Second Prize

SOPC-Based Word Recognition System

Institution: National Institute Of Technology, Trichy

Participants: S. Venugopal, B. Murugan, S.V. Mohanasundaram

Instructor: Dr. B. Venkataramani

Design Introduction

Real-world processes generally produce observable outputs, which can be characterized as signals. The signals can be discrete in nature (e.g., characters from a finite alphabet, quantized vectors from a codebook), or continuous in nature (e.g., speech samples, temperature measurements, music). The signal source can be stationary (i.e., its statistical properties do not vary with time), or non-stationary (i.e., the signal properties vary over time). The signals can be pure (i.e., coming strictly from a single source), or can be corrupted from other signal sources (e.g., noise) or by transmission distortions, reverberation, etc.

A problem of fundamental interest is characterizing such real-world signals in terms of signal models. There are several reasons to be interested in applying signal models. First, a signal model can provide the basis for a theoretical description of a signal processing system, which processes the signal to provide a desired output.

A second reason why signal models are important is that they are potentially capable of letting us learn a great deal about the signal source (that is, the real-world process that produced the signal) without having the source available. This property is especially important when the cost of getting signals from the actual source is high. In this case, with a good signal model, we can simulate the source and learn as much as possible via simulations.

Finally, signal models are important because they often work extremely well in practice, and enable us to realize important practical systems (such as prediction systems, recognition systems, identification systems, etc.) in a very efficient manner. There are several possible choices for the type of signal model used for characterizing the properties of a given signal. Broadly, one can dichotomize the types of signal models into the class of deterministic models and the class of statistical models.

Deterministic models generally exploit some known specific properties of the signal (e.g., that the signal is a sine wave or a sum of exponentials). In these cases, specification of the signal model is generally

straightforward; all that is required is to determine (estimate) the values of the signal model parameters (e.g., amplitude, frequency, phase of a sine wave, amplitudes and rates of exponentials, etc.).

The second broad class of signal models is the set of statistical models in which one tries to characterize only the statistical properties of the signal. The underlying assumption of the statistical model is that the signal is characterized as a parametric random process, and that the parameters of the stochastic process can be determined (estimated) in a precise, well-defined manner. The Hidden Markov Model (HMM) falls in this second category.

In simple terms, a HMM is a model used to create another model about which we know nothing except its output sequence. The HMM is trained to produce an output that closely matches the available output sequence, and can be assumed to model the unknown model with sufficient accuracy.

Speech recognition systems have been developed for many real-world applications, often using low-cost speech recognition software. However high-performance and robust isolated word recognition, particularly for digits, is still useful for many applications, such as recognizing telephone numbers for use by physically challenged persons. This formed the motivation for taking up this project.

Efficient implementation of a complete system on a programmable chip (SOPC) got an impetus with the advent of high-density FPGAs integrated with high-capacity RAMs and the availability of implementation support for soft-core processors such as the Nios® II processor. FPGAs enable the best of both worlds to be used gainfully for an application—the microcontroller or RISC processor is efficient for performing control and decision-making operations, while the FPGA efficiently performs digital signal processing (DSP) operations and other computation intensive tasks.

We aim to produce an efficient hardware speech recognition system with an FPGA acting as a coprocessor that is capable of performing recognition at a much faster rate than software. Implementation of systems using an Altera®-based system on a programmable chip enables time-critical functions to be implemented on hardware synthesized with VHDL/Verilog HDL code. The soft-core Nios II processor that is part of the FPGA can execute the programs, written in a high-level language. Custom instructions enable the feasibility of implementing the whole system on an FPGA with better partitioning of the software and hardware implementation of the speech recognition system.

Our project aims at developing a HMM-based speech recognition system with a vocabulary of 10 digits (digits zero to nine). We trained the system for three users for all the mentioned digits with a recognition accuracy of nearly 100%. Energy and zero crossings-based voice activity detection (VAD) was used for segmentation of the input samples and removing background noise. We used linear predictive coding (LPC-10) analysis, followed by cepstral analysis for feature vector extraction from speech frames. HMM was used for training the speech models and Viterbi decoding for recognition. Vector quantization (VQ) was used for reducing the memory requirement. We used direct parameter averaging of the HMM parameters during training, which has several advantages over Rabiner's approach, such as a lower data requirement, higher detection accuracy, and less computation complexity.

We implemented the feature extraction, training, and other preprocessing stages of HMM in software (C++/MATLAB) in the offline mode and the recognition process, including floating-point multiplication operation of the Viterbi decoding process, as custom hardware in hardware implementation and as an online process.

Functional Description

The aim is to design a system that will recognize an uttered digit from the recorded speech samples (recorded as .wav files and converted to text files using MATLAB). The digits have to be from a

predetermined vocabulary set for which the system is trained. The design can be split into software and hardware partitioning to exploit the facilities present in the Nios II processor. The overall design can be divided into two functional parts: training and recognition.

Training involves iteratively fine-tuning the parameters of the HMMs with digits from the given vocabulary set until it converges. The sequence of steps in training are:

- Preprocessing
- Codebook generation
- Generating models for individual digits using a combination of the Forward algorithm, the Backward algorithm, and the Baum-Welch algorithm.

Recognition involves testing the given digit with each of the available digit HMMs, finding the model that gives the maximum probability, and concluding that the result corresponds with the digit uttered. The sequence of steps in recognition are:

- Preprocessing
- Finding the best fitting model using maximum likelihood algorithm, the Viterbi decoding algorithm

Preprocessing

Preprocessing involves the following steps:

1. *Recording*—Record the speech at a sampling frequency of 8 KHz with 16-bit quantization.

2. *VAD*—Using the endpoint detection algorithm, the starting and ending points are found. The speech is sliced into frames 450 samples in length. The energy and number of zero crossings of each frame is found. A threshold energy and zero crossing value is determined based on the computed values, and only frames crossing the threshold are considered. This removes most of the unnecessary background noise. A small vestige of frames beyond the starting and ending frames are included so as not to leave the starting and ending parts of the speech that do not cross the threshold but that may prove important in recognition.

3. *Pre-emphasis*—The digitized speech signal s(n) is put through a low-order LPF to spectrally flatten the signal and to make it less susceptible to finite precision effects later in the signal processing. The filter is represented by $H(z)=1-az^{-1}$ where we have chosen the value of "a" as 0.9375.

4. *Frame blocking*—Speech frames are formed with durations of 56.25 ms (N = 450 sample length) and with an overlap of 18.75 ms (M=150 sample length) between adjacent frames. The overlapping is performed so that the resulting LPC spectral estimates correlate from frame to frame and are quite smooth.

   $x_q(n)=s(Mq+n)$ n=0 to N-1; q=0 to L-1 where L is the number of frames.

5. *Windowing*—Each frame with a Hamming window is windowed to minimize signal discontinuities at the beginning and end of the frames.

   $x'_q(n)=x_q(n).\ w(n)$

Where

$w(n)=0.54=0.46 \cos(2\ n/N-1)$

6. *Autocorrelation analysis*—Perform autocorrelation analysis for each frame and find P+1 (P=10) autocorrelation coefficients.

7. $$r_q(m)=\sum_{n=0}^{N-1-m} x'_q(n)x'_q(n+m) \qquad m=0,1,\ldots..P$$

8. The zeroth autocorrelation coefficient is the frame's energy, which was previously used for VAD.

9. Perform LPC analysis by employing Levinson and Durbin's algorithm to convert the autocorrelation coefficients to LPC parameter set.

$$E^{(0)}=r_q(0)$$

$$K_i=\{\ r_q(i)-\{\ \sum_{j=1}^{L-1}\ \alpha_j^{(i-1)}.\ r_q(|i=j|)\ \}\ \}/E^{(i-1)} \qquad 1\le i\le P$$

$$\alpha_i^{(i)}=k_i$$

$$\alpha_j^{(i)}=\alpha_j^{(i-1)}-k_i.\ \alpha_{(i-j)}^{(i-1)}$$

$$E^{(i)}=(1-k_i^2)\ E^{(i-1)}$$

Where α_m^{10} $1\le m\le P$ are the LPC coefficients.

10. LPC parameters to cepstral coefficient conversion: The cepstrum coefficients are a more robust and reliable feature set than the LPC coefficients.

$c_0=\ln \sigma^2$ where σ is the gain of the LPC model.

$$c_m=\alpha_m+\sum_{k=1}^{m-1}\ (k/m)\ .\ c_k.\ \alpha_{m-k} \qquad 1\le m\le P$$

$$c_m=\sum_{k=1}^{m-1}\ (k/m)\ .\ c_k.\ \alpha_{m-k} \qquad P<m\le Q$$

11. *Parameter weighing*—Sensitivity of the lower order cepstral coefficients to overall slope and the higher order coefficients to noise has necessitated weighing of the cepstral coefficients by a tapered window to minimize these sensitivities. We have used weighing by a band pass filter of the form

    $w_m=[1+ (Q/2). \sin (m/Q)] \quad 1 \leq m \leq Q$

12. *Temporal cepstral derivative*—Temporal cepstral derivatives are an improved feature vector for forming the speech frames. They can be used with the cepstral derivative in case the cepstral coefficients do not give acceptable recognition accuracy.

