Second Prize

SOPC-Based Speech-to-Text Conversion

Institution: **National Institute of Technology, Trichy**

Participants: **M.T. Bala Murugan and M. Balaji**

Instructor: **Dr. B. Venkataramani**

Design Introduction

For the past several decades, designers have processed speech for a wide variety of applications ranging from mobile communications to automatic reading machines. Speech recognition reduces the overhead caused by alternate communication methods. Speech has not been used much in the field of electronics and computers due to the complexity and variety of speech signals and sounds. However, with modern processes, algorithms, and methods we can process speech signals easily and recognize the text.

Objective

In our project, we developed an on-line speech-to-text engine, implemented as a system-on-a-programmable-chip (SOPC) solution. The system acquires speech at run time through a microphone and processes the sampled speech to recognize the uttered text. We used the hidden Markov model (HMM) for speech recognition, which converts the speech to text. The recognized text can be stored in a file on a PC that connects to an FPGA on a development board using a standard RS-232 serial cable.

Our speech-to-text system directly acquires and converts speech to text. It can supplement other larger systems, giving users a different choice for data entry. A speech-to-text system can also improve system accessibility by providing data entry options for blind, deaf, or physically handicapped users.

Project Outline

The project implements a speech-to-text system using isolated word recognition with a vocabulary of ten words (digits 0 to 9) and statistical modeling (HMM) for machine speech recognition. In the training phase, the uttered digits are recorded using 16-bit pulse code modulation (PCM) with a sampling rate of 8 KHz and saved as a wave file using sound recorder software. We use the MATLAB software's **wavread** command to convert the **.wav** files to speech samples.

Generally, a speech signal consists of noise-speech-noise. The detection of actual speech in the given samples is important. We divided the speech signal into frames of 450 samples each with an overlap of 300 samples, i.e., two-thirds of a frame length. The speech is separated from the pauses using voice activity detection (VAD) techniques, which are discussed in detail later in the paper.

The system performs speech analysis using the linear predictive coding (LPC) method. From the LPC coefficients we get the weighted cepstral coefficients and cepstral time derivatives, which form the feature vector for a frame. Then, the system performs vector quantization using a vector codebook. The resulting vectors form the observation sequence. For each word in the vocabulary, the system builds an HMM model and trains the model during the training phase. The training steps, from VAD to HMM model building, are performed using PC-based C programs. We load the resulting HMM models onto an FPGA for the recognition phase.

In the recognition phase, the speech is acquired dynamically from the microphone through a codec and is stored in the FPGA's memory. These speech samples are preprocessed, and the probability of getting the observation sequence for each model is calculated. The uttered word is recognized based on a maximum likelihood estimation.

FPGA Design Significance

The trend in hardware design is towards implementing a complete system, intended for various applications, on a single chip. The advent of high-density FPGAs with high-capacity RAMs, and support for soft-core processors such as Altera's Nios® II processor, have enabled designers to implement a complete system on a chip. FPGAs provide the following benefits:

- FPGA systems are portable, cost effective, and consume very little power compared to PCs. A complete system can be implemented easily on a single chip because complex integrated circuits (ICs) with millions of gates are available now.
- SOPC Builder can trim months from a design cycle by simplifying and accelerating the design process. It integrates complex system components such as intellectual property (IP) blocks, memories, and interfaces to off-chip devices including application-specific standard products (ASSPs) and ASICs on Altera® high-density FPGAs.
- SOPC methodology gives the designer flexibility when writing code, and supports both high-level languages (HLLs) and hardware description language (HDLs). The time-critical blocks can be implemented in an HDL while the remaining blocks are implemented in an HLL. It is easy to change the existing algorithm in hardware and software by simply modifying the code.
- FPGAs provide the best of both worlds: a microcontroller or RISC processor can efficiently perform control and decision-making operations while the FPGA can perform digital signal processing (DSP) operations and other computationally intensive tasks.
- An FPGA supports hardware/software co-design in which the time-critical blocks are written in HDL and implemented as hardware units, while the remaining application logic is written C. The challenge is to find a good tradeoff between the two. Both the processor and the custom hardware must be optimally designed such that neither is idle or under-utilized.

Nios II-Based Design

We decided to use an FPGA after analyzing the various requirements for a speech-to-text conversion system. A basic system requires application programs, running on a customizable processor, that can implement custom digital hardware for computationally intensive operations such as fast Fourier transform (FFT), Viterbi decoding, etc. Using a soft-core processor, we can implement and customize various interfaces, including serial, parallel, and Ethernet.

The Altera development board, user-friendly Quartus® II software, SOPC Builder, Nios II Integrated Development Environment (IDE), and associated documentation enable even a beginner to feel at ease with developing an SOPC design. We can perform hardware design and simulation using the Quartus II software and use SOPC Builder to create the system from readily available, easy-to-use components. With the Nios II IDE, we easily created application software for the Nios II processor with the intuitive click-to-run IDE interface. The development board's rich features and customization, SOPC Builder's built-in support for interfaces (such as serial, parallel, Ethernet, and USB), and the easy programming interface provided by the Nios II hardware application layer (HAL) make the Nios II processor and an FPGA the ideal platform for implementing our on-line speech-to-text system.

Application Scope and Targeted Users

The design is very generic and can be used in a variety of applications. The basic system with the vocabulary of numbers from 0 to 9 can be used in applications such as:

- Interactive voice response system (IVRS)
- Voice-dialing in mobile phones and telephones
- Hands-free dialing in wireless bluetooth headsets
- PIN and numeric password entry modules
- Automated teller machines (ATMs)

If we increased the system's vocabulary using phoneme-based recognition for a particular language, e.g., English, the system could be used to replace standard input devices such as keyboards, touch pads, etc. in IT- and electronics-based applications in various fields. The design has a wide market opportunity ranging from mobile service providers to ATM makers. Some of the targeted users for the system include:

- Value added service (VAS) providers
- Mobile operators
- Home and office security device providers
- ATM manufacturers
- Mobile phone and bluetooth headset manufacturers
- Telephone service providers
- Manufacturers of instruments for disabled persons
- PC users

Function Description

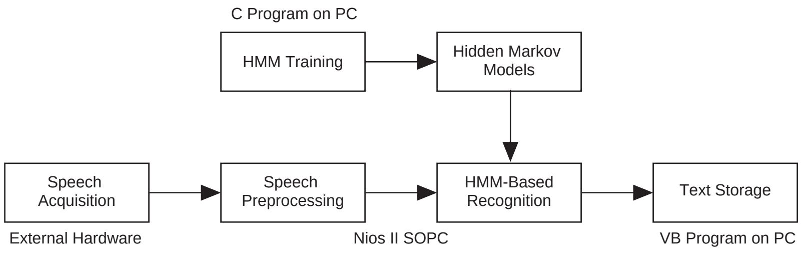
The project converts speech into text using an FPGA. Run-time speech data is acquired from a transducer through analog-to-digital (A/D) conversion and is sent to the FPGA, which provides speech preprocessing and recognition using the Nios II processor. The system sends text data to a PC for storage. Functionally, the project has the following blocks:

- Speech acquisition
- Speech preprocessing

- HMM training
- HMM-based recognition
- Text storage

Figure 1 shows the system's functional block diagram

Figure 1. Functional Block Diagram



Speech Acquisition

During speech acquisition, speech samples are obtained from the speaker in real time and stored in memory for preprocessing. Speech acquisition requires a microphone coupled with an analog-to-digital converter (ADC) that has the proper amplification to receive the voice speech signal, sample it, and convert it into digital speech. The system sends the analog speech through a transducer, amplifies it, sends it through an ADC, and transmits it to the Nios II processor in the FPGA through the communications interface. The system needs a parallel/serial interface to the Nios II processor and an application running on the processor that acquires and stores data in memory. The received samples are stored into memory on the Altera Development and Education (DE2) development board.

We easily implemented speech acquisition with the Altera DE2 development board. The microphone input port with the audio codec receives the signal, amplifies it, and converts it into 16-bit PCM digital samples at a sampling rate of 8 KHz. The codec requires initial configuration, which is performed using custom hardware implemented in the Altera Cyclone® II FPGA on the board.

The audio codec provides a serial communication interface, which is connected to a UART. We used SOPC Builder to add the UART to the Nios II processor to enable the interface. The UART is connected to the processor through the Avalon® bus. The C application running on the HAL transfers data from the UART to the SDRAM. Direct memory access (DMA) transfers data efficiently and quickly, and we may use it instead in future designs.

Speech Preprocessing

The speech signal consists of the uttered digit along with a pause period and background noise. Preprocessing reduces the amount of processing required in later stages. Generally, preprocessing involves taking the speech samples as input, blocking the samples into frames, and returning a unique pattern for each sample, as described in the following steps.

1. The system must identify useful or significant samples from the speech signal. To accomplish this goal, the system divides the speech samples into overlapped frames.

2. The system checks the frames for voice activity using endpoint detection and energy threshold calculations.

3. The speech samples are passed through a pre-emphasis filter.

4. The frames with voice activity are passed through a Hamming window.

5. The system performs autocorrelation analysis on each frame.

6. The system finds linear predictive coding (LPC) coefficients using the Levinson and Durbin algorithm.

7. From the LPC coefficients, the system determines the cepstral coefficients and weighs them using a tapered window. The cepstral coefficients serve as feature vectors.

