A VLSI WORDPROCESSING SUBSYSTEM FOR A REAL TIME LARGE VOCABULARY CONTINUOUS SPEECH RECOGNITION SYSTEM

by

Anton Stölzle

Memorandum No. UCB/ERL M89/133

14 December 1989

A VLSI WORDPROCESSING SUBSYSTEM FOR A REAL TIME LARGE VOCABULARY CONTINUOUS SPEECH RECOGNITION SYSTEM

by

Anton Stölzle

Memorandum No. UCB/ERL M89/133

14 December 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A VLSI WORDPROCESSING SUBSYSTEM FOR A REAL TIME LARGE VOCABULARY CONTINUOUS SPEECH RECOGNITION SYSTEM

by

Anton Stölzle

Memorandum No. UCB/ERL M89/133

14 December 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

Hidden Markov model based speech recognition for connected speech and large vocabularies can be done with recognition accuracies of up to 99% [14, 11, 6], but due to the complexity of the algorithms involved, general purpose computers need several minutes to compute the results. For many applications however, it is essential to have real time speech recognition. This paper presents a word processing system that is based on two integrated circuits. It is part of a recognition hardware [12] that gives real time performance to large vocabulary continuous speech recognition systems. The integrated circuits perform the recognition algorithm (Viterbi algorithm [7]) for 50,000 states in real time using a breadth first recognition search. Since the bottleneck of the system is the acquisition of data (670 Megabits per second), an architecture with dual ported cache memories is used. With this system applications that require real time speech recognition can be implemented, for example automatic typewriting or database query.

Contents

Chapter 1

Introduction

It is generally accepted that Hidden Markov Models (HMM) are currently the most efficient technique to model speech for use in automatic speech recognition [7, 5, 1]. For large vocabulary connected speech recognition systems it has largely replaced the technique of template matching, where speech templates are compared to the speech that has to be recognized using the dynamic time warp algorithm [4, 2]. HMM based speech recognition systems have proven to yield high recognition accuracy even for speaker independent systems. For speaker dependent systems they have the advantageous feature that only a small part of the speech model is speaker dependent. This is important if the system has to adapt or switch to a new speaker.

The task in speech recognition is to recognize a sequence of utterances by comparing it to a speech model. In HMM based speech recognition systems the model is the following:

Speech is segmented into *frames* which are time intervals of typically 10 ms. The characteristics of every frame is described with a set of features (o_i for the feature values at frame i). These features could for example be the energy of the signal in different frequency bands. The states s of a HMM speech process correspond to generic speech sounds (e.g. a phoneme or a part of a phoneme). To take into account the variation in pronunciation, a probability distribution is obtained that gives the output probability, $P(o|s)$, that the process outputs a set of features o when it is in s. Speech can be considered as being generated by transitions between these states yielding a state sequence, $S = \{s_i\}$. These transitions occur according to transition probabilities, $A(s,t)$, which reflect the likelihood that state s follows state t.

The task that has to be performed during speech recognition is to find, for a given HMM, the state sequence that most likely could have produced the speech that has to be recognized. If the most probable state sequence has been determined, it is possible to reconstruct the sequence of words that was spoken. A very efficient search algorithm for the most likely state sequence is the Viterbi algorithm. Frame for frame and for every state, this algorithm computes the probability of the most probable state sequence that ends in this state.

This paper describes hardware which is able to perform the Viterbi equation for up to 50,000 states in real time for frame durations of 10ms. [12, 13]. It thus improves the performance for a class of HMM-based speech recognition systems such as the BYBLOS system developed by Bolt Beranek and Newman [1] and the DECYPHER system developed at SRI International [11].

The hardware is part of a recognition system that includes grammar processing by considering transitions between HMMs that describe words [12, 13]. This constrains the possible word sequences thus increasing the recognition accuracy. However, this paper focuses on the word processing subsystem which only considers state transitions within words.

Chapter 2

Algorithm

Fig. 2.1 shows the graph of a HM chain typical for a vocabulary word. The nodes represent the states in the HMM which correspond to periods of steady behavior in speech. The arcs correspond to the transitions between the states which have transition probabilities. This graph defines the ***topology*** of the HMM: for any state the ***predecessors*** (states that have a transition to the state) and the ***successors*** (states in which this state can transition to) can be determined. In order to find out the values for the transition probabilities and the probability distributions that are associated with the states, a large amount of speech data has to be analyzed. For example, 4,000 sentences are needed to train SRI's speaker independent system for a vocabulary size of 1000 words [11]. However, during speech recognition it is assumed that all the probability values have already been determined so that a well trained HMM is available.

The task in speech recognition is to find the state sequence that most likely could have produced the incoming speech. If this state sequence is known, the word sequence that corresponds to the incoming speech can be derived.

2.1 State probabilities

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

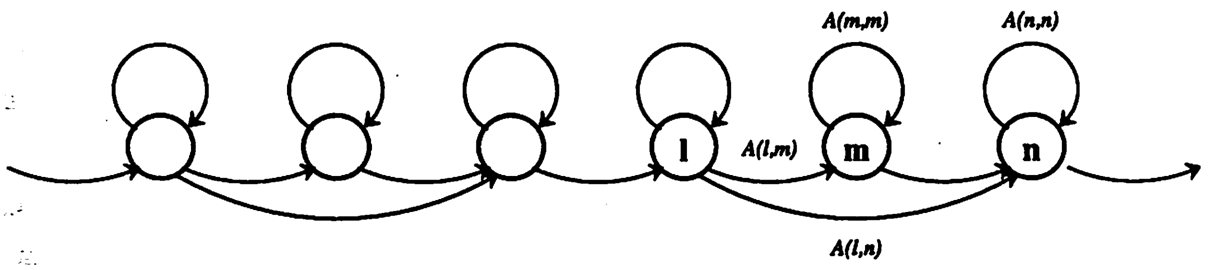


Figure 2.1: State transition graph of a typical HMM used in the word processing subsystem

and is given by

$$P(S_N, O_N) = \pi(s_1) \cdot P(o_1|s_1) \cdot \prod_{i=2}^{N} [A(s_{i-1}|s_i) \cdot P(o_i|s_i)] \tag{2.1}$$

In this equation, $\pi(s)$ is a probability distribution that gives the probabilities of the states in the first frame. The goal is to find the most likely state sequence given O_N. Therefore, if the path probabilities of all the possible state sequences are known, the sequence corresponding to the highest path probability represents the most likely state sequence. However, it is impractical to compute all the possible state sequences: among all the possible sequences ending in a *certain* state s, it is sufficient to compute only the probability of the most likely sequence, $P(O_N, s)$ that leads to s. Is this probability known for every state in the HMM, the state that maximizes $P(O_N, s)$ is the endpoint of the most likely state sequence.

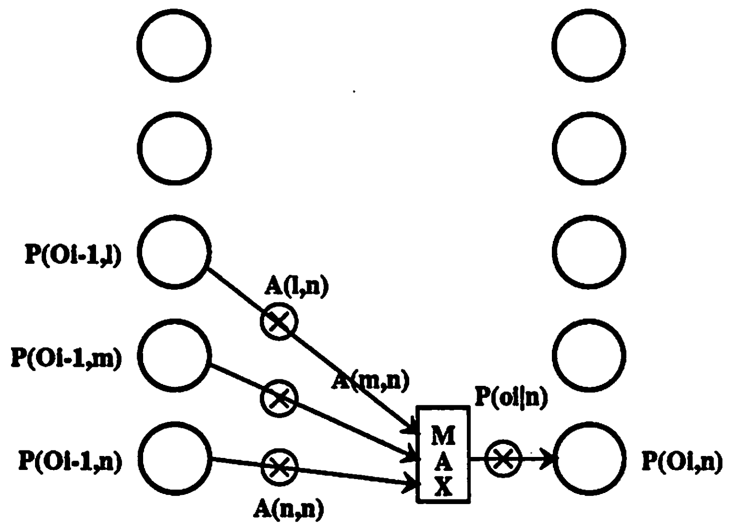
Let us define the ***state probability***, $P(O_i, s)$, as the probability of the most probable state sequence that ends in s and generates O_i, a sequence of i feature values. This probability will be used to compute the desired state probability $P(O_N, s)$ by means of a dynamic programming scheme called the Viterbi algorithm [7]:

$$P(O_1, s) = \pi(s) \cdot P(o_1|s) \tag{2.2}$$

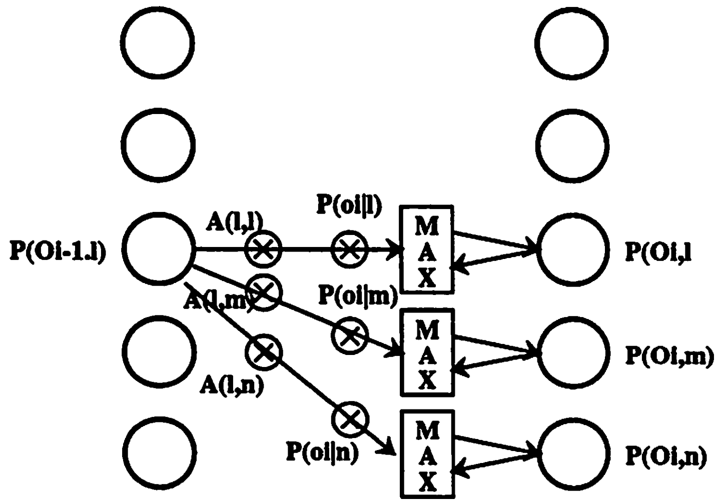
$$P(O_i, s) = \max_p [P(O_{i-1}, p) \cdot A(p, s)] \cdot P(o_i|s) \tag{2.3}$$

The states p stand for all the predecessors of state s, which are defined in 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 state probability then is the final state of the most likely state sequence.

This computation can be visualized using the computation graph as shown in Fig. 2.2a. Associated with the nodes in the left column are the state probabilities of all the states at frame $i-1$, $P(O_{i-1}, s)$. The task is to compute the state probabilities $P(O_i, s)$ which are symbolized with the nodes in the right column. Two nodes on the same line correspond to state probabilities of the same state, and probabilities



a)



b)

Figure 2.2: a) lattice structure for predecessor transitions b) lattice structure for successor transitions

on neighboring lines correspond to states that are neighbors in the HM chain of Fig. 2.1.

To compute $P(O_i, s)$ for state s, the probability values of the predecessor states have to be multiplied with the transition probabilities to s. The product that represents the highest probability is then multiplied with the output probability $P(o_i|s)$, which is defined in the HMM. This implementation will be called *predecessor* implementation since all the transitions *to* a state are considered.

Another possible implementation is to compute the state probabilities based on the successors z of a state s that has the state probability $P(O_{i-1}, s)$ at frame $i-1$. This implementation is formalized in Eq. 2.4 and visualized in Fig. 2.2b:

$$P(O_i, z) = \max[P(O_i, z),\ \ P(O_{i-1}, s) \cdot A(s, z)] \cdot P(o_i|z) \qquad (2.4)$$

There are several states that share the same successor state, and the goal is to find the transition that results in the highest state probability at frame i for every possible transition to the state. Therefore, the state probability of the successor is only updated if the currently considered path results in a higher state probability than a path that might have been considered previously (and that originated from another state). For this implementation it is essential that all the state probabilities for the successor states are set to zero before the computation starts for a new frame. If Eq. 2.4 has been performed for the successors of every state in the vocabulary, all the state probabilities have the right values $P(O_i, s)$ for frame i as if they where computed using Eq. 2.3.

2.2 Backtracing

As discussed above, the state with the highest state probability in the last frame is the endpoint of the most probable state sequence. The task that has to be performed after the incoming speech pauses, is to recover this state sequence starting from the endpoint.

This *backtracing* can be done if there is a list that, for every state in the path, contains a pointer to the preceding state. However, during the generation of this list the most probable state sequence is not yet known, therefore all state sequences should be stored. Since every state in the HMM is the endpoint of a state sequence, there are as many state sequences as there are states in the HMM. Storing this list would result in a huge memory requirement since there would be an entry for every state and for every frame.

However, it is not necessary to exactly recover the sequence of states, it is sufficient to know the sequence of words by storing word transitions. To be able to do this, it is important to know which word most likely preceded the current word. This information is provided by the grammar processing subsystem. In every frame, the first state in a word gets a tag from the grammar processing system that identifies the most likely predecessor word. If, in the succeeding frames, the most probable state sequence develops through the states in a word, the tag is transferred to the different states. Thus, if the last state in the word has a high probability (which means it might be the final state of the most probable path), it contains the tag that points to the predecessor word. Using this scheme it is possible to reduce the memory requirement by just storing the backtrace tags of final states with high state probabilities. The generation and storage of the tags is not part of the word processing system, all it does is to transfer the tag from one state to another according to Eq. 2.5:

$$TAG(s,i) = \arg\max_{p}[TAG(p,i-1)] \tag{2.5}$$

$TAG(s,i)$ is the tag associated with state s at frame i. Eq. 2.5 defines that $TAG(s,i)$ is the copy of the tag associated with the predecessor state p of s that is part of the most likely path to s.

2.3 Multiple output probabilities

The output probability is the probability that a certain speech segment is generated while the speech process is in a certain state. In our system it is assumed that the Markov process outputs a speech segment that is described using up to four different representations, $o^{(1)}..o^{(4)}$. In other words, every state has associated with it up to four probability distributions that simultaneously output four different representations of a speech segment. Therefore the output probability is the joint probability of the individual outputs. This is formalized in Eq. 2.6:

$$P(o|s) = P(o^{(1)}|s) \cdot P(o^{(2)}|s) \cdot P(o^{(3)}|s) \cdot P(o^{(4)}|s) \tag{2.6}$$

In the DECYPHER system developed at SRI International [11] the different features are cepstral coefficients, the time derivative of the cepstral coefficients, the energy and the delta energy of the speech signal. The individual outputs are vector quantized or quantized using an 8 bit representation. Therefore the probability distributions of the stochastic functions are discrete and every distribution is composed of 256 probabilities.

If the HMM is set up that the process outputs only one representation of the speech segment, this equation becomes obsolete and the output probability is defined in the probability distribution of the (single) random function associated with the current state.

