TEXAS INSTRUMENTS

Acoustic Echo Cancellation

Algorithms and Implementation on the TMS320C8x

Application Report

1996 Digital Signal Processing Solutions

Texas Instruments

Printed in U.S.A., May 1996

SPRA063

Acoustic Echo Cancellation

Algorithms and Implementation on the TMS320C8x

David Qi
Digital Signal Processing Solutions

SPRA063
May 1996

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 applications 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.

Contents

Appendices

List of Illustrations

List of Tables

Introduction

In recent years, speakerphones and hands-free cellular phones have been used widely around the world for audio-conferencing and video teleconferencing applications. A speakerphone or a hands-free cellular phone allows *full-duplex* communication without having to hold the phone. *Full-duplex* means voices on both ends of the line are transmitted continuously, as with a normal telephone.

The speech from the far-end caller is broadcast by the speakerphone or the hands-free cellular phone and then repeats itself by bouncing off the inside surfaces of the room. This repetition of sound is called an *echo*. Echoes are picked up by the near-end microphone, creating a feedback loop where the far-end caller hears an echo of his or her own voice. To solve this problem, developers are using the digital signal processing technique of acoustic echo cancellation (AEC) to stop the feedback and allow full-duplex communication.

This application report describes the implementation of an integrated N-tap digital acoustic echo canceller on the Texas Instruments TMS320C8x parallel processor (PP). The report presents a brief discussion of generic echo cancellation algorithms. The implementation considerations for a 512-tap (64-ms span) echo canceller on the TMS320C8x are described in detail, as well as the software logic and flow for each program module. Line echoes are not considered in this report.

Algorithm Overview

Figure 1 shows the principle of the echo canceller for one direction of transmission. In this figure, *y(n)* represents the far-end signal, *r(n)* is the undesired echo, and *x(n)* is the near-end signal. The near-end signal is superimposed with the undesired echo on port D. The received far-end signal is available as a reference signal for the echo canceller, and is used by the canceller to generate a replica of the echo called $\hat{r}(n)$. This replica is subtracted from the near-end signal plus the echo to yield the transmitted near-end signal *u(n)* where $u(n) = x(n) + r(n) - \hat{r}(n)$. Ideally, the residual echo error $e(n) = r(n) - \hat{r}(n)$ will be very small after echo cancellation.

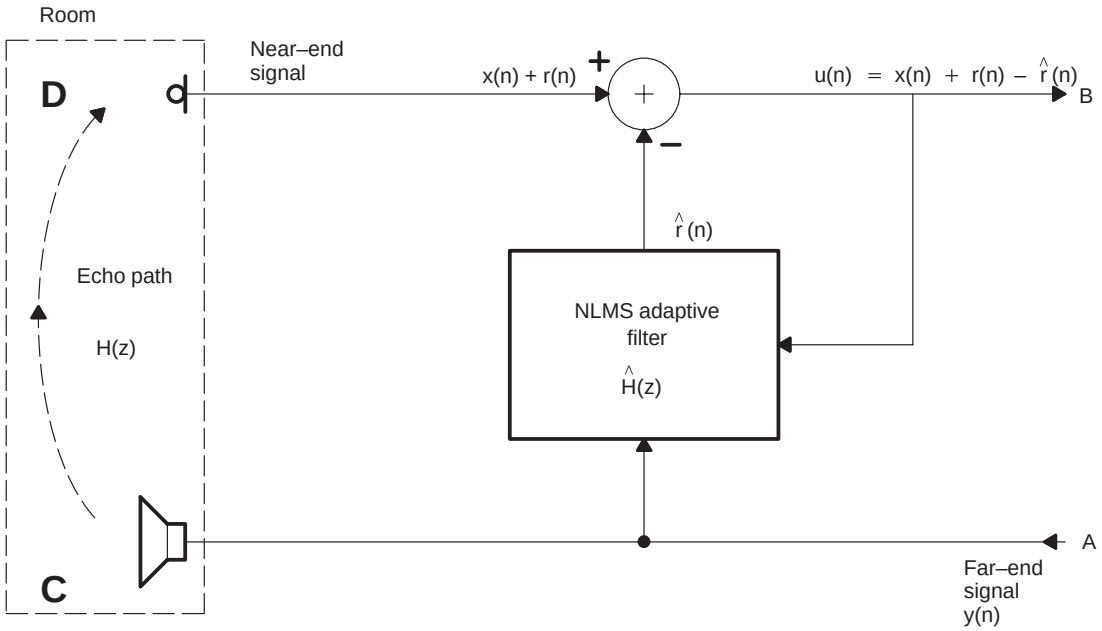

Figure 1. Echo Canceller Configuration

Adaptive Transversal Filter

The echo canceller generates the echo replica by applying the reference signal to an adaptive transversal filter (tapped-delay line), as shown in Figure 2. If the transversal filter's transfer function is identical to that of the echo path, then the echo replica will be identical to the echo, thus achieving total cancellation.

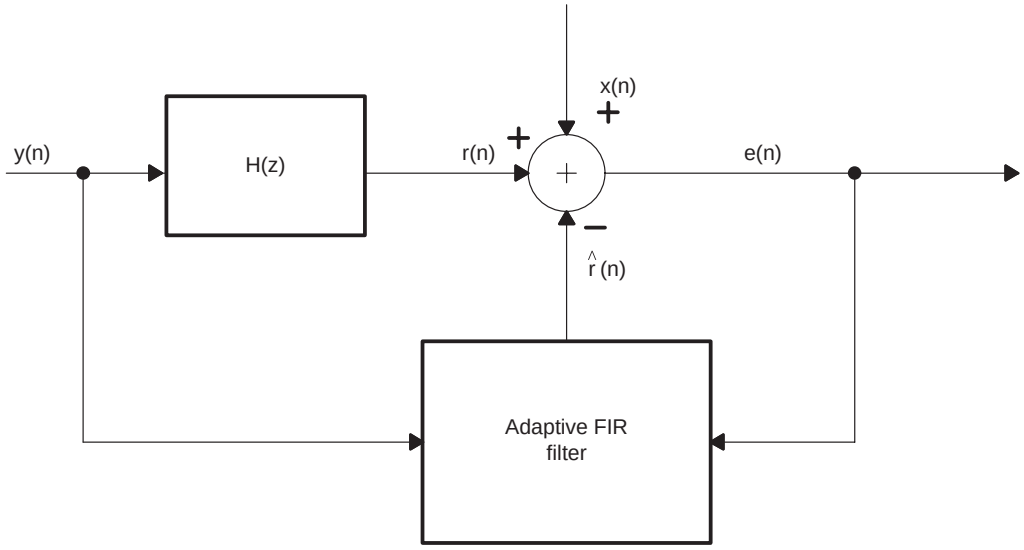

Figure 2. Echo Replication Using an Adaptive FIR Filter

Adaptive FIR Filter

The AEC adaptive filter software, *lms.p*, implements a least-mean-squared (LMS) algorithm and an adaptive finite impulse response (FIR) filter. The algorithm uses the previous sample values and errors to update the FIR filter's coefficients. It then uses the updated new coefficients and the latest sample values to calculate the FIR filter's output. This output is used to calculate the next error. Equations for the calculations are given below. These equations assume that the echo path has a finite impulse response.

FIR Filter Equations

The following equation is used to calculate the FIR filter's output:

$$\hat{r}(n) = \sum_{k=0}^{N-1} a_k \ * \ y\ (n - k) \tag{1}$$

where

N = Number of filter taps

a_k = Filter coefficients

y = Filter memory (taps delay)

$a_k = a(0), a(1), \ldots, a(N-1); y = y(n-0), y(n-1), \ldots, y(n-N+1)$

If there is no near-end signal, that is, *x(n)* = 0, then the equation for the error signal *e(n)* is:

$$e(n) = r(n) - \hat{r}(n) \tag{2}$$

A normalized LMS (NLMS) algorithm is used in the LMS adaptive filter function to update the FIR filter's coefficients. NLMS is almost the same as LMS, except that you use equation 5 to normalize the step size.

Following is an example of the NLMS equation that is used to update the FIR filter's coefficients:

$$a_k(n+1) = a_k(n) + \frac{\mu e(n)}{P(n)} * y(n) \tag{3}$$

where

$\frac{\mu e(n)}{P(n)}$ = NLMS constant at given sample time *n*

where

μ = Step size

The following example shows an extension of equation 3:

$$\begin{bmatrix} \alpha(0) \\ \alpha(1) \\ \alpha(2) \\ \vdots \\ \vdots \\ \alpha(N-1) \end{bmatrix} = \begin{bmatrix} \alpha(0) \\ \alpha(1) \\ \alpha(2) \\ \vdots \\ \vdots \\ \alpha(N-1) \end{bmatrix} + \frac{\mu e(n)}{P(n)} * \begin{bmatrix} y(0) \\ y(1) \\ y(2) \\ \vdots \\ \vdots \\ y(N-1) \end{bmatrix} \tag{4}$$

where

$\mu e(n) = \left[r(n) - \hat{r}(n) \right] * stepsize$

P(n) = Estimated signal power of far-end speech at sample time *n*

The convergence properties of the AEC algorithms are largely determined by the step size parameter and the power of the far-end signal *y(n)*. In general, making the step size larger speeds the convergence, while a smaller step size reduces the asymptotic cancellation error. The convergence time constant is inversely proportional to the power of *y(n)*, so the algorithms converge very slowly for low-power signals. (For more information, see the Signal Power Estimation section.)

The short window power estimate of far-end speech (fes_short_pwr) is used in equation 5 to normalize the step size. This equation requires execution of the **divi** instruction, which is discussed in detail in Appendix B.

$$2B = 2B(n) = \frac{B1}{P_y(n)} \tag{5}$$

where

$B1$ = Compromise value of the step size constant

$P_y(n)$ = Estimate of the average power of *y(n)* at sample time *n*

The generic equation for estimating the average power is:

$$P_y(n) = L_y^2(n) \tag{6}$$

The next two equations are related to equation 6. These two equations show two ways of estimating average power. The first method uses the following equation for $L_y(n)$:

$$L_y(n) = (1- \rho)L_y(n-1) + \rho|y(n-1)| \tag{7}$$

where

ρ is constant

The second method uses the following equation for $P_y(n)$:

$$P_y(n) = (1- \rho)P_y(n-1) + \rho * y^2(n) \tag{8}$$

LMS Adaptive Filter Implementation

The LMS adaptive filter software, *lms.p*, is written in TMS320C8x's PP assembly code. It implements an N-tap LMS adaptive filter and uses the PP's compactor and register allocator (PPCA) to allocate variables in the program.

The filter function is designed to work on one sample input per update for all the filter's coefficients. It also can work with blocks of data by modifying the AEC's main program, *main.c*. The block method is recommended because it is more efficient than using one sample at a time.

Basic Operations

The basic operations in an LMS adaptive filter function are multiplication, accumulation, and a memory shift. Since the algorithm requires old memory data to update the coefficients before doing the filtering, always make the data buffer one tap longer than the coefficients' buffer to retain one old data set after the memory shift. This allows you to use old memory values to calculate new coefficient values, as discussed below.

After loading new data into the data buffer, the *lms.p* software performs a multiplication and accumulation loop that updates the coefficients and shifts the entire data buffer down by one. The direction in which the multiplication and accumulation operations are performed is not important. The software starts these operations at the end of the coefficients buffer and the data buffer, and moves upward through the buffers. It calculates each new coefficient value by multiplying the old data times the error. Next, it overwrites the old coefficient value with the new value. The updated coefficients then are used to calculate the filter output. Figure 3 illustrates the update sequence and memory shift.

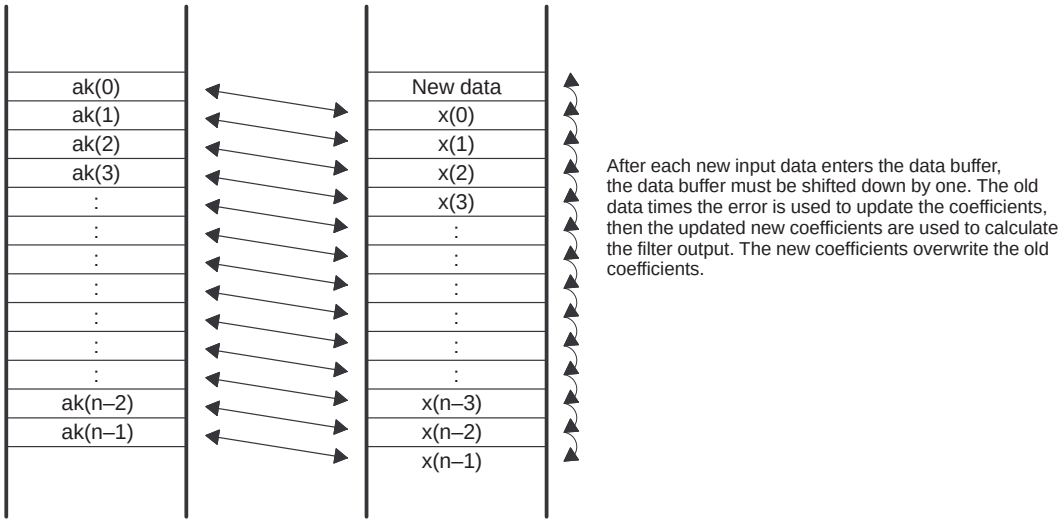

Figure 3. Updating Coefficients and Shifting Memory

Following is an example of PP assembly code for the main loop of the LMS adaptive filter function:

```
LMS_LOOP_START:                          ; LMS loop starts here
     up_prod_1 =r (x_1 * erf)<<1         ; get update product
     || ak_new_2 = ealu(SHIFT_ADD:ak_old_2+up_prod_2>>16)
                                         ; update coefficient
     || ak_old_1 =h *--Ga_ak             ; load next coefficient
     || *(La_x + [1]) =h x_1             ; shift memory
     prod_2 = x_1 * ak_new_2             ; get multiplication product
     || y = y + prod_1                   ; accumulate filter output
     || *(Ga_ak + [1]) =h ak_new_2       ; store new filter coeff
     || x_2 =h *--La_x                   ; load filter memory sample
     up_prod_2 =r (x_2 * erf)<<1         ; get update product
     || ak_new_1 = ealu(SHIFT_ADD:ak_old_1+up_prod_1>>16)
                                         ; update coefficient
     || ak_old_2 =h *--Ga_ak             ; load next coefficient
     || *(La_x + [1]) =h x_2             ; shift memory
LMS_LOOP_END:
     prod_1 = x_2 * ak_new_1             ; get multiplication result
     || y = y + prod_2                   ; accumulate filter output
     || *(Ga_ak + [1]) =h ak_new_1       ; store new coefficient
     || x_1 =h *--La_x                   ; load next memory sample
```

The sample code shows a four-cycle loop that processes two signal samples. This loop not only performs FIR filtering, but it also updates the filter coefficients and shifts the filter memory at the same time. So, there are two cycles of LMS adaptive filter processing for each signal sample.

The above four instructions are fully optimized. Each instruction takes advantage of the TMS320C8x parallel processor's ability to do parallel operations in every clock cycle. These parallel operations include multiply, arithmetic logic unit (ALU) or extended ALU (EALU) operation, global transfer, and local transfer. Rounded multiply is used to improve the output precision in all multiplication.

Equation 1 indicates that the following operations are needed to perform each coefficient update:

- One multiply
- One EALU (shift and add)
- One load
- One store

Each new sample also requires FIR filtering, which needs these operations:

- One multiply
- One EALU
- One load
- One store

The operations for the coefficient update and the FIR filtering can be combined. To do this, the LMS adaptive filter function requires at least two PP instructions which can perform two multiplies, two EALU operations, and two loads and two stores. Although each PP instruction can perform one multiply, one EALU, and two loads in parallel, the LMS adaptive filter function cannot finish in two cycles because of the operation sequence. To produce each output from a new input, the following calculations must done:

1. New sample * old error → update product
2. Update product + old coefficient → new coefficient
3. New coefficient * old sample → new accumulator product
4. New accumulator product + old accumulated value → new accumulated value

Improving Performance

The above four calculations indicate that at least four cycles are required by the LMS algorithm, and these cycles do not include load, store, and memory shift. If the standard operations in a four-instruction cycle are used to perform the LMS adaptive filter function, many free slots are left after each instruction completes. Better performance is achieved by calculating two inputs in four instructions using the following operations:

- Four multiplies
- Four EALUs
- Four loads
- Four stores

These operations fit perfectly into four PP instructions, which leave no free slots. This tight loop makes a two-cycle LMS adaptive filter function a reality. Appendix C provides an example of how to implement an LMS adaptive filter using four PP instructions. For more information about software optimization, refer to the *Parallel Processor User's Guide*.[1]

Signal Power Estimation

The power estimate software, *power.p*, is written in TMS320C8x's PP assembly code. This software implements the signal power estimation used in AEC to normalize the loop gain (step size). In addition, the power estimation output is used by the speech detection function to determine the sequence of operations to be performed in that function.

The basic operations in signal power estimation are multiplication and addition in the manner of a simple infinite impulse response (IIR) filter. Because the IIR filter is a recursive filter, each output of the filter depends on the previous one.