Vector Quantization

A codebook of size 128 is obtained by the VQ of the weighted cepstral coefficients of all reference digits, by all users. The advantages of VQ are:

- Reduced storage for spectral analysis information
- Reduced computation for determining similarity of spectral analysis vectors.
- Discrete representation of speech sounds. By associating phonetic label(s) with each codebook vector, the process of choosing a best codebook vector to represent a given spectral vector becomes equivalent to assigning a phonetic label to each spectral frame of speech. This makes the recognition process more efficient.

One obvious disadvantage of VQ is the reduced resolution in recognition. Assigning a codebook index to an input speech vector amounts to quantizing it. This results in quantization error, which increases with decrease in codebook size.

There are two commonly used algorithms, the K-Means algorithm and the Binary Split algorithm. The K-Means algorithm describes the way in which a set of L training vectors can be clustered into M (<L) codebook vectors with:

1. *Initialization*—Arbitrarily choose M vectors as the initial set of code words in the codebook.
2. *Nearest neighbor search*—For each training vector, find the code word in the current codebook that is closest and assign that vector to the corresponding cell.
3. *Centroid update*—Update the codeword in each cell using the centroid of the training vectors assigned to that cell.
4. *Iteration*—Repeat the above two steps until the average distance falls below a preset threshold.

The Binary Split algorithm is a more efficient method than the K-Means algorithm because it builds the codebook in stages. First, it designs a 1-vector codebook, then uses a splitting technique on the code words to initialize the search for a 2-vector codebook, and then continues the splitting process until the desired M-vector codebook is received.

1. Design a 1-vector codebook, which is the centroid of the entire training set and hence needs no iteration.
2. Double the size of the codebook by splitting each current codebook yn according to the rule

    $y_n^+ = y_n(1+e)$

$y_n^- = y_n(1-e)$

where n varies from 1 to the codebook size and e is the splitting parameter.

3. Use the K-Means iterative algorithm to obtain the best set of centroids for the split codebook.

4. Iterate the above two steps until the required codebook size is received.

Hidden Markov Model

Recognition is achieved by maximizing the probability of the linguistic string, W, given the acoustic evidence, A. For example, choose the linguistic sequence W such that

$$P(W' \mid A) = \max_{W} P(W \mid A)$$

An HMM is characterized by the following:

- N, the number of states in the model. Although the states are hidden, for many practical applications there is often some physical significance attached to the states or to sets of states of the model. The states are interconnected in such a way that any state can be reached from any other state (e.g., an ergodic model); however other possible interconnections of states are often used by restricting the transitions. We denote the individual states as $S = \{S_1, S_2, \ldots S_N\}$, and the state at time t as q_t.

- M, the number of distinct observation symbols per state, i.e., the discrete alphabet size. The observation symbols correspond to the physical output of the system being modeled. We denote the individual symbols as $V=\{V_1, V_2, \ldots V_M\}$

- The state transition probability distribution $A = \{a_{ij}\}$ where $a_{ij}=P[q_{t+1}=S_j \mid q_t=S_i] \quad 1\le i,j\le N$

- The observation symbol probability distribution in state j, $B = \{bj(k)\}$,

  where $b_j(k) = P[\, v_k \text{ at } t \mid q_t = S_j \,] \quad 1\le j\le N, \; 1\le k\le M$

- The initial state distribution $\quad = \{ \ _i\}$ where $\ _i=P[q_1=S_i] \;\; 1\le i\le N$

Given appropriate values of N, M, A, B, and , the HMM can be used as a generator to give an observation sequence $O=O_1 O_2 O_3 \ldots O_T$ (where each observation O_t, is one of the symbols from V, and T is the number of observations in the sequence).

Three Basic Problems for HMM

Given the form of HMM, there are three basic problems of interest that must be solved for the model to be useful in real-world applications. These problems are as follows:

- *Problem 1, the scoring problem*—Given the observation sequence $O = \{ O_1, O_2, \ldots O_T \}$ and a model $\lambda = (A, B, \pi)$, how to efficiently compute P(O/A), the probability of the observation sequence, given the model? The algorithm normally used to solve this is the Forward-Backward algorithm

- *Problem 2, the matching problem*—Given the observation sequence $O = \{ O_1, O_2, \ldots O_T \}$, and the model $\boldsymbol{\lambda}$, how to choose a corresponding state sequence $Q = q_1 q_2 \ldots q_T$ which is optimal in

some meaningful sense (i.e., best "explains" the observations)? The algorithm normally used to solve this is the Viterbi algorithm

- *Problem 3, the training problem*—How to adjust the model parameters $\lambda = (A, B, \pi)$ to maximize P(O/A)? The algorithm normally used to solve this is the Baum-Welch Re-Estimation Procedures.

Solution to Problem 1

One of the most straightforward and highly inefficient methods to solve this problem is to enumerate all the possible state sequences of length **T** and finding the sum over all such state sequences to find the required probability. The forward-backward is a more efficient algorithm, which can be explained as follows:

Consider the forward variable $\alpha_t(i)$ defined as

$$\alpha_t(i)=P[O_1, O_2, O_3, O_t, q_t=S_i \mid \lambda]$$

That is, the probability of the partial observation sequence, $O_1, O_2 \ldots O_t$ and state S_i at time t, given the model λ. We can solve for $\alpha_t(i)$ inductively, as follows:

1. Initialization:

   $$\alpha_1(i)= \pi_i . b_i(O_1) \quad 1\leq i\leq N$$

2. Induction:

   $$\alpha_{t+1}(j)=\left[\sum_{i=1}^{N} \alpha_t(i).a_{ij} \right] b_j(O_{t+1}) \quad 1\leq t\leq T-1;\ 1\leq j\leq N$$

3. Termination:

   $$P(O \mid \lambda)= \sum_{i=1}^{N} \alpha_T(i)$$

In a similar manner we can solve for the backward variable $\beta_t(i)$ iteratively as follows:

1. Initialization:

   $$\beta_T(i)= 1 \quad 1\leq i\leq N$$

2. Induction:

$$\beta_t(i) = \sum_{j=1}^{N} \beta_{t+1}(j).a_{ij}.\ b_j(O_{t+1}) \quad T-1 \geq t \geq 1;\ 1 \leq i \leq N$$

Solution to Problem 2

Unlike Problem 1, for which an exact solution can be given, there are several possible ways of solving Problem 2, namely finding the "optimal" state sequence associated with the given observation sequence. The difficulty lies with the definition of the optimal state sequence; i.e., there are several possible optimality criteria. To implement this solution to Problem 2, we define the variable

$$\gamma_t(i) = P(q_t = S_i \mid O, \lambda)$$

$$\gamma_t(i) = \alpha_t(i).\ \beta_t(i) / P(O \mid \lambda)$$

$$= \alpha_t(i).\ \beta_t(i) / \sum_{i=1}^{N} \alpha_t(i).\ \beta_t(i)$$

Because $\alpha_t(i)$ accounts for the partial observation sequence $O_1, O_2 \ldots O_t$, and state S_i at t , while $\beta_t(i)$ accounts for the remainder of the observation sequence $O_{t+1}, O_{t+2}, \ldots O_T$ given state S_i at t.

Viterbi Algorithm

To find the single best state sequence, $Q = \{q_1, q_2, \ldots q_T\}$, for the given observation sequence $O = (O_1, O_2, \ldots O_T\}$, we need to define the quantity

$$\delta_t(i) = \max_{q_1, q_2, \ldots q_t} P[q_1, q_2, \ldots q_t = i, O_1, O_2 \ldots O_t / \lambda]$$

That is, $\delta_t(i)$ is the best score (highest probability) along a single path, at time t, which accounts for the first t observations and ends in state S_i. By induction we have

$$\delta_{t+1}(j) = [\max \delta_t(i)\ a_{ij}\ ] .\ b_j(O_{t+1}).$$

To retrieve the state sequence, we need to keep track of the argument that maximized, for each t and j. We do this via the array $\psi_t(j)$. The complete procedure for finding the best state sequence can now be stated as follows:

1. Initialization:

$$\delta_1(i) = \pi_i\, b_i(O_1),\ 1 \leq i \leq N$$

$$\psi_1(i) = 0$$

2. Recursion:

$$\delta_t(j) = \max [\delta_{t-1}(i)a_{ij}] \, b_j(O_t) , \quad 2 \le t \le T ; \quad 1 \le j \le N$$

$$\psi_t(j) = \text{argmax} \, [\delta_{t-1}(i)a_{ij}], \quad 2 \le t \le T ; 1 \le i \le N ; \quad 1 \le j \le N$$

