# ABSTRACT

PAZHAYAVEETIL, ULLAS CHANDRASEKHAR. Hardware Implementation of a Low Power Speech Recognition System. (Under the direction of Dr. Paul Franzon.)

Speech is envisioned as becoming an important mode of communication and interaction with computing devices and systems in the future. The potential for speech recognition applications both in our living environment as well as our workplace is well understood, and is driving the shift from current command and control applications to full fledged speech recognition systems. While these systems would be especially useful in future mobile embedded domains, the real-time performance requirements of such systems cannot be met by current embedded processors. Even modern high performance microprocessors are barely able to keep up with the real time requirements of sophisticated speech recognition applications often straining the resources of the host processor while incurring a power consumption that is prohibitive in the embedded space. Custom ASIC solutions in the past have focused on faster clock rates and logic speeds, and have largely ignored the power reduction aspect of the problem. In this dissertation, we approach the speech recognition problem by a) designing a custom ASIC that is flexible enough to adapt to evolutionary improvements in the design and take advantage of these improvements at the algorithmic level to achieve low power operation, and b) restructuring the memory and adapting a lexical style dictionary along with an innovative 'Timestamp' scheme to reduce overall memory requirements, bandwidth requirements, power and energy consumption.

Our Gaussian Estimator achieves real-time performance while reducing power consumption by 2 orders of magnitude over a software implementation running on a Pentium 4 processor, and by 43% over the best previous comparable ASIC design. Our design also achieves 3 orders of magnitude improvement in energy consumption over the Pentium 4 and 35% improvement in energy consumption over the previous ASIC design.

Similarly our Viterbi Decode unit performs real-time speech recognition while achieving an improvement of 3 orders of magnitude over the Pentium 4 and 1 order of magnitude improvement over the previous design – the perception processor – in both power and energy savings.

Our final design achieves real-time recognition over a vocabulary that is 6-12 times as much as competing designs while taking up only 2.5 times as much area.

# HARDWARE IMPLEMENTATION OF A LOW POWER SPEECH RECOGNITION SYSTEM

by

**ULLAS CHANDRASEKHAR PAZHAYAVEETIL**

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

**ELECTRICAL ENGINEERING**

Raleigh, North Carolina

2007

**APPROVED BY:**

| | |
|---|---|
| Dr. Suleyman Sair | Dr. Min Kang |
| Dr. Paul Franzon | Dr. W. Rhett Davis |

Chair of Advisory Committee

# DEDICATION

Two very special people have been responsible for helping me complete this work, and their contribution began far before I even joined this project. They are the two greatest people I know and whom I have the privilege of proudly calling my parents – Mr. Chandra Sekhar and Mrs. Rajalakshmi Chandra Sekhar. This dissertation is a testament to their foresight, love, support and their willingness to sacrifice so that I never had to. None of this could have been possible without them and I dedicate this dissertation to them.

# BIOGRAPHY

Ullas Chandra Sekhar received his Bachelor of Technology in Electrical Engineering from the Indian Institute of Technology, Madras in 2001. He was also awarded the Dr. Ing Dieter Kind Prize for the best B.Tech project in Electrical Engineering for 2001. He received his Master of Science in Electrical Engineering in 2003 from the University of Texas at Arlington. During his MS program, he worked as a research assistant at the Automation and Robotics Research Institute. He joined North Carolina State University in 2003, where he worked both as a research assistant and a teaching assistant while working towards his Ph.D. His research interests are primarily hardware and ASIC design.

# ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Paul Franzon, for his patient mentoring and guidance through out this research. He managed to maintain the perfect balance between giving me a free hand in exploring research ideas, and steering me towards focusing on the core goals of the project. His confidence in this fledgling project during rough patches and roadblocks, and his support both in matters relating to this research as well as outside of it, have been two of the greatest assets to this research dissertation. I would also like to thank the rest of my committee – Dr. Rhett Davis, Dr. Suleyman Sair, and Dr. Min Kang – for their generous advice and help, especially in finding solutions to practical issues that came up during the course of this research. Their open-door policy has been extremely reassuring. I would like to thank my friends for providing a balance to my lifestyle, which would otherwise have been ‘a graduate student’s life’. Last, but certainly not the least, I would like to thank my family – my father Mr. Chandra Sekhar and my mother Mrs. Rajalakshmi Chandra Sekhar, for their love, support and sacrifice, my brother Unmesh who could not have cared less about the details of what I did, but listened to me nevertheless just because it was important to me, and finally my fiancé Gayathri for getting me through this last year, and lifting my spirits when things looked tough. Thank you all!!

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# Introduction

Speech recognition has been an area of active research for more than 40 years [1], maturing from an area of pure academic research to one with growing use in the marketplace. Opportunities for application of speech recognition are immense and diverse. The trend of a constantly increasing number of computing devices, both in our living environment, as well as our workplace, calls for a better way of interacting with them. Speech is already an established mode of communication in many mobile embedded environments, and the value of speech recognition applications in such environments is immeasurable. When compared to other forms of communication, speech has some attractive advantages. A person can speak about 3-4 times faster than they can type, allowing for greater communication efficiency. It is ideal for multi-modal tasks since the hands and eyes are free to do other tasks. Speech accessories are cheap, easily available and small, creating mobile capacity. Thus this hands-free, user friendly nature of speech coupled with improvements in processor speeds and trends of ubiquitous computing, promise to make speech a primary human/machine interface in the near future.

## 1.1 The Problem

A variety of software packages for speech recognition are available in the mass market today, such as Dragon Systems' Dragon Naturally Speaking, IBM's ViaVoice, Lernout & Hauspie's Voice Xpress, and Philips FreeSpeech98[2]. Vocabularies in commercial systems today range from 20,000 to 150,000 words. Recognition accuracies have been steadily improving as well, though current systems are still not sufficiently accurate to easily take dictation without straining the resources of the microprocessor. The CMU 'Speech in Silicon'[3] is another project that is working at developing a hardware solution to speech recognition.

A successful design of a speech recognition system involves achieving accuracy levels of >95% while being fast enough to be able to process speech in real time. The system should be able to achieve speaker-independent speech recognition for multiple languages. (i.e. the system must be deployed with base training (speaker independent models) with update and training facilities). The system should be flexible enough to handle different dialects and speech model parameters with minimal effort to change dialects, ideally requiring only a download of a new model. Power savings have also come to be of significant importance during the designing of such systems due to their application in the embedded domain.

Even modern high performance microprocessors are barely able to keep up with the real time requirements of sophisticated speech recognition applications. The run times [4] for a 29.3 sec segment of speech over the SPHINX III speech recognition system is shown in Figure 1.1. The theoretical run times are based on ideal scaling of performance with frequency. It is evident that for speech recognition the performance of the processor

does not scale ideally. In theory a 2.4 GHz processor should achieve real time performance. In practice a processor frequency of approximately 2.9 GHz is required to satisfy real time requirements.

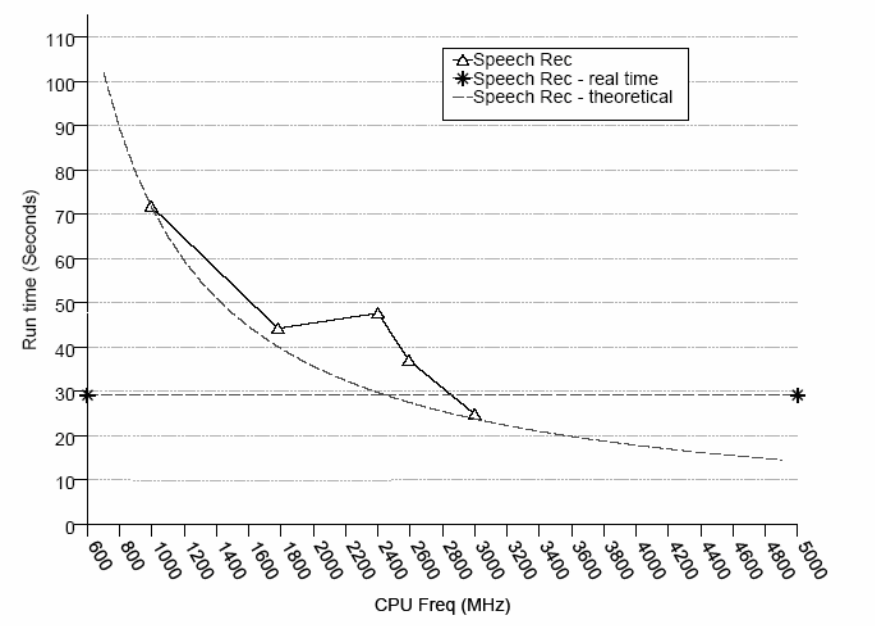



Figure 1.1 - Performance of SPHINX 3 on Intel Pentium 3 and later processors (900MHz to 3GHz)

This performance gap suggests that when moving to more complex future speech recognition workloads higher frequencies alone are not the solution, fundamental architectural improvements are called for. The speech recognition system also severely limits the processor's availability for other task. The results clearly show that speech applications stress the performance limits of high-end processors.

By their very nature, applications such as speech are likely to be most useful in mobile embedded systems. A fundamental problem that plagues these applications is that they require significantly more performance than current embedded processors can deliver. Most embedded and low-power processors, such as the Intel XScale, do not have the hardware resources and performance that would be necessary to support a full-

featured speech recognizer. The energy consumption that accompanies the required performance level is often orders of magnitude beyond typical embedded power budgets. Figure 1.2 provides a rough estimate of the speech recognition rate achievable on various modern computing systems [5]. Annotated above each bar is the time each processor class would operate on a single "AA" rechargeable battery (1600 mA·Hr). It is clear that, while high-end systems are within the performance range necessary for real-time speech recognition, they far exceed the power budget of portable devices.

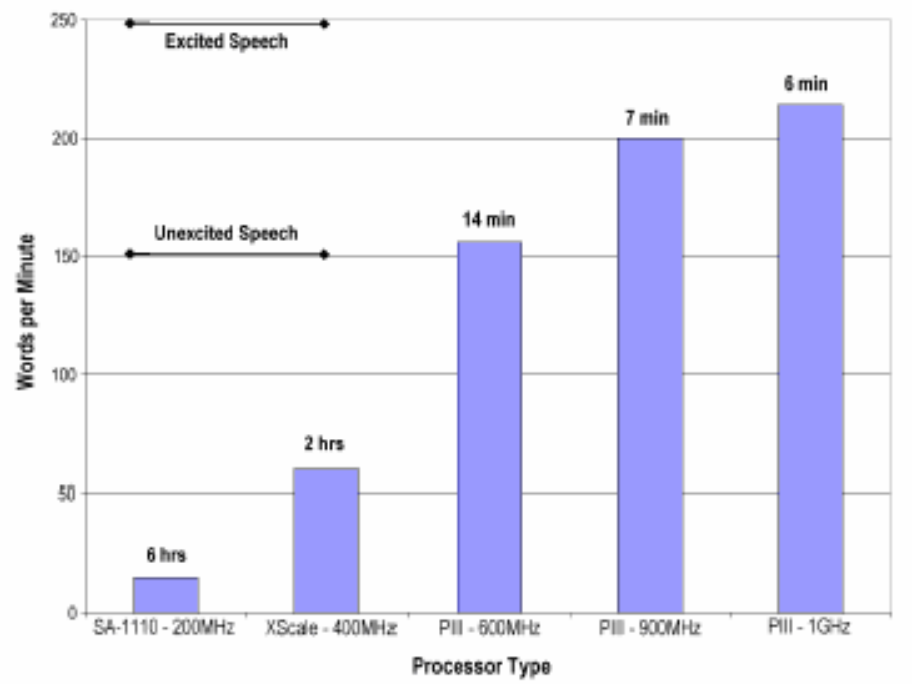



Figure 1.2 - Performance and power considerations for speech recognition on modern architectures

Another problem that these systems face especially in the mobile domain is that with new and emerging techniques in speech recognition, it is more attractive to find a solution that can readily adapt to most if not all of the developments and techniques. Often many of these new techniques prove to be useful in more than one domain. As an example, 'Frame skipping' is a technique used in speech recognition that proves to

reduce both processing and power consumption as well as required memory bandwidth. It is prudent to look into finding a way to incorporate these into our own system.

## 1.2 The Solution

The problem can now be redefined as finding a solution that provides high accuracy speech recognition at speeds high enough for real time processing while consuming minimum power and also being able to incorporate a degree of flexibility to new and emerging techniques. These problems are easily dealt with by a custom ASIC coprocessor with some flexibility built into it. Speech and security interfaces are by nature always on. Stressing the host processor with the speech recognition task limits its availability for other tasks. Thus a speech coprocessor can help free-up resources so that the host processor can focus on other tasks. An ASIC can also help achieve high-end speech recognition within the power budget of embedded processors. Similar to video cards in a standard PC, speech on a chip has the ability to perform better than a speech recognition system in software running on a processor.

While an ASIC solution is attractive, it limits the flexibility and level of generality offered. Speech is a rapidly growing field and the techniques used to process and recognize speech improve constantly. These improvements reflect not only such factors such as accuracy and speed, but also on the reduction in power consumption and a general improvement of the process as a whole. In this dissertation, one of our strategies was to preserve a level of flexibility in the architecture that we developed. The advantage to this was two-fold. Firstly, new and emerging techniques may become a standard for how speech recognition is done in the future. By making sure that our architecture can

adapt to these techniques, we ensure that our design remains competitive. Secondly, our design will be able to take advantage of the performance improvements offered by these new techniques to achieve even better performance numbers.

A bottleneck that has been commonly identified in speech recognition designs is the memory bandwidth required by this application especially when trying to perform real-time recognition. The total amount of data that is needed to support and perform an application such as speech recognition is quite large. A large training set, complex acoustic and language models, and a very large parameter set is usually required to support the complex nature of the application. Having a larger parameter set and better-trained models also contribute to the performance of the application in terms of accuracy and recognized vocabulary size. However the tradeoff is speed and power.

While the *total* amount of data required for speech recognition is large, not all of this data is used all the time. By controlling and manipulating the amount of data *accessed* each time frame, it is possible to maximize the use of data accessed, while at the same time minimizing or eliminating unnecessary accesses.

Keeping this in mind, we restructured how the data is arranged and accessed, switching from a flat vocabulary structure to a lexical structure. By coupling this with our innovative 'Timestamp' technique and dynamic memory allocation, we eliminated redundancies and reduced the total data that needed to be processed every time frame. This led to a design that provided immediate savings in terms of memory bandwidth, speed and power.

## 1.3 Organization of the Dissertation

Chapter 2 will provide an introduction to the basics of speech recognition. Chapter 3 will describe previous research and related work. Chapter 4 will provide an overview of the system design and also discuss the front end in some detail. Chapter 5 presents the design(s) of the Gaussian Estimation unit. Chapter 6 presents the Viterbi Decoder Unit with emphasis on the design issues and advantages of switching from a flat tree structure to a lexical tree structure. The performance of these designs is analyzed in Chapter 7. Chapter 8 draws conclusions and highlights important points in the design methodologies.

# CHAPTER 2

# Background

## 2.1 Acoustic Modeling

### 2.1.1 A brief overview of speech recognition and HMMs [1, 4, 6, 7, 8, 9]

The front-end converts an unknown speech waveform into a sequence of acoustic vectors $Y=y_1,y_2,y_3\ldots$each representing a short time (10 ms) speech spectrum of the speech signal. This is also known as the observation sequence. This sequence may correspond to a number of actual word sequences $W= w_1,w_2,w_3\ldots$(It should noted that W actually refers to a sequence of representative models which could be words, but usually are sub word units). The basic speech recognition task is to determine the most probable word sequence $W_p =w_1,w_2,w_3\ldots$, given the observed acoustic signal $Y$:

$$W_p = \arg\max_w P(W|Y) = \arg\max_w ((P(W)P(Y|W)/P(Y)) \qquad (2.1)$$

The first term $P(W)$ is the probability of observing $W$ independent of the observed signal (sequence) $Y$, which is determined by a language model. The probability P(Y|W) is determined by an acoustic model. Figure 2.1 shows the computation of the probabilities of a postulated word sequence $W$ = "This is speech". Each word is converted into a sequence of phones applying a pronouncing dictionary, and for each phone there is a corresponding statistical model, a hidden Markov model (HMM). The sequences of

HMMs (representing the postulated utterance) are concatenated to form a single composite model. The probability of that model generating the observed signal *Y* is calculated, yielding the wanted probability *P(Y|W)*. This decoding process may be repeated for all possible word sequences, and the most likely sequence is selected for the recognizer output as the 'recognized speech'.

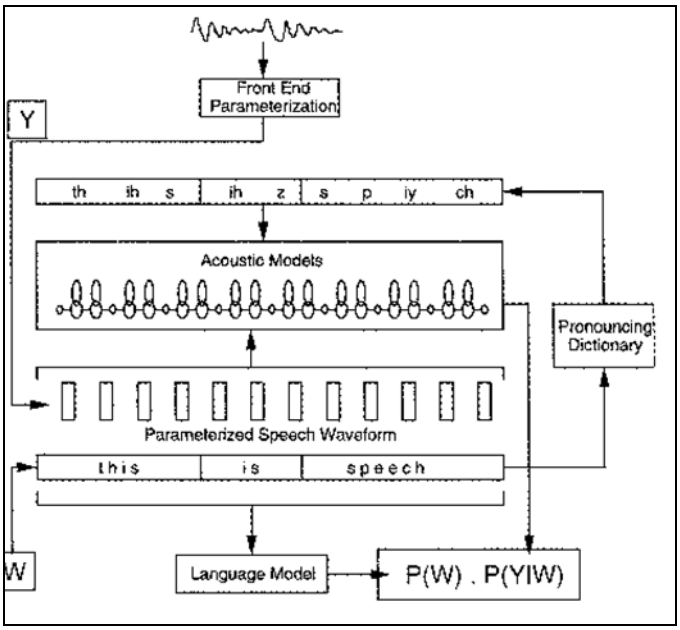



Figure 2.1 - Schematics of an LVR (Large Vocabulary Recognition) system. It shows the computation of the probabilities of a postulated word sequence *W* = "This is speech" [7]

An *N*-state Markov Model is completely defined by a set of *N* states forming a finite state machine, and an *NxN* stochastic matrix defining transitions between states, whose elements $a_{ij}$ represent the probability of transitioning from state *i* to *j* at time *t*; these are the transition probabilities. With a Hidden Markov Model, each state additionally has associated with it a probability density function $b_j(y_t)$ which determines the probability

that state $j$ emits a particular observation $Y_t$ at time $t$ (the model is "hidden" because any state could have emitted the current observation). The probability density function (p.d.f). can be continuous or discrete; accordingly the pre-processed speech data can be a multi-dimensional vector or a single quantized value. The quantity $b_j(y_t)$ is known as the observation probability. Such a model can only generate an observation sequence $Y=y_1,y_2,y_3...y_T$ via a state sequence of length $T$, as a state only emits one observation at each time $t$. The set of all such state sequences can be represented as routes through the state-time trellis shown in Figure 2.2. The $(j,t)^{th}$ node (a state within the trellis) corresponds to the hypothesis that observation $Y_t$ was generated by state $j$. Two nodes $(i,t-1)$ and $(j,t)$ are connected if and only if $a_{ij} > 0$.

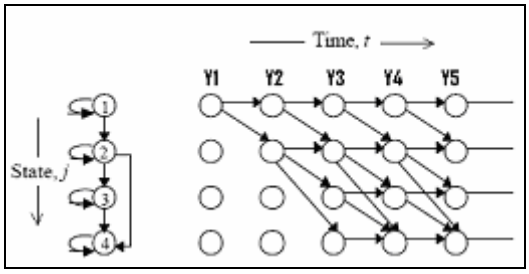



Figure 2.2 - Hidden Markov Model showing the finite state machine for the HMM (left), the Observation sequence (top),and all possible routes through the trellis

As described above, we compute $P(W|Y)$ by first computing $P(Y|W)$. Given a state sequence $Q=q_1\ q_2...q_T$, where the state at time $t$ is $q_t$, the joint probability, given a model $W$, of state sequence $Q$ and observation sequence $Y$ is given by:

$$P(Y,Q|W) = b_1(y_1)\prod_{t=2}^{T} a_{qt-1qt}b_{qt}(y_t) \qquad (2.2)$$

Assuming the HMM is in state 1 at time $t = 1$, $P(Y,Q|W)$ is the sum of all possible routes through the trellis, i.e.

$$P(Y|W) = \sum_{allQ} P(Y,Q|W) \tag{2.3}$$

In practice, the probability *P(Y|W)* is approximated by the probability associated with the state sequence which maximizes *P(Y,Q|W)*. This probability is computed efficiently using Viterbi decoding. Firstly, we define the value $\delta_t$ *(j),* which is the maximum probability that the HMM is in state *j* at time *t*. It is equal to the probability of the most likely partial state sequence $Q=q_1\ q_2 \ldots\ q_t$, which emits observation sequence $Y=y_1,y_2,y_3\ldots$, and which ends in state *j*:

$$\delta_t(j) = \max_{q1,q2\ldots qt} P(q_1,q_2\ldots q_t\ ;\ q_t = j\ ;\ y_1,y_2\ldots y_t|W) \tag{2.4}$$

It follows from Equation 2.2 and 2.4 that the value of $\delta_t(j)$ can be computed recursively as follows:

$$\delta_t(j) = \max_{1\le i\le N} [\ \delta_{t-1}(i)a_{ij}].b_j(y_t) \tag{2.5}$$

where *i* is the previous state (i.e. at time *t-1*). This value determines the most likely predecessor state $\psi_t(j)$, for the current state *j* at time *t*, given by:

$$\psi_t(j) = \arg\max_{1\le i\le N} [\ \delta_{t-1}(i)a_{ij}] \tag{2.6}$$

At the end of the observation sequence, we backtrack through the most likely predecessor states in order to find the most likely state sequence. Each utterance has an HMM representing it, and so this sequence not only describes the most likely route through a particular HMM, but by concatenation provides the most likely sequence of HMMs, and hence the most likely sequence of words or sub-word units uttered.

Each node in the trellis must evaluate Equation 2.5 and Equation 2.6. This consists of multiplying each predecessor node's probability $\delta_{t-1}(i)$ by the transition probability $a_{ij}$, and comparing all of these values. The most likely is multiplied by the

observation probability $b_j(y_t)$ to produce the result. After a number of stages of multiplying probabilities in this way, the result is likely to be very small. In addition, without some scaling method, it demands a large dynamic range of floating point numbers, and implementing floating point multiplication requires more resources than for fixed point. A convenient alternative is therefore to perform all calculations in the log domain[10]. This converts all multiplications to additions, and narrows the dynamic range. Hence Equation 2.5 becomes

$$\delta_t(j) = \max_{1 \le i \le N} [\, \delta_{t-1}(i) + \log a_{ij}] + \log [b_j(y_t)] \qquad (2.7)$$

The result of these changes means that a node can have the structure shown in Figure 2.3. The figure highlights the fact that each node is dependent only on the outputs of nodes at time *t-1*, hence all nodes in all HMMs at time *t* can perform their calculations in parallel. The way in which this can be implemented is to deal with an entire column of nodes of the trellis in parallel.

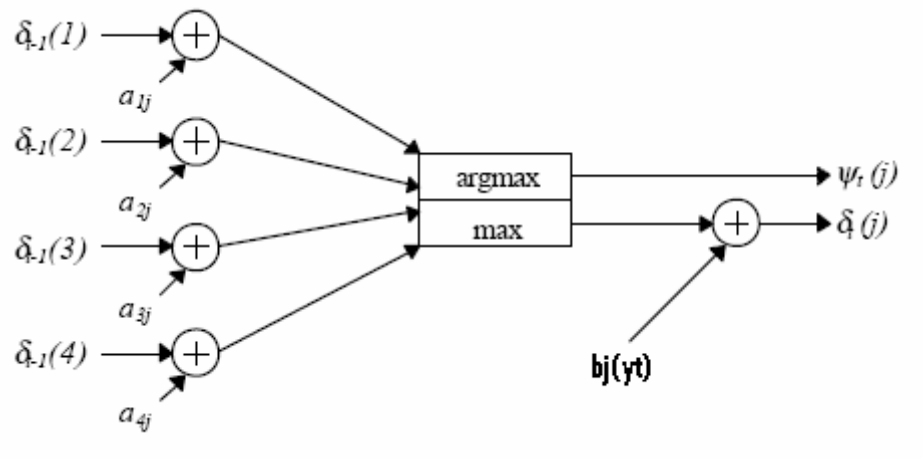



Figure 2.3 - Block diagram of node representing state *j*

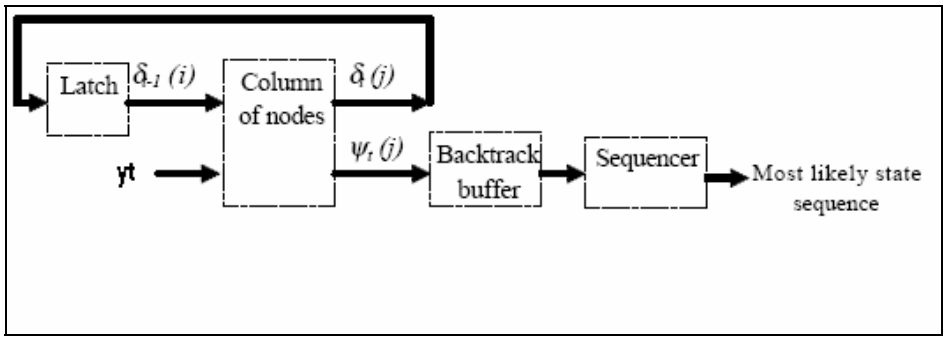



Figure 2.4 - Decoder structure showing forward computation and backtracking

As the speech data comes in as a stream, we can only deal with one observation vector at a time, and so we only need to implement one column of the trellis (In actuality, implementing the entire trellis is area heavy and unnecessary as will be seen later. Instead an optimum number of 'node units' are designed so that each unit will update a set number of nodes). The new data values (observation vector $y_t$ and maximal path probabilities $\delta_{t-1}(j)$) pass through the column, and the resulting $\delta_t$ values are latched, ready to be used as the new inputs to the column when the next observation data appears.

Each node outputs its most likely predecessor state $y_t(j)$, which is stored in a sequential buffer external to the nodes. When the current observation sequence reaches its end at time $T$, a sequencer module reads the most likely final state from the buffer, chosen according to the highest value of $\delta_T(j)$. It then uses this as a pointer to the collection of penultimate states to find the most likely state at time $T-1$, and continues with backtracking in this way until the start of the buffer is reached. As the resulting state sequence will be produced in reverse, it is stored in a sequencer until the backtracking is complete, before being output. This state sequence reveals which HMMs have been traversed, and hence which words or sub-word units have been uttered. This information can then be passed to software, which assembles the utterances back into words and sentences.

### 2.1.2 Phones & Triphones[11,12,13,14]

Equation 2.1 needs the quantity *P(Y|W),* the probability of an acoustic vector sequence *Y* given a word sequence *W* to find the most probable word sequence. A simplistic approach to achieve this would be to obtain several samples of each possible word sequence, convert each sample to the corresponding acoustic vector sequence and compute a statistical similarity metric for the given acoustic vector sequence *Y* to the set of known samples. For large vocabulary speech recognition this is not feasible because the set of possible word sequences is very large. Instead words may be represented as sequences of basic sounds. Knowing the statistical correspondence between the basic sounds and acoustic vectors, the required probability can be computed. The basic sounds from which word pronunciations can be composed are known as *phones* or *phonemes*. Approximately 50 phones may be used to pronounce any word in the English language. For example, the sentence 'This is speech' is represented as 'th ih s ih z s p iy ch'.

While phones are an excellent means of encoding word pronunciation, they are less than ideal for recognizing speech. The mechanical limits of the human vocal apparatus leads to co-articulation effects where the beginning and end of a phone are modified by the preceding and succeeding phones. Recognizing multiple phone units in context tends to be more accurate than recognizing individual phones. Current speech recognition systems deal with three-tuples of phones called *triphones*. It is customary to denote triphones as left context-current phone+right context. For example 'th-ih+s' is a triphone that represents the context of the 'ih' phone in the word 'this'. The final 'ch' phone in dissertation can be modeled with a cross-word triphone whose right context is the first phone in the next word or by the triphone 'iy+ch-sil' where 'sil' is a special

phone that denotes silence. Though there are approximately 50x50x50 = 125000 possible triphones, only about 60,000 actually occur in English.

The front-end processes every 10ms data input and extracts relevant features that will enable the recognition process. This involves converting every 10ms of speech into a 39-element vector, that has statistical information about itself (each element having means, variances, mixture weights and scale factors). This then needs to be compared using some distance measure to every triphone phone model and the observation output probability for this input was obtained.

Initial HMM recognizers used discrete OPFs and sub-vector quantized (VQ) models, which are easy to compute. The acquired acoustic vector was replaced by the index of the closest codebook vector, and OPFs were just look-up tables containing the VQ index probabilities. While this is computationally efficient, the discretization of observation probability leads to excessive quantization error and thereby poor recognition accuracy. To obtain better accuracy, modern systems use a continuous probability density function and the common choice is a multivariate mixture Gaussian in which case the computation may be represented as [8]:

$$b_j(y_t) = \sum_{m=1}^{M} \frac{w_m}{(2\pi)^{D/2} \sqrt{\prod_{n=1}^{N} \sigma_{jm}^2[n]}} \exp\left( -\frac{1}{2} \sum_{n=1}^{N} \frac{(y_t[n] - \mu_{jm}[n])^2}{\sigma_{jm}^2[n]} \right) \tag{2.8}$$

Here $y_t$ is the input vector, $\mu_{jm}$ and $\sigma_{jm}$ represent the mean and standard deviation of the multivariate Gaussian, $w_m$ is the mixture weight, $m$ is the no. of mixtures and $n$ is the no. of feature vectors. The term before the exponent does not depend on the input and can be pre-calculated. Doing all calculations in the log domain significantly simplifies Equation

2.8 and reduces the exponent calculation to simple multiplications. Performing these modifications, Equation 2.8 reduces to

$$\log b_j(y_t) = \sum_{m=1}^{M} c_{im} \sum_{n=1}^{N} (y_t[n] - \mu_{jm}[n])^2 \times V_{im}[n] \tag{2.9}$$

$C_{im}$ being the final mixture weight and $V_{im}$ being the variance. The HUB4 speech database which was considered for this research chose the values of $M$ and $N$ to be 8 and 39 respectively. The outer term represents an addition in the log domain. Every triphone model would require pre-training which means that the number of parameters to be estimated is about 60000 x (39x8x2+8) = 11.37 million parameters. The training data usually available is insufficient to estimate so many parameters. The usual solution is to cluster together HMM states and share a probability density function among several states (Figure 2.6). Systems using such clustered probability density functions are called semi-continuous or tied-mixture systems. Groups of states that are tied together are called 'senones'. The total number of senones in the English Language ranges between 4000 and 6000.

The utterance hierarchy for the word 'HI' is shown in Figure 2.5. Each triphone is represented by a 4 state HMM. Only the first 3 states are emitting states (i.e., they can produce an observation vector $y_t$). The last state is a null state. The overall effect is that of combining all the triphone HMMs by adding null transitions between the final states of one triphone HMM to the initial state of its successor. To model continuous speech, null transitions are added from the final state of each word to the initial state of all words.

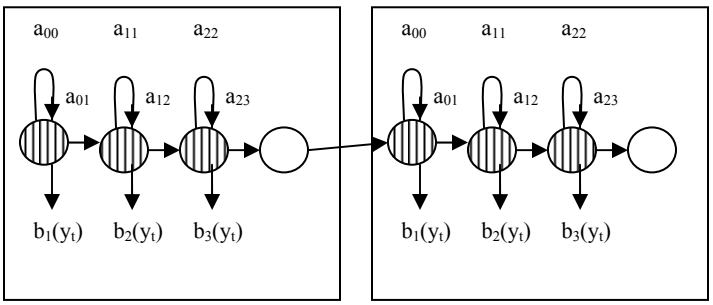



Figure 2.5 - HMM for word 'HI', with phones 'hh'&'ay, and triphones 'sil-hh+ay'&'hh-ay+sil'. Each HMM has 3 emitting senone states (striped oval) and one nullstate (plain oval).

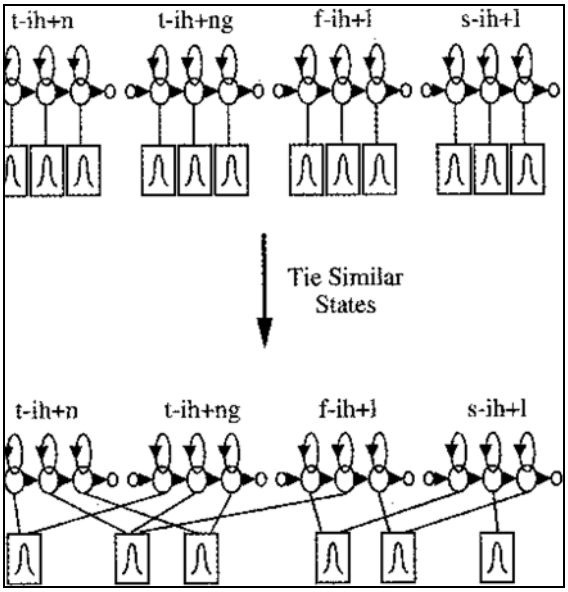



Figure 2.6 - State-tying[7]

The viterbi search is modeled as a lexical tree search[15,16]. The roots of the tree correspond to the set of all triphones that start any word in the dictionary. Each node in the tree points to the next triphone in the expanded pronunciation of a word etc. Triphones that occur at the end of a word are specially marked so that a language model may be consulted at those points. Thus the lexical tree is a multi-rooted tree where each node points to an HMM and a successor node. In the case of word exit triphones there are multiple successors. Given an acoustic vector sequence Y, each vector in the sequence is

applied successively to the HMMs and the probability that the HMM generated that vector is noted. Transitions are made in each step to successor nodes. On reaching a word exit triphone, the state sequence history is consulted to find the word that has been recognized. The last n words (usually n=3) are checked against a language model for further analysis. Acoustic vectors are evaluated successively and on evaluating an HMM for the current vector, if the HMM generates a probability above a certain threshold, the successors of the HMM will be evaluated in the next time step. Thus there is always a list of currently active HMMs/lexical tree nodes and a list of nodes that will be active next. This combination of the Viterbi search combined with pruning techniques (comparing with threshold) is known as the Viterbi Beam Search[17-20]. Pruning prevents uncontrolled generation and maintenance of nodes with time by deactivating low-probability paths.

## 2.2 Language modeling

The introduction of a language model to the speech recognition unit increases accuracy of the recognition hypothesis [21-23]. It helps introduce additional biases to the several alternate similar words that the acoustic model recognizes and cannot choose between. This also helps in the quick pruning of improbable paths and the unnecessary explosion of node generation. All state-of-the-art speech recognition systems implement a language model in one form or the other, and so it is necessary to study the language model in order to build a system that is commercially competitive.

*N*-gram models [24,25] that predict the probability of a word sequence (in other words the probability of a word given the previous *N-1* words) prove to be an effective

and common approach. They encode simultaneously the syntax, semantics and pragmatics and they concentrate on local dependencies. They are thus extremely effective for languages like English in which word order is important and the strongest contextual effects come from near neighbors. They also have the distinct advantage of being easy to train. *N*-grams can be trained automatically from a large corpus of text.

The complexity of the modeling increases logarithmically with the size of the vocabulary V (the complexity for an N-gram vocabulary will be V^N). Large values of N lead to complexities both in the viterbi decoding stage as well as the training stage, where sparse training data leads to incorrect parameter estimation. This may even have a deteriorating effect on the recognition accuracy. Thus a modest value of 3 is chosen, and this has proven to be sufficient in systems like Sphinx and HTK. Such models are called trigrams. A trigram model may be trained using the following equation [26]:

$$P(w3 \mid w1, w2) = \frac{F(w1, w2, w3)}{F(w1, w2)} \tag{2.10}$$

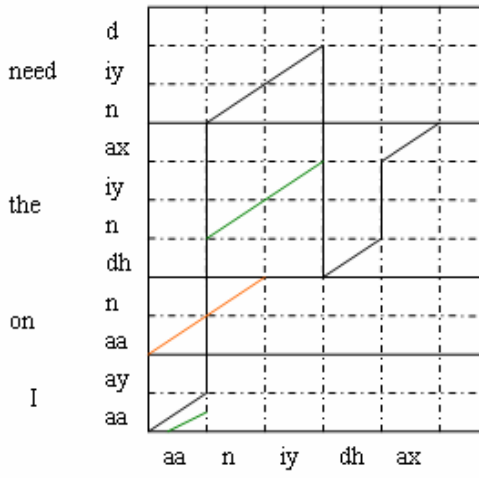
Here, *F(w1,w2,w3)* refers to the frequency of occurrence of the trigram (*w1,w2,w3*) in the training text and *F(w1,w2)* refers to the frequency of occurrence of the bigram (*w1,w2*). In practice, for a large vocabulary all possible trigrams will not be present in the training corpus. In that case bigram or unigram probabilities are used in the place of trigram probabilities after reducing the probability by a back-off weight, which accounts for the fact that the next higher *N*-gram has not been seen and therefore has a lower chance of occurring [27].