Following is the generic equation for estimating power:

$$P(n) = L^2(n) \tag{9}$$

The next two equations are related to equation 9. These equations show two ways to estimate power, depending on whether *L(n)* or *P(n)* is given. Equation 10 uses the absolute value of the input to estimate the signal power, while equation 11 uses the input squared to calculate the signal power. Equation 11 is used in the power estimate program, *power.p*.

$$L(n) = (1-\alpha) * L(n-1) + \alpha * |X(n)| \tag{10}$$

or

$$P(n) = (1-\alpha) * P(n-1) + \alpha * X^2(n) \tag{11}$$

where

$\alpha = 1/32$ (very short window power estimate, 4 ms)

$\alpha = 1/128$ (short window power estimate, 16 ms)

$\alpha = 1/16384$ (long window power estimate, 2048 ms)

$X(n) =$ the input signal

A different α value is chosen for each different window-size power estimate.

The far-end signal power (fes_short_pwr) is estimated by using a short window size of 16 ms. This estimate is used in the NLMS algorithm to normalize the step size. A window size of 4 ms is used to determine the very short power estimates of near-end power (nes_vshort_pwr) and far-end power (fes_vshort_pwr). These estimates are used in far-end and near-end speech detection.

It is important that the functionality of speech detectors be accurate to avoid erroneous detection, which could lead to an unstable system. Therefore, all power estimation is carried out in double-precision.

Block Sample Signal Power Estimate

The power estimate function is designed for block sample processing. The three different window sizes of near-end signal power are calculated in one tight loop. The software assumes that certain amounts of samples already have been stored in the PP's on-chip data RAM. Since three different window-size powers need to be calculated for near-end speech, the D0 register loads different EALU labels alternately for different shift amounts. To increase the output's precision, the software keeps the previous power in Q31 format and uses it for the next power estimate.

The following example code shows the six cycles that are required for each near-end signal's power estimate in very short, short, and long window sizes:

```
nes_power_estimates:
   d0 = SHORT_POWER
   dummy = ealu(SHORT_POWER: s_pwr_2_16 – s_pwr_2_16>>7)
   || *(sp + [d0_SHORT_POWER]) = d0
   d0 = LONG_POWER
   dummy = ealu(LONG_POWER: l_pwr_2_16 – l_pwr_2_16>>14)
   || *(sp + [d0_LONG_POWER]) = d0
   d0 = VSHORT_POWER
   dummy = ealu(VSHORT_POWER: v_pwr_2_16 – v_pwr_2_16>>5)
   || *(sp + [d0_VSHORT_POWER]) = d0
   La_nes_long_pwr_Q15 = &*(xba + L_NES_LONG_POWER_Q15)
   La_nes_short_pwr_Q15 = &*(xba + L_NES_SHORT_POWER_Q15)
   La_nes_vshort_pwr_Q15 = &*(xba + L_NES_VSHORT_POWER_Q15)
   s_pwr_2_16 = *(Ga_nes_short_pwr_Q31 = xba + L_NES_SHORT_POWER_Q31)
   ; initialize pointer to previous short power estimate of
   ; near end signal, and load to a register
   l_pwr_2_16 = *(Ga_nes_long_pwr_Q31 = xba + L_NES_LONG_POWER_Q31)
   ; initialize pointer to previous long power estimate of
   ; near end signal, and load to a register
   v_pwr_2_16 = *(La_nes_vshort_pwr_Q31 = xba + L_NES_VSHORT_POWER_Q31)
   ; initialize pointer to previous very short power estimate of
   ; near end signal, and load to a register
   nes =h *(Ga_nes = xba + L_NEAR_END_SPEECH)
   lrs0 = SAMPLES – 1
   le0 = NES_POWER_EST_LOOP_END
   dummy =h *––La_nes_long_pwr_Q15
   || d0 = *(sp + [d0_SHORT_POWER])
   ; load d0 with ALU configuration for power estimate
NES_POWER_EST_LOOP:
   nes_squared = nes * nes
   || s_pwr_2_16 =ealu(SHORT_POWER: s_pwr_2_16 – s_pwr_2_16>>7)
   || nes_long_pwr =h1 l_pwr_2_16    ; get upper half of nes long
                                     ; power estimate
   s_pwr_2_16 = s_pwr_2_16 + nes_squared>>7
   || *La_nes_long_pwr_Q15++ =h nes_long_pwr
                                     ; store near end long power
   || d0 = *(sp + [d0_VSHORT_POWER])
```

```
   || d0 = *(sp + [d0_VSHORT_POWER])
   v_pwr_2_16 =ealu(VSHORT_POWER: v_pwr_2_16 – v_pwr_2_16>>5)
   || nes_short_pwr =h1 s_pwr_2_16   ; get upper half of nes short
                                     ; power estimate
   v_pwr_2_16 = v_pwr_2_16 + nes_squared>>5
   || *La_nes_short_pwr_Q15++ =h nes_short_pwr
                                     ; store near short power
   || d0 = *(sp + [d0_LONG_POWER])
   l_pwr_2_16 =ealu(LONG_POWER: l_p_2_16 – l_pwr_2_16>>14)
   || nes_vshort_pwr =h1 v_pwr_2_16
   || d0 = *(sp + [d0_SHORT_POWER])
NES_POWER_EST_LOOP_END:
   l_pwr_2_16 = l_pwr_2_16 + nes_squared>>14
   || nes =h *++Ga_nes               ; load a near end signal sample
   || *La_nes_vshort_pwr_Q15++ =h nes_vshort_pwr
                                     ; store v_short power
   .cjump NES_POWER_EST_LOOP
nes_long_pwr =h1 l_pwr_2_16          ; get upper half word (Q15)
   br = iprs
   *Ga_nes_long_pwr_Q31 = l_pwr_2_16 ; store nes long power (Q31)
   || *La_nes_vshort_pwr_Q31 = v_pwr_2_16; store nes very short power
   *Ga_nes_short_pwr_Q31 = s_pwr_2_16
   || *La_nes_long_pwr_Q15++ =h nes_long_pwr
   .uexit
```

Sample-By-Sample Signal Power Estimate

Although the power estimate function is designed for block processing, it can be adjusted to perform sample-by-sample processing by changing the sample size to one. The following example shows how to do this by implementing a short-window signal power estimate using equation 11 and the TMS320C8x's PP assembly code.

```
short_power_estimates:
sample                  .set    d1
prev_pwr_31             .set    d2
La_prev_pwr_Q31         .set    a0
power                   .set    d5
sample_squared          .set    d7
   .ptext
   .lock d0
   .entry d1,d2, a0
   d0 = *(sp + [d0_SHORT_POWER])
```

```
sample_squared = sample * sample
|| prev_pwr_31 = ealu(SHORT_POWER: prev_pwr_31 –prev_pwr_31>>7)
|| br = iprs
prev_pwr_31 = prev_pwr_31 + sample_squared>>7
power =h1 prev_pwr_31
|| *La_prev_pwr_Q31 = prev_pwr_31
.uexit
```

Speech Detection

Speech detection is a very important part of AEC. It must be done before the software can determine whether to filter, update, filter and update, or freeze the adaptive filter. There are three speech detectors:

- Far-end speech detector
- Double-talk detector
- Near-end speech detector

The speech detection software always checks for the presence of far-end speech first, then it goes to double-talk detection. It performs double-talk detection even if it does not detect far-end speech. This avoids false detection due to the small signal level of far-end speech. If the software does not detect either far-end speech or double-talk, it goes to near-end speech detection. All detection is based on the signal power estimate algorithm, which is discussed in detail in the Signal Power Estimation section of this report.

Figure 4 shows a block diagram of the speech detection function that is used in AEC algorithms.

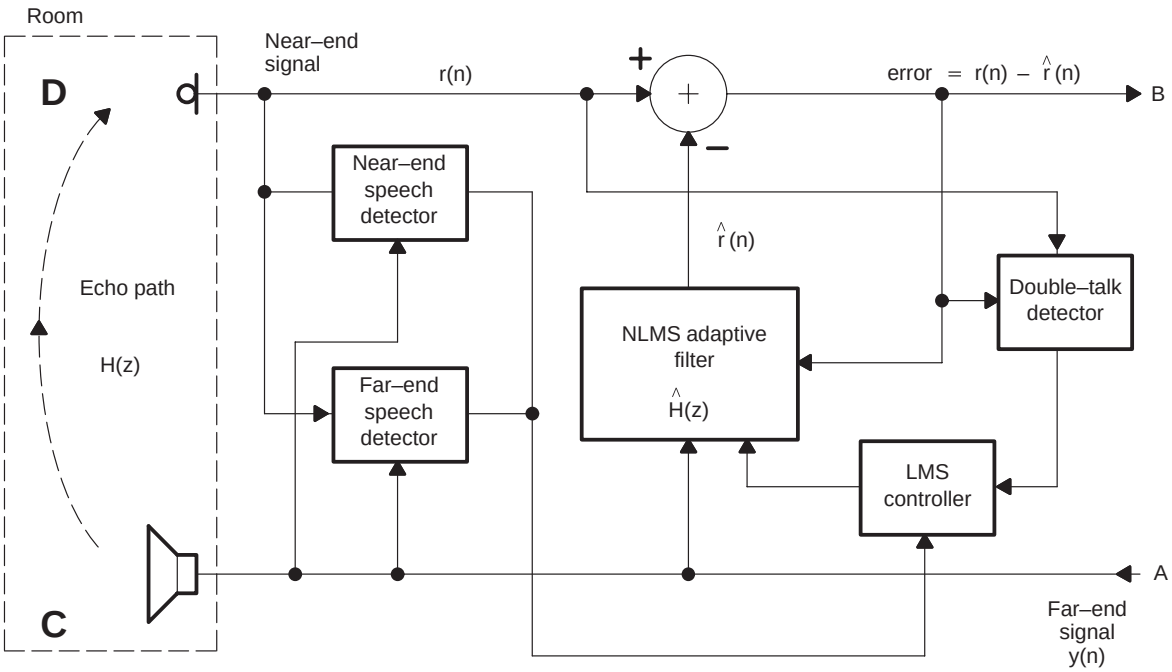

Figure 4. Speech Detection Block Diagram

Far-End Speech Detection

Far-end speech means that only the far-end speaker is active. This is the only time the AEC program performs both filtering and updating. The very short power estimates of the far-end and near-end speech signals are used to determine if far-end speech is present. So, far-end speech is detected only if

$$fes\_vshort\_pwr + FES\_MARGIN > nes\_vshort\_pwr$$

where

FES_MARGIN = Threshold constant

The value of the threshold constant must be chosen carefully from real-time experiments. If the threshold value is too small, the background noise picked up by the microphone results in a false detection. On the other hand, if the threshold value is too large, part of the speech is not transmitted when the speech signal levels are low.

Double-Talk Detection

In AEC algorithms, the presence of both far-end speech and near-end speech is known as double-talk. The following equation implements and defines a double-talk detector based on echo return loss enhancement (ERLE):

$$ERLE = 10 \log \frac{P_x^2(n)}{P_e^2(n)}$$

where

$ERLE$ = 8db (chosen from real-time experiments)

$P_x(n)$ = Short window power estimate of the near-end signal

$P_e(n)$ = Short window power estimate of the residual error signal

So, double-talk is detected if

$$err\_short\_pwr > C * nes\_short\_pwr + D$$

where

$C = 10^{ERLE/10}$

D = Threshold constant (determined from real-time experiments)

The higher the value of the D constant, the less double-talk is detected, but the more the coefficients will be updated. In addition, a higher threshold constant results in a greater difference between the echo and the echo replica which means less echo cancellation, but it also results in less noise interference.

After double-talk is detected, the program freezes the FIR filter's coefficient updates; however, filtering is still done, and the double-talk hangover counter is reset to high (see the Hangover Counters section for more information).

Near-End Speech Detection

Near-end speech exists when there is no far-end speech and no double-talk. Near-end speech is detected by calculating the very short power estimate and the very long power estimate of the near-end signal as follows:

$$nes\_short\_pwr + NES\_MARGIN > nes\_long\_pwr$$

where

NES_MARGIN = Threshold constant (chosen from real-time testing)

If near-end speech is detected, the program sets the near-end-speech mode bit and freezes the LMS adaptive filter function.

Hangover Counters

Two hangover counters are used in the speech detection algorithm:

- DT_HANG
- NES_HANG

Each hangover counter is set to a hangover time of 600 samples or 75 ms after its corresponding type of speech is detected (assume the sampling frequency is 8 kHz). If a type of speech is not detected, its hangover counter is decreased by one. Table 1 shows how the counters determine when to do filtering, updating, filtering and updating, or nothing.

Hangover counters play a very important role in AEC algorithms. After each different speech is detected, its corresponding mode bit is set. For example, the FAR_SPEECH mode bit sets to 1 to indicate that only far-end speech is detected. The AEC_UPDATE mode bit is not set to 1 until the double-talk and near-end hangover counters are both less than zero. This method avoids erroneous detection and gives some buffer time to turn on the adaptive filter.

Table 1. Hangover Counters and LMS Mode Bit Settings

HANGOVER COUNTERS		SET FILTERING MODE BIT	SET UPDATING MODE BIT
DT_HANG ≥ 0	NES_HANG ≥ 0	YES	NO
	NES_HANG < 0	YES	NO
DT_HANG < 0	NES_HANG ≥ 0	NO	NO
	NES_HANG < 0	YES	YES

AEC Program Flow

This section outlines the flow of the AEC program, and shows which tasks run on the master processor (MP) and which tasks run on the parallel processor (PP).

MP Tasks

Figure 5 shows a flowchart of the AEC program running on the MP. The MP is responsible for interrupt handling, transmitting speech signals, and sending tasks to the PP.

Audio Codec is an audio capture and playback program that was written in C and runs on the MP. The AEC initialization task and the AEC task are two tasks that the MP sends to the PP. The initialization task runs once at the beginning of the AEC program. The AEC task is the AEC program's main task. It includes the power estimate function, the LMS adaptive filter function, and the speech detection function.

Table 2 and Table 3 list the prototypes for the AEC initialization task and the AEC task.

Table 2. Prototype for the AEC Initialization Task

AEC INITIALIZATION TASK PROTOTYPE	
Syntax	void AEC_State_Init(void) ;
Parameters	None
Return value	None
Description	This function is called to initialize the LMS filter's coefficients, memories, and all required variables.

Table 3. Prototype for the AEC Task

AEC TASK PROTOTYPE	
Syntax	void AEC(int *fes_ptr, int *nes_ptr, *out_ptr) ;
Parameters	*fes_ptr /* Pointer to far-end speech */ *nes_ptr /* Pointer to near-end speech */ *out_ptr /* Pointer to AEC output */
Return value	int *out_ptr
Description	This function is called to process AEC by using the LMS algorithm.

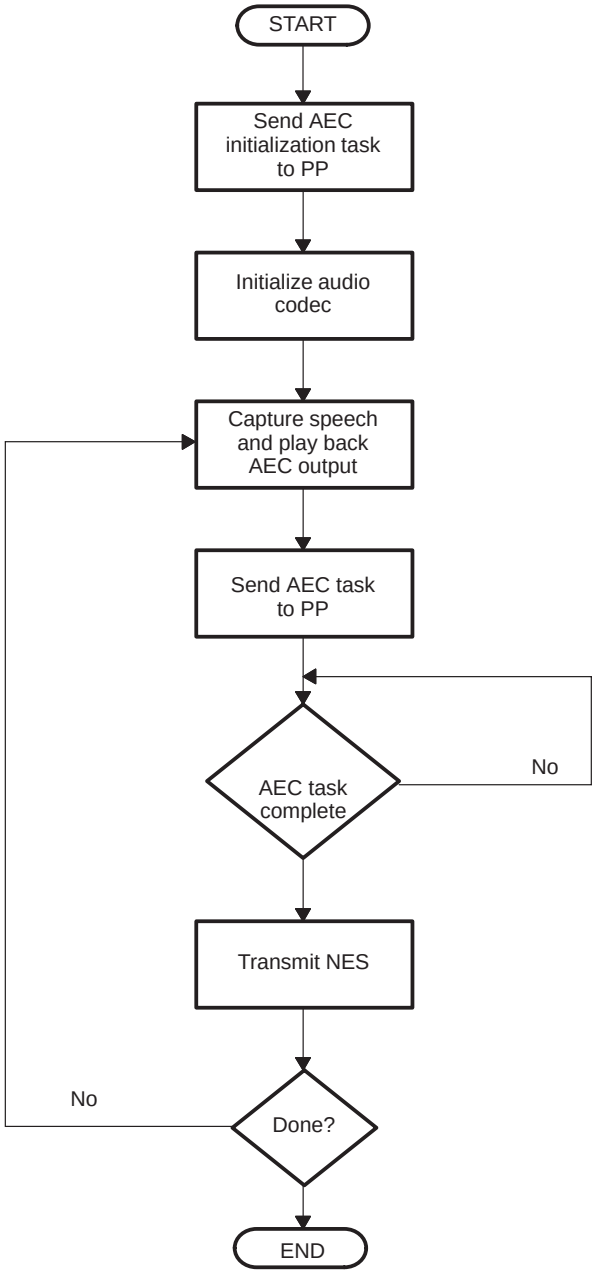

Figure 5. Flowchart of the AEC Program Running on the MP

PP Tasks

Figure 6 shows a flowchart of the AEC program running on the PP. The AEC initialization task performs two procedures. First, it restores the initial values of the filter's coefficients. It does this by transferring external memory to the PP's on-chip data RAM using packet transfer, a very useful feature provided by

TMS320C8x's architecture. Second, it initializes all variables and zeroes out the filter's memory. This process is important because the PP's data RAM could be used by other tasks when the AEC task is idle.