3. Termination:

$$P^* = \max \, [\delta_t(i)]$$

$$1 \le i \le N$$

$$q_T^* = \text{argmax} \, [\delta_t(i)]$$

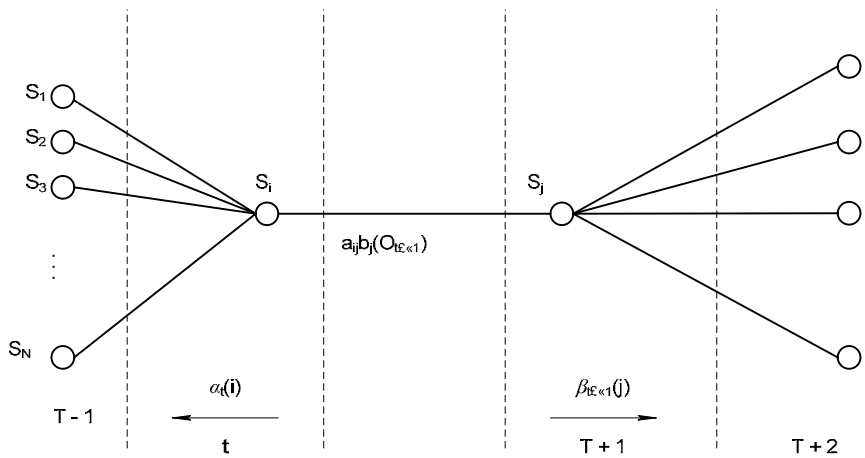
$$1 \le i \le N$$

4. Path (state sequence) backtracking:

$$q_t^* = \psi_{t+1}(q_{t+1}^*), \; t = T - 1, T - 2, .. 1.$$

The lattice (or trellis) structure given in Figure 1 efficiently implements the computation of the Viterbi procedure.

Figure 1. Lattice Structure



Solution to Problem 3

The third, and by far the most difficult, problem of HMMs is to determine a method to adjust the model parameters (A, B, π) to maximize the probability of the observation sequence given the model. There is no known way to analytically solve for the model, which maximizes the probability of the observation sequence. In fact, given any finite observation sequence as training data, there is no optimal way of estimating the model parameters. We can, however, choose $\lambda = (A, B, \pi)$ such that $P(O \mid \lambda)$ is locally maximized using an iterative procedure such as the Baum-Welch method.

To describe the procedure for re-estimation (iterative update and improvement) of HMM parameters, we first define $\xi_t(i,j)$, the probability of being in state S_i at time t, and state S_j, at time t+1, given the model and the observation sequence,

$\xi_t(i,j) = P(q_t=S_i, q_{t+1}=Sj \mid O, \lambda)$.

We can write $\xi_t(i,j)$ in the form

$$\xi_t(i,j) = P(q_t=S_i, q_{t+1}=S_j, O \mid \lambda) / P(O \mid \lambda)$$

$$= \alpha_t(i)\, a_{ij}\, b_i(O_{t+1})\, \beta_{t+1}(j) / P(O \mid \lambda)$$

$$= \alpha_t(i)\, a_{ij}\, b_i(O_{t+1})\, \beta_{t+1}(j) \Big/ \left[\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_t(i)\, a_{ij}\, b_i(O_{t+1})\, \beta_{t+1}(j) \right]$$

We have previously defined $\gamma_t(i)$ as the probability of being in state S_i at time t, given the observation sequence and the model; hence we can relate $\gamma_t(i)$ to $\xi_t(i,j)$ by summing over j, giving

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j)$$

If we sum $\gamma_t(i)$ over the time index t, we get a quantity that can be interpreted as the expected (over time) number of times that state S_i is visited, or equivalently, the expected number of transitions made from state S_i (if we exclude the time slot t = T from the summation). Similarly, summation of $\xi_t(i,j)$ over t (from t = 1 to t = T - 1) can be interpreted as the expected number of transitions from state S_i to state S_j. That is,

$$\sum_{t=1}^{T-1} \gamma_t(i) = \text{expected number of transitions from } S_i$$

$$\sum_{t=1}^{T-1} \xi_t(i,j) = \text{expected number of transitions from } S_i \text{ to } S_j \text{ in O.}$$

Using the above formulas (and the concept of counting event occurrences), the method for re-estimation of the parameters of an HMM is as follows:

Π'_j = expected frequency (number of times) in state S_i at time (t = 1) = $\gamma_1(i)$

a'_{ij} = expected number of transitions from state S_i to state S_j / expected number of transitions from state Si

$$= \sum_{t=1}^{T-1} \xi_t(i,j) \Big/ \sum_{t=1}^{T-1} \gamma_t(i)$$

$b'_j(k)$= expected number of times in state S_j and observing symbol v_k / expected number of times in state S_j

$$= \sum_{t=1}^{T} \gamma_t(i) \Big/ \sum_{t=1}^{T} \gamma_t(i)$$

such that Ot=vk

Based on the above procedure, we iteratively use λ' in place of λ and repeat the re-estimation calculation, to improve the probability of O being observed from the model until some limiting point is reached. The result of this re-estimation procedure is a maximum likelihood estimate of the HMM. We use a terminating condition of λ' not varying by more than a certain fraction (say 10%) from λ.

Training

Training the HMM for each digit in the vocabulary is done using the Baum-Welch algorithm. The codebook index will be the observation vector for the HMM.

Recognition

Recognition of the uttered digit is found by employing the maximum likelihood estimate such as Viterbi decoding algorithm. This implies the model with the maximum probability will be the uttered digit.

Performance Parameters

This section describes the performance parameters for the project.

Recognition Accuracy

100% recognition accuracy for a three–user dependent system implemented with input from the trained vocabulary alone.

Design Implementation Times

Total design run time for recognition:

- 96.53 s (full software implementation)
- 93.936 s (with floating-point multiplication operation of the Viterbi decoding process block implemented as custom block and the rest as software)

Implementation time for the recognition process (excluding preprocessing step) and finding the probability for the input digit for all models using the Viterbi block in software: 5.48 s.

Implementation time for a floating-point multiplication operation of the Viterbi decoding process block implemented as a custom block: 0.5 ms (Refer to Snapshot 4 in the Appendix). The Viterbi block uses 4810 such multiplications, so the estimated run time of the whole Viterbi process using the custom block is 2.405 sec.

Implementation time for a floating-point multiplication operation of the Viterbi decoding process block implemented as software: 5.7 ms (Refer to Snapshot 5 in the Appendix).

Speed-up factor achieved by using custom block: 11.4 times.

Design Metrics

Memory requirement—M4Ks (Refer to Snapshot 1 in the Appendix):

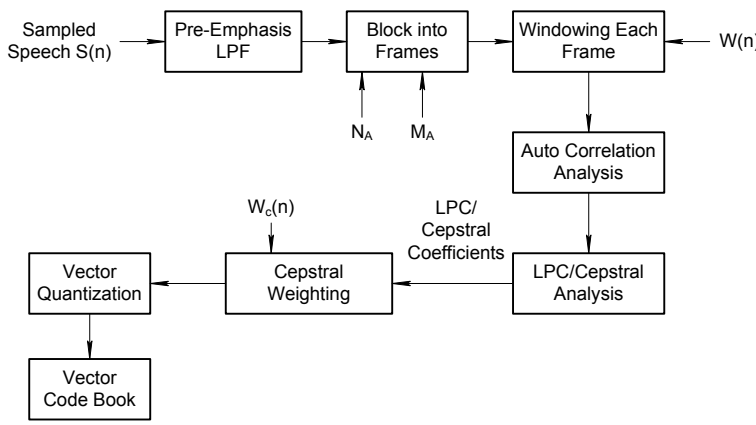
- Whole design: 9 out of 20
- Custom block alone: 2 out of 9

Logic area used: 4,233/5,980 logic elements (LEs).

Design Architecture

Figure 2 shows the software block diagram.

Figure 2. Software Design Block Diagram of Front-End Feature Analysis for HMM



LPC Feature Analysis

Observation vectors are obtained from speech samples by performing VQ of LPC coefficients. The overall system is a block processing model in which a frame of N_A samples is processed and a vector for each frame is computed.

The steps in the processing are as follows:

1. *Pre-Emphasis*—A first-order digital network processes the digitized speech signal to spectrally flatten the signal.

2. *Blocking into Frames*—Sections of N_A consecutive speech samples are used as a single frame with the overlap between adjacent frames being M_A.

3. *Frame Windowing*—Each frame is multiplied by an N_A sample hamming window w(n) to minimize the adverse effects of chopping an N_A-sample section out of the running speech signal.

4. *Autocorrelation Analysis*—Each windowed frame of speech samples is autocorrelated with a lag of 10.

5. *LPC/Cepstral Analysis*—The LPC coefficients (Q) of order 10 are computed from the autocorrelation coefficients using Durbin's recursion method, and LPC derived cepstral vectors are computed.