A typical example of the use of a language model is shown in Figure 2.7. The Bigram probabilities were taken from the Brown Switchboard Corpus. Without the language model, the speech recognition system recognizes the phrase ‘on’ and has a close

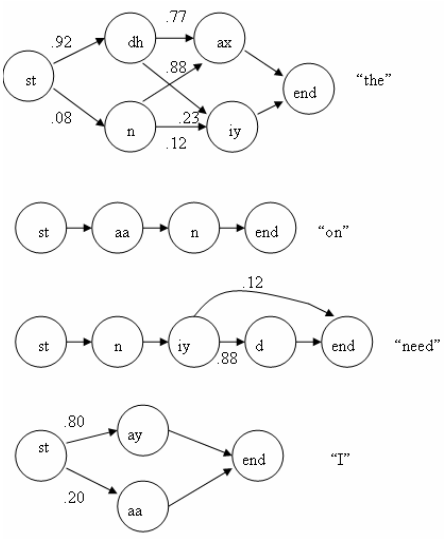
contention in recognizing the phrase ‘I the’, instead of the actual phrase ‘I need the’. With the language model, the improbable paths are pruned out leading to a good recognition and increase in accuracy.

Figure 2.7 - Use of Language model

(a) Shows the four words ‘I’,’Need’,’The’ and ‘on’, on the y-axis and the paths that the different hypotheses take as the input phones ‘aa n iy dh ax’ come in.
(b) Shows the word models for the 4 words.
(c) Shows the Bigram Probabilities obtained from the Brown Switchboard Corpus
(d) Shows the path probabilities without the use of the language model.
(e) Shows the path probabilities with the language model.



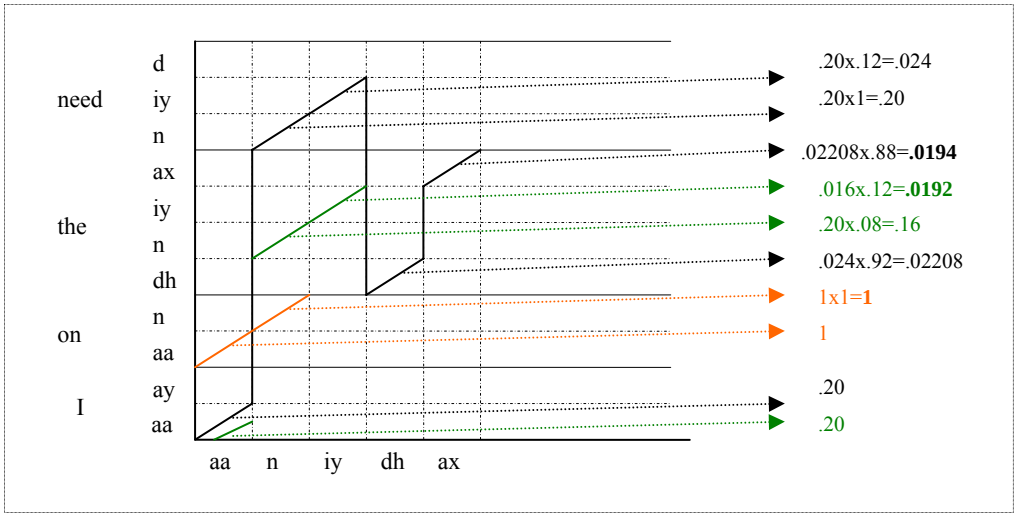

(a)



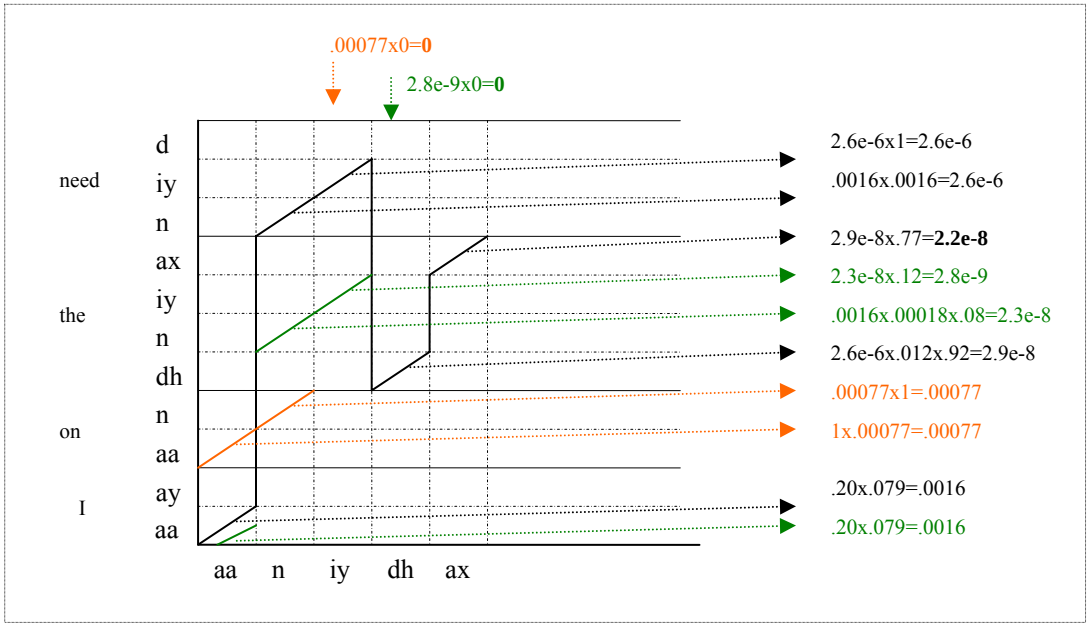

(b)

| Bigram Probabilities | |
|---|---|
| | |
| # Need | 0.00018 |
| # The | 0.016 |
| # On | 0.00077 |
| # I | 0.079 |
| I need | 0.0016 |
| I the | 0.00018 |
| I on | 0.000047 |
| I I | 0.039 |
| on need | 0.000055 |
| on the | 0.094 |
| on on | 0.0031 |
| on I | 0.00085 |
| need need | 0.000047 |
| need the | 0.012 |
| need on | 0.000047 |
| need I | 0.000016 |
| the need | 0.00051 |
| the the | 0.0099 |
| the on | 0.00022 |
| the I | 0.00051 |

(c)


need
the
on
I
d
iy
n
ax
iy
n
dh
n
aa
ay
aa
aa n iy dh ax
.20x.12=.024
.20x1=.20
.02208x.88=.0194
.016x.12=.0192
.20x.08=.16
.024x.92=.02208
1x1=1
1
.20
.20

(d)


.00077x0=0
2.8e-9x0=0
need
the
on
I
d
iy
n
ax
iy
n
dh
n
aa
ay
aa
aa n iy dh ax
2.6e-6x1=2.6e-6
.0016x.0016=2.6e-6
2.9e-8x.77=2.2e-8
2.3e-8x.12=2.8e-9
.0016x.00018x.08=2.3e-8
2.6e-6x.012x.92=2.9e-8
.00077x1=.00077
1x.00077=.00077
.20x.079=.0016
.20x.079=.0016

(e)

# CHAPTER 3

# Related Work

Several speech recognition applications have been developed in the industry and sold as commercial products. Several speech applications and training kits are also available from universities that are mainly oriented at speech research. Speech Recognition is inherently a computationally demanding task and hence software solutions running on a general-purpose processor are not good at real-time speech recognition. These systems are not particularly designed keeping underlying architecture in mind and hence end up squeezing all available resources of the processor. Some of the systems available are discussed below.

## 3.1 Commercially Available Systems

Commercially available software systems are IBM's ViaVoice and Dragon Systems Dragon Naturally Speaking, claim to be able to support between 10k and 150k vocabulary sizes. They are speaker independent though some amount of training is encouraged. The systems can support multiple languages and report a >95% accuracy. They are however slow and sensitive requiring the user to speak slowly and discontinuously, and cannot be truly called a continuous speech recognition system. It is also not real-time since there is a lag between the spoken sentence and the recognized

output. These also take up a large amount of the CPU processing power effectively blocking it from doing other tasks and also consume a large amount of power. Considering the 'always on' nature of speech, these software at best work as a 'speech-to-text' solution used for small periods of time for very specialized functions, and are not the best step towards making speech a general user interface.

Since a lot of the research has been done with a product in mind, the systems used at Dragon [28-30] usually require a lot less resources than other research systems. For instance, while most systems were using a 10ms frame size, Dragon decided to use a coarser sampling of 20ms frame sizes. While most systems use 32-bit floating point formats for their input features, Dragon used 8-bit fixed-point parameters. To reduce the resource requirements even more, the commercial version applies automatic vocabulary switching to restrict the search space.

The IBM speech recognition system differs form other competing systems by the early use of a rank-based approach for the computation of observation probabilities that allows avoiding certain search problems related to extreme probability values. The search strategy is a combination of an A*[31,32] with a time-synchronous Viterbi search (the so called 'envelope search') and is therefore difficult to compare to the fully time-synchronous search of other systems. Other features that made this system substantially different from its competitors include that during the first recognition pass the usual mixture of Gaussian probabilities were replaced by a probability derived from the rank of the models score [33-36]. The system has a recognition vocabulary of 65,000 words.

A version of ViaVoice is also available for embedded processors and mobile devices such as PDA's. However this is a command and control type application working

on template recognition rather than speech processing. It has a maximum of 16k vocabulary and maintains a dynamically swappable 4,000 word phonetic flat list. It is speaker independent but has small database support. This system requires several minutes of read speech to adapt to a new user.

The AT&T Watson speech recognition engine [27,37-39] is a software implementation of the AT&T voice processing technology. It uses a gender based triphone model and a 5-gram language model. The recognition takes place in 2 phases – the first pass does the phone recognition, word recognition and builds the word lattice; the second pass rescores the word graph.

Microsoft's Whisper [31,40,41] is an adaptation of CMU's SPHINX and incorporates speaker adaptation and noise cancellation. The acoustic models are compressed and hence Whisper claims to be memory efficient. The software supports a 60,000-word vocabulary with the ability to add new words. It works with any Windows application and has two specialized applications for use in Windows - "Dictation Pad" provides enhanced dictation features while "IntelliSense" converts spoken numbers and times automatically.

SRI's DECIPHER [42-44] is another HMM-based system that uses multi-pass time synchronous Viterbi decoding. A first forward-backward pass generates word-lattices using a 60,000-word bigram language model and context-dependent hidden Markov models. Only within-word context dependent models were used in the first pass. The Gaussian computation was sped up using vector quantization and Gaussian shortlists. The second pass performed a forward-backward N-best search on the word-lattices using the first-pass hidden Markov models. The N-best lists were then re-scored using more

expensive acoustic and language models. Most of the development effort went into the reduction of the error rate, and only little research was reported on means for achieving real time recognition.

SRI‘s ‘Phraselator’ [45] is a template based phrase recognition and translation system. The Phraselator is a handheld, wireless computer used to translate more than 1,000 spoken English phrases into languages such as Arabic and Pashto. This is again a lookup engine rather than a recognition engine.

Several smaller companies have developed IC’s for small end applications like voice-responsive toys and voice dialers [46-50].

## 3.2 Research Systems

SPHINX speech recognition systems [4,13,20,51-56] are CMU’s state-of-the-art large vocabulary speech recognition systems. They are semi continuous HMM based [13] systems and use state-tied triphone models. It uses scaled integer observation probability computation. Decoding is done in 2 phases with a first pass viterbi decoding done to reduce the search space and a second pass A* algorithm to combine the results of the first pass and the language models. Beam searches and various pruning strategies ate used to reduce the computations and prune paths quickly. However the CMU-SPHINX system uses an extraordinary amount of power while running on desktops and the power required is prohibitive for mobile applications. The SPHINX–IV system is CMU’s speech recognition system for the mobile domain. It however supports a much lower vocabulary size and is still not real-time.

Cambridge University's ABBOT system differs significantly from other systems by being the only neural net driven system used in large-scale evaluations [57-59]. It is also among the few systems to use a stack decoder rather than a time synchronous Viterbi algorithm for the search process. The ABBOT system supports a 65,000-word vocabulary. The recognition speed for this system in the evaluation was estimated to about 60 times slower than real time on a 170MHz UltraSparc.

Cambridge University's HTK system was the best recognizer in the 1994 LVCSR evaluation with a word error rate of only 10.5% [60,61]. All promising algorithms such as quinphone models, cross-word models, 4-gram language models and unsupervised speaker adaptation were applied to this system. Considerable effort was also invested in the pronunciation dictionaries. The commercial cousin of this system 'Entropics' is more stable and more toolkit-oriented.

SONIC [62] is a toolkit for enabling research and development of new algorithms for continuous speech recognition. The system uses HMM acoustic models and a two-pass search strategy. Model-based adaptation methods such as Maximum Likelihood Linear Regression (MLLR), Structured MAP Linear Regressions as well as feature-based adaptation methods such as Vocal Tract Length Normalization, cepstral mean & variance normalization, and Constrained MLLR are implemented [63].

## 3.3 Implementations on general-purpose processors

Several approaches have been taken towards finding solutions to the problems of speed, accuracy and power consumption. Research has been traditionally geared towards improving accuracy, with performance as a secondary goal. Power efficiency has been

largely ignored. Even the yearly HUB speech recognition evaluation reports typically emphasize improvements in recognition accuracy and mention improvements in performance as a multiple of "slow down over real-time" [64, 65].

Binu [4] used rapid semi-automatic generation of low-power high performance VLIW processors for the perception domain. Energy efficiency was primarily achieved by minimizing communication and activity using complier-controlled fine-grain clock gating. Ravishanker's research improved the performance of the Sphinx speech recognition system by trading off accuracy in a computationally intensive phase for faster run time and then recovered the lost accuracy by doing additional processing in a computationally cheaper phase of the application [52]. This research also reduced the memory footprint of speech recognition by using a disk based language model cached in memory by the software. Agram, Burger and Keckler characterized the Sphinx II speech recognition system in a manner useful for computer architects [66]. They focused on ILP as well as memory system characteristics including cache hit rates and block sizes and concluded that available ILP was low. They compared the characteristics of the Sphinx II system with those of Spec benchmarks and also hinted at the possibilities and problems associated with exploiting thread level parallelism.

Intel ICRC lab researchers executed the Intel speech recognition system on several versions of the x86 processor [67]. The study focused on the run time and size of the working set, and the language was Mandarin Chinese. They reported a decrease in ILP with increased clock rate. IPC decreased from between 1 and 1.2 at 500MHz to .4 at 1.5 GHz. Obviously increasing clock rate is not the solution to improving speech recognition performance. The decrease in ILP was attributed to memory system behavior,

but a detailed explanation was not provided. The ICRC speech system is not publicly available, and details of the ICRC workload are not available.

## 3.4 Hardware solutions

A common approach to finding a solution to the problems of speed, accuracy and power consumption is to build hardware accelerators to speed up parts of the speech recognition process or build a complete hardware speech recognition system.

An earlier attempt to accelerate speech recognition may be found in the work of Anantharaman and Bisiani [68], who presented a multi-processor architecture as well as a custom architecture for improving the beam search algorithm used in the CMU distributed speech recognition system. The paper also describes the design process of the custom architectures and presents a number of ideas on the automatic design of custom systems for data dependent computations.

Researchers at the Norwegian University of Science and Technology designed a custom probability density function (PDF) coprocessor in a 0.8μ CMOS process that could accelerate the computation of Gaussian observation probabilities in a hidden Markov model based speech recognizer [69]. This research concluded that memory bandwidth was a limiting factor for Gaussian computation. They approached the memory bandwidth problem by using a new fixed point representation called dynamical circular fixed-point format which reduced the memory bandwidth in half. The PDF coprocessor could evaluate 40,000 39-element Gaussian components in real time using this format at 154 MHz consuming 853 mW of power. The workload has worsened by a factor of 15.3 since this effort. Also the design was a fixed-point implementation and it is obvious from

current systems that fixed point does not meet the required range of generated probabilities necessary for high accuracy speech recognition.

Sergiu.et.al[70] implemented a low power speech recognition system that performs real-time speech recognition. This is a complete design that has its own language and acoustic model. It exploits parallelism existing in speech recognition algorithm with multiple Processing Elements (PE). Parallel execution helps in reducing clock frequency resulting in reduced power usage. Dynamic loading of speech models is used for changing language grammar and retraining, while reprogramming is used to support evolution of recognition algorithms. The focus on small sets of words (at one time) reduces the complexity, cost and power consumption. The recognizer is extremely flexible and can support multiple languages or dialects with speaker-independent recognition. The average power dissipation for the logic part of the decoder was about 5.125mW in the 0.18μm process, and area of the design was about 2.5mm$^2$. Evaluations demonstrate an order of magnitude improvement in power compared with optimized recognition software running on a low-power embedded general-purpose processor of the same technology and of similar capabilities. The design however is only good for a very small vocabulary and is phoneme based, therefore it is not good for real life use.

Binu [71] improved memory bandwidth on the SPHINX –III systems by using a blocking scheme whereby a single retrieved set of variances, means, weights and scales were used to calculate the observation probabilities for 10 frames. A special-purpose accelerator for the dominant Gaussian probability phase was developed for a 0.25$\mu$ CMOS process. Area, power and bandwidth efficiency are achieved by reducing the floating-point precision (a 24-bit format rather than the standard 32-bit format),

restructuring the computation, and sharing memory bandwidth. The accelerator improves performance of a Pentium 4 (.13μ) system running the SPHINX – III system by a factor of 2, while simultaneously improving on the energy consumption by 2 orders of magnitude. The Gaussian accelerator consumed 1.8 watts while the Pentium 4 consumed 52.3 watts during Gaussian computation, representing an improvement of 29 fold.

Researchers at the Tsinghua University developed a single chip speech recognition system based on an 8051 microcontroller core [72]. The chip was designed based on the SOC (system on chip) philosophy and an 8-bit MCU, RAM, ROM, ADC/DAC, PWM, I/O ports and other peripheral circuits were all embedded in it. Software modules including control/communication, speech coding and speech recognition algorithms were implemented in an 8051 compatible microcontroller core, resulting in the extremely low cost of the chip. The speech recognition adopted the template matching technique, and recognized up to 20 phrases with an average length of 1 second and a recognition accuracy reaching more than 95% with the background SNR above 10 dB. The design operated at 40 MHz and consumed 60mW of power.

Borgatti et.al [73] developed a low-power, low-voltage speech processing intended to be used in remote speech recognition applications where feature extraction is performed on terminal and high-complexity recognition tasks are moved to a remote server accessed through a radio link. Power optimization of portable terminals featuring speech recognition was pursued by partitioning speech recognition complexity between on-terminal circuitry and remote hosts. The proposed system was based on a CMOS feature extraction chip for speech recognition that computed 15 cepstrum parameters, each 8 ms, and dissipated 30 μW using a 0.9-V supply. Single-cell battery operation was

achieved. Processing relied on a novel feature extraction algorithm using 1-bit A/D conversion of the input speech signal. The chip was implemented as a gate array in a standard 0.5-μm, three-metal CMOS technology. Recognition rates above 98% were achieved in isolated-word speech recognition tasks.

LOGOS [74] is a real time hardware speech recognition system that uses both parallel and pipelined processing techniques, matching up to several hundred words from a previously stored vocabulary of whole word "templates" in real time. An efficient single pass dynamic programming algorithm is used to find the sequence of templates that best represents the input. Continuous recognition is achieved using a trace back technique on partial recognition results. Vargas et.al [75] proposed a novel HW/SW co-design with redundancy techniques to implement a speech recognition system. Special attention needs to be taken when partitioning digital signal processing algorithms into hardware and software parts. The design methodology in this work partitions the HW and SW parts in such a way as to boost system performance while maintaining low area overhead. The proposed approach is called the "speech recognition-oriented HW/SW partitioning and fault-tolerant design" approach (or simply SCORPION approach). Other VLSI-based designs[76,77] for HMM speech recognition also exist.

The most recent research effort towards a hardware solution to the speech recognition problem is CMU's 'Moving Speech Recognition from Software to Silicon: the In Silico Vox' research project [3,78]. The project is looking at both an FPGA approach as well as an ASIC solution to building massively parallel, energy efficient speech chips that will attain high performance. The FPGA approach plans to use a Xilinx Vertex II based chip with a clock rate of 50MHz to recognize about 1000 words. The

design will operate about 2.3 times slower than real-time due to memory bandwidth constraints. The ASIC implementation will be designed to recognize about 5000 words, operating at about 6 times faster than real time requirements. The overall size of the chip is estimated to be about $10mm^2$. The latest publication [79] from this group indicates successful design of an ASIC implementation that recognizes speech at .6xRT (real time), and runs at 125MHz. Hardware prototyping on a Xilinx XC2VP30 FPGA, using a Xilinx XUP development board has also been completed. The prototype recognizes about 1000 words at roughly 2xRT. No power readings were reported.

The use of reconfigurable logic and FPGA devices is another common approach to the speech recognition problem [6, 80]. Techniques vary from power aware mapping of designs onto commercially available FPGA devices to hybrid methods where specialized function blocks are embedded into a reconfigurable logic array [81,82]. The inherent reconfigurability of FPGAs provides a level of specialization while retaining significant generality. FPGAs, however, have a significant disadvantage both in performance and power when compared to either ASIC or CPU logic functions.

Melnikoff et. tried to exploit the parallel nature of the algorithm to implement the decoder part of the speech recognition system onto an FPGA[83]. The question of how to deal with limited resources, by reconfiguration or otherwise, was also addressed. A later publication [84] indicated that the design was implemented on a Xilinx Virtex XCV1000 FPGA, sitting on Celoxica's RC1000-PP development board. The design occupied 5,590 of the XCV1000's slices, equal to 45%, and ran at 44 MHz. The correctness values (less than 60%) are clearly much lower than those found in commercial speech recognition products. This is because such products use significantly more complex models.

Vargas et a.[85] used a new approach by which the Viterbi algorithm is built in with the HMM structure. The probabilistic state machines run as parallel processes and the entire system is built as a hardware/software co-design. The design is intended for isolated word recognition, and runs 500 faster than classic implementations.

## 3.5 Digital Signal Processing solutions

It is common practice to use special DSP techniques to try to reduce the number of operations performed and speed up the algorithm. Bocchieri [86] used vector quantization of the input vector to identify a subset of gaussian neighbors so that only a smaller subset of likelihood computations needed to be calculated with only a small loss in accuracy. In [87], the authors reduced the word error rate for speaker-independent continuous speech recognition by modeling subphonetic events with HMM states and treating the state in phonetic hidden Markov models as the basic subphonetic unit. Lee et al [88] used context-independent models for selection and back off of corresponding triphone models. Another study [89] used lookahead HMMs and frame skipping to skip the gaussian calculation for frames that do not show significant change from the previous frame(s). Much effort has been spent on optimizing the computation of likelihood for all tied triphone states [90-94]. The authors in [95] categorize the different schemes into 4 layers, and described how the different layers can interact with each other and compliment each other. These techniques and their relevance are discussed further in detail in Chapter 5.

# CHAPTER 4

# System Architecture

The Speech Recognition System can be separated into 3 separate units – the Front-End, the Gaussian Estimator, and finally, the Viterbi Decoder. Figure 4.1 shows the overview of the system. The Front end processes the spoken input and provides mel-frequency cepstral coefficients (MFCCs) of the input data to the Gaussian Estimator. The Gaussian Estimator then uses this input along with the acoustic models to provide the phone/senone scores. The Viterbi Decoder uses this information along with the language models to calculate the state transitions and word-to-word transition probabilities and search for the most likely sequence of words.

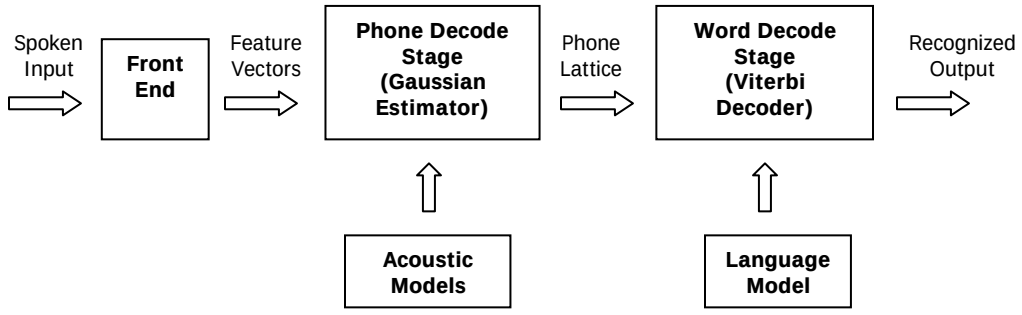



Figure 4.1 - System Overview

Profiling of the speech recognition process shows that system spends approximately .89%, 49.8% and 49.3% [4] of its compute cycles in the Front-End, Gaussian Estimation

and Viterbi Decoding Stages respectively. Fortunately, both the compute heavy stages are extremely parallelizable, making them ideal candidates for translation into hardware.

In our implementation we used a 4 state HMM model that was the most common choice among speech recognition systems. The Gaussian Estimator unit was modeled as a 32-bit unit though 20-bit implementations have proven to be sufficient [101]. This was so that we would be able to work with standard 32-bit acoustic scores, and optimize down once the initial implementation was complete.

Figure 4.2 shows the overview of the implementation of the complete system. The system consists of a 'Gaussian Estimator', a 'Viterbi Decoder', a 'System Control', an 'Arbiter', three DRAMS and an SRAM. The system acts like a co-processor that interfaces with a host processor. The front-end will be implemented on the host processor providing inputs to our co-processor at 16kbps. The Arbiter acts as control to the Gaussian Estimator and also directs the inputs from the front-end to it. Initially it was envisioned that the Arbiter would be used to control any feedback from the Viterbi Decoder that we may implement. The final system does not contain feedback. The Arbiter is also responsible for initializing the DRAM Unit (with the Acoustic Models). The Gaussian Estimator updates and places the phone/senone scores in the SRAM every frame. The Viterbi Decoder accesses these scores as well as other data in the two DRAM units to complete the process, and place the outputs in the form of a sequence of word indexes on the 'output' bus. The Viterbi Decoder is an extremely self-sufficient unit requiring minimal external control. System Control acts as control unit for the entire system and is also responsible for initializing DRAM UNIT –1 and DRAM UNIT –2 which hold the Language Models.

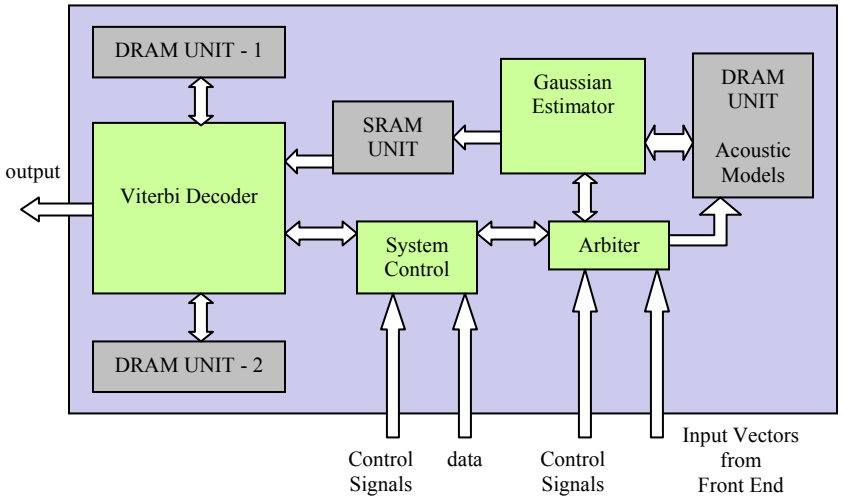



Figure 4.2 - System Implementation Overview

The different parts of the system are discussed in detail in the following chapters. We use this chapter to also briefly describe the Front End. This part of the system is not implemented in hardware as it is responsible for less that 1% of the total computation workload and can easily be done by a host processor

## 4.1 Front-End

Even though the front end only occupies less than 1% of the compute time on speech systems, it is very important for two reasons – (a) The front-end is responsible for generating a good smooth spectral estimate of the incoming speech waveform and is directly responsible for obtaining good output observation probability estimates (b) Understanding acoustic vectors is a crucial prerequisite to understanding the operation of the acoustic model. The Front-End is not dealt with in detail in this research, and the implementation (which is almost standardized at this point) is obtained from [102]. The overview is shown in Figure 4.3.

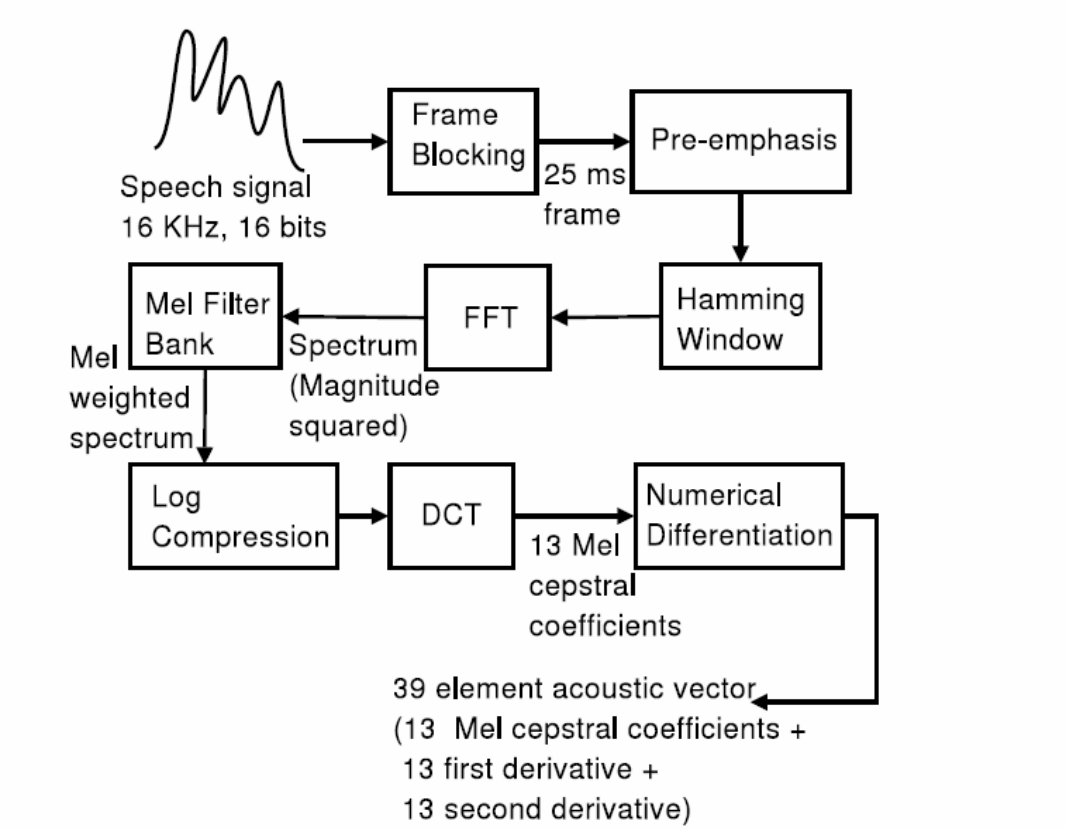



Figure 4.3 - Front-End [102]

The human vocal apparatus has mechanical limitations that prevent rapid changes to sound generated by the vocal tract. Thus, speech signal are considered to be quasi-stationary, i.e., stationary in short time intervals (typically 5-20 ms), during which the spectral characteristics are relative constant. DSP techniques may be used to summarize the spectral characteristics of a speech signal into a sequence of acoustic observation vectors, with a single vector representing about 10 ms of speech.

The front-end segments the signal into blocks and makes a smooth spectral estimate for each block. The (constant) length of the blocks is typically chosen to be 10 ms, and the blocks are overlapped in time to give a longer analysis window of 25 ms (commonly a Hamming window, i.e., a raised cosine). The raw signal is also pre-emphasized, i.e., high frequencies are amplified in order to compensate for their

attenuation because of the mouth directivity. Other processing such as noise suppression and band-pass filtering (usually frequencies limited to 300-3400 Hz) and removal of long silences is also necessary.

The spectral estimates can be computed via linear prediction or discrete Fourier analysis or cepstrum analysis, and the coefficients, i.e., the final acoustic vectors can be obtained via a number of transformations. The most typical method of modern LVR systems is to use the mel-frequency cepstral coefficients (MFCCs). The processing is mainly done in order to satisfy constrains in the acoustical modeling component. The Fourier spectrum of each speech block is smoothed by a mel-scale filter-bank that consists of 24 band-pass filters that simulate the human cochlea processing. The mel-scale is linear up to 1000 Hz and logarithmic thereafter, creating a so-called perceptual weighting to the signal.

From the output of the filter-bank a squared logarithm is computed, which discharges the unnecessary phase information and performs a dynamic compression making the feature extraction less sensitive to dynamic variations. This also makes the estimated speech power spectrum approximately Gaussian.

Finally, the inverse DFT is applied to the log filter-bank coefficients, which actually is reduced to a discrete cosine transformation (DCT). DCT compresses the spectral information into lower-order coefficients, and it also decorrelates them allowing simpler statistical modeling. The acoustic modeling assumes that each acoustic vector is uncorrelated with its neighbors. Due to human articulatory system, this requirement is not well satisfied; there is continuity between consecutive spectral estimates. Second and third order differentials greatly reduce this problem. A linear regression is fitted over two

preceding and two following vectors resulting the final acoustic vector with 39 components (each 32 bits wide).

# CHAPTER 5

# Gaussian Estimator

Chapter 2 described how modern systems use a continuous probability density function to evaluate the senone scores every time frame, and the common choice is a multivariate mixture Gaussian. This Gaussian Estimation is perhaps the most computationally intensive part of the speech recognition process. The complex nature of these computations require several IEEE 754 format floating point operations to take place per time frame. Several multiplications, additions, multiply-accumulate-compares, $(a-b)^2$ type operations, scaling and weighing need to be carried out for the computation of each senone score. Current research trends also show that these computations are steadily increasing in complexity and number. These factors impact the real-time performance of the system, requiring faster evaluations of scores. They also have a significant impact on the total power consumption of the system. Porting the computation to hardware is justified both by the need to speed up the number of computations that are carried out during the process as well as the need to lower the power consumption of the process.

This chapter discusses two and a half versions of Gaussian Estimators that were designed. The first version is the baseline system. It is a highly pipelined floating-point unit that efficiently ports the algorithm to hardware. This baseline system forms the foundation for the second design. It was also used to test the improvements in

performance of the later designs. An improved and more power efficient unit – the Reduced Calculation Gaussian Estimation was also designed. This version was not completely implemented and tested due to a shift in research focus (hence its reference as a 'half' version). However, the ideas of this design were incorporated in the final version of the working design and hence it merits a brief discussion.

The constant push to improve both speed and accuracy of the gaussian estimation has led to the emergence of several new techniques [86-90]. Chan et al. [95] has categorized these techniques into four unique layers and discussed their interaction. They also showed that any new technique that emerges could be placed in one of the four layers. The final version of the Gaussian Estimator is a highly flexible floating-point unit that can be programmed to adapt to new techniques at three of these four layers. The gains offered by an ASIC design in terms of speed and power consumption at the circuit level are obvious. However, this design, also offers the ability to incorporate new techniques in speech recognition and use it to reduce power consumption at the algorithm level. The area of speech recognition is fast-paced and constantly growing. A technique that is new today may become a standard among speech recognition systems in the future. Hence, an important implication of the flexibility of this design is that it can adapt (at least in part) to such changes in the future and not become obsolete.

## 5.1 Baseline Gaussian Estimator

The baseline Gaussian Estimator (GE) is shown in Figure 5.1. The 3 values of mean (Mean_0), variance (Variance_0) and the input vector component enter the GE pipeline, and are processed. During this time, the next values of mean and variance are brought

into buffers Mean_1 and Variance_1. One the next iteration, these values are used along with the input, and the buffers Mean_0 and Variance_0 are updated to be used for the next cycle.

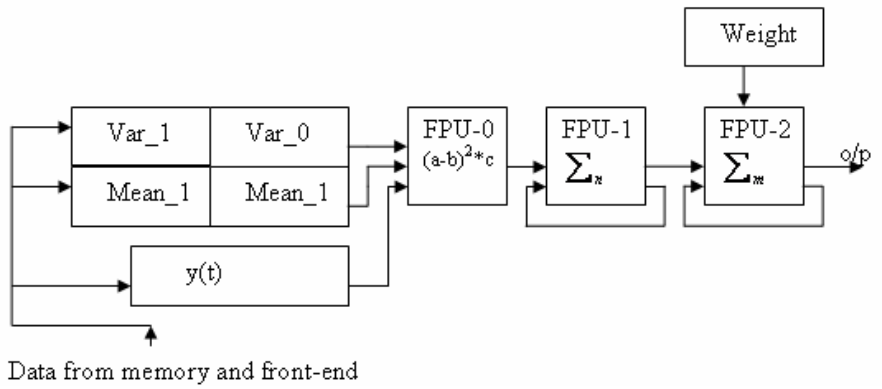



Figure 5.1 - Baseline Gaussian Estimator

Each vector codebook consists of 8 mixtures of 39 variance and mean values each. Each of these are 32 bits wide. Every mixture also has its own weight value. Effectively, this would mean that each Gaussian Estimator unit would have to read in (39x2+1)x8x32=20224 bits of data per senone. Thus for about 6000 senones, this amounts to 15.16MB of data every 10ms, or 1.516GBps. The input data accounts for another 16KBps.