After initializing the processor, the power estimate function calculates all required power estimates for both the near-end and far-end signals, and stores them into on-chip memory, ready for the speech detection function to use. The power estimate function is designed for processing blocks of data. By changing the sample size to one, it also works for sample-by-sample processing. The program then determines if it needs to start the LMS adaptive filter. If it does, it runs the LMS adaptive filter function; otherwise, it goes to the speech detection function.

The PP remains in command interpreter mode before and after it executes the AEC tasks. This function is provided automatically by the multitasking executive, which is used to manage the AEC program. Refer to the *Multitasking Executive User's Guide* for more information about the command interpreter and sending tasks.(2)

Table 4 through Table 6 show the prototypes used for the LMS adaptive filter function, the power estimate function, and the speech detection function.

Table 4. Prototype for the Adaptive Filter Function

AEC LMS ADAPTIVE FILTER FUNCTION PROTOTYPE	
Syntax	short LmsFilter (int filter_size, int erf, short *coefficients, short *mem, short input, *output) ;
Parameters	filter_size /* The size of filter. Determined by how long the echo tail wants to be cancelled. */ erf /* Normalized error times stepsize*/ *coefficients /* Point to filter coefficients */ *mem /* Point to filter memory */ input /* A new signal sample */
Return value	*output /* Point to output buffer */
Description	This function is called to adapt the filter coefficients and calculate an FIR filter output by using a sample input and previous error.

Table 5. Prototype for the Power Estimate Function

AEC POWER ESTIMATE FUNCTION PROTOTYPE	
Syntax	short PowerEstimates (short near_end_signal, short far_end_signal, short error_signal, short output_power);
Parameters	near_end_signal /* Near-end signal */ far_end_signal /* Far-end signal */ error_signal /* Error signal */
Return value	output_power /* Estimated power of input signal */
Description	This function is called to estimate the signal power which is used in speech detection and to normalize the step size.

Table 6. Prototype for the Speech Detection Function

AEC SPEECH DETECTION FUNCTION PROTOTYPE	
Syntax	void SpeechDetector (void);
Parameters	None
Return value	None
Description	This function is called to detect far-end speech, near-end speech, and double-talk. After each detection, the corresponding mode bit is set.

Figure 7 shows a flowchart that presents more detailed information about how the AEC program handles different situations during speech detection.

The speech detection function uses data from the power estimate function to determine the sequence of operations to be performed for the corresponding mode. For example, if a near-end signal is detected by the near-end detector, the NEAR_SPEECH flag is set to 1, and the NES_HANG hangover counter is reset to the highest value (600) to avoid toggling of the speech modes.

Next, the program goes to double-talk detection even if no far-end speech is detected. This avoids false detection. If double-talk is not detected and there is no far-end speech, the program decreases the DT_HANG hangover counter by one and goes to the *control_lms* routine. If neither far-end speech nor double-talk is detected, the program goes to the near-end speech detector. If near-end speech is detected, the NES_HANG hangover counter is set to the highest value, and the program goes to the *control_lms* routine. This routine checks all hangover counters before setting the AEC_UPDATE and AEC_FILTERING mode bits. See Table 1 for the relationship between hangover counters and mode bit settings.

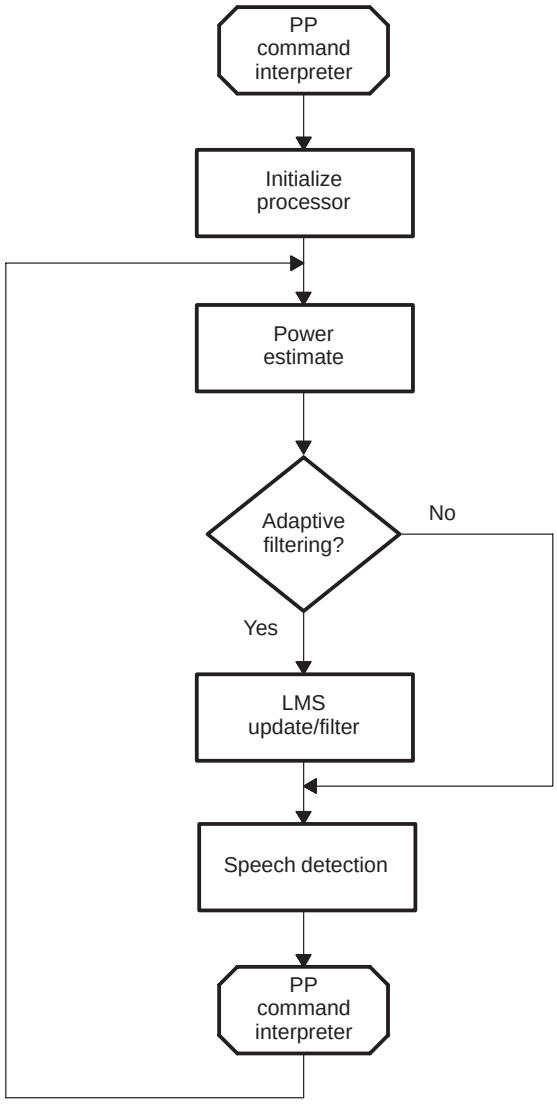

Figure 6. Flowchart of the AEC Program Running on the PP

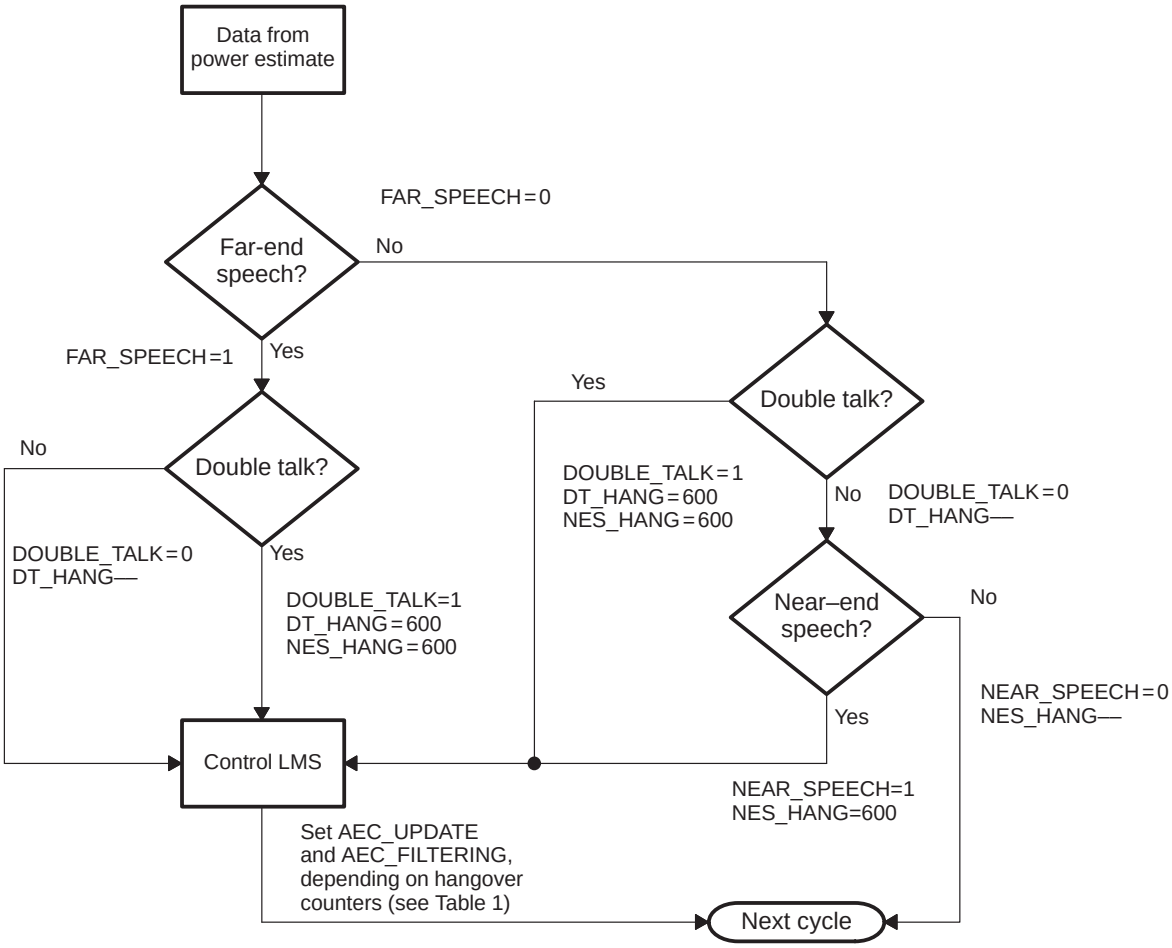

Figure 7. Flowchart of the Speech Detection Function

Hardware Setup

Real-time testing was performed on a software development board (SDB). Figure 8 shows the hardware connection.

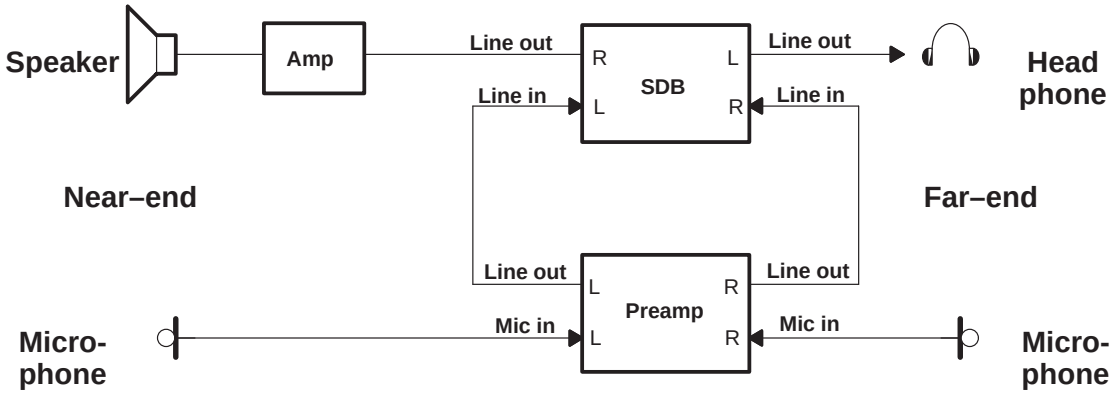

Figure 8. AEC Real-Time Testing on an SDB

Since the SDB has only line inputs, the pre-amplifier was used to convert microphone output to line level. Note that audio codec on the SDB has microphone input which can be used by modifying the SDB. Also, although the channels setting in audio codec was set to 2, only one channel of the AEC process is shown in Figure 8.

During testing, the pre-amplifier's gain was set to +45 dB. This gain can be changed by using a different microphone setup, or by changing the input gain setting in audio codec. For more information, refer to the *AD1848k Technical Reference* manual.(3) Table 7 shows the SDB audio input/output connections.

Table 7. SDB Input/Output Signals

LINE	SIGNAL
SDB left line-in	Near-end-signal
SDB right line-in	Far-end-signal
SDB left line-out	AEC output
SDB right line-out	Far-end-signal

The sampling frequency in the SDB's audio codec was set to 8 kHz for capture. The sample data format was set to 16-bit 2's complement pulse code modulation (PCM). The bandwidths of the AEC software depend on sampling frequency and filter taps (assume the TMS320C8x PP is running at 50 MHz). Theoretically, if the sampling frequency is set to 8 kHz, the maximum filter size can be set to 2000. However, in real-time testing, the filter size was limited to about 800 taps because of the size of the PP data RAM and because hardware access took longer than expected. With a filter size of 800 taps, up to a 100 ms echo tail is cancelled.

Table 8 shows the estimated performance of the AEC task, while Table 9 shows an estimate of how much time is required to cancel the echo. Refer to Appendix C for more details about the link control file and memory usage.

Table 8. Computational Estimates for the AEC Task (Standalone)†

	PROGRAM MEMORY (BYTES)	DATA MEMORY (BYTES)‡	CYCLES (N TAPS)§
LMS Filter	20*8	4N + Sample	2N + 12
Speech Detection	71*8	40	71
Power Estimate	43*8	10*Sample+20	43
Initialization	99*8	64	99
Others	85*8	5*Sample	85
Total	2544	4N+16Sample+124	2N+310

† Assumes the PP clock cycle is 50 MHz and sampling frequency is set to 8 kHz
‡ N is the number of LMS filter taps.
§ Sample is the number of samples per block of data.

Table 9. Estimated PP Loading Time for Echo Cancellation¶

ECHO TAIL	PP LOADING
32 ms (256 Taps)	10.4% (5.2 MIPS)
64 ms (512 Taps)	18.6% (9.3 MIPS)
100 ms (800 Taps)	27.8% (14.9 MIPS)

¶ Based on block processing for 80 samples

Summary

The implementation of acoustic echo cancellation on the TMS320C8x offers high performance and the flexibility to meet varying user needs. Users can easily change the software to fit their specific application requirements. Because the TMS320C8x has the highest engine power of any parallel processor currently available anywhere, it provides the fastest convergence time and the smallest residual error after echo cancellation.

The simple LMS-adapted tapped-delay-line filter plays an important role in this implementation. Large numbers of taps, stepsize normalization, and a new speech-detection algorithm improve the implementation capabilities. Applications of this technology can be widely used in other communication tasks, including speech coding, digital cellular phone systems, and teleconferencing systems.

References

1. *Parallel Processor User's Guide* literature number (SPRU110A), Texas Instruments, 1995.
2. *Multitasking Executive User's Guide*, literature number (SPRU112A), Texas Instruments, 1995.
3. *AD1848k Technical Reference*, Analog Devices, 1993.
4. David Messerschmitt, David Hedberg, Christopher Cole, Amine Haoui, and Peter Winship, "Digital Voice Echo Canceller with a TMS32020," *Digital Signal Processing Applications With the TMS320 Family*, Volume 1, 1989.
5. Padma P. Mallela, *Full-Duplex Speaker-Phone, Theory and Implementation*, 1993.
6. C. Richard Johnson, Jr., "On the Interaction of Adaptive Filtering, Identification, and Control," *IEEE Signal Processing Magazine*, March 1995.

APPENDIX A—Variables Used in the AEC Algorithms

Table 10. Algorithm Variable Names and Descriptions

NAME	DESCRIPTION
l_pwr_2_16	Near-end signal long power estimate in Q31 format
s_pwr_2_16	Near-end signal short power estimate in Q31 format
v_pwr_2_16	Near-end signal very short power estimate in Q31 format
nes_long_pwr_Q15	Near-end signal long power estimate in Q15 format
nes_long_pwr_Q31	Near-end signal long power estimate in Q31 format
nes_short_pwr_Q15	Near-end signal short power estimate in Q15 format
nes_short_pwr_Q31	Near-end signal short power estimate in Q31 format
nes_vshort_pwr_Q15	Near-end signal very short power estimate in Q15 format
nes_vshort_pwr_Q31	Near-end signal very short power estimate in Q31 format
fes_short_pwr_Q31	Far-end signal short power estimate in Q31 format
fes_vshort_pwr_Q15	Far-end signal very short power estimate in Q15 format
fes_vshort_pwr_Q31	Far-end signal very short power estimate in Q31 format
err_short_pwr_Q15	Error signal short power estimate in Q15 format
err_short_pwr_Q31	Error signal short power estimate in Q31 format
prev_fes_pwr	Previous far-end speech short power estimate in Q31 format
nes	Near-end signal picked up by microphone
fes	Far-end signal when far-end speech is active
error	Output signal to far-end speaker (near-end signal minus output of LMS filter)
mode	AEC status mode which is designed to do filtering, updating, or nothing
nes_hang	Hold near-end speech hangover counter
fes_hang	Hold far-end speech hangover counter
dt_hang	Hold double-talk hangover counter
r	Near-end signal picked up by the microphone
r_hat	Replica of the echo generated by the LMS filter
erf	Error input of the LMS/NLMS filter
ue	stepsize*error
ueM1	Intermediate value of the divi operation
db_margin	Double-talk detection margin
C	Double-talk detection margin (determined by ERLE)
fes_margin	Far-end speech detection margin
nes_margin	Near-end speech detection margin
step size	Step size of the LMS filter
filter_size	Hold adaptive filter taps
speech_out	Speech to the far-end caller

APPENDIX B—Q16 unsigned Divi Instructions

When the dividend is less than the divisor, the quotient will always be a fractional number. Since the parallel processors on the TMS320C8x are fixed-point processors, the output must be changed to Q16 format.

The *Parallel Processor User's Guide* contains a detailed discussion of the **divi** operation, but all the examples are for integer **divi** operations.(1) This section explains how to use the **divi** operation to perform the Q16 format number division and produce the Q16 output. The **divi** operations for other Q format numbers are similar.

The general syntax for the **divi** operation is:

dst1 =**[**cond[.pro]**]divi**(src2,dst2=[cond]src1**[**[n]src1-1**]**)

All parameters are described on page 8-99 of the *Parallel Processor User's Guide*, and therefore are not repeated here.(1) The pair of src1:mf registers is set to the 64-bit dividend, and the src2 registers are set to the negative of the 32-bit divisor.

