Implementation of a CELP Speech Coder for the TMS320C30 using SPOX

APPLICATION REPORT: SPRA401

Mark D. Grosen
Spectron Microsystems, Inc

Digital Signal Processing Solutions

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Implementation of a CELP Speech Coder for the TMS320C30 using SPOX

Abstract

This chapter shows how a 4.8-kbps CELP (code-excited linear prediction) can be quickly developed using SPOX. Using TMS320C30 DSP power, the ease of use provided by C, and the SPOX DSP library, an efficient and portable coder can be developed quickly and compiled and executed on a variety of hardware platforms.

The chapter's main sections include:

- A 4.8-kbps CELP coder
- Using SPOX in development
- Implementation
  - Input/Output
  - Spectrum Analysis
  - Filters
  - Pitch and codebook search
  - Assembly language enhancements
  - Performance

Certain applications require the TMS320C30's high arithmetic throughput but in the IEEE floating-point format. These applications benefit from a custom chip that performs conversions between the TMS320C30 native format and the single-precision IEEE Standard 754-1985. This chapter describes this custom chip.

The description includes the following specific topics:

- External interfaces
- Architectural overview
- Converter operating modes
- Interrupts
- Software application examples
- Hardware application examples
- JTAG/IEEE-1149.1 scan interface

Product Support

World Wide Web

Our World Wide Web site at www.ti.com contains the most up to date product information, revisions, and additions. New users must register with TI&ME before they can access the data sheet archive. TI&ME allows users to build custom information pages and receive new product updates automatically via email.

Email

For technical issues or clarification on switching products, please send a detailed email to *dsph@ti.com.* Questions receive prompt attention and are usually answered within one business day.

Introduction

Speech coders are critical to many speech transmission and store-and-forward systems. With the emergence of universal standards, it is possible to develop systems that are interoperable. Quality and bit rate for speech coders vary from toll quality at 32 kilobits/second (kbps) (CCITT ADPCM) to intelligible quality at 2.4 kbps (DOD LPC-10). Recently, a new standard for 4.8 kbps with near toll-quality has been proposed and is based on code-excited linear prediction (CELP) techniques [1,2]. Unfortunately, products based on new coding algorithms are often slow to appear because of the considerable time and effort required to develop real-time implementations.

The purpose of this article is to demonstrate how a CELP coder based on this new standard can be quickly developed using SPOX. Utilizing the power of the TMS320C30 DSP plus the ease of use provided by C and the SPOX DSP library, an efficient and portable coder can be written in a much shorter period of time than that required by conventional assembly language methods. Because of the portability of SPOX and C, the coder can also be compiled and executed on a variety of hardware platforms.

A 4.8-kbps CELP Coder

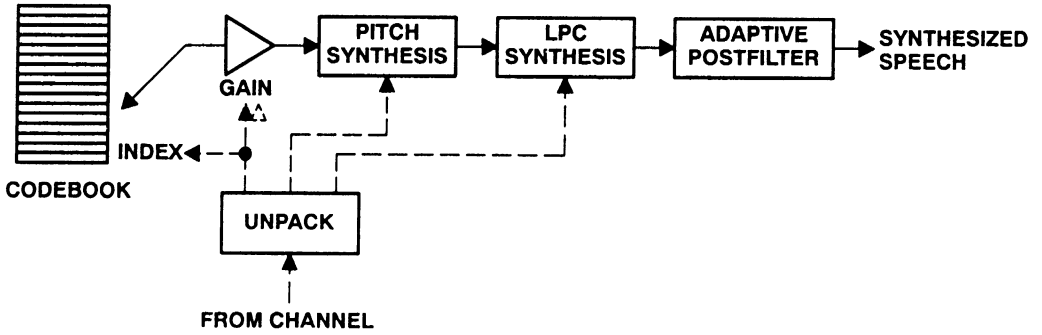
CELP coders were first introduced by Atal and Schroeder in 1984 [3]. These coders offer high quality at low bit rates, but at a high computational cost. Implementing the original systems directly required several hundred million instructions per second (MIPS). Much of the research on CELP techniques has concentrated on reducing this computational load to facilitate real-time implementations.