The total on-chip memory required would be the buffers for the variance and mean values, as well as the buffer memory on the accumulators (to hold the temporary values during the looping stage). This works out to be about (32*4+32*39*2)+32*2 +32*8= 2.944Kbits of memory per GE unit.

Binu [71] used a blocking technique and data-reuse to arrive at a partial solution to the memory bandwidth problem on a PC. On an custom ASIC, the bottleneck will be the maximum operating speed of each of the three FPU's. Correspondingly multiple such units would have to work in parallel to realize real-time requirements. The FPU units

were written in Verilog and simulated in Synopsys for a .25μ technology. The total area occupied by the coprocessor design (not including memory) was about 1.955mm$^2$. The fully pipelined designed functioned at about 233Mhz. At this speed, evaluating about 6000 senones in real time would mean implementing about 6 such units in parallel.

Table 5.1 gives the current memory speed and area measures for SRAMs and DRAMs. The required on-chip memory area would be 6x2.944Kbits/(3Mbits/10mm$^2$) = .0588mm$^2$. The total logic circuitry area is about 11.73mm$^2$. The required off-chip is about 15.16 MB and the required bandwidth of 1.5GBps is also met.

- **Off-chip DRAM**
  - E.g. 800 MHz DDR2 (x16 bit)
    - Bandwidth : 160 – 1600 MBps
    - 256 Mbit – 1Gbit
- **On-chip DRAM**
  - ~4 GBps (256-bit embedded DRAM)
  - 16 Mbit/10 sq.mm
- **On-chip SRAM**
  - ~16 GBps (256-bit, 500 MHz embedded SRAM)
  - 3 Mbit/10 sq.mm

Table 5.1 – SRAM and DRAM specifications

While real time implementation is possible, we also need to focus on building a low-power design. Let us look carefully at Equation 2.9. The higher the value of the output in the log domain (the o/p is a probability and the negative is implicit and not used for calculations), the worse off the input is at matching this particular senone. With this insight, we can reduce the number of performed calculations by using this algorithm - if the value of the accumulators reaches/exceeds a particular predetermined threshold, the GE unit squashes all further calculations for that input on that senone and returns a high value (modeling negative log (0)). This reduced-calculation GE is shown in Figure 5.2. Obviously, the efficiency of this technique depends on how quickly a good estimate of

the output observation probability can be obtained. Thus the components of the vector codebook that are the heaviest contributors to the output probability need to be brought in and computed first, followed by components contributing less. Thus if the vector code book is properly setup after profiling, unnecessary calculations leading to incremental differences can be avoided and in the case of speech recognition, improbable paths can be pruned quickly. This can lead to huge power savings both at the GE level as well as during the Viterbi decoder search. Note: This technique is dependant on the ability of the profiling to clearly demarcate heavy and light contributors to the distortion. Also it is important to note that for inputs that closely match the acoustic vector codebook, all contributions will be small, and the final output will also be small (indicating a high probability).

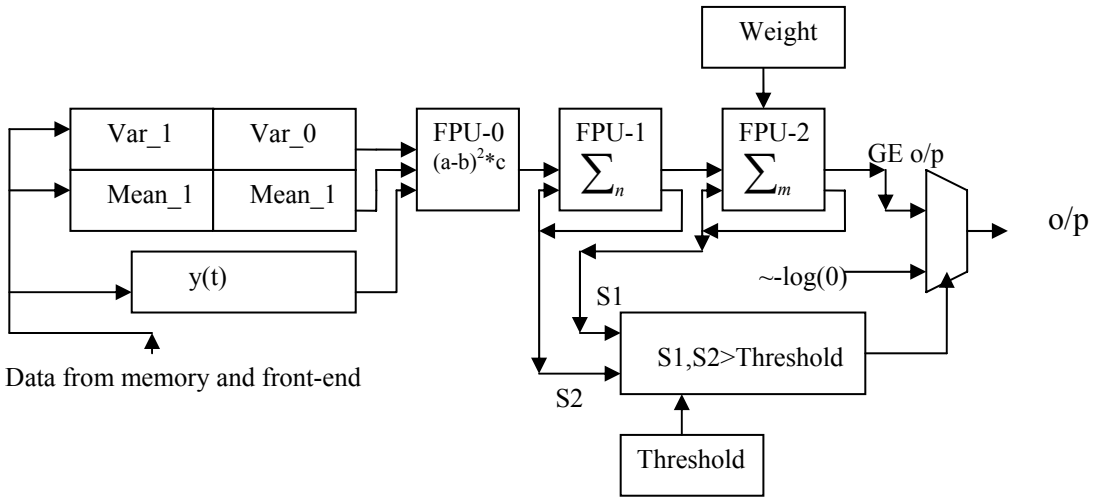



Figure 5.2 - Reduced Calculation Gaussian Estimator

## 5.2 Flexible Gaussian Estimator

It was realized during the course of this research that the speech research community has increased emphasis on the DSP end of the application with a large amount of effort being

spent on trying to reduce computation at the algorithm level. In particular, several techniques have been proposed to speed up the computation of the senone score [86-89]. We realized that in order to be commercially competitive, any coprocessor that we build has to be able to adapt (at least in part) to these new techniques.

The techniques have been proposed to speed up the computation of the senone scores have been categorized into different layers [95]. Individual fast GMM computation techniques can be associated with specific layers. This allows two things – first, representative techniques associated with each layer can be compared for effectiveness, and secondly techniques from different layers can be used in tandem to improve the speeds of different parts of the computation simultaneously. It should be noted that using two techniques associated with the same layer will perhaps not be as effective as they will be working on the same part of the computation and in fact may reduce performance due to increased overhead with low returns. We will briefly go through these layers and techniques.

## 5.2.1 Layer Categorization

### 5.2.1.1 Frame-Layer Algorithms

Frame-layer algorithms decide whether the senone score of the current frame should be computed or skipped. Speech is a slowly changing signal, and so the observation probabilities do not usually change dramatically from one frame to the next. The score of a skipped frame is assumed to be copied from the most recently computed frame. The simplest technique called Simple-Down Sampling (SDS) computes the frame scores only for every other frame. There is no faster way to compute a score than to assume it has not

changed and not compute it at all. This technique can be extended to skipping every two out of three frames as well. Errors introduced by this system can be, at least in part, recovered by using wider beams (more relaxed pruning). The technique is discussed in detail in [89].

Another technique used that falls into this category is the Conditional Down Sampling (CDS) [89]. A VQ (Vector Quantized) codebook is trained from all means of GMMs of a set of trained acoustic models. Then, in decoding, every frame's feature vector is quantized using that codebook. A frame is skipped if the feature vector is quantized to a codeword, which is the same as that of the previous frame. For rapid speech the error rate introduced by SDS can be prohibitive, especially if the concept is extended from skipping only every other frame to skipping every two out of three frames.

It is thus important to estimate if the signal remains static for the next couple of frames. This is done by calculating the output of the HMMs for the lookaheads [89]. Lookahead HMMs are similar to monophone (context independent) HMMs. These are small in number allowing the computation of the observation probabilities $b_j(O_t)$(senones scores) for these models to be fast enough to be performed for every input frame $t$ and every state $j$ of the lookahead HMM models.

The Euclidean distance between the vectors of these scores for the next two frames is used to determine whether or not the signal is currently changing:

$$D(t) = \sum_{j=0}^{J} \left( b_j(O_t) - b_j(O_{t-1}) \right)^2 \tag{5.1}$$

The maximum score $D_{max} = max_{(0<i<t)}D(i)$ is also recorded. The normalized value if then compared to find out how many frames need to be skipped.

$$D_{norm}(t) = \frac{D(t)}{\max_{0 \le i \le t} D(i)} \tag{5.2}$$

$0.0 < D_{norm}(t) \le 0.3$ : skip 2 frames

$0.3 < D_{norm}(t) \le 0.6$ : skip 1 frame

$0.6 < D_{norm}(t) \le 1.0$ :skip no frames (5.3)

#### 5.2.1.2 GMM-Layer algorithms

Algorithms that decide which senone scores need to be computed in each computed frame are placed in the Gaussian mixture model (GMM) layer. One such representative technique is the Context-Independent (CI) GMM-based selection (CIGMMS) [88]. CI GMM scores (scores of monophone models instead of triphone models) are first computed (50 in number). A beam or threshold is then applied to these scores. For those scores that are within the preset threshold, the detailed context dependent CD GMM (senone) scores are computed. The rest are backed-off by their corresponding CI GMM score.

#### 5.2.1.3 Gaussian-Layer algorithms

The different techniques used to decide which Gaussians dominate the senone score computation are categorized as Gaussian-Layer techniques. One such technique is the Sub-Vector Quantization Gaussian Selection (SVQGS) [86] where a rough model computation is first used to decide which Gaussians (in the multidimensional Gaussian distribution) need to be computed.

In this scheme, all mixture components are clustered into neighborhoods[86] during system training. A vector quantizer, consisting of one codeword for each neighborhood of gaussians is also defined. During the recognition, vector Quantization of

the input frame vector allows the selection of a small subset (neighborhood) of Gaussians whose likelihoods must be exactly computed, and a complimentary set whose likelihoods can be quickly approximated by table look-up or by a small constant.

An input observation $O_t$ is said to be quantized to a particular codeword if

$$\frac{1}{D}\sum_{d=1}^{D}\frac{\left(O_t[d]-\mu_j[d]\right)^2}{U_j[d]} > \varphi \qquad (5.4)$$

Here $O_t[d]$ represents the $d^{th}$ element (or dimension) of the input, and $\mu_t[d]$, and $U_t[d]$, represents the $d^{th}$ element (or dimension) of the $j^{th}$ mean and covariance. φ is the quantization threshold. Once this codeword has been identified, only the input vector likelihoods of the Gaussians of the codeword neighborhood are exactly computed and added into the state likelihoods.

### 5.2.2 Implementation

The Front-End takes up less than 1% of the total computation, and can be implemented using the host processor. The system context for our Gaussian Estimator (GE) is shown in Figure 5.3. The extracted feature vectors are fed into the GE through an arbiter unit. The arbiter also initializes the DRAM with the acoustic models for all senones, and obtains feedback from the Viterbi Decoder. It uses this information to provide the input to the GE. The results or senone scores are stored in an SRAM, from which the Viterbi Decoder accesses them. Figure 5.4 shows the block diagram of the GE itself.

The Gaussian Estimator is a highly pipelined IEEE 754 32-bit floating-point unit. The data path consists of an $(a-b)^2*c$ floating point unit followed by an adder that completes the inner loop of Equation 2.9. A fused multiply-add unit – the Scale Weight

Adjust unit - (labeled SWA in Figure 5.4) then performs the scale and weight adjustment. A log_add unit completes the outer loop.

The basic building units (adders /multipliers) for this design have a 3-stage pipeline needing three buffers at both adder units to complete the calculations. The internal control unit has a course grain control over most of the arithmetic units, and multiplexers (all shaded boxes in Figure 5.4). The different mode settings provide course-grain control of different stages of the pipeline, as well as control over the interaction between the different units. This will be discussed in detail. From this point on, 'a','b' and 'c' will be used to refer to the inputs to the Gaussian Estimator.

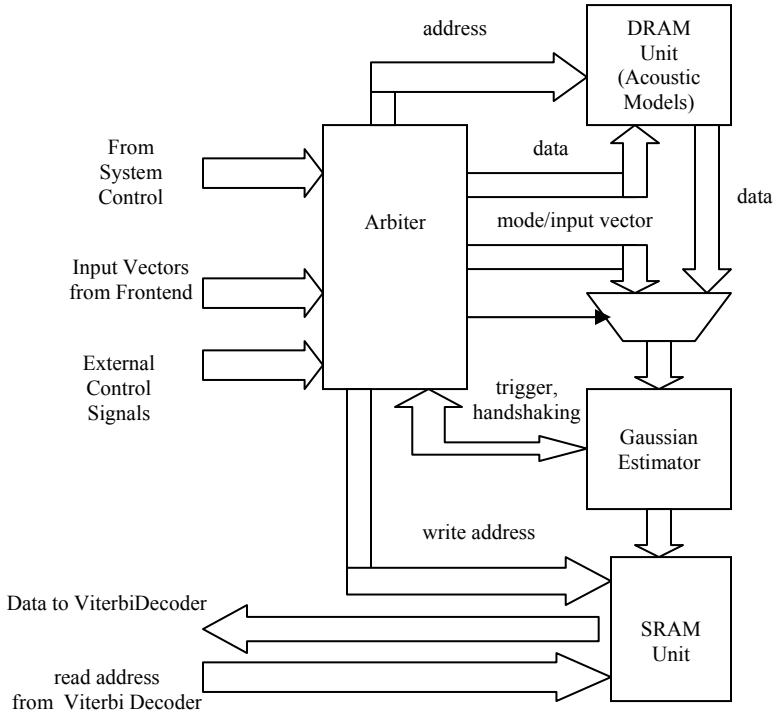



Figure 5.3 - Gaussian Estimator Interfacing

During the initialization and setup phases, the parallel inputs to the unit, In1 and In2, initialize the internal LUT for the log_add module, and also setup the vector length

or dimension (*dim*), number of mixture (*mix*), scales, weights, thresholds and other setup information. It should be noted that many of these parameters change during the course of the computation, (for example, scale and weight values change for every GMM), and can be controlled separately and quickly. The SW (Scale Weight) register array stores the scale and weight values during the normal operation mode. The threshold array stores the different beam values, which the compare (x>y) unit uses to compare outputs at different stages of the pipeline against.

During the normal observation probability estimation process, the input feature vector is first stored in the internal register array (labeled 'Input Vector' in Figure 5.4). Mean and variance values for each senone are then fed in parallel to the mean and variance buffers m0, v0. Buffers m1 and v1 are updated with these values in the next cycle, which in turn feeds the data path. The output of each of the completed internal loop (over the entire vector length) is a gaussian. This output is scaled and weighed and passed onto the log_add unit, which performs the outer loop calculations in the log domain (mixture of gaussians). The output (ScoreOut) of this is sent to the SRAM. A crude power save mode compares each of the individual gaussians as well as the summation (o/p of the log_add module) to one of four threshold values. If the observation probability falls below a particular threshold, further calculations for that particular senones are squashed and a preset 'Constant backoff' is sent to the output.

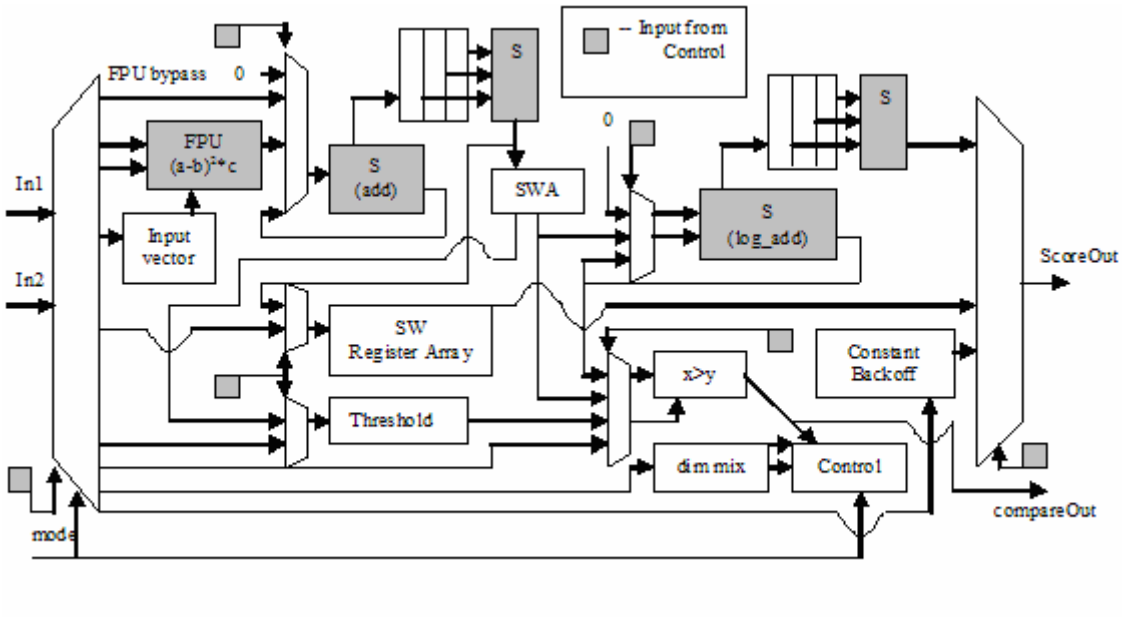



Figure 5.4 - Gaussian Estimator

The complete design has multiplexed inputs and outputs as well as trigger and handshake signals, which are not shown for the sake of simplicity.

### 5.2.2.1 Adaptation to the layer techniques

***Frame layer***

For the SDS [89] implementation, the arbiter simply feeds the GM unit every other frame, and in turn updates senones score value every other frame. The arbiter contains a counter that can be externally set and triggered. For the SDS, this counter is activated and its last bit is monitored to find out which frames are to be skipped.

For the CDS implementation, we first calculate the senone scores $b_j(O_t)$ for lookahead HMM models[89]. Two sets of scores are maintained in the SRAM memory, one for the previous calculated frame, and one for the current frame. Scores of consecutive frame models (the Euclidean distance) are compared and recorded. The maximum score $D_{max} = max_{(0<i<t)}D(i)$ is also recorded.

For the *D(t)* calculations, *dim* is first modified to fit the required vector length, and then $b_j(O_t)$ is fed to a, $b_j(O_{t-1})$ is fed to b, and '1' fed to c. *D(t)* is temporarily stored in the SW register array. $D_{max}$ is recorded by continually feeding the result of the first sumer unit to the compare unit and updating the threshold if the current output value is greater than the recorded $D_{max}$ till now.

In our implementation, instead of the normalization (Equation 5.2), we scale the max score to obtain the thresholds (.3*$D_{max}$, .6*$D_{max}$). This is performed in 2 separate runs of the Gaussian Estimator, where √.3 and √.6 are fed into 'a', 0 into 'b', and $D_{max}$ into 'c'. The outputs (for both values) from the first sumer unit as well as $D_{max}$ are sent to three of the four threshold buffers. Finally the values of *D(t)* are compared to these values using the compare_unit(labeled 'x>y' in Figure 5.4). The output of the compare unit (compareOut) is sent to the arbiter to set its internal counter in turn setting how many frames to skip.

***GMM layer***

Context Independent observation probabilities [88] are first calculated, and compared to a threshold value using the compare unit. The output 'compareOut' sets a bit in memory signaling whether the corresponding context dependent phones need to be computed or not. Else the CD scores are backed off by the CI score (ScoreOut).

***Gaussian Layer***

Codeword and cluster definitions are done offline. A pre-calculated threshold value is sent to the threshold register array. The output of the first adder unit is used to identify the codeword and neighborhood of the input vector [86]. The output of the compare unit

(compareOut) is sent to the arbiter and is used to select the codeword and its neighbors using a LUT. Now each GMM is made up of a reduced set of mixtures. Finally the *mix* parameter is varied for each senone and the observation probability of each one is calculated using the reduced set of mixture values.

# CHAPTER 6

# Viterbi Decoder

In Chapter 2, we discussed in detail the components that contribute to determining the most probable sequence of spoken words. One of these components was $P(Y|W)$ - the probability with which a given HMM could have generated a particular observation sequence $Y$. This probability can be calculated using the Forward/Backward algorithm for HMMs[1]. However it is more common to do a Viterbi search and update, even though it is more expensive. The optimal state sequence is needed at a later stage anyway, and the Viterbi search can compute the probability and uncover the optimal state sequence simultaneously. One problem with the Viterbi search is that the number of active states can exponentially increase. Hence a heuristic classically called "beam search" is used to prune unlikely triphones that have little chance of having the best score in the future time step. The combinations of the two processes is called *Viterbi beam search*. In this dissertation, when we talk about the *Viterbi search*, we imply the *Viterbi beam search*.

Another component that contributes to the recognition process is $P(W)$, which is the probability of observing the sequence of words $W$ independent of the observed signal (sequence) $Y$, which is determined by a language model. In our implementation, we integrate the language model search into the Viterbi Decoder.

The dictionary words can be phonetically broken down and are stored in either *flat* form or *lexical tree* form. Figure 6.1 shows two form of dictionary arrangement representing words – Start, Starting, Started and Startup.

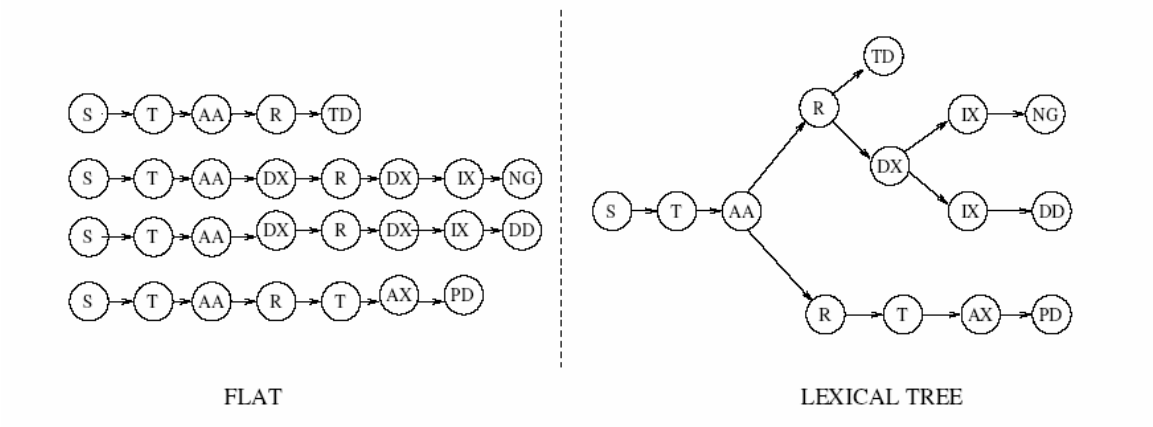



Figure 6.1 - Flat vs. Lexical Dictionary

In this chapter we will discuss two implementations of the Viterbi Decoder. The first version is based on the conventional flat dictionary style. The advantage to this was that it was easy to implement. It was easier to keep track of word endings, and easier to transition in between words. The flat dictionary style was a good initial choice allowing us to lay down the foundations for the different operating stages of the Viterbi Decoder. This design has been discussed in Section 6.1. The initial design provided us with estimates on memory bandwidth and helped us identify critical memory components and bottlenecks.

We realized early on that restructuring the memory was critical to reducing the number of operations and HMM transition evaluations per frame. This would also have a direct impact on the speed of the overall system in terms of real-time performance, as well as the power consumed. At this point the lexical tree dictionary seemed like a good option in terms of eliminating redundant calculations and also reducing the overall

memory requirements. However the main problem with switching to the lexical tree dictionary was the difficulty in word-to-word transition. This is discussed in detail in Section 6.2. We solved this problem in our second implementation of the Viterbi Decoder using an innovative 'TimeStamp' concept and a unique memory arrangement style. The improvements made to the initial design are discussed in Section 6.2. We then move on to the implementation of the final design which has been discussed in Section 6.3.

## 6.1 Implementation of the Viterbi Decoder (Flat Dictionary)

The Viterbi decoder unit computes each state transition using data from several sources including the transition score, past score, and output probability score. It uses this information to compute the probability of being in a particular state at a given time, given the sequence of inputs till that time. Monitoring the last states of triphones allows us to search for and identify potential within-word triphone transitions and word-to-word transitions during state updates using the language model.

We partition this search process into 2 stages – the State-Update-Stage and Word-Transition-Stage. In the State-Update-Stage, each state within all active triphones are updated using transition scores, past scores and output probability scores. Last states of all active triphones that are not the last in a word are also monitored for transition into the next triphone of the word. In the Word-Update-Stage, the last triphone of all words are monitored for transition into the next word. Every triphone state must be updated within the 10ms window, for the application to run in real time.

### 6.1.1 State-Update-Stage

We will first list out the different memory elements of the Viterbi Decoder in Section 6.1.1.1 going into such detail as content and total size of each element. Next we will discuss how these memory elements are accessed and manipulated to perform the search process in Section 6.1.1.2.

#### 6.1.1.1 Viterbi Decoder Memory Elements

**Triphone_Block**

The Triphone_Block is the central element in the Viterbi decoder. Every triphone of every word in the language has an entry in the Triphone_Block. Each entry maintains the current score and history of each of the triphone states, its history, the ID of each state senone, and the transition ID of the triphone. The first 3 bits are the *valid*, *second last* and *last* bits indicating whether the triphone is an active one, whether it is the second last, or the last triphone in a word respectively. The next 4 blocks of 13 bits each are *senone-ids* for each of the 4 states (each senone is 1 of 6000 entries). These are used to index into the Senone_Score block. The next 4 blocks of 32 bits each are the *current-scores* of each of the states. The next 4 blocks of 16 bits each are the *word-history* for each state, which indicate the last word from which the transition to the current word was made. The 16 bits are an index into the Identified_Words block. Since each time frame of 10ms can hypothetically lead to a new word to word transition, words that transition into the current word can differ for different time frames. The last 16 bits give the ID of the triphone – the *transition_id* (1 of 60,000 triphones), and are used to index into the Transition_Block.

Each word was assumed to have an average of 8 triphones (this is statistically true [103]). Thus a total of 60000x8 entries are present in the Triphone_Block. However since it is not possible to predict what the last triphone of any word might be (note: the last triphone depends on the first phone of the next word), the last triphone entry is simply a pointer to 50 locations in the Triphone_Block, each being one of the 50 possible last triphones of the word. With this in mind, the total number of entries increases by another 60000x50. Thus a 60K vocabulary takes about 114.405MBytes of memory. Figure 6.2 shows the Triphone_Block and Table 6.1 summarizes its contents.

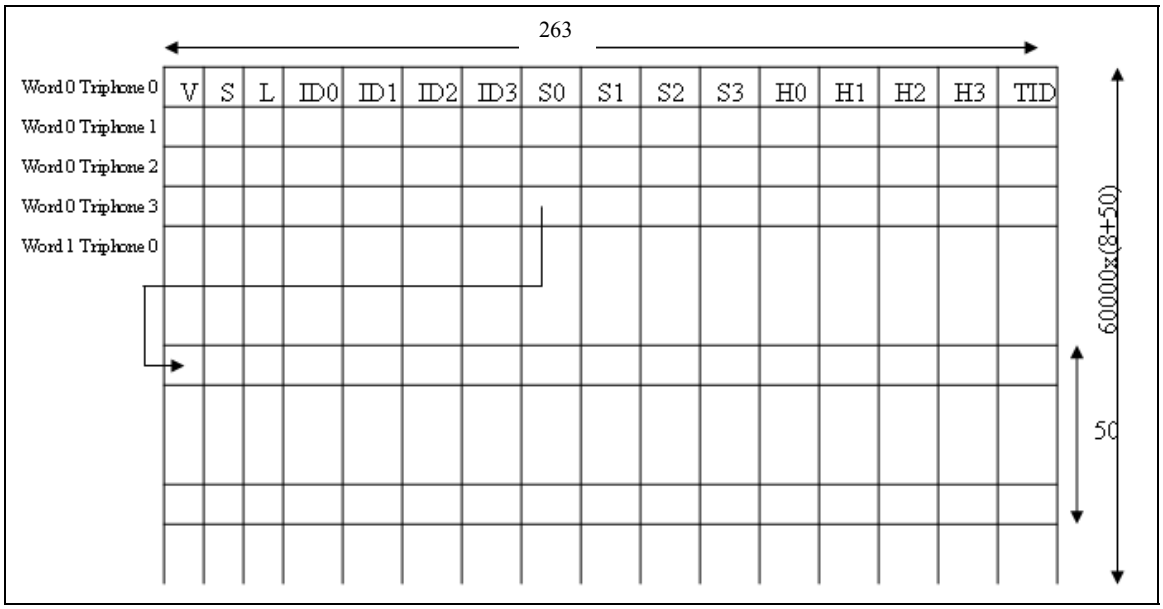



Figure 6.2 - Triphone_Block

Table 6.1 - Contents of Triphone_Block

| NAMES | SYMBOL | WIDTH |
|---|---|---|
| Valid Bit | V | 1 |
| Second-Last Bit | S | 1 |
| Last Bit | L | 1 |
| Senone-ID (4 states) | ID0, ID1, ID2, ID3 | 4x13 |
| State current-scores (4 states) | S0, S1, S2, S3 | 4x32 |

Table 6.1 (continued)

| | | |
|---|---|---|
| Word –history (4 states) | H0, H1, H2, H3 | 4x16 |
| Transition ID | TID | 16 |

**Transition_Block**

The 6 transition probabilities ($a_{00}$, $a_{01}$, $a_{11}$, $a_{12}$, $a_{22}$, $a_{23}$) for each triphone of each word are stored in the Transition_Block as 32-bit entries. They are indexed using the *transition_id*. The total number of entries is 60000 (one for every possible triphone), and the total memory requirement is about 1.44MBytes. The Transition_Block is shown in Figure 6.3 (a).

**Senone_Score**

The Senone_Score contains the output observation probabilities (senone-scores) of each of the senones that occur in the language as 32 bit scores. As indicated in Chapter 5, this is updated by the Gaussian Estimator every frame. The last bit indicates whether the senone is active or not. This is used during the feedback from the decoder unit to the Gaussian Estimator unit. The *senone-ids* are used to index into this block. Using a total senone count of 6000, the memory required is about 24KBytes. The Senone_Score block is shown in Figure 6.3 (b).

**Word_Lookup**

The unique address of each triphone is used to index into the Word_Lookup table. This table basically helps identify which word has been identified after the last state of the last triphone passes pruning. Each entry is 16 bits (used to uniquely identify 1 of 60000 words

in the dictionary), and the total number of entries is 60000x8. This takes up about 960KBytes of memory. The Word_Lookup block is shown in Figure 6.3 (c).

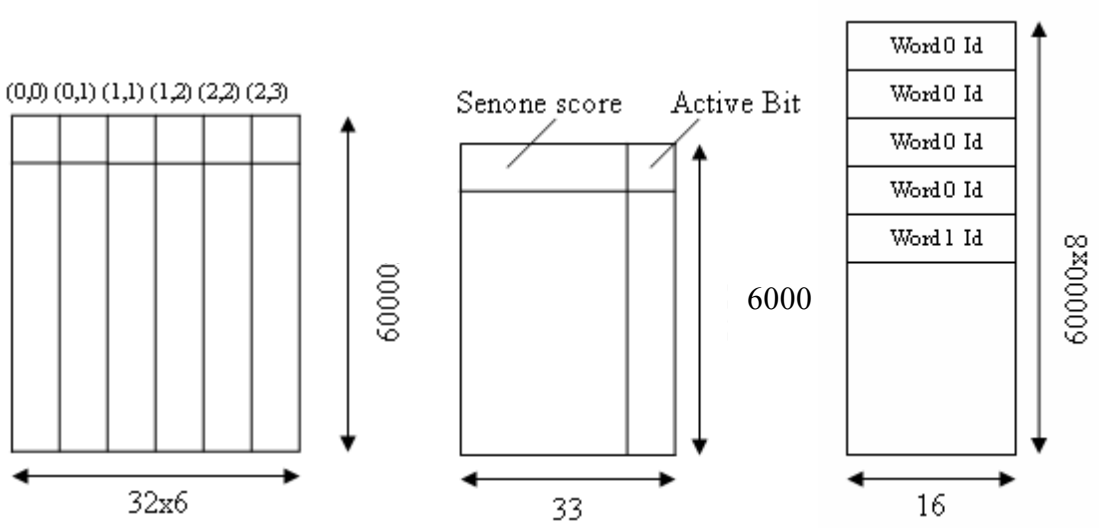



Figure 6.3 (a) Transition_Block (b) Senone_Score (c) Word_Lookup

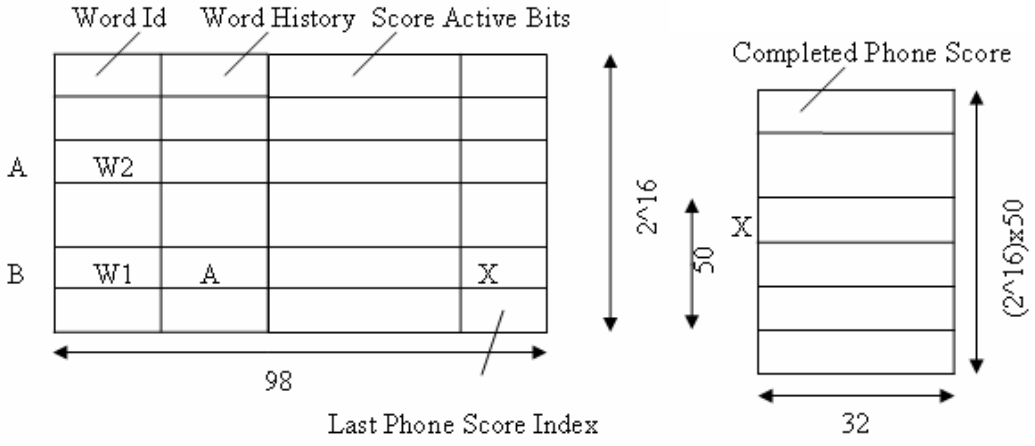
**Identified_Words**

Words that pass the pruning stage of the Viterbi decoder are inserted into the Identified_Words block as a 16-bit *Word_ID* obtained from the Word_Lookup block. The history index of the word (in this case the history index of the last state of the last triphone of the word) is also inserted into this block as a 16-bit *Word_History*. When the final backtracking takes place, this block is traced back to obtain the final 'hypothesis'. The number of entries into this block was chosen as a worst case of about 60000. In actuality only about 5000-10000 (including multiple instances of a word) will be present. The next 50 bits – the *score_active_bits* - indicate which of the 50 final triphones passed the pruning stage for this word. They are an indicator of which phones are active and need to be checked by the language models. The final 16-bit entry – the

*Last_Phone_Score_index* - is the index into the Last_Phone_Score block. The total memory size needed is about 802 KBytes. The Identified_Words block is shown in Figure 6.4(a).

**Last_Phone_Score**

The Last_Phone_Score contains the scores from the last 50 triphones of every recognized word. They are indexed using the Last_Phone_Score_index. The 16-bit *Last_Phone_Score_index* form the most significant bits of the 22-bit index. The *Last_Phone_Score_index* is the start location of the 1st of the 50 scores for a word. Adding the bit location of the valid bits from the Identified_Words block gave the other scores locations. The maximum score for all blocks is also maintained for backtracking. There are a total of $50x2^{16}$ entries each 32 bits wide. The total memory requirement for this is about 13.1MBytes. The Identified_Words block is shown in Figure 6.4(b).




(a) (b)

Figure 6.4 (a) Identified_Words (b) Last_Phone_Score

#### 6.1.1.2 The Viterbi Decode Process

Initially the pre-trained of the transition probabilities are placed in the Transition_Block. The *valid* bits for all the start triphones of all the words are also set to '1'. All senones of these triphones in the Senone_Score block are also activated (valid bit set to '1').

During the State-Update-Stage, the Triphone_Block is scanned and all triphones that have their *valid* bit set are sent for Viterbi decoding and update. During this process, the *senone-ids* of each of the states are read and are used as indexes into the Senone_Score block. The senone scores for each state is obtained from here. The *transition_id* is also used to access the Transition_Block and obtain the transition probabilities of the triphone HMM. Each update-unit then reads the senone scores, the transition probabilities and the past state score and produces a new score for the state. The number of update-units will depend both on the speed of the unit (to obtain real-time requirements) as well as memory bandwidth constraints. A typical update-unit is shown in Figure 6.5. Once the score for all states have been calculated, the updated score as well as the history bits of a state are written back. The best score for all the state calculations is maintained so that pruning can be done.

In the transition phase of this stage, the last state of every triphone is monitored. For within word triphones, if this state score is a beam distance from the threshold, the next triphone of the word is activated (its *valid* bit is set to '1'). All senones for these triphones are also activated if not already active. If all 4 states of a triphone are a beam distance away from the threshold, then the triphone is deactivated (unless it is the first triphone in a word).

For word end triphones, once the last state of the triphone(s) passes the prune threshold, the word-id number is looked up in the Word_Lookup block using the

*transition_id*, and this is entered along with the *word-history* into the Identified_Words block. Fifty entries are assigned for this word in the Last_Phone_Score block and the start location is written into the *Last_Phone_Score_index*.

The 50 active bits are updated as the Triphone_Block is progressed (within the same 10ms window), and updates are also made to the Last_Phone_Score block. The word-id, the scores from the Last_Phone_Score block and the active bits in the Identified_Words block are used along with the language model in the Word-Transition-Stage to determine the most probable next word following this word (word-word transitions). Once these words are determined, the score from the Last_Phone_Score block (now updated with some language model probabilities) are passed to the first state of the first triphone of these words, and the process repeats.

