An Implementation of Adaptive Filters with the TMS320C25 or the TMS320C30

APPLICATION REPORT: SPRA116

Sen Kuo
Northwestern Illinois University
Chein Chen
Digital Signal Processor Products
Semiconductor Group
Texas Instruments

Digital Signal Processing Solutions

TEXAS INSTRUMENTS

TRADEMARKS

CONTACT INFORMATION

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

An Implementation of Adaptive Filters with the TMS320C25 or the TMS320C30

Abstract

Adaptive filtering techniques are necessary considerations when a specific signal output is desired but the coefficients of that filter cannot be determined at the outset. Sometimes this is because of changing line or transmission conditions. An adaptive filter is one which contains coefficients that are updated by an adaptive algorithm to optimize filter response to the desired performance criterion.

Two devices, the TMS320C25 and TMS320C30, combine the power, high speed, flexibility and architecture optimized for adaptive signal processing.

This book discusses the topic of adaptive filter implementation as they apply to these two processors.

The book begins with a description of the two parts of an adaptive filter: the filter and the adaptive algorithm. The book goes on to discuss:

- The applications of adaptive filters (including adaptive prediction, equalization, noise cancellation and echo cancellation).
- The implementation of adaptive structures and algorithms (including transversal structure with the LMS algorithm, symmetric transversal structure, lattice structure, and modified LMS algorithms)
- Implementation considerations (including dynamic range constraint, finite precision errors, and design issues)

- Software development (assembly function libraries, C function libraries, development process and environment)

The book also contains:

- Tables showing transversal structure, symmetric transversal structure and lattice structure for both the TMS320C25 and TMS320C30 processors
- Extensive references
- Multiple appendices of sample code for both TMS320C25 and TMS320C30 processors

Product Support

World Wide Web

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

Email

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

Introduction

A filter selects or controls the characteristics of the signal it produces by conditioning the incoming signal. The coefficients of the filter determine its characteristics and output *a priori* in many cases. Often, a specific output is desired, but the coefficients of the filter cannot be determined at the outset. An example is an echo canceller; the desired output cancels the echo signal (an output result of zero when there is no other input signal). In this case, the coefficients cannot be determined initially since they depend on changing line or transmission conditions. For applications such as this, it is necessary to rely on adaptive filtering techniques.

An adaptive filter is a filter containing coefficients that are updated by an adaptive algorithm to optimize the filter's response to a desired performance criterion. In general, adaptive filters consist of two distinct parts: a filter, whose structure is designed to perform a desired processing function; and an adaptive algorithm, for adjusting the coefficients of that filter to improve its performance, as illustrated in Figure 1. The incoming signal, x(n), is weighted in a digital filter to produce an output, y(n). The adaptive algorithm adjusts the weights in the filter to minimize the error, e(n), between the filter output, y(n), and the desired response of the filter, d(n). Because of their robust performance in the unknown and time-variant environment, adaptive filters have been widely used from telecommunications to control.

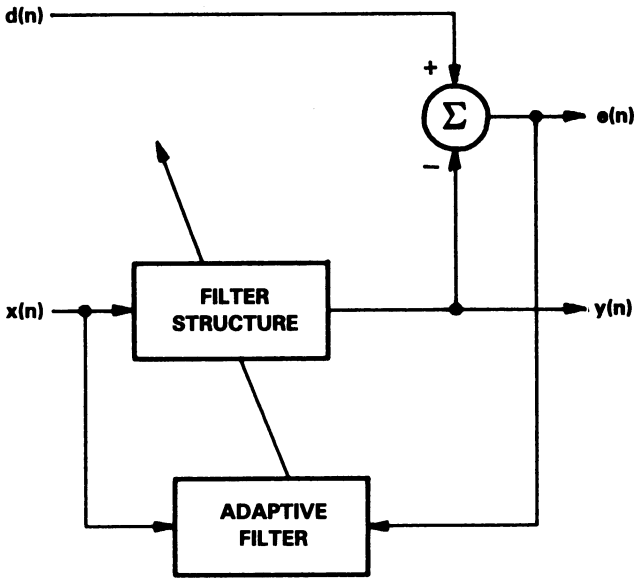


Figure 1. General Form of an Adaptive Filter

Adaptive filters can be used in various applications with different input and output configurations. In many applications requiring real-time operation, such as adaptive prediction, channel equalization, echo cancellation, and noise cancellation, an adaptive filter implementation based on a programmable digital signal processor (DSP) has many advantages over other approaches such as a hard-wired adaptive filter. Not only are power, space, and manufacturing requirements greatly reduced, but also programmability provides flexibility for system upgrade and software improvement.

The early research on adaptive filters was concerned with adaptive antennas [1] and adaptive equalization of digital transmission systems [2]. Much of the reported research on the adaptive filter has been based on Widrow's well-known Least Mean Square (LMS) algorithm, because the LMS algorithm is relatively simple to design and implement, and it is well-understood and well-suited for many applications. All the filter structures and update algorithms discussed in this application report are Finite Impulse Response (FIR) filter structures and LMS-type algorithms. However, for a particular application, adaptive filters can be implemented in a variety of structures and adaptation algorithms [1, 3 through 9]. These structures and algorithms generally trade increased complexity for improved performance. An interactive software package to evaluate the performance of adaptive filters has also been developed [10].

The complexity of an adaptive filter implementation is usually measured in terms of its multiplication rate and storage requirement. However, the data flow and data manipulation capabilities of a DSP are also major factors in implementing adaptive filter systems. Parallel hardware multiplier, pipeline architecture, and fast on-chip memory size are major features of most DSPs [11, 12] and can make filter implementation more efficient.

Two such devices, the TMS320C25 and TMS320C30 from Texas Instruments [13, 14], have been chosen as the processors for fixed-point and floating-point arithmetic. They combine the power, high speed, flexibility, and an architecture optimized for adaptive signal processing. The instruction execution time is 80 ns for the TMS320C25 and only 60 ns for the TMS320C30. Most instructions execute in a single cycle, and the architectures of both processors make it possible to execute more than one operation per instruction. For example, in one instruction, the TMS320C25 processor can generate an instruction address and fetch that instruction, decode the instruction, perform one or two data moves (if the second data is from program memory), update one address pointer, and perform one or two computations (multiplication and accumulation). These processors are designed for real-time tasks in telecommunications, speech processing, image processing, and high-speed control, etc.

To direct the present research toward realistic real-time applications, three adaptive structures were implemented:

1. Transversal
2. Symmetric transversal
3. Lattice

Each structure utilizes five different update algorithms:

1. LMS
2. Normalized LMS
3. Leaky LMS
4. Sign-error LMS
5. Sign-sign LMS

Each structure with its adaptation algorithms is implemented using the TMS320C25 with fixed-point arithmetic and the TMS320C30 with floating-point arithmetic. The processor assembly code is included in the Appendix for each implementation. The assembly code for each structure and adaptation strategy can be readily modified by the reader to fit his/her applications and could be incorporated into a C function library as callable routines.

In this application report, the applications of adaptive filters, such as adaptive prediction, adaptive equalization, adaptive echo cancellation, and adaptive noise cancellation are presented first. Next, the implementation of the three filter structures and five adaptive algorithms with the TMS320C25 and TMS320C30 is described. This is followed by the practical considerations on the implementation of these adaptive filters. The remainder of the application report covers coding options, such as the routine libraries that support both assembly and C languages.

Applications of Adaptive Filters

The most important feature of an adaptive filter is the ability to operate effectively in an unknown environment and track time-varying characteristics of the input signal. The adaptive filter has been successfully applied to communications, radar, sonar, control, and image processing. Figure 1 illustrates a general form of an adaptive filter with input signals, x(n) and d(n), output signal, y(n), and error signal, e(n), which is the difference between the desired signal, d(n), and output signal, y(n). The adaptive filter can be used in different applications with different input/output configurations. In this section we briefly discuss several potential applications for the adaptive filters [15].

Adaptive Prediction

Adaptive prediction [16 through 18] is illustrated in Figure 2. In the general application of adaptive prediction, the signals are x(n) – delayed version of original signal, d(n) – original input signal, y(n) – predicted signal, and e(n) – prediction error or residual.

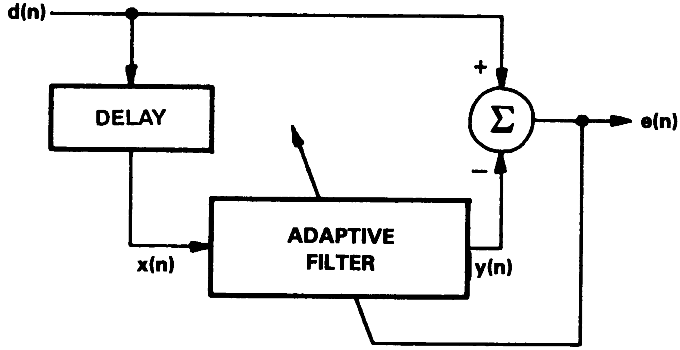


Figure 2. Block Diagram of an Adaptive Predictor

A major application of the adaptive prediction is the waveform coding of a speech signal. The adaptive filter is designed to exploit the correlation between adjacent samples of the speech signal so that the prediction error is much smaller than the input signal on the average. This prediction error signal is quantized and sent to the receiver in order to reduce the number of bits required for the transmission. This type of waveform coding is called Adaptive Differential Pulse-Code Modulation (ADPCM) [17] and provides data rate compression of the speech at 32 kb/s with toll quality. More recently, in certain on-line applications, time recursive modeling algorithms have been proposed to facilitate speech modeling and analysis.

The coefficients of the adaptive predictor can be used as the autoregressive (AR) parameters of the nonstationary model. The equation of the AR process is

$$u(n) = a_1{*}\ u(n-1) + a_2{*}\ u(n-2) + \ldots\ldots + a_m{*}\ u(n-m) + v(n)$$

where a_1, a_2, ..., a_m are the AR parameters. Thus, the present value of the process u(n) equals a finite linear combination of past values of the process plus an error term v(n). This adaptive AR model provides a practical means to measure the instantaneous frequency of input signal. The adaptive predictor can also be used to detect and enhance a narrow band signal embedded in broad band noise. This Adaptive Line Enhancer (ALE) provides at its output y(n) a sinusoid with an enhanced signal-to-noise ratio, while the sinusoidal components are reduced at the error output e(n).

Adaptive Equalization

Figure 3 shows another model known as adaptive equalization [2, 9, 15]. The signals in the adaptive equalization model are defined as x(n) – received signal (filtered version of transmitted signal) plus channel noise, d(n) – detected data signal (data mode) or pseudo random number (training mode), y(n) – equalized signal used to detect received data, and e(n) – residual intersymbol interference plus noise.

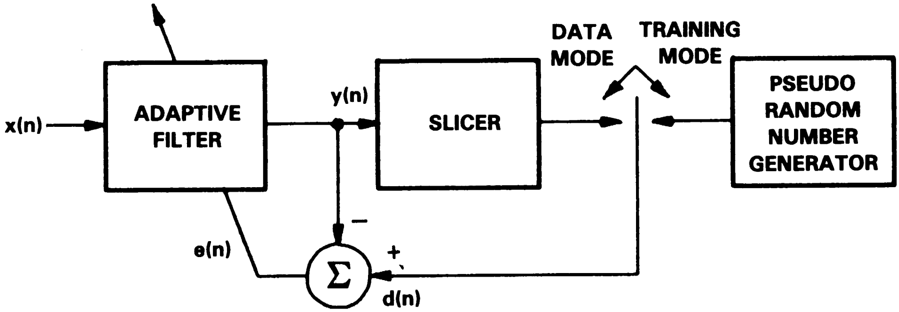


Figure 3. Block Diagram of an Adaptive Equalizer

The use of adaptive equalization to eliminate the amplitude and phase distortion introduced by the communication channel was one of the first applications of adaptive filtering in telecommunications [19]. The effect of each symbol transmitted over a time-dispersive channel extends beyond the time interval used to represent that symbol, resulting in an overlay of received symbols. Since most channels are time-varying and unknown in advance, the adaptive channel equalizer is designed to deal with this intersymbol interference and is widely used for bandwidth-efficient transmission over telephone and radio channels.

Adaptive Echo Cancellation

Another application, known as adaptive echo cancellation [20, 21] is shown in Figure 4. In this application, the signals are identified as x(n) − far-end signal, d(n) − echo of far-end signal plus near-end signal, y(n) − estimated echo of far-end signal, and e(n) − near-end signal plus residual echo.

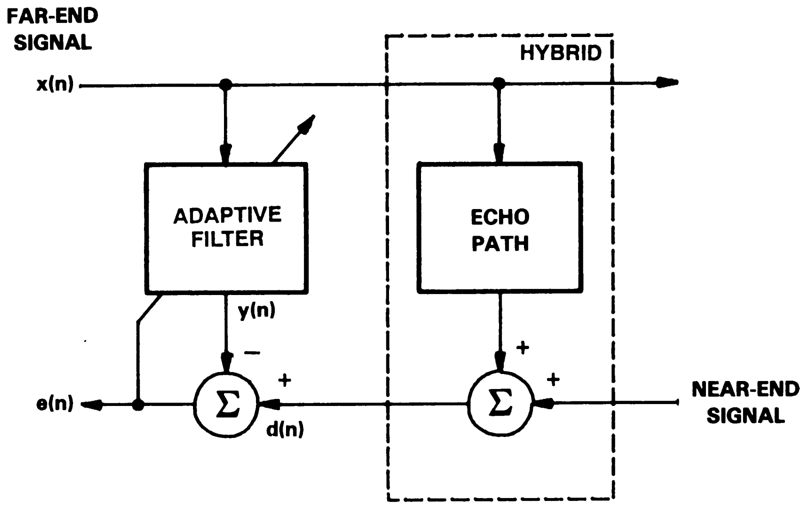


Figure 4. Block Diagram of an Echo Canceller

The adaptive echo cancellers are used in practical applications of cancelling echoes for long-distance telephone voice communication, full-duplex voiceband data modems, and high-performance audio-conferencing systems. To overcome the echo problem, echo cancellers are installed at both ends of the network. The cancellation is achieved by estimating the echo and subtracting it from the return signal.

Adaptive Noise Cancellation

One of the simplest and most effective adaptive signal processing techniques is adaptive noise cancelling [1, 22]. As shown in Figure 5, the primary input d(n) contains both signal and noise, where x(n) is the noise reference input. An adaptive filter is used to estimate the noise in d(n) and the noise estimate y(n) is then subtracted from the primary channel. The noise cancellation output is then the error signal e(n).

The applications of noise cancellation include the cancellation of various forms of interference in electrocardiography, noise in speech signals, noise in fighter cockpit environments, antennas sidelobe interference, and the elimination of 60-Hz hum. In the majority of these noise cancellation applications, the LMS algorithm has been utilized.

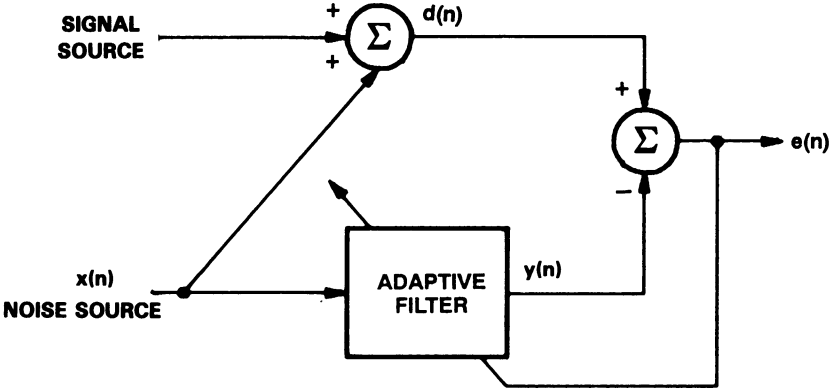


Figure 5. General Form of a Noise Canceller

Application Summary

The above list of applications is not exhaustive and is limited primarily to applications within the field of telecommunications. Adaptive filtering has been used extensively in the context of many other fields including, but not limited to, instantaneous frequency tracking, intrusion detection, acoustic Doppler extraction, on-line system identification, geophysical signal processing, biomedical signal processing, the elimination of radar clutter, beamforming, sonar processing, active sound cancellation, and adaptive control.

Implementation of Adaptive Structures and Algorithms

Several types of filter structures can be implemented in the design of the adaptive filters such as Infinite Impulse Response (IIR) or Finite Impulse Response (FIR). An adaptive IIR filter [1, 5], with poles as well as zeros, makes it possible to offer the same filter characteristics as the FIR filter with lower filter complexity. However, the major problem with adaptive IIR filter is the possible instability of the filter if the poles move outside the unit circle during the adaptive process. In this application report, only FIR structure is implemented to guarantee filter stability.

An adaptive FIR filter can be realized using transversal, symmetric transversal, and lattice structures. In this section, the adaptive transversal filter with the LMS algorithm is introduced and implemented first to provide a working knowledge of adaptive filters.

Transversal Structure with LMS Algorithm

Transversal Structure Filter

The most common implementation of the adaptive filter is the transversal structure (tapped delay line) illustrated in Figure 6. The filter output signal y(n) is

$$y(n) = \underline{w}^T(n)\underline{x}(n) = \sum_{i=0}^{N-1} w_i(n)\, x(n-i) \tag{1}$$

where $\underline{x}(n)=[x(n)\ x(n-1)\ \ldots\ x(n-N+1)]^T$ is the input vector, $\underline{w}(n)=[w_0(n)\ w_1(n)\ \ldots\ w_{N-1}(n)]^T$ is the weight vector, T denotes transpose, n is the time index, and N is the order of filter. This example is in the form of a finite impulse response filter as well as the convolution (inner product) of two vectors $\underline{x}(n)$ and $\underline{w}(n)$. The implementation of Equation (1) is illustrated using the following C program:

```
y[n] = 0.;
for (i = 0; i < N; i++) {
      y[n] += wn[i]*xn[i];
      {
```

where wn [i] denotes wi(n) and xn[i] represents x(n−i).

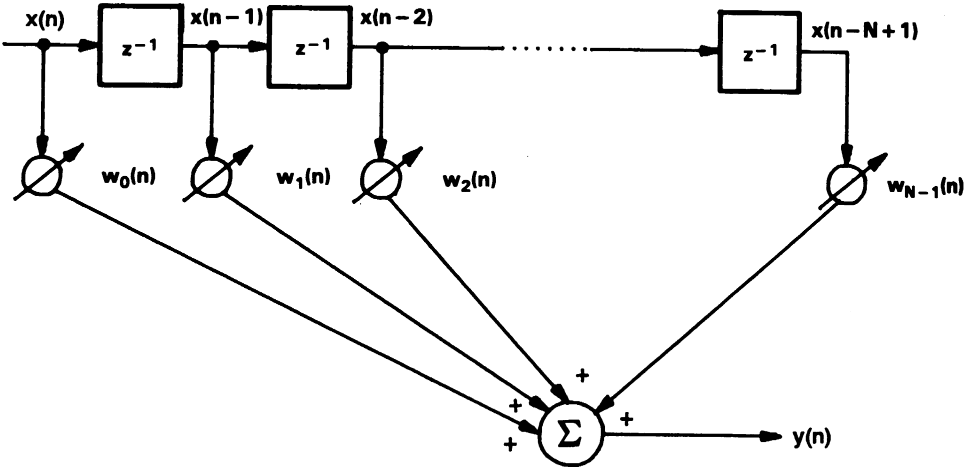


Figure 6. Transversal Filter Structure

TMS320C25 Implementation

The architecture of TMS320C25 [13] is optimized to implement the FIR filter. After execution of the CNFP (Configure Block B0 as Program Memory) instruction, the filter coefficients $w_i(n)$ from RAM block B0 (via program bus) and data $x(n-i)$ from RAM block B1 (via data bus) are available simultaneously for the parallel multiplier (see Figure 7).

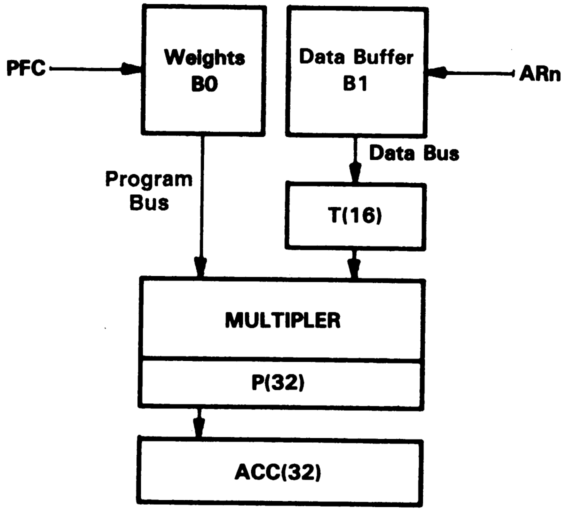
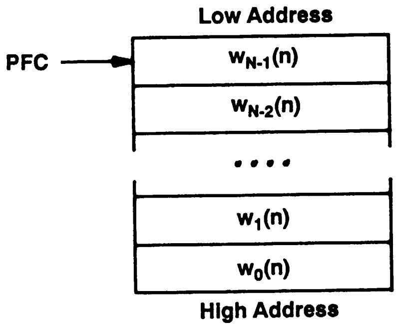


Figure 7. TMS320C25 Arithmetic Unit (after execute CNFP instruction)

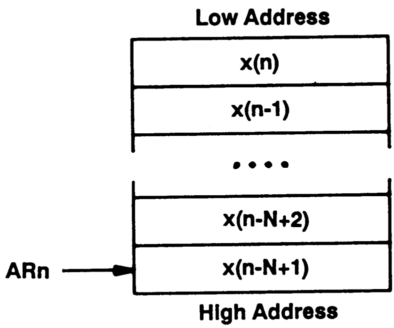
The MACD instruction enables complete multiply/accumulate, data move, and pointer update operations to be completed in a single instruction cycle (80 ns) if filter coefficients are stored in on-chip RAM or ROM or in off-chip program memory with zero wait states. Since the adaptive weights $w_i(n)$ need to be updated in every iteration, the filter coefficients must be stored in RAM. The implementation of the inner product in Equation (1) can be made even more efficient with a repeat instruction, RPTK. An N-weight transversal filter can be implemented as follows [23]:

```
LARP    ARn
LRLK    ARn,LASTAP
RPTK    N-1
MACD    COEFFP,*-                                   (A)
```

Where ARn is an auxiliary address register that points to $x(n-N+1)$, and the Prefetch Counter (PFC) points to the last weight $w_{N-1}(n)$ indicated by COEFFP. When the MACD instruction is repeated, the coefficient address is transferred to the PFC and is incremented by one during its operation. Therefore, the components of weight vector $\underline{w}(n)$ are stored in B0 as



The MACD in repeat mode will also copy data pointed to by ARn, to the next higher on-chip RAM location. The buffer memories of transversal filter are therefore stored as



In general, roundoff noise occurs after each multiplication. However, the TMS320C25 has a 16×16-bit multiplier and a 32-bit accumulator, so there is no roundoff during the summing of a set of product terms in Program (A). All multiplication products are represented in full precision, and rounding is performed after they are summed. Thus y(n) is obtained from the accumulator with only one roundoff, which minimizes the roundoff noise in the output y(n). Since both the tapped delay line and the adaptive weights are stored in data RAM to achieve the fastest throughput, the highest transversal filter order for efficient implementation on the TMS320C25 is 256. However, if necessary, higher order filters can be implemented by using external data RAM.

TMS320C30 Implementation

The architecture of TMS320C30 [14] is quite different from TI's second generation processors. Instead of using program/data memory, it provides two data address buses to do the data memory manipulations. This feature allows two data memory addresses to be generated at the same time. Hence, parallel data store, load, or one data store with one data load can be done simultaneously. Such capabilities make the programming much easier and more flexible. Since the hardware multiplier and arithmetic logic unit (ALU) of TMS320C30 are separated, with proper operand arrangement, the processor can do one multiplication and one addition or subtraction at the same time. With these two combined features, the TMS320C30 can execute several other parallel instructions. These parallel instructions can be found in Section 11 of the *Third-Generation TMS320 User's Guide* [14]. Associating with single repeat instruction RPTS, an inner product in Equation (1) can be implemented as follows:

```
      MPYF3    *AR0++(1)%,*AR1++(1)%,R1     ; w[0].x[0]
      RPTS     N-2                          ; Repeat N-1 times
      MPYF3    *AR0++(1)%,*AR1++(1)%,R1     ; y[] = w[].x[]
   || ADDF3    R1,R2,R2
      ADDF3    R1,R2,R2                     ; Include last product
```

where auxiliary registers AR0 and AR1 point to x and w arrays. The addition in the parallel instruction sums the previous values of R1 and R2. Therefore, R1 is initialized with the first product prior to the repeat instruction RPTS.

Note that the implementation above does not move the data in the x array like MACD does in TMS320C25. For filter delay taps, the TMS320C30 uses a circular buffer method to implement the delay line. This method reserves a certain size of memory for the buffer and uses a pointer to indicate the beginning of the buffer. Instead of moving data to next memory location, the pointer is updated to point to the previous memory location. Therefore, from the new beginning of the buffer, it has the effect of the tapped delay line. When the value of the pointer exceeds the end of the buffer, it will be circled around to the other end of the buffer. It works just like joining two ends of the buffer together as a necklace. Thus, new data is within the circular queue, pointed to by AR0, replacing the oldest value. However, from an adaptive filter point of view, data doesn't have to be moved at this point yet.