Using temporal cepstral derivatives improves the speech frame feature vectors. We use them in case the cepstral coefficients do not give acceptable recognition accuracy.

We implemented these steps with C programs, which execute based on the algorithms. Although we used the same C programs for training and recognition, the C code executes on a PC during training and it runs on the Nios II processor during recognition.

HMM Training

An important part of speech-to-text conversion using pattern recognition is training. Training involves creating a pattern representative of the features of a class using one or more test patterns that correspond to speech sounds of the same class. The resulting pattern (generally called a reference pattern) is an example or template, derived from some type of averaging technique. It can also be a model that characterizes the reference pattern statistics. Our system uses speech samples from three individuals during training.

A model commonly used for speech recognition is the HMM, which is a statistical model used for modeling an unknown system using an observed output sequence. The system trains the HMM for each digit in the vocabulary using the Baum-Welch algorithm. The codebook index created during preprocessing is the observation vector for the HMM model.

After preprocessing the input speech samples to extract feature vectors, the system builds the codebook. The codebook is the reference code space that we can use to compare input feature vectors. The weighted cepstrum matrices for various users and digits are compared with the codebook. The nearest corresponding codebook vector indices are sent to the Baum-Welch algorithm for training an HMM model.

The HMM characterizes the system using three matrices:

- *A*—The state transition probability distribution.
- *B*—The observation symbol probability distribution.
- *n*—The initial state distribution.

Any digit is completely characterized by its corresponding A, B, and n matrices. The A, B, and n matrices are modeled using the Baum-Welch algorithm, which is an iterative procedure (we limit the iterations to 20). The Baum-Welch algorithm gives 3 matrices for each digit corresponding to the 3 users with whom we created the vocabulary set. The A, B, and n matrices are averaged over the users to generalize them for user-independent recognition.

For the design to recognize the same digit uttered by a user for which the design has not been trained, the zero probabilities in the B matrix are replaced with a low value so that it gives a non-zero value on recognition. To some extent, this arrangement overcomes the problem of less training data.

Training is a one-time process. Due to the complexity and resource requirements, it is performed using standalone PC application software that we created by compiling our C program into an executable. For recognition, we compile the same C program but target it to run on the Nios II processor instead. We were able to accomplish this cross-compilation because of the wide support for the C language in the Nios II processor IDE.

The C program running on the PC takes the digit speech samples from a MATLAB output file and performs preprocessing, feature vector extraction, vector quantization, Baum-Welch modeling, and averaging, which outputs the normalized A, B, and n matrices for each digit. The normalized A, B, and n matrices are then embedded in the recognition C program code and stored in the DE2 development board's SDRAM using the Nios II IDE.

HMM-Based Recognition

Recognition or pattern classification is the process of comparing the unknown test pattern with each sound class reference pattern and computing a measure of similarity (distance) between the test pattern and each reference pattern. The digit is recognized using a maximum likelihood estimate, such as the Viterbi decoding algorithm, which implies that the digit whose model has the maximum probability is the spoken digit.

Preprocessing, feature vector extraction, and codebook generation are same as in HMM training. The input speech sample is preprocessed and the feature vector is extracted. Then, the index of the nearest codebook vector for each frame is sent to all digit models. The model with the maximum probability is chosen as the recognized digit.

After preprocessing in the Nios II processor, the required data is passed to the hardware for Viterbi decoding. Viterbi decoding is computationally intensive so we implemented it in the FPGA for better execution speed, taking advantage of hardware/software co-design. We wrote the Viterbi decoder in Verilog HDL and included it as a custom instruction in the Nios II processor. Data passes through the `dataa` and `datab` ports and the `prefix` port is used for control operations. The custom instruction copies or adds two floating-point numbers from `dataa` and `datab`, depending on the `prefix` input. The output (`result`) is sent back to the Nios II processor for further maximum likelihood estimation.

Text Storage

Our speech-to-text conversion system can send the recognized digit to a PC via the serial, USB, or Ethernet interface for backup or archiving. For our testing, we used a serial cable to connect the PC and RS-232 port on the DE2 board. The Nios II processor on the DE2 board sends the digital speech data to a PC; a target program running on the PC receives the text and writes it to the disk.

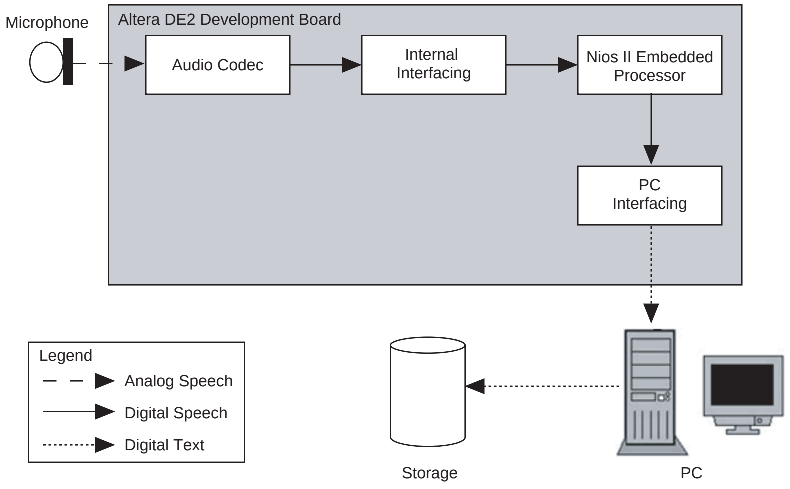
We wrote the PC program using Visual Basic 6 (VB) using a Microsoft serial port control. The VB program must be run in the background for the PC to receive the data and write it to the hard disk. The Windows HyperTerminal software or any other RS-232 serial communication receiver could also be used to receive and view the data. The serial port communication runs at 115,200 bits per second (bps) with 8 data bits, 1 stop bit, and no parity. Handshaking signals are not required.

The speech-to-text conversion system can also operate as a standalone network device using an Ethernet interface for PC communication and appropriate speech recognition server software designed for the Nios II processor.

Design Architecture

Figure 2 shows the system's block diagram.

Figure 2. Block Diagram



The system has the following parts:

- Training software
- Audio input hardware
- FPGA interface
- Nios II software
- PC software for storage

Design Description

The design is an efficient hardware speech-to-text system with an FPGA coprocessor that recognizes speech much faster than software. With an Altera FPGA, we implemented the time-critical functions (written in VHDL/Verilog HDL) in hardware. The Nios II processor, which is downloaded onto the FPGA, executes C programs. Custom instructions let us implement the whole system on an FPGA with better software and hardware partitioning.

The phases of the project were:

- Speech sampling at 8 KHz and encoded as 16-bit PCM.
- Preprocessing (voice activity detection, autocorrelation analysis, and feature vector extraction).
- Training (codebook generation, finding codebook index of uttered word, Baum-Welch for training HMM).
- Recognition (Viterbi decoding and maximum likelihood estimation).

The following table shows the methods we used and their purpose.

Methods Used

Method	Purpose
Threshold detection	Detects the voice activity.
LPC analysis	Extracts the feature vector.
HMM	Characterizes real signals in terms of signal models.
Viterbi decoding	Detects the unknown word.
Maximum likelihood estimation	Chooses the best matching digit.

We used the following software and hardware to develop the design:

- Sound recorder
- MATLAB version 6
- C – DevC++ with gcc and gdb
- Quartus II version 5.1
- SOPC Builder version 5.1
- Nios II processor
- Nios II IDE version 5.1
- MegaCore® IP library
- Altera Development and Education (DE2) board
- Microphone and headset
- RS-232 cable

Speech Acquisition

Speech acquisition requires a microphone coupled with an amplified ADC to receive the voice speech signal, sample it, and convert it into digital speech for input to the FPGA. The DE2 development board has a WM8731 audio codec chip connected both to the microphone input pins and the Altera Cyclone II FPGA pins through an I^2C serial controller interface. The WM8731 is a versatile audio codec that provides up to 24-bit encoding/decoding of audio signals in various sampling rates ranging from 8 to 96 KHz.

