TEXAS INSTRUMENTS

Implementing Speech-Recognition Algorithms on the TMS320C2xx Platform

Application Report

1998 *Digital Signal Processing Solutions*

Printed in U.S.A., April 1998

SPRA178

Implementing Speech-Recognition Algorithms on the TMS320C2xx Platform

Lim Hong Swee
Texas Instruments Singapore (P&E) Ltd.

SPRA178
April 1998

Contents

List of Figures

List of Examples

Implementing Speech-Recognition Algorithms on the TMS320C2xx Platform

ABSTRACT

This application report presents the hidden-Markov-model-based speech-recognition system using a 16-bit fixed-point digital signal processor (DSP) from Texas Instruments (TI™)**1**. It addresses the problems that have been encountered and the compromises that have been made to implement the hidden-Markov-model-based isolated-word recognizer using the TI TMS320C2xx DSP. This report addresses fixed-point representation issues and techniques used to prevent overflows. All recognizer functions, including signal processing, model evaluation, and control, are performed by a single TMS320C2xx DSP.

1 Speech-Recognition Algorithm

Hidden-Markov models (HMMs) are popular statistical models used to implement speech-recognition technologies.[1] The time variances in the spoken language are modeled as Markov processes with discrete state spaces. Each state produces speech observations according to the probability distribution characteristics of that state. The speech observations can take on a discrete or a continuous value. In either case, the speech observations represent a fixed time duration (i.e., a frame). The states are not directly observable, which is why the model is called the hidden-Markov model.

The original minimal HMM (min_HMM) algorithm was implemented on a floating-point C language program platform running under the UNIX operating system.[2] As a result of changes in the technology, a project has been called for to transfer this algorithm from floating-point C language to TMS320C2xx assembly language (ASM). The primary objectives of this project are to have the algorithm running in real time and to have assembly code that is user-supportable.

Since HMM is an extremely complicated algorithm, it is difficult, if not impossible, to trace the algorithm directly from the assembly-code version to the C-language version. To ensure user-supportability, the assembly-language code was written in a manner that is similar to the C-language code. The primary advantage of the similarity is that any modifications made to the algorithm can be tested in the C-language platform, before any work is done to port the algorithm to the TMS320C2xx.

1. TI is a trademark of Texas Instruments Incorporated.

The speech-recognition algorithm, as shown in Figure 1, contains two fundamental parts, which are the acoustic front end and the search algorithm itself.[3] The acoustic front end is the process of converting sequences of raw-speech data to observation vectors, which represent events existing in a probability space. The search algorithm then finds the most probable sequence of these events while operating under a set of syntactic constraints.

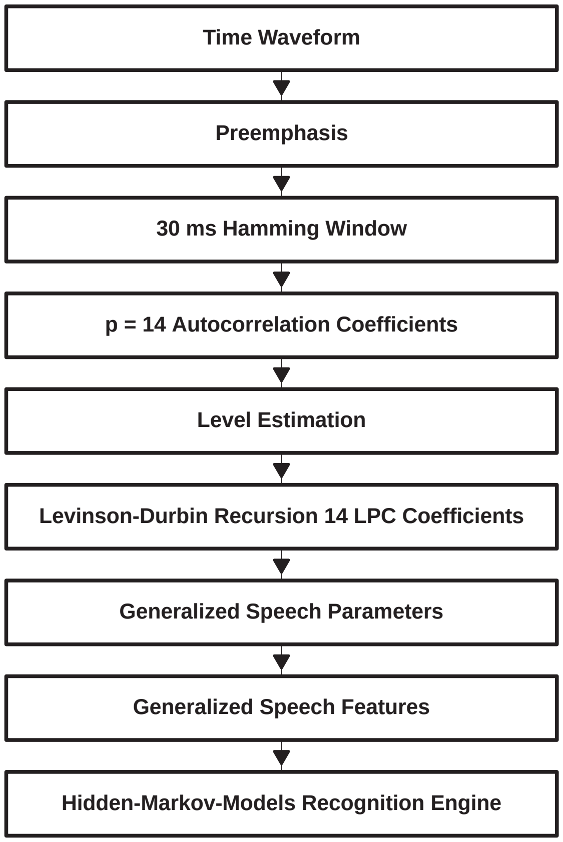
Time Waveform
Preemphasis
30 ms Hamming Window
p = 14 Autocorrelation Coefficients
Level Estimation
Levinson-Durbin Recursion 14 LPC Coefficients
Generalized Speech Parameters
Generalized Speech Features
Hidden-Markov-Models Recognition Engine

Figure 1. Block Diagram of the Speech-Recognition System

1.1 The Acoustic Front End

Figure 2 shows an overview of the acoustic front end. The first stage in the process converts speech to a digital representation by sampling speech at the rate of 8 kHz/second. Spectral shaping is performed to allow the more important frequency components to be emphasized in the speech signal by using a finite impulse response (FIR) filter. Normally, this is a 1-coefficient digital filter, which is also known as an audio preemphasis filter.

