Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2006, Article ID 48085, Pages 1–19
DOI 10.1155/ES/2006/48085

Speech Silicon: An FPGA Architecture for Real-Time Hidden Markov-Model-Based Speech Recognition

Jeffrey Schuster, Kshitij Gupta, Raymond Hoare, and Alex K. Jones

University of Pittsburgh, Pittsburgh, PA 15261, USA

Received 21 December 2005; Revised 8 June 2006; Accepted 27 June 2006

This paper examines the design of an FPGA-based system-on-a-chip capable of performing continuous speech recognition on medium-sized vocabularies in real time. Through the creation of three dedicated pipelines, one for each of the major operations in the system, we were able to maximize the throughput of the system while simultaneously minimizing the number of pipeline stalls in the system. Further, by implementing a token-passing scheme between the later stages of the system, the complexity of the control was greatly reduced and the amount of active data present in the system at any time was minimized. Additionally, through in-depth analysis of the SPHINX 3 large vocabulary continuous speech recognition engine, we were able to design models that could be efficiently benchmarked against a known software platform. These results, combined with the ability to reprogram the system for different recognition tasks, serve to create a system capable of performing real-time speech recognition in a vast array of environments.

1. INTRODUCTION

Many of today's state-of-the-art software systems rely on the use of hidden Markov model (HMM) evaluations to calculate the probability that a particular audio sample is representative of a particular sound within a particular word [1, 2]. Such systems have been observed to achieve accuracy rates upwards of 95% on dictionaries greater than 1000 words; however, this accuracy comes at the expense of needing to evaluate hundreds of thousands of Gaussian probabilities resulting in execution times of up to ten times the real-time requirement [3]. While these systems are able to provide a great deal of assistance in data transcription and other offline collection tasks, they do not prove themselves as effective in tasks requiring real-time recognition of conversational speech. These issues combined with the desire to implement speech recognition on small, portable devices have created a strong market for hardware-based solutions to this problem. Figure 1 gives a conceptual overview of the speech recognition process using HMMs. Words are broken down into their phonetic components called phonemes. Each of the grey ovals represents one phoneme, which is calculated through the evaluation of a single three state HMM. The HMM represents the likelihood that a given sequence of inputs, senones, is being traversed at any point in time. Each senone in an HMM represents a subphonetic sound unit, defined by the particular speech corpus of interest. These senones are generally composed of a collection of multivariant Gaussian distributions found through extensive offline training on a known test set. In essence, each HMM operates as a three-state finite-state machine that has fixed probabilities associated with the arcs and a dynamic "current state" probability associated with each of the states, while each word in the dictionary represents a particular branch of a large, predefined tree style search space.

The set of senones used during the recognition process is commonly referred to as the acoustic model and is calculated using a set of "features" derived from the audio input. For our research we chose to use the RM1 speech corpus which contains 1000 words, and uses an acoustic model comprised of 2000 senones [4]. The RM1 corpus represents the most common words used in "command-and-control" type tasks and can be applied to a large number of tasks from navigation assistance to inventory ordering systems. This particular dictionary also represents a medium-sized task (100–10 000 words) and presents a reasonable memory requirement for a system looking to be implemented as a single-chip solution. This corpus requires that every 10 milliseconds, 300 000 operations must be performed to determine the probability that a particular feature set belongs to a given multivariant

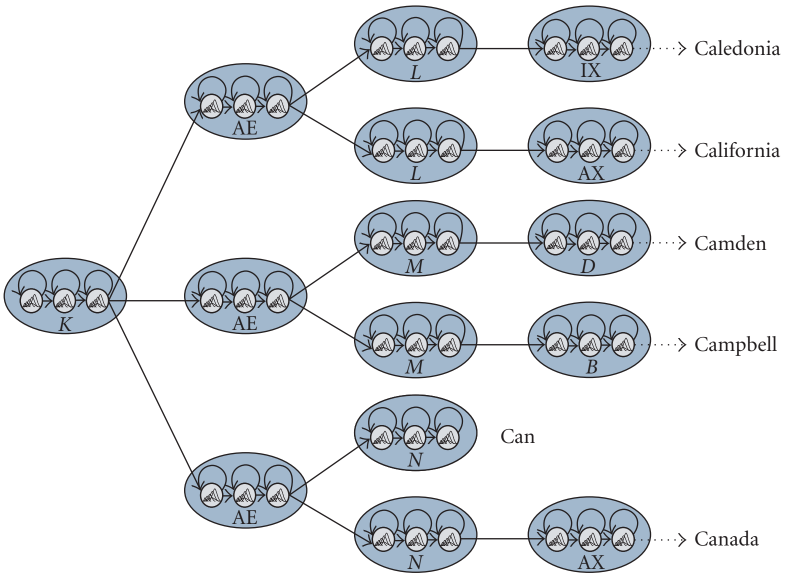


Figure 1: Conceptual overview of speech recognition using hidden Markov models.

Gaussian distribution, resulting in over 60 million calculations per second, just to calculate the senones.

1.1. Background

Although several years of research has gone into the development of speech recognition, the progress has been rather slow. This is a result of several limiting factors, amongst which recognition accuracy is the most important. The ability of machines to mimic the human auditory perceptory organs and the decoding process taking place in the brain has been a challenge, especially when it comes to the recognition of natural, irregular speech [5].

To date, however, state-of-the-art recognition systems overcome some of these issues for systems with regular speech structures, such as command- and control-based applications. These systems provide accuracies in excess of 90% for speaker independent systems with medium sized dictionaries [6]. Despite the satisfactory accuracy rate achieved for such applications, speech recognition has yet to penetrate our day-to-day lives in a meaningful way.

The majority of this problem stems from the computationally intensive nature of the speech recognition process, which generally requires several million floating-point operations per second. Unfortunately using general purpose processors (GPPs) with traditional architectures is inefficient due to limited numbers of arithmetic logic units (ALUs) and insufficient caching resources. Cache sizes in most processors available today, especially those catering towards embedded applications, are very limited: only on the order of tens of kBs [7]. Therefore, accessing tens of MBs of speech data using tens of kBs of on-chip cache results in a high cache miss rate thereby leading to pipeline stalls and significant reduction in performance.

Further, since several peripherals and applications running on a device need access to a common processor, bus-based communication is required. Thus, all elements connected to the bus are synchronized by making use of bus transaction protocols thereby incurring several cycles of additional overhead. Because of these inefficiencies, speech recognition systems execute less than one instruction per cycle (IPC) [1, 2] on GPPs. As a result, the process of recognizing speech by such machines is slower than real time [3].

To counter these effects, implementers have two options. They could either use processors with higher clock-rates to account for processor idle time caused by pipeline stalls and bus arbitration overheads, or they could redesign the processor that caters to the specific requirements of the application. Since software-based systems are dependent on the underlying processor architecture, they tend to take the first approach. This results in the need for devices with multi-GHz processors [1, 2] or the need to reduce the model complexity. However, machines with multi-GHz processors are not always practical, especially in embedded applications. The alternative is to reduce bit-precision or use a more coarse-grain speech model to decrease the data size. While this helps in making the system practically deployable, the loss in computational precision in most cases, leads to degraded performance (in terms of accuracy) and decreases the robustness of the system. For example, a speaker-independent system becomes a speaker-dependent system or continuous speech recognition moves to discrete speech recognition.

The second option involves designing a dedicated architecture that optimizes the available resources required for processing speech and allows for the creation of dedicated data-paths that eliminate significant bus transaction overhead.

Projects at the University of California at Berkeley, Carnegie Mellon University, and the University of Birmingham in the United Kingdom have made some progress with hardware-based speech recognition devices in recent years [8, 9]. These previous attempts either had to sacrifice model complexity for the sake of memory requirements or simply encountered the limit of the amount of logic able to be placed on a single chip. For example, the solution in [8] is to create a hardware coprocessor to accelerate a portion of speech recognition, beam search. The solution in [9] requires device training. In contrast, our work presents a novel architecture capable of solving the entire speech recognition problem in a single device with a model that does not require training through the use of task specific pipelines connected via shared, multiport memories. Thus, our implementation is capable of processing a 1000 word command and control-based application in real time with a clock speed of approximately 100 MHz.

The remainder of this paper describes the speech silicon project, providing an in-depth analysis of each of the pipelines derived for the system-on-a-chip (SoC). Specifically, we introduce a novel architecture that enables real-time speech recognition on an FPGA utilizing the 90 nm ASIC multiply-accumulate and block RAM features of the Xilinx Virtex 4 series devices. Final conclusions as well as a summary of synthesis and post place-and-route results will be given at the end of the paper.

2. THE SPEECH SILICON PROJECT

The hardware speech processing architecture is based on the SPHINX 3 speech recognition engine from Carnegie Mellon University [10]. Through analysis of this algorithm, a model of the system was created in MATLAB. As a result, complex statistical analysis could be performed to find which portions of the code could be optimized. Further, the data was able to be rearranged into large vectors and matrices leading to the ability to parallelize calculations observed to be independent of one another. Preliminary work on this topic has been discussed in [11, 12].

The majority of automatic speech recognition engines on the market today consist of four major components: the feature extractor (FE), the acoustic modeler (AM), the phoneme evaluator (PE), and the word modeler (WM), each presenting its own unique challenge. Figure 2 shows a block diagram for the interaction between the components in a traditional software system, with inputs from a DSP being shown on the left of the diagram.

The FE transforms the incoming speech into its frequency components via the fast fourier transform, and subsequently generates mel-scaled Cepstral coefficients through mel-frequency warping and the discrete cosine transform. These operations can be performed on most currently available DSP devices with very high precision and speed and will therefore not be considered for optimization within the scope of this paper.

The AM is responsible for evaluating the inputs received from the DSP unit with respect to a database of known Gaussian probabilities. It produces a normalized set of scores, or senones, that represent the individual sound units in the database. These sound units represent subphonetic components of speech and are traditionally used to model the beginning, middle, and end of a particular phonetic unit. Each of the senones in a database is comprised of a mixture of multivariant Gaussian probability distribution functions (PDFs) each requiring a large number of complex operations. It has been shown that this phase of the speech recognition process is the most computationally intensive, requiring up to 95% of the execution time [2, 13], and therefore requires a pipeline with very high bandwidth to accommodate the calculations.