2.4 Difference from Viterbi in communication systems

Viterbi decoding is widely applied for various problems in communications, such as decoding convolutional codes or trellis codes. The hardware available for these applications can however not be used for speech recognition, and the intention of this chapter is to show the reason for that.

A Hidden Markov Model that describes the process of speech production typically has a large number of states and a large amount of transitions between the states. The specifications for the system described in this paper allow for up to 256K states with an average of three transitions per state within the word models. To store the topology of a model of this size corresponds to a memory requirement of 3,6 Mbyte. Storing the probability distributions of the stochastic functions that are associated to each state takes another 64 Mbytes (8 bits per output probability, 256 probabilities per function, 256K functions). Therefore, high density dynamic memories (DRAMs) are necessary to store this information, and the hardware that performs the Viterbi algorithm has to access all this information within a frame duration of typically 10ms. Thus the bottleneck in performing Viterbi for speech recognition is the acquisition of data.

On the other side, the Markov process that models the generation of a sequence of data in a communication system [10] has at most 64 states. Also, there are not very many transitions between the states, typically two or four per state. That 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 one state in the HMM. Another difference is that, instead of using precompiled, fixed probabilities to describe transitions between the states, a distance measure between the received signal and the signal represented by the transition is used. Therefore the hardware performing Viterbi has an arithmetic that can't be used for speech recognition.

In summary, it is necessary to have a new dedicated architecture with custom integrated circuits that is capable of accessing all the necessary off chip data. This system is targeted towards a real time performance for a vocabulary size of 50,000 states, which means that 670 Mbyte/sec have to be accessed.

Chapter 3

Architectural tradeoffs

The intention of this section is to show different possible architectures and implementations. The tradeoffs between the alternatives will be discussed and our choice will be motivated.

3.1 Hierarchical storage of output probabilities

The storage of the output probabilities constitutes the most critical memory requirement in the system. This system is targeted towards real time performance for 50,000 states, but it is designed to be able to deal with 256K states with an approximately five times slower performance than real time.

If the output probabilities are stored individually for every state, there are 64 Mbyte of memory required. (8bits per probability value, 256K states $\cdot$ 256 probabilities per state = 64 Mbyte) This requirement gets even bigger if every state has several associated output distributions.

However, not all the states in the HMM are unique states, some of them actually describe the same speech segment. Therefore it is possible to to reduce the memory requirement by sharing probability distributions among several states. Conceptually, a single phoneme occurs in several words, and the states that describe this phoneme in the different words can share one probability distribution. Our architecture supports that feature in the following way: Only unique output probability distributions are stored and to find the unique probability distribution for a certain state, a lookup table is used. Using this scheme, the memory requirement could be reduced by a factor of eight with the cost of a lookup table (256Kx14).

The tradeoff however is, that the memory containing the output probabilities is now accessed in a random fashion even though the states in the HMM are processed sequentially. Since this memory is a DRAM, this degrades the performance since now the fast column access mode cannot be used. However, the reduction in memory size was important enough to justify this decision.

3.2 Successor processing versus predecessor processing

In chapter 2 two different implementations of the Viterbi algorithm were demonstrated: the computation of the state probabilities based on the successors of a state and the computation considering the predecessors of a state.

The advantageous feature of processing transitions to successor states is that it is possible to save computations in a very straightforward way. States with very low state probabilities very unlikely terminate the most likely path. Therefore, it is sufficient to compute the probabilities only for states that yield high state probabilities. This method is called *pruning* or *beam search*, states with low state probabilities are just not considered. Performing the Viterbi algorithm using transitions to successors (Eq. 2.4) makes it easy to implement pruning, because states with low state probabilities, $P(O_{i-1}, s)$, cannot contribute towards high state probabilities $P(O_i, z)$ for the successor states z. Thus, if a state with low probability $P(O_{i-1}|s)$ is encountered, all the computations for its successors can be dropped.

There are, however, drawbacks of this implementation. One is, that the computations have to be performed sequentially for every transition to a successor state because for each transition some $P(O_{i-1}, z)$ has to be accessed. A speedup by performing this equation in parallel for several successors is not possible since it would result in contention for the memory containing $P(O_i, z)$ (i memory) and the memory containing $P(o_{i-1}|s)$ (output probability memory). The other drawback is, that the i memory has to be accessed twice, for reading $P(O_i, z)$ and for eventually updating it if the current transition yields a higher probability. This produces a bottleneck which is the access of the i memory. Yet another drawback is, that to find out if a state can be pruned, still some computation has to be done. $P(O_{i-1}, s)$ has to be read and compared to a threshold value to make the pruning decision. Sorting the states according to their state probabilities would be a way to avoid that, but sorting the states is too time consuming.

This bottleneck can be eased if the Viterbi algorithm is done based on the predecessors of a state (Eq. 2.3). For this implementation the i memory has to be accessed only once per state for writing the new state probability. The memory containing $P(O_{i-1}, p)$ (i-1 memory) on the other side has one read access per predecessor. Now this memory constitutes the new (but faster) bottleneck.

However, implementing pruning for this method is not straightforward. Every state has to be processed since any predecessor state could have a high state probability. This is however not known before every single transition from a predecessor has been processed.

Nevertheless, our decision was to implement the algorithm based on the predecessors of a state (Eq. 2.3). There are two major reasons for this decision. Firstly, the bottleneck to the $i-1$ memory could be eliminated using cache memories on the custom processors (see chapter 3.3). Therefore, it is possible to simultaneously perform the computations for several transitions from predecessor states. Secondly, pruning is still possible on a higher level: the subsystem that considers transitions between the words has the information if a word has a high probability to be the last word on the most likely path. Based on this probability, it can prune the words with low probabilities so that the word processing subsystem doesn't have to consider any state within a pruned word. The tradeoff is, that even though a word has a high probability, there might be states within a word that have a low state probability (for example, if the speech process just started with a word, all the states at the end of this word will have low state probabilities). However, the potential speedup in the performance of the Viterbi algorithm justified this extra computations.

3.3 On chip cache memories

The system requirement is to compute the Viterbi equations for 50,000 states within a frame duration of 10ms. This translates to a computation rate of one state probability per 200ns. This corresponds to the cycle time of the DRAMs that are used throughout the system, therefore it is necessary to access within one memory cycle all the data needed to perform the computations associated with one state (Eqs. 2.3,2.5).

The data that can be accessed in parallel are the locations of the predecessor states of the current state and their transition probabilities to this state. Therefore the memory containing these informations keeps all these data in one single word. The output probability has to be accessed only once for the computation of one state probability, so this information is also easily accessible.

The bottleneck however is to access the state probabilities of the predecessor states, $P(O_{i-1}, p)$. Unless there are multiple copies of the memory that contains these probabilities (i-1 memory), accessing these data is not straightforward. Having several copies of the i-1 memory however is not desirable because of two reasons. Firstly, it is a large memory (256Kx32bits) and having multiple copies of this memory would be a waste. Also, in the next frame the memory that currently is the memory where $P(O_i, s)$ is written (i memory), becomes the i-1 memory in the next frame so that this memory would also have to be replicated. The second reason not to replicate the memories is that they all have to be interfaced to the processing element that performs the Viterbi algorithm. However, since the number of pins per processor is limited, this solution would not be feasible: every single memory would require both, an extra address- and data bus.

The solution to this problem is to use multiple copies of only a subset of the i-1 memory. This is possible because of two properties of HMMs at the word level: every predecessor of a state is close to this state and certainly not the state of another word, and every predecessor is situated to the left of this state or it is the state itself. Thus, if there are multiple copies of the subset of the i-1 memory that contain all the state probabilities $P(O_{i-1}, p)$ of a set of states that are to the left of the current state (see Fig. 2.1), all the relevant information is replicated. Since the predecessors are very close to the current state, this subset is small: 16 predecessor probabilities are sufficient, which corresponds to the fact that within a word no state has a predecessor that is more than 16 states away. Therefore, this subset can be replicated on the chip that performs the Viterbi algorithm.

The state probabilities are entered in memory analogous to Fig.2.2 such that consecutive memory addresses correspond to neighboring states in the HMM. Therefore the state probabilities that have to be replicated are the ones between the current address corresponding to the current state and an address with a negative offset of 16. This subset can be updated by sequentially reading the off chip i-1 memory into the on-chip memories. In this process the predecessor probability that has the biggest offset to the current address is being overwritten. Fig. 3.1 shows the principle:

In order to compute $P(O_i, t)$, the state probabilities of the predecessors $P(O_{i-1}, f)$,$P(O_{i-1}, e)$ and $P(O_{i-1}, b)$ have to be read simultaneously from the on-chip cache memories. To update the set of probabilities on these cache memories, a new predecessor probability, $P(O_{i-1}, g)$ is copied to the three memories. Thus, the write address is sequential and can be generated using a modulo counter, so that the oldest value in the memories is replaced with a new value. On the other side, the read address is generated relative to the current write address using an offset that locates a specific predecessor.

With this architecture the Viterbi algorithm can be performed simultaneously for several predecessors without replicating off chip memory or stressing the number of pins on the processor that does the computations.

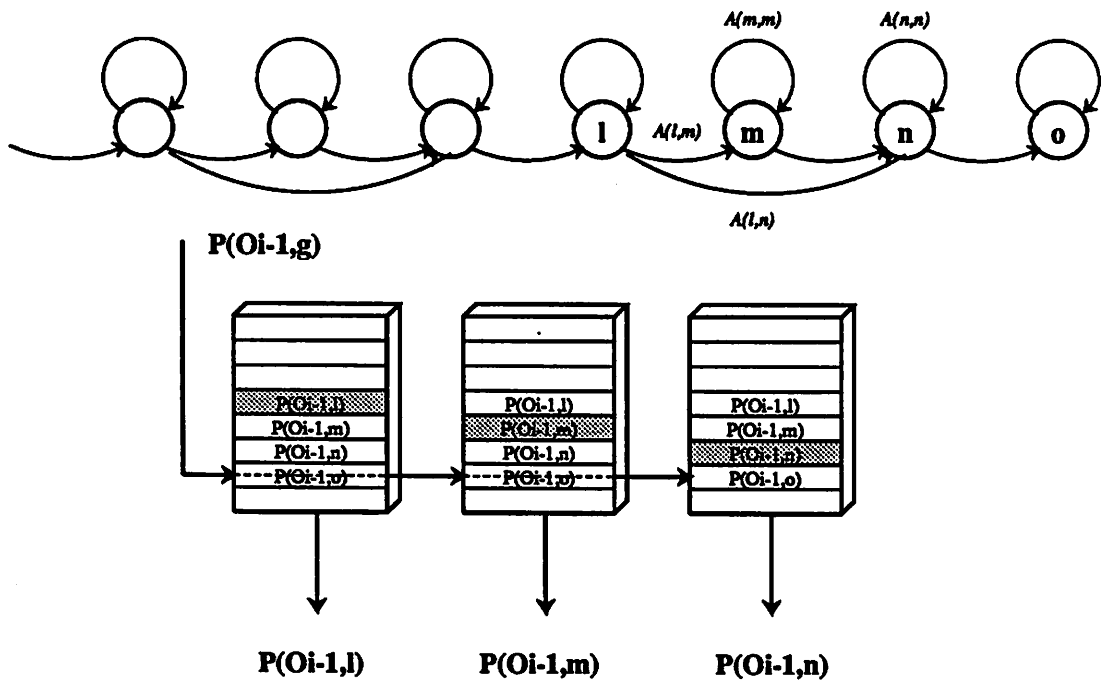


Figure 3.1: dual ported cache memories that keep three identical copies of a local subset of the predecessor probabilities.

Chapter 4

Subsystem architecture

Fig.4.1 shows a block diagram of the word processing subsystem. It has three major components. The first stores all the information associated with the HMMs of the vocabulary words. Another component stores intermediate results ($P(O_{i-1}, p)$ and $P(O_{i-1}, s)$). Finally, the third component is a processing element that is partitioned into two integrated circuits, the Viterbi processor that computes the state probabilities (Eq. 2.3) and the Backtrace processor that copies the backtrace tag (Eq. 2.5).

All the probabilities associated with the HMM, $A(s,t)$ and $B(o|s)$, are represented with their absolute logarithmic values using 8 bits. The state probabilities $P(O_i, s)$ have the same logarithmic representation using 14 bits. Therefore, all the multiplications involved in computing $P(O_i, s)$ can be implemented with additions, and a maximum operation corresponds to a minimum operation since the absolute value of the logarithm of a high probability is a small number.