The codec clock input is generated by dividing the system clock by four using a custom hardware block. The block combines the clock divider logic and the I2S digital audio interface logic as well as options for programming the control registers. The DE2 board’s microphone input port connects the microphone and headset for speech acquisition. The following table shows the codec's control register settings:

Codec Register Settings

Register	Setting
ADCDAT	16 bit
USB/normal mode	Normal mode
Master/slave mode	Master
Digital audio interface	I2S interface
MCLK input	50 MHz divided by 4
Bit oversampling rate	256 fs
ADC sampling rate	8 KHz
Serial control mode	2-wire interface

These settings are programmed by setting or resetting of various bits in the control registers as shown in the following table:

Control Register Settings

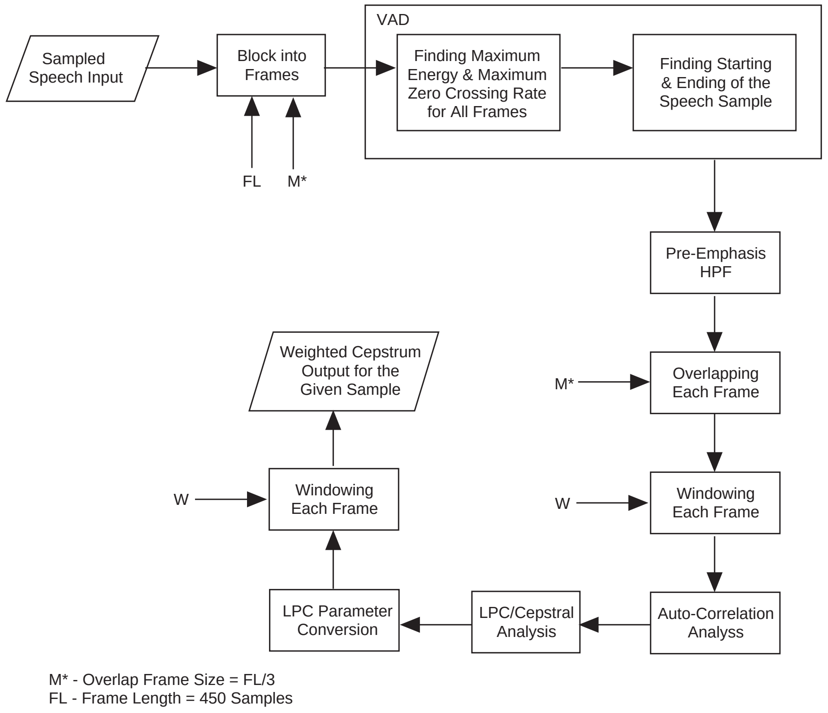
Register	Address	Register Name	Value
R0	0000000	Left line in	0017
R1	0000001	Right line in	0217
R2	0000010	Left headphone Out	047F
R3	0000011	Right headphone Out	067F
R4	0000100	Analog audio path control	0838
R6	0000110	Power down control	0C00
R7	0000111	Digital audio interface format	0EC2
R8	0001000	Sampling control	1030
R9	0001001	Active control	1201

The I2S digital audio interface connects to the Nios II processor via a UART. The UART has two additional pins, one for serial transmit data and one for serial receive data, and SOPC Builder connects the UART to the Nios II processor via the Avalon bus. The UART is also supported by the HAL, which makes it easy to get the input from the codec through the UART by reducing communication to simple file-handling functions. The `fopen()` function opens the UART (**/dev/micinput**) as a file and the `fread()` function reads 16 bits by 16 bits whenever required. The data is stored directly in the SDRAM, which contains a buffer array in the heap region. The buffer array is a circular buffer that loads the speech samples and overwrites with new samples when processing finishes.

Preprocessing

Preprocessing takes the speech samples as the input, blocks it into frames, and returns a unique pattern for each sample. Figure 3 shows the preprocessing block diagram.

Figure 3. Preprocessing Block Diagram


VAD
Sampled Speech Input
Block into Frames
Finding Maximum Energy & Maximum Zero Crossing Rate for All Frames
Finding Starting & Ending of the Speech Sample
FL
M*
Pre-Emphasis HPF
M*
Overlapping Each Frame
W
Windowing Each Frame
Auto-Correlation Analyss
LPC/Cepstral Analysis
LPC Parameter Conversion
W
Windowing Each Frame
Weighted Cepstrum Output for the Given Sample
M* - Overlap Frame Size = FL/3
FL - Frame Length = 450 Samples

Voice Activity Detection

The system uses the endpoint detection algorithm to find the start and end points of the speech. The speech is sliced into frames that are 450 samples long. Next, the system finds the energy and number of zero crossings of each frame. The threshold energy and zero crossing value is determined based on the computed values and only frames crossing the threshold are considered, removing most background noise. We include a small number of frames beyond the starting and ending frames so that we do not miss starting or ending parts that do not cross the threshold but may be important for recognition.

Pre-Emphasis

The digitized speech signal s(n) is put through a low-order LPF to flatten the signal spectrally and make it less susceptible to finite precision effects later in the signal processing. The filter is represented by the equation:

$H(z) = 1 - az^{-1}$ where a is 0.9375.

Frame Blocking

Speech frames are formed with a duration of 56.25 ms (N = 450 sample length) and an overlap of 18.75 ms (M = 150 sample length) between adjacent frames. The overlapping ensures that the resulting LPC spectral estimates are correlated from frame to frame and are quite smooth. We use the equation:

$Xq(n) = s(Mq + n)$

with n = 0 to N - 1 and q = 0 to L - 1, where L is the number of frames.

Windowing

We apply a Hamming window to each frame to minimize signal discontinuities at the beginning and end of the frame according to the equation:

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

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

Autocorrelation Analysis

We perform autocorrelation analysis for each frame and find P + 1 (P = 10) autocorrelation coefficients according to the equation:

$$r_q m = \sum_{n=0}^{N-1-m} x'_q(n) x'_q(n+m)$$

where m = 0, 1, ..., P

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

LPC

The system performs LPC analysis using Levinson and Durbin’s algorithm to convert the autocorrelation coefficients to the LPC parameter set according to the following equations:

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

$$K_i = \left\{ r_q(i) - \left\{ \sum_{j=1}^{L-1} a_j^{(i-1)} \cdot \ r_q(|(i = j)|) \right\} \right\} / \ E^{i-j}$$

for $1 \leq i \leq P$

$a_i^{(i)} = k_i$

$a_j^{(i)} = a_j^{(i-1)} - k_i.\ a_{(i-j)}^{(i-1)}$

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

where a_m^{10} $1 \leq m \leq P$ are the LPC coefficients.

Cepstral Coefficient Conversion

The cepstrum coefficients provide a more robust, reliable feature set than the LPC coefficients. We use the following equations:

$c_0 = \ln\sigma^2$

where σ is the gain of the LPC model.

$$c_m = a_m + \sum_{k=1}^{m-1} \left(\frac{k}{m}\right) \cdot c_k \cdot a_{m-k}$$

for $1 \le m \le P$

$$c_m = \sum_{k=1}^{m-1} \left(\frac{k}{m}\right) \cdot c_k \cdot a_{m-k}$$

for $P < m \le Q$

Parameter Weighing

Because the lower-order cepstral coefficients are sensitive to the overall slope and the higher-order coefficients are sensitive to noise, we need to weigh the cepstral coefficients with a tapered window to minimize the sensitivity. We weighed the coefficients using a band-pass filter of the form:

$w_m = [1 + (Q/2). \sin(\pi m/Q)]$ for $1 \le m \le Q$

Temporal Cepstral Derivative

We can obtain improved feature vectors for the speech frames using temporal cepstral derivatives. We use them with the cepstral derivative if the cepstral coefficients do not have an acceptable recognition accuracy.

Vector Quantization

A codebook of size 128 is obtained by vector quantizing the weighted cepstral coefficients of all reference digits generated by all users. The advantages of vector quantization are:

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

One obvious disadvantage of vector quantization is the reduced resolution in recognition. Assigning a codebook index to an input speech vector amounts to quantizing it, which results in quantization errors. Errors increase as the codebook size decreases. Two com algorithms are commonly used for vector quantization: the K-means algorithm and the binary split algorithm**.**

In the K-means algorithm, a set of L training vectors can be clustered into M (<L) codebook vectors, as follows:

- *Initialization*—Arbitrarily choose M vectors as the initial set of codewords in the codebook.
- *Nearest neighbour search*—For each training vector, find the codeword in the current codebook that is closest and assign that vector to the corresponding cell.

- *Centroid update*—Update the codeword in each cell using the centroid of the training vectors assigned to that cell.
- *Iteration*—Repeat the above two steps until the average distance falls below a preset threshold.

Our implementation uses the binary split algorithm, which is more efficient than the K-means algorithm because it builds the codebook in stages as described in the following steps:

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

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

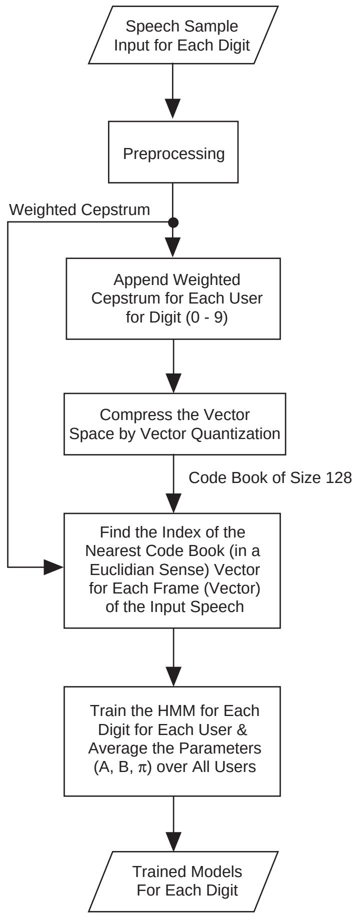
   $y_n^- = y_n(1 - e)$

   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 obtained.

Training

The system trains the HMM for each digit in the vocabulary. The same weighted cepstrum matrices for various users and digits are compared with the codebook and their corresponding nearest codebook vector indices is sent to the Baum-Welch algorithm to train a model for the input index sequence. The codebook index is the observation vector for the HMM model. The Baum-Welch model is an iterative procedure and our system limits the iterations to 20. After training, we have three models for each digit that correspond to the three users in our vocabulary set. We find the average of the A, B, and n matrices over the users to generalize the models. Figure 4 shows the HMM training flow chart.