Example 1 and Example 2 below show the different data formats of the dividend placement in the registers between the regular 16-bit integer **divi** operations and the Q16 number **divi** operations.

Example 1. Dividend Placement of 16-bit Integer Divi Operations

dividend:	32 0s	dividend (16-bit)	16 0s

`src1(32 MSBs of dividend) + mf(32 LSBs of dividend) = 64-bit`

Example 2. Dividend Placement of Q16 Number Divi Operations

dividend:	16 0s	dividend (16-bit)	32 0s

`src1(32 MSBs of dividend) + mf(32 LSBs of dividend) = 64-bit`

Instead of placing the dividend in the upper half of the mf register, put it in the lower half of src1, which is equal to left shift divided by 16 bits. After the sixteen divide iterates, the quotient will be in Q16 format. Example 3 shows sample code for placing the dividend in scr1.

Example 3. Code for Placing the Dividend in the Lower Half of Scr1

```
src1          .set  d1   ; must odd register.
src1M1        .set  d0
src2          .set  d4
dividend      .set  d6
divisor       .set  d7
remainder     .set  d0
      .ptext
      .system $divi
      .system _divi
$divi:
_divi:
      dividend = *a8            ;load dividend.
            || divisor = *a0    ;load divisor.
      src2 = –divisor           ;negate the divisor.
            || mf = &*(0)       ;make 32 LSBs of 64–bit
                                ;dividend == 0.
      src1 = dividend           ;src1 = 32 MSBs of 64–bit dividend.
            || lrse2 = &*(12)   ;14 loops + 2 delay slots == ;16 divis
      src1 = divi(src2, src1M1 = src1)            ;1st divi. iterate.
      src1 = divi(src2, src1M1 = src1 [n] src1M1) ;2nd divi. iterate.
      src1 = divi(src2, src1M1 = src1 [n] src1M1) ;3rd through 15th
                                                  ;divis iterate.
      remainder = [n] src1      ;Remainder may be in src1M1 register or
                                ;src1 register, depending on the n
                                ;status bit. You may never need the
                                ;remainder.
; At the end of the 16 divide iterates, the 16-bit quotient is found in
; the 16 LSBs of the mf register, and the 16-bit remainder is found in
; remainder register.
      br = iprs
      nop
      nop
```

APPENDIX C—aec Software Files

This appendix contains listings of all software files required for implemention of a 512-tap (64-ms span) echo canceller on the TMS320C80.

File: audinit.c

```
#include <stdlib.h>
#include <sdbembed.h>
#define SAMPLE_RATE    (8000)
#define CHANNELS       (2)
void audinit()
{
        UCHAR reg_set;
        /* Initialize codec to bidirectional mode                  */

        SDB_InitAudio(1);
        /* Set codec sample rate, channel and sampling data format */
        SDB_SetClockDataFormat(SAMPLE_RATE, CHANNELS,
                               SDB_16BIT_PCM);

        /* Set left and right line input source and gain           */

        SDB_SetLeftInput(3,0);
        SDB_SetRightInput(3,0);
        /* Enable mic inputs' 20db gain                            */
        /*reg_set = 0xaa;
        set_codec_reg(0x0,reg_set);
        set_codec_reg(0x1,reg_set);
        /* Set left and right output attenuation and mute           */

        SDB_SetLeftDac(0,0);
        SDB_SetRightDac(0,0);
        /* Start audio use loopback mode enable both
           capture and playback                                     */
        SDB_StartAudio(SDB_AUDIO_LOOPBACK);
}
```

File: main.c

```
/*
 *------------------------------------------------------------------------
 *       Copyright (C) 1993–1994  Texas Instruments Incorporated.
 *                         All Rights Reserved
 *------------------------------------------------------------------------
 *
 *  main.c  --  C source code for application based on MVP executive
 *
 *------------------------------------------------------------------------
 *  History:
 *    03/30/93...Original version written ................. J. Van Aken
 *    03/15/94...Updated to run under new kernel ........... J. Van Aken
 *    01/10/95...Modified to run a test program ............ Varadi Gyorgy
 *------------------------------------------------------------------------
 */
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
/*#include "audiodrv.h"*/
#include <mvp.h>                /* MP hardware functions */
#include <mvp_hw.h>             /* MVP hardware parameters */
#include <task.h>
#include <mp_int.h>
#include <mp_ppcmd.h>
#include "main.h"
#include "aec.h"
#include "pp3func.h"
#include <sdbembed.h>
extern short OUTPUT;
extern short NEAR_END_SIGNAL;
extern short FAR_END_SIGNAL;
/*extern short COEFF_INIT[]; */
extern void AEC_State_Init();
extern void AEC();
extern void audinit();
long port;
/* ========================================================================*/
```

```
/*
 * Dynamic memory allocation functions with multitasking capability
 */
long mySemaId;                   /* mutex protection for heap accesses */
void *myMalloc  (size_t size)
{
    void *p;

    TaskWaitSema (mySemaId);
    p = malloc (size);
    TaskSignalSema (mySemaId);
    return (p);
}
void myFree (void *ptr)
{
    TaskWaitSema (mySemaId);
    free (ptr);
    TaskSignalSema (mySemaId);
}
/* ========================================================================= */
void MPServerTask (void *dummy)
{
       short fes_low;
       short fes_high;
       short nes_low;
       short nes_high;
       short temp, temp2, temp1;
       int acal=0;
       short fes_lacal, fes_hacal, nes_lacal,nes_hacal;
       int i = 0;
       short delay_buff[1000];
    port = TaskOpenPort (–1);
    TaskReceiveMsg (port);
    Pp3CmdBufIssue (AEC_State_Init);
    Pp3WaitCompletion ();
/* Initialize audio drive */
    audinit();
for (;;)
  {
```

```
        /* Run audio drive capture speech */
        /* Find DC offset*/
        if (acal <1000){
        while (!(IO_REG(AUDIO_STATUS) & 0x20));
        fes_lacal = IO_REG(AUDIO_PIO_DATA);
        fes_hacal = IO_REG(AUDIO_PIO_DATA);

        nes_lacal = IO_REG(AUDIO_PIO_DATA);
        nes_hacal = IO_REG(AUDIO_PIO_DATA);
        acal ++;
        }
        else {
        while (!(IO_REG(AUDIO_STATUS) & 0x20));
        fes_low = IO_REG(AUDIO_PIO_DATA);
        fes_high = IO_REG(AUDIO_PIO_DATA);

        nes_low = IO_REG(AUDIO_PIO_DATA);
        nes_high = IO_REG(AUDIO_PIO_DATA);
        }
        /* Subtract the DC offset */
        temp = ((fes_low & 0x00ff) | fes_high << 8)–
((fes_lacal&0x00ff)|(fes_hacal<<8));
        NOCACHE_SHORT(FAR_END_SIGNAL) = temp;

        temp = ((nes_low & 0x00ff) | (nes_high << 8))–
((nes_lacal&0x00ff)|(nes_hacal<<8));
        NOCACHE_SHORT(NEAR_END_SIGNAL) = temp;

        /* run AEC */

        run_aec_test();
        /* Run audio drive playback speech*/
        while (!(IO_REG(AUDIO_STATUS) & 0x02));
        temp = NOCACHE_SHORT(FAR_END_SIGNAL);
        fes_low = temp;
        fes_high = temp >> 8;

        IO_REG(AUDIO_PIO_DATA) = fes_low;
        IO_REG(AUDIO_PIO_DATA) = fes_high;
```

```
       temp = NOCACHE_SHORT(OUTPUT);

       if (i<999){
       delay_buff[i]=temp;
/*    temp = NOCACHE_SHORT(NEAR_END_SIGNAL);  */
/* Software delay for testing the difference between
   AEC is work or not.*/
       nes_low = delay_buff[i+1] ;
       nes_high = delay_buff[i+1] >> 8;

       IO_REG(AUDIO_PIO_DATA) = nes_low;
       IO_REG(AUDIO_PIO_DATA) = nes_high;
       i++;
       }
       else {
       delay_buff[i]=temp;
       i = 0;
       nes_low = delay_buff[i] ;
       nes_high = delay_buff[i] >> 8;

       IO_REG(AUDIO_PIO_DATA) = nes_low;
       IO_REG(AUDIO_PIO_DATA) = nes_high;
       }
  }
}
run_aec_test()
{
    short *fes_ptr;
    short *nes_ptr;
    short *out_ptr;
    PPCMDBUF          *cmdbuf;
    AEC_PP_ARGBUF     *argbuf;
         cmdbuf = Pp3GetCmdBuf();
                  argbuf = PpCmdBufGetArgs(cmdbuf);
                  argbuf–>fes_src_pkt = fes_ptr;
                  argbuf–>nes_src_pkt = nes_ptr;
                  argbuf–>dst_pkt = out_ptr;
         Pp3CmdBufIssue(AEC);
                 Pp3WaitCompletion();
```

```
}
/* ========================================================================== */
main ()
{
  long    taskId;
  MESSAGE msg;
  long    result;
  unsigned long  i;
  PpResetAll ();
  InterruptInit ();
  InterruptSetVect (_isrPpMsg, 0x000f0000);
  IE |= 0x000f0000;
  TaskInitTasking ();           /* call before any other task functions */
  mySemaId = TaskOpenSema (–1, 1);
  TaskInstallMalloc (myMalloc, myFree);
  taskId = TaskCreate (–1, MPServerTask, (void *)NULL, 16, 4096);
  TaskResume (taskId);
  Pp3Init ();
  TaskSendMsg (&msg, port);
  for (;;) {}                   /* endless loop */
  return ;
}
```

File: pp3func.c

```
/*****************************************************************************
* PP3FUNC.C                                                                  *
 *                                                                           *
* Created by Varadi Gyorgy, Texas Instruments                                *
* Copyright (c) 1995 Texas Instruments                                       *
*                                                                            *
* Last modified: 21. Feb. 1995                                               *
*****************************************************************************/
#include <stddef.h>
#include <task.h>
#include <mp_ppcmd.h>
#include <mp_int.h>
#include ”pp3func.h”
static long     Pp3RunSemaId;
static PPCMDBUF *Pp3CmdBuf;
/*
 *      Initializes the PP3’s command buffers.
 *      If the PP3 initialization is not successful then the return value
 *      is NULL, otherwise is a pointer of the command buffers.
 */
PPCMDBUF *Pp3Init (void)
{
  PPCMDBUF *cmdbuf;
  cmdbuf = PpCmdBufInit (PPNUM, PpCmdInterp, 2);
  PpCmdBufSetArgs (cmdbuf, (void *)(0x1000240 + (PPNUM << 12)));
  cmdbuf = PpCmdBufNext (cmdbuf);
  PpCmdBufSetArgs (cmdbuf, (void *)(0x1000290 + (PPNUM << 12)));
  cmdbuf = PpCmdBufNext (cmdbuf);
  if ((Pp3RunSemaId = TaskOpenSema (–1, 1)) == –1)
    return (NULL);
  PpMsgIntSetSema (PPNUM, INTLEVEL, Pp3RunSemaId);
  Pp3CmdBuf = cmdbuf;
  return (cmdbuf);
}
/*
 *      This function returns the pointer of the PP3’s command buffers.
 */
```

```
PPCMDBUF *Pp3GetCmdBuf (void)
{
  return (Pp3CmdBuf);
}
/*
 *      The Pp3CmdBufIssue function issues a command to the PP3. It's not
 *      waiting for the completion of this command. But before it issues
 *      this command, waits for completion of the last command.
 */
void Pp3CmdBufIssue (void (*func)())
{
  TaskWaitSema (Pp3RunSemaId);
  PpCmdBufSetFunc (Pp3CmdBuf, func);
  PpCmdBufNotifyIssue (Pp3CmdBuf);
  Pp3CmdBuf = PpCmdBufNext (Pp3CmdBuf);
}

/*
 *      This function is waiting for execution of the last command.
 */
void Pp3WaitCompletion (void)
{
  TaskWaitSema (Pp3RunSemaId);
  TaskSignalSema (Pp3RunSemaId);
}
```

File: aec.h

```
#ifndef _AEC_H
#define _AEC_H
typedef struct
{
  void *fes_src_pkt;
                      /* pointer to far end speech input packet – decoded
                         speech */
  void *nes_src_pkt;  /* pointer to near end speech input packet – speech
                         sampled from the microphone at the same time the far
                         end speech is being played through the speaker */
  void *dst_pkt;      /* pointer to output speech packet – raw audio for
                         encoder */
} AEC_PP_ARGBUF;
#endif
```

File: main.h

```
/*
 *------------------------------------------------------------------------
 *         Copyright (C) 1993  Texas Instruments Incorporated.
 *                          All Rights Reserved
 *------------------------------------------------------------------------
 *
 *  main.h  --  C header file for AEC main program
 *
 *------------------------------------------------------------------------
 *  History:
 *    03/24/93...Original version written ................. J. Van Aken
 *    03/03/94...Updated to run under new kernel ........... J. Van Aken
 *    01/10/95...Updated to run a test program ............. Varadi Gyorgy
 *------------------------------------------------------------------------
 */
 /*
 * Message body for the MP server task
 */
typedef struct
{
  long a;
  long b;
  long *result;
  char command;
} MESSAGE;
/*
 * Commands for the MP server task
 */
#define ADD  0x1
#define SUB  0x2
/*
 * Argument passed to newly created server task
 */
typedef struct
{
  long     ppNum;
  void     (*ppProg) ();
```

```
  long     semaId;
  long     portId;
  PPCMDBUF *cmdBuf;
} SRVARG;
/*
 * Declare entry-point addresses to PP-resident routines
 */
extern void PPCMD_ADD ();                  /* command 0 */
extern void PPCMD_SUB ();                  /* command 1 */
```

File: pp3func.h

```
/*****************************************************************************
* PP3FUNC.H                                                                  *
*                                                                            *
* Created by Varadi Gyorgy, Texas Instruments                                *
* Copyright (c) 1995 Texas Instruments                                       *
*                                                                            *
* Last modified: 21. Feb. 1995                                               *
*****************************************************************************/
#ifndef PP3FUNC_H
#define PP3FUNC_H
/* some macros */
#define PPNUM    3
#define INTLEVEL 1
/* Function definitions */
PPCMDBUF *Pp3Init (void);
PPCMDBUF *Pp3GetCmdBuf (void);
void     Pp3CmdBufIssue (void (*)());
void     Pp3WaitCompletion (void);
#endif
```

File: echan.i

```
*******************************************************************************
*       Copyright (C) 1995 Texas Instruments Incorporated.                    *
*                         All Rights Reserved                                 *
*-----------------------------------------------------------------------------*
*                                                                             *
*  echan.i  --  Include file for C80 Acoustic Echo Cancellation software      *
*                                                                             *
*  Environment:                                                               *
*    -- Assemble with versions 1.10 and above of TI's PP assembler.           *
*-----------------------------------------------------------------------------*
*  History:                                                                   *
*    04MAY95...Original version written ................... R. Matusiak     *
*                                        ................... David Qi        *
*******************************************************************************
STACK_SIZE              .set     9
FES_MARGIN              .set    -7
NES_MARGIN              .set    -6
DB_MARGIN               .set    -5
CONST1                  .set    -4
d0_LONG_POWER           .set    -3
d0_SHORT_POWER          .set    -2
d0_VSHORT_POWER         .set    -1
IPR                     .set     1
MODE                    .set     2
NES_HANG                .set     3
FES_HANG                .set     4
DT_HANG                 .set     5
HANG_COUNT              .set     6
SIGN                    .set     7
*-----------------------------------------------------------------------------
*  For Acoustic Echo Cancellation, there are several modes of operation.
*  For program control, we need this mode information.  Each mode will
*  be given one bit of a mode register.  This mode register then could
*  be pushed and popped from the local stack as needed.
*-----------------------------------------------------------------------------
AEC_UPDATE              .set    0
AEC_FILTERING           .set    1
```

```
DOUBLE_TALK             .set    2
NEAR_SPEECH             .set    3
FAR_SPEECH              .set    4
*------------------------------------------------------------------------
*
*  mode[0] – controls AEC coefficient updates.  If set to 1, it indicates
*  that the AEC filter coefficients can be updated for that cycle. If 0,
*  do not update coefficients for that cycle.
*
*  mode[1] – controls AEC filtering.  If set to 1, it indicates
*  that the AEC filter filtering should be performed that cycle. If 0,
*  do not do AEC filtering.
*
*  mode[2] – double talk detector flag.  A 1 indicates double talk has
*  been detected.  A 0 indicates no double talk has been detected.
*
*  mode[3] – near end speech talk detector flag.  1 indicates near end
*  speech has been detected.  A 0 indicates no near end speech has been
*  detected.
*
*  mode[4] – far end speech talk detector flag.  1 indicates far end
*  speech has been detected.  A 0 indicates no far end speech has been
*  detected.
*
*------------------------------------------------------------------------
*------------------------------------------------------------------------
*  The instructions below are used to show how the above defined mode
*  bits are set.  AEC_FILTERING has been chosen for the example, but
*  others would be similar.
*
*       ; Set mode bit to indicate AEC filtering should be performed
*
*       mode = *(sp + [MODE])                ; get mode reg from stack
*       mode = mode | 1\\AEC_FILTERING       ; set AEC filtering mode bit
*       *(sp + [MODE]) = mode                ; put mode reg on stack
*
*       ; Clear mode bit to indicate AEC filtering should not be performed
*
*       mode = *(sp + [MODE])                ; get mode reg from stack
```

```
*       mode = mode & ~(1\\AEC_FILTERING)     ; clear AEC filtering mode bit
*       *(sp + [MODE]) = mode                 ; put mode reg on stack
*
*       ; Using the mode reg. test to see if AEC filtering is to be
*       ; performed.
*
*       mode = *(sp + [MODE])                 ; get mode reg from stack
*       test = mode & 1\\AEC_FILTERING        ; check if AEC filtering
*                                             ; mode bit is set
*       call =[nz] aec_fitering               ; call AEC filtering routine
*       <delay 1>
*       <delay 2>
*-------------------------------------------------------------------------
FILTER_SIZE     .set    512   ; LMS filter size – set for 64 msec echo tail
SAMPLES         .set    1     ; number of speech samples to be processed
STEPSIZE        .set    0x1   ; LMS filter step size.
```