TMS320C30 has a 32-bit floating point multiplier and the result from the multiplier is put and accumulated into a 40-bit extended precision register. If the input from A/D converter is equal to or less than 16 bits, there is no roundoff noise after multiplication. Theoretically, the TMS320C30 can implement a very high order of adaptive filter. However, for the most efficient implementation, the limitation of filter order is 2K because the TMS320C30 external data write requires at least two cycles. If the filter coefficients are put in somewhere other than internal data RAM, the instruction cycles will be increased.

LMS Adaptation Algorithm

The adaptation algorithm uses the error signal

$$e(n) = d(n) - y(n), \tag{2}$$

where d(n) is the desired signal and y(n) is the filter output. The input vector $\underline{x}(n)$ and e(n) are used to update the adaptive filter coefficients according to a criterion that is to be minimized. The criterion employed in this section is the mean-square error (MSE)ϵ:

$$\epsilon = E[e^2(n)] \tag{3}$$

where E [.] denotes the expectation operator. If y(n) from Equation (1) is substituted into Equation (2), then Equation (3) can be expressed as

$$\epsilon = E[d^2(n)] + \underline{w}^T(n)R\underline{w}(n) - 2\,\underline{w}^T(n)\underline{p} \tag{4}$$

where $R = E[x(n)x^T(n)]$ is the N x N autocorrelation matrix, which indicates the sample-to-sample correlation within a signal, and $\underline{p} = E[d(n)\,\underline{x}(n)]$ is the N x 1 cross-correlation vector, which indicates the correlation between the desired signal d(n) and the input signal vector $\underline{x}(n)$.

The optimum solution $w^* = [w_0^*\ w_1^* \ldots w_{N-1}^*]^T$, which minimizes MSE, is derived by solving the equation

$$\frac{\delta\epsilon}{\delta\underline{w}(n)} = 0 \tag{5}$$

This leads to the normal equation

$$R\,\underline{w}^* = \underline{p} \tag{6}$$

If the R matrix has full rank (i.e., R^{-1} exists), the optimum weights are obtained by

$$\underline{w}^* = R^{-1} \underline{p} \tag{7}$$

In Linear Predictive Coding (LPC) of a speech signal, the input speech is divided into short segments, the quantities of R and $\underline{p}$ are estimated, and the optimal weights corresponding to each segment are computed. This procedure is called a block-by-block data-adaptive algorithm [24].

A widely used LMS algorithm is an alternative algorithm that adapts the weights on a sample-by-sample basis. Since this method can avoid the complicated computation of R^{-1} and $\underline{p}$, this algorithm is a practical method for finding close approximate solutions to Equation (7) in real time. The LMS algorithm is the steepest descent method in which the next weight vector w(n+1) is increased by a change proportional to the negative gradient of mean-square-error performance surface in Equation (7)

$$\underline{w}(n+1) = \underline{w}(n) - u\underline{\nabla}(n) \tag{8}$$

where u is the adaptation step size that controls the stability and the convergence rate. For the LMS algorithm, the gradient at the nth iteration, $\underline{\nabla}(n)$, is estimated by assuming squared error $e^2(n)$ as an estimate of the MSE in Equation (3). Thus, the expression for the gradient estimate can be simplified to

$$\underline{\nabla}(n) = \frac{\delta[e^2(n)]}{\delta \underline{w}(n)} = -2\, e(n)\, \underline{x}(n) \tag{9}$$

Substitution of this instantaneous gradient estimate into Equation (8) yields the Widrow-Hoff LMS algorithm

$$\underline{w}(n+1) = \underline{w}(n) + 2\, u\, e(n)\, \underline{x}(n) \tag{10}$$

where 2 u in Equation (10) is usually replaced by u in practical implementation.

Starting with an arbitrary initial weight vector $\underline{w}(0)$, the weight vector $\underline{w}(n)$ will converge to its optimal solution $\underline{w}^*$, provided u is selected such that [1]

$$0 < u < \frac{1}{\lambda_{max}} \tag{11}$$

where λ_{max} is the largest eigenvalue of the matrix R. λ_{max} can be bounded by

$$\lambda_{max} < Tr\ [R] = \sum_{i=0}^{N-1} r\ (0) = N\ r(0) \tag{12}$$

where Tr [.] denotes the trace of a matrix and $r(0) = E\ [x^2(n)]$ is average input power.

For adaptive signal processing applications, the most important practical consideration is the speed of convergence, which determines the ability of the filter to track nonstationary signals. Generally speaking, weight vector convergence is attained only when the slowest weight has converged. The time constant of the slowest mode is [1]

$$t = \frac{1}{u\lambda_{min}} \tag{13}$$

This indicates that the time constant for weight convergence is inversely proportional to u and also depends on the eigenvalues of the autocorrelation matrix of the input. With the disparate eigenvalues, i.e., $\lambda_{max} >> \lambda_{min}$, the setting time is limited by the slowest mode, λ_{min}. Figure 8 shows the relaxation of the mean square error from its initial value ϵ_0 toward the optimal value ϵ_{min}.

Adaptation based on a gradient estimate results in noise in the weight vector, therefore a loss in performance. This noise in the adaptive process causes the steady state weight vector to vary randomly about the optimum weight vector. The accuracy of weight vector in steady state is measured by excess mean square error (excess MSE $= E\ [\epsilon - \epsilon_{min}]$). The excess MSE in the LMS algorithm [1] is

$$\text{excess MSE} = u\ Tr[R]\ \epsilon_{min} \tag{14}$$

where ϵ_{min} is minimum MSE in the steady state.

Equations (13) and (14) yield the basic trade-off of the LMS algorithm: to obtain high accuracy (low excess MSE) in the steady state, a small value of u is required, but this will slow down the convergence rate. Further discussions of the characteristics and properties of the LMS algorithm are presented in [1, 3 through 9]. The implementations of LMS algorithm with the TMS320C25 and TMS320C30 are presented next.

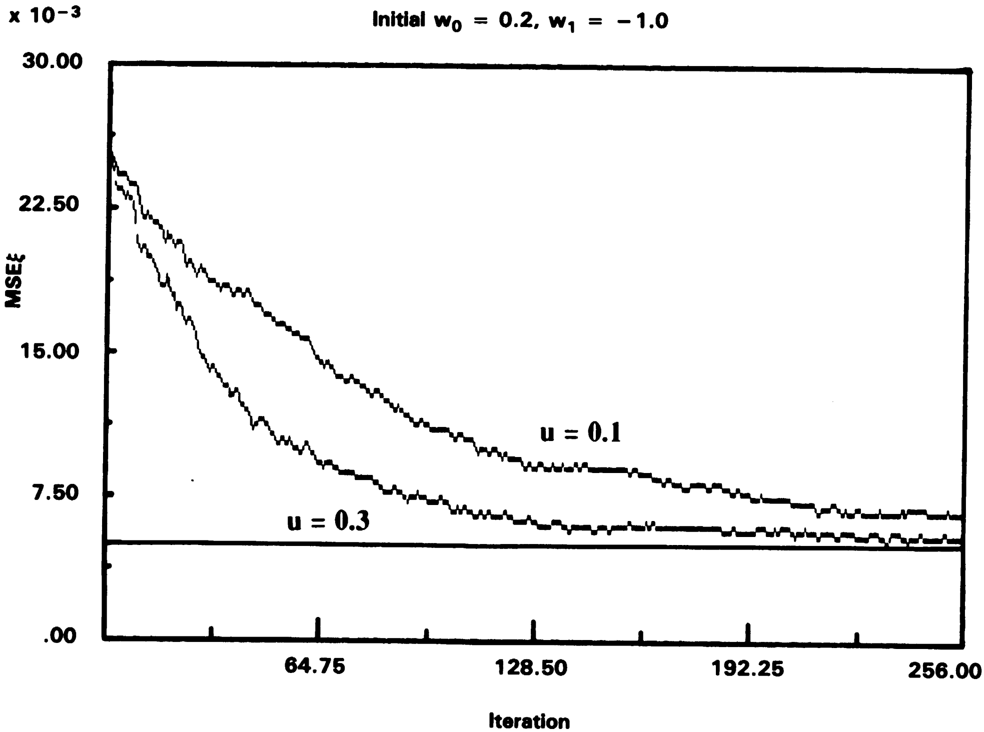


Figure 8. Learning Curve of an Adaptive Transversal Filter and an LMS Algorithm with Different Step Sizes

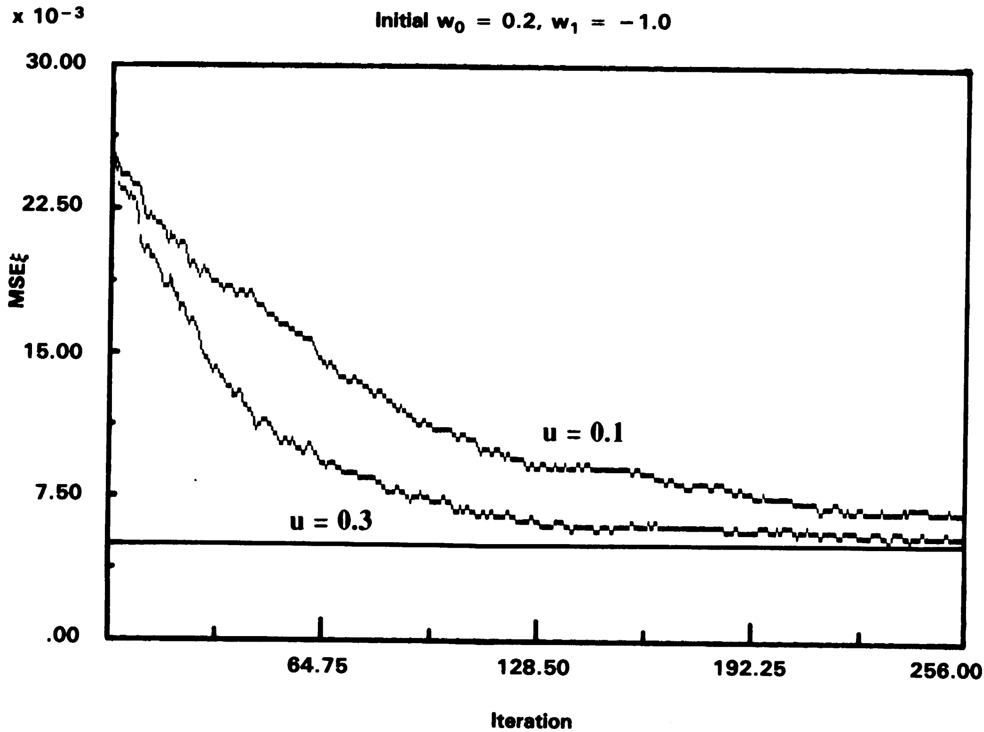


Figure 8. Learning Curve of an Adaptive Transversal Filter and an LMS Algorithm with Different Step Sizes

Since u*e(n) is constant for N weights update, the error signal e(n) is first multiplied by u to get ue(n). This constant can be computed first and then multiplied by x(n) to update w(n). An implementation method of the LMS algorithm in Equation (10) is illustrated as

```
ue(n) = u*e[n];
for (i=0; i<N; i++) {
      wn[i] += uen * xn[i];
}
```

TMS320C25 Implementation

The TMS320C25 provides two powerful instructions (ZALR and MPYA) to perform the update example in Equation (10).

- ZALR loads a data memory value into the high-order half of the accumulator while rounding the value by setting bit 15 of the accumulator to one and setting bits 0-14 of the accumulator to zero. The rounding is necessary because it can reduce the roundoff noise from multiplication.
- MPYA accumulates the previous product in the P register and multiplies the operand with the data in T register.

Assuming that ue(n) is stored in T and the address pointer is pointing to AR3, the adaptation of each weight is shown in the following instruction sequence:

```
      LRLK  AR1,N-1        ; Initialize loop counter
      LRLK  AR2,COEFFD     ; Point to w_{N-1}(n)
      LRLK  AR3,LASTAP+1   ; Point to x(n-N+1), since MACD in (A)
                           ; Already moved elements of current
                           ; x(n) to the next higher location
      MPY   *-,AR2         ; P=ue(n) * x(n-N+1)
ADAP  ZALR  *,AR3          ; Load w_i(n) and round
      MPYA  *-,AR2         ; ACC=P+w_i(n) and P=ue(n) * x(n-i)
      SACH  *+,0,AR1       ; Store w_i(n+1)
      BANZ  ADAP,*-,AR2    ; Test loop counter, if counter not
                           ; Equal to 0, decrement counter,
                           ; Branch to ADAP and select AR2 as
                           ; Next pointer.
```

For each iteration, N instruction cycles are needed to perform Equation (1), 6N instruction cycles are needed to perform weight updates in Equation (10), and the total number of instruction cycles needed is 7N+28. An example of a TMS320C25 program implementing a LMS transversal filter is presented in Appendix A1. Note that BANZ needs three instruction cycles to execute. This can be avoided by using straight line code, which requires 4N+33 instruction cycles [25].

TMS320C30 Implementation

Although the TMS320C30 doesn't provide any specific instruction for adaptive filter coefficients update, it still can achieve the weight updating in two instructions because of its powerful architecture. The TMS320C30 has a repeat block instruction RPTB, which allows a block of instructions to be repeated a number of times without any penalty for looping. A single repeat mode, RM, in the status register, ST, and three registers – repeat start address (RS), repeat end address (RE), and repeat counter (RC) – control the block repeat. When RM is set, the PC repeats the instructions between RS and RE a number of times, which is determined by the value of RC. The repeat modes repeat a block of code at least once in a typical operation. The repeat counter should be loaded with one less than the desired number of repetitions. Assuming the error signal e(n) in Equation (10) is stored in R7, the adaptation of filter coefficients is shown as follows:

```
          MPYF3   *AR0++(1)%,R7,R1   ; R1 = u*e(n)*x(n)
          LDI     order-3,RC         ; Initialize repeat counter
          RPTB    LMS                ; Do i = 0, N-3
          MPYF3   *AR0++(1)%,R7,R1   ; Compute u*e(n)*x(n-i-1)
       || ADDF3   *AR1,R1,R2         ; Compute wi(n) + u*e(n)*x(n-i)
LMS       STF     R2,*AR1++(1)%      ; Store wi(n+1)

          MPYF3   *AR0,R7,R1         ; For i = N-2
       || ADDF3   *AR1,R1,R2
          STF     R2,*AR1++(1)%      ; Store wN-2(n+1)
          ADDF3   *AR1,R1,R2         ; Include last w
          STF     R2,*AR1++(1)%      ; Store wN-1(n+1)
```

where auxiliary register AR0 and AR1 point to x and w arrays. R1 is updated before loop since the accumulation in the parallel instruction uses the previous value in R1. In order to update x array pointer to the new beginning of the data buffer for next iteration (i.e., perform the data move), one of the loop instruction set has been taken out of loop and modified by eliminating the incrementation of AR0.

To perform an N-weight adaptive LMS transversal filter on TMS320C30 requires 3N+15 instruction cycles. There are N and 2N instruction cycles to perform Equations (1) and (10), respectively. The TMS320C30 example program is given in Appendix A2.

The LMS algorithm considerably reduces the computational requirements by using a simplified mean square error estimator (an estimate of the gradient). This algorithm has proved useful and effective in many applications. However, it has several limitations in performance such as the slow initial convergence, the undesirable dependence of its convergence rate on input signal statistics, and an excess mean square error still in existence after convergence.

Symmetric Transversal Structure [5]

A transversal filter with symmetric impulse response (weight values) about the center weight has a linear phase response. In applications such as speech processing, linear phase filters are preferred since they avoid phase distortion by causing all the components in the filter input to be delayed by the same amount. The adaptive symmetric transversal structure is shown in Figure 9.

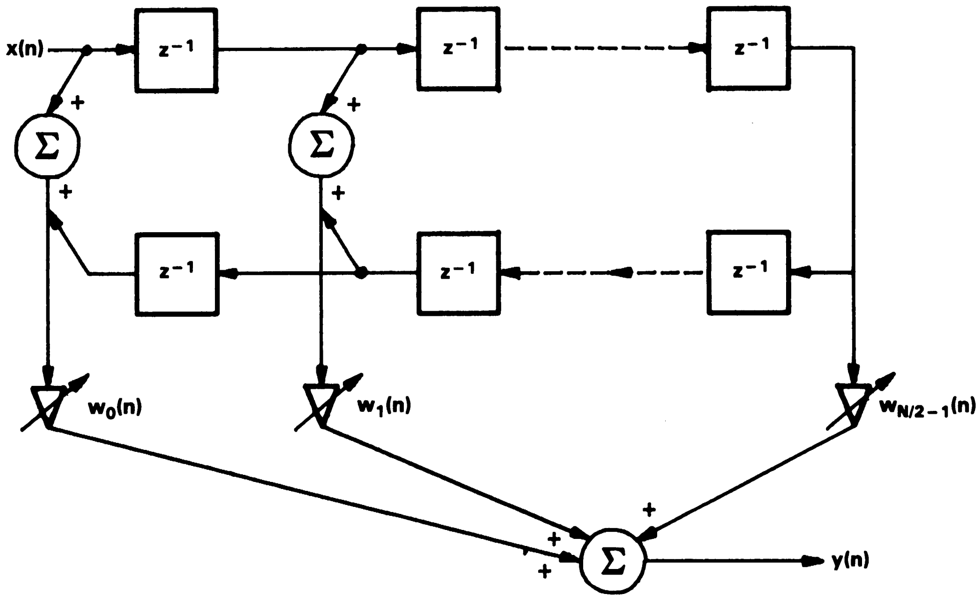


Figure 9. Symmetric Transversal Structure (even order)

This filter is actually an FIR filter with an impulse response that is symmetric about the center tap. The output of the filter is obtained as

$$y(n) = \sum_{i=0}^{N/2-1} w_i(n) \, [x(n-i) + x(n-N+i+1)] \qquad (15a)$$

where N is an even number. Note that, for fixed-point processors, the addition in the brackets may introduce overflow because the input signals $x(n-i)$ and $x(n-N+i+1)$ are in the range of -1 and $1-2^{-15}$. This problem can be solved by shifting x(n) to the right one bit. The update of the weight vector is

$$w_i(n+1) = w_i(n) + ue(n)[x(n-1) + x(n-N+i+1)] \qquad (15b)$$

for $i=0,1,\ldots,(N/2-1)$, which requires N/2 multiplications and N additions. Theoretically, this symmetric structure can also reduce computational complexity since such filters require only half the multiplications of the general transversal filter. However, it is true only for the TMS320C30 processor. When a filter is implemented on the TMS320C25, the transversal structure is more efficient than the symmetric transversal structure due to the pipeline multiplication and accumulation instruction MACD, which is optimized to implement convolution in Equation (1).

TMS320C25 Implementation

For TMS320C25, in order to implement the instructions MAC, ZALR, and MPYA, we can trade memory requirements for computation saving by defining

$$z(n-i) = x(n-i) + x(n-N+i+1) \ , \ i=0,1,\ldots,N/2-1 \tag{16a}$$

Now, Equation (15) can be expressed as

$$y(n) = \sum_{i=0}^{N/2-1} w_i(n)\, z(n-i) \tag{16b}$$

$$w_i(n+1) = w_i(n) + u\, e(n)\, z(n-i) \ , \ i=0,1,\ldots,N/2-1 \tag{16c}$$

Equation (16a) can be implemented using the TMS320C25 as

```
        LARK    AR1, N/2-1     ; Counter = N/2 -1
        LRLK    AR2,LAST_X     ; Point to x(n-N+1)
        LRLK    AR3,FIRST_X    ; Point to x(n)
        LRLK    AR4,FIRST_Z    ; Point to z(n)
        LARP    AR3
SYM     LAC     *+,0,AR2
        ADD     *-,0,AR4
        SACL    *+,0,AR1
        BANZ    SYM,*-,AR3
```

The instruction sequence to implement the LMS algorithm in Equations (1) and (10) can be used to implement Equations (16b) and (16c), except using MAC instead of MACD in Program (A). Therefore, N instruction cycles are needed to shift data in x(n), 3N instruction cycles are needed to implement Equation (16a), N/2 for Equation (16b), and 3N for Equation (16c). The total number of instruction cycles required to implement the symmetric transversal filter with the LMS algorithm is 7.5N+38. Where 7.5N is an integer because N is chosen as an even number. The 0.5N instruction cycles come from Equation (15a) since symmetric transversal structure folds the filter taps into half of the order N (see Figure 9). The maximum filter length for most efficient code, 256, is the same as for the FIR filter. The use of the additional data memory can be obtained from the reduced data memory requirement for weights of the symmetric transversal filter. The complete TMS320C25 program is given in Appendix B1.

Note that instead of storing buffer locations x(n) contiguously, then using DMOV to shift data in the buffer memory (requiring N cycles) at the end of each iteration, we can use a circular buffer with pointers pointing to x(n) and x(n−N+1). Since pointer updating requires several instruction cycles, compared with N cycles using DMOV to update the buffer memory contents, the circular buffer technique is more efficient if N is large.

TMS320C30 Implementation

As mentioned above, the TMS320C30 uses a circular buffer instead of data move technique. Therefore, it does not have to implement tapped delay line separately as TMS320C25. Equations (1) and (16a) can be combined and implemented in the same loop. The advantage of this is that a parallel instruction reduces the number of the instruction cycles. The implementation is shown as follows:

```
        LDF     0.0,R2                      ; Clear R2
        LDI     order/2-2,RC                ; Set up loop counter
        RPTB    INNER                       ; Do i = 0, N/2 -2
        ADDF3   *AR4++(1)%,*AR5--(1)%,R1;   z(i) = x(n-i) + x(n+N-i)
        MPYF3   R1,*AR1++(1),R3             ; R3 = w[] * z[]
     || STF     R1,*AR2++(1)                ; Store z(i)
INNER   ADDF3   R3,R2,R2                    ; Accumulate the result for y

        ADDF3   *AR4++(1)%,*AR5--(1)%,R1;   For i = N/2 -1
        MPYF3   R1,*AR1--(IR0),R3
     || STF     R1,*AR2--(IR0)
        ADDF3   R3,R2,R2                    ; Include last product
```

where AR4 and AR5 point to x[0] and x[N−1]. AR1 and AR2 point to w and z array, respectively. IR0 contains value of $N/2^{-1}$. The same instruction codes of weight update of transversal filter can be used in symmetric transversal structure by changing the x array pointer to the z array pointer. Appendix B2 presents an example program. The total number of instructions needed is 2.5N+15, which is less than that of the transversal structure.

Lattice Structure [6]

An alternative FIR filter realization is the lattice structure [26]. A discussion of the transversal filter with the LMS algorithm shows that the convergence rate of the transversal structure is restricted by the correlation of signal components; i.e., the eigenvalue spread, $\lambda_{max}/\lambda_{min}$. The lattice structure is a decorrelating transform based on a family of prediction error filters as illustrated in Figure 10. The recursive equations that describe the lattice predictor are

$$f_0(n) = b_0(n) = x(n) \tag{17a}$$

$$f_m(n) = f_{m-1}(n) - k_m(n) b_{m-1}(n-1), \; 0 < m <= M \tag{17b}$$

$$b_m(n) = b_{m-1}(n-1) - k_m(n) f_{m-1}(n), \; 0 < m <= M \tag{17c}$$

where $f_m(n)$ represents the forward prediction error, $b_m(n)$ represents the backward prediction error, $k_m(n)$ is the reflection coefficients, m is the stage index, and M is the number of cascaded stages. The lattice structure has the advantage of being order-recursive. This property allows adding or deleting of stages from the lattice without affecting the existing stages.