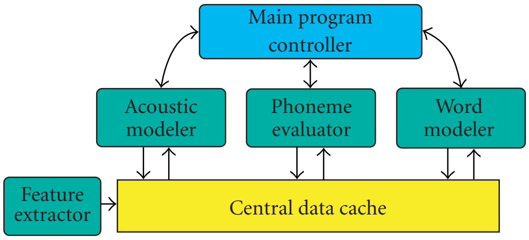


Figure 2: Block diagram of software-based automatic speech recognition system.

The PE associates groups of senones into HMMs representing the phonetic units, phonemes, allowable in the systems dictionary. The basic calculations necessary to process a single HMM are not extremely complex and can be broken down into a simple ADD-COMPARE-ADD pipeline, described in detail in Section 4. The difficulty in this phase is in managing the data effectively so as to minimize unnecessary calculations. When the system is operational not all of the phonemes in the dictionary are active all the time, and it is the PE that is responsible for the management of the active/inactive lists for each frame. By creating a pipeline dedicated to calculating HMMs and combining it with a second piece of logic that acts as a pruner for the active list, a two step approach was conceived for implementing the PE, allowing for maximal efficiency.

The WM uses a tree-based structure to string phonemes together into words based on the sequences defined in the system dictionary. This block serves as the linker between the phonemes in a word as well as the words in a phrase. When the transition from one word to another is detected, a variable penalty is applied to the exiting word's score depending on what word it attempts to enter next. In this way, basic syntax rules can be implemented in addition to pruning based on a predefined threshold for all words. WM is also responsible for resetting tokens found inactive by the PE. The pruning stage of the PE passes two lists to the WM, one for active tokens and the other for newly inactive tokens. Much like the PE, the WM takes a two stage approach, first resetting the inactive tokens and then processing the active tokens. By doing the operations in this order we ensure that while processing the active tokens, all possible successor tokens are available if and when they are needed.

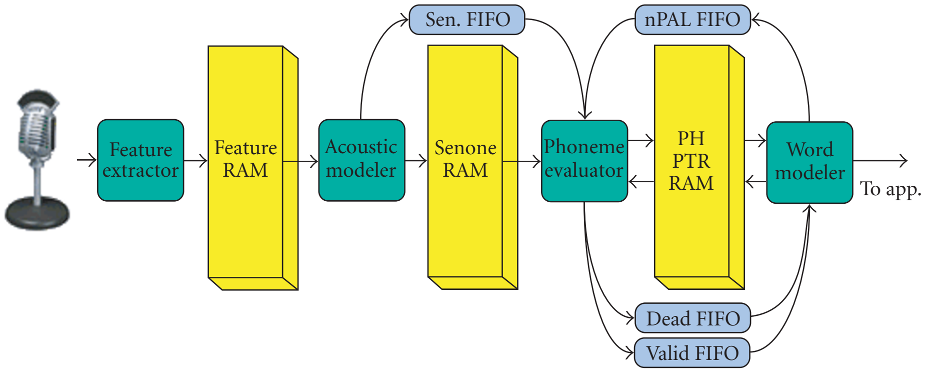


Figure 3: Block diagram of the *speech silicon* hardware-based ASR system.

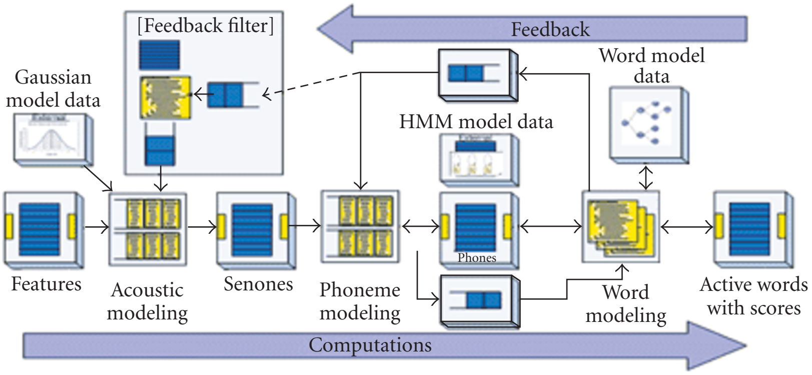


Figure 4: Conceptual diagram of high-level architecture.

When considering such systems for implementation on embedded platforms the specific constraints imposed by each of the components must be considered. Additionally, the data-dependencies between all components must be considered to ensure that each component has the data it requires as soon as it needs it. To complicate matters, the overall size of the design and its power consumption must also be factored into the design if the resultant technology is to be applicable to small, hand-held devices. The most effective manner for accommodating these constraints was determined to be the derivation of three separate cells, one for each of the major components considered, with shared memories creating interface between cells. To minimize the control logic and communication between cells, a token-passing scheme was implemented using FIFOs to buffer the active tokens across cell boundaries. A block diagram of the component interaction within the system is shown in Figure 3.

By constructing the system in this fashion and keeping the databases necessary for the recognition separate from the core components, this system is not bound to a single dictionary with a specific set of senones and phonemes. These databases can in fact be reprogrammed with multiple dictionaries in multiple languages, and then given to the system for use with no required changes to the architecture. This flexibility also allows for the use of different model complexity in any of the components, allowing for a wide range of input models to be used, and further aiding in the customizability of the system. Figure 4 shows a detailed diagram of the high-level architecture of the speech recognition engine.

2.1. Preliminary analysis

During the conceptual phase of the project, one major requirement was set: the system must be able to process all data in real time. It was observed that speech recognition for a 64 000 word task was 1.8 times slower than real time on a 1.7 GHz AMD Athalon processor [14]. Additionally, the models for such a task are 3 times larger than the models used for the 1000-word command and control task on which our project is focused. Therefore, extending this linearly in terms of the number of compute cycles required, it can be said that a 1000-word task would take 1.6 times real time, or 160% longer than real time, to process at 1.7 GHz. Thus, a multi-GHz processor cannot handle a 1000-word task in real time, and custom hardware must be considered to help expedite the process. This certainly eliminates real-time speech

Table 1: Number of compute cycles for three different speech corpuses.

Speech corpus	No. of words	No. of gaussians	No. of evaluations per frame
TI digits	12	4816	192 600
RM1	1000	15 480	619 200
HUB-4	64 000	49 152	1 966 080

Table 2: Timing requirements for frame evaluation.

	AM	PE	WM	Total
No. of cyles [per 10 ms frame]	603 720	8192	102 400	714 312
Memory bandwidth [MB/sec]	495	—	5	—

processing from mobile phones and PDAs due to the far more limited capabilities of embedded processors.

In modern speech processing, incoming speech is sampled every 10 milliseconds. By assuming a frame latency of one for DSP processing, it can be said that a real-time hardware implementation must execute all operations within 10 milliseconds. To find our total budget a series of experiments were conducted on open-source SPHINX models [15, 16] to observe the cycle counts for different recognition tasks. Table 1 summarizes the results of these tests for three different sized tasks: digit recognition [TI Digits], command and control [RM1], and continuous speech [HUB-4].

The table shows the number of "compute cycles" required for the computation of all Gaussians for different tasks assuming a fully pipelined design. It can be seen that assuming one-cycle latency for memory accesses, the RM1 task would require 620 000 compute cycles, while HUB4 would require 2 million cycles. Knowing that we need to process all of the data within a 10- milliseconds window we observe that the minimum operating speeds for systems performing these tasks would be 62 MHz and 200 MHz, respectively.

Since the computation of Gaussian probabilities in AM constitutes the majority of the processing time, keeping some cushion for computations in the PHN and WRD blocks, it was determined that 1 million cycles would be sufficient to process data for every frame for RM1 task. Therefore a minimum operating speed of 100 MHz was set for our design. Having set the target frequency, a detailed analysis of the number of compute cycles was performed and is summarized in Table 2.

The number of cycles presented in this table is based on the assumption that all computations are completely pipelined. While a completely pipelined design is possible in the case of AM and PHN, computations in the WRD block do not share such luxury. This is a direct result of the variable branching characteristic of the word tree structure. Hence, to account for the loss in parallelism, the computation latency (estimated at a worst case of 10 cycles) has been accounted into the projected cycles required by the WRD block.

Further, the number of cycles required by the PE and WM blocks is completely dependent on the number of phones/words active at any given instant. Therefore, an analysis of the software was performed to obtain the maximum number of phones active at any given time instant. It was observed from SPHINX 3.3 for an RM1 dictionary, a maximum of 4000 phones were simultaneously active. Based on this analysis a worst case estimate of the number of cycles required for the computation is presented in the table.

3. ACOUSTIC MODELER

Acoustic modeling is the process of relating the data received from the FE, traditionally Cepstral coefficients and their derivatives, to statistical models found in the system database, which can account for 70% to 95% of the computational effort in modern HMM-based ASR systems [2, 13]. Each of the i senones, in the database are made up of c components, each one representing a d-dimensional multivariant Gaussian probability distribution. The components of a senone are log-added [17] to one another to obtain the probability of having observed the given senone. The equations necessary to derive a single senone score are shown in (1)–(6).

$$P(\overline{X}) = \frac{1}{\sqrt{(2\pi)^D \,|\, \overline{V}^* \,|}} e^{-\sum_{d=1}^{D} ((X_d - \mu_d)^2 / 2 * \sigma_d^2)} \tag{1}$$

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

Consider the first term on the left-hand side of (2). If the variance matrix V is constant, then the V^* term will also be constant, making the entire term a predefined constant K. Additionally, the denominator of the second term can be factored out and replaced with a new variable Ω_d that can be used to create a simplified version of the term Dist(X). Dist(X) becomes solely dependent on the d-dimensional input vector X. These simplifications are summarized in the three axioms below with a simplified version of (2) given as (3)

$$\begin{aligned}
&\text{let: } K = -0.5 \ln\left[(2\pi)^D \,|\, \overline{V}^* \,|\,\right],\\
&\text{let: } \text{Dist}(\overline{X}) = \sum_{d=1}^{D} \frac{\left(X_d - \mu_d\right)^2}{2 * \sigma_d^2} = \sum_{d=1}^{D} \left(X_d - \mu_d\right)^2 * \Omega_d,\\
&\text{let: } \Omega_d = \left(\frac{0.5}{\sigma_d^2}\right),\\
&\ln\left(P(\overline{X})\right) = K - \text{Dist}(\overline{X}).
\end{aligned} \tag{3}$$

Equation (3) serves to represent the calculations necessary to find a single multidimensional Gaussian distribution, or component. From here we must combine multiple components with an associated weighting factor to create senones as summarized in (4):

$$S_i(\overline{X}) = \sum_{c=1}^{C} \left[W_{i,c} * P_{i,c}(\overline{X})\right]. \tag{4}$$

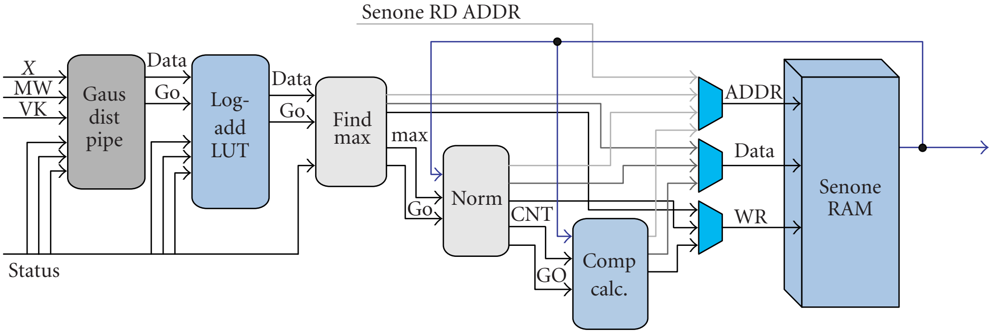


Figure 5: Block diagram of acoustic modeling pipeline.