During the backtracking phase, the maximum score of the Last_Phone_Score block is used to backtrack through the Identified_Words block and provide a hypothesis.

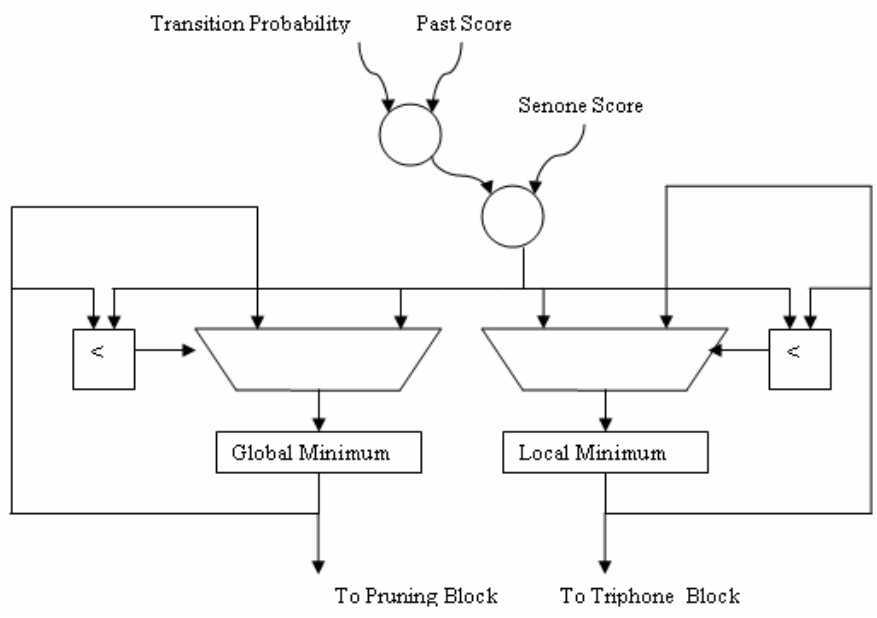



Figure 6.5 - Update Unit

### 6.1.2 Word-Update-Stage

The Word-Update-Stage does 3 levels of search using the language models– the trigram, bigram and finally the unigram search. Once again an understanding of the memory elements are required to proceed further. A typical 60K vocabulary contains about 64,001 unigrams, 9,382,014 bigrams, and 13,459,879 trigrams [22]. We will first list out the different memory elements of the Language Model Memory Units in Section 6.1.2.1 going into such detail as content and total size of each element. Next we will discuss how these memory elements are accessed to obtain the language model probability in Section 6.1.2.2.

#### 6.1.2.1 Language Model Memory Blocks

**Unigram_Block**

The Unigram_Block contains the unigram score and backoff information, as well as Bigram_Block access information. The *unigram_score* of a word is the probability of that particular word appearing within a dictionary. It is a 32-bit value that is used as part of the language model score when both the bigram search and the trigram search fails. The *unigram_backoff_score* is a 32-bit weight attached to the language model score to indicate that the both the bigram search and trigram searches have failed. Similarly the *bigram_backoff_score* is a 32-bit weight attached to the language model score indicating that the trigram search has failed. A 24-bit *bigram_block_pointer* provides the pointer (start address) into the Bigram_Block, while a 16-bit *bigram_block_access_size* indicate the number of enteries belonging to this word in the Bigram_Block. Together they are

used to access the Bigram_Block to obtain the bigram/trigram scores. The total memory required is 1.02MBytes.

**Bigram_Block**

The Bigram_Block contains a word-id tag, a pointer to its corresponding score, and Trigram_Block access information. The *bigram_block_pointer* is used to index into the Bigram_Block. It provides the location of the first allowed access into the Bigram_Block for this triphone with the *bigram_block_access_size* providing the total number of allowed accesses from this start point.

A 16 bits *word-id* tag is used during the search of 2-word sequences. The *bigram_score* is stored in the Bigram_Score and is accessed using an 18-bit pointer into this block – the *bigram_score_block_index*. We had mentioned previously that the *bigram_block_pointer* and the *bigram_block_access_size* of the Unigram_Block are used to access the Bigram_Block. Similarly the 24-bit *trigram_block_pointer* and the 16-bit *trigram_block_access_size* of the Bigram_Block are used to access the Trigram_Block. The total memory requirement for this block is 86.78MBytes of memory.

**Bigram_Score_Block**

The Bigram_Score_Block is accessed using the *bigram_score_block_index* and stores all *bigram_score* information. This 32-bit value will be used as part of the language model score when the trigram search fails and the bigram search succeeds. A separate block is used since the number of probability values used is far less than the number of entries in the Bigram_Block, and so this mapping reduces memory requirements. Based on the data file sizes from the HTK Speech ToolKit, the total required memory size is about 1MByte.

**Trigram_Block**

The Trigram_Block is accessed using the *trigram_block_pointer* and the *trigram_block_access_size*. This block holds only 2 values, the first of which is a 16- bit *word-id* tag similar to the one found in the Bigram_Block, and the second being a *trigram_score_block_index* used as a pointer to the Trigram_Score_Block (once again similar to the *bigram_score_block_index* of the Bigram_Block). The total memory size required is about 57.2MBytes.

**Trigram_Score_Block**

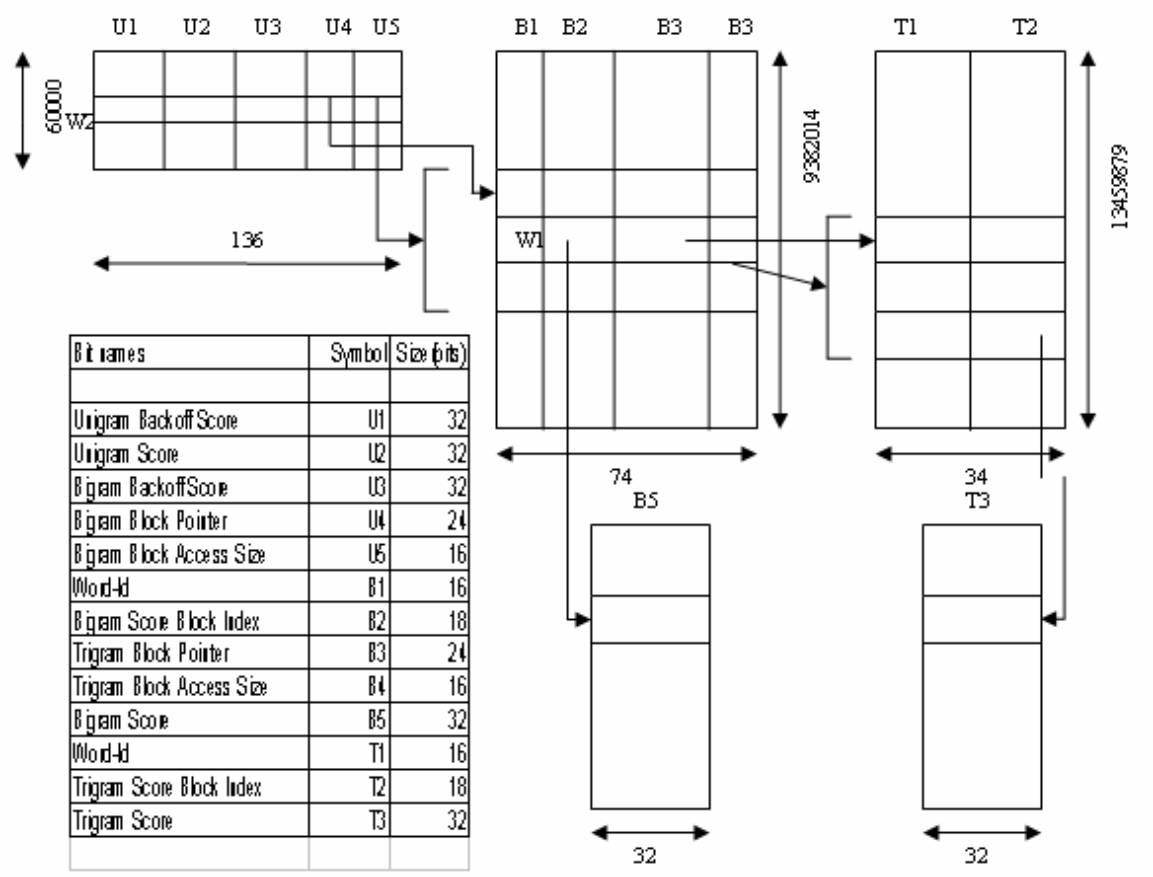
This is exactly similar to the Bigram_Score_Block, the only difference being that it holds trigram score information. Once again a 1MByte memory size should be sufficient.

### 6.1.2.2 Language Model Search

Let us denote the current recognized word as 'W1', its history word (the word from which a transition to this word was made) as 'W2' and finally any possible next words as 'x'. The language model score will be one of three quantities –

- The *trigram_score* obtained using the sequence of words 'W2-W1-x'.
- The *bigram_score* obtained using the sequence of words 'W1-x' along with a *bigram_backoff* score added to it to signify that the trigram sequence of 'W2-W1-x' did not return a match.
- The *unigram_score* obtained by using the word 'x' along with a *bigram_backoff* score and a *unigram_backoff* score to signify that the

trigram sequence of ‘W2-W1-x’ and the bigram sequence of ‘W1-x’ returned no match.




| Bit names | Symbol | Size (bits) |
|---|---|---|
| | | |
| Unigram Backoff Score | U1 | 32 |
| Unigram Score | U2 | 32 |
| Bigram Backoff Score | U3 | 32 |
| Bigram Block Pointer | U4 | 24 |
| Bigram Block Access Size | U5 | 16 |
| Word-Id | B1 | 16 |
| Bigram Score Block Index | B2 | 18 |
| Trigram Block Pointer | B3 | 24 |
| Trigram Block Access Size | B4 | 16 |
| Bigram Score | B5 | 32 |
| Word-Id | T1 | 16 |
| Trigram Score Block Index | T2 | 18 |
| Trigram Score | T3 | 32 |
| | | |

Figure 6.6 - Language Model Memory Blocks

The three language model searches are discussed next.

**Trigram Search**

In the trigram search, the Unigram_Block is first accessed using the word history (W2) as the index. The *bigram_block_pointer* and the *bigram_block_access_size* is used to index into the Bigram_Block. The *word-id* bits are compared with the current word W1 for all entries defined by the *bigram_block_pointer* and the *bigram_block_access_size*. If an entry is found and the trigram pointer and size values for this entry are valid, this means

that the trigram for the word sequence ‘W2-W1-x’ exists. The range of ‘x’ is given by the *trigram_block_pointer* and the *trigram_block_access_size* with which we access the Trigram_Block. The valid values of ‘x’ are searched for those with valid start-phones (those phones that have valid entries for the word W1 according to the *score_active_bits* in the Identified_Words Block). If valid values exist, the *trigram_score_block_index* of those entries are used to index into the Trigram_Score_Block and access the *trigram scores* for those word sequences.

**Bigram Search**

If no word sequences ‘W2-W1-x’ can be found using the Trigram search, then a *bigram_backoff* probability is added and a bigram search is conducted. In the bigram search the Unigram_Block is accessed using the word W1. If an entry is found, and if the bigram pointer and size values for this entry are valid, then this means the bigram for the sequence ‘W1-x’ exists. The range of ‘x’ is given by the *bigram_block_pointer* and the *bigram_block_access_size* with which we access the Bigram_Block. The valid values of ‘x’ are searched for those with valid start-phones (those phones that have valid entries for the word W1 according to the *score_active_bits* in the Identified_Words Block). If valid values exist, the *bigram_score_block_index* of those entries are used to index into the Bigram_Score_Block and access the *bigram_scores* for those word sequences.

**Unigram Search**

If no word sequence ‘W1-x’ is found, a *unigram_backoff* probability is added, and the Unigram search is conducted. In the unigram search, the valid values of ‘x’ are once

again looked up according to the score_active_bits and these are used to index into the Unigram_Block and obtain the *unigram_scores* for those words.

The final step is to take the scores from the Last_Phone_Score block and add the backoff scores as well as the language model scores from the unigram, bigram and trigram searches to it. The final score obtained by combining all these scores forms the starting score of the new word 'x'. This value is then compared with the threshold to see if passes the word-transition pruning requirement. If it does, it is written back as the new entry for the word 'x' into the Triphone_Block.

### 6.1.3 Analysis

Table 6.2 summarizes the memory element sizes for the state-update and word-update stages. The total memory requirements come up to about 280Mb with about 150Mb being taken up by the language model. It is obvious that the two areas that make up the bulk of memory requirements are the Triphone_Block and the language models stored in the Trigram_Block and the Bigram_Block.

Table 6.2 - Viterbi Decoder Memory Element Sizes

| State-Update-Stage | | Word-Update-Stage | |
|---|---|---|---|
| | | | |
| Triphone_Block | 114405KB | Unigram_Block | 1020KB |
| Transition_Block | 1440KB | Bigram_Block | 86780KB |
| Senone_Score | 24KB | Trigram_Block | 57200KB |
| Identified_Words | 802KB | Bigram_Score_Block | 1000KB |
| Last_Phone_Score | 13100KB | Trigram_Score_Block | 1000KB |
| Word_Lookup | 960KB | | |
| | | | |

The information stored in the Bigram_Block and Trigram_Block is part of the language model. Reducing the size of these blocks requires perhaps a different method of language model training and is outside the scope of this research. However, it is important to notice that while the language model takes up a large chunk of the memory, it is accessed only during the language model lookup. This happens only when the last triphone passes pruning and a word has completed and has been identified. Compare this with the fact that every active row in the Triphone_Block is accessed and updated every frame.

Assume that about 40% of the first 60000x8 rows of the Triphone_Block are active at any given time (this an aggressive estimate). We also know that about 3000 words also have their last 50 triphones active as well. Let us keep an aggressive worst-case estimate of 4000 for this. Also assume that about 1000 rows (from amidst these 4000 words) are activated for language model lookup every second (again a very aggressive estimate). These amounts to .0167% of total rows activated for language model lookup every frame. The Sphinx-III language model contains 60,000 unigrams, 9,382,014 bigrams and 13,459,879 trigram[13]. This indicates that while 60,000 words can potentially lead to $(60000)^3$ trigrams, only 13,459,879 or less that .00001% were trained. Allowing for the fact that many of these $(60000)^3$ combinations may be illegal and grammatically incorrect, we hike this percentage to 1%. Similarly out of a potential $(60000)^2$ bigrams, only 9,382,014 or about .2% were trained. Once again factoring in the illegal combinations, we hike this percentage to 9%.

The same statistics can be applied to the number of words with trigram, bigram and unigram scores in the test set as well, assuming that the training set and the test set

will not differ by an order of magnitude. Hence we can conclude that 1% of total words that initiate the language model lookup pass the trigram search (and use the trigram score), and 9% of the total words that initiate the language model lookup pass the bigram search(and use the bigram score). The rest 90% use the unigram score.

Next, let us look at how many accesses each search will take. During the trigram search, the word sequence 'W2-W1-x' needs to be looked up. We use 'W2' to index into the Unigram_Block and then use try to find 'W1' among the list of bigram values in the Bigram_Block. Once we find this entry, we try to find 'x' among the list of trigram entries in the Trigram_Block. Let us assume that each entry is found (if it is present) after accessing 75% of the total entries on an average. We assume the same statistic for the bigram search. The unigram search is conducted using the word as a direct index and hence no search is required.

We can now break down the accesses as follows:

- Trigram match: This involves, accessing the 16-bit *bigram_block_pointer* and 24-bit *bigram_block_access_size* of the Unigram_Block. Next, the 16-bit *word_ids* from 75% of $2^{16}$ locations in the Bigram_Block are accessed till a match is found. The 16-bit *trigram_block_pointer* and 24-bit *trigram_block_access_size* are read in. Next, the 16-bit *word_ids* from 75% of $2^{16}$ locations in the Trigram_Block are accessed till a match is found. The 18-bit *trigram_score_block_index* is read in and used to access the 32-bit *trigram_score.*
- Bigram match: This involves 2 steps – 1) Failing the trigram search for 'W2-W1-x', and 2) passing the bigram search for 'W1-x'.

- Failing the trigram search: This can happen in 2 ways – finding valid entries for 'W2-W1-$y_i$' (where $y_i$ are the valid words following the 'W2-W1' sequence) but not finding x amoung $y_i$, OR finding no valid entries for 'W2-W1-$y_i$' and proceeding directly to the bigram search for 'W1-x'.

  The former case is more severe in terms of wasted accesses and hence is the one explored. In this case, the 16-bit *bigram_block_pointer* and 24-bit *bigram_block_access_size* of the Unigram_Block are first accessed. Next, the 16-bit *word_ids* from 75% of $2^{16}$ locations in the Bigram_Block are accessed till a match is found. The 16-bit *trigram_block_pointer* and 24-bit *trigram_block_access_size* are read in. Next, the 16-bit *word_ids* from 100% of $2^{16}$ locations in the Trigram_Block are accessed and no match is found. The 32-bit *bigram_backoff* score is recorded.

- Passing the Bigram search: Once again, the 16-bit *bigram_block_pointer* and 24-bit *bigram_block_access_size* of the Unigram_Block are first accessed. Next, the 16-bit *word_ids* from 75% of $2^{16}$ locations in the Bigram_Block are accessed till a match is found. The 18-bit *bigram_score_block_index* is read in and used to access the 32-bit *bigram_score.*

- Unigram Search - This involves 3 steps – 1) Failing the trigram search for 'W2-W1-x', 2) failing the bigram search for 'W1-x' and 3) using the

unigram search results. Once again we explore only the most severe case(s).

- Failing the trigram search: This can happen in 2 ways – finding valid entries for 'W2-W1-$y_i$' (where $y_i$ are the valid words following the 'W2-W1' sequence) but not finding x amoung $y_i$, OR finding no valid entries for 'W2-W1-$y_i$' and proceeding directly to the bigram search for 'W1-x'.

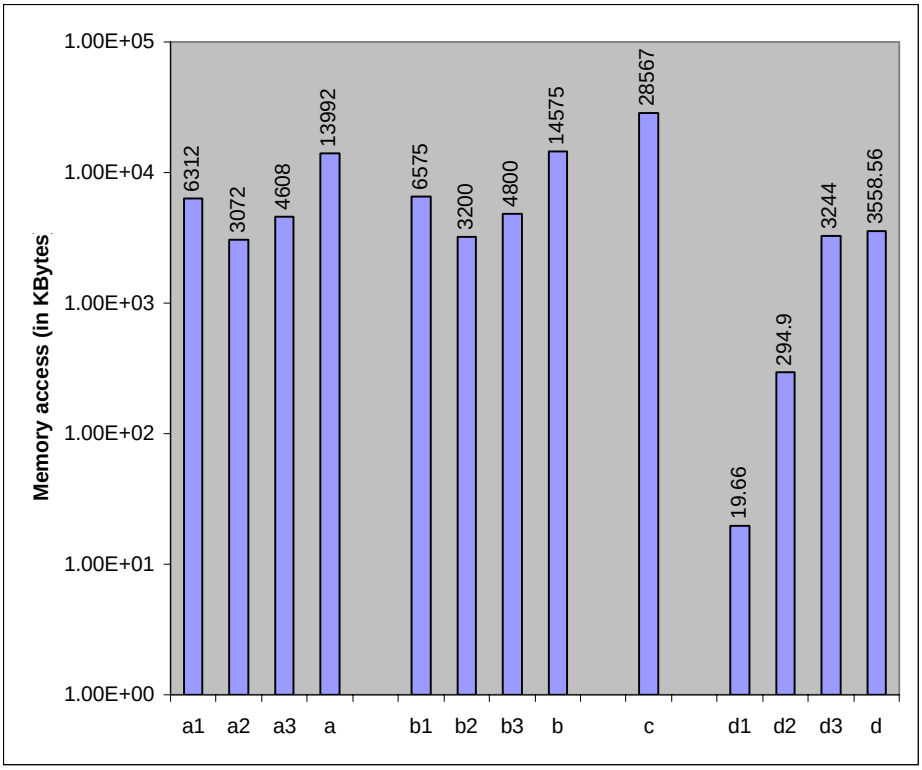
  The former case is more severe in terms of wasted accesses and hence is the one explored. In this case, the 16-bit *bigram_block_pointer* and 24-bit *bigram_block_access_size* of the Unigram_Block are first accessed. Next, the 16-bit *word_ids* from 75% of $2^{16}$ locations in the Bigram_Block are accessed till a match is found. The 16-bit *trigram_block_pointer* and 24-bit *trigram_block_access_size* are read in. Next, the 16-bit *word_ids* from 100% of $2^{16}$ locations in the Trigram_Block are accessed and no match is found. The 32-bit *bigram_backoff* score is recorded.
- Failing the Bigram search: Once again, the 16-bit *bigram_block_pointer* and 24-bit *bigram_block_access_size* of the Unigram_Block are first accessed. Next, the 16-bit *word_ids* from 100% of $2^{16}$ locations in the Bigram_Block are accessed and no match is found. The 32-bit *unigram_backoff* score is recorded.
- Using the Unigram Search values: The 32-bit *unigram_score* is recorded.

This discussion has been summarized in Table 6.3. The numbers indicate the number of bits needed *per language model lookup of that type* and not *per frame*.

Table 6.3 - Language Model Search Access Breakup

| LANGUAGE MODEL SEARCH | | | | |
|---|---|---|---|---|
| Search | Steps | Access Breakup (in bits) | Step total(bits) | Total (bits) |
| Trigram Search | Trigram Pass | 16 + 24 + .75 x 2^16 x 16 + .75 x 2^16 x 16 + 18 + 32 | 1572954 | |
| | Total Trigram Search | | | 1572954 |
| Bigram Search | Trigram Fail | 16 + 24 + .75 x 2^16 x 16 + 1.0 x 2^16 x 16 + 32 | 1835080 | |
| | Bigram Pass | 16 + 24 + .75 x 2^16 x 16 + 18 + 32 | 786522 | |
| | Total Bigram Search | | | 2621602 |
| Unigram Search | Trigram Fail | 16 + 24 + .75 x 2^16 x 16 + 1.0 x 2^16 x 16 + 32 | 1835080 | |
| | Bigram Fail | 16 + 24 + 1.0 x 2^16 x 16 + 32 | 1048648 | |
| | Unigram Pass | 32 | 32 | |
| | Total Unigram Search | | | 2883760 |

We now need to apply the percentage of such accesses taking place each frame. Remember that as per our prior discussions, about 1000 rows per second (or per 100 frames) initiate the language model lookup with frequencies of 1%, 9% and 90% accessing the *trigram_score*, *bigram_score* and *unigram_score* respectively. Figure 6.7 shows the breakup of the number of bytes accessed *per frame* for updating the first n-1 triphones of the words(those that are active – about 40% of total), the set of 50 $n^{th}$ triphones of words (those that are active - worst case estimate of 4000 total), and the language model lookup. Note that the scale is logarithmic.




a1 – First n-1 triphones (Triphone Block)
a2 – First n-1 triphones (Senone Scores)
a3 – First n-1 triphones (Transition Scores)
a – Total memory access for updating first n-1 triphone states (a1+a2+a3)
b1 – First set of 50 nth triphones (Triphone Block)
b2 – First set of 50 nth triphones (Senone Scores)
b3 – First set of 50 nth triphones (Transition Scores)
b – Total memory access for updating first set of 50 nth triphone states (b1+b2+b3)
c – Total memory access for updating triphone states(a+b)
d1 – Trigram access
d2 – Bigram access
d3 – Unigram access
d – Total Language Model access(d1+d2+d3)

Figure 6.7 - Memory access per frame

It is obvious that the language model lookup (and hence its corresponding memory accesses) form a very small part of the total memory accesses as compared to updating the Triphone_Block. In other words, while the language model *memory* requirement is large, its *memory bandwidth* requirement is quite small.

We must therefore concentrate on the other Triphone_Block and try to reduce its size or restructure it. Reducing the size of the Triphone_Block is beneficial in many

ways. Since it contains the core memory elements that are processed every frame, reducing its overall size implies reducing the number of calculations and operations involved. Reducing the number of operations increases the speed at which each frame can be processed as a whole and hence it improves the real-time performance of the system. The reduced number of calculations also translates to reduced power consumption. Smaller memory elements imply fewer blocks needing to be updated, fewer reads and writes and reduced bandwidth requirements.

## 6.2 Improvements to the initial design

### 6.2.1 Switching to the lexical tree structure

In our previous implementation we adopted a flat vocabulary structure. The advantage to this was that it was easy to implement. It was easier to keep track of word endings, and easier to transition in between words. However it required a large amount of memory and also led to many redundant calculations. In spite of this, the main reason for not choosing the lexical tree was that if two words that shared the same triphones at the start had two different word histories i.e. two different words transitioning into them it was not possible to continue both possibilities since they could only keep one or the other. Assume for example that 'Is-Starting' and 'Has-Started' are 2 possible transitions *that occur at the same time*. With the flat dictionary the start score of 'Starting' would be the end score of 'Is' combined with the probability of an 'Is' to 'Starting' transition. Similarly the start score of 'Started' would be the end score of 'Has' combined with the probability of a 'Has' to 'Started' transition. However with the lexical tree structure, the start score of 'S-T-AA' would be either of the two above but not both (unless 2 copies are instantiated which would defeat the whole purpose of collapsing the dictionary) and so one of the paths is eliminated. Thus word-to-word transitions become a problem.

Another issue was the memory structure for the lexical tree form and within word triphone to triphone transitions. We needed to come up with a simple method of being able to transition from one triphone to another within words. In the flat tree arrangement, this was simple enough because the next triphone was simply the one following the current one in the Triphone_block. However with the new scheme, a single triphone may have multiple successor triphones, and having a separate look up table (LUT) and

elaborate transition schemes would complicate the design while also adding power for all the extra processing.

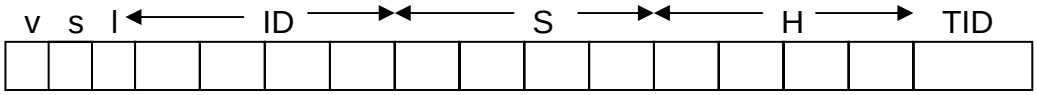
However if these problems could be solved, the lexical tree structure afforded many advantages. It eliminated redundant processing of words that shared start triphones. The scores for the same set of start triphones would not have to be evaluated for every word; rather one set would suffice. Only the end triphones would be unique and would need to be evaluated. This reduction in calculation would lead to an immediate reduction in required memory bandwidth, which is a big concern especially in a real-time bandwidth constrained design like this. We would also achieve huge power savings both from the reduced number of calculations required as well as the reduced number of reads and writes to memory.

### 6.2.2 Implementing the tree structure

In our new implementation, we have solved both these problems, and are able to take advantage of the lexical tree structure. As mentioned before most of the old design has been preserved with some fundamental changes in the new one. Thus instead of pulling up the entire design we simply highlight the differences from the old design in these discussions.

Figure 6.8 shows how the memory structure for the Triphone_block has changed. One row each of the old and new structures are shown. Table 6.4 summarizes the different components of the Triphone_Block (old and new).

**Old**




**New**

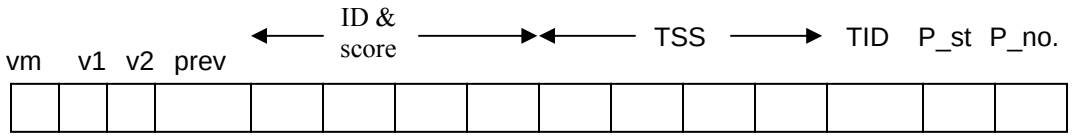



Figure 6.8 - Triphone_block row (old and new)

Table 6.4 - Triphone_block row bits (old and new)

| OLD DESIGN | NEW DESIGN |
|---|---|
| v-'valid' bit (1)<br>s – 'second last triphone' bit(1)<br>l – 'last triphone' bit(1)<br>ID – Senone ID(4x13)<br>S – State current score(4x32)<br>H – Word History (4x16)<br>TID – Transition Block index (16)<br><br><br><br><br><br>Total - 263 | v-'valid' bit (1)<br>v1,v2- Used to indicate the type of triphone. (2)<br>00 - 1st triphone of the word<br>01 - 2nd to n-2th triphone of the word<br>10 - n-1th triphone of the word<br>11 - Last triphone of the word<br>prev – Pointer to previous triphone (18)<br>ID & score – Senone ID & State current score (4x13) + (4x32)<br>TSS – Timestamp Start (4x11)<br>TID – Transition Block index (16)<br>P_st – Start position of next set of triphones (18)<br>P_no. – Number of next triphones (6)<br><br>Total - 285 |

### 6.2.2.1 Handling within word transitions – Memory structure for the lexicon tree

As mentioned before, within word transitions needed to be implemented in a simple way (a generic non-mapping scheme must be developed) if we were to succeed in implementing the lexical tree structure. We did this through the addition of 2 sets of bits at the end of each triphone –

P_st - which is a 18 bit address of the start position of the next *set* of triphones.

P_no. - which is a 6 bit number that indicates how many next triphone transitions are possible from the current triphone.

An example is shown in Figure 6.9. It should be noted that to keep things simple, the example has been shown with letters of the words rather than the actual triphones. This is an example of a set of 3 words ‘verify’, verified’ and ‘verification’. Note that all bits have not been shown.

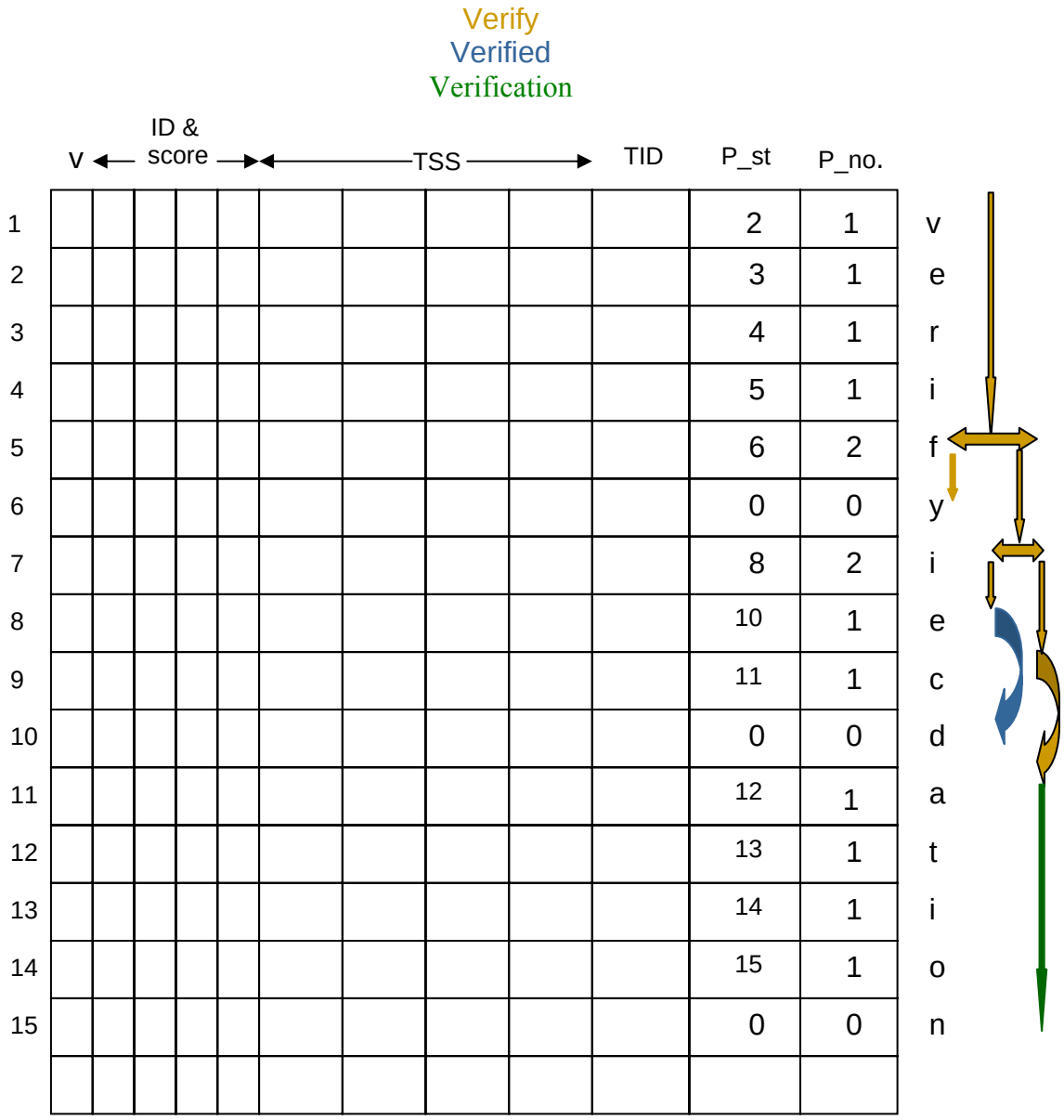



Figure 6.9 - Within-word transition example

The example shows that any type of lexical tree can be handled by this memory structure. As a coarse estimate, for this example, we would be using 15*285 bits = 4275 bits as opposed to (6+8+12) *263 = 6838 bits.

Estimates for the memory reduction for a 60000-word vocabulary are discussed later in this document. Also note that the example shown above is a slightly simplified version of the actual one. For example once 'Verification' is complete, row 14 would not point to simply row 15, but actually a set of 50 locations each being the 50 possible last triphones of the word.

#### 6.2.2.2 Handling word to word transitions – Timestamps

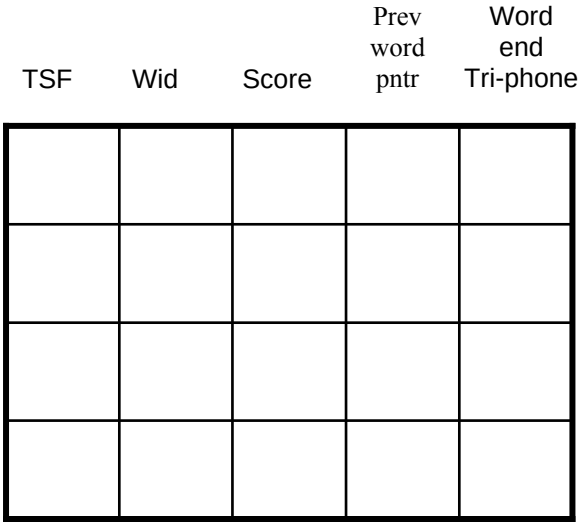
Word to word transitions were a problem when using the lexical tree. Language models gave the probability for transition between single words and not a single word to a group of words. Hence conventionally when a word (say A) completes, we then look at the language model for the probability of word B occurring after A, and combine this language model probability and the observation probability density of B (GE score of B) with the end score of A. This score is then assigned as the score of the first state of the first triphone of B, and we say that A 'transitions into' B with this score. However with a lexical tree, we do not know the language model probability of transitioning into a *group* of words from one word. More importantly, it would be difficult to separate out individual probability once shared triphone states are completed and the individual words start separating out.

We solve this problem by the use of timestamps. Timestamps are basically a counter that runs in the background. This counter is updated every 10ms, so that we have

a new counter value for every frame. Obviously this timestamp has to be unique for every timeframe, yet the counter value cannot be infinite. Hence we make the reasonable assumption that a decision about all words that are actively being considered will be made at least (worst case estimate) 10sec after the start of that word. We would need a 10bit counter to be able to give a unique counter values for 10sec. By adding an extra bit to this, we get twice the required count value. We then use the MSB and MSB-1 to monitor the current value of the counter and invalidate entries that are older than 10secs. For example if the current count value is 00XXXXXXXXX , all values with timestamp 01XXXXXXXXX are invalid. (We assume the counter counts up in which case all words with timestamps '01XX..' come more than 15 secs before '00XX..'

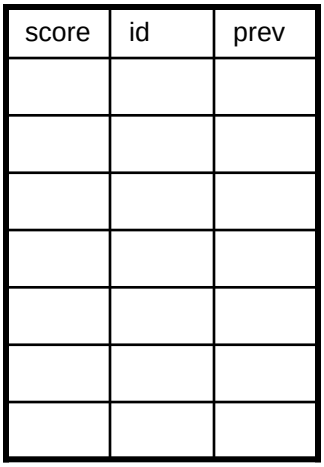
Now, instead of assigning the score of the first state of the first triphone of a new word at the beginning (i.e. while transitioning *into* it), we assign it at the end (after it's last state has passed pruning). When a word begins, we do not assign a transition score to it at the beginning. Instead we process the word as though it is the start of a new sentence (i.e. with no history score). But we assign a Start Timestamp (TSS) to it when it begins. We also assign a timestamp to a word once it gets completed (Finish Timestamp TSF).