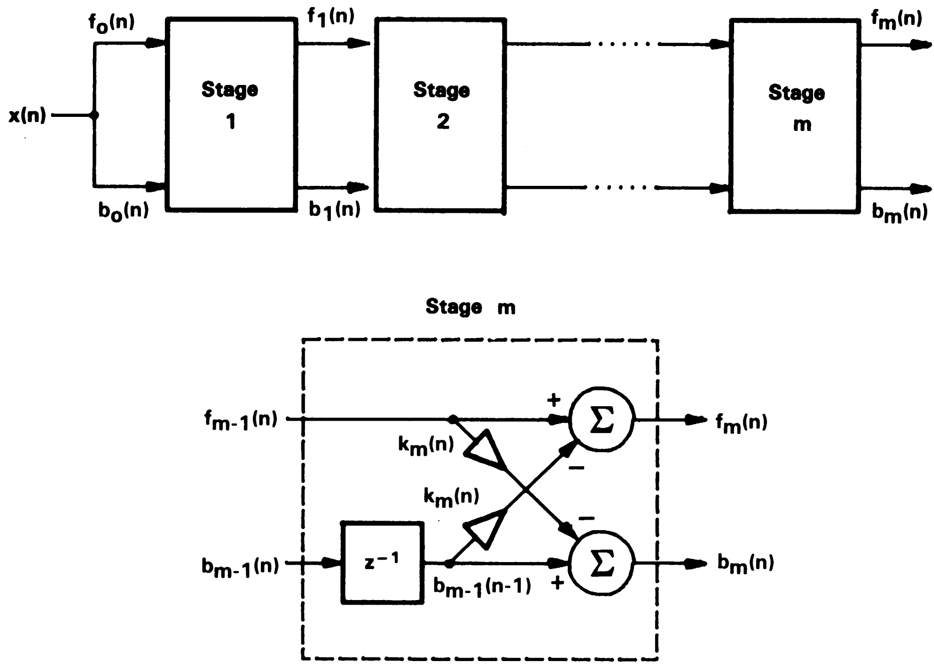


Figure 10. Lattice Structure

To implement the lattice filter for processing actual data, the reflection coefficients $k_m(n)$ are required. These coefficients can be computed according to estimates of the autocorrelation coefficients using Durbin's algorithm. However, it would be more efficient if these reflection coefficients could be estimated directly from the data and updated on a sample-by-sample basis, such as LMS algorithm [6]. The reflection coefficient $k_m(n+1)$ can be recursively computed [7]:

$$k_m(n+1) = k_m(n) + u[f_m(n)b_{m-1}(n-1) + b_m(n)f_{m-1}(n)], \quad 0 < m <= M \quad (18)$$

For applications such as noise cancellation, channel equalization, line enhancement, etc., the joint-process estimation [3] illustrated in Figure 11 is required. This device performs two optimum estimations: the lattice predictor and the multiple regression filter. The following equations define the implementation of the regression filter

$$e_0(n) = d(n) - b_0(n)g_0(n) \quad (19a)$$

$$e_m(n) = e_{m-1}(n) - b_{m-1}(n)g_{m-1}(n), \quad 0 < m <= M \quad (19b)$$

$$g_m(n+1) = g_m(n) + u_{em}(n)b_m(n), \quad 0 <= m <= M \quad (20)$$

where the LMS algorithm is used to update the coefficients of the regression filter. For noise cancellation application, $e_m(n)$ corresponds to the output e(n) in Figure 5. For applications such as adaptive line enhancer and channel equalizer, filter output y(n) is obtained as

$$y(n) = \sum_{m=0}^{M} g_m(n)\, b_m(n) \quad (21)$$

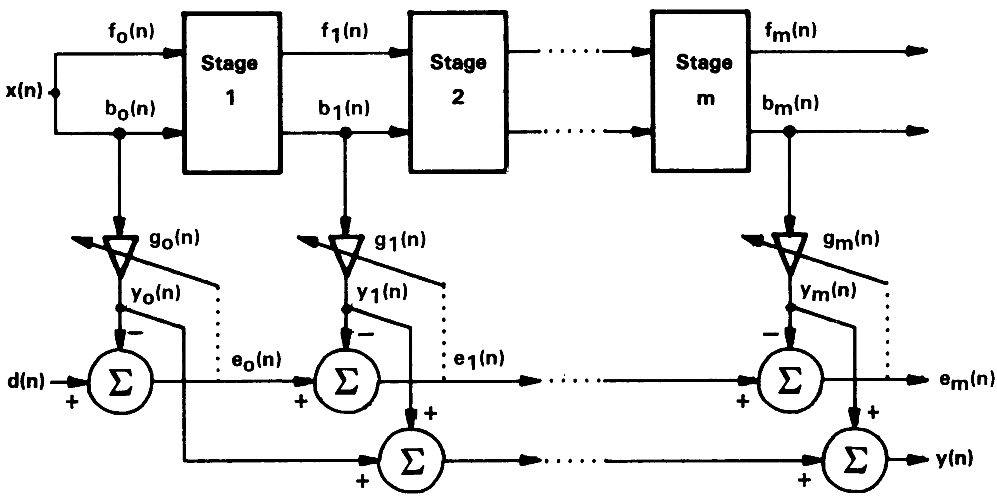


Figure 11. Lattice Structure with Joint Process Estimation

TMS320C25/TMS320C30 Implementation

There are five memory locations—$f_m(n)$, $b_m(n)$, $b_m(n-1)$, $k_m(n)$, and $g_m(n)$—required for each stage. The limitation of on-chip data RAM is 544 words for the TMS320C25 and 2K words for the TMS320C30. A maximum of 102 stages can therefore be implemented on a single TMS320C25 for the highest throughput. Here, another advantage of TMS320C30 architecture design is shown. Since the operands of the mathematic operations can be either memory or register on the TMS320C30, and there is no need to preserve the values of f_m array for the next iteration (refer to Equations (17) and (18)), the f_m array can be replaced by an extended precision register. Thus, for the most efficient codes, the stage limitation of lattice structure for TMS320C30 is 512, or one-fourth of the 2K on-chip RAM.

Lattice structures have superior convergence properties relative to transversal structures and good stability properties; e.g., low sensitivity to coefficient quantization, low roundoff noise, and the ability to check stability by inspection. The disadvantages of lattice filter algorithms are that they are numerically complex and require mathematical sophistication to thoroughly understand their derivations. Furthermore, as shown in Appendixes C1 and C2, lattice structures cannot take advantage of the TMS320C25 and TMS320C30's pipeline architecture to achieve high throughput. The total number of instruction cycles needed is $33M+32$ for TMS320C25 and $14M+4$ for TMS320C30.

Modified LMS Algorithms [5]

The LMS algorithm described in previous sections is the most widely used algorithm in practical applications today. In this section, a set of LMS-type algorithms (all direct variants of the LMS algorithm) are presented and implemented. The motivation for each is some practical consideration, such as faster convergence, simplicity in implementation, or robustness in operation. The description of these algorithms is based on the transversal structure. However, these algorithms can be applied to the symmetric transversal structure and the lattice structure as well.

Normalized LMS Algorithm

The stability, convergence time, and fluctuation of the adaptation process is governed by the step size u and the input power to the adaptive filter. In some practical applications, you may need an automatic gain control (AGC) on the input to the adaptive filter. The normalized LMS algorithm is one important technique used to improve the speed of convergence. This is accomplished while maintaining the steady-state performance independent of the input signal power. This algorithm uses a variable convergence factor u(n), which represents a u that is a function of the time index,

$$u(n) = a \;/\; var(n) \tag{22}$$

and

$$\underline{w}(n+1) = \underline{w}(n) + u(n)e(n)\underline{x}(n) \tag{23}$$

where a is a convergence parameter, and var(n) is an estimate of the input average power at time n using the recursive equation

$$\text{var}(n) = (1 - b)\ \text{var}(n-1) + b\ x^2(n) \tag{24}$$

where $0 < b << 1$ is a smoothing parameter. In practice, a is chosen equal to b.

For fixed-point processors, there is a way to reduce the computation of power estimation. Since b in Equation (24) doesn't have to be an exact number, it is computationally convenient to make b a power of 2. If $b = 2^{-m}$, the multiplication of b can be implemented by shifting right m bits. Therefore, the var(n) in Equation (24) is computed by

$$\begin{aligned}\text{var}(n) &= \text{var}(n-1) - b\ \text{var}(n-1) + b\ x^2(n) \\ &= \text{var}(n-1) - \text{var}(n-1) * 2^{-m} + x^2(n) * 2^{-m}\end{aligned}$$

Then, assuming the variance var(n) of input signal is stored in the data memory VAR and its initial value is 0.99997 ($= 1 - 2^{-15}$), The implementation of this equation using TMS320C25 assembly code is

```
LARP   AR3
LRLK   AR3,FRSTAP     ; Point to input signal x
SQRA   *              ; Square input signal
SPH    ERRF
ZALH   VAR            ; ACC = var(n-1)
SUB    VAR,SHIFT      ; ACC = (1-b) var(n-1)
ADD    ERRF,SHIFT     ; ACC = (1-b) var(n-1) + b x2(n)
SACH   VAR            ; Store var(n)
```

The normalized LMS algorithm can be implemented as

```
var = b1 * var + b * xn[0] * xn[0];
unen = e[n] * a / var;
for (i = 0; i< N; i++)
wn[i] += unen * xn[i];
```

where $b_1 = (1-b)$, xn[0] = x(n), and unen = u(n)*e(n). This normalized technique reduces the dependency of convergence speed on input signal power at the cost of increased computational complexity, especially the division in Equation (22). The algorithms of implementing the fixed-point and floating-point division on the TMS320C25 and

TMS320C30 can be found in the user's guide for each device [13, 14]. Since the power of input signal is always positive, those codes can be simplified to save computation time.

Since the power estimation in Equation (24) and step size normalization in Equation (22) are performed once for each sample x(n), the computation increase can be ignored when N is large. As shown in Appendixes D1 and D2, the total number of instruction cycles needed for the normalized LMS algorithm (7N+57 for the TMS320C25 and 3N+47 for the TMS320C30) is slightly higher than for the LMS algorithm (7N+34 and 3N+15) when N is large.

Sign LMS Algorithms

The LMS algorithm requires 2N multiplications and additions for each iteration; this amount is much lower than the requirements for many other complicated adaptive algorithms, such as Kalman and Recursive Least Square (RLS) [3]. However, there are three simplified versions of the LMS algorithm (sign-error LMS, sign-data LMS, and sign-sign LMS) that save the number of multiplications required and extend the real-time bandwidth for some applications [5, 27].

First, the sign-error LMS algorithm can be expressed as

$$\underline{w}(n+1) = \underline{w}(n) + u \; \text{sign}[e(n)] \; \underline{x}(n) \tag{25}$$

where $$\text{sign}[e(n)] = \begin{cases} 1, & \text{if } e(n) \geq 0 \\ -1, & \text{if } e(n) < 0 \end{cases}$$

The C program implementation of sign-error LMS algorithm is

```
tu = u;
if (e[n] < 0.) {
    tu = -u; }
for (i=0; i<N; i++) {
    wn[i] += tu * xn[i];
}
```

As shown in Appendixes E1 and E2, the instruction sequence to implement weight update with the sign-error LMS algorithm is identical to that with the LMS algorithm. The difference is that the sign-error LMS algorithm uses the sign [e(n)]*u instead of e(n)*u before the update loop. Note that, for fixed-point processors, if u is chosen to be a power of two, the u x(n) can be accomplished by shifting right the elements in x(n). This algorithm keeps the same convergence direction as the LMS algorithm. Thus, the sign-error LMS algorithm should remain efficient, provided the variable gain u(n) is matched to this change. However, the use of constant step size u to reduce computation comes at the expense of a slow convergence rate since smaller u is normally used for stability reasons.

The programs in Appendixes E1 and E2 implement a transversal filter with sign-error LMS algorithm in looped code. The total number of instruction cycles needed for this algorithm using the TMS320C25 is 7N+26, which is slightly less than for the LMS algorithm's 7N+28. Computing u*e(n) takes 5 instruction cycles. The sign-error LMS algorithm determines the sign of the u by checking the sign of e(n), which takes only 3 instruction cycles. The total number of instruction cycles needed for the sign-error LMS algorithm using the TMS320C30 is 3N+16, which is slightly higher than for the LMS algorithm. This occurs because the TMS320C30 takes only one instruction cycle to compute u*e(n) and two instruction cycles to determine the sign of the u.

Secondly, the sign-data LMS algorithm is

$$\underline{w}(n+1) = \underline{w}(n) + u\, e(n)\, \mathrm{sign}[\underline{x}(n)] \tag{26}$$

This equation can be implemented as

$$\begin{aligned} w_i(n+1) &= w_i(n) + ue(n)\ , \text{ if } x(n-i) >= 0 \\ &= w_i(n) - ue(n)\ , \text{ if } x(n-i) < 0 \end{aligned}$$

for $i=0,1,\ldots,N-1$. Since the sign determination is required inside the adaptation loop to determine the sign of $x(n-i)$, slower throughput is expected. The total number of instruction cycles needed is 11N+26 for the TMS320C25 and 5N+16 for the TMS320C30.

Finally, the sign-sign LMS algorithm is

$$\underline{w}(n+1) = \underline{w}(n) + u\, \mathrm{sign}[e(n)]\, \mathrm{sign}[\underline{x}(n)] \tag{27}$$

which requires no multiplications at all and is used in the CCITT standard for ADPCM transmission. As we can see from the above equations, the number of multiplications is reduced. This simplified LMS algorithm looks promising and is designed for VLSI or discrete IC implementation to save multiplications.

The sign-sign LMS algorithm can be implemented as

```
for (i=0; i<N; i++) {
    if (e[n] >= 0.) {
        if (xn[i] >= 0.)
            wn[i] += u;
        else
            wn[i] -= u; }
                else {
        if (xn[i]> = 0.)
            wn[i]-= u;
```

```
        else
              wn[i] += u; } }
```

When this algorithm is implemented on TMS320C25 and TMS320C30 with pipeline architecture and a parallel multiplier, the performance of sign-sign LMS algorithm is poor compared to standard LMS algorithm due to the determination of sign of data, which can break the instruction pipeline and can severely reduce the execution speed of the processors.

In order to avoid double branches inside the loop, the XOR instruction is utilized to check the sign bit of e(n) and x(n−i). The sign-sign LMS algorithm can be implemented as

$$
\begin{aligned}
w_i(n+1) &= w_i(n) + u \text{ , if } \operatorname{sign}[e(n)] = \operatorname{sign}[x(n-i)] \\
&= w_i(n) - u \text{ , otherwise}
\end{aligned}
$$

The following TMS320C25 instruction sequence implements this algorithm without branching (assuming that the current address register used is AR3):

```
      LRLK   AR1,N-1         ; Set up counter
      LRLK   AR2,COEFFD      ; Point to wi(n)
      LRLK   AR3,LASTAP+1    ; Point to x(n-i)
ADAP  LAC    *-,0,AR2        ; Load x(n-i)
      XOR    ERR             ; XOR with e(n)
      SACL   ERRF            ; Save sign bit, sign = 0 if same signs
                             ; Sign = 1 if different signs
      LAC    ERRF            ; Sign extension to ACCH,
                             ; ACCH = 0 If ERRF > = 0
                             ; ACCH = 0FFFFh if ERRF < 0
      XORK   MU,15           ; Take one's complement of m
                             ; If sign = 1
      ADD    *,15            ; Weight update
      SACH   *+,1,AR1        ; Save new weight
      BANZ   ADAP,*-,AR3
```

The one's complement of u is used instead of −u, because they are only slightly different and the step size does not require the exact number. The weight update with this technique requires 10N instruction cycles and FIR filtering requires N instruction cycles so that the total number of instruction cycles needed is 11N+21. The complete TMS320C25 assembly program is given in Appendix F1.

To determine whether a positive or negative u should be used without branching is trickier in the TMS320C30. Fortunately, the extended precision registers of TMS320C30 interpret the 32 most-significant bits of the 40-bit data as the floating-point number and the 32 least-significant bits of the 40-bit data as an integer. When a floating-point number changes its sign, its exponent remains the same. Therefore, the sign of step size u can be determined by using XOR logic on its mantissa. The following code shows how the sign-sign LMS algorithm is implemented on the TMS320C30.

```
        ASH     -31,R7              ; R7 = Sign[e(n)]
        XOR3    R0,R7,R5            ; R5 = Sign[e(n)] * u
        LDF     *AR0++(1)%,R6       ; R6 = x(n)
        ASH     -31,R6              ; R6 = Sign[x(n-i)]
        XOR3    R5,R6,R4            ; R4 = Sign[x(n-i)]*Sign[e(n)] * u
        ADDF3   *AR1,R4,R3          ; R3 = wi(n) + R4

        LDI     order-3,RC          ; Initialize repeat counter
        RPTB    SSLMS               ; Do i = 0, N-3
        LDF     *AR0++(1)%,R6       ; Get next data
||      STF     R3,*AR1++(1)%       ; Update wi(n+1)
        ASH     -31,R6              ; Get the sign of data
        XOR3    R5,R6,R4            ; Decide the sign of u
SSLMS   ADDF3   *AR1,R4,R3          ; R3 = wi(n) + R4

        LDF     *AR0,R6             ; Get last data
||      STF     R3,*AR1++(1)%       ; Update wN-2(n+1)
        ASH     -31,R6              ; Get the sign of data
        XOR3    R5,R6,R4            ; Decide the sign of u
        ADDF3   *AR1,R4,R3          ; Compute wN-1(n+1)
        STF     R3,*AR1++(1)%       ; Store last w(n+1)
```

Here, R0, R4, and R5 contain the value of u before updating. AR0 and AR1 point to x array and w array, respectively. R7 contains the value of error signal e(n). The complete program is given in Appendix F2. The total number of instruction cycles is $5N+16$, which is much higher than LMS algorithm.

The sign-sign LMS algorithm is developed to reduce the multiplication requirement of the LMS algorithm. Since DSPs provide the hardware multiplier as a standard feature, this modification does not provide any advantage when implementing this algorithm on the DSPs. On the contrary, it causes some disadvantages since decision instructions will destroy the instruction pipeline. If you use the XOR logic operation in order to avoid using the decision instructions, the complexity of the program will be increased and the total number of instruction cycles will be greater than the regular LMS algorithm.

Leaky LMS Algorithm

When adaptive filters are implemented on signal processors with fixed word lengths, roundoff noise is fed back to adaptive weights and accumulates in time without bound. This leads to an overflow that is unacceptable for real-time applications. One solution is based upon adding a small forcing function, which tends to bias each filter weight toward zero. The leaky LMS algorithm has the form

$$\underline{w}(n+1) = r\ \underline{w}(n) + u\ e(n)\ \underline{x}(n) \tag{28a}$$

where r is slightly less than 1.

Since r can be expressed as 1 − c and c << 1, the TMS320C25 can take advantage of the built-in shifters to implement this algorithm. Therefore, Equation (28a) can be changed to

$$\underline{w}(n+1) = \underline{w}(n) - c\ \underline{w}(n) + u\ e(n)\ \underline{x}(n) \tag{28b}$$

In order to achieve the highest throughput by using ZALR and MPYA, cw(n) can be implemented by shifting $w_i(n)$ right by m bits where 2^{-m} is close to c. Since the length of the accumulator is 32 bits and the high word (bits 16 to 31) is used for updating w(n), shifting right m bits of $w_i(n)$ can be implemented by loading $w_i(n)$ and shifting left 16 − m bits. The sequence of TMS320C25 instructions to implement Equation (28b) is shown as

```
        LRLK    AR1,N-1         ; Set up counter
        LRLK    AR2,COEFFD      ; Point to wi(n)
        LRLK    AR3,LASTAP+1    ; Point to x(n - i)
        LT      ERRF            ; T = ERRF =u*e(n)
        MPY     *-,AR2
ADAPT   ZALR    *,AR3
        MPYA    *-,AR2
        SUB     *,LEAKY         ; LEAKY=16-m
        SACH    *+,0,AR1
        BANZ    ADAPT,*-,AR2
```

For each iteration, 7N instruction cycles are needed to perform the adaptation process (6N for the LMS algorithm). The total number of instruction cycles needed is 8N+28 (see Appendix G1 for the complete program). The leaky factor r has the same effect as adding a white noise to the input. This technique not only can solve adaptive weights overflow problem, but also can be beneficial in an insufficient spectral excitation and stalling situation [5].

The method used above is especially for the TMS320C25, which has a free shift feature. Since TMS320C30 is a floating-point processor, r can simply multiply to filter coefficient. However, in order to reduce the instruction cycles, this multiplication can combine with another instruction to be a parallel instruction inside the loop. The following code shows how to rearrange the instructions from the LMS algorithm to include this multiplication without an extra instruction cycle.

```
        MPYF     @u_r,R7               ; R7 = e(n)*u/r
        MPYF3    *AR0++(1)%,R7,R1      ; R1 = e(n)*u*x(n)/r
        MPYF3    *AR0++(1)%,R7,R1      ; R1 = e(n)*u*x(n-1)/r
     || ADDF3    *AR1,R1,R2            ; R2 = w0(n) + e(n)*u*x(n)/r
        LDI      order-4,RC            ; Initialize repeat counter
        RPTB     LLMS                  ; do i = 0, N-4
        MPYF3    *AR2,R2,R0            ; R0 = r*wi(n) + e(n)*u*x(n-i)
     || ADDF3    *+AR1(1),R1,R2        ; R2 = wi+1(n) + e(n)*u*x(nz-i-1)/r
LLMS    MPYF3    *AR0++(1)%,R7,R1      ; R1 = e(n)*u*x(n-i-2)/r
     || STF      R0,*AR1++(1)%         ; Store wi(n+1)

        MPYF3    *AR2,R2,R0            ; R0 = r*wN-3(n) + e(n)*u*x(n-N+3)
     || ADDF3    *+AR1(1),R1,R2        ; R2 = wN-2(n) + e(n)*u*x(n-N+2)/r
        MPYF3    *AR0,R7,R1            ; R1 = e(n)*u*x(n-N+1)/r
     || STF      R0,*AR1++(1)%         ; Store wN-3(n+1)
        MPYF3    *AR2,R2,R0            ; R0 = r*wi(n) + e(n)*u*x(n-N+2)
     || ADDF3    *+AR1(1),R1,R2        ; R2 = wN-1(n) +
*                                      ;      e(n)*u*x(n-N+1)/r
        MPYF3    *AR2,R2,R0            ; R0 = r*wi(n) + e(n)*u*x(n-N+1)
     || STF      R0,*AR1++(1)%         ; Store wN-2(n+1)
        STF      R0,*AR1++(1)%         ; Update last w
```

Auxiliary registers AR0 and AR1 point to x and w arrays. AR2 points to the memory location that contains value r. R7 contains the value of error signal e(n). R1 and R2 are updated before the loop because the parallel instructions inside the loop use the previous values in R1 and R2. Note that R1 is updated twice before the loop because the updating of R2 requires the previous value of R1. In order to update x array pointer to the new beginning of the data buffer for next iteration, two of the loop instruction sets have been taken out of loop and modified by eliminating the incrementation of AR0. The TMS320C30 assembly program of an adaptive transversal filter with the leakage LMS algorithm is listed in Appendix G2 as an example. The total number of instruction cycles for this algorithm is 3N+15, which is the same as the LMS algorithm. This example shows the power and flexibility of the TMS320C30.

Implementation Considerations

The adaptive filter structures and algorithms discussed previously were derived on the basis of infinite precision arithmetic. When implementing these structures and algorithms on a fixed integer machine, there is a limitation on the accuracy of these filters due to the fact that the DSP operates with a finite number of bits. Thus, designers must pay attention to the effects of finite word length. In general, these effects are input quantization, roundoff in the arithmetic operation, dynamic range constraints, and quantization of filter coefficients. These effects can either cause deviations from the original design criteria or create an effective noise at the filter output. These problems have been investigated extensively, and techniques to solve these problems have been developed [28, 29].

The effects of finite precision in adaptive filters is an active research area, and some significant results have been reported [30 through 32]. There are three categories of finite word length effects in adaptive filters:

- Dynamic Range Constraint (scaling to avoid overflow). Since this is not applicable for a floating-point processor, the TMS320C30 is not mentioned in this portion.
- Finite Precision Errors (errors introduced by roundoff in the arithmetic).
- Design Issues (design of the optimum step size u that minimizes system noise).

Dynamic Range Constraint

As shown in Figure 1, the most widely used LMS transversal filter is specified by the difference equations

$$y(n) = \sum_{i=0}^{N-1} w_i(n)\ x(n-i) \tag{29}$$

and

$$w_i(n+1) = w_i(n) + u{*}e(n){*}x(n-i), \text{ for } i = 0, 1, \ldots, N-1 \tag{30}$$

where $x(n-i)$ is the input sequence and $w_i(n)$ are the filter coefficients.

If the input sequence and filter coefficients are properly normalized so that their values lie between -1 and 1 using Q15 format, no error is introduced into the addition. However, the sum of two numbers may become larger than one. This is known as overflow. The TMS320C25 provides four features that can be applied to handle overflow management [13]:

A. Branch on overflow conditions.
B. Overflow mode (saturation arithmetic).
C. Product register right shift.
D. Accumulator right shift.

One technique to inhibit the probability of overflow is scaling, i.e., constraining each node within an adaptive filter to maintain a magnitude less than unity. In Equation (29), the condition for $|y(n)| < 1$ is

$$x_{max} < 1 / \sum_{i=0}^{N-1} |w_i(n)| \tag{31}$$

where x_{max} denotes the maximum of the absolute value of the input. The right shifter of the TMS320C25, which operates with no cycle overhead, can be applied to implement scaling to prevent overflow of multiply-accumulate operations in Equation (29). By setting the PM bits of status register ST1 to 11 using the SPM or LST1 instructions, the P register output is right-shifted 6 places. This allows up to 128 accumulations without the possibility of an overflow. SFR instruction can also be used to right shift one bit of the accumulator when it is near overflow.

Another effective technique to prevent overflow in the computation of Equation (29) is using saturation arithmetic. As illustrated in Figure 12, if the result of an addition overflows, the output is clamped at the maximum value. If saturation arithmetic is used, it is common practice [28] to permit the amplitude of $x(n-i)$ to be larger than the upper bound given in Equation (31). Saturation of the filter represents a distortion, and the choice of scaling on the input depends on how often such distortion is permissible. The saturation arithmetic on the TMS320C25 is controlled by the OVM bit of status register ST0 and can be changed by the SOVM (set overflow mode), ROVM (reset overflow mode), or LST (load status register).