At this point in our models it is necessary to define a log-base conversion factor, ψ, in order to stay in line with the SPHINX models used as our baseline. The use of a conversion factor in these equations is useful in transforming the $P_{i,c}(X)$ term of (4) into the required $\ln(P_{i,c}(X))$ term required for insertion of (3), but the use of the specific value is unique to the SPHINX system. By moving into the log-domain, the multiplication of (4) can also be transformed into an addition helping to further simplify the equations. The following axioms define the conversion factor with the result of its insertion shown in (5)–(6):

$$\text{let: } \psi = 1.0003,$$

$$\text{let: } f = \frac{1}{\ln(\psi)},$$

$$f * \ln\left[S_i(\overline{X})\right] = \log_{\psi}\left[S_i(\overline{X})\right], \tag{5}$$

$$\log_{\psi}\left[S_i(\overline{X})\right] = \log \sum_{c=1}^{C}\left[\log_{\psi}\left(W_{i,c}\right) + \log_{\psi}\left(P_{i,c}(\overline{X})\right)\right],$$

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

$$\log_{\psi}\left[S_i(\overline{X})\right] = \log \sum_{c=1}^{C}\left[W'_{i,c} + \log_{\psi}\left(P_{i,c}(\overline{X})\right)\right]. \tag{6}$$

The values μ, σ, V, K, and W relate to specific speech corpus being used and represent the mean, standard deviation, covariance matrix, scaling constant, and mixture weight, respectively. These values are stored in ROMs that are otherwise unassociated with the system and can be replaced or reprogrammed if a new speech corpus is desired. The f & Ψ values are log-base conversion factors ported directly out of the SPHINX 3 algorithm and the X vector contains the Cepstral coefficient input values provided by the FE block.

For our system we chose to use the 1000-word RM1 dictionary provided by the Linguistic Data Consortium [16], which utilizes 1935 senones, requiring over 2.5 million floating-point operations to calculate scores for every senone. For any practical system these calculations become the critical path and need to be done as efficiently as possible. By performing an in-depth analysis of these calculations, it was found that the computationally intensive floating-point Gaussian probability calculations could be replaced with fixed-point calculations while only introducing errors on the order of 10^{-4}. The ability to use fixed-point instead of floating-piont calculations allowed for the implementation of a pipelined acoustic modeling core running at over 100 MHz post place-and-route on a Virtex-4 SX35-10. Figure 5 illustrates the main components of the AM pipe.

Each of the stages in the pipeline sends a "go" signal to the following stage along with any data to be processed, allowing for the system to be stalled anywhere in the pipe without breaking. The first three stages also receive data from a status bus regarding the particular nature of the calculation being performed (i.e., is this the first, middle, or last element of a summation), which removes the need for any local FSM to control the pipeline.

3.1. Gaussian distance pipe

The Gaussian distance pipe is the heart of AM block and is responsible for calculating (1)–(3) for each senone in the database. This pipe must execute (1) over 620 000 times for each new frame of data and therefore must have the highest throughput of any component in the system. To accommodate this requirement while still trying to minimize the resources consumed by pipeline, the inputs to crucial arithmetic operations are multiplexed, allowing the inputs to the operation to be selected based on the bits of the status bus. The bits of the status bus, the *calc-bits*, provide information as to which element of the summation is being processed so that the output of the given stage can be routed properly to the next stage. Figure 6 shows a data-flow graph (DFG) for the order of operations inside the Gaussian distance pipe.

In order to help with low-power applications, the Gaussian distance pipe has a "pipe freeze" feature included, which is not shown in the DFG. If the last bit of the calculation is seen at the end of the pipe before a new first bit to be calculated has arrived, the pipe will completely shut down

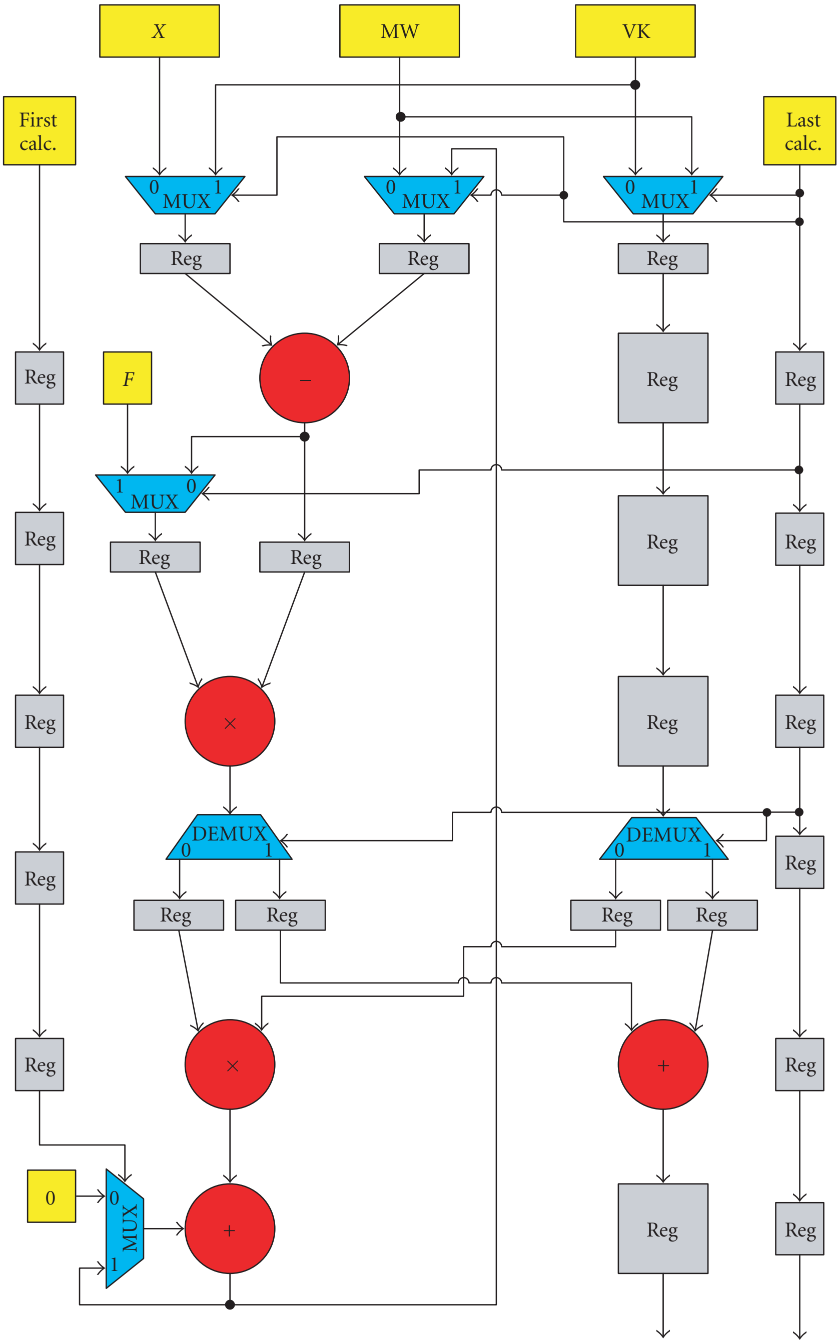


Figure 6: Data-flow graph for Gaussian distance pipe.

and wait for the presence of a new data. Internal to the pipe, each stage passes a valid bit to the successive stage that serves as a local stall, which will freeze the pipe until the values of the predecessor stage have become valid again.

Examining (2)–(4) reveals that to calculate a single component based on a d-dimensional Gaussian PDF actually requires $d + 1$ cycles, since the result of the summation across the d-dimensions must be subtracted from a constant and then scaled. As shown in Figure 6, the data necessary for the subtraction and scaling (K & W) can be interleaved into the data for the means and variances (M & V), leading to the need to read $d+1$ values from the ROM for each component in the system. This creates a constraint for feeding data into the pipe such that once the $d+1$ values have been read in, the system must wait for 1 clock cycle before feeding the data for the next component of the pipe. This necessity comes from the need to wait for the output of the final addition shown at the bottom of Figure 6. At the beginning of clock cycle $d + 1$, the K & W values are input into the pipe, but these values cannot be used until the summation of $DIST\ (X)$ is complete. This does not occur until clock cycle $d + 2$, resulting in the need to hold the input values to the pipe for one extra cycle.

Figure 6 further indicates that it takes seven clock cycles to traverse from one end of the pipe to the next. However, the next stage of the design, the log-add lookup table (LUT), described in Section 3.2, takes ten cycles to traverse. Therefore we must add three extra cycles to the Gaussian distance pipe to keep both stages in sync. To ensure that the additional cycles are not detrimental to the system, a series of experiments were conducted examining the effects of additional pipeline stages on the achieved $f_{\max}$ of the system. The results of these experiments, as well as the synthesis and post place-and-route results for this block are summarized in Section 4.