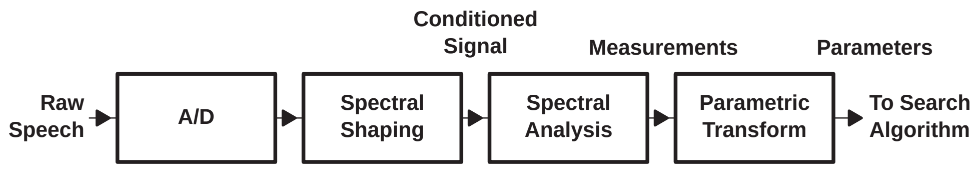

Figure 2. An Overview of Acoustic Front-End Signal Processing

The sampled signal is split into frames; each frame represents a finite time slice. Each slice is short enough to allow a speech wave segment to be considered stationary within the frame. A technique called *windowing* is used to achieve this result. Windowing allows the portion of the sample that is closest to the center of a window to be more heavily weighted than the parts of the sample that are further away from the center of the window. This weighting technique minimizes spectral leakage. Another window function, the hamming window, is an error-trapping routine that is used to detect data errors inside individual windows.

By stepping a data window along a sampled speech signal, a sequence of frames that represent the whole speech wave sequence is obtained. Typical window lengths are 30 ms (where the length is a representation of a forward time-slice sequence), but the frames can be stepped at a shorter elapsed time interval, such as 20 ms, so that the frames overlap.

Individual windowed speech frames are then processed further to capture the characteristic information in a more compact form. Texas Instruments uses linear predictive coding (LPC) to perform speech spectral analysis.[4,5] LPC conversions use a pitch-asynchronous automatic-correlation method with a frame rate of 20 ms. The gestures generated by the acoustic front end of the recognizer are obtained by orthogonalization of the 14-mel spaced filter-bank outputs. These values, in addition to the corresponding 14 difference values, the actual speech level, the relative speech level, the transitional characteristics of the speech, and the energy difference values, make up a 32-element vector called the generalized speech parameters (GSPs). This 32-element vector is then multiplied by a linear transform (LT) value [6] to provide a 10-element vector named the generalized speech features (GSFs), which is used in the final acoustic-distance calculation.

1.2 Search Algorithm

Figure 3 shows an HMM collection of state-connected transitions. It begins in a designated initial state. In each discrete time-interval step, this transition is taken to a new state. One output symbol is generated in each state, and the user has a choice between the new transition or the symbol selected for that state. While the choice is made randomly, it is governed by probability distributions.

HMMs have a variety of applications. When an HMM is applied to speech recognition, the states are interpreted as acoustical models, indicating which sounds are most likely to be heard during the corresponding segments of speech being evaluated. The transitions also provide a set of temporal constraints that indicate how the states follow each other in sequence. Since speech is always going forward in time, the transitions in a speech application must always go forward or be made to make a self-loop, which allows each state to have an arbitrary duration.

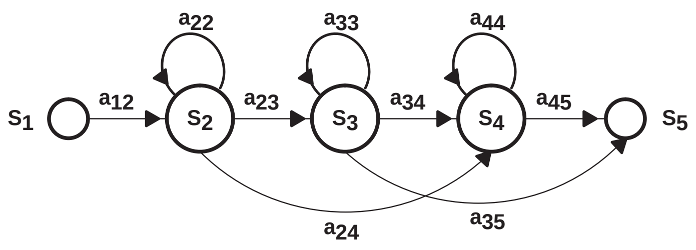

Figure 3. A Simple HMM With Three States

If, for example, a speech utterance is represented by a series of T-parameter speech frames that are referred to as observation vectors *O*, then

$O = o_1, o_2, \ldots o$

The object of the pattern-matching stage is to determine the most likely sequence of words that produce the desired observational sequence. An observation vector is generated according to a probability-density function associated with that state. The probabilities are also associated with the transition between states; therefore, the likelihood of generating *O* using any sequence of states can be computed.

