A REAL TIME LARGE VOCABULARY SPEECH RECOGNITION SYSTEM

by

Anton Manfred Stölzle

Memorandum No. UCB/ERL M91/109

6 December 1991

To my family,

Christine, Alexander, Sebastian, and Michaela Stölzle

and to my parents,

Helene and Franz Stölzle

A REAL TIME LARGE VOCABULARY SPEECH RECOGNITION SYSTEM

by

Anton Manfred Stölzle

Memorandum No. UCB/ERL M91/109

6 December 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

To my family,

Christine, Alexander, Sebastian, and Michaela Stölzle

and to my parents,

Helene and Franz Stölzle

A Real Time Large Vocabulary Speech Recognition System

by

Ph.D. **Anton Manfred Stölzle** **Department of EECS**

Abstract

This thesis describes hardware for connected speech recognition based on Hidden Markov Models. It can perform real time speech recognition for vocabularies of up to 60,000 words using the Viterbi dynamic programming algorithm. It is not customized to a particular HMM topology, thus supporting multiple recognition systems. To minimize hardware implementation, the recognition algorithm was modified to a pruned frame synchronous beam search, a fixed point number representation using the logarithm of the probability parameters, and a hierarchical HMM representation which allows the Viterbi algorithm to be processed in parallel on two hierarchy levels, the phone level and the grammar level. The specific properties of HMMs on the two hierarchy levels (left-to-right vs. ergodic) motivated different implementations of the Viterbi algorithm (predecessor processing and successor processing). The system configuration is based on the VME bus, and uses 3 custom triple-height boards that include a combination of general purpose hardware and full custom VLSI hardware (2 sets of 6 custom VLSI processors).

Chairman of Committee

To my family,

Christine, Alexander, Sebastian, and Michaela Stölzle

and to my parents,

Helene and Franz Stölzle

Table of Contents

List of Figures

Acknowledgments

A large number of people have contributed towards the successful completion of this project. Above all, I want to thank my advisor, Professor Robert W. Brodersen. He is a great teacher who inspired, guided, and supported me in the past 4 years in so many ways.

In the early phases of the project, I benefitted immensely from numerous discussions I had with Professor Jan Rabaey, Professor Brodersen, and Dr. Hy Murveit. This project contains a large amount of design work, and without the support of many great people it would have been impossible to finish in such a short time. Many thanks to Shankar Narajanaswamy, who designed the ToActiveWord chip set, Robert Yu, who designed the distribution board, and Phil Schrupp, who's knowledge in PCB design was invaluable and who did the physical design of the Viterbi board. Also many thanks to Mani Srivastava, who gave me his parametrized DSP models for the DSP board design, and Brian Richards, who designed the address computation chip, debugged the output distribution board, and who was always available and gave great advice whenever there were problems. This project was a collaboration with the Stanford Research Institute (SRI) in Menlo Park. My thanks to Hy Murveit, Mitch Weintraub, George Chen, and Psi Mankoski. Hy and Mitch made that collaboration possible, and George and Psi spent countless days and nights and weekends to get the hardware working. Also many thanks to the Siemens Research Laboratories in Munich. They made it possible for me to spend these years in Berkeley.

Finally, special thanks to my friends and family. My biggest debt of gratitude goes to my wife, Christine, who gave me so much love and support. She had the strength to take care of our three kids and a stressed graduate student.

1 Introduction

Speech is a natural way of transmitting messages between humans, which is more efficient than handwriting or typewriting. It is therefore desirable to provide a speech recognizing man-machine interface that has the ability to listen to a human voice and to recognize the words spoken.

1.1. Advantages of Speech Recognition

A speech recognition system offers a number of advantages for entering data into a computer. The average word duration for a continuous speech recognition database that contains 1,529 sentences was computed, and the result shows that, in this database, a word has an average length of 0.352 seconds [Lee89], corresponding to a rate of 170 words per minute. If a speech recognition system were available to keep up with this rate, speech input would be clearly faster than keyboard entry. Another advantage is, that controlling equipment or entering data with speech allows hands free of eyes free operation. For example, a driver would not be distracted if he controlled a car radio by voice: he does not have to push buttons, nor look at the radio to locate these buttons. The hands free feature also is desirable for physically handicapped people to work with a computer or to control equipment.

Speech recognition can also be used to make future portable products feasible by reducing their volume and weight. A normal sized keyboard considerably adds to the bulk of a portable product and could be eliminated if a recognizer were used as the entry mechanism.

Speech recognition systems can be classified in the following way:

- isolated words or connected words. Isolated speech recognition is less difficult than connected speech recognition, but, since the user has to pause between each word, it is not user friendly, and the speech rate is by a factor of 2.5 slower than connected speech recognition [Lee89].
- speaker dependent or speaker independent. Speaker dependent systems have a higher recognition accuracy than speaker independent systems, but the recognition accuracy significantly degrades for other users.
- small vocabulary and large vocabulary. As the vocabulary size of a system increases, the number of confusable words grows substantially. Also, in large vocabulary systems each word cannot be modelled individually, and this degrades the recognition accuracy. Large vocabulary typically means a vocabulary of 1,000 words or more [Lee89].

1.2. Algorithms in Speech Recognition

Speech recognition has been an active area of research over the last 40 years, and it yielded speech recognition algorithms that roughly can be classified in four groups: template based systems [Wai90a], knowledge-based systems [Wai90b], connectionist systems [Wai90c], and stochastic systems [Wai90d]. The most successful and most widely used speech recognition approach is stochastic modelling, in particular, stochastic modelling using hidden Markov models (HMM) in conjunction with the Viterbi algorithm (2. 3. 1.) for recognition [Wai90d]. The hardware described

in this thesis is used to perform the Viterbi algorithm for HMM based speech recognition systems.

1. 2. 1. Template Based Speech Recognition [Wai90a]

Template based speech recognition systems have a database of prototype speech patterns (templates) that define the vocabulary. The generation of this database is performed during the training mode. During recognition, the incoming speech is compared to the templates in the database, and the template that represents the best match is selected. Since the rate of human speech production varies considerably, it is necessary to stretch or compress the time axes between the incoming speech and the reference template. This can be done efficiently using a dynamic programming based strategy called dynamic time warping (DTW). Template based systems usually are speaker dependent, so each person who wants to use the system has to generate a personal template database by uttering each vocabulary word several times. Usually, each vocabulary word has its own template, and therefore this method becomes impractical as the vocabulary size is increased (>1,000 words). Template based systems have been most successful for speaker dependent, isolated word recognition, however, there are methods to extend these systems to connected speech [Sak79], or towards speaker independence [Rab79].

1. 2. 2. Knowledge Based Speech Recognition [Wai90b]

Knowledge based speech recognition systems incorporate expert speech knowledge that is, for example, derived from spectograms, linguistics, or phonetics. The "existence proof" that speech recognition can be performed using a multitude of knowledge sources comes from experiments with expert spectogram[1] readers. An expert was able to segment spectograms of discrete and continuous speech into phonetic units

1. In a spectrogram, the energy of the speech signal in different frequency bands is graphed against time. The intensity of the image at a certain frequency-time point indicates the energy of the speech signal corresponding to that point.

(100% correct for isolated speech, 97% correct for connected speech), and label these units with an error rate of 7% to 19% [Zue85]. The goal of a knowledge based speech recognition system is to incorporate this knowledge using rules or procedures. The drawback of these systems is the difficulty of quantifying expert knowledge and to integrate the multitude of knowledge sources [Wai90b]. This gets increasingly difficult if the speech is continuous, and the vocabulary size is increased. The knowledge based speech recognition system, HEARSAY, developed at CMU, is a speaker-dependent continuous recognition system with a vocabulary of 1011 words. Using a very restrictive syntax (perplexity[2] 4.5), it achieved a recognition accuracy of 87% [Les75].

1. 2. 3. Stochastic Speech Recognition Systems [Wai90d]

The most widely used and most successful speech recognition approach is stochastic modelling. Here, probabilistic models of speech are used to deal with incomplete information or uncertainty. The most widely used model is the hidden Markov model (HMM). It uses *states* that model generic speech sounds and transitions between the states with associated *transition probabilities* to model the temporal behavior of speech. This model assumes that speech was produced by a hidden Markov process. At any given time the process occupies one state in the HMM, and this state outputs a small segment of speech (observation) based on a probability distribution that gives the likelihood that a certain speech sound could have been produced by that state (*output probability*). Then, the speech process makes a state transition based on the transition probabilities between the states.

To derive these HMM parameters (output and transition probabilities), an efficient estimate-maximize algorithm, the forward-backward algorithm, is often used [Wai90d]. Because of it's efficiency, it is possible to derive these parameters from a

2. The perplexity Q is an information theoretic measure of a tasks difficulty. It is defined as $Q=2^H$, where H is the entropy, or the number of bits necessary to specify the next word using an optimal encoding scheme [Lee89].

large body of speech data (for example, several thousand words for any vocabulary word). Thus, compared to knowledge-based approaches, it is easy to compile knowledge sources into a compiled architecture [Wai90]. For speech recognition, it is necessary to find the most likely state transition given the incoming speech. This state transition can be found using the dynamic programming Viterbi algorithm (see 2. 3. 1.).

A short coming of HMMs is, that speech observations generated by the HMM are only conditionally independent given the underlying state sequence [Ost90]. Another stochastic modelling approach that attempts to overcome this problem are stochastic segment models (SSM) [Rou87]. In this approach, there are templates that model the distributions of entire speech segments consisting of a sequence of observations, and the incoming speech has to be aligned to these template segments (re-sampling transformation). In [Rou87], it was shown that SSMs achieve - for a 350 connected word, speaker-dependent task - an average word recognition accuracy of 83%, while in the same task, a HMM based recognizer achieved 76%. The drawback of SSMs is, however, that recognition requires significantly more computation than HMM based speech recognition [Ost90].

1. 2. 4. Connectionist Speech Recognition Systems [Wai90c]

Connectionist speech recognition is based on artificial neural networks that use learning strategies to organize and optimize a network of processing elements (neurons). These networks are used as classifiers or mapping functions to recognize the incoming speech. Thus, speech knowledge or constraints used for speech recognition are distributed among many, but simple processing elements [Wai90c]. This approach to speech recognition is the youngest, and researchers are investigating a number of approaches. For example, new physiological-based front end processing, and combined recognizers, that implement neural net and conventional recognition approaches are being investigated. Preliminary results look promising, a Time-Delay Neural Network recognizer was compared to a HMM recognizer in the task to recognize the phones "B",

"D", and "G" out of a database of 5,240 Japanese words. For different speakers, the HMM had a recognition accuracy of 90.9% to 97.2%, while the neural net achieved accuracies of 97.5%-99.1% [Wai88].

1.3. Language Processing

The speech recognition described above concentrated on the problem of recognizing speech given an acoustic representation of the speech patterns. However, a human uses many other sources of knowledge that are non-acoustic: for example, knowing the person talking and what he is talking about makes it possible to understand the person, even if there is noise and not every individual word can be understood. These non-acoustic sources of information are collectively called language processing, and modeling these non-acoustical sources is called language modelling [Wai90e].

As an example how language constraints can improve recognition accuracy, let us consider a car radio control application. After a person said the word "*turn*", it is possible to constrain the recognition vocabulary to the words "*on*" or "*off*". This increases recognition accuracy, since this constraint eliminates words that might be acoustically similar and thus hard to distinguish. On the other hand, this grammar constrains the recognition system just to car radio control application, other applications might use a different grammar and the user is constrained to use the appropriate one. This is termed finite state grammar, since there is only a set of allowable sentences which are modelled using fixed networks.

Another language modelling technique is to use statistical methods. One approach is to assume, that the probability of a word depends on the previous N words, and that the probabilities of different words are independent. The most common grammars use N=1 (bigram grammar) or N=2 (trigram grammar). Using such a statistical grammar reduces the perplexity, which is a measure of a task's difficulty. It is

roughly the number of words that can follow a word[3]. Reducing the perplexity, however, does not contribute to speech understanding. For that, models for syntax (which sentences are acceptable), prosody (pitch, loudness, rhythm, stress), and semantics (the meaning of a sentence) have to be integrated to the acoustic recognizer. These methods are powerful to increase the usability for speech recognition systems.

For example, SRI has an experimental airline travel information system (ATIS) that can be used to query a database of flight schedules [Murv91]. The user can ask questions to the system, and the words of these questions are recognized by an HMM recognition system. Then, the semantics of the question is extracted, and a database query generated. The user can speak in a natural way, not constrained to any grammar or vocabulary. The word recognition accuracy of the speech recognition system is 86.4%, which corresponds to a sentence error rate of 60%. Despite this large sentence recognition error, the ATIS system generates a valid database query for 66.2% of the input sentences, for 7.5% of the sentences it generates a false query, and with 26% it generates no query [Pal91].

1.4. Performance and Limitations of Speech Recognition

Ideally, a speech recognition system should be usable by several people, and not just dedicated to a certain person. It should have a high recognition accuracy, a very large vocabulary, have the ability to recognize connected words, be task independent, and operate in real time. Given state-of-the-art speech technology, however, it is necessary to make some compromises to achieve acceptable recognition accuracies (>90%). The task of speech recognition gets more complicated as the system moves from speaker dependence to speaker independence, from discrete words to connected

3. The perplexity Q is an information theoretic measure of a tasks difficulty. It is defined as $Q=2^H$, where H is the entropy, or the number of bits necessary to specify the next word using an optimal encoding scheme [Lee89].

words, from small vocabulary to large vocabulary, and from task dependence to natural tasks. For example, IBM's Tangora system [Jel85] has a large vocabulary (20,000 words), the grammar is not restricted to a certain task, the recognition accuracy averages 94.3%, and it operates in real time. However, the system is speaker dependent and can only recognize discrete words, so the user is confined to speak in an unnatural way. Another example is SRI's DECYPHER system: It has a 1,000 word vocabulary, is speaker independent, accepts natural, connected speech and has a recognition accuracy of 95.6% [Pal90]. However, in order to achieve high recognition accuracy, the vocabulary and the grammar are restricted to a certain task (resource management, [Pri88]). In this task, a statistical bigram grammar is used that reduces the perplexity to 60 (without grammar, any word can follow a given word). Using this grammar, the recognition accuracy for DECIPHER improves from 75.7% (no grammar) to 95.6% [Pal90]. Since DECIPHER is a connected speech recognition system, the recognition algorithm involves more computation than a discrete recognition system (by a factor of 3 [Lee90], [Bah81]), and real time performance is more difficult to achieve.

Figure 1 shows the influences that speaker independence and grammar has on recognition accuracy, and it summarizes the progress that has been made in speech recognition over the last years. In this graph, the word error rates for a specific task (resource management, [Pri89]) is shown over time. It demonstrates, that speaker dependent speech recognition that uses a grammar yields the best recognition accuracy, while speaker independent recognition without grammatical constraints performs worst.

1.5. VLSI for Real Time Speech Recognition

To fully realize the advantage of speech recognition, it is critical that the recognition system operates in real time. This means, it must continuously process the speech input so the user does not have to pause in order for the recognition system to catch up with the computation.

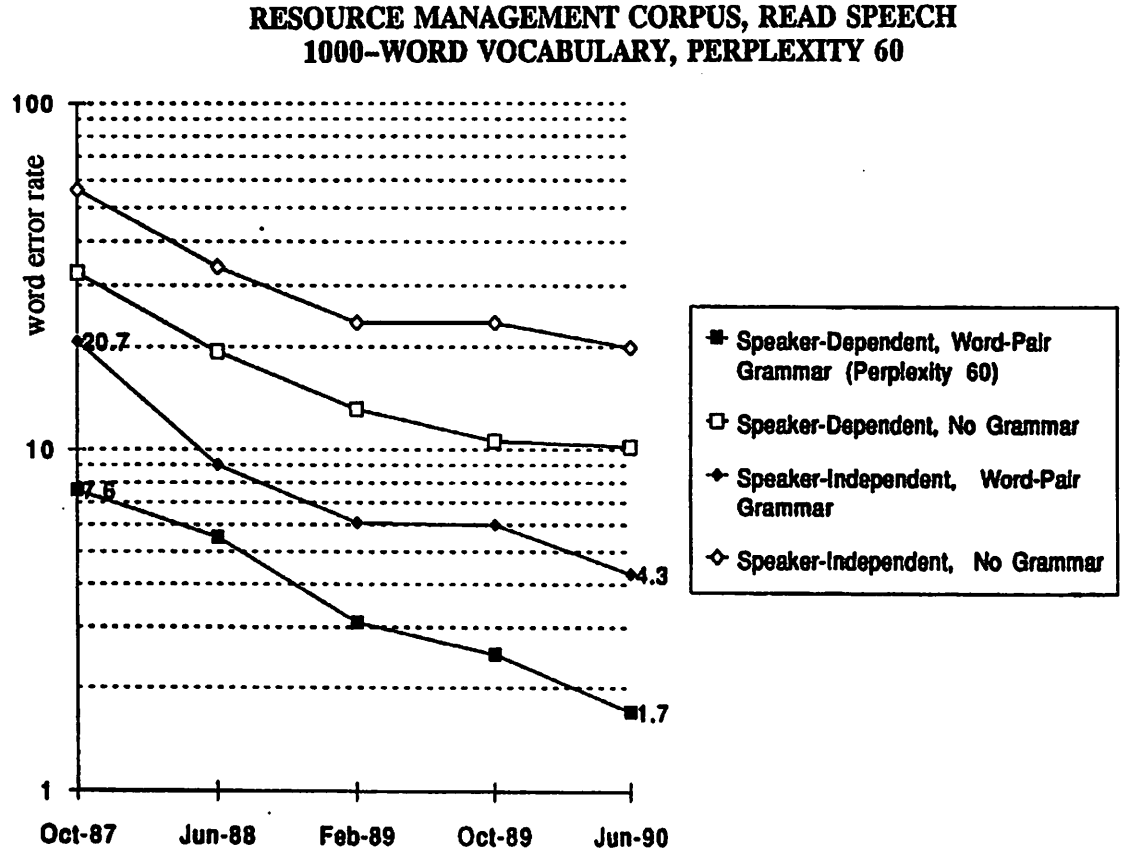


Figure 1: Recognition Accuracies for the Resource Management Task [Way91]

Speech recognition based on HMMs is computational demanding, particularly for continuous speech and large vocabulary. A signal processor, the TMS320C30, can process in real time about 100,000 states of a HMM per second using the Viterbi recognition algorithm, however, at least 400,000 states per second are required for a 1,000 word continuous speech recognition system [Bis89]. Since real time performance is an important issue, commercial and experimental real time recognition systems have been developed, but no system has been reported that can recognize connected speech in real time for a vocabulary that has significantly more than 1,000 words. A few real time systems are described in 3.1. and 3.2.

1. 5. 1. General vs. Special Purpose Implementation

The question is, should a real time speech recognition system be implemented using full custom processors that are tailored to the recognition algorithm, or should systems be used that consist of general purpose processors. The conventional view is, that general purpose systems can be developed much faster and thus cheaper than custom VLSI systems, also it is believed that general purpose systems can be programmed using a high level language which makes them more versatile, since a new recognition algorithm can be supported just by re-writing the software for the general purpose processors. Since speech recognition algorithms are still evolving, this feature is desirable.

However, one of the goals of this thesis is to demonstrate that there is no reason why VLSI systems cannot be developed as fast as general purpose systems. Furthermore, the HMM algorithms have stabilized so that an appropriate custom architecture is able to adapt to new algorithms.

The approach to reduce the time for development of VLSI systems is based on advances that were made in computer aided design (CAD) tools. In particular, the LAGER silicon compilation system can be used to compile VLSI chips using either a hierarchical structural description, or an architectural template in conjunction with microcode [Shu91]. The LAGER system evolved into the SIERA design environment for rapid VLSI *system* prototyping. In SIERA, it is possible to specify an algorithm in a variety of ways including a high level flowgraph and structural description.Within the environment are a number of generation and a synthesis tools which produce a complete structural description using scheduling and re-timing. This structure is then used to compile the individual VLSI processors and to generate PCB layouts or multichip modules. The VLSI systems thus obtained effectively have the algorithm coded in the structure of the datapaths, or in microcode of programmable processors [Rab91].

General purpose systems, on the other hand, often make use of programmable hardware. Thus, the task of scheduling or re-timing is moved to the software level. For the BEAM general purpose real time speech recognition system, 80% of the development time was spent for software development [Bis89]. This time was almost equally divided between support software development, algorithm restructuring, and algorithm coding. 20% of the time was spent for designing and building a custom board that contains three commercial processors. Designing and building a board, however, has to be done for both approaches, and there is no fundamental difference in the time required for the board design. In the system described in the thesis, it took about 5 months to design and simulate the most complex board (Viterbi board), but it took only 2 months to design the full custom processors that already implement the algorithm. If the time is also considered that it took to structure and modify the algorithm, and to design the system architecture, the "time to market" for this full custom system is comparable to the time it takes to design a general purpose system.

Of course, in the same way CAD tools for VLSI systems advanced, there are synthesis tools for software development, for example [Rab91]. Thus, with the advent of these design tools, a high level description of the algorithm can either be synthesized into VLSI, or into code, and the choice between one implementation over the other can be independent of the "time to market" issue, and only depend on issues like cost, performance, or quantity.

The advantage of VLSI systems over systems that use general purpose components is, however, that they can be much more powerful while being small (see 3.2.). The reason for that is, that the architecture of the system and the VLSI chips can be tailored to the specific needs of the algorithm that is implemented. For example, if there is a memory bottleneck, it can be eliminated just by implementing parallel interconnect. Also, VLSI systems only contain the essential hardware needed to perform a particular algorithm, while general purpose systems include a number of

unused features. These are advantages that are decisive for competitiveness, be it production cost or performance.

This thesis describes the algorithm, architecture and implementation of a VLSI-based connected speech recognition system capable of recognizing words using a vocabulary of up to 60,000 words. Thus, it can be used for applications such as word processing where most of the English language is in the recognition vocabulary (Merriam Webster's Seventh New Collegiate Dictionary has 60,000 words). In this system, it is necessary to achieve a high performance: in section 4. 2. 2., we will derive that such a system has to process 20 million states per second (a factor of 50 over what has been previously reported [Bis89]). Thus, it was implemented in an architecture based on 6 full custom VLSI processors that directly map the recognition algorithm into hardware.

2

Algorithm

It is generally accepted that Hidden Markov Models (HMM) are currently the most accurate technique for modelling speech for use in automatic speech recognition [Rab86],[Lee88],[Schw87]. For large vocabulary speaker independent connected speech recognition systems it has better recognition accuracies than the preciously popular dynamic time warp algorithm (DTW) [Kav86,Stö87]. The DTW algorithm was implemented by a number of researchers in custom hardware, and the basic search mechanism is similar to the HMM approach. So, it is worthwhile to compare the two methods [Kav86, Stö87].

For both techniques, the task is to recognize a sequence of speech utterances by comparing it to a model that describes speech segments such as phones or words. To do that efficiently, the incoming speech is first processed to reduce its data rate. This will be called *front end processing* in which speech is segmented into *frames*, which are time intervals of typically 10-20 msec, and the characteristics of frame i is described with a vector of features, $o_i = \{o_i^1 .. o_i^n\}$. One feature vector, o_i^j, could be the energy of the speech signal in different frequency bands, or if this vector is vector quantized [Gra84], the address of the codebook vector that has the best match to these energies.

The difference between DTW and HMM based speech recognition is the way in which a speech segment is modelled. In DTW based systems, the speech representation

is a *template*, which is the sequence of trained features that describe a certain word. Thus, to find the unknown word or phrase, it is necessary to find the template that is most similar to the unknown word. This is done by computing a distance between the features of the template and the unknown word: the smaller the distance between these two representations, the more similar the words. To take into account a possible time distortion between the two representations, the time axis are warped to minimize the distance using the recursive dynamic time warp algorithm.

In HMM based systems, the speech model is statistical and based on a Hidden Markov model. The assumption is that the unknown speech was produced by an HMM speech process that produces speech features based on state transitions. The task in speech recognition is to recover the state transitions that most likely produced the input speech features. When the most probable state sequence is determined, it is straightforward to reconstruct the sequence of words that was spoken. A very efficient search algorithm for the most likely state sequence is the Viterbi algorithm.

2.1. Front End Processing

Front end processing takes the speech waveform that has to be recognized and converts it into sets of features. The set of features at time i is an observation, o_i, and a whole sentence can be described by a sequence of observations, $O_N = o_1 \,..\, o_N$. This observation sequence is then used to find the most probable state sequence in the HMM.

In general, there are two major classes of front end processing algorithms, parametric and non-parametric. Parametric algorithms extract speech parameters such as LPC[1] coefficients. Thus, the observations are the parameters of the autoregressive

1. Here, the generation of the speech waveform is modeled with an all-pole filter (autoregressive model) that has two inputs: a stream of pulses to generate voiced speech, and white noise to generate non-voiced speech. Thus, to describe a segment of speech it is only necessary to specify the filter coefficients and the input. These parameters are estimated using linear predictive coding (LPC, [Rab78]).

speech model itself or some features that were derived from these parameters (for example, LPC-based cepstral coefficients). Non-parametric algorithms, on the other hand, analyze speech data to directly measure certain features such as zero crossings or the energy of the signal in certain frequency bands.

Both classes are widely used in speech recognition, with active proponents of both approaches [Lee89, Murv89]. This system uses a general purpose signal processor for front end processing, so either method (or both) can be implemented. We chose to implement a non-parametric feature extraction algorithm that is used in DECIPHER [Murv89]. A block diagram of this algorithm is shown in Figure 2.

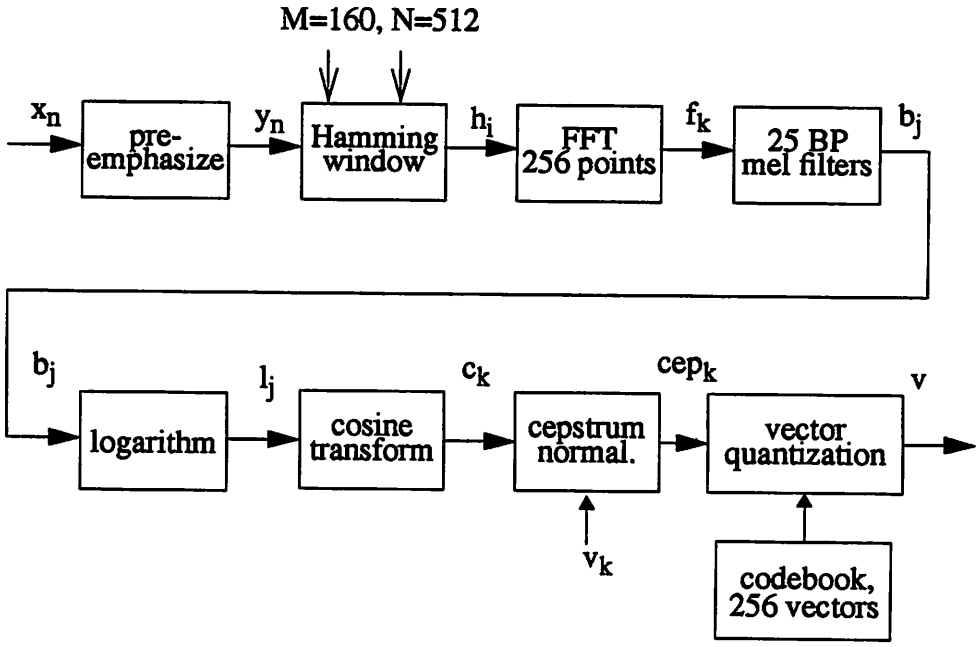


Figure 2: Block Diagram of the Front End Algorithm

The incoming speech, x_n, is sampled at 16KHz and linearly quantized to 16 bits. It then is pre-emphasized using the following equation

$$y_n = x_n - A \cdot x_{n-1}$$
$$A = 0.95 \qquad \text{(EQ 1)}$$

The pre-emphasized speech is then blocked into frames of N=512 samples (32 msec) which are spaced M=160 samples (10ms) apart. Thus, consecutive frames overlap by 352 samples (22ms). Each frame is smoothed by a Hamming window, (EQ 2) shows the corresponding equation for the *k*th frame:

$$h_i = m_i \cdot y_{kM-i}$$
$$m_i = 0.54 - 0.46\cos\left(\frac{2\pi i}{N-1}\right) \qquad \text{(EQ 2)}$$
$$i = 0 \ldots N-1$$

This resulting windowed frame is then used to compute a 256 point Fast Fourier Transform (FFT). The FFT spectrum is then integrated into L=25 mel-bandpass filters[2]. These filters are shown in Figure 3, and a table of the band edges is shown in Table 1.

1 FFT point = 31.25 Hz	lower stopband edge, in FFT points	lower passband edge, in FFT points	upper passband edge, in FFT points	upper stopband edge, in FFT points
filter 0	0	0	6	11
filter 1	3	6	10	14
filter 2	6	10	13	17
filter 3	10	13	16	20
filter 4	13	16	19	23
filter 5	16	19	22	27
filter 6	19	22	26	30
filter 7	22	26	29	33
filter 8	26	29	32	36
filter 9	29	32	35	41

Table 1: Band Edges of the Mel BP Filter

2. The mel frequency scale is a physiological scale that takes into account the performance of the human auditory system to differentiate between frequencies. The differences in frequencies below 1 KHz can be very well distinguished, while frequencies above 1KHz are more difficult. This behavior roughly translates into a frequency scale which is linear to the physical frequency below 1 KHz, but the logarithm of the frequency above. The mel bandpass filters are equally spaced in the mel frequency scale, and they are used to copy human performance in the perception of frequencies in the hope that information relevant for speech recognition is extracted from the speech signal.

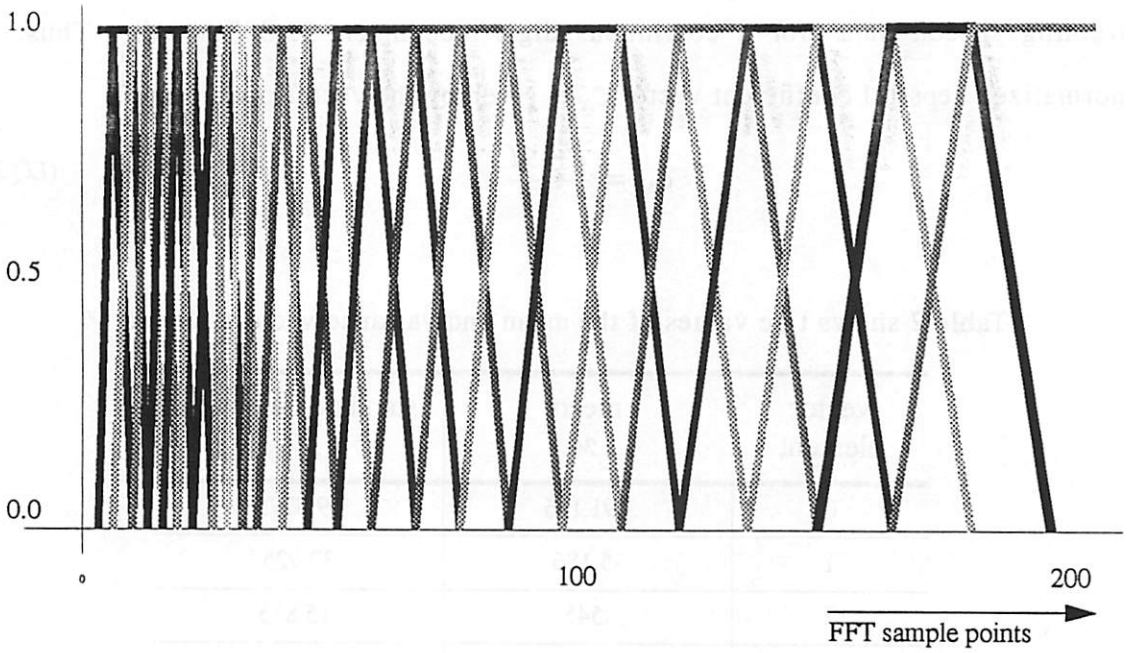


Figure 3: Mel Bandpass Filters

1 FFT point = 31.25 Hz	lower stopband edge, in FFT points	lower passband edge, in FFT points	upper passband edge, in FFT points	upper stopband edge, in FFT points
filter 10	32	35	40	46
filter 11	35	40	45	52
filter 12	40	45	51	59
filter 13	45	51	58	68
filter 14	51	58	67	77
filter 15	58	67	76	87
filter 16	67	76	86	98
filter 17	76	86	97	110
filter 18	86	97	109	122
filter 19	97	109	121	136
filter 20	109	121	135	150
filter 21	121	135	149	165
filter 22	135	149	164	181
filter 23	149	164	180	197
filter 24	164	180	255	255

Table 1: Band Edges of the Mel BP Filter

After that, the energy of each bandpass filter gets converted to the logarithm of the energy, l_j, $j=0..L-1$. These energy values are then used to get the cepstral coefficient[3] vector $C = [c_0, .. ,c_{12}]$ using the cosine transform, which is described in the following equation (EQ 3):

The cepstral coefficient vector C is then normalized with respect to its mean vector $\mathcal{M}$ and variance vector $\mathcal{V}$. $\mathcal{M}$ and $\mathcal{V}$ were computed off-line using a large set of training speech data (for a continuous digit recognizer, 52,000 words). Thus, the normalized cepstral coefficient vector C_N is given by the vector operation

$$C_N = \frac{C - \mathcal{M}}{\mathcal{V}} \qquad \text{(EQ 3)}$$

Table 2 shows t he values of the mean and variance vectors, $\mathcal{M}$ and $\mathcal{V}$.

vector element	mean $\mathcal{M}$	standard deviation $\mathcal{V}$
0	291.116	89.5275
1	-5.186	32.925
2	.545	15.855
3	11.278	14.443
4	-1.881	10.535
5	-.388	7.206
6	-2.318	7.237
7	.724	5.851
8	-.465	5.519
9	-.238	4.699
10	0.252	3.847
11	-1.063	3.371
12	.028	2.944

Table 2: Mean and Standard Deviation for the Cepstrum (TI Digit Database)

Finally, the cepstral vector $C_N' = [c_{1N}, .. , c_{12N}]$ is vector quantized using a codebook with size 256·12 (note that C_N' does not contain c_{0N}, this element is used later). The result of the vector quantization step is the address o of the codebook vector

3. Thus, the multiplication of the speech components in the frequency domain (the exaltation by the vocal cords and the vocal tract filter) is replaced by and addition, and the cosine transform yields a time domain signal that corresponds to the sum of the two contributions.

with the closest match to the cepstral coefficient vector. It is an 8 bit number that is computed every 10 msec. The initial codebook was generated using the Lloyd algorithm on a large amount of speech data (about 9,000 sentences). [Gra84]

The resulting address o is o_i^1, the vector quantized cepstral feature for frame i. Another feature, o_i^2, is the vector quantized difference of cepstral vectors. This differenced cepstral coefficient is computed by

$$D(t) = C'_N(t+\delta) - C'_N(t-\delta) \qquad \text{(EQ 4)}$$

Here, $C_N'(t)$ is the cepstral vector of the t^{th} frame of the incoming speech, and δ was chosen to be 2, so the difference between 4 frames (40msec) is computed [Lee89]. Further features are the quantized energy of the signal (o_i^3), which is represented by the first element of the cepstral vector (c_o), and the quantized differenced energy (o_i^4), which is computed in analogous way to (EQ 3). The codebook that is used to quantize o_i^3 and o_i^4 has 256 scalars.

2.2. Hidden Markov Model

In hidden Markov model based speech recognition, we assume that speech was produced by a Hidden Markov (HMM) process. A HMM is described with a finite set of states and a set of transitions between these states. Figure 4 shows the graph of a typical HMM that could model a vocabulary word.

The graph in Figure 4 is called *left to right*, since the predecessors of a state are always to the left. If the graph has the property, that every state can be reached from every other state in a finite number of steps, it is called *erogdic* [Rab86]. Typically,

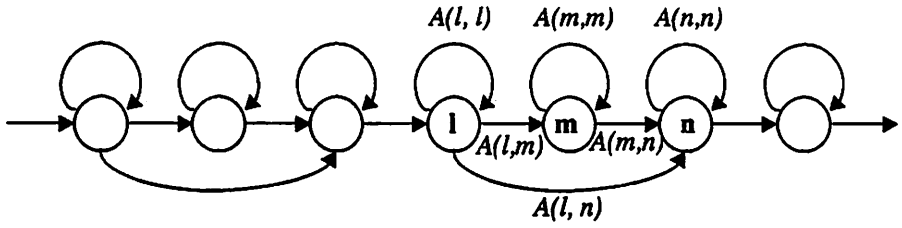


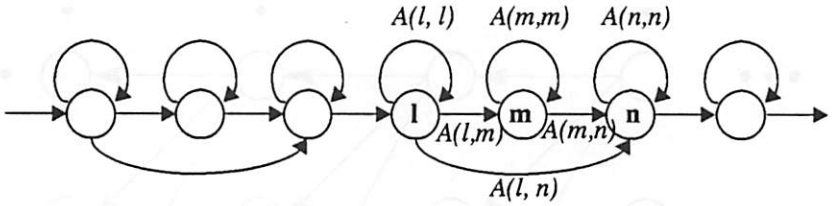
Figure 4: Graph of a HMM

words are modelled with left to right HMMs, while grammar is modeled with ergodic HMMs.

A node represents a state in the HMM, and such a state corresponds to a generic speech sound (e.g. a part of a phoneme). In the same way as speech progresses from one sound to another, the HMM can make transitions between states using the arcs in Figure 4. These transitions have associated transition probabilities, $A(s,t)$ for the transition from state s to state t. Thus, the connectivity of the arcs (topology) defines the allowable state sequence and determines the predecessors (states that have a transition to the state) and the successors (states to which this state can transition to) of every state.

Speech can be considered as being generated during transitions between these states yielding a state sequence, $S_N = s_1 .. s_N$, where the likelihood of these transition is described with the transition probabilities, $A(s,t)$. After entering a state s at time i, the hidden Markov process emits a small segment of speech (observation), o_i, with probability $P(o_i|s)$. Thus, every state s in the HMM has an associated probability distribution $P(o_i|s)$ that gives the probability for all possible speech sounds o given the HMM process is in state s. This distribution can be discrete or continuous, depending on the random variable o_i. In this system, o_i has only a finite set of 256 symbols, thus $P(o_i|s)$ is discrete.

Figure 5 shows the HMM speech production process. At any given time (frame) the hidden Markov speech process occupies a certain state. Assuming the process is in state l at frame i-1, it will generate the observation $o = o_{i-1}$ with probability $P(o_{i-1}|l)$ and then progress to the next state based on the transition probabilities $A(l,l)$, $A(l,m)$ and $A(l,n)$. Assuming state m is the next state, the process will output feature o_i with probability $P(o_i|m)$ and so on. The model is *hidden* since we can only observe the output sequence $O_N = o_1 \,..\, o_N$ (speech features), and the underlying state sequence $S_N = s_1 \,..\, s_N$ of the process is not observable.



frame number	i-1	i	i+1	i+2
state sequence $\{s_i\}$	s_{i-1}= **l**	s_i = **m**	s_{i+1}= **n**	s_{i+2}= **n**
output sequence $\{o_i\}$	o_{i-1}	o_i	o_{i+1}	o_{i+2}
speech				

Figure 5: Hidden Markov Model for Speech Production

2.3. Recognition

The task to be performed during speech recognition is to find, for a given HMM, the state sequence that most likely produced the speech being recognized. When the most probable state sequence has been determined, it is straightforward to reconstruct the sequence of words that was spoken. A very efficient search algorithm

for the most likely state sequence is the Viterbi algorithm. This algorithm computes the probability of the most probable state sequence $S_N = \{s_1 .. s_N\}$ given a set of observations $O_N = \{o_1 .. o_N\}$.

Figure 6 shows a trellis structure that visualizes the state transitions over time: states in the same column correspond a single frame. The rows in the trellis correspond to a certain state during a sequence of frames. Transitions between the states are indicated with arrows. Thus, horizontal arrows correspond to "self loops" which are transitions into the same state.

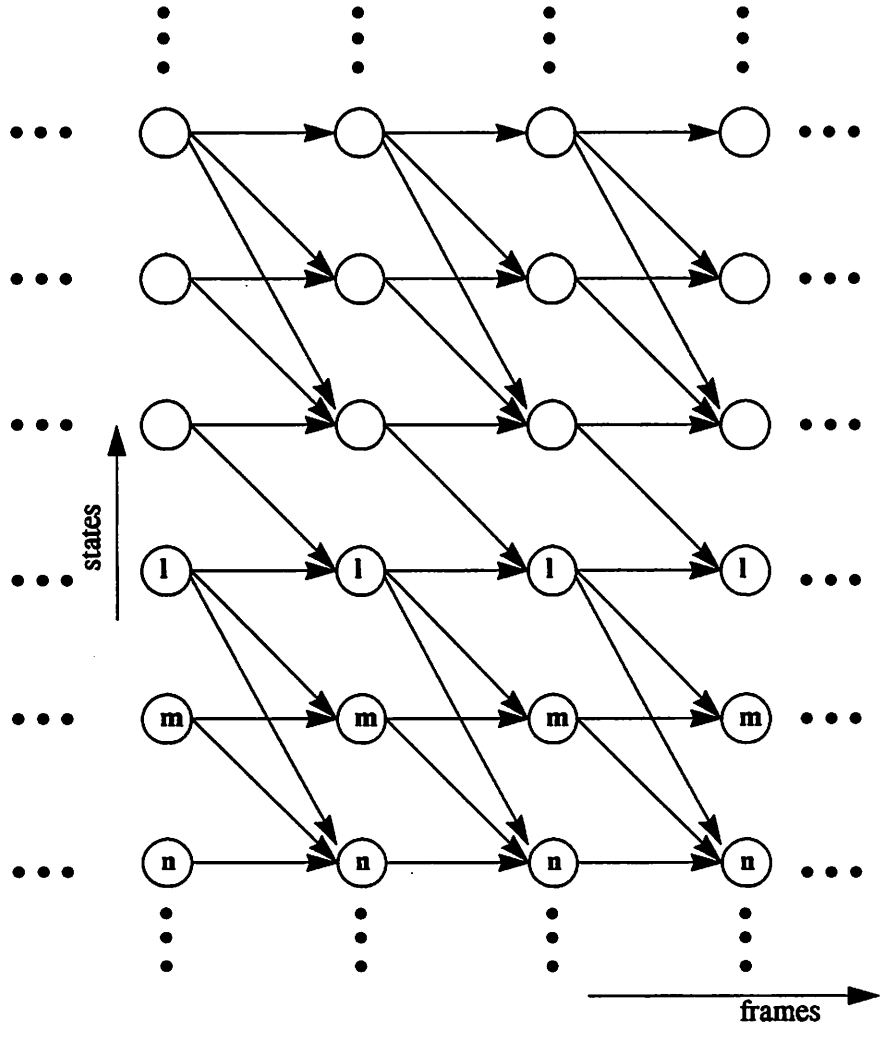


Figure 6: Trellis Structure for the Model in Figure 4

2. 3. 1. State probabilities, Viterbi algorithm

The joint probability of a sequence of N states, $S_N = \{s_1 .. s_N\}$, and a sequence of N speech features (observations), $O_N = \{o_1 .. o_N\}$, is the *path probability* and is given by

$$P(S_N, O_N) = \pi(s_i) \cdot P(o_1|s_1) \cdot \prod_{i=2}^{N} [A(s_{i-1}, s_i) \cdot P(o_i|s_i)] \qquad \text{(EQ 5)}$$

In this equation, $\pi(s)$ is a probability distribution that gives the probabilities of the states in the first frame (first column). The goal is to find the most likely state sequence for a given O_N. Therefore, if the path probabilities of all the possible state sequences S_N are known, the sequence with the highest path probability represents the most likely state sequence. However, the number of possible state sequences is $(mN)^{T-1}$, where N is the number of states n the model, m is the average number of transitions into a state, and T is the number of frames in a sentence (there are mN state transitions at every frame). For a 60,000 word speech recognition system, we assume m=3, N=200,000, and T=200 (see section 4. 2. 2., two seconds of speech), so there are $(6 \cdot 10^5)^{199}$ possible state sequences, and it is not practical to compute all of them. If we consider all the possible state sequences that lead to a state s at frame i, we are only interested in the sequence that yields the highest path probability. The paths that are less likely will not be part of the overall highest path probability. Thus, all we really need to know is the highest probable path into each state at each time, and the probability of this path is termed the *state probability*, $P(O_i, s)$ for state s in frame i. If this probability is known for every state in the HMM for the last frame N, we know that the most likely state sequence terminates at the state that has the highest state probability $P(O_N, s)$.

Thus, the state probability $P(O_i, s)$ is the probability of the most likely state sequence that ends in s at frame i and generates O_i, a sequence of i features. The

approach to compute the state probabilities $P(O_N, s)$ is to use a dynamic programming scheme called the Viterbi algorithm [Rab86]:

$$P(O_1,s) = \pi(s) \cdot P(o_1|s) \qquad \text{(EQ 6)}$$

$$P(O_i, s) = MAX_{p \in \{pred\}} [P(O_{i-1}, p) \cdot A(p,s)] \cdot P(o_i|s)$$

The maximum operation is performed over all the states, p, in the set *{pred}* which contains the predecessors of state s, as defined by the topology of the HMM. Given these equations, $P(O_i,s)$ can be computed for all states for O_1, then for all states for O_2 and so on until all probabilities $P(O_N,s)$ for all states are computed. The state with the highest probability is the final state of the most likely state sequence.

2. 3. 2. Backtracking

As discussed above, the state with the highest state probability in the last frame is the endpoint of the most probable state sequence. After an unknown sentence ended which is indicated by the energy of the signal (pause), this state sequence terminating at the state with the highest state probability has to be recovered by *backtracking*.

This could be done if there is a list that, for every state and for every frame, contains a pointer to the most probable preceding state. However, during the generation of this list the most probable state sequence is not yet known, therefore all state sequences have to be stored. Since every state in the HMM is the endpoint of a single state sequence (the most likely state sequence), there are as many state sequences as there are states in the HMM.

To identify these state sequences, each state in each frame has a tag that identifies the predecessor state on the most likely path. To generate the tag, we use the following equation:

$$TAG(s,i) = argmax_p [TAG(p,i-1)] \qquad \text{(EQ 7)}$$

TAG(s,i) is the tag associated with state s at frame i. It is the copy of the tag associated with the predecessor state p of s, *TAG(p,i-1)*, and p is the state that is on the most likely path to s.

Using these tags, we can recover the most likely state sequence at the end of a sentence.

2. 3. 3. Multiple output probabilities

The output probability is the probability that a certain speech segment is generated given the speech process is in a certain state. This speech segment can be described using several different speech features, o_i^1 .. o_i^n. In other words, every state has n probability distributions that give the probabilities that the state can output n different features of a particular speech segment. Assuming statistical independence, the joint output probability is given by (EQ 8) :

$$P(o_i|s) = P(o_i^1|s) \cdot P(o_i^2|s) \ldots \cdot P(o_i^n|s) \qquad \text{(EQ 8)}$$

3

Previous Work

Speech recognition has come to a stage where the recognition accuracies -or at least the recognition of the meaning of a phrase- is high enough to justify its feasibility in real, task specific applications (1.3.). For that, it is important that the recognition system operates in real time. If a user can type faster than the recognition system can recognize its speech, some advantages of speech recognition are not utilized. It would be relatively easy to achieve real time performance if the goals for accuracy would be relaxed. For example, a method called beam search (see chapter 4) gives a strategy to skip computation. To obtain real time performance for large vocabulary, we could simply skip a large amount of computation, but at the expense of reduced recognition accuracy.

This chapter describes previous work on real time connected speech recognition systems based on HMMs. They can be classified into two groups, depending on whether they use general purpose or full custom processors. It is demonstrated that existing general purpose hardware or multiprocessor systems are not powerful enough to implement a 60,000 word recognition system in real time. Therefore, it is necessary to use full custom hardware to efficiently implement a real time large vocabulary speech recognition system that is flexible enough to support common HMMs for speech recognition.

3.1. Hardware Based on General Purpose Processors

This section describes speech recognition systems that use general purpose processors to perform the recognition algorithms. The attraction is the belief that systems with general purpose components can be designed quickly and cheaply, and that they are programmable using high level languages (e.g., C), thus making the system versatile so it can be tailored to different recognition systems [Bis89].

3. 1. 1. The BEAM Hardware Accelerator

BEAM was developed at the Carnegie Mellon University (CMU) to run the SPHINX speaker-independent, continuous speech recognition system in real time [Bis89, Lee89]. It is a VME-based multiprocessor system that uses 3 general purpose Weitek XL-8032 processors, each capable of 10 million instructions per second. The architecture of the system is shown in Figure 7.

The processors have two local memories each, one with 256Kbytes for data, and the other with 32Kbytes for program storage (corresponds to 8,000 instructions). The three processors share a dual-bank global memory. The local memories can be accessed with a cycle time of 100 nsec, while the global memories have an access time of 200nsec (if there is no memory contention). To support processor synchronization, each word in the shared memory is augmented with a bit that is used for special read and write operations: If the bit is set, write operations are forbidden, if it is zero, read operations are forbidden. This way, producer-consumer synchronization can be implemented with a minimum of overhead [Bis89].