The proposed U. S. Federal Standard 4.8-kbps CELP coder (USFS CELP), Version 2.3, uses several techniques to reduce the complexity to a level where a one- or two-processor implementation is possible. These are the main characteristics of the coder:

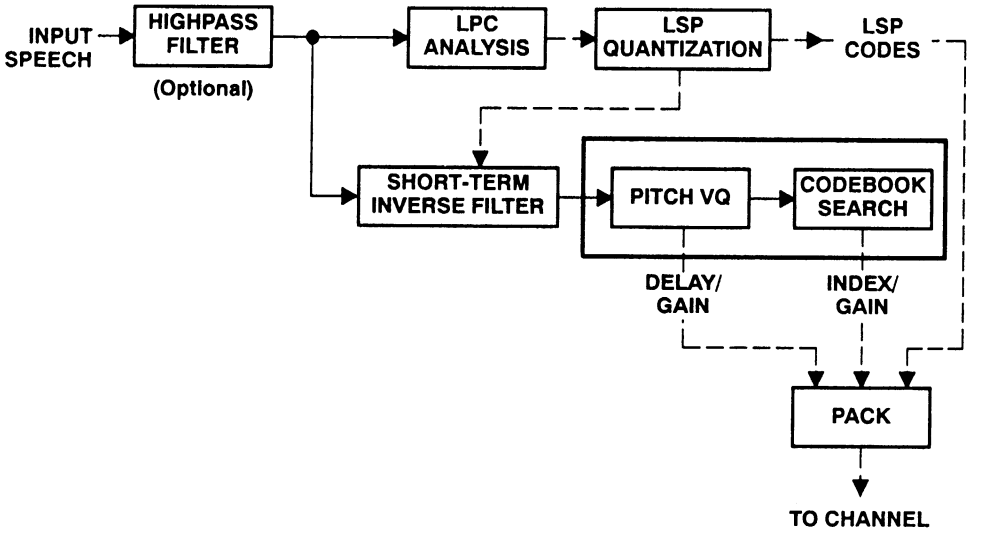
- 240-sample frame size at 8-kHz sampling rate
- Tenth-order short-term predictor
  - Calculated once per frame, open loop
  - Autocorrelation with Hamming window
  - LSP quantization
- Four subframes (60 samples)
  - One tap pitch predictor
    1) Closed loop analysis
    2) Even/odd subframe delta search method
  - 1024-element codebook
    1) Overlapped by 2 (see Pitch and Codebook Search)
    2) 75% of elements are zero

Block diagrams of the decoder and encoder are shown in Figure 1.

Figure 1. USFS CELP Decoder and Encoder Structures



DECODER



ENCODER

Bit allocations are given in Table 1 [2,4].

Table 1. 4.8-kbps CELP Parameters

	Spectrum	Pitch	Codebook
Update Parameters Bps	30 ms (240 samples) 10 LSP 1133.3	7.5 ms (60) 1 delay, 1 gain 1466.7	7.5 ms (60) 1 of 1024 index, 1 gain 2000
Remaining 200 bps reserved for expansion, error protection, and synchronization			

The standard also specifies an error protection scheme utilizing forward error-correcting Hamming code and parameter smoothing.

The major computational parts of the algorithm are the pitch search and the codebook search, both of which are performed four times per frame. An important technique to reduce the computations is the end-correction convolution technique (see Pitch and Codebook Search). This is a recursive convolution method that reduces the number of multiply-adds by an order of magnitude.

In addition, the codebook is designed to have approximately 75% of the samples equal to zero. This allows many of the convolution updates in the codebook search to be reduced to a simple shift of a vector of samples. On DSP processors with circular addressing, this shift can be replaced by using circular buffers.

To further reduce complexity, the pitch search is limited in range for every other subframe. During even-numbered subframes, the optimal pitch value is performed over the range 20 to 147 (128 values). On the odd subframes, the search is only over the range 16 from the previous pitch value. This also decreases the bit rate with a negligible effect on speech quality.

If adequate processing power is not available, you can implement an interoperable coder by using a subset of the full codebook. For example, if only the first 128 vectors from the codebook could be used, the sub-optimal coder would work with an optimal coder if the same frame structure and bit rate were used.

These techniques produce complexity estimates for the USFS CELP coder ranging from 5.3 MIPS to 16.0 MIPS for a 128-vector and 1024-vector codebook, respectively[4].