The purpose of the speech-recognition algorithm is to take an observation sequence *O* and compute which words are most likely to have produced this particular set of observation vectors. If the observation sequence is known to represent only a single word from a limited set of possible words, the task is to compute

$$\max p(\omega_i|O)$$

which is the probability of word ω_i, given the observation vector *O*.

A forward algorithm is used for min_HMM. To perform an isolated-word recognition, it is necessary to evaluate the probabilities that a given min_HMM-word model has produced a given observation sequence. Sequences have a score assigned to each word model, based on the probability of accuracy. The models can be compared to that score for each word model, and the model with the highest score can be chosen.

2 Fixed-Point Implementation of a min_HMM

In any high-level language (HLL) implementation, the acoustic front end and search algorithms are implemented using floating-point (FP) arithmetic. However, when these search algorithms are being implemented using the TMS320C2xx DSP, only a finite amount of arithmetic precision is available. This limitation causes some deviation from the original design, and a considerable amount of effort is required to ensure that the digital hardware implementation is as precise as the HLL implementation. This section discusses some of the design considerations for implementing the algorithm in the final digital hardware platform.

When given the observation sequence *O*, the problem is to find the single best state sequence that would explain the observed data. The Viterbi algorithm, which is shown in equation (1), was used to find the best state sequence.

$$pathscore_t(j) = \max\left[pathscore_{t-1}(i) + ln\left(prob_{ij}\right)\right] + d_j \; 1 \le i \le n \tag{1}$$

d_j	= probability that state *j* matches the observation *O* at time *t*
$prob_{ij}$	= the transition probability from state *i* to state *j*
n	= the number of states in the model

A linear transformation is designated in such a manner that the resulting features can be approximated by a Gaussian distribution. Hence, *dj* at time *t* reduces to a simple Euclidean distance between the observed vector *O* at time *t* and the reference vector associated with state *j*. Then *dj* can be implemented efficiently by the TMS320C2xx DSP.

A floating-point processor implementation of the HMM algorithm reduces development time but is not necessary. A fixed-point DSP, conversely, does require a thorough understanding of the dynamic range of the HMM variables so a suitable Q-point can be used. However, since a fixed-point DSP is significantly less expensive than a floating-point DSP, using a fixed-point processor allows a more cost-effective implementation of hardware. Fixed-point DSPs are normally an order of magnitude less expensive than floating-point DSPs.

Direct implementation of equation (1) results in a monotonically decreasing pathscore. This pathscore obviously causes data overflows in a fixed-point arithmetic processor. To prevent such overflows, the pathscores are normalized and the best score across all the models at time *i-1* is subtracted from the pathscore at time *i*. By constraining all pathscores to lie between a value of 1 and 0, all bits in a 16-bit word can be used to represent a pathscore. Using this system, the pathscores are normalized to within 1 percent of the unconstrained pathscore values implemented on a floating-point platform.

2.1 Dynamic-Range Determination

To determine the best Q-point for a fixed-point implementation, the dynamic ranges of all the variables involved in the score computation were computed on two sets of files (military alphabet and 10-digit strings). The military alphabet (alpha through zulu) is designed to cover all phonetic event, and therefore the dynamic ranges estimated from this set of data are considered reliable. Ten-digit strings are long utterances that are expected to augment any phenomena that short utterances (military alphabet) cannot show.

Since the linear-transformation matrix (GSP to GSF) does a mean-global subtraction, all the GSFs are expected to have their mean values around 0. All the scores should be less than 0 since the logarithmic probabilities are less than 0. The minimum value of the best score is greater than −1 because −1 is the rejection score value. For all the existing slot scores (including pruned and unpruned), the minimum score is lower than −1. The check point is located before the point at which the pruning occurs. Because only the scores that are better than rejection scores need to be kept, computing the dynamic range of the scores is not necessary. It is known that all scores will be between −1 and 0.

2.2 Q-Point Setting

The probability of a search path is a product of all the transition and observational probabilities along the search path. Because it is more likely that an underflow will occur by multiplying small probabilites (values < −1), the logarithmic probability is used as the score. Because of the logarithmic operation, scores are computed by adding all the logarithmic probabilities together.