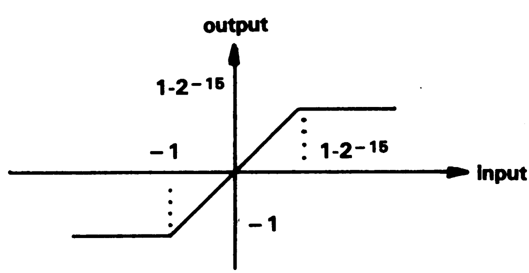


Figure 12. Saturation Arithmetic

Filter coefficients are updated using Equation (30). As illustrated in Figure 13, a new technique presented in reference 31 uses the scaling factor a to prevent filter's coefficients overflow during the weight updating operation. Suppose you use $a = 2^{-m}$. A right shift by m bits implements multiplication by a, while a left shift by m bits implements the scaling factor 1/a. Usually, the required value of a is not expected to be very small and depends on the application. Since a scales the desired signal, it does not affect the rate of convergence.

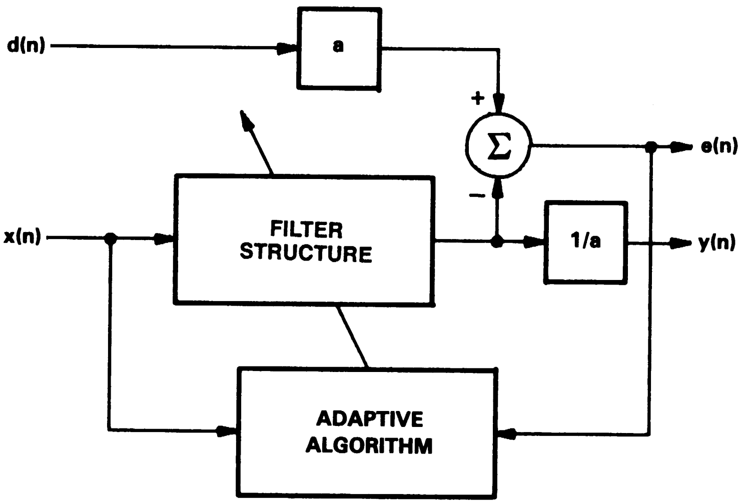


Figure 13. Fixed-Point Arithmetic Model of the Adaptive Filter

Finite Precision Errors

The TMS320C25 is a 16/32-bit fixed point processor. Each data sample is represented by a fractional number that uses 15 magnitude bits and one sign bit. The quantization interval

$$\delta = 2^{-b}, \tag{32}$$

(b = 15), is called the width of quantization since the numbers are quantized in steps of δ.

The products of the multiplications of data by coefficients within the filter must be rounded or truncated to store in memory or a CPU register. As shown in Figure 14, the roundoff error can be modeled as the white noise injected into the filter by each rounding operation. This white noise has a uniform distribution over a quantization interval and for rounding

$$-\,1/2\,\delta < e \leq 1/2\,\delta \tag{33a}$$

and

$$\delta_e^2 = (1/12)\ \delta^2 \qquad (33b)$$

where δ_e^2 is the variance of the white noise.

In general, roundoff noise occurs after each multiplication. However, the TMS320C25 has a full precision accumulator, i.e., a 16×16-bit multiplier with a 32-bit accumulator, so there is no roundoff when you implement a set of summations and multiplications as in Equation (29). Rounding is performed when the result is stored back to memory location y(n), so that only one noise source is presented in a given summation node.

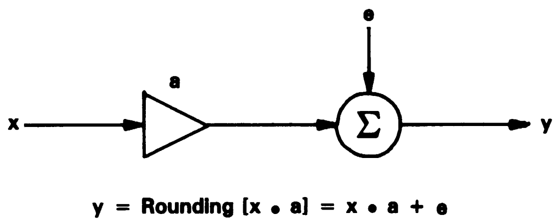


Figure 14. Fixed-Point Roundoff Noise Model

For floating-point arithmetic, the variance of the roundoff noise [31] is slightly different from Equation (33b),

$$\sigma_e^2 = 0.18\ \delta^2 \qquad (33c)$$

Since TMS320C30 has a 40/32-bit floating-point multiplier and ALU, the result from arithmetic operation has the mantissa of [31] bits plus one sign bit. Therefore, the δ in Equation (33c) is equal to 2^{-31}. Another roundoff noise is introduced when you restore the result back to memory. This noise has the power of 2^{-23} because the mantissa of TMS320C30 floating-point data is 23 bits plus one sign bit. Therefore, unless the filter order is high, the roundoff noise from arithmetic operation is relatively small.

The steady-state output error of the LMS algorithm due to the finite precision arithmetic of a digital processor was analyzed in reference [31]. It was found that the power of arithmetic errors is inversely proportional to the adaptation step size u. The significance of this result in the adaptive filter design is discussed next. Furthermore, roundoff noise is found to accumulate in time without bound, leading to an eventual overflow [32]. The leaky LMS algorithm presented in the previous section can be used to prevent the algorithm overflow.

Design Issues

The performance of digital adaptive algorithms differs from infinite precision adaptive algorithms. The finite precision LMS algorithm is given as

$$\underline{w}(n+1) = \underline{w}(n) + Q[u*e(n)*\underline{x}(n)] \tag{34}$$

where Q [.] denotes the operation of fixed point quantization. Whenever any correction term $u*e(n)*x(n-i)$ in the update of the weight vector in Equation (34) is too small, the quantized value of that term is zero, and the corresponding weight $w_i(n)$ remains unchanged. The condition for the ith component of the vector w(n) not to be updated when the algorithm is implemented with the TMS320C25 is

$$| u\ e(n)\ x(n-i) | < \delta/2 \tag{35a}$$

where $\delta = 2^{-15}$. The condition for TMS320C30 is

$$| u\ e(n)\ x(n-i) | < 2^{exp} * \delta/2 \tag{35b}$$

where exp is the exponent of $w_i(n)$ and $\delta = 2^{-23}$.

Since the adaptive algorithms are designed to minimize the mean squared value of the error signal, e(n) decreases with time. If u is small enough, most of the time the weights are not updated. This early termination of the adaptation may not allow the weight values to converge to the optimum set, resulting in a mean square error larger than its minimum value. The conditions for the adaptation to converge completely [30] is $u > u_{min}$ where

$$u^2_{min} = \frac{\delta^2}{4\sigma_x^2\epsilon_{min}} \tag{36a}$$

for the TMS320C25 and the TMS320C30

$$u^2_{min} = \frac{\delta^2 * 2^{exp}}{4\sigma_x^2\epsilon_{min}} \tag{36b}$$

where σ_x^2 is the power of input signal x(n) and ϵ_{min} is the minimum mean squared error at steady state.

In the Leaky LMS Algorithm section, it was mentioned that the excess MSE given in Equation (14) is minimized by using small u. However, this may result in a large quantization error since the most significant term in the total output quantization error is [31]

$$\frac{\cdot N\sigma_e^2}{2\ a^2\ u} \tag{37}$$

The optimum step size u_0 reflects a compromise between these conflicting goals. The value of u_0 is shown to be too small to allow the adaptive algorithm to converge completely and also to give a slow convergence. In practice, $u > u_0$ is used for faster convergence. Hence, the excess MSE becomes larger, and the roundoff noise can typically be neglected when compared with the excess mean square error.

Finally, recall Equations (11) and (12). The step size u has an upper limit to guarantee the stability and convergence. Therefore, the adaptive algorithm requires

$$0 < u < \frac{1}{N\sigma_x^2} \tag{38}$$

On the other hand, the step size u also has a lower limit. The optimum u_0, which minimizes the sum of the excess MSE and roundoff noise, is smaller than u_{min}, i.e., too small to allow the adaptive weight to converge. For an algorithm implemented on the TMS320C25, the word-length of 16 bits is fixed, and the minimum step-size that can be used is given in Equation (36). The most important design issue is to find the best u to satisfy

$$u_{min} < u < \frac{1}{N\sigma_x^2} \tag{39}$$

Therefore, in order to make the condition in Equation (39) valid, the initial values of filter coefficients are better close to zero for the floating-point processor if the situation in unknown.

Software Development

The TMS320C25 and TMS320C30 combine the high performance and the special features needed in adaptive signal processing applications. The processors are supported by a full set of software and hardware development tools. The software development tools include an assembler, a linker, a simulator, and a C compiler. The most universal software development tool available is a macro assembler. However, the assembly language programming for DSP can be tedious and costly. For adaptive filter applications, an assembly language programmer must have knowledge of adaptive signal processing. The challenge lies in compressing a great deal of complex code into the fairly small space and most efficient code dictated by the real-time applications typical of adaptive signal processing.

Recently, C compilers for the processors were developed to make DSP programming easier, quicker, and less costly compared with the work associated with programming in assembly language. Due to the general characteristics of a compiler, the code it generates is not the most efficient. Since the program efficiency consideration is important for adaptive filter implementation, the code generated from the C compiler has to be modified before implementing. Thus, two alternative ways, besides writing an assembly program, to implement adaptive signal processing on DSP are presented. First is the automatic adaptive filter code generator [12], which can be found on Texas Instruments TMS320 Bulletin Board Service (BBS), and second are the adaptive filter function libraries that support assembly and C programming languages.

In this report, two adaptive filter libraries have been developed: one can be called from an assembly main program; the other can be called from the C main program. Note that, for the TMS320C25 only, certain data memory locations have been reserved for storing the necessary filter coefficients, previous delayed signal, etc. In other words, these data memories are used as global variables.

Assembly Function Libraries

The basic concept of creating an assembly subroutine for an adaptive filter is to modify in module the assembly programs discussed above. Then, the user can implement the adaptive filter by writing his own assembly main program that calls the subroutine.

TMS320C25 Assembly Subroutine

The TMS320C25 has an eight-level deep hardware stack. The CALL and CALA subroutine calls store the current contents of the program counter (PC) on the top of the stack. The RET (return from subroutine) instruction pops the top of the stack back to the PC. For computational convenience, the processor needs to be set as follows before calling the assembly callable subroutine.

1. PM status bits equal to 01.
2. SXM status bit set to 1.
3. The current DP (data memory page pointer) is 0.

The following example is the TMS320C25 assembly main routine, which performs an adaptive line enhancement by calling the LMS algorithm subroutine. The filter order is 64, delay is equal to one, and the convergence factor u is 0.01.

```
*       DEFINE AND REFER SYMBOLS
*
                .global   ORDER,U,ONE,D,Y,ERR,XN,WN,LMS
*
```

```
        DEFINE SAMPLING RATE, ORDER, AND MU
*
ORDER:    .equ    20
MU:       .equ    327             ; mu = 0.01 in Q15 format
PAGE0:    .equ    0
*
        DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
X0:       .usect  ''buffer'',ORDER-1
XN:       .usect  ''buffer'',1
WN:       .usect  ''coeffs'',ORDER
*
*       RESERVE ADDRESSES FOR PARAMETERS
*
ONE:      .usect  ''parameters'',1
U:        .usect  ''parameters'',1
ERR:      .usect  ''parameters'',1
Y:        .usect  ''parameters'',1
D:        .usect  ''parameters'',1
ERRF:     .usect  ''parameters'',1
*
*       INITIALIZATION
*
START     LDPK    PAGE0           ; Set DP = 0
          SPM     1               ; Set PM equal to 1
          SSXM                    ; Set sign extension mode
          LRLK    AR7,X0          ; AR7 point to >300
          LACK    1               ; Initialize ONE = 1
          SACL    ONE
          LALK    MU              ; Initialize U = MU = 0.01
          SACL    U
**********************************************************************
*       PERFORM THE PREDICTOR
**********************************************************************
INPUT:    IN      D,PA2           ; Get the input
*
          CALL    LMS             ; Call subroutine
*
OUTPUT:   OUT     Y,PA2           ; Output the signal
*
          LAC     D               ; Insert the newest sample
          LARP    AR7
          SACL    *
          B       INPUT
          .end
```

The symbols, such as ORDER, U, ONE, D, LMS, Y, and ERR, are defined and referred to for the purpose of modular programming. The uninitialized sections specified by the directive .usect can be placed in any location of memory according to the linker command file. Note that MACD instruction requires the sources of the operands on program memory and data memory separately, and CNFP instruction configures RAM block 0 as program memory. Therefore, the coeffs section has to be in data RAM block 0, and the buffer has to be in RAM block 1. Appendix H1 contains the adaptive transversal filter with LMS algorithm subroutine using the TMS320C25, and Appendix H2 contains an example of a linker command file.

TMS320C30 Assembly Subroutine

Instead of a hardware stack, TMS320C30 uses a software stack, which is more flexible and convenient for a high-level language compiler. The stack memory location is pointed to by the stack pointer SP. In order to maintain the proper program sequence, the programmer must make certain that no data is lost and that the stack pointer always points to proper location. The PUSH, PUSHF, POP, POPF, CALL, CALLcond, RETIcond, and RETScond instructions will change the value of the stack pointer; in addition, writing data into it and using the interrupt will also change that value. It is the programmer's responsibility to initialize the stack pointer in the beginning of the program. The same adaptive line enhancer example above using TMS320C30 is listed below. The adapfltr.int program that initializes the stack pointer and the data RAM is given in Appendix H3.

```
*
*       DEFINE GLOBAL VARIABLES AND CONSTANTS
*
                .copy     "adapfltr.int"
                .global   LMS30,order,u,d,y,e
N               .set      20
mu              .set      0.01
*
*       INITIALIZE POINTERS AND ARRAYS
*
                .text
begin           .set      $
                LDI       N,BK                ; Set up circular buffer
                LDP       @xn_addr            ; Set data page
                LDI       @xn_addr,AR0        ; Set pointer for x[]
                LDI       @wn_addr,AR1        ; Set pointer for w[]
                LDF       0.0,R0              ; R0 = 0.0
                RPTS      N-1
                STF       R0,*AR0++(1)%       ; x[] = 0.
```

```
        ||STF    R0,*AR1++(1)%     ; w[] = 0.
          LDI    @in_addr,AR6      ; Set pointer for input ports
          LDI    @out_addr,AR7     ; Set pointer for output ports
*
*   PERFORM ADAPTIVE LINE ENHANCER
*
input:
          LDF    *AR6,R7           ; Input d(n)
        ||LDF    *+AR6(1),R6       ; Input x(n)
          STF    R7,@d             ; Insert d(n)
          STF    R6,*AR0           ; Insert x(n) to buffer
*
*   CALL ASSEMBLY SUBROUTINE
*
*         CALL   LMS30
*   OUTPUT y(n) AND e(n) SIGNALS
*
          LDF    @y,R6             ; Get y(n)
          BD     input             ; Delay branch
          LDF    @e,R7             ; Get e(n)
          STF    R6,*AR7           ; Send out y(n)
          STF    R7,*+AR7(1)       ; Send out e(n)
*
*   DEFINE CONSTANTS
*
n         .usect  "buffer",N
wn        .usect  "coeffs",N
in_addr   .usect  "vars",1
out_addr  .usect  "vars",1
xn_addr   .usect  "vars",1
wn_addr   .usect  "vars",1
u         .usect  "vars",1
order     .usect  "vars",1
d         .usect  "vars",1
y         .usect  "vars",1
e         .usect  "vars",1
cinit     .sect   ".cinit"
          .word   6,in_addr
          .word   0804000h
          .word   0804002h
          .word   xn
          .word   wn
```

```
        .float    mu
        .word     N-2
        .end
```

In the above example, data memory order is initialized to N−2 for computation convenience. The linker command files and the subroutine that implements the LMS transversal filter can be found in Appendixes H4 and H5.

C Function Libraries

The TMS320C25 and TMS320C30 C language compilers provide high-level language support for these processors. The compilers allow application developers without an extensive knowledge of the device's architecture and instruction set to generate assembly code for the device. Also, since C programs are not device-specific, it is a relatively straightforward task to port existing C programs from other systems.

To allow fast development of efficient programs for adaptive signal processing applications, C function libraries have been developed. These libraries include functions for adaptive transversal, symmetric transversal, and lattice structures.

TMS320C25 C-Callable Subroutines

In a C program, the memory assignments are chosen by the compiler. There are two ways to use the most efficient instruction MACD:

A. Use inline assembly code to assign memory locations for filter coefficients and buffers.

B. Reserve the desired memory locations for them and do the assignment in the linker command file.

The latter method is used in this report.

For a C main program, the parameters passed to and returned from the subroutines are all within the parentheses following the subroutine name, as shown below:

```
lms(n,mu,d,x,&y,&e)     n - Filter order
                        mu - Convergence factor
                        d - Desired signal
                        x - Input signal
                        y - Address of output signal
                        e - Address of error signal
```

Since the TMS320C25 C compiler pushes the parameters from right to left into software stack pointed by AR1 , the subroutine gets the parameters in reverse order, as shown below:

```
MAR     *-          ; Set pointer for getting parameters
LAC     *-          ; ACC = N
```

```
SUBK   1
SACL   ORDER        ; ORDER = N - 1
LAC    *-           ; Getting and storing the mu
SACL   U
LAC    *-           ; Getting and storing the D
SACL   D
LAC    *-,0,A-R3    ; Insert the newest sample
LRLK   AR3,FRSTAP
SACL   *
```

The assembly subroutine returns the parameters y and e as follows:

```
LARP   AR1
LAR    AR2,*-,AR2   ; Get the address of y in main
LAC    Y
SACL   *,0,AR1      ; Store y
LAR    AR2,*,AR2    ; Get the address of e in main
LAC    ERR
SACL   *,0,AR1      ; Store e
```

Therefore, the parameters should be entered in the order given above. If there are other parameters, they should be inserted right after the convergence factor mu. The leaky LMS algorithm subroutine is given as an example.

```
llms(n,mu,r,d,x,&y,&e)
```

the r is defined in Equation (28a). Note that the values of the AR registers, which will be used in subroutine, and the status registers must be saved at the beginning of the subroutine and restored right before returning to calling routine. An example of a C-callable program is given in Appendix I1. Memory locations 0200h to 0200h+N−1 and 0300h to 0300h+N−1 are reserved for filter coefficients and buffers, respectively. N denotes the filter order.

TMS320C30 C Subroutine

As previously mentioned, the TMS320C30 architecture has features designed for a high-level language compiler. Note that the callable word is dropped in this section title because the TMS320C30 is so flexible that the restrictions for the TMS320C25 no longer exist. Since the memory locations of filter buffers and coefficients are determined by the parameters that pass from the calling routine, the same subroutine can be used in different places. However, the only restriction is that the memory locations of filter buffers must align to the circular addressing boundary [14]. The features of TMS320C30 architecture that make a major contribution toward these improvements are dual data address buses, software stack, and flexible addressing mode. The parameters passed to subroutine are pushed into the stack. Therefore, after returning from the subroutine, the stack pointer, SP, must be updated to point to the location where SP pointed before pushing the parameters into the stack. However, this will be done by the C compiler. The usage example of the C function subroutine is given as follows:

```
tlms(n,u,d,&w,&x,&y,&e) where    n - Filter order
                                 u - Step size
                                 d - Desired signal
                                 &w - Filter coefficients
                                 &x - Input signal buffers
                                 &y - Addr of output signal
                                 &e - Addr of error signal
```

The example below shows how the C subroutine receives and manipulates the parameters passed from the caller program and how the result is returned to the caller routine.

```
*
*       SET FRAME POINTER FP
*
FP      .set     AR3
        PUSH     FP
        LDI      SP,FP
*
*       GET FILTER PARAMETERS
*
        LDI      *-FP(2),R4     ; Get filter order
        LDI      *-FP(6),AR0    ; Get pointer for x[]
        LDI      *--FP(5),AR1   ; Get pointer for w[]
*
*       COMPUTE ERROR SIGNAL e(n) AND STORE y(n) AND e(n)
*
        LDI      *-FP(2),AR2    ; Get y(n) address
        SUBF3    R2,*+FP(1),R7  ; e(n) = d(n) - y(n)
      ||STF      R2,*AR2        ; Send out y(n)
        LDI      *-FP(3),AR2    ; Get e(n) address
        STF      R7,*AR2        ; Send out e(n)
        MPYF     *+FP(2),R7     ; R7 = e(n) * u
        POP      FP
```

Note that AR3 is used as the frame pointer in TMS320C30 C compiler. Appendix I2 contains the complete LMS transversal filter example subroutine program.

Development Process and Environment

Following a four stage procedure [33] to minimize the amount of finite word length effect analysis and real-time debugging, adaptive structures and algorithms are implemented on the TMS320C25. Figure 15 illustrates the flowchart of this procedure. Since the implementation on TMS320C30 is done only by the simulator, the last stage, real-time testing, is not implemented.

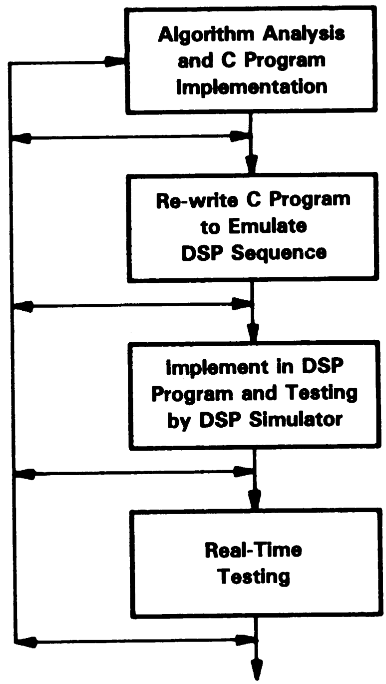


Figure 15. Adaptive Filter Implementation Procedure

In the first stage, algorithm design and study is performed on a personal computer. Once the algorithm is understood, the filter is implemented using a high-level C program with double precision coefficients and arithmetic. This filter is considered an ideal filter.

In the second stage, the C program is rewritten in a way that emulates the same sequence of operations with the same parameters and state variables that will be implemented in the processors. This program then serves as a detailed outline for the DSP assembly language program or can be compiled using TMS320C25 or TMS320C30 C compiler. The effects of numerical errors can be measured directly by means of the technique shown in Figure 16, where H(z) is the ideal filter implemented in the first stage and H'(z) is a real filter. Optimization is performed to minimize the quantization error and produce stable implementation.

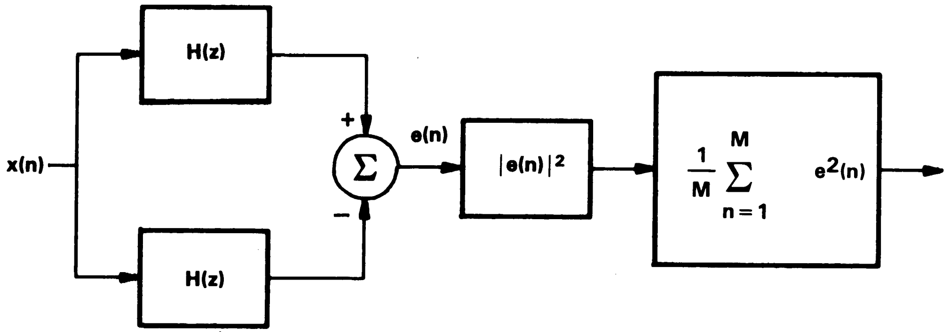


Figure 16. A Commutational Technique for Evaluating Quantization Effects

In the third stage, the TMS320C25 and TMS320C30 assembly programs are developed; then they are tested using the simulators with test data from a disk file. Note that the simulation of TMS320C25 can also be implemented on the SWDS with the data logging option. This test data is a short version of the data used in stage 2 that can be internally generated from a program or data digitized from a real application environment. Output from the simulation is compared against the equivalent output of the C program in the second stage. Since the simulation requires data files to be in Q15 format, certain precision is lost during data conversion. When a one-to-one agreement within tolerable range is obtained between these two outputs, the processor software is assured to be essentially correct.

The final stage is applied only to the TMS320C25. First, you download this assembled program into the target TMS320C25 system (SWDS) to initiate real-time operation. Thus, the real-time debugging process is constrained primarily to debugging the I/O timing structure of the algorithm and testing the long-term stability of the algorithm. Figure 17 shows an experimental setup for verification, in which the adaptive filter is configured for a one-step adaptive predictor illustrated in Figure 18. The data used for real-time testing is a sinusoid generated by a Tektronix FG504 Function Generator embedded in white noise generated by an HP Precision Noise Generator. The DSP gets a quantized signal from the Analog Interface Board (AIB), performs adaptive prediction routines, and outputs an enhanced sinusoid to the analog interface board. The corrupted input and predicted (enhanced) output waveforms are compared on the oscilloscope or on the HP 4361 Dynamic Signal Analyzer. The corresponding spectra of input and output can be compared on the signal analyzer. The signal-to-noise ratio (SNR) improvement can be measured from the analyzer, which is connected to an HP plotter.