3.2. Log-add lookup

After completing the scoring for one component, that component is sent to the log-add LUT for evaluation of (4)–(6). This block is responsible for accumulating the partial senone scores and outputting them when the summation is complete. Equations (7)–(10) show the calculations necessary to perform the log-add of two components $P_{1,1}$ and $P_{1,2}$,

$$D = |P_{1,1} - P_{1,2}|, \tag{7}$$

$$R = P_{1,1} \longrightarrow \text{if} : P_{1,1} > P_{1,2}, \tag{8}$$

$$\begin{aligned} &\text{else: } R = P_{1,2}, \\ &\text{let: } \Psi = 1.0003, \\ &\text{let: } f = \frac{1}{\log(\psi)}, \end{aligned} \tag{9}$$

$$\text{RES} = R + 0.5 + f^* \left(\log\left(1 + \psi^{-D}\right)\right). \tag{10}$$

Due to the complexity of (10), it has been replaced by a LUT, where D serves as the address into the table. By using this table, (10) can be simplified to the result seen in (11),

$$\text{RES} = R + \text{LUT}(D). \tag{11}$$

While the use of a lookup to perform the bulk of the computation is a more efficient means of obtaining the desired result, it creates the need for a table with greater than 20 000 entries. In an effort to maximize the speed of the LUT, it was divided into smaller blocks and the process was pipelined over 2 clock cycles. The address is demultiplexed in the first cycle and the data is fetched and multiplexed onto the output bus during the second.

Equations (7)-(8) illustrate the operations necessary to find the address to this LUT. We chose to implement these operations as a three stage pipeline. The first stage of operation performs a subtraction of the two raw inputs and strips the sign bit from the output. In the second cycle the sign bit is used as a select signal to a series of multiplexers that assign the larger of the two inputs to the first input of the subtraction and the smaller to the second input of the summation. The third cycle of the pipe registers the larger value for use after the lookup and simultaneously subtracts the two values to obtain the address for the table. Similarly to the Gaussian distance pipe, the log-add LUT also has a pipe-freeze function built in. Figure 7 shows a detailed data-flow graph of the operations being performed inside the log-add lookup.

As mentioned in Section 3.1, the entire log-add calculation takes a minimum of 10 clock cycles to process a single input and return the partial summation for use by the next input. When this block is combined with the Gaussian distance pipe to form the main pipeline structure for the AM block the result is a 20 stage pipeline capable of operating at over 140 MHz, and requiring no local FSM for managing the traffic through the pipe, or possible stalls within the pipe.

3.3. Find Max/normalizer

Once a senone has been calculated, it must first pass through the find Max block before being written to the senone RAM. This block is a 2-cycle pipeline that compares the incoming data to the current best score and overwrites the current best when the incoming data is larger. Once the larger of the two values has been determined, the raw senone is output to the senone RAM. This is accompanied by a registered write signal ordinarily supplied by the log-add LUT. A data-flow graph for the find Max block is shown in Figure 8.

As mentioned in Section 3.2, the find Max unit only needs to operate once every 10 cycles, or whenever a new senone is available, therefore the values being fed to the compare are only updated when the senone valid bit is high. Aside from this local stall, the find Max unit has a similar pipe freeze function to conserve power.

When the last raw senone is put into the senone RAM, the "*MAX done*" signal in Figure 8 will be set high, signaling to the normalizer block that it can begin. During the process of normalization the raw senones are read sequentially out of the senone RAM and subtracted from the value seen at the "*Best Score*" output of the find Max block. The normalizer block consists of a simple 4-stage pipeline that first registers the input, then reads from the RAM, performs the normalization, and finally writes the value back to the RAM. The normalizer block also has pipe-freeze and local stall capabilities.

3.4. Composite senone calculation

In the RM1 speech corpus there are two different types of senones. The first type is "normal" or "base" senones, which are calculated via the processes described in Sections 3–3.3. The second type is a subset of the normal senones called composite senones. Composite senones are used to represent more difficult or easily confusable sounds, as well as nonverbal anomalies such as silence or coughing. Each composite senone is pointer to a group of normal senones, and for a given frame the composite senone takes the value of the best scoring normal senone in its group.

In terms of computation this equates to the evaluation of a series of short linked lists, where the elements of the list

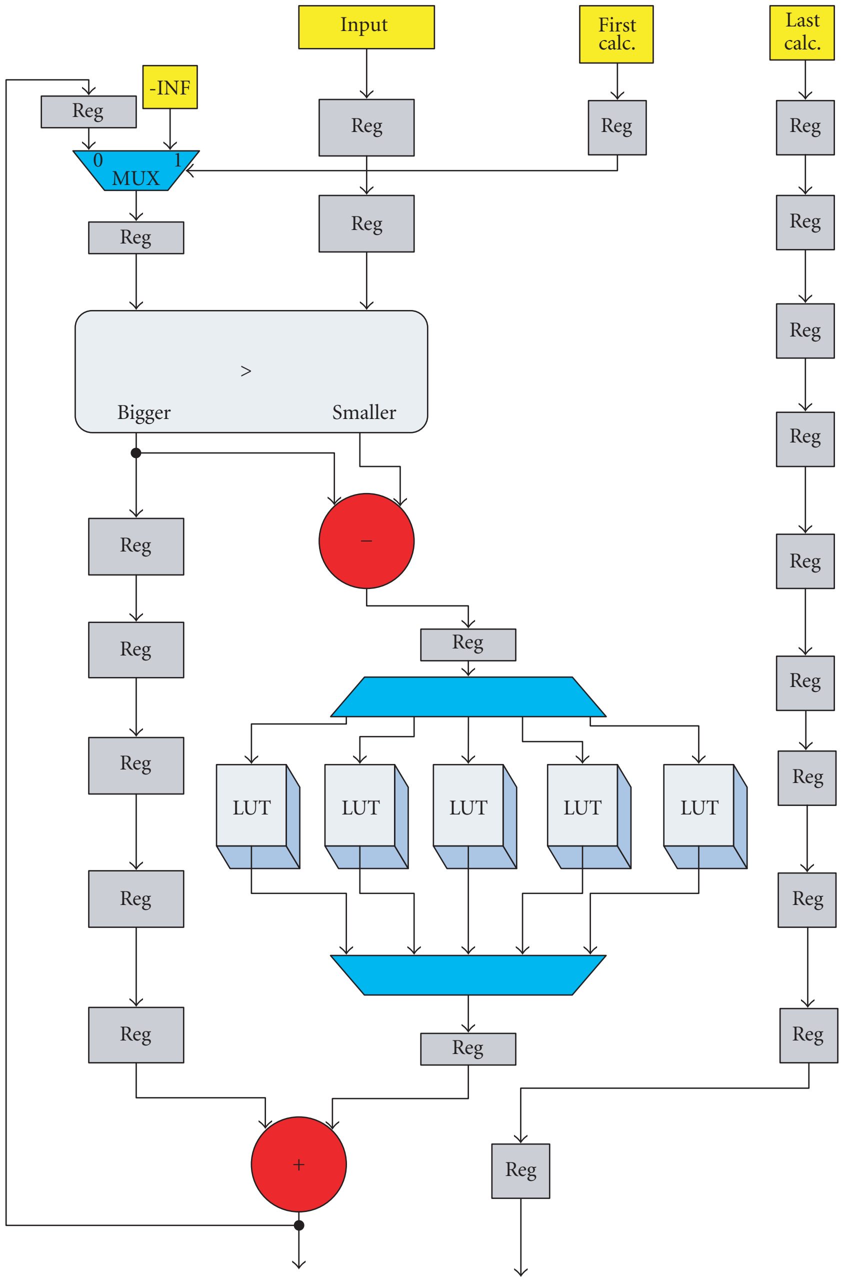


Figure 7: Data-flow graph for log-add LUT.

must be compared to find the greatest value. Once this greatest value is found it is written to a unique location in the senone RAM at some address above the address of the last normal senone. By writing this entry into its own location in the senone RAM instead of creating a pointer to its original location, the phoneme evaluation block is able to treat all senones equally, thus simplifying the control for that portion of the design.

The composite calculation works through the use of two separate internal ROMs to store the information needed for processing the linked-lists. The first ROM (*COUNT ROM*) contains the same number of entries as the number of composite senones in the system, and holds information about the number of elements in each composite's linked list. When a count is obtained from this ROM, it is added to a base address and used to address a second ROM (*ADDR ROM*) that contains the specific address in the senone RAM, where the normal senone resides.

Once the normal senone has been obtained from the senone RAM, it is passed through a short pipeline similar to

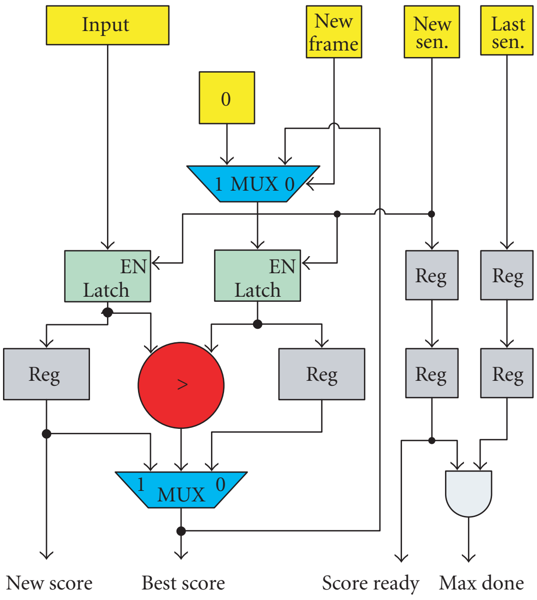


FIGURE 8: Data-flow graph for find Max unit.

the find MAX block except that only the best score is written back to the senone RAM. The count is then decremented and the process repeated until the count equals zero. At this point the next element of the count ROM is read and the process is repeated for the next composite senone. Once all elements of the count ROM have been read and processed, the block will assert a done signal indicating that all the senone scores for a given frame have been calculated. A DFG for the composite senone calculation is shown in Figure 9.

Like the other blocks of the AM calculation, the composite senone calculation has the built-in ability for locally stalling during execution and freezing completely when no new data is present at the input. This feature is more significant because composite senone calculations can only be performed after all of the normal senones have been completely processed. This results in a significant portion of the runtime where this block can be completely shut down leading to notable power savings. Specifically, it takes approximately 650 000 clock cycles to calculate all of the normal senones, during which the composite senone calculation block is active for only 2200 cycles.