Recognition scores are the accumulation of transition scores and observation scores along the search paths. All scores are negative during computation, because logarithm probabilities are negative. Q15 is used for all scores. Transition scores are already-weighted Q15 scores and are simply added to the path score. Observation scores are computed by multiplying the square of the Euclidean distance by a factor (− observation weight). See the euclid.asm program listing in Example 1. If, for example, $\overline{X}$ is the input GSF vector, and $\overline{R}$ is the model frame GSF vector, the observation score is

$$\frac{-1}{250}(\overline{X} - \overline{R})^2 \qquad (2)$$

Both $\overline{X}$ and $\overline{R}$ are 10-dimensional vectors; for precision reasons, they are in Q9 format. For a 16-bit word, Q9 format can represent −64 to +64. From the dynamic range statistics obtained from section 2.1, it should cover the GSF dynamic range. The factor −1/250 is represented in Q22. The product (observations score) is in Q15. For a 16-bit word, Q15 can represent −1 to +0.99997. The scores can fall below −1; however, because −1 is the rejection threshold, all scores that are less than −1 are eliminated and only the scores between −1 and 0 are kept. The architecture of the ’C2xx is optimized to implement multiply and add instructions. Figure 4 shows the architecture of the ’C2xx that affects the way euclid.asm is written.