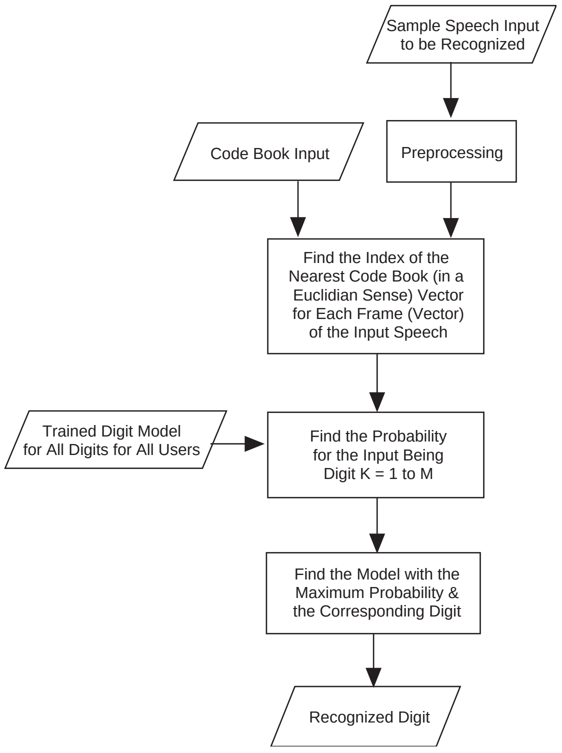
Figure 4. HMM Training Flow Chart


Speech Sample Input for Each Digit
Preprocessing
Weighted Cepstrum
Append Weighted Cepstrum for Each User for Digit (0 - 9)
Compress the Vector Space by Vector Quantization
Code Book of Size 128
Find the Index of the Nearest Code Book (in a Euclidian Sense) Vector for Each Frame (Vector) of the Input Speech
Train the HMM for Each Digit for Each User & Average the Parameters (A, B, π) over All Users
Trained Models For Each Digit

Recognition

The system recognizes the spoken digit using a maximum likehood estimate, i.e., a Viterbi decoder. The input speech sample is preprocessed to extract the feature vector. Then, the nearest codebook vector index for each frame is sent to the digit models. The system chooses the model that has the maximum probability of a match. Figure 5 shows the recognition flow chart.

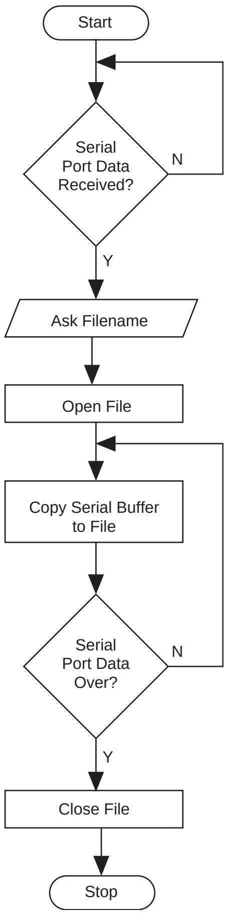
Figure 5. Recognition Flow Chart


Sample Speech Input to be Recognized
Code Book Input
Preprocessing
Find the Index of the Nearest Code Book (in a Euclidian Sense) Vector for Each Frame (Vector) of the Input Speech
Trained Digit Model for All Digits for All Users
Find the Probability for the Input Being Digit K = 1 to M
Find the Model with the Maximum Probability & the Corresponding Digit
Recognized Digit

PC Software for Storage

Data goes to the PC for storage as shown in Figure 6.

Figure 6. Text Storage on PC Flow Chart



Design Features

Our system has the following features:

- It is a standalone system for converting speech to text without using a PC for recognition.
- It optimally uses resources with custom instructions implemented as custom hardware blocks and peripherals.
- Runtime data acquisition and recognition makes the system suitable for practical use in ATMs, telephones, etc.
- Storing the output text to a PC aids logging and archiving.
- It uses a variety of features and components available for Nios II-based development, such as PIO, UART, and RS-232 communication.

In the future, we would like to implement a TCP/IP stack so that we can provide a standalone speech-to-text conversion engine/server on the network.

Performance Parameters

The important performance parameters for our project are:

- *Recognition accuracy*—The most important parameter in any recognition system is its accuracy. A recognition accuracy of 100% for all digits, independent of the speaker, is the goal.
- *Recognition speed*—If the system takes a long time to recognize the speech, users would become restless and the system loses its significance. A recognition time of less than 1 second is required for the project.
- *Resource usage*—Because the system is implemented in an FPGA, the resources required to implement the project are important. The FPGA has limited resources, such as logic blocks, memory cells, phase-locked loops (PLLs), hardware multipliers, etc. that must be used efficiently. Our project had to fit into a Cyclone II EP2C35F672C6 device, which has 33,216 logic elements, 82,016 memory bits, 475 pins, 4 PLLs, and 70 embedded multiplier elements.

Although we tried to maximize the system performance, there were some bottlenecks. The following sections describe the actual system performance.

Recognition Accuracy

We studied the recognition accuracy for four cases:

- *Case 1*—Used samples from speakers 1, 2, and 3 that were also used for training.
- *Case 2*—Used samples from speakers 1, 2, and 3 that were **not** used for training.
- *Case 3*—Used samples of 3 untrained speakers (4, 5, and 6) whose voices were not used for training.
- *Case 4*—Used samples from speakers 1, 2, 3, 4, 5, and 6 to obtain the overall recognition performance.

The results are summarized in the following table.

Recognition Accuracy Test Results

Digit	Recognition Percentage			
	Case 1	Case 2	Case 3	Case 4
0	100	100	85.7	95.2
1	100	85.7	71.4	85.7
2	100	85.7	57.1	80.9
3	100	85.7	57.1	80.9
4	100	100	71.4	90.5
5	100	100	71.4	90.5
6	100	85.7	57.1	80.9
7	100	100	71.4	90.5
8	100	100	57.1	85.7
9	100	100	71.4	90.5

Our system had an overall recognition accuracy of more than 85%.

Recognition Speed

The recognition time of the system was 3.54 seconds for a vocabulary of ten digits.

Resource Usage

The Cyclone II FPGA on the DE2 board provided many more resources than we needed to implement our system: we used fewer than 20% of the available resources. The following table summarizes the resource usage of the **vitercopy** custom instruction and the overall system. Refer to the Screenshots section for more information.

vitercopy Custom Instruction Resource Usage

Resource	vitercopy Custom Instruction Usage	Complete System Usage
Total combinational functions	389	3,650
Logic element usage by number of LUT inputs		
4 input functions	85	1,809
3 input functions	168	1,146
<=2 input functions	136	695
Combinational cells for routing	0	1
Logic elements by mode		
Normal mode	222	3,147
Arithmetic mode	167	503
Total registers	264	2,699
I/O pins	109	47
Total memory bits	10,848	82,016
Maximum fan-out node	`clk`	4
Maximum fan-out	281	1
Total fan-out	2593	CLOCK_50
Average fan-out	3.23	2684

Conclusion

Our project implements a speech-to-text system in which a Nios II processor performs the recognition process. The Nios II processor is very useful for implementing the application logic using C code. We implemented computationally intensive tasks as custom blocks and the SOPC Builder helped us integrate these blocks into the Nios II system.

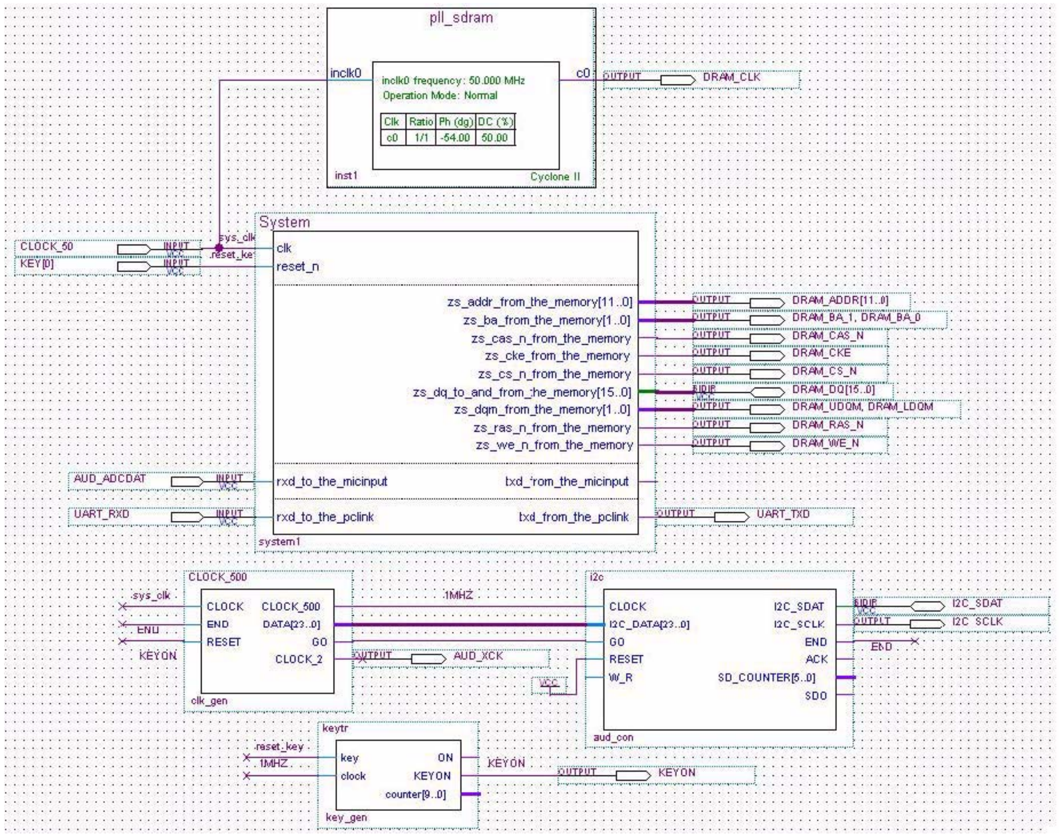
The Cyclone II FPGA, DE2 board, Quartus II software, Nios II IDE, and SOPC Builder provided a suitable platform for implementation our embedded system, which required both hardware and software designs.