In order to minimize the data access latency of later stages in the design, the senone RAM is replicated three times. When processing AM, the address and data lines of each of the RAMs are tied together so that one write command from the pipeline will place the output value in each of the RAMs during the same clock cycle. When the control of these RAMs is traded off to the phoneme evaluator (PE), the address lines are decoupled and driven independently by the three senone ID outputs from the PE. While this design choice does create a nominal area increase, the 3x improvement in latency is critical for achieving real-time performance.

4. PHONEME EVALUATOR

During phoneme evaluation, the senone scores calculated in the AM are used as state probabilities within a set of HMMs. Each HMM in the database represents one context-dependent phone or phoneme. In most English speech corpuses, a set of 40–50 base phones is used to represent the phonetic units of speech. These base phones are then used to create context-dependent phones called mono-, bi-, or triphones based on the number of neighbors that have influence on the original base phone. In order to stay close to the SPHINX 3 system, we chose to use a triphone set from the RM1 speech corpus represented by 3-state Bakis-topology HMMs. Figure 10 shows an example Bakis HMM with all states and transitions labeled for later discussion.

The state shown at the end of the HMM represents a null state called the exit state. While this exit state has no probability associated with it, it does have a probability for entering it. It is this probability that defines the cost of transitioning from one HMM to another. One of the main advantages of HMMs for speech recognition is the ability to model time-varying phenomena. Since each state has a self transition as well as a forward transition, it is possible to remain inside an HMM for a very large amount of time or conversely, to exit an HMM in as little as four cycles, visiting each state only once. To illustrate this principle, Figure 11 maps a hypothetical path through an HMM on a two-dimensional trellis.

By orienting the HMM along the Y-axis and placing time on the X-axis, Figure 11 shows all possible paths through an HMM with the hypothetical best path shown as the darkened line through the trellis. In our HMM decoder we chose to use the Viterbi algorithm to help minimize the amount of data needed to be recorded during calculation. The Viterbi algorithm states that if, at any point in the trellis, two paths converge, only the best path need be kept and the other discarded. This optimization also is widely used in speech recognition systems, including SPHINX 3 [18].

For each new set of senones, all possible states of an active HMM must be evaluated to determine the actual probability of the HMM for the given inputs. The operations necessary to calculate these values are described in (12)–(15),

$$\begin{gathered}\left.\begin{array}{l}\mathrm{H}_3(t-1)+\mathrm{T}_{22} \\ \mathrm{H}_2(t-1)\mathrm{T}_{12}\end{array}\right\rangle \textcircled{$>$} + S_2(t) = \mathrm{H}_3(t), \\ \left.\begin{array}{l}\mathrm{H}_2(t-1)+\mathrm{T}_{11} \\ \mathrm{H}_1(t-1)+\mathrm{T}_{01}\end{array}\right\rangle \textcircled{$>$} + S_1(t) = \mathrm{H}_2(t),\end{gathered} \tag{12}$$

$$\left.\begin{array}{l}\mathrm{H}_1(t-1)+\mathrm{T}_{00} \\ \mathrm{H}_0\end{array}\right\rangle \textcircled{$>$} + S_0(t) = \mathrm{H}_1(t), \tag{13}$$

$$\mathrm{H}_{\mathrm{BEST}}(t) = \mathrm{MAX}\left\{\mathrm{H}_1(t), \mathrm{H}_2(t), \mathrm{H}_3(t)\right\} \tag{14}$$

$$\mathrm{H}_{\mathrm{EXIT}}(t) = \mathrm{H}_2 + \mathrm{T}_{2e}. \tag{15}$$

Equations (12)-(13) show that the probability of an HMM being in a given state at a particular time is dependent not only on that state's previous score and associated transition penalties, but also on the current score of its associated senone. This relationship helps to enhance the accuracy of the model when detecting time-varying input patterns.

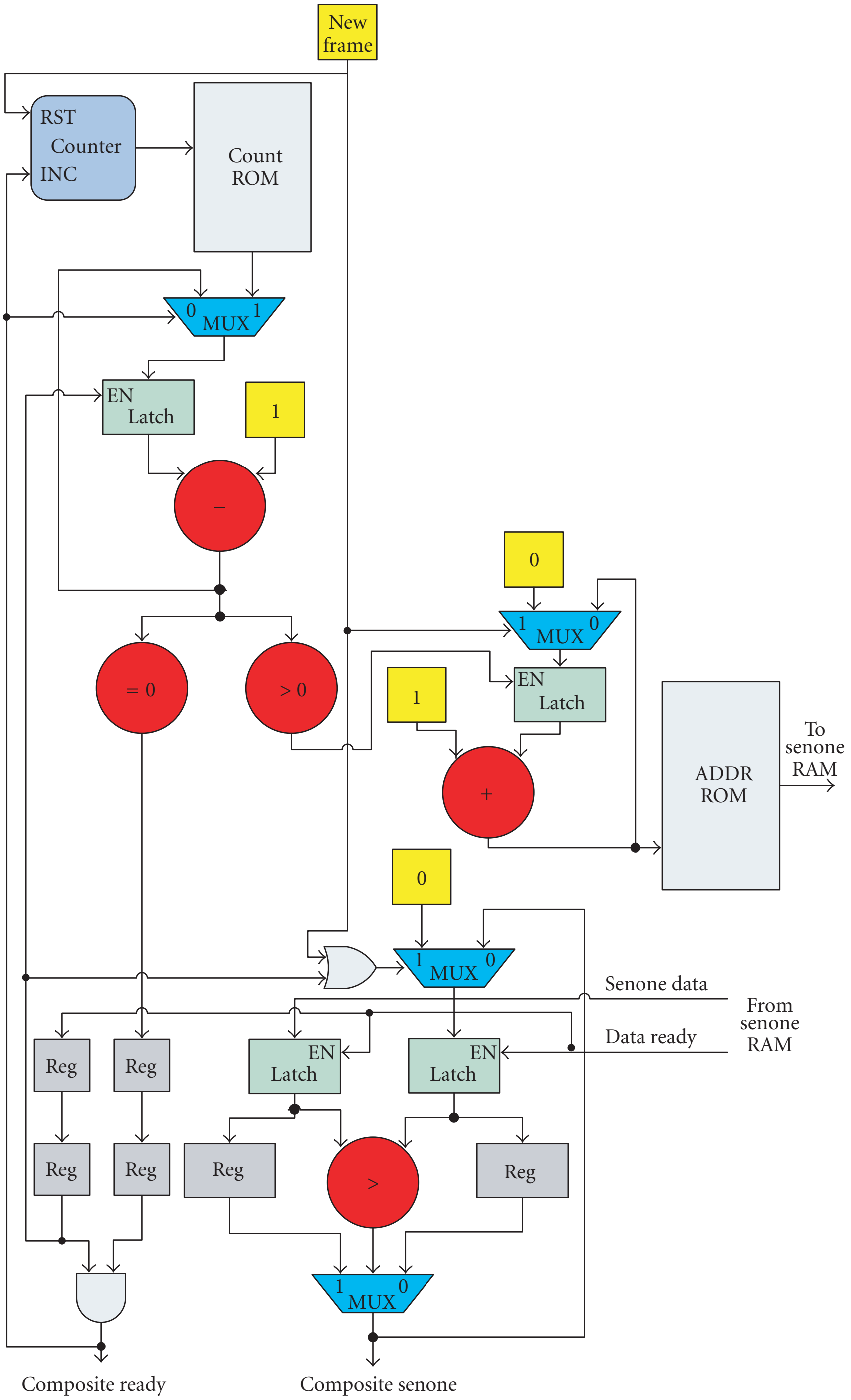


Figure 9: Data-flow graph for composite senone calculation.

Looking specifically at (13) it can be seen that the H_0 input is not time-dependent. While H_0 is not completely constant, it does not change with every new time tick, and therefore is not considered strictly time-dependent. The specific reasons for this functionality relate to the way HMMs are activated and deactivated in the system and are described in more detail in Section 5. However, it should be noted that this value only changes on the transition from inactive to active. Equations (12)-(13) show the insertion of the Viterbi algorithm in that only the best of the possible transitions will be held in the value use in the next time.

It is also relevant to note that, while we may have a very large number of bi- or tri-phones in the database, we only have as many unique sets of transition matrices as we have

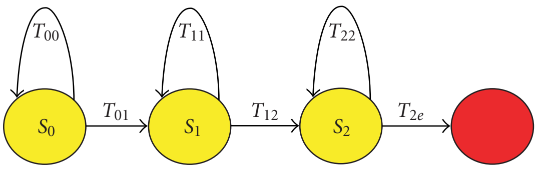


Figure 10: Sample HMM topology.

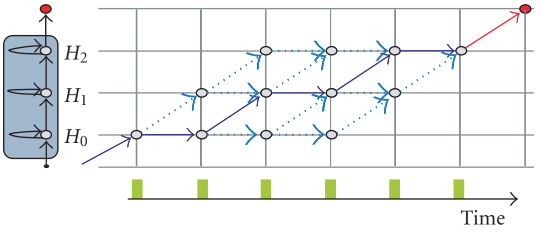


Figure 11: Sample HMM trellis.

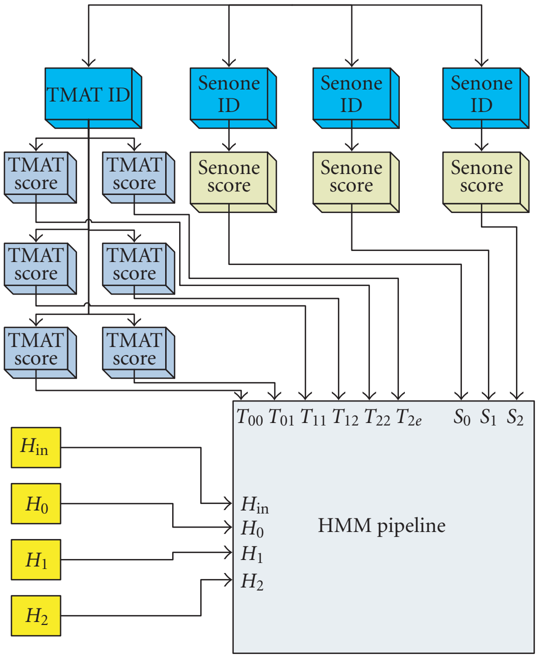


Figure 12: Data hierarchy for HMM calculation.

base phones. When a new HMM is to be processed, its ID is put into a decoder that outputs the ID of the transition matrix to be used for the evaluation. This transition score ID, or TMAT ID, is then used to address the ROMs that contain the actual scores associated with the arcs of Figure 10. For each evaluation of an HMM the best of all state scores as well as the HMM exit score must be calculated to facilitate the pruning of the HMM in the next stage of the PE.