BEAM was successfully used for speech recognition using the SPHINX recognition system in the 1,000 word DARPA resource management task [Pri88]. BEAM is capable of updating 4,000 states within a frame duration of 10msec, or computing 8,000 transitions per frame (in SPHINX, there are, in average, 2

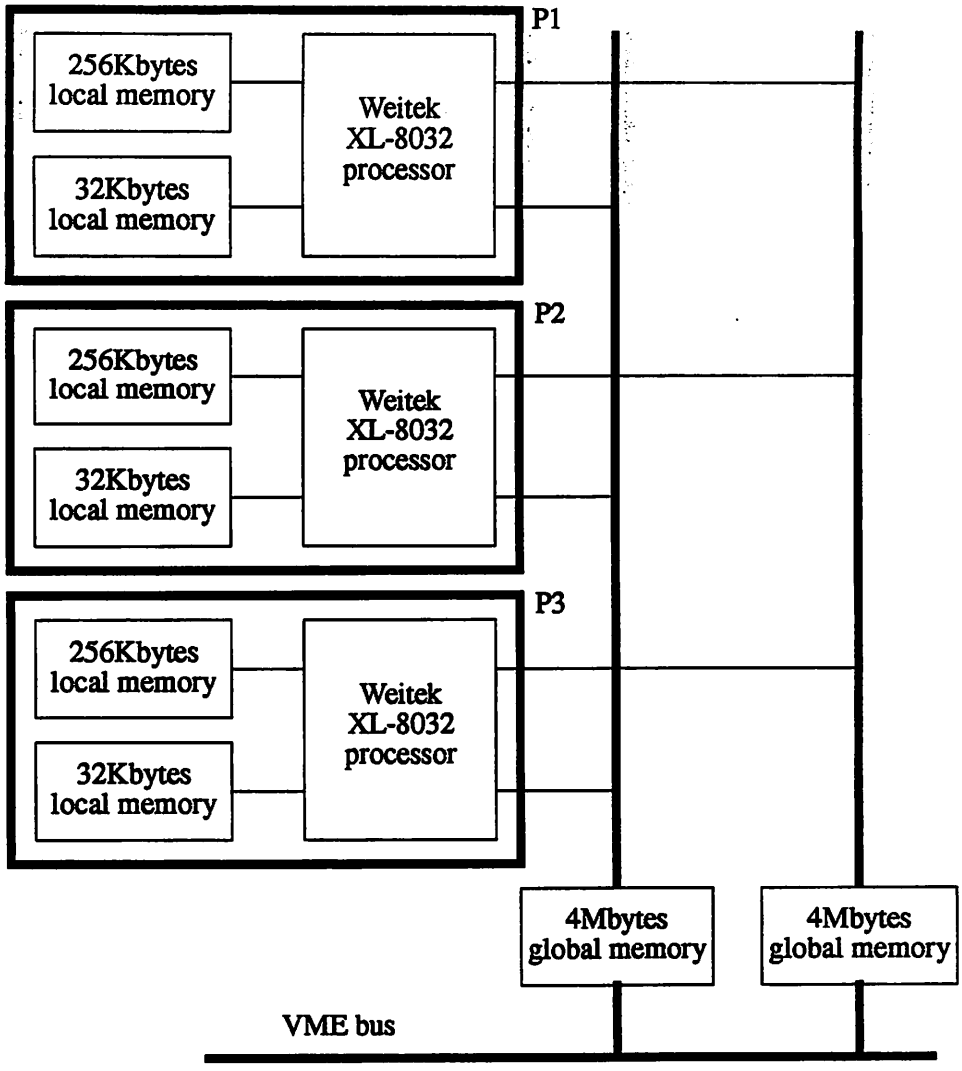


Figure 7: The Architecture of BEAM

predecessors for a state). To perform real time recognition for the resource management task, it would be necessary to update 40,000 states within a frame [Bis89], so the system uses a pruning algorithm (see 4. 2. 2.) that discards 9/10th of the states and only updates the most likely 10%. For a grammar with perplexity[1] 20, the performance of BEAM was in the average 1.15 times real time (recognition time divided by the length

1. The perplexity Q is an information theoretic measure of a tasks difficulty. It is defined as $Q=2^H$, where H is the entropy, or the number of bits necessary to specify the next word using an optimal encoding scheme [Lee89].

of the incoming sentence), and for a perplexity 60 grammar, the real time factor averaged 1.38.

3. 1. 2. The AT&T BT-100 ASPEN Parallel Computer

The AT&T BT-100 (ASPEN) is a tree-structured parallel processor which is targeted for pattern recognition applications such as speech recognition. The system is partitioned into several boards, 8" x 11" each. One board contains 8 processing elements, and a processing element consists of an AT&T DSP32 floating point digital signal processor (8 million floating point operations per second, MFLOPs), 64KBytes of local memory, and a communication chip [Roe89]. The basic architecture of this system is shown in Figure 8. The referenced paper [Roe89] investigates a system that consists of 16 boards (127 PE's), and it is capable of 1000 MFLOPs.

To perform speech recognition on this hardware, the features that were observed in a frame have to be broadcast to all PE's. Then, the state probabilities of all states in the system are concurrently computed on the PEs, and with a method called *psr* (pipelined sort and report), the K best grammar nodes (see section 4. 3. 1.) are communicated to the host PC. These K best phone instances are then broadcast to the PEs, which compute (concurrently) the initial scores for all subsequent phone instances using the scores of the K best grammar nodes.

The referenced paper [Roe89] estimates the time required for the 1,000 word vocabulary resource management task [Pri88]. In this estimate, the HMM uses continuous output probability distributions (tied mixtures, [Bah90]), in which the PEs compute the output probability (the probability of an observation given the speech process occupies a certain state, see 2.2.) using 3 gaussian distributions.

For this estimation, it is assumed that the PEs compute, sort and transmit the K=20 best grammar nodes to the host PC, which collects and broadcasts these nodes back to the PEs. The authors estimate, that the time spent for communication is just

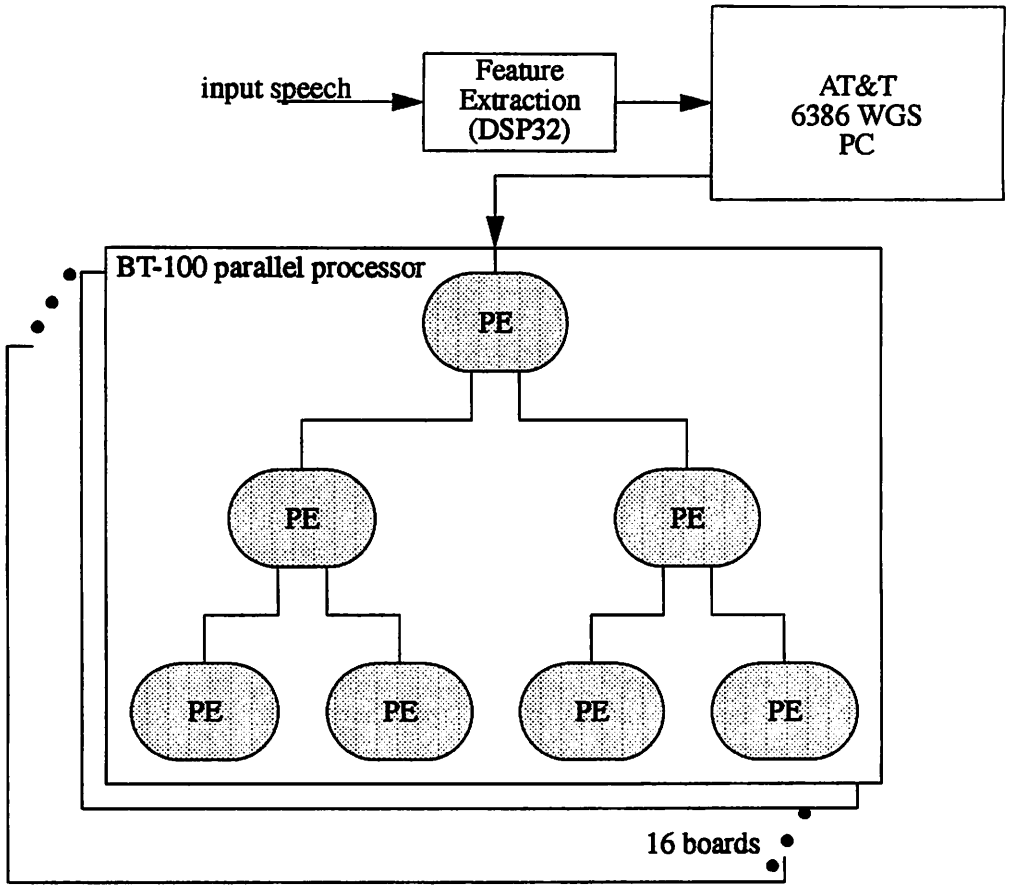


Figure 8: Architecture of the AT&T ASPEN Processor

under 5 ms per frame, and the time spent for computation is 12 msec. Thus, it takes about 17ms to perform the recognition algorithm for an incoming speech segment of 10ms, therefore the real time factor is about 1.7.

3. 1. 3. The MARS Multiprocessor Pipeline

The MARS hardware is organized using 2 levels of hierarchy. On the lowest level, there is a 16-bit microprogrammable processing element with special features for message passing, list and table manipulation, and bit field operations. Each processing element has it's own local memory, and communication between PE's is performed entirely by message passing [Chat89]. There is no shared memory. 15 PE's comprise a

cluster, and the PE's inside a cluster are connected through a crossbar switch that can be re-configured each clock cycle. Thus, with message passing between PEs using the crossbar switch, it is possible to form a re-configurable pipeline. The MARS system consists of 256 clusters, so it has a total of 3840 PEs.

The referenced paper [Chat89] deals with a simulation model for the MARS hardware. Using this simulation model, the SPHINX speech recognition system [Lee89] was coded. For that, the algorithm was partitioned on a functional basis, and the functional modules were implemented on the PEs. The local memories of each PE contain the data structures required by that module of the algorithm.

Based on the simulation results, it is projected that MARS can process 400,000 states per second, or 4,000 states per frame. The recognition algorithm is time-synchronous, which means, all processing for a frame is completed before processing for the next frame can be started. Thus, the MARS pipeline has to be drained after each frame, and this takes 25μsec.

Thus, MARS is capable of processing 4,000 states, so it should perform on the SPHINX recognition task as well as the BEAM hardware accelerator.

3. 1. 4. The Dragon Real Time Continuous Speech Recognition System

The commercial Dragon speech recognition system [Bam90] uses the *"rapid match"* method to reduce the amount of computation. As a result, real time connected speech recognition for a 842 word vocabulary can be performed using a 486-based personal computer (15MIPS) that has a 17MIPS add-on board (AMD27000).

The rapid match algorithm generates a short list of words (typically 100-200 words) that might begin at a particular time, based on the acoustic speech data beginning at that time [Gil90]. To achieve this, the words in the vocabulary are grouped into words whose beginnings are acoustically similar (word start groups, WSGs). Each

WSG has a single model that consists of an acoustic representation of the word start typical to that group. This representation is generated using the first 240 msec of speech of all the words in the WSG.

During recognition, the rapid matcher computes the same acoustic representation of a 240 ms segment of the incoming speech, and computes a score (WSG score) that gives the probability that the smoothed representation of the unknown word matches the smoothed representation of the various WSGs. The rapid matcher then looks up the words in the highest probable WSGs, removes duplicates and scores the remaining words using the WSG score and a language model score. The top scoring words of this list are then passed to the recognition system which executes the traditional recognition algorithm (Viterbi algorithm).

In the real time Dragon system, the average number of words that are passed to the Viterbi recognition algorithm is 40. In the 843 word system, this results in a system speed by a factor of 5 to 10 [Gil90]. The hardware used for this recognition system is a 486-based personal computer (15MIPS) with an add-on board (AT super) that has an AMD 29000 processor (17MIPS). During recognition, these processors operate in parallel: The AMD29000 processor performs the rapid match algorithm and the 486 microprocessor the Viterbi algorithm. With this setup, recognition for the 842 word system is effectively performed in real time (1.1 times real time). The recognition accuracy is 96.6% (speaker dependent) using a grammar with perplexity 66[2].

The speedup is of a factor of 5 to 10 over a system without rapid match. Though impressive, this method introduces errors: in 4% of the frames, the rapid match algorithm does not pass the correct word to the Viterbi algorithm [Gil90]. However, some level of recovery is possible since the correct word might be passed in a nearby

2. The perplexity Q is an information theoretic measure of a tasks difficulty. It is defined as $Q=2^H$, where H is the entropy, or the number of bits necessary to specify the next word using an optimal encoding scheme [Lee89].

frame. Another drawback is, that this method introduces a new set of problems for speaker adaption in a speaker independent system, and training of the WSGs.

3.2. General Purpose Speech Recognition Hardware.

In this section, we investigate two full custom processors that are used for HMM-based connected speech recognition. Using full custom processors, it is possible to customize the datapaths and the instruction set for speech recognition algorithms, and as a result, small systems that use these special purpose chips can outperform the large general purpose systems described in the previous section.

3. 2. 1. The AT&T Graph Search Machine

The Graph Search Machine (GSM) is a VLSI processor that was designed for graph search operations such as dynamic programming for DTW and HMM based speech recognition systems, and metric computations for vector quantization and distance measurement. The processor was designed using a 1.77μm CMOS process. It has 34,500 transistors, comes in a 68-pin package, and operates on a 8MHz clock. A typical system architecture that uses multiple GSMs is shown in Figure 9:

A general purpose microprocessor is the host, and it is used to load the system at start-up, and to run the support software tools. The individual GSMs are connected to the microprocessor bus with an 8 bit data bus, and each GSM has a local memory with an address space of 64K for program and data storage.

The GSM has a 6 bit datapath with a minimum unit, an adder, and an address generation unit. It also has an on-chip program cache memory that can hold 32 instruction words, and a 32-word by 32-bit ROM for utility programs. The datapath and the instruction set support the performance of the Viterbi algorithm, and the

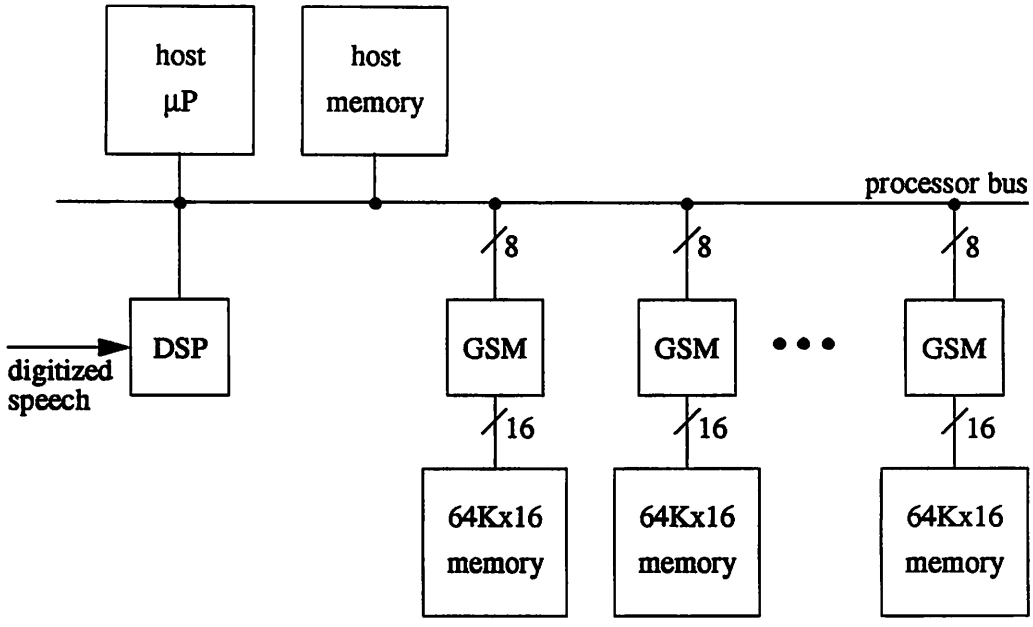


Figure 9: System Architecture for the GSM

computation of backtrack pointers that are necessary to recover the most probable sentence after recognition.

The referenced paper [GLI87] states, that for graph search tasks, this processor can outperform typical microprocessors (of that time) by a factor of 10, and typical signal processors by a factor of 5. This processor, however, is not suited for real time large vocabulary connected speech recognition. This is mainly because of two reasons: memory bandwidth and the small size of the local memory (64Kx16).

To obtain all the information necessary to update one state, it is necessary to access memory 17 times (with the assumptions described in 4.1.). To perform real time speech recognition, it is necessary to update 200,000 states within 10msec (4. 2. 2.). Neglecting time that might be used to access instructions from the local memories, or excess computation that does not overlap with memory operations, it takes 17 · 125nsec = 2.125µsec to process a state. Thus, a GSM could update 4,700 state in real time, and

the combined bandwidth of about 40 GSMs would be required for a 60,000 word real time speech recognition system.

The more severe restriction of the GSM is the small local memory. In 4. 3. 1., it was sown that a memory with an address space of 82 million words is required to store the parameters of a hierarchical HMM that uses 4 discrete output distributions per state, and has 64,000 unique states (no grammar). Neglecting the fact that some of these structures require more than the 16 bits offered by the GSM's local memory, and assuming that the HMM could be perfectly partitioned between the local memories of GSMs (which is not even possible), it would take $(82 \cdot 10^6) / (64 \cdot 10^3) \cong 1{,}300$ local memories of GSM memories to store the parameters. Also, it is not possible to dynamically swap the parameters into the local memories as they are needed during recognition: the 8-bit interface to the shared bus of the microprocessor is not nearly fast enough.

As a result, using the GSM for a 60,000 word vocabulary connected speech recognition would require an unreasonable number of GSM chips.

3.3. Comparison to other Dynamic Programming Applications

This section relates the Viterbi algorithm for speech recognition to Viterbi decoding used in communications and to the Dynamic Time Warp algorithm. Various special purpose VLSI processors for these applications have been designed and will be reviewed as to their suitability for this task.

3. 3. 1. Viterbi in Communication Systems

Viterbi decoding is used for error correction in communication systems to improve bit error rate performance. Here, the signal is encoded using convolutional

encoding, and Viterbi decoding is used to make a maximum likelihood decision to recover the signal.

The Markov process that models the generation of a sequence of data in a communication system [Lin89] usually has not more than 64 states. There are typically two or four transitions per state. This means, a processor that performs the Viterbi algorithm for this class of applications can store on chip all the necessary data that describe the HMM. However, the bottleneck in this application is throughput. The "frame" durations (called stages) are in the range of 100ns down to 5ns, therefore there is typically one processing element provided for each state in the HMM. This is clearly an inadequate architecture, as in speech recognition 200,000 states have to be processed within 10ms. Another difference is that instead of using pre-compiled, fixed probabilities to describe transitions between the states, a distance is calculated between the received signal and the signal represented by the transition. Therefore the hardware performing the Viterbi algorithm has an arithmetic and an architecture that can not be used for large vocabulary speech recognition.

3. 3. 2. Dynamic Time Warp Algorithm

The computations for the Viterbi algorithm are very similar to computations for the Dynamic Time Warp (DTW) algorithm, another forward programming scheme that is successfully used for isolated speech recognition. Several special purpose DTW processors for real time speech recognition have been designed, for example [Kav86] and [Stö87]b but as this section demonstrates, they cannot be used for HMM based large vocabulary speech recognition.

In DTW systems, a word in the recognition vocabulary is called the template and its representation is a time ordered sequence of speech feature vectors. Typically, a feature vector is a spectral representation of the speech waveform. The underlying computation in the recognition algorithm is a word-to-word distance measure which is

computed between all template words and the unknown word. This distance between two words is the sum of the squared Euclidean distances between the feature vectors and it is used as an error measure: the smaller the distance, the more similar the words.

Since a person might say a particular word faster or slower than the corresponding word in the template, the word that has to be recognized might have a different number of feature vectors than the corresponding template word. Therefore, the time axis must be stretched (warped) to align the two representations and to minimize the word-to-word distance. This is done using the Dynamic Time Warp algorithm: it minimizes the word-to-word distance over all possible alignments of the feature vectors using dynamic programming.

Figure 10 shows how two utterances of the word /six/ have to be aligned to minimize the word-to-word distance [Kav86]. The notation /siiiks/ describes the sequence of features corresponding to the sounds of the utterance ("s", "i", "i", "i", "k" and "s").

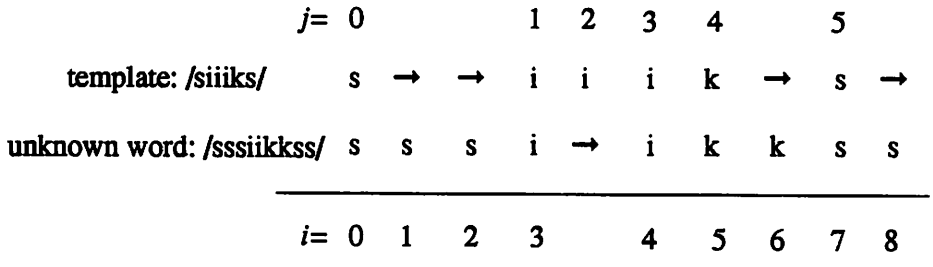


Figure 10: Alignment of Two Utterances of /six/

The DTW algorithm computes the minimum word-to-word distance using the following recursion:

$$d(i,j) = L_2(U_i, T_j)$$

$$D(i,j) = min\,[D(i-1,j), D(i-1,j-1), D(i,j-1)] + d(i,j) \quad \text{(EQ 9)}$$

In these equations, *i* refers to the sequential count of feature vectors of the unknown utterance (see Figure 10) which is equal to the count of frames, *i=0..I-1*. The index j refers to the features of a template word, *j=0..J-1*. The value *d(i,j)* is the Euclidean distance (L_2 norm) between U_i and T_j, the *i*th feature vector of the unknown word and the *j*th feature vector of the template word. The accumulated distance (the error measure) *D(i,j)* is the sum of the Euclidean distances accumulated over a path that yields the smallest sum. Thus, if the unknown word has *I* features (*i=0..I-1*) and the template word has *J* features (*j=0..J-1*), the minimum accumulated distance between the two words is *D(I-1, J-1)*. This word-to-word distance is computed for all template words and the template with the smallest *D(I-1, J-1)* is the recognized word.

To visualize these equations, *d(i,j)* can be displayed as a matrix of Euclidean distance measures between all features of the template and the unknown word. To compute the matrix *D(i,j)*, the distance *d(i,j)* at *i,j* is added to the minimum of the accumulated distances at the upper (*D(i,j-1)*), upper left (*D(i-1,j-1)*) and left (*D(i-1,j)*) vicinity of the matrix location *i,j*. The final result is the minimum accumulated distance at the lower right corner of the matrix.

To find a representation of a template word similar to the HMM, a sequence of spectral features could be modeled as a sequence of states where transitions between states correspond to the allowable sequence of spectral features. To model the nonlinear stretching of the time axes, each state has a self transition, and a *null-arc* from the immediate state at its left. The self transitions are used to delay the speech process: it

can stay longer than one frame in a certain state. On the other side, a null-arc is a transition that does not consume a frame delay, which means, the speech process can occupy several states within one frame. In other words, the speech process can skip through a state using a null-arc transition, and this is the case when the input speech is spoken faster than the corresponding reference template. This model is shown in Figure 10. There are two transitions from left-to right: the solid arrows refer to state transitions that consume a frame delay while the striped arrows refer to the null-arcs.

template: /siiiks/

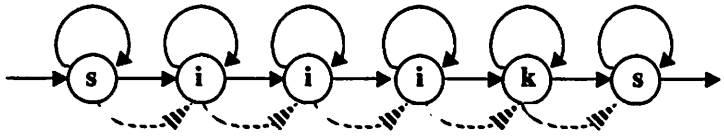


Figure 11: State Transition Model of the Template Word /siiiks/

The computations in (EQ 9) can be visualized in a trellis similar to Figure 12. States in a column correspond to the features of a template word while states in a row correspond to the same feature at different time indexes *i*. Thus, horizontal arrows correspond to self loops, vertical arrows to null-arcs and diagonal arrows to a state transition from the immediate state to the left (Figure 12).

Every *i* corresponds to a feature of the unknown word, and therefore the *i*th column in the trellis visualized the computations for index *i*. The minimum accumulated distance of a certain trellis location *i,j* can be thought of as the minimum accumulated distance associated with state *j* at frame *i*. To compute this distance, we have to find the minimum of the accumulated distances of the predecessor states (states that have a transition to that state) and add *d(i,j)*, the Euclidean distance between the *i*th feature of the unknown word and the *j*th feature of the template word. This computation is done

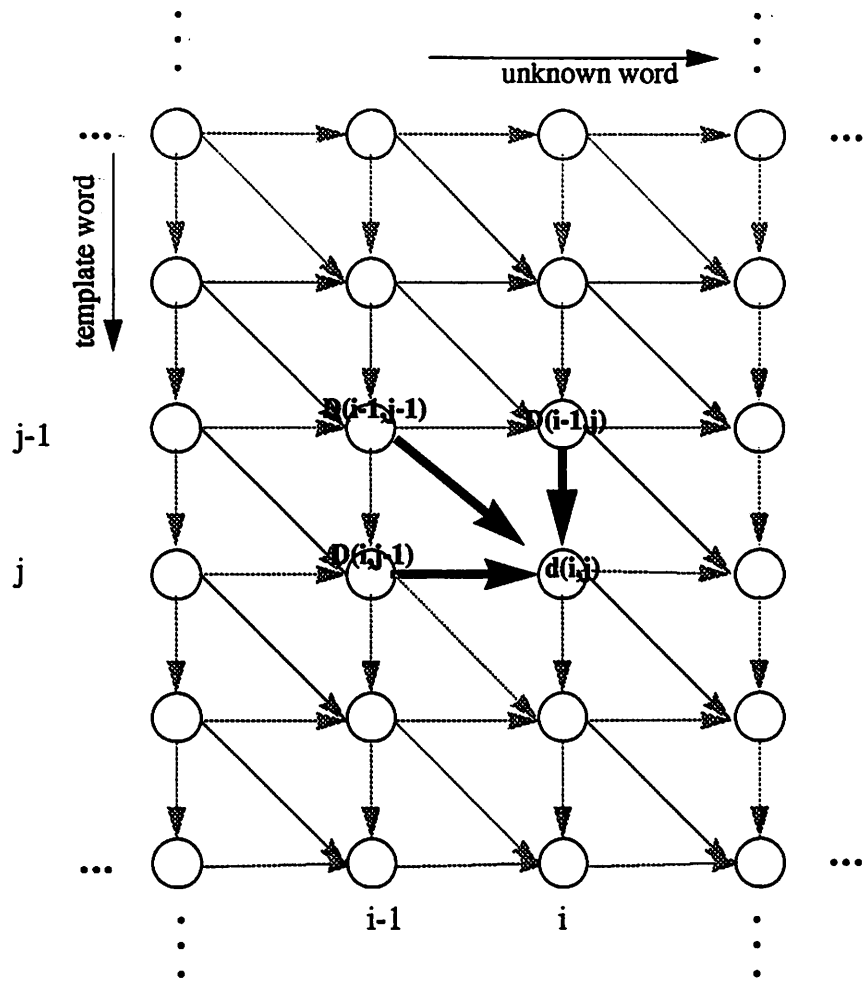


Figure 12: Trellis Structure for the Dynamic Time Warp Algorithm

sequentially for all the states of the template word at frame *i*, then for frame *i+1* and so on (time-synchronous streaming algorithm) until frame *I-1*. The minimum distance is then the accumulated distance associated with the state *J-1* at frame *I-1* (lower left state).

The computations of (EQ 10) are very similar to the computations that have to be performed for the Viterbi algorithm if probabilities are represented using their negative logarithmic values: Multiplications are implemented using additions and the

MAX operation becomes a MIN operation. (EQ 10) shows the equation for the Viterbi algorithm in the logarithmic domain:

$$\mathcal{P}(O_i, s) = MIN_{p \in \{pred\}} [\mathcal{P}(O_{i-1}, p) + \mathcal{A}(p, s)] + \mathcal{P}(o_i | s) \qquad \text{(EQ 10)}$$

The value of a probability is between 0 and 1, therefore the logarithm of a probability is negative. Thus, we can ignore the sign and treat the logarithm of a probability as a positive number. By comparing (EQ 9) and (EQ 10), we notice that the distance measure $d(i,j)$ corresponds to the negative logarithm of the output distribution, $\mathcal{P}(o_i|s)$, while the accumulated distance of a predecessor corresponds to the sum of the state probability of a predecessor state and the transition probability, $\mathcal{P}(O_{i-1},p)+\mathcal{A}(p,s)$.

The difference in the Viterbi algorithm is, that the trellis in the DTW algorithm for speech recognition is regular. That means, every state has predecessor states on the same relative position. In HMMs, the predecessor states can be on different relative positions. Also, in DTW the transition from one state to another has no associated transition probability. Another major difference is that the distance between the features of the unknown word and a template word is computed using the Euclidean distance while the Viterbi algorithm uses probability distributions to relate a state to an observation.

In general, to perform the Viterbi recursion, data have to be accessed that aren't necessary for the DTW recursion, such as transition probabilities, topology information and output probabilities. These differences have a significant impact on the hardware implementation. DTW processors can utilize the regular trellis of the DTW algorithm and map the access of the accumulated distances directly into hardware. Instead of addressing predecessors, this allows the output of registers that keep the accumulated distances to be directly routed to a MIN logic.

In summary, the computation of the DTW algorithm and the Viterbi algorithm are very similar, but a processor that implements the Viterbi algorithm requires a higher external bandwidth than a DTW processor since more data have to be accessed. Also, additional logic is required to address the predecessors and their corresponding transition probabilities.

The French LIMSI-CNRS system is an example of exploiting the similarities of the two recognition algorithms.

3. 3. 2. a The LIMSI-CNRS Dynamic Programming (DP) Processor

This processor was designed for isolated and connected speech recognition using the DTW algorithm. It operates on a 20MHz clock, and can execute 10 million instructions per second. It is capable of real-time recognition of a 1000 isolated words, or 300 connected words. The chip has of 127,300 transistors, and it comes in a 84-pin package [Qué89].

Figure 9 shows a block diagram of speech recognition system that uses several of these processors.

The system architecture is almost identical the system architecture for the GSM, the major difference is, that this processor can accommodate a larger local memory.

The processor architecture of the DP processor has 3 datapaths: a distance calculation unit (DCU), a general purpose arithmetic unit (GAU), and a memory address calculation unit (ACU). These datapaths can operate in parallel, and they all have access to the local memory bus. The GAU has a 32 word register file, that can be configured as circular buffers, and the ACU has a register file that can be used to locally buffer the features observed in the current frame.

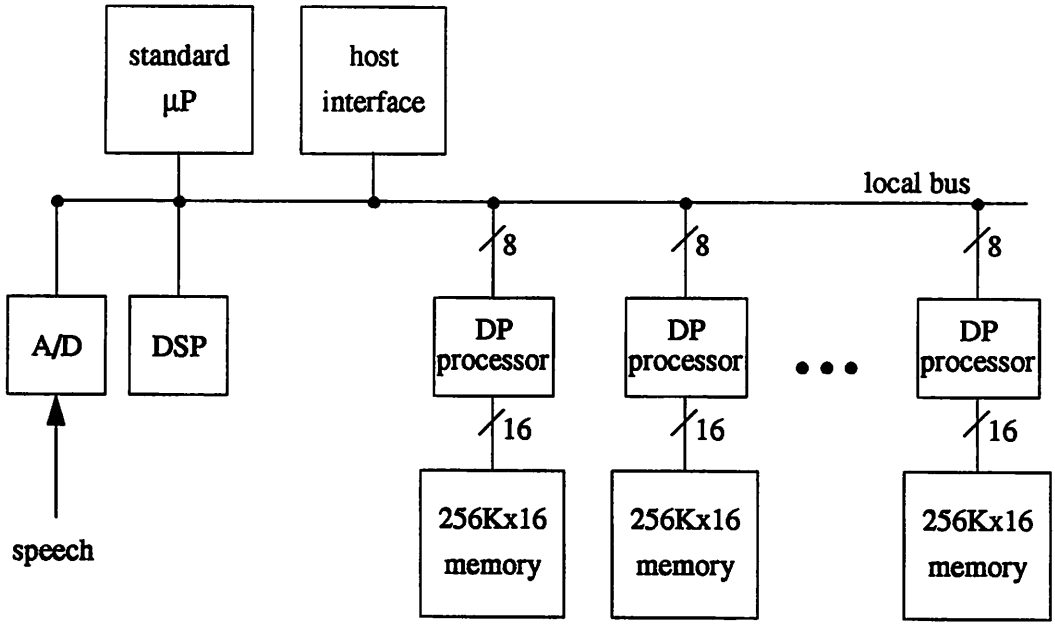


Figure 13: System Architecture for the Dynamic Programming Processor

LIMSI-CNRS has successfully used this processor for HMM based speech recognition. For that, the HMM was simplified in two ways: first, the topology of the HMM of vocabulary words is regular, each state has the same number of predecessors and the same relative predecessors, and transitions between states have no associated transition probability. Second, the output probability is computed using an approximation of tied gaussian mixtures: instead of using gaussian distributions, the mixtures are modelled as triangular functions. Using this model, the DCU can compute the output probability using, in the log domain, the sum of differences between the observed features and the means of the triangular distributions.

To process a state, it is possible to utilize the circular buffer of the GAU: Since the topology of the HMM is regular, each state has the same relative predecessors. To update the circular buffer after a state has been processed, the predecessor state that was the longest time in the buffer gets replaced with the predecessor state that is closest to the next state processed. Using this scheme, it is possible to cut down the number of

external memory accesses (14-16, depending on how many mixture vectors have to be read), and to process one state within 2 to 3 μsec [Qué91].

Thus, one DP processor chip can update about 4,000 states within a frame of 10ms, and LIMSI-CNRS implemented a 500 word real-time speaker-dependent connected speech recognition system using one such processor.

To use this processor in a real time 60,000 vocabulary system, it would be necessary to process 200,000 states within a frame, therefore 50 DP processors would have to be used. Also, the restricted topologies of the HMMs would impose severe restrictions on the usefulness of the system.

3.4. Advantages of a System with Full Custom VLSI Processors

The previous sections showed, that the general purpose systems described cannot be used for large vocabulary speech recognition, and that a system that employs the described full custom processors would be large. The computational requirements for a 1,000 word recognition system are, to update 4,000 states within a frame [Bis89], while 200,000 states per frame have to be updated in a 60,000 word recognition system (4. 2. 2.). This corresponds to a factor of 50, and this performance gain can be achieved with a system that uses dedicated VLSI processors that implement the recognition algorithm directly in hardware.

While it is necessary to use full custom hardware for large vocabulary systems it might be advantageous to use a custom hardware implementation even for recognition systems with a small vocabulary. General purpose systems are often more expensive and consume more power than what can be achieved with a more optimized implementation. The reason for the increased power consumption is that these processors use a fast clock (33MHz and more) and generally have additional circuitry that is not needed for that specific task. Also, there usually are additional peripheral

circuits necessary to interface to the general purpose components. A full custom solution, on the other hand, uses the smallest amount of circuitry and the lowest clock speed necessary to implement a certain task, and it is possible to implement architectural decisions and circuit techniques to minimize power consumption [Cha91].

Future portable systems such as notebook and laptop computers or personal communication services (PCS) will have speech recognition as an alternative to the keyboard or to replace the keyboard entirely [Bur91]. In these applications, the cost, size, weight and power consumption of speech recognition hardware is an important issue. The recognition accuracy of HMMs for specific tasks is acceptable, but the cost and the power consumption of general purpose solutions is not acceptable for most of these future products. Special purpose solutions, on the other hand, can be tailored just for a specific algorithm. Thus, they can be smaller, less expensive, and can be designed in such a way that they consume less power than general purpose solutions.

4 Algorithmic Modifications for Hardware Implementation

This chapter describes modifications to the speech recognition algorithm which were necessary to relax the hardware requirements for a real time implementation. These modifications involve the representation of the HMM speech model and the implementation of the Viterbi algorithm.

4.1. "Brute Force" Requirements for a 60,000 Word System

This thesis describes the implementation of a 60,000 word recognition system. To evaluate the requirement of the system, we extrapolated from the DECIPHER systems implementation of a 1,000 word connected speech recognition system [Murv89]. This section demonstrates the hardware requirements if the recognition algorithm is implemented without modifications.

On the average, a word is modeled using 17 states, therefore the recognition system has to store the model parameters of 60,000 * 17 $\approx 10^6$ states. In the DECIPHER system, states have an average of 3 predecessor states and they have four associated discrete output distributions with 256 probabilities $P(o_i|s)$ each. Therefore, to store the model parameters for one state, 4*256 memory locations are needed to store the output distributions and 6 memory locations to store the transition probabilities and the pointers to predecessor states. This results in a requirement of $(4 \cdot 256+6)10^6 \approx 10^9$

memory locations to store the model parameters. Also, two memories with 10^6 locations are needed to store the state probabilities for the 10^6 states, $P(O_{i-1}, s)$ and $P(O_i, s)$.

Since the system computes the Viterbi equation over all the states in real time, (EQ 6), (EQ 7), and (EQ 8) must be evaluated for 10^6 states within the frame duration of 10 msec. (EQ 6) requires 4 memory accesses per predecessor (topology, $P(O_{i-1}, s)$, $Tag(s,i-1)$, and transition probability), 4 memory accesses to compute the output probability (EQ 8), and one write access to store the final result $P(O_i, s)$. Thus, the system would have to keep up with $(3 \cdot 4+5) \cdot 10^6$ memory accesses per 10 msec = $1.7 \cdot 10^9$ memory accesses per second. The performance of the above equations involve 1 MAX operation, 1 ARGMAX operation, and 7 multiplications per state, so that the computational throughput of that system would be $900 \cdot 10^6$ operations per second.

Fortunately it is possible to use hierarchy and pruning strategies to dramatically reduce these memory and computation requirements.

4.2. Changes that Affect the Performance of the Algorithm

4. 2. 1. Number Representation, Normalization

One level of freedom in the design of a full custom hardware implementation is the number representation for the variables in the algorithm. To achieve the required dynamic range, a floating point representation could be used, but this would result in complex floating point datapaths and relatively large storage requirements.

To avoid this increased complexity, the hardware was implemented using a fixed point representation, but with re-normalization to retain the desired accuracy. Since the recursive nature of the Viterbi algorithm results in state probabilities that become smaller with every recursion (EQ 6), it is necessary to normalize the state probability variables after each frame. This not only avoids arithmetic underflow, it also

makes it possible to represent these variables with a minimal number of bits. The normalization is done using the following equations:

$$P_n(O_i, s) = \frac{P(O_i, s)}{MAX_t[P_n(O_{i-1}, t)]} \qquad \text{(EQ 11)}$$

In this equation, $MAX_t\,[P_n(O_{i-1}, t)]$ is the largest (normalized) state probability that occurred in the previous frame.

The division in (EQ 16), however, is costly to implement as are the multiplications of the predecessor state probabilities with the transition probabilities in (EQ 6). A solution to this problem is a logarithmic number representation of all variables in the recognition system. Multiplications can then be implemented using additions, and divisions are reduced to subtract operations. Thus, the Viterbi recursion (EQ 6) is implemented using:

$$\mathcal{P}(O_i, s) = MIN_{p \in \{pred\}}[\mathcal{P}(O_{i-1}, p) + \mathcal{A}(p, s)] + \mathcal{P}(o_i|s) \qquad \text{(EQ 12)}$$

$$\mathcal{P}(O_i, s) = -\log P(O_i, s)$$

$$\mathcal{A}(p, s) = -\log A(p, s)$$

$$\mathcal{P}(o_i|s) = -\log P(o_i|s)$$

Since all variables in the system describe probabilities, which have a value in the range between 0 and 1, the logarithm of these variables will always be non-positive. Thus, it is possible to ignore the sign and treat the log numbers as positive numbers. As a result of this, MAX operations are implemented as MIN operations. Probability 0 is coded using the largest number in the fixed point logarithmic representation. The logarithmic representation also has the advantage that the range of values for a given wordlength is much higher than for a linear fixed point representation. The accuracy for

small values in the logarithmic domain (probabilities close to 1) is reduced, but the range for small probabilities is larger.

Simulations using the DECIPHER system showed, that recognition accuracies do not degrade compared to a floating point representation if the variables use the following wordlengths in the logarithmic domain:

State Probabilities -log $[P(O_i, s)]$	16 bits
Output Probabilities -log $[P(o_i\|s)]$	10 bits
Transition Probabilities -log $[A(s, t)]$	4 bits

Table 3: Wordlengths for the Representation of the Probabilities

Table 3 shows that transition probabilities are quantized using only 4 bits. However, to assure a higher dynamic range than 0 to 15, they are shifted 4 bits to the left before they are added to the state probabilities. Simulations showed that this dynamic range and quantization does not degrade recognition accuracy while minimizing the wordlength. The representation for an output probability is 10 bits, therefore the wordlength of a joint distribution which is the product (sum in the log domain) of 4 output probabilities uses 12 bits (EQ 8). Figure 14 graphically shows the wordlengths and their weights relative to the state probabilities:

4. 2. 2. Pruning

To reduce the amount of computations that have to be performed, this system uses a pruning algorithm which discards word models whose states have low state probabilities. The decision to prune a word is based on a pruning threshold, Θ. If every state in a word has a state probability lower than the pruning threshold, the state

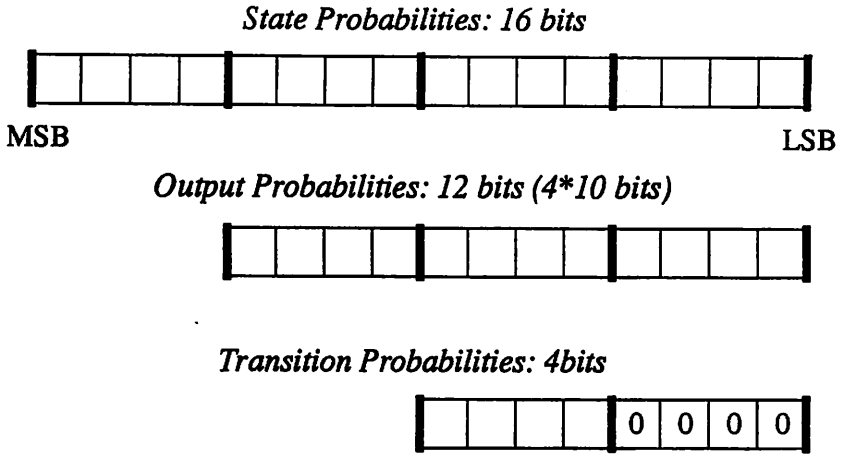


Figure 14: Wordlength Representations for Probabilities

probabilities of this word will not be computed in the next frame. The pruning threshold is computed using the following equation (in the linear domain):

$$\Theta_{new} = MAX\,[\Theta_{old}, P\,(O_i, s) \cdot \Theta_{offset}] \qquad \text{(EQ 13)}$$

Θ_{offset} is a fixed probability value that is preset before processing new frame and it determines the number of states that are evaluated. The threshold, Θ_{new}, is adjusted as new state probabilities are being computed. Thus, the probability, Θ_{new}, increases as states are processed and have high state probabilities. At the beginning of a frame it is possible that initial states with low state probabilities are active until Θ_{new} gets adjusted to a higher probability. To avoid that, it is possible to preset Θ_{new} before a frame is processed.

Simulations for the DECIPHER 1,000 word recognition system showed that recognition accuracies do not degrade if only 1/6th of the states are left active. If a more conservative number is used (1/5th) and extrapolated to the number of active states of a 60,000 word recognitions system, only ≈ 200,000 active states would have to

be processed, instead of 10^6 active states for a non-pruned system (see chapter 2). Thus, the computational throughput for the Viterbi algorithm would drop from $900 \cdot 10^6$ operations per second to $\approx 180 \cdot 10^6$; the number of memory accesses, from $1.4 \cdot 10^9$ to $\approx 0.3 \cdot 10^9$ per second. The pruning algorithm adds, however additional computations: one addition and one MIN operation (log domain) for (EQ 13), and one operation to compare each state probability to the pruning threshold. These operations have to be performed for 200,000 active states per 10 ms, therefore pruning requires 80 million operations per second.

4.3. Changes that do not Affect the Performance

4. 3. 1. Hierarchical HMM

The memory requirement for a system with a vocabulary size of 60,000 words was derived and requires an address space of about 10^9 words. Obviously, this requirement, which is the result of a flat HMM representation, is costly to implement in a system that uses memory chips for fast access times.

A flat representation of an HMM is also undesirable for training an HMM. As mentioned above, one needs many utterances of each speech segment to train the corresponding HMM. In a hierarchical description, there is a small set of basic HMMs that describe small units of speech (phones). These basic HMMs are then instantiated and concatenated to build words. Obviously, this representation yields - for a given set of training utterances - more training data per basic HMM unit than if the system is described flat. For example, the National Institute of Standards and Technology (NIST) offers a database to train a speaker independent connected digit recognizer, and this database has 4331 training utterances for each of the words. This number of training utterances per word is adequate for training a speaker independent HMM, using less data for training results in decreased recognition accuracy. Extrapolating this number to

a 60,000 word recognition system with a flat HMM, we need at least 60,000 · 4331 $\approx 260 \cdot 10^6$ words for training and it is likely that the increased size of the vocabulary would require an even larger training set.

It is more desirable to find small speech units that are the building blocks for a complete language model. The English language, for example, has about 30 basic phones, and using these phones we could describe all possible words. However, the phones are not always pronounced in the same way, depending on the predecessor or successor phone (coarticulation). Thus, we need more phone models to take into account the left and right phonetic context [Lee90]. In the worst case, every phone could be influenced by all the possible neighboring phones, and $30^3 = 27{,}000$ context dependent phones would be required. In practice however, much fewer context dependent phones on the order of $(2 .. 10) \cdot 10^3$ are required.

The description of a basic speech unit such as context dependent phones will be called a *unique phone*. It contains the topology of the model, the transition probabilities, and the output distributions associated with the states. Many unique phones share the same topology and it would be redundant to replicate this information in every unique phone. Thus, this system stores a basic set of phone topologies and the description of a unique phone references its corresponding topology.

Given a basic set of unique phones, we can describe words or complete languages in the next level of hierarchy as shown in Figure 15.

Using only one or two basic phone topologies, unique phones are differentiated by specifying the transition probabilities and the output probability distributions, and by referring to their phone topologies. To describe a word or a sentence, these unique phones are instantiated and the instances concatenated using transitions with associated transition probabilities. These instances of unique phones are called *phone instances*[1].

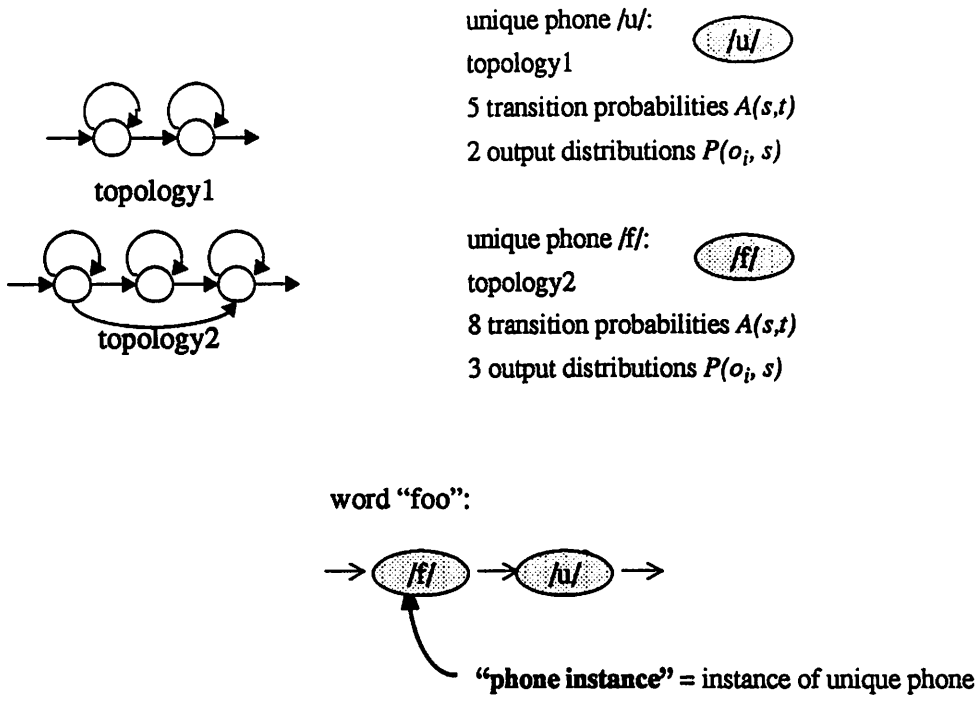


Figure 15: hierarchical HMM

In this system, we use two levels of hierarchy: the *phone level* describes unique phones while the *grammar level* describes transitions between phone instances. Therefore, the basic building blocks of a grammar are phone instances and transitions between them, and the grammar level of this system describes both, transitions between phone instances inside a word, and transitions between words.