6. *Cepstral Weighting*—The Q-coefficient cepstral vector $C_l(m)$ at time frame '*l*' is weighted by a window $W_C(m)$ of the form.

   $W_C(m)= 1+ (Q/2) \sin(\Pi m/Q), \quad 1\leq m \leq Q$

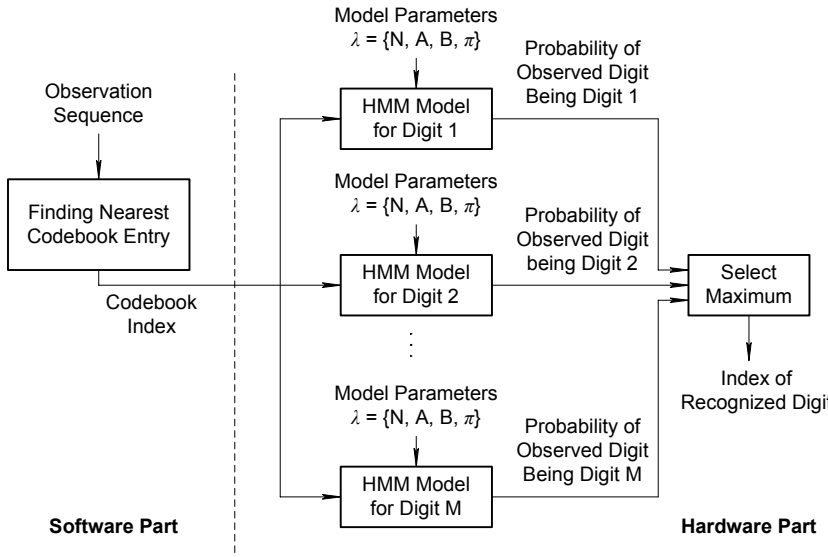
   to yield the weighted cepstral coefficient as

   $C'_l(m)=C_l(m) \cdot W_C(m)$

7. *VQ*—For a HMM with discrete observation symbol density, VQ is required to map each continuous observation vector into a discrete codebook index. Once the codebook of vectors has been obtained, the mapping between continuous vectors and codebook indices becomes a simple nearest neighbor computation, that is, the continuous vector is assigned the index of the nearest (in a spectral distance sense) codebook vector. Thus, the major issue in VQ is the design of an appropriate codebook for quantization. We have taken codebook size as 128.

Figure 3 shows the hardware design block diagram.

Figure 3. Hardware Design Block Diagram



The Hidden Markov Model (HMM)

Recognition is achieved by maximizing the probability of the digit W**,** given the acoustic evidence, A, that is, choose the digit W such that

$P(\hat{W}|A)=\max_{w} P(W|A)$

Elements of a Discrete Hidden Markov Model

N is the number of states in the model

M is the number of distinct observation symbols per state. We denote the individual symbols as

$$V = \{v_1, v_2, \cdots, v_M\}.$$

A **=** $\{a_{ij}\}$, the state transition probability distribution where

$$a_{ij} = P[q_{t+1} = S_j | q_t = S_i], \quad 1 \le i, j \le N.$$

q_t is the state of the HMM at time t

B is the observation symbol probability distribution in state j, B = $\{b_j(k)\}$, where

$$b_j(k) = P[v_k \text{ at } t | q_t = S_j], \quad 1 \le j \le N \quad 1 \le k \le M.$$

π is the initial state distribution $\pi = \{ \pi_i \}$ where

$$\pi_i = P[q_1 = S_i], \quad 1 \le i \le N.$$

Given appropriate values of N, M, A, B, and π, the HMM can be used as a generator to give an observation sequence

$$O = O_1 O_2 \cdots O_T$$

where each observation O_t is one of the symbols from V, and T is the number of observations in the sequence. A complete specification of an HMM requires specification of two model parameters (N and M), specification of observation symbols, and the specification of the three probability measures A, B, and π.

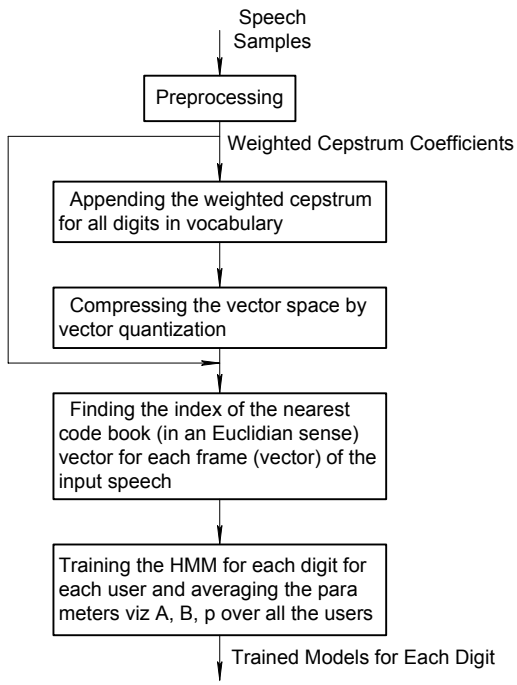
Design Methodology

We used an Altera UP3 board using a Cyclone™ EP1C6Q240C8 FPGA with a Nios II soft-core CPU. The board has 128 Mbytes of SRAM and 8 Mbytes of SDRAM, and we used both memories for our design. The SDRAM is particularly important because the SRAM alone cannot handle our data and program (Refer to Snapshot 6 in the Appendix). We used a PLL for clock input to the SDRAM for clock skew minimization. The PLL was generated using the Altera MegaWizard® Plug-In Manager.

Design Flow in Training

First, the input speech samples were preprocessed to extract the feature vectors. Then the codebook was built, serving as the reference code space with which we compare the input feature vectors. We have worked with both K-Means and Binary split algorithms and we decided to use the Binary split algorithm in our final design since it is more efficient. Then for training the HMMs, the same weighted cepstrum matrices for various users and digits are compared with the codebook, and their corresponding nearest codebook vector indices are sent to the Baum-Welch algorithm for training a input index sequence model. The Baum-Welch model is an iterative procedure and we have kept the iteration limit as 20. We now have three models for each digit corresponding to the three users in our vocabulary set, and we average the A, B, and π matrices over the users to generalize it. For the design to recognize the same digit uttered by a user for whom the design has not been trained, the zero probabilities in the B matrix have been replaced by a low value so that on recognition it gives a non-zero value. This overcomes the problem of less training data to some extent. Training is done in software and we have included the speech samples required for the software design as arrays. See Figure 4.

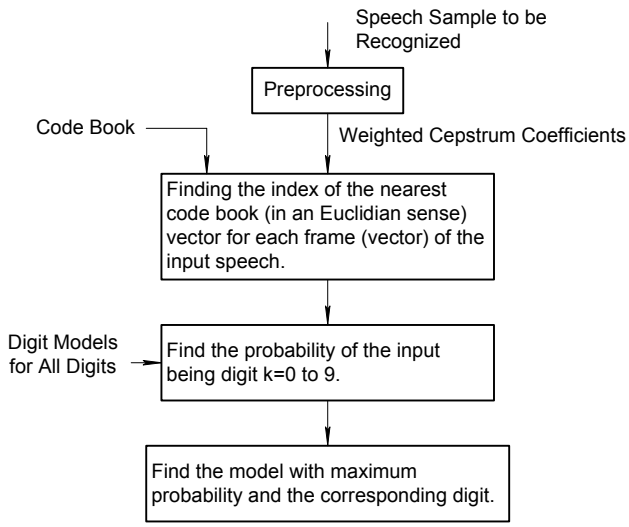
Figure 4. Training



Design Flow for Recognition

The input speech sample is preprocessed to get extract the feature vector. Then the index of the nearest codebook vector for each frame is sent to all the digit models out of which the model giving the maximum probability is chosen. Viterbi is computation intensive, so the processing steps in it have been ported to the FPGA for better speed of execution. After the soft-core processor has completed all the preprocessing steps, the required data is passed to the hardware to do the rest of the processing. Data is through the dataa and datab ports and the prefix port used for the control operations. (Refer Snapshots 2a and b in the Appendix.) See Figure 5.

Figure 5. Data Processing



Implementation Summary

Number of states of the HMMs, N: 10

Codebook size, M: 128

Order of LPC, P: 10

Number of weighted cepstrum coefficients per frame vector, Q: 11

Number of digits: 0 – 9

Speech sampling rate: 8 KHz, PCM encoded

End point detection (VAD): short-time energy-based thresholding

Feature analysis: LPC analysis

HMM training for multiple observation sequence: ensemble training (direct parameter averaging)

Design Features

Performance comparison of hardware and full software implementation could be done with the help of the facility to measure the running time of the code (Refer to comparison Snapshots 4 and 5 in the Appendix).

The HMM is rich with mathematical structure as the training of the model uses the Baum-Welch algorithm and the recognition decoding employs the Viterbi algorithm. Hence, these algorithms can be efficiently implemented using FPGAs.