Once the H_{BEST} and H_{EXIT} values have been found for all the active HMMs, a beam pruning algorithm is applied to the set to help mitigate the amount of active data in the system. During this process a constant offset or beam is added to the best H_{BEST} and H_{EXIT} values. Only values with scores above this beam remain active for the next stage. As the HMMs are being pruned, a token with the HMMs unique ID is written to a status FIFO based on the result of pruning. During the word modeling portion of the SR process, these FIFOs will be read and the data in the shared RAM will be processed accordingly.

When implementing the PE in hardware, the majority of the logic necessary resides in the large amount of constant data that must be stored and retrieved in order to process an HMM. Figure 12 illustrates the data structure that must be traversed to acquire all of the necessary data to process a single HMM.

In the structure shown in Figure 12 the HMM ID input is used to address 4 separate LUTs, one for the transition score ROMs, and one for each of the senones needed for the HMM. As mentioned previously, the TMAT ID ROM serves as a decoder to map one of a large set of HMM IDs to one of the relatively small set of TMAT IDs. This single TMAT ID is then used to address six TMAT score ROMs in parallel in order to decrease the latency of the data access. The other three LUTs receive the same HMM ID, but are used to decode the appropriate senone IDs needed for the HMM. Again in an effort to increase parallelism and decrease latency, these senone IDs are found in parallel and output from the block back to the shared senone RAM described in Section 3. Once the TMAT scores, current senone scores, and previous state scores have been retrieved, the actual processing of the HMM begins. As previously stated, the remainder of the calculation can be implemented as a high throughput pipeline and is described in detail in the following sections.

4.1. HMM control logic

While developing our architecture, it was necessary to consider the fact that unlike AM, the amount of work that needs to be done at any one time is variable and therefore some control must be included to monitor the amount of active data in the system. As illustrated in Figures 3 and 4 the data to be processed by the PE is most efficiently managed by a series of FIFOs containing lists of active HMM IDs. Specifically, data entering PE is provided via either the new phoneme active list (nPAL) FIFO, and data exiting PE is written to either the exit (VALID) FIFO, the inactive (DEAD) FIFO, or the phoneme active list (PAL) FIFO. Figure 13 illustrates the relationships between these FIFOs.

The first observation made when looking at Figure 13 is that the PAL FIFO is actually completely internal to the PE block and can be loaded by either the HMM pipeline or the pruner. In order for this to work properly a special end of phase (EOP) token was created to serve as a place marker in the FIFO. To create the EOP token the PAL FIFO was designed to be 1 bit wider than the other FIFOs so that the extra

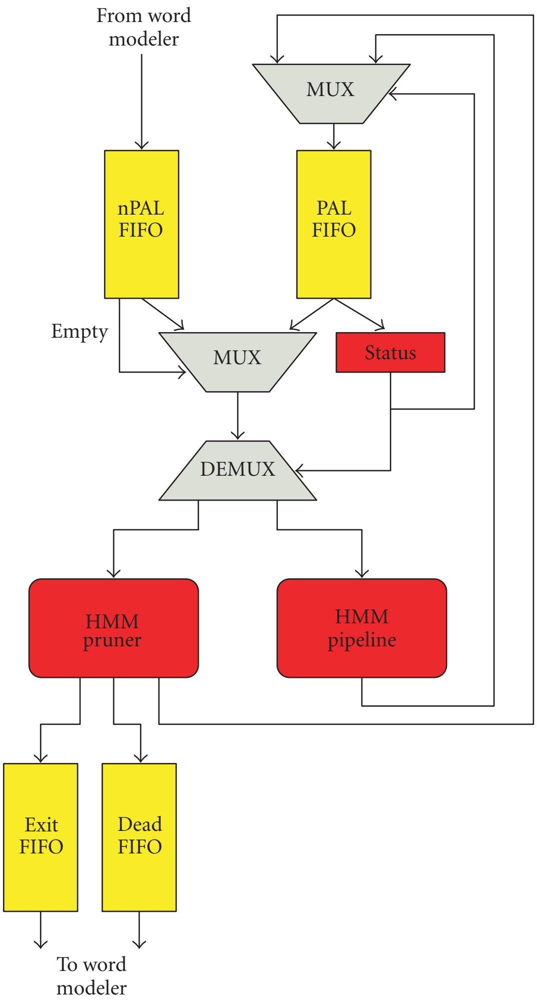


Figure 13: Control for the phoneme evaluator.

bit could be used to hold information about the EOP token. All standard tokens in the queue are appended with a “0” for this extra bit, while the EOP token is appended with a “1,” making it very easy to detect the presence of the EOP token. At the beginning of each new frame PE will start pulling tokens from the nPAL FIFO until the FIFO is completely empty. When the FIFO is emptied, or in the case there was nothing there to start, the PAL FIFO is then read until and EOP token is detected by the STATUS block. When the EOP token is seen it is then known that all HMMs requiring processing have been completed and pruning may commence.

At the beginning of pruning the EOP token is written back to the PAL FIFO. Pruning continues until the EOP token is popped back out of the FIFO. Upon this second observation, all data has been processed for that frame and the word modeler may begin processing the tokens in the DEAD and EXIT FIFOs.

4.2. HMM pipeline

The execution of (12)–(15) constitutes the majority of the calculations necessary to perform phoneme evaluation and therefore a significant amount of time was put into examining the optimal way to perform these operations. After establishing the control for this pipeline the calculations were examined and it was found that to find all H values for a given HMM a simple ADD-COMPARE-ADD-COMPARE pipeline can be constructed as shown in the DFG in Figure 14.

The DFG in Figure 14 highlights the regularity of the structure for the pipeline and leads directly to a high-throughput low-latency design for calculation of the HMM scores in the system. Further, the complexity of the pipeline is actually quite small and requires a noticeably smaller amount of logic than even the ROMs required to drive it. As each of the active HMMs are evaluated, the five output values are written to a shared RAM called the phone-pointer RAM (PH RAM) and the HMM ID token is written into the pruner queue. Results on synthesis for this block are summarized in Section 6 of this document.

4.3. HMM pruner

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

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

5. WORD MODELER

Word modeling can be broken down into two major steps: resetting of newly inactive tokens and updating of currently active tokens. Given that the functionality of these two components is distinctly different, two separate components were designed, one for each task. The token deactivator reads data from the DEAD FIFO and resets the scores in the PH RAM. Simultaneously, the token activator reads from the EXIT FIFO and processes the word tree to determine which new tokens need to be placed in the nPAL FIFO. The creation of

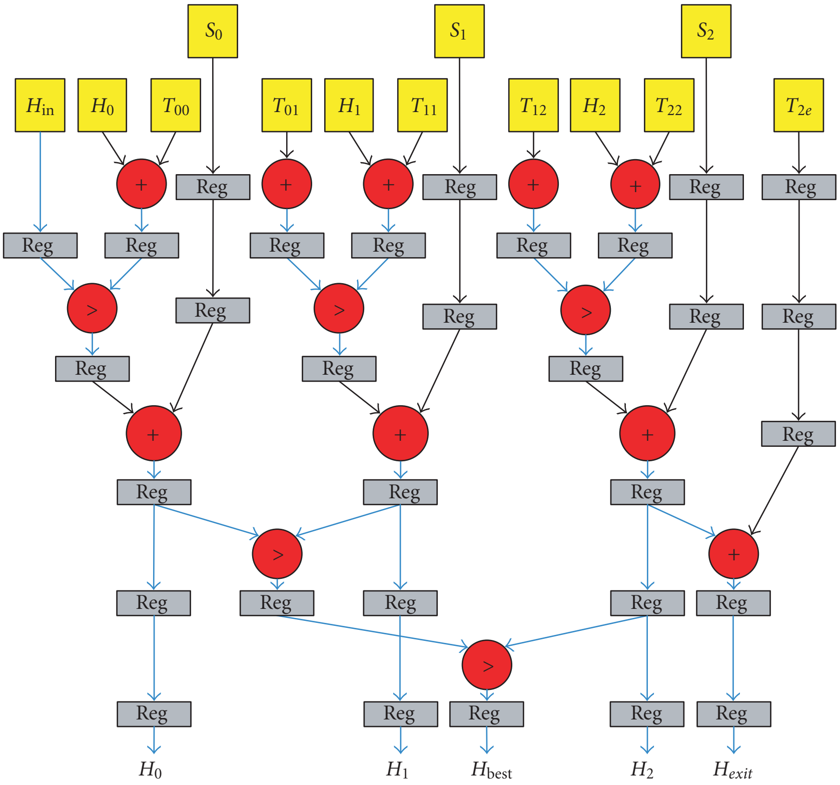


FIGURE 14: Data-flow graph for HMM pipeline.

two separate blocks also minimizes the amount of logic active at any one point in time leading to power savings crucial to ensuring sustainability on a mobile platform. The operations of these blocks are described in detail in Sections 5.1 and 5.2, respectively.

5.1. *Token deactivator*

After the PE block has pruned the active HMMs and placed the appropriate tokens in the DEAD FIFO the word modeler can begin the task of resetting the PH RAM entries corresponding to these tokens. This process is simplified by the fact that all PH RAM values need to be reset to the exact same value. This means that to deactivate a token the token must be popped from the DEAD FIFO and used to address the PH RAM. When this address is applied to the RAM the reset constant is then written to the RAM and the process is repeated until the FIFO is empty. This process can be performed in a simple-two stage, POP-WRITE pipeline, and is capable of running at close to the f_{MAX} of the target device.

5.2. *Token activator*

The token activator portion of the word modeler is noticeably more complicated than deactivator portion. When an HMM is found to have a valid exit score, the word modeler must determine the location of that HMM in the word tree and which HMMs are tied to its exit state. As shown in Figure 1, a word tree can have a large number of branches stemming from one root. Mapping these types of structures into hardware is not obvious. Another unique problem in token activation is that while a given HMM may be used multiple times in multiple different word trees such as the *"CH"* sound at the end of *pouch* and *couch*, these two sounds must be represented by completely unique events. This means that while a given dictionary may not need all possible phonemes in a language, it will most likely need multiple instances of some of the phones.