The hierarchical HMM also allows major reduction in the memory requirements. Using the same model size assumptions as in section 4.1., we need to store the model parameters of at most 30^3 = 27,000 phones. Again, assuming that each phone has 3 states, that each state has 3 predecessor states and 4 discrete output distributions with 256 probabilities each (chapter 2), we need 27K * (3*256*4 + 3) ≅ $82 \cdot 10^6$ memory locations to store the model parameters, a reduction by a factor of 10 over the $900 \cdot 10^6$ locations requirement of 4.1.

---

1. Phone instances are sometimes called wordarcs in the literature. This comes from the fact that typical grammar topologies represent speech units with arcs rather than nodes.

However, the trade off is that accessing the model parameters now involves address computations. For example, we need 4 pointers to uniquely determine the predecessor of a particular state: the phone instance, the unique phone, the reference to the unique topology, and the state.

4. 3. 2. Grammar Processing and Phone Processing

Despite the hierarchical representation of the output and transition probabilities, the HMM has to be flat in order to perform the Viterbi algorithm of (EQ 6). If a given unique phone is instantiated twice, the different instances will have different state probabilities. This means states inside phone instances have to be represented without hierarchy in order to store their state probabilities.

In this system, the Viterbi recursion (EQ 6) is simultaneously performed on two levels of the hierarchy. In *phone processing*, the state probabilities of phone instances are updated considering only local transitions inside the phone instance. Using the results of these computations (state probabilities) and transition probabilities between phone instances, the probabilities of paths to the successor phone instances are computed in *grammar processing*.

This scheme has the advantage that the computations on the two hierarchy levels can be done in parallel, however, there is a considerable amount of data (state probabilities) that have to be passed between them. In the worst case, every state inside a phone instance can have a transition to a state in the successor phone instance, and this successor phone instance (phone instance z in Figure 9) could have several predecessors. To compute the state probability of the first state s of phone instance z, it is therefore necessary to access all probabilities $P(O_t, p)$ from the states inside the predecessor phone instances (see Figure 17).

To reduce this data rate between the hierarchies, two "compressed" state probability are defined. $P(S)$, the *source grammar node probability*, gives the

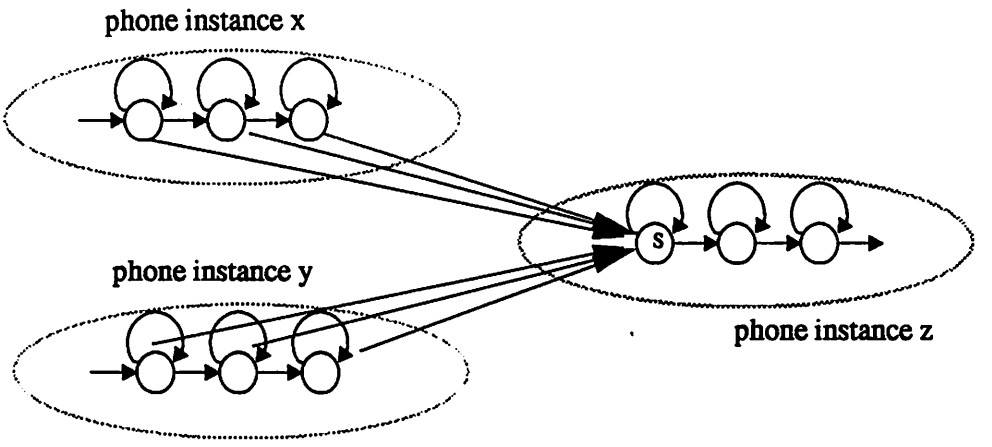


Figure 16: Concatenation of Unique Phones in Grammar Processing

probability that the most probable state sequence terminates at the beginning (source) of a phone instance. This probability corresponds to the result of the inner loop of the Viterbi recursion, $MAX_p\ [P(O_i,\ p)\ \cdot\ A(p,s)]$, for all predecessor states p of all predecessor phone instances.

Since a unique phone will be instantiated, it will have in every case different transitions to successor phone instances. The source grammar node probability is computed in two steps, the first step is on the phone level, in which a a *destination grammar node probability* $P(D)^i$ is computed using the following equation:

$$P(D)^i = MAX_{p \in \{pred\}} [P(O_i, p) \cdot A(p, D)] \qquad \text{(EQ 14)}$$

$P(D)^i$ is the likelihood that the most probable path at frame i terminates at the end of a phone instance, and it is computed by finding the most likely transition from the states inside the phone instance to the end of the phone instance. In (EQ 14), *{pred}* is the set of states that have a transition to the destination grammar node. This

probability gets computed for every phone instance in every frame. The transition probabilities at the end of a particular phone instance, *A(p,D)*, are defined in the unique phone that was instantiated by that phone instance. Therefore, every instance of the same unique phone has the same set of transition probabilities *A(p,D)* to the outside and they cannot be used to describe transitions on the grammar level (after all, different instances of the same unique phone can have different grammar transitions). Therefore, transitions on the grammar level are described using another set of transition probabilities between destination and source grammar nodes. For that, the phone processing system sends $P(D)^i$ to the grammar processing system which computes the source grammar node probability of a successor phone instance using transitions *A(D,S)* between destination and source grammar nodes (or, in other words, between phone instances):

$$P(S)^i = MAX_{D \in \{pred\}} [P(D) \cdot A(D,S)] \qquad \text{(EQ 15)}$$

Here, *{pred}* is the set of all destination grammar nodes *D* that have a transition to the source grammar node *S*. This scheme is shown in Figure 17. Every phone instance has, besides the description of the states and the transitions, two additional "grammar nodes". These grammar nodes have no associated output distribution, which means a transition into these nodes does not produce a speech output (observation). The only information they keep are the grammar node probabilities that are passed between the hierarchy levels.

The shaded nodes in Figure 17 are the grammar nodes. The source grammar node keeps the probability that a certain phone instance starts and the destination grammar node keeps the probability that a phone instance ends. All transitions between phone instances originate and terminate in grammar nodes only. For example, the destination grammar node probability for phone instance x in Figure 17 is computed

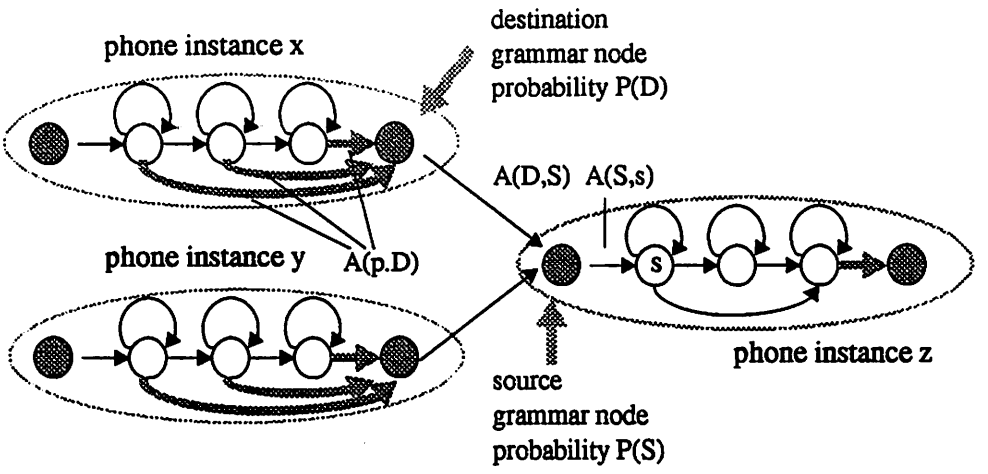


Figure 17: Concatenation of Unique Phones (inside dashed ellipses) using Grammar Nodes

using the states in the phone instance and destination grammar node transitions *A(p,D)*. The source grammar node of phone instance z keeps the maximum of the product of the destination grammar nodes of phone instances x and y multiplied with the transition probabilities *A(D,S)* to this source grammar node.

Thus, to update the first state s of phone instance z there are only two predecessors: the source grammar node and state s itself. In a more complicated model, there could be several transitions from the source grammar node to various states in the phone instance.

Similar to (EQ 16), the grammar node probabilities have to be normalized to prevent overflow for the fixed point representation. This normalization is performed on the destination grammar node according to (EQ 16):

$$P_n(D)^i = \frac{P(D)^i}{MAX_t[P_n(O_{i-1}, t)]} \qquad \text{(EQ 16)}$$

In this equation, $MAX_t\ [P_n(O_{i-1},\ t)]$ is the highest (normalized) state probability (of state t) that occurred in the previous frame.

Using this scheme, it is possible to process (EQ 6) in parallel on different processes, phone processing and grammar processing. This not only yields faster performance than using just one process, it also allows us to customize the implementation of these processes for the distinct requirements of the two levels.

Phone processing requires every state in the system that is not pruned to be processed within a frame duration of 10ms. This involves, for every state, the computation of the state probability, the backtrack tag, normalization, updating the pruning threshold, and pruning. The amount of computation in grammar processing is less, only transitions between non-pruned phone instances have to be processed. Inside a word, there is only one transition between two phone instances, while in phone processing, there are 3 transitions for every state. Also, pruning can be used to cut the number of phone instance transitions that have to be processed at word boundaries (see below).

On the other hand, the topology (or the trellis structure) of state transitions inside phones is fairly simple since the transitions strictly go left-to right, and there are only a few predecessor states for a given state. Most importantly, transitions inside phones are local, there is no state transition that skips more than a few states. This is different for transitions between phone instances, since while they are only left-to-right inside a word, at word boundaries they can go to any other phone instance (ergodic HMM, see section 2.2.). Thus, they are not local and a given phone instance can have a large number of predecessor phone instances.

As a result, the difference in the computational requirements between phone processing and grammar processing requires different strategies for the hardware architecture of each. The isolation of the computations for these two levels of hierarchy

using the source and destination grammar nodes makes it possible to do these optimizations.

4. 3. 3. Predecessors Processing vs. Successor Processing

This section introduces two implementations of the Viterbi algorithm. One is based on computing the state probability of a state by accessing all its predecessor states (predecessor implementation) while the other implementation updates successor state probabilities (successor implementation). It is shown, that the former is the most appropriate implementation for phone processing, while the latter is well suited for grammar processing.

4. 3. 3. a Predecessor Implementation

Figure 18 shows the flowgraph of the Viterbi recursion described in (EQ 6) . This equation chooses the maximum of the state probabilities of the predecessors of a state multiplied with their transition probabilities to that state. In Figure 18 , the state probabilities of states at frame *i-1*, $P(O_{i-1}, p)$, are associated with the nodes in the left column. The task is to compute the state probabilities $P(O_i, s)$, which are represented by the nodes in the right column. Nodes on the same row correspond to state probabilities of the same state at different frames, and the rows are ordered in such a way that probabilities on neighboring rows correspond to states that are neighbors in the HMM of Figure 4 . A node above a particular node is associated to a state that is to the left, and a node below is associated with a state that is to the right of a particular state.

To compute $P(O_i,n)$ for state n in Figure 18 , the state probabilities of the predecessor states, $P(O_{i-1},l)$, $P(O_{i-1},m)$ and $P(O_{i-1},n)$, have to be multiplied with the transition probabilities to n. The maximum of the resulting three products is then multiplied with the output probability $P(o_i \mid s)$. Thus, a state probability is computed by accessing all predecessors of a state, hence the name *predecessor* implementation.

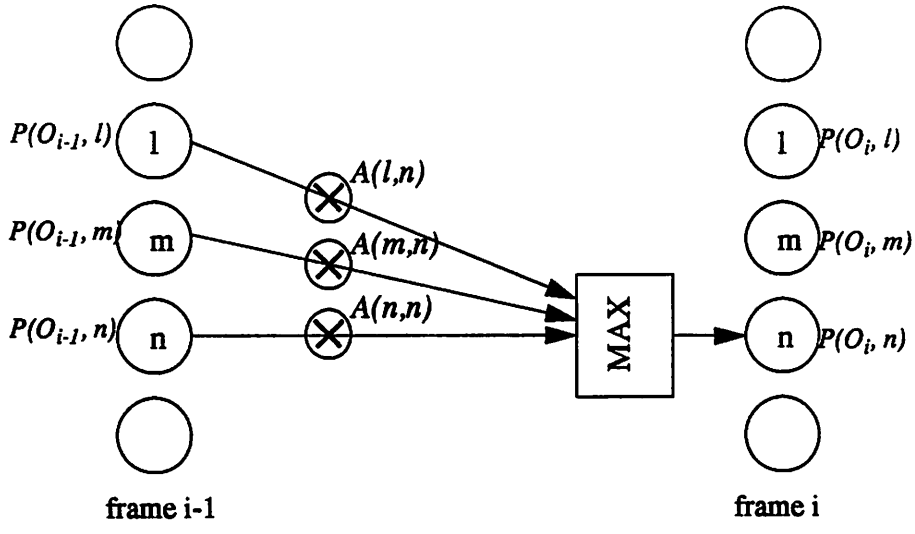


Figure 18: Lattice Structure for the Predecessor Implementation

4. 3. 3. b Successor Implementation

Another implementation of the Viterbi algorithm is the *successor* implementation formalized in the following equation:

$$P_{new}(O_i, z) = MAX\,[P_{old}(O_i, z), P(O_{i-1}, s) \cdot A(s, z) \cdot P(o_i, z)] \qquad \text{(EQ 17)}$$
$$z \in \{\text{successor states of } s\}$$

Figure 18 shows the data flow for this implementation.

Here, the algorithm accesses all successor states z of a certain state s that has the state probability $P(O_{i-1}, s)$ at frame $i-1$.

There are several states that share the same successor, and the goal is to find the transition to this successor state that results in the overall highest state probability $P_{new}(O_i, z)$. Therefore, if the successor state z has a state probability $P_{old}(O_i, z)$ that was computed previously (originating from a different state), it has to be updated if the

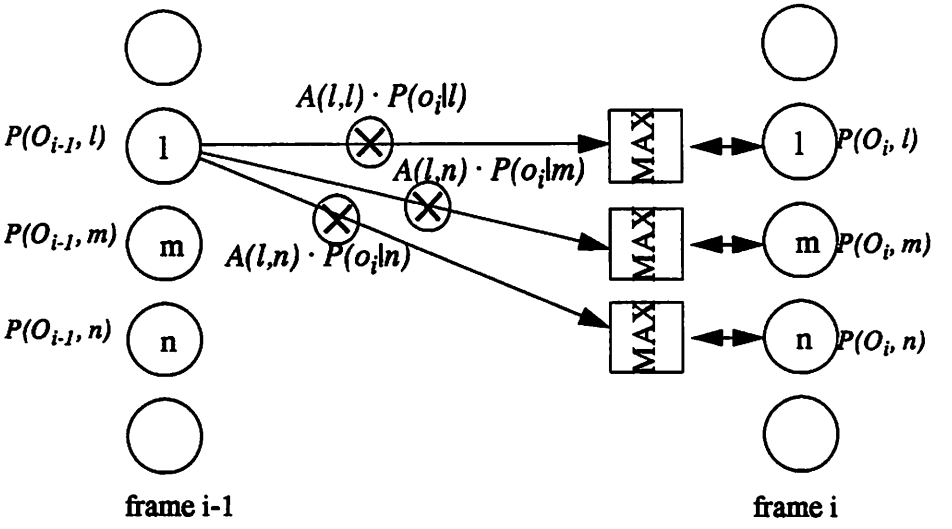


Figure 19: Lattice Structure for the Successor Implementation

transition from another state yields a higher state probability. Before this computation starts at the beginning of a frame, it is necessary to initialize the state probabilities of all successor states to probability 0. If (EQ 17) has been performed for all successors of every state in the vocabulary for a certain frame i, the states will have a new set of state probabilities $P_{new}(O_i,z) = P(O_i,z)$ that correspond to the highest probable paths.

4. 3. 3. c Trade Offs between Successor and Predecessor Implementation

The successor implementation has the advantage that it is straightforward to save computation using the pruning algorithm described above: states with low state probabilities have a low likelihood of terminating the most probable path. If the Viterbi algorithm is implemented using transitions to successors (EQ 17) it is obvious that states with low state probabilities, $P(O_{i-1},s)$, cannot contribute towards high state probabilities $P_{new}(O_i,z)$ for the successor states z. Therefore, if such a state with low $P(O_{i-1},s)$ is encountered, all computation for it's successors can be dropped.

The successor implementation, however, results in a higher memory bandwidth than the predecessor implementation. To update $P_{new}(O_i,z)$, the memory containing the state probabilities for time i (state probability memory i) has to be read to get $P_{old}(O_i,z)$ and then $P_{new}(O_i,z)$ has to be updated if it has a higher probability than $P_{old}(O_i,z)$. This operation has to be repeated for every predecessor of z. In the predecessor implementation, this memory is only accessed once, when the final result, $P(O_i,z)$, is written.

Thus, to improve the memory bottleneck for the performance of the Viterbi algorithm, it is better to use the predecessor implementation. Here, the bottleneck is the state probability memory i-1 which has to be accessed once per predecessor, while in the successor implementation the bottleneck is the state probability memory i which has to be accessed twice per successor. But the drawback of the predecessor implementation is that pruning is not straightforward to implement: we can only skip the computation of a state probability after we processed all predecessor states, because there might be a predecessor state with a high state probability contributing to a high state probability of the current state. Thus, we can only prune after doing all the work, and pruning does not save any computation.

In this system, the computation of the Viterbi algorithm is done on two hierarchy levels: phone processing and grammar processing (see section 4. 3. 2.). As already mentioned, the computational requirements on these two levels are quite different:

- phone level: high throughput, but left-to right HMM with a few local transitions
- grammar level: less throughput, but ergodic HMM with many irregular global transitions

Since it is necessary to achieve a high computational throughput in phone processing, it was implemented using the predecessor implementation because it results

in a smaller memory bandwidth than the successor implementation. Grammar processing, on the other hand, is based on the successor implementation for two reasons. First, the inputs to the grammar processing system are destination grammar nodes sent from the phone processing system. Implementing the Viterbi recursion based on successors makes it possible to process these inputs as they arrive. If grammar processing were implemented based on predecessors, all destination grammar nodes had to be stored before the source grammar nodes could be processed. Secondly, in successor processing it is very straightforward and effective to make pruning decisions: if a destination grammar node has a state probability $P(O_{i-1}, s)$ that is smaller than the pruning threshold Θ (see 4. 2. 2.), it is not necessary to update the successor nodes since the result would also be less than Θ (probability always decreases). Even further, if $P(O_{i-1}, s)$ is slightly above the threshold and the successors are sorted based on the values of the transition probabilities $A(s,z)$, we can stop processing the successor nodes (source grammar nodes) as soon as the first $P(O_i, z)$ is smaller than Θ.

As a result, the pruning algorithm is implemented in the following way: The grammar processing system maintains a list of active phone instances. Thus, the phone processing system only processes the states and destination grammar nodes of phone instances on that list. If a phone instance that had been processed by the phone processing system has at least one state with a high state probability ($>\Theta$), the phone instance is shipped to the grammar processing system which keeps the phone instance on the list of active phone instances. An active phone instance that was processed but not returned from the phone processing system is deactivated. On the other hand, if an active phone instance has, after phone processing, a high destination grammar node probability ($>\Theta$), it is also sent to the grammar system. In this case, the phone processing system gives a request to compute the source grammar node probabilities of the successor phone instances. Transitions to the successors of a destination grammar node in the grammar processor model memories are sorted based on their probability, therefore it is possible to stop processing a destination grammar node as soon as the

first successor source grammar node has a probability smaller than Θ. Only the phone instances that have a source grammar node higher than the pruning threshold are added to the list of active phone instances. If the destination grammar node probability is lower than the pruning threshold, it is not necessary to generate a request to compute the successor transitions, since a low destination grammar node probability cannot contribute to high source grammar node probabilities of the successor phone instances.

In this implementation, pruning is done based on phone instances: If a phone instance has one state with a high state probability, it is activated for the next frame. Thus, all states in this phone instance will be processed, even those with a low probability. This results in some computational overhead, but implementing a list of active phone instances is easier than implementing a list of active states since it requires less information to be stored as there are fewer active phone instances than active states.

4. 3. 4. Backtrack Algorithm

To recover the most likely state sequence after a sentence has been processed, we traverse a list of backtrack tags that were generated during the performance of the Viterbi Algorithm (see section 2. 3. 2.). This list contains, for every active state in every frame, a tag that identifies the state's predecessor on the most likely path.

An implementation of that list would be very costly since a sentence might have several hundred frames, and for each frame we might have to store 200,000 tags. Instead, it is possible to implement another scheme in which it is not necessary to exactly recover the sequence of states, since the important information is the sequence of phone instances. So all we need to store are tags associated with grammar nodes: each active source grammar node in each frame has a tag that identifies its predecessor phone instance. In the succeeding frames, this tag gets copied into the states according to (EQ 7), and it eventually arrives at the destination grammar node of that phone

instance. Thus, the destination grammar node has a tag that points to the predecessor of that phone instance.

If the destination grammar node has a high probability ($>\Theta$), it could be part of the most probable path and therefore it is important to store the tag in a backtrack list along with the identification of the current phone instance. After that, the memory address that was used to store the old tag is passed as a new tag to the source grammar node of the successors of that phone instance, and in the succeeding frames it will eventually be passed to the destination grammar node (see above) and so on.

Thus, tags are memory pointers that specify the location of a structure in the backtrack list that contains a phone instance id and a pointer to its predecessor phone instance. Using this scheme, we generate an array of linked lists where each element contains the identification of the word and a pointer to the list that contains the structure associated with a predecessor word (Figure 20).

By storing backtrack tags only at word boundaries, it is possible to reduce the memory requirement for the backtrack list. At the end of the sentence, this linked list has to be traversed starting from the tag of the state with the highest state probability in the last frame.

Using this scheme, the backtrack list has to have a size which is the number of destination grammar nodes per frame with probability higher than the pruning threshold Θ, times the number of frames in a sentence. Simulations of the DECIPHER system showed, that about 5% of the active phone instances have a destination grammar node probability that is higher than the pruning threshold, so in the worst case, if each frame has 60,000 active phone instances (200,000 active states divided by 3 states per phone instance), the phone processing system sends about 3,000 phone instances with high grammar node probabilities to the grammar system. Thus, the grammar system has to store the phone instance and the backtrack tag of 3,000 grammar nodes per frame, and

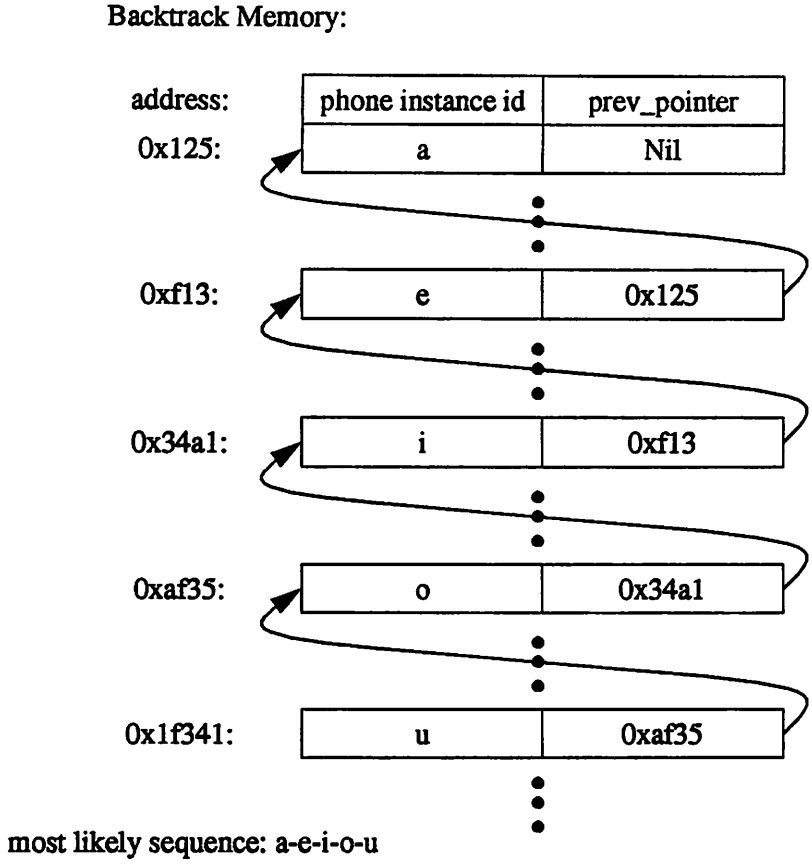


Figure 20: Implementation of the Linked List in the Backtrack Memory

use the address as a new tag for the successor phone instances of that grammar node. In this system, 1 million addresses are allocated for the backtrack list, so we can generate a list for 1,000,000 / 3,000 ≈ 333 frames (3.3 seconds of speech), and the backtrack tag has to be 20 bits wide.

5

System Architecture

This chapter describes the system architecture of the speech recognition hardware. It was implemented as a VME based system, and the basic components were a VME card cage, a general purpose processing board based on the Motorola 68020 microprocessor (Heuricon board, [Vw90]), an ethernet card that interfaces the Heuricon board to various workstations, and three custom designed boards that implement the speech recognition algorithm.

The first section describes the functional blocks of the hardware, which were implicitly described in chapters 2 and 4. In the second section, we relate the functional blocks to a general purpose hardware or to a full custom hardware implementation.

5.1. Functional Partition

A high level functional block diagram of the speech recognition system is shown in Figure 24. The speech recognition functions are front end processing, phone processing, grammar processing, and backtracking.

The algorithm used for front end processing was described in chapter 3. In essence, the incoming speech waveform is sampled and processed yielding a stream of

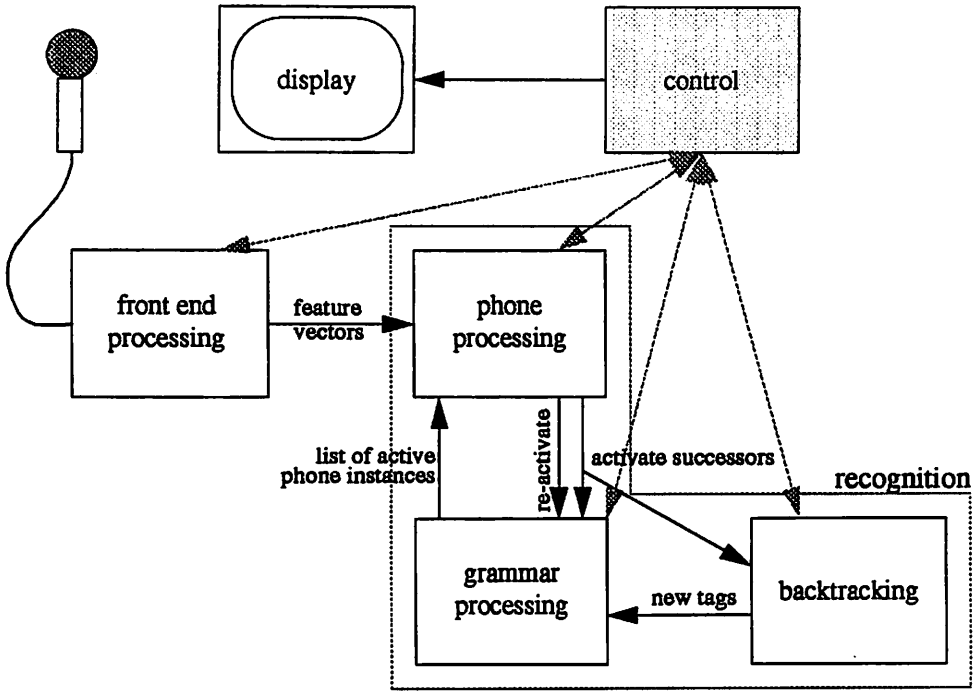


Figure 21: Functional System Partition

vector quantized features at a rate of 4 features every 10 ms. These features are sent to the phone processing system, and a feature is represented using 8 bits.

5. 1. 1. Phone Processing

The phone processing system (Figure 23) computes the state probabilities and the backtrack tags of states in active phone instances, and their destination grammar node probabilities and backtrack tags (chapter 2). The active phone instances of a particular frame are stored in a memory called active phone instance memory. This memory is part of the grammar processing system, and it will be described in more detail in the following section.

To process active phone instances, the following data are required:

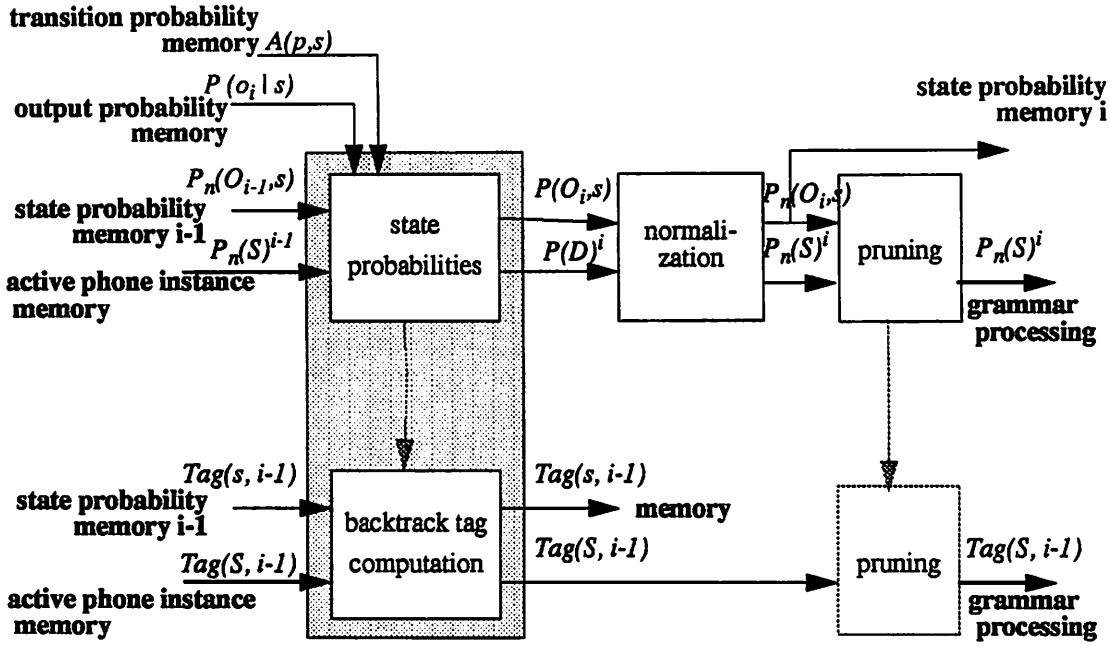


Figure 22: Functional Diagram of the Phone Processing System

- HMM model parameters of phone instances. These parameters (topology, transition probabilities and output distributions) are stored locally in the phone processing system.

- Pointers to these parameters. These pointers are provided by the active phone instance memory which stores the unique identification of active phone instances (phone instance id, unique state id, topology address). To derive the output probabilities, we also need the feature vectors that come from the front end processing system.

- State probabilities and backtrack tags of active phone instances that were computed in the previous frame. These parameters are also stored locally in the phone processing system. One memory stores these parameters associated with the previous frame (state probability memory i-1), and another memory stores the updated parameters (state probability memory i).

- If a phone instance was active in the previous frame, the active phone instance memory provides a pointer to the state probabilities and backtrack tags in the state probability memory i-1. This information is also stored in the active phone instance memory.
- The source grammar node probabilities and backtrack tags of active phone instances which are also stored in the active phone instance memory.

After they are computed, the normalized state probabilities and backtrack tags of active phone instances are locally stored in the state probability memory i. If at least one of the state probabilities of a phone instance is higher than the pruning threshold, information associated with that phone instance is sent to the grammar system with a request to re-activate this phone instance by adding it to the list of active phone instances for the next frame. This information contains the destination grammar node probability, the backtrack tag and an address to the state probability memory to locate the state probabilities of that phone instance. If the destination grammar node has a high probability ($>\Theta$), the phone processing system sends a request to the grammar system to compute the source grammar nodes of the successors of this phone instance, and to activate these successors by adding them to the list of active phone instances if the source grammar node probabilities are higher than the pruning threshold.

5. 1. 2. Grammar Processing

Another process in the recognition system is grammar processing (Figure 24). In section 4. 3. 2. we specified that it's function is to compute the source grammar node probabilities of phone instances that are to be activated, and to generate a list of active phone instances that have to be processed by the phone processing system in the next frame. phone instances are added to that list as a result of requests from the phone processing system (see above). For these computations, the following data are used:

- Information associated with phone instances that have a destination grammar node probability $>\Theta$ (who's successors should be activated), or that have at least one state with a state probability $> \Theta$ (who should be re-activated). This information includes the destination grammar node probability, the backtrack tag, the phone instance id, the unique state id and the topology address, and it is provided by the phone processing system along with the corresponding requests.
- The HMM model of the grammar. It defines the phone instances by referring to their unique phone id and topology address, and it specifies the successors of a phone instance and the transition probabilities to these successors. This information is stored locally in the grammar processing system.

If the successors of a phone instance are added to the list of active phone instances, the source grammar node probabilities of the successor phone instances are computed using the successor implementation of the Viterbi algorithm (section 4. 3. 3.). Therefore, the grammar processing system can activate a certain phone instance several times within a frame, each time with a different source grammar node probability (contributed from different predecessors). It is important not to replicate phone instances in the active phone instance memory, but to locate that phone instance in the memory and to maximize the source grammar node probability. Figure 23 shows a block diagram of the grammar processing subsystem.

If the phone processing system generates a request to re-activate a certain phone instance for the next frame, it is not necessary to specify a source grammar node (probability or backtrack tag). After all, only transitions between phone instances can generate a source grammar node probability, and the phone processing system does not deal with that. Therefore, only the phone instance id, the unique phone id, the topology address and the address that was used to store the state probabilities on the phone processing system (state probability address) have to be passed to the grammar processing system. The source grammar node probability is set to 0.

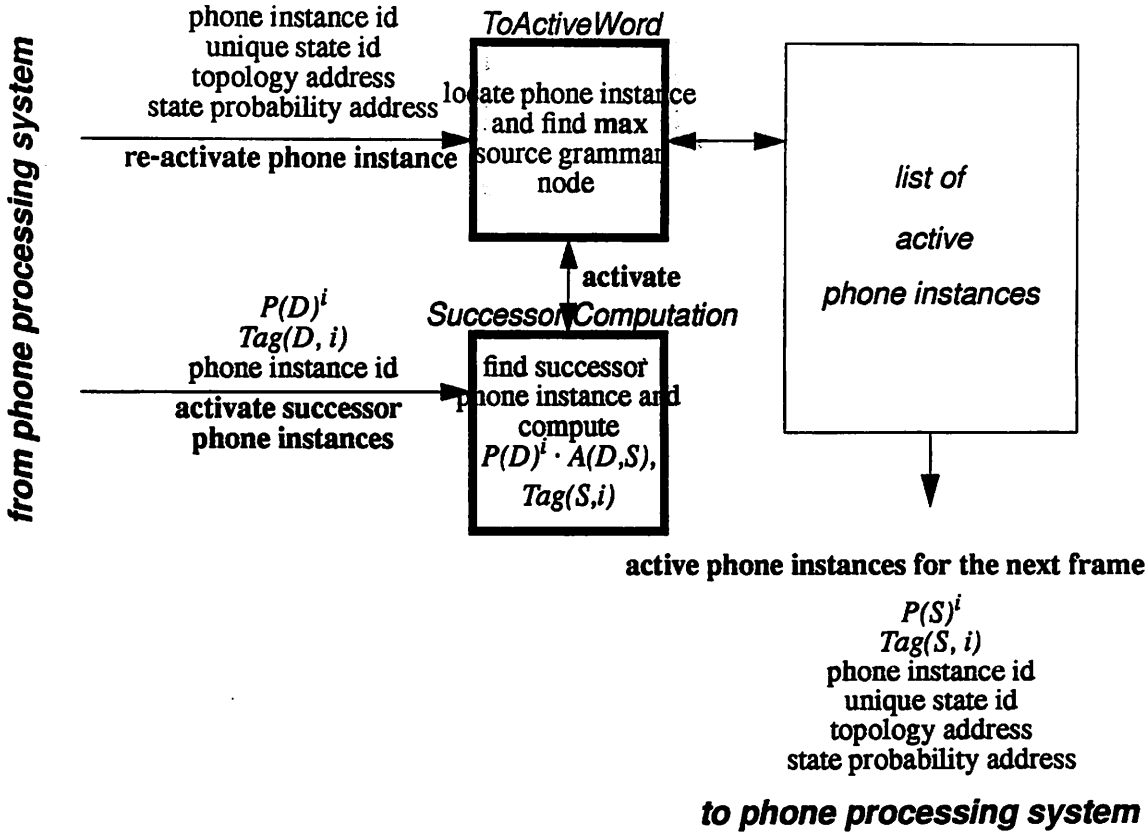


Figure 23: Block Diagram of the Grammar Processing System

A request to activate the successors of a phone instance, on the other hand, does not require any state probability addresses: such a request activates new phone instances with state probabilities that are 0. However, the phone instance id and the destination grammar node (probability and backtrack tag) of the phone instance that activates its successors has to be specified. For such a request, the grammar system has to perform the following tasks:

- The successor phone instances (phone instance id, unique state id and topology address) and the transition probabilities to these successors have to be determined using the grammar topology, and $P(D)^i \cdot A(D,S)$ has to be computed.
- This result along with the new backtrack tag and the phone instance specification then has to be added to the list of active phone instances (see Figure 16).

The *ToActiveWord* process is responsible for adding a phone instance to the list of active phone instances or to update the source grammar nodes (probability and backtrack tags) of phone instances that are already on that list (see Figure 23). This process determines if a certain phone instance is already on the list of active phone instances and if so, locates it and determines the maximum of the already existing source grammar node probability, $P(S)_{old}^{i}$ (EQ 17) , and the new probability, $P(D)^{i} \cdot A(D,S)$. This scheme also works if a phone instance was already re-activated as a result of a request from the phone processing system: in this case, the source grammar node probability was set to 0 (see above).

After the grammar system has computed all the requests of the phone processing system for a certain frame, it has generated a list of active phone instances that has to be processed by the phone processing system in the next frame. To completely specify an active phone instance, this list contains, for every active phone instance, the source grammar node probability, the backtrack tag, the phone instance id, the unique phone id, the topology address, and an address to locate the state probabilities on the phone processing system.

There is no frame delay between the input (destination grammar nodes) and the output (source grammar nodes) of the grammar system. This is because transitions between grammar nodes are null-arcs: a null-arc transition does not have an associated frame delay (see section 4. 3. 2.).

5. 1. 3. Backtracking

The backtracking process is described in section 4. 3. 4. In summary, this process generates a list of backtrack pointers during recognition, and after the incoming speech pauses (at the end of a sentence), the backtracking process reads that list starting from the state with a highest state probability.

To generate this list, the backtrack process writes the tag and the associated phone instance ids of destination grammar nodes that have a high probability (>Th). The phone processing system sends these grammar nodes, and they are the same that are send to the grammar processing system with a request to activate the successors (Figure 24). The backtracking process writes the list, and returns a new tag to the grammar processing system which gets attached to the source grammar nodes of the successor phone instances.

5.2. Hardware Partition

In this section, the functional blocks of the speech recognition system are partitioned into full custom hardware and general purpose hardware. To be able to relate a function to a specific hardware implementation, we examine the requirements of that function.

5. 2. 1. Front End Processing

The first step in front end processing is A/D conversion with anti-alias filtering. This function was implemented using commercially available hardware.

After A/D conversion, the front end dsp algorithm described in (EQ 1) through (EQ 4) in chapter 3, has to be performed within a frame of 10 ms. This algorithm is a typical dsp algorithm, using functions such as the fast fourier transform (FFT) and vector quantization which are well supported by digital signal processors. Thus, it is possible to implement the front end algorithm in real time on a single TMS320C30 DSP. This approach was chosen as a special purpose implementation of these algorithms was determined not worth the effort and would result in inflexibility of a portion of the algorithm that is not well defined.

To ease the task of interfacing the A/D hardware with the DSP, commercial hardware was used that contains - besides anti aliasing filters and the A/D converter - a processor to set up a serial protocol so the sampled data can be directly received by the serial port of the DSP. Furthermore, the hardware can be configured by the DSP for various sample rates (8kHz to 48kHz) and dynamic ranges (up to 16 bits per sample).

5. 2. 2. Phone Processing

In this process, the state probabilities and the backtrack tags, and the destination grammar nodes of active phone instances are computed. In chapter 4 we derived that for a 60,000 word recognition system about 200,000 state probabilities have to be computed within 10 msec (20 million states per second). Other operations in this process include normalization of the state probabilities, comparing the pruning threshold to every state probability, and eventually updating it. Also, the destination grammar node probabilities and backtrack tags of 60,000 active phone instances are computed. For pruning, the phone processing system compares the state probabilities and the destination grammar node probabilities to the pruning threshold to decide if a phone instance should be re-activated for the next frame, or to request activating the successors of a phone instance. It is obvious that these computations are the most critical "inner loop" of the Viterbi algorithm, they are very demanding in terms of computational throughput and memory bandwidth.

To process one state, the phone processing system has to read $A(p,s)$, $P(o_i|s)$ and the phone topology. Also, the state probabilities $P(O_{i-1},s)$ and the backtrack tags $Tag(p, i-1)$ of all the predecessor states have to be read from the state probability memory i-1. In addition to that, the source grammar node probabilities and backtrack tags of active phone instances have to be read from the active phone instance memory on the grammar processing system, as well as the complete specification of this active phone instance (phone instance id, unique phone id and topology) and a pointer into the state probability memory i-1 to locate the state probabilities of its states.

Clearly, this function has to be implemented using a dedicated architecture and full custom processors. A general purpose system would not be capable of processing 20 million states per second, even less capable of accessing within 50 nsec all data that are necessary to process one state. It is because of this critical function that hardware implementations using general purpose components are limited to a small vocabulary.

5. 2. 3. Grammar Processing

The inputs to the grammar processing system are requests from the phone processing system. This can either be a request to re-activate a certain phone instance if it has at least one high state probability ($> \Theta$), or to activate the successors of a phone instance if the destination grammar node has a high probability, or both. A phone instances is activated by adding it to a list of active phone instances which gets processed by the phone processing system in the next frame.

The most demanding computation in grammar processing is the ToActiveWord process since it is activated each time a phone instance has to be added to the list of active phone instances (see Figure 23). Simulations showed that about 5% of the phone instances that are active in a frame cause a request to activate their successors. Thus, out of 60,000 active phone instances, 3,000 requests to activate successors can be generated each frame. If each phone instance has an average of S successors that have a high source grammar node probability, S · 3,000 successors have to be added to the list of active phone instances. Clearly, the ToActiveWord process is definitely very demanding, especially if S is large which tends to be the case as the perplexity[1] of the grammar gets higher. Therefore, it is desirable to support this function with dedicated full custom hardware. The multiplications (additions in the log domain) necessary to compute the source grammar node probabilities $P(S)_{new}^{i}$ (EQ 17) are less demanding,

1. The perplexity Q is an information theoretic measure of a tasks difficulty. It is defined as $Q=2^{H}$, where H is the entropy, or the number of bits necessary to specify the next word using an optimal encoding scheme [Lee89].

but if the grammar perplexity is high, they also cannot be implemented on a single general purpose processor. For each successor, the processor that implements successor computation has to read memory to identify the successor (phone instance id, unique state id, topology address), read the transition probability, compute the source grammar node probability, compare it to the pruning threshold, update the pruning threshold, and send the structure to the ToActiveWord system. Assuming this takes not more than 500 nsec (optimistic), we can process in real time 20,000 successors per frame. Thus, a general purpose system is only feasible for small values of S (in this case, S<6.7).

Even though the successor computation process can be very demanding for grammars with a high perplexity, this was not done because grammar processing algorithms in which researchers are experimenting with various grammar topologies are not stable. Also, natural language processing (chapter 1) makes it necessary to dynamically change transition probabilities between phone instances. For a full custom implementation it is desirable to customize the architecture to a certain grammar topology, and since this topology is not determined yet, it is more feasible to use a general purpose grammar system.

As a result, the ToActiveWord process is implemented using full custom hardware: it is an operation that is not only the most time critical, it is also common across various grammar topologies.

5. 2. 4. Backtracking

The least demanding function in Figure 24 is the backtracking function. To generate the backtrack list, there are 2 memory accesses per incoming destination grammar node tag, one to store the old tag and one to store the associated phone instance. About 3,000 destination grammar nodes have a high probability at the end of a frame (see above), therefore there are about 6,000 memory accesses in 10 msec. Also the backtracking function that reads the backtrack list at the end of a sentence is not

very demanding: there is one memory access per recognized phone instance, thus the whole operation can easily be performed on general purpose hardware.

5. 2. 5. Final Hardware Partition

Figure 24 summarizes the partition of the recognition system into full custom and general purpose boards.

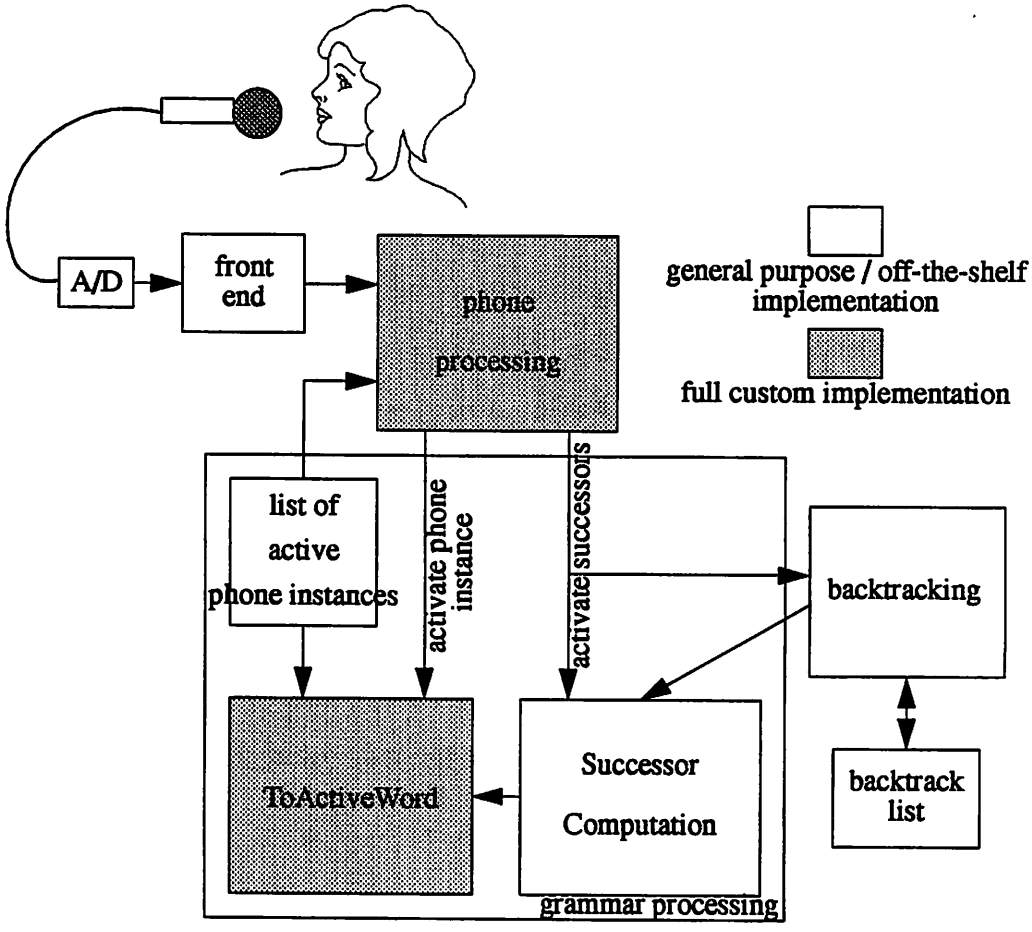


Figure 24: Hardware Partition

After relating functions to full custom and general purpose hardware components, we can now define a physical partition of the system. The most natural way is to separate the full custom functions and the general purpose functions and to implement them on different boards. Following this strategy, Figure 24 shows the resulting hardware partition. The computations for the recognition system are

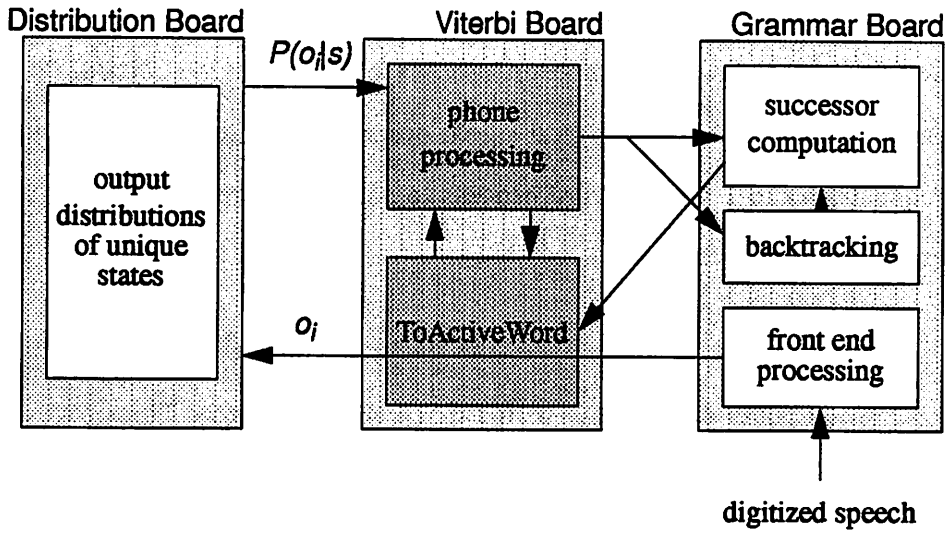


Figure 25: Hardware Partition of the Recognition System

implemented on 3 custom boards: a grammar board that contains digital signal processors for the successor computation, backtracking, and front end processing. The Viterbi board contains full custom hardware for phone processing, and for the ToActiveWord process. The third board is a memory board that contains the output distributions of the states of unique phones.