The Nios II processor enables the optimum sharing of hardware and software implementations by executing more computation intensive tasks in hardware and the remaining algorithm blocks in software.

Implementation issues include:

- The Viterbi design was too big to be implemented as such in the Cyclone device. So we decided to implement only the main processing part (i.e., the floating point multiplication) in hardware. We included the synthesis report of the whole Viterbi block alone using the LeonardoSpectrum™ tool using another APEX™ FPGA (Refer to Snapshot 3 in the Appendix).

- Scaling. As T becomes sufficiently large, the range of the multiplication factors starts to reach exponentially to zero. The basic scaling procedure is used is to multiply these factors by a scaling coefficient so that they do not exceed the precision range of the machine. Over and above this, since all the values in the Baum-Welch and Viterbi algorithms are probabilities, maintaining is of utmost importance for proper results. Therefore, we have used a floating-point data format in the C program with a structure (mantissa and exponent) and used a normalization function, which removes the leading zeros and accordingly adjusts the exponent whenever it is called.

- Initial Estimates of HMM Parameters. There is no simple or straightforward answer to the above question. Either random (subject to the stochastic and the nonzero value constraints) or uniform initial estimates of the π and A parameters have to be used. However, for the B parameters, good initial estimates are helpful in the discrete symbol case. We have used uniform initial values for A and B and random values for π.

- Insufficient Training Data. The observation sequence used for training is finite. Thus there is often an insufficient number of occurrences of different model events (e.g., symbol occurrences within states) to give good estimates of the model parameters. One solution to this problem is to increase the size of the training observation set, which is often impractical. A second possible solution is to reduce the size of the model (e.g., number of states, number of symbols per state, etc) at the expense of the recognition rate. For the design to recognize a digit in the vocabulary uttered by a user for whom the design has not been trained, the zero probabilities in B matrix have been replaced by a low value so that on recognition it gives a non-zero value. This implementation overcomes the problem of less training data to some extent.

Conclusion

We learned that the Nios II processor is powerful enough to implement a full speech recognition process. Its memory and logic capabilities enabled us to implement the design. We implemented the whole design in C++ in a PC environment to check the functionality. We then ported it to the FPGA environment. That gave us a good knowledge of the issues involved in porting a software code to the SoC environment. Finally, we learnt how to include custom block in a design, how to communicate with it from the soft-core processor, what the problems are that may crop up in such a process, and how to solve them.

Probable Future Improvements

- Implementing floating point vector dot product as custom block in the short term.

- Optimizing the Viterbi part so that it can be fitted into the Cyclone device.

- Implementing other computation intensive tasks in the design like LPC processing in hardware to improve recognition time.

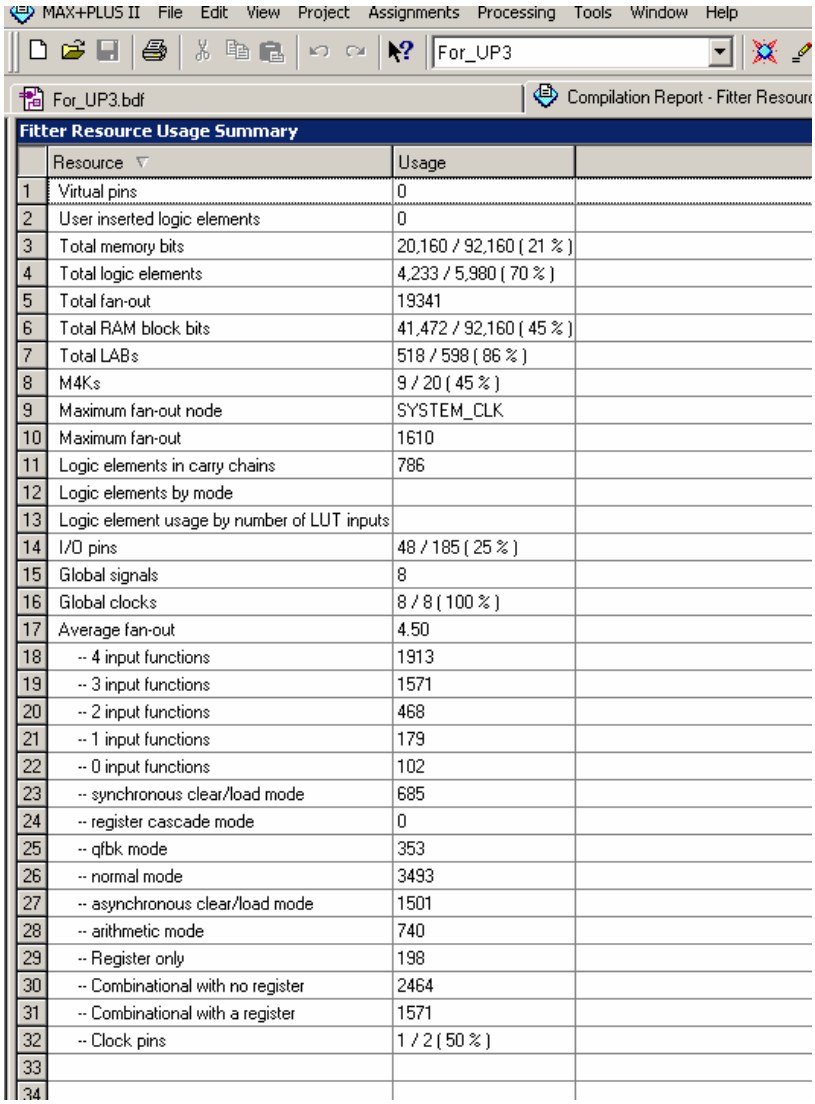
- Using Bakis model, which is claimed to model speech signals better.
- Using flash memory for using file read/write functions in the soft-core processor itself.
- Accessing SDRAM from software and hardware to pass large data, which is our requirement.

Results

The hardware and software results were verified and were found to match. The process of software/hardware co-design using SOPC is much different than conventional methods such as using microprocessor-based software routines alone or ASIC/FPGA-based full hardware implementation alone. It helped us to achieve the best of both worlds. The ability to include our hardware as a custom design in the FPGA and calling it from the software using custom instructions provided added flexibility to our design. We identified the computation intensive blocks in the design and were able to port it to the hardware for better speed. The soft IP cores helped us speed up our design time. We learned to pass data from the soft core to the hard core processor and tackling the issues in the process helped us to gain better understanding of the Nios II processor.

Appendix: Implementation Snapshots

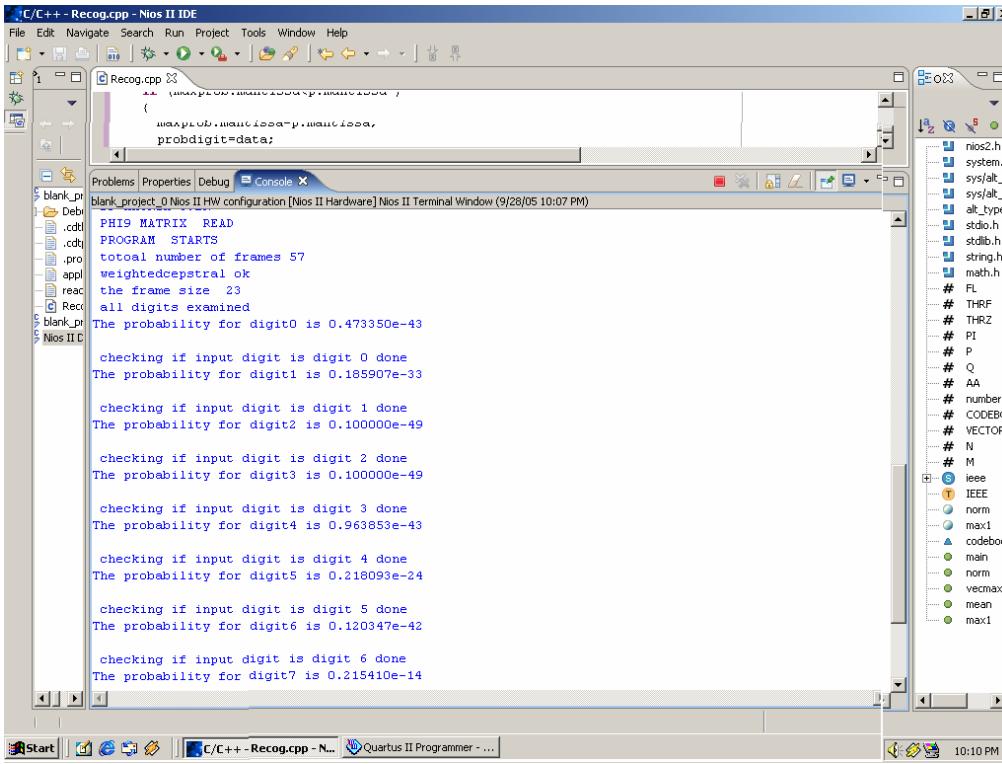
Snapshot 1. Fitter Summary for Custom Block