File: ckmem.p

```
*-------------------------------------------------------------------------
*       Copyright (C) 1995 Texas Instruments Incorporated.
*                         All Rights Reserved
*-------------------------------------------------------------------------
*
*  ckmem.p      -- C callable PP assembly language function used to check
*                  for divergence of AEC filter memory.  Used in C80
*                  implementation of AEC.
*
*  Environment:
*    -- Assembles with versions 1.10 and above of TI's PP assembler.
*    -- Allocates with PPCA versions 1.0.
*-------------------------------------------------------------------------
*  History:
*    05MAY95...Original version written ................... R. Matusiak
*-------------------------------------------------------------------------
*************************************************************************
* C PROTOTYPE
*
* void check_filter_memory(short input)
*
*       where input is a sample of far end speech
*
*************************************************************************
*
* DESCRIPTION
*
* Check AEC filter memory values for possible divergence.  This
* is done as a safety measure.  If the memory values become large,
* our chances for overflow increase.  One method for checking filter
* memory divergences is to keep a running sum of the memory values, if
* the sum gets larger than some predefined threshold, then scale down
* the memory values.
*
* Concern – if this is done, do we need to worry about the Q point in
* the remainder of the calculations?
*
```

```
* This only needs to be called when AEC filtering is done.
*
*******************************************************************************
*
* BENCHMARK
*
*          number of cycles = 13 + N
*
*                    where N is the filter order
*
*******************************************************************************
     .include ”echan.i”
input                .set     d2
sum                  .reg     d
mem                  .reg     d
test                 .reg     d
new_mem              .reg     d
La_memory            .reg     la
La_sum               .reg     la
Lx_mem_end           .reg     lx
     .ref L_AEC_FILTER_MEM
     .ref L_SUM
     .entry  d2
     .lock  d1, d6, d7, a4, a12, x0
     .system $check_filter_memory
     .system _check_filter_memory
     .ptext
$check_filter_memory:
_check_filter_memory:
     *(sp + [IPR]) = iprs
     Lx_mem_end = FILTER_SIZE – 1               ; index to get oldest memory
                                                ; value
     La_memory = &*(xba + L_AEC_FILTER_MEM)     ; pointer to filter memory
     sum = *(La_sum = xba + L_SUM)              ; get previous sum,
                                                ; initial value is zero
     sum = sum + |input|                        ; add the newest input (far
                                                ; end) to previous sum
          || mem =h *(La_memory + [Lx_mem_end])  ; get oldest memory value
                                                ; this one is shifted out
```

```
     sum = sum – mem                          ; subtract the oldest memory
                                              ; value from sum
     test = sum – 1\\14                       ; test to see if sum >
                                              ; 0x4000
     br =[lt] *(sp + [IPR])                   ; if sum < 0x4000, no
                                              ; divergence –> no need to
                                              ; scale memory
     lrs0 =[ge] FILTER_SIZE – 2               ; if sum > 0x4000, divide
                                              ; each memory value by 2
                                              ; (right shift by 1)
     le0 = ipe + (MEM_SHIFT_END – $)
          || mem =h *La_memory++
     .cexit
     new_mem = mem >> 1
          || mem =h *La_memory++
MEM_SHIFT:
     new_mem = mem >> 1
          || *(La_memory – [2]) =h new_mem
MEM_SHIFT_END:
     mem =h *La_memory++
     .cjump MEM_SHIFT
     br = *(sp + [IPR])
     new_mem = mem >> 1
          || *(La_memory – [2]) =h new_mem
     *(La_memory – [2]) =h new_mem
     .uexit
     .end
```

File: detector.p

```
*****************************************************************************
*       Copyright (C) 1995 Texas Instruments Incorporated.                  *
*                         All Rights Reserved                               *
*---------------------------------------------------------------------------*
*                                                                           *
*  detector.p  --  Speech detection file for C80 Acoustic Echo Cancellation *
*                  software. (Total 66 instructions.)                       *
*                                                                           *
*  Environment:                                                             *
*    -- Assembles with versions 1.10 and above of TI's PP assembler.        *
*    -- Allocates with PPCA version 1.0.                                    *
*---------------------------------------------------------------------------*
*  History:                                                                 *
*    04JULY95...Original version written ................... R. Matusiak    *
*                                        ................... David Qi       *
*    10/01/95...Modified version runs on SDB................ David Qi       *
*****************************************************************************
; Total instructions 67.
        .include        echan.i
mode            .set    d1
nes_hang        .set    d2
fes_hang        .set    d3
dt_hang         .set    d4
error           .set    d6
prev_err_pwr    .set    d2
err_short_pwr   .set    d5
thresh          .reg    d
prod            .reg    d
test            .reg    d
D               .reg    d
C               .reg    d
fes_margin      .reg    d
nes_margin      .reg    d
fes_vshort_pwr  .reg    d
nes_vshort_pwr  .reg    d
nes_short_pwr   .reg    d
nes_long_pwr    .reg    d
```

```
dummy           .dummy
La_error_prev_pwr       .set    a0
La_error                .reg    la
La_nes_margin           .reg    la
La_fes_margin           .reg    la
Ga_nes_vshort_pwr       .reg    ga
Ga_fes_vshort_pwr       .reg    ga
Ga_nes_short_pwr        .reg    ga
Ga_nes_long_pwr         .reg    ga
Gx_sample               .set    x8
        .ptext
        .entry  x8, d5
        .ref short_power_estimates
        .ref L_ERR_SHORT_POWER_Q31
        .ref L_NEAR_END_SPEECH
        .ref L_FAR_END_SPEECH
        .ref L_NES_SHORT_POWER_Q15
        .ref L_NES_SHORT_POWER_Q31
        .ref L_NES_VSHORT_POWER_Q15
        .ref L_NES_VSHORT_POWER_Q31
        .ref L_NES_LONG_POWER_Q15
        .ref L_NES_LONG_POWER_Q31
        .ref L_ERROR
        .ref L_ERR_SHORT_POWER_Q15
        .ref L_ERR_SHORT_POWER_Q31
        .ref L_FES_SHORT_POWER_Q31
        .ref L_FES_VSHORT_POWER_Q15
        .def SPEECH_DETECTOR
        .def far_end_speech_detector
        .def double_talk_detector
        .def nes_speech_detector
        .def control_lms

SPEECH_DETECTOR:
*******************************************************************************
* Far End Speech Detector:
*
* If the very short power estimate of FES plus some margin is greater than
* the very short power estimate of NES, then FES is detected.  If FES
```

```
* is detected, set FAR_SPEECH mode bit, reset fes hangover counter, then
* branch to double talk detector. If FES is not detected, decrease the FES
* hangover counter by one, clear FAR_SPEECH mode bit, then go to double talk
* detector.
*******************************************************************************
far_end_speech_detector:
;-------------------------------------------------------------------------------
; Far end speech is detected if:
;                   fes_vshort_pwr + fes_margin > nes_vshort_pwr
;-------------------------------------------------------------------------------
       dummy = *(Ga_fes_vshort_pwr = xba + L_FES_VSHORT_POWER_Q15)
       fes_vshort_pwr =h *(Ga_fes_vshort_pwr + [Gx_sample])
       dummy = *(Ga_nes_vshort_pwr = xba + L_NES_VSHORT_POWER_Q15)
       nes_vshort_pwr =h *(Ga_nes_vshort_pwr + [Gx_sample])
            || fes_margin = *(sp + [FES_MARGIN])
       thresh = fes_vshort_pwr + fes_margin
            || mode = *(sp + [MODE])             ; load MODE from stack.
          ;nes_hang = *(sp + [NES_HANG])
       test = thresh – nes_vshort_pwr            ; test fes or not.
            || fes_hang = *(sp + [FES_HANG])     ; load FES_HANG from stack.
;-------------------------------------------------------------------------------
; If fes is detected, do the following:
;-------------------------------------------------------------------------------
      br =[gt] double_talk_detector             ; br to double_talk_detector.
      mode =[gt.nvz] mode | (1\\FAR_SPEECH)     ; set fes mode bit.
             || *(sp + [IPR]) = iprs              ; save iprs to stack.
      fes_hang =[gt]0x50;*(sp+[HANG_COUNT])       ; reset fes_hang to high.
     ;  *(sp + [MODE]) = mode                     ; store back MODE to stack.
     ;  *(sp + [FES_HANG]) = fes_hang             ; store FES_HANG to stack.
       .cjump double_talk_detector
;-------------------------------------------------------------------------------
; If fes is not detected, do the following:
;-------------------------------------------------------------------------------
       mode = mode & ~(1\\FAR_SPEECH)            ; clear far speech mode.
       fes_hang = fes_hang – 1                   ; decrement fes_hang by one.
       ;*(sp + [MODE]) = mode                    ; store back MODE to stack.
       *(sp + [FES_HANG]) = fes_hang             ; store FES_HANG to stack.
```

```
*****************************************************************************
* Double Talk Detector:
*
* Double talk is detected when the error power estimate is greater
* than the near end power estimate.
* ERLE (echo return loss enhancement) = 10 log Pd/Pe
* where Pd is the power estimate of the near end signal and
* Pe is the error power estimate.
*
* When Pe > C*Pd + D (where C is a threshold constant and D is threshold
* constant determined from real time implementation) then double talk
* is detected –> reset DT, FES and NES hangover counters, set DOUBLE_TALK
* mode bit, clear FAR_SPEECH mode bit, clear NEAR_SPEECH mode bit, then
* branch to control_lms.
*
* If Pe <= C*Pd + D then double talk is not detected –> decrease double talk
* hangover counter by one, clear DOUBLE_TALK mode bit, then test FAR_SPEECH
* mode bit. If FAR_SPEECH mode bit is not set, branch to nes_speech_detector,
* otherwise decrease NES hangover counter by one then branch to control_lms.
*****************************************************************************
double_talk_detector:
;-----------------------------------------------------------------------------
; Compute the short power estimate of the error signal
; int power = short_power_estimate(short sample, int prev_pwr)
;-----------------------------------------------------------------------------
       call = short_power_estimates
       error =h *(La_error = xba + L_ERROR)      ; load error.
       prev_err_pwr = *(La_error_prev_pwr = xba + L_ERR_SHORT_POWER_Q31)
                                                 ; load the previous 32 bit
       .uexit
;-----------------------------------------------------------------------------
; Double talk is detected when:
;             err_short_pwr >= C*nes_short_pwr + D
;-----------------------------------------------------------------------------
        .entry  x8, d5
       dummy = *(Ga_nes_short_pwr = xba + L_NES_SHORT_POWER_Q15)
       nes_short_pwr =h *(Ga_nes_short_pwr + [Gx_sample])
                                                 ; load nes_short_pwr.
            || C = *(sp + [CONST1])              ; C = 6
```

```
          ; nes_short_pwr =h1 nes_short_pwr
       prod = nes_short_pwr * C                  ; multiply.
            || D = *(sp + [DB_MARGIN])           ; load D from stack.
       thresh = prod + D                         ; add D to product.
      ;      || mode = *(sp + [MODE])            ; load MODE from stack.
            || dt_hang = *(sp +[DT_HANG])        ; load DT_HANG from stack.
       err_short_pwr = err_short_pwr << 4
       test = err_short_pwr – thresh             ; test double talk or not.
       ;     || fes_hang = *(sp + [FES_HANG])    ; load FES_HANG from stack.
            ||nes_hang = *(sp + [NES_HANG])      ; load NES_HANG from stack.
;-------------------------------------------------------------------------
; If double talk is detected, do the following:
;-------------------------------------------------------------------------
       mode =[gt.nvz] mode | (1\\DOUBLE_TALK)      ; set double talk mode.
            || dt_hang =[gt]*(sp + [HANG_COUNT])   ; reset double talk
                                                   ; counter.
       mode =[gt.nvz] mode & ~(1\\FAR_SPEECH)      ; clear far speech mode.
             fes_hang =[gt]0x50;*(sp+[HANG_COUNT]) ; reset fes_hang.
       mode =[gt.nvz] mode & ~(1\\NEAR_SPEECH)     ; clear near speech mode.
            || nes_hang =[gt]*(sp+[HANG_COUNT])    ; reset nes_hang counter.
       *(sp + [MODE]) = mode                       ; store back MODE to
                                                   ; stack.
       *(sp + [DT_HANG]) = dt_hang                 ; store DT_HANG to stack.
       br =[gt] control_lms                        ; branch to control_lms.
       *(sp + [NES_HANG]) = nes_hang               ; store NES_HANG to stack.
       *(sp + [FES_HANG]) = fes_hang               ; store FES_HANG to stack.
       .cjump control_lms
;-------------------------------------------------------------------------
;If double talk is not detected, do the following:
;-------------------------------------------------------------------------
       mode = mode & ~(1\\DOUBLE_TALK)           ; clear DOUBLE_TALK mode
                                                 ; bit.
       dt_hang = dt_hang – 1                     ; decrement dt_hang by one.
       test = mode & (0x1\\FAR_SPEECH)           ; test FAR_SPEECH mode bit.
;-------------------------------------------------------------------------
; If no far end speech detected:
;-------------------------------------------------------------------------
       br =[z] nes_speech_detector               ; br to nes_speech_detector.
       *(sp + [DT_HANG]) = dt_hang               ; store DT_HANG to stack.
```

```
       *(sp + [MODE]) = mode                    ; store back MODE to stack.
       .cjump nes_speech_detector
;-------------------------------------------------------------------------
; If far end speech detected and no double talk, i.e. not near end speech:
;-------------------------------------------------------------------------
       nes_hang = nes_hang – 1                 ; decrease nes_hang by one.
       br = control_lms                        ; branch to control_lms.
       mode = mode & ~(1\\NEAR_SPEECH)         ; clear near speech mode bit.
           || *(sp + [NES_HANG]) = nes_hang    ; store NES_HANG to stack.
       *(sp + [MODE]) = mode                   ; store back MODE to stack.
       .ujump control_lms
*************************************************************************
* NEAR End Speech Detector:
*
* If the very short power estimate of NES plus some margin is greater or
* equal than the very long power estimate of NES, then NES is detected. If
* NES is detected, reset the NES hangover counter, set NEAR_SPEECH mode bit,
* clear FAR_SPEECH mode bit, clear DOUBLE_TALK mode bit, then branch to
* control_lms.
* If the NES is not detected, decrease NES hangover counter, clear
* NEAR_SPEECH mode bit. If the double talk  hangover counter is greater than
* zero, set AEC_FILTERING mode bit, and clear AEC_UPDATE mode bit, then
* branch out speech detection routine. If the double talk hangover counter is
* less than zero, clear AEC_UPDATE and AEC_FILTERING mode bits, then branch
* out speech detection routine.
*************************************************************************
nes_speech_detector:

;-------------------------------------------------------------------------
; Near end speech is detected when:
;               nes_vshort_pwr + nes_margin >= nes_long_pwr
;-------------------------------------------------------------------------
       dummy = *(Ga_nes_vshort_pwr = xba + L_NES_VSHORT_POWER_Q15)
       nes_vshort_pwr =h *(Ga_nes_vshort_pwr + [Gx_sample])
       dummy = *(Ga_nes_long_pwr = xba + L_NES_LONG_POWER_Q15)
       nes_long_pwr =h *(Ga_nes_long_pwr + [Gx_sample])
            || nes_margin = *(sp + [NES_MARGIN]) ; load nes margin.
       thresh = nes_vshort_pwr + nes_margin
            || mode = *(sp + [MODE])            ; load mode bit.
```

```
       test = thresh – nes_long_pwr             ; test near end speech.
            || dt_hang = *(sp +[DT_HANG])       ; load DT_HANG from stack.
       nes_hang = *(sp + [NES_HANG])            ; load NES_HANG from stack.
;-------------------------------------------------------------------------
; If near speech is detected, do the following:
;-------------------------------------------------------------------------
        mode = [ge.nvz] mode | (1\\NEAR_SPEECH)   ; set near speech mode bit.
            || nes_hang =[ge]*(sp+[HANG_COUNT])   ; reset nes_hang.
        br =[ge] control_lms                      ; branch to control_lms.
        mode = [ge.nvz]mode & ~(1\\FAR_SPEECH)    ; clear far speech mode.
            || *(sp+[NES_HANG]) = nes_hang        ; store NES_HANG to stack.
        mode =[ge] mode & ~(1\\DOUBLE_TALK)       ; clear double talk mode.
            || *(sp+[MODE]) = mode                ; store back MODE to stack.
           .cjump control_lms
;-------------------------------------------------------------------------
; If near speech is not detected, i.e., no speech
; detected at this time, do the following:
;-------------------------------------------------------------------------
;       nes_hang = nes_hang – 1                ; decrement nes_hang by one.
;       br =[ge] control_lms                   ; branch to control_lms.
;       mode = mode & ~(1\\NEAR_SPEECH)        ; clear NEAR_SPEECH mode bit.
;            || *(sp + [NES_HANG]) = nes_hang  ; store NES_HANG to stack.
;       dt_hang = dt_hang                      ; test DT_HANG.
;           .ujump control_lms
;-------------------------------------------------------------------------
; If (DT_HANG >= 0):
;-------------------------------------------------------------------------
;       mode =[ge.nvz] mode | (1\\AEC_FILTERING)  ; set filtering mode bit.
;            || br =[ge] *(sp + [IPR])            ; branch out.
;
;       mode =[ge.nvz] mode & ~(1\\AEC_UPDATE)    ; clear update mode bit.
;       *(sp + [MODE]) = mode                     ; store back MODE to stack.
;       .cexit
;
;-------------------------------------------------------------------------
; If (DT_HANG < 0):
;-------------------------------------------------------------------------
;       mode =[lt.nvz] mode & ~(1\\AEC_FILTERING) ; clear filtering mode bit.
;            || br =[lt] *(sp + [IPR])            ; branch out.
```

```
;
;       mode =[lt] mode & ~(1\\AEC_UPDATE)        ; clear update mode bit.
;       *(sp + [MODE]) = mode                     ; store back MODE to stack.
;       .cexit
*******************************************************************************
* Control LMS:
*
* After any speech detected, go to this part to set the updating and
* filtering mode bit only depends on the DT_HANG and NES_HANG. To do LMS or
* not depends on this subroutine.
*******************************************************************************
control_lms:
       dt_hang = *(sp +[DT_HANG])                ; load DT_HANG from stack.
       dt_hang = dt_hang                         ; test DT_HANG.
               || mode = *(sp + [MODE])          ; load MODE from stack.
;------------------------------------------------------------------------------
; If (DT_HANG >= 0):
;------------------------------------------------------------------------------
       mode =[ge.nvz] mode | (1\\AEC_FILTERING)  ; set filtering mode bit.
            || br =[ge] *(sp + [IPR])            ; branch out.
       mode =[ge.nvz] mode & ~(1\\AEC_UPDATE)    ; clear update mode bit.
            || nes_hang = *(sp + [NES_HANG])     ; load NES_HANG from stack.

       nes_hang = nes_hang                       ; test NES_HANG.
            || *(sp + [MODE]) = mode             ; store back MODE to stack.
       .cexit
;------------------------------------------------------------------------------
; If (DT_HANG < 0) and (NES_HANG < 0):
;------------------------------------------------------------------------------
       mode =[lt.nvz] mode | (1\\AEC_FILTERING)  ; set filtering mode bit.
            || br =[lt] *(sp + [IPR])            ; branch out.
       mode =[lt.nvz] mode | (1\\AEC_UPDATE)     ; set update mode bit.
       *(sp + [MODE]) = mode                     ; store back MODE to stack.
       .cexit
;------------------------------------------------------------------------------
; If (DT_HANG < 0) and (NES_HANG >= 0):
;------------------------------------------------------------------------------
       mode =[ge.nvz] mode & ~(1\\AEC_FILTERING) ; clear filtering mode bit.
            || br =[ge] *(sp + [IPR])            ; branch out.
```

```
mode =[ge] mode & ~(1\\AEC_UPDATE)        ; clear update mode bit.
*(sp + [MODE]) = mode                     ; store back MODE to stack.
.cexit
```