4.1 Representation of the HMM

The part that stores the HMM has two major components, the *topology memory* and the *output memory*. Every word in the topology memory is associated with a state in the HMM and describes the location of three predecessor states relative to the current state along with their transition probabilities to this state. There is also

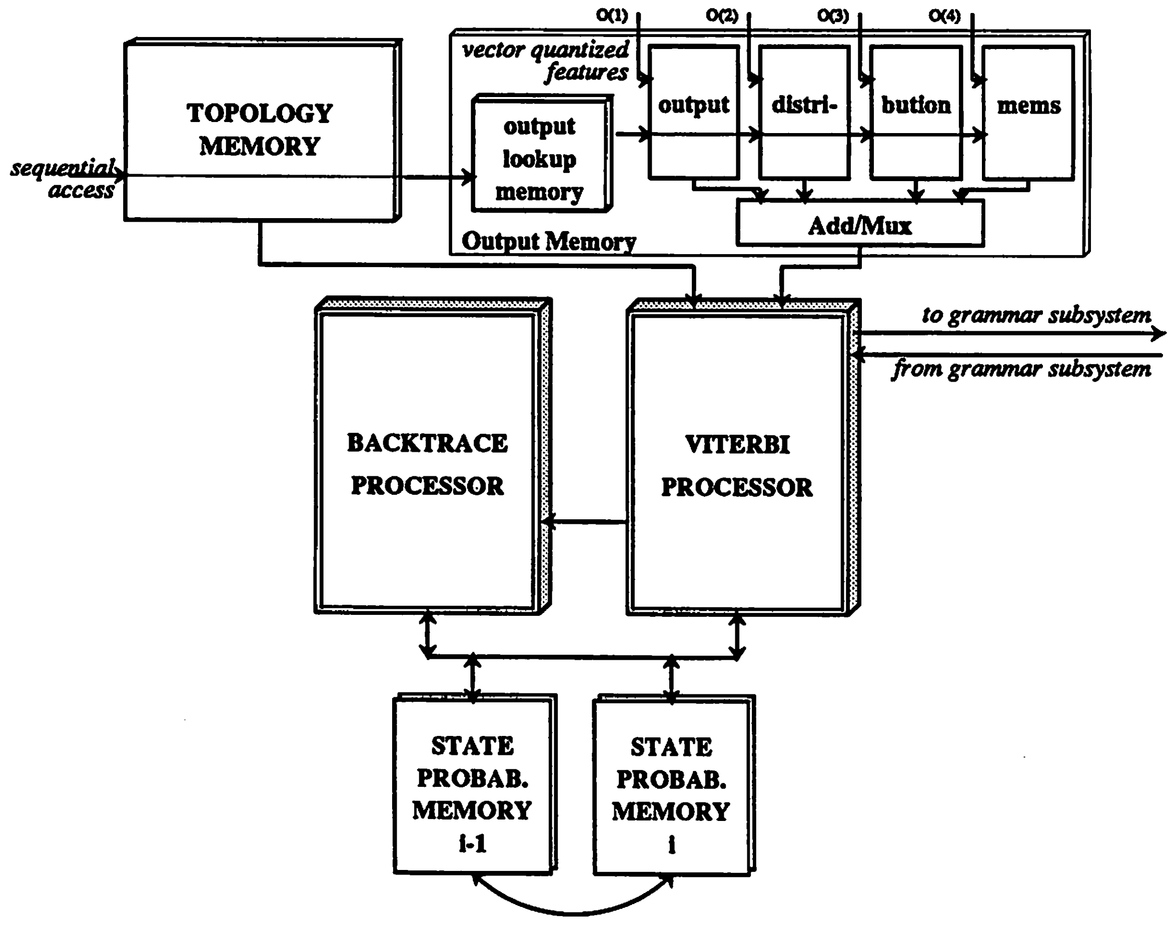


Figure 4.1: Block diagram of the Word Processing Subsystem

a control bit that indicates if the state can start a word (has a transition that comes from another word) and a value that gives the probability that this state can end the word (transits to another word). Finally, there are two more bits that indicate if a state is the last state in the word or the last state in the frame. Three predecessors are simultaneously described because of the fact that a state in the word models mostly has not more than three predecessors. It is therefore sufficient to simultaneously access the information related to three predecessors in order to compute $P(O_i, s)$ within one memory cycle. If a state has more than three predecessors, a control bit indicates that the next memory location is dedicated to the same state where the next set of predecessors is described. In this case two memory cycles have to be spent to compute $P(s, t)$.

The *output memory* is a combination of 5 memories and some discrete logic to implement the computation of the joint output probability as defined in Eq. 2.6. Addressed with the same address as the topology memory, the output lookup memory (see Fig.4.1) provides the high address bits for the 4 output distribution memories. As discussed above, this indirection is used because many states share the same output distributions. The other part of the address for the individual output distribution memories are the vector quantized features $o^{(1)}$ through $o^{(4)}$ of the current speech frame. According to Eq. 2.6, the resulting output distributions $P(o^{(1)}|s)$ through $B_1(s, O^{(4)}|s)$ are either multiplied (added) or, if an output probability based on one feature is desired, multiplexed to get $P(o|s)$.

This operation has been implemented using discrete components. The rationale not to implement it on one of the custom processors is to save pins: the inputs to this operation are four output probabilities, 8 bit each. The output is one output probability represented with 8 bits, so implementing this function on the board level saves 24 pins. Also, the operation is a straightforward addition/multiplex operation which is easy to implement using off the shelf components.

4.2 Memories for intermediate results

The two state probability memories contain the state probabilities and the backtrace tags for two consecutive frames, $i-1$ and i. Using the same address that the memories containing the HMM use, every location of the i-1 memory keeps the state probability and the backtrace tag of the state that is associated with this address. In the case where a state has more than three predecessors (= occupies more than on address location), only the location with the highest address describing that state has meaningful values. In order to perform Eqs. 2.3 and 2.5, the system reads $P(O_{i-1}, p)$ and $TAG(p, i-1)$ from the i-1 memory and writes the result, $P(O_i, s)$ and $TAG(s, i)$ into the i memory. After a whole frame has been processed, the memories are "flipped" in such a way that the i-1 memory becomes the i memory and vice versa.

4.3 Processing elements

The custom processors that perform the Viterbi equations are partioned into two processors: the Viterbi processor that computes $P(O_i, s)$ (Eq. 2.3) and the backtrace processor that transfers the backtrace tag, $TAG(s, i)$, from the best predecessor to the current state according to Eq. 2.5.

It was necessary to make a partition into two processors because of the limitation in the pin count. The partitioning we selected was natural in the sense that each processor implements a different computation. Since the data needed for these computations are different, this partition made it possible not to exceed 208 pins per processor. Both processors are simultaneously working on the same state, therefore all the input and output data of both processors can be stored in the same address location throughout the system. The only communication between the processors are some control pins that instruct the backtrace processor which predecessor state is contained in the most probable path so that the proper tag can be selected.

Chapter 5

Custom processors

Both custom processors on the system, the Viterbi processor and the Backtrace processor, have some common architectural features that will be described before the individual processor description.

5.1 Pipelining

The system architecture as described above makes it possible to access within one memory cycle all data that are necessary to perform the Viterbi equations (2.2), (2.3) and (2.5). To make use of this potential, an processor architecture has to be defined that is able to cope with this throughput.

We decided to implement a pipelined structure with parallel data paths for the computation of the transitions from the predecessor states, $(P(O_{i-1}, p) \cdot A(p, s))$. To minimize the design effort for the custom chips, the processor uses the same cycle time that the memorys have (200ns). Therefore, the timing requirements for the processors are manageable and the control can be implemented in a straightforward way.

This basic clocking strategy has the consequence, that a memory access has to be pipelined: data can only be expected one cycle after the address has been put out. This does not complicate the architecture since the address computation is not dependent on data. Therefore it can be guaranteed that the processors can read and write continuously.

In order to have the potential to increase the clock frequency in the case when faster memories are used, we tried to avoid critical paths throughout the processors. Since the latency introduced by the pipelined processors is not an issue, we allocated a pipeline stage for every operation like additions or comparisons. To avoid critical paths due to wiring delays, also a pipeline stage was allocated whenever there was global communication on the chip. Thus, broadcasting data on the chips corresponds to a pure register transfer operation. For example, this was done for the control signals that are provided by the control unit and that have to be distributed to the individual data paths and for data communication between data paths. Using the strategy, the critical path could be reduced to the critical paths of the slowest logic blocks, which in our design is a saturating 14 bit adder.

5.2 Control logic

We chose for the controller a data stationary control architecture. This means that all the necessary control for all the different pipeline stages is generated simultaneously for one set of data. To have the right timing for the individual bits in the control word, the controller has pipeline registers that delay the control bits so that they follow the data through the pipeline. Whenever a control bit is required at a certain pipeline stage, it is diverted from the control delay register that corresponds to the pipeline stage.

This architecture was chosen because it minimizes the size of the finite state machine (FSM) that generates the control word. This is because the sequentiality is reduced and therefore fewer states are required with respect to a direct implementation using time stationary control: in time stationary control all control bits are generated for a particular instance of time and directly routed to the individual pipeline stages. Since for a heavily pipelined structure data stationary control results in a smaller FSM, it can be clocked with higher clock rates than time stationary control.

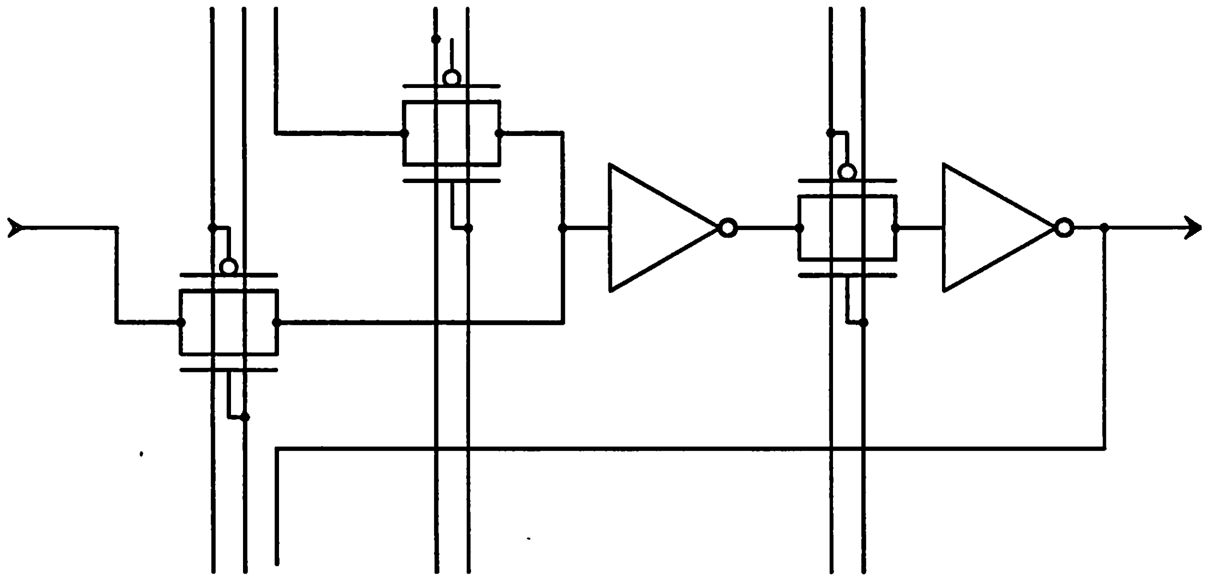


Figure 5.1: Dynamic Scanpath Register.

5.3 Testing strategy

The pipelined architecture introduces a highly sequential behavior, which makes it hard to control or to observe internal nodes solely using inputs or outputs of the processor. Therefore we chose to use the scanpath test methodology. Every register is implemented as a scanpath register so that the inputs and outputs of every logic between registers are completely controllable and observable.

To minimize the overhead in area and pins introduced by using scanpath registers throughout the design, we designed pipeline registers that have only one additional transfer gate over a regular dynamic master slave register (see Fig. 5.1). The register uses the same two phase clock for testing as well as for the normal mode of operation. The input transfer gates of the register are driven by a signal that is a combination of a conditional load signal, a signal indicating the scantest mode and the master clock. Therefore only two additional pads had to be provided per scanpath (scanin, scanout) plus one control pin that indicates operation in sequential mode.

The scanpath was partitioned into several smaller scanpaths so that in case of an error in one part of the chip still other parts can be tested.

5.4 Viterbi processor

Fig. 5.2 shows a block diagram of the Viterbi processor which updates state probabilities according to Eq. 2.3. As discussed above, the processor has on chip bidirectional cache memories to store 16 state probabilities $P(O_{i-1}, p)$. We provided 3 dual ported memories so that the path probabilities for three predecessors can be computed in parallel. Since there are mostly not more than 3 predecessors for a state in the word model, most states will be updated within one processor cycle. The write address, which is the same for all memories, is generated by a modulo counter. The read addresses are computed relative to the write address using offset addresses that are provided by the topology memory. The three state probabilities at the output of the memories then have to be multiplied with the transition probabilities, $A(p, s)$ (see Eq. 2.3), which is implemented with an addition of the logarithm of the probabilities. After that operation, the probability with the highest value is selected corresponding to a minimum operation of the logarithmic values. Whenever a state has more than 3 predecessors, this minimum operation is done sequentially on sets of 3 predecessor probabilities to find out the overall best value. To this selected value the output probability, $P(o|s)$, provided by the output probability memory, is multiplied (added) to finally compute $P(O_i, s)$.

In order to minimize the wordlength of the probability values the processor performs a normalization based on the overall best probability of the previous frame. Also, to support a pruning operation on the wordlevel, the highest state probability of every word is computed.

Parallel to the computation of Eq. 2.3, the Viterbi processor computes the probability that a word ends. This probability is the overall maximum of the state probabilities of a given word multiplied (added) with their transition probabilities to the end of the word. The implementation is such that potentially every node can end a word, but only meaningful transitions have a non-zero transition to the end of the

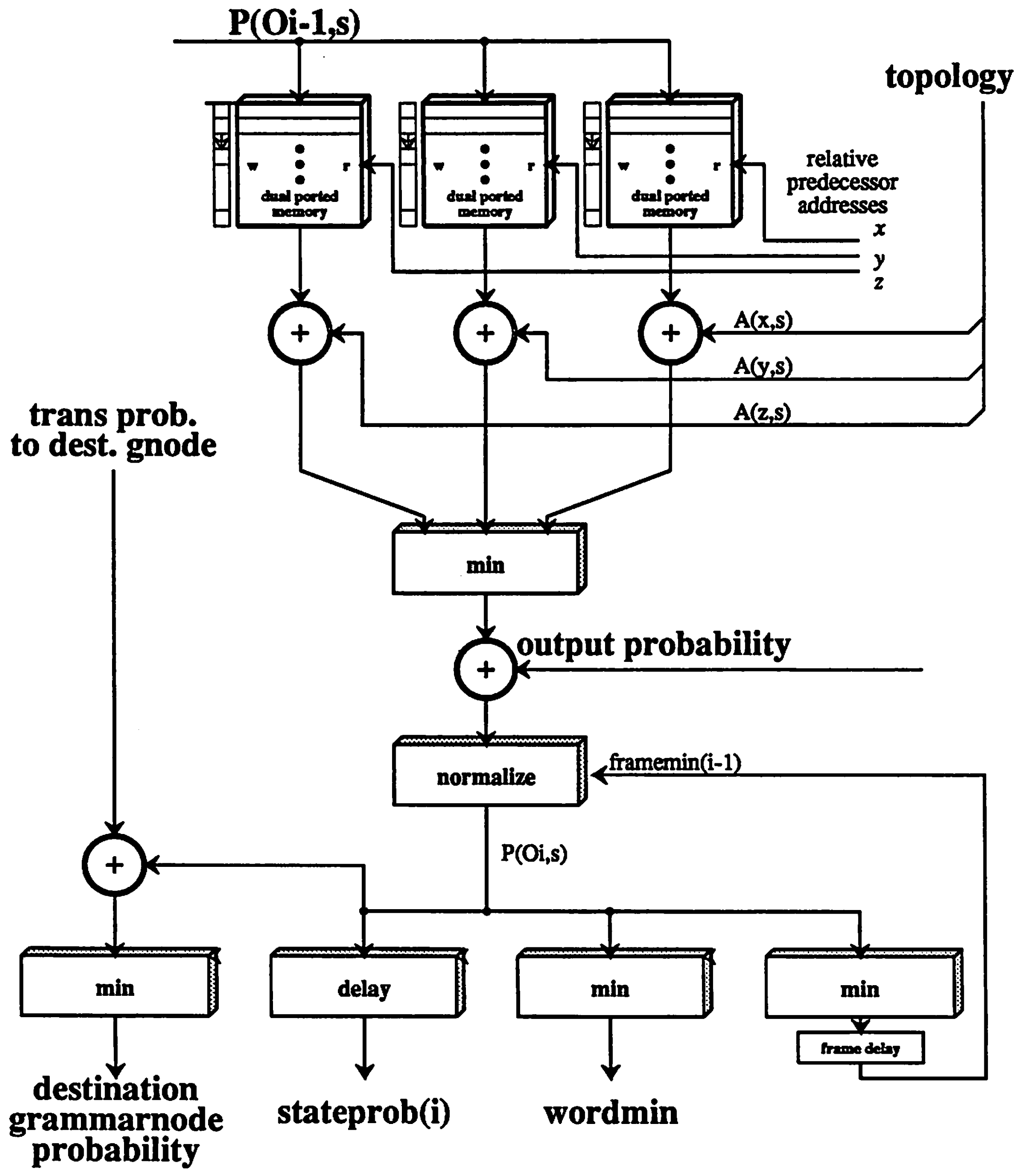


Figure 5.2: Architecture of the Viterbi Processor.

word.