During recognition, speech is A/D converted and frame for frame, the features o_i are computed using a TMS320C30 digital signal processor that resides on the grammar board. These features are then sent to the output distribution board so that the output probabilities for the active states in that frame can be accessed.

Given these output probabilities and a list of active phone instances from the ToActiveWord process, the phone processing system in the Viterbi board updates the states of the active phone instances. All data needed for these computations (except the list of active phone instances and the output distributions) reside on local memory inside the phone processing system. If, after processing the state probabilities of an

active phone instance, there is at least one state that has a probability higher than the threshold, the phone processing system sends a request to the ToActiveWord system to put this phone instance on the active list of phone instances for the next frame. If the destination grammar node probability is high, the phone processing system sends a request to the successor computation process on the grammar board. This process is implemented on TMS320C30 digital signal processors. It generates a backtrack list and send requests to the ToActiveWord process to add successors to the list of active phone instances for the next frame.

After the end of a sentence, the backtrack process on the grammar board is activated which generates a list of words that were recognized. This process runs on one of the DSPs that implemented successor processing during the sentence. Finally, Figure 24 shows a schematic of the complete speech recognition hardware.

The general purpose microprocessor board (68020 based board by Heuricon) and the ethernet interface board are used to control the system (see Figure 24) and to interface it to the computer network. The Heuricon board runs a real time operating system (VxWorks) and has the driver software necessary to open a process remotely from a workstation. At system start-up, the microprocessor reads the HMM parameters via ethernet from a disk and loads them into the recognition hardware. During recognition and at the end of a sentence, it also has the task to coordinate the data transfer between the boards and synchronize the boards after every frame.

Communication between the boards is done in two ways: through dedicated busses that are implemented using ribbon cables and through the standard VME back plane. Any communication that has a high data rate had to be implemented on dedicated busses since the VME back plane is very band limited (about one read or write cycle every 300 nsec, maximal 32 bits data). These high bandwidth communications are accessing the output distributions from the distribution board, generating requests to the successor computation process, or generating requests to add successor phone

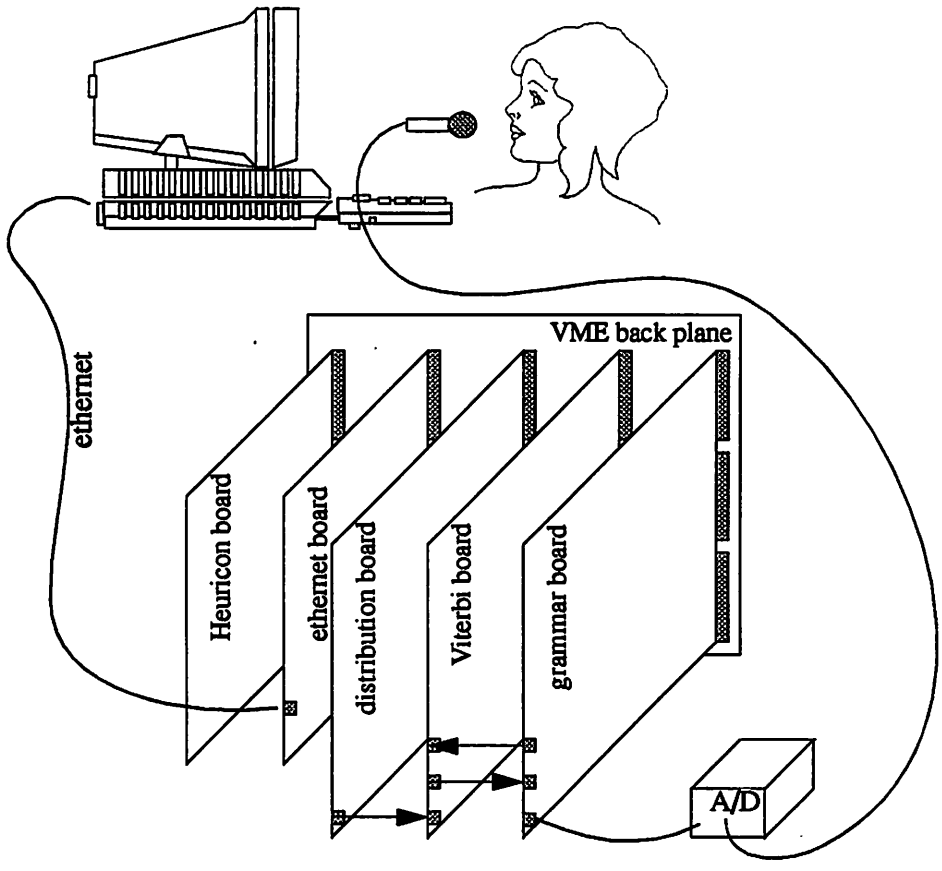


Figure 26: The Complete Speech Recognition Hardware

instances to the list of active phone instances. Communications that use the VME bus are transferring the feature vectors of a frame to the output distribution board, and loading the recognition hardware at system start-up. Also, the Heuricon board uses the VME bus to control the boards, to read local memories and registers on the various boards for debugging purposes, and to load status registers on the boards. For synchronization, the interrupt lines that are provided by the VME back plane are used.

6 The Viterbi Board

This chapter describes the architecture and implementation of the full custom Viterbi board which performs the computations for phone processing and for the ToActiveWord process (chapter 5). It also contains memories to store HMM parameters of unique phones, intermediate results, and the list of active phone instances generated by the ToActiveWord process.

6.1. Phone Processing

The most time critical function in the recognition system is phone processing. As derived earlier, this process has to compute the state probabilities and backtrack tags of 20 million states and 6 million destination grammar node probabilities and backtrack tags per second. Also, it performs normalization and pruning for all states and grammar nodes processed. We also showed in chapter 4 that due to the properties of HMMs that describe phones (locality, left to right transitions), the most feasible implementation of this process is to compute the state probabilities based on predecessor states.

6. 1. 1. Data Access

Figure 28 visualizes the data flow for phone processing. If a state has an

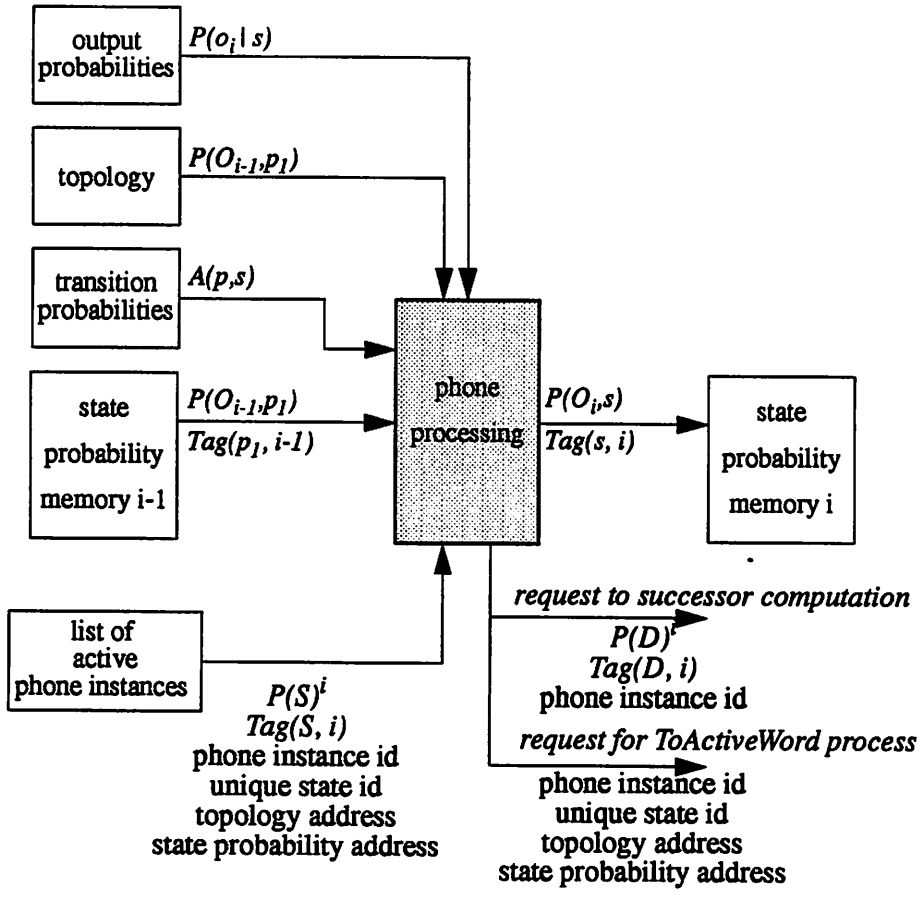


Figure 27: Data Flow Diagram of the Viterbi Process on the Phone Level

average of 3 predecessor states, the following data have to be accessed in order to update the state probabilities:

- First, the phone process reads information associated with an active phone instance from the list of active phone instances in the grammar system. This information contains the probability and backtrack tag of the source grammar node (36 bits), the phone instance id (20 bits), the unique phone id 16 bits), the topology address (4 bits) and a pointer to the state probabilities (16 bits, see chapter 5).

Also, there are flags to indicate if a phone instance has been activated for the first time, and a flag that is set in the highest valid address location of the active list memory to indicate the end of the list of active phone instances. This memory gets accessed sequentially, once per phone instance, yielding 96 bits of data per phone instance.

- for every state in an active phone instance, $P(O_{i-1}, p)$ and $Tag(p, i-1)$ has to be accessed for all the predecessors p of that state. The backtrack tag is coded using 20 bits and the state probability has 16 bits (section 4.2.). We assume that a state has 3 predecessors, thus this access yields $3 \cdot 36$ bits of data for every state that has to be updated.
- $P(o_i \mid s)$, the output probability (12 bits)
- $A(p,s)$, the transition probabilities from the 3 predecessors to the state ($3 \cdot 4$ bits)
- Topology information to identify the 3 predecessors (relative offset, $3 \cdot 3$ bits)

If we assume that a phone instance has an average of 3 states, about 60,000 active phone instances (200,000 active states) have to be processed per frame. The phone processing system has to access a total of 145 bits per state and 96 bits per phone instance. This corresponds to ≈ 34.8 Mbits per frame or 3.48 Gbits per second. In addition, the results of the phone processing operation have to be stored or passed to the grammar processing system:

- $P(O_i, s)$ and $Tag(s, i)$, the state probabilities and Tags of the states processed (36 bits). These results are stored in the state probability memory i, even if the phone instance is not re-activated (pruned).
- $P(D)^i$, $Tag(D, i)$, the state probability and Tag of the destination grammar node (36 bits) and the phone instance id (20 bits). This structure only gets passed after a phone instance has been processed and if $P(D)^i$ is above the pruning threshold.

In this case, a request to activate the successors of that phone instance is issued. In the worst case however, the system must support a long sequence of these requests.

- If there is a state inside the processed phone instance with a state probability that is above the pruning threshold, the phone processing system sends a request to the ToActiveWord system to re-activate this phone instance for the next frame. This request contains the complete specification of the phone instance (20+16+4 bits) and a pointer to the state probabilities in the state probability memory (16 bits).

Thus, the results of phone processing that are stored or passed to other processes, are 36 bits per state and 112 bits per phone instance (worst case). This corresponds to 14 Mbits per frame or 1.4 Gbits per second.

Clearly, the most severe bottleneck in this process is the access of the memories that contain information associated with the predecessor states (state probabilities i-1, transition probabilities and topology). These memories have to be accessed once for every predecessor of a state. The other memories (output probabilities and state probabilities i) only get accessed once per state.

The computation of the Viterbi algorithm and the address computation for the memories are implemented on full custom processors. Therefore, the processor architecture can be pipelined and customized to the Viterbi algorithm to achieve a very high computational throughput. This means the real system bottleneck is memory access.

6. 1. 2. Parallel Processing of Active States

This section describes an architecture that could reduce the memory bottleneck, but it is shown that it requires too much hardware and thus is not feasible. Here, several Viterbi processes operate in parallel and the memories that contain information about the predecessor states can be partitioned so that each process

operates on a subset of the active states. This would result in a linear speedup as we increase the number of Viterbi processes.

Since each Viterbi process can operate on a different state s, the various Viterbi processes have to access different output probabilities $P(o_i \mid s)$ thus increasing the bandwidth to that memory. To partition the output probability memory among the Viterbi processes, it would be necessary to dedicate a certain Viterbi process to only operate on a subset of unique phones: if only a subset of unique phones is processed by a single Viterbi process, the output distribution memory accessed by this process only has to have a subset of the output distributions. To implement this scheme, it is necessary to distribute active phone instances in such a way that they are processed by the according Viterbi process.

Also, the hardware for the memories that contain information about the predecessors (state probability i-1, transition probabilities, topology memory) would increase even though they can be partitioned. Memories are available with address spaces of up to 4M, but the recognition system is specified for a maximum number of 256K active states. Thus, it is important to use memories with a large word length, but 64K word memories (that would be used if the system is partitioned into 4 parallel processes) are not available with wider words than the memories with an address space of 256K. Therefore, partitioning the address space would result in additional memory chips, in this case, 4 times as many.

In conclusion, a subsystem architecture that is based on parallel processing certainly is possible, but it inevitably results in more hardware. It would require too much hardware to process the active states in parallel. So a time multiplexed approach was used to process all active states using one custom process. However, a parallel architecture certainly can be used for an even larger system (multiple person speech server) where serial processing of active states is not fast enough.

6. 1. 3. Serial Processing of Active States

Here, the state probabilities of active phone instances are processed sequentially and the bandwidth to the memories that have to be accessed for every predecessor of a state limits the system performance. To speed up this bottleneck the following techniques were used.

6. 1. 3. a Transition Probabilities A(p,s)

The HMMs of unique phones define the predecessors and the transition probabilities from the predecessors to a certain state. It is possible to arrange the transition probability memory in such a way that one memory access yields the transition probabilities from all predecessors of a state. Typical HMMs that describe phones do not have more than 3 predecessor states for any state in the phone. Therefore, one word in the transition probability memory contains 4 transition probabilities, 3 transitions from predecessor states (here, the source grammar node can be a predecessor) and one transition probability to the destination grammar node. Since these probabilities are coded using only 4 bits (see chapter 4), the width of this memory is 16 bits. Whenever a state has less than 3 predecessors or no destination grammar node transition, the probability in the memory is 0 (1111_{bin} in the log domain).

6. 1. 3. b Topology Memory

The topology memory contains the addresses of predecessor states. Again, this memory can be arranged in such a way that one memory access yields all the necessary address information to process one state.

Typical phone topologies have only local transitions, strictly left to right. Usually there is no predecessor state that has an offset of more than 8 states. Therefore, the most efficient way to store the location of predecessors is using offset addresses relative to the current state. Again, there are not more than 3 predecessors for a state

inside a phone, therefore the topology information of a state can be coded in a single 9 bit wide word (3 · 3 bits).

6. 1. 3. c State Probability Memories

To increase the memory bandwidth to the state probability memory i-1, cache memories were used that store a small subset of the state probabilities, the subset relevant to process a state.

To describe this method, consider the following architecture: if the state probability memory i-1 is replicated 3 times we can simultaneously access the information associated with the 3 predecessors of a state using different addresses for each of these memories. Thus, each predecessor state probability is accessed from a different memory and there is no contention. When the result is computed, it has to be broadcast to the state probability memories that keep the current result (state probability memory i) and stored using the (sequential) address corresponding to that state. Figure 28 shows this architecture that assumes that a state has not more than 3 predecessors.

Using this scheme, it is possible to access all information necessary to process one state probability in one memory cycle. However, the drawback is that the state probability memories have to be replicated 3 times.

To avoid this, we can exploit the fact that state transitions inside phones are local and strictly left to right. Thus, it is sufficient to replicate only a small subset of the state probability memory i-1, the subset that is relevant to update a certain phone. Assuming that a state has no predecessor that has an offset bigger than 8 (no predecessor is more than 8 states to the left of that state), it is sufficient to replicate the subset of 8 states that precede the current state. Since the states are processed and stored sequentially as they occur in the HMM of phones, only the states probabilities

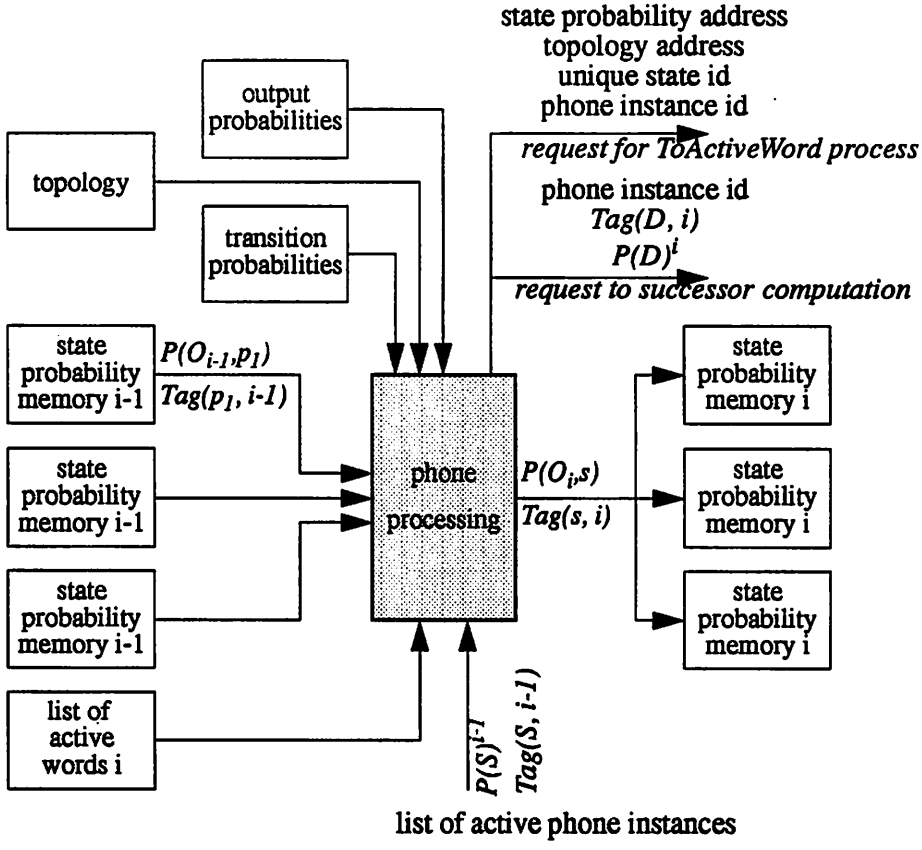


Figure 28: Viterbi Process on the Phone Level using Multiple State Probability Memories

(and tags) that are stored on the 8 neighboring lower address locations have to be replicated. This scheme is shown in Figure 28:

To compute the state probability of state n at frame i, we simultaneously want to access $P(O_{i-1}, l)$, $P(O_{i-1}, m)$ and (O_{i-1},n). This can be done by simultaneously accessing 3 small cache memories that contain these data. Because of the locality of state transitions inside phones, there are no "misses" when that cache memory is read: it is guaranteed that the all predecessors of a state are in that small subset of the state probability memory i-1. As a state is processed and the caches are read, they simultaneously get updated by transferring the state probabilities (and backtrack tags) from sequential addresses of the state probability memory i-1, discarding the content that was longest resident in the caches, which is in effect a circular buffer.

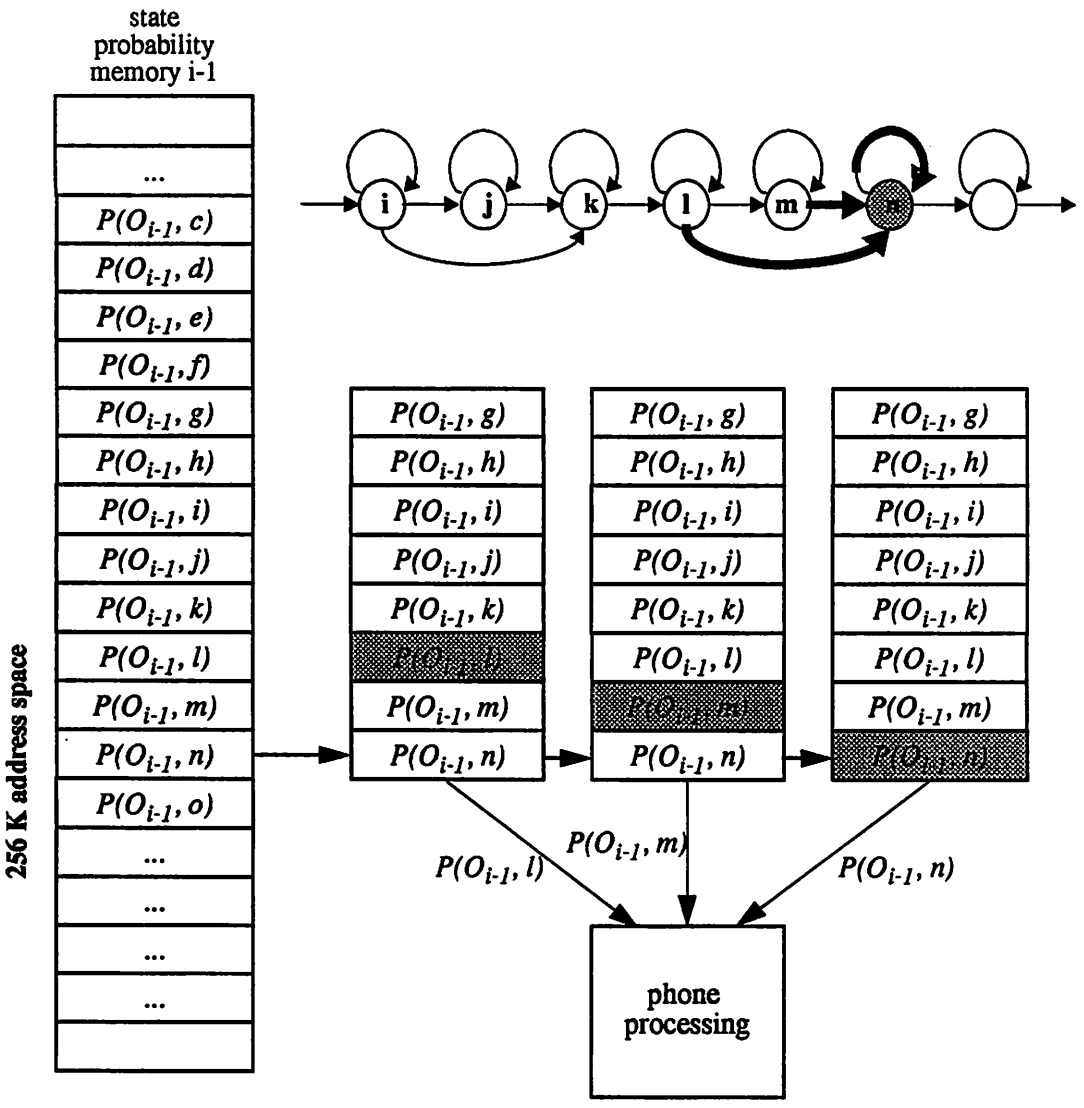


Figure 29: Replication of a Local Subset of the State Probability Memory i-1

6. 1. 3. d Output Probability Memory

Using the above memory architecture, it is possible to access all information associated with a state in one memory cycle: the topology information and the transition probabilities for predecessor states can be accessed using one memory address, while the state probabilities and backtrack tags of predecessor states can be simultaneously accessed from 3 small cache memories. Since the hardware has to process 200,000 states within a frame duration of 10msec, it is necessary to implement memory cycles of

50nsec. In general, there are three ways to achieve a fast memory cycle time: Using a fast, expensive memory technology (static memories), using interleaved memories, or using fast page mode in dynamic memories.

In phone processing, the memory that is most critical to implement this fast access cycle time is the output distribution memory, not only because of its size (80MBytes), but also because it is accessed almost randomly (see below). This memory contains 4 output distributions per unique state, and each output distribution has 256 probabilities. When the memory is accessed, the output probability corresponding to a certain state in a phone instance and a certain set of observations is read. However, the order in which active states are processed is sequential only within a phone instance, but the sequence of phone instances (which are instances of unique phones) is random because of pruning. However, phone instances typically are very small (2-5 states), therefore there are as many random accesses as there are active phone instances in a frame, and techniques like memory interleaving or fast page mode access to increase the memory bandwidth for inexpensive DRAM are not very successful.

Instead, this system uses static memories (SRAM) to achieve the fast cycle time. Since these memories are not very dense, have a high power consumption (4W for a 64Kx32 module), and are fairly expensive, only a small subset of the memories is implemented using this technology. Similar to memory architectures in computer systems, most data reside on inexpensive (but slow) dynamic memories and only the subset that is actually required to process a frame with a certain speech observation is loaded to a fast static memory. This subset is called output probability memory.

The output probability memory contains the output probabilities of all unique states, but only for the set of speech features that was observed in a frame. Thus, instead of storing 4 output distributions, each with 256 output probabilities for 256 different features (output distribution memory), the output probability SRAM contains one probability per unique state. This probability is the product of the 4 probabilities

that were derived from the output distribution memory for that unique state and for the set of 4 speech features o_t that were observed in that frame (EQ 8).

The data transfer from the DRAM output distribution memory to the SRAM output probability memory can be done sequentially. For that, the output distribution memory is partitioned into 256 blocks for each speech feature, each block corresponds to a certain speech feature vector (observation) and it contains the output probabilities of all unique states for this particular observation. Thus, to load the SRAM output distribution memory, we simultaneously access 4 different blocks corresponding to 4 observations of different speech features and sequentially read the probabilities for all unique states, add them (log domain) and store the result on the output probability SRAM. This data transfer has to happen before the frame that corresponds to these observations is processed.

Using the memory architecture described above, it is possible to access all data necessary to process an active state within one memory cycle. All memories that are accessed by the phone processing system are implemented with fast static memories that allow an access cycle time of 50ns. Using this scheme, it is possible to sequentially compute 200,000 state probabilities in one frame.

6.2. ToActiveWord Process

The ToActiveWord Process has to generate and maintain a list of active phone instances for the next frame. For that, it implements the MAX operation for the successor based Viterbi process on the grammar level by updating the source grammar node probabilities of successor phone instances (successor implementation).

6. 2. 1. Active Phone Instance Memory

The purpose of the active phone instance memory is to store a set of active phone instances that has to be processed in the next frame by the phone processing system. This list is necessary because of pruning: only phone instances with a high source grammar node probability or with at least one state with a high state probability are processed on the phone level (see section 4. 3. 3.).

The recognition system has two active phone instance memories: one that contains the active phone instances that have to be processed in the current frame and another list that is being generated for the next frame. In the next frame, the memories are swapped: the one that was generated in the previous frame now gets processed while the other one gets re-built. There are two sources that can add a new active word onto the active word memory, the phone processing system and the successor computation process (see section 5. 2. 3.).

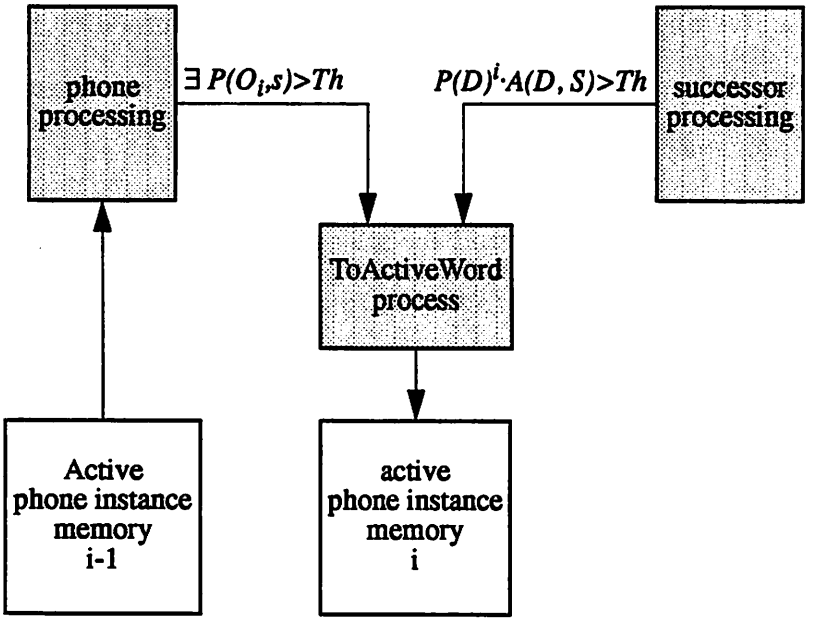


Figure 30: Sources for Requests to the ToActiveWord Process

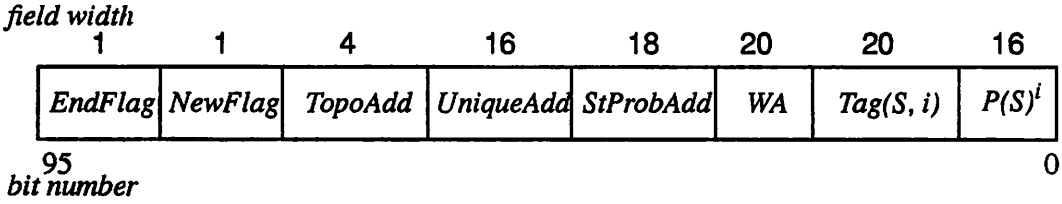
The phone processing system compares the state probabilities of all states in an active phone instance to the current pruning threshold after the states were processed. If there is at least one state that has a state probability that is higher than the pruning threshold, this phone instance has to be processed again in the next frame. In this case, the phone process generates a request to re-activate this phone instance. The other type of request comes from the successor computation system if the probability $P(S)^i = P(D)^i \cdot A(D,S)$ of a successor source grammar node is higher than the pruning threshold. In this case, the successor phone instance with source grammar node probability $P(S)^i$ has to be added to the active phone instance memory if it is not yet on that list. However, if this successor phone instance was already added to that list, the source grammar node probability of that successor has to be updated based on (EQ 18):

$$P(S)^i_{new} = MAX[P(S), P(S)^i_{old}] \qquad \text{(EQ 18)}$$

The active phone instance memory has to store information about each phone instance so that its states can be updated in phone processing. Figure 31 shows the content of this memory for a phone instance.

All the necessary information about an active phone instance is contained in a single address location. $P(S)^i$ and $Tag(S, i)$ are required in phone processing for states that have a transition from the source grammar node. *PI*, *UniqueAdd* and *TopoAdd* specify the active phone instance: *PI* is the phone instance identification, *UniqueAdd* specifies the address of the corresponding unique phone, and *TopoAdd* is required to specify the topology of the unique phone.

It is also necessary to specify *StProbAdd* if the phone instance was already active previously. In this case, the Viterbi process has to access the state probabilities $P(O_{i-1},p)$ that were computed in the previous frame. *StProbAdd* specifies the address of the state probability of the first state, and since the state are processed sequentially, the



$P(S)^i$	source grammar node probability
Tag(S, i)	source grammar node tag
PI	phone instance identification
StProbAdd	address in the state probability memory
UniqueAdd	address of the unique phone
TopoAdd	address in the topology memory
NewFlag	Flag that indicates phone instance is new on that list
EndFlag	Flag that indicates end of active phone instance list

Figure 31: Content of the Active Word Memory

other states of that phone instance are in the next higher address locations. The number of states in this phone instance is specified in the topology memory, and this information can be accessed using the *TopoAdd* field. *NewFlag* specifies if the active phone instance was already active (*NewFlag=0*) in the previous frame or if it is active the first time (*NewFlag=1*). In the latter case, *StProbAdd* is meaningless since all the predecessor state probabilities are assumed to be 0. Therefore, the access of the state probability memory i-1 is controlled by *NewFlag* and *StProbAdd*.

Finally, the active phone instance memory contains a bit that indicates the end of the list of active phone instances. If the phone processing system reads this bit, it can initiate process the next frame since all active phone instances in this frame were processed.

6. 2. 2. ToActiveWord Process

Figure 30 shows how the ActiveWord Process is embedded into the phone processing system and the successor computation process. It receives requests to add a

new active phone instance to the active phone instance memory from both of these systems and has to update the active phone instance memory accordingly.

However, this operation is not just a memory write operation on successive write addresses: as described earlier, there might be several requests to add the same phone instance to the active word memory. For example, a phone instance might have been already added to the list of active words due to a request from the phone processing system, when the grammar system requests to activate the same phone instance because a predecessor contributes to a high source grammar node probability. Or, a phone instance that has many predecessors which have a high destination grammar node probability will cause multiple requests to be put on the list of active phone instances. Since the Viterbi algorithm on the grammar level is based on successor processing, these requests can arrive at arbitrary times.

To implement the Viterbi algorithm correctly, it is necessary to merge transitions that terminate in the same phone instance. This means a specific phone instance can only be on the list of active phone instances once. Therefore, if the ToActiveWord process gets a request for certain phone instance, it first has to find out if this phone instance is already on the active phone instance memory. If so, there are three possible actions:

1. If the NewFlag bit is 0, then the phone instance is on the active phone instance memory due to a request from the phone processing system. Also, since the phone processing system generates only one request per phone instance (it only processes a phone instance once), the current request comes from the successor computation process. In this case, the ToActiveWord process has to update the source grammar node probability and backtrack tag according to (EQ 18), but StProbAdd and NewFlag may not be altered.

2. If the NewFlag bit is 1, then all previous request came from the successor computation process. If the current request was originated by the phone processing system, the NewFlag has to be set to 0 and StProbAdd inserted. However, the source grammar node probability and backtrack tag may not be changed since the phone processing system does not contribute to a source grammar node probability.

3. Finally, if NewFlag is 1 and the current request comes from the successor computation system, the source grammar node probability and backtrack tag has to be updated according to (EQ 18), and NewFlag remains 1. In this case, the StProbAdd field is meaningless.

These actions of the ToActiveWord system are summarized in Figure 32:

	new request from phone processing system for the same phone instance	new request from successor computation system for the same phone instance
phone instance in memory was requested from phone processing system	not possible	- update prob and tag of source grammar node - NewFlag = 0 - don't change StProbAdd
phone instance in memory was requested only from successor computation system	- don't change prob and tag of source grammar node - NewFlag = 0 - insert StProbAdd	- update prob and tag of source grammar node - NewFlag = 1 - StProbAdd meaningless

Figure 32: Actions of the ToActiveWord System if Phone Instance was Already Activated

If the ToActiveWord system gets a request for a phone instance that was not yet activated, it inserts the following structure into the active phone instance memory:

1. If the request came from the phone processing system (re-activate), the phone instance specification (WA, UniqueAdd and TopoAdd) and StProbAdd are inserted using the next available successive memory address. NewFlag is set to 0. In this case, the source grammar node probability is set to 0.
2. If the request came from the successor computation system, the ToActiveWord process inserts the phone instance specification, the source grammar node probability and backtrack tag, and sets NewFlag to 1. In this case, StProbAdd is meaningless.

If the ToActiveWord system gets a flag from the successor computation system that indicates that all requests have been processed, a dummy structure (all ones) is added to the next available successive memory location in the active phone instance memory. In this structure, the EndFlag is set to 1.

To determine if a certain phone instance is already on the list of active words, the ToActiveWord process uses a list that, if accessed using the phone instance identification as address, yields an address for the active phone instance memory (this list will be called ActiveList memory). To update the phone instance, the active phone instance memory gets accessed using this address and the phone instance that was already active gets updated. If the phone instance is not yet on the list of active words, the content of the ActiveList memory is 0. In this case the new active phone instance will be appended to the active phone instance memory and this new address stored in the ActiveList memory.

For this scheme, it is important that the ActiveList memory is cleared before a new list of active phone instances gets generated. If the active phone instance memory is empty, there should only be "0" in every location of this memory. To implement this,

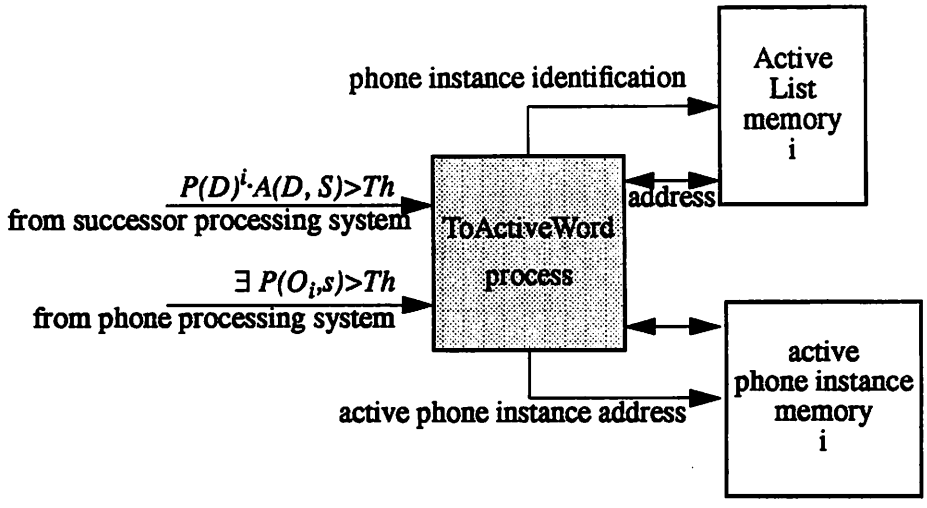


Figure 33: Generation of the Active Word List

I use two ActiveList memories, i-1 and i: while the ActiveList memory i is used to build the active phone instance memory i, the ActiveList memory i-1 gets cleared as the phone processing system sequentially reads phone instances from the active phone instance memory i-1. Only the memory locations that correspond to the phone instances read from the active phone instance memory are cleared as they are read from the active phone instance memory (the other ones don't have to be cleared). After the active phone instance memory i was generated at the end of a frame, the memories get swapped such that memories i become memories i-1 and memories i-1 becomes memories i. Figure 30 shows the corresponding block diagram:

6.3. Board Architecture

The Viterbi board contains the two processes and the associated memories that were discussed above: phone processing and the ToActiveWord process. Figure 35 shows how these processes communicate with each other and the memories.

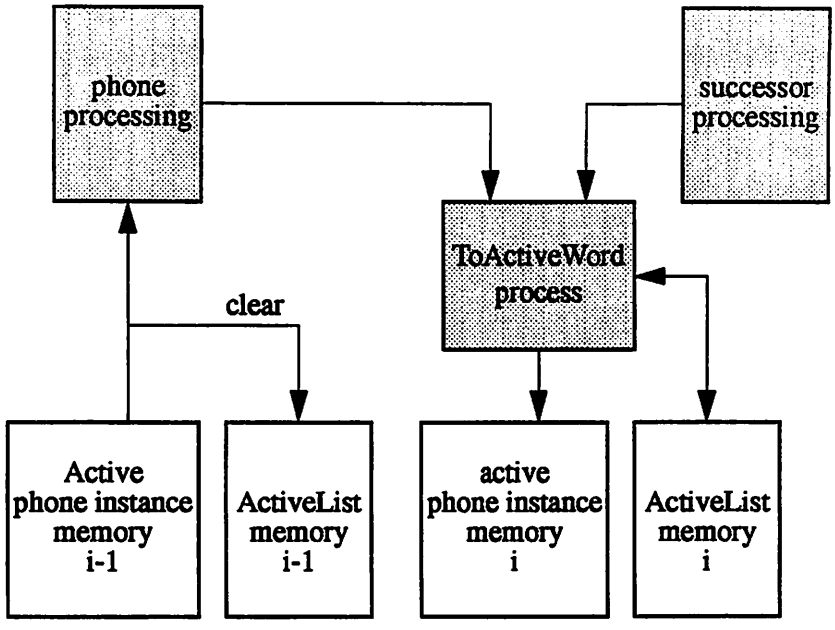


Figure 34: Clearing the ActiveList Memory

The phone processing system sequentially reads active phone instances that are stored on the active phone instance memory i-1 and processes their states. For that, the state probability memory i-1, the transition probability memory, and the output probability memory have to be read. The phone processing system also has to read the topology memory, but since this is a very small memory (128 words), it is implemented inside the phone processing system. As the active phone instance memory i-1 is read, the address location in the ActiveList memory that corresponds to the phone instance id gets cleared. If, after phone processing, a state in a phone instance has a high state probability, the phone processing system sends a request to the ToActiveWord system to re-activate that phone instance. If a phone instance has a high destination grammar node probability, the phone processing system sends a request to the successor computation process to activate the successors of that phone instance. The successor computation process is implemented on another board (see 5. 2. 5.).

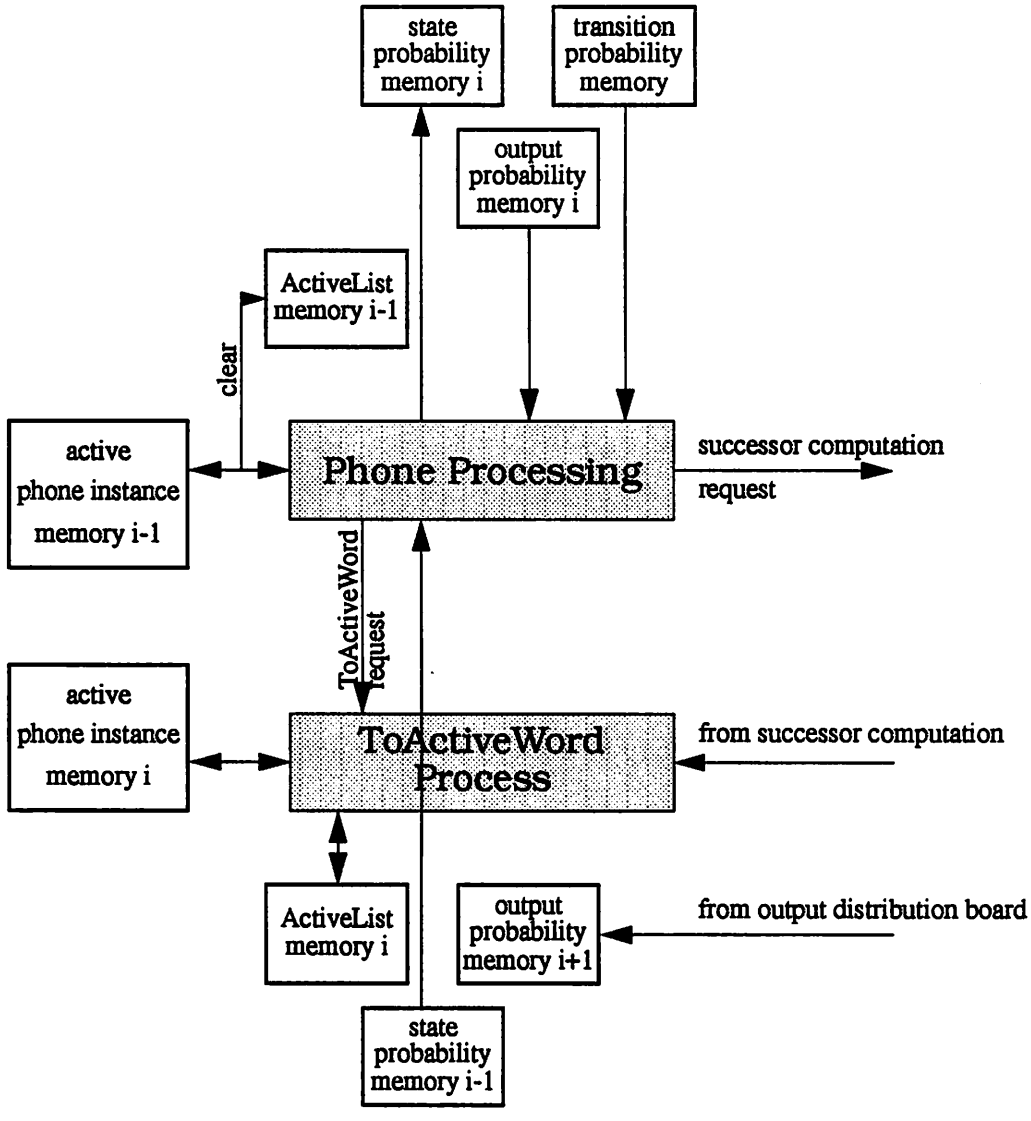


Figure 35: Processes and Memories that were implemented on the Viterbi Board

The task of the ToActiveWord process is to generate a new list of active phone instances for the next frame. It receives requests from the phone processing system and from the successor computation system to add a certain phone instance to that list. To find out if this phone instance is already on that list, the ToActiveWord process reads the ActiveList memory using as address the phone instance id. This yields a pointer into the active phone instance memory. If this pointer is 0, the phone instance is not yet on the list of active phone instances, and it will be added using the next unused sequential

address location. This address is then written into the ActiveList memory for further reference. However, if the pointer that was accessed from the ActiveList memory is not 0, the phone instance is already in the active phone instance memory. In this case, the phone instance is updated according to Figure 32.

While the phone processing system and the ToActiveWord system are busy, a new set of output probabilities is loaded into the output probability memory i+1. These probabilities correspond to the values $P(o_{t+1} \mid s)$ for all unique states s and the observation in the next frame, o_{t+1}.

6. 3. 1. Switching Processor Architecture

This section describes an architecture that was used for the Viterbi board. It a redundant set of processors that get activated alternately to eliminate discrete multiplexors that would have to be used to switch memory busses.

The phone processing system has completely processed a frame if it reads the EndFlag from the active phone instance memory (see 6. 2. 1.). In this case, no more requests will be issued to the successor computation system and to the ToActiveWord system which is indicated with a status bit. Thus, the successor computation system has finished after it processed all requests and the status bit is set. In this case, the successor computation system asserts a flag to the ToActiveWord system.

The ToActiveWord system has finished a frame after it received the flag from the successor computation system and the status bit from the phone processing system. If, in addition to that, all output probabilities for the next frame have been transferred to the output probability memory i+1, the system can proceed to the next frame. To be able to process the next frame, the active phone instance memories and the state probability memories have to be swapped so that the i-1 memories become the i memories and vice versa. Also, the output probability memory i+1 now becomes output probability memory i and vice versa. This means, the address and data busses of the

active phone instance memories have to be multiplexed so that the memory that was connected to the ToActiveWord system now is connected to the phone processing system (and vice versa), and the state probability memory that was read by the phone processing system now has to be written (and vice versa), and the ActiveList memory that was used to generate the active phone instance memory now has to be cleared (and vice versa).

The total number of address and data bits that have to be multiplexed is (96+16) · 2 (active phone instance memories) + (36+18) · 2 (state probability memories) + (20+16) · 2 (ActiveList memories) + (12+16) · 2 (output probability memories) = 460 bits. To multiplex these busses, they have to be duplicated and fed to the multiplexer input pins. This multiplexing operation requires a total of 460 · 2 = 920 multiplexer input pins and 460 output pins. A hardware implementation of this scheme would consume a lot of board area: if the multiplexers were implemented using TTL logic (octal 74LS604), there would be 58 chips required.

To avoid the switching of memory busses, I use an architecture that switches processors (see Figure 36). For that, a second set of processors which implement a second phone processing system and a second ToActiveWord process is added to the board. For a certain frame, only one set of processors is active (for example, the shaded processors in Figure 36). The other set is idle with its output pins floating. After the shaded processors finished a frame, they get de-activated (output pins are floating) and the other set of processors, which is connected to the memories as required for the next frame, gets activated. Thus, there are no additional pins for multiplexers required since the multiplex operation is done using tristate pins in the processors.

The phone processing system and the ToActiveWord system are implemented with 3 full custom processors each. Thus, the additional hardware overhead for the redundant processors is 6 custom chips.