Let us say that a current path consists of n words, the $n^{th}$ word being under consideration now. The $n\text{-}2^{nd}$ and earlier words are stored in the Identified_words block in the form of a linked list, with each entry giving the word id, the path score till that word, and a pointer to the previous word (this pointer is simply an index to the previous word in the path which is stored somewhere in the Identified_words list). The $n\text{-}1^{th}$ word is stored in the temp_list along with its Finish Timestamp, a pointer to the $n\text{-}2^{nd}$ word in the Identified_words list as well as the index of its last triphone.




TSF – Timestamp Finish (11)
Wid – Word ID (16)
Score – score (of the path upto this word) (32)
Prev word pntr – Index of previous word in the Identified_words list (13)
Word end triphone – Index of the last phone of this word.(16)

(a)




Score – score of the path till this word (32)
Id – Word Id(16)
Prev – Index of previous word (in the Identified_words list)(13)

(b)

Figure 6.10 - (a) temp_list (b) Identified_words

Once the $n^{th}$ word completes (its last triphone state passes pruning), we compare its TSS with all TSF's in the temp_list. If there is a match, this means that the word in the temp_list finished at the same time that the current word started, and so a transition could have been possible between these 2 words. Next, we recall that every word has 50 possible last triphones (Remember, the last triphone of any word is made up of its last 2 phones + the first phone of the next word. Since there are 50 possible first phones, we have 50 different last triphones for each word). Once the timestamps match, we compare the index of the last triphone of the word in the temp_list with the index of the first triphone of the current word. If this matches as well, we use the $n^{th}$, $n\text{-}1^{th}$ and $n\text{-}2^{nd}$ words

to access the language model. We then combine the language model probabilities along with the score of the n-1$^{th}$ word/and the individual score of the n$^{th}$ word to assign the final score for the n$^{th}$ word. If this passes pruning, this word is inserted into the temp_list, and the n-1$^{th}$ word is copied to the Identified_words list. Thus using these 2 compare techniques we can link words together.

Let us go back to the previous example (‘Is starting’ and ‘has started’). We would calculate the scores for ‘Started’ and ‘Starting’ individually with no scores associated with them at the start of these words. They would each be associated with a timestamp (in this case since they both start at the same time, the timestamp would be the same). Once each of these words completes and passes pruning, their start timestamps would be compared with all end timestamps of words in the temp_list. Let us assume 3 words ‘Is’, Has’ and ‘Dog’ give a match (i.e. their end timestamps are the same as the candidate word start timestamps). Now comparing the start and end triphones, both ‘Starting’ and ‘Started’ would eliminate ‘Dog’. For the word ‘Started’, the language model would eliminate ‘Is’ and so only ‘Has started’ would pass through. ‘Started’ would then combine its own score with the score of ‘Has’ and the language model score to get its final score which would be inserted along with its entry into the temp_list. Similarily ’Starting’ would only inherit from ‘Is’.

Another possibility if the 2 compare techniques fail is that the current word is the start of a new sentence. To keep this possibility alive, once a word completes and fails both compares, it is inserted into the temp_list with a null pointer to the Identified_words list. This means that it has no predecessor.

All entries within the temp_list that have invalid timestamps (as discussed previously) would be eliminated, effectively pruning paths that show no promise.

### 6.2.3 Modified Triphone_block

Part of the modified Triphone_block is shown in Figure 6.11 illustrating how the final 50 initialization takes place. Once the second last triphone of a word passes pruning, a Free Memory List is looked up to find available spaces for initiating the final 50 triphones. The valid bits and the pruning blocks continually update this list. The last 50 triphone do not need to point to any other triphones and so do not need the P_st and P_no. bits.

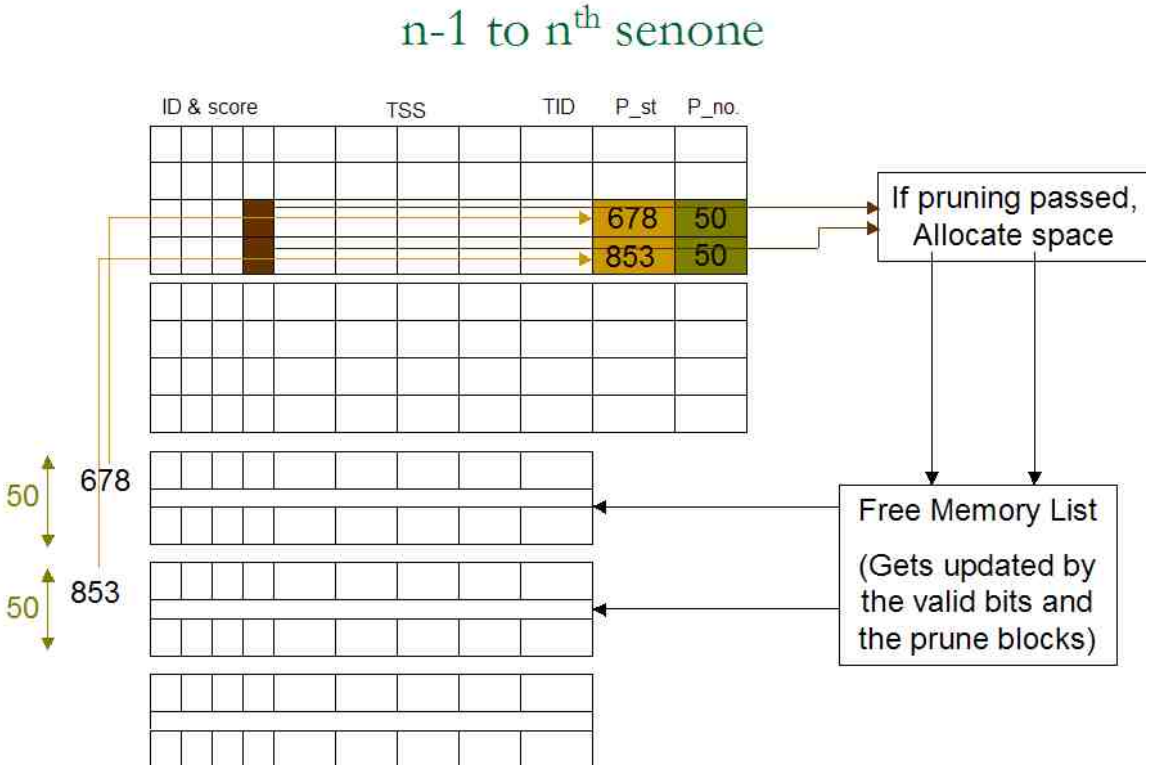



Figure 6.11 - Final 50 initialization

### 6.2.4 Memory Savings

It is safe to assume that each word in the vocabulary would have at least 4 'forms' of it (Verify, Verified, Verification, Verifying etc). This assumption is true even if we consider other languages. Assuming an average of 8 senones per word and assuming at least 5 of these can be shared by these 4 words, we have a total of 5 + 3x4 or 17 triphones as opposed to 8x4 = 32 previously. Thus for a 60000 word vocabulary, we have 60000/4 x (5+3x4) = 255000 rows for the first n-1 triphones as opposed to 60000x8 = 480000 rows. Each of these rows will have 285 bits as opposed to 263 bits (as shown in Figure 6.8). Let us call these as type 'A' rows.

We also know that only about ~3000 words (worst case estimate 4000) pass pruning all the way to the final triphone at any given time. Hence for the last triphones, we have about 4000x50 = 200000 rows as opposed to 60000x50 = 3000000 rows. Let us call these type 'B' rows.

Total memory for the Triphone_block in the new scheme is 285x255000 + (285-18-6-2)x200000 = 15.56MBytes as opposed to 114.405 MBytes from the previous implementation. The Last_Phone_Score block is no longer required giving an additional saving of 13MBytes. The completed words list would reduce by a small amount since we no longer need to keep track of the last phone score index. The additional memory needed for the temp_list is about (11+16+32+13 (13 bit index into Identified_Words list)+16)x(~5000). Thus total memory required for this block is 88x5000 = ~55KBytes.

### 6.2.5 Implications of new implementation

There are many advantages to the new implementation as compared to the older one:

- The memory savings alone may not be of importance especially since with the language model (which is a purely DSP aspect of the system), the total required memory will still be over 100MBytes. What is important however is that the memory savings come from an area that contains very *volatile data*. The Triphone_block is a block that is updated *every* frame and hence reducing the *amount of data* that needs to be updated by 7x is significant. Note that only a *fraction* of the memory holding the language model is looked up every frame, and that too only a 'read' operation takes place. We achieve significant savings in the number of reads and writes to the Triphone_block.
- This translates to a direct savings in terms of power (both due to the reduced calculation as well as due to the reduced memory operations).
- Since fewer calculations take place, redundant calculations are eliminated and lesser data need to be updated every frame, this scheme significantly improves the real-time performance of the system.
- We also free up the memory-bandwidth bottleneck that may be encountered from using slower memories.
- In our previous implementation since each word inherits the score of the path till now from the previous word in the path, *only one transition* to a word at any given time is possible. This means that say 2 paths try to transition into a word at any time, there will be a contention and only one of these paths will be allowed to continue. We solved this problem in the previous implementation by allowing duplicate copies of a word in the Triphone_Block.

With the new design, since we only allow histories to be inherited *after* the word has completed, no paths are lost. Any number of paths can be continued now since the score of the word can simply be added to each of the paths at the end of the word completion, instead of the path score being added to the word before the word beginning.

- Word to word transitioning becomes easier now we no longer need to keep track of word and path histories. Assigning the right histories to the right words is critical to the application and is also something that is easy to get wrong. Now with the new scheme, we only to need to assign a global timestamp to the words when it begins and ends. This is much easier to keep track of and assign.
- There is uniformity in the memory structure unlike the older memory structure where the last triphone row of each word simply served as a pointer to the last set of 50 triphones.

There are some tradeoffs to the scheme as well:

- Additional hardware would be required to perform the compare operations. It may be required to develop a hashing function to complete this operation in real time. This additional hardware translates to additional power as well as area. Since a mux/comparator array is all that is required, we do not assume the power and area addition would enough to degrade performance/overhead.
- Unlike the old scheme dynamic addition of words would not be possible. We would need to re-map the Triphone_block statically and preload it every time a new word is added. To circumvent this, we would be adding a small section of memory in the Triphone block to implement a mixed flat-lexical tree type

arrangement. Any new words added would be added as a tree with only one branch thus being modeled as an inefficient tree structure. It should be noted that this is only an 'On the go' problem. We could still perform the re-mapping statically and reload the Triphone_block, but this would have to be done offline.

- In the previous implementation it is possible that many words do not reach their last triphone due to the history scores and are pruned off. This leads to many paths being eliminated early and hence reduced number of calculations being performed. In the new scheme if the GE scores for a word are high, they will transition to their last triphone, and will be pruned only *after* their history scores are added in. This could potentially lead to a large number of calculations if many paths are kept active by the GE score. Statistically this has a low chance of happening, but is entirely dependent on speaker dialect, noise, as well as how well the acoustic model are trained.

## 6.3 Implementation of the Viterbi Decoder (Lexical Tree Dictionary)

The Viterbi Decoder implementation can be broken up into 4 phases. In Phase 1 the first n-1 triphones are updated. In Phase 2 the last 50 triphones are initiated. In Phase 3 the last 50 triphones are updated. Phase 4 is the language model lookup phase. These 4 phases are discussed in Section 6.3.1. Figure 6.12 shows the VITERBI DECODER and its interfacing with memory. The individual modules are discussed in detail in Section 6.3.4.

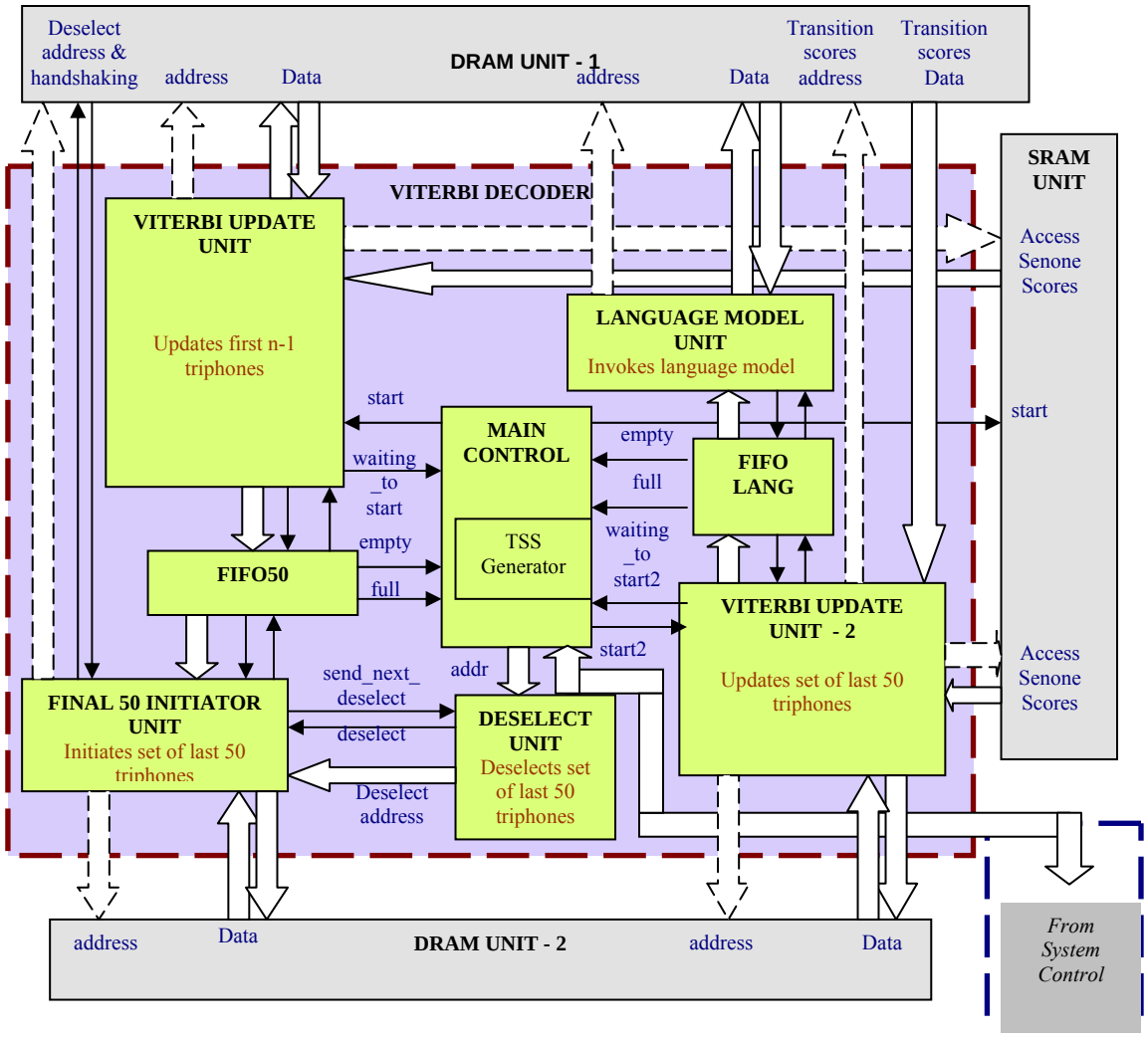



Figure 6.12 - Viterbi Decoder and its interfacing

### 6.3.1 Overview of the implementation

During Phase 1, the *first n-1 triphones* are read into the VITERBI DECODER and the new values for each of the state scores are calculated. The corresponding timestamps for each of the four states are also updated. Three different processes are initiated at this point.

Firstly the new scores (and their corresponding timestamps) are *written back* into memory – this process is called 'Writeback'. If all four states of any triphone do not pass pruning, that triphone is deactivated, unless it is the first triphone of the word (The first triphone of a word is never deactivated because that word could be spoken in any frame, and it is the first triphone that will pick up on this and start a new word possibility when this happens).

Secondly if the last state of the triphone passes pruning, and the triphone is one of the first n-2 triphones, it then *activates* the triphones that follow it in the word(s) (sets the *vm* bit of the successor triphones to '1') – this process is called 'vm-Activation'. By keeping only the relevant triphones activated, we speed up the update process and also save power due to reduced computations.

Finally if the last state of the triphone passes pruning, and the triphone is the n-1$^{th}$ triphone, we need to *initiate* the last 50 triphones. The first step in this process is to insert the information required for this initiation into a FIFO (called FIFO50). This process is called 'FIFO-insertion' and is also done in Phase 1.

Phase 1 is done entirely by 'VITERBI UPDATE UNIT –1', the details of which are discussed in Section 6.3.4.1. Phase 1 is shown in Figure 6.13.

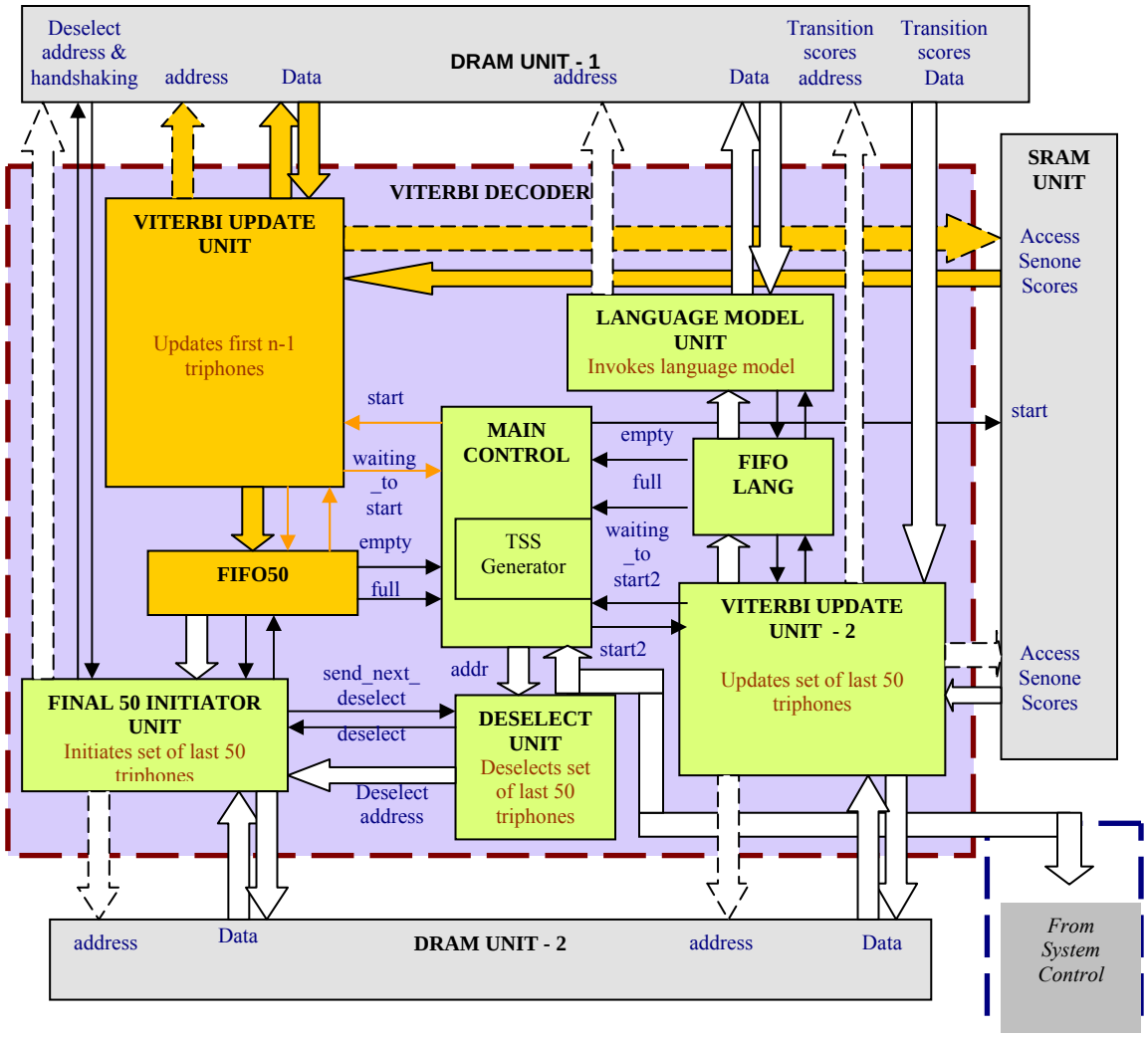



Figure 6.13 - Phase 1

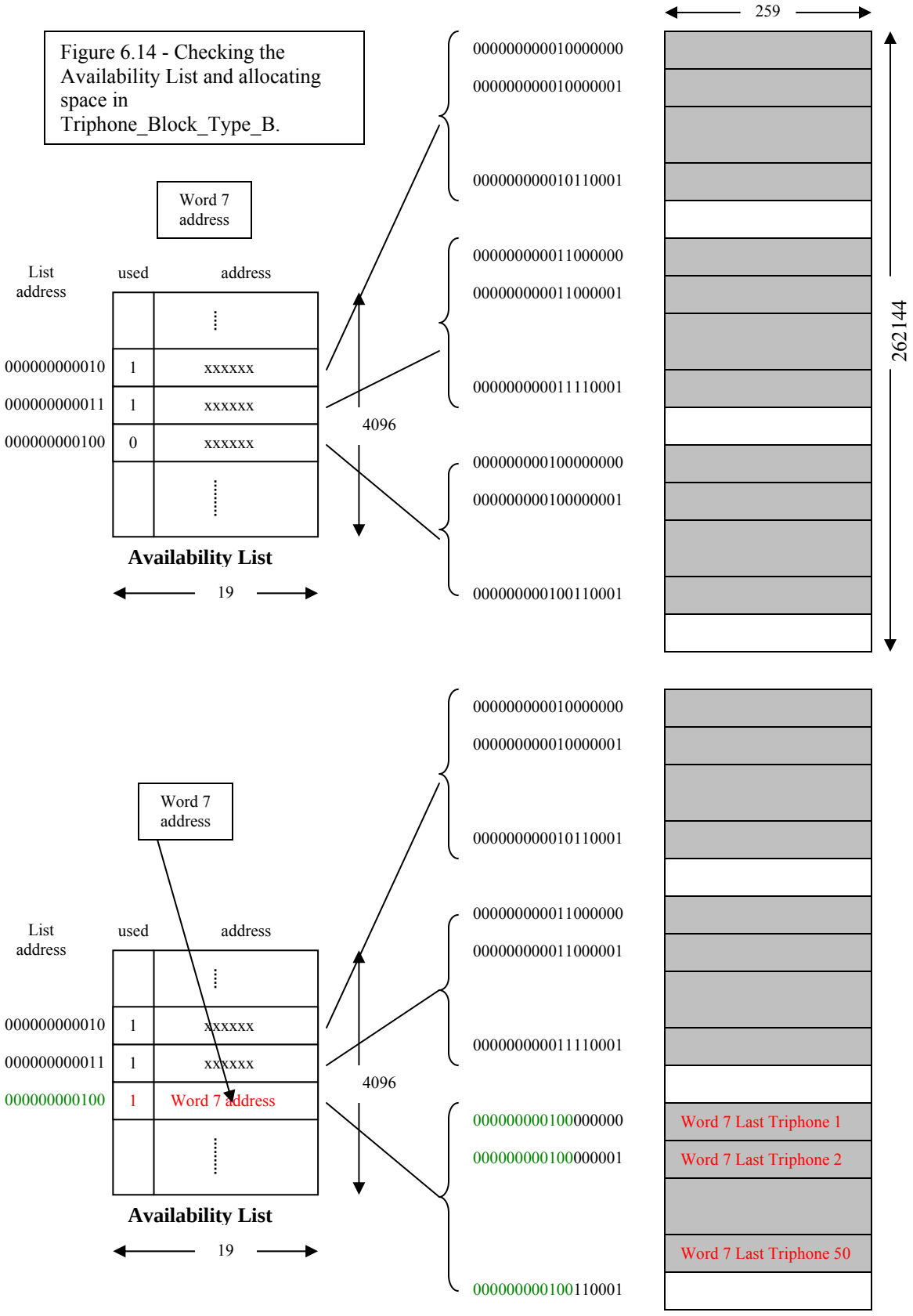
Placing a FIFO in between Phase 1 and Phase 2 essentially decouples the 'updating and writeback' process from the 'initiation' process. Updating the triphones takes place quickly, but the active triphones from the entire 'Triphone_block' needs to be updated and written back and so this process takes place frequently. Initiation of the last 50 triphones is a lengthy process, but is not very frequent (happens only if the last state of the n-1$^{th}$ triphone passes pruning). The two processes also access two different memory units. Phase 1 accesses DRAM UNIT 1 and Phase 2 accesses DRAM UNIT 2 (this is explained in detail in Section 6.3.2 and by Figure 6.13 and Figure 6.15). The FIFO allows

for both processes to take place in *parallel* thus hiding the processing time (including the memory access) of Phase 2.

Note that there are 60k words in the dictionary. If space were allocated for all of the 50 triphones for each of these 60k words, we would need 60k x 50 = 3000000 rows. This translates to about 97.125 MBytes of memory required. Instead we use the fact that only a small number of these last triphone states for all the words are active at any given moment. We adopt a dynamic allocation scheme whereby all the words share a much smaller common space (called Triphone_Block_Type_B), which is allocated to words depending on which ones are currently nearing completion. Memory savings using this scheme had been discussed in Section 6.2.4.

The actual initiation is done in Phase 2. In this phase, data is first extracted from FIFO50. Next the 'Availability List' is checked for free space. As shown in Figure 6.14, each location in this list maps to 50 locations in the Triphone_Block_Type_B and gives information about what the active words are, which sections they are currently occupying, and also which sections are currently free. For example say 'word 7' needs its final 50 triphones initiated. The availability list is checked for free space. In this example, List address 000000000100 is free. This List Address is made unavailable by setting the *used* bit to 1 and placing the address of Word 7 (which is the data extracted from FIFO50) into *address*. Six bits are then appended to this *List Address* and used to access the memory locations '000000000100000000' through to '000000000100110001'. These are the locations that have now been allotted to Word 7. These locations are populated with the relevant information for each of the 50 last triphones of Word 7, and are ready for processing during the next run (next frame).

Figure 6.14 - Checking the Availability List and allocating space in Triphone_Block_Type_B.



The *set50* bit of the second last (n-1$^{th}$) triphone of each word indicates whether or not the word has been allotted its last 50 triphones or not. It is set to '1' if it has, and '0' otherwise. This bit also dictates what Phase 2 should do with the information in FIFO50. The initiation explained previously takes place if the bit was previously '0'. If it was '1', this means that the word already has a set of 50 locations allotted to it, and all that needs to be done is to *reactivate* any of the 50 locations that had become deactivated due to pruning. Hence, Phase 2 compares the Data in FIFO50 with each of the entries in the Availability List. Once it finds a match, it uses the List Address to access and reactivate the 50 locations. Phase 2 is done entirely by FINAL 50 INITIATOR UNIT, the details of which are discussed in Section 6.3.4.2. Phase 2 is shown in Figure 6.15.

During Phase 3 the *last set of 50 triphones* are read into the VITERBI DECODER (from Triphone_Block_Type_B) and the new values for each of the state scores are calculated. Similar to Phase 1, two processes are initiated. Firstly the new scores are *written back* into the Triphone_Block_Type_B memory – this process is called 'Writeback'. The current triphone is also if all the states do not pass pruning. Secondly if the last state of the triphone passes pruning, this means the *word has completed* and we invoke the *language model* to add word-to-word transition probabilities to it. The first step in this process is to insert the information required for this language model lookup into a FIFO (called FIFO_Lang). This process is called 'FIFO-insertion' and is also done in Phase 3.

These processes are done by VITERBI UPDATE UNIT –2, the details of which are discussed in Section 6.3.4.3. After every set of 50 triphones is updated, a DESELECT UNIT checks to see if all the 50 triphones are active.

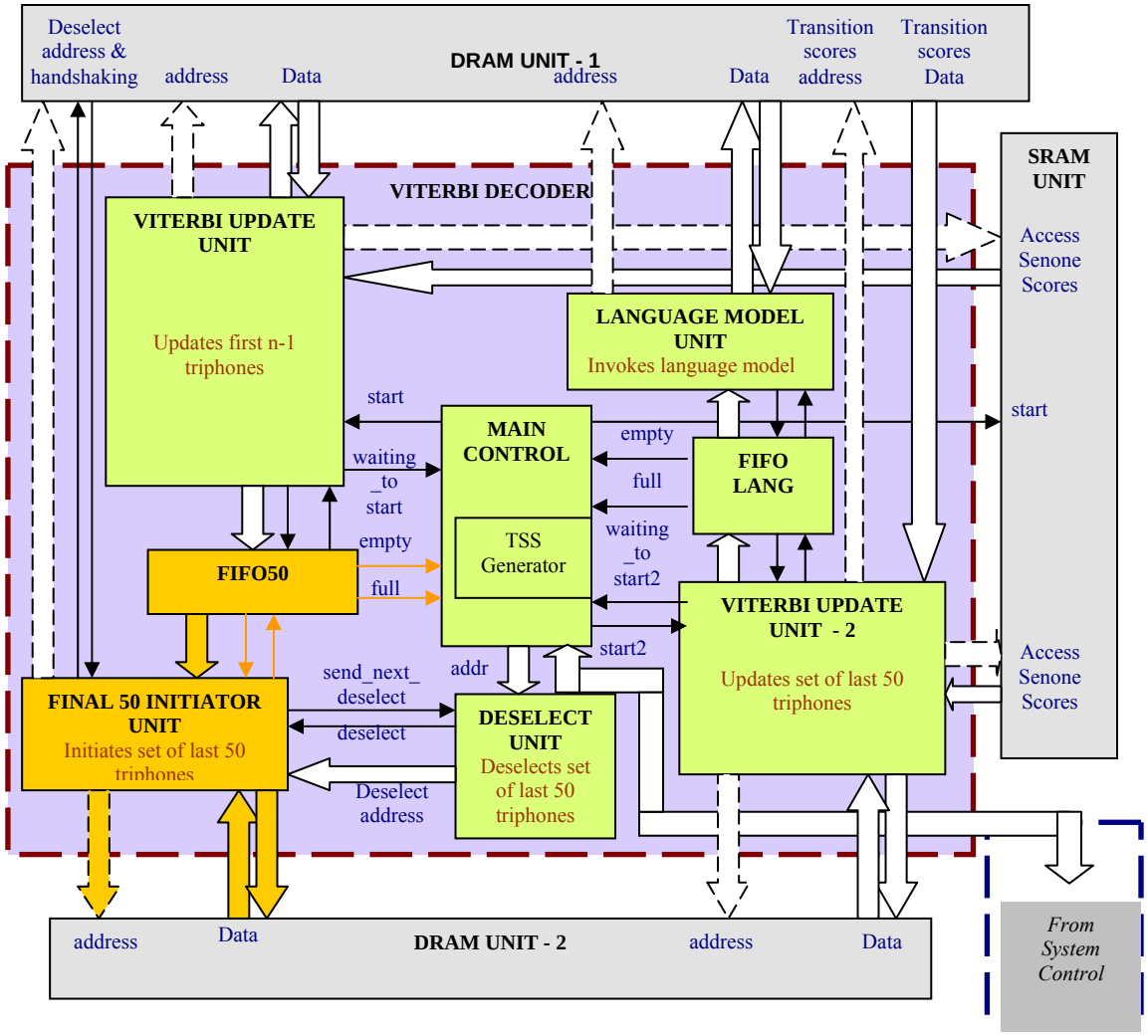



Figure 6.15 – Phase 2

If they are all inactive, the DESELECT UNIT signals the FINAL 50 INITIATOR UNIT to *free up* these memory locations. The FINAL 50 INITIATOR UNIT sets the *used* bit at that location in the Availability List to '0' indicating that the set of locations have now become free. For example, if the addresses in the Triphone_Block Type B from '000000000100000000' through to '000000000100110001' are inactive (meaning their *vm* bits are '0'), then the location in the availability list corresponding to these set of addresses (which is the 1st 12 bits of these addresses) – '000000000100' is freed up. It

then extracts the address from this location in the Availability List, uses it to access the triphone (n-1$^{th}$ triphone of the word that initially requested space allocation for its final 50 triphones) and sets the *set50* bit of that triphone to '0'. This indicates that this triphone (or word) no longer has space allocated for it. The DESELECT UNIT has been discussed in detail in Section 6.3.4.4. Phase 3 has been illustrated in Figure 6.16.

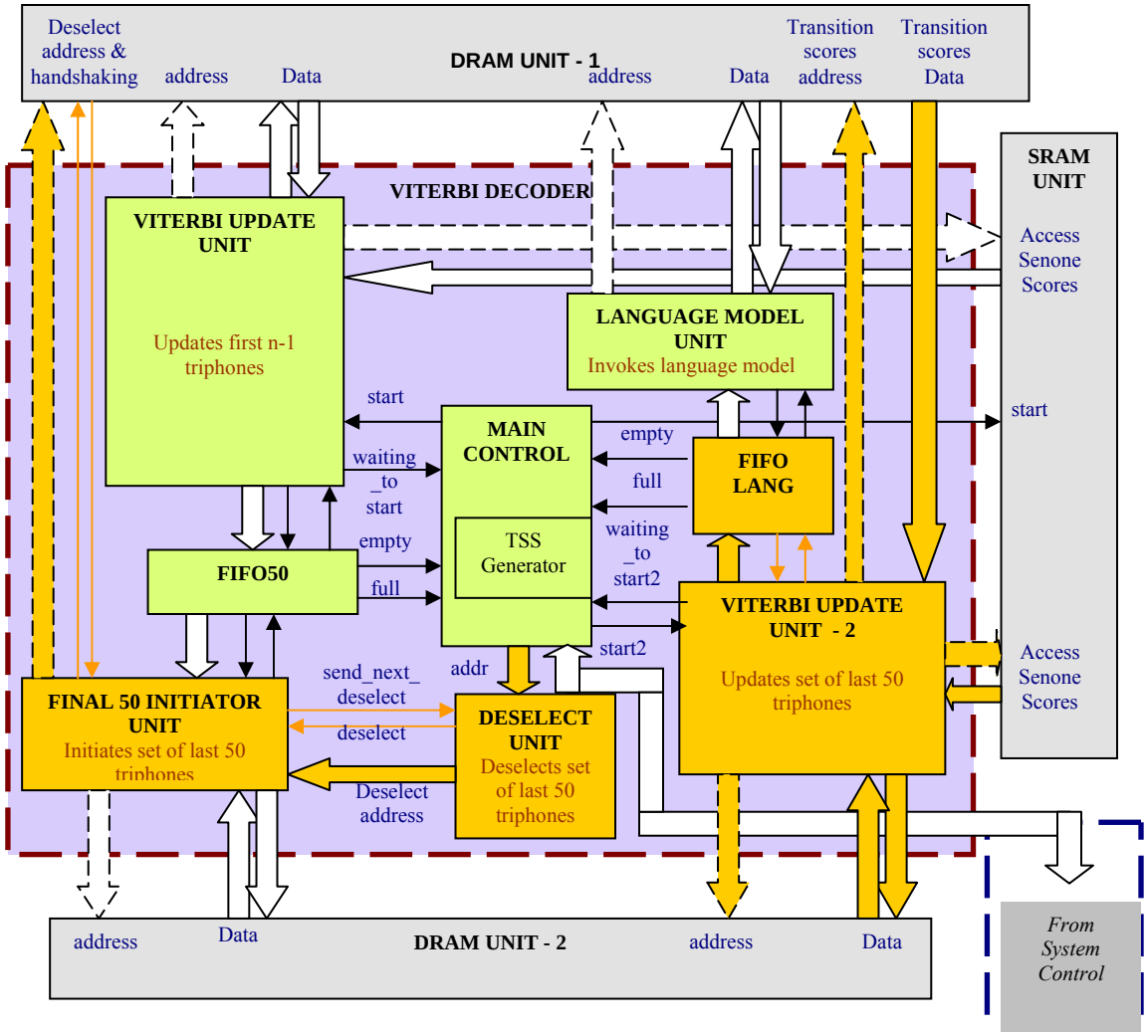



Figure 6.16 – Phase 3

The Final Phase – Phase 4 – performs the language model lookup to insert word-to-word transition probabilities to the completed word. Phase 4 has been illustrated in Figure 6.17. The information about every completed word (its TSS, Score and ID) are extracted by the

LANGUAGE MODEL UNIT. In the next step the 2 conditions under which the word could be part of a sentence (discussed in Section 6.2.2.2) – the timestamp check and the last triphone check are performed. Upon passing these, the word is inserted as part of a sentence into the temp_list (i.e. it has a history). If it does not pass the 2 checks, it is still inserted into the temp_list, but with no history. (i.e. it is first word of a new sentence). The language model is consulted (using this word and the previous words in the sentence) to obtain a language model probability for this word.