The processor (Fig.5.3) has up to 11 levels of pipelining. The basic blocks are 8 data paths, 3 bidirectional memories and a control unit with a data stationary control architecture. The chip has been fabricated through MOSIS using a 2 μm CMOS technology. Including 204 pads, it has a die size of $11.6 \times 9.8mm^2$ with 25,000 transistors.

5.5 Backtrace processor

The architecture of the Backtrace processor is shown in Fig. 5.4. It implements Eq. 2.5, the backtrace portion of the Viterbi algorithm. Like the Viterbi processor, this processor has 3 on-chip bidirectional memories to store the backtrace tags of 3 predecessors of a given state.

The tags are passed through a series of delay registers to synchronize them to the associated probability data that are simultaneously processed on the Viterbi processor. Multiplexors are used to generate two different outputs based on control signals that are supplied by the Viterbi processor. The first output is $TAG(s, i)$, the updated backtrace tag for every state within a word, which is stored in the state probability memory along with $P(O_i, s)$. The second output is the backtrace tag that is given to the grammar processing subsystem along with the probability that the word ends. There it is eventually stored in memory and later used to recover the most likely word sequence.

The backtrace processor has a die size of 6.8 mm by 7.5 mm including 132 pads in a 2 micron technology and uses 12,000 transistors.

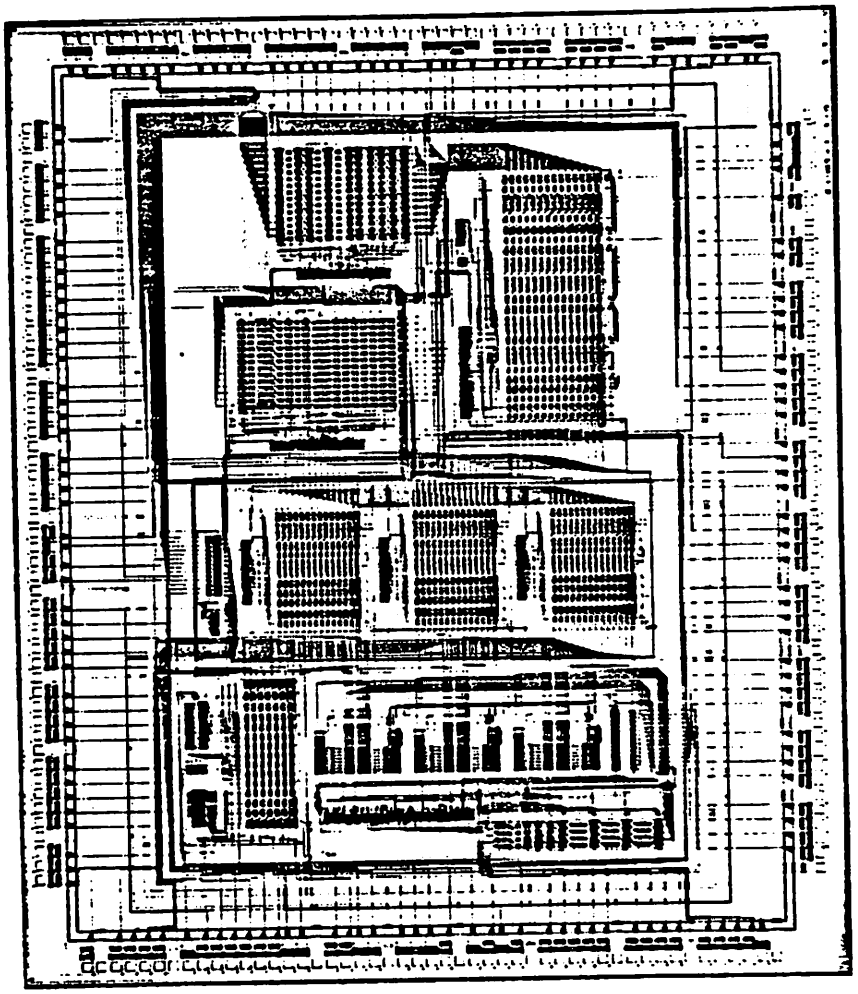


Figure 5.3: Layout of the Viterbi Processor.

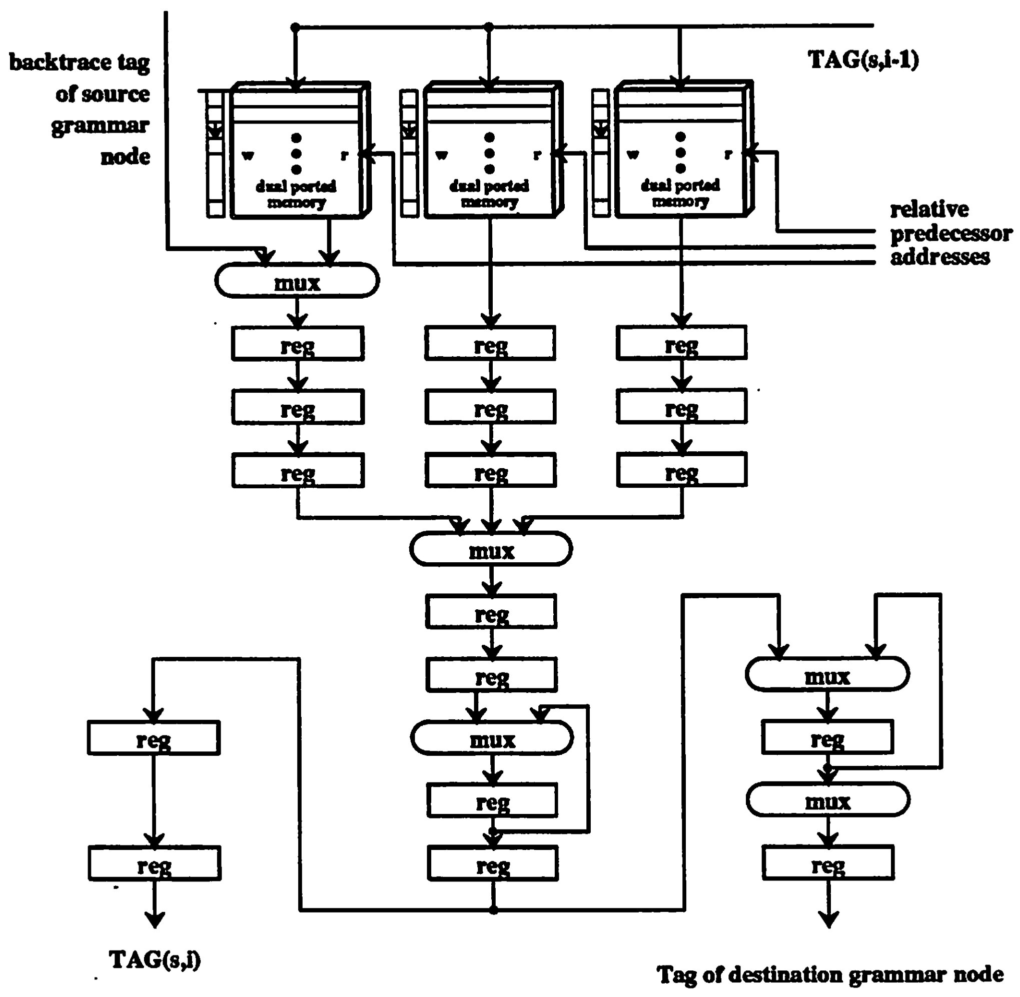

backtrace tag of source grammar node
TAG(s,i-1)
w
r
dual ported memory
relative predecessor addresses
mux
reg
TAG(s,i)
Tag of destination grammar node

Figure 5.4: Architecture of the Backtrace Processor.

Chapter 6

Implementation Issues

6.1 Simulation of the Subsystem

The functionality of the system was simulated before the physical implementation. The tool we used was THOR, a behavioral simulator developed at Stanford University [3].

The behavior of the system was hierarchically described using two different languages, *CHDL* and *CSL*. *CHDL* uses the C programming language with some extensions to support the description of hardware. It is used to describe the behavior of functional blocks on the lowest level of the system hierarchy. These functional blocks are then interconnected using a pin oriented netlist description called *CSL* which supports arbitrary levels of hierarchy.

One problem in the system description was to find the appropriate level of detail in which the system should be described and to find the appropriate hierarchy level where functional modules should be described using CHDL. The tradeoffs are as follows:

If the description is too detailed or the hierarchy level where CHDL is used is too low the description of the whole system gets very complex. This complexity makes it difficult to change the description. It is however very likely that in this initial system simulation phase the description has to be changed frequently. On the other side, a detailed description is fairly easy to debug since every signal is directly observable using the monitors that are part of the THOR simulator. Also, the simulation description can be used directly for the actual implementation.

Choosing a description where the level of detail is very low, or where the hierarchy level in which CHDL is used is very high, means that the hardware is not described in every detail. Subsequently, a new design process has to be entered to change from this high level description to the description using the SDL syntax accepted by the LagerIV System [8]. Verifying the functionality of this high level description is not straightforward, since in the functional *CHDL* description signal values are coded as variables. To observe these values the signal monitoring support offered by THOR can not be utilized, thus a standard software debugging tool has to be used. However, changes can be made very easily which is important in this early design process. Also, since the description is very compact, it can be accomplished relatively fast.

The way the system was actually described was to use CHDL for describing macro blocks on the hierarchy level of data paths, controllers, memories and so on. The description however was detailed enough to reflect data transitions between pipeline stages within these modules using variables that reflect the various registers. To model a register, two variables were used so that distinct values for the master and the slave state of a master slave register can be distinguished. This hierarchy level of macrocells was the lowest level used in the simulation description (see appendix A).

The next higher hierarchies were then described using a hierarchical CSL description that specifies the interconnection of these macrocells (see appendix A). To minimize the amount of interconnections only logically necessary signals were used. That means for example that the description of the clock consists only of the master and the slave clock, there were no inverted master or inverted slave clock specified as it is necessary for the actual description.

6.2 Test Generation Program

To automatically generate test vectors for processors following the scanpath testing strategy the program **tpgen** has been written [15]. It uses a novel approach to test pattern generation by using the functionality of basic logic blocks like adders or multiplexors as opposed to the functionality of gates. The algorithm used is a derivative of the classical D-Algorithm. It recursively generates input vectors by using implication, that means by trying to ripple input test patterns of a certain logic block in a bigger network of logic blocks to the primary inputs of the network. The result vectors are generated using the propagation approach of the classical D-Algorithm, that means, by trying to set the blocks at the outputs of a certain logic block such that they propagate the result of this block to the primary output.

The program reads the hierarchical OCT Structure Instance View [9], which is a detailed description of the processor designs. This description is automatically generated in the process of generating a chip using the LagerIV Silicon Assembly System [8]. In the first step the program extracts the topology of the scanregister chain. Then, starting from the inputs of the individual scanregisters, it recursively propagates through the logic blocks that are the sources of that register and generates test patterns for testing the blocks along with the expected result vectors.

When the chips came back from fabrication, the state of the program was not yet fully debugged so that the test patterns were actually generated manually. However, the program took over the tedious and error prone process of recovering the scanpath topology.

6.3 Layout Issues

This section gives a short discussion on some decisions that where made on the layout level.

Due to the huge amount of pads needed to accomplish a high off chip communication bandwidth the area of all the chips is determined by the pad frame rather than by the circuitry. To minimize the chip area narrow pads with a pitch of 175 μm have been used.

The registers used to delay control signals as well as data signals are implemented using a data path macrocell. This has the problem, especially when control signals have to be delayed, that outputs of a single bit register can not be routed to the periphery of the data path macrocell, only the N bit register bus as a unit. However, since the registers are optimized for data path macros and due to the uniform topology of the data paths this approach still results in reasonable area consumption.

The control logic for the data paths (see Fig. A.2) was implemented using the standardcell design approach because of it's flexibility and ease of design. However, the trade off is that due to hierarchical routing two routing ares are generated which results in a substantial waste of area. The solution for this problem would be to minimize the glue logic along with the number of connections needed by introducing an additional slice with buffers to the bitsliced data path. This buffer slice would be part of the data path where routing to the data path is done by abutment. To minimize the area allocated for the routing channel it would be desirable to partially flatten the design. However, since all the designs of this subsystem are pad limited rather than circuit limited these solutions were nor pursued.