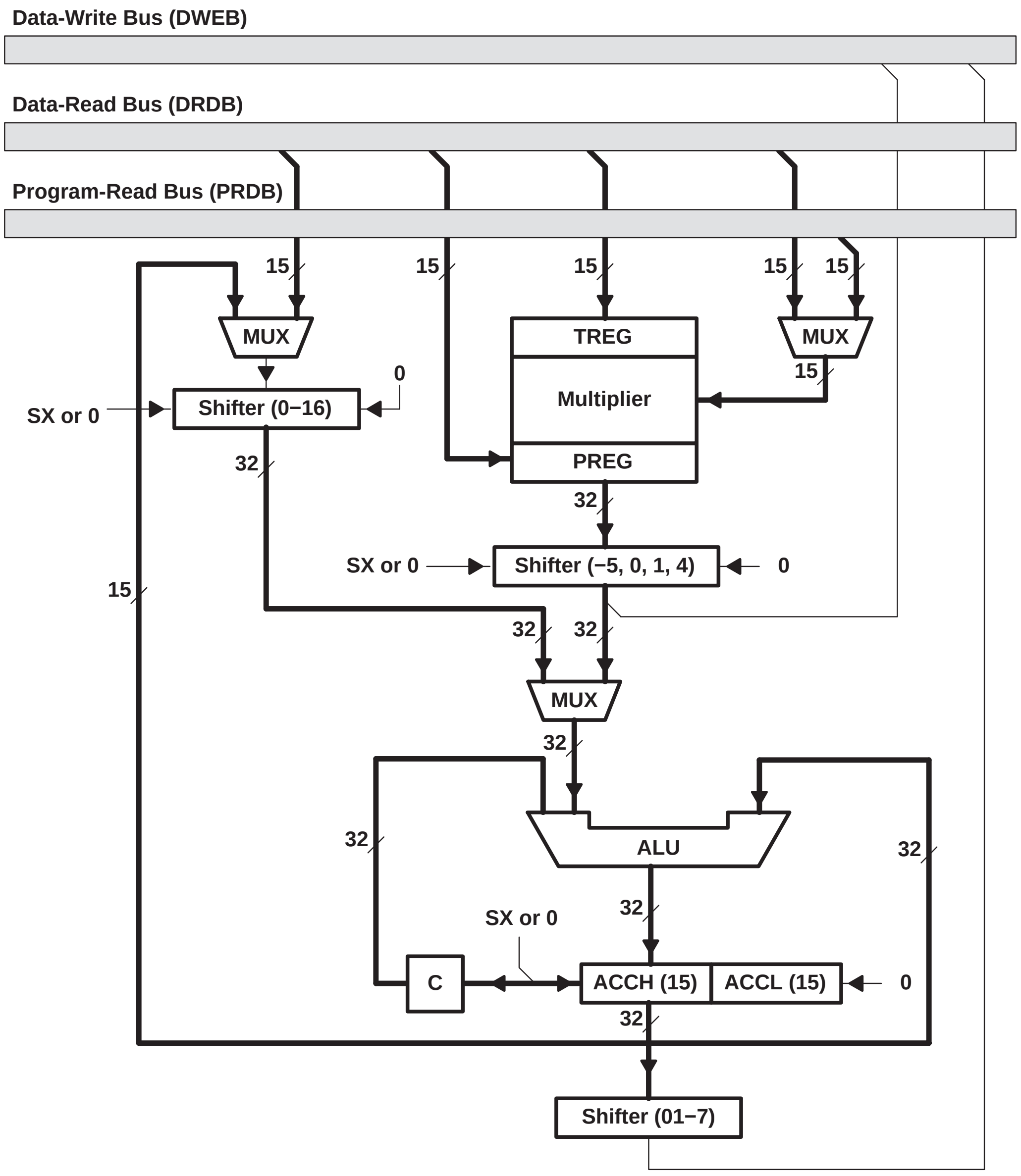

Figure 4. Block Diagram of the ’C2xx Internal Hardware

Example 1. euclid.asm Assembly-Language Program

```
RPOBSW          .equ      16777         ;1/250 in q22
euclid    ldp   #addr2
          sar   AR2,addr2
          lacc  #RPOBSW
          sacl  inv250
                                        ;10        2
          lar   AR1,#(GSF_ORD−1)        ;sum(GSF_1 − GSF_r)
          lar   AR2,#di_buf             ;i=1
          mar   *,AR4
eucl1     lacc  *+,AR3                  ;for(i=10,i>0,i−−)
          sub   *+,AR2                  ;GSF_i − GSF_r,let this be x
          sacl  *+,AR1                  ;q9
          banz  eucl1,*−,AR4
          lar   AR4,#di_buf
          lacl  #0                      ;zero ACC
          mpy   #0                      ;zero P register
          rpt   #(GSF_ORD−1)            ;10        2
          sqra  *+                      ;sum (x)
          apac                          ;i=1
          sach  scrhi                   ;store the 32 bits
          sacl  scrlo
          lt    inv250                  ;load LT with 1/250
          mpyu  scrlo
          pac
          add   #1,15                   ;rounding
          sach  scrlo                   ;store the MSB and discard the LSB
          lacl  scrlo
          mpy   scrhi
          apac                          ;q24
          neg
          add   #1,8                    ;rounding
          sach  obsscr,7                ;obsscr is in q15
          rpt   #8
          sfr                           ;get q15
          sub   #8000h                  ;check for overflow
          bend  $1,GT
          lacc  #8000h                  ;saturate it to −1
          sacl  obsscr
$1        lar   AR2, addr2
          ret                           ;return from euclid
```

The Q-point chosen can provide precision. It is simulated using direct Q9 GSF input (i.e., bypassing the front end) to the assembly code and comparing the numerical results with the C-code listing (floating-point). The recognition alignment is exactly the same for begin time, end time, and validation time, and the scores are also similar.

2.3 Utterance Detection

In a UNIX environment, the recognizer assumes the start and end of an utterance is known (for example, using the start and end of a file for reference). The recognition search starts from the beginning frame and proceeds sequentially to the end frame. However, in a real application, an utterance segmentation is not given, and an algorithm is needed to detect when an utterance starts and when it ends. In a real-time implementation, an EVM5X board was used. Since there is no push-to-talk button on this board, an utterance-detection algorithm was used to decide when to start recognition, when to stop recognition, and when to output the results.

The algorithm used to detect an utterance is based on frame root-mean-square (rms) energy and the HMM search engine, so it is initialized once before each utterance and called for every frame. Unlike the acoustic front-end and recognition codes, which make exact assembly-code tracing complicated, it is possible to trace the utterance-detection assembly code and understand the algorithm easily.

Example 2 is a C-code segment that illustrates the level-estimator portion of the utterance-detection algorithm. The HMM-search engine works with the level estimator to stop the detection process when a best-word score is detected.

Example 2. Level-Estimator C-Language Code

```
void level_est(float rms,float*speech_est)
{
 float       db_frm;
 static             float db_level=INITIAL_SPEECH_LEVEL;
 static             float db_noise=INITIAL_NOISE_LEVEL;
 float       delta;
 float       frm_egy;
/*-----------------------------------------------------------------
*      Get the current signal input level in DB domain
*---------------------------------------------------------------*/
frm_egy = MAX (rms, RMS_MIN);
db_frm = RMS_to_DB (frm_egy);
/*-----------------------------------------------------------------
*      Adapt the speech and noise levels
*---------------------------------------------------------------*/
delta = db_frm – db_level;
if(delta>0)
       db_level = db_level + 0.1813 * delta;
else
       db_level = db_level + MAX(0.01 * delta, –20);
delta = db_frm – db_noise;
if(delta>0)
       db_noise=db_noise + MIN (0.01 * delta, 20);
else
       db_noise=db_noise + MAX (0.1813 * delta, –20);
*/-----------------------------------------------------------------
*      Return the current speech estimate in RMS domain
*---------------------------------------------------------------*/
*speech_est=DB_to_RMS (MAX (db_level,db_noise + 15));
```