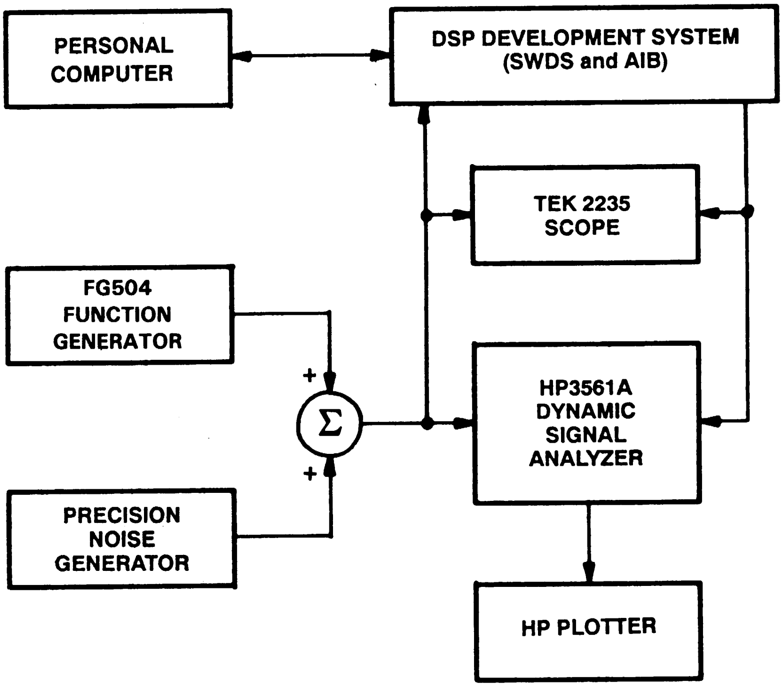


Figure 17. Real-Time Experiment Setup

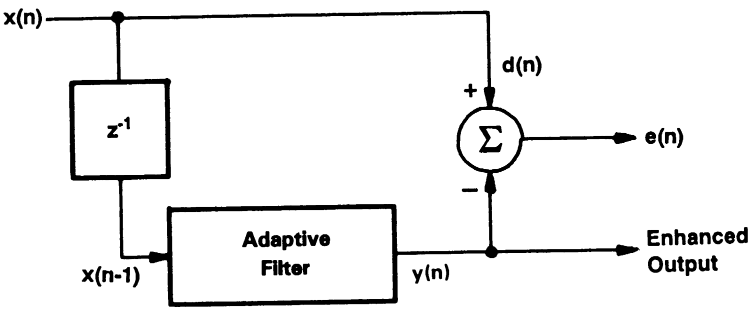


Figure 18. Block Diagram of a One-Step Adaptive Predictor

To illustrate the operation in a nonstationary environment, the adaptive predictor is implemented using a TMS320C25, and the following experiment is performed. The input signal is swept from 1287 Hz to 4025 Hz, then jumps back to 1287 Hz. The time for each sweep is one second. The input spectra at every second are shown in Figure 19a; the corresponding output spectra are shown in Figure 19b. From the observations on the oscilloscope and signal analyzer, the significant SNR improvement, convergence speed, ability to track nonstationary signals, and long-term stability of the adaptive predictor are observed.

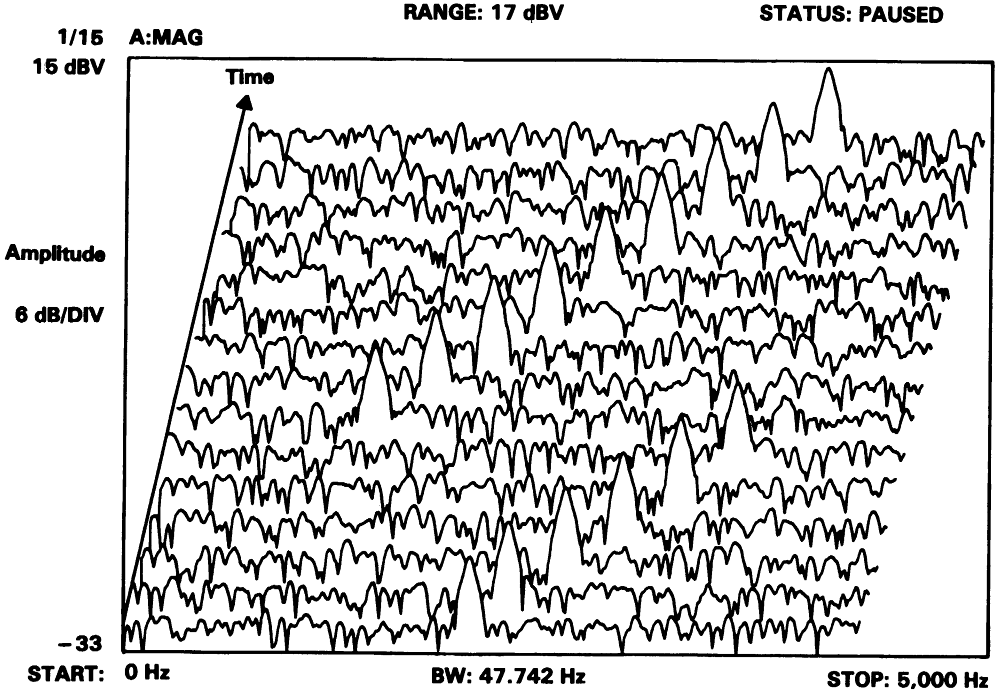


Figure 19(a). Spectrum of Input Signal

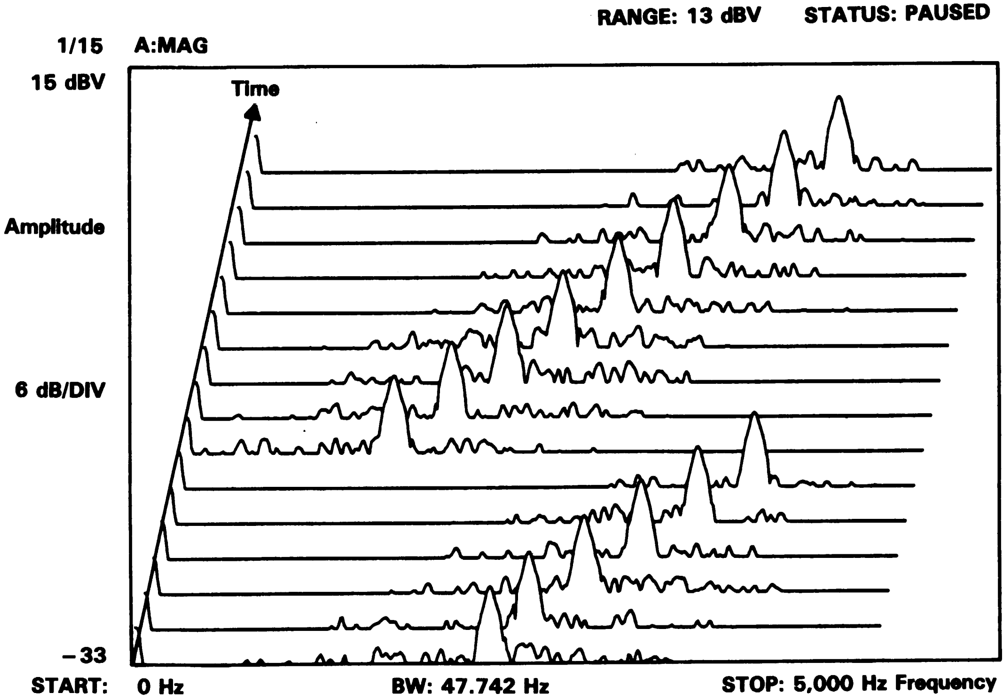


Figure 19(b). Spectrum of Enhanced Output Signal

Summary

Three adaptive structures and six update algorithms are implemented with the TMS320C25 and TMS320C30. Applications of adaptive filters and implementation considerations have been discussed. Two subroutine libraries that support both C language and assembly language for two processors were developed. These routines can be readily incorporated into TMS320C25 or TMS320C30 users' application programs.

The advancements in the TMS320C25 and TMS320C30 devices have made the implementation of sophisticated adaptive algorithms oriented toward performing real-time processing tasks feasible. Many adaptive signal processing algorithms are readily available and capable of solving real-time problems when implemented on the DSP. These programs provide an efficient way to implement the widely used structures and algorithms on the TMS320C25 and TMS320C30, based on assembly-language programming. They are also extremely useful for choosing an algorithm for a given application. The performances of adaptive structures and algorithms that have been implemented using the TMS320C25 and TMS320C30 have been summarized in Tables 1 and 2.

Table 1. The Performance of Adaptive Structures and Algorithms of TMS320C25

TMS320C25			
Transversal Structure	LMS	Instruction Cycles	7N+28
		Program Memory (Word)	33
	Leaky LMS	Instruction Cycles	8N+28
		Program Memory (Word)	34
	Sign-Data LMS	Instruction Cycles	11N+26
		Program Memory (Word)	41
	Sign-Error LMS	Instruction Cycles	7N+26
		Program Memory (Word)	30
	Sign-Sign LMS	Instruction Cycles	11N+21
		Program Memory (Word)	30
	Normalized LMS	Instruction Cycles	7N+57
		Program Memory (Word)	47
Symmetric Transversal Structure	LMS	Instruction Cycles	7.5N+38
		Program Memory (Word)	50
	Leaky LMS	Instruction Cycles	8N+38
		Program Memory (Word)	51
	Sign-Data LMS	Instruction Cycles	9.5N+36
		Program Memory (Word)	58
	Sign-Error LMS	Instruction Cycles	7.5N+36
		Program Memory (Word)	47
	Sign-Sign LMS	Instruction Cycles	9.5N+31
		Program Memory (Word)	47
	Normalized LMS	Instruction Cycles	7.5N+69
		Program Memory (Word)	66
Lattice Structure	LMS	Instruction Cycles	33N+32
		Program Memory (Word)	63
	Leaky LMS	Instruction Cycles	35N+32
		Program Memory (Word)	65
	Sign-Error LMS	Instruction Cycles	36N+32
		Program Memory (Word)	65
	Normalized LMS	Instruction Cycles	90N+34
		Program Memory (Word)	92

Note: N represents filter order.

Table 2. The Performance of Adaptive Structures and Algorithms of TMS320C30

TMS320C30			
Transversal Structure	LMS	Instruction Cycles	3N + 15
		Program Memory (Word)	17
	Leaky LMS	Instruction Cycles	3N + 15
		Program Memory (Word)	19
	Sign-Data LMS	Instruction Cycles	5N + 16
		Program Memory (Word)	24
	Sign-Error LMS	Instruction Cycles	3N + 16
		Program Memory (Word)	18
	Sign-Sign LMS	Instruction Cycles	5N + 16
		Program Memory (Word)	24
	Normalized LMS	Instruction Cycles	3N + 47
		Program Memory (Word)	49
Symmetric Transversal Structure	LMS	Instruction Cycles	2.5N + 15
		Program Memory (Word)	23
	Leaky LMS	Instruction Cycles	2.5N + 19
		Program Memory (Word)	26
	Sign-Data LMS	Instruction Cycles	3.5N + 18
		Program Memory (Word)	30
	Sign-Error LMS	Instruction Cycles	2.5N + 18
		Program Memory (Word)	24
	Sign-Sign LMS	Instruction Cycles	3.5N + 17
		Program Memory (Word)	30
	Normalized LMS	Instruction Cycles	2.5N + 50
		Program Memory (Word)	56
Lattice Structure	LMS	Instruction Cycles	14N + 9
		Program Memory (Word)	20
	Leaky LMS	Instruction Cycles	16N + 9
		Program Memory (Word)	22
	Sign-Error LMS	Instruction Cycles	16N + 9
		Program Memory (Word)	22
	Normalized LMS	Instruction Cycles	67N + 9
		Program Memory (Word)	73

Note: N represents filter order.

References

[1] B. Widrow and S. Stearns, *Adaptive Signal Processing*, Prentice-Hall, 1985.

[2] R. Lucky, J. Salz, and E. Weldon, *Principles of Data Communications*, McGraw-Hill, 1968.

[3] S. Haykin, *Adaptive Filter Theory*, Prentice-Hall, 1986.

[4] M. Honig and D. Messerschmit, *Adaptive Filters: Structures, Algorithms, and Applications*, Kluwer Academic, 1984.

[5] J.R. Treichler, C.R. Johnson, and M.G. Larimore, *Theory and Design of Adaptive Filters*, Wiley, 1987.

[6] T. Alexander, *Adaptive Signal Processing*, Springer-Verlag, 1986.

[7] G. Goodwin and K. Sin, *Adaptive Filtering Prediction and Control*, Prentice-Hall, 1984.

[8] M. Bellanger, *Adaptive Digital Filters and Signal Analysis*, Marcel Dekker, 1987.

[9] J. Proakis, *Digital Communications*, McGraw-Hill, 1983.

[10] C. Chen and S. Kuo, "An Interactive Software Package for Adaptive Signal Processing on an IBM Person Computer," *19th Pittsburgh Conference on Modeling and Simulation*, May 1988.

[11] S. Kuo, G. Ranganathan, P. Gupta, and C. Chen, "Design and Implementation of Adaptive Filters," *IEEE 1988 International Conference on Circuits and Systems*, June 1988.

[12] S. Kuo, G. Ma, and C. Chen, "An Advanced DSP Code Generator for Adaptive Filters," *1988 ASSP DSP workshop*, Sept. 1988.

[13] Texas Instruments, *Second-Generation TMS320 User's Guide*, 1987.

[14] Texas Instruments, *Third-Generation TMS320 User's Guide*, 1988.

[15] S. Qureshi, "Adaptive Equalization," Invited Paper, *Proceedings of the IEEE*, Sept. 1985.

[16] L. Rabiner and R. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, 1978.

[17] N. Jayant and P. Noll, *Digital Coding of Waveforms: Principles and Applications to Speech and Video*, Prentice-Hall, 1984.

[18] J. Makhoul, "Linear Prediction: A Tutorial Review," *Proceedings of the IEEE*, April 1975.

[19] C. Cowan and P. Grant, *Adaptive Filters*, Prentice-Hall, 1985.

[20] C. Gritton and D. Lin, "Echo Cancellation Algorithms," *IEEE ASSP Magazine*, April 1984.

[21] D. Messerschmitt, et al, "Digital Voice Echo Canceller with a TMS32020," in *Digital Signal Processing Applications with the TMS320 Family*, Prentice-Hall, 1986.

[22] B. Widrow, et al, "Adaptive Noise Cancelling: Principles and Applications," *Proceedings of the IEEE*, December 1975.

[23] A. Lovrich and R. Simar, "Implementation of FIR/IIR Filter with the TMS32010/TMS32020," in *Digital Signal Processing Applications with the TMS320 Family*, Texas Instruments, 1986.

[24] S. Orfanidis, *Optimum Signal Processing*, MacMillan, 1985.
[25] G. Frantz, K. Lin, J. Reimer, and J. Bradley, ''The Texas Instruments TMS320C25 Digital Signal Microcomputer,'' *IEEE Micro*, December 1986.
[26] B. Friedlander, ''Lattice Filters for Adaptive Processing,'' *Proceedings of the IEEE*, August 1982.
[27] A. Gersho, ''Adaptive Filtering with Binary Reinforcement,'' *IEEE Transactions on Information Theory*, March 1984.
[28] A. Oppenheim and R. Schafer, *Digital Signal Processing*, Chap. 9, Prentice-Hall, 1975.
[29] L. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Chap. 5, Prentice-Hall, 1975.
[30] J. R. Gitlin et al, ''On the Design of Gradient Algorithms for Digitally Implemented Adaptive Filters,'' *IEEE Transactions on Circuit Theory*, March 1973.
[31] C. Caraiscos and B. Liu, ''A Roundoff Error Analysis of the LMS Adaptive Algorithm,'' *IEEE Transactions on ASSP*, February, 1984.
[32] J. Cioffi, ''Limited-Precision Effects in Adaptive Filtering,'' *IEEE Transactions on Circuits and Systems*, July 1987.
[33] R. Crochier, R. Cox, and J. Johnson, ''Real-Time Speech Coding,'' *IEEE Transactions on Communications*, April 1982.

List of Appendices for Implementation of Adaptive Filters with the TMS320C25 and TMS320C30

Appendix A1. Transversal Structure with LMS Algorithm Using the TMS320C25

```
        .title   'TLMS'
*****************************************************************
*
*   TLMS :  Adaptive Filter Using Transversal Structure
*           and LMS Algorithm, Looped Code
*
*   d(n) --------------------------:
*                                  :
*                                  :+
*                             (SUM)-------> e(n)
*                                  :-
*                   ------         :
*   x(n) ----------: AF :----------:--------> y(n)
*                   ------
*
*   Algorithm:
*
*             63
*     y(n) = SUM w(k)*x(n-k)  k=0,1,2,...,63
*            k=0
*
*     e(n) = d(n) - y(n)
*
*     w(k) = w(k) + u*e(n)*x(n-k) k=0,1,2,..63
*
*     Where we use filter order = 64 and mu = 0.01.
*
*   Note:  This source program is the generic version; I/O configuration has
*          not been set up. User has to modify the main routine for specific
*          application.
*
*   Initial condition:
*          1) PM status bit should be equal to 01.
*          2) SXM status bit should be set to 1.
*          3) The current DP (data memory page pointer) should be page 0.
*          4) Data memory ONE should be 1.
*          5) Data memory U should be 327.
*
*                 Chen, Chein-Chung  February, 1989
*
*****************************************************************
*
*   DEFINE PARAMETERS
*
ORDER:     .equu    64
PAGE0:     .equ     0
*
*   DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
X0:        .usect   "buffer",ORDER-1
XN:        .usect   "buffer",1
WN:        .usect   "coeffs",ORDER
*   RESERVE ADDRESSES FOR PARAMETERS
*
D:         .usect   "parameters",1
Y:         .usect   "parameters",1
ERR:       .usect   "parameters",1
ONE:       .usect   "parameters",1
U:         .usect   "parameters",1
ERRF:      .usect   "parameters",1
*********************************
*   PERFORM THE ADAPTIVE FILTER
*********************************
           .text
*
*   ESTIMATE THE SIGNAL Y
*
           LARP    AR3
           CNFP                       ; Configure B0 as program memory
           MPYK    0                  ; Clear the P register
           LAC     ONE,15             ; Using rounding
           LRLK    AR3,XN             ; point to the oldest sample
FIR        RPTK    ORDER-1            ; Repeat N times
           MACD    WN+0fd00h,*-       ; Estimate Y(n)
           CNFD                       ; Configure B0 as data memory
           APAC
           SACH    Y                  ; Store the filter output
*
*   COMPUTE THE ERROR
*
           NEG                        ; ACC = - Y(n)
           ADDH    D
           SACH    ERR                ; ERR(n) = D(n) - Y(n)
*
*   UPDATE THE WEIGHTS
*
           LT      ERR                ; T = ERR(n)
           MPY     U                  ; P = U * ERR(n)
           PAC
           ADD     ONE,15             ; Round the result
           SACH    ERRF               ; ERRF = U * ERR(n)
*
           LARK    AR1,ORDER-1        ; Set up counter
           LRLK    AR2,WN             ; Point to the coefficients
           LRLK    AR3,XN+1           ; Point to the data sample
           LT      ERRF               ; T register = U * ERR(n)
           MPY     *-,AR2             ; P = U * ERR(n) * X(n-k)
ADAPT      ZALR    *,AR3              ; Load ACCH with A(k,n) & round
           MPYA    *-,AR2             ; W(k,n+1) = W(k,n) + P
*                                     ; P = U * ERR(n) * X(n-k)
           SACH    *+,0,AR1           ; Store W(k,n+1)
           BANZ    ADAPT,*-,AR2
*
FINISH     .end
```

Appendix A2. Transversal Structure with LMS Algorithm Using the TMS320C30

```
****************************************************************
*
*   T30 - Adaptive transversal filter with LMS algorithm
*         using the TMS320C30
*
*   I/O configuration:
*
*    d(n) -------------------------:
*                                  :
*                                  :+
*                             (SUM)------> e(n)
*                                  :-
*                  ------          :
*    x(n) ---------: AF :----------:--------> y(n)
*                  ------
*
*    Algorithm:
*
*              63
*       y(n) = SUM w(k)*x(n-k) k=0,1,2,...,63
*              k=0
*
*       e(n) = d(n) - y(n)
*
*       w(k) = w(k) + u*e(n)*x(n-k) k=0,1,2,..63
*
*       Where we use filter order = 64 and mu = 0.01.
*
*                   Chen, Chein-Chung  March, 1989
*
************************************************************************
          .copy     "adapfltr.int"
***************************************************************
*    PERFORM ADAPTIVE FILTER
***************************************************************
order      .set      64
mu         .set      01
*
*    INITIALIZE POINTERS AND ARRAYS
*
           .text
begin      .set      $
           LDI       order,BK            ; Set up circular buffer
           LDP       @xn_addr            ; Set data page
           LDI       @xn_addr,AR0        ; Set pointer for x[]
           LDI       @wn_addr,AR1        ; Set pointer for w[]
           LDF       0.0,R0              ; R0 = 0.0
           RPTS      order-1
           STF       R0,*AR0++(1)%       ; x[] = 0
      ::   STF       R0,*AR1++(1)%       ; w[] = 0
           LDI       @in_addr,AR6        ; Set pointer for input ports
           LDI       @out_addr,AR7       ; Set pointer for output ports

input:
           LDF       *AR6,R7             ; Input d(n)
      ::   LDF       *+AR6(1),R6         ; Input x(n)
           STF       R6,*AR0             ; Insert x(n) to buffer
*
*    COMPUTE FILTER OUTPUT y(n)
*
           LDF       0.0,R2              ; R2 = 0.0
           MPYF3     *AR0++(1)%,*AR1++(1)%,R1
           RPTS      order-2
           MPYF3     *AR0++(1)%,*AR1++(1)%,R1
      ::   ADDF3     R1,R2,R2            ; y(n) = w[].x[]
           ADDF      R1,R2               ; Include last result
*
*    COMPUTE ERROR SIGNAL e(n) AND OUTPUT y(n) AND e(n) SIGNALS
*
           SUBF      R2,R7               ; e(n) = d(n) - y(n)
           STF       R2,*AR7             ; Send out y(n)
      ::   STF       R7,*+AR7(1)         ; Send out e(n)
*
*    UPDATE WEIGHTS w(n)
*
           MPYF      @u,R7               ; R7 = e(n) * u
           MPYF3     *AR0++(1)%,R7,R1    ; R1 = e(n) * u * x(n)
           LDI       order-3,RC          ; Initialize repeat counter
           RPTB      LMS                 ; Do i = 0, N-3
           MPYF3     *AR0++(1)%,R7,R1    ; R1 = e(n) * u * x(n-i-1)
      ::   ADDF3     *AR1,R1,R2          ; R2 = wi(n) + e(n) * u * x(n-i)
LMS        STF       R2,*AR1++(1)%       ; wi(n+1) = wi(n) + e(n) * u * x(n-i)
           MPYF3     *AR0,R7,R1          ; For i = N - 2
      ::   ADDF3     *AR1,R1,R2
           BD        input               ; Delay branch
           STF       R2,*AR1++(1)%       ; wi(n+1) = wi(n) + e(n) * u * x(n-i)
           ADDF3     *AR1,R1,R2
           STF       R2,*AR1++(1)%       ; Update last w
*
*    DEFINE CONSTANTS
*
xn         .usect    "buffer",order
wn         .usect    "coeffs",order
in_addr    .usect    "vars",1
out_addr   .usect    "vars",1
xn_addr    .usect    "vars",1
wn_addr    .usect    "vars",1
u          .usect    "vars",1
cinit      .sect     ".cinit"
           .word     5,in_addr
           .word     0804000h
           .word     0804002h
           .word     xn
           .word     wn
           .float    mu
           .end
```