The topology of the clock distribution network is set up in such a way that the clock is locally buffered at the registers. This is also done using standardcells. With this approach, the clock scew is uniformly distributed along the chip since the branches of the clock network have the same capacitive load, which is the input of the standardcell buffer. Also, only the master and the slave clocks are distributed over the chips, while the inverted clocks are generated locally.

Chapter 7

Conclusion

The architecture and implementation of a word processing subsystem for a real time speech recognition system using Hidden Markov Models has been described. The bottleneck of this system, which is the acquisition of data, has been demonstrated and an architecture that speeds up this bottleneck using on chip dual ported cache memories has been presented.

I want to conclude this report by shortly reflecting about the following issues: how much time was spent on the different design stages, what was done wrong and what right.

- Where did the time go?

  Most of the time was spent in defining the architecture of the system. This design phase involved understanding the algorithm well enough in order to be able to predict the influence of architectural decisions on performance figures like computational speed or recognition accuracy. We spent a lot of time "negotiating" with the algorithm experts, the objective was mostly versatility of the system versa complexity of the hardware implementation. One outcome of these negotiations is that the hardware only supports a statistical grammar and not a variety of grammars like finite-state grammars or unification grammars. The total time spent on this phase was roughly 6 months.

After this design phase, code for the behavioral simulation of the system had to be written. This also involved the architectural design of the chips and the coding of behavioral descriptions of every single functional module on the system. All in all, this task took about 3 months.

Once the behavioral description was in place, the custom chips could be designed and verified. Due to the use of sophisticated design tools (LagerIV) this stage took only 2 months, even though it involved the design of basic library modules (scan registers and dual ported RAMs). The verification of the chip designs was also facilitated by the fact that the behavioral simulator that was used to simulate the system behavior (THOR) could drive the simulator that was used on the layout level (IRSIM). Thus, no additional test patterns had to be generated an the correctness of the chip design was verified by automatically comparing the outcome of the two simulations.

Finally boards had to be designed to test the chips: A general purpose board that plugs into an IBM PC and has the task of writing and reading scantest vectors to the device under test (DUT) and a special purpose board that contains a chip carrier and the necessary connections to test the DUT. This was a rather time consuming task since the chips had many pins (208) that had to be wire wrapped. In order to test the pads also discrete boundary scan registers were implemented on the custom test board. The general purpose scantest board had the problem that it shorted out portions of the microprocessor bus thus destroying the mother board. The reason of this failure was mechanically, caused by bending the board. All in all, this design stage took roughly 2 months.

Finally, it should be noted that these times don't reflect "man months", but rather "student months". This means, included in those time figures are tasks like classwork and preliminary examinations.

- What went wrong?

In order to facilitate the board design for the first prototype system, we decided not ot implement pruning on the word level: the state probabilities of all words in the vocabulary were updated in every frame. On the other side, we made the decision to use high density dynamic memories to store all the necessary data. Both of these decisions substantially influenced the performance. But had we implemented word pruning in conjunction with using fast static memories the performance of the system would have been better by a factor of eight: the system could have been operated with 20 MHz as opposed to 5 MHz and only 50% of all the words would have to be processed. Since the use of DRAMs made the board design more complex than if we had used SRAMS, we could have had a better system with presumably the same board complexity.

Another architectural decision that increased the board complexity was that all the memories were implemented as dual ported memories: They can be accessed by the host processor via the VME bus as well as by the custom chips on the board. This required a vast amount of discrete components like multiplexors and tristate buffers to properly select the address and data busses. The better solution would have been to always access the memories via the custom chips. If the host processor wanted to access a memory location he then had to do it giving a control word and the address to the custom processor. Thus the multiplexing circuitry could be part of the custom IC's, which complicates the IC design but in return makes the board design a lot easier.

- what went right?

The decision to design the processors in such a way that they can operate with higher speed (about 20 MHz) proved to be a lucky one: the next generation system implements natural language processing to further constrain the number of possible word sequences. However, this increases the task that has to be performed by the word processing subsystem. Replacing the dynamic memories with static memories already increases the system performance by a factor of 4 without the need to redesign the processors. The fact that the viterbi processor also implements functions to support pruning on the wordlevel (computation of the best probability within a word and the possibility to stall the processors) furthermore boosts the performance by a factor of two, again without changing the custom processors. The testing strategy that was chosen also turned out to be an important asset: once the scan test board used in conjunction with an PC was in place, the task of testing the processors was a very easy one.

All the custom processors have been designed using the Berkeley LagerIV Silicon Assembly System. The architecture was described in a textual form and the layout data were completely automatically generated. The chips have been fabricated through MOSIS using a 2μm CMOS nwell technology. The functionality of the processors was successfully tested using the scanpath test methodology, and all the processors were first time working silicon.

Appendix A

Detailed Description of the Viterbi Processor

A.1 Block Diagrams

While the functionality of the processor as well as it's architecture was described in chapter 5, this chapter mainly gives block diagrams to show the physical implementation of the Viterbi processor.

Fig.A.1 gives an overview how the various blocks of the Viterbi processor are hierarchically composed and interconnected. The hierarchy of the interlaced boxes reflects the hierarchy of the textual description using *SDL* (structural description language, [8]). The names of the blocks in Fig.A.1 give the prefix of the corresponding files of the textual design description. The meaning of the suffices of the filenames are the following:

name.**sdl** : description of the pure data path without control logic.

*name*_**ctr.sdl** : sdl file of the control logic associated with *name*. These files describe a standard cell design that is used to generate control signals for data paths as well as generating the clocks for the scanpath registers. Fig.A.2 shows a block diagram of the circuitry used to generate these clocks.

*name*w**ctr.sdl** : sdl file that combines *name* and *name*_ctr.sdl.

name.**bds** : BDSYN [9] description of logic (only used for the sequencer).

name.**eqn** : Equations used to automatically generate the file *name*_ctr.sdl using Mani's program *eqn2sdl*.

Figures A.3 - A.8 show the detailed block diagrams of the various data paths and controllers of the processor. It should be pointed out that the adders shown in these diagrams are actually saturating adders that are implemented using a 2:1 multiplexor in series with an adder. Also, the diagrams don't show the low level control signals like the clocks, the inverted clocks and the inverted control signals for the multiplexors. Fig.A.10 shows the state diagram of the finite state machine "sequencer" that is used in the controller.