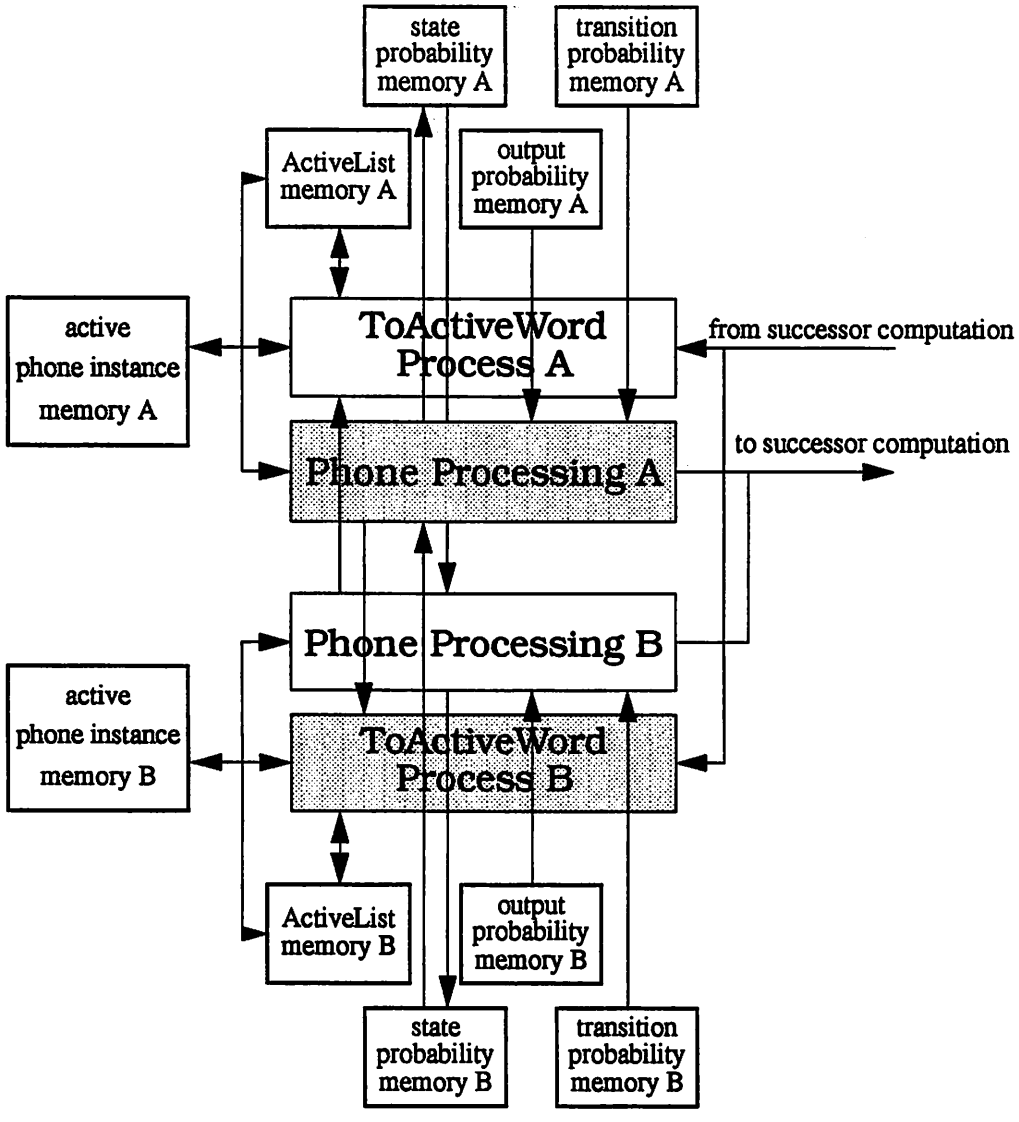


Figure 36: Switching Processor Architecture

Another advantage of this architecture is that it is symmetric if the transition probability memory is duplicated (all the other memories already had to be duplicated). This can be seen in Figure 36: the Viterbi board consists of two identical halves, and therefore it is less effort to implement that board since the design complexity is roughly cut in half. The transition probability memory is fairly small (64Kx16), thus a duplication of this memory does not add a lot of additional hardware.

6. 3. 2. VME Access to Memories

The switching processor architecture eliminated multiplexers which otherwise would be needed to switch memory busses between processors. However, it is desirable that the memories on the system can also be accessed by the VME host processor (see 5. 2. 5.)because the memories have to be loaded with initial data at system start-up and the ability to read and write memories on the Viterbi board it is desirable for debugging and testing. Thus, the memory busses have to be routed and multiplexed to a VME interface that handles the communication between the host and the Viterbi board. This again would result in a large number of multiplexors that would have to be implemented using discrete components.

To avoid this, I implemented VME interface controllers on the full custom processors that are already connected to the corresponding memory busses. These controllers use the VME address bus to decode commands like reading or writing certain memories or register locations. To access a certain memory, the VME host processor identifies the full custom processor chip that is connected to that memory and specifies a memory operation using the lower 6 address bits of the VME address bus. The VME data bus contains the memory address which is loaded to the address register of the corresponding memory. If the memory operation is a write operation, a second VME cycle supplies the data to the corresponding processor which then writes the data into the memory. If the access was a read operation, the processor reads the memory and writes the data onto the VME bus.

Figure 37 shows this architecture: the VME bus is only connected to the full custom processor chips that implement phone processing and the ToActiveWord process. These processes then use the memory busses to access the various memories.

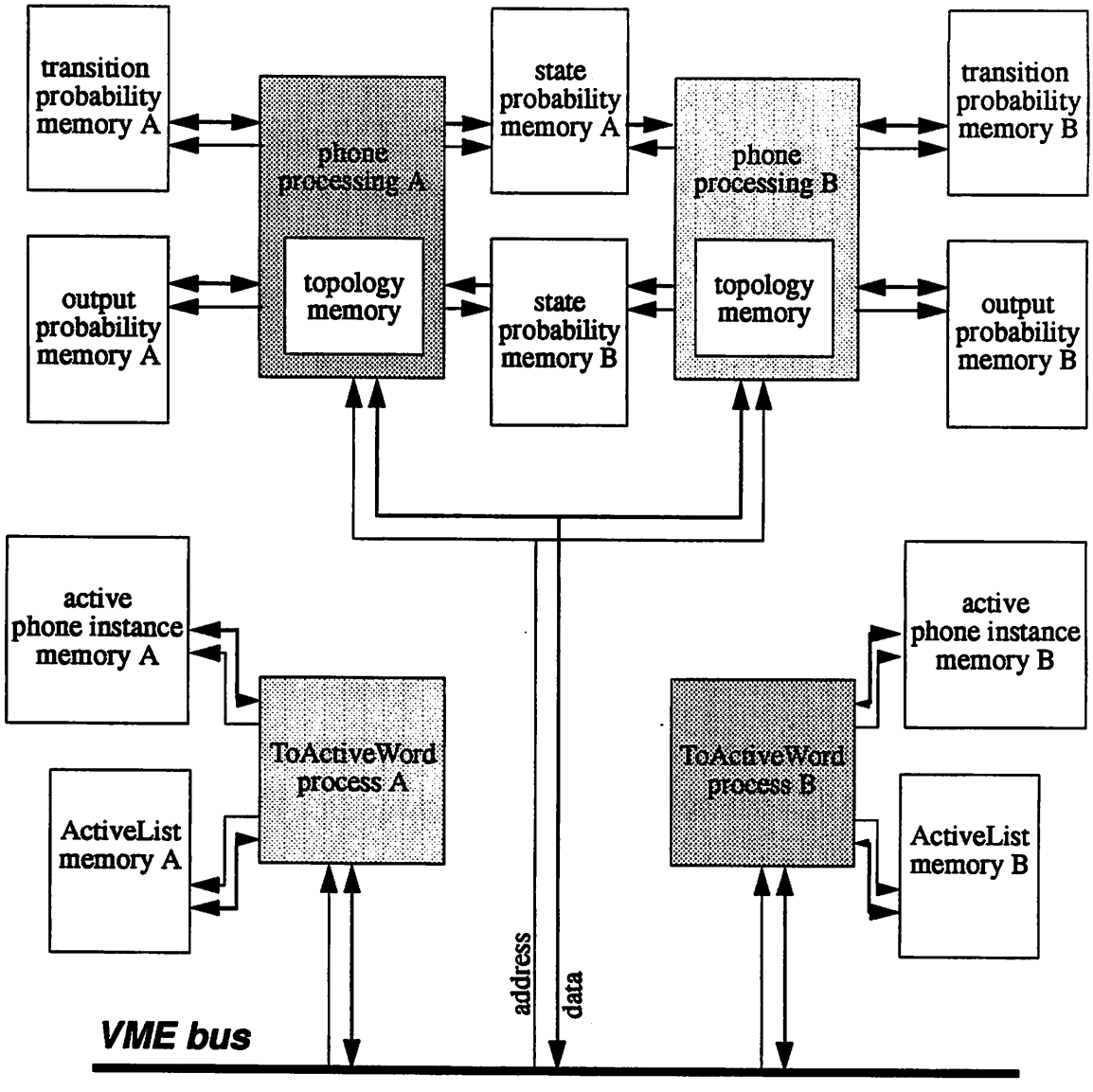


Figure 37: VME Host Access to the Viterbi Board Memories

6. 3. 3. The VLSI Phone Processing System

The phone processing system computes the state probabilities and backtrack tags of active phone instances. For a 60,000 word recognition system, it is necessary to process one state every 50 nsec.

The most critical bottleneck in phone processing is to access the state probabilities and backtrack tags of predecessor states from the state probability memory. In section 6. 1. 3. c, we derived an architecture that eliminates the bottleneck using 3 copies of a small local subset of this memory. This subset - consisting of only 8 memory locations - is stored on the VLSI processors that implement phone processing, and sequentially transferring data from the state probability memory to these local copies updates this subset. Thus, the bandwidth to the state probability memory i-1 is 36 bits per 50 nsec (16 bit probability and 20 bit backtrack tags). The topology memory i implemented on the processors because it is sufficiently small (128 words).

6. 3. 3. a Processor Partition

The phone processing system has to be implemented on three different VLSI processors because of the large number of pins necessary to access data. This section lists the busses at the periphery of the phone processing chips and explains how phone processing was partitioned into 3 chips.

The total number of bits that have to be accessed from external memory is 36 (state probability memory i-1) + 12 (output probability) + 16 (transition probabilities). In addition, the result is stored on the state probability memory i. Since these data are read once every cycle, each of these memories has its own bus to the processors. Also, to address these memories, we need 18 bits (times 2) for the state probability memories and 16 bits for the output probability memory. Since the address for the output probability memory is the unique phone address, it can be shared with the address to the transition probability memory. In total, the chipset requires 152 pins (not including control pins) to interface to the memories.

There are also data associated with active phone instances. When the phone processing system reads the active phone instance memory, it accesses 96 bits (see 6. 2. 1.). These data are used to update the state probabilities (source grammar node

probability, backtrack tag and state probability address) and to identify the phone instance (phone instance id, unique phone address and the topology address). If, after processing that phone instance, the phone processing system generates a request for the ToActiveWord system or the successor computation system, it again passes data associated with that phone instance to these systems: 58 bits to the ToActiveWord process (phone instance id, unique state address, topology address and state probability address) and 56 bits to the successor computation system (source grammar node probability and backtrack tag, phone instance id). This adds up to 190 bits (the phone instance id is send to both, the ToActiveWord process and the successor computation, therefore it can be shared).

In total, the phone processing system has 342 pins dedicated to data and address busses for memories and the other processes. Since the packaging technology that was available for this project supported up to 208 pins per chip, it was necessary to partition the process into more than one full custom chip. Thus, the partition was done according to Figure 38.

This partition is based on the various functions of the phone processing system: the Viterbi processor computes the state probabilities, the destination grammar node probabilities, and performs normalization and pruning. In addition to that, it contains the controller for the phone processing system. The backtrack processor computes the backtrack tags of states and destination grammar nodes and buffers the phone instance id of the active phone instances that are processed. Finally, the address processor computes the addresses that are needed to access the various memories on the phone processing system.

This partition minimizes the number of connections between the chips: the only signals that have to be distributed among the processors are a few control signals. There are no data or address busses that have to be fed to more than one processor.

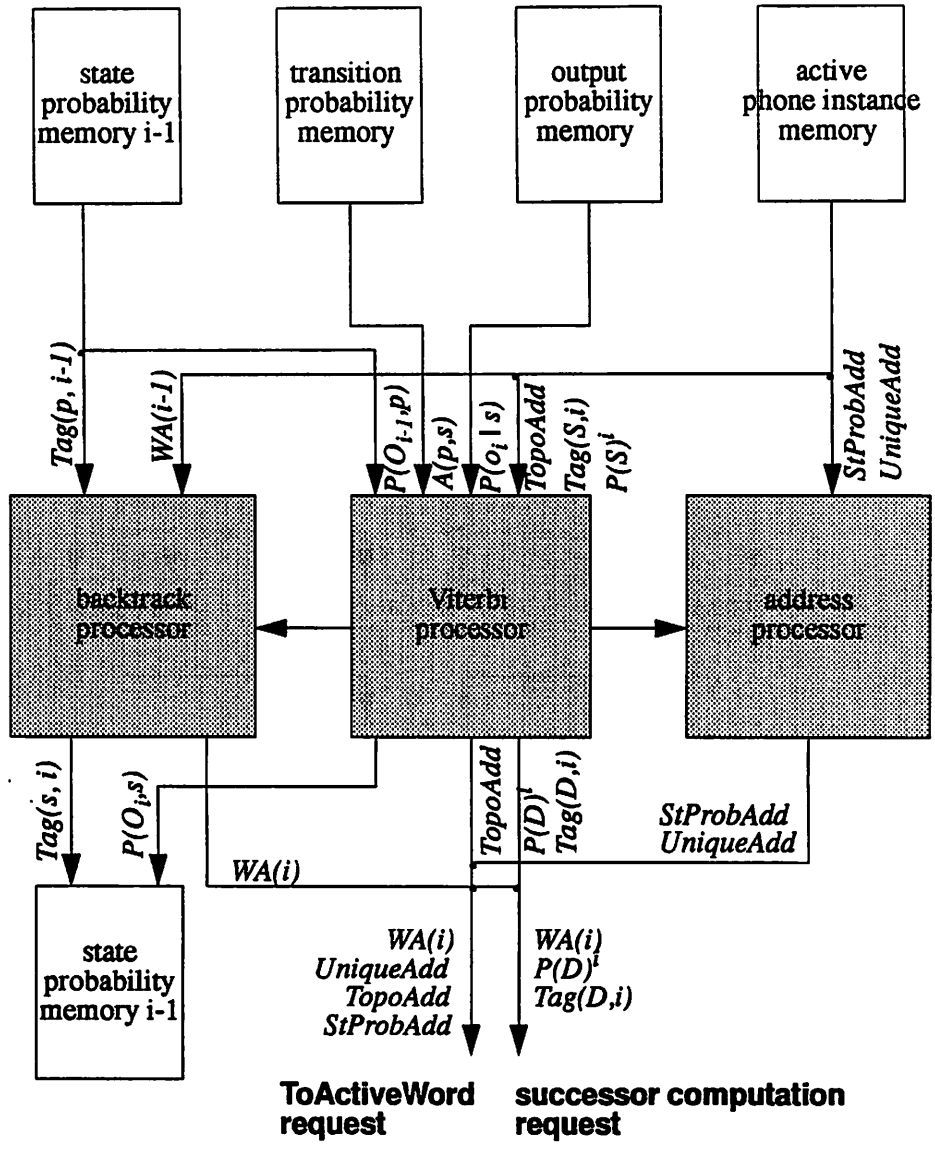


Figure 38: Viterbi Process Chip Partition

6. 3. 3. b Chip Architecture

The phone processing system uses a pipelined architecture (Figure 39) to implement a throughput of one state probability computation per clock cycle. The chips have to process 20 million states per second, therefore it is necessary to compute one state every 50 nsec, which is the cycle time of the static memories that are accessed by the phone processing system. To make the design simple, I decided to use the same

cycle time to clock the processors; this means, we have to process one state per clock cycle.

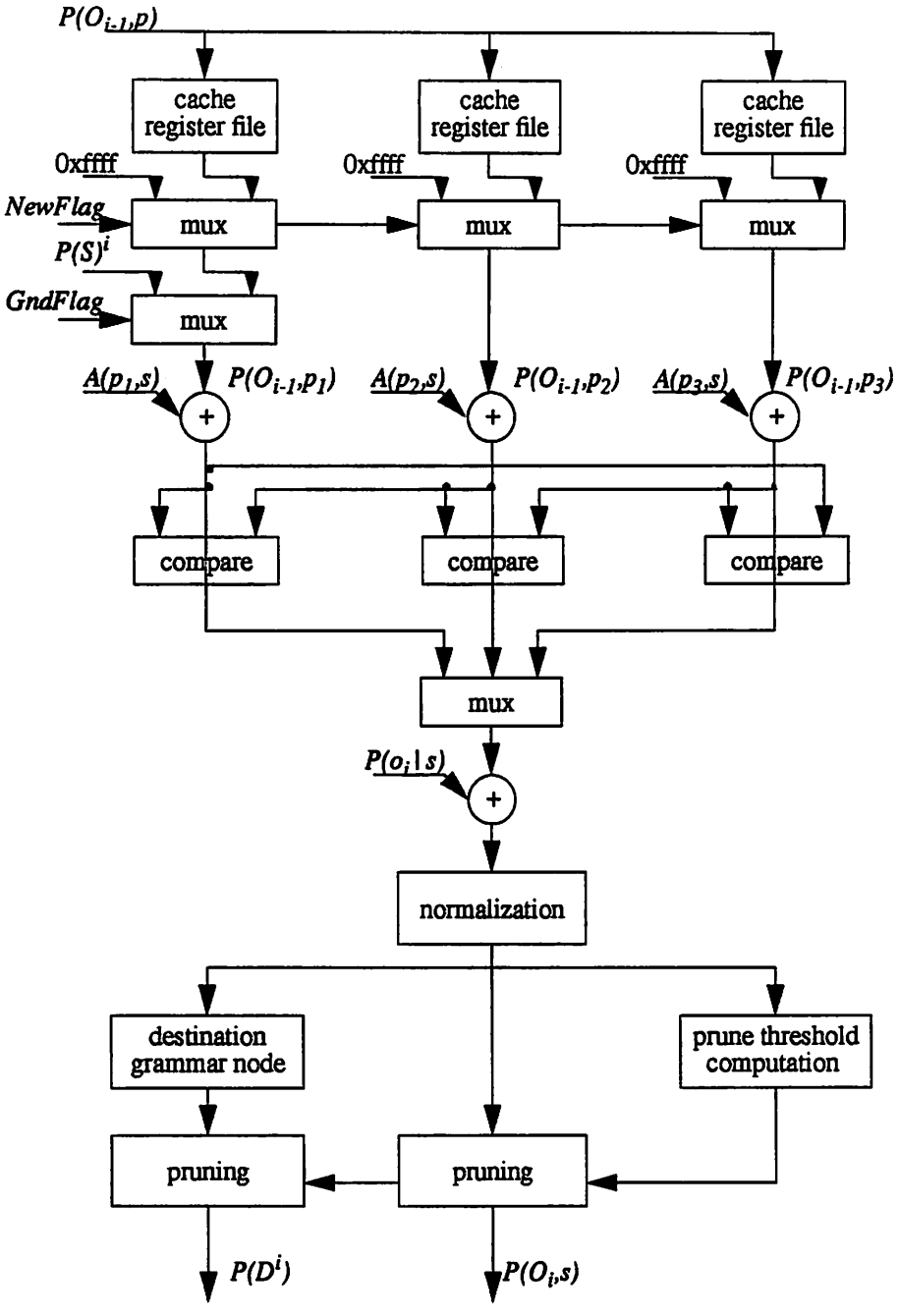


Figure 39: Architecture of the Viterbi Processor

The Viterbi processor sequentially computes the state probabilities and destination grammar nodes of active phones. For the computation of the inner loop,

($P(O_{i-1}, p) \cdot A(p,s)$), there are three identical pipelined datapaths that work in parallel (see Figure 39). Each of these datapaths accesses the predecessors state probability, $P(O_{i-1}, p)$, from a register file that keeps the state probabilities of the 8 states closest to the state that is currently processed (see section 5. 2. 2.). To identify the predecessors of a state, the Viterbi processor has an internal topology memory that contains the relative offsets to the predecessor states. This offset is used to address the register file. The topology memory also indicates if a state has a transition from the source grammar node, and in this case, one datapath accesses the source grammar node probability (which was read from the active phone instance memory) instead of a state probability from the register file. The result $P(O_{i-1}, p) \cdot A(p,s)$, that yields the highest probability is then selected and, to compute the state probability, the output probability is added. The processor then normalizes the state probability (EQ 13) and sends it off chip to be stored in the state probability memory i. The processor also computes destination grammar nodes, the current pruning threshold, and compares the state probabilities to that threshold to make pruning decisions (re-activate a phone instance or request to activate successors).

A chip layout of the Viterbi processor is shown in Figure 44. The chip was implemented using a 23µm CMOS process (Orbit). It has about 50,000 transistors, 204 pads, and has a chip area of 11.4mm x 12.1 mm.

The task of the backtrack processor is to copy the backtrack tag of the predecessor that yields the highest state probability. To have these backtrack tags readily available, the backtrack processor - like the Viterbi processor - has 3 register files that keep the backtrack tags of the states closest to the current state and a copy of the topology memory to access the register files. If a state has a transition from the source grammar node (indicated by the topology memory), the backtrack tag of the source grammar node probability is fed into one of the datapaths. After the 3 tags of the predecessors have been accessed, they are delayed in the same way the corresponding state probabilities are delayed in the pipeline of the Viterbi processor. In the same

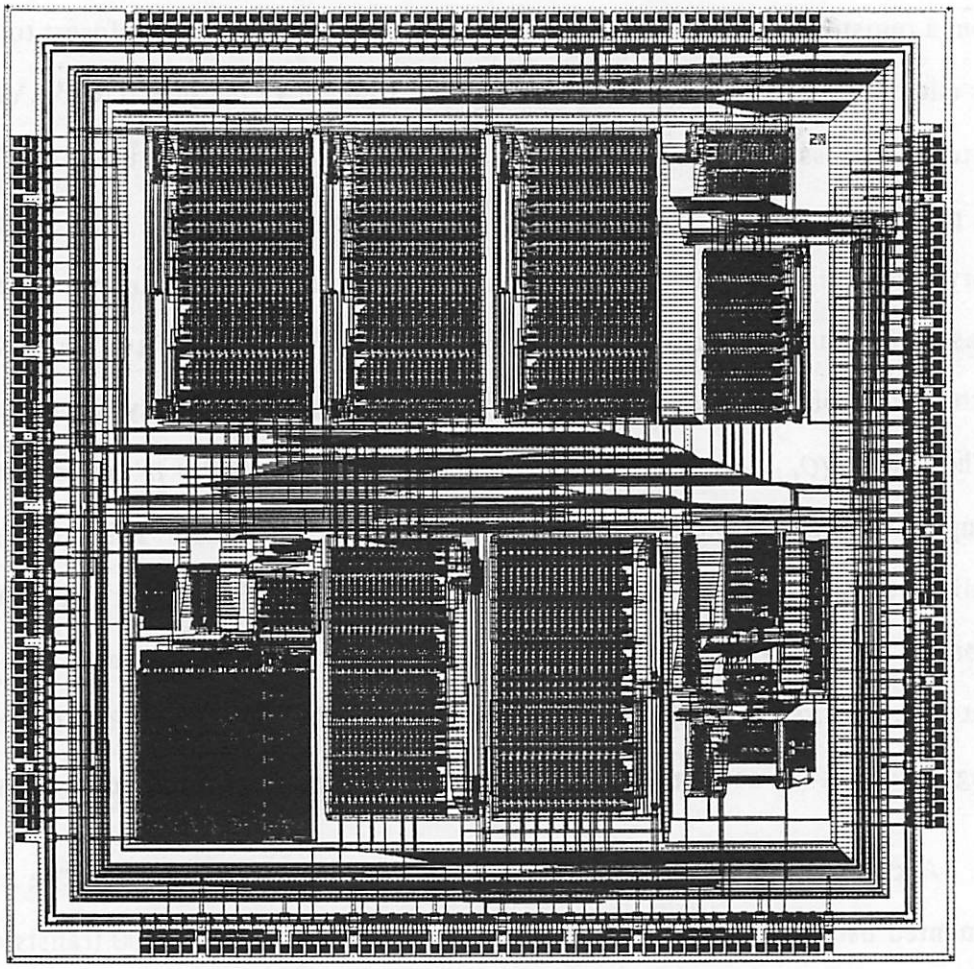


Figure 40: Layout of the Viterbi Processor

pipeline stage where - in the Viterbi chip - a multiplexor selects the best probability, the backtrack processor selects the associated backtrack tag. For that, the control signals that control the multiplexor on the Viterbi processor are routed to the backtrack processors. The same scheme is used to select the backtrack tag for the destination grammar node: whenever the Viterbi processor selects a transition from a state to the destination grammar node, the corresponding backtrack tag is selected on the backtrack processor. Figure 39 shows the datapath and Figure 44 the layout of the backtrack processor.

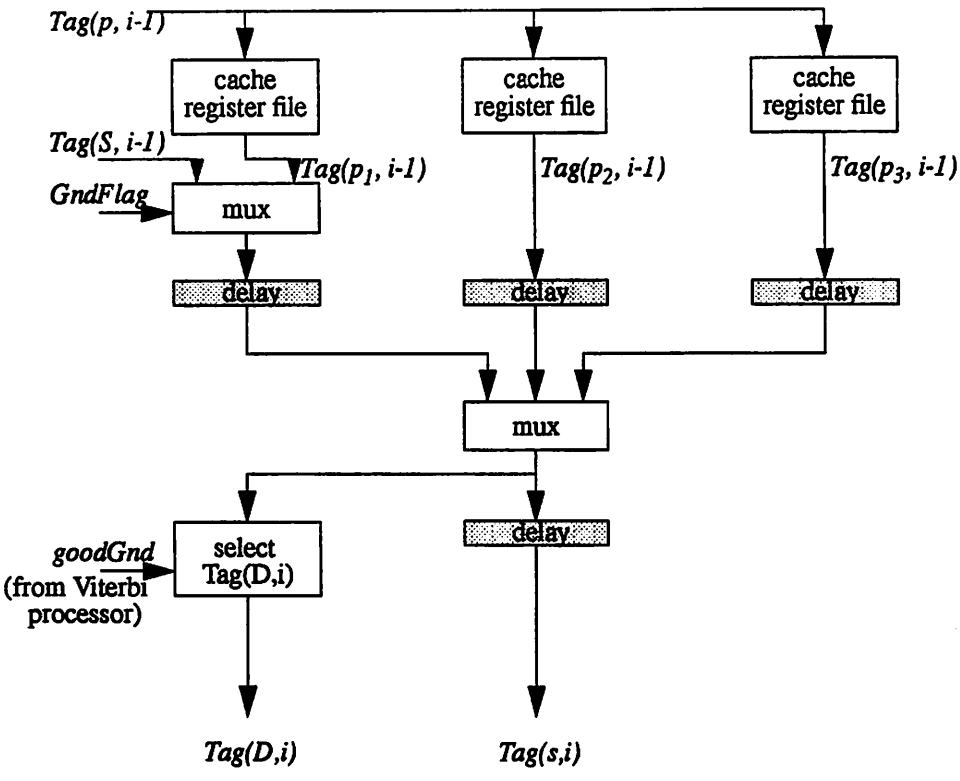


Figure 41: Architecture of the Backtrack Processor

The third chip in the phone processing system is the address processor. This chip computes all addresses that are needed to access the various memories on the phone processing system. Figure 39 shows the datapaths and Figure 44 the layout of this chip.

The address processor generates the addresses for the state probability memories i-1 and i, the topology and output probability memories, and the active phone instance memory i-1. The datapaths used for these address computations, however, are identical: They implement counters that can be set and incremented individually.

Since the phone processing system sequentially computes the phone instances in the active phone instance memory, the address for the active phone instance memory is the output of a counter. This counter is incremented each time an active phone

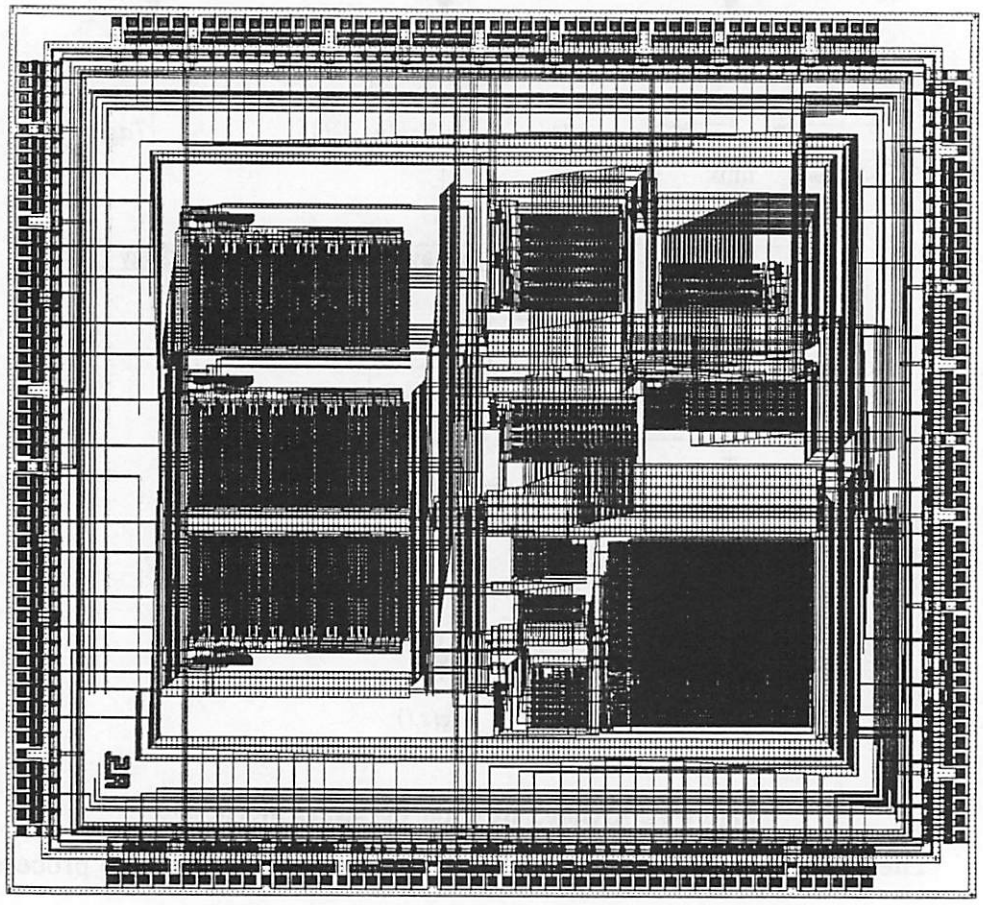


Figure 42: Layout of the Backtrack Processor

instance has been computed. At the start of a frame, the counter is set to 1 (address 0 is not used since a 0 in the ActiveList memory corresponds to a nil pointer).

The address to the state probability memory i-1 is set to *StProbAdd* when a new phone instance is read from the active phone instance memory (*StProbAdd* is provided by the active phone instance memory). This address points to the state probability of the first state in that phone instance, and the other states are stored in the succeeding memory locations. Therefore, the addresses corresponding to the states in the phone instance other than the first state are computed by incrementing *StProbAdd* once every clock cycle. Processing the phone instance is finished when the topology memory inside

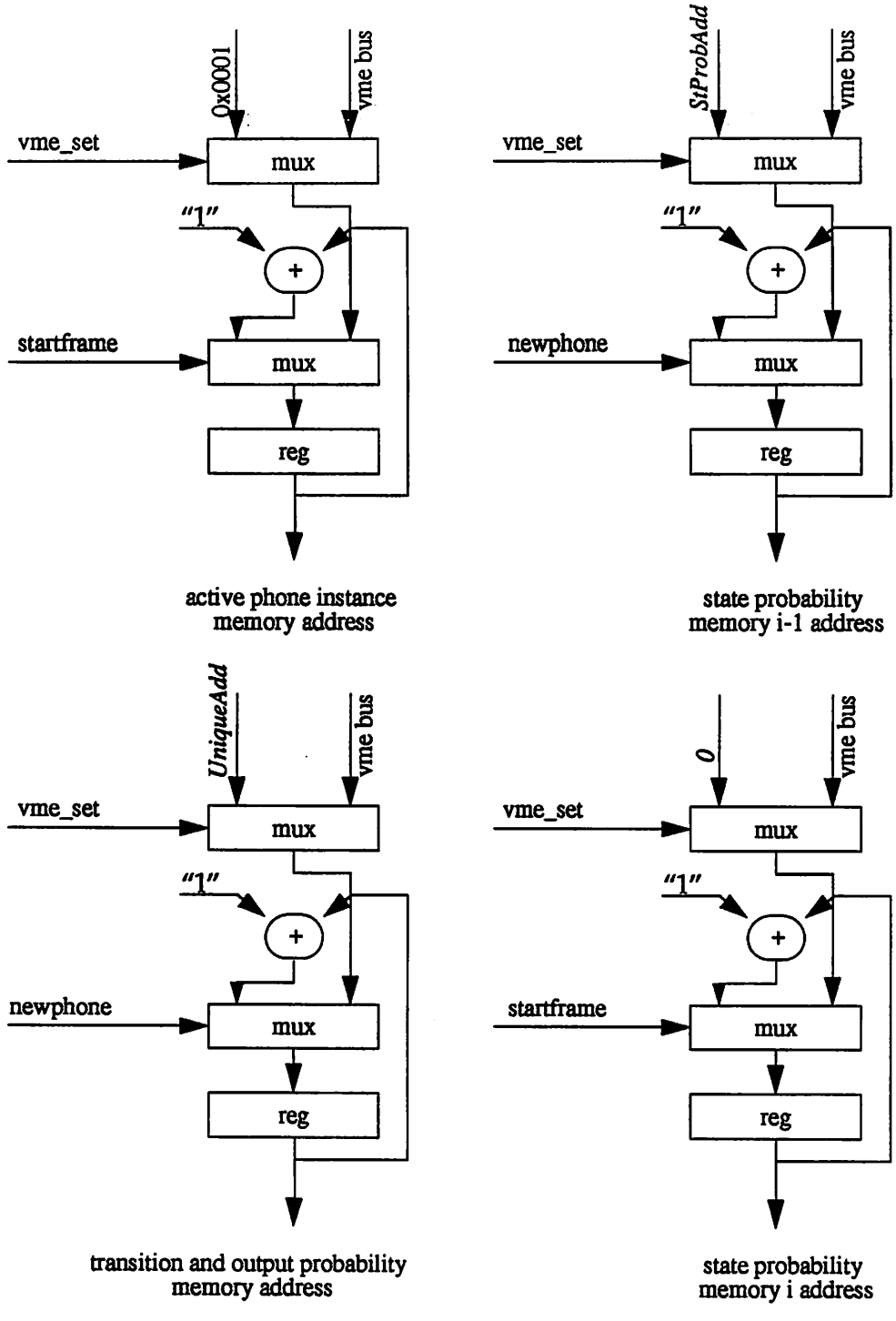


Figure 43: Datapaths of the Address Processor

the Viterbi chip yields an EndFlag (section 4. 3. 1.). In this case, the address counter is set to *StProbAdd* of the next active phone instance.

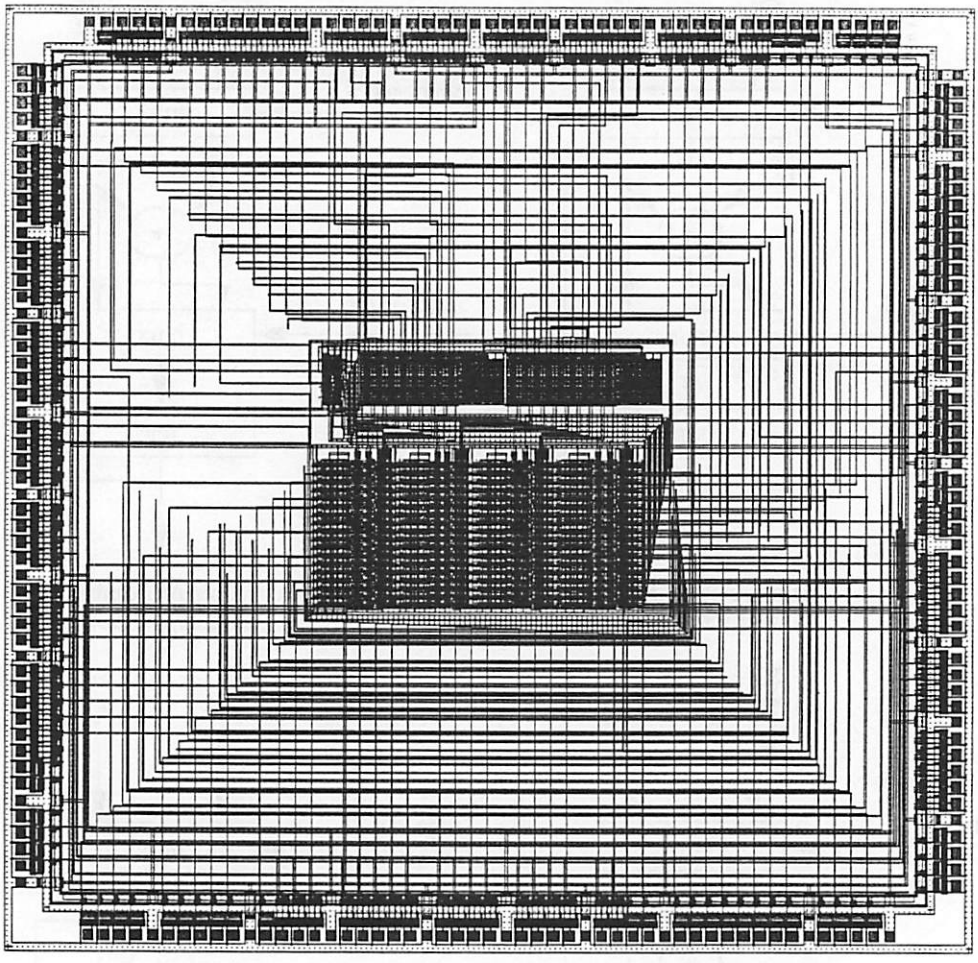


Figure 44: Layout of the Address Processor

The state probabilities that are computed in the current frame, $P(O_i,s)$, are sequentially stored in the state probability memory i. Thus, the counter that generates the address to this memory is incremented every clock cycle. At the start of a frame it gets reset to 0. To be able to locate the state probabilities of a certain phone instance in the next frame, the address processor stores the address of the first state of a phone instance and feeds it to the ToActiveWord system if the Viterbi process generates a request to the ToActiveWord system.

At the start of a new phone instance, the counter that computes the address for the transition probability and output probability memories gets set to *UniqueAdd* which

is read from the active phone instance memory. UniqueAdd points to the first state of a unique phone, and addresses of the succeeding states of that phone instance are in the next sequential address locations, so the counter gets incremented every clock cycle. The end of the phone instance is again indicated by the topology memory, and in this case a new *UniqueAdd* is loaded from the active phone instance memory.

6.3.3.c Control

The phone processing system has 3 independent controllers: the main controller handles the proper operation of the chipset during phone processing, the VME controller takes care of the communication with the VME host, and the test controller is used to put the chipset in scanpath mode and accept scan commands from the VME host (write, read a scan bit or clock the system for one cycle).

The structure of the main controller is data stationary control: for a set of data, the controller generates a control word. As the data are processed in the various pipeline stages, the control word is delayed using the same pipeline delay as the data. If, in a certain pipeline stage, the datapath requires a control, the necessary control bits from the same pipeline stage are fed to the datapath.

In the phone processing system, the main controller has to generate control based on only 2 events: the beginning and the end of a frame. Figure 45 shows a state transition diagram of the main controller. To specify the start of a frame, the main controller is interfaced to the VME bus so that the VME host can issue a "StartFrame" command by writing dummy data to a specific address location. The main controller decodes the VME address, and, if the address corresponds to the "StartFrame" command (and the controller is in the idle state), it goes to a state that generates a "NewFrame" bit. Also, to release the VME bus, an acknowledge signal is generated. After that, the controller goes to a state which generates an EndOfPhone bit which causes that the first active phone instance to be read from the active phone instance

memory. The states in the phone instance are sequentially processed until the EndOfPhone of the corresponding unique phone topology in the topology memory is read. This EndOfPhone causes the next active phone instance to be read from the active phone instance memory: this is done without the interference of the controller. To control the datapath for all actions that are necessary if a new phone instance gets processed, the EndOfPhone flag read from the topology memory is logically OR'ed with the EndOfPhone flag that was generated by the controller and then fed into the control delay registers. All states in the main controller that are not the idle state also output an "active" bit that is used to activate the i/o pads for the phone processing system.

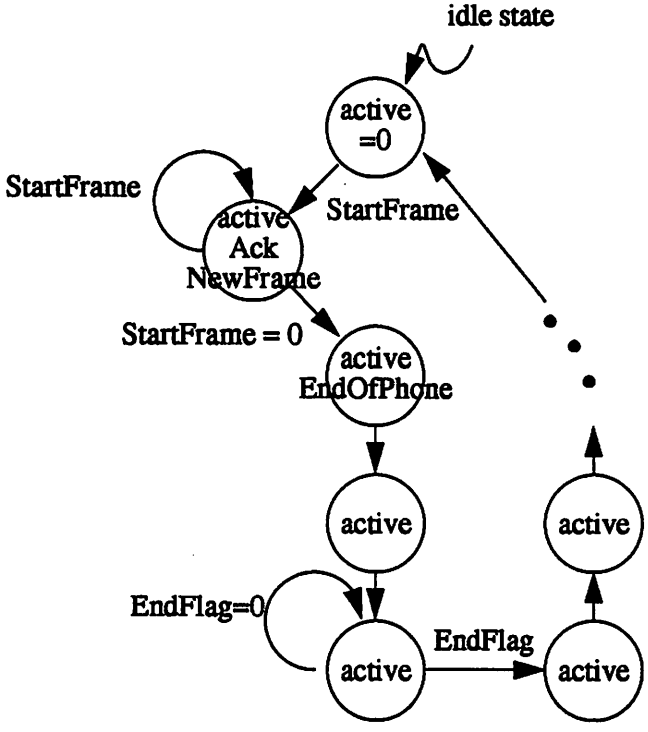


Figure 45: State Transition Diagram of the Main Controller

The end of a frame is encountered if the active phone instance memory yields the EndFlag. This means all active phone instances for this frame have been processed and the main controller can go to the idle state, disabling the active flag which causes

all outputs of the chipset to be disabled. However, this active flag has to be delayed because, when the EndFlag is read, there are still states in the pipeline that have to be processed. This delay s implemented in the controller: it goes through a sequence of states before it enters the idle state and de-asserts the active bit. This delay cannot be implemented using external pipeline registers (as it is done with NewFrame and EndOfPhone), because the active flag has to go into effect immediately after the StartFrame command. If it were delayed using external registers, the io pads would not be enabled on time.

The VME controller implements an interface to the VME host so it can access some registers on the processors, and the memories that are connected to the phone processing system. This interface is implemented using a finite state machine (FSM), since the VME bus cycles are not synchronous to the clock of the Viterbi board. The assumption is, however, that the clock of the FSM is faster than the VME cycle time, otherwise the VME controller would be too slow to react to the VME bus and it would not go through the correct sequence of states (this would result in VME bus errors).

All VME commands are identified by the VME address. Accessing a register is implemented in one VME cycle while accessing a memory takes two cycles if the wordlength of the memory is less than 32 bits (the width of the VME data bus). In this case, the first VME cycle identifies the memory using the 5 LSBs of the VME address and supplies the address of the desired memory using the VME data bus. This address gets latched in the corresponding address register of the address processor (see Figure 43). In the second cycle, the VME host reads or writes the data from or to a VME register on the Viterbi processor. Since the state probability memories have 32 bits, it takes 3 VME cycles to transfer data from the host to these memories. The first cycle identifies the memory and supplies the address, the second cycle gives the probability which gets latched on the Viterbi processor, and the third cycle gives the backtrack tag which is latched on the backtrack processor. After these data have been received, the

Viterbi processor initiates a write operation to the state probability memory using the supplied address and data. Since a certain phone process has one state probability memory that is being read only (i-1) and one that is being written to only (i), just the state probability memory i can be written. To write data to the state probability memory i-1, the other phone process has to be used (see Figure 37). A read from the state probability memory i-1 is implemented with 2*2 cycles: 2 cycles (address + data) to read the probability and 2 cycles (address + data) to read the backtrack tag.

Bit 5 of the VME address is used to differentiate between the phone processing systems A and B. To identify a certain chipset, The Viterbi processors has a pin that specifies if the processor is in system A or B. If the pin is 0, the controller reacts for memory transactions associated with system A, if it is 1, the controller of the B system reacts.

Finally, the test controller is responsible for system level scanpath testing. A command from the VME host activates this controller, and it generates a TestMode signal. Based on this signal, all registers on the entire Viterbi board are no longer clocked, even the state registers of the other controllers. In testmode, the test controller accepts 4 commands from the VME host: to write a bit into the scan path, to read a bit from the scan path, to clock the system for one cycle, or to terminate the test mode.

Using this test feature together with fact that all memories on the system can be accessed by the VME host, there is no storage node on the entire Viterbi board that is not observable or controllable by the VME host.

6. 3. 4. The VLSI ToActiveWord System

The ToActiveWord system generates a new list of active phone instances for the next frame by adding or updating a phone instance in the active phone instance memory based on a request. There are two sources for these requests: the phone processing system and the general purpose successor computation system. When the

ToActiveWord system processes a phone instance, it has to find out if that phone instance was already added to the active phone instance memory. In this case, the corresponding data have to be updated according to Figure 32. If the phone instance was not yet on that list of active phone instances, it has to be added to the next successive available memory location.

6. 3. 4. a Processor partition

Because of the large pin count on the periphery of the ToActiveWord process, the implementation was partitioned on three full custom chips. Here, the partition was based on a bitslice architecture: a certain processor operates on a subset of the 96 bit wide word of the active phone instance memory. Figure 46 shows a block diagram of the ToActiveWord partition.

The request processor operates on the WA id of a request either from the phone processing or the successor computation system. It accesses the ActiveList memory using the phone instance id as address. The data read from the ActiveList memory is a pointer into the active phone instance memory. If this pointer is 0, the request processor uses an address generated by a counter to address the active phone instance memory. Also, the request processor generates the flags (NewFlag and EndFlag, section 6. 2. 1.) for the active phone instance memory and contains the controller for the ToActiveWord system.

The grammar node processor performs the MAX operation for the source grammar node of the current phone instance. If the phone instance is already in the active phone instance memory, it reads the grammar node probability associated with that phone instance and compares it to the grammar node probability of the current request, The higher of the two probabilities is written back into the active phone instance memory. If the phone instance was not yet in the memory, it just inserts the probability associated with the current request. If there is a request from the phone

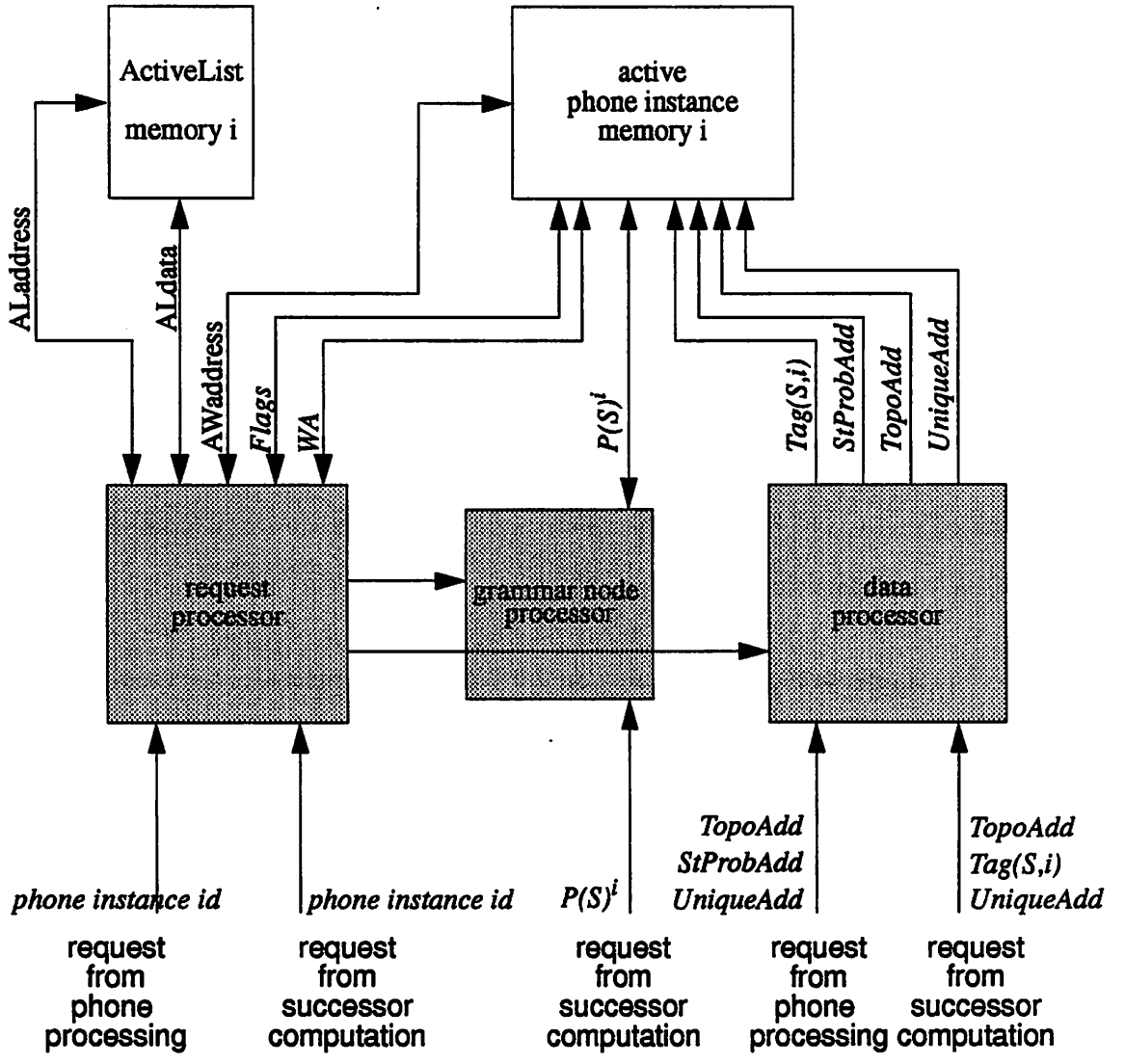


Figure 46: Partition of the ToActiveWord System

processing system and the phone instance was not yet in the phone instance memory, the grammar node probability written into the new location in the active phone instance memory is 0.