Appendix B1. Symmetric Transversal Structure with LMS Algorithm Using the TMS320C25

```
        .title   'Y25'
*****************************************************************
*
*   Y25 :  Adaptive Filter Using Symmetry Transversal Structure
*            and LMS Algorithm, Looped Code
*
*   d(n) -------------------------------------------:
*                                                   :
*                         ---------  y(n)     +  :+
*                         :  A. F. :----------->(SUM)---> e(n)
*                         ---------
*                   z1(n)      : : : :    z1(n-k)
*              :-----------: : : :-------------:
*              :           :---: :-----:       :
*              :           :        :          :
*              :   :---:   :   :---:    :  :---:  :
*   x(n) -------:---: Z :--:---: Z :...:---: Z :--:-----:
*              : : :---:  : : :---:   : : :---:  : :   :
*              :/         :/          :/        :/  :---:
*            (SUM)      (SUM)       (SUM)     (SUM) : Z :
*              :          :           :         :   :---:
*              \  :---:   \  :---:    \  :---:   \    :
*               :-: Z :----:-: Z :...--:-: Z :----:---:
*                 :---:      :---:       :---:
*
*   Algorithm:
*
*      z1(n-k) = x(n-k) + x(n-63+k) k=0,1,...,31
*
*            31
*      y(n) = SUM w(k)*x(n-k)  k=0,1,2,...,31
*            k=0
*
*      e(n) = d(n) - y(n)
*
*      w(k) = w(k) + u*e(n)*z1(n-k) k=0,1,2,..31
*
*      Where we use filter order = 64 and mu = 0.01.
*
*   Note:  This source program is the generic version; I/O configuration has
*          not been set up. User has to modify the main routine for specific
*          application.
*
*   Initial condition:
*          1) PM status bit should be equal to 01.
*          2) SXM status bit should be set to logic 1.
*          3) The current DP (data memory page pointer should be page 0.
*          4) Data memory ONE should be 1.
*          5) Data memory U should be 327.
*
*                 Chen, Chein-Chung  February, 1989
```

```
*****************************************************
*
*   DEFINE PARAMETERS
*
ORDER:      .equ      64
ORDER2:     .equ      32
*
*   DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
FRSBUF:     .usect    "buffer",ORDER2-1
LASBUF:     .usect    "buffer",1
WN:         .usect    "coeffs",ORDER2
FRSDAT:     .usect    "coeffs",ORDER-1
LASDAT:     .usect    "coeffs",1
*
*   RESERVE ADDRESSES FOR PARAMETERS
*
D:          .usect    "parameters",1
Y:          .usect    "parameters",1
ERR:        .usect    "parameters",1
ONE:        .usect    "parameters",1
U:          .usect    "parameters",1
ERRF:       .usect    "parameters",1
**********************************
*   PERFORM THE ADAPTIVE FILTER
**********************************
            .text
*
*   SYMMETRIC BUFFER ADDITION
*
            LARP      AR3
            LARK      AR1,ORDER2-1        ; Set up the counter
            LRLK      AR2,LASDAT          ; Point to oldest data
            LRLK      AR3,FRSDAT          ; Point to newest data
            LRLK      AR4,FRSBUF          ; Point to first buffer
SYM         LAC       *+,0,AR2
            ADD       *-,0,AR4
            SACL      *+,0,AR1            ; Buffer(k) = DAT(n+k) + DAT(n-N+k)
            BANZ      SYM,*-,AR3
*
*   ESTIMATE THE SIGNAL Y
*
            CNFP                          ; Configure B0 as program memory
            MPYK      0                   ; Clear the P register
            LAC       ONE,15              ; Using rounding
            LRLK      AR3,LASBUF          ; Point to the oldest buffer
FIR         RPTK      ORDER2-1            ; Repeat N/2 time
            MACD      WN+0fd00h,*-        ; Estimate Y(n)
            CNFD                          ; Configure B0 as data memory
            APAC
            SACH      Y                   ; Store the filter output
*
*   COMPUTE THE ERROR
```

```
        NEG                         ; ACC = - Y(n)
        ADD     D,15
        SACH    ERR                 ; ERR(n) = D(n) - Y(n)
*
*   UPDATE THE WEIGHTS
*
        LT      ERR                 ; T = ERR(n)
        MPY     U                   ; P = U * ERR(n)
        PAC
        ADD     ONE,15              ; Round the result
        SACH    ERRF                ; ERRF = U * ERR(n)
*
        LARK    AR1,ORDER2-1        ; Set up counter
        LRLK    AR2,WN              ; Point to the coefficients
        LRLK    AR3,LASBUF          ; Point to the last buffer
        LT      ERRF                ; T register = U * ERR(n)
        MPY     *-,AR2              ; P = U * ERR(n) * X(n-k)
ADAPT   ZALR    *,AR3               ; Load ACCH with A(k,n) & round
        MPYA    *-,AR2              ; W(k,n+1) = W(k,n) + P
*                                   ; P = U * ERR(n) * X(n-k)
        SACH    *+,0,AR1            ; Store W(k,n+1)
        BANZ    ADAPT,*-,AR2
*
*   UPDATE DATA POSTION FOR NEXT ITERATION
*
FINISH  LRLK    AR2,LASDAT-1        ; Set pointer
DATMOV  RPTK    ORDER-2             ; Repeat N-1 times
        DMOV    *-                  ; Shift data for next iteration
*
        .end
```

Appendix B2. Symmetric Transversal Structure with LMS Algorithm Using the TMS320C30

```
****************************************************************
*
*   Y30 - Adaptive symmetric transversal filter with
*         LMS algorithm using the TMS320C30
*
*    Algorithm:
*       z(n-k) = x(n-k-1) + x(n-63+k) k=0,1,...,31
*                31
*       y(n) = SUM w(k)*z(n-k) k=0,1,2,...,31
*              k=0
*
*       e(n) = d(n) - y(n)
*
*       w(k) = w(k) + u*e(n)*z(n-k) k=0,1,2,..31
*
*       Where we use filter order = 64 and mu = 0.01
*
****************************************************
*     PERFORM ADAPTIVE FILTER
****************************************************
            .copy   "adapfltr.int"
order       .set    64              ; Filter order
mu          .set    0.01            ; Step size
*
*   INITIALIZE POINTERS AND ARRAYS
            .text
begin       .set    $
            LDI     order,BK        ; Set up circular buffer
            LDP     @xn_addr        ; Set data page
            LDI     @xn_addr,AR0    ; Set pointer for x[]
            LDI     @wn_addr,AR1    ; Set pointer for w[]
            LDI     @zn_addr,AR2    ; Set pointer for z[]
            LDI     order/2-1,IR0   ; Set index pointer
            LDF     0.0,R0          ; S0 = 0.0
            RPTS    order-1
            STF     R0,*AR0++(1)%   ; x[] = 0
            RPTS    order/2-2
            STF     R0,*AR1++(1)    ; w[] = 0
         :: STF     R0,*AR2++(1)    ; z[] = 0
            STF     R0,*AR1--(IR0)  ; w[] = 0
         :: STF     R0,*AR2--(IR0)  ; z[] = 0
            LDI     @in_addr,AR6    ; Set pointer for input ports
            LDI     @out_addr,AR7   ; Set pointer for output ports
input:
            LDF     *AR6,R7         ; Input d(n)
         :: LDF     *+AR6(1),R6     ; Input x(n)
            LDI     AR0,AR4         ; Set forward pointer for x[]
            STF     R6,*AR0--(1)%   ; Insert x(n) to buffer
*
*    COMPUTE FILTER OUTPUT y(n)
*
            LDF     0.0,R2              ; R2 = 0.0
            LDI     AR0,AR5             ; Set backward pointer for x[]
            LDI     order/2-2,RC
            RPTB    INNER
*
            ADDF3   *AR4++(1)%,*AR5--(1)%,R1
                                        ; z(n) = x[n-i] + x[n+N-i]
*
            MPYF3   R1,*AR1++(1),R3     ; y[] = w[].z[]
         :: STF     R1,*AR2++(1)        ; Store z(n)
INNER       ADDF3   R3,R2,R2            ; Accumulate the result
*
            ADDF3   *AR4++(1)%,*AR5--(1)%,R1
                                        ; z(n) = x[n-i] + x[n+N-i]
*
            MPYF3   R1,*AR1--(IR0),R3   ; y[] = w[].z[]
         :: STF     R1,*AR2--(IR0)      ; Store z(n)
            ADDF    R3,R2               ; Include last result
*
*    COMPUTE ERROR SIGNAL e(n) AND OUTPUT y(n) and e(n) SIGNALS
*
            SUBF    R2,R7               ; e(n) = d(n) - y(n)
            STF     R2,*AR7             ; Send out y(n)
         :: STF     R7,*+AR7(1)         ; Send out e(n)
*
*    UPDATE WEIGHTS w(n)
*
            MPYF    @u,R7               ; R7 = e(n) * u
            MPYF3   *AR2++(1),R7,R1     ; R1 = e(n) * u * z(n)
            LDI     order/2-3,RC        ; Initialize repeat counter
            RPTB    LMS                 ; Do i = 0, N-3
            MPYF3   *AR2++(1),R7,R1     ; R1 = e(n) * u * z(n-i-1)
         :: ADDF3   *AR1,R1,R2          ; R2 = wi(n) + e(n) * u * z(n-i)
LMS         STF     R2,*AR1++(1)        ; wi(n+1) = wi(n) + e(n) * u * z(n-i)
            MPYF3   *AR2--(IR0),R7,R1   ; For i = N - 2
         :: ADDF3   *AR1,R1,R2
            BD      input               ; Delay branch
            STF     R2,*AR1++(1)        ; wi(n+1) = wi(n) + e(n) * u * z(n-i)
            ADDF3   *AR1,R1,R2          ; Include last w
            STF     R2,*AR1--(IR0)      ; Update last w
*
*    DEFINE CONSTANTS
*
xn          .usect  "buffer",order
wn          .usect  "coeffs",order/2
zn          .usect  "coeffs",order/2
in_addr     .usect  "vars",1
out_addr    .usect  "vars",1
xn_addr     .usect  "vars",1
wn_addr     .usect  "vars",1
zn_addr     .usect  "vars",1
u           .usect  "vars",1
cinit       .sect   ".cinit"
            .word   6,in_addr
```

```
.word     0804000h
.word     0804002h
.word     xn
.word     wn
.word     zn
.float    mu
.end
```

Appendix C1. Lattice Structure with LMS Algorithm Using the TMS320C25

```
        .title    'L25'
*****************************************************************
*
*    L25 :  Adaptive Filter Using Lattice Structure
*           and LMS Algorithm, Looped Code
*
*      f0(n)    +  -     f1(n)     fi-1(n)          +
*        ----:-->(SUM)-------->....----------:-->(SUM)----->fi(n)
*        :   :     :-                        :     :-
*        :   :-----:--:                      :-----:--:
*        :  *k0    :  :                *ki-1      :  :
*  x(n)--:         :  :                           :  :
*        :      *k0 :  :                   *ki-1 :  :
*        :      :---:  :                    :---:  :
*        : :-: :      -:                 :-: :      -:
*        --:Z:-:-->(SUM)----->....--------:Z:-:-->(SUM)--->bi(n)
*     b0(n):-:    +      b1(n)    bi-1(n) :-:       +
*
*    Algorithm:
*
*      fi(n) = fi-1(n) - Ki(n) * bi-1(n-1)  i=1,2,...,64
*
*      bi(n) = bi-1(n-1) - Ki(n) * fi-1(n)  i=1,2,...,64
*                       i-1
*      ei(n) = d(n) - SUM yk(n) = ei-1 - bi-1(n)*Gi-1(n)  i=1,2,...,64
*                       k=0
*              64          64
*      y(n) = SUM yi(n) = SUM bi(n)*Gi(n)
*              i=0         i=0
*
*      Ki(n+1) = Ki(n) + mu * [ fi(n)*bi-1(n-1) + bi(n)*fi-1(n) ]
*
*      Gi(n+1) = Gi(n) + mu * ei(n) * bi(n)  i=1,2,..64
*
*      Where filter order = 64 and mu = 0.01.
*
*    Note: This source program is the generic version; I/O configuration has
*          not been set up. User has to modify the main routine for specific
*          application.
*
*    Initial condition:
*          1) PM status bit should be equal to 01.
*          2) SXM status bit should be set to logic 1.
*          3) The current DP (data memory page pointer) should be page 0.
*          4) Data memory U should be 327.
*          5) The B1 & BO1 pointer (AR3 & AR4) should be exchanged every
*             iteration. For example,
*                For odd iteration:  AR3 --> B1
*                                    AR4 --> BO1
*                For even iteration: AR3 --> BO1
*                                    AR4 --> B1
*
```

```
*                   Chen, Chein-Chung  February, 1989
*****************************************************************
*
*    DEFINE PARAMETERS
*
ORDER:      .equ     64
*
*    DEFINE ADDRESSES OF BUFFERS AND COEFFICIENTS
*
G1:         .usect   "coeffs",ORDER
K1:         .usect   "coeffs",ORDER
F1:         .usect   "coeffs",ORDER+1
B1:         .usect   "buffer",ORDER+1
BO1:        .usect   "buffer",ORDER+1
*
*    RESERVE ADDRESSES FOR PARAMETERS
*
D:          .usect   "parameters",1
X:          .usect   "parameters",1
Y:          .usect   "parameters",1
E:          .usect   "parameters",1
U:          .usect   "parameters",1
TEMP:       .usect   "parameters",1
***********************************
*    PERFORM THE ADAPTIVE FILTER
***********************************
            .text
*
*    INITIALIZE THE POINTERS
*
            LARP     AR3
            LARK     AR1,ORDER-1
            LRLK     AR2,F1
            LRLK     AR3,B1
            LRLK     AR4,BO1
            LRLK     AR5,G1
            LRLK     AR6,K1
*
*    INITIALIZE THE B1 AND F1
*
            LAC      X
            SACL     *,0,AR2
            SACL     *,0,AR3
*
*    INITIALIZATION
*
            LT       *,AR5            ; T = B1
            MPY      *,AR2            ; P = B1 * G1
            PAC                       ; ACC = B1 * G1
            SACH     Y                ; Initialize Y(0) = B1 * G1
            NEG                       ; ACC = -(B1 * G1)
            ADDH     D                ; ACC = D(n) - B1 * G1
            SACH     E                ; Initialize E(0) = D(n) - B1 * G1
*
```

```
*    COMPUTE THE Fi AND Bi
*
LAT1      ZALR    *,AR6          ; ACC = Fi-1
          LT      *,AR4          ; T = ki
          MPY     *,AR2          ; P = ki * BDi-1
          MPYS    *+             ; ACC = Fi-1 - ki*BDi-1, P = ki*Fi-1
          SACH    *-,0,AR4       ; Store Fi
          ZALR    *,AR3          ; ACC = BDi-1
          MPYS    *+             ; ACC = BDi-1 - ki * Fi-1
          SACH    *-             ; Store Bi
*
*    COMPUTE GAIN G(n)
*
          LT      U              ; T = MU
          MPY     E              ; P = MU * Ei-1
          SPH     TEMP           ; Store MU * Ei-1
          LT      TEMP           ; T = MU * Ei-1
          MPY     *+,AR5         ; P = MU * Ei-1 * Bi-1
          ZALR    *,AR3          ; ACC = Gi(n)
          LTA     *,AR5          ; ACC = Gi(n) + MU*Ei-1*Bi-1, T = Bi
NEXT1     SACH    *+,0,AR5       ; Store Gi(n+1)
*
*    COMPUTE E AND UPDATE Ki
*
          MPY     *,AR2          ; P = Bi * Gi
          ZALR    E              ; ACC = Ei-1
          MPYS    *+,AR2         ; ACC = Ei-1 - Bi * Gi, P = Bi*Fi-1
          SACH    E              ; Store Ei
          LTP     *,AR4          ; T = Fi, ACC = Bi*Fi-1
          MPY     *+,AR6         ; P = Fi * BDi-1
          APAC                   ; ACC = Bi*Fi-1 + BDi-1 * Fi
          SACH    TEMP
          LT      TEMP           ; T = Bi*Fi-1 + BDi-1 * Fi
          MPY     U              ; P = MU * (Fi*BDi-1+Fi-1*Bi)
          ZALR    *              ; ACC = Ki(n)
          APAC                   ; ACC = Ki(n) + MU
*                                ; * (Fi*BDi-1+Fi-1*Bi)
          SACH    *+,0,AR1       ; Store Ki(n+1)
          BANZ    LAT1,*-,AR2
*
*    COMPUTE Y
*
          CNFP                   ; Configure B0 as program memory
          MPYK    0              ; Clear the P register
          ZAC                    ; Clear accumulator
          LRLK    AR2,B1         ; Set the pointer
FIR       RPTK    ORDER-1        ; Repeat N times
          MAC     G1+0fd00h,*+   ; Compute Y
          CNFD                   ; Configure B0 as data memory
          APAC                   ; Include last data
          SACH    Y              ; Store the filter output
*
          .end
```

Appendix C2. Lattice Structure with LMS Algorithm Using the TMS320C30

```
****************************************************************
*   L30 :  Adaptive Lattice Structure Filter with LMS Algorithm
*          using the TMS320C30
*
*   Algorithm:
*
*      fi(n) = fi-1(n) - Ki(n) * bi-1(n-1)  i=1,2,...,64
*
*      bi(n) = bi-1(n-1) - Ki(n) * fi-1(n)  i=1,2,...,64
*                         i-1
*      ei(n) = d(n) - SUM yk(n) = ei-1 - bi-1(n)*Gi-1(n)  i=1,2,...,64
*                         k=0
*               64            64
*      y(n) = SUM yi(n) = SUM bi(n)*Gi(n)
*              i=1           i=1
*
*      Ki(n+1) = Ki(n) + mu * [ fi(n)*bi-1(n-1) + bi(n)*fi-1(n) ]
*
*      Gi(n+1) = Gi(n) + mu * ei(n) * bi(n)  i=1,2,..64
*
*      Where filter order = 64 and mu = 0.04.
*
*                  Chen, Chein-Chung  March, 1989
*
****************************************************************
           .copy     adapfltr.int"
****************************************************
*    PERFORM ADAPTIVE FILTER
****************************************************
order      .set      64                  ; Filter order
mu         .set      0.04                ; Step size
*
*   INITIALIZE POINTERS AND ARRAYS
*
           .text
begin      .set      $
           LDI       order*2,BK          ; Set up circular buffer
           LDP       @kn_addr            ; Set data page
           LDI       @kn_addr,AR0        ; Set pointer for k[]
           LDI       @bn_addr,AR1        ; Set pointer for b[]
           LDI       @gn_addr,AR2        ; Set pointer for g[]
           LDI       order,IR0
           LDF       0.0,R0              ; R0 = 0.0
           RPTS      order*2-1
           STF       R0,*AR0++(1)%       ; k[] = 0.0 and g[] = 0.0
     ::    STF       R0,*AR1++(1)%       ; b[] = 0.0 and bd[] = 0.0
           ADDI      AR1,IR0,AR4
           LDI       @in_addr,AR6        ; Set pointer for input ports
           LDI       @out_addr,AR7       ; Set pointer for output ports
input:
           LDF       *AR6,R7             ; Input d(n)
     ::    LDF       *+AR6(1),R5         ; Input x(n)
           MPYF3     R5,*AR2,R6          ; B1 * G1
     ::    STF       R5,*AR1             ; Insert B1
           SUBF      R6,R7               ; E = D - B1 * G1
*
           LDI       order-1,RC
           RPTB      lattice
           MPYF3     *AR0,R5,R3          ; R3 = kFi-1
           MPYF3     R7,*AR1++(1)%,R0    ; R0 = Ei-1 * Bi-1
     ::    SUBF3     R3,*AR4,R3          ; R3 = Bi = BDi-1 - kFi-1
           MPYF      @u,R0               ; R0 = u * Ei-1 * Bi-1
           ADDF3     R0,*AR2,R0          ; R0 = Gi-1 + u * Ei-1 * Bi-1
     ::    STF       R3,*AR1             ; Store Bi
           MPYF3     R5,*AR1,R1          ; R1 = Fi-1 * Bi
     ::    STF       R0,*AR2++(1)        ; Store Gi
           MPYF3     *AR0,*AR4,R0        ; R0 = kBDi-1
           SUBF      R0,R5               ; R5 = Fi
           MPYF3     R5,*AR4++(1)%,R0    ; R1 = Fi * BDi-1
           ADDF      R1,R0               ; R0 = Fi*BDi-1 + Fi-1*Bi
           MPYF      @u,R0               ; R0 = u * (Fi*BDi-1 + Fi-1*Bi)
           ADDF3     R0,*AR0,R0          ; ki = ki-1 + R0
           MPYF3     R3,*AR2,R4          ; R4 = Yi
     ::    STF       R0,*AR0++(1)        ; Store ki
           ADDF      R4,R6               ; Compute y(n)
lattice    SUBF      R4,R7               ; Compute e(n)
*
*    OUTPUT y(n) AND e(n) SIGNALS
*
           BD        input               ; Delay branch
           SUBF      R4,R6               ; Take out last term
           STF       R6,*AR7             ; Send out y(n)
     ::    STF       R7,*+AR7(1)         ; Send out e(n)
           LDI       *AR0--(IR0),R5      ; Update k[] pointer
     ::    LDI       *AR2--(IR0),R7      ; Update g[] pointer
*
*   DEFINE CONSTANTS
*
kn         .usect    "coeffs",order
gn         .usect    "coeffs",order
bn         .usect    "buffer",2*order
in_addr    .usect    "vars",1
out_addr   .usect    "vars",1
kn_addr    .usect    "vars",1
bn_addr    .usect    "vars",1
gn_addr    .usect    "vars",1
u          .usect    "vars",1
cinit      .sect     ".cinit"
           .word     6,in_addr
           .word     0804000h
           .word     0804002h
           .word     kn
           .word     bn
           .word     gn
           .float    mu
           .end
```

Appendix D1. Transversal Structure with Normalized LMS Algorithm Using the TMS320C25

```
        .title    'TM25'
*************************************************************
*
*  TM25 :  Adaptive Filter Using Transversal Structure
*          and Normalized LMS Algorithm ,Looped Code
*
*  Algorithm:
*
*            63
*     y(n) = SUM w(k)*x(n-k)  k=0,1,2,...,63
*            k=0
*
*     e(n) = d(n) - y(n)
*
*     var(k) = (1.-r) * var(K-1) + r * x(n) * x(n)
*
*     w(k) = w(k) + u*e(n)*x(n-k)/var(k)  k=0,1,2,..63
*
*     Where we use filter order = 64 and mu = 0.01.
*
*  Note:  This source program is the generic version; I/O configuration has
*         not been set up. User has to modify the main routine for specific
*         application.
*
*  Initial condition:
*         1) PM status bit should be equal to 01.
*         2) SXM status bit should be set to 1.
*         3) The current DP (data memory page pointer) should be page 0.
*         4) Data memory ONE should be 1.
*         5) Data memory U should be 327.
*         6) Data memory VAR should be initialized to 07fffh.
*
*                 Chen, Chein-Chung  February, 1989
*
*************************************************
*
*   DEFINE PARAMETERS
*
ORDER:     .equ     64
SHIFT:     .equ     7
PAGE0:     .equ     0
*
*   DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
X0:        .usect   "buffer",ORDER-1
XN:        .usect   "buffer",1
WN:        .usect   "coeffs",ORDER
*
*   RESERVE ADDRESSES FOR PARAMETERS
*
D:         .usect   "parameters",1
Y:         .usect   "parameters",1
ERR:       .usect   "parameters",1
ONE:       .usect   "parameters",1
U:         .usect   "parameters",1
ERRF:      .usect   "parameters",1
VAR:       .usect   "parameters",1
*******************************************
*   PERFORM THE ADAPTIVE FILTER
*******************************************
           .text
*
*   ESTIMATE THE POWER OF SIGNAL
*
           LARP     AR3
           LRLK     AR3,X0          ; Point to input signal X
           SQRA     *               ; Square input signal
           SPH      ERRF
           ZALH     VAR             ; ACC = VAR(n-1)
           SUB      VAR,SHIFT       ; ACC = (1-r) * VAR(n-1)
           ADD      ERRF,SHIFT      ; ACC = (1-r) * VAR(n-1) + r * X(n)
*                                   ; * X(n)
           SACH     VAR             ; Store VAR(n)
*
*   ESTIMATE THE SIGNAL Y
*
           CNFP                     ; Configure B0 as program memory
           MPYK     0               ; Clear the P register
           LAC      ONE,15          ; Using rounding
           LRLK     AR3,XN          ; Point to the oldest sample
FIR        RPTK     ORDER-1         ; Repeat N times
           MACD     WN+0fd00h,*-    ; Estimate Y(n)
           CNFD                     ; Configure B0 as data memory
           APAC
           SACH     Y               ; Store the filter output
*
*   COMPUTE THE ERROR
*
           NEG                      ; ACC = - Y(n)
           ADDH     D
           SACH     ERR             ; ERR(n) = D(n) - Y(n)
*
*   UPDATE THE WEIGHTS
*
           LT       ERR             ; T = ERR(n)
           MPY      U               ; P = U * ERR(n)
           PAC
           ADD      ONE,15          ; Round the result
*
*   NORMALIZE CONVERGE FACTOR
*
           ABS                      ; Make dividend positive
           RPTK     14              ; Repeat 15 times
           SUBC     VAR             ; Perform U * !ERR(n)! / VAR
           BIT      ERR,0           ; Check sign of ERR(n)
```

```
        BBZ     NEXT
        NEG                     ; ERRF = - U * |ERR(n)| / VAR
NEXT    SACL    ERRF            ; Store ERRF
*
        LARK    AR1,ORDER-1     ; Set up counter
        LRLK    AR2,WN          ; Point to the coefficients
        LRLK    AR3,XN+1        ; Point to the data samples
        LT      ERRF            ; T register = U * ERR(n)
        MPY     *-,AR2          ; P = U * ERR(n) * X(n-k)
ADAPT   ZALR    *,AR3           ; Load ACCH with A(k,n) & round
        MPYA    *-,AR2          ; W(k,n+1) = W(k,n) + P
*                               ; P = U * ERR(n) * X(n-k)
        SACH    *+,0,AR1        ; Store W(k,n+1)
        BANZ    ADAPT,*-,AR2
FINISH  .end
```