Using SPOX in Development

The computational complexity of CELP coders, even with use of the various techniques to reduce it, has made real-time implementations impractical on first- and second-generation DSPs. The recent introduction of the third-generation TMS320C30[5], however, makes it feasible to implement the USFS CELP coder with one or two processors. Furthermore, because of the general-purpose capabilities of the TMS320C30 and the availability of a C compiler and SPOX, development of a real-time coder can be significantly expedited.

In particular, SPOX provides the following functions to facilitate software development.

- C standard I/O functions
  - **printf( ), scanf( )**
  - **fopen( ), fread( ), fwrite( )**
- Stream I/O to move data efficiently
- Standard set of DSP math functions
  - Filters
  - Vector operations
  - Windows
  - Levinson-Durbin algorithm
- Processor independence

Both FORTRAN and C versions of the Version 2.3 USFS CELP coder were available as starting points for the real-time implementation. The initial development was done on a Sun worksta-

tion equipped with SPOX/SUN [6] and the usual UNIX programming tools, such as the symbolic debugger dbx. SPOX/SUN is a library of SPOX DSP math functions that can be used for developing SPOX applications on Sun workstations. The new version of the coder utilizing SPOX was checked against the existing implementation for correctness. After the new version was debugged on the workstation, the source code was recompiled employing the Texas Instruments TMS320C30 C compiler and linked with the SPOX/XDS library for the XDS1000 development system.

The same facilities for testing the code on the workstation were available on the XDS1000. A SPOX stream function (see Input/Output section) read digitized speech from a disk file. Status information was printed to the console screen. Command line arguments were used to vary the encoder's parameters such as the codebook size.

The software development process for the USFS CELP coder followed three evolutionary steps:

- C program using standard I/O
- C program using SPOX functions for faster math and I/O
- C program using SPOX and assembly language optimizations

The first step was taken because an existing C implementation was available. The C standard I/O provided by SPOX made it possible to run the application code written in C directly on the XDS1000. For example, functions (**fscanf**( )) that read control information from a disk file on the Sun also worked on the XDS1000 using the PC's hard disk.

In general, it would have been easier to start with the SPOX library functions to implement some of the common operations contained in the coder. Many of the functions needed (filtering, correlation, dot-product) are in the SPOX DSP library. In this case, the C implementations of these standard vector and filter functions in the existing program were replaced with the corresponding SPOX functions. The SPOX functions, written in optimized assembly language, execute several times faster than the corresponding C functions.

The last step was needed to meet real-time constraints. XDS1000 timing capabilities allowed the identification of two time-critical sections of the code which were then rewritten in TMS320C30 assembly code. Since the interface to the SPOX math functions is open, new math functions can be written that work with SPOX data structures such as vectors and filters.

Implementation

Several major parts of the USFS CELP encoder are implemented with a mixture of C, SPOX, and TMS320C30 assembly language functions. The decoder can be easily constructed from the material presented here. An adaptive postfilter for the decoder is not described here.

The framework of the resulting encoder is shown in Figure 2. A description of the major functions performed can be found in the following sections. Appendix A provides a short summary of the SPOX functions employed in the next four sections (Input/Output, Spectrum Analysis, Filters, and Pitch and Codebook Search).

Figure 2. Structure of the Encoder Function

```
encoder(instream, outstream)
    SS_Stream    instream;
    SS_Stream    outstream;
{

    while ( SS_get(instream, SV_array(speech)) ) {

    /* Apply a high pass filter to the input speech */
        SF_apply(hpfilter, speech, speech);

    /* Find the coefficients of the short-term prediction filter */
        calculateLP(speech, invcoeffs);

    /*
     * Convert the direct form coefficients to line spectrum pairs.
     * Then quantize the LSP's and convert back to direct form.
     */
        SV_a2lsp(invcoeffs, lsps);
        quantizeLSP(lsps, qntzlsps);
        SV_lsp2a(qntzlsps, invcoeffs);

    /*
     * For each of the 4 subframes, determine the pitch prediction
     * parameters and codebook (excitation) parameters
     */
        for (i = 0; i < 4; i++) {

            genShortResidual(s[i], res[i]);/* generate short term residual */
            pitchSearch(s[i], res[i]);     /* find optimum pitch predictor */
            genFullResidual(s[i], res[i]); /* generate residual */
            codeSearch(res[i], reshat);    /* find best codebook vector */
            updateFilters(reshat);         /* update filter states */
        }
        packParams();   /* pack parameters into output array */
        SS_put(outstream, params);
    }
}
```