Our speech-to-text system has a recognition time of about 3 seconds and a recognition accuracy of ~85%.

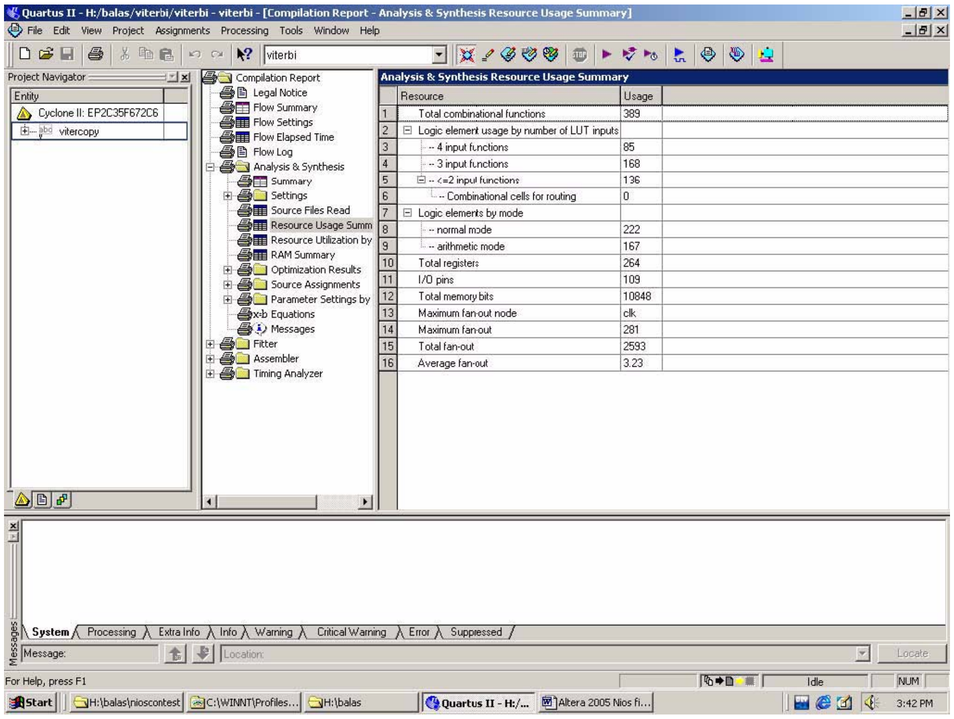
Screenshots

The following figures provide additional information.

The Final Project Block Description File



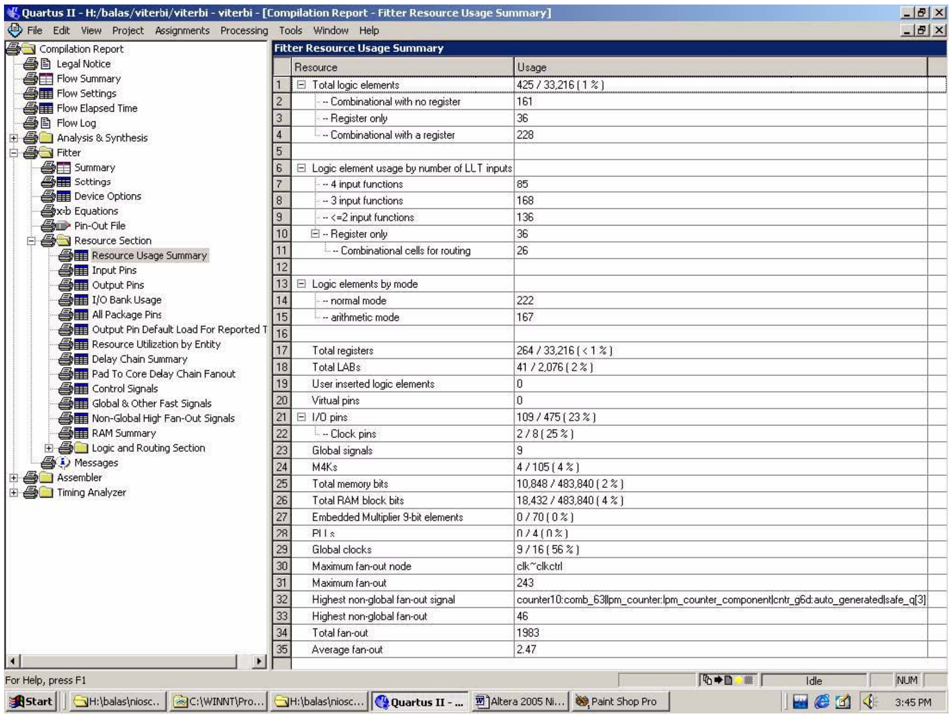
Synthesis Report of Custom Hardware Block



Analysis & Synthesis Resource Usage Summary

	Resource	Usage
1	Total combinational functions	389
2	Logic element usage by number of LUT inputs	
3	-- 4 input functions	85
4	-- 3 input functions	168
5	-- <=2 input functions	136
6	-- Combinational cells for routing	0
7	Logic elements by mode	
8	-- normal mode	222
9	-- arithmetic mode	167
10	Total registers	264
11	I/O pins	109
12	Total memory bits	10848
13	Maximum fan-out node	clk
14	Maximum fan-out	281
15	Total fan-out	2593
16	Average fan-out	3.23