Appendix D2. Transversal Structure with Normalized LMS Algorithm Using the TMS320C30

```
##############################################################
*
*  TM30 - Adaptive transversal filter with Normalized LMS algorithm
*         using the TMS320C30
*
*   Algorithm:
*
*             63
*      y(n) = SUM w(k)*x(n-k) k=0,1,2,...,63
*             k=0
*
*      var(n) = r*var(n-1) + (1-r)*x(n)*x(n)
*
*      e(n) = d(n) - y(n)
*
*      w(k) = w(k) + u*e(n)*x(n-k)/var(n)  k=0,1,2,..63
*
*      Where we use filter order = 64 and mu = 0.01.
*
*                 Chen, Chein-Chung  March, 1989
*
##################################################
           .copy     "adapfltr.int"
#############################################
*    PERFORM ADAPTIVE FILTER
#############################################
order      .set      64               ; Filter order
mu         .set      0.01             ; Step size
power      .set      1.0              ; Input signal power
alpha      .set      0.996
alpha1     .set      0.004            ; 1.0 - alpha
*
*   INITIALIZE POINTERS AND ARRAYS
*
           .text
begin      .set      $
           LDI       order,BK         ; Set up circular buffer
           LDP       @xn_addr         ; Set data page
           LDI       @xn_addr,AR0     ; Set pointer for x[]
           LDI       @wn_addr,AR1     ; Set pointer for w[]
           LDF       0.0,R0           ; R0 = 0.0
           RPTS      order-1
           STF       R0,*AR0++(1)%    ; x[] = 0
        :: STF       R0,*AR1++(1)%    ; w[] = 0
           LDI       @in_addr,AR6     ; Set pointer for input ports
           LDI       @out_addr,AR7    ; Set pointer for output ports

input:
           LDF       *AR6,R7          ; Input d(n)
        :: LDF       *+AR6(1),R6      ; Input x(n)
           STF       R6,*AR0          ; Insert x(n) to buffer

*  ESTIMATE THE POWER OF THE INPUT SIGNAL
*
           MPYF      R6,R6            ; R6 = x2
           MPYF      @r_1,R6          ; R6 = (1-r) * x2
           LDF       @r,R3
           MPYF      @var,R3          ; R3 = r * var(n-1)
*
*  COMPUTE FILTER OUTPUT y(n)
*
           LDF       0.0,R2           ; R2 = 0.0
*
           MPYF3     *AR0++(1)%,*AR1++(1)%,R1
        :: ADDF      R6,R3
           STF       R3,@var          ; Restore var(n)
           RPTS      order-2
*
           MPYF3     *AR0++(1)%,*AR1++(1)%,R1
        :: ADDF3     R1,R2,R2         ; y(n) = w[].x[]
           ADDF      R1,R2            ; Include last result
*
*  COMPUTE ERROR SIGNAL e(n)
*
           SUBF      R2,R7            ; e(n) = d(n) - y(n)
*
*  OUTPUT y(n) AND e(n) SIGNALS
*
           STF       R2,*AR7          ; Send out y(n)
        :: STF       R7,*+AR7(1)      ; Send out e(n)
*
*  UPDATE WEIGHTS w(n)
*
           PUSHF     R3               ; Compute  1/var(n)
           POP       R2               ; var(n) = a * 2e
           ASH       -24,R2
           NEGI      R2
           SUBI      1,R2             ; Now we have 2-e-1
           ASH       24,R2
           PUSH      R2
           POPF      R2               ; Now R2 = x[0] = 1.0 * 2-e-1.
*
           MPYF      R2,R3,R0         ; R0 = v * x[0]
           SUBRF     2.0,R0           ; R0 = 2.0 - v * x[0]
           MPYF      R0,R2            ; R2 = x[1] = x[0] * (2.0 - v * x[0])
*
           MPYF      R2,R3,R0         ; R0 = v * x[1]
           SUBRF     2.0,R0           ; R0 = 2.0 - v * x[1]
           MPYF      R0,R2            ; R2 = x[2] = x[1] * (2.0 - v * x[1])
*
           MPYF      R2,R3,R0         ; R0 = v * x[2]
           SUBRF     2.0,R0           ; R0 = 2.0 - v * x[2]
           MPYF      R0,R2            ; R2 = x[3] = x[2] * (2.0 - v * x[2])
*
           MPYF      R2,R3,R0         ; R0 = v * x[3]
```

```
        SUBRF   2.0,R0              ; R0 = 2.0 - v * x[3]
        MPYF    R0,R2               ; R2 = x[4] = x[3] * (2.0 - v * x[3])
        RND     R2                  ; This minimizes error in the LSBs.
*
        MPYF    R2,R3,R0            ; R0 = v * x[4] = 1.0..01.. => 1
        SUBRF   1.0,R0              ; R0 = 1.0 - v * x[4] =
*                                   ; 0.0..01... => 0
        MPYF    R2,R0               ; R0 = x[4] * (1.0 - v * x[4])
        ADDF    R0,R2               ; x[5] = (x[4]*(1.0-(v*x[4])))+x[4]
        RND     R2,R0               ; Round since this is followed
*                                   ; by a MPYF.
*
        MPYF    @u,R7               ; R7 = e(n) * u
        MPYF    R0,R7               ; R7 = e(n) * u / var(n)
        MPYF3   *AR0++(1)%,R7,R1    ; R1 = e(n) * u * x(n) / var(n)
        LDI     order-3,RC          ; Initialize repeat counter
        RPTB    LMS                 ; Do i = 0, N-3
        MPYF3   *AR0++(1)%,R7,R1    ; R1 = e(n) * u * x(n-i-1) / var(n)
     :: ADDF3   *AR1,R1,R2          ; R2 = wi(n) + R1
LMS     STF     R2,*AR1++(1)%       ; Store wi(n+1)
        MPYF3   *AR0,R7,R1          ; For i = N - 2
     :: ADDF3   *AR1,R1,R2
        BD      input               ; Delay branch
        STF     R2,*AR1++(1)%       ; Store wi(n+1)
        ADDF3   *AR1,R1,R2
        STF     R2,*AR1++(1)%       ; Update last w
*
*   DEFINE CONSTANTS AND VARIABLES
*
xn      .usect  "buffer",order
wn      .usect  "coeffs",order
in_addr .usect  "vars",1
out_addr .usect "vars",1
xn_addr .usect  "vars",1
wn_addr .usect  "vars",1
u       .usect  "vars",1
var     .usect  "vars",1
r       .usect  "vars",1
r_1     .usect  "vars",1
cinit   .sect   ".cinit"
        .word   8,in_addr
        .word   0804000h
        .word   0804002h
        .word   xn
        .word   wn
        .float  mu
        .float  power
        .float  alpha
        .float  alpha1
        .end
```

Appendix E1. Transversal Structure with Sign-Error LMS Algorithm Using the TMS320C25

```
        .title   'TSE25'
*************************************************************
*
*   TSE25 :  Adaptive Filter Using Transversal Structure
*            and Sign-Error LMS Algorithm ,Looped Code
*
*   Algorithm:
*
*               63
*     y(n) = SUM w(k)*x(n-k)  k=0,1,2,...,63
*              k=0
*
*     e(n) = d(n) - y(n)
*
*     For k = 0,1,2,...,63
*           w(k) = w(k) + u*x(n-k) if e(n) >= 0
*           w(k) = w(k) - u*x(n-k) if e(n) <  0
*
*     Where we use filter order = 64 and mu = 0.01.
*
*   Note: This source program is the generic version; I/O configuration has
*         not been set up. User has to modify the main routine for specific
*         application.
*
*    Initial condition:
*         1) PM status bit should be equal to 01.
*         2) SXM status bit should be set to 1.
*         3) The current DP (data memory page pointer) should be page 0.
*         4) Data memory ONE should be 1.
*         5) Data memory U should be 327.
*         6) Data memory NEGMU should be -327.
*
*                 Chen, Chein-Chung  February, 1989
*
*************************************************
*
*   DEFINE PARAMETERS
*
ORDER:      .equu     64
PAGE0:      .equ      0
*
*   DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
X0:         .usect    "buffer",ORDER-1
XN:         .usect    "buffer",1
WN:         .usect    "coeffs",ORDER
*
*   RESERVE ADDRESSES FOR PARAMETERS
*
D:          .usect    "parameters",1
Y:          .usect    "parameters",1
ERR:        .usect    "parameters",1
ONE:        .usect    "parameters",1
U:          .usect    "parameters",1
ERRF:       .usect    "parameters",1
NEGMU:      .usect    "parameters",1
*******************************
*   PERFORM THE ADAPTIVE FILTER
*******************************
            .text
*
*   ESTIMATE THE SIGNAL Y
*
            LARP     AR3
            CNFP                        ; Configure B0 as program memory
            MPYK     0                  ; Clear the P register
            LAC      ONE,15             ; Using rounding
            LRLK     AR3,XN             ; Point to the oldest sample
FIR         RPTK     ORDER-1            ; Repeat N times
            MACD     WN+0fd00h,*-       ; Estimate Y(n)
            CNFD                        ; Configure B0 as data memory
            APAC
            SACH     Y                  ; Store the filter output
*
*   CHECK THE SIGN OF ERROR
*
            LT       U                  ; T register = U
            NEG                         ; ACC = - Y(n)
            ADDH     D                  ; ACC = D(n) - Y(n)
            BGEZ     NEXT
            LT       NEGMU              ; T register = -U
*
*   UPDATE THE WEIGHTS
*
NEXT        LARK     AR1,ORDER-1        ; Set up counter
            LRLK     AR2,WN             ; Point to the coefficients
            LRLK     AR3,XN+1           ; Point to the data sample
            MPY      *-,AR2             ; P = U * X(n-k)
ADAPT       ZALR     *,AR3              ; Load ACCH with W(k,n) & round
            MPYA     *-,AR2             ; W(k,n+1) = W(k,n) + P
*                                       ; P = U * X(n-k)
            SACH     *+,0,AR1           ; Store W(k,n+1)
            BANZ     ADAPT,*-,AR2
*
FINISH      .end
```

Appendix E2. Transversal Structure with Sign-Error LMS Algorithm Using the TMS320C30

```
**********************************************************
*
*  TSE30 - Adaptive transversal filter with Sign-Error LMS
*    algorithm using the TMS320C30
*
*   Algorithm:
*
*              63
*      y(n) = SUM w(k)*x(n-k) k=0,1,2,...,63
*             k=0
*
*      e(n) = d(n) - y(n)
*
*      for k=0,1,2,..63
*      w(k) = w(k) + u*x(n-k) if e(n) >= 0.0
*      w(k) = w(k) - u*x(n-k) if e(n) <  0.0
*
*      Where we use filter order = 64 and mu = 0.01.
*
*                 Chen, Chein-Chung  March, 1989
*
*********************************************
           .copy     "adapfltr.int"
*****************************************
*    PERFORM ADAPTIVE FILTER
*****************************************
order      .set      64
mu         .set      0.01
*
*   INITIALIZE POINTERS AND ARRAYS
*
           .text
begin      .set      $
           LDI       order,BK          ; Set up circular buffer
           LDP       @xn_addr          ; Set data page
           LDI       @xn_addr,AR0      ; Set pointer for x[]
           LDI       @wn_addr,AR1      ; Set pointer for w[]
           LDF       0.0,R0            ; R0 = 0.0
           RPTS      order-1
           STF       R0,*AR0++(1)%     ; x[] = 0
       ||  STF       R0,*AR1++(1)%     ; w[] = 0
           LDI       @in_addr,AR6      ; Set pointer for input ports
           LDI       @out_addr,AR7     ; Set pointer for output ports
           LDF       @u,R4             ; R4 = mu
           LDF       @u,R5             ; R5 = mu
*
input:
           LDF       *AR6,R7           ; Input d(n)
       ||  LDF       *+AR6(1),R6       ; Input x(n)
           STF       R6,*AR0           ; Insert x(n) to buffer
*
*   COMPUTE FILTER OUTPUT y(n)
*
           LDF       0.0,R2 II         ; R2 = 0.0
*
           MPYF3     *AR0++(1)%,*AR1++(1)%,R1
           RPTS      order-2
           MPYF3     *AR0++(1)%,*AR1++(1)%,R1
*
       ||  ADDF3     R1,R2,R2IIII      ; y(n) = w[].x[]
           ADDF      R1,R2             ; Include last result
*
*   COMPUTE ERROR SIGNAL e(n)
*
           SUBF      R2,R7             ; e(n) = d(n) - y(n)
*
*   OUTPUT y(n) AND e(n) SIGNALS
*
           STF       R2,*AR7           ; Send out y(n)
       ||  STF       R7,*+AR7(1)       ; Send out e(n)
*
*   UPDATE WEIGHTS w(n)
*
           ASH       -31,R7            ; Get Sign[e(n)]
           XOR3      R4,R7,R5          ; R5 = S[e(n)] * u
           MPYF3     *AR0++(1)%,R5,R1  ; R1 = S[e(n)] * u * x(n)
           LDI       order-3,RC        ; Initialize repeat counter
           RPTB      SELMS             ; Do i = 0, N-3
           MPYF3     *AR0++(1)%,R5,R1  ; R1 = S[e(n)] * u * x(n-i-1)
       ||  ADDF3     *AR1,R1,R2        ; R2 = wi(n) + S[e(n)] * u * x(n-i)
SELMS      STF       R2,*AR1++(1)%     ; wi(n+1) = wi(n) + S[e(n)]*u*x(n-i)
           MPYF3     *AR0,R5,R1        ; For i = N - 2
       ||  ADDF3     *AR1,R1,R2
           BD        input             ; Delay branch
           STF       R2,*AR1++(1)%     ; wi(n+1) = wi(n) + S[e(n)]*u*x(n-i)
           ADDF3     *AR1,R1,R2
           STF       R2,*AR1++(1)%     ; Update last w
*
*   DEFINE CONSTANTS
*
xn         .usect    "buffer",order
wn         .usect    "coeffs",order
in_addr    .usect    "vars",1
out_addr   .usect    "vars",1
xn_addr    .usect    "vars",1
wn_addr    .usect    "vars",1
u          .usect    "vars",1
cinit      .sect     ".cinit"
           .word     5,in_addr
           .word     0804000h
           .word     0804002h
           .word     xn
           .word     wn
           .float    mu
           .end
```

Appendix F1. Transversal Structure with Sign-Sign LMS Algorithm Using the TMS320C25

```
        .title    'TSS25'
*******************************************************
*
*   TSS :  Adaptive Filter Using Transversal Structure
*          and Sign-Sign LMS Algorithm ,Looped Code
*
*   Algorithm:
*
*                63
*     y(n) = SUM w(k)*x(n-k)  k=0,1,2,...,63
*               k=0
*
*     e(n) = d(n) - y(n)
*
*     For k = 0,1,2,...,63
*           w(k) = w(k) + u if e(n)*x(n-k) >= 0
*           w(k) = w(k) - u if e(n)*x(n-k) <  0
*
*     Where we use filter order = 64 and mu = 0.01.
*
*   Note: This source program is the generic version; I/O configuration has
*         not been set up. User has to modify the main routine for specific
*         application.
*
*   Initial condition:
*         1) PM status bit should be equal to 01.
*         2) SXM status bit should be set to 1.
*         3) The current DP (data memory page pointer) should be page 0.
*         4) Data memory ONE should be 1.
*         5) Data memory U should be 327.
*
*                  Chen, Chein-Chung  February, 1989
*
*****************************************
*
*   DEFINE PARAMETERS
*
ORDER:      .equ      64
PAGE0:      .equ      0
*
*   DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
X0:         .usect    "buffer",ORDER-1
XN:         .usect    "buffer",1
WN:         .usect    "coeffs",ORDER
*
*   RESERVE ADDRESSES FOR PARAMETERS
*
D:          .usect    "parameters",1
Y:          .usect    "parameters",1
ERR:        .usect    "parameters",1
ONE:        .usect    "parameters",1
U:          .usect    "parameters",1
ERRF:       .usect    "parameters",1
**************************************
*   PERFORM THE ADAPTIVE FILTER
**************************************
            .text
*
*   ESTIMATE THE SIGNAL Y
*
            LARP      AR3
            CNFP                      ; Configure B0 as program memory
            MPYK      0               ; Clear the P register
            LAC       ONE,15          ; Using rounding
            LRLK      AR3,XN          ; Point to the oldest sample
FIR         RPTK      ORDER-1         ; Repeat N times
            MACD      WN+0fd00h,*-    ; Estimate Y(n)
            CNFD                      ; Configure B0 as data memory
            APAC
            SACH      Y               ; Store the filter output
*
*   SET UP THE POINTERS
*
            LARK      AR1,ORDER-1     ; Set up counter
            LRLK      AR2,WN          ; Point to the coefficients
            LRLK      AR3,XN+1        ; Point to the data sample
*
*   CHECK THE SIGN OF ERROR
*
            NEG
            ADDH      D               ; ACC = D(n) - Y(n)
            SACH      ERR
*
*   UPDATE THE WEIGHTS
*
ADAPT       LAC       *-,0,AR2        ; ACC = X(n-k)
            XOR       ERR             ; Get the sign of ERR(n) * X(n-k)
            SACL      ERRF            ; Store the sign
            LAC       ERRF            ; Get the sign with its sign extension
            XORK      MU,15           ; Get the convergent factor MU or -MU
            ADD       *,15            ; Update W(k)
            SACH      *+,1,AR1
            BANZ      ADAPT,*-,AR3
*
FINISH      .end
```

Appendix F2. Transversal Structure with Sign-Sign LMS Algorithm Using the TMS320C30

```
***********************************************************************
*
*   TSS30 - Adaptive transversal filter with Sign-Sign LMS
*           algorithm using the TMS320C30
*
*   Algorithm:
*
*               63
*       y(n) = SUM w(k)*x(n-k) k=0,1,2,...,63
*              k=0
*
*       e(n) = d(n) - y(n)
*
*       for k=0,1,2,...,63
*           w(k) = w(k) + u, if x(n-k)*e(n) >= 0.0
*           w(k) = w(k) - u, if x(n-k)*e(n) <  0.0
*
*       Where we use filter order = 64 and mu = 0.01.
*
*                   Chen, Chein-Chung  March, 1989
*
***********************************************************************
            .copy   "adapfltr.int"
order       .set    64
mu          .set    0.01
*
*   INITIALIZE POINTERS AND ARRAYS
*
            .text
begin       .set
            LDI     order,BK            ; Set up circular buffer
            LDP     @xn_addr            ; Set data page
            LDI     @xn_addr,AR0        ; Set pointer for x[]
            LDI     @wn_addr,AR1        ; Set pointer for w[]
            LDF     0.0,R0              ; R0 = 0.0
            RPTS    order-1
            STF     R0,*AR0++(1)%       ; x[] = 0
         || STF     R0,*AR1++(1)%       ; w[] = 0
            LDF     @u,R0               ; R0 = mu
            LDF     @u,R4               ; R4 = mu
            LDF     @u,R5               ; R5 = mu
            LDI     @in_addr,AR6        ; Set pointer for input ports
            LDI     @out_addr,AR7       ; Set pointer for output ports
input:
            LDF     *AR6,R7             ; Input d(n)
         || LDF     *+AR6(1),R6         ; Input x(n)
            STF     R6,*AR0             ; Insert x(n) to buffer
*
*   COMPUTE FILTER OUTPUT y(n)
*
            LDF     0.0,R2              ; R2 = 0.0
*
            MPYF3   *AR0++(1)%,*AR1++(1)%,R1
            RPTS    order-2
            MPYF3   *AR0++(1)%,*AR1++(1)%,R1
         || ADDF3   R1,R2,R2            ; y(n) = w[].x[]
            ADDF    R1,R2               ; Include last result
*
*   COMPUTE ERROR SIGNAL e(n) AND OUTPUT y(n) AND e(n) SIGNALS
*
            SUBF    R2,R7               ; e(n) = d(n) - y(n)
            STF     R2,*AR7             ; Send out y(n)
         || STF     R7,*+AR7(1)         ; Send out e(n)
*
*   UPDATE WEIGHTS w(n)
*
            ASH     -31,R7              ; R7 = Sign[e(n)]
            XOR3    R0,R7,R5            ; R5 = Sign[e(n)] * u
            LDF     *AR0++(1)%,R6       ; R6 = x(n)
            ASH     -31,R6              ; R6 = Sign[x(n-i)]
            XOR3    R5,R6,R4            ; R4 = Sign[x(n-i)]*Sign[e(n)] * u
            ADDF3   *AR1,R4,R3          ; R3 = wi(n) + R4

            LDI     order-3,RC          ; Initialize repeat counter
            RPTB    SSLMS               ; Do i = 0, N-3
            LDF     *AR0++(1)%,R6       ; Get next data
         || STF     R3,*AR1++(1)%       ; Update wi(n+1)
            ASH     -31,R6              ; Get the sign of data
            XOR3    R5,R6,R4            ; Decide the sign of u
SSLMS       ADDF3   *AR1,R4,R3          ; R3 = wi(n) + R4

            LDF     *AR0,R6             ; Get last data
         || STF     R3,*AR1++(1)%       ; Update wN-2(n+1)
            ASH     -31,R6              ; Get the sign of data
            BD      input               ; Delay branch
            XOR3    R5,R6,R4            ; Decide the sign of u
            ADDF3   *AR1,R4,R3          ; Compute wN-1(n+1)
            STF     R3,*AR1++(1)%       ; Store last w(n+1)
*
*   DEFINE CONSTANTS
*
xn          .usect  "buffer",order
wn          .usect  "coeffs",order
in_addr     .usect  "vars",1
out_addr    .usect  "vars",1
xn_addr     .usect  "vars",1
wn_addr     .usect  "vars",1
u           .usect  "vars",1
cinit       .sect   ".cinit"
            .word   5,in_addr
            .word   0804000h
            .word   0804002h
            .word   xn
            .word   wn
            .float  mu
.end
```

Appendix G1. Transversal Structure with Leaky LMS Algorithm Using the TMS320C25

```
        .title    'TL25'
**********************************************************************
*
*   TL25 :  Adaptive Filter Using Transversal Structure
*           and Leaky-LMS Algorithm, Looped Code
*
*   Algorithm:
*
*           63
*     y(n) = SUM w(k)*x(n-k)  k=0,1,2,...,63
*           k=0
*
*     e(n) = d(n) - y(n)
*
*     w(k) = v*w(k) + u*e(n)*x(n-k) k=0,1,2,..63
*
*     Where we use filter order = 64 and mu = 0.01.
*
*   Note: This source program is the generic version; I/O configuration has
*         not been set up. User has to modify the main routine for specific
*         application.
*
*   Initial condition:
*         1) PM status bit should be equal to 01.
*         2) SXM status bit should be set to 1.
*         3) The current DP (data memory page pointer) should be page 0.
*         4) Data memory ONE should be 1.
*         5) Data memory U should be 327.
*
*                 Chen, Chein-Chung  February, 1989
*
*****************************************************************
*
*   DEFINE PARAMETERS
*
ORDER:      .equu     64
LEAKY:      .equ      7
PAGE0:      .equ      0
*
*   DEFINE ADDRESSES OF BUFFER AND COEFFICIENTS
*
X0:         .usect    "buffer",ORDER-1
XN:         .usect    "buffer",1
WN:         .usect    "coeffs",ORDER
*
*   RESERVE ADDRESSES FOR PARAMETERS
*
D:          .usect    "parameters",1
Y:          .usect    "parameters",1
ERR:        .usect    "parameters",1
ONE:        .usect    "parameters",1
U:          .usect    "parameters",1
ERRF:       .usect    "parameters",1
**************************************************
*   PERFORM THE ADAPTIVE FILTER
**************************************************
            .text
*
*   ESTIMATE THE SIGNAL Y
*
            LARP      AR3
            CNFP                          ; Configure B0 as program memory
            MPYK      0                   ; Clear the P register
            LAC       ONE,15              ; Using rounding
            LRLK      AR3,XN              ; Point to the oldest sample
FIR         RPTK      ORDER-1             ; Repeat N times
            MACD      WN+0fd00h,*-        ; Estimate Y(n)
            CNFD                          ; Configure B0 as data memory
            APAC
            SACH      Y                   ; Store the filter output
*
*   COMPUTE THE ERROR
*
            NEG                           ; ACC = - Y(n)
            ADDH      D
            SACH      ERR                 ; ERR(n) = D(n) - Y(n)
*
*   UPDATE THE WEIGHTS
*
            LT        ERR                 ; T = ERR(n)
            MPY       U                   ; P = U * ERR(n)
            PAC
            ADD       ONE,15              ; Round the result
            SACH      ERRF                ; ERRF = U * ERR(n)
*
            LARK      AR1,ORDER-1         ; Set up counter
            LRLK      AR2,WN              ; Point to the coefficients
            LRLK      AR3,XN+1            ; Point to the data sample
            LT        ERRF                ; T register = U * ERR(n)
            MPY       *-,AR2              ; P = U * ERR(n) * X(n-k)
ADAPT       ZALR      *,AR3               ; Load ACCH with A(k,n) & round
            MPYA      *-,AR2              ; W(k,n+1) = W(k,n) + P
*                                         ; P = U * ERR(n) * X(n-k)
            SUB       *,LEAKY             ; ACC = R * W(k,n) + P
            SACH      *+,0,AR1            ; Store W(k,n+1)
            BANZ      ADAPT,*-,AR2
*
FINISH      .end
```