Input/Output

Input speech samples are obtained by employing a function (**SS_get( )**), which reads data from a named stream (**instream**). The creation of instream during program initialization determines the source of the data. During development, the easiest source is a disk file with digitized speech. When real-time testing is needed, a codec connected to a TMS320C30 serial port could be utilized. For example, instream could be created to read from standard input with the following code segment.

```
#define FRAMESIZE        240 * sizeof(Float)

instream = SS_create(DF_FILE, DF_STDIN, FRAMESIZE, NULL);
```

The output stream (**outstream**) consists of the packed frame parameters. It could also go to a disk file or a serial port by using **SS_put( )**.

Spectrum Analysis

After preconditioning the signal with a highpass filter (see the Filters section), the coefficients of the short term prediction filter can be found by using the function **calculateLP( )** shown below.

```
SV_Vector           window, rc, error, cor, gammavec;

calculateLP(s, coeffs)
    SV_Vector       s, coeffs;
{
    SV_window(s, window, s);                /* window the speech in-place */
    SV_corr(s, s, cor);                     /* autocorrelation */
    SV_autorc(cor, coeffs, rc, error);      /* Levinson-Durbin */
    SV_mul2(gammavec, coeffs);              /* bandwidth expansion */
}
```

The vector window is initialized to contain the desired window; in this case, a Hamming window is used. The autocorrelation terms are stored in the vector **cor** that has the same length as the order of the short term filter. **SV_autorc( )** uses a Levinson-Durbin type algorithm to compute the inverse filter coefficients. As a side effect, the reflection coefficients are also stored in **rc**. Finally, a 15-Hz bandwidth expansion is produced by the multiplication of the inverse filter coefficient vector by a vector (**gammavec**) consisting of the terms

$$g[i] = 0.994^i \qquad \text{for } i = 0, 1, \ldots, m-1$$

Efficient quantization is obtained by:

- Transforming the prediction coefficients into line spectrum pairs (LSPs)
- Then quantizing the LSPs

The conversions between prediction coefficients and LSPs are not currently in the SPOX library. The existing C implementation evaluates cosine values directly, which is too expensive computationally. A more efficient routine (**SV_a2lsp( )**), that employs table-lookup of cosine values, has been written utilizing the algorithm outlined in [7]. The quantized LSPs are transformed back to direct-form coefficients for use in the short-term predictor.

Filters

Three filters in the encoder can be realized by use of SPOX filter objects. The inverse filter $A(z)$ and the short term predictor $1/A(z)$ share the same filter coefficients. The former is an FIR filter and the latter an all-pole filter. The final filter is the all-pole weighting filter $W(z)$ with coefficients given by $1/A(\lambda z)$, with $\lambda = 0.8$.

During the initialization of the encoder, the filters are created with the code fragment shown below.

```
#define FILTERSIZE    11 * sizeof(Float)

    SF_Filter         invfilter, predfilter, wgtfilter;
    SV_Vector         invcoeffs, wgtcoeffs;
    SA_Array          array;

     array = SA_create(SG_CHIP, FILTERSIZE, NULL);
     invfilter = SF_create(array, NULL, NULL);
     SF_bind(invfilter, invcoeffs, NULL);

     array = SA_create(SG_CHIP, FILTERSIZE, NULL);
     predfilter = SF_create(NULL, array, NULL);
     SF_bind(predfilter, NULL, invcoeffs);
```

```
array = SA_create(SG_CHIP, FILTERSIZE, NULL);
wgtfilter = SF_create(NULL, array, NULL);
SF_bind(invfilter, NULL, wgtcoeffs);
```

Note that the inverse and prediction filters are both bound to the same coefficient vector. For each new frame of speech, this vector is updated when it is passed to **calculateLP( )**.