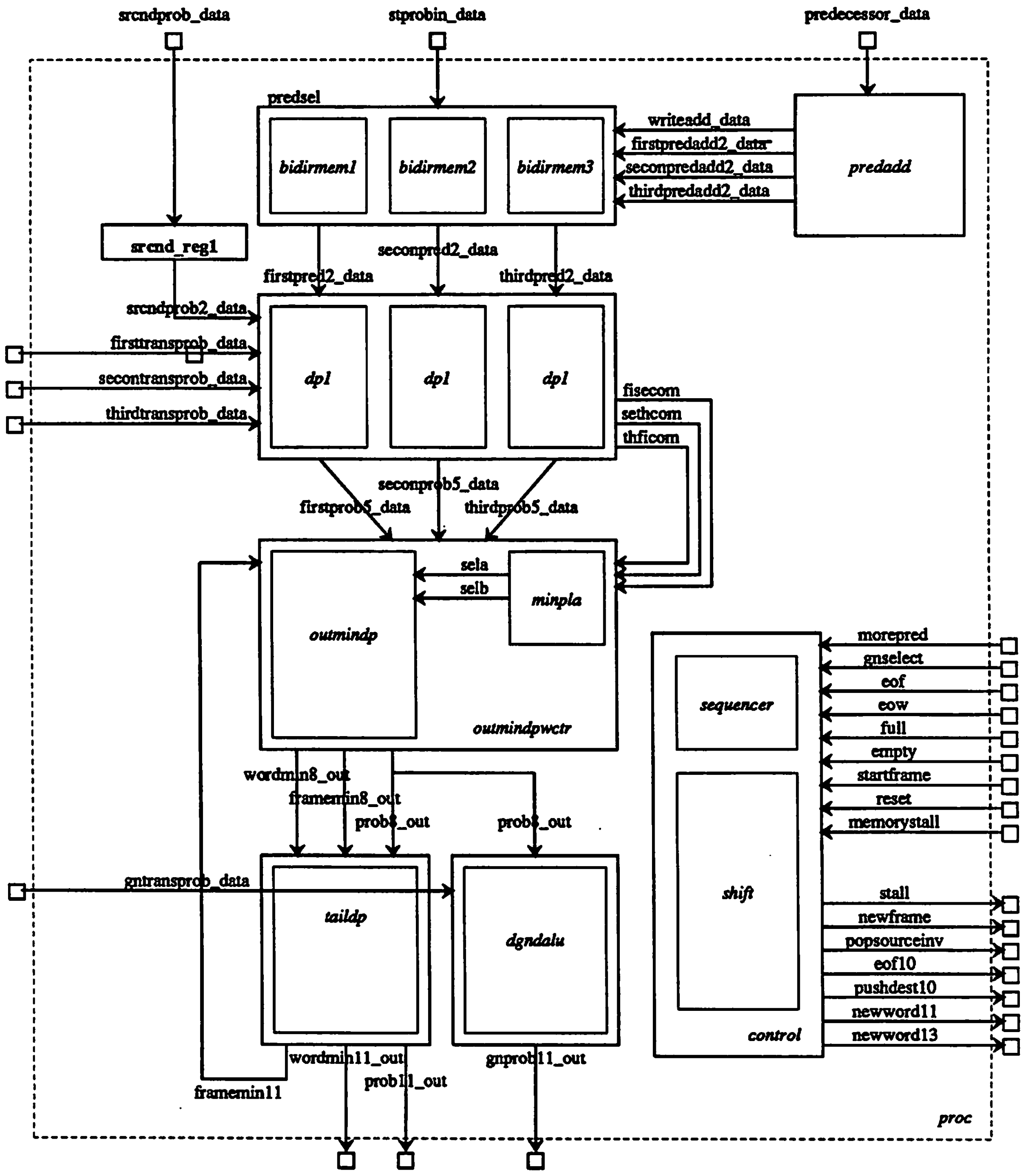


Figure A.1: Block diagram of the Viterbi Processor

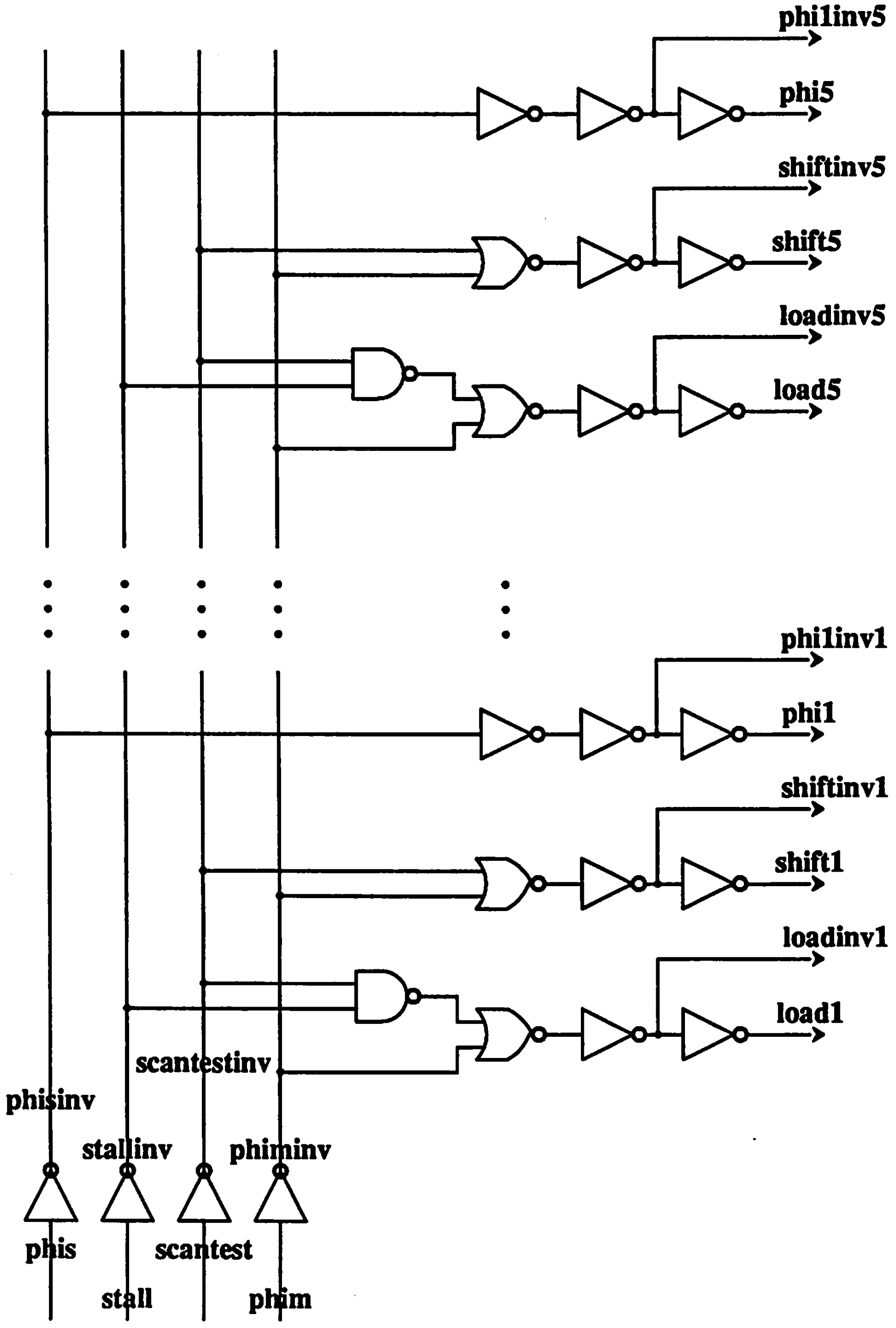


Figure A.2: Block diagram of the clock generation circuit

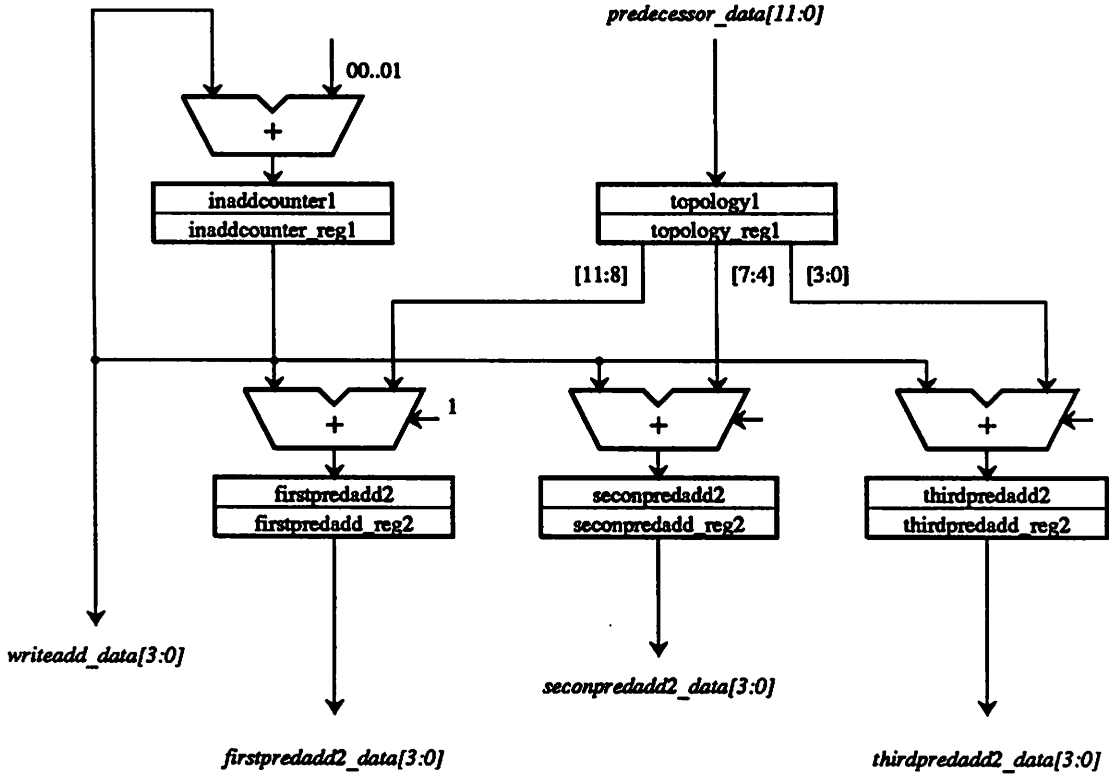


Figure A.3: Block diagram of address computation unit

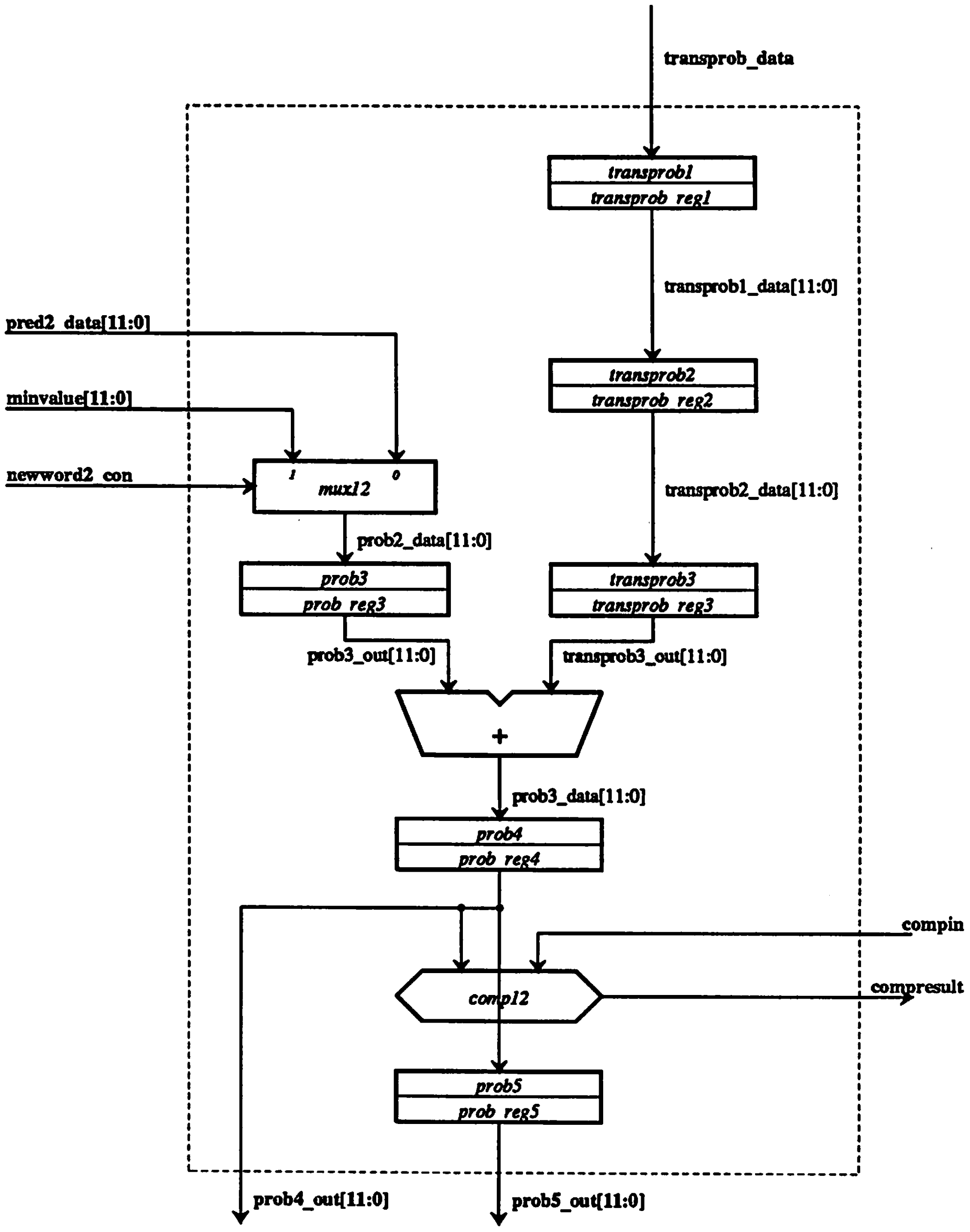


Figure A.4: Block diagram of the data path *dp1*

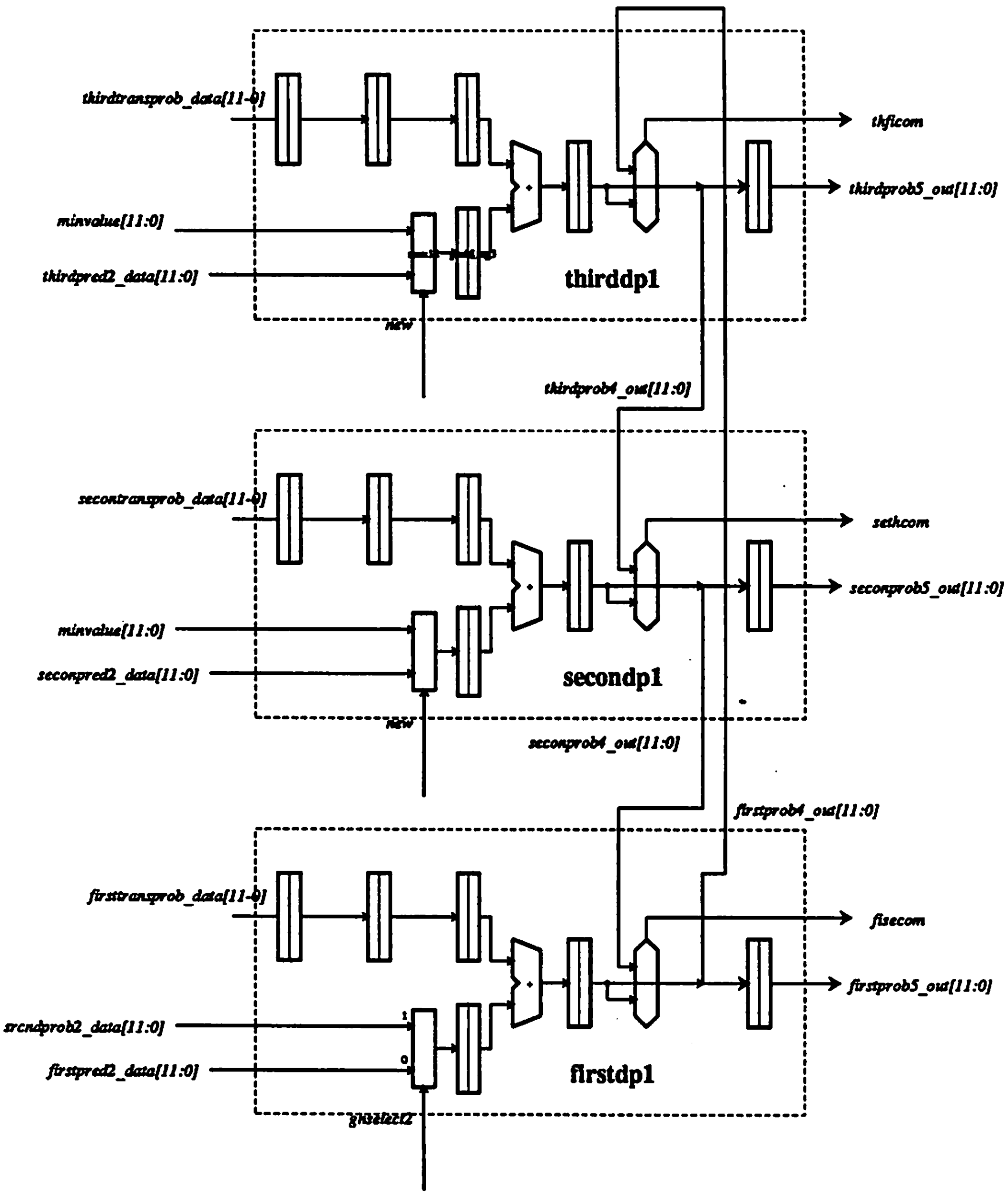


Figure A.5: Block diagram of the data path *predcom*

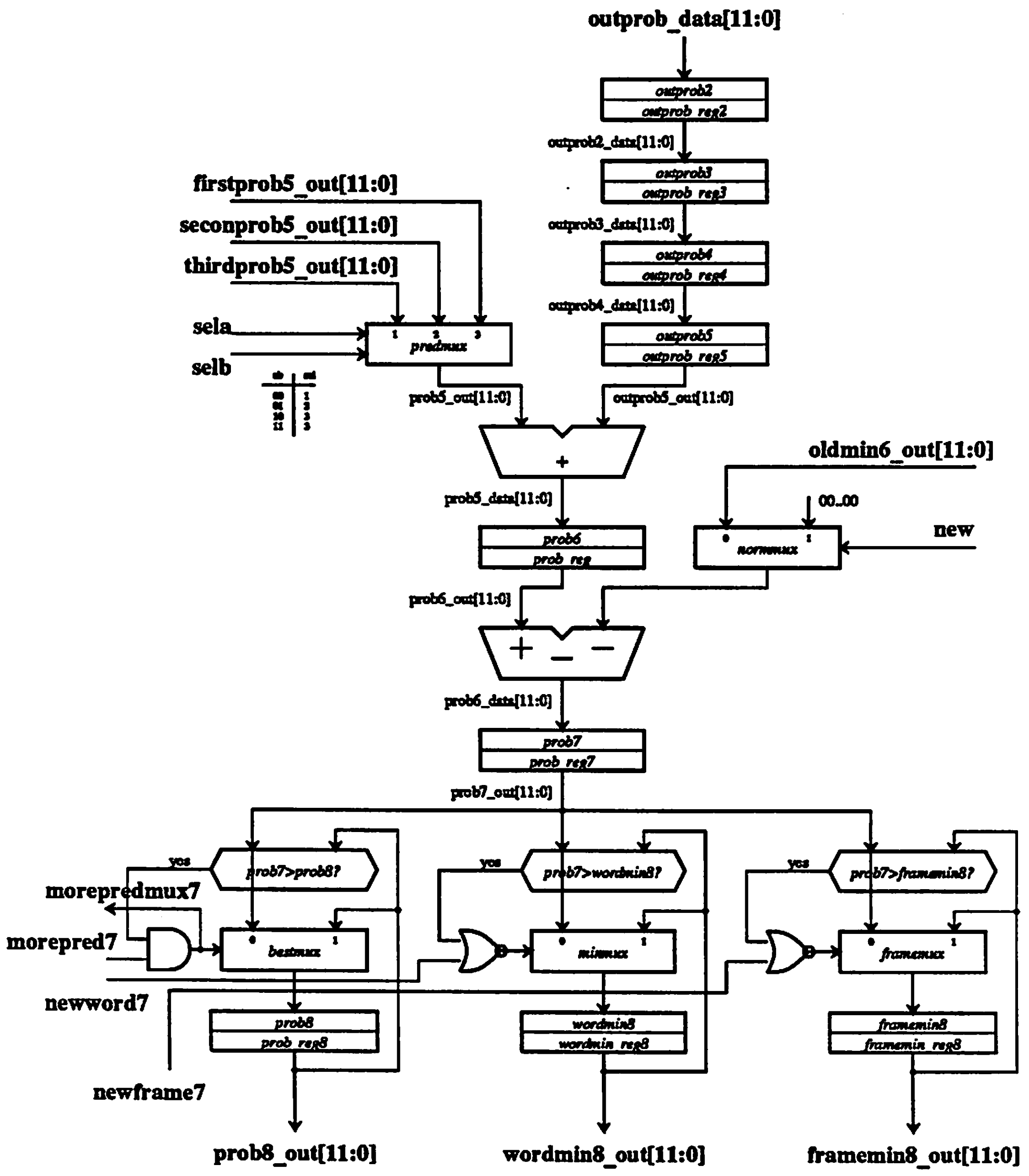


Figure A.6: Block diagram of the data path *outmindp*