Finally, the data processor writes data associated with the phone instance into the active phone instance memory. If the request comes from the phone processing system, the data are the UniqueAdd, the TopoAdd, and the StProbAdd. If the request

was originated by the successor computation system, the data are the UniqueAdd, the TopoAdd, and the Tag that is associated with the most probable grammar node.

6. 3. 4. b Architecture

To process a request to activate a phone instance, the ToActiveWord system has to perform up to 3 memory accesses: First, it reads the ActiveList memory to find out if the phone instance was already activated and to get its address in the active phone instance memory. Then it reads the data associated with that phone instance from the active phone instance memory and performs the operations shown in Figure 32, and finally it writes the updated data back into the active phone instance memory. If the phone instance was not yet activated, the ToActiveWord system only has a write access to the active phone instance memory, but there are two accesses to the ActiveList memory: to read the pointer and to update it.

To increase the throughput of the ToActiveWord system, the memory accesses can be pipelined. However, the pipeline has to allocate all possible memory accesses: reading and writing both the active phone instance and the ActiveList memories. Figure 47 shows the hardware allocation table for the pipeline of the ToActiveWord system. A row in that table corresponds to a certain pipeline stage during various clock cycles while the columns represent the pipeline at a certain clock cycle. Different shadings of pipeline stages refer to different data.

This table shows that the ToActiveWord system has 7 pipeline stages. The ActiveList memory is accessed during pipeline stages 2 and 4, and the active phone instance memory is accessed during pipeline stage 4 and after pipeline stage 6. Using this scheme, we can enter new data into the pipeline every other cycle. In this case, the memories are fully utilized and the maximum throughput is achieved.

Since it is possible that there is a request for a phone instance which is currently being processed in the datapaths of the ToActiveWord system, the processors

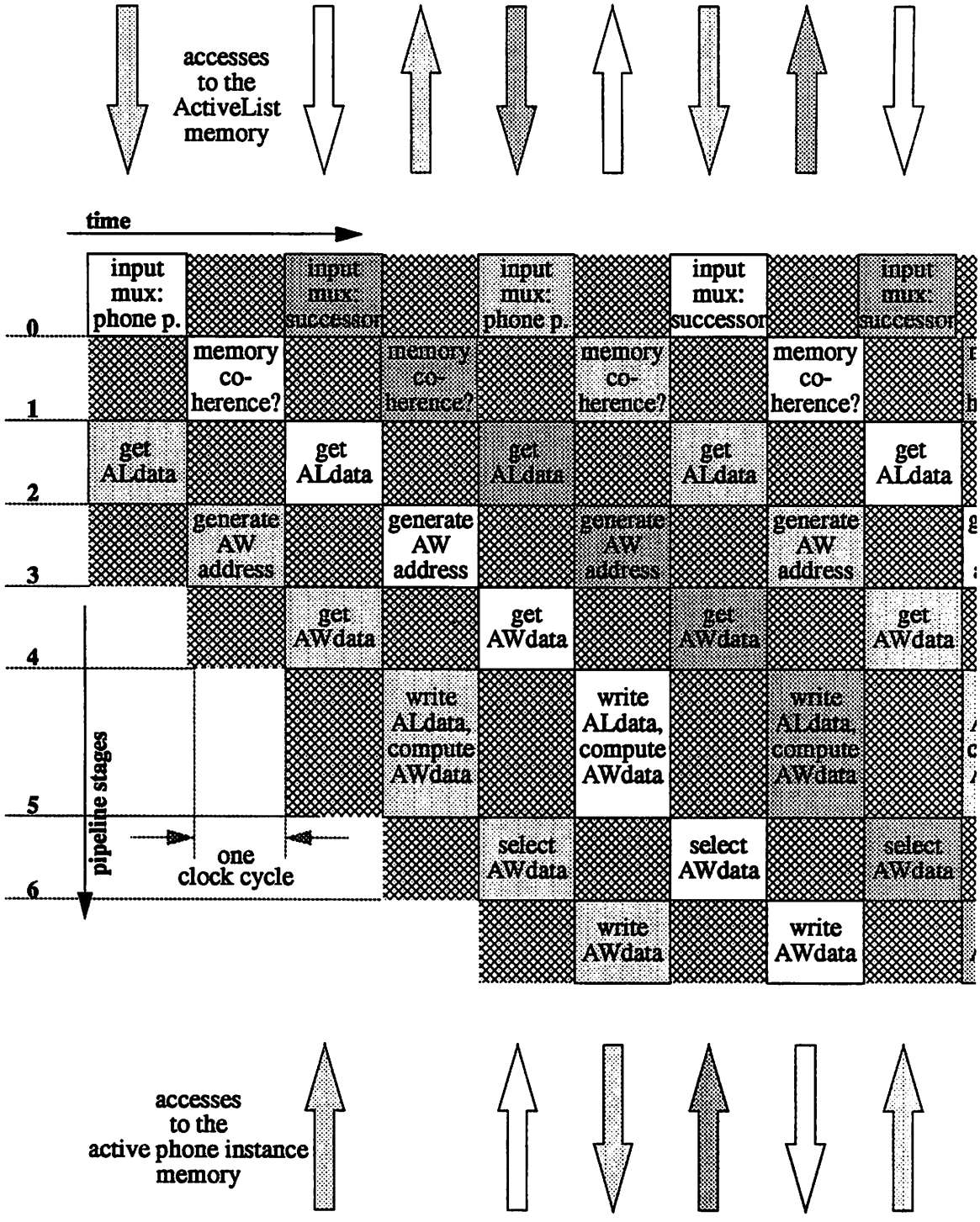


Figure 47: Hardware Allocation Table for the ToActiveWord system

could potentially access data from memories that are not yet updated (memory incoherence). To avoid this, there are three memory coherence registers at pipeline

stage 0. These registers keep the phone instance ids of the phone instances that are currently in the pipeline. The phone instance ids of these registers are compared and if there is a match between two registers, the ToActiveWord processors do not accept any more input until after the pipeline is flushed. Then the memories contain relevant data and new requests are accepted.

The ToActiveWord system accepts requests from two subsystems: the phone processing system, which generates requests that are synchronous to the clock of the ToActiveWord system (since both use the same clock generation circuitry), and the successor computation system which generates requests that are not synchronous. To arbitrate between the two requests, there is a multiplexor at the input of the datapaths which toggles between the two requests every other clock cycle. If there is a request from the successor computation system, it gets acknowledged after the input multiplexor accepted the data. Requests from the phone processing system don't get acknowledged, but if there is a request that cannot be processed right away, the ToActiveWord system generates a stall signal that halts the pipeline registers of the phone processing system.

Figure 48 through Figure 50 show the layouts of the 3 ToActiveWord processors.

6. 3. 4. c Control Structure

The control structure for the ToActiveWord system is time stationary. Here, the controller generates a control word for all the pipeline stages for a certain clock period. This control structure has the advantage that controller decisions can be applied to the pipeline stage without delay. This is important in the ToActiveWord system: there are some decisions that have to take immediately effect in all the pipeline stages. For example, if there is a match in the memory coherence registers, pipeline stages 0 and 12 have to be stopped while the other stages have to continue in order to flush the pipeline.

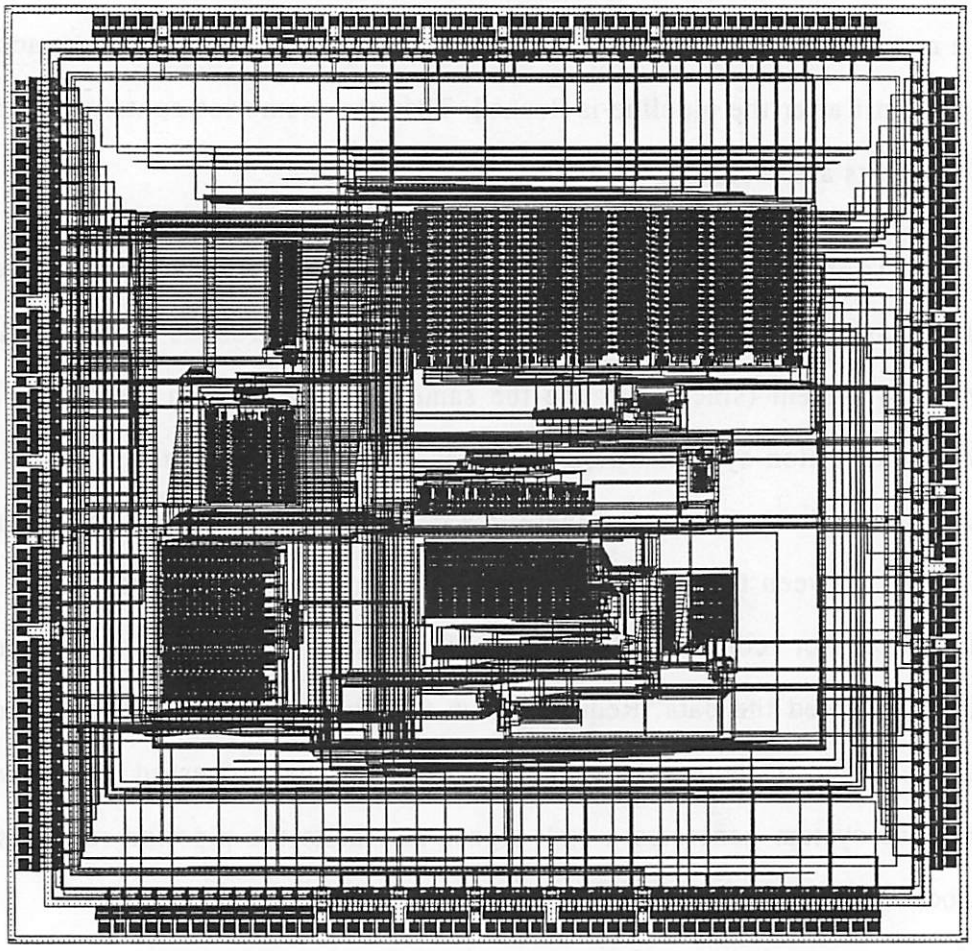


Figure 48: Layout of the Request Processors

The drawback of this implementation is, however, that control decisions for certain data have to be delayed until this decision can take effect in the corresponding pipeline stage. Therefore, the controller has to have several state transitions before the control bit goes into effect.

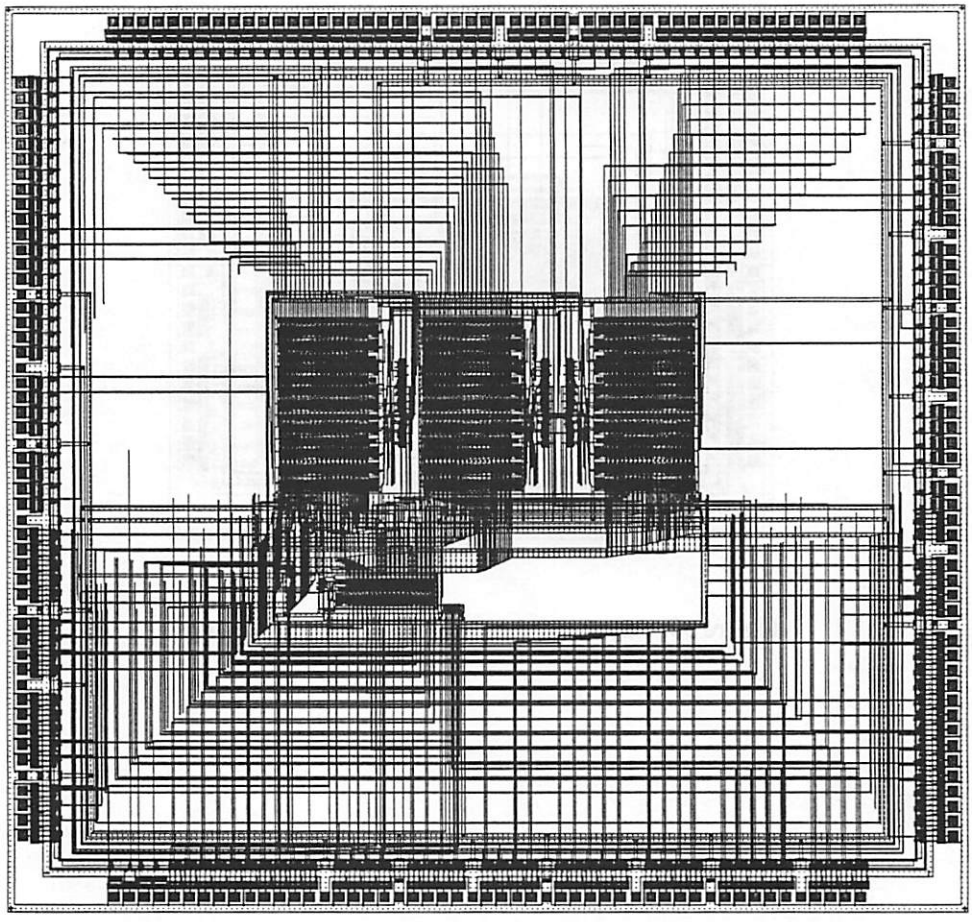


Figure 49: Layout of the Data Processors

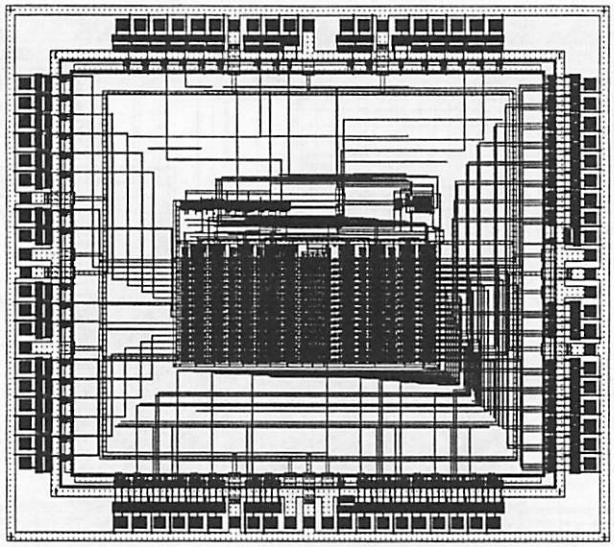


Figure 50: Layout of the Grammar Node Processors

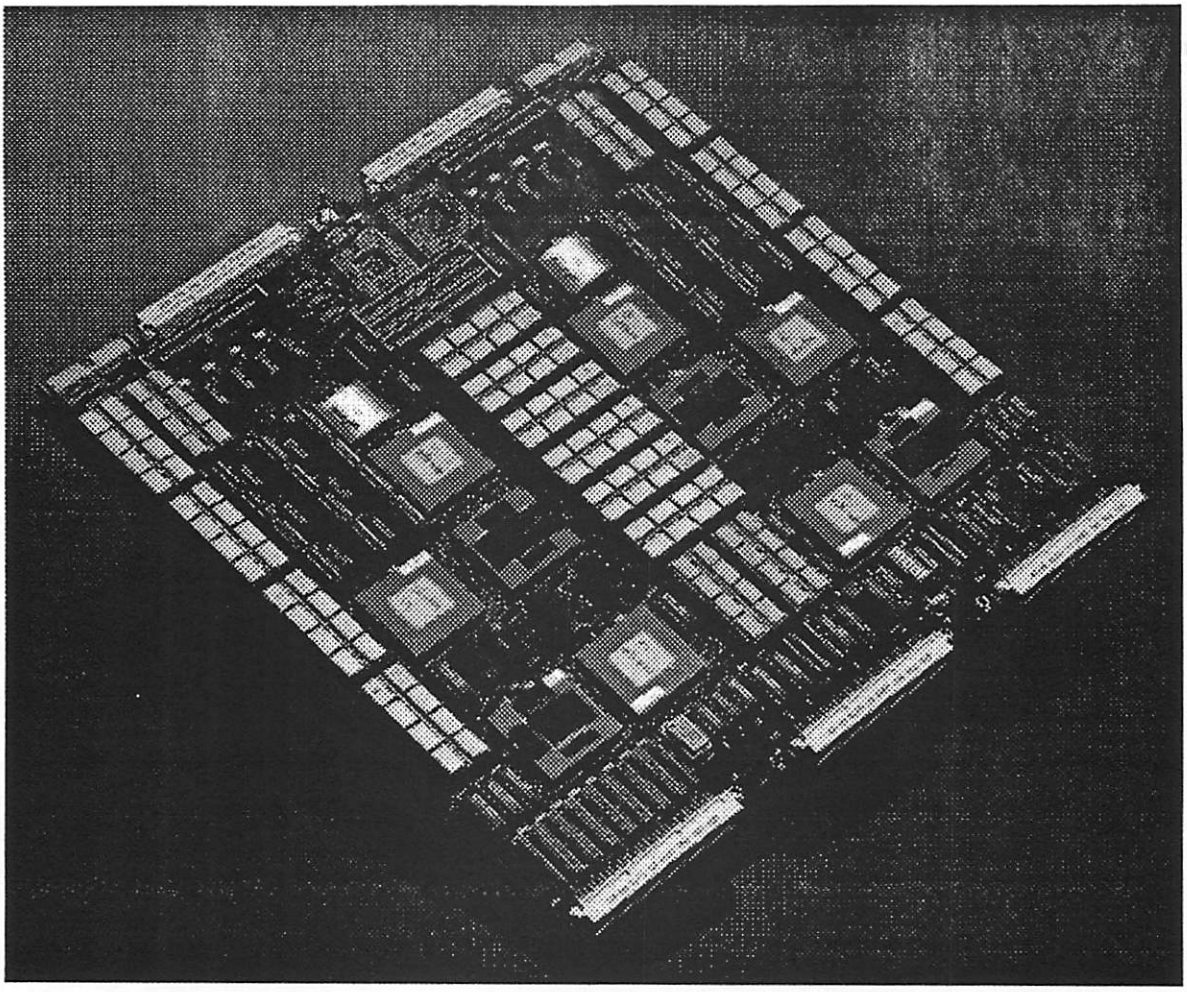


Figure 51: The Viterbi Board

7

The Distribution Board

The distribution board stores the output probability distributions of unique states (= states of unique phones). This system can handle a maximum number of 64,000 unique states, and associated with every state are 4 discrete distributions, where each distribution corresponds to a certain speech feature. A distribution contains 256 probability values, one probability for every possible observation. Thus, a particular observation for a certain speech feature is represented with an 8 bit feature vector, and since our front end processing system generates feature vectors for 4 different speech features, a speech segment is represented with 4 feature vectors. The probabilities in the distributions are stored with their negative logarithms and quantized to 10 bits (see 4.2.). Thus, the distribution board has to store 64K x 256 x 10 bits = 80MBytes.

7.1. Board Operation

At system start-up, the VME host loads the output distributions of the desired HMM into the memories of the output distribution board. These distributions were generated using the forward-backward training algorithm on a large amount of speech data. They were generated off line and stored on a disk, and the VME host accesses them via ethernet and transfers them into dynamic memory (DRAM) on the output distribution board. For that, the distribution board has a VME interface that uses the

lower 26 VME address bits to specify the memory address on the distribution board and the lower 10 bits of the VME data bus supply the probability values. For testing purposes, the memory can also be read by the VME host. The distribution board is organized in 4 DRAM banks. One bank contains, for all unique states, the probability distributions of a certain speech feature.

During recognition, the VME host reads every 10 msec 4 feature vectors from the front end processing system. The feature vectors in our system correspond to the following speech features: the cepstrum, the delta cepstrum, the energy and the delta energy. The task of the distribution board is, to generate, for all unique states, the output probability that corresponds to the set of feature vectors for a certain frame. These probabilities have to be written into one of the output probability memories on the Viterbi board. For that, the distribution board has two ribbon cables that connect it to the address and data busses of these memories (see Figure 52). The output

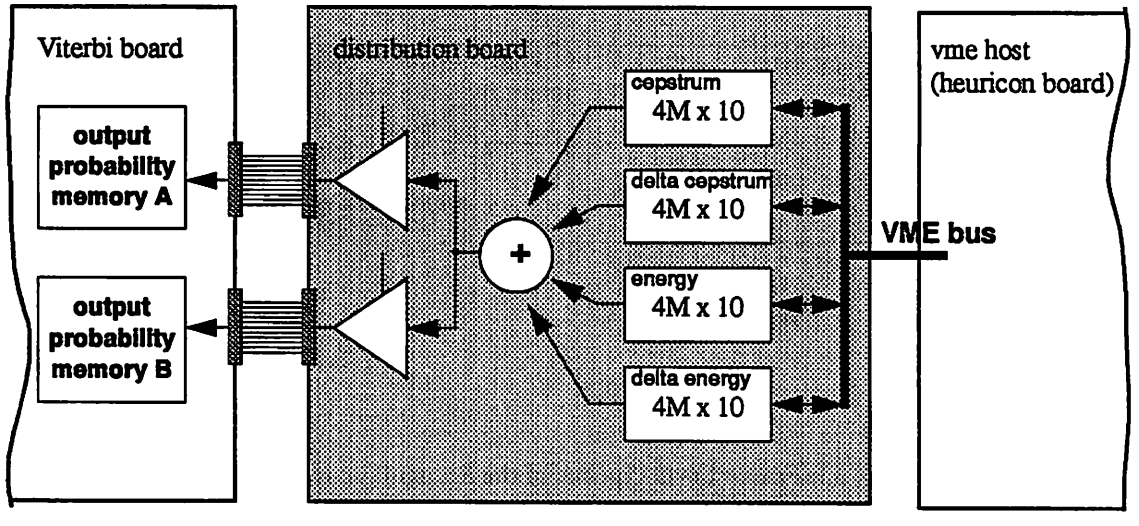


Figure 52: Basic Function of the Distribution Board

probabilities that get loaded to the output probability memory will be used by the phone processing system in the next frame. Thus, the speech recognition system has a

pipelining scheme on the frame level as sown in Figure 53. The feature vectors are delayed by one frame when the front end processing system computed the feature vectors. Then, the corresponding output probabilities for all unique states are loaded into one of the output probability memories on the Viterbi board, and in the next frame, the phone processing system uses this memory to update the state probabilities of active phone instances. Thus, during a certain frame, one output probability memory gets written by the distribution board, while the other memory is read by the phone processing system.

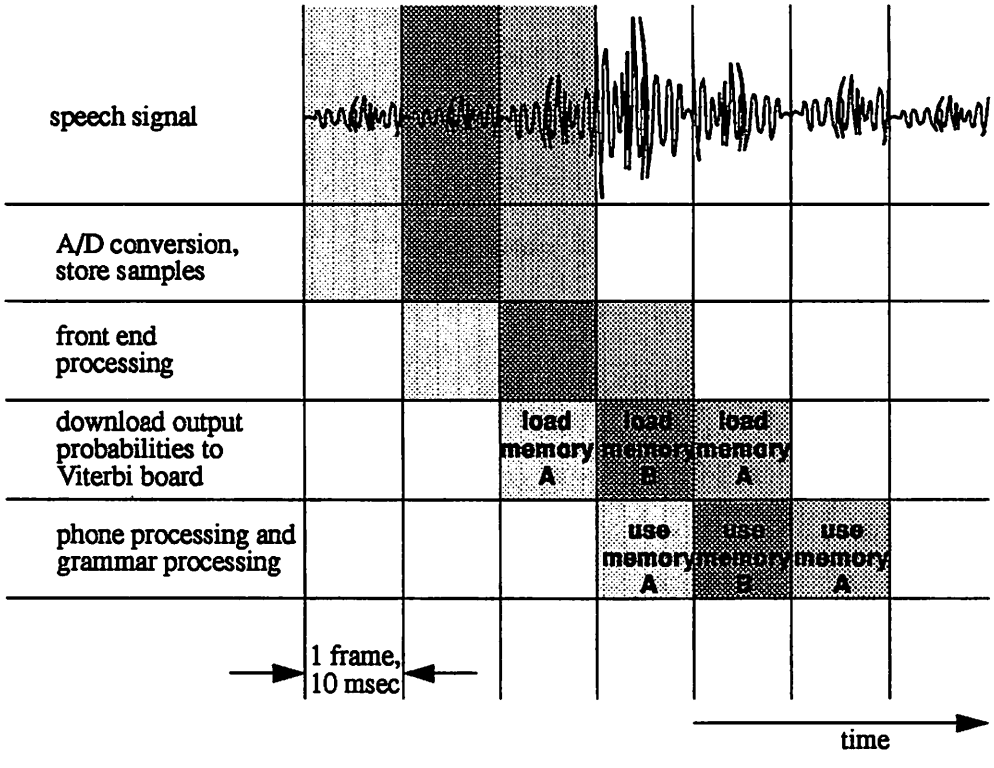


Figure 53: Frame Level Pipeline of the Recognition Hardware

To compute the output probabilities for a certain frame, we assume that the 4 features are statistically independent. Thus, the joint output probability of a unique state for a set of observations is the product of the 4 output probabilities that correspond to the observations (EQ 8). Since the probabilities are represented in their

logarithmic values, the multiplication corresponds to an addition. The VME host writes the 4 feature vectors into registers on the distribution board and specifies the number of unique states in the current HMM (this is the number of probabilities that have to be transferred each frame to the Viterbi board). Then, the VME host specifies which output probability memory has to be loaded on the Viterbi board, and gives a command to the distribution board to start downloading output probabilities.

After that command was received, the distribution board reads the 4 output probabilities associated with the set of 4 feature vectors. These probabilities get added and written into the specified output probability memory on the Viterbi board. For that, the distribution board generates the address, data, and the memory control signals (cs, we) for the output distribution board. The output probability memory gets written sequentially, starting from the address that corresponds to the number of unique states down to address 0. The address that is used for the output probability memory is the address of the unique state.

The distribution board interrupts the VME host after the output probabilities of all unique states are transferred. Also, the memory busses that were used get disabled so that the phone processing system can access the memory in the next frame. Thus, the distribution board is ready to accept a new set of features from the VME host and to download data for another frame, this time to the other output probability memory.

7.2. Architecture

7. 2. 1. Memory Organization

The distribution board has 4 identical DRAM banks, 16M x 10 bits each. Each bank contains the output distributions of unique states for a certain speech feature. In this system, one bank contains the distributions associated with the cepstral coefficient, another the distributions associated with the delta cepstrum and so on. The memory

banks are partitioned into 256 blocks, and each block contains the output probabilities of up to 64K unique states for a certain 8 bit feature vector. This memory architecture is shown in Figure 54:

The unique states inside a block are on consecutive memory addresses, therefore, if the VME host specifies a feature vector, the corresponding output probabilities can be sequentially accessed. Since our system uses 4 speech features, the VME host specifies 4 blocks, one block one each memory bank. When the probabilities are transferred to the Viterbi board, 4 probabilities corresponding to a certain unique state are simultaneously read from the banks. The sequential ordering of states has the advantage, that the memories can be read using fast page mode, where only the column address has to be changed after a read. The DRAM memory banks are implemented with memories that have an address space of 4M, so one page contains probabilities for 2K unique states with can be read with a fast access time (80nsec).

7. 2. 2. Address Generation

All blocks in the memory banks have the unique states on the same relative address locations. Thus, only one counter is required to address these relative locations on the four banks. The difference between the addresses on the various memory banks is a base address that corresponds to the 8 bit feature vectors. This is indicated with the shaded blocks in Figure 54. Figure 55 shows the architecture of the memory address unit. The address for a certain memory bank is composed of a feature vector and an index address. These two addresses are merged in such a way, that the lower 11 bits of the address are the lower 11 bits of the index counter address. Thus, the address unit accesses a new row only after 2K columns had been accessed. The index address itself is the sum of a DRAM base address that can be set by the VME host, and the output of a decremental counter. Before the transfer of data, this counter is preset by the VME host to the number of unique states. Then, the VME host issues a "go" command and the counter starts decremental. The counter generates a flag if all the column addresses of a

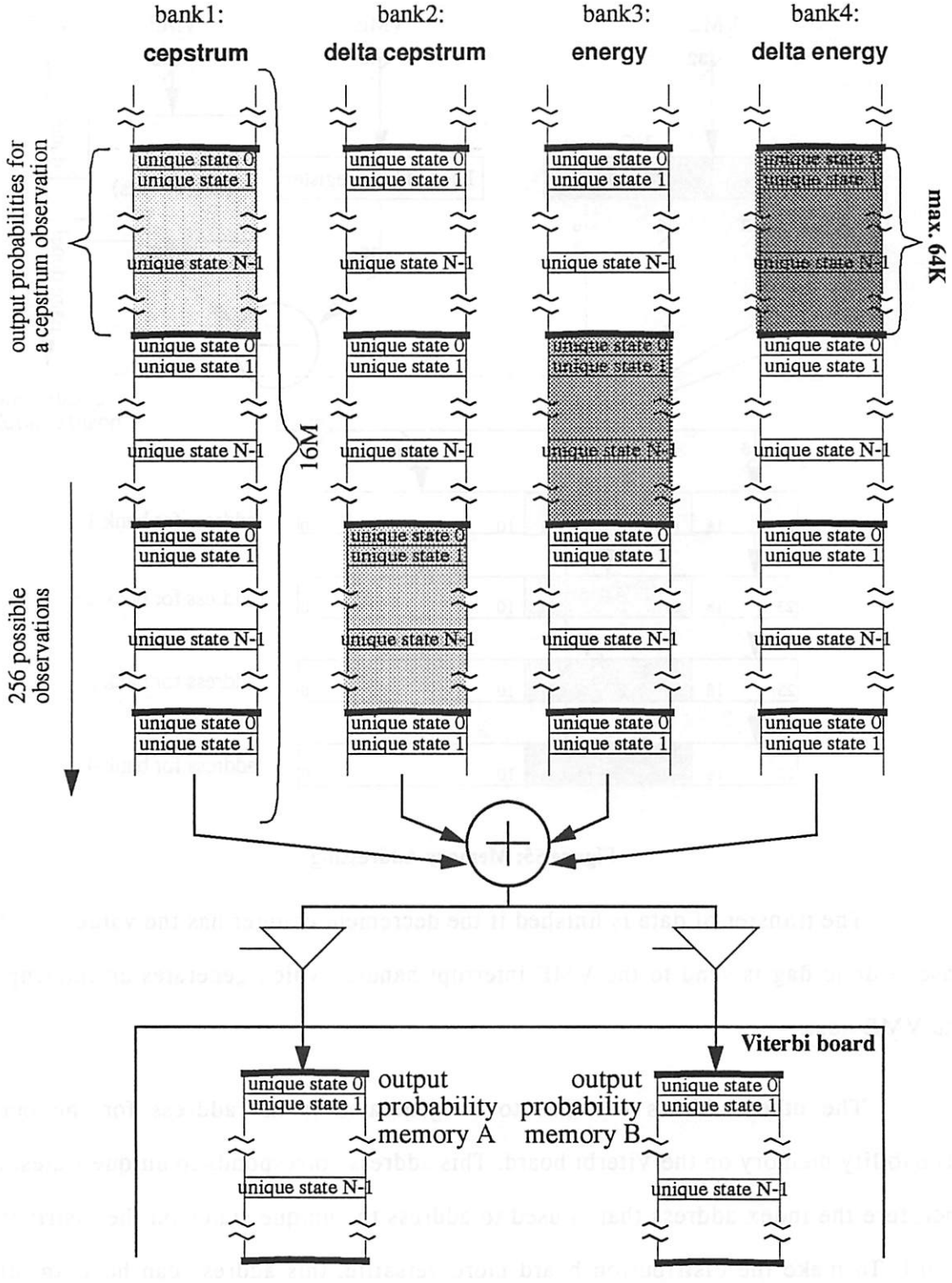


Figure 54: Memory Architecture

page have been accessed, so a controller on the distribution board can take care of the different timing involved in a new row access.

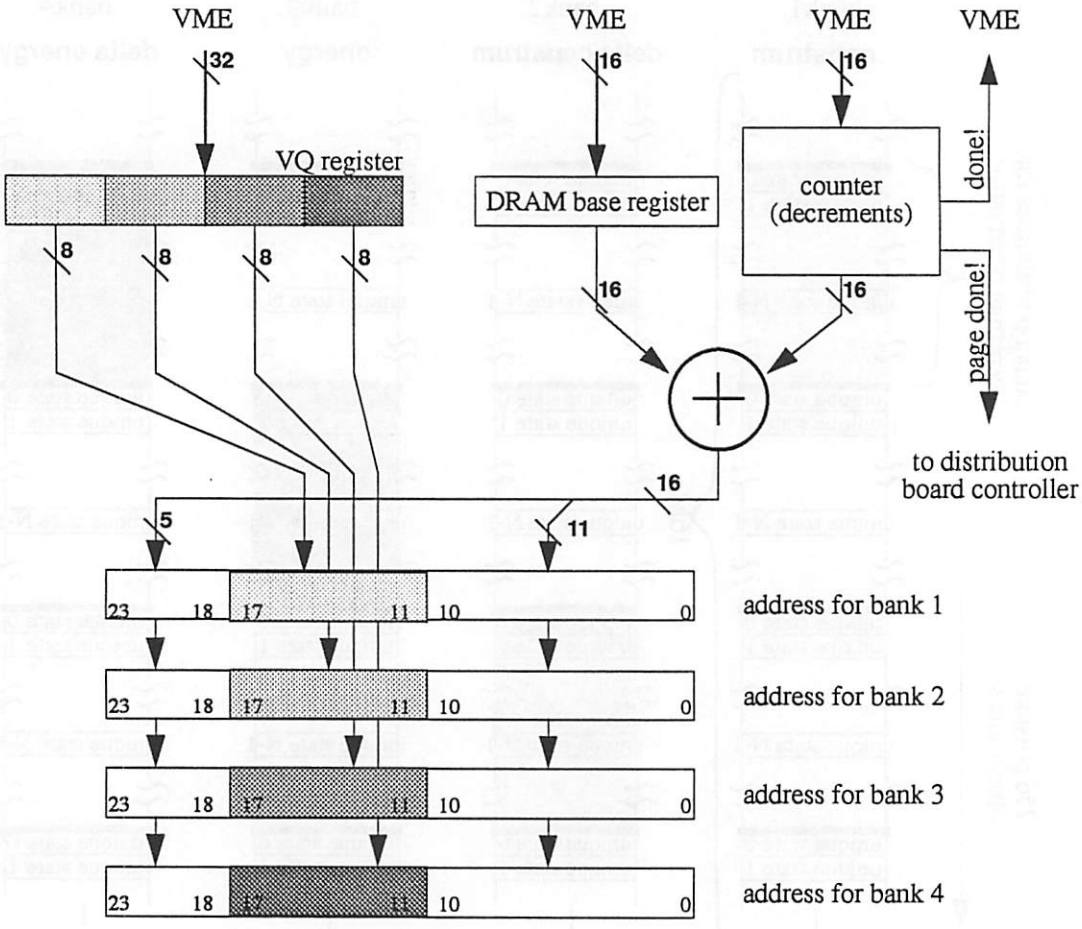


Figure 55: Memory Addressing

The transfer of data is finished if the decrement counter has the value 0. In this case, a done flag is send to the VME interrupt handler which generates an interrupt to the VME host.

The other address that has to be generated is the address for the output probability memory on the Viterbi board. This address corresponds to unique states, and therefore the index address that is used to address the unique states on the distribution board. To make the distribution board more versatile, this address can have an offset "SRAMbase" which is provided by the VME host and stored in a register. Thus, the output probability address is the sum of the index address and SRAMbase.

For testing purposes, the VME host can read the output probability memories on the Viterbi board through the distribution board. For that, there is an output probability address register which can be set using the VME data bus, and a data register which latches the corresponding data from the output probability memory. This register can then be read by the VME host.

Table 4 gives a list of the VME registers on the distribution board.

VME address	register name	size	purpose
0x48xx,xxxx	VQ register	32 bits	keeps 4 feature vectors
0x49xx,xxxx	DRAM base register	16 bits	offset to index address
0x4Axx,xxxx	SRAM base register	16 bits	offset to output probability memory address
0x4Bxx,xxxx	Index register	16 bits	keeps number of unique states
0x4Cxx,xxxx	ActiveA/ActiveB* register	1 bit	specify which output probability memory to load
0x4Dxx,xxxx	output probability address register	16 bits	store address if VME host access to output probability memory
0x4Exx,xxxx	go-ahead-signal	1 bit	activates transfer of data
0x4Fxx,xxxx	output probability data register	12 bits	store data if VME host access to output probability memory

Table 4VME Registers on the Distribution Board

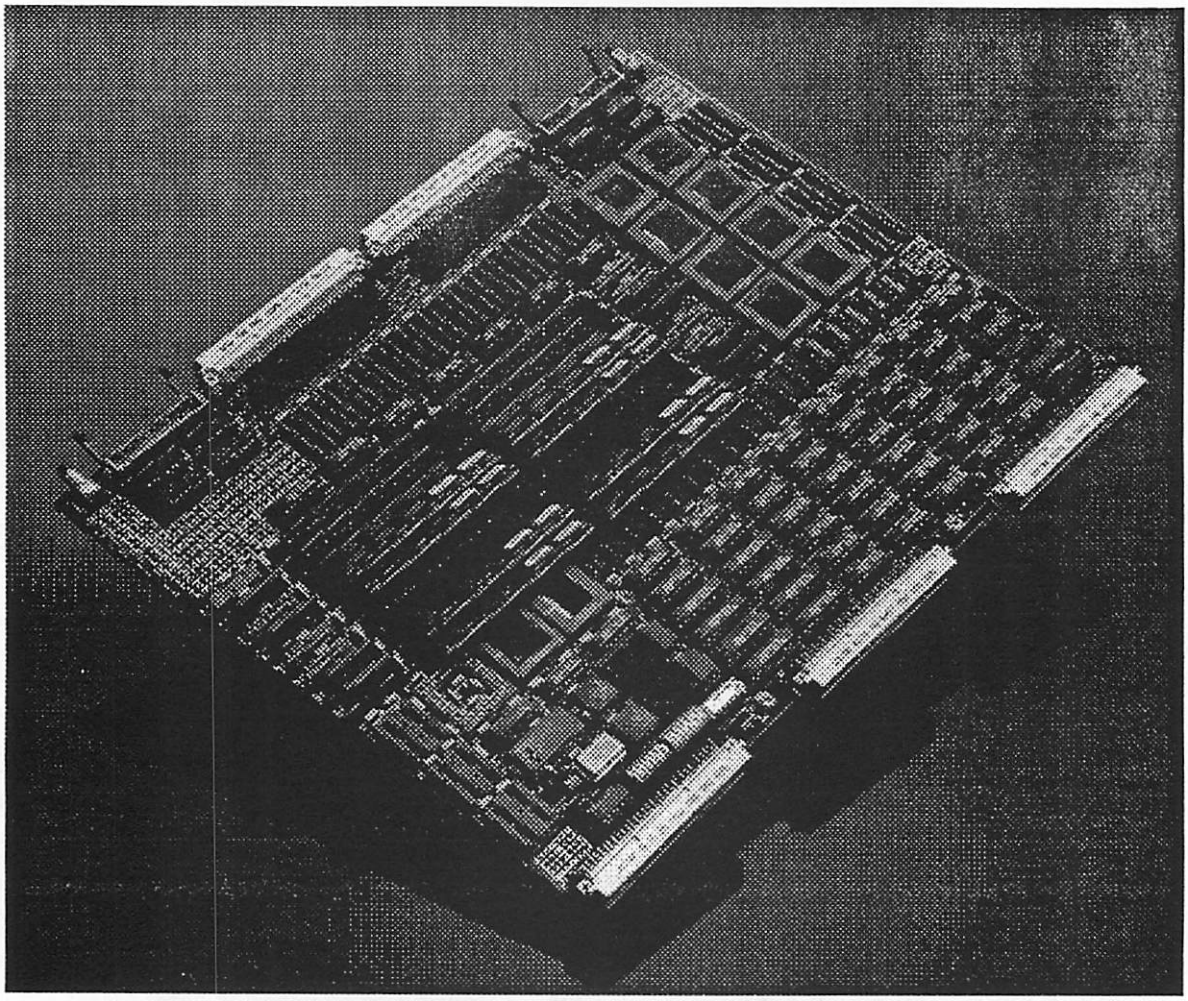


Figure 56: The Distribution Board

8

The DSP Board

The task of the DSP board is to perform successor computation (see 5. 2. 3.) and front end processing (see 2.1.) Both processes use algorithms that vary among different recognition systems. Therefore, it is essential to implement these processes with general purpose hardware (5.2.). Computing the successor grammar node probabilities in grammar processing is very demanding for complex grammars, therefore the DSP board described in this chapter uses an architecture that supports successor computation with multiple digital signal processors. It is even possible to extend the system and to use multiple DSP boards for the successor computation task. The outputs of the DSP board are - besides the feature vectors from front end processing - 94 bit wide data structures associated with phone instances that get activated, and they have to be passed to the ToActiveWord system on the Viterbi board. A certain phone instance might be activated several times from different predecessor phone instances (6.2.), therefore the DSP board has to have a full custom high bandwidth interface to the Viterbi board.

8.1. The Board Architecture

The architecture of the DSP board is shown in Figure 57. It has 3 TI320C30 digital signal processors (DSPs) with an identical configuration. Each DSP has 2

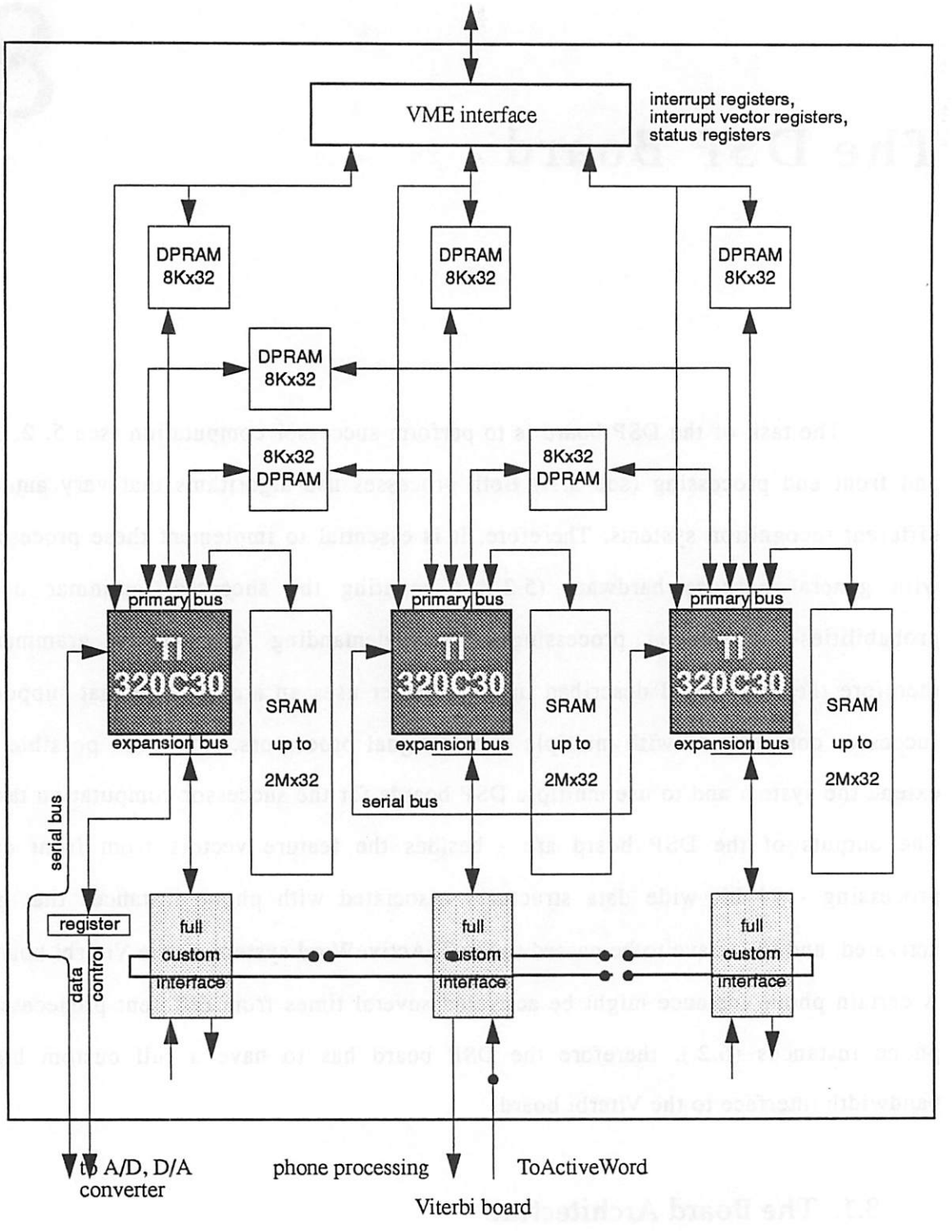


Figure 57: DSP Board Architecture

external busses, a primary bus and a secondary bus with 32 bits of data each. To store data and programs, each processor has its own local SRAM that is connected to the primary bus. The size of this memory can be up to 512Kx32 if 64Kx32 memory modules are used, or up to 2Mx32 if 256Kx32 memory modules are used. To access this memory, the DSP uses one wait state, so a memory cycle requires 60nsec. This local memory is not shared with other processors on the board or with the VME host.

To be able to communicate with the VME host, every DSP has a dual ported memory (DPRAM)attached to the primary bus. This DPRAM is accessible by the various DSP processors and by the VME host. To indicate if a certain memory portion is being used by one of the DSPs or by the VME host, there are several semaphores. The DPRAM is accessed by the VME host through a VME interface. This interface also has interrupt registers that can be set by the VME host to interrupt any of the processors. If one of the processors interrupts the VME host, there is also an interrupt vector register that identifies the interrupting processor. These interrupt vector registers are accessible by the VME host processor only. If the interrupting processor has to send information to the VME host to specify the interrupt, the dual ported memory can be used.

The board architecture also allows inter processor communication (IPC) using dual ported memories (IPCRAM). The IPC network implements a fully connected ring, therefore every DSP processor can communicate to any other processor. These dual ported memories are connected to the primary bus, and each processor has 2 IPCRAMs to communicate to the left and right neighboring processor. To avoid contention, these memories also have semaphore banks.

Using this architecture, there are two possible configurations for successor processing: parallel processing of successors where each processor executes the same program, of bitsliced processing, where each processor executes a different program and processes different data structure associated with phone instances.

8. 1. 1. Parallel Processing of Successors

In this configuration, each processor on the DSP board implements a successor computation processes, and these processes work in parallel. For that, each processor computes the successors of a different phone instance using the same algorithm, and the requests that come from the phone processing system are distributed among the processors. Figure 58 shows this configuration:

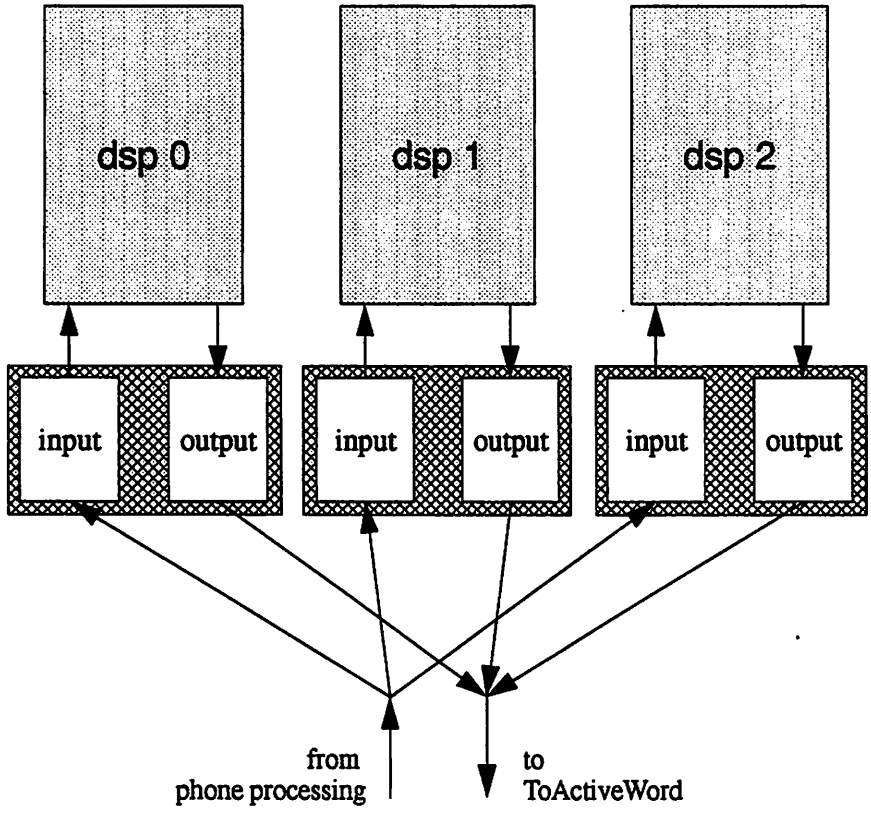


Figure 58: Parallel Processing of Successors

Each processor receives the complete grammar node structure for a certain phone instance from the phone processing system (phone instance id, grammar node probability and backtrack pointer), and sends the complete source grammar node structure of its successor phone instances (source grammar node probability, backtrack tag, phone instance id, UniqueAdd, Topology address) back to the Viterbi board to the

ToActiveWord system. This configuration requires, that data associated with the grammar have to be replicated in the memories of the individual processors. However, the processes are completely independent, so there is no need for inter processor communication.