File: echan.p

```
*******************************************************************************
*       Copyright (C) 1995 Texas Instruments Incorporated.                    *
*                         All Rights Reserved                                 *
*-----------------------------------------------------------------------------*
*                                                                             *
*  echan.p  --  Main functions file for C80 Acoustic Echo Cancellation        *
*               software. (Total 84 instructions.)                            *
*                                                                             *
*  Environment:                                                               *
*    -- Assembles with versions 1.10 and above of TI's PP assembler.          *
*    -- Allocates with PPCA version 1.0.                                      *
*-----------------------------------------------------------------------------*
*  History:                                                                   *
*    04JULY95...Original version written ................. R. Matusiak      *
*                                        ................. David Qi         *
*    10/01/95...Modified version runs on SDB.............. David Qi         *
*******************************************************************************
        .include        echan.i
erf             .set    d1
fes             .set    d6
prev_fes_pwr    .set    d2
r_hat           .set    d5
;arg1           .set    d1
mode            .reg    d
test            .reg    d
ue              .reg    d
ueM1            .reg    d
result          .reg    d
prod            .reg    d
;stepsize       .reg    d
error           .reg    d
r               .reg    d
nes             .reg    d
speech_out      .reg    d
fes_short_pwr   .reg    d
nes_and_echo    .reg    d
fes_signal      .reg    d
```

```
nes_signal              .reg    d
hang_counter            .set    d3
nes_margin              .set    d3
fes_margin              .set    d2
constant_C              .set    d4
db_margin               .set    d4
La_margin               .set    a1
La_hang                 .set    a1
dummy           .dummy

La_near_end_speech      .set    a1
La_mem                  .set    a2
Ga_coeff                .set    a8
;Ga_struct              .set    a9  ; input pointer to structure.
La_prev_fes_pwr         .reg    la
La_pwr_Q15              .reg    la
La_error                .reg    la
La_filter_out           .reg    la
La_NEAR_END_SIGNAL      .reg    la
La_FAR_END_SIGNAL       .reg    la
La_nes_and_echo         .reg    la
La_SPEECH_OUT           .reg    la
Ga_fes_signal           .reg    ga
Ga_fes                  .reg    ga
Lx_sample               .set    x0
Gx_sample               .set    x8
Lx_count                .set    x2
Lx_mem_end              .reg    lx
        .ptext
        .system $AEC
        .system _AEC
        .ref nes_power_estimates
        .ref $lms_filter
;        .ref $check_filter_memory
        .ref short_power_estimates
        .ref fes_vshort_power_estimates
        .ref far_end_speech_detector
        .ref double_talk_detector
        .ref nes_speech_detector
```

```
        .ref L_AEC_FILTER_MEM
        .ref L_AEC_COEFFICIENTS
        .ref L_ERROR
        .ref L_NEAR_END_SPEECH
        .ref L_FAR_END_SPEECH
        .ref L_SPEECH_OUT
        .ref L_FES_SHORT_POWER_Q31
        .ref L_FILTER_OUTPUT
        .ref _NEAR_END_SIGNAL
        .ref _FAR_END_SIGNAL
        .ref _OUTPUT
        .ref echan_restore_state
        .ref get_speech
        .ref $AEC_State_Init
        .ref SPEECH_DETECTOR
        .ref _AEC_State_Init
        .def NEXT_CYCLE
        .def OUTPUT_CALC
$AEC:
_AEC:
        .entry  a9
     *(sp –= [STACK_SIZE]) = iprs     ; put function return pointer on stack
*-------------------------------------------------------------------------------
* This function call to restore all AEC required data from external
* memory. Not used in here because this demo code is AEC stand alone.
*-------------------------------------------------------------------------------
;     call = echan_restore_state
;     nop
;     nop
;     .uexit
*-------------------------------------------------------------------------------
* Initialize variables in internal memory.
*-------------------------------------------------------------------------------
src             .set    d1
dst             .set    d2
count   .set    d3
    Lx_count = SAMPLES                      ; local index counter.
    Lx_sample = 0                           ; local index sample counter.
    fes_margin = 0x2                        ; fes margin.
```

```
    nes_margin = 0x200                      ; nes margin.
    ;*(sp + [NES_MARGIN]) = nes_margin      ; store nes margin to stack.
     ;  || *(sp + [FES_MARGIN]) = fes_margin ; store fes margin to stack.
    hang_counter = 0x258                    ; hangover counter.
    constant_C = 0x06                       ; load constant C
                                            ; ERLE/10
                                            ; C = 10  ,ERLE = 8db
    La_NEAR_END_SIGNAL = _NEAR_END_SIGNAL   ; point to nes_speech.

     *(sp + [HANG_COUNT]) = hang_counter    ; store hangover counter.
        || *(sp + [CONST1]) = constant_C    ; store constant C.
     db_margin = 0x500                      ; double talk margin.
     count = 2*SAMPLES                      ; load packet transfer counter.
*-------------------------------------------------------------------------------
* This function call get near-end-signal from external memory by
* using packet transfer.
*-------------------------------------------------------------------------------
     call = get_speech                      ; call get nes.
     src = La_NEAR_END_SIGNAL               ; source address.
       ;|| *(sp + [DB_MARGIN]) = db_margin  ; store margin D to stack.
     dst = &*(xba + L_NEAR_END_SPEECH)      ; destination address.
     .cexit  d1,d2
*-------------------------------------------------------------------------------
        .entry  a9
     La_FAR_END_SIGNAL = _FAR_END_SIGNAL
                                            ; point to fes_speech.
     count = 2*SAMPLES                      ; load packet transfer counter.
*-------------------------------------------------------------------------------
* This function call get far-end-signal from external memory by
* using packet transfer.
*-------------------------------------------------------------------------------
     call = get_speech                      ; call get fes.
     src = La_FAR_END_SIGNAL                ; source address.
     dst = &*(xba + L_FAR_END_SPEECH)       ; destination address.
     .cexit  d1,d2
*-------------------------------------------------------------------------------
* This function call to calculate near-end-signal power estimate for
* all windows size.
*-------------------------------------------------------------------------------
```

```
     call = nes_power_estimates              ; calculate # of samples
                                             ; nes_pwr,
     nop                                     ; include short, very short and
     nop                                     ; long windows.
     .cexit
*-------------------------------------------------------------------------
* This function call to calculate far-end-signal power estimate for
* very short windows size.
*-------------------------------------------------------------------------
     call = fes_vshort_power_estimates       ; calculate # of samples very
                                             ; short window power.
     Gx_sample = 0                           ; global index sample counter.
     nop
     .cexit
*************************************************************************
* AEC Filtering
*
* We first need to check if we need to do AEC filtering for this cycle.
* We do AEC filtering if Far End Speech (FES) of the previous cycle was
* detected. This implies that either FES or Double Talk (DT) was detected
* on the previous cycle.  AEC filtering is only turned off when FES is
* NOT detected for 600 consecutive FES samples.  If AEC filtering is not
* to be done in this cycle then we know no AEC coefficient update will
* be done either, therefore, we do not need to check if coefficient
* updating should be done, just go to error signal power estimation code.
*
* AEC Coefficient Update
*
* Do AEC coefficient update if far end speech is detected and NO double
* talk is detected.
*
* The AEC filtering and coefficient updating are done in the same loop
* to reduce processing when both are needed to be done.  If only AEC
* filtering is needed (meaning no updating of the coefficients), we
* still use the same routine, only we zero the error term which reduced
* the update equation to ak_new = ak_old, thus not changing them.
*************************************************************************
*-------------------------------------------------------------------------
* If we proceed to this part of the code, AEC coefficient update is going
```

```
* to be done or not depend on mode bit.  We first need to calculate the
* short power estimate of the FES. Second we need to decide if we will use
* LMS or NLMS method to update the coefficients. The function prototype of
* the short power estimate is given below:
*
* int power = short_power_estimate(short sample, int prev_pwr)
*-------------------------------------------------------------------------
NEXT_CYCLE:
       Ga_fes = &*(xba + L_FAR_END_SPEECH)  ; Ga_fes –> buffer that contains
                                            ; far end speech
       call = short_power_estimates
       prev_fes_pwr = *(La_prev_fes_pwr = xba + L_FES_SHORT_POWER_Q31)
                                            ; load the previous 32 bit
                                            ; short power estimate of the FES
       fes =h *(Ga_fes + [Gx_sample])       ; load a FES sample
       .cexit
*-------------------------------------------------------------------------
* In this part we are going to check the MODE bit. If in filtering mode, we
* freeze the coefficients by error equal to zero. If in updating mode, do
* NLMS. If both mode not set, set filter output equal to zero.
*-------------------------------------------------------------------------
                .entry a9, x0
       mode = *(sp + [MODE])                ; load MODE from stack.
       test = mode & 1\\AEC_FILTERING       ; check if AEC filtering
                                            ; mode bit is set
       br =[z] OUTPUT_CALC                  ; branch to OUTPUT_CALC routine
                                            ; if AEC filtering bit is not
                                            ; set

       r_hat = 0                            ; we do not do AEC filtering
                                            ; therefore echo estimate is 0
       nop
       .cjump OUTPUT_CALC
       test = mode & 1\\AEC_UPDATE          ; check if AEC coefficient
                                            ; update mode bit is set
       br =[z] CHECK_FILTER_MEM
       nop
       erf =[z] 0                           ; if AEC_UPDATE bit is not
                                            ; set, set erf = 0.  This
```

```
                                          ; causes the AEC coefficients
                                          ; to remain unchanged when
                                          ; lms_filter() is called.
       .cjump CHECK_FILTER_MEM
*-------------------------------------------------------------------------
* We always use NLMS to update coefficients.
*
* erf = error*stepsize/fes_short_pwr
*
* The divi function we use here only works when the dividend is less than the
* devisor and both are positive numbers. The dividend is STEPSIZE and also
* add it to fes_short_pwr to make sure the devisor is never zero and larger
* than the dividend. The output result is Q15 format.
*-------------------------------------------------------------------------
       ue = STEPSIZE                              ; load stepsize.
       fes_short_pwr = *(La_prev_fes_pwr = xba + L_FES_SHORT_POWER_Q31)

       fes_short_pwr = fes_short_pwr>>16          ; load fes power Q31
                                                  ; and right shift
                                                  ; 16 bit.
       fes_short_pwr = fes_short_pwr + ue         ; make sure devisor not
                                                  ;  equal to zero.
       fes_short_pwr = –fes_short_pwr             ; negate the divisor.
         || mf = &*(0)                            ; clear mf register.
       lrse2 = 12                                 ; 13 loops + 2 delay
                                                  ; slots == 15 divis
       erf = divi(fes_short_pwr, ueM1 = ue)       ; 1st divi. iterate.
       erf = divi(fes_short_pwr, ueM1 = ue [n] ueM1) ; 2nd divi. iterate.
DIVI_LOOP:
       erf = divi(fes_short_pwr, ueM1 = ue [n] ueM1) ; 3rd through 15th
                                                  ; divis iterate.
       .cjump DIVI_LOOP
       result = mf                                ; result equal to mf.
       error =h *(La_error = xba + L_ERROR)       ; load error.
       erf = result*error                         ; erf=stepsize*error
           erf = erf >> 15                        ; change erf back to
Q15.
CHECK_FILTER_MEM:
```

```
*-------------------------------------------------------------------------
* Check AEC filter memory values for possible divergence.  This is done
* as a safety measure.  If the memory values become large, our chances for
* overflow increase.  One method for checking filter memory divergences
* is to keep a running sum of the memory values, if the sum gets larger
* than some predefined threshold, then scale down the memory values.
* The function prototype for checking the filter memory is:
*
* void check_filter_memory(short input, int length)
*-------------------------------------------------------------------------
;       call = $check_filter_memory    ; Didn't use because no overflow found
;       fes =h *(Ga_fes + [Gx_sample]) ; in real time testing.
;       Lx_mem_end = FILTER_SIZE – 1

*-------------------------------------------------------------------------
* Call LMS filter routine.  This routine updates the AEC filter coefficients
* if the erf term is non–zero, then performs filtering with these updated
* coefficients.  The prototype of this function is:
*
* short output = lms_filter(int filter_size, short *coefficients, short *mem)
*-------------------------------------------------------------------------
dummy =h *(La_far_end_speech = xba + L_FAR_END_SPEECH)
       dummy = *(La_mem = xba + L_AEC_FILTER_MEM)
       nes =h *(La_far_end_speech += [Lx_sample])
       call = $lms_filter
       Ga_coeff =&*(xba + L_AEC_COEFFICIENTS)
       *La_mem =h nes
       .cexit  d1
*-------------------------------------------------------------------------
* At this part of program, we calculate the output signal. If the update
* mode is set, that means far–end only signal is detected. The output equal
* to error and suppressed by –24db.
* If update mode is not set, i.e. in double talk or near–end speech only
* mode. The output signal equal to near–end signal and no any suppress.
*-------------------------------------------------------------------------
OUTPUT_CALC:
                .entry d5, a9, x0
       dummy =h *(La_near_end_speech = xba + L_NEAR_END_SPEECH)
       r =h *(La_near_end_speech + [Lx_sample])
```

```
        error =h r – r_hat
          || mode = *(sp + [MODE])            ; get mode bit from stack

        *(La_error = xba + L_ERROR) =h error  ; store error to memory.
        test = mode & 1\\AEC_UPDATE           ; check if far end speech only
                                              ; was detected
        speech_out =[nz.nvz] error >> 4       ; if only far end speech was
                                              ; detected, suppress the output
                                              ; signal
        La_SPEECH_OUT =&*(xba + L_SPEECH_OUT) ; point to out_speech buffer.

        speech_out =[z] r;error               ; we may want to try other
                                              ; equations here to improve
                                              ; echo suppression
        *(La_SPEECH_OUT + [Lx_sample]) =h speech_out
                                              ; store output speech.
        count = 2*SAMPLES                     ; load packet transfer counter.

*-------------------------------------------------------------------------
* This function call put output signal from on chip data RAM to external
* memory by using packet transfer.
*-------------------------------------------------------------------------
        call = get_speech                     ; call get nes.
        src = La_SPEECH_OUT                   ; source address.
        dst = _OUTPUT                         ; destination address.
        call = SPEECH_DETECTOR
            nop
            nop
        .uexit
                .entry  x0, x2, x8
        Lx_sample = Lx_sample + 1             ; local index sample + 1.
        Lx_count = Lx_count – 1               ; local index count – 1.
        br =[nz] NEXT_CYCLE
        br =[z] *sp
        Gx_sample = Gx_sample + 1             ; global index sample + 1.
        .cjump NEXT_CYCLE
        a15 = *(sp+=[STACK_SIZE])             ; reset stack pointer.
        .cexit
```