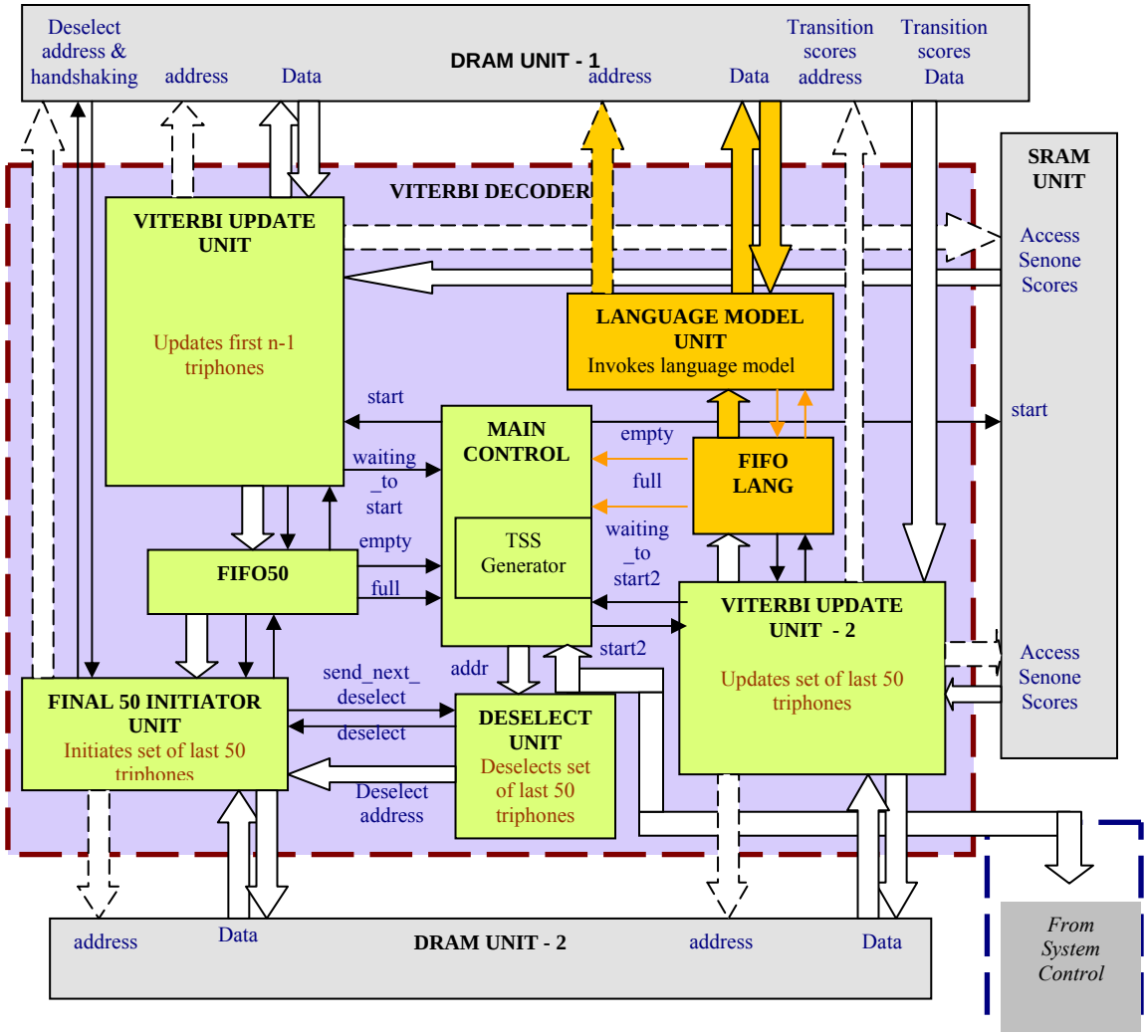



Figure 6.17 – Phase 4

At this point the final score of the word will be made up of a) the individual score of the word, b) the score of the sentence till now (the history score), and c) the language model probability for this word. This final score is written into the temp_list and forms the new score for the sentence till now. The promotion of a 'completed' word into an 'identified' word has been discussed in detail in Section 6.2.2.2. Phase 4 is performed completely by the LANGUAGE MODEL UNIT and is discussed in detail in Section 6.3.4.5.

As mentioned before, the MAIN CONTROL UNIT uses the *start1* signal to start off Phase 1 and Phase 2. It then uses the *waiting to start 1* signal from the VITERBI UPDATE UNIT –1 and the *empty* signal from FIFO50 to monitor when Phase 1 is done. Similarly it then uses the *waiting to start 2* signal from the VITERBI UPDATE UNIT –2 and the *empty* signal from FIFO-Lang to monitor when Phase 2 is done. It also uses some handshaking and assert signals to monitor and prevent memory overflow (during the initialization of the last 50 triphones).

### 6.3.2 Accessing the Memory

The VITERBI DECODER shown in Figure 6.12 interfaces with 2 DRAM units and 1 SRAM unit. The design and memory interfacing was chosen to minimize contesting memory accesses. This is especially important given the high memory bandwidth nature of this application. The SRAM unit stores the Senone Scores for each of the 6000 senones. All 6000 senones needs to get written to or updated every frame (10ms) by the Gaussian Estimator. Simultaneously, the Viterbi Decoder needs to access the Senone Scores for updating the states of the triphones. In order to facilitate this, 2 separate memories are maintained. While one of them gets updated by the Gaussian Estimator, the

other one is accessed by the Viterbi Decoder. The next frame, the process is switched as shown in Figure 6.18. It is not possible to predict the sequence in which the senone scores will be accessed from this unit. Maintaining two separate memories guarantees that the wrong values of senone scores are not read (we don't want the Viterbi Decoder to access values from a partially updated SRAM). While this means the memory requirement doubles, the extra 24Kbytes, is a small price to pay to reduce the operating speed requirements of both the Gaussian Estimator AND the Viterbi Decoder. This implies that the speech recognition process will always be 10ms behind actual speech. However it is still real-time speech recognition for all practical purposes.

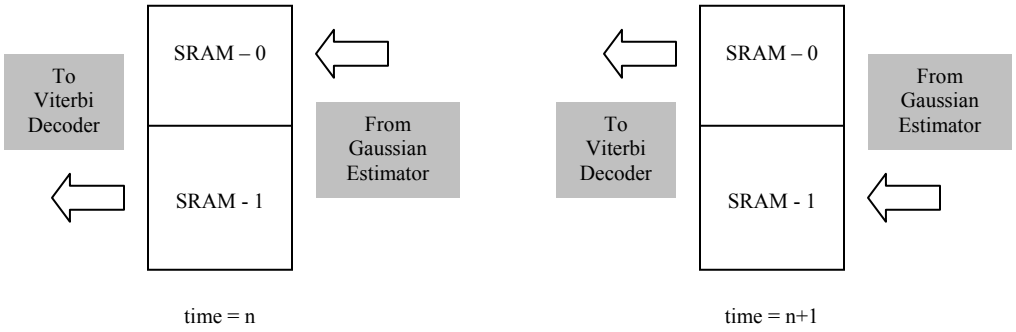



Figure 6.18 - Senone Score Updating

The DRAMs acts as both static storage for all the preloaded trained data as well as dynamic storage for all the triphone data. Phase 1 and Phase 2 take place in parallel. Phase 1 accesses DRAM UNIT –1 and Phase 2 accesses DRAM UNIT –2. Accessing different memories allow both phases to take place in parallel without each one having to wait on the other for memory access. Similarly Phase 3 and Phase 4 take place in parallel. Phase 3 mostly accesses DRAM UNIT-2 and accesses a small amount of data from DRAM UNIT –1. Phase 4 accesses DRAM UNIT –1. Even though a conflict may occur due to both Phase 3 and Phase 4 accessing DRAM UNIT –1, it does not slow down the process and because a) Phase 3 extracts only about 5% of its data from DRAM UNIT

–1 and b) Phase 4 is invoked only once a word is identified and is not as frequent as Phase 3.

### 6.3.3 DRAM Interface

Figure 6.19 shows the general interface for the DRAM UNIT(s). The DRAM controller [Appendix A] takes care of the DRAM initialization sequence, the precharge delay, auto refresh etc. while also generating the correct control and address signals. Several functional units request memory access and so the MEM-control unit takes care of

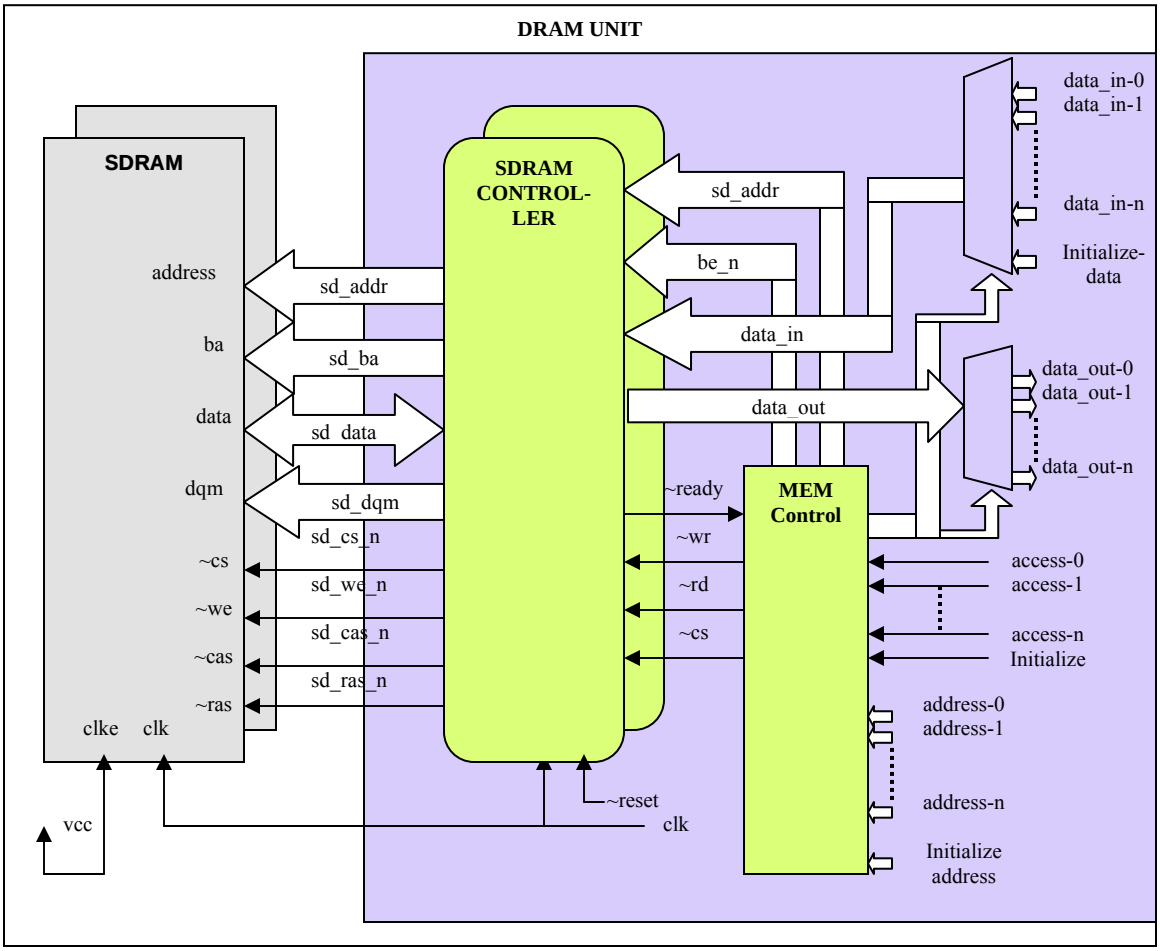



Figure 6.19 - DRAM Interfacing

chip/bank contention, scheduling and providing the right address to the DRAM controller. The MEM-control unit also provides the control signals to the DRAM controller as well as the control/select signals to the multiplexers/demultiplexors to correctly channel the data in and out of the DRAM controller. It is also responsible for initializing the SDRAM.

### 6.3.4 Functional Units

In this section we go over each of the functional units of the VITERBI DECODER in detail. We will be discussing the role of each unit as well as its interaction with other units. It should be noted global signals such as 'clock' and 'reset', as well as some handshaking signals have not been shown in the block diagrams of the functional units so as not to clutter the diagrams. The widths of the different signals along with the functionality are given in the tables corresponding to each block diagram. Please note that from this point, 'row' refers to the row of the Triphone_block(s) and not the DRAM row unless otherwise mentioned.

#### 6.3.4.1 Viterbi Update Unit-1

The VITERBI UPDATE UNIT-1 is shown in Figure 6.20. Table 6.5 gives details about the different signals within the design. Once the *start* bit is asserted (starting Phase 1), the 'validChecker' proceeds through the Triphone_Block checking for rows that are active (rows with *vm* bit equal to '1'). Once a row is found it hands off this address to 'DataRetrieval' using the *valid_address* and *valid* pins, and awaits a *done* signal that indicates that the current row has been updated. It then hands off the next active row to

'DataRetrieval' and the process continues. When it has reached the last address and receives a *done* signal, it signals the MAIN CONTROL UNIT that the Triphone_Block has been completely updated for this frame using the *waiting_to_start* signal.

On receiving a valid address, 'DataRetrieval' proceeds to access and store all the data that will be required to update the row. Scores, TID, SID(s), v1, v2, set50, prev, p_st and p_no are all addressed and accessed using the *valid_address*, and are stored in the register banks inside 'DataRetrieval'. SID(s) are redirected as addresses to the Senone_Score block (in SRAM UNIT) to obtain the senone scores for each of the four senones. TID is redirected as the address to the Transition_Block (in DRAM UNIT-1) to obtain the transition scores of the triphone. Once all the data has been retrieved, the 'DataRetrieval' signals that the data in its registers are the right values and is ready to be used by the rest of the system using the *done_DataRetreival* signal.

'NewScore' actually does the Viterbi decoding, taking in the scores, the transition scores and the senone scores and feeding them to smaller 'Update' units (similar to those shown in Figure 6.5). The 'Update' units calculate the new scores as well as the corresponding TSS for each of the four states. The new scores are put through a 'thresholdCheck' unit to see if the new scores of the four states pass pruning. This data is relayed to 'WriteBack'. The 'allValid' unit is a counter that gets activated when the *done_DataRetrieval* signal is asserted and counts a fixed number of cycles till it reaches a value at which the new scores, TSS values and the 'compare' information would have filtered through the pipeline and is valid. It then signals that 'WriteBack' may begin the final process of Phase1.

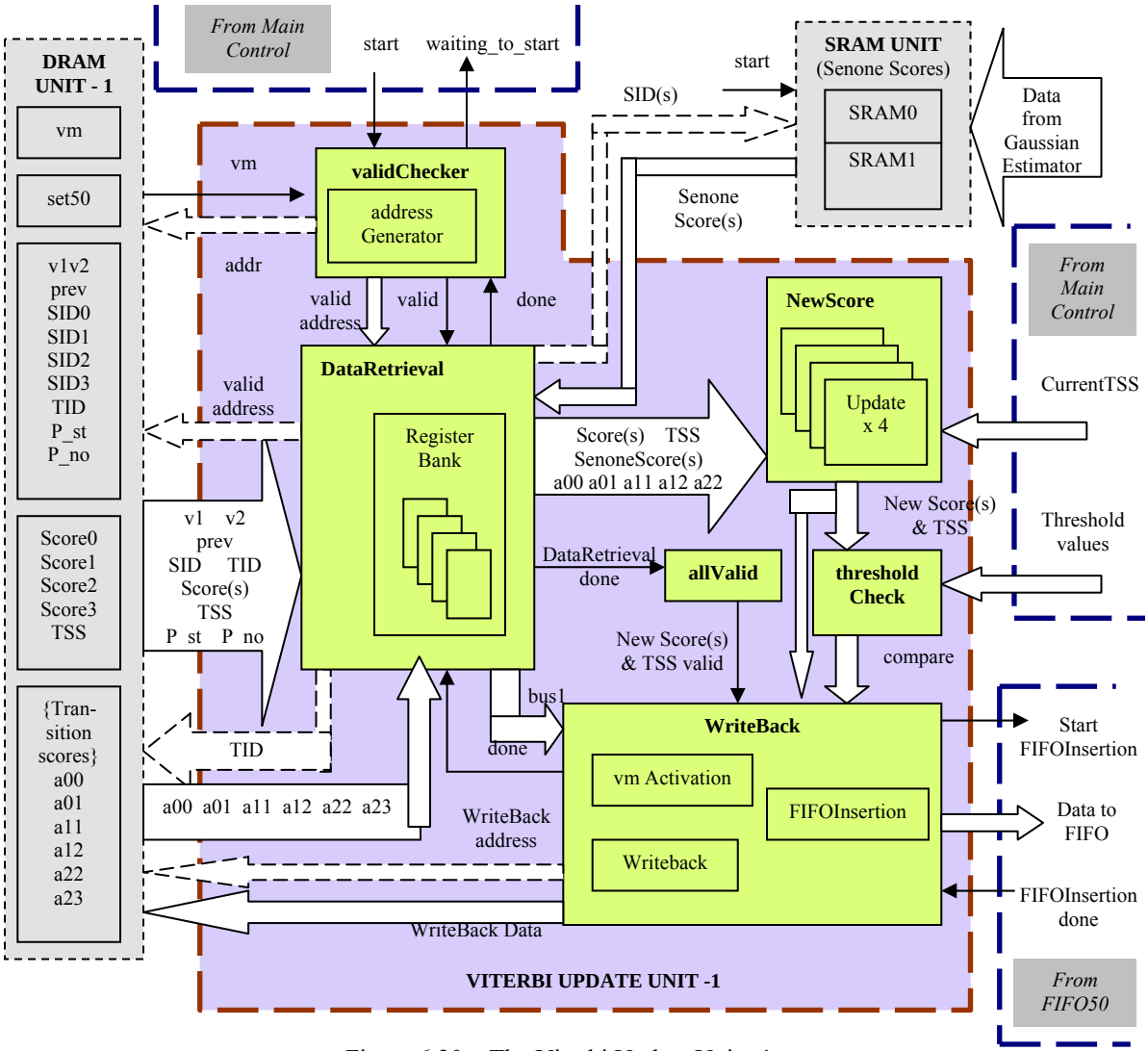



Figure 6.20 – The Viterbi Update Unit –1

Table 6.5 – Signals in the Viterbi Update Unit -1

| NAME OF SIGNAL | WIDTH | DESCRIPTION |
|---|---|---|
| start | 1 | Begins Phase1 for a new frame |
| waiting_to_start | 1 | Indicates that Phase1 has been completed |
| addr | 18 | Address to access vm bit of row |
| vm | 1 | vm bit |
| valid_address | 18 | Address of row that is valid |
| valid | 1 | Indicates the valid_address is valid |
| done | 1 | Indicates that row has been processed |
| v1,v2 | 1+1 | Indicates the type of triphone(of word having 'n' total triphones)<br>00 – 1st triphone of word<br>01 – 2nd to n-2th triphone of a word<br>10 – n-1th triphone of a word<br>11 – nth triphone of a word |
| prev | 18 | Address of predecessor triphone – Used to access the Score and TSS of the predecessor triphone. |
| SID(0,1,2,3) | 13x4 | ID(s) of the senones – Used as address to access Senone Scores from SRAM UNIT |
| TID | 16 | ID of the triphone – Used as address to access Transition Scores |

Table 6.5 (continued)

| | | |
|---|---|---|
| Score(s) | 32x(4+1) | Scores of each of the four states of the triphone as well as the score of the last state of the predecessor triphone |
| TSS | 11x4 | TimeStamps of each of the four states of the triphone as well as the TimeStamp of the last state of the predecessor triphone |
| P_st | 18 | First address of the set of triphones that succeed this triphone |
| P_no | 6 | No. of triphones that succeed this triphone |
| a00 a01 a11 a12 a22 a23 | 32x6 | Transition scores |
| Senone Scores | 32x4 | Senone Scores of each senone (each state) |
| DataRetrieval_done | 1 | Indicates that the row has been completely read and required Data has been retrieved |
| bus1 | | {v1,v2, P_st, P_no} |
| CurrentTSS | 11 | TSS for Current Frame |
| Threshold values | 32x4 | Values used for pruning |
| New Scores(s) and TSS | 32x4 + 11x4 | New values calculated for Score(s) and corresponding Timestamps |
| compare | 4x1 | Indicates which of the new Scores(s) passed pruning |
| StartFIFOInsertion | 1 | Signals FIFO50 to accept a new entry |
| Data to FIFO | 18+1 | {addr,set50} – Data to be inserted into FIFO50 |
| FIFOInsertionDone | 1 | Signals that the Data has been inserted into FIFO50 |
| WriteBack address | 18 | Address used to writeback new values |
| WriteBack Data | 1+1+32x4 +11x4 | Data that needs to be written back – {vm,set50,Score(s),TSS} |

As mentioned previously, the 'WriteBack' module is responsible for 3 different processes –'writeback', 'vm-activation' and 'FIFO-insertion'. Note that v1 and v2 give information about what type of triphone the current one is (See Table 6.5). During WriteBack, if all the new scores fail the threshold check (as indicated by the *compare* signal), the current triphone is deactivated (*vm* set to '0') unless the current triphone is the first triphone of a word (A word may *start* at any time frame and so the first triphone is always kept active to allow this). If at least one of the scores pass pruning, the triphone is kept active and the new scores and TSS are written back to the Triphone_Block using the *WriteBack address* and *WriteBack Data* buses. During 'vm-Activation', the 'Writeback' module gets the P_st value that provide the *start address* of the set of triphones that succeed the current triphone, and the P_no value that gives the *number* of triphones that succeed the current one. 'WriteBack' then proceeds to activate each of these triphones in case they had been deactivated, to ensure that a transition from current triphone to each of these ones triphones occur in the next time frame. During the 'FIFO-insertion' process,

'Writeback' sends a *startFIFOInsertion* signal along with the address of the current triphone and the set50 value to FIFO50. Next, it also sets the *set50* bit of this triphone to '1' and waits on a *FIFOInsertionDone* signal from FIFO50. This will indicate that the 'FIFO-Insertion' has been completed. The WriteBack then sends a *done* signal to 'DataRetrieval', which propagates it to the 'validChecker' signaling that the row (current triphone) has been processed and updated.

#### 6.3.4.2 Final 50 Initiator Unit

The FINAL 50 INITIATOR UNIT is comprised of 2 modules – the 'Initiator' and the 'MEM_allocator'. The Initiator is activated when FIFO50 signals that a new triphone is available for initialization/reactivation. FIFO50 does this by placing the required data (which is the *address* of the triphone and its *set50* bit) on the data bus and asserting the *Advance done* signal . At this point one of two things take place depending on the value of the *set50* bit.

If the *set50* bit had been '0', this means that the triphone is requesting space allocation. The 'Initiator' looks down the Availability List for a space whose *used* bit is not set to '1'. Once such a space is found, it places the address of the triphone into this location and sets the *used* bit to '1'. The address of this location – know as the *List address* – along with the the *address* of the triphone and the *set50* bit are relayed to the 'MEM_Allocator'.

If the *set50* bit had been '1', this means that the triphone has already been allocated space and is requesting a reactivation. The 'Initiator' simply looks down the list

for the location at which the address of the triphone is stored. Once found, it sends the *address*, the *List address* and the *set50* bit to the MEM_Allocator.

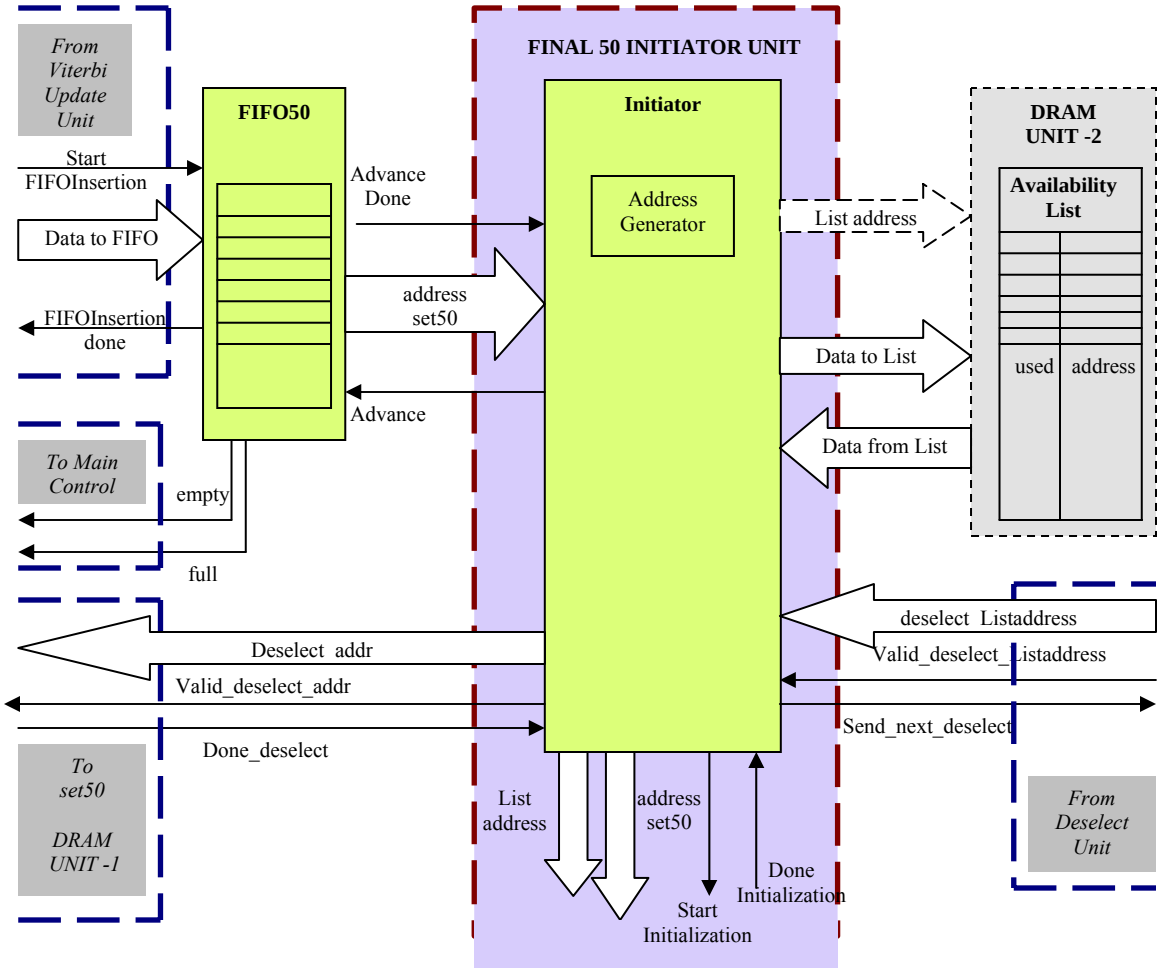



Figure 6.21 - Final 50 Initiator Unit - Initiator

Table 6.6 - Signals in the Initiator

| NAME OF SIGNAL | WIDTH | DESCRIPTION |
|---|---|---|
| AdvanceDone | 1 | Indicates that FIFO50 has placed a valid data on the data lines – {address & set50} |
| address | 18 | Address of the triphone that requested the initialization of the last 50 triphones |
| set50 | 1 | set50 bit – used to indicate if space has been previously allocated or not |
| advance | 1 | Indicates that the initialization process has completed and the FIFO50 may send the next value(s) |
| List address (to Availability List) | 12 | Address used to access the Availability List in DRAM UNIT-2 |
| List address (to MEM_allocator) | 12 | Final List Address that will be used by the MEM_allocator to do the initialization/reactivation. |
| Data to List | 18+1 | Data sent to(written to) the Availability List |
| Data from List | 18+1 | Data retrieved from the Availability List |

Table 6.6 (continued)

| | | |
|---|---|---|
| deselect_Listaddress | 12 | During deselect process, this gives the location in Availability List that needs to be 'freed up' |
| Valid_deselect_Listaddress | 1 | Used to start the deselection process |
| Send_next_deselect | 1 | Used to indicate that the deselection process is complete |
| Deselect_addr | 18 | Address of triphone/word that has been deallocated space in the Triphone_Block Type B – used to access 'set50' bit and set it to '0' |
| Valid_deselect_addr | 1 | Used to indicate that the 'set50' bit needs to be set to '0' |
| Done_deselect | 1 | Used to indicate that the 'set50' bit has been set to '0' |
| Start_Initialization | 1 | Used to signal the MEM_allocator to start the initialization/reactivation process |
| Done_Initialization | 1 | Used to indicate that the MEM_allocator has completed the initialization/reactivation process |

When the DESELECT UNIT identifies a set of addresses that are inactive it sends the corresponding *List address* to the 'Initiator' using the *deselect_Listaddress* bus and signals 'Initiator' to start the deselection process by asserting the *valid_deselect_Listaddress* line. The 'Initiator' uses this address to access the Availability List. It sets the *used* bit to '0', sends the *address* at that location to the *Deselect_addr* bus and asserts the *valid_deselect_addr* line signaling that the *set50* bit of the triphone at this address needs to be set to '0'. The deselection process is completed when the *Done_deselect* line is asserted, at which point the 'Initiator' asserts the *Send_next_*deselect line to signals DESELECT UNIT that it is ready to deselect another set of locations if required.

'MEM_Allocator is responsible for initializing the Triphone_Block_Type_B with the right values (if the *set50* bit was '0') or reactivating a set of locations (if the *set50* bit was '1'). Either process is started when the *Start_Initialization* signal is asserted. During initialization, the WordLookup table is accessed using the *address*. It provides the ID of the word that requested the initialization. The address of the location at which that the final set of 50 triphones will be stored, is obtained by appending the *ListAddress* with a 6 bit extension as shown in Figure 6.22 and Figure 6.14. Appending the' wordID' to a 6-bit

extension will provide the address to the TID and SID of the triphones. This information is stored in the SIDTIDLoc block and is accessed during Phase 3. It acts as the 22 bit address to the TID's and SID's of the last 60000x50 triphones.

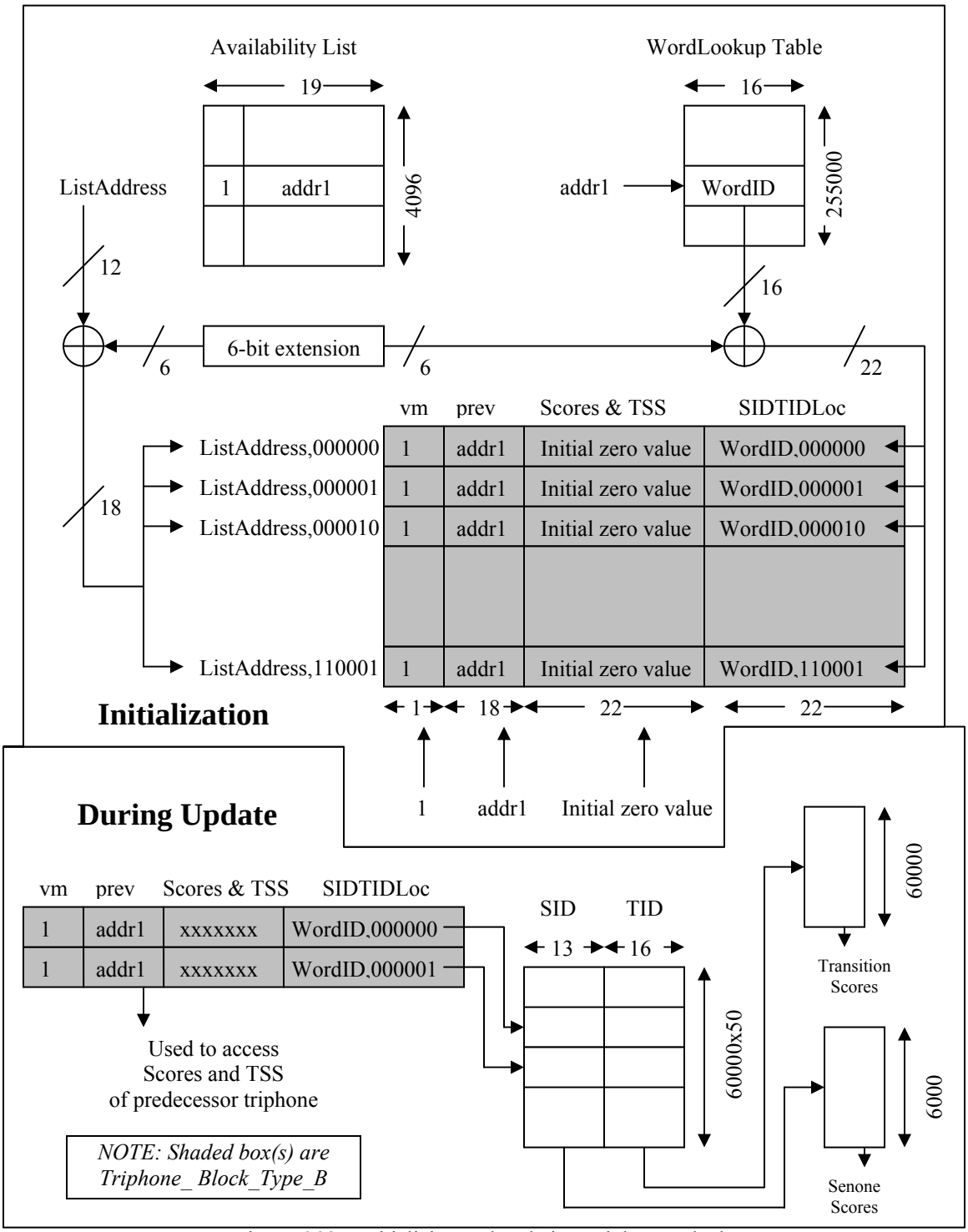



Figure 6.22 – Initializing and updating Triphone_Block_Type_B

The *vm* bit is set to '1', the *prev* is initialized with *address* and the *Scores* and *TSS* bits are set to the initial zero value. The initialization process is shown in Figure 6.22.

On the other hand, during reactivation, the 'MEM_Allocator' uses the *List_address* to access the *vm* bits of the 50 triphones and set them to '1'. At the end of either process, the 'MEM_Allocator' asserts *Done_Initialization*, to signal the end of the initialization/reactivation process. The 'MEM_Allocator' is shown in Figure 6.23.

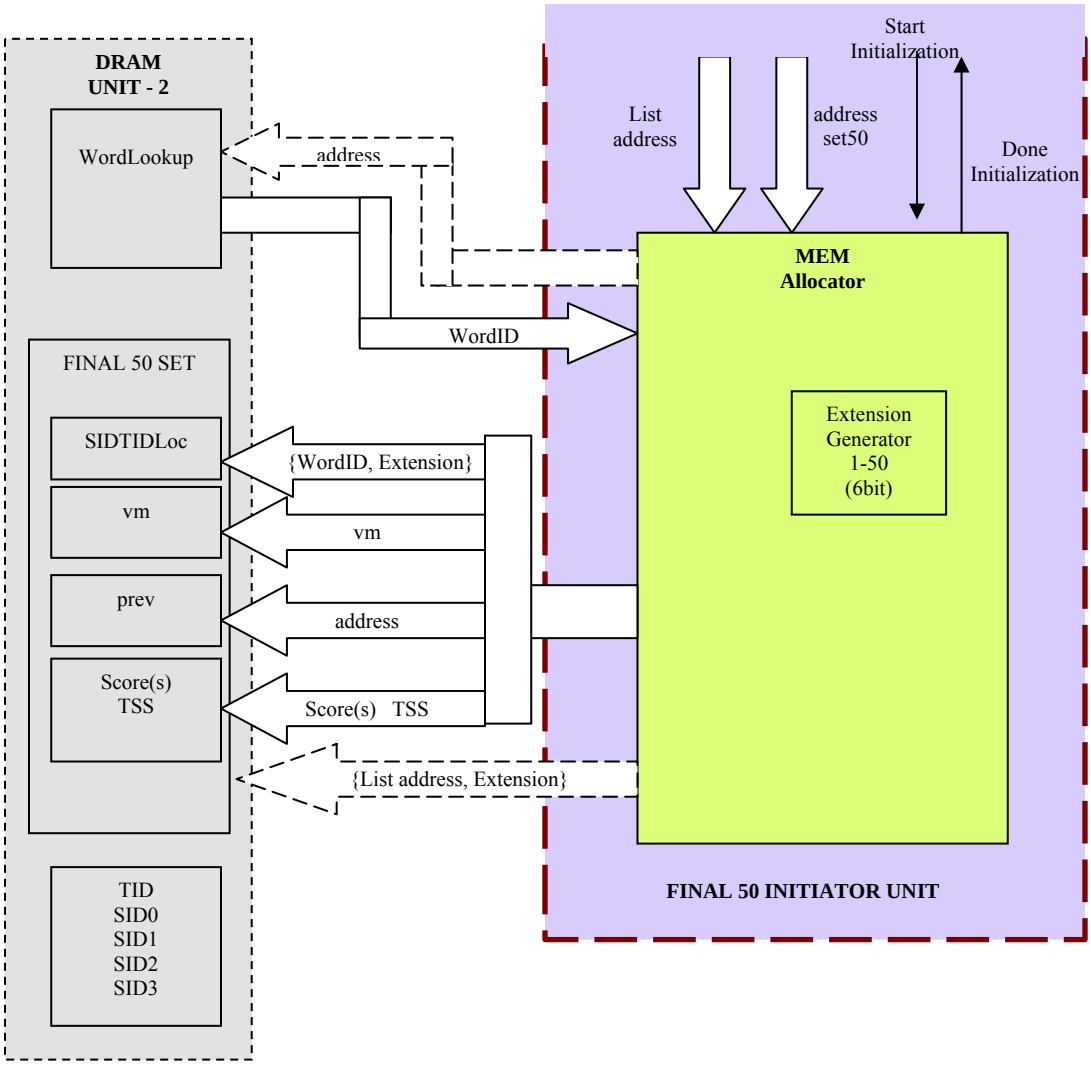



Figure 6.23 - Final 50 Initiator Unit – MEM_Allocator

Table 6.7 - Signals in the MEM_Allocator

| NAME OF SIGNAL | WIDTH | DESCRIPTION |
|---|---|---|
| Start_Initialization | 1 | Used to signal the MEM_Allocator to start the initialization/reactivation process |
| Done_Initialization | 1 | Used to indicate that the MEM_Allocator has completed the initialization/reactivation process |
| address | 18 | Address of the triphone that requested the initialization of the last 50 triphones – this is also used to access the WordLookup Block and get the word ID |
| set50 | 1 | set50 bit – used to indicate if space has been previously allocated or not |
| advance | 1 | Indicates that the initialization process has completed and the FIFO50 may send the next value(s) |
| List address (from MEM_Allocator) | 12 | Final List Address that will be used by the MEM_Allocator to do the initialization/reactivation. |
| WordID | 16 | ID of the word |

### 6.3.4.3 Viterbi Update Unit-2

The VITERBI UPDATE UNIT –2 is quite similar to the VITERBI UPDATE UNIT-1 and we refrain from repeating all the details. The main difference between the 2 units is that instead of updating the first n-1 triphones, VITERBI UPDATE UNIT -2 updates the last set of 50 triphones in Phase 3. The VITERBI UPDATE UNIT –2 accesses the DRAM UNIT –1 for the transition scores and DRAM UNIT –2 for everything else. During the 'FIFO-Insertion' The 'Writeback' module inserts the completed word into the FIFO-Lang i.e. it places the TID, the WordID, the Score of the last state, and the TSS of the last state on *Data_to_FIFO* and asserts the *start_FIFOInsertion*. The VITERBI UPDATE UNIT –2 signals the end of Phase 3 by asserting the *waiting_to_start2* line. Figure 6.24 illustrates the VITERBI UPDATE UNIT –2.