Thus, it was necessary to determine a way of indexing specific nodes in the search space such that their information could remain in a unique location in the PH RAM. To do this, the entire word search space was mapped and each of the nodes was given a unique ID. An example of this process is shown in Figure 15.

Based on this mapping scheme, even though nodes 12–14 in Figure 15 all relate to the *AE* phoneme, they all have unique IDs and will be treated separately in search algorithm. The mappings determined in the process relate directly to the HMM IDs stored in the tokens passed between the PE and WM blocks, and define the core of the token passing algorithm as implemented in our system.

Having established our mapping scheme it was necessary to implement a tree structure in hardware. Unlike the previous section of the design, this portion is less arithmetically

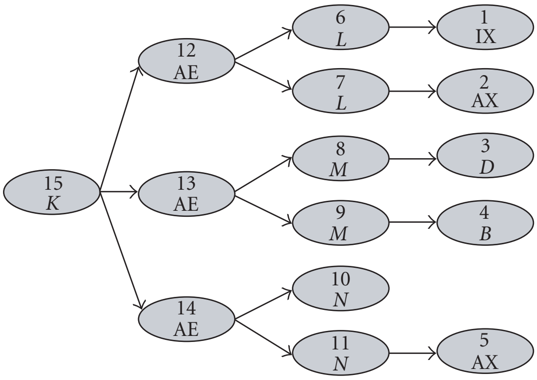


FIGURE 15: Sample search space node mapping.

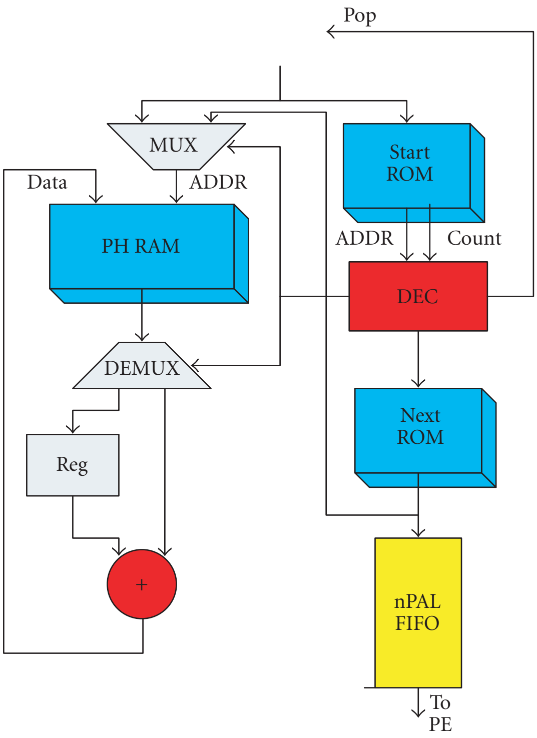


FIGURE 16: Data-flow graph for token activator.

intensive and involves searching instead of computational overheads. One of our immediate observations when looking at the data structures was that each node in the search space can be thought of as a short linked list. Evaluating linked lists in hardware is a well-defined process, so to create our tree we created a linked list of linked lists.

During each evaluation a token must be read from the EXIT FIFO and its linked list retrieved. Further the H_{EXIT} score from the exiting HMM must be added to a word penalty and propagated to the H_0 score of the HMMs in the linked list. Since the word penalty will be different for each HMM in the linked list, it becomes necessary to process the HMMs one at a time until the end of the list is reached. To efficiently keep track of the linked list, two ROMs were designated: the START ROM and the NEXT ROM. The START ROM is directly addressed by the token in the EXIT FIFO and contains both a starting address in the NEXT ROM for the linked list and a count of how many values are in the list. The NEXT ROM holds all of the HMM IDs necessary to process the linked lists. Figure 16 shows the DFG for the token activator.

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

6. SYNTHESIS AND PLACE-AND-ROUTE

While performing the synthesis and place-and-route operations on our hardware architecture implemented in VHDL, two distinctly different approaches were taken in an effort to observe the effect of design tools on overall system design. The extremely computationally heavy AM block was written using VDHL syntax specific to the Synplify synthesis engine in an effort to focus the bulk of the work load on the tools as opposed to the designers. This approach allows for the generation greatly simplified VHDL which in turn makes the processes of debugging and optimization significantly easier. Additionally, the Synplify synthesis tool provides built-in features to perform pipelining and retiming of the target VHDL.

As a second design strategy we implemented the PE and the WM blocks in traditional VHDL and synthesized using precision synthesis from Mentor Graphics. Precision synthesis tends to leave more optimizations up to the designers, and knowing that the implementation of the PE and the WM are significantly more intuitive than the implementation for AM, we chose to leave the bulk of the design optimization to ourselves. To compare the overall quality of the design as well as the time for development, limited portions of PE were also written for Synplify so a one to one comparison of results could be obtained.

6.1. *Acoustic modeling results*

To examine the effects of pipelining and retiming in Synplify, a series of experiments was conducted on the Gaussian distance pipe to find the optimal number of pipeline stages. To do this, extra registers were put into the design and the pipelining and retiming options were enabled in the synthesis tool. When the synthesis was executed, the tool was able to move these registers to what it determined were the optimal locations in the design, minimizing the amount of analysis done by the designer. Our primary target in these

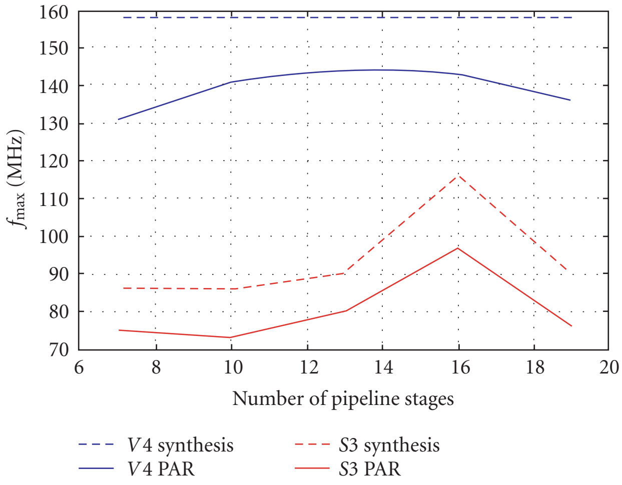


Figure 17: Analysis of $f_{\max}$ versus pipeline stages of the gaussian distance pipe for virtex-4 SX and spartan-3 FPGAs.

experiments was the Xilinx Virtex-4 SX35 FPGA due to its high performance and large number of embedded DSP (DSP48) and RAM (BRAM) cells. For sake of comparison we also targeted a smaller device, the Xilinx Spartan-3, a 90 nm FPGA with embedded 18 × 18 multipliers. The graph in Figure 17 shows the results of these experiments for a pipeline between 7 and 19 stages deep, with the dotted lines representing the projected $f_{\max}$ of the system from the synthesis engine and the solid lines representing the post place-and-route $f_{\max}$.

While the $f_{\max}$ obtained for the Virtex-4 device is noticeably higher than the $f_{\max}$ of the Spartan-3 devices, it is also observed that increasing the number of pipeline stages improves the speed of the Spartan-3 device more significantly than the speed of the Virtex-4 device. It is also observed that for both devices there are an optimum number of stages beyond which the performance of the device actually degrades due to the increased amount of area consumed by the design.

Another interesting result of these experiments was that regardless of the number of pipeline stages the projected synthesis speed for the Virtex-4 did not change. This implies that even when the pipeline is configured with the minimum number of allowable stages, the results of the pipelining and retiming processes are the same. The post place-and-route timing results for the Virtex-4, however, do change with the number of pipeline stages implemented. Since we know we are utilizing the embedded DSP slices on the chip and we can trace the critical path of the circuit we can conclude that the physical distance between two individual DSP cells is great enough that adding extra registers along the path will in fact increase the speed of the design. Figure 16 further shows that when targeting the Virtex-4, a 10-stage pipe will provide an acceptable operating frequency for the system with only minor improvements being gained with each additional pipe stage beyond 10. This is a promising result because we know that the depth of the Log-Add LUT is 10 cycles as well, allowing us to match the depths of the two pipelines without having to sacrifice a considerable amount of speed.

In addition to the experiment described above all individual components of the AM block were synthesized and routed on the chip to fully characterize their performance. Table 3 summarizes the results of these tests and makes note of any special cells used by each stage of the design. Figure 18 shows the FPGA layout for the AM block.

Table 3: Summary of synthesis and place-and-route results for Virtex-4 SX35.

Component	Synthesis (MHz)	PAR (MHz)	Area
Gaussian dist. pipe	157	145	6 DSP48s 423 slices
Log-add LUT	164	150	13 BRAMs 307 slices
Find max	181	160	90 slices
Normalizer	197	172	155 slices
Composite senone calc.	197	140	2 BRAMs 147 slices
AM block (total)	164	125	6 DSP48s, 30 BRAMs 1527 slices

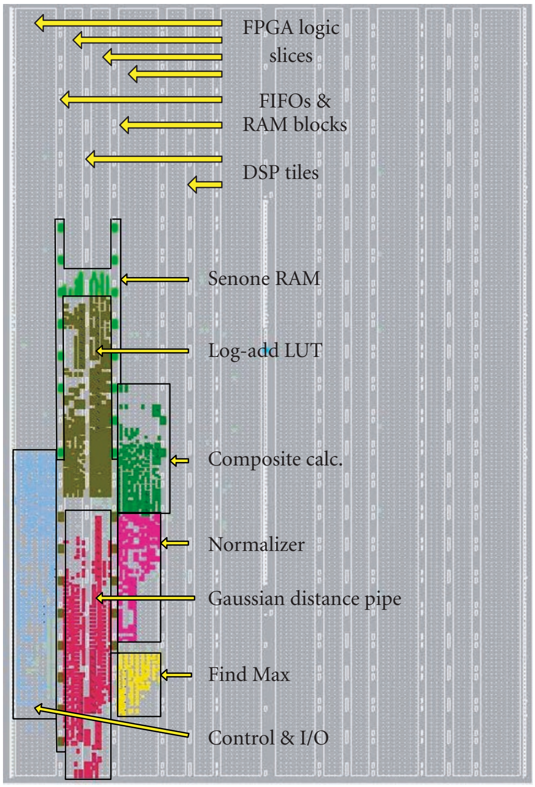


Figure 18: Floor plan for AM pipeline.

Table 4: Power analysis for multiple target FPGA devices.