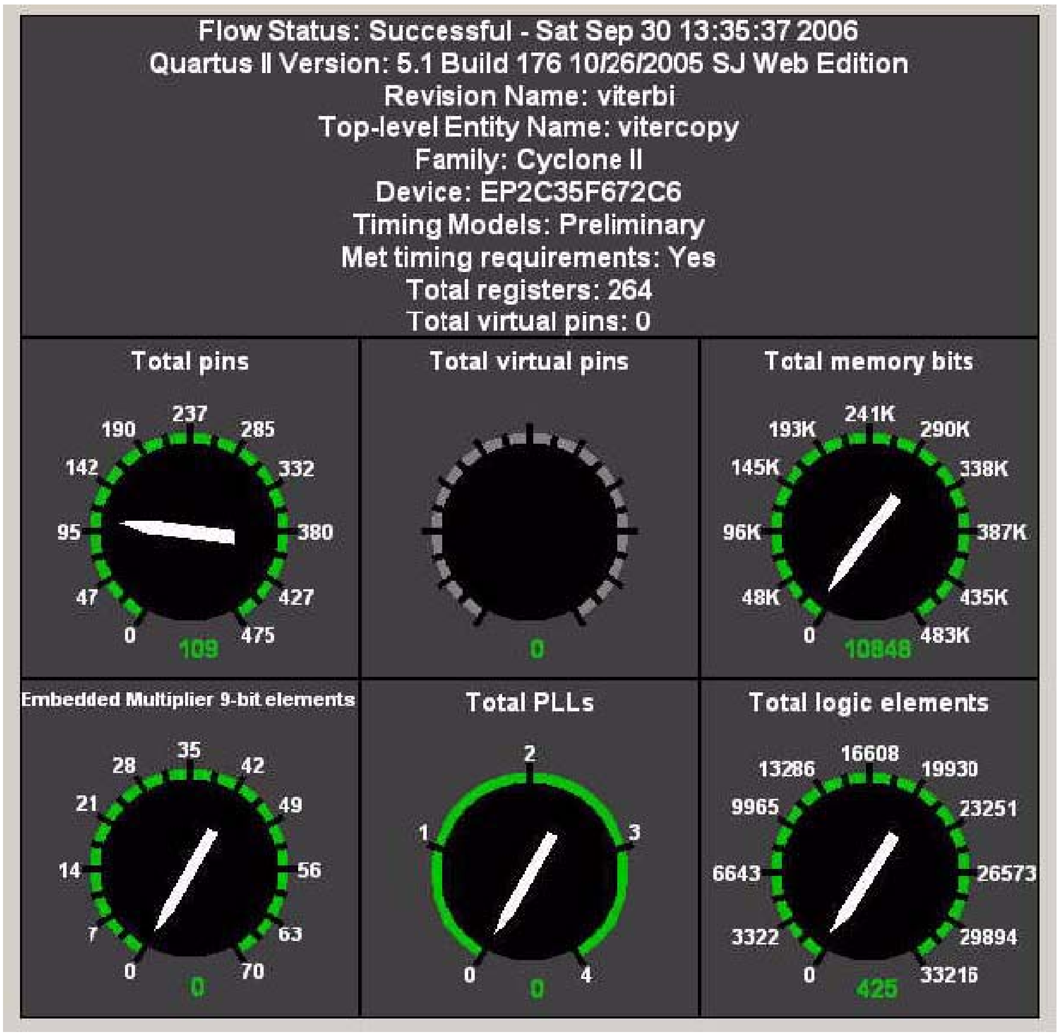
Fitter Report of Custom Hardware Block



Fitter Resource Usage Summary

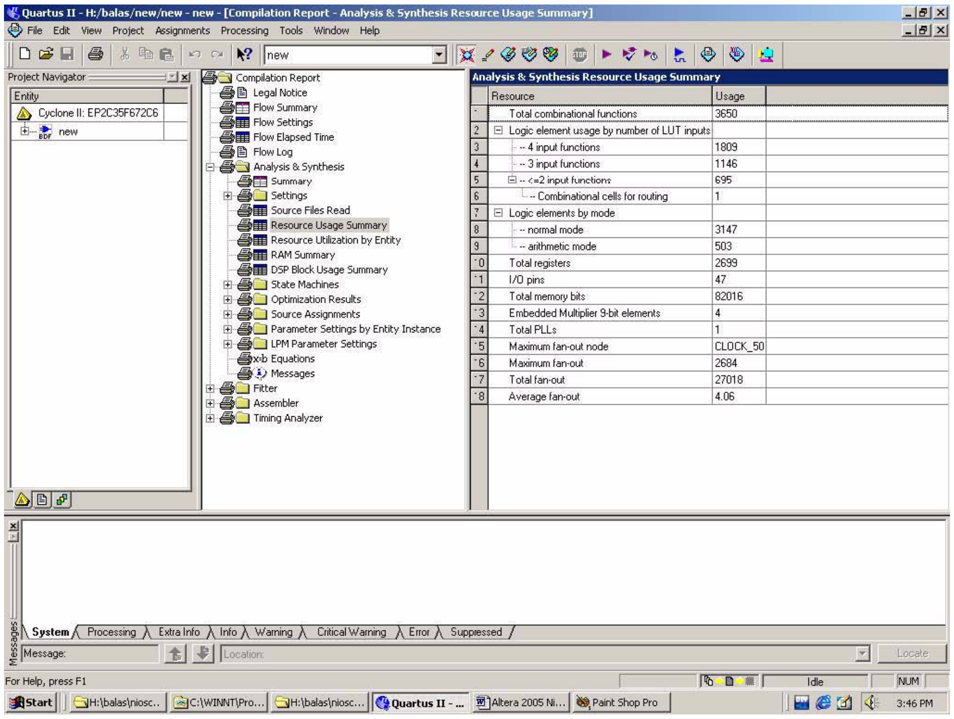
	Resource	Usage
1	Total logic elements	425 / 33,216 (1 %)
2	-- Combinational with no register	161
3	-- Register only	36
4	-- Combinational with a register	228
5		
6	Logic element usage by number of LUT inputs	
7	-- 4 input functions	85
8	-- 3 input functions	168
9	-- <=2 input functions	136
10	-- Register only	36
11	-- Combinational cells for routing	26
12		
13	Logic elements by mode	
14	-- normal mode	222
15	-- arithmetic mode	167
16		
17	Total registers	264 / 33,216 (< 1 %)
18	Total LABs	41 / 2,076 (2 %)
19	User inserted logic elements	0
20	Virtual pins	0
21	I/O pins	109 / 475 (23 %)
22	-- Clock pins	2 / 8 (25 %)
23	Global signals	9
24	M4Ks	4 / 105 (4 %)
25	Total memory bits	10,848 / 483,840 (2 %)
26	Total RAM block bits	18,432 / 483,840 (4 %)
27	Embedded Multiplier 9-bit elements	0 / 70 (0 %)
28	PLLs	0 / 4 (0 %)
29	Global clocks	9 / 16 (56 %)
30	Maximum fan-out node	clk~clkctrl
31	Maximum fan-out	243
32	Highest non-global fan-out signal	counter10:comb_63|lpm_counter:lpm_counter_component|cntr_g6d:auto_generated|safe_q[3]
33	Highest non-global fan-out	46
34	Total fan-out	1983
35	Average fan-out	2.47

Compilation Report of Custom Hardware Block



Flow Status: Successful - Sat Sep 30 13:35:37 2006
Quartus II Version: 5.1 Build 176 10/26/2005 SJ Web Edition
Revision Name: viterbi
Top-level Entity Name: vitercopy
Family: Cyclone II
Device: EP2C35F672C6
Timing Models: Preliminary
Met timing requirements: Yes
Total registers: 264
Total virtual pins: 0

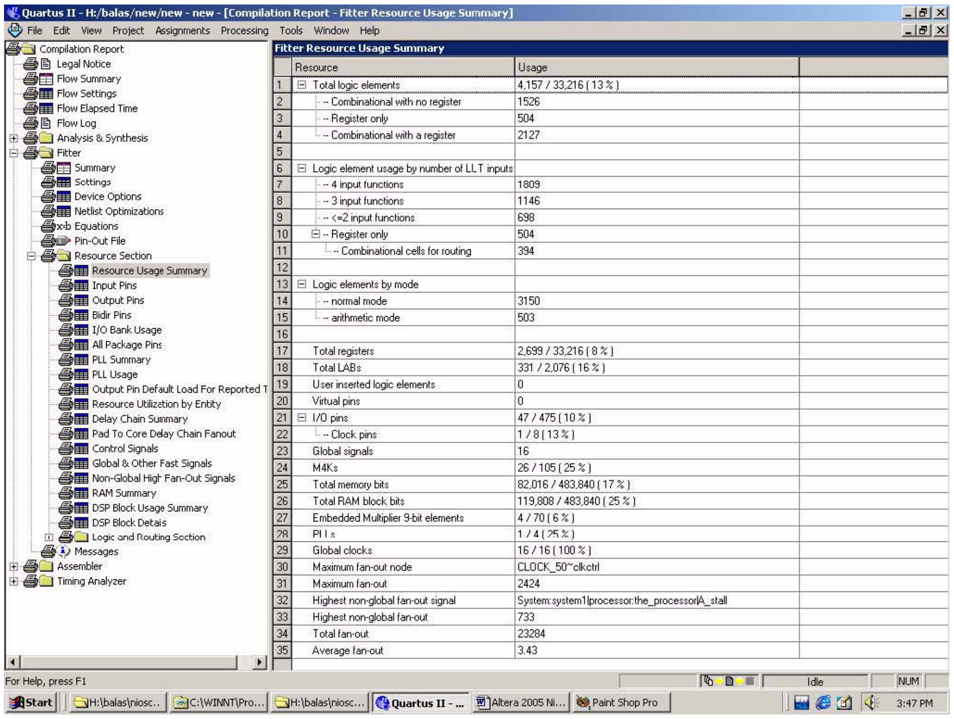
Synthesis Report of Complete Design


Quartus II - H:/balas/new/new - new - [Compilation Report - Analysis & Synthesis Resource Usage Summary]

Analysis & Synthesis Resource Usage Summary

	Resource	Usage
1	Total combinational functions	3650
2	Logic element usage by number of LUT inputs	
3	-- 4 input functions	1809
4	-- 3 input functions	1146
5	-- <=2 input functions	695
6	-- Combinational cells for routing	1
7	Logic elements by mode	
8	-- normal mode	3147
9	-- arithmetic mode	503
10	Total registers	2699
11	I/O pins	47
12	Total memory bits	82016
13	Embedded Multiplier 9-bit elements	4
14	Total PLLs	1
15	Maximum fan-out node	CLOCK_50
16	Maximum fan-out	2684
17	Total fan-out	27018
18	Average fan-out	4.06