File: lms.p

```
*****************************************************************************
*            Copyright (C) Texas Instruments Incorporated.                  *
*                         All Right Reserved                                *
*---------------------------------------------------------------------------*
*                                                                           *
* lms.p   -- Least-mean-squared(LMS) algorithm implemented on one of the    *
*            the C80's Parallel Processors (PP).                            *
*                                                                           *
* Environment:                                                              *
*         -- This PP-callable code executes on a PP (TMS320C80 devices).    *
*         -- Allocates with version 1.01 and above.                         *
*         -- Assembles with versions 1.10 and above of TI's PP assembler.   *
*---------------------------------------------------------------------------*
*  History:                                                                 *
*    04JULY95...Original version written ..................... R. Matusiak    *
*    10/01/95...Modified version runs on SDB.................. David Qi       *
*****************************************************************************
*
*  BENCHMARK
*
*     Number of cycles =  4 * N/2  + 19
*
*             Where, (Assume no ICACHE misses)
*                    N is number of filter order (must be even)
*
*************************************************************************
*
*  MEMORY
*
*     Program Memory:        19*8 = 152 (bytes)
*     On-chip Data RAM:      4*M + 4*N + 2*4*M (bytes)
*     Parameter RAM:         0 bytes
*
*************************************************************************
;File name : lms.p
;FIR filter and LMS coefficients update.
;                               type              description
```

```
;
    .include  echan.i
prod_1          .reg    d ; intermediate        product reg.
prod_2          .reg    d ; intermediate        product reg.
up_prod_1       .reg    d ; intermediate        update coeff. product reg
up_prod_2       .reg    d ; intermediate        update coeff. product reg
ak_new_1        .reg    d ; output              filter new coeff. reg.
ak_new_2        .reg    d ; output              filter new coeff. reg.
ak_old_1        .reg    d ; input/output        filter old coeff. reg.
ak_old_2        .reg    d ; input/output        filter old coeff. reg.
x_1             .reg    d ; input               filter memory reg.
x_2             .reg    d ; input               filter memory reg.
;input          .set    d2 ; input              input samples.
erf             .set    d1 ; input              constant in time i.
N               .reg    d ; input               number of filter taps – must
                          ;                     be even!!!
y               .set    d5; output              fir–filter output.
Ga_ak           .set    a8; input               pointer to filter coeff.
Ga_output       .reg    ga; output              pointer to output buffer.
La_x            .set    a2; input               pointer to filter memory.
La_input        .set    a1; input               pointer to input samples.
;La_x_start     .reg    la; input               pointer to start of filter
                          ;                     memory.
Lx_N            .set    x1; constant            holds the filter order.
Gx_N            .set    x10; constant           holds the filter order.
                .ref L_FILTER_OUTPUT
        .system  $lms_filter
        .system  _lms_filter
        .ptext
        .lock    d0, a4, a12
        .entry   a1, a2, a8, d1,prod_2
$lms_filter:
_lms_filter:
                                              ; set EALU label.
        Lx_N = FILTER_SIZE
        Gx_N = Lx_N – 1                       ; adjust offset.
        d0 = SHIFT_ADD
        ak_old_1 =h *(Ga_ak += [Gx_N])        ; get last coeff. in memory.
          || x_1 =h *(La_x += [Lx_N])         ; get last number in filter
```

```
                                        ; memory.
        le0 = LMS_LOOP_END              ; set fir loop end label.
        y = 1\\14                       ; clear accumulator.
          || *(La_x + [1]) =h x_1       ; shift memory.
          || prod_2 =&*(0)

        up_prod_1 =r (x_1 * erf)<<1     ; get first update product
          || ealu(SHIFT_ADD)
          || x_2 =h *--La_x
        lrs0 = (FILTER_SIZE/2) – 2      ; set fir loop count.
        up_prod_2 =r (x_2 * erf)<<1     ; get second update product.
          || ak_new_1 = ealu(SHIFT_ADD:ak_old_1+up_prod_1>>16)
                                        ; update coeff.
          || ak_old_2 =h *--Ga_ak       ; pointer to next coeff.
          || *(La_x + [1]) =h x_2       ; shift memory.
        prod_1 = x_2 * ak_new_1
          || *(Ga_ak + [1]) =h ak_new_1
          || x_1 =h *--La_x
LMS_LOOP_START:                         ; fir loop starts here.
        up_prod_1 =r (x_1 * erf)<<1     ; get first update product.
          || ak_new_2 = ealu(SHIFT_ADD:ak_old_2+up_prod_2>>16)
                                        ; update coeff.
          || ak_old_1 =h *--Ga_ak       ; load next coefficient.
          || *(La_x + [1]) =h x_1       ; shift memory.
        prod_2 = x_1 * ak_new_2         ; get second multiplication
                                        ; product.
          || y = y + prod_1             ; accumulate filter output.
          || *(Ga_ak + [1]) =h ak_new_2 ; load next coeff.
          || x_2 =h *--La_x             ; load next filter memory number.
        up_prod_2 =r (x_2 * erf)<<1     ; get second update product.
          || ak_new_1 = ealu(SHIFT_ADD:ak_old_1+up_prod_1>>16)
                                        ; update coeff.
          || ak_old_2 =h *--Ga_ak       ; pointer to next coeff.
          || *(La_x + [1]) =h x_2       ; shift memory.
LMS_LOOP_END:
        prod_1 = x_2 * ak_new_1         ; get first multiplication result.
          || y = y + prod_2             ; accumulate output
          || *(Ga_ak + [1]) =h ak_new_1 ; store new coefficient.
          || x_1 =h *--La_x             ; load next memory sample.
```

```
    .cjump LMS_LOOP_START

    up_prod_1 =r (x_1 * erf)<<1
      || ak_new_2 = ealu(SHIFT_ADD:ak_old_2+up_prod_2>>16)
                                    ; update coeff.
    prod_2 = x_1 * ak_new_2         ; get second multiplication product.
      || y = y + prod_1             ; accumulate filter output.
      || *Ga_ak =h ak_new_2         ; load next coeff.
      || *(La_x + [1]) =h x_1       ; shift memory.
    y = y + prod_2                  ; get final result.
    br = iprs
    y = y >> 15                     ; right-shift output 15 bits.
*(Ga_output = xba + L_FILTER_OUTPUT) =h y
    .uexit
```

File: power.p

```
*******************************************************************************
*       Copyright (C) 1995 Texas Instruments Incorporated.                    *
*                         All Rights Reserved                                 *
*-----------------------------------------------------------------------------*
*                                                                             *
*  power.p  --  Speech power calculation file for C80 Acoustic Echo           *
*               Cancellation software. (Total 42 instructions.)               *
*                                                                             *
*  Environment:                                                               *
*    -- Assemble with versions 1.10 and above of TI's PP assembler.           *
*    -- Allocate with PPCA version 1.01 and above.                            *
*-----------------------------------------------------------------------------*
*  History:                                                                   *
*    04JULY95...Original version written ..................... R. Matusiak    *
*                                        ..................... David Qi       *
*******************************************************************************
     .include echan.i
     .ref L_NES_SHORT_POWER_Q31
     .ref L_NES_LONG_POWER_Q31
     .ref L_NES_VSHORT_POWER_Q31
     .ref L_NES_SHORT_POWER_Q15
     .ref L_NES_LONG_POWER_Q15
     .ref L_NES_VSHORT_POWER_Q15
     .ref L_FES_VSHORT_POWER_Q31
     .ref L_FES_VSHORT_POWER_Q15
;     .ref L_FES_SHORT_POWER_Q15
     .ref L_FES_SHORT_POWER_Q31
     .ref L_ERR_SHORT_POWER_Q15
     .ref L_ERR_SHORT_POWER_Q31
     .ref L_NEAR_END_SPEECH
     .ref L_FAR_END_SPEECH
     .ref L_ERROR
     .def nes_power_estimates
     .def fes_vshort_power_estimates
     .def short_power_estimates
dummy                .dummy
s_pwr_2_16           .reg     d
```

```
l_pwr_2_16             .reg     d
v_pwr_2_16             .reg     d
nes_squared            .reg     d
fes_squared            .reg     d
nes                    .reg     d
fes                    .reg     d
nes_long_pwr           .reg     d
nes_short_pwr          .reg     d
fes_short_pwr          .reg     d
nes_vshort_pwr         .reg     d
fes_vshort_pwr         .reg     d
La_nes_long_pwr_Q15    .reg     la
La_nes_short_pwr_Q15   .reg     la
La_nes_vshort_pwr_Q15  .reg     la
La_nes_vshort_pwr_Q31  .reg     la
La_fes_vshort_pwr_Q15  .reg     la
La_fes_vshort_pwr_Q31  .reg     la
Ga_nes_long_pwr_Q31    .reg     ga
Ga_nes_short_pwr_Q31   .reg     ga
Ga_nes                 .reg     ga
Ga_fes                 .reg     ga
Gx_fes                 .reg     gx
     .ptext
     .lock d0
     .entry  s_pwr_2_16, l_pwr_2_16, v_pwr_2_16
*************************************************************************
* Very short, short and long window power estimate of near end speech is used
* for all speech detection algorithm.
*                                        2
*   Vshort_nes_pwr = (1–B)*V_prev_nes_pwr + B*sample  , B = 1/32 , 4msec
*   Short_nes_pwr = (1–B)*S_prev_nes_pwr + B*sample  , B = 1/128 , 16msec
*   Long_nes_pwr = (1–B)*L_prev_nes_pwr + B*sample  , B = 1/16384 , 2048msec
*************************************************************************
nes_power_estimates:
     d0 = SHORT_POWER
     dummy = ealu(SHORT_POWER: s_pwr_2_16 – s_pwr_2_16>>7)
          || *(sp + [d0_SHORT_POWER]) = d0
     d0 = LONG_POWER
     dummy = ealu(LONG_POWER: l_pwr_2_16 – l_pwr_2_16>>14)
```

```
          || *(sp + [d0_LONG_POWER]) = d0
     d0 = VSHORT_POWER
     dummy = ealu(VSHORT_POWER: v_pwr_2_16 – v_pwr_2_16>>5)
          || *(sp + [d0_VSHORT_POWER]) = d0
     La_nes_long_pwr_Q15 = &*(xba + L_NES_LONG_POWER_Q15)
     La_nes_short_pwr_Q15 = &*(xba + L_NES_SHORT_POWER_Q15)
     La_nes_vshort_pwr_Q15 = &*(xba + L_NES_VSHORT_POWER_Q15)

     s_pwr_2_16 = *(Ga_nes_short_pwr_Q31 = xba + L_NES_SHORT_POWER_Q31)
          ; initialize pointer to previous short power estimate of
          ; near end signal, and load to a register
     l_pwr_2_16 = *(Ga_nes_long_pwr_Q31 = xba + L_NES_LONG_POWER_Q31)
          ; initialize pointer to previous long power estimate of
          ; near end signal, and load to a register
     v_pwr_2_16 = *(La_nes_vshort_pwr_Q31 = xba + L_NES_VSHORT_POWER_Q31)
          ; initialize pointer to previous very short power estimate of
          ; near end signal, and load to a register
     nes =h *(Ga_nes = xba + L_NEAR_END_SPEECH)
     lrs0 = SAMPLES – 1
     le0 = NES_POWER_EST_LOOP_END
     dummy =h *––La_nes_long_pwr_Q15
          || d0 = *(sp + [d0_SHORT_POWER])
          ; load d0 with alu configuration for power estimate
NES_POWER_EST_LOOP:
     nes_squared = nes * nes
       || s_pwr_2_16 =ealu(SHORT_POWER: s_pwr_2_16 – s_pwr_2_16>>7)
     s_pwr_2_16 = s_pwr_2_16 + nes_squared>>7
       || *La_nes_long_pwr_Q15++ =h nes_long_pwr     ; store near end long
                                                     ; power
     nes_long_pwr = nes_long_pwr – s_pwr_2_16>>14
       ||  nes_short_pwr =h1 s_pwr_2_16              ; get upper half of nes
                                                     ; short power estimate
       || d0 = *(sp + [d0_VSHORT_POWER])
     v_pwr_2_16 =ealu(VSHORT_POWER: v_pwr_2_16 – v_pwr_2_16>>5)
       || nes_long_pwr =h1 l_pwr_2_16                ; get upper half of nes
                                                     ; long power estimate
     v_pwr_2_16 = v_pwr_2_16 + nes_squared>>5
       || *La_nes_short_pwr_Q15++ =h nes_short_pwr   ; store near short
                                                     ; power
```

```
       || d0 = *(sp + [d0_LONG_POWER])
     l_pwr_2_16 =ealu(LONG_POWER: l_pwr_2_16 – l_pwr_2_16>>14)
       why(l_pwr_2_16,l_pwr_2_16\\d0,%d0)
       || nes_vshort_pwr =h1 v_pwr_2_16
       || d0 = *(sp + [d0_SHORT_POWER])

NES_POWER_EST_LOOP_END:
     l_pwr_2_16 = l_pwr_2_16 + nes_squared>>14
       || nes =h *++Ga_nes                          ; load a near end
                                                    ; signal sample
       || *La_nes_vshort_pwr_Q15++ =h nes_vshort_pwr  ; store v_short power

     .cjump NES_POWER_EST_LOOP
     nes_long_pwr =h1 l_pwr_2_16                    ; get upper half word
                                                    ; (Q15).
     br = iprs
     *Ga_nes_long_pwr_Q31 = l_pwr_2_16               ; store nes long power
                                                    ; (Q31).
       || *La_nes_vshort_pwr_Q31 = v_pwr_2_16; store nes very short power.

     *Ga_nes_short_pwr_Q31 = s_pwr_2_16
       || *La_nes_long_pwr_Q15++ =h nes_long_pwr

     .uexit
*************************************************************************
* Very short window power estimate of far end speech is used
* for far end talk detection algorithm.
*                                      2
*   Vshort_fes_pwr = (1–B)*prev_fes_pwr + B*sample  , B = 1/32 , 4msec
*************************************************************************
fes_vshort_power_estimates:
     fes =h *(Ga_fes = xba + L_FAR_END_SPEECH)
     v_pwr_2_16 = *(La_fes_vshort_pwr_Q31 = xba + L_FES_VSHORT_POWER_Q31)
     La_fes_vshort_pwr_Q15 = &*(xba + L_FES_VSHORT_POWER_Q15)
     lrs0 = SAMPLES – 1
     le0 = ipe + (FES_VSHORT_POWER_EST_LOOP_END – $)
          || d0 = *(sp + [d0_VSHORT_POWER])
     dummy =h *––La_fes_vshort_pwr_Q15
FES_VSHORT_POWER_EST_LOOP:
```

```
     fes_squared = fes * fes
          || v_pwr_2_16 = ealu(VSHORT_POWER: v_pwr_2_16 – v_pwr_2_16>>5)
          || fes_vshort_pwr =h1 v_pwr_2_16
FES_VSHORT_POWER_EST_LOOP_END:
     v_pwr_2_16 = v_pwr_2_16 + fes_squared>>5
          || *La_fes_vshort_pwr_Q15++ =h fes_vshort_pwr
          || fes =h *++Ga_fes
     .cjump FES_VSHORT_POWER_EST_LOOP
     br = iprs

     fes_vshort_pwr =h1 v_pwr_2_16
          || *La_fes_vshort_pwr_Q31 = v_pwr_2_16
     *La_fes_vshort_pwr_Q15++ =h fes_vshort_pwr
     .uexit
```

```
*****************************************************************************
* Short window power estimate of far end speech and error signal is used
* for NLMS and double talk detection algorithm.
*                                        2
*   short_pwr = (1–B)*prev_pwr + B*sample  , B = 1/128 , 16 msec
*****************************************************************************
short_power_estimates:

sample              .set       d6
prev_pwr_31         .set       d2
La_prev_pwr_Q31     .set       a0
power               .set       d5
sample_squared      .set       d7

        .ptext
        .lock d0
        .entry  d6,d2, a0

        d0 = *(sp + [d0_SHORT_POWER])
        sample_squared = sample * sample
          || prev_pwr_31 = ealu(SHORT_POWER: prev_pwr_31 – prev_pwr_31>>7)
          || br = iprs
        prev_pwr_31 = prev_pwr_31 + sample_squared>>7
        power =h1 prev_pwr_31
          || *La_prev_pwr_Q31 = prev_pwr_31
        .uexit
```