Power (mW)	Virtex II Pro XC2VP100	Virtex 4 XCE4VSX35	Spartan 3 XC3S1500
Dynamic	163.46	35.46	41.75
Static	571.88	395.50	178.00
Total	735.34	430.97	219.76

Table 5: Summary of post-place-and-route results for the phoneme evaluator.

Component	f_{MAX} (MHz)	Slices	Flip-flops	LUTs	MISC
HMM pipeline	189	1475	2393	593	24 BRAMs
Pruner	115	48	24	77	—
PE core	115	1713	2645	861	24 BRAMs
PE control	213	8	14	13	—
PE block (total)	117	1941	2983	1050	25 BRAMs

Further, since AM consumes over 90% of the run-time, it was relevant to analyze the power consumption of this block to get a feel for the overall consumption of the device. To do this, post-place-and-route simulations were performed in ModelSim using data generated in Xilinx Xpower and the results of our experiments are shown in Table 4 for Virtex-4, Virtex-2pro, and Spartan-3 devices all running at $f_{\max}$ for the specific device. The stimulus for these experiments was based on randomly generated inputs to the system.

6.2. Phoneme evaluator results

Given that the design for the PE was noticeably less complex than the design for the AM, we chose to implement all optimizations by hand and derive our own custom models for all necessary components as opposed to allowing the tools to do this. We used FPGAdvantage GUI to derive the code and precision synthesis to do our placement analysis. Our post-place-and-route results are summarized in Table 5.

6.3. Word modeler results

Observing that word modeling took less than 5% of the overall execution effort, little time was spent analyzing the synthesis results for this block. As with the PE block we opted for custom derivation of VHDL and synthesis via precision synthesis. The results of the place-and-route operations are summarized in Table 6.

Table 6: Summary of the word modeler synthesis results.

Component	f_{MAX} (MHz)	Slices	Flip- flops	LUTs	MISC
Token deactivate	319	51	48	88	—
Token activate	145	174	66	320	2 BRAMs
WM block (total)	142	357	131	638	2 BRAMs

6.4. Hardware development summary

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

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

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

During the final portion of the design, word modeling, a tree-search algorithm was designed in hardware. The hardware was designed as a large linked list evaluation unit capable of propagating information throughout the tree while also deactivating nodes in the tree and connecting multiple

Table 7: Summary of hardware performance results.

Component	f_{MAX} synthesis/PAR (MHz)	Area	Cycle count
Gaussian distance pipeline	157/145	6 DSP tiles, 411 slices	10 cycle/Gauss
Log-add lookup	164/150	13 BRAMs, 307 slices	10 cycle/comp
AM block total	164/125	6 DSP tiles, 30 BRAMS, 1328 slices	162 cycle/senone 640 K cycle total
Hidden Markov model pipeline	261/140	775 slices	8 cycle/load 5 cycle/calc
Pruner	277/177	112 slices	4 cycle/HMM
PE block total	115/111	84 BRAMs, 1866 slices	22 cycle/HMM
Token deactivator	377/170	54 Slices	2 cycle/dead HMM
Token activator	184/120	3 BRAMs, 160 slices	10*branch cycle/active HMM
WM block total	166/129	3 BRAMs, 414 slices	22 cycle/dead HMM + 10*branch cycle/active HMM

trees for the creation of word strings. The deactivation portion of the hardware is capable of running at 170 MHz post-place-and-route but the activation logic can only operate at 120 MHz, limiting the overall performance of the word modeler but not impacting the overall performance of the system.

The majority of the verification for the design was done through post-place-and-route simulations models and comparing their results to the results obtained in the MATLAB environment for the SPHINX 3 models. Knowing that our MATLAB model provided a one-to-one representation of the SPHINX system, we were confident that if the MATLAB results were identical to the hardware results, we had correctly implemented the algorithms. Table 7 summarizes the performance results for the major portions of the hardware development.

7. CONCLUSIONS

In this work we have shown the ability to implement high-performance acoustic modeling pipeline on an FPGA device. Further we designed a unique architecture for a design capable of performing critical operations in the speech recognition process in real time with minimized power consumption and maximum processing bandwidth. Our system also highlights an architecture built to be driven completely by the data leading to a system that can be reprogrammed for multiple applications and dictionaries by altering the input to the system. This research has proven the effectiveness of the proposed design methodology and helped further the development of portable low-power speech recognition systems.

REFERENCES

[1] K. K. Agaram, S. W. Keckler, and D. Burger, "Characterizing the SPHINX speech recognition system," Tech. Rep. TR2001-18, Department of Computer Sciences, University of Texas at Austin, Austin, Tex, USA, January 2001.

[2] C. Lai, S.-L. Lu, and Q. Zhao, "Performance analysis of speech recognition software," in *Proceedings of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads*, Cambridge, Mass, USA, February 2002.

[3] M. Ravishankar, R. Singh, B. Raj, and R. Stern, "The 1999 CMU 10x real time broadcast news transcription system," in *Proceedings of DARPA Workshop on Automatic Transcription of Broadcast News*, Washington, DC, USA, May 2000.

[4] L. Rabiner and B. H. Juang, *Fundamentals of Speech Recognition*, Prentice Hall Signal Processing Series, Prentice Hall, Englewood Cliffs, NJ, USA, 1993.

[5] X. Huang, A. Acero, and H. Hon, *Spoken Language Processing*, Prentice Hall, Englewood Cliffs, NJ, USA, 2001.

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

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

[8] T. S. Anantharaman and R. Bisiani, "A hardware accelerator for speech recognition algorithms," in *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA '86)*, pp. 216–223, Tokyo, Japan, June 1986.

[9] S. Nedevschi, R. K. Patra, and E. A. Brewer, "Hardware speech recognition for user interfaces in low cost, low power devices," in *Proceedings of Design Automation Conference (DAC '05)*, pp. 684–689, Anaheim, Calif, USA, June 2005.

[10] P. Placeway, S. Chen, M. Eskenazi, et al., "The 1996 Hub-4 Sphinx-3 system," in *Proceedings of the DARPA Speech Recognition Workshop*, pp. 85–89, Chantilly, Va, USA, February 1997.

[11] R. Hoare, K. Gupta, and J. Schuster, "Speech silicon: a data-driven SoC for performing hidden Markov model based speech recognition," in *Proceedings of High Performance Embedded Computing Workshop (HPEC '05)*, MIT, Lexington, Mass, USA, September 2005.

[12] R. Hoare, et al., "A hardware based acoustic modeling pipeline for hidden Markov model based speech recognition," in *Proceedings of 13th Reconfigurable Architectures Workshop (RAW '06)*, Rhodes Island, Greece, April 2006.

[13] J. Nouza, "Feature selection methods for hidden Markov model-based speech recognition," in *Proceedings of the 13th International Conference on Pattern Recognition*, vol. 2, pp. 186–190, Vienna, Austria, August 1996.

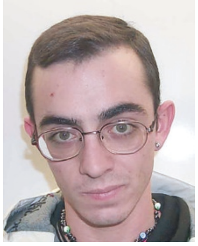
[14] B. Mathew, A. Davis, and Z. Fang, "A low-power accelerator for the SPHINX 3 speech recognition system," in *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '03)*, pp. 210–219, San Jose, Calif, USA, November 2003.

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

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

[17] X. Li and J. Bilmes, "Feature pruning in likelihood evaluation of HMM-based speech recognition," in *Proceedings of IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU '03)*, pp. 303–308, St. Thomas, Virgin Islands, USA, November-December 2003.

[18] M. Ravishankar, *Efficient algorithms for speech recognition*, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, Pa, USA, May 1996, CMU-CS-96-143.



Jeffrey Schuster received his B.S.E.E from the University of Pittsburgh's Department of Electrical Engineering in 2004 focusing his work on signal processing, specifically for audio and speech implementations. During his studies he began work for Drs. Raymond Hoare and Amro El-Jaroudi designing parallel speech processing systems based on the SPHINX 3 engine from Carnegie Mellon University. Through this work funding was obtained from the Tech Collaborative, formerly the Pittsburgh Digital Greenhouse, to complete his M.S.E.E. developing his work into hardware based solutions to the speech recognition problem. The work completed at the University of Pittsburgh let to the derivation of numerous papers presented at conferences such as the HPEC 2005 event at MIT and the RAW 2006 event in Rhodes, Greece. After completion of his M.S.E.E. he accepted a position with the hardware engineering division of Exegy Inc. located in St. Louis, MO working to design the next generation of high speed data search appliances. He currently resides in the St. Louis area where his free time is spent applying his love of engineering and computers to his passion for music.

Kshitij Gupta received his M.S. degree in electrical engineering from the University of Pittsburgh, Pittsburgh (USA) in 2005. With primary focus on FPGA/system-on-chip design, he has over three years of research experience in analyzing, optimizing, and implementing algorithms employed in speech recognition systems onto hardware platforms, towards speech-on-silicon. Prior to this, he received his B.E. degree in electronics & communication engineering from Osmania University, Hyderabad (India) in 2002. His current interests include design, implementation & verification of custom hardware, and multiprocessor SoC-based designs using FPGAs, ASICs, and configurable processors for high-performance computing and embedded consumer applications.

Raymond Hoare received his B.E. degree from Steven's Institute of Technology in 1991. He received his M.S. degree from the University of Maryland in 1994. He received his Ph.D. degree from Purdue University in 1999. He is currently the President and Founder of Concurrent Design Automation in Pittsburgh, Pennsylvania. Previously, he was an Assistant Professor of electrical and computer engineering at the University of Pittsburgh in Pittsburgh, Pennsylvania. His research interests include high-performance parallel architectures, communication networks, systems-on-a-chip, and design automation.

Alex K. Jones received his B.S. degree in 1998 in physics from the College of William and Mary in Williamsburg, Virginia. He received his M.S. and Ph.D. degrees in 2000 and 2002, respectively, in electrical and computer engineering from Northwestern University. He is currently an Assistant Professor of Electrical and Computer Engineering and Computer Science at the University of Pittsburgh, Pennsylvania. He was formerly a Research Associate in the Center for Parallel and Distributed Computing and Instructor of electrical and computer engineering at Northwestern University. He is a Walter P. Murphy Fellow of Northwestern University, a distinction he was awarded twice. His research interests include compilation techniques for behavioral and low-power synthesis, embedded systems, radio frequency identification (RFID), and high-performance computing. He is the author of over 40 publications in these areas.