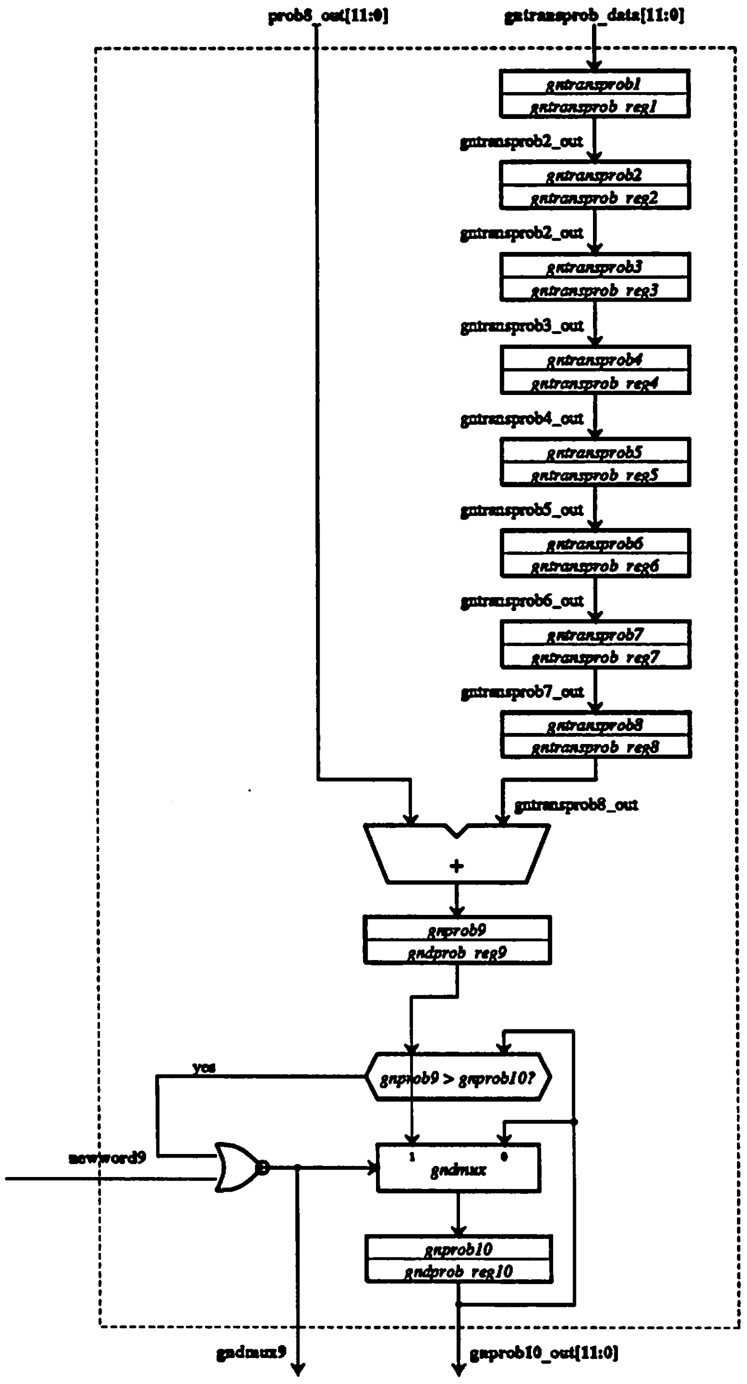


Figure A.7: Block diagram of the data path *dgndalu*

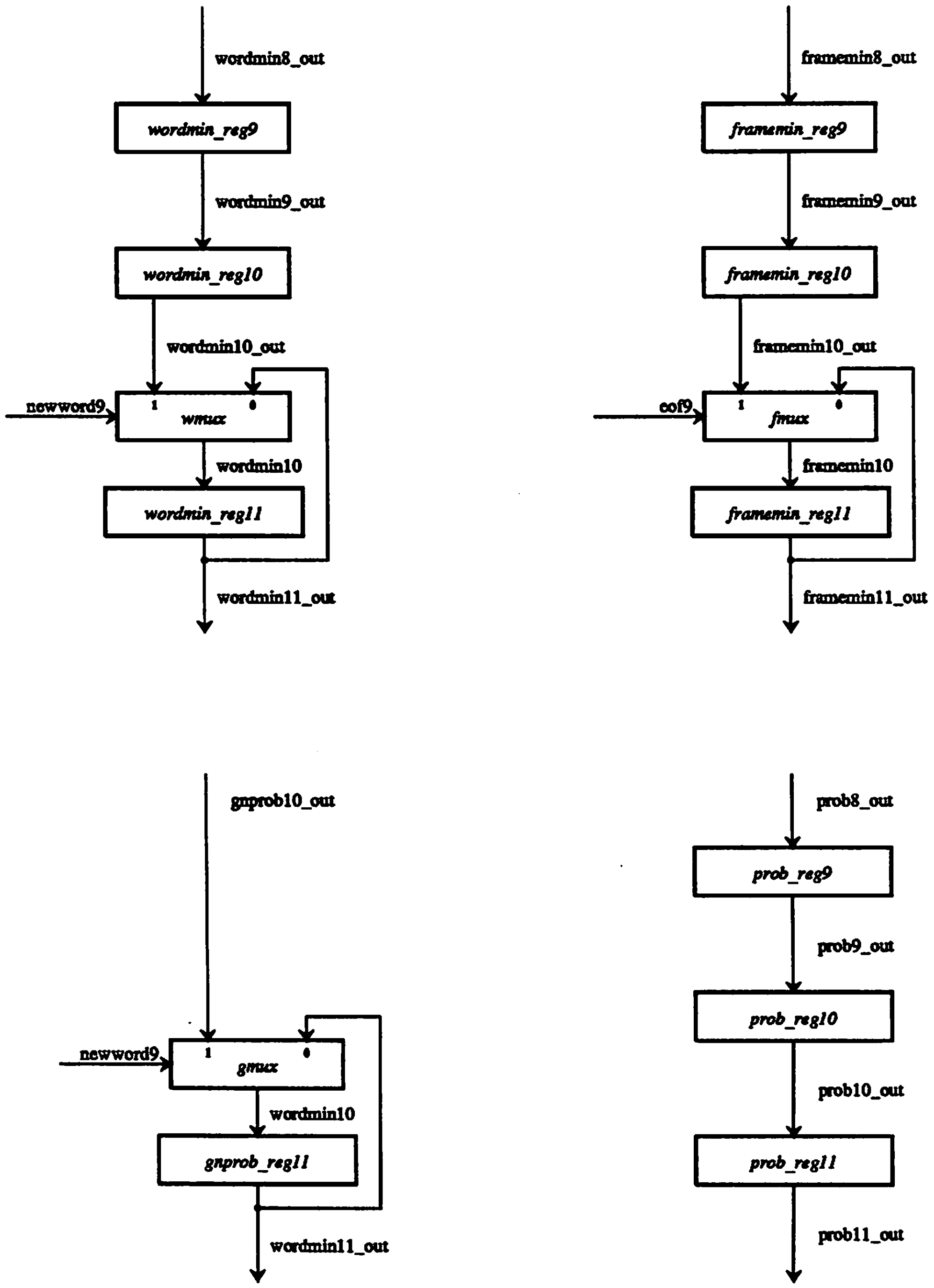


Figure A.8: Block diagram of the data path *taildp*

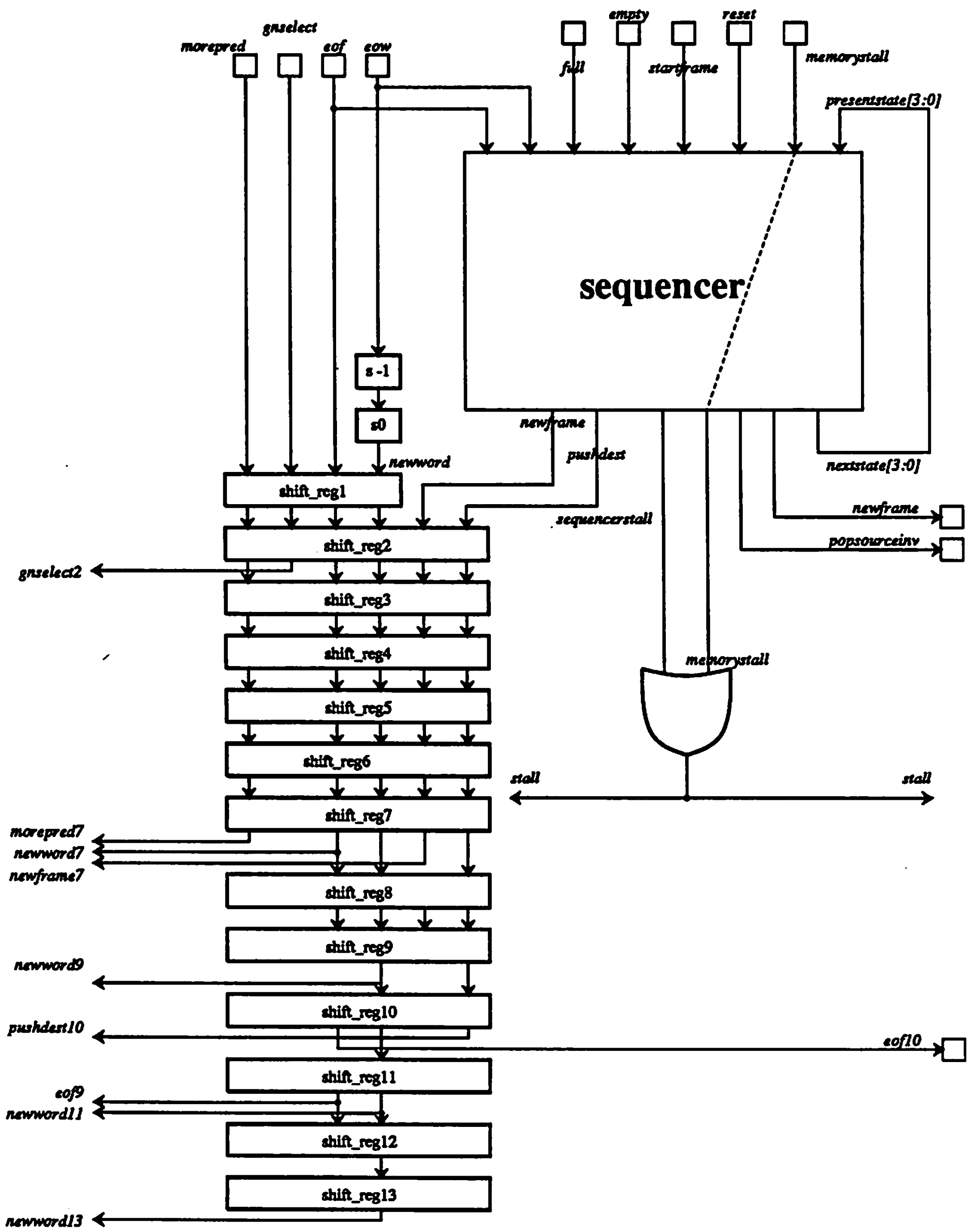


Figure A.9: Block diagram of the controller

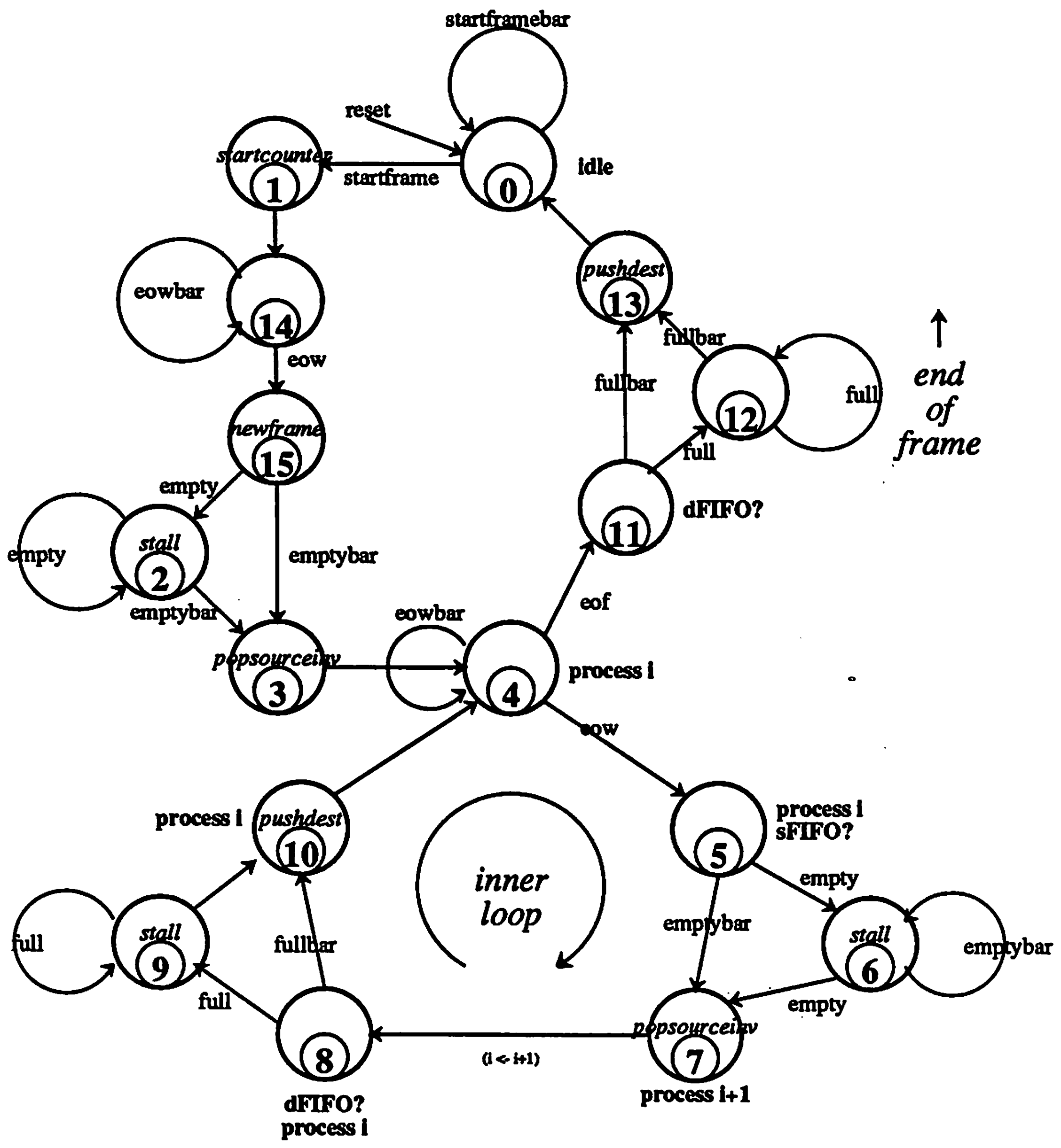


Figure A.10: State diagram of the sequencer

A.2 Input and Output Pins

This is a listing of all the pins of the Viterbi Processor along with their logic meaning:

stprobin_data 14 bit data bus that inputs the state probabilities at time $t-1$ from the state probability memories.