Two estimators are used to track speech and noise (background noise) levels of the input signals. They work in a complementary fashion, and each has a fast-adaptation direction and slow-adaptation direction. The speech estimator adapts upward rapidly, with a first-order time constant of 0.1 second, and downward slowly with a first-order time constant value of 5 seconds. The background time constants are reversed; therefore, speech estimation approximates a peak selector, while the noise estimator approximates a valley selector.

Utterance detection asks the recognizer to start when the start of an utterance is detected and to stop and report results when the end of an utterance is detected. One primary consideration in using this mechanism is that there is a 3-frame delay in detecting speech. The recognizer receives a frame only after speech is detected, which means that two frames are cut off and not processed by the recognizer. To solve this problem, an 8-frame circular buffer is used. The current frame is entered into the circular buffer while the 8-frame-old frame is sent to the recognizer. This allows a noticeable improvement in detection when an utterance starts with, for example, an “h” sound.

The parameters of this utterance detector can be adjusted for different purposes. For example, 50 low-energy frames were used as the criteria for an utterance end, which allows a person to break a sentence for up to one second (20 ms/frame * 50 = 1 second). If the purpose is to capture a segment of speech and make a model, for example to enter a name, five low-energy frames should be used as the token-end criterion. If 50 frames are used for a token enrollment, too much silence is included in the model, but if five frames are used, voices may be cut off in the middle of a sentence.

3 Real-Time Software Implementation

Because of time constraints, the algorithm was ported without simulating the algorithm’s performance in the fixed-point C-language program environment. Figure 5 shows the flow chart of real-time software implementation for the speech-recognition project.

The algorithm was implemented and simulated on a UNIX workstation, which is a non real-time environment. With satisfactory file-input simulation results, the algorithm was changed into a more DSP-like language. This step was taken to make the algorithm or assembly-language code more manageable when debugging. See Figure 5.

In the second stage, the C-language algorithm was ported to the TMS320C2xx DSP platform. Extensive effort was made to place the organization of the assembly-language code and the implementation of the memory overlays in a manner that is tailored to the architecture and memory constraints of the ’C2xx DSP. After the ’C2xx DSP assembly-language program was developed, tests were conducted using the EVM with inputs from existing files. The output from the assembly was compared with the output of the C-language program; when the two are within a tolerable range (two decimal places of accuracy), the Q-point and the dynamic range used in the DSP assembly-language program are essentially correct.

The final stage is to perform a full simulation of the assembly-language code using files as input. After this has been accomplished, the DSP assembly is essentially complete. Real-time testing was accomplished using EVM files after some slight modification of the codes.

Figure 6 shows the ’C203 with 544 words of dual-access RAM divided into three blocks:

B0	256 words
B1	256 words
B2	32 words

In this implementation, B0 is configurable as storage for all local and global variables. Memory overlays are also implemented in this block. B1 is used as data-buffer memory. B2 is used for command interrupt, A/D interrupt, and data variables.

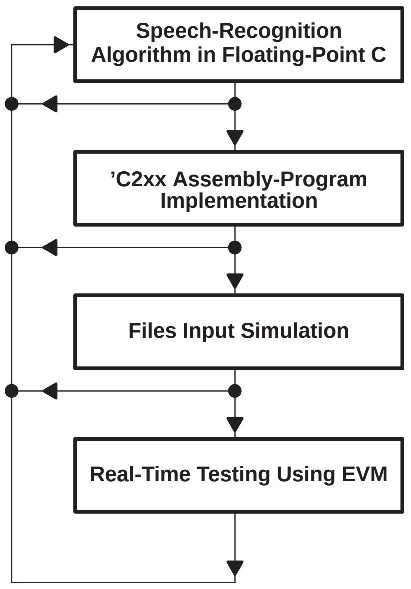
Speech-Recognition Algorithm in Floating-Point C
'C2xx Assembly-Program Implementation
Files Input Simulation
Real-Time Testing Using EVM