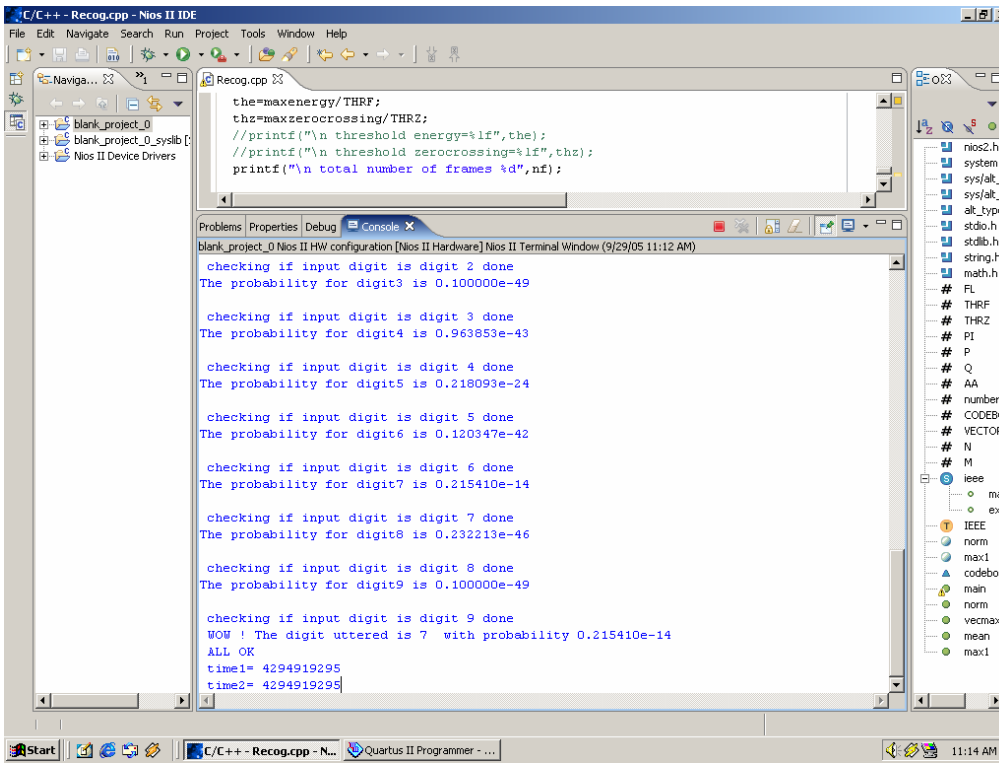
Fitter Resource Usage Summary

	Resource	Usage
1	Virtual pins	0
2	User inserted logic elements	0
3	Total memory bits	20,160 / 92,160 (21 %)
4	Total logic elements	4,233 / 5,980 (70 %)
5	Total fan-out	19341
6	Total RAM block bits	41,472 / 92,160 (45 %)
7	Total LABs	518 / 598 (86 %)
8	M4Ks	9 / 20 (45 %)
9	Maximum fan-out node	SYSTEM_CLK
10	Maximum fan-out	1610
11	Logic elements in carry chains	786
12	Logic elements by mode	
13	Logic element usage by number of LUT inputs	
14	I/O pins	48 / 185 (25 %)
15	Global signals	8
16	Global clocks	8 / 8 (100 %)
17	Average fan-out	4.50
18	-- 4 input functions	1913
19	-- 3 input functions	1571
20	-- 2 input functions	468
21	-- 1 input functions	179
22	-- 0 input functions	102
23	-- synchronous clear/load mode	685
24	-- register cascade mode	0
25	-- qfbk mode	353
26	-- normal mode	3493
27	-- asynchronous clear/load mode	1501
28	-- arithmetic mode	740
29	-- Register only	198
30	-- Combinational with no register	2464
31	-- Combinational with a register	1571
32	-- Clock pins	1 / 2 (50 %)
33		
34		

Snapshot 2a. Recognition Process for Digit 7 Uttered by User 1 as Input



Snapshot 2b. Recognition Process for Digit 7 Uttered by User 1 as Input



Snapshot 3. Synthesis Report of Whole Viterbi Block Implemented in EP20K1000EFC896 Device

```
Info: Attempting to checkout a license to run as LeonardoSpectrum Level 1 Altera
Info: License passed
Session history will be logged to file 'C:/Exemplar/LeoSpec/v19991j/bin/win32/exemplar.his'
Info, Working Directory is now 'C:\Exemplar\LeoSpec\v19991j\bin\win32'
->set _xmp_enable_renoir FALSE
FALSE
Info: system variable EXEMPLAR set to "C:\Exemplar\LeoSpec\v19991j"
Info: Loading Exemplar Blocks file: C:\Exemplar\LeoSpec\v19991j/data/xmplrblks.ini
Messages will be logged to file 'C:/Exemplar/LeoSpec/v19991j/bin/win32/exemplar.log'...
LeonardoSpectrum Level 1 Altera - v1999.1j (build 6.108, compiled Apr  6 2000 at 12:57:38)
Copyright 1990-1999 Exemplar Logic, Inc.  All rights reserved.

--
-- Welcome to LeonardoSpectrum Level 1 Altera
-- Run By ecad@VLSI-33
-- Run Started On Fri Sep 23 12:13:10 India Standard Time 2005
--
No constraint for register2register
No constraint for input2register
No constraint for input2output
No constraint for register2output
->set register2register 1073741824.0
1073741824.0
->set input2register 1073741824.0
1073741824.0
->set register2output 1073741824.0
1073741824.0
->set input2output 1073741824.0
1073741824.0
->_gc_read_init
->_gc_run_init
->set input_file_list { "D:/nios_modelsim/viterbi_ver4.v" }
 "D:/nios_modelsim/viterbi_ver4.v"
->set part EP20K1000EFC896
EP20K1000EFC896
->set process 1
1
->set flex_use_cascades TRUE
TRUE
->set chip TRUE
```