The interface of a certain processor has to properly select incoming requests from the phone processing system: a certain request should only be processed by one processor. Also, requests to the ToActiveWord system have to be synchronized between the processors to avoid contention at the ToActiveWord system.

Another possibility to partition the successor computation between parallel processes is, that every processor computes a subset of the successor phone instances of the same phone instance. This has the advantage, that a certain processor only has to store a subset of the grammar topology, while still every processor performs the same algorithm. In this case, a request from the phone processing system has to be accepted by all processors, and requests to the ToActiveWord system again have to be synchronized to avoid contention.

8. 1. 2. Bitsliced Successor Processing

In this configuration, every processor computes the successors of the same phone instance. However, a certain processor generates different data associated with the successor phone instances: one processor could, for example, be dedicated to compute the source grammar node probabilities, another to compute the backtrack tags, and yet another to generate (read from memory) the phone instance id, UniqueAdd and topology address (see section 6. 2. 1.). This configuration is shown in Figure 59.

Here, each processor performs a different program, and it is not necessary to replicate information in the memories of the various processors. However, the processors have to communicate. For example, the processor that computes the source grammar node probabilities of successor phone instances has to read the transition

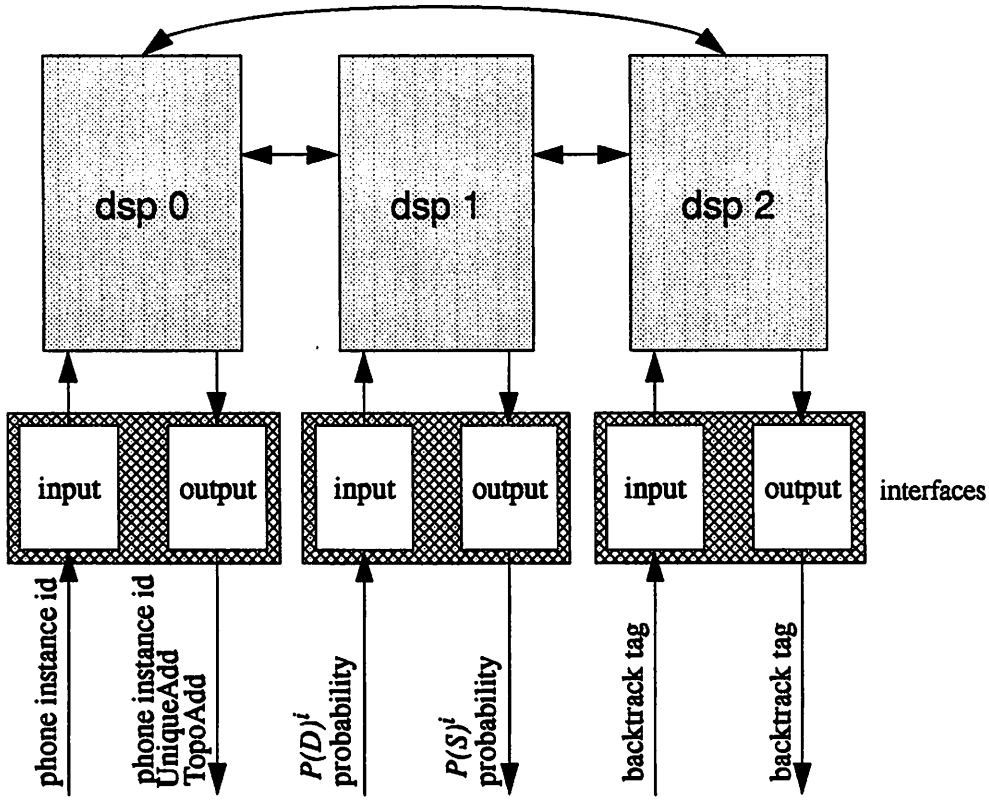


Figure 59: Bitsliced Successor Computation System

probabilities from its memories. For that, it needs the phone instance id of the predecessor phone instance which has to be read from the IPCRAM that connects to the DSP that processes the phone instance id.

In this configuration, the interfaces have to be synchronized: a request from the phone processing system has to be accepted by all interfaces, but only a subset of the data is relevant. In the other direction, a request to the ToActiveWord system can only be generated if all processors finished the computation of their contribution to the successor phone instance structure.

8.2. Full Custom Interface

This section describes the full custom interface to the Viterbi board. It accepts requests from the phone processing system to activate the successor phone instances of a certain phone instance, and it generates requests to the ToActiveWord system to add these successors to a list of active phone instances. This interface is flexible to support various processor configurations as discussed in the previous section.

8. 2. 1. Input Interface

The part of the interface that accepts requests from the phone processing system (called "input" in Figure 58 and Figure 59) has two possible configurations:

1. incoming requests have to be distributed to the various processors if each processor computes all the successors of a certain phone instance. For that, one request is only accepted by one interface.
2. incoming requests have to be broadcast to all interfaces if a certain processor computes only a subset of the successors or if a certain processor computes only a subset of the data structure associated with a successor (bitsliced configuration). Thus, a request is accepted by all interfaces.

These requirements are met by the input interface that has the structure shown in Figure 60. To implement configuration 1 (see above), an interface only accepts a request form the phone processing system if there is a token (InToken) at the input. If the request was accepted, the token is passed to the neighboring interface using the clock of the Viterbi board. This is possible because requests are always synchronous to the Viterbi board clock, and it always takes at least two clock cycles between requests (a phone instance has at least 2 states, and it takes one clock cycle to process a state). Using this scheme, the interfaces of the individual processors take turns in accepting

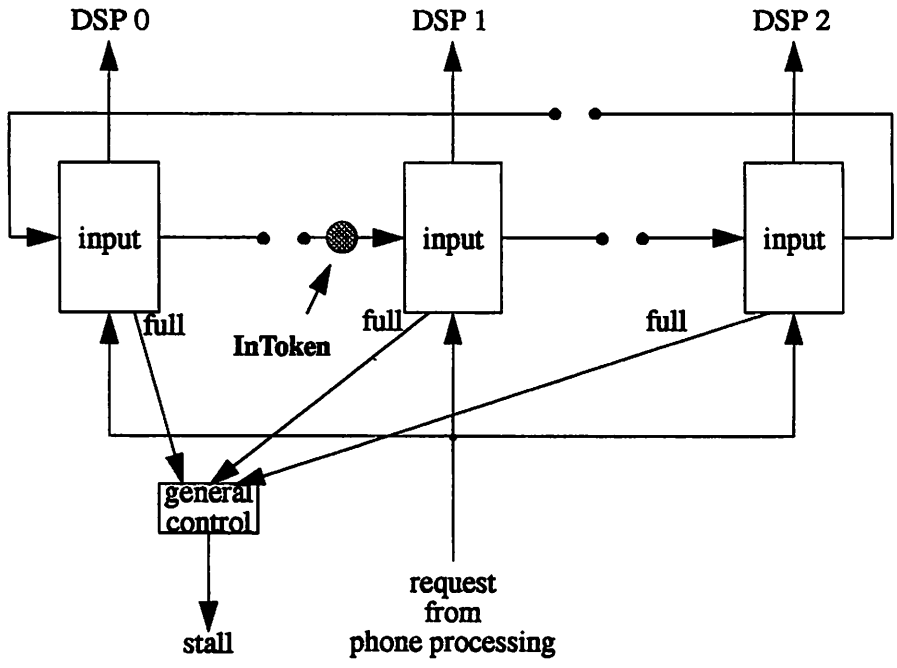


Figure 60: Structure of the Input Interface

requests. To be able to configure the interfaces if less than three processors are used for successor computation, there are jumpers on the board to route the InToken.

To implement configuration 2 (see above), each interface accepts requests regardless of the position of the InToken.

Since the Viterbi board and the DSP board do not operate synchronously, the interface has to buffer the data associated with requests. This is done using FIFOs (first in first out memories). Thus, if a request is accepted by the input interface, the data associated with that request are pushed onto the input FIFO. To read these data, the DSP processor pops the FIFO.

If the phone processing system generates more requests than the DSP board can process, the input FIFOs could overflow and data might be lost. To avoid that, the phone processing system gets stalled as soon as the input FIFOs are full. This stall signal is generated by a control block (general control) using the full flags of the

individual FIFOs. The empty flags of the input FIFOs can be polled by the dsp processors. If the empty flags indicate that the input FIFO is not empty, the DSP can read the corresponding input structure (phone instance id, destination grammar node and tag). Due to the limited wordlength of the DSP bus, one structure has to be read at a time.

8. 2. 2. Output Interface

Similar to the input interface, the output interface has two possible configurations:

1. Requests to the ToActiveWord system have to be serialized between the different interfaces to avoid contention. This is the case if the successor computation system is configured with identical parallel processes.
2. In the bitsliced configuration, requests to the ToActiveWord system have to be synchronized so that data associated with a phone instance that has to be added to the list of active phone instances, are send simultaneously.

This output interface structure is shown in Figure 60. To serialize requests to the ToActiveWord system (configuration 1), there is an output token (OutToken) that gets passed between the individual interfaces. If an output FIFO is not empty and the OutToken is at the input of that interface, a request (request 0 through 2) is generated. If, however, the FIFO is empty, no request is generated and OutToken is passed to the neighboring interface. Passing the OutToken is done using an acknowledge signal (see Figure 60, OutToken_Ack). The OutToken at the input of an interface is released if that interface asserts OutToken_Ack. Requests from the individual interfaces are passed to logic which coordinates requests to the ToActiveWord system. In this case (configuration 1), a request to the ToActiveWord system is the logic OR of the requests from the individual interfaces. To implement configuration 2, the individual requests are generated as soon as the output FIFO is not empty, and a request to the

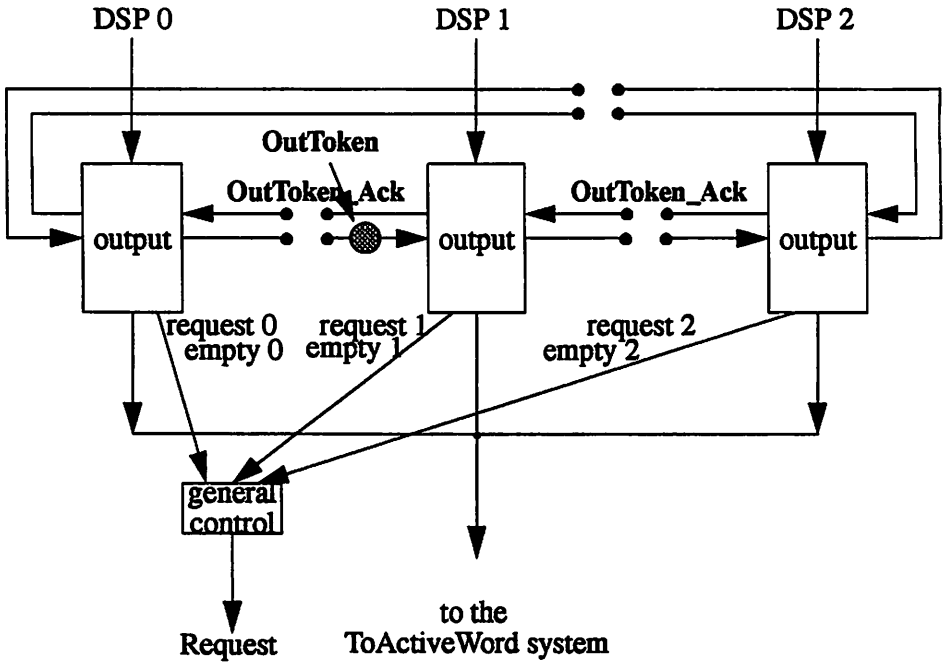


Figure 61: Structure of the Output Interface

ToActiveWord system is generated if none of the fifos is empty (logic AND of the individual empty signals).

Similar to the InToken, the routing of OutToken and the OutToken_Ack can be configured with jumpers to accommodate different processor configurations.

8. 2. 3. Interface Implementation

Figure 62 shows both, the output and the input interface which are connected to the expansion bus of a DSP.

In order to save board area, I designed a full custom FIFO chip that is customized to the width of the structures that have to be buffered (Figure 63). This chip was used to implement the input and output FIFOs. The FIFOs are controlled by field programmable logic devices (FPGAs, Altera EP1810). Thus, they can be re-programmed to implement different configurations as discussed above. These FPGAs

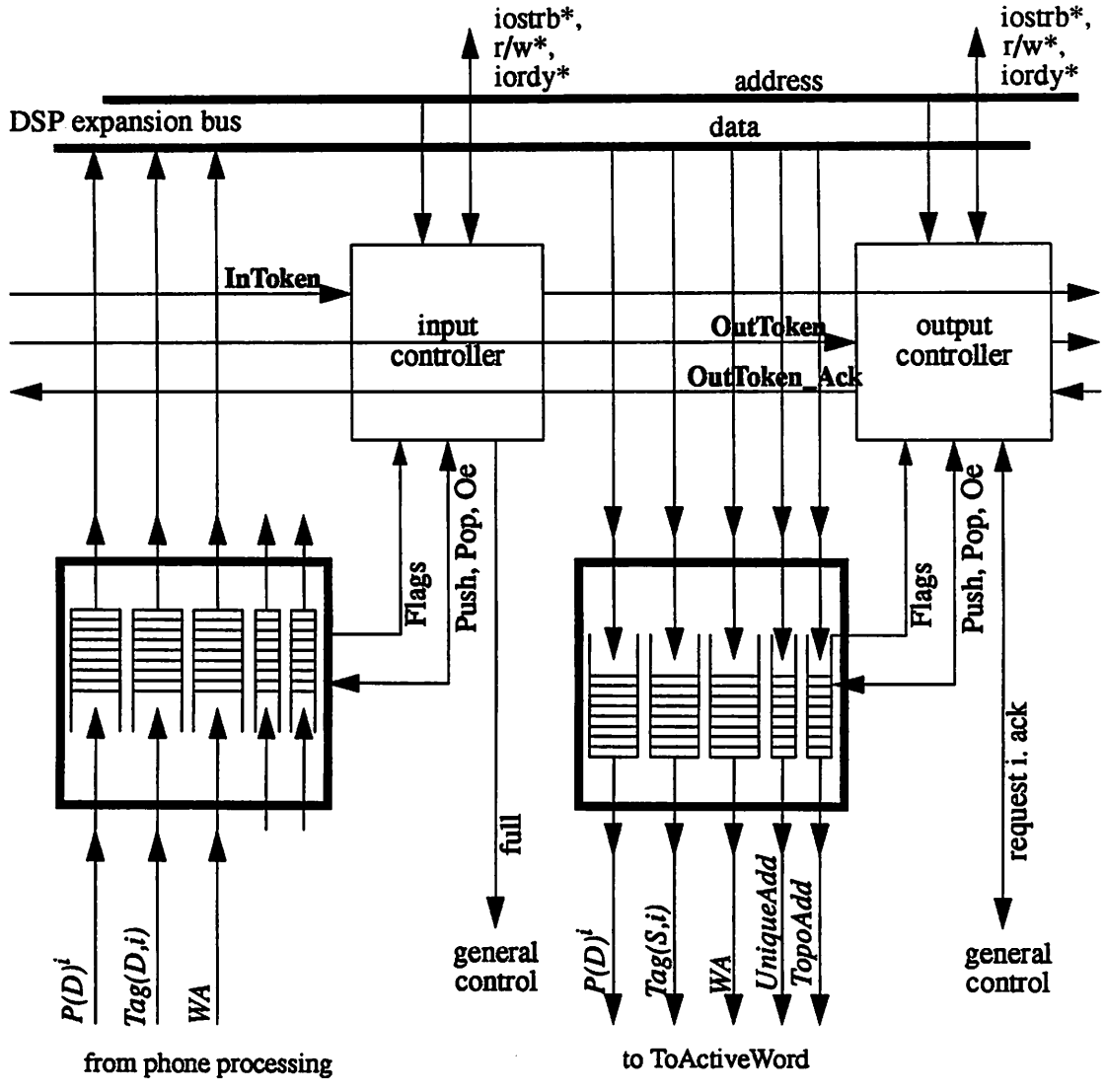


Figure 62: Full Custom Interface

also have internal registers, so the token passing scheme could be implemented without external registers.

The controllers also implements address decoders so that the DSP extension bus can read (input fifo) and write (output fifo) to the interface.

In addition to the Viterbi board interface, one processor (DSP 0) also has an interface to the external A/D converter. To initialize this converter, there is a 4 bit

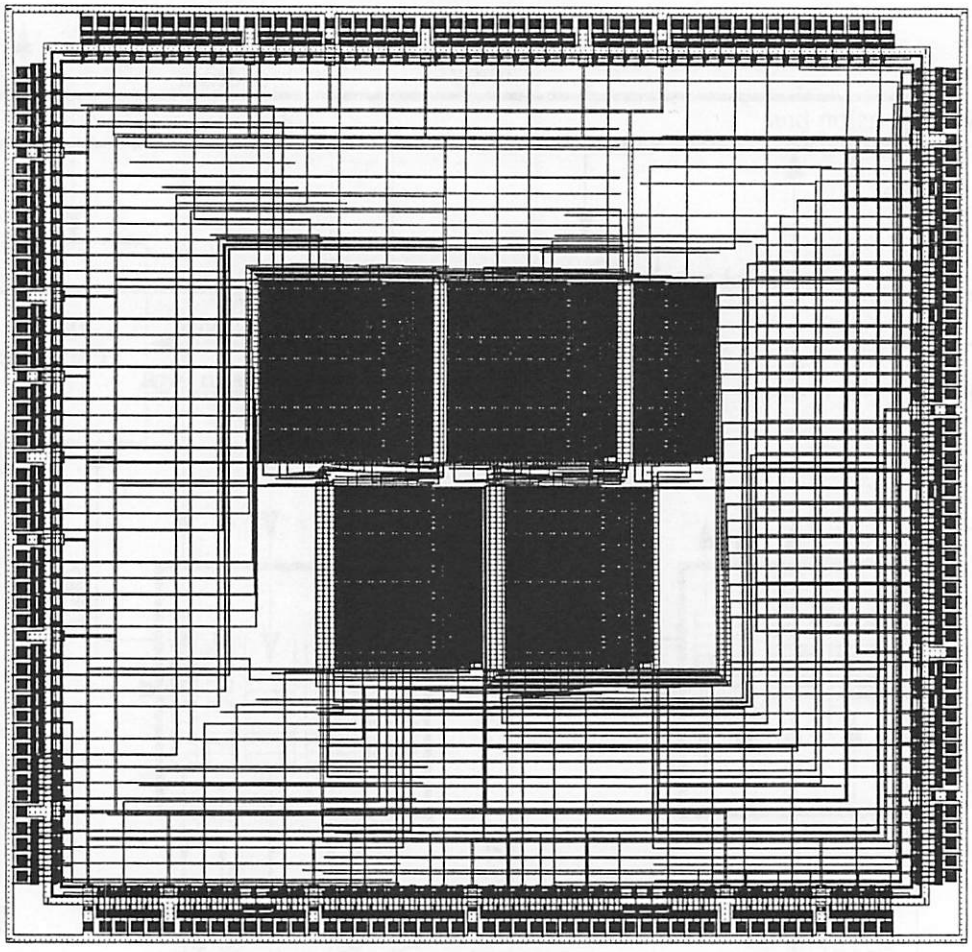


Figure 63: Chip Layout of the Full Custom Interface FIFO

parallel interface which is connected to the dsp's expansion bus (Figure 57). The digitized speech data are send to the DSP using one of the serial busses.

8.3. Processor Synchronization

The DSP board has special hardware to synchronize processors at the end of a frame. This is necessary before an EndFlag can be send to the ToActiveWord system (see 6. 2. 2.) to indicate that a frame is processed. This EndFlag then causes the ToActiveWord system to generate an interrupt for the VME host processor.

To generate this EndFlag, there are two necessary conditions: first, the phone processing system has to be finished processing phone instances from the active phone instance memory, so no more requests are issued to the successor computation system. This is indicated by the Active flag (generated by the main controller of the phone processing system, see 6. 3. 3. c). Second, each processor on the DSP board has to be finished processing the successors of the phone instances that were last requested in that frame. Only then can one processor send a structure to the ToActiveWord system that has the EndFlag.

Thus, DSP 0 (Figure 57) is dedicated as "master processor" to generate the EndFlag, and this processor has to have information about the status of the other processors. This synchronization is implemented as shown in Figure 64.

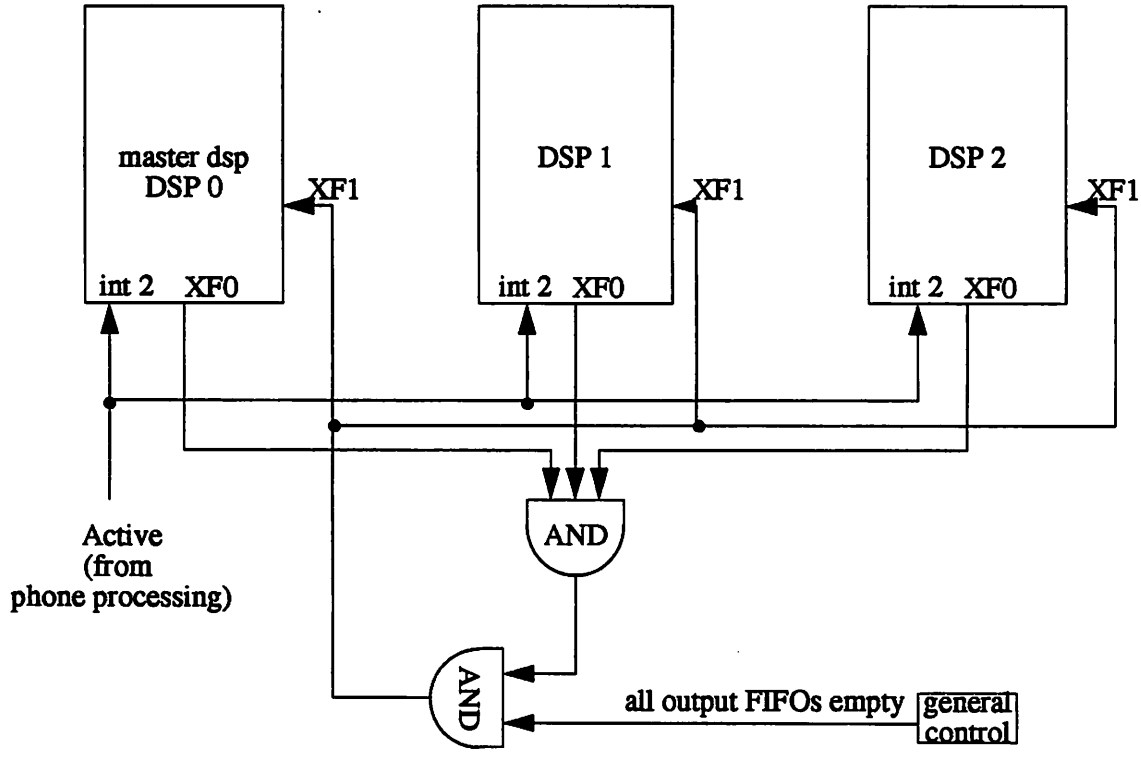


Figure 64: Processor Synchronization at the End of a Frame

The Active signal from the phone processing system is connected to an interrupt input of the DSPs (int2). Thus, if Active is de-asserted (phone processing system has finished processing phone instances), the DSPs get interrupted. Thus, a DSP has finished a frame if it received the interrupt and processed all the pending requests from the input FIFOs. In this case, it executes the command SIGI [TI88], an interlocked operation that uses two external signals, XF0 and XF1 to synchronize processors. XF0 is an output signal of the processor which gets asserted as a result of the SIGI instruction, and its purpose is to request synchronization. The DSP that executes the SIGI instruction is waiting for XF1, the synchronization acknowledge signal, to be asserted. On the DSP board, the synchronization can be acknowledged if all DSPs executed the SIGI command (all DSPs finished processing their pending requests) and all output fifos are empty.

Thus, the signal XF1 is the logical AND of the three XF0 signals and a flag that indicates that all FIFOs are empty which was generated by the general control block from the individual empty signals of the output fifos. If the master processor, f.e, DSP 0, receives the XF1 signal, it can send a structure to the ToActiveWord system that contains the EndFlag. For this operation, any processor can be the master processor.

8.4. Multi Board Operation

For complex or large grammars it is necessary to use more than 3 processors for grammar processing. In this case, control signals have to be routed between boards, and there are two connectors on the DSP board for this purpose. The control signals are shown in Figure 65.

To synchronize the input and output interfaces across boards, the InToken (Figure 60), OutToken, and OutToken_Ack signals (Figure 60) have to be routed to the Viterbi board interfaces on the various boards. These signals can be routed to

connectors via jumpers. One connector (gr-conn-in) is used as input for these signals, while the other one (gr-conn-out) is used as output to other boards.

Also, the signal Gr_Stall (used to stall the phone processing system if the input fifos are full) and Gr_Strobe (request signal to the ToActiveWord signal) have to be distributed across boards. Here, it is necessary that one of the DSP boards coordinates the generation of these two signals. For that, the global control logic blocks (pal22v10_int) of the various boards are connected through the connectors gr-conn-in and gr-conn-out, and the Gr_Stall and Gr_Strobe output of the last logic block in the chain (Gr_Stall_sync and Gr_Strobe_or) is connected to the ToActiveWord system via jumpers.

Finally, XF1 has to be passed between boards. XF1 is the logic AND of the XF0 signals from the individual processors (Figure 64) and the empty signals (Out_Empty) of the output FIFOs. Thus, to generate the overall XF1 across boards, XF1_out of the local board is generated as the logic AND of XF1_in, which was generated on another board, and the local XF0 and Out_Empty signals. XF1_out is then passed to another board. Thus, the last board in that chain generates the overall XF1 signal which is broadcast between the boards, and XF1_out of this board is connected to the overall XF1 using a jumper (Figure 65).

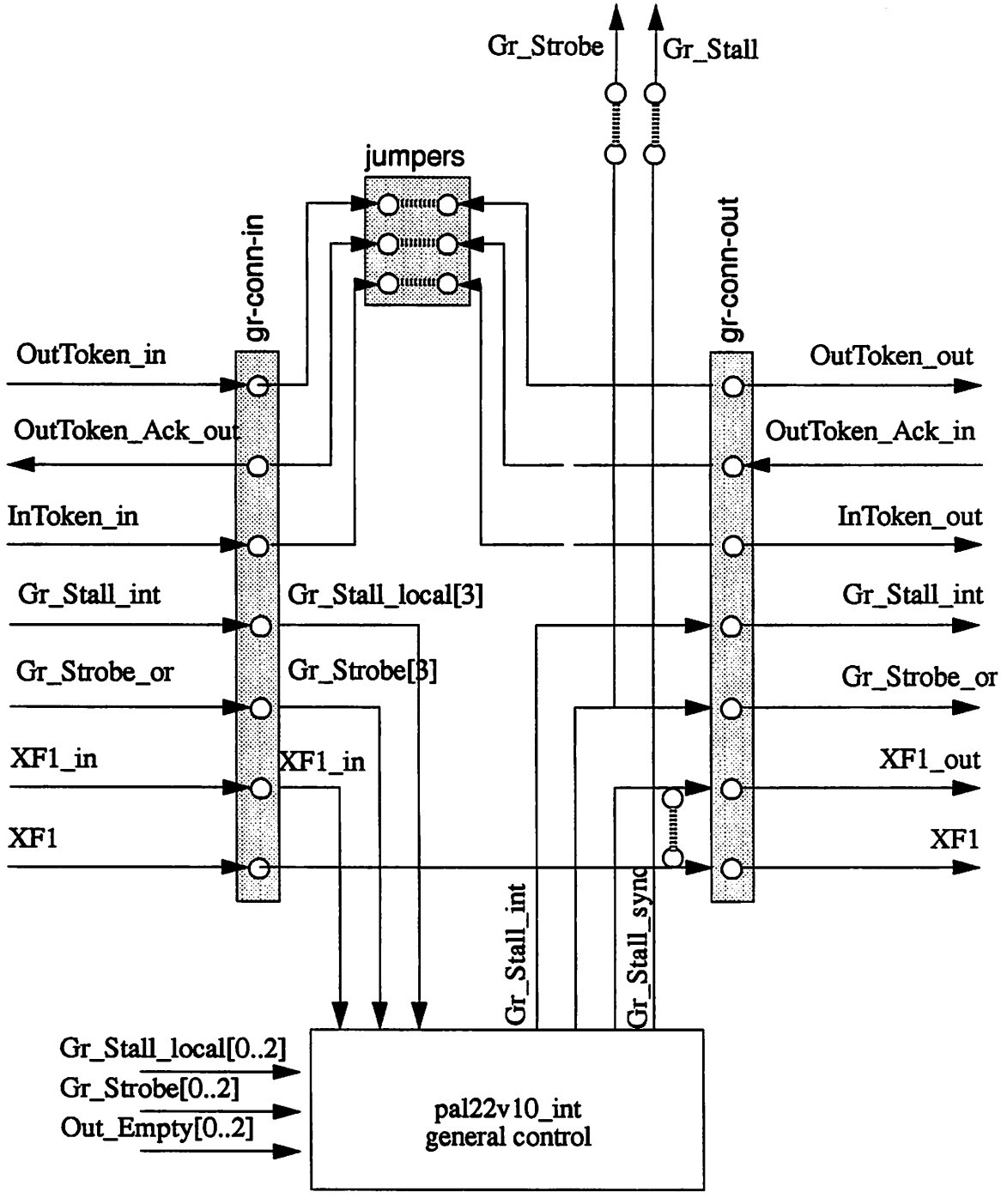


Figure 65: Connectors and Jumpers for Multi Board Operation

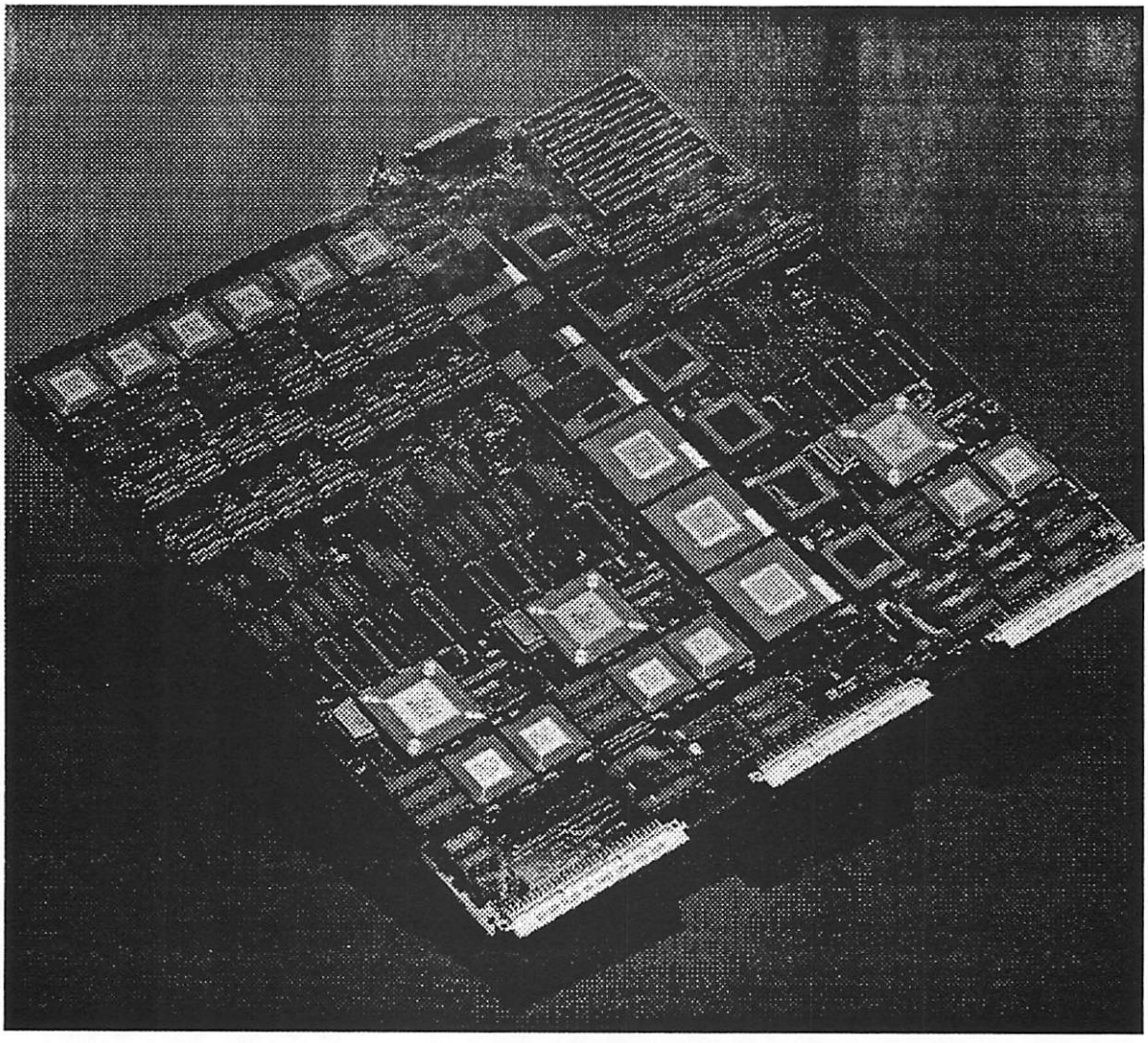


Figure 66: The DSP Board

9

System Software and Recognition Results

To test and demonstrate the speech recognition system, a speaker independent, connected digit recognition task was implemented. This chapter describes the software that loads the HMM parameters onto the memories of the hardware and controls the system during recognition, and it gives the recognition accuracy obtained for the digit recognition task. For this test, the front end algorithm was not ported onto the DSP board. Instead, files were read that contained the already processed speech data.

9.1. System Startup

At system startup, the Heuricon board (section 5. 2. 5.) has to load the HMM parameters onto the corresponding hardware memories. These parameters are the output probabilities, the transition probabilities, and the topologies of unique states. The program that controls the transfer of these parameters is called *Load_Hmm_File* (see appendix), and it has one argument which is the name of the HMM parameter file.

9. 1. 1. Format of the HMM Parameter File

The HMM parameter file has two major sections, one section that describes unique phone topologies, and another section that describes the states of unique phones. To illustrate this file format, the example phone topology of Figure 67 is used.

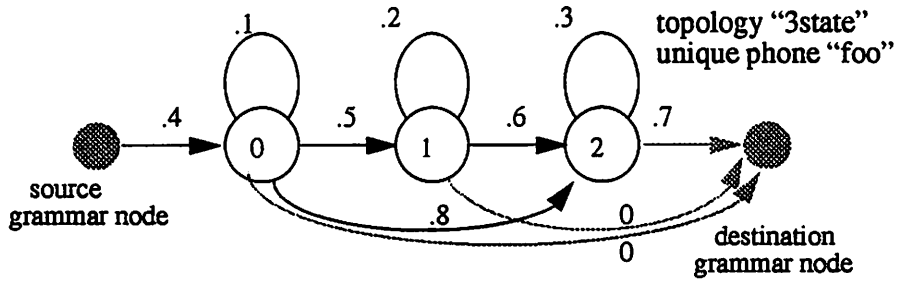


Figure 67: Example Phone Topology

The topology section of the HMM file describes the prototype phone topologies, and the example in Figure 67 is represented in the following way:

```
topology 3state 3
state[0] -1 0
state[1] 0 1
state[2] 0 1 2
```

Each topology description starts with the keyword "topology", followed by the name of that topology and the number of states. The following lines describe the predecessor states of each state in the prototype topology. By default, every state has a transition to the destination grammar node, therefore this transition is not mentioned. A transition from the source grammar node is indicated with the value "-1".

The next section of the HMM file describes the states of unique phones. The unique phone "foo" in Figure 67 uses the topology "3state", and it is be described using the following syntax:

```
instance foo 3state
state[0] -1 -log(.4) -log(.1)
OutputPDF 256 4
{
prob0_1 prob0_2 ... prob0_256
prob1_1 prob1_2 ... prob1_256
prob2_1 prob2_2 ... prob2_256
prob3_1 prob3_2 ... prob3_256
}
state[1] -1 -log(.2) -log(.5)
OutputPDF 256 4 ...
```

The first line defines the unique phone "foo" and references the topology "3state". Then, the transition probabilities for that state are described. The first probability value corresponds to the destination grammar node transition, and the value -1 corresponds to probability 0 (it is necessary to specify a keyword, because the probabilities are represented using the negative logarithm and probability 0 is $+\infty$). The probabilities that follow next correspond to transitions from predecessors, and they occur in the same sequence as in the phone topology section. The next line specifies the size and the number of probability distributions associated with that unique state. In the example of Figure 67, we assume that each state has 4 distributions, and each distribution has 256 probabilities. The probabilities inside the square brackets describe these distributions, starting from the feature vector values 0 through 255 of observation 0, to feature vector value 0 through 255 of observation 3.

9. 1. 2. HMM Representation in the Memories of the Speech Recognition Hardware

The program *Load_Hmm_File()* (see appendix) reads the HMM file and loads the HMM parameters into the topology memories of the two Viterbi processors, the transition probabilities of the two subsystems on the Viterbi board, and the memory banks on the distribution board.

First, the program reads the prototype topologies and loads them onto the topology memories (64 x 11) on the Viterbi processors of the Viterbi board. In these memories, the topology information of a state resides in one memory location, and the description of the first state of a prototype topology has to start at an address location that is a multiple of 4. This start address is used to reference to this topology, and the constraint that this address has to be a multiple of 4 makes it possible to use only 4 bits for that reference. Assuming that the example topology of Figure 67 is loaded starting at address 0 in the topology memory, it would be represented in the following way:

address (binary)	state	EndFlag	source grammar node transition?	third transition offset	second transition offset	first transition offset
000000	s1	0	1	0	0	x
000001	s2	1	0	1	1	0
000010	s3	0	0	2	1	0
000011			unused	unused	unused	unused
000100	t1		...	...	...	...

In this description, a transition offset of 0 means that the state has a self loop, and the maximum offset is 7 (111_{bin}). Whenever the state has a transition from the source grammar node, the first transition offset becomes a "don't care". During recognition, the Viterbi processor sequentially reads the states of the prototype topology, and the end of a topology is indicated with the EndFlag. Because of pipelining effects, this endflag is set at the state before the last state. In the above example, he next starting address that can be used for the description of the next phone topology is 000100_{bin}.

After all prototype topologies have been loaded, *Load_Hmm_File()* writes the transition probability memories and the output distribution memories. In the transition probability memory, each state occupies one memory location that contains transition probabilities from all 3 predecessor states and a transition probability to the destination grammar node. The states of unique phones are in consecutive memory addresses, and the address of the first state defines the identification of the unique phone. Transition probabilities are coded by quantizing the negative logarithm to 4 bits (see 4. 2. 1.). Probability 0 corresponds to 1111_{bin}. Assuming that the example unique phone "foo" (Figure 67) has the phone identification 64, the transition probabilities are represented in the following way in memory:

address	destination grammar node transition probability	transition probability from third predecessor	transition probability from second predecessor	transition probability from first predecessor
63	...	...	...	...
64	1111	1111	$-\log(.1)\&f0_{hex}$	$-\log(.4)\&f0_{hex}$
65	1111	1111	$-\log(.5)\&f0_{hex}$	$-\log(.2)\&f0_{hex}$

address	destination grammar node transition probability	transition probability from third predecessor	transition probability from second predecessor	transition probability from first predecessor
66	- log(.7)&f0$_{hex}$	- log(.8)&f0$_{hex}$	- log(.6)&f0$_{hex}$	- log(.3)&f0$_{hex}$
67	...	...	...	...

If a state has a transition from a source grammar node, the transition probability from that node is coded in the field "transition to first predecessor".

The probability values of the output distributions are loaded onto the 4 banks of the distribution board. The unique state address, which is a part of the output distribution memory address (see Figure 55), corresponds to the memory address of the transition probability memory.

The source code for the loader program (loader.c) is listed in the appendix.

9. 1. 3. Representation of the Digit Recognition Model

The vocabulary of the connected digit recognition task consists of 13 words, which are the words *zero*, *oh*, *oh-oh*, *one*, *two*, ... , *nine*, and *pause*. Because of the small vocabulary, it is not necessary to represent a word using prototype phones that are shared between the words. There is enough training data to train the words without re-using phonetic units, and it is no problem to store the parameters of such a small vocabulary without using the hierarchy level of phone instances. Therefore, each word is modelled as a whole in the hierarchy level of unique phones. These words, however, share 4 unique topologies:

2-state topology: pause
10-state topology: oh
15-state topology: oh-oh, one, two, three, four, five, six, eight, nine
20-state topology: zero, seven

All prototype topologies have the same structure (Figure 68), and their only difference is the number of states.

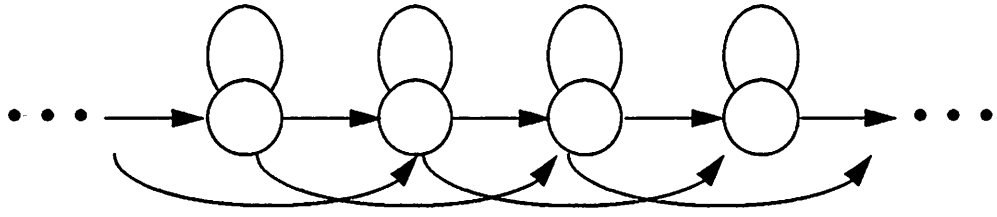


Figure 68: Topology for the Digit Recognition Task

9.2. System Control During Recognition

During recognition, the Heuricon board controls the three boards of the speech recognition system using the interrupt lines and the communication software shown in Figure 69.

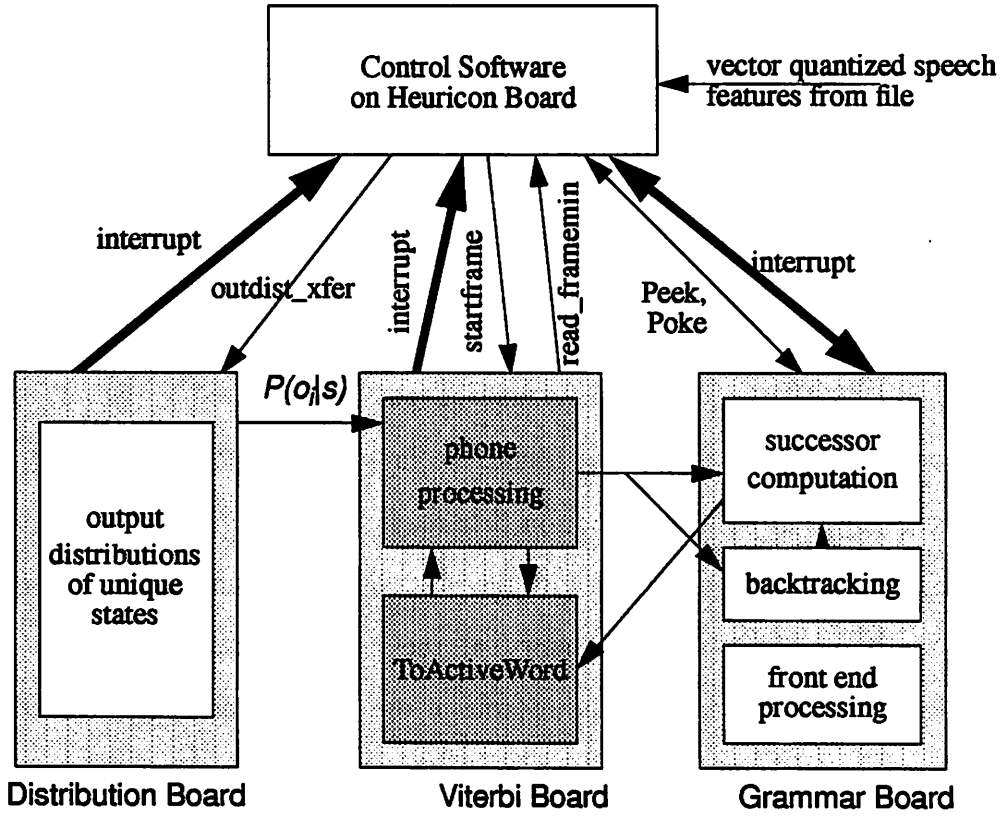


Figure 69: System Control During Recognition

The Heuricon board can initiate a transfer of output probabilities from the distribution board to the Viterbi board using the function *outdist_xfer*, and it can activate the Viterbi board using the function *startframe* (for a listing of these functions, see appendix). The algorithm for successor processing and backtracking is implemented on processor 2 on the DSP board, and this processor constantly polls its input FIFO for active phone instances that are send from the Viterbi board. Since in this digit recognition task any word can follow a given word, the DSP activates all words of the vocabulary each time a word was sent from the Viterbi chip, using the received destination grammar node probability as the source grammar node probability for the successor words. After the Viterbi board finished processing all words, the DSP that implements the grammar algorithm gets interrupted and sends an EndFlag to the Viterbi board. After that, it continues monitoring the input FIFO. Thus, the successor computation process needs no interaction by the Heuricon board.

After the Viterbi board received the EndFlag from the DSP board, it interrupts the Heuricon board. Also, the distribution board indicates to the Heuricon board the completion of a probability transfer with an interrupt. After the Heuricon board received the two interrupts, it reads the best probability that occurred in that frame from the Viterbi chip, and initiates the next frame.

After the last frame of a sentence was read (as indicated by end of file) and the Viterbi board has processed the last frame, the function *Get_Result* is called to initiate the backtracking routine. This function first reads the active phone instance memory, gets the backtrack tag of the word with the best source grammar node probability, and writes it onto the dual ported memory of the grammar DSP. Then, the DSP gets interrupted and the interrupt service routine of the DSP performs backtracking, writes the result into the dual ported memory, and interrupts the Heuricon board after completion. Then, the Heuricon board can read the most likely path and display the result.

The source codes for the system control program (system_control.c) and successor processing and backtracking (grammar.c) are listed in the Appendix.

9.3. Recognition Results

To verify the hardware, 924 sentences with a total of 5,160 words (digits) were recognized and compared to the recognition results that were obtained on a computer with double precision arithmetic. The HMM was trained using 55 male and 57 female speakers, 77 sentences for each speaker. The recognition was performed for 12 randomly selected male speakers that were not in the training set. Table 5 shows the results that were obtained with the recognition hardware.

speaker	ins	del	sub	sentence error	word error	sentence error [%]	word error [%]
ae	2	0	1	2	3	**2.6**	**.7**
aj	0	2	1	3	3	**3.9**	**.7**
al	12	0	0	12	12	**16**	**2.8**
aw	0	1	2	3	3	**3.9**	**.7**
bd	0	0	1	1	1	**1.3**	**.2**
cb	0	0	0	0	0	**0**	**0**
cf	7	0	1	6	8	**10**	**1.9**
cr	1	2	0	3	3	**3.9**	**.7**
dl	4	1	0	4	5	**6.5**	**1.2**
dn	7	0	1	8	8	**10**	**1.9**
eh	2	1	0	3	3	**3.9**	**.7**
el	7	0	1	7	8	**3.9**	**.7**
total	42	7	8	52	57	**5.6**	**1.1**

Table 5: Recognition Results using the Hardware

10

Conclusions

The system described in this thesis has been implemented and tested. There are two working versions of the hardware, one at SRI (Stanford Research Institute, Menlo Park), and another version at UCB for research purposes. The system at SRI is used for an interactive database query system where the user can ask questions to get information about airline travel (ATIS = airline travel information system), for example, flight schedules or availability. The speech input is recognized by the recognition hardware, and then passed to a natural language back end. This back end extracts the meaning of the question and generates a database query for the ATIS database, which in turn gives the answer. For successor processing, the SRI system uses a commercial board that has two TMS320C30 processors (SkyBolt). Front-end processing is done on an add-on board for an IBM PC. The commercial dsp board, however, has an architecture that does not support successor processing: the interface to the Viterbi board is not customized, therefore the bandwidth is not high enough to support complicated grammars or large vocabularies. SRI now plans to build three more systems (using the custom dsp boards), and these systems will be given to speech recognition researchers.