### 6.3.4.4 Deselect Unit

The DESELECT UNIT (shown in Figure 6.25) serves to identify sets of locations in the Triphone_Block_Type_B that are inactive and can be freed up. Remember that the dynamic allocation scheme requires begin successfully able to identify such locations so that the design does not run out of memory.

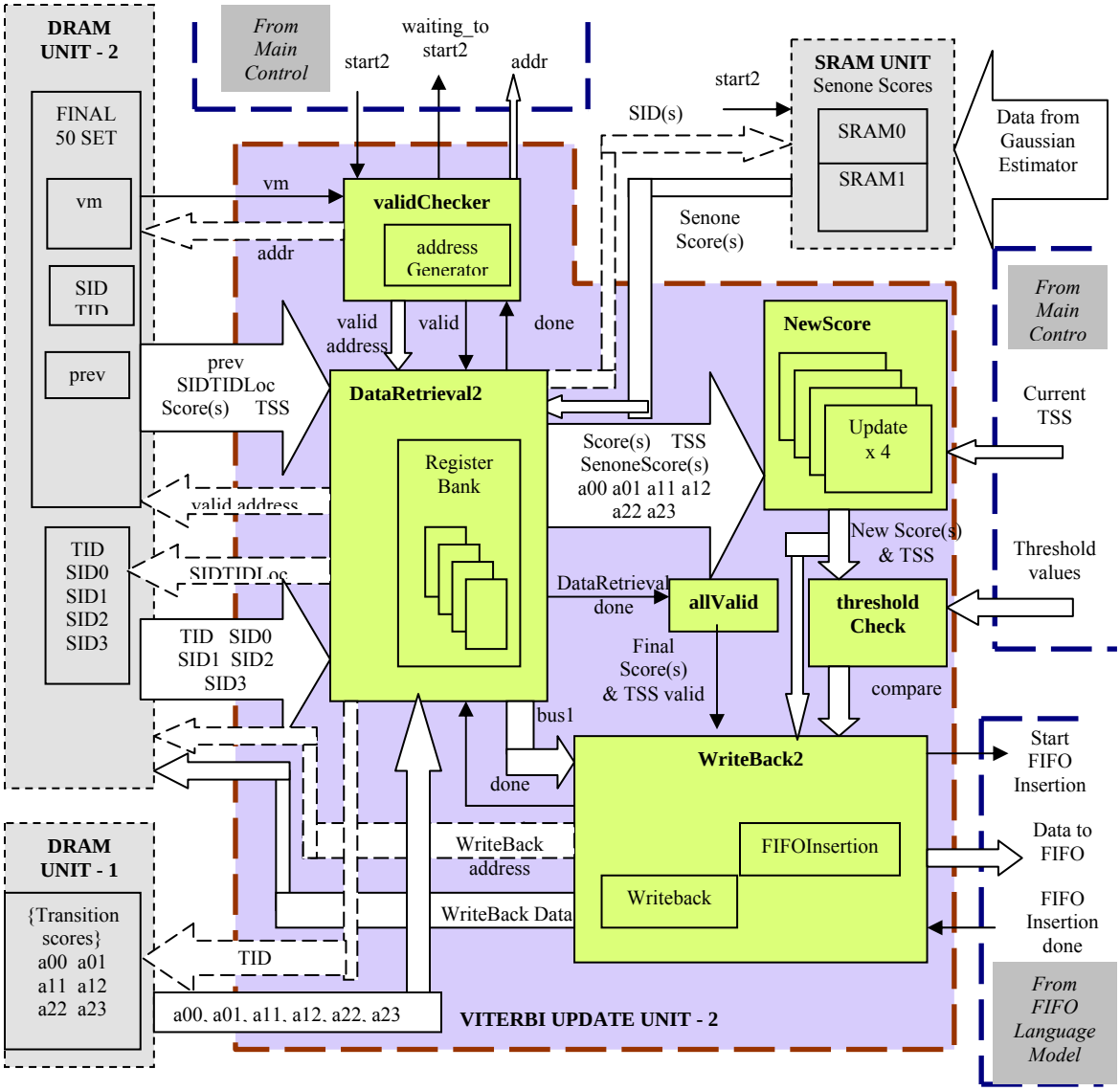



Figure 6.24 – The Viterbi Update Unit –2

Table 6.8 – Signals in the Viterbi Update Unit -2

| NAME OF SIGNAL | WIDTH | DESCRIPTION |
|---|---|---|
| start | 1 | Begins Phase3 for a new frame |
| waiting_to_start | 1 | Indicates that Phase3 has been completed |
| addr | 18 | Address to access vm bit of row |
| vm | 1 | vm bit |
| valid_address | 18 | Address of row that is valid |
| valid | 1 | Indicates the valid_address is valid |
| done | 1 | Indicates that row has been processed |
| prev | 18 | Address of predecessor triphone – Used to access the Score and TSS of the predecessor triphone. |
| SID(0,1,2,3) | 13x4 | ID(s) of the senones – Used as address to access Senone Scores from SRAM UNIT |
| TID | 16 | ID of the triphone – Used as address to access Transition Scores |
| Score(s) | 32x(4+1) | Scores of each of the four states of the triphone as well as the score of the last state of the predecessor triphone |

Table 6.8 (continued)

| | | |
|---|---|---|
| TSS | 11x4 | TimeStamps of each of the four states of the triphone as well as the TimeStamp of the last state of the predecessor triphone |
| a00 a01 a11 a12 a22 a23 | 32x6 | Transition scores |
| Senone Scores | 32x4 | Senone Scores of each senone (each state) |
| DataRetrieval_done | 1 | Indicates that the row has been completely read and required Data has been retrieved |
| bus1 | | {v1,v2, WordID,TID} |
| CurrentTSS | 11 | TSS for Current Frame |
| Threshold values | 32x4 | Values used for pruning |
| New Scores(s) and TSS | 32x4 + 11x4 | New values calculated for Score(s) and corresponding Timestamps |
| compare | 4x1 | Indicates which of the new Scores(s) passed pruning |
| StartFIFOInsertion | 1 | Signals FIFO50 to accept a new entry |
| Data to FIFO | 16+16+32+11 | {{WordID, TID,Score,TSS}} – Data to be inserted into FIFO-Lang |
| FIFOInsertionDone | 1 | Signals that the Data has been inserted into FIFO50 |
| WriteBack address | 18 | Address used to writeback new values |
| WriteBack Data | 1+32x4 +11x4 | Data that needs to be written back – {vm,Score(s),TSS} |

Once the VITERBI UPDATE UNIT –2 is done with updating a set of locations, it sends the first 12 bits of this address to DESELECT UNIT. The DESELECT UNIT then checks the 50 corresponding locations. If all the active bits (*vm*s) are '0' then this set of locations is marked for de-allocation. The 12-bit *List address* is sent to the 'Initiator' module (within the FINAL 50 INITIATOR UNIT) using the *deselect_Listaddress* bus and signals 'Initiator' to start the deselection process by asserting the *valid_deselect_Listaddress* line.

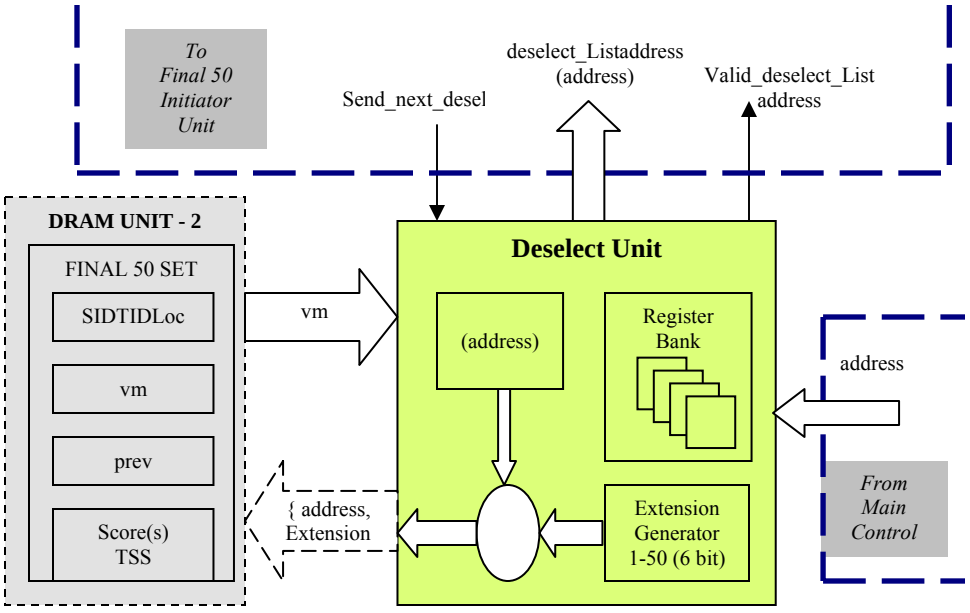



Figure 6.25 – Deselect Unit

#### 6.3.4.5 Language Model Unit

The 'Identified_words' block consist of words that have completed AND have a successor in the 'temp_list' block(meaning that they are part of a sentence). The 'temp_list' block consist of words that may have a predecessor in the 'Identified_words' block OR may be the first word of a sentence. Both of these are placed in DRAM UNIT –1.

The first step in the process now is to check if the TSS of the completed word matches the TSF of any of the words in the 'temp_list'. If there are one (or more) matches, this would indicate that the time this word started corresponds to the time the word in the 'temp_list' completed - meaning that the word pair could be part of a sentence. The ID of the completed word (or more specifically, the ID of the last triphone of the word) is used to lookup possible next triphones for this word from the 'Possible_next_Triphone' list. If any entry in this list corresponding to this ID (there can be max of 50 for any ID) matches the last triphone of the word in the 'temp_list' (which is a separate column entry in the 'temp_list'), this means that a transition could have taken place. This condition checking is done by the 'Conditon_Check Unit'.

Scenario A - A match is found (a word pair is found that passes both compare steps). The temp_list word is checked to see if the it had any predecessor in the Identified_words list. At this point the word in the Identified_words list is called the n-2$^{nd}$ word (A null is used if no predecessor exists). The word in the temp_list is called the n-1$^{st}$ word. The completed word is called the n$^{th}$ word. The 3 words are then sent to the 'Trigram Search' to check for the language model

probabaility. The final scores and entries are written back into the ‘temp_list’ and ‘Identified_word’ block.

Senario B – No match is found. The word is inserted as a the *first word* of a new sentence. Once again the ‘Trigram Search’ is used to check the probability of this word being the first of a sentence, and the final scores and entries are written back into the ‘temp_list’ and ‘Identified_word’ block.

Figure 6.26 shows the LANGUAGE MODEL UNIT.

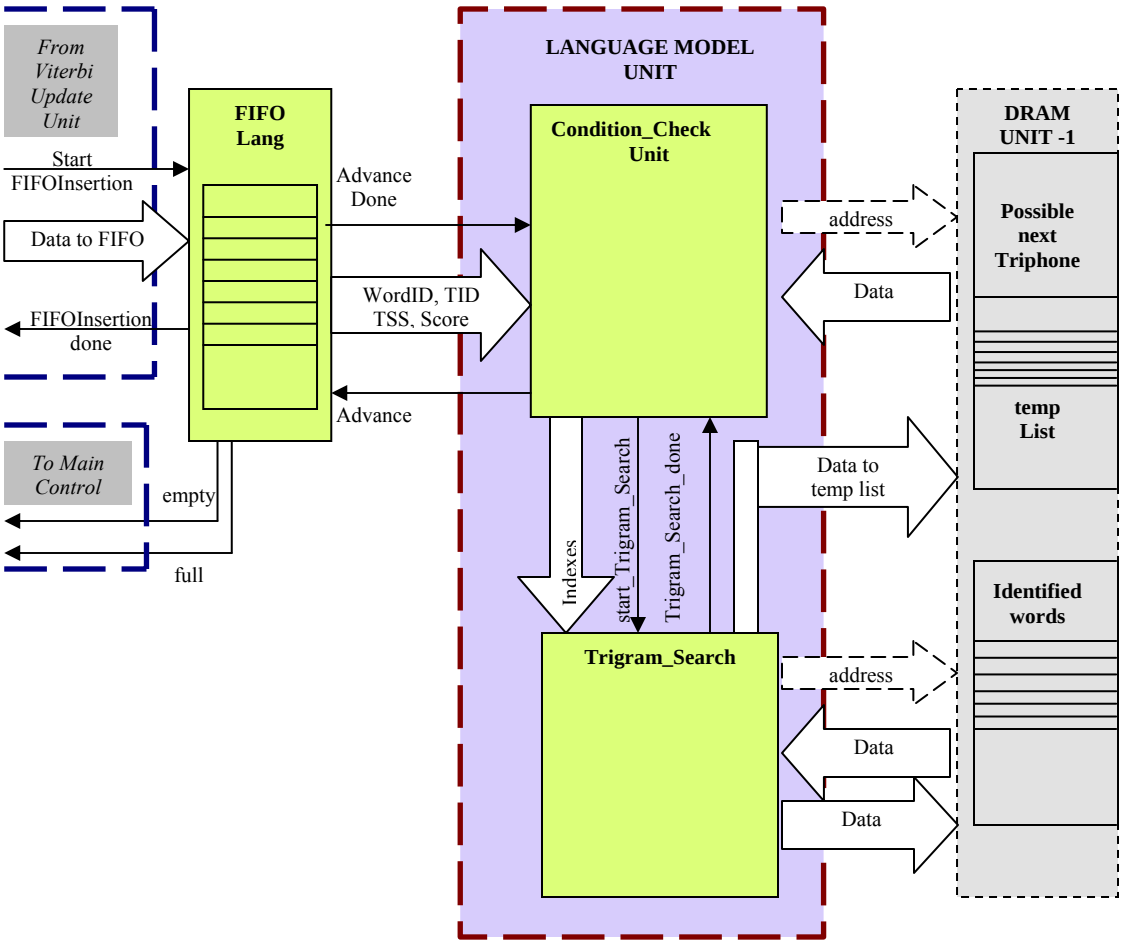



Figure 6.26 – Language Model Unit

# CHAPTER 7

# Evaluation

In our evaluation of the systems developed, we will be considering two sets of results. Firstly we will be showing improvements and savings achieved by each of our designs compared to the baseline designs (the conventional design upon which we made improvements). Secondly, we will be comparing each of our designs to other implementations of the speech recognition system.

Comparison of speech recognition systems, in part or as a whole, is often difficult owing to the vast majority of parameters that specify any such system. As an example, more often than not each system differs by the size of the vocabulary it targets. Real-time specifications for such systems differ, as it is hardly fair to compare a system that supports a 5k vocabulary set with one that is designed for a 60k set. The nuances of each design and design methodology may not allow for assumptions of complete parallel operation and linear scalability. For example the design developed by researchers at the Norwegian University of Science and Technology [69] is an earlier design of a gaussian estimation unit that reports real-time operation. However the results were based on an earlier test set, and the workload has worsened by 15.3 times since then. The design itself is based on a fixed-point format, while current designs are based on the IEEE 754 floating-point format. Another example is the design by Nedevschi et al. [70], which is a

complete speech recognition system doing both the gaussian estimation and the viterbi search in a single design. It shows 1-2 orders of magnitude improvement in power consumption as compared to other designs. However this design handles only 28 phonemes and 40 words, and cannot be considered a 'complete' speech recognition system by today's standards. Even the researchers at CMU (Speech in Silicon project) [79] have compared their latest design, with implementations of the SPHINX software on different CPU configurations, perhaps for the same reasons.

Binu [4] evaluated his perception processor to other implementations of speech recognition systems, and it is these results that we will be comparing our system with. This particular study was chosen because of the variety of systems in different domains that this study was conducted on. The first of these systems is a software implementation of SPHINX running on a 2.4GHz Intel Pentium 4 processor – a system that is optimized for performance rather than energy efficiency. The second system is software running on a 400MHz Intel XScale (StrongARM) processor, which represents an energy efficient embedded processor. The third system is the perception processor developed by Binu. Finally, for the gaussian estimation task, we also compare our Gaussian Estimator with another custom gaussian accelerator [4]. Two benchmarks are run on each of these systems and compared for power and energy efficiency. The first benchmark developed for the gaussian estimation task compares the performance of these systems on a single input packet - where one input packet corresponds to evaluating a single acoustic model state over 10 frames of a speech signal. The second benchmark developed for the viterbi decoding task compares the performance of these systems on a single input packet – where one input packet to the consists of updating 32 triphones. While the first

benchmark is floating-point operation dominated, the second benchmark is dominated by integer compares and select operations.

Even with these implementations, we need to normalize across process, to make an accurate comparison. The perception processor and the Pentium 4 are both implemented in 0.13µ CMOS technology and their results need not be normalized. The XScale is implemented using a .18µ technology and Binu's [4] custom Gaussian Accelerator is implemented in a .25µ technology. Our Gaussian Estimator and Viterbi Decoder designs have been implemented in a .25µ and .18µ technology respectively. All these results have been normalized to the to a .13µ technology. We used constant field scaling i.e. when the minimum feature size is scaled from $\lambda$ to $s\lambda$, where s is a scale factor, the length and width of the channel, the oxide thickness, substrate concentration density and the operating voltage are all scaled by the same factor s so that the electric field in the transistor remains constant. The net result is that the dynamic power consumption P is scaled to $s^2P$ and energy consumption scales as $s^3$.

The metrics chosen for these comparison was power and energy efficiency. Note that each design was required to meet real-time recognition, and the *total* energy and power consumption of an appropriate number of units that would ensure this was used for the comparisons. Power was chosen as a metric to evaluate the portability of our system where the power supply limits the maximum power that such designs can draw. Each design deals with the speech recognition process differently – some requiring a large number of cycles to complete processing a given set of inputs, but operating at higher frequencies (such as the Pentium-4), while others operating much slower, but requiring fewer cycles to completely process the set of inputs. Energy efficiency, measured as

energy per input (what each 'input' is defined as, is discussed later) was chosen as a metric, both for its ability to normalize across such operating parameters, as well as to provide an 'absolute' measure of battery life.

## 7.1 Gaussian Estimator

We have designed and simulated the HMM-based GE using Verilog HDL and Synopsys. The design was synthesized using the Synopsys Design Analyzer tool in a .25µ CMOS technology. Feature vectors are extracted from the input speech waveforms using the Sphinx-3 front-end. An operational frequency of 50Mhz was achieved. Three of these units were sufficient to support real-time speech recognition for about 6000 senones. The die size was about 2.856 $mm^2$.

The design was evaluated for real time performance and power consumption for six configurations – Baseline system, crude threshold check (CTC), Simple Down Sampling (SDS), Conditional Down Sampling (CDS), Context Independent Gaussian Mixture Model Selection (CIGMMS) and Sub-Vector Quantization Gaussian Selection (SVQGS). Figure 7.1 shows the real time performance of each of these techniques on our design. Real-time performance is measured as a fraction of the sampling frame rate (every 10ms). Completion times are measured by dividing the number of cycles for each test set in the simulation by the operating frequency. Real time performance is achieved if all required senone scores are computed before the next set of input observation vectors is available (10ms later). Performance is thus reported as a fraction of this time of 10ms required to complete the computations. It should be noted that for the SDS, the frequency of operation was cut down to 25Mhz so as to reduce power consumption, which explains

its 1xRT performance. All results are shown for both the arbiter and the Gaussian Estimator units working together. A 128Mb (x32) Micron SDRAM - MT48LC4M32B2 – was chosen to model the DRAM Unit used for the tests, and a corresponding DRAM controller was also developed (Appendix A). A generic SRAM unit was used for modeling the SRAM for the senone scores. An additional 40% was added to both the area and power numbers to compensate for post placement and routing.

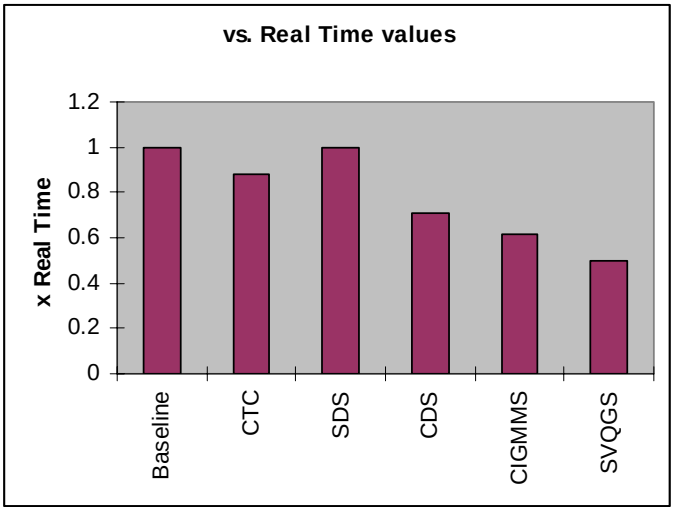



Figure 7.1 - Real Time Performance (speed)

Figure 7.2 shows the power consumption for each of these techniques with our design. Power consumption was computed by annotating the activity of individual modules with each adaptation to the power consumption of individual modules (as reported by the 'report_power' command in Synopsis Design Analyzer. The power consumption values reported are for a single GE. It should be noted that since three units are needed, the total power consumption would be three times the values reported. We achieve a power consumption reduction of up to 48% (as compared to the baseline system) using these techniques.

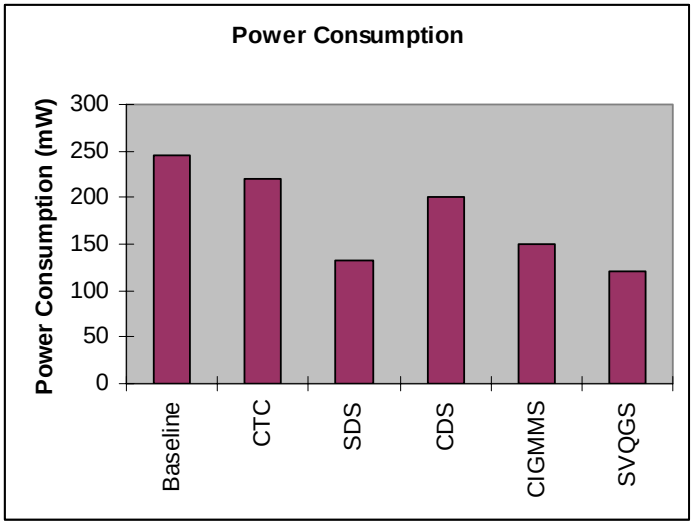



Figure 7.2 - Power Consumption

Next we evaluate our system with the benchmarks and systems described at the beginning of this chapter to put our design in perspective.

The baseline system was chosen for these evaluations to signify that the results we obtain are worst case, and will only improve depending on context and input. The results shown are for 3 of our units (required for real-time operation). Figure 7.3 shows the process normalized steady state power consumption of the different implementations – XScale, Pentium 4 processor, the Perception Processor, Binu's custom Gaussian Accelerator, and our Gaussian Estimator.While it was expected that our custom implementation of the Gaussian Estimator would achieve about 2 orders of magnitude power savings over the general purpose Pentium 4 processor, it is worthwhile to note that our design achieves a 43% reduction in power consumption over the previous custom ASIC design. Figure 7.4 shows the energy consumption per input packet. Once again while achieving significant improvement (3 orders of magnitude) over the Pentium 4 implementation, we also achieve a 35% improvement in total energy consumption as compared to the previous custom ASIC design.

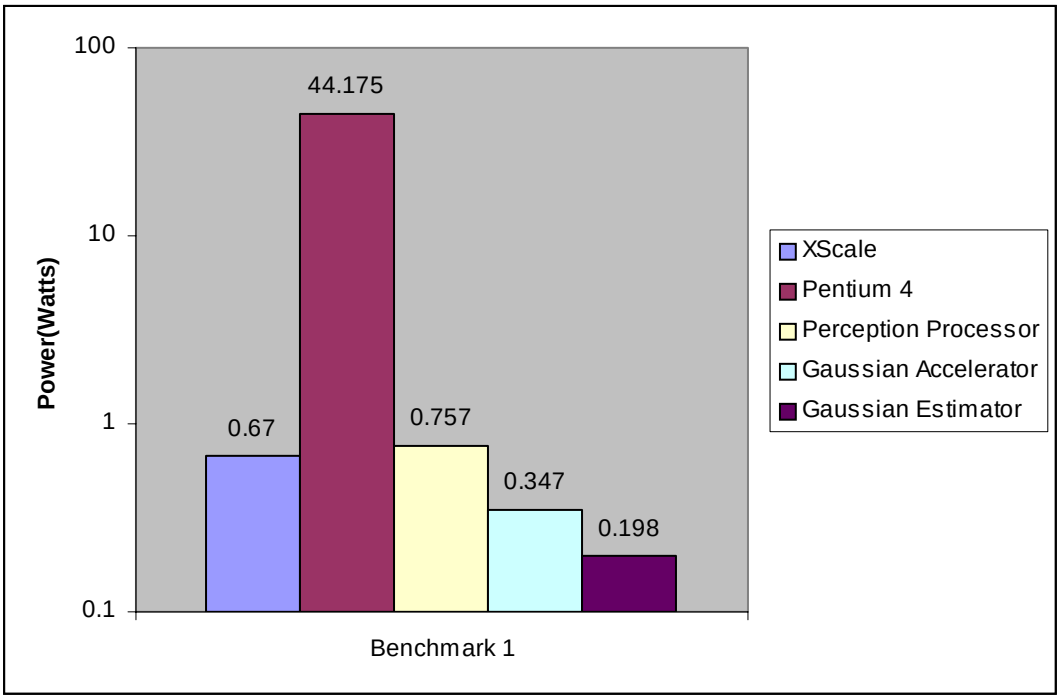



Figure 7.3 - Power Consumption across systems – Benchmark 1

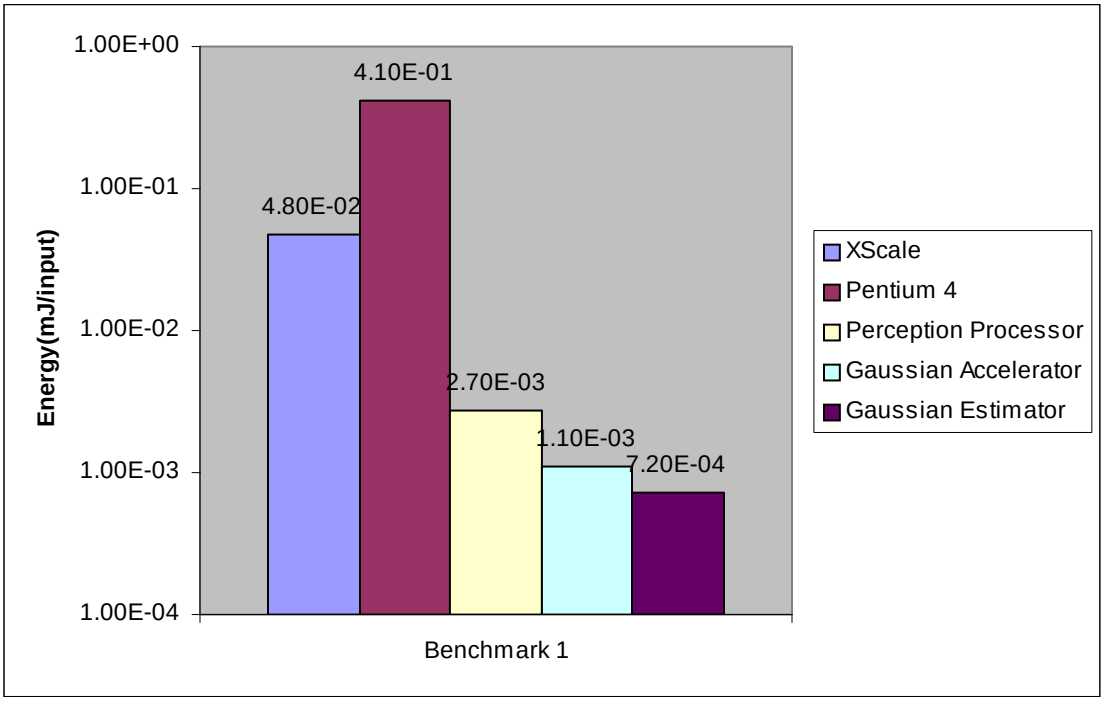



Figure 7.4 – Process Normalized Energy Consumption across systems – Benchmark 1

## 7.2 Viterbi Decoder

We have designed and simulated the Viterbi Decoder Unit using Verilog HDL and Synopsys. The design was synthesized using the Synopsys Design Analyzer tool in a .18μ CMOS technology. Eight of these units operating at 200MHz achieve real-time operation. The die size was about 2.278 $mm^2$ per unit. The complete system of 8 units consumes about 80.5 mW of power. A test bench of 1000 words modeled after the 1997 Hub-5E dataset was used for the evaluation of the design. A 128Mb (x32) Micron SDRAM - MT48LC4M32B2 – was chosen to model the DRAM Units used for the tests, and a corresponding DRAM controller was also developed. A generic SRAM was used for the senone score SRAM unit(s).

The tests did not include the Gaussian Estimator running in parallel. Instead, senone scores for 50 frames were developed and used for the simulations. While the complete design could be ported to the Xilinx Vertex-II FPGA, it could not be tested due to the limited memory resources available on the Xilinx Virtex-II Pro™ Prototype Platform board[96]. However, the design was implemented part by part on the FPGA and tested on a smaller dataset to verify functionality. Table 7.1 shows the memory breakup of the initial and final implementations and this is further illustrated in Figure 7.5 and Figure 7.6.

Table 7.1 – Memory Requirement Breakup comparison

| FINAL IMPLEMENTATION MEMORY BREAKUP | (MBYTES) | | INITIAL IMPLEMENTATION MEMORY BREAKUP | (MBYTES) |
|---|---|---|---|---|
| Triphone Block | 15.38 | | Triphone Block | 114.405 |
| Transition Block | 1.44 | | Transition Block | 1.44 |
| Senone Scores | 0.048 | | Senone Scores | 0.048 |
| Availability List | 0.009728 | | Identified_words | 0.802 |
| WordLookup Table | 0.51 | | Last_Phone_Score | 13.1 |
| SID TID | 10.875 | | Word_Lookup | 0.96 |
| temp_list | 0.055 | | Language Model | 147 |
| Identified_words | 0.062464 | | | |
| Possible_next_triphones | 6 | | | |
| Language Model | 147 | | | |
| Total | 181.38019 | | Total | 277.755 |

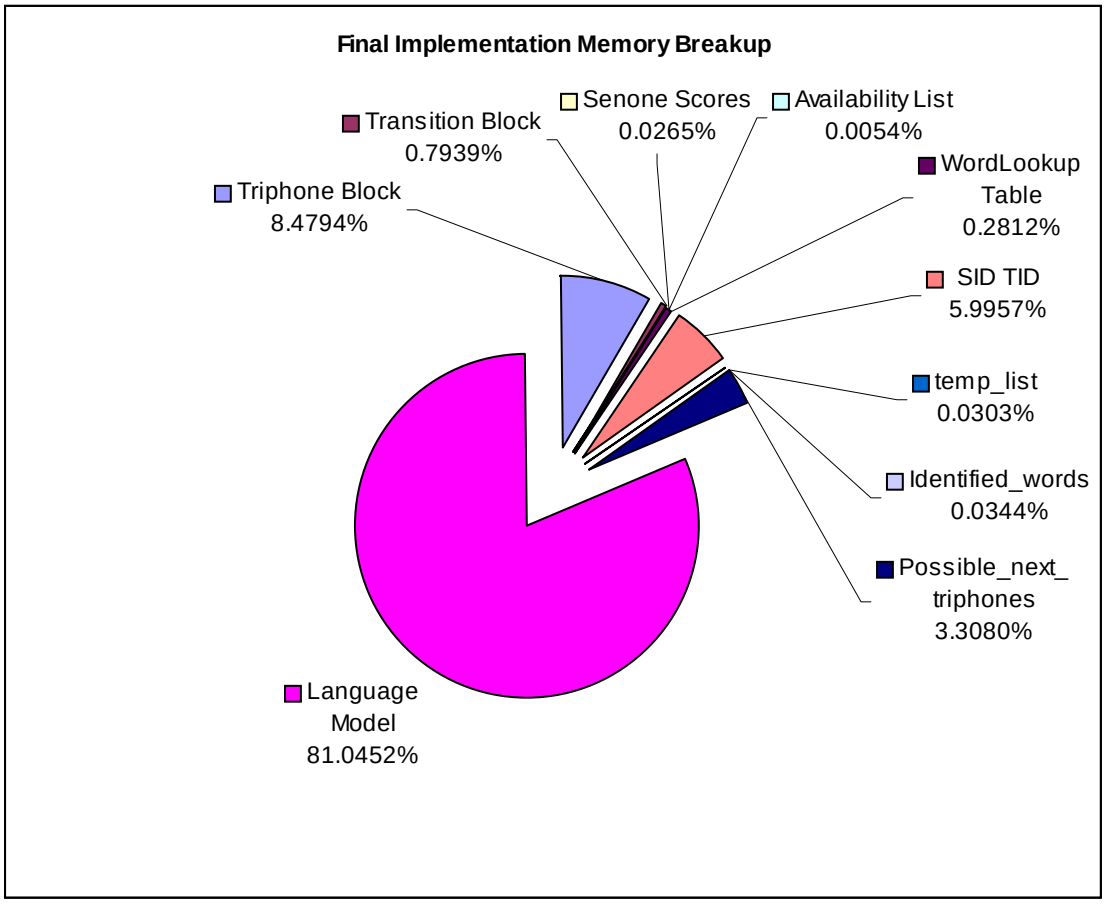



Figure 7.5 – Final Implementation Memory Breakup

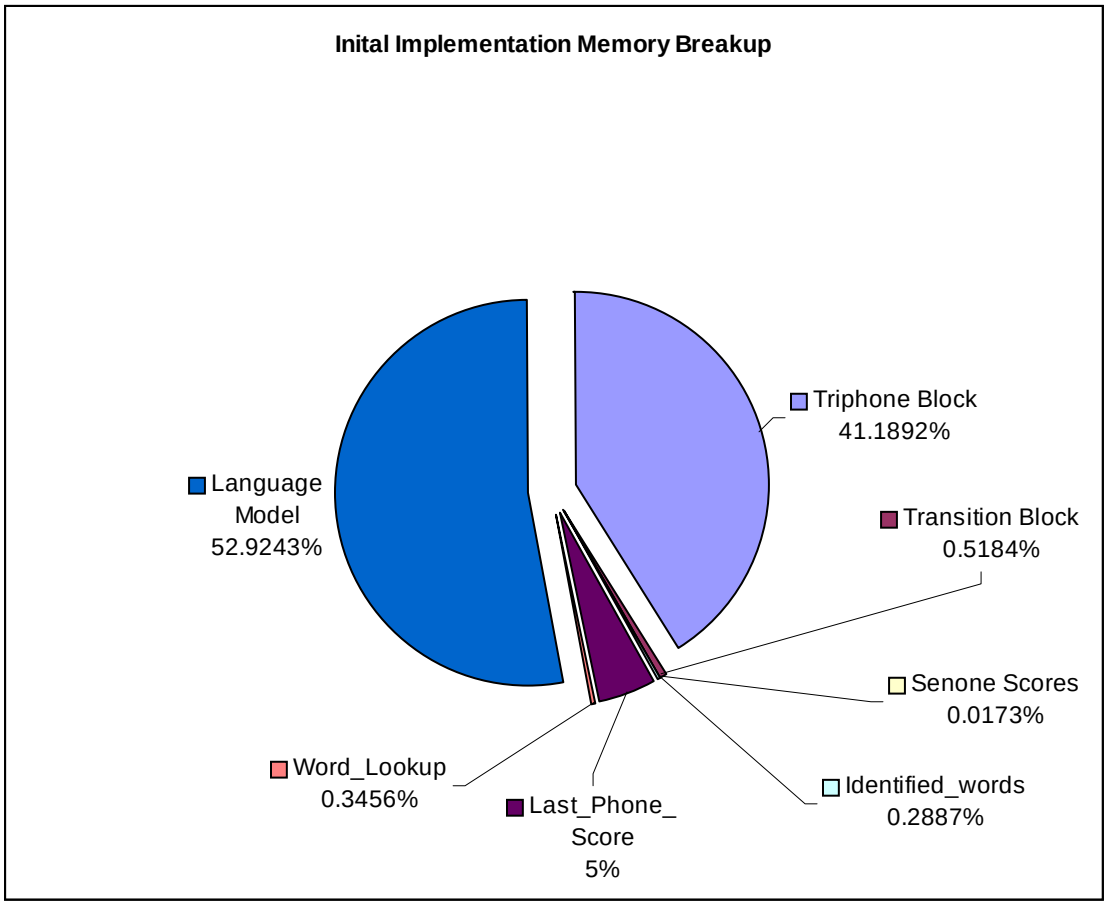



Figure 7.6 – Initial Implementation Memory Breakup

The total memory usage is slightly different from our initial estimates in Section 6.2. Nevertheless, we see that we have managed to reduce the total memory requirement from the initial 277.755 MBytes to about 181.38 MBytes – savings of about 35%. More significantly, the language model now takes up about 81% of the total required memory. As mentioned before, a) the only way to reduce this model size is to adopt a radically new method of language model training and is outside the scope of this research, and b) while the total memory requirement of this block is high, its total bandwidth requirement is low. Secondly, the Triphone_Block now takes up only a small fraction of the total memory requirements - less than 9% as compares to the previous 41% - which was our goal to begin with since this is the bandwidth hungry memory block.