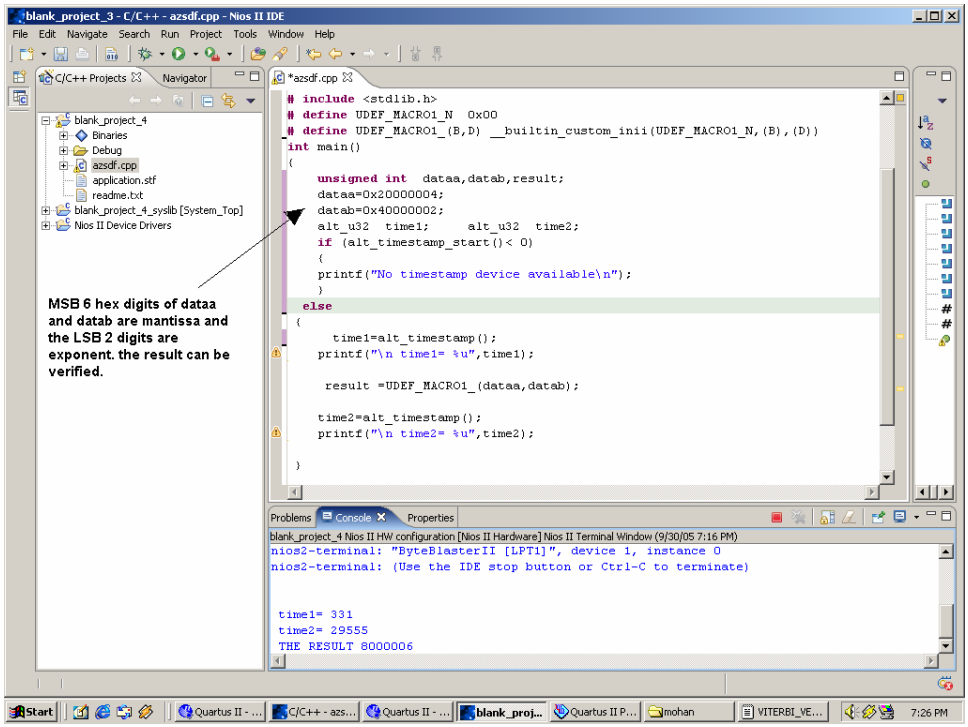
```
->set macro FALSE
FALSE
->set area TRUE
->set delay FALSE
FALSE
->set report brief
brief
->set hierarchy_auto TRUE
TRUE
->set output_file "C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf"
C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf
->set novendor_constraint_file FALSE
FALSE
->set target apex20e
apex20e
->_gc_read
-- Reading target technology apex20e
Reading library file `C:\Exemplar\LeoSpec\v19991j\lib\apex20e.syn`...
Library version = 1.6
Delays assume: Process=1
-- read -tech apex20e { "D:/nios_modelsim/viterbi_ver4.v" }
-- Reading file 'D:/nios_modelsim/viterbi_ver4.v'...
-- Loading module viterbi_ver4
-- Compiling root module 'viterbi_ver4'
"D:/nios_modelsim/viterbi_ver4.v",line 92: Info, Enumerated type _STATE_NAME_ with 28 elements encoded as onehot.
Encodings for _STATE_NAME_ values
        value    _STATE_NAME_[27-0]
==================================
       _STATE_0  ---------------------------1
       _STATE_1  --------------------------1-
       _STATE_2  -------------------------1--
       _STATE_3  ------------------------1---
       _STATE_4  -----------------------1----
       _STATE_5  ----------------------1-----
       _STATE_6  ---------------------1------
       _STATE_7  --------------------1-------
       _STATE_8  -------------------1--------
       _STATE_9  ------------------1---------
      _STATE_10  -----------------1----------
      _STATE_11  ----------------1-----------
      _STATE_12  ---------------1------------
      _STATE_13  --------------1-------------
      _STATE_14  -------------1--------------
      _STATE_15  ------------1---------------
      _STATE_16  -----------1----------------
      _STATE_17  ----------1-----------------
      _STATE_18  ---------1------------------
      _STATE_19  --------1-------------------
      _STATE_20  -------1--------------------
      _STATE_21  ------1---------------------
      _STATE_22  -----1----------------------
      _STATE_23  ----1-----------------------
      _STATE_24  ---1------------------------
      _STATE_25  --1-------------------------
      _STATE_26  -1--------------------------
      _STATE_27  1---------------------------

"D:/nios_modelsim/viterbi_ver4.v",line 164: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 215: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 248: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 307: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 285: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 307: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 343: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 378: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 440: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 417: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 440: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 417: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 440: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 487: Error, static loops are not allowed in implicit state machines.
No constraint for register2register
No constraint for input2register
No constraint for input2output
No constraint for register2output
->_gc_read_init
->_gc_run_init
->set input_file_list { "D:/nios_modelsim/viterbi_ver4.v" }
 "D:/nios_modelsim/viterbi_ver4.v"
->set chip TRUE
->set macro FALSE
FALSE
->set delay FALSE
FALSE
->set hierarchy_auto TRUE
TRUE
->set output_file "C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf"
C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf
->set sdf_write_flat_netlist TRUE
TRUE
->set target apex20e
apex20e
->_gc_read
-- Reading target technology apex20e
Reading library file `C:\Exemplar\LeoSpec\v19991j\lib\apex20e.syn`...
Library version = 1.6
```

```
Delays assume: Process=1
-- read -tech apex20e { "D:/nios_modelsim/viterbi_ver4.v" }
-- Reading file 'D:/nios_modelsim/viterbi_ver4.v'...
-- Loading module viterbi_ver4
-- Compiling root module 'viterbi_ver4'
"D:/nios_modelsim/viterbi_ver4.v",line 92: Info, Enumerated type _STATE_NAME_ with 28 elements encoded as onehot.
Encodings for _STATE_NAME_ values
        value    _STATE_NAME_[27-0]
=================================
       _STATE_0  ---------------------------1
       _STATE_1  --------------------------1-
       _STATE_2  -------------------------1--
       _STATE_3  ------------------------1---
       _STATE_4  -----------------------1----
       _STATE_5  ----------------------1-----
       _STATE_6  ---------------------1------
       _STATE_7  --------------------1-------
       _STATE_8  -------------------1--------
       _STATE_9  ------------------1---------
      _STATE_10  -----------------1----------
      _STATE_11  ----------------1-----------
      _STATE_12  ---------------1------------
      _STATE_13  --------------1-------------
      _STATE_14  -------------1--------------
      _STATE_15  ------------1---------------
      _STATE_16  -----------1----------------
      _STATE_17  ----------1-----------------
      _STATE_18  ---------1------------------
      _STATE_19  --------1-------------------
      _STATE_20  -------1--------------------
      _STATE_21  ------1---------------------
      _STATE_22  -----1----------------------
      _STATE_23  ----1-----------------------
      _STATE_24  ---1------------------------
      _STATE_25  --1-------------------------
      _STATE_26  -1--------------------------
      _STATE_27  1---------------------------

"D:/nios_modelsim/viterbi_ver4.v",line 216: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 249: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 308: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 286: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 308: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 344: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 379: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 441: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 418: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 441: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 418: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 441: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 488: Error, static loops are not allowed in implicit state machines.
No constraint for register2register
No constraint for input2register
No constraint for input2output
No constraint for register2output
->_gc_read_init
->_gc_run_init
->set input_file_list { "D:/nios_modelsim/viterbi_ver4.v" }
 "D:/nios_modelsim/viterbi_ver4.v"
->set chip TRUE
->set macro FALSE
FALSE
->set delay FALSE
FALSE
->set hierarchy_auto TRUE
TRUE
->set output_file "C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf"
C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf
->set target apex20e
apex20e
->_gc_read
-- Reading target technology apex20e
Reading library file `C:\Exemplar\LeoSpec\v19991j\lib\apex20e.syn`...
Library version = 1.6
Delays assume: Process=1
-- read -tech apex20e { "D:/nios_modelsim/viterbi_ver4.v" }
-- Reading file 'D:/nios_modelsim/viterbi_ver4.v'...
"D:/nios_modelsim/viterbi_ver4.v", line 526: Warning, system task enable ignored for synthesis
-- Loading module viterbi_ver4
-- Compiling root module 'viterbi_ver4'
"D:/nios_modelsim/viterbi_ver4.v",line 93: Info, Enumerated type _STATE_NAME_ with 28 elements encoded as onehot.
Encodings for _STATE_NAME_ values
        value    _STATE_NAME_[27-0]
===================================
       _STATE_0  ---------------------------1
       _STATE_1  --------------------------1-
       _STATE_2  -------------------------1--
       _STATE_3  ------------------------1---
       _STATE_4  -----------------------1----
       _STATE_5  ----------------------1-----
       _STATE_6  ---------------------1------
       _STATE_7  --------------------1-------
       _STATE_8  -------------------1--------
       _STATE_9  ------------------1---------
      _STATE_10  -----------------1----------
      _STATE_11  ----------------1-----------
```