outprob2_data 8 bit data bus, used to input the output probabilities $b_{j,k}$ from the output probability memory. The data on this bus have to be delayed by three clock cycles. This takes care of the fact that the output probability memory with the output probability memory lookup table introduces this delay.

srcndprob_data 14 bit bus that inputs the source grammar node probabilities from the FIFO that interfaces the word processing subsystem with the grammar processing subsystem.

morepredmux7 Control output that has to be connected to the Backtrace Processor. It is active if a state has more than 3 predecessors and if the first set of predecessors has probability values that are better than the second set of predecessors.

dgnenable input control pin to indicate that a state has a non-zero transition probability to the destination grammar node.

odpscanout scanout of the data path *outmindp.*

phis input of the slave clock.

phim input of the master clock.

udpscanout scanout of the data path *upperdp.*

scantest control input signal for scantest.

new control input signal, valid during the first frame of the recognition. The signal should be asserted by the μprocessor to indicate that all the values read from the state probability memories have to be disregarded.

sela control output signal that has to be connected to the Backtrace Processor. Indicates (together with selb) which predecessor has to be selected.

selb control output signal that has to be connected to the Backtrace Processor. Indicates (together with selb) which predecessor has to be selected.

odpscanin scan input for the data path *outmindp.*

gntransprob_data 8 bit input bus for the destination grammar node transition probabilities read from the topology memory.

gndmux9 control output signal that has to be connected to the backtrace processor. It is used to select the overall best destination grammar node probability within a word.

dascanin scan input for the data path *dgndalu*.

dascanout scan output for the data path *dgndalu*.

gnprob11_out 14 bit output bus with for the destination grammar node probability. This bus should be connected to the FIFO that interfaces to the grammar processing subsystem.

prob11_out 14 bit output bus for the state probabilities at time *t*. This bus has to be connected to the state probability memories. The data on this bus are delayed by 11 cycles relative to the current state (relative to the address counter).

wordmin11_out 14 bit output bus for the best probability value within a word. This value is needed by the Backtrace Memory Processor to generate a pruning threshold.

startcounter control output signal used to reset the addresscounter in the Backtrace Processor.

tdpscanout scan output for the data path *taildp*.

tdpscanin scan input for the data path *taildp*.

pcscanout scan output for the data path *predcom*.

pcscanin scan input for the data path *predcom*.

thirdtransprob_data 8 bit input bus for the transition probability from the first predecessor. This bus has to be connected to the topology memory.

secontransprob_data 8 bit input bus for the transition probability from the second predecessor. This bus has to be connected to the topology memory.

firsttransprob_data 8 bit input bus for the transition probability from the third predecessor. This bus has to be connected to the topology memory.

phi14 output signal to probe the slave clock generated internally on the chip.

startframe input control signal that indicates the start of a frame. This signal does not have to be synchronized to the clock on the subsystem.

memorystall input control signal that indicates that one of the memories can not be accessed. The action that the Viterbi Processor takes in this case is to stop any operation by keeping the current states in all the internal registers.

full input control signal that indicates that the FIFO containing the generated destination grammar node probabilities is full. The action that the Viterbi Processor takes in this case is to stop any operation by keeping the current states in all the internal registers.

empty input control signal that indicates that the FIFO containing the source grammar node probabilities is empty. The action that the Viterbi Processor takes in this case is to stop any operation by keeping the current states in all the internal registers.

reset input control signal that puts the processor in the reset state.

popsourceinv output control signal that pops the FIFO containing the source grammar node probabilities.

newframe control output signal used to start the addresscounter. This synchronous signal is generated as a result of the asynchronous startframe signal asserted by the μprocessor.

eof control input from topology memory. Used to indicate that all states of the vocabulary have been processed ($\equiv$ end of a frame).

eow control input from topology memory. Used to indicate the end of a word. This signal has to be valid one state *before* the final state of a word.

pushdest13 control output signal to the backtrace memory processor. This signal conceptually implements a slave clock since it is the delayed version of the **pushdest11** signal.

load4 output signal to probe the master clock generated internally on the chip.

pushdest11 control output signal to push a new destination grammar node probability into the FIFO that interfaces with the grammar processing subsystem. This signal is also used to trigger the Backtrace Memory Processor.

newword9 control output meant to interface with the Backtrace Processor.

eof10 control output signal used to reset and stop the address counter on the subsystem. This signal has to be still delayed by 2 clock cycles or, the last word in the vocabulary has to be a dummy word.

controlscanin scanin signal for the controller.

controlscanout scanout signal from the controller.

gnselect control input signal from the topology memory. It is high whenever a state in a wordmodel has the source grammar node as predecessor.

gnselect2 delayed version of **gnselect**. This signal is needed by the Backtrace Processor.

morepred control input signal from the topology memory. It indicates that a state has more than three predecessors.

predecessor_data 12 bit input bus containing the three 4 bit offset addresses of the predecessors. These offsets are coded using the two's complement notation. Bit 3 to 0 contain the (negative) offset address of the third predecessor, bits 7 to 4 the offset address of the second and bits 11 to 8 the offset address of the first predecessor. Bit 0 is the least significant bit.

udpscanin scan input for the macrocell *upperdp*.

stall control output signal that indicates if the processor stalls its operation. It has to be connected to the Backtrace Processor.

Appendix B

Thor description of the Subsystem

This appendix shows THOR descriptions of two different hierarchy levels in the subsystem. The first example shows the lowest level of hierarchy, the description of the behavior of the data path *dp1* (see Fig.A.1, Fig.A.4). The syntax of the description is CHDL (C hardware description language), an extended C syntax [3].

```
MODEL(dp1)
{
   IN_LIST
      SIG(phim);
      SIG(phis);
      SIG(stall);
      GRP(transprob_data, 14);
      GRP(pred2_data, 14);
      GRP(minvalue, 14);
      GRP(compin, 14);
      SIG(newword2_con);
   ENDLIST;
   OUT_LIST
      GRP(prob4_out, 14);
      GRP(prob5_out, 14);
      SIG(compresult);
   ENDLIST;
   ST_LIST
      GRP(transprob1, 14);
      GRP(transprob2, 14);
      GRP(transprob3, 14);
      GRP(transprob_reg1, 14);
      GRP(transprob_reg2, 14);
      GRP(transprob_reg3, 14);
      GRP(prob3, 14);
      GRP(prob4, 14);
      GRP(prob5, 14);
      GRP(prob_reg3, 14);
      GRP(prob_reg4, 14);
      GRP(prob_reg5, 14);
   ENDLIST;

   if (stall == ONE)   EXITMOD(0);

   if ((phim == UNDEF) || (phis == UNDEF))  {
      fprintf(stderr, "dp1:  phim || phis undefined\n");
      EXITMOD(0);
   }
   if (phim == ONE)   {
/*
 * PIPELINESTEP 1:
 */
         if (fckbin(transprob_data,13,0) != PASSED) {
            fprintf(stderr, "dp1:  transprob_data[13:0] undefined\n");
            EXITMOD(0);
         }
         transprob1[] = transprob_data[];
/*
 * PIPELINESTEP 2:
 */
         transprob2[] = transprob_reg1[];
/*
 * PIPELINESTEP 3:
 */
         if (fckbin(pred2_data,13,0) != PASSED) {
            fprintf(stderr, "dp1:  pred2_data[13:0] undefined\n");
            EXITMOD(0);
         }
         if (fckbin(minvalue,13,0) != PASSED) {
            fprintf(stderr, "dp1:  minvalue[13:0] undefined\n");
            EXITMOD(0);
         }

         transprob3[] = transprob_reg2[];
         if (newword2_con == ZERO)  prob3[] = pred2_data[];
      else                              prob3[] = minvalue[];
/*
 * PIPELINESTEP 4:
 */
      if ((prob_reg3[] + transprob_reg3[]) > 4095)
         fsetword(prob4, 13, 0, ONE);
      else prob4[] = prob_reg3[] + transprob_reg3[];

/*
 * PIPELINESTEP 5:
 */
      if (fckbin(compin,13,0) != PASSED) {
         fprintf(stderr, "dp1:  compin[13:0] undefined\n");
         EXITMOD(0);
      }
      if (prob_reg4[] <= compin[])  compresult = ONE;
      else                          compresult = ZERO;
      prob5[] = prob_reg4[];
   }

   if (phis == ONE)  {
      transprob_reg1[] = transprob1[];
      transprob_reg2[] = transprob2[];
      transprob_reg3[] = transprob3[];
      prob_reg3[] = prob3[];
      prob_reg4[] = prob4[];
      prob_reg5[] = prob5[];
   }
   prob4_out[] = prob_reg4[];
   prob5_out[] = prob_reg5[];
   EXITMOD(0);
}
```

The second example describes the interconnection of blocks according to Fig.A.5. The syntax used in this hierarchy level is *csl* [3] which supports a pin oriented netlist description.

```
(sub = predcom)
   (i=
      phim,
      phis,
      stall,
      newword2,
      gnselect2,
      minvalue[0-13],
      srcndprob2_data[0-13],
      firsttransprob_data[0-13],
      firstpred2_data[0-13],
      secontransprob_data[0-13],
      seconpred2_data[0-13],
      thirdtransprob_data[0-13],
      thirdpred2_data[0-13])
   (o=
      firstprob5_out[0-13],
      seconprob5_out[0-13],
      thirdprob5_out[0-13],
      fisecom,
      sethcom,
      thficom)

       {
         (f=../dp1/dp1)(n=firstdp1)
         (i= phim,phis,stall,
             firsttransprob_data[0-13], firstpred2_data[0-13],
             srcndprob2_data[0-13], seconprob4_out[0-13], gnselect2)
            (o= firstprob4_out[0-13], firstprob5_out[0-13], fisecom);

         (f=../dp1/dp1)(n=secondp1)
         (i= phim,phis,stall,
             secontransprob_data[0-13], seconpred2_data[0-13],
             minvalue[0-13], thirdprob4_out[0-13], newword2)
            (o= seconprob4_out[0-13], seconprob5_out[0-13], sethcom);

         (f=../dp1/dp1)(n=thirddp1)
   (i= phim,phis,stall,
       thirdtransprob_data[0-13], thirdpred2_data[0-13],
       minvalue[0-13], firstprob4_out[0-13], newword2)
      (o= thirdprob4_out[0-13], thirdprob5_out[0-13], thficom);

}
```

List of Figures

Bibliography

[1] Y.L. Chow, M.D. Dunham, O.A. Kimball, M.A. Krasner, G.F. Kubala, J. Makhoul, P.J . Price, S. Roucos, and R.M. Schwartz. Byblos: The bbn continuous speech recognition system. In *Proc ICASSP 87: 1987 International Conference on Acoustics Speech and Signal Processing*, pages 89–92, April 1987.

[2] W. Drews, R. Laroia, J. Pandel, A. Schumacher, and A. Stölzle. A cmos processor for a 1000 word speech recognition system. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 559–562, May 1987.

[3] VLSI/CAD Group. *THOR*. Stanford University, release 3.2 edition, 1986.

[4] Robert A Kavaler. *The Design and Evaluation of a Speech Recognition System for Engineering Wordstations*. PhD thesis, University of California at Berkeley, May 1986.

[5] K.Lee and H. Hsiao-Wuen. Large vocabulary speaker-independent continuous-speech recognition using hmm. In *Proc. ICASSP 88: 1988 International Conference on Acoustics Speech and Signal Processing*, pages 123–126, April 1988.

[6] K.Lee, H. Wuen, and M.-Y. Hwang. recent progress in the sphinx recognition system. In *Proc Speech and Natural Language Workshop*, pages 125–130, February 1989.

[7] B H Juang L R Rabiner. An introduction to hidden markov models. *IEEE ASSP Magazine*, pages 4–16, January 1986.

[8] Electronic Research Laboratory. *LagerIV Distribution 1.0 Silicon Assembly System Manual.* University of Claifornia at Berkeley, June 1988. Distribution 1.0.

[9] Electronic Research Laboratory. *Oct Tools Distribution 2.1.* University of Claifornia at Berkeley, March 1988. Distribution 2.1.

[10] H.-D. Lin and D. Messerschmitt. High speed viterbi decoding. Submitted to IEEE Trans. on Communications, 1989.

[11] H. Murveit, M. Cohen, P. Price, G. Baldwin, M. Weintraub, and J. Bernstein. Sri's decipher system. In *Proc Speech and Natural Language Workshop*, pages 238–242, February 1989.

[12] J. Rabaey, R. Brodersen, A. Stölzle, S. Narayanaswamy, D. Chen, R. Yu, P. Schrupp, H. Murveit, and A. Santos. *VLSI Signal Processing III*, chapter A Large Vocabulary Real Time Continuous Speech Recognition System, pages 61–74. IEEE Press, 1988.

[13] J. Rabaey, R. Brodersen, A. Stölzle, S. Narayanaswamy, D. Chen, R. Yu, P. Schrupp, H. Murveit, and A. Santos. Real-time large-vocabulary speech recognition. In *Proc ICASSP '89, 1987 International Conference on Acoustics Speech and Signal Processing*, May 1989.

[14] R.M. Schwartz, C. Barry, Y.L. Chow, A. Derr, M.-W. Feng, O. Kimball, F. Kubala, J. Makhoul, and J. Vandegrift. The bbn byblos continuous speech recognition system. In *Proc Speech and Natural Language Workshop*, pages 94–99, February 1989.

[15] A. Stölzle. Tpgen: Automatic tespattern generation for pipelined parallel processors. Final project report for eecs 244, University of California at Berkeley, EECS Cory Hall, Fall '88 1989.