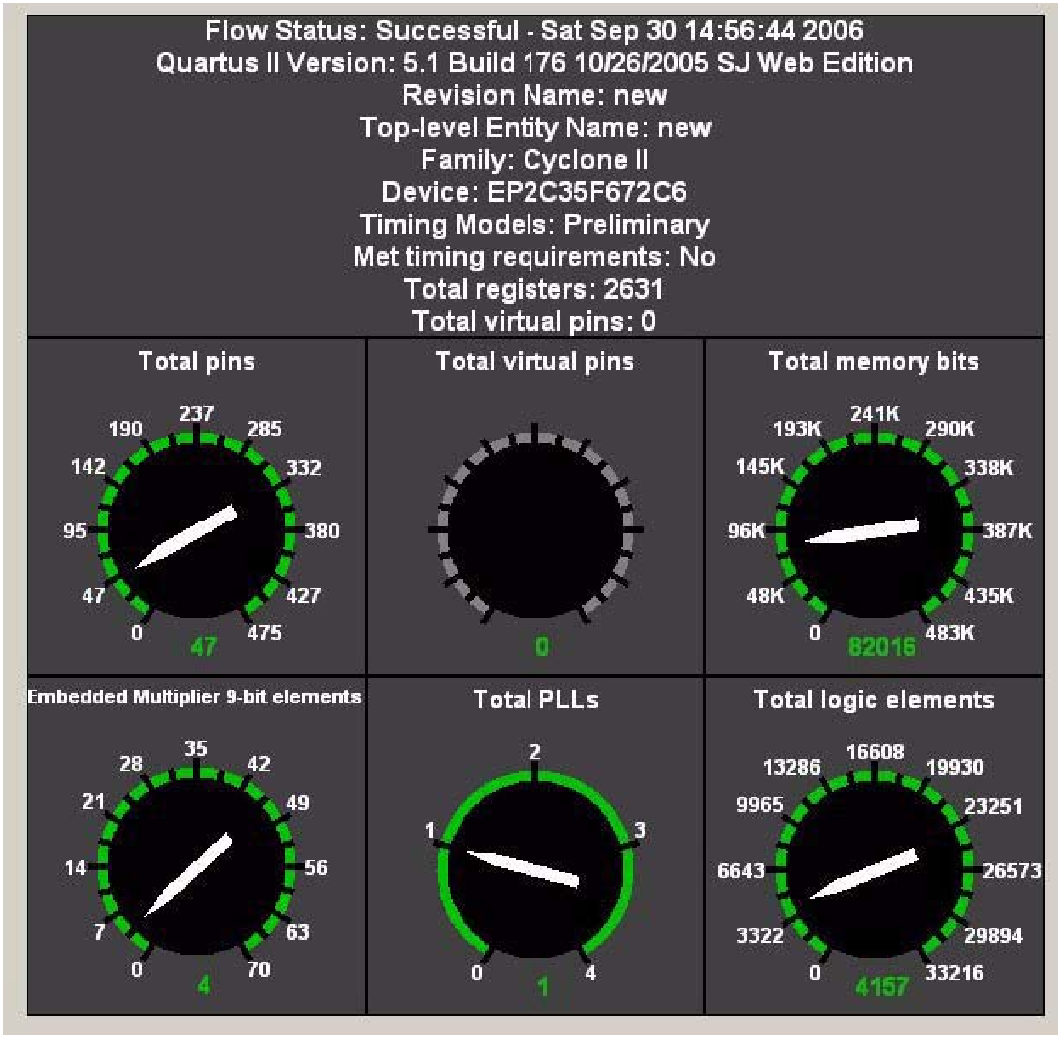
Fitter Report of Complete Design


Quartus II - H:/balas/new/new - new - [Compilation Report - Fitter Resource Usage Summary]

Fitter Resource Usage Summary

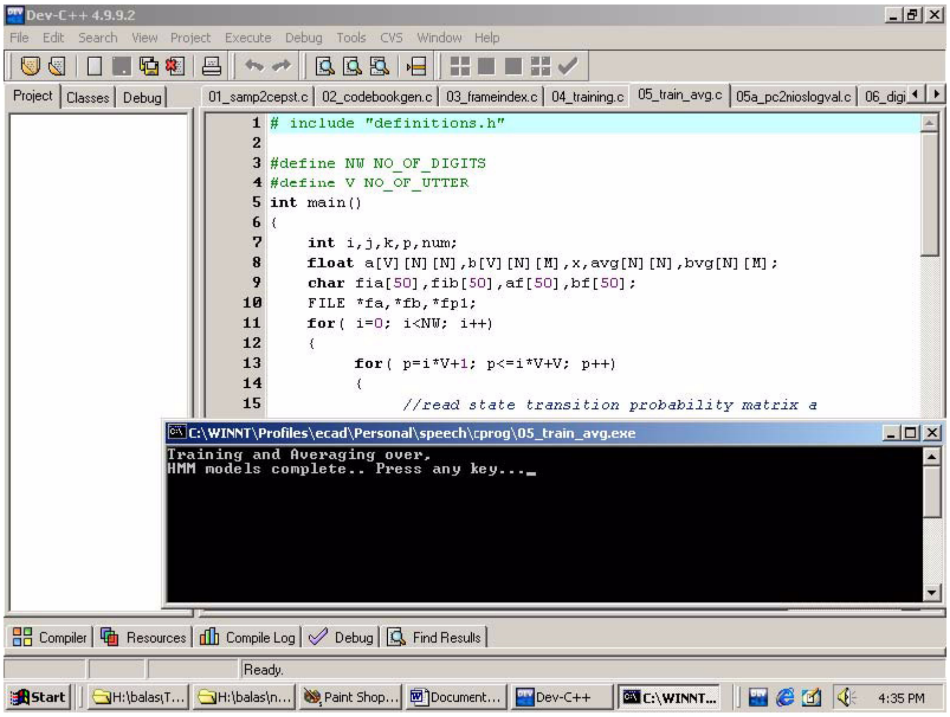
	Resource	Usage
1	Total logic elements	4,157 / 33,216 (13 %)
2	-- Combinational with no register	1526
3	-- Register only	504
4	-- Combinational with a register	2127
5		
6	Logic element usage by number of LUT inputs	
7	-- 4 input functions	1809
8	-- 3 input functions	1146
9	-- <=2 input functions	698
10	-- Register only	504
11	-- Combinational cells for routing	394
12		
13	Logic elements by mode	
14	-- normal mode	3150
15	-- arithmetic mode	503
16		
17	Total registers	2,699 / 33,216 (8 %)
18	Total LABs	331 / 2,076 (16 %)
19	User inserted logic elements	0
20	Virtual pins	0
21	I/O pins	47 / 475 (10 %)
22	-- Clock pins	1 / 8 (13 %)
23	Global signals	16
24	M4Ks	26 / 105 (25 %)
25	Total memory bits	82,016 / 483,840 (17 %)
26	Total RAM block bits	119,808 / 483,840 (25 %)
27	Embedded Multiplier 9-bit elements	4 / 70 (6 %)
28	PLLs	1 / 4 (25 %)
29	Global clocks	16 / 16 (100 %)
30	Maximum fan-out node	CLOCK_50~clkctrl
31	Maximum fan-out	2424
32	Highest non-global fan-out signal	System:system1|processor:the_processor|A_stall
33	Highest non-global fan-out	733
34	Total fan-out	23284
35	Average fan-out	3.43

Compilation Report of Complete Design



Flow Status: Successful - Sat Sep 30 14:56:44 2006
Quartus II Version: 5.1 Build 176 10/26/2005 SJ Web Edition
Revision Name: new
Top-level Entity Name: new
Family: Cyclone II
Device: EP2C35F672C6
Timing Models: Preliminary
Met timing requirements: No
Total registers: 2631
Total virtual pins: 0

Training Software and Output

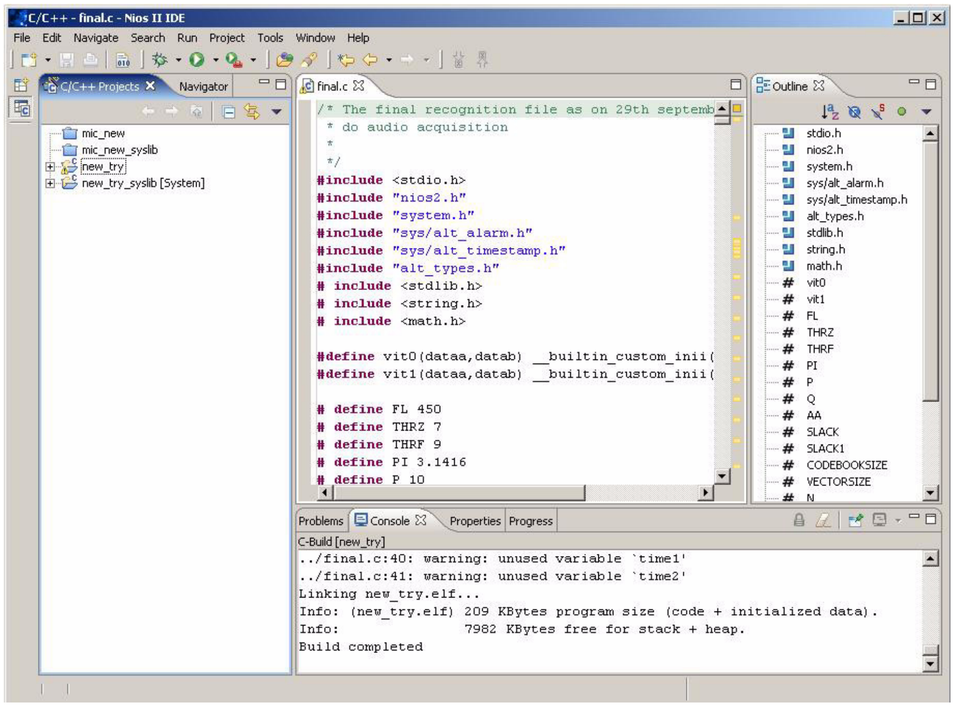

Dev-C++ 4.9.9.2
File Edit Search View Project Execute Debug Tools CVS Window Help
Project Classes Debug
01_samp2cepst.c 02_codebookgen.c 03_frameindex.c 04_training.c 05_train_avg.c 05a_pc2nioslogval.c 06_digi
C:\WINNT\Profiles\ecad\Personal\speech\cprog\05_train_avg.exe
Compiler Resources Compile Log Debug Find Results
Ready.
Start H:\balas\T... H:\balas\n... Paint Shop... Document... Dev-C++ C:\WINNT... 4:35 PM

```
# include "definitions.h"

#define NW NO_OF_DIGITS
#define V NO_OF_UTTER
int main()
{
    int i,j,k,p,num;
    float a[V][N][N],b[V][N][M],x,avg[N][N],bvg[N][M];
    char fia[50],fib[50],af[50],bf[50];
    FILE *fa,*fb,*fp1;
    for( i=0; i<NW; i++)
    {
        for( p=i*V+1; p<=i*V+V; p++)
        {
            //read state transition probability matrix a
```

```
Training and Averaging over.
HMM models complete.. Press any key..._
```