Table 7.2 – Memory access breakup for single frame

| STAGE | | MBYTES |
|---|---|---|
| First n-1 triphones | Triphone Block | 3.63375 |
| | Senone Scores | 1.632 |
| | Transition Scores | 2.448 |
| | Total | 7.71375 |
| | | |
| Last set of 50 triphones | Triphone Block | 4.9175 |
| | Senone Scores | 2.24 |
| | Transition Scores | 3.36 |
| | Total | 10.5175 |
| | | |
| Language Model | Trigram search | 0.01966 |
| | Bigram search | 0.2949 |
| | Unigram search | 3.244 |
| | Total for Language Model Lookup | 3.55856 |

Table 7.3 –Memory access breakup by phase and memory unit for single frame

| | | MBYTES |
|---|---|---|
| By Phase | Phase1 | 7.71375 |
| | Phase2 | 0.013 |
| | Phase3 | 10.5175 |
| | Phase4 | 3.55856 |
| | | |
| By Memory Unit | SDRAM1 | 9.64032 |
| | SDRAM2 | 8.2905 |
| | SRAM | 3.872 |

Table 7.2 shows the average memory access requirement for the different memory blocks as well as the language model lookup. Table 7.3 shows the memory access breakup by phase and by memory unit. This information is used to estimate the memory power savings of our design as shown in Table 7.4. Note that we need to convert the data in Table 7.3 from *MBytes* to *number of 16 bit accesses* which is used in Table 7.4. Note that data for the old design was obtained from Figure 6.7. SDRAM operating value data

were obtained from the datasheets[97]. SRAM operating value data was obtained from [70].

Table 7.4 – Memory Power Savings

| | VOLTAGE | CURRENT | ACCESS TIME | BW (Macc/sec) | POWER(mW) |
|---|---|---|---|---|---|
| New Design | | | | | |
| | | | | | |
| SDRAM1 | 3.3 | 190m | 5.5n | 482.016 | 1662.16 |
| SDRAM2 | 3.3 | 190m | 5.5n | 414.5248 | 1429.44 |
| SRAM | 1.8 | 20m | 70n | 193.6 | 487.84 |
| | | | | | 3579.44 |
| Old Design | | | | | |
| | | | | | |
| SDRAM | 3.3 | 190m | 5.5n | 1292.678 | 4457.8 |
| SRAM | 1.8 | 20m | 70n | 313.6 | 790.272 |
| | | | | | 5248.072 |

External memory power consumption forms a large portion of the total power consumption of speech recognition systems, but is largely ignored in most studies. Our lexical tree design gives a 32% improvement in external memory power consumption. It is clear that reducing the bandwidth requirement of such designs using schemes like those used in this research is a huge contributor to reducing the overall power requirements of the system.

As before, we also evaluate our system with the benchmarks and systems described at the beginning of this chapter to put our design in perspective. These results compare *only* the Viterbi Decoder unit itself to other such designs and do not reflect the underlying architectural power savings and memory power savings that our unit provides. The results shown are for 8 of our units (required for real-time operation). Figure 7.7 and Figure 7.8 show the process normalized steady state power consumption and the energy efficiency, respectively, of the different implementations – XScale, Pentium 4 processor,

the Perception Processor and our Viterbi Decoder. We achieve an improvement of 3 orders of magnitude in both power consumption and energy efficiency over the software implementation on the Pentium 4 processor as was expected. We also achieve 1 order of

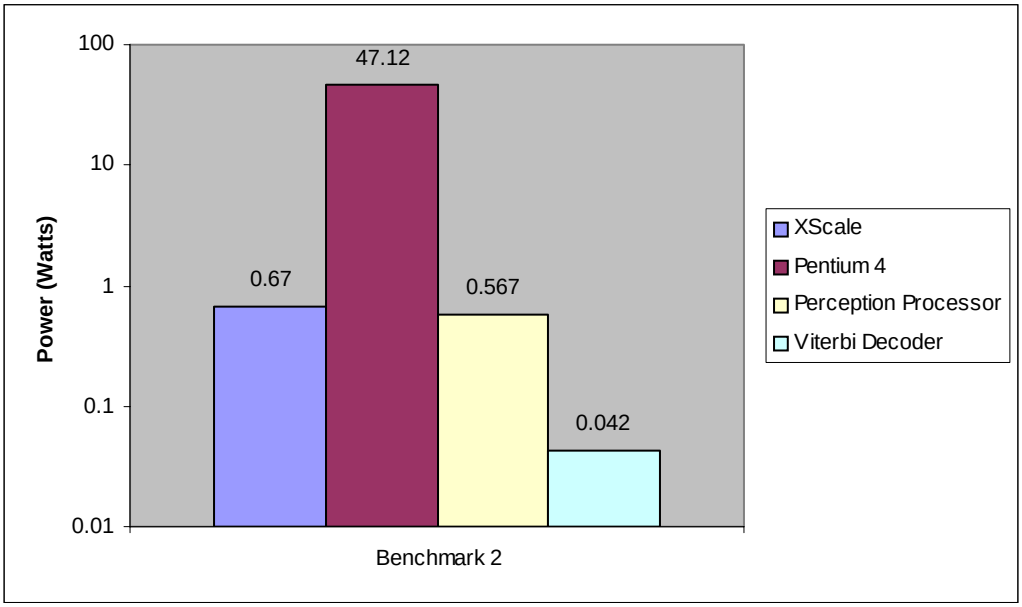



Figure 7.7 - Power Consumption across systems – Benchmark 2

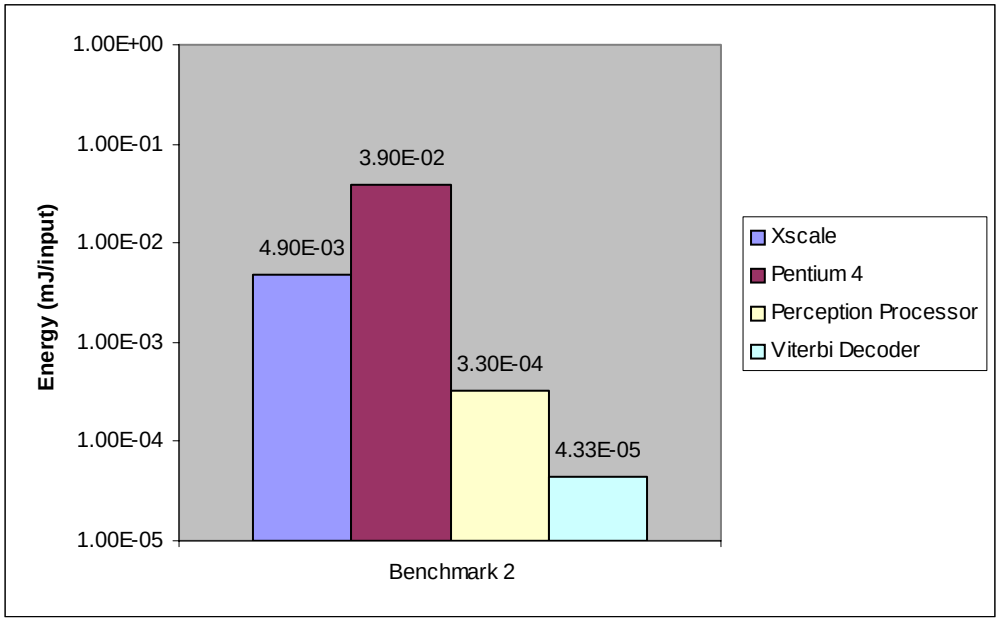



Figure 7.8 – Process Normalized Energy Consumption across systems – Benchmark 2

magnitude improvement in power consumption and energy efficiency over the viterbi decoder phase of the Perception Processor (note that this processor is designed especially for the speech recognition task).

# CHAPTER 8

# Conclusion

The desirability of portable operation of electronic systems and the applications they perform has become very clear in recent years. A major factor that governs the weight and size of such portable systems is the amount of batteries they carry, which in turn is governed by the power dissipated by such devices. The cost of providing this power, and the associated cooling has led to significant interest in power reduction for such applications. However, until recently, there had not been a major focus on design methodologies for such systems with power reduction in mind; the focus rather was on faster clock rates and logic speeds. This dissertation takes a different viewpoint, focusing on the design of a speech recognition system with power reduction in mind, while meeting the real-time requirements of the system, and supporting a large vocabulary. While system level design techniques such as sharing execution units and buses and reducing activity for arithmetic computation have been followed, the goal of this research was not to achieve power using standard power reduction such as voltage and frequency scaling, but to achieve power reduction of speech recognition applications at the algorithmic and structural level. It was envisioned that standard techniques could be applied *on top of* the power reduction schemes developed in this study.

We have designed dedicated low power hardware accelerators for the gaussian estimation phase and the viterbi-decoding phase – the two resource hungry and computation intensive parts of the speech recognition system. These specialized units allow us to achieve real-time speech recognition within the power budget of portable devices. While the power savings itself is a step towards porting our gaussian estimator design to mobile domains, what is more important is perhaps the degree of flexibility that our design incorporates. With new techniques emerging continually in speech recognition, it is important that any hardware accelerator built will be able to incorporate these techniques at least to some extent, and take advantage of the savings they provide. Much work on energy reduction has taken place in the circuit and device technology domains [8], and there has been an increasing emphasis on designing for power efficiency at the architectural level. This implies that the desired energy and power goals must be targeted early in the design cycle and that the system microarchitecture must work in concert with advances in circuit technology to reduce power demands. Our design has been able to incorporate new techniques in speech recognition and use it to reduce power consumption at the algorithm level.

Note that Figure 7.3 and Figure 7.4 show the savings achieved by the gaussian estimator *before* taking into account algorithmic savings. The fact that the design is low power to start off with, and that its potential can be further extended by taking advantage of the flexible nature of the design to incorporate algorithm-level power and energy savings makes this an attractive solution to the gaussian estimation phase of the speech recognition system.

Similarly by restructuring our memory and adopting the lexical tree dictionary style, we have achieved our goal of reducing both the total memory required as well as reducing memory bandwidth and power consumption of the viterbi decoder unit. It is important to note that while the unit itself is low power, and reduces activity and power by eliminating redundant calculations and operations, the lexical trees structure also reduces the total external memory power consumption. The ability of this design to successfully address the problems with switching to the lexical tree structure thus taking advantage of the improvements it provides makes it a viable solution to the viterbi decoding phase of the speech recognition system.

## 8.1 Summary

### 8.1.1 Real time performance & area

Three of our Gaussian Estimation units working in parallel achieved real-time performance for 6000 senone updates per 10ms time frame. In addition by adapting to the representative layer techniques we achieve a speed up of up to .5x RT (real time). Eight of our Viterbi Decoder units working in parallel achieved real-time performance for a 60000-word vocabulary size. The total area for our chip was 11.825 $mm^2$. Adding 100% for post routing and packaging, we get a total area of 23.6 $mm^2$.

In comparison the latest Speech-in-Silicon chip from CMU has a area of 10 $mm^2$ that supports only 5000 words with a real time performance of .6 x RT. Assuming that this chip can support twice as much words in real-time, our design still manages to support 6 times as much vocabulary for about 2.5 times the area.

### 8.1.2 Memory Requirement

Using our lexical tree structure, and our dynamic memory allocation scheme, we have manage to reduce the total required memory for the Viterbi Decoding part from 227.75MBytes to 181.38Mbytes (a 35% reduction). Adding the 15.16MBytes required by the Gaussian Estimation part, the total memory requirement is now 196.54MBytes. We also achieved a 7x reduction in the size of the high traffic Triphone_Block (from 114.405MBytes to 15.38MBytes).

### 8.1.3 Memory Bandwidth requirement

The required memory bandwidth for the Gaussian Estimator unit was about 15Mbytes per frame. The memory bandwidth requirement for the Viterbi Decoder was about 18MBytes per frame for the Triphone_Block update for the lexical structure. This represented a 36% improvement over the previous flat structure. The bandwidth requirements are also balanced between both DRAM units ensuring parallel operation.

### 8.1.4 Power & Energy

The Gaussian Estimation Unit has a worst-case power budget of 3x245mW and a best-case power rating of 3x121mW. The design also achieves a 43% improvement in power and 35% improvement in energy consumption over the previous comparable ASIC implementation before the savings due to adaptation are taken into account. As compared to the state-of-the-art software implementation of SPHINX on a general-purpose processor, the design also shows 2 orders of magnitude improvement in power and 3 orders of magnitude in energy consumption. Clearly an ASIC solution is needed to push

the speech recognition application to within the power and energy budgets of the mobile domain.

While the ASIC implementation of the Viterbi Decoder is power efficient, most of the power savings for the unit itself can be attributed to the reduction in calculations and elimination of redundancy due to the lexical tree structure and the dynamic memory allocation. The total power required by the Viterbi Decoder unit(s) was 80.5mW. The design achieves 3 orders of magnitude improvement in both power and energy consumption compared to the software implementation of SPHINX on a general purpose processor, and 1 order of magnitude improvement in both power and energy consumption when compared to the closest custom processor for speech recognition – the perception processor. These results are before applying the additional savings due to the BW and operation reductions due to the new structure.

In addition the reduced BW requirement has lead to a 32% reduction in external memory power consumption (from about 5.2W to 3.6W) – a factor ignored by most studies. Clearly reducing the BW requirements is a huge contributor to reducing the overall power consumption of a system such as this.

## 8.2 Contributions

The main contributions of this dissertation are:

- We re-designed the Gaussian Estimation unit to be flexible, thus allowing it to recast itself to algorithmic changes at 3 out of the 4 layers. This allows it to harvest the power and energy savings offered by these algorithms – savings that

can be applied *on top of* standard techniques such as voltage and frequency scaling.

- We successfully transitioned from the flat dictionary to the lexical tree dictionary – a move that has led to a reduced memory requirement, a reduced memory bandwidth requirement, reduced external memory power consumption, and elimination of redundant calculations and memory access. A simple but novel memory structure was used to solve the mapping problems associated with this shift while maintaining a structure that did not require a complicated network of LUTs.
- A novel 'timestamp' technique were used to solve the problems associated with word-to-word transitions .
- A dynamic memory allocation scheme was used to reduce the total memory requirement for the Viterbi Decoding phase for the last set of 50 triphones of every word.
- The search sequence was analyzed for optimizations and parallelism, and the design was partitioned in a manner that allowed for these optimizations to be incorporated.

# REFERENCES


[1] Rabiner, L. R.: A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE* Vol 77, 2 Dec. 1989, 257–286.

[2] http://www.speech.cs.cmu.edu/comp.speech/Section6/Q6.5.html

[3] http://amp.ece.cmu.edu/projects/SIS/

[4] Mathew, B.: The Perception Processor.

[5] Krishna, R., Mahlke, S., Austin, T.: Architectural Optimizations for Low-Power, Real-Time Speech Recognition, *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems CASES '03,* October 2003.

[6] Melnikoff, S. Quigley, S.F: Performing speech recognition on multiple parallel files using continuous hidden Markov models on an FPGA, *Proc. IEEE International Conference on Field Programmable Technology (FPT 2002)*, 2002, pp.399-402.

[7] Young, S: Large Vocabulary Continuous Speech Recognition. *IEEE Signal Processing Magazine*, 13(5), 1996, 45-57.

[8] Hagen, A., Pellom, B., Connors, D. A.: Analysis and Design of Architecture Systems for Speech Recognition on Modern Handheld-Computing Devices, *Proceedings of the of the 11th International Symposium on Hardware/Software Codesign.* October, 2003.

[9] Buchsbaum, A. Giancarlo, R: Algorithmic aspects in Speech Recognition - An Introduction.

[10] Srivastava, S.: Fast gaussian evaluations in large vocabulary continuous speech recognition. M.S. Thesis, Department of Electrical and Computer Engineering, Mississippi State University, Oct. 2002.

[11] Waibel, A., Lee, K.,: Readings in speech recognition, Morgan Kaufmann Publishers Inc. 1990.

[12] Gupta, Vishwa N.,: Phoneme based speech recognition, *The Journal of the Acoustical Society of America*, Vol 98, Issue 3, September 1995.

[13] Huang, X., Alleva, F., Hon, H.-W., Hwang, M.-Y., Lee, K.-F., Rosenfeld, R.: The SPHINX-II speech recognition system: an overview. *Computer Speech and Language*, Vol 7, Issue 2 ,1993, 137–148.

[14] Fosler-Lussier,E.,: Dynamic Pronunciation Models for Automatic Speech Recognition, Ph.D. thesis, University of California, Berkeley, 1999.

[15] Soong, J.K.C, Lin-Shan, F.K.: Large vocabulary word recognition based on tree-trellis search, *Acoustics, Speech, and Signal Processing, 1994. ICASSP-94., 1994 IEEE International Conference on*, Vol 2, Apr 1994, 137-140.

[16] Fissore, L. Laface, P. Micca, G. Pieraccini, R.,: Lexical access to large vocabularies for speech recognition**,** *Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on*, Vol 37, Issue 2, Aug 1989, 1197-1213.

[17] Jang, Jyh-Shing Roger / Lin, Shiuan-Sung (2002): "Optimization of viterbi beam search in speech recognition", *ISCSLP 2002*, paper 114.

[18] Antoniol, G. Brugnara, F. Cettolo, M. Federico, M.: Language model representations for beam-search decoding, *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, Vol 1, May 1995, 588-591.

[19] Ney, H. Haeb-Umbach, R. Tran, B.-H. Oerder, M.: Improvements in beam search for 10000-word continuous speech recognition, *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*, Vol 1, March 1992, 9-12.

[20] Alleva, F. Huang, X. Hwang, M.-Y.: An improved search algorithm using incremental knowledge for continuous speech recognition, *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, Vol 2, Apr 1993, 307-310.

[21] Clarkson, P., Rosenfeld. R.: Statistical language modeling using the CMU-Cambridge toolkit. *Eurospeech '97*.

[22] Suhm, B., Waibel, A.: Towards Better Language Models for Spontaneous Speech. *ICSLP 94*, Yokohama, Vol. 2, pp. 831-834.

[23] Baggia, P., Gauvain, J.L., Kellner A., Perennou G., Popovici C., Sturm J., Wessel F.: Language Modelling and Spoken Dialogue Systems - the ARISE experience,

*Proc. Sixth European Conference on Speech Communication and Technology*, Budapest, Hungary, September 1999.

[24] GoodMan, J., Chen, S.F.: An empirical study of smoothing techniques for language modeling, *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, 1996, 310 – 318.

[25] Kneser, R. Ney, H.: Improved backing-off for M-gram language modeling, *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, Vol 1, May 1995, 181-184.

[26] Gopalakrishnan, P.S., Nahamoo, D.: Models and algorithms for continuous speech recognition - a brief tutorial, *Circuits and Systems, 1993., Proceedings of the 36th Midwest Symposium on*, Vol 2, Aug 1993, 1535-1538.

[27] Sharp ,R., Bocchieri, E.,: The Watson speech recognition engine, *Proceedings of the IEEE Int. Conf. Acoust., Speech, Signal Processing*, May 1997.

[28] Ellermann, C., Even, S. V., Huang, C., and Manganaro, L.: Dragon Systems' Experiences in Small to Large Vocabulary Multi-Lingual Speech Recognition Applications. *EUROSPEECH, 1993*, Vol 3, 2077-2080.

[29] Chevalier, H., Ingold, C., Kunz, C., Moore, C., Roven, C. et. al: A Large Vocabulary Speech Recognition in Specialized Domains. *ICASSP, 1995*, Vol 1, 217-220.

[30] Peskin, B., Gillick, L., Liberman, N., Newman, M., van Mulbregt et. al: Progress in Recognizing Conversational Telephone Speech. *ICASSP, 1997*, Vol 3, 1811-1814.

[31] Alleva, F.: Search organization in the Whisper continuous speech recognition system: *Automatic Speech Recognition and Understanding, 1997. Proceedings. 1997 IEEE Workshop on*, Dec 1997, 295-302.

[32] Klein, D., Manning, C. D.: A parsing - fast exact Viterbi parse selection, *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1 NAACL '03*, May 2003.

[33] Bahl, L., Balakrishnan-Aiyer, S., Bellegarda, J., Franz, M., et al.: Performance of the IBM Large Vocabulary Continuous Speech Recognition System on the ARPA Wall Street Journal Task, *EUROSPEECH, 1995*, Volume 1, 41-44.

[34] Bakis, R., Chen, S., Gopalakrishnan, P., Gopinath, R., Mes, S.,Polymenakos, L.: Transcription of broadcast news shows with the IBM Large Vocabulary Speech Recognition System. *DARPA 1997 Speech Recognition Workshop*.

[35] Gopalakrishnan, P., Bahl, L., Mercer, R. A.: A tree search strategy for large vocabulary continuous speech recognition, *ICASSP, 1995*, Vol 1, 572-575.

[36] Gopalakrishnan, P., Nahamoo, D., Padmanabhan, M., Pincheny, M. A.: A Channel-Bank-Based Phone Detection Strategy. *ICASSP, 1994*, Vol 2, 161-164.

[37] Ljolje, A., Riley, M., Hindle, D., Pereira, F.: The AT&T 60,000 Word Speech-To-Text System, *Proceedings of the ARPA SLT Workshop 1995*.

[38] Mohri, M. and Riley, M.: Weighted Determinization and Minimization for Large Vocabulary Speech Recognition. *EUROSPEECH, 1997*, Vol 1, 131-134.

[39] Phillis, S. and Rogers, A.: Parallel Speech Recognition, *EUROSPEECH, 1997*, Vol 1, 242-245.

[40] http://research.microsoft.com/srg/

[41] Alleva, F., Huang, X., Hwang, M.-Y.: Improvements on the Pronunciation Prefix Tree Search Organisation, *ICASSP, 1996*, Vol 1, 133-136.

[42] http://www.speech.sri.com/projects/decipher/

[43] Murveit, H., Butzberger, J.: Performance of SRI's Decipher Speech Recoginition System on DARPA's CSR Task, *Proceedings of the workshop on Speech and Natural Language*, 1992, 410-414.

[44] Murveit, H., Butzberger, J., Digalakis, V., Weintraub, M.: Large-Vocabulary Dictation using SRI's DECIPHER Speech Recognition System - Progressive Search Techniques. *ICASSP, 1993*, Vol 2, 319-322.

[45] http://www.speechatsri.com/products/enablers.shtml

[46] ftp://svr-ftp.eng.cam.ac.uk/pub/comp.speech/info/VoiceRecognitionProcessors

[47] http://www.sensoryinc.com/

[48] http://www.voicecontrol.com/

[49] http://www.voicegate.com/voiceics.htm

[50] http://www.futurlec.com/News/Philips/SpeechChip.html

[51] http://cmusphinx.sourceforge.net/html/cmusphinx.php

[52] Ravishankar, M. K.: Efficient Algorithms for Speech Recognition, PhD thesis, Carnegie Mellon University.

[53] Ravishankar, M., Bisiani, R., Thayer, E.: Sub-Vector Clustering to Improve Memory and Speed Performance of Acoustic Likelihood Computation. *EUROSPEECH, 1997*, Vol 1, 151-154.

[54] Lee, K.-F., Alleva, F.: Continuous Speech Recognition, Advances in Speech Signal Processing, Pub. Marcel Dekker.

[55] Huang, X., Alleva, F., Hon, H.-W., Hwang, M.-Y., Rosenfeld, R.: The SPHINX-II Speech Recognition System. Technical Report CS-92-112, Carnegie Mellon University (USA).

[56] Hwang, M.-Y., Alleva, F., Huang, X.: Senones, Multi-Pass Search, and Unified Stochastic Modeling in SPHINX-II, *EUROSPEECH, 1993*, Vol 3, 2143-2146.

[57] Renals, S. Hochberg, M. E.: Efficient Search Using Posterior Phone Probability Estimates, *ICASSP, 1995*, Volume 1, 596-599.

[58] Hochberg, M., Renals, S., Robinson, A. J., Cook, G. D.: Recent Improvements to the ABBOT Large Vocabulary CSR System, *ICASSP, 1995*, Vol 1, 69-72.

[59] Hochberg, M. M., Cook, G. D., Renals, S. J., Robinson, A. J., Schechtman, R. S.: The 1994 ABBOT Hybrid connectionist-hmm Large-Vocabulary Recognition System, *Proceedings of the ARPA SLT Workshop 1995*, 1995.

[60] Woodland, P. C., Gales, M. J. F., Pye, D., Young, S.: Broadcast news transcription using HTK, *DARPA 1997 Speech Recognition Workshop*.

[61] Woodland, P. C., Leggetter, C. H., Odell, J. J., Valtchev, V., Young, S. J.: The Development of the 1994 HTK Large Vocabulary Speech Recognition System, *Proceedings of the ARPA SLT Workshop 1995*

[62] http://cslr.colorado.edu/beginweb/speech_recognition/sonic.html

[63] Pellom. B., Hacioglu, K.: Recent Improvements in the CU Sonic ASR System for Noisy Speech: The SPINE Task, *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP) 2003*, April, 2003.

[64] Pallett, D., Przybocki, M. A.: 1996 preliminary broadcast news benchmark tests. *Proceedings of the 1997 DARPA Speech RecognitionWorkshop* (Feb. 1997).

[65] Stern, R. M.: Specification of the 1996 hub 4 broadcast news evaluation.

[66] Agaram, K., Keckler, S. W., Burger, D.: A characterization of speech recognition on modern computer systems, *Proceedings of the 4th IEEE Workshop on Workload Characterization,* (Dec. 2001).

[67] Lai, C., Lu, S.-L., Zhao, Q.: Performance analysis of speech recognition software, *Proceedings of the Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads* (Feb. 2002).

[68] Anantharaman, T., Bisiani, R.: A hardware accelerator for speech recognition algorithms, *Proceeedings of the 13th International Symposium on Computer Architecture* (June 1986).

[69] Pihl, J., Svendsen, T., Johnsen, M. H.: A VLSI Implementation of Pdf Computations in HMM Based Speech Recognition. In *Proceedings of the IEEE Region Ten Conference on Digital Signal Processing Applications (TENCON'96)*, Nov 1996.

[70] Nedevschi, S., Patra, R. K.: Hardware speech recognition for user interfaces in low cost, low power devices, *DAC '05: Proceedings of the 42nd annual conference on Design automation* (New York, NY, USA, 2005), ACM Press, 684-689.

[71] Mathew, B., Davis, A. Fang, Z.: A Low-Power Accelerator for the SPHINX 3 Speech Recognition System, *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '03)*, October 2003, 210-219.

[72] Yuanyuan, S., Jia, L., Runsheng, L.: Single-chip speech recognition system based on 8051 microcontrollercore, *Consumer Electronics, IEEE Transactions on*, Vol 47, Issue 1, Feb 2001, 149-153.

[73] Borgatti, M., Felici, M., Ferrari, A., Guerrieri, R.: A low-power integrated circuit for remote speech recognition, *Solid-State Circuits, IEEE Journal of*, Vol 33, Issue 7, Jul 1998, 1082-1089.

[74] Peckham, J., Green, J., Canning, J., Stephens, P.: LOGOS - A real time hardware continuous speech recognition system, *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '82*, Vol 7, May 1982, 863-866.

[75] Vargas, F., Fagundes, R.D., Barros, D.: A New Approach to Design Reliable Real-Time Speech Recognition Systems, *On-Line Testing Workshop, 2001. Proceedings. Seventh International*, July 2001, 187-191.

[76] Weintraub, M., Chen, G., Mankoski, J., Murveit, J.: Hardware for Hidden Markov-Model-Based, Large-Vocabulary Real-Time Speech Recognition, *Proceedings of the workshop on Speech and Natural Language, 1990*.

[77] Fernandez, J. M., Moreno, F., Alexandres, S., Meneses, J.: A flexible VLSI-based hardware system for medium-large-vocabulary real-time speech recognition, *Selected papers of the short notes session on Euromicro '94* , 825-828.

[78] csdl.computer.org/comp/mags/co/2006/11/ry015.pdf

[79] Lin, E., Yu. K., Rutenbar, R., Chen, T.: Moving Speech Recognition from Software to Silicon: the In Silico Vox Project, *Proceedings of Interspeech 2006* Sept 2006.

[80] DeHon, A.: DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, 1994*, IEEE Computer Society Press, 31–39.

[81] Lysaght, P.: Aspects of dynamically reconfigurable logic, 1999.

[82] Memik, S., Bozorgzadeh, E., Kastner, R., Sarrafzadeh, M.: A strategically programmable system, 2001.

[83] Melnikoff, S.J., James-Roxby, P.B., Quigley, S.F., Russell M.J.: Reconfigurable Computing for Speech Recognition: Preliminary Findings, Book Series – Lecture Notes in Computer Science.

[84] Melnikoff, S.J., James-Roxby, P.B., Quigley, S.F., Russell M.J.: Implementing a Simple Continuous Speech Recognition System on an FPGA, *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*

[85] Vargas, F.L., Fagundes, R.D.R., Junior, D.B.: A FPGA-based Viterbi algorithm implementation for speech recognition systems, *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on*, Vol 2, 2001, 1217-1220.

[86] Bocchieri, E.: Vector quantization for efficient computation of continuous density likelihoods. *ICASSP, 1993,* Vol 2, 1993, 692-695.

[87] Hwang, M. H., Huang, X.: Subphonetic Modeling with Markov States – Senone, *IEEE Int. Conf. Acoust., Speech, Signal Processing*, 1992, 33-36.

[88] Lee, A., Kawahara, T., Shikano, K.: Gaussian mixture selection using context-independent hmm. *IEEE ICASSP*, 2001.

[89] Woszczyan, M.: Fast Speaker Independent Large Vocabulary Continuous Speech Recognition. *Universitat Karlsruhe; Institut fur Logik, Komplexitat and Deduktionssysteme*, 1998.

[90] Mosur, R., Singh, R., Raj B. Stern, R.M.: The 1999 CMU 10X Real Time Broadcast News Transcription System, *2000 Speech Transcription Workshop*.

[91] Ortmanns, S., Ney, H., Coenen, N. Eiden, A.: Lookahead Techniques for Fast Beam Search, *ICASSP 1997*.

[92] Gales, M. J. F., Knill, K. M. Young, S.: Use of Gaussian Selection in Large Vocabulary Continuous Speech Recognition Using HMMs, *ICSLP 1996.*

[93] Mosur R., Bisiani, R., Thayer, E.: Sub-Vector Clustering to Improve Memory and Speed Performance of Acoustic Likelihood Computation, *Eurospeech*, Sep 1997.

[94] Bocchieri, E. Mak, B.: Subspace Distribution Clustering for Continuous Observation Density Hidden Markov Models, *EUROSPEECH, 1997*, Vol 1, 107-110.

[95] Chan, A., Mosur, R., Rudnicky, A., Sherwani, J.: Four layer categorization scheme of fast gmm computation techniques in large vocabulary continuous speech recognition systems. In *Intl. Conf. on Spoken Language Processing*, 2004, 689-692.

[96] http://www.ece.ncsu.edu/msl/ncsu/manuals/0402044.pdf

[97] http://download.micron.com/pdf/datasheets/dram/sdram/128MbSDRAMx32.pdf

[98] http://bwrc.eecs.berkeley.edu/Publications/1995/Min_pwr_consump_**CMOS**_crct /paper.fm.pdf

[99] Smith, Franzon, P.: Verilog Styles for Synthesis of Digital Systems, 2000, Prentice Hall.

[100] Lee, C., Soong, F. K., Paliwal, K. K.: Automatic speech and speaker recognition: advanced topics, 1996

[101] Tong, Y. F., Rutenbar, R.: Minimizing floating-point power dissipation via bit-width reduction, *Proceedings of the 1998 International Symposium on Computer Architecture Power Driven Microarchitecture Workshop*, 1998.

[102] Huang, X., Acero, A.: Spoken Language Processing – A guide to Theory, Algorithm and System Development, Prentice Hall (2001).

[103] http://www.speech.cs.cmu.edu/sphinx/models/

# APPENDIX

# Appendix A

## DRAM Controller

We had used the 128Mb (x32) Micron SDRAM - MT48LC4M32B2 -for our simulations, and needed to develop a DRAM controller to interface with this memory unit as well as take care of the initialization sequence, the precharge delay, auto refresh etc. while also generating the correct control and address signals. The Altera SOPC Builder is used to create and integrate a custom version of the DDR or DDR2 SDRAM controller MegaCore function to a larger system. The Avalon Switch Fabric is used to provide an interface that can seamlessly integrate all Avalon peripherals to the Altera Cyclone 2 FPGA. The problem is that this interface requires at least one Avalon master device such as the NIOS II embedded processor, which serves as the platform for porting the software component of large designs. Our design is entirely built in hardware and does not require the onboard processor. We can however use the DRAM controller generated by the Altera Quartus tool for our own designs. Small changes are necessary to make the generated design work independently with the rest of the hardware.

- The original code generated timing violations which were fixed after the following code was inserted/replaced.
    - A condition for refresh following the initialization sequence is necessary- Immediately following this code:

```
if (init_done)
```

Insert this code:

```
        begin
            if (refresh_request)
                  begin
                        zs_cs_n <= 0;
                        zs_ras_n <= 1;
                        zs_cas_n <= 1;
                        zs_we_n <= 1;
                  end
            else
                  begin
                        zs_cs_n <= 1;
                        zs_ras_n <= 1;
                        zs_cas_n <= 1;
                        zs_we_n <= 1;
end
```

……..rest of code…..

- immediately following this code:

```
9'b001000000: begin
 m_state <= 9'b000000100;
 m_addr <= {12{1'b1}};
```

Insert this code:

```
 if (refresh_request)
     {zs_cs_n, zs_ras_n, zs_cas_n, zs_we_n} <= 4'b0010;
 else
     {zs_cs_n, zs_ras_n, zs_cas_n, zs_we_n}
                                      <={csn_decode,3'h2};
```

- Replace

```
assign cmd_code = {zs_ras_n, zs_cas_n, zs_we_n};
assign cmd_all = {zs_cs_n, zs_ras_n, zs_cas_n, zs_we_n};
```

instead of

```
assign cmd_code = {zs_ras_n, zs_cas_n, {1{1'b0}}};
assign cmd_all = {{1{1'b0}}, zs_ras_n, zs_cas_n,
                                          {1{1'b0}}};
```

- The clock to the SDRAM and the SDRAM controller need to be out of phase (Clk & ~Clk).

- The SOPC builder generates its own test bench when it generates the SDRAM controller. This test bench makes sure that a sufficient startup time elapses before any read/write occur, and is essential for completing the proper startup sequence, as well as for establishing operational mode. In a complete design, we need to ensure this, using alternate methods instead of leaving this to the test bench. We have done this in our design using condition checking in our DRAM unit(s).