File: aec_vars.s

```
*------------------------------------------------------------------------
*       Copyright (C) 1995 Texas Instruments Incorporated.
*                         All Rights Reserved
*------------------------------------------------------------------------
*
*  aec_vars.s  -- Assembly language declaration of variables and arrays
*                 for the C80 implementation of AEC.
*
*  Environment:
*    -- Assemble with versions 1.10 and above of TI's PP assembler.
*------------------------------------------------------------------------
*  History:
*    05MAY95...Original version written ................... R. Matusiak
*------------------------------------------------------------------------
      .include "echan.i"
      .def L_AEC_FILTER_MEM
      .def L_SUM
          .def L_NEAR_END_SPEECH
          .def L_FAR_END_SPEECH
      .def L_NES_SHORT_POWER_Q15
      .def L_NES_SHORT_POWER_Q31
          .def L_NES_VSHORT_POWER_Q15
          .def L_NES_VSHORT_POWER_Q31
          .def L_NES_LONG_POWER_Q15
          .def L_NES_LONG_POWER_Q31
      .def L_FES_VSHORT_POWER_Q15
      .def L_FES_VSHORT_POWER_Q31
      .def L_ERR_SHORT_POWER_Q31
      .def L_ERR_SHORT_POWER_Q15
      .def L_FES_SHORT_POWER_Q31
;          .def L_FES_SHORT_POWER_Q15
      .def L_ERROR
      .def L_FILTER_OUTPUT
      .def L_SPEECH_OUT
      .def L_AEC_COEFFICIENTS
      .def L_FES_MARGIN
      .def L_NES_MARGIN
```

```
;           .def HANG_CONSTANT
;           .def SAMPLE_COUNT
      .def FES_HANG
      .def ECHAN0
      .def ECHAN1
      .def ECHAN2
      .def GEN_PTR
L_AEC_FILTER_MEM        .usect   "AEC_MEM",2*FILTER_SIZE+4,4
L_AEC_COEFFICIENTS      .usect   "COEFF",2*FILTER_SIZE+4,4
L_SUM                   .usect   "SUM",4,4
L_NEAR_END_SPEECH       .usect   "NES",2*SAMPLES,4
L_FAR_END_SPEECH        .usect   "FES",2*SAMPLES,4
L_NES_SHORT_SPACE       .usect   "NESSPCE",4,4
L_NES_SHORT_POWER_Q15   .usect   "NESPEQ15",2*SAMPLES,4
L_NES_SHORT_POWER_Q31   .usect   "NESPEQ31",4,4
L_NES_VSHORT_SPACE      .usect   "NEVSPCE",4,4
L_NES_VSHORT_POWER_Q15  .usect   "NEVPEQ15",2*SAMPLES,4
L_NES_VSHORT_POWER_Q31  .usect   "NEVPEQ31",4,4
L_NES_LONG_SPACE        .usect   "NELSPCE",4,4
L_NES_LONG_POWER_Q15    .usect   "NELPEQ15",2*SAMPLES,4
L_NES_LONG_POWER_Q31    .usect   "NELPEQ31",4,4
L_FES_VSHORT_SPACE      .usect   "FEVSSPCE",4,4
L_FES_VSHORT_POWER_Q15  .usect   "FEVPEQ15",2*SAMPLES,4
L_FES_VSHORT_POWER_Q31  .usect   "FEVPEQ31",4,4
;L_FES_SHORT_POWER_Q15  .usect   "FEPEQ15",2,4
L_FES_SHORT_POWER_Q31   .usect   "FEPEQ31",4,4
L_ERR_SHORT_POWER_Q15   .usect   "ERPEQ15",2,4
L_ERR_SHORT_POWER_Q31   .usect   "ERPEQ31",4,4
L_ERROR                 .usect   "ERROR",2,4
L_FILTER_OUTPUT         .usect   "OUT",2*SAMPLES,4
L_SPEECH_OUT            .usect   "SPOUT",2*SAMPLES,4
L_NES_MARGIN            .usect   "NMARGIN",2,4
L_FES_MARGIN            .usect   "FMARGIN",2,4
;SAMPLE_COUNT           .usect   "SAMPLE",4,4
ECHAN0                  .usect   ".aec_st0",2048,4
ECHAN1                  .usect   ".aec_st1",2048,4
ECHAN2                  .usect   ".aec_st2",2048,4
GEN_PTR                 .usect   "PTR",64,64
```

File: einit.s

```
*******************************************************************************
*       Copyright (C) 1995 Texas Instruments Incorporated.                    *
*                         All Rights Reserved                                 *
*-----------------------------------------------------------------------------*
*                                                                             *
*  einit.s  --  AEC initialization file for C80 Acoustic Echo Cancellation    *
*                                                                             *
*                                                                             *
*  Environment:                                                               *
*    -- Assembles with versions 1.10 and above of TI's PP assembler.          *
*    -- Allocates with PPCA version 1.01 and above.                           *
*-----------------------------------------------------------------------------*
*  History:                                                                   *
*    04JULY95...Original version written ..................... R. Matusiak    *
*                                        ..................... David Qi       *
*******************************************************************************
        .include "echan.i"
        .include "packetpp.i"
        .ref GEN_PTR
  ;      .ref FES_HANG
        .ref ECHAN0
        .ref ECHAN1
        .ref ECHAN2
        .ref echan0_start
        .ref echan0_end
        .ref echan1_start
        .ref echan1_end
        .ref echan2_start
        .ref echan2_end
        .ref COEFF_INIT
        .ref L_AEC_FILTER_MEM
        .ref L_AEC_COEFFICIENTS

        .def echan_save_state
        .def echan_restore_state
        .def generic_ptr
        .def get_speech
```

```
        .system _AEC_State_Init
        .system $AEC_State_Init
src             .set    d1
dst             .set    d2
count           .set    d3
Ga_coeff        .set    a8
La_mem          .set    a2
        .ptext
_AEC_State_Init:
$AEC_State_Init:
     *(sp -= [STACK_SIZE]) = iprs
         *(sp + [2])=a15
         *(sp + [3])=a15
         *(sp + [4])=a15
         *(sp + [5])=a15
****************************************
* Restore initial values of coefficients.
****************************************
     count = 2*FILTER_SIZE
     Ga_coeff = COEFF_INIT
     call = get_speech
     src = Ga_coeff
     dst = &*(xba + L_AEC_COEFFICIENTS)
****************************************
* Clear filter memory
****************************************
     La_mem =&*(xba + L_AEC_FILTER_MEM)
     lrse0 = FILTER_SIZE/2 - 3
     *La_mem++ = a15
     *La_mem++ = a15
     *La_mem++ = a15
****************************************
* Save all on chip data to extern memory
****************************************
     call = echan_save_state
     nop
     nop
     br = *sp
     a15=*(sp+=[STACK_SIZE])
```

```
     nop

*****************************************************************
src             .set    d1
dst             .set    d2
end             .set    d4
La_packet_table .set    a0

****************************
* void echan_save_state()
****************************
echan_save_state:
     *(sp + [IPR]) = iprs
     src = &*(xba + echan0_start)
     end = &*(xba + echan0_end)
     La_packet_table = &*(xba + GEN_PTR)
     call = generic_ptr
     dst = ECHAN0
     count = end – src
     src = &*(xba + echan1_start)
     end = &*(xba + echan1_end)
     La_packet_table = &*(xba + GEN_PTR)
     call = generic_ptr
     dst = ECHAN1
     count = end – src
     src = &*(xba + echan2_start)
     end = &*(xba + echan2_end)
     La_packet_table = &*(xba + GEN_PTR)
     call = generic_ptr
     dst = ECHAN2
     count = end – src
     br = *(sp + [IPR])
     nop
     nop
****************************
* void echan_restore_state()
****************************
echan_restore_state:
     *(sp + [IPR]) = iprs
```

```
     dst = &*(xba + echan0_start)
     end = &*(xba + echan0_end)
     La_packet_table = &*(xba + GEN_PTR)
     call = generic_ptr
     src = ECHAN0
     count = end – dst
     dst = &*(xba + echan1_start)
     end = &*(xba + echan1_end)
     La_packet_table = &*(xba + GEN_PTR)
     call = generic_ptr
     src = ECHAN1
     count = end – dst

     dst = &*(xba + echan2_start)
     end = &*(xba + echan2_end)
     La_packet_table = &*(xba + GEN_PTR)
call = generic_ptr
     src = ECHAN2
     count = end – dst
     br = *(sp + [IPR])
     nop
     nop
*****************************************************************************
* void generic_ptr(long &src, long &dst, long count, long &packet_table)
*****************************************************************************
generic_ptr:
*****************************************
* Arguments passed by the calling function
*****************************************
src:                    .set    d1
dst:                    .set    d2
count:                  .set    d3
La_PR:                  .set    a0
*********************
* Internal variables
*********************
PT_entry:               .set    d4
```

```
*********************************************************************
* Before modifying the Packet Transfer Parameter table, make sure that
* the previous PTR has completed.  This is done by polling the Q bit.
*********************************************************************
     a15 = comm & 0x1\\PT_QueuedShift         ; check if Q bit is zero
     br =[nz.z] ipe
     nop
     a15 = comm & 0x1\\PT_QueuedShift         ; check if Q bit is zero
     *(pba + ePT_LinkedListStart) = La_PR
     PT_entry = 1\\31                         ; PT_entry = 0x80000000
          || *La_PR.sPT_Next = La_PR
     *La_PR.sPT_Options = PT_entry            ; set stop bit in PT options
     *La_PR.sPT_SrcStartAddress = src         ; set source start address src
     *La_PR.sPT_DstStartAddress = dst         ; set destination start address
                                              ; to dst
     *La_PR.sPT_SrcBACount = count
     PT_entry = 0
           || *La_PR.sPT_DstBACount = count
     *La_PR.sPT_SrcCCount = PT_entry
     *La_PR.sPT_DstCCount = PT_entry
     br = iprs
     comm = comm | 1\\PT_SubmitShift          ; issue the PTR
     nop
*******************************************************
* void get_speech(char *src, char *dst, int num_bytes)
*******************************************************
get_speech:
     *(sp + [IPR]) = iprs
     call = generic_ptr
     La_packet_table = &*(xba + GEN_PTR)
     nop
     br = *(sp + [IPR])
     nop
     nop
*******************************************************
* void put_speech(char *src, char *dst, int num_bytes)
*******************************************************
put_speech:
     *(sp + [IPR]) = iprs
```

```
call = generic_ptr
La_packet_table = &*(xba + GEN_PTR)
nop
a15 = comm & 0x1\\PT_QueuedShift          ; check if Q bit is zero
br =[nz.z] ipe
nop
a15 = comm & 0x1\\PT_QueuedShift          ; check if Q bit is zero
br = *(sp + [IPR])
nop
nop
```

File: init.cmd

```
if $$MVP_MP$$
;  map C80 internal memory
ma 0x00000000,0x00000800,ram  ; PP0 Data RAM 0
ma 0x00000800,0x00000800,ram  ; PP0 Data RAM 1
ma 0x00001000,0x00000800,ram  ; PP1 Data RAM 0
ma 0x00001800,0x00000800,ram  ; PP1 Data RAM 1
ma 0x00002000,0x00000800,ram  ; PP2 Data RAM 0
ma 0x00002800,0x00000800,ram  ; PP2 Data RAM 1
ma 0x00003000,0x00000800,ram  ; PP3 Data RAM 0
ma 0x00003800,0x00000800,ram  ; PP3 Data RAM 1
ma 0x00008000,0x00000800,ram  ; PP0 Data RAM 2
ma 0x00009000,0x00000800,ram  ; PP1 Data RAM 2
ma 0x0000A000,0x00000800,ram  ; PP2 Data RAM 2
ma 0x0000B000,0x00000800,ram  ; PP3 Data RAM 2
ma 0x01000000,0x00000800,ram  ; PP0 Parameter RAM
ma 0x01001000,0x00000800,ram  ; PP1 Parameter RAM
ma 0x01002000,0x00000800,ram  ; PP2 Parameter RAM
ma 0x01003000,0x00000800,ram  ; PP3 Parameter RAM
ma 0x01010000,0x00000800,ram  ; MP Parameter RAM
ma 0x01810000,0x00000800,ram  ; MP Data Cahce 0
ma 0x01810800,0x00000800,ram  ; MP Data Cahce 1
ma 0x01818000,0x00000800,ram  ; MP Instruction Cahce 0
ma 0x01818800,0x00000800,ram  ; MP Instruction Cahce 1
ma 0x01820000,0x00000200,ram  ; Memory-Mapped TC Registers
ma 0x01820200,0x00000200,ram  ; Memory-Mapped FC Registers
;  map SDB specific memory
ma 0x80000000,0x00800000,ram  ; DRAM 8 Meg
ma 0xC0000000,0x00200000,ram  ; VRAM 2 Meg
load echan.out
;go main
endif
if $$MVP_PP$$
;  map C80 internal memory
ma 0x00000000,0x00000800,ram  ; PP0 Data RAM 0
ma 0x00000800,0x00000800,ram  ; PP0 Data RAM 1
ma 0x00001000,0x00000800,ram  ; PP1 Data RAM 0
ma 0x00001800,0x00000800,ram  ; PP1 Data RAM 1
```

```
ma 0x00002000,0x00000800,ram  ; PP2 Data RAM 0
ma 0x00002800,0x00000800,ram  ; PP2 Data RAM 1
ma 0x00003000,0x00000800,ram  ; PP3 Data RAM 0
ma 0x00003800,0x00000800,ram  ; PP3 Data RAM 1
ma 0x00008000,0x00000800,ram  ; PP0 Data RAM 2
ma 0x00009000,0x00000800,ram  ; PP1 Data RAM 2
ma 0x0000A000,0x00000800,ram  ; PP2 Data RAM 2
ma 0x0000B000,0x00000800,ram  ; PP3 Data RAM 2
ma 0x01000000,0x00000800,ram  ; PP0 Parameter RAM
ma 0x01001000,0x00000800,ram  ; PP1 Parameter RAM
ma 0x01002000,0x00000800,ram  ; PP2 Parameter RAM
ma 0x01003000,0x00000800,ram  ; PP3 Parameter RAM
ma 0x01010000,0x00000800,ram  ; MP Parameter RAM
ma 0x01801800,0x00000800,ram  ; PP0 Instruction Cache
ma 0x01803800,0x00000800,ram  ; PP1 Instruction Cache
ma 0x01805800,0x00000800,ram  ; PP2 Instruction Cache
ma 0x01807800,0x00000800,ram  ; PP3 Instruction Cache
ma 0x01820000,0x00000200,ram  ; Memory–Mapped TC Registers
ma 0x01820200,0x00000200,ram  ; Memory–Mapped FC Registers
;  map SDB specific memory
ma 0x80000000,0x00800000,ram  ; DRAM 8 Meg
ma 0xC0000000,0x00200000,ram  ; VRAM 2 Meg
sload echan.out
;ba AEC_State_Init
ba AEC
endif
```

File: echan.lnk

```
-c
-x
-heap  0x2000
-stack 0x2000
-l mp_rts.lib
-l mp_task.lib
-l mp_ppcmd.lib
-l ppcmd.lib
-l mp_int.lib
-l sdbembed.lib
-l mp_ptreq.lib
MEMORY
{
    DRAM0     : o = 0x00000000   l = 0x0800    /* DRAMS 0                */
    DRAM1     : o = 0x00000800   l = 0x0800    /* DRAMS 1                */
    DRAM2     : o = 0x00008000   l = 0x0800    /* DRAMS 2                */
/*  PRAM_RES  : o = 0x01000300   l = 0x0240    /* RESERVED PART OF PRAM  */
    PRAM      : o = 0x01000000   l = 0x07ff    /* PRAM                   */
    EXTMEM    : o = 0x80000000   l = 0x800000  /* EXTERNAL MEMORY        */
}

SECTIONS
{
    .text    :       > EXTMEM
    .ptext   :       > EXTMEM
    .cinit   :       > EXTMEM
    .const   :       > EXTMEM
    .switch  :       > EXTMEM
    .data    :       > EXTMEM
    .bss     :       > EXTMEM
    .pbss    :       > EXTMEM
    .aec_st0 :       > EXTMEM
    .aec_st1 :       > EXTMEM
    .aec_st2 :       > EXTMEM
    GROUP > DRAM0
    {
         .echan0 {echan0_start = . ;}
```

```
{
          *(COEFF)
          *(NES)
          *(NEVSPCE)
          *(NEVPEQ15)
          *(NEVPEQ31)
          *(NESSPCE)
          *(NESPEQ15)
          *(NESPEQ31)
          *(NELSPCE)
          *(NELPEQ15)
          *(NELPEQ31)
     }     {echan0_end = . ;}
}
GROUP > DRAM1
{
     .echan1 {echan1_start = . ;}
{
           *(AEC_MEM)
           *(FES)
           *(FEVSSPCE)
           *(FEVPEQ15)
           *(FEVPEQ31)
           *(FEPEQ31)
           *(ERPEQ15)
           *(ERPEQ31)
           *(OUT)
           *(SPOUT)
     }     {echan1_end = . ;}

}
GROUP > DRAM2
{
     .echan2 {echan2_start = . ;}
{
           *(ERROR)
           *(SUM)
     }     {echan2_end = . ;}
```

```
    }
    .pram: > PRAM
    {
        *(PTR)
    }

}
```