The architecture of the speech recognition system is expandable to even larger tasks (larger vocabularies, multiple users), continuous output probabilities, and a full custom grammar board.

Speech recognition systems for larger tasks or multi-user operation can be necessary in personal communication systems (PCS, [Bur91]). Here, portable hardware can have a wireless link to a powerful "speech server" that is centrally located and can serve multiple users simultaneously. For that, the recognition hardware processes different observations (from multiple users) within each frame. The only change that might be necessary in the current hardware is, that the output probability memory has to be expanded because probabilities of several different observations have to be stored.

In the current system, we use discrete output probabilities that are stored on the distribution board, and the probabilities corresponding to an observation in a frame are loaded to the SRAM output probability memory on the Viterbi board. However, it has been shown that recognition accuracies can be improved if the output probability distributions are continuous (section 2. 3. 3., [Bah90]). For that, the distribution board can be replaced by a board that can download output probabilities from continuous distributions: instead of reading probabilities from memory, the board reads parameters of gaussians and computes the output probabilities using tied mixtures [Bah90]. In fact, the next board that is being developed for this system is a tied mixture board.

References

[Bah81]: L. Bahl, R. Bakis, P. Cohen, A. Cole, F. Jelinek, B. Lewis, R. Mercer, "Speech Recognition of a Natural Text Read as Isolated Words", Proc. ICASSP 81: 1981 International Conference on Acoustics Speech and Signal Processing, April 1981

[Bam90]: P. Bamberg, Y. Chow, L. Gillick, R. Roth, D. Sturtevant, *"The Dragon Continuous Speech Recognition System: A Real-Time Implementation"*, Proc. of the Speech and Natural Language Workshop, June 1990, pp 78-81.

[Bis89] R. Bisiani, T. Anantharaman, L. Butcher: *"BEAM: An Accelerator for Speech Recognition"*, IEEE International Conference on Acoustics, Speech, and Signal Processing, 1989, S15.3, pp782-784

[Bur91]: A. Burstein, A. Stölzle, R. Brodersen: *"Using Speech Recognition in a Personal Communications System"*, submitted to the ICC Conference, session 841 (personal communications)

[Chat89]: S, Chatterjee, P. Agrawal, *"Connected Speech Recognition on a Multiprocessor Pipeline"*, IEEE International Conference on Acoustics, Speech, and Signal Processing, 1989, S15.1, pp774-778.

[Cha91]: A. Chandrakasan, S. Sheng, R. Brodersen: "Low Power CMOS Digital Design", submitted to the IEEE Journal of Solid State Circuits, Oct. 1991

[Gil90]: L. Gillick, R. Roth: *"A Rapid Match Algorithm for Continuous Speech Recognition"*, Proc. of the Speech and Natural Language Workshop, June 1990, pp 170-172.

[Gli87]: S. Glinski, T. Lalumia, D. Cassiday, T. Koh, C. Gerveshi, G. Wilson, J. Kumar, *"The Graph Search Machine (GSM): A VLSI Architecutre for Connected*

Speech Recognition and Other Applications", Proceedings of the IEEE, VOL. 75, No. 9, September 1987, pp1174-1184

[Gra84]: R. Gray, "*Vector Quantization*", IEEE ASSP magazine 1(2), April 1984, pp. 4-29

[Ita74]: F. Itakura, "*Minimum Prediction Residual Principle Applied to Speech Recognition*", IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP-23(1); February 1975, pp. 67-72

[Jel85]: F. Jelinek, "*The Development of an Experimental Discrete Dictation Recognizer*", Proc. of the IEEE, Nov. 1985, pp. 1616-1624

[Kav86]: R. Kavaler: "*The Design and Evaluation of a Speech Recognition System for Engineering Wordstations*", PhD thesis, University of California at Berkeley, May 1986, UCB/ERL M86/39

[Lee88]: K.Lee, H. Hsiao-Wuen: "*Large Vocabulary Speaker-Independent Continuous-Speech Recognition Using HMM*", Proc. ICASSP 88: 1988 International Conference on Acoustics Speech and Signal Processing, New York, April 1988, pp. 123-126

[Lee89]: K. Lee: "*Automatic Speech Recognition: The Development of the SPHINX System*", Kluwer internatioanl series in engineering and computer science, ISBN 0-89383-296-3, 1989.

[Lee90]: K. Lee: "*Context-Dependent Phonetic Hidden Markov Models for Speaker-Independent Continuous Speech Recognition*", IEEE Trans, on Acoustics, Speech and Signal Processing, Vol. 38, pp. 599 - 609, Apr. 1990

[Les75]: V. Lesser, R. Fennel, L. Erman, R. Reddy, "The HEARSAY II Speech Understanding System", IEEE Transactions on Scoustics, Speech, and Signal Processing ASSP-23(1), Feb. 1975, pp. 11-24

[Lin89]: H.-D. Lin, D. Messerschmitt: *"High Speed Viterbi Decoding"*, Submitted to IEEE Trans. on Communications, 1989

[Murv89]: H. Murveit et al, *"SRI's DECIPHER System"*, Proc. of the Speech and Natural Language Workshop, pp. 238 - 242, Feb. 1989

[Murv91]: H. Murveit, J. Butzberger, M. Weintraub, *"Speech Recognition in SRI's Resource Management and ATIS System"*, Proceedings of the Speech and Natural Language Workshop, Feb. 1991, pp. 94-100.

[Ost90]: V. Dikalakis, M. Ostendorf, J. Rohlicek, *"Fast Search Algorithms for Connected Phone Recognition Using the Stochastic Segment Model"*, Proceedings of the Speech and Natural Language Workshop, June 1990, pp. 173-178

[Pal90]: D. Pallett, J. Fiscus, J. Garofolo: *"DARPA Research Management Benchmark Test Results"*, Proc. of the Speech and Natural Language Workshop, June 1990, pp. 298-305.

[Pal91]: D. Pallett, "DARPA Resource Management and ATIS Benchmark Test Poster Session", Proceedings of the Speech and Natural Language Workshop, Feb. 1991, pp. 49-58.

[Pri88]: P. Price, W. Fisher, J. Bernstein, D. Pallet, *"A Database for Continuous Speech Recognbition in a 1000-Word Domain"*, IEEE International Conference on Acoustics, Speech, and Signal Processing, April 1988

[Qué89]: G. Quénot, J. Gauvain, J. Gangolf, J. Mariani, *"A Dynamic Programming Processor for Speech Recognition"*, IEEE Journal of Solid State Circuits, Vol. 24, No. 2, April 1989, pp. 349-357.

[Qué91]: personal communication with Georges Quénot, Nov. 1991.

[Rab78]: L. Rabiner, R. Schafer, *"Digital Processing of Speech Signals"*, Prentice Hall signal processing series, ISBN 0-13-213603-1, 1918, chapter 8

[Rab79]: L. R. Rabiner, S. Levinson, A. Rosenberg, J. Wilpon, *"Speaker-Independent Recognition of Isolated Words using Clustering Techniques"*, IEEE Transactions on Acoustics, Speech, and Signal Processing ASSP-27(4), Aug. 1979, pp. 336-349

[Rab86]: L R Rabiner, B H Juang, *"An Introduction to Hidden Markov Models"*, IEEE ASSP Magazine, Jan. 1986, pp. 4-16

[Rab89]: L. R. Rabiner, B. H. Juang, *"A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition"*, Proceedings of the IEEE, pp. 257-2586, Feb. 1989.

[Rab91]: J. Rabaey, C. Chu, P. Phu, M. Potkoniak, *"Fast Prototyping of Datapath -Intensive Architectures"*, IEEE Design & Test of Computers, June 1991, pp. 40-51.

[Roe89]: D. Roe, A. Gorin, P. Ramesh, *"Incorporating Syntax into the Level-Building Algorithm on a Tree-Structured Parallel Computer"*, IEEE International Conference on Acoustics, Speech, and Signal Processing, 1989, S15.3, pp778-784.

[Rou87]: S. Roucos, M. Dunham, *"A Stochastic Segment Model for Phoneme-Based Continuous Speech Recognition"*, IEEE International Conference on Acoustics Speech and Signal Processing, Apr. 1987, pp. 73-76.

[Sak79]: H. Sakoe, *"Two-Level DP-Matching - A Dynamic Programming-Based Pattern Matching Algorithm for Connected Word Recognition"*, IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP-27(6), pp588-595, Dec. 1979

[Sak78]: H. Sakoe, S. Chiba: *"Dynamic Programming Algorithm Optimization for Spoken Word Recognition"*, IEEE Transactions on Acoustic, Speech and Signal Processing, vol. ASSP-26, pp. 43-49, Feb. 1987

[Shu91]: C. Shung, R. Jain, K. Rimey, E. Wang, M. Shrivastava, B. Richards, E. Lettang, S. Azim, L. Thon, P. Hilfinger, J. Rabaey, R. Brodersen, *"An Integrated CAD System for Algorithm-Specific IC Design"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 10, No. 4, Apr. 1991, pp.447-475

[Sch87]: Y.L. Chow, M.D. Dunham, O.A. Kimball, M.A. Krasner, G.F. Kubala, J. Makhoul, P.J. Price, S. Roucos, R.M. Schwartz: *"BYBLOS: The BBN Continuous Speech Recognition System"*, Proc ICASSP 87: 1987 International Conference on Acoustics Speech and Signal Processing, Piscataway, N.J., April 1987, pp. 89-92

[Sch89]: R.M. Schwartz, C. Barry, Y.L. Chow, A. Derr, M.-W. Feng and O. Kimball, F. Kubala, J. Makhoul , J. Vandegrift: *"The BBN Byblos Continuous Speech Recognition System"*, Proc Speech and Natural Language Workshop, Philadelphia, Feb, 1989, pp. 94-99

[Stö87]: A. Stölzle, W. Drews, R. Laroia, J. Pandel, A. Schumacher: *"A CMOS Processor for a 1000 Word Speech Recognition System"*, Proceedings of the IEEE Custom Integrated Circuits Conference, Portland, Or, May 1987, pp. 559-562

[Stö90]: A. Stölzle et al., *"A Flexible VLSI 60,000 Word Real Time Continuous Speech Recognition System"*, IEEE Press book: VLSI Signal ProcessingIV, ISBN 0-87942-271-8, Chapter 27, pp 274 - 284, Nov. 1990

[Stö91]: A. Stölzle et al, "*Integrated Circuits for a Real-Time Large-Vocabulary Continuous Speech Recognition System*", IEEE Journal of Solid State Circuits, vol. 26, no.1, pp. 2-11, Jan. 1990

[TI88]: "*Third Generation TMS320 User's Guide*", Texas Instruents Incorporated, document number SPRZ048, 1988.

[Vw90]: Vxworks manual

[Wai88]: A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, K. Lang, "Phoneme Recogniton using Time-Delay Neural Networks", IEEE International Conference on Acoustic, Speech and Signal Processing, May 1989.

[Wai90]: "*Readings in Speech Recognition*", edited by A. Waibel and K. Lee. Morgan Kaufmann Publishers, Inc., ISBN 1-55860-124-4, 1990.

[Wai90a]: "*Template Based Approaches, Introduction*", Readings in Speech Recognition, edited by A. Waibel and K. Lee. Morgan Kaufmann Publishers, Inc., ISBN 1-55860-124-4, 1990, pp 113-114

[Wai90b]: "*Knowledge Based Approaches, Introduction*", Readings in Speech Recognition, edited by A. Waibel and K. Lee. Morgan Kaufmann Publishers, Inc., ISBN 1-55860-124-4, 1990, pp 198-199

[Wai90c]: "*Connectionist Approaches, Introduction*", Readings in Speech Recognition, edited by A. Waibel and K. Lee. Morgan Kaufmann Publishers, Inc., ISBN 1-55860-124-4, 1990, pp 371-372

[Wai90d]: "*Stochastic Approaches, Introduction*", Readings in Speech Recognition, edited by A. Waibel and K. Lee. Morgan Kaufmann Publishers, Inc., ISBN 1-55860-124-4, 1990, pp 263-264

[Wai90e]: *"Language Processing for Speech Recognitin, Introduction"*, Readings in Speech Recognition, edited by A. Waibel and K. Lee. Morgan Kaufmann Publishers, Inc., ISBN 1-55860-124-4, 1990, pp 447-448

[Way91]: C. Wayne, "A Snapshot of Two DARPA Speech and Natural Language Programs", Proceedings of the Speech and Natural Language Workshop, Feb. 1991, pp. 403-404.

[Whi76]: G. White, R. Nely, *"Speech Recognition Experiments with Linear Prediction, Bandpass Filtering and Dynamic Programming"*, IEEE Transactions on Acoustic, Speech and Signal Processing, vol. ASSP-24, pp. 183-188, April 1976

[Zue85]: V. Zue, *"The Use of Speech Knowledge in Automatic Speech Recognition"*, Proceedings of the IEEE, Nov. 1985, pp. 1602-1615

Appendix

Program to load HMM parameters onto the hardware: loader.c

```
/*
 *  File:   loader.c
 *  Author: tony
 *  Date:   Sept. 12, 1991
 *
 *  Description:
 *     This program loads the speech hardware boards.
 *     The single optional command-line
 *     argument  to this program is the pathname to
 *     a file in a Hmm file format to read.  Otherwise,
 *     the default filename, "Hmm.data" is read.
 *
 *
 */

/* Include declarations files. */

#include <sys/types.h>
#include "vxWorks.h"
#include "stdioLib.h"
#include "ioLib.h"
#include "../include/background.h"
#include "../include/grammar.h"
#include "../include/speech.h"

#define MIN_TRANS 7000
#define MAX_TRANS 7077888
#define MIN_OUTPROB 50237
#define MAX_OUTPROB 882554

#define OUTPROBMASK   0xff80000
#define TRANSPROBMASK 522240  /* 0xff << 11 */
#define MAX_STRING 80
#define MAX_NR_OF_UNIQUE_PHONES 200
#define MAX_NR_OF_TOPOLOGIES 20
#define abs(A)  ((A) > 0 ? (A) : -(A))
#define BG_PROB_MEM (u_short *) 0x40000000
#define BgAddress(bank,vector,addr)  (u_short *)((BG_PROB_MEM) + \
                              ((bank)<<24) + \
```

```
                        ((vector)<<11) + \
                        (((addr) & 0xf800)<<8) + \
                        ((addr) & 0x7ff))

typedef struct TOPO_REFERENCE
{
     char name[MAX_STRING];
     int address;
     int number_of_states;
}topo_reference;

typedef struct PHONE_REFERENCE
{
     char name[MAX_STRING];
     char topo_id[MAX_STRING];
     int topo_address;
     int phone_address;
     int number_of_states;
}phone_reference;

topo_reference *topo_table;
phone_reference *phone_table;

int topo_table_index = 0;
int phone_table_index = 0;

int Debug = 0;
int TransDebug = 0;
int TopoDebug = 0;
int Delay = 0;

void
A_write_trans(address, data) /* fun */
unsigned long address, data;
{
             unsigned long*vme_address;

             vme_address = (unsigned long *) A_LOAD_ADDRESS;
             *vme_address = address;
             { int i; for (i = 0; i < Delay; i++) ; }
             vme_address = (unsigned long *) A_WRITE_TRANSPROB;
             *vme_address = data;
             { int i; for (i = 0; i < Delay; i++) ; }
}

void
B_write_trans(address, data) /* fun */
unsigned long address, data;
{
             unsigned long*vme_address;

             vme_address = (unsigned long *) B_LOAD_ADDRESS;
             *vme_address = address;
             { int i; for (i = 0; i < Delay; i++) ; }
             vme_address = (unsigned long *) B_WRITE_TRANSPROB;
             *vme_address = data;
             { int i; for (i = 0; i < Delay; i++) ; }
}
```

```
void
A_write_phone(address, data) /* fun */
              unsigned long address, data;
{
              unsigned long *vme_address;

              vme_address = (unsigned long *) A_LOAD_ADDRESS;
              *vme_address = address;
              { int i; for (i = 0; i < Delay; i++) ; }
              vme_address = (unsigned long *) A_WRITE_NEWSTATEPROB;
              *vme_address = data;
              { int i; for (i = 0; i < Delay; i++) ; }
              vme_address = (unsigned long *) A_WRITE_NEWSTATEBACK;
              *vme_address = data;
              { int i; for (i = 0; i < Delay; i++) ; }
              vme_address = (unsigned long *) A_LOAD_ADDRESS;
              *vme_address = address;
              { int i; for (i = 0; i < Delay; i++) ; }
              vme_address = (unsigned long *) A_WRITE_PHONETOPOLOGY;
              *vme_address = data;
              { int i; for (i = 0; i < Delay; i++) ; }
}

void
B_write_phone(address, data) /* fun */
              unsigned long address, data;
{
              unsigned long *vme_address;

              vme_address = (unsigned long *) B_LOAD_ADDRESS;
              *vme_address = address;
              { int i; for (i = 0; i < Delay; i++) ; }
              vme_address = (unsigned long *) B_WRITE_NEWSTATEPROB;
              *vme_address = data;
              { int i; for (i = 0; i < Delay; i++) ; }
              vme_address = (unsigned long *) B_WRITE_NEWSTATEBACK;
              *vme_address = data;
              { int i; for (i = 0; i < Delay; i++) ; }
              vme_address = (unsigned long *) B_LOAD_ADDRESS;
              *vme_address = address;
              { int i; for (i = 0; i < Delay; i++) ; }
              vme_address = (unsigned long *) B_WRITE_PHONETOPOLOGY;
              *vme_address = data;
              { int i; for (i = 0; i < Delay; i++) ; }
}


void
enter_in_topo_table(name, address, number_of_states)
     char * name;
     int address;
     int number_of_states;
{
     strcpy((topo_table+topo_table_index)->name, name);
     (topo_table+topo_table_index)->address = address;
     (topo_table+topo_table_index)->number_of_states = number_of_states;
     topo_table_index++;
}
```

```
void
enter_in_phone_table(name, topo_id, topo_address, phone_address,
  number_of_states)
     char * name;
     char * topo_id;
     int topo_address;
     int phone_address;
     int number_of_states;
{
     strcpy((phone_table+phone_table_index)->name, name);
     strcpy((phone_table+phone_table_index)->topo_id, topo_id);
     (phone_table+phone_table_index)->topo_address = topo_address;
     (phone_table+phone_table_index)->phone_address = phone_address;
     (phone_table+phone_table_index)->number_of_states = number_of_states;
     phone_table_index++;
}

int
get_topo_reference(name)
     char * name;
{
     int i;

     for (i=0; i<topo_table_index;i++) {
               if (strcmp((topo_table+i)->name, name) == 0) {
                    return (i);
               }
     }
     printf(" get_reference: name %s not in topo_table!\n", name);
     return (-1);
}

int
scale_and_pack_transprobs(gndtransprob, firsttransprob, secondtransprob,
  thirdtransprob)
     int gndtransprob, firsttransprob;
     int secondtransprob, thirdtransprob;
{
     int transprob_data = 0;

     /*
      * scale transprobs and mask them.
      */

     if (gndtransprob != -1) gndtransprob = ((gndtransprob-MIN_TRANS) * 0xf) / (
  MAX_TRANS);
     else gndtransprob = 0xf;
     if (firsttransprob != -1) firsttransprob = ((firsttransprob-MIN_TRANS) *
  0xf) / (MAX_TRANS);
     else firsttransprob = 0xf;
     if (secondtransprob != -1) secondtransprob = ((secondtransprob-MIN_TRANS) *
  0xf) / (MAX_TRANS);
     else secondtransprob = 0xf;
     if (thirdtransprob != -1) thirdtransprob = ((thirdtransprob-MIN_TRANS) *
  0xf) / (MAX_TRANS);
     else thirdtransprob = 0xf;
```

```
    /*
     *pack these values into one word ...
     * SGndTransProb ThirdTransProb SecondTransProb FirstTransProb
     *     0000           0000            0000          0000
     */

    transprob_data |= (gndtransprob << 12);
    transprob_data |= (thirdtransprob << 8);
    transprob_data |= (secondtransprob << 4);
    transprob_data |= (firsttransprob);

    return(transprob_data);
}


int
setup_topology(fp, topo_address)
    FILE *fp;
    int topo_address;

{
    char *keyword = "topology";
    char topo_id[MAX_STRING];
    char state_id[MAX_STRING];
    int offset1, offset2, offset3;
    char buf[MAX_STRING];

    int number_of_states;
    int i, j;
    int topo_data;

    /*
     * get topo_id and the number of states in this topology
     * from the input stream
     */
    fscanf(fp, "%s %d\n", topo_id, &number_of_states);

    /*
     * for later reference, let's store this information in a
     * table. Later we have to retrieve topo_address and number_of_states
     * for a certain topo_id
     */
    enter_in_topo_table(topo_id,topo_address,number_of_states);

    if (TopoDebug) printf("setup_topology: topology %s, %d states:\n",
                          topo_id, number_of_states);

    /*
     * now, let's set up the topology memory for all the states
     * in this topology
     */
    for(j=0; j<number_of_states; ++j) {

            /*
             * initialize offsets and topo_data.
             * this is important if we have less than 3 offsets in
             * the input stream.
             */
            offset1= offset2= offset3= j;
```

```
  topo_data = 0;

  /*
   * get the name of the state and the offsets from the input
   * stream. I scan in a whole line because I don't know
   * how many offsets to expect
   */
  fgets(buf,MAX_STRING,fp);       /* get a line of input */
  sscanf(buf, "%s %d %d %d",state_id, &offset1, &offset2,
             &offset3);

  /*
   * convert to word that actually gets loaded into the topo memory
   *
   * EndFlag    GndTrans     3rd     2nd     1st
   *    0          0         000     000     000
   *
   * take care of the EndFlag, one state before the last state
   */
  if(j==(number_of_states - 2)) topo_data |= 0x400;

  /*
   * see if we have a source grammar node transition
   * this is coded with offset = -1 in the input stream
   */
  if(offset1 == -1) {
            /*
            * well, we have a grammar node transition, so let's set
            * the GndTrans flag
            */
       topo_data |= 0x200;
            /*
            * offset1 is now meaningless
            */
       offset1 = j;
  }

  /*
   * compute the relative distance of the predecessor
   * right now we have absolute transitions
   */
  offset1 = -(offset1 -= j);
  offset2 = -(offset2 -= j);
  offset3 = -(offset3 -= j);

  /*
   * pack the offsets into one word that gets downloaded
   * to the topology memories A and B
   */
  topo_data |= (((unsigned long)offset3 & 0x7) << 6);
  topo_data |= (((unsigned long)offset2 & 0x7) << 3);
  topo_data |=  ((unsigned long)offset1 & 0x7);

{ int k; for (k = 0; k < Delay; k++) ; }
  A_write_phone(topo_address + j, topo_data);
{ int k; for (k = 0; k < Delay; k++) ; }
  B_write_phone(topo_address + j, topo_data);

  if (TopoDebug) printf("setup_topology:\t\t%2d:\t0x%3x\n",
```

```
                                topo_address + j, topo_data);

        }
        /*
         * finished all the states for this topology
         * return pointer to next possible location in topo memory (mod 4)
         */
        return((topo_address + j) + ( 4- (topo_address + j) % 4));
}

int
load_output_prob(unique_state_address, bank, vector, output_prob)
        int unique_state_address, bank, vector, output_prob;
{

        *BgAddress(bank, vector, unique_state_address) = (u_short) output_prob;
        return(0);
}

int
read_and_load_opdf(fp, unique_state_address)
        FILE *fp;
        int unique_state_address;
{
        char *keyword = "OutputPDF";
        char *curly_bracket = "{";
        int codebook_size, number_of_codebooks;
        unsigned int output_prob;
        int i, j;

        fscanf(fp, "%s %d %d\n", keyword, &codebook_size, &number_of_codebooks);
        fscanf(fp, "%s\n", curly_bracket);/* just dummy to advance read-pointer */

        for(j=0; j<number_of_codebooks; j++) {
                  for(i=0; i<codebook_size; i++) {
                        fscanf(fp, "%d", &output_prob);
                        /* scale */
                        output_prob = ((output_prob-MIN_OUTPROB)*0x3fe)
                             /(MAX_OUTPROB);

                        load_output_prob(unique_state_address, j, i, output_prob);
                  }
        }
        fscanf(fp, "%s\n", curly_bracket);/* just dummy to advance read-pointer */
}

int
setup_unique_states(fp, unique_state_address)
        FILE *fp;
        int unique_state_address;
{
        char *keyword = "instance";
        char unique_phone_id[MAX_STRING];
        char topo_id[MAX_STRING];
        char state[MAX_STRING];
        char buf[MAX_STRING];
        unsigned int    gndtransprob=-1, firsttransprob=-1;
        unsigned int secondtransprob=-1, thirdtransprob=-1;
        unsigned int transprob_data = 0;
```

```
    int topo_address;
    int number_of_states, i, j;

    /*
     * get the unique phone id and the pointer to the unique topology
     * from the input stream
     */
    fscanf(fp, "%s %s\n", unique_phone_id, topo_id);
    /*
     * now, using the unique topology pointer, we access the
     * topo_table to find out where this topology is stored and
     * how many states this unique phones has
     */
number_of_states = (topo_table + get_topo_reference(topo_id))->number_of_states;
    topo_address = (topo_table + get_topo_reference(topo_id))->address;
    /*
     * this information gets entered in the phone_table for later reference.
     * This table stores all the pointers necessary to completely access
     * the structures necessary for that phone
     */
    enter_in_phone_table(unique_phone_id,topo_id,
  topo_address,unique_state_address,number_of_states);

    printf("loading unique phone %s:\t",unique_phone_id);
    /*
     * now, let's take care of all  the unique states that
     * belong to this unique phone:
     */
    for (i=0; i<number_of_states; i++) {
              /*
               * initialize transprobs to infinity. This is important when
               * there are less than 4 transprobs in the input stream
               */
              gndtransprob = -1;
              firsttransprob = -1;
              secondtransprob = -1;
              thirdtransprob = -1;

              /*
               * get the line of input that contains the trans probs
               */
              fgets(buf,MAX_STRING,fp);
              sscanf(buf, "%s %d %d %d %d", state, &gndtransprob,
                        &firsttransprob, &secondtransprob, &thirdtransprob);

              /*
               * scale probabilities for maximum dynamic range in HW and
               * pack them into one word that gets loaded to the HW
               */
              transprob_data = scale_and_pack_transprobs(gndtransprob,
                        firsttransprob, secondtransprob, thirdtransprob);
              printf(".");

              /*
               * transprob_data is now ready to be downloaded to the
               * transition probability memories A and B.
               * Address = unique_state_address
               */
```

```
                A_write_trans(unique_state_address, transprob_data);
                B_write_trans(unique_state_address, transprob_data);

                /*
                 * the next inputs from the file set up the output distributions
                 * of that state. For that we call a seperate routine ...
                 */
                read_and_load_opdf(fp, unique_state_address);

                /* this state is now completely downloaded, increment the
                 * unique state counter (= unique state address) and go on
                 * to the next state
                 */
                unique_state_address++;
        }
        printf("\n");
        return(unique_state_address);
}


Load_Hmm_File(filename)
        char *filename;
{
        char *keyword = "just some crazy caracters";
        char *topology = "topology";
        char *instance = "instance";
        int topo_add = 4;
        int uniquestate_add = 0;

        extern topo_reference *topo_table;
        extern phone_reference *phone_table;
        FILE *fp;

        /*
         * initialize reference table
         */
        if ((topo_table = (topo_reference *)malloc(sizeof(topo_reference) *
                                MAX_NR_OF_TOPOLOGIES)) == NULL)
                {
                     printf("malloc failed on topo_table!\n");
                     return;
                }


        if ((phone_table = (phone_reference *)malloc(sizeof(phone_reference) *
                                MAX_NR_OF_UNIQUE_PHONES)) ==
                NULL) {
                 printf("malloc failed on phone_table!\n");
                 return;
        }


        /*
         * Open the Hmm file.
         */
        if ((fp = fopen(filename, "r")) == NULL) {
                 printf("Error opening \"%s\"!\n", filename);
                 return;
        }
```

```
    while (fscanf(fp, "%8s", keyword) != 0) {
              if (strcmp(keyword, topology) == 0) {
                   topo_add = setup_topology(fp, topo_add);
              }
              else if (strcmp(keyword, instance) == 0) {
                   uniquestate_add = setup_unique_states(fp, uniquestate_add);
              }
              else printf("keyword neither topology nor instance: %s\n",
  keyword);
    }

    printf("\n  done loading the recognition hardware!\n");
/*
    print_topo_table();
    print_phone_table();
*/
}
```

Program for successor processing and backtracking: grammar.c

```
#include "grammar.h"
#include "tms320c30.h"
#include "vmeipc.h"

#define MAX_NR_OF_TAGS 0x1000
#define MAX_NR_OF_WAS  0x1000
#define MAX_LIST_SIZE  0x8000

volatile unsigned int * fifo = PROBADDRESS;
volatile unsigned int backtrack[MAX_LIST_SIZE];
volatile unsigned int wordarc_topos[MAX_NR_OF_WAS];
volatile unsigned int prob, oldtag, wordarc, tag=0;
volatile unsigned int bestprob = 0xffff;
volatile unsigned int bestwa, besttag;

main()
{
    volatile unsigned int * in_flag = INPUTEMPTYFLAGSADDRESS;
    volatile unsigned int GndProb = 0xfff0;

    setup_tms();
    setup_wa_topologies();

    PollForNewFrame_ClearIFreg(in_flag); /* poll input fifo */

    while (TRUE) {
              if ((*in_flag & 0x7) ==0) {
                        /*
                        * we can receive a wordarc from the HMM board
                        */
                        ReceiveWordArc();
                        if (bestprob > prob) {
                            bestprob = prob;
                            bestwa = wordarc;
                            besttag = oldtag;
                        }

                        /*
```

```
                        * store the oldtag pointer and get another one.
                        * I use the 16 lower bits for the tag and the 16 upper
                        * bits for the WordArc ID.
                        */
                        *(backtrack + tag)= ((wordarc << 16) | (oldtag &
                           0xffff));

                        /*
                        * send the wordarc we just received, using the same
                        * GndProb, the same WA, a modified tag and the topology
                        */
                         SendWordArc(tag);

                        /*
                        * increment the tag for the next structure received ...
                        */
                        tag++;
                }

                else {
                     /* the input fifo is empty. Is it the end of a frame? */
                     if ((GetIF() & 0x8) == 0x8){
                              /* it's the end of this frame */
                              SendLastWordarc();
                              PollForNewFrame_ClearIFreg(in_flag);
                     }
                     else; /* not eof, so keep on polling */
                }
     }
}

void
ReceiveWordArc()
{
     volatile unsigned int * i = 0;
     *IN_POPADDRESS = DUMMY;       /* pop input fifo to get the next WA */
     prob     = *fifo;
     oldtag   = *(fifo-1);
     wordarc  = *(fifo-2);
}

void
SendWordArc(tag)
     volatile unsigned int tag;
{
     unsigned int i;

     for (i=0; i< 13; i++) {
               *fifo     = prob;
               *(fifo-1) = tag;
               *(fifo-2) = i;
               *(fifo-3) = wordarc_topos[i];
     }
}

void
SendLastWordarc()
{
```

```
    *MY_DPRAM_BASE_ADDRESS = bestprob;
    *(MY_DPRAM_BASE_ADDRESS+1) = bestwa;
    *(MY_DPRAM_BASE_ADDRESS+2) = besttag;

    bestprob = 0xffff;

    *fifo     = -1;       /* Gr_Prob */
    *(fifo-1) = -1;       /* Gr_Tag  */
    *(fifo-2) = -1;       /* Gr_WA   */
    *(fifo-3) = 0x1fffff; /* Gr_Topo + EndFlag*/

}

void
PollForNewFrame_ClearIFreg(in_flag)
    volatile unsigned int * in_flag;
{
    unsigned int i=0;

    /*
     * poll input for non-empty status
     */
    while((*in_flag & 0x7) != 0)  i++;

    /*
     * ok, input fifo now has some data. In this case,
     * Vit_Active is high and we can reset the IF register
     * so that Vit_Active(-) generates a new interrupt
     */
    ClrCpuRegBits(IF,IF_INT3);
}

void
setup_wa_topologies()
{
    extern volatile unsigned int wordarc_topos[MAX_NR_OF_WAS];

    /*
     * the wordarc topology is set up the following way:
     * topo_address >> 2  bits 16..19
     * unique state address bits 0..15
     */
    wordarc_topos[0] = 0x20000; /* word_0,   topo= 4, unique state =   0 */
    wordarc_topos[1] = 0x5000a; /* word_1,   topo=16, unique state =  10 */
    wordarc_topos[2] = 0x50019; /* word_2,   topo=16, unique state =  25 */
    wordarc_topos[3] = 0x50028; /* word_3,   topo=16, unique state =  40 */
    wordarc_topos[4] = 0x50037; /* word_4,   topo=16, unique state =  55 */
    wordarc_topos[5] = 0x50046; /* word_5,   topo=16, unique state =  70 */
    wordarc_topos[6] = 0x50055; /* word_6,   topo=16, unique state =  85 */
    wordarc_topos[7] = 0x90064; /* word_7,   topo=32, unique state = 100 */
    wordarc_topos[8] = 0x50078; /* word_8,   topo=16, unique state = 120 */
    wordarc_topos[9] = 0x50087; /* word_9,   topo=16, unique state = 135 */
    wordarc_topos[10]= 0x90096; /* word_Z,   topo=32, unique state = 150 */
    wordarc_topos[11]= 0x100aa; /* word_P,   topo= 0, unique state = 170 */
    wordarc_topos[12]= 0x500ac; /* word_0_0, topo=16, unique state = 172 */
}

/*
 * Interrupt handler: performs backtracking at the end of a sentence
```

```
 * and writes the result into the dual ported memory, starting at  0x1000.
 *
 * If the backtrack opperation is complete, it interrupts the heuricon board
 * to tell it that the results are there.
 */
void c_int01()
{

     unsigned int i= 0;
     unsigned int prev_pointer;
     unsigned int * result = MY_DPRAM_BASE_ADDRESS;

     /*
      * perform a dummy read using IACK to clear external latch
      * we pick a location in the internal RAM for this dummy read
      */
     asm("        PUSH DP");
     asm("        LDI 80h, DP");
     asm("        IACK @9800h");
     asm("        POP  DP");

     /*
      * clear the internal flag for INT0
      */
     ClrCpuRegBits(IF,IF_INT0);

      /*
      * first, get pointer to the best wordarc at the end of the sentence.
      * we assume, this pointer is in the DPRAM at location 0x1000
      */
     prev_pointer = *result++;

     i = 0;

     while ((i<0x1000) && (prev_pointer != 0)) {
               *result++ = (*(backtrack + prev_pointer) >> 16);
               prev_pointer = (*(backtrack+prev_pointer) & 0xffff);
               i++;
     }

     *MY_DPRAM_BASE_ADDRESS = i;

     /* send an interrupt to vme ... */
     p_vmeipc->ireg->v = 1;

     /* reset tag for the next sentence */
     tag = 0;
}

void
setup_tms()
{
     void c_int01(void);

     /*
      * note: by default the software waitstate generation is set to 111
      * set wait state generation of expansion bus to SWW = 11,
      * at least 1 wait state (because of input fifo)
      */
```

```
    this_cpu.peribus->bus_control.expansion.f.sww = 3;
    this_cpu.peribus->bus_control.expansion.f.wtcnt = 1;

    /*
     * use external wait state generator for primary bus
     */
    this_cpu.peribus->bus_control.primary.f.sww = 0;

    /*
     * start the cache
     */
    SetCpuRegBits(ST,ST_CE);

    /*
     * install the interrupt handler for int0
     */
    this_cpu.vectortable->int0 = (unsigned int)c_int01;

    /*
     * enable int0
     */
    SetCpuRegBits(ST,ST_GIE);
    SetCpuRegBits(IE,IE_EINT0_CPU);

    /*
     * clear interrupt to vme ...
     */
    p_vmeipc->ireg->v = 0;

    /*
     * reset interface fifos
     */

    *RESETADDRESS = DUMMY;
    *RESETADDRESS = DUMMY;
    *RESETADDRESS = DUMMY;
    *RESETADDRESS = DUMMY;
    *RESETADDRESS = DUMMY;
}
```

Program for system control during recognition: systemcontrol.c

```
/*****************************************************************
 * Speech Recognition control code for the heuricon board
 *
 * Anton Stolzle, 9-13-91
 *****************************************************************/


#include "vxWorks.h"
#include "stdio.h"
#include "fioLib.h"
#include "ioLib.h"
#include "semLib.h"
#include "math.h"
#include "../grammar/grammar.h"
#include "tmsipc.h"
```

```
#define OUTDIST_SIZE 256
#define NUM_OF_UNIQUE_STATES 200
#define MAX_STRING 80
#define flip(A)  ((A) == 0 ? 1 : 0)

extern SEM_ID dist_board_int_sem, hmm_board_int_sem, dsp_board_back_sem;
extern void outdist_xfer();
extern void hmm_reset();
extern unsigned long A_read_framemin(), B_read_framemin();
extern void Clear_ActiveList();
extern void Clear_ActiveWord();
extern void A_aw_newframe(), B_aw_newframe();
extern void A_vit_newframe(), B_vit_newframe();
extern void A_load_framemin(), B_load_framemin();
extern void A_load_normvalue(), B_load_normvalue();
extern void A_load_prunoffset(), B_load_prunoffset();
extern void A_write_awmem();
extern int setup_hw_int();
extern unsigned long Get_Best();
extern unsigned long Print_OutProb();
extern unsigned long PrintState();
extern void MpeekProc(), PokeProc();
extern unsigned int PeekProc();

/*
extern feature_struct * feature_table;
extern int feature_readidx;
extern int feature_writeidx;
 */

int sent_debug = 0;

int
read_framemin(side)
     unsigned short side;
{
     if (side == 0) return(A_read_framemin());
     else if (side == 1) return(B_read_framemin());
     else  {
               printf("read_framemin: side not binary, %d\n", side);
               return(-1);
     }
}

void
startframe(side, framemin, norm, offset)
     unsigned short side;
     int framemin;
     int norm;
     int offset;
{
     if (side == 0) {
               if (sent_debug) printf("Starting side A, framemin = %4x, norm =
  %4x, offset = %4x\n",
                              framemin,norm,offset);
               A_load_framemin(framemin);
               A_load_normvalue(norm);
               A_load_prunoffset(offset);
               B_aw_newframe();
```

```
                A_vit_newframe();
        }
        else if (side == 1) {
                if (sent_debug) printf("Starting side B, framemin = %4x, norm =
   %4x, offset = %4x\n",
                                framemin,norm,offset);
                B_load_framemin(framemin);
                B_load_normvalue(norm);
                B_load_prunoffset(offset);
                A_aw_newframe();
                B_vit_newframe();
        }
}

int
get_speech_frame(fp, cep_ptr, dcep_ptr, egy_ptr, degy_ptr)
        FILE *fp;
        int *cep_ptr, *dcep_ptr, *egy_ptr, *degy_ptr;
{
        if ((fscanf(fp, "%d", cep_ptr) > 0) &&
                  (fscanf(fp, "%d", dcep_ptr) > 0) &&
                  (fscanf(fp, "%d", egy_ptr)  > 0) &&
                  (fscanf(fp, "%d", degy_ptr) > 0)){
                  return(0);
        }
        else return (EOF);
}


void
Get_Result(side)
        unsigned short side;
{
        unsigned long tag;
        unsigned long number_of_wordarcs, wordarc, i;
        unsigned int dpram = (unsigned int) MY_DPRAM_BASE_ADDRESS;

        /*
         * this is the end of a sentence, let's interrupt the grammar dsp and
         * get the backtrack pointers. First, find the grammar node with the
         * best probability ...
         */
        tag = Get_Best(side);
        printf("BackTracking, send tag %x to grammar ...\n", tag);
        /*
         * now, send tag, interrupt and wait 'till dsp is finished ...
         */
        PokeProc(GRAMMAR_PROC, dpram, tag);
        PutIREG(GRAMMAR_PROC,1);  /* verify .... */
        semTake(dsp_board_back_sem);

        /* void PokeProc
         * now, the backtrack info is in the DPRAM, the first location
         * is 0x1000 and it contains the number of wordarcs
         */
        number_of_wordarcs = PeekProc(GRAMMAR_PROC, dpram);
        if (number_of_wordarcs == 0x1000) {
                  printf("backtrack error, no NIL pointer found!\n");
                  printf("try to recover sentences by setting pointer to 0x30\n");
```

```
                number_of_wordarcs = 30;
        }

        MpeekProc(GRAMMAR_PROC, dpram, number_of_wordarcs);
        printf("\n\t");
        for(i=dpram+number_of_wordarcs; i>dpram; i--) {
                if (PeekProc(GRAMMAR_PROC, i) != 0xb) {
                        printf("..%d..",  PeekProc(GRAMMAR_PROC, i));
                }
        }
        printf("\n");
}

void
rec_sentence(vq_file, prun_offset)
        char *vq_file;
        int prun_offset;
{
        FILE *fp;
        int offset = prun_offset;
        int framemin = 0xffff;
        unsigned short side = 0;
        int norm_value = 0;
        unsigned short  count = NUM_OF_UNIQUE_STATES;
        int cep, dcep, egy, degy;
        int i=0;
        char buf[MAX_STRING];

        /* Open the VQ file. */
        if ((fp = fopen(vq_file, "r")) == NULL) {
                printf("Error opening \"%s\"!\n", vq_file);
                exit(0);
        }
        /*
         * perform a few dummy reads because of the file headers
         */
        fgets(buf,MAX_STRING,fp);
        fgets(buf,MAX_STRING,fp);
        fgets(buf,MAX_STRING,fp);

        /*
         * start up the pipeline and download first observation to hmm board
         */
        get_speech_frame(fp, &cep, &dcep, &egy, &degy);
        outdist_xfer(side, cep, dcep, egy, degy, count);

        /*
         * wait until all data are transfered
         */
        semTake(dist_board_int_sem);

        /*
         * put the pause wordarc into the active list memory
A_write_awmem(address,gndprob,gndtag,wordarc,stateprobaddr,topoaddr,flags)
         */
        A_write_awmem(1, 0, 0, 11, 0, 0x100aa, 1);
        A_write_awmem(2, -1, -1, -1, -1, -1, 3);

        /*
```

```
 * this is very uneffective since we read a file between
 * two frames. But let's just do it this way for a while ...
 */
while (get_speech_frame(fp, &cep, &dcep, &egy, &degy) != EOF) {

          i++;
          if (i%50 == 0) printf("frame %4d ...\n", i);

          outdist_xfer(flip(side), cep, dcep, egy, degy, count);
          startframe(side, framemin, norm_value, offset);

          semTake(dist_board_int_sem);

          /*
            Print_OutProb(side, 0, 192);
           */

          semTake(hmm_board_int_sem);

          /*
            PrintState(9, 0xd0, 1);
           */

          norm_value = read_framemin(side);
          framemin = framemin;
          offset = offset;
          hmm_reset();                /* just in case ... */

          if(sent_debug) {
                    printf("finished frame %d: bestprob = %4x, wa= %5x,
                         tag= %5x\n",i, PeekProc(GRAMMAR_PROC, 0x1000),
                          PeekProc(GRAMMAR_PROC, 0x1001),
                          PeekProc(GRAMMAR_PROC, 0x1002));
                     Print_ActiveWord(flip(side), 0x0, 0xf);
          }

          /*
           if (i%10 == 0) Get_Result(flip(side));
           */

          side = flip(side);
}
/*
 * now we still have to take care of the last frame
 */
printf("frame %4d, last frame.\n", i);

startframe(side, framemin, norm_value, offset);
semTake(hmm_board_int_sem);
hmm_reset();

Get_Result(side);

Clear_ActiveList(0, 0, 0x20, 0);
Clear_ActiveList(0, 0xffff0, 0xfffff, 0);
Clear_ActiveList(1, 0, 0x20, 0);
Clear_ActiveList(1, 0xffff0, 0xfffff, 0);
Clear_ActiveWord(0, 0, 0x30, 0);
Clear_ActiveWord(1, 0, 0x30, 0);
```

```
    StopAllProc();
    StartProc(2);

    printf("\n");
    fclose(fp);

}

void
Get_Result_in_char(side, result)
    unsigned short side;
    char *result;
{
    unsigned long tag;
    unsigned long number_of_wordarcs, wordarc, i;
    unsigned int dpram = (unsigned int) MY_DPRAM_BASE_ADDRESS;

    /*
     * this is the end of a sentence, let's interrupt the grammar dsp and
     * get the backtrack pointers. First, find the grammar node with the
     * best probability ...
     */
    tag = Get_Best(side);

    /*
     * now, send tag, interrupt and wait 'till dsp is finished ...
     */
    PokeProc(GRAMMAR_PROC, dpram, tag);
    PutIREG(GRAMMAR_PROC,1);  /* verify .... */
    semTake(dsp_board_back_sem);

    /* void PokeProc
     * now, the backtrack info is in the DPRAM, the first location
     * is 0x1000 and it contains the number of wordarcs
     */
    number_of_wordarcs = PeekProc(GRAMMAR_PROC, dpram);

    if (number_of_wordarcs == 0x1000) {
            printf("backtrack error, no NIL pointer found, try again ..\n");
            PokeProc(GRAMMAR_PROC, dpram, tag);
            PutIREG(GRAMMAR_PROC,1);  /* verify .... */
            semTake(dsp_board_back_sem);
            number_of_wordarcs = PeekProc(GRAMMAR_PROC, dpram);
            if (number_of_wordarcs == 0x1000) {
                printf("again ... try to recover sentences
                        by setting pointer to 0x30\n");
                number_of_wordarcs = 30;
            }
    }

    printf("\n\t");
    for(i=dpram+number_of_wordarcs; i>dpram; i--) {
            if (PeekProc(GRAMMAR_PROC, i) != 0xb) {
                printf("..%1x..",  PeekProc(GRAMMAR_PROC, i));
                sprintf(result++, "%1x", PeekProc(GRAMMAR_PROC, i));
            }
    }
    printf("\n");
```

```
    sprintf(result, "\0");
}

char
*char_rec_sentence(vq_file, prun_offset)
    char *vq_file;
    int prun_offset;
{
    FILE *fp;
    int offset = prun_offset;
    int framemin = 0xffff;
    unsigned short side = 0;
    int norm_value = 0;
    unsigned short  count = NUM_OF_UNIQUE_STATES;
    int cep, dcep, egy, degy;
    int i=0;
    char buf[MAX_STRING];
    char result[MAX_STRING];

    /* Open the VQ file. */
    if ((fp = fopen(vq_file, "r")) == NULL) {
              printf("Error opening \"%s\"!\n", vq_file);
              exit(0);
    }
    /*
     * perform a few dummy reads because of the file headers
     */
    fgets(buf,MAX_STRING,fp);
    fgets(buf,MAX_STRING,fp);
    fgets(buf,MAX_STRING,fp);

    /*
     * start up the pipeline and download first observation to hmm board
     */
    get_speech_frame(fp, &cep, &dcep, &egy, &degy);
    outdist_xfer(side, cep, dcep, egy, degy, count);

    /*
     * wait until all data are transfered
     */
    semTake(dist_board_int_sem);

    /*
     * put the pause wordarc into the active list memory
     * A_write_awmem(address,gndprob,gndtag,wordarc,stprobaddr,topoaddr,flags)
     */
    A_write_awmem(1, 0, 0, 11, 0, 0x100aa, 1);
    A_write_awmem(2, -1, -1, -1, -1, -1, 3);

    /*
     * this is very uneffective since we read a file between
     * two frames. But let's just do it this way for a while ...
     */
    while (get_speech_frame(fp, &cep, &dcep, &egy, &degy) != EOF) {

              i++;
              /*
               * if (i%50 == 0) printf("frame %4d ...\n", i);
               */
```

```
            outdist_xfer(flip(side), cep, dcep, egy, degy, count);
            startframe(side, framemin, norm_value, offset);

            semTake(dist_board_int_sem);
            semTake(hmm_board_int_sem);

            norm_value = read_framemin(side);
            framemin = framemin;
            offset = offset;
            hmm_reset();                /* just in case ... */

            side = flip(side);
    }
    /*
     * now we still have to take care of the last frame
     */
    printf("frame %4d, last frame.\n", i);

    startframe(side, framemin, norm_value, offset);
    semTake(hmm_board_int_sem);
    hmm_reset();

    Get_Result_in_char(side, result);

    Clear_ActiveList(0, 0, 0x20, 0);
    Clear_ActiveList(0, 0xffff0, 0xfffff, 0);
    Clear_ActiveList(1, 0, 0x20, 0);
    Clear_ActiveList(1, 0xffff0, 0xfffff, 0);
    Clear_ActiveWord(0, 0, 0x30, 0);
    Clear_ActiveWord(1, 0, 0x30, 0);

    StopAllProc();
    StartProc(2);

    printf("\n");
    fclose(fp);
    return(result);
}
```