An important consideration is that the filters are used more than once during a frame. A different signal is filtered each time, but the state (history) of the filter must be the same. This is accomplished before each filter operation by using the

- **SF_getstate( )** function to recover a vector with the state of the filter at the end of the previous frame
- **SF_setstate( )** function to restore the filter's state

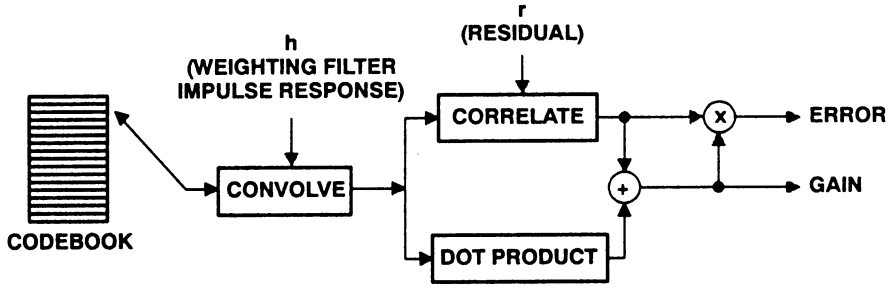
The following code segment shows how the short term prediction residual is generated for the pitch search.

```
SF_setstate(predfilter, NULL, predstate);
SV_fill(residual, 0.0);
SF_apply(predfilter, residual, residual); /* zero input of filter */

SV_sub3(residual, speech, residual);      /* speech - history */

SF_setstate(invfilter, invstate, NULL);
SF_apply(invfilter, residual, residual);  /* filter with inverse */

SF_setstate(wgtfilter, NULL, wgtstate);
SF_apply(wgtfilter, residual, residual);  /* filter with weighting */
```

Pitch and Codebook Search

After the program finds the short-term predictor and generates the corresponding residual, the pitch predictor and code book parameters are found for each of the four subframes. The pitch and codebook search functions are similar: both search over a set of values to minimize an error term. In this section, only the codebook search is illustrated (see Figure 3). Many of the functions, however, can be applied to the pitch predictor calculations.

Figure 3. Codebook Search Block Diagram



The search in Figure 3 minimizes the distance between the input vector and one of many generated vectors. The quantity being minimized is the Euclidean norm:

$$e = \| r - \hat{r} \|^2 = r^t r - 2\, r^t \hat{r} + \hat{r}^t\, \hat{r} \quad (1)$$

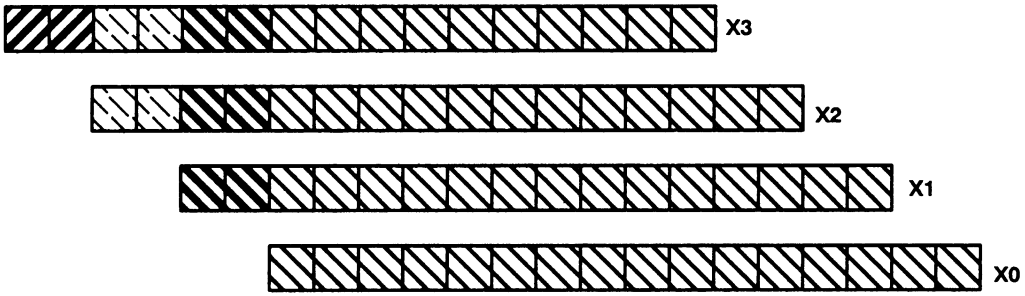
where

r = the original residual
$\hat{r}$ = the synthesized residual

It can be seen from the vector definition that only two terms need to be computed – the correlation of r and $\hat{r}$ and the energy of $\hat{r}$; this is because the energy of the original residual is invariant over all the generated residuals. It appears that there would be N convolutions and 2N dot products to perform for each sub-frame. Implemented directly, the codebook search would thus require 66 MIPS if N = 256 and a sub-frame length of 60 are specified.

Instead, the USFS CELP coder uses a specially structured codebook that greatly reduces the computational load. The biggest savings comes from the elimination of all but one of the convolutions for each subframe. The codebook is overlapped, as shown in Figure 4.