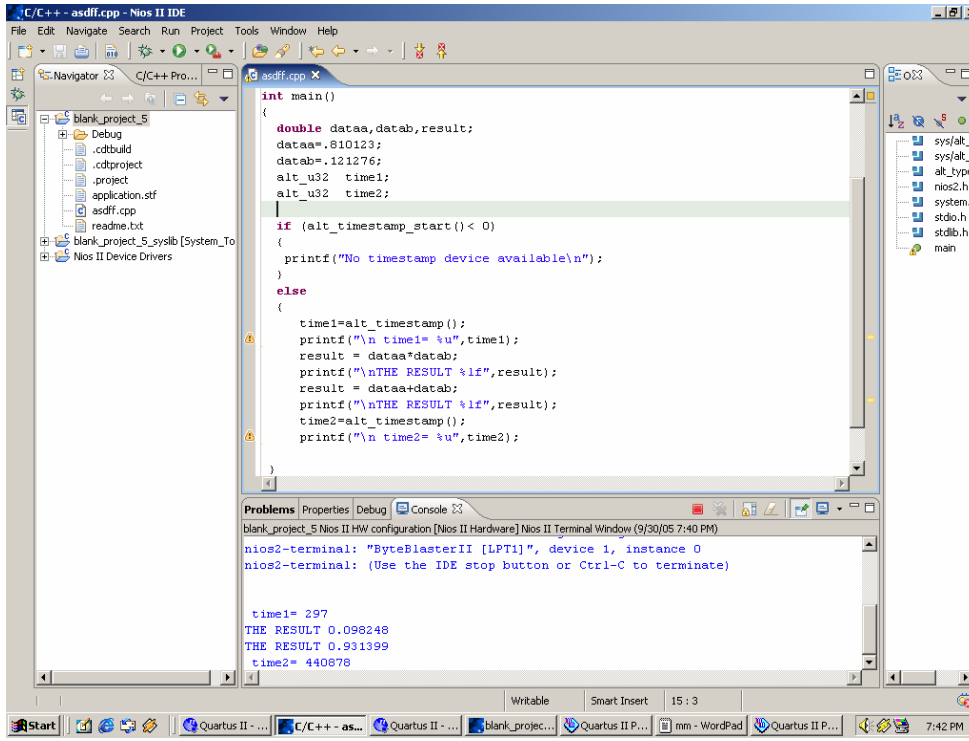
```
      _STATE_12  ---------------1------------
      _STATE_13  --------------1-------------
      _STATE_14  -------------1--------------
      _STATE_15  ------------1---------------
      _STATE_16  -----------1----------------
      _STATE_17  ----------1-----------------
      _STATE_18  ---------1------------------
      _STATE_19  --------1-------------------
      _STATE_20  -------1--------------------
      _STATE_21  ------1---------------------
      _STATE_22  -----1----------------------
      _STATE_23  ----1-----------------------
      _STATE_24  ---1------------------------
      _STATE_25  --1-------------------------
      _STATE_26  -1--------------------------
      _STATE_27  1---------------------------

"D:/nios_modelsim/viterbi_ver4.v", line 39:Info, D-Flipflop reg_si(237)(8) is unused, optimizing...
"D:/nios_modelsim/viterbi_ver4.v", line 39:Info, D-Flipflop reg_si(238)(8) is unused, optimizing...
"D:/nios_modelsim/viterbi_ver4.v", line 39:Info, D-Flipflop reg_si(239)(8) is unused, optimizing...
Info: Finished reading design
->_gc_run
-- Run Started On Fri Sep 23 13:24:10 India Standard Time 2005
--
-- optimize -target apex20e -effort quick -chip -area -hierarchy=auto
Using default wire table: apex20e_default
Warning, View .work.viterbi_ver4.INTERFACE needs partitioning, you may want to optimize this block in "auto
dissolve" hierarchy mode...
Warning, View .work.viterbi_ver4.INTERFACE needs partitioning, you may want to optimize this block in "auto
dissolve" hierarchy mode...
Warning, View .work.cx0.partition_vx0 needs partitioning, you may want to optimize this block in "auto dissolve"
hierarchy mode...
-- Start optimization for design .work.cx2.partition_vx0
Using default wire table: apex20e_default
             est est
      Pass    LCs Delay DFFs TRIs  PIs POs    --CPU--
                                              min:sec
      1       6961    106 1809    0  810 688      07:07
-- Start optimization for design .work.cx0.partition_vx0
Using default wire table: apex20e_default
             est est
      Pass    LCs Delay DFFs TRIs  PIs POs    --CPU--
                                              min:sec
      1       8351    116 2411    0  467 545      07:43
-- Start optimization for design .work.cx1.partition_vx0
Using default wire table: apex20e_default
             est est
      Pass    LCs Delay DFFs TRIs  PIs POs    --CPU--
                                              min:sec
      1       6087     73 1472    0  606 331      01:56
-- Start optimization for design .work.viterbi_ver4.INTERFACE
Using default wire table: apex20e_default
             est est
      Pass    LCs Delay DFFs TRIs  PIs POs    --CPU--
                                              min:sec
      1       7435     58 1767    0    2  18      03:24
Using default wire table: apex20e_default
-- Start timing optimization for design .work.viterbi_ver4.INTERFACE
No critical paths to optimize at this level

*****************************************************

Cell: viterbi_ver4    View: INTERFACE    Library: work

*****************************************************

 Number of ports :                     20
 Number of nets :                   31206
 Number of instances :              29214
 Number of references to this view:     0

Total accumulated area:
 Number of GND:                         4
 Number of I/Os:                       20
 Number of LCs:                     28938
 Number of Memory Bits:              7680
 Number of VCC:                         1

*********************************************
Device Utilization for EP20K1000EFC896
*********************************************
Resource                Used    Avail   Utilization
---------------------------------------------
I/Os                     20      896       2.23%
LCs                    28938   38400    75.36%
Memory Bits            7680    540672    1.42%

---------------------------------------------
                       Clock Frequency Report

     Clock                : Frequency
      ------------------------------------

     clk                  : 12.1 MHz

                       Critical Path Report
```

```
Critical path #1, (unconstrained path)
NAME                                                                    GATE              ARRIVAL              LOAD
-------------------------------------------------------------------------------------------------------------------
clock information not specified
delay thru clock network                                                                  0.00 (ideal)


reg_i5(7)/regout                                                        apex20_lcell_normal  0.00  2.30 up             1.45
modgen_add_6752_ix86/combout                                            apex20_lcell_arithmetic  4.96  7.25 up            2.05
modgen_mux_6996_ix304/cascout                                           apex20_lcell_normal  2.01  9.26 up             1.22
modgen_mux_6996_ix306/combout                                           apex20_lcell_normal  0.53  9.79 up             1.22
modgen_mux_6996_ix310/cascout                                           apex20_lcell_normal  2.04  11.83 up             1.22
modgen_mux_6996_ix312/combout                                           apex20_lcell_normal  0.53  12.36 up             1.22
ix1500353/cascout                                                       apex20_lcell_normal  2.01  14.36 up             1.22
ix1500226/combout                                                       apex20_lcell_normal  0.53  14.89 up             1.22
ix1415575/Y                                                             NOT         1.49  16.38 up             1.22
modgen_mux_6996_ix730/combout                                           apex20_lcell_normal  3.10  19.48 up             1.55
modgen_gt_7005_ix39/combout                                             apex20_lcell_arithmetic  4.96  24.43 up            2.05
ix1507344/combout                                                       apex20_lcell_normal  3.39  27.82 up             1.64
ix6362/Y                                                                NOT         4.20  32.02 up             2.05
ix1504380/combout                                                       apex20_lcell_normal  2.18  34.21 up             1.22
modgen_mux_7532_ix316/cascout                                           apex20_lcell_normal  1.98  36.19 up             1.22
modgen_mux_7532_ix318/combout                                           apex20_lcell_normal  0.53  36.71 up             1.22
modgen_mux_7532_ix342/combout                                           apex20_lcell_normal  2.18  38.90 up             1.22
ix1504445/cascout                                                       apex20_lcell_normal  1.98  40.88 up             1.22
ix1503646/combout                                                       apex20_lcell_normal  0.53  41.41 up             1.22
ix15950/Y                                                               NOT         1.49  42.90 up             1.22
modgen_mux_7532_ix730/combout                                           apex20_lcell_normal  3.10  45.99 up             1.55
modgen_gt_7522_ix23/cout                                                apex20_lcell_arithmetic  2.01  48.00 up            1.22
modgen_gt_7522_ix25/cout                                                apex20_lcell_arithmetic  0.09  48.09 up            1.22
modgen_gt_7522_ix27/cout                                                apex20_lcell_arithmetic  0.09  48.18 up            1.22
modgen_gt_7522_ix29/cout                                                apex20_lcell_arithmetic  0.09  48.27 up            1.22
modgen_gt_7522_ix31/cout                                                apex20_lcell_arithmetic  0.09  48.35 up            1.22
modgen_gt_7522_ix33/cout                                                apex20_lcell_arithmetic  0.09  48.44 up            1.22
modgen_gt_7522_ix35/cout                                                apex20_lcell_arithmetic  0.09  48.53 up            1.22
modgen_gt_7522_ix37/cout                                                apex20_lcell_arithmetic  0.09  48.62 up            1.22
modgen_gt_7522_ix39/combout                                             apex20_lcell_arithmetic  0.53  49.15 up            1.90
ix1507591/combout                                                       apex20_lcell_normal  4.96  54.11 up             2.05
ix1507333/combout                                                       apex20_lcell_normal  4.90  59.00 up             2.05
ix1501713/combout                                                       apex20_lcell_normal  3.70  62.70 up             1.73
ix1502628/combout                                                       apex20_lcell_normal  2.21  64.91 up             1.22
ix1503203/combout                                                       apex20_lcell_normal  2.16  67.07 up             1.22
ix1507603/combout                                                       apex20_lcell_normal  2.47  69.53 up             1.34
ix1507602/combout                                                       apex20_lcell_normal  3.10  72.63 up             1.55
ix1502721/combout                                                       apex20_lcell_normal  2.21  74.84 up             1.22
ix1503153/combout                                                       apex20_lcell_normal  2.21  77.05 up             1.22
ix1497841/combout                                                       apex20_lcell_normal  4.87  81.92 up             2.05
reg_e_delta(5)(7)/ena                                                   apex20_lcell_normal  0.00  81.92 up             0.00
data arrival time                                                                         81.92


data required time  (default specified - setup time)                                 not specified
-------------------------------------------------------------------------------------------------------------------
data required time                                                                   not specified
data arrival time                                                                       81.92
                                                                                     ----------
                                                                                   unconstrained path
-------------------------------------------------------------------------------------------------------------------


-- Design summary in file 'C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.sum'
-- Writing file C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf
Info, Writing xrf file 'C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.xrf'
-- Writing file C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.xrf
Info, Writing batch file 'C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.tcl'
-- CPU time taken for this run was  3017.14 sec
-- Run Successfully Ended On  Fri Sep 23 14:14:27 India Standard Time 2005
0
Info: Finished Synthesis run
```

Snapshot 4. Hardware Floating Multiplier Custom Block & its Run Time with System Frequency of 48 MHz (Run Time Mentioned as Number of Clock Ticks)



Snapshot 5. Software Floating Multiplier & its Run Time With System Frequency of 48 MHz (Run Time Mentioned as Number of Clock Ticks)



Snapshot 6. Memory Usage Summary in Nios II IDE