Nios II Program Code Memory Usage



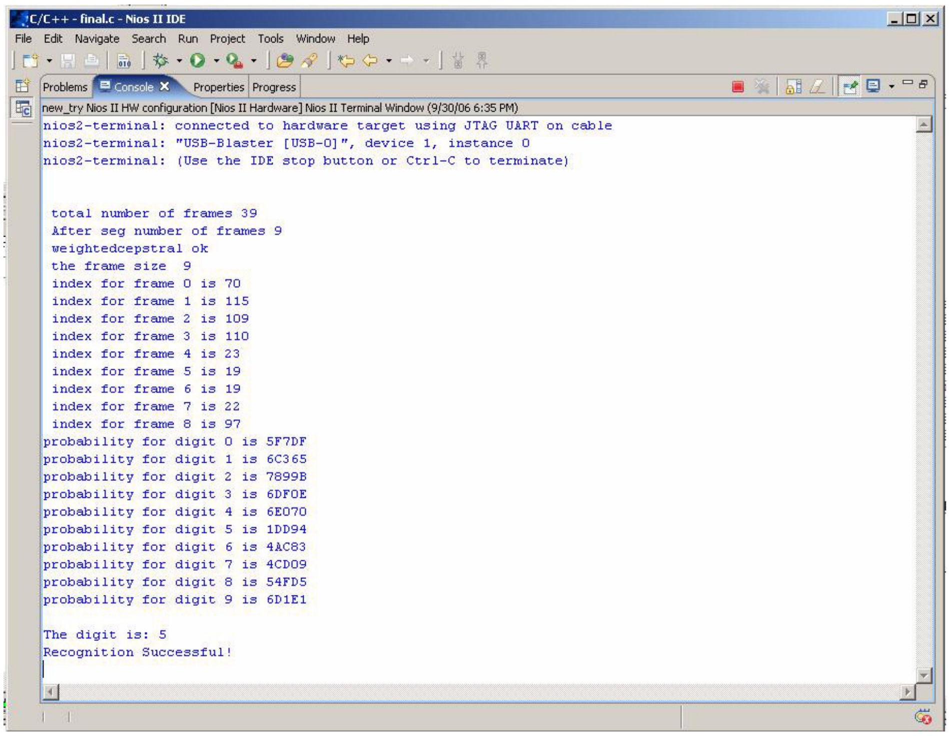
```
/* The final recognition file as on 29th septemb
 * do audio acquisition
 *
 */
#include <stdio.h>
#include "nios2.h"
#include "system.h"
#include "sys/alt_alarm.h"
#include "sys/alt_timestamp.h"
#include "alt_types.h"
# include <stdlib.h>
# include <string.h>
# include <math.h>

#define vit0(dataa,datab)  __builtin_custom_inii(
#define vit1(dataa,datab)  __builtin_custom_inii(

# define FL 450
# define THRZ 7
# define THRF 9
# define PI 3.1416
# define P 10
```

```
C-Build [new_try]
../final.c:40: warning: unused variable `time1'
../final.c:41: warning: unused variable `time2'
Linking new_try.elf...
Info: (new_try.elf) 209 KBytes program size (code + initialized data).
Info:               7982 KBytes free for stack + heap.
Build completed
```

Sample Output for Digit 5


C/C++ - final.c - Nios II IDE
File Edit Navigate Search Run Project Tools Window Help
Problems Console Properties Progress

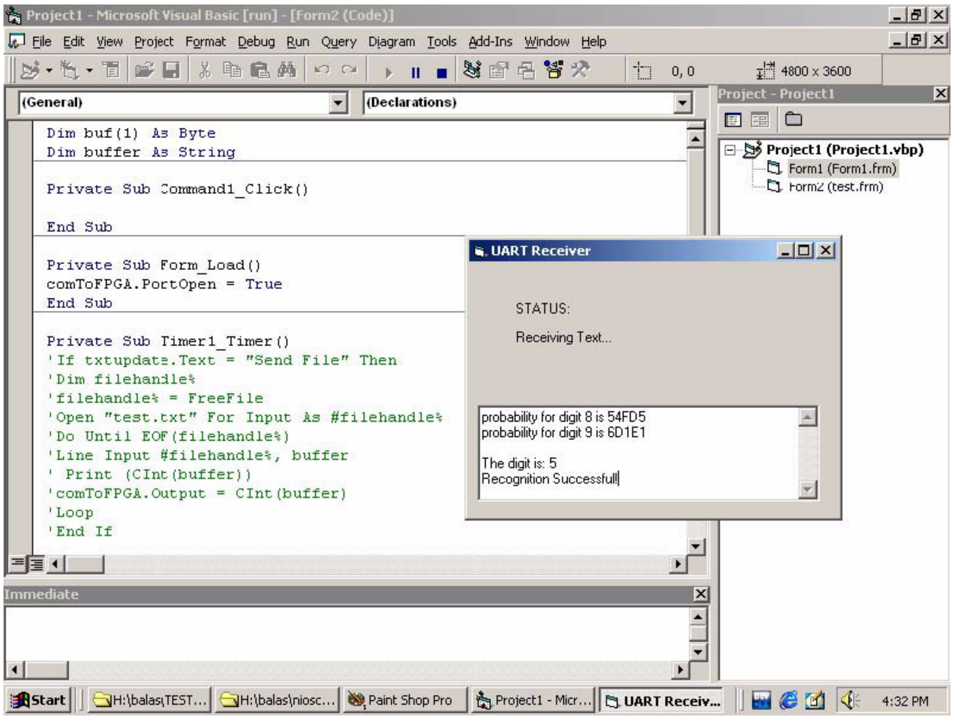
new_try Nios II HW configuration [Nios II Hardware] Nios II Terminal Window (9/30/06 6:35 PM)

```
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)


 total number of frames 39
 After seg number of frames 9
 weightedcepstral ok
 the frame size  9
 index for frame 0 is 70
 index for frame 1 is 115
 index for frame 2 is 109
 index for frame 3 is 110
 index for frame 4 is 23
 index for frame 5 is 19
 index for frame 6 is 19
 index for frame 7 is 22
 index for frame 8 is 97
probability for digit 0 is 5F7DF
probability for digit 1 is 6C365
probability for digit 2 is 7899B
probability for digit 3 is 6DF0E
probability for digit 4 is 6E070
probability for digit 5 is 1DD94
probability for digit 6 is 4AC83
probability for digit 7 is 4CD09
probability for digit 8 is 54FD5
probability for digit 9 is 6D1E1

The digit is: 5
Recognition Successful!
```

VB Program for PC Storage



```
Dim buf(1) As Byte
Dim buffer As String

Private Sub Command1_Click()

End Sub

Private Sub Form_Load()
comToFPGA.PortOpen = True
End Sub

Private Sub Timer1_Timer()
'If txtupdate.Text = "Send File" Then
'Dim filehandle%
'filehandle% = FreeFile
'Open "test.txt" For Input As #filehandle%
'Do Until EOF(filehandle%)
'Line Input #filehandle%, buffer
'  Print (CInt(buffer))
'comToFPGA.Output = CInt(buffer)
'Loop
'End If
```

STATUS:

Receiving Text...

```
probability for digit 8 is 54FD5
probability for digit 9 is 6D1E1

The digit is: 5
Recognition Successfull
```

References

Garg, Mohit. *Linear Prediction Algorithms*. Indian Institute of Technology, Bombay, India, Apr 2003.

Li, Gongjun and Taiyi Huang. *An Improved Training Algorithm in Hmm-Based Speech Recognition. National Laboratory of Pattern Recognition*. Chinese Academy of Sciences, Beijing.

Quatieri, T.F. *Discrete-time Speech Signal Processing*. Prentice Hall PTR, 2001.

Rabiner, Lawrence and Biing-Hwang Juang. *Fundamentals of Speech Recognition*. Prentice Hall PTR, 1993.

Rabiner, Lawrence. *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*. Proceedings of the IEEE, vol 77, issue 2, Feb 1989 (257-286).

Turin, William. *Unidirectional and Parallel Baum–Welch Algorithms*. IEEE Transactions on Speech and Audio Processing. vol 6, issue 6. Nov 1998 (516-523).

Altera Nios II documentation.

Acknowledgements

We are very grateful to our project instructor and guide, Dr. B.Venkataramani, Professor, Department of ECE, NIT, Trichy for his valuable support and ideas from start to finish.

We also want to thank Ms. Amutha, Research Assistant, Department of ECE, NIT, Trichy for always being there with us to help in the lab.

We express our gratitude to all the students, scholars, and faculty who work around-the-clock in our VLSI lab for their suggestions, help, and comments.

We also thank all peer Nios II developers from **niosforum.com**, other universities, and worldwide for their valuable help and support.

Finally, we thank Altera for providing with tools for the project, including the FPGA, software, board, documentation about this contest, and a platform to showcase our designs.