Figure 4. Structure of Overlapped Codebook



This structure permits a recursive convolution computation. The first codebook vector is convolved normally with the weighting filter. Subsequent convolutions, however, make use of the following relationships.

$$V_{i+1}(z) = z^{-1}\hat{R}_i(z) + x_{i+1}[1]H(z)$$
$$\hat{R}i + 1(z) = z^{-1}V_{i+1}(z) + x_{i+1}[0]H(z) \quad (2)$$

where $\hat{R}_i(z)$ is the Z-transform of the generated residual. Given the convolution of the previous codebook vector with the weighting filter, the convolution employing the next vector can be found with only 120 (2 × 60) multiplies and adds.

This number can be further reduced by another property of the codebook. The vectors are generated by center-clipping a gaussian noise source, which causes approximately 75% of the elements to be zero. Thus, 75% of the updates to the convolutions require no multiplications or additions; however, the convolution elements must still be shifted. The following function **update( )** implements the recursive update operation. Note that it must be called twice per codebook vector, once for each new term.

```
update(x, res, wgtimpulse)
    Float       x;
    SV_Vector   res, wgtimpulse;
{
    Float       *rptr, *rptrm1, *wptr;
    Int         len;

    len = SV_getlength(res);
    rptr = (Float *) SV_loc(res, len - 1);
    rptrm1 = rptr - 1;

    if ( x == 0.0 ) {                               /* no input, so just shift */
        for (; len > 1; len--) {
            *rptr-- = *rptrm1--;
        }
        *rptr = 0.0;
    }
    else {                                          /* update using new input */
        wptr = (Float *) SV_loc(wgtimpulse, len - 1);
        for (; len > 1; len--) {
            *rptr-- = *rptrm1-- + x * *wptr--;
        }
        *rptr = x * *wptr;
    }
}
```

Once the convolution has been determined, the corresponding error and gain can be found.

The following function calculates the error and gain terms.

```
Float error(res, reshat, gain)
    SV_Vector   res, reshat;
    Float       *gain;
{
    Float       cor, energy;

    SV_dotp(reshat, reshat, &energy);
    SV_dotp(reshat, res, &cor);
    *gain = cor / energy;
    return( *gain * cor );
}
```

The codebook search function with **update( )** and **error( )** functions is shown below. The first convolution must be calculated directly, so it is done outside of the main **for** loop. The error for each entry is compared against the current maximum; if it is greater than the maximum, this entry becomes the new best vector. The process is repeated for each of the *N* vectors.

```
SV_Vector       codebook, wgtimpulse;

codeSearch(res, reshat)
    SV_Vector res, reshat;
{
    Float       errmax, gain, err;
    Float       *cbptr;
    Int         i, best;

    findImpulse(wgtimpulse);

    SV_setbase(codebook, FIRSTVEC);

    convolve(codebook, wgtimpulse, reshat);
    errmax = error(res, reshat, &gain);
```

```
    best = 0;
    cbptr = (Float *) SV_loc(codebook, 0) - 1;

    for (i = 1; i < N; i++) {
        update(*cbptr--, reshat, wgtimpulse);
        update(*cbptr--, reshat, wgtimpulse);
        if ( (err = error(res, reshat, &gain)) > errmax ) {
            errmax = err;
            best = i;
        }
    }
}
```

After the search is completed, the gain of the best vector is recomputed and quantized. The corresponding gain index and index of the codebook element can then be readied for transmission.

Assembly Language Enhancements

The codebook and pitch searches require the largest share of the computation cycles in the encoder. One way to increase performance is to recode critical parts of these functions in assembly language. One such function is the **update( )** function described above for the recursive convolution computation.

An assembly language version of **update( )** was written to take advantage of the parallel instructions and repeat block capabilities of the TMS320C30. The assembly language function utilizes the same calling structure as the C version. The function was written using the assembly language macros provided with SPOX to work with the vector, matrix, and filter objects in the DSP library[8]. The new version of **update( )** is listed in Figure 5.