Figure 5. Real-Time Implementation

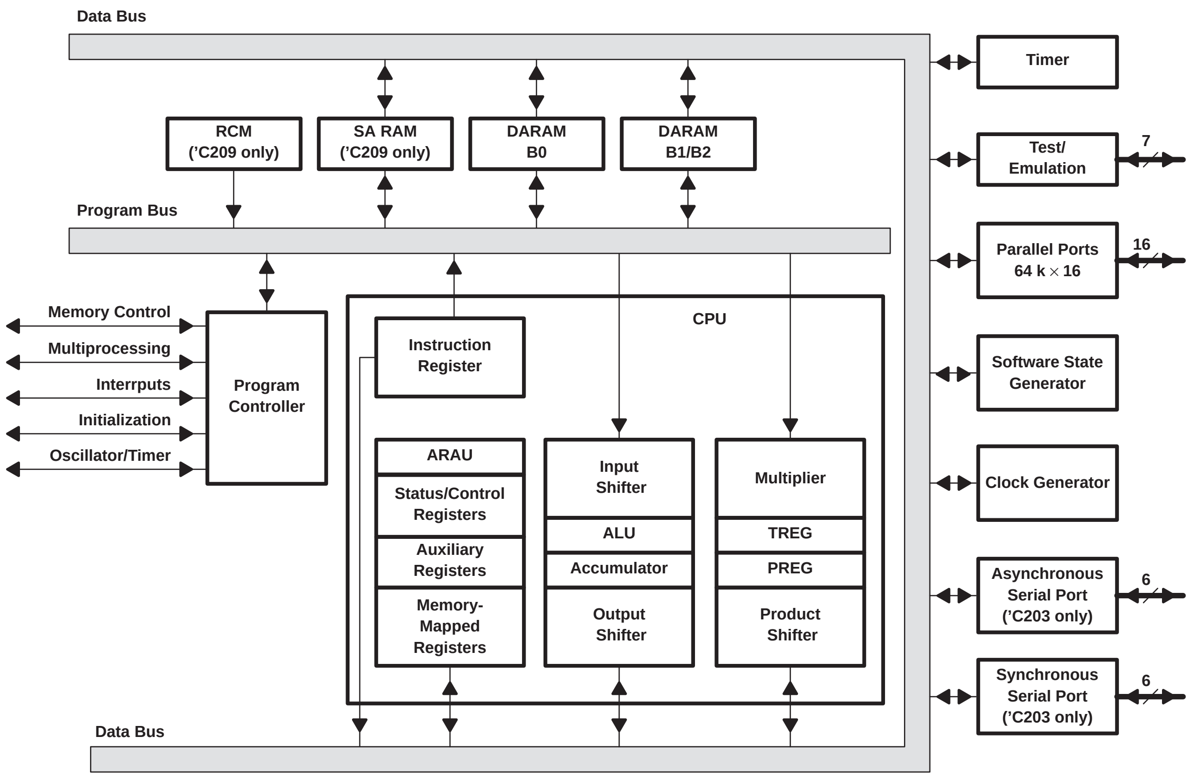

Figure 6. TMS320C203 Block Diagram

4 Conclusion

This application report presents the challenges in implementing speech recognition using an HMM-based algorithm based on the speaker-independent isolated-word recognizer. This report explains the various implementation issues associated with speech recognition from the min_HMM C-language code to the ’C2xx assembly language. It emphasizes techniques to prevent overflows of probability scores and efficiently represent some of the key variables used in fixed-point notation. The result is the presentation of a cost-effective implementation.

References

1. Baum, L.E., and Petrie, T., “Statistical Inference for Probabilistic Functions of Finite-State Markov Chains,” *Annotated Mathematical Statistics*, Volume 37, pp. 1554−1563, 1966.
2. Anderson, Wallace, “A Minimal HMM Speech Recognition, min_HMM,” Texas Instruments (TAR).
3. Netsch, Lorin, “Acoustic Feature Processing Reference Guide,” Texas Instruments (TAR).
4. Rabiner, L.R. and Shafer, R.W., “Digital Signal Processing of Speech Signals.”
5. Deller, John R., Proakis, John G., and Hausen, John H.L., “Discrete-Time Processing of Speech Signals.”
6. Yu-Hung Kao, “Robustness Study of Free-Text Speaker Identification and Verification,” PhD Thesis, University of Maryland, 1992.