Figure 5. Update Function Written in TMS320C30 Assembly Language

```
*
* Synopsis:
*
*       Void update(x, res, wgtimpulse)
*           Float       x;
*           SV_Vector   res, wgtimpulse;
*
#include <sv30.h>

FP      .set    ar3
        .global _update

        .text

_update:
        push    FP
        ldi     sp, FP
*
*       Set the following registers by using vector object macros
*           ar0 - SV_loc(wgtimpulse, 0)
*           ar1 - SV_loc(res, 0)
*           rc  - the length of the vectors
*           r2  - x
*
        ldi     *-FP(2), ar2
        SV_get1 ar2, SV_LOC0, ar0
        ldi     *-FP(3), ar2
        SV_get2 ar2, SV_LEN|SV_LOC0, rc, ar1
*
        ldf     *-FP(4), r1                     ; x
        bzd     shift                           ; x is 0 so just shift
        subi    1, rc
        addi    rc, ar1                         ; ar1 -> res[l - 1]
        ldi     ar1, ar2                        ; ar2 -> res[i - 1]
*
*  General case when x != 0.0
*
        addi    rc, ar0                         ; ar0 -> wgt[l - 1]
        subi    2, rc                           ; set loop count

        mpyf    r1, *ar0--, r2                  ; x * wgt[i]
        addf    r2, *--ar2, r0

      rptb    lp20
        mpyf    r1, *ar0--, r2                  ; x * wgt[i]
lp20:   addf    r2, *--ar2, r0
||      stf     r0, *ar1--

         bud     end
         stf     r0, *ar1--
         mpyf    r1, *ar0, r0                       ; res[0] = x*wgt[0]
         stf     r0, *ar1
*
*  Case for x == 0.0
*
shift:   subi    2, rc                              ; loop l - 1 times
         ldf     *--ar2, r0                         ; prime the pipe

        rptb    slp
slp:    ldf     *--ar2, r0
||      stf     r0, *ar1--

        stf     r0, *ar1--                      ; final store
        ldf     0.0, r0                         ; first term = 0.0
        stf     r0, *ar1
*
end:    pop     FP
        rets
```

Performance

A complete CELP encoder was implemented as described above. Two versions were tested:

- One encompassing C and standard SPOX functions
- One having C, SPOX, and two custom TMS320C30 assembly language functions

Table 2 shows the execution times for different combinations of codebook size, processor, and implementation. To achieve near real-time performance for a codebook with 128 vectors, the codebook and pitch search functions were completely rewritten in assembly language. Each function required approximately 130 lines of assembly code.

Table 2. Timing of Various Implementations of the CELP Encoder for One Frame of Speech

Codebook Size	Sun (C/SPOX)	C30 (C/SPOX)	C30 (C/SPOX/ASM)
128	16,000 ms	88.2 ms	39.0 ms
256	24,000 ms	114.6 ms	54.3 ms

Memory requirements for the program on the TMS320C30 were approximately 14,000 words for instructions and approximately 6,000 words for data. The application code required approximately 4500 words of instructions. The SPOX operating system and DSP math functions consumed the remaining 9500 words of memory. This figure reflects many functions that are essential for easing development but unnecessary for a real-time implementation.

Once a real-time implementation has been achieved, the SPOX memory requirements can be greatly reduced by porting (or customizing) SPOX to a custom hardware implementation. In this case, the SPOX memory requirements can be reduced to approximately 4000 words, making a 12K-word implementation feasible (both data and instruction memory requirements).

These timings show that a real-time CELP coder can be implemented on a single TMS320C30. They also illustrate the power of the TMS320C30 compared to a standard microprocessor. Note that a TMS320C30 implementation has approximately 500,000 instruction cycles available in a 30-ms frame.

Version 3.0 of the USFS CELP coder has significant improvements in computational complexity, including:

- Ternary codebook to eliminate multiplications
- Shorter codebook
- Faster LSP conversion and quantization

Work to bring the SPOX implementation up to Version 3.0 is continuing. An investigation of a two-processor implementation is also being performed.

Summary

A 4.8-kbps CELP coder based on a Department of Defense-proposed standard has been implemented on a TMS320C30. Several of the functions used in the encoder were illustrated. A suboptimal implementation of the encoder using a 128-vector codebook is possible on only one TMS320C30. Work is continuing on both the algorithm and the software implementation to improve the coder's real-time performance.

With SPOX, the encoder was developed in less than one month. The resulting source (with the exception of two TMS320C30 assembly language functions) can be compiled and run on a Sun workstation, a PC, or a TMS320C30 system such as the Texas Instruments XDS1000. This represents a considerable improvement in development time and effort over previous implementation methods.

References

1) Kemp, D.P., Sueda, R. A., and Tremain, T. E., "An Evaluation of 4800 bps Voice Coders," *Proceedings of ICASSP '89*, IEEE, May 1989.
2) Campbell, J. P., Welch, V. C., and Tremain, T. E., "An Expandable Error-Protected 4800 bps CELP Coder," *Proceedings of ICASSP '89*, IEEE, May 1989.
3) Atal, B. S., and Schroeder, M. R., "Stochastic Coding of Speech at Very Low Bit Rates," *Proceedings of ICC '84*, pages 1610–1613, 1984.
4) Tremain, T. E., Campbell, J. P., and Welch, V. C., "A 4.8 kbps Code Excited Linear Predictive Coder," *Proceedings of Mobile Satellite Conference*, pages 491-496, May 1988.
5) Texas Instruments, Inc., *Third-Generation TMS320 User's Guide*, 1988.
6) Spectron MicroSystems, Inc., *SPOX/SUN User's Guide*, April 1989.
7) Soong, F. K., and Juang, B. H., "Line Spectrum Pair (LSP) and Speech Data Compression," *Proceedings of ICASSP '84*, pages 1.10.1-1.10.4, IEEE, 1984.
8) Spectron MicroSystems, Inc., *Adding Math Functions to SPOX*, March 1989.

Appendix A

The SPOX functions used in the code examples are briefly described below. Complete descriptions can be found in *Getting Started With SPOX* and the *SPOX Programming Reference Manual*. These manuals are supplied with the XDS1000. They are also available from Spectron Micro-Systems, Inc.

Stream Functions

```
SS_get - get data from a stream into an array

          Int SS_get(stream, array)
              SS_Stream   stream;
              SA_Array    array;

SS_put - put data from an array to a stream

          Int SS_put(stream, array)
              SS_Stream  stream;
              SA_Array   array;
```

Vector Functions

```
SV_autorc - perform inverse filter calculations

          Void SV_autorc(cor, inv, rc, alpha)
              SV_Vector    cor;
              SV_Vector    inv;
              SV_Vector    rc;
              SV_Vector    alpha;

SV_corr - calculate correlation of two vectors

          SV_Vector SV_corr(src1, src2, dst)
              SV_Vector    src1;
              SV_Vector    src2;
              SV_Vector    dst;

SV_dotp - calculate the dot product of two vectors

          SV_Vector SV_corr(src1, src2, result)
              SV_Vector    src1;
              SV_Vector    src2;
              Float        *result;

SV_fill - fill a vector with a value

          SV_Vector SV_fill(vector, value)
              SV_Vector    vector;
              Float        value;

SV_getlength - return the length of a vector

          Int SV_getlength(vector)
              SV_Vector    vector;

SV_loc - return the address of a vector element
```

```
            Ptr SV_loc(vector, num)
                SV_Vector    vector;
                Int          num;

SV_mul2 - multiply elements of two vectors

            SV_Vector SV_mul2(src, dst)
                SV_Vector    src;
                SV_Vector    dst;

SV_setbase - set the base of a vector

            Void SV_setbase(vector, base)
                SV_Vector    vector;
                Int          base;

SV_sub3 - subtract elements of two vectors and store results in a third
          vector

            SV_Vector SV_sub3(src1, src2, dst)
                SV_Vector    src1;
                SV_Vector    src2;
                SV_Vector    dst;

SV_window - apply a symmetric window to a vector

            SV_Vector SV_window(src, wnd, dst)
                SV_Vector src;
                SV_Vector wnd;
                SV_Vector dst;
```

Filter Functions

```
SF_apply - apply a filter to a vector

            SV_Vector SF_apply(filter, input, output)
                SF_Filter    filter;
                SV_Vector    input;
                SV_Vector    output;

SF_bind - bind coefficient vectors to a filter

            Void SF_bind(filter, num, den)
                SF_Filter    filter;
                SV_Vector    num;
                SV_Vector    den;

SF_getstate - copy filter state arrays into vectors

            Void SF_getstate(filter, hisinv, hisoutv)
                SF_Filter    filter;
                SV_Vector    hisinv;
                SV_Vector    hisoutv;

SF_setstate - copy vectors into filter state arrays

            Void SF_setstate(filter, hisinv, hisoutv)
                SF_Filter    filter;
                SV_Vector    hisinv;
                SV_Vector    hisoutv;
```