Appendix G2. Transversal Structure with Leaky LMS Algorithm Using the TMS320C30

```
**********************************************************
*   TL30 - Adaptive transversal filter with Leaky LMS algorithm
*          using the TMS320C30
*
*   Algorithm:
*
*             63
*      y(n) = SUM w(k)*x(n-k) k=0,1,2,...,63
*             k=0
*
*      e(n) = d(n) - y(n)
*
*      w(k) = r*w(k) + u*e(n)*x(n-k) k=0,1,2,...,63
*
*      Where we use filter order = 64, r = 0.995 and mu = 0.01.
*
*                  Chen, Chein-Chung  March, 1989
*
**************************************************
           .copy    "adapfltr.int"
**********************************************
*    PERFORM ADAPTIVE FILTER
**********************************************
order      .set     64
mu_leaky   .set     0.01005          ; mu / leaky
leaky      .set     0.995
*
*   INITIALIZE POINTERS AND ARRAYS
*
           .text
begin      .set     $
           LDI      order,BK         ; Set up circular buffer
           LDP      @xn_addr         ; Set data page
           LDI      @xn_addr,AR0     ; Set pointer for x[]
           LDI      @wn_addr,AR1     ; Set pointer for w[]
           LDI      @r_addr,AR2      ; Set pointer for r
           LDF      0.0,R0           ; R0 = 0.0
           RPTS     order-1
           STF      R0,*AR0++(1)%    ; x[] = 0
      ||   STF      R0,*AR1++(1)%    ; w[] = 0
           LDI      @in_addr,AR6     ; Set pointer for input ports
           LDI      @out_addr,AR7    ; Set pointer for output ports
input:
           LDF      *AR6,R7          ; Input d(n)
      ||   LDF      *+AR6(1),R6      ; Input x(n)
           STF      R6,*AR0          ; Insert x(n) to buffer
*
*   COMPUTE FILTER OUTPUT y(n)
*
           LDF      0.0,R2           ; R2 = 0.0
*
           MPYF3    *AR0++(1)%,*AR1++(1)%,R1
           RPTS     order-2
           MPYF3    *AR0++(1)%,*AR1++(1)%,R1
      ||   ADDF3    R1,R2,R2         ; y(n) = w[].x[]
           ADDF     R1,R2            ; include last result
*
*   COMPUTE ERROR SIGNAL e(n) AND OUTPUT y(n) AND e(n) SIGNALS
*
           SUBF     R2,R7            ; e(n) = d(n) - y(n)
           STF      R2,*AR7          ; Send out y(n)
      ||   STF      R7,*+AR7(1)      ; Send out e(n)
*
*   UPDATE WEIGHTS w(n)
*
           MPYF     @u_r,R7          ; R7 = e(n)*u/r
           MPYF3    *AR0++(1)%,R7,R1 ; R1 = e(n)*u*x(n)/r
           MPYF3    *AR0++(1)%,R7,R1 ; R1 = e(n)*u*x(n-1)/r
      ||   ADDF3    *AR1,R1,R2       ; R2 = w0(n) + e(n)*u*x(n)/r
           LDI      order-4,RC       ; Initialize repeat counter
           RPTB     LLMS             ; Do i = 0, N-4
           MPYF3    *AR2,R2,R0       ; R0 = r*wi(n) + e(n)*u*x(n-i)
      ||   ADDF3    *+AR1(1),R1,R2   ; R2 = wi+1(n) + e(n)*u*x(n-i-1)/r
LLMS       MPYF3    *AR0++(1)%,R7,R1 ; R1 = e(n)*u*x(n-i-2)/r
      ||   STF      R0,*AR1++(1)%    ; store wi(n+1)
           MPYF3    *AR2,R2,R0       ; R0 = r*wN-3(n) + e(n)*u*x(n-N+3)
      ||   ADDF3    *+AR1(1),R1,R2   ; R2 = wN-2(n) + e(n)*u*x(n-N+2)/r
           MPYF3    *AR0,R7,R1       ; R1 = e(n)*u*x(n-N+1)/r
      ||   STF      R0,*AR1++(1)%    ; Store wN-3(n+1)
           BD       input            ; Delay branch
           MPYF3    *AR2,R2,R0       ; R0 = r*wi(n) + e(n)*u*x(n-N+2)
      ||   ADDF3    *+AR1(1),R1,R2   ; R2 = wN-1(n) + e(n)*u*x(n-N+1)/r
           MPYF3    *AR2,R2,R0       ; R0 = r*wi(n) + e(n)*u*x(n-N+1)
      ||   STF      R0,*AR1++(1)%    ; Store wN-2(n+1)
           STF      R0,*AR1++(1)%    ; Update last w
*
*   DEFINE CONSTANTS
*
xn         .usect   "buffer",order
wn         .usect   "coeffs",order
in_addr    .usect   "vars",1
out_addr   .usect   "vars",1
xn_addr    .usect   "vars",1
wn_addr    .usect   "vars",1
u_r        .usect   "vars",1
r          .usect   "vars",1
r_addr     .usect   "vars",1
cinit      .sect    ".cinit"
           .word    7,in_addr
           .word    0804000h
           .word    0804002h
           .word    xn
           .word    wn
           .float   mu_leaky
           .float   leaky
           .word    r
           .end
```

Appendix H1. Assembly Subroutine of Transversal Structure with LMS Algorithm Using the TMS320C25

```
        .title   'BLMS'
*****************************************************************
*
*   BLMS :  Adaptive Filter subroutine using Transversal Structure
*           and LMS Algorithm, Looped Code
*
*   Algorithm:
*
*
*              N-1
*      y(n) = SUM w(k)*x(n-k) k=0,1,2,...,N-1
*              k=0
*
*      e(n) = d(n) - y(n)-
*
*      w(k) = w(k) + u*e(n)*x(n-k) k=0,1,2,...,N-1
*
*      Where we use filter order = N
*
*   Note: This subroutine performs Adaptive Filter using the LMS Algorithm.
*         There are some initial conditions to meet before calling it.
*
*   Initial conditions:
*         1) Data memory ONE should be equal to 1.
*         2) Data memory U should be equal to MU (Q15 format).
*         3) PM status bit should be equal to 01.
*         4) SXM status bit should be set to logic 1.
*         5) OVM status bit should be set to 1.
*         6) The current DP (data memory page pointer) should be page 0.
*
*    p.s. 1) The return current auxiliary register will be AR2.
*         2) AR1  AR3 have been used in this subroutine.
*
*                 Chen, Chein-Chung  February, 1989
*
**********************************************
*
*   DEFINE AND REFER SYMBOLS
*
        .global   LMS,ORDER,U,D,ONE,Y,ERR,XN,WN
*
*   RESERVE ADDRESS FOR PARAMETER
*
SAVE1:      .usect    "parameters",1
SAVE2:      .usect    "parameters",1
SAVE3:      .usect    "parameters",1
ERRF:       .usect    "parameters",1
******************************
*   PERFORM THE ADAPTIVE FILTER
******************************
*
*   ESTIMATE THE SIGNAL Y

            .text
LMS         LARP    AR3             ; Set current register
            SAR     AR1,SAVE1       ; Save register AR1
            SAR     AR2,SAVE2       ; Save register AR2
            SAR     AR3,SAVE3       ; Save register AR3
            CNFP                    ; Configure B0 as program memory
            MPYK    0               ; Clear the P register
            LAC     ONE,15          ; Using rounding
            LRLK    AR3,XN          ; Point to the oldest sample
FIR         RPTK    ORDER-1         ; Repeat N times
            MACD    WN+0fd00h,*-    ; Estimate Y(n)
            CNFD                    ; Configure B0 as data memory
            APAC
            SACH    Y               ; Store the filter output
*
*   COMPUTE THE ERROR
*
            NEG                     ; ACC = - Y(n)
            ADDH    D
            SACH    ERR             ; ERR(n) = D(n) - Y(n)
*
*   UPDATE THE WEIGHTS
*
            LT      ERR             ; T = ERR(n)
            MPY     U               ; P = U * ERR(n)
            PAC
            ADD     ONE,15          ; round the result
            SACH    ERRF            ; ERRF = U * ERR(n)
*
            LARK    AR1,ORDER-1     ; Set up counter
            LRLK    AR2,WN          ; Point to the coefficients
            LRLK    AR3,XN+1        ; Point to the data sample
            LT      ERRF            ; T register = U * ERR(n)
            MPY     *-,AR2          ; P = U * ERR(n) * X(n-k)
ADAPT       ZALR    *,AR3           ; Load ACCH with A(k,n) & round
            MPYA    *-,AR2          ; W(k,n+1) = W(k,n) + P
*                                   ; P = U * ERR(n) * X(n-k)
            SACH    *+,0,AR1        ; Store W(k,n+1)
            BANZ    ADAPT,*-,AR2
*
            LAR     AR1,SAVE1       ; Restore register AR1
            LAR     AR2,SAVE2       ; Restore register AR2
            LAR     AR3,SAVE3       ; Restore register AR3
*
FINISH RET
*
            .end
```

Appendix H2. Linker Command File for Assembly Main Program Calling a TMS320C25 Adaptive LMS Transversal Filter Subroutine

```
/***************************************************************************/
/* ALINK.CMD -  COMMAND FILE FOR LINKING A TMS320C25 ASSEMBLY PROGRAM      */
/*              Copyright 1988, 1989 Texas Instruments Incorporated       */
/*                                                                         */
/*      Usage: dsplnk <obj files...> -o <out file> -m <map file> c.cmd    */
/*                                                                         */
/* Description: This file is a sample command file that can be used      */
/*              for linking the TMS320C25 assembly programs; use it as a  */
/*              guideline. You may want to change the allocation          */
/*              scheme according to the size of the program and the      */
/*              memory configuration of your TMS320C25.                   */
/*                                                                         */
/* Notes:       MEMORY SPECIFICATION                                       */
/*                                                                         */
/*              Block B0 is configured as data memory (CNFD) and          */
/*              MP/MC- = 1 (microprocessor mode). Data memory locations   */
/*              6h--5Fh and 80h--1FFh are not configured.                 */
/***************************************************************************/

MEMORY
{
   PAGE 0 : Ints      : origin =   0h, length =   020h        /* Program */
            Ext_Prog  : origin = 020h, length = 0FEE0h

   PAGE 1 : Regs      : origin =   0h, length =     06h      /* Data    */
            Block_B2  : origin =  060h, length =   020h
            Int_RAM   : origin = 0200h, length =  0100h      /* B0      */
            Int_RAM1  : origin = 0300h, length =  0100h      /* B1      */
            Ext_Data  : origin = 0400h, length = 0FC00h
}

/*-----------------------------------------------------------------------*/
/* SECTIONS ALLOCATION                                                   */
/*-----------------------------------------------------------------------*/
SECTIONS
{
    vectors   : { } > Ints      PAGE 0   /* Interrupt vector table      */
    .text     : { } > Ext_Prog  PAGE 0   /* Code                        */
    parameters: { } > Block_B2  PAGE 1   /* Parameters                  */
    coeffs    : { } > Int_RAM   PAGE 1   /* Block B0                    */
    buffer    : { } > Int_RAM1  PAGE 1   /* Block B1                    */
    .bss      : { } > Ext_Data  PAGE 1   /* Global VARS, STACK, HEAP    */
}
```

Appendix H3. TMS320C30 Adaptive Filter Initialization Program

```
          .width    132
*****************************************************************
*
*   This is the initial boot routine for TMS320C30 adaptive
*        filter Programs.
*
*   This module performs the following actions:
*        1) Allocates and initializes the system stack.
*        2) Performs auto-initialization, which copies section
*           ".const" data from ROM to DATA RAM.
*        3) Prepare to start the user's assembly program.
*
*****************************************************************
STACK_SIZE  .set      40h                 ; Size of system stack
FP          .set      AR3                 ; Frame pointer
*
            .sect     "vectors"
RESET       .word     adap_init
*
* ALLOCATE SPACE FOR THE SYSTEM STACK. INITIALIZE THE FIRST WORDS IN
* .text TO POINT TO THE STACK AND INITIALIZATION TABLES.
*
stack       .usect    ".stack",STACK_SIZE
            .text
*
stack_addr  .word     stack               ; Address of stack
init_addr   .word     cinit               ; Address of init tables
*****************************************************************
* ADAPTIVE FILTER INITIALIZATION ENTRY POINT FUNCTION
*****************************************************************
adap_init:
*
* SET UP THE INITIAL STACK POINTER
*
            LDP       stack_addr          ; Get page of stored address
            LDI       @stack_addr,SP      ; Load the address into SP
            LDI       SP,FP               ; And into FP too
*
* DO AUTOINITIALIZATION
*
            LDP       init_addr           ; Get page of stored address
            LDI       @init_addr,AR0      ; Get address of init tables
            CMPI      -1,AR0              ; If RAM model, skip init
            BEQ       done
            LDI       *AR0++,R1           ; Get first count
            BZD       done                ; If 0, nothing to do
            LDI       *AR0++,AR1          ; Get dest address
            LDI       *AR0++,R0           ; Get first word
            SUBI      1,R1                ; Count - 1
*
do_init:
            RPTS      R1                  ; Block copy
            STI       R0,*AR1++
         || LDI       *AR0++,R0
            LDI       R0,R1               ; Move next count into R1
            BNZD      do_init             ; If there is more, repeat
            LDI       *AR0++,AR           ; Get next dest address
            LDI       *AR0++,R0           ; Get next first word
            SUBI      1,R1                ; Count - 1
*
done:
 BR         begin
            .end
```

Appendix H4. Assembly Subroutine of Transversal Structure with LMS Algorithm Using the TMS320C30

```
****************************************************************
*
*  BT30 - TMS320C30 adaptive transversal filter with
*     LMS algorithm assembly subroutine.
*
*   Algorithm:
*
*             N-1
*      y(n) = SUM w(k)*x(n-k) k=0,1,2,...,N-1
*             k=0
*
*      e(n) = d(n) - y(n)
*
*      w(k) = w(k) + u*e(n)*x(n-k) k=0,1,2,...,N-1
*
*      Where we use filter order = N and mu = 0.01.
*
*   Initial condition:
*
*       1) AR0 and AR1 should point to x[0] and w[0].
*       2) Data memory u should contain step size.
*       3) Data memory order should contain N-2, where N is filter order.
*       4) Data memories d, y, and e should be defined in caller routine.
*
*                  Chen, Chein-Chung  March, 1989
*
****************************************************
          .global   LMS30,u,d,y,e,order
****************************************************
*    PERFORM ADAPTIVE FILTER
****************************************************
          .text
LMS30     .set      $
          PUSH      R1
          PUSHF     R1
          PUSHF     R2
          PUSH      R3
          PUSHF     R3
*
*   COMPUTE FILTER OUTPUT y(n)
*
          LDF       0.0,R3             ; R3 = 0.0
*
          MPYF3     *AR0++(1)%,*AR1++(1)%,R1
          RPTS      @order
          MPYF3     *AR0++(1)%,*AR1++(1)%,R1
       || ADDF3     R1,R3,R3           ; y(n) = w[].x[]
          ADDF      R1,R3              ; Include last result
*
*   COMPUTE ERROR SIGNAL e(n) AND STORE y(n) AND e(n)
*
          STF       R3,@y              ; Store y(n)
          SUBRF     @d,R3              ; e(n) = d(n) - y(n)
          STF       R3,@e              ; Store e(n)
*
*   UPDATE WEIGHTS w[] AND SHIFT x[]
*
          MPYF      @u,R3              ; R3 = e(n) * u
          MPYF3     *AR0++(1)%,R3,R1   ; R1 = e(n) * u * x(n)
          LDI       @order,RC          ; Initialize repeat counter
          SUBI      1,RC
          RPTB      LMS                ; Do i = 0, N-3
          MPYF3     *AR0++(1)%,R3,R1   ; R1 = e(n) * u * x(n-i-1)
       || ADDF3     *AR1,R1,R2         ; R2 = wi(n) + e(n) * u * x(n-i)
LMS       STF       R2,*AR1++(1)%      ; wi(n+1) = wi(n) + e(n) * u * x(n-i)
          MPYF3     *AR0,R3,R1         ; for i = N - 2
       || ADDF3     *AR1,R1,R2
          STF       R2,*AR1++(1)%      ; wi(n+1) = wi(n) + e(n) * u * x(n-i)
          ADDF3     *AR1,R1,R2
          STF       R2,*AR1++(1)%      ; Update last w
*
          POPF      R3
          POP       R3
          POPF      R2
          POPF      R1
          POP       R1
*
          RETS
          .end
```

Appendix H5. Linker Command/file for Assembly Main Program Calling the TMS320C30 Adaptive LMS Transversal Filter Subroutine

```
/***************************************************************************/
/* ADAP.CMD  -  COMMAND FILE FOR LINKING TMS320C30 ADAPTIVE FILTER        */
/*              PROGRAMS                                                   */
/*                                                                         */
/*      Usage: lnk30 <obj files...> -o <out file> -m <map file> adap.cmd  */
/*                                                                         */
/* Description: This file is a sample command file that can be used      */
/*              for linking adaptive filter assembly programs.             */
/*              All the adaptive filter programs have to link with the    */
/*              ADAPINIT.ASM file to do the auto_initialization.          */
/*                                                                         */
/* Notes:       When using the small (default) memory model, be sure      */
/*              that the ENTIRE .bss section fits within a single page.   */
/*              To satisfy this, vars must be smaller than 64K words and  */
/*              must not cross any 64K boundaries.                         */
/***************************************************************************/

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
   VECS:   org = 0         len = 0xc0
   ROM:    org = 0xc0      len = 0x7fff40
   RAM0:   org = 0x809800 len = 0x3c0    /* RAM block 0                  */
   _STACK: org = 0x809bc0 len = 0x40     /* System stack                 */
   RAM1:   org = 0x809c00 len = 0x3c0    /* RAM block 1                  */
   _VARS:  org = 0x809fc0 len = 0x1040   /* VARS block - 40h + 4k of EXT */
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
   vectors: {} > VECS                 /* Interrupt vectors               */
   .text:   {} > ROM                  /* Code                            */
   .cinit:  {} > ROM                  /* Initialization tables           */
   .stack:  {} > _STACK               /* System stack                    */
   vars:    {} > _VARS                /* Memory for variables            */
   buffer:  {} > RAM0                 /* Memory for data buffer          */
   coeffs:  {} > RAM1                 /* Memory for filter coefficients  */
   gains align(32) : {} > RAM1        /* Memory for lattice filter gains */
}
```

Appendix I1. C Subroutine of Transversal Structure with LMS Algorithm Using the TMS320C25

```
        .title    'CLMS'
**************************************************************
*
*   CLMS :  Adaptive Filter C subroutine using Transversal Structure
*           and LMS Algorithm, Looped Code
*
*   Algorithm:
*
*               N-1
*       y(n) = SUM w(k)*x(n-k) k=0,1,2,...,N-1
*               k=0
*
*       e(n) = d(n) - y(n)
*
*       w(k) = w(k) + u*e(n)*x(n-k) k=0,1,2,...,N-1
*
*       Where we use filter order = N
*
*   Usage:     lms(n,mu,d,x,&y,&e)
*              n  - order of filter
*              mu - convergence factor
*              d  - desired signal
*              x  - input signal
*              &y - addr of output signal
*              &e - addr of error signal
*
*   Note:      Data memory 0200h  0200h+N-1 & 0300h  0300h+N-1 are reserved.
*
*                   Chen, Chein-Chung  February, 1989
*
*********************************************************
        .def      _lms
*
*   RESERVE ADDRESSES FOR PARAMETERS
*
DST0:       .usect    "parameters",1
DST1:       .usect    "parameters",1
SAVE1:      .usect    "parameters",1
SAVE2:      .usect    "parameters",1
SAVE3:      .usect    "parameters",1
SAVE4:      .usect    "parameters",1
ORDER:      .usect    "parameters",1
X:          .usect    "parameters",1
D:          .usect    "parameters",1
U:          .usect    "parameters",1
Y:          .usect    "parameters",1
ERR:        .usect    "parameters",1
ERRF:       .usect    "parameters",1
ADRLST:     .usect    "parameters",1
*
*   DEFINE ADDRESSES OF BUFFER AND COEFFICIENTSS
COEFFP:     .equ      0ff00h
COEFFD:     .equ      0200h
FRSTAP:     .equ      0300h
*****************************************
*   PERFORM THE ADAPTIVE FILTER
*****************************************
*
*   SAVE THE VALUES OF THE REGISTERS
*
            .text
_lms        SAR       AR1,SAVE1
            SAR       AR2,SAVE2
            SAR       AR3,SAVE3
            SAR       AR4,SAVE4
            SST       DST0
            SST1      DST1
*
*   GET THE ADAPTIVE FILTER PARAMETERS
*
            SPM       1                  ; Set P register shift mode
            SSXM                         ; Set sign extension mode
            SOVM                         ; Set overflow mode
            LDPK      0                  ; Set data page = 0
            MAR       *-                 ; Set pointer for getting papameter
            LAC       *-                 ; ACC = N
            SUBK      1
            SACL      ORDER              ; ORDER = N - 1
            ADLK      FRSTAP
            SACL      ADRLST             ; Store address of last tap
            LAC       *-
            SACL      U                  ; Get and store the MU
            LAC       *-
            SACL      D                  ; Get and store the D
            LAC       *-,0,AR3
            LRLK      AR3,FRSTAP
            SACL      *                  ; Insert newest sample
*
*   ESTIMATE THE SIGNAL Y
*
            CNFP                         ; Configure B0 as program memory
            MPYK      0                  ; Clear the P register
            LALK      1,15               ; Using rounding
            LAR       AR3,ADRLST         ; Point to the oldest sample
FIR         RPT       ORDER              ; Repeat N times
            MACD      COEFFP,*-          ; Estimate Y(n)
            CNFD                         ; Configure B0 as data memory
            APAC
            SACH      Y                  ; Store the filter output
*
*   COMPUTE THE ERROR
*
            NEG                          ; ACC = - Y(n)
            ADDH      D
```

```
        SACH    ERR             ; ERR(n) = D(n) - Y(n)
*
*   UPDATE THE WEIGHTS
*
        LT      ERR             ; T = ERR(n)
        MPY     U               ; P = U * ERR(n)
        PAC
        ADLK    1,15            ; Round the result
        SACH    ERRF            ; ERRF = U * ERR(n)
*
        LAR     AR4,ORDER       ; Set up counter
        LRLK    AR2,COEFFD      ; Point to the coefficients
        LAR     AR3,ADRLST      ; Point to the data sample
        MAR     *+
        LT      ERRF            ; T register = U * ERR(n)
        MPY     *-,AR2          ; P = U * ERR(n) * X(n-k)
ADAPT   ZALR    *,AR3           ; Load ACCH with A(k,n) & round
        MPYA    *-,AR2          ; W(k,n+1) = W(k,n) + P
*                               ; P = U * ERR(n) * X(n-k)
        SACH    *+,0,AR4        ; Store W(k,n+1)
        BANZ    ADAPT,*-,AR2
*
*   STORE THE Y AND ERR
*
        LARP    AR1
        LAR     AR2,*-,AR2      ; Get the address of Y (in MAIN)
        LAC     Y
        SACL    *,0,AR1         ; Store Y
        LAR     AR2,*,AR2       ; Get the address of ERR (in MAIN)
        LAC     ERR
        SACL    *,0,AR1         ; Store ERR
*
*   RESTORE THE REGISTERS
*
        LST     DST0
        LST1    DST1
        LAR     AR1,SAVE1
        LAR     AR2,SAVE2
        LAR     AR3,SAVE3
        LAR     AR4,SAVE4
*
FINISH RET
*
        .end
```

Appendix I2. C Subroutine of Transversal Structure with LMS Algorithm Using the TMS320C30

```
****************************************************************
*
*  CT30 - TMS320C30 C subroutine adaptive transversal filter with
*     LMS algorithm.
*
*  Algorithm:
*
*           N-1
*     y(n) = SUM w(k)*x(n-k) k=0,1,2,...,N-1
*           k=0
*
*     e(n) = d(n) - y(n)
*
*     w(k) = w(k) + u*e(n)*x(n-k) k=0,1,2,...,N-1
*
*     Where we use filter order = N and mu = 0.01.
*
*  Usage:  tlms(n,mu,d,&w,&x,&y,&e)
*          n  - order of filter
*          mu - convergence factor
*          d  - desired signal
*          &w - filter coefficients
*          &x - input signal buffer
*          &y - addr of output signal
*          &e - addr of error signal
*
*                 Chen, Chein-Chung  March,  1989
*
****************************************************************
          .global   _tlms
FP        .set      AR3
****************************************************************
*    PERFORM ADAPTIVE FILTER
****************************************************************
          .text
_tlms     .set      $
          PUSH      FP
          LDI       SP,FP
          PUSH      AR0
          PUSH      AR1
          PUSH      AR2
          PUSH      R1
          PUSHF     R1
          PUSH      R2
          PUSHF     R2
          PUSH      R4
          PUSHF     R6
          PUSHF     R7
*
*  GET FILTER PARAMETERS
*
          LDI       *-FP(2),R4          ; Get filter order
          LDI       *-FP(6),AR0         ; Get pointer for x[]
          LDI       *--FP(5),AR1        ; Get pointer for w[]
          SUBI      2,R4                ; Set loop counter
*
*  COMPUTE FILTER OUTPUT y(n)
*
          LDF       0.0,R2              ; R2 = 0.0
*
          MPYF3     *AR0++(1),*AR1++(1),R1
          RPTS      R4
          MPYF3     *AR0++(1),*AR1++(1),R1
       || ADDF3     R1,R2,R2            ; y(n) = w[].x[]
          ADDF      R1,R2               ; Include last result
*
*  COMPUTE ERROR SIGNAL e(n) AND STORE y(n) AND e(n)
*
          LDI       *-FP(2),AR2         ; Get y(n) address
          SUBF3     R2,*+FP(1),R7       ; e(n) = d(n) - y(n)
       || STF       R2,*AR2             ; Send out y(n)
          LDI       *-FP(3),AR2         ; Get e(n) address
          STF       R7,*AR2             ; Send out e(n)
*
*  UPDATE WEIGHTS w[] AND SHIFT x[]
*
          MPYF      *+FP(2),R7          ; R7 = e(n) * u
          MPYF3     *--AR0(1),R7,R1     ; R1 = e(n) * u * x(n-N+1)
          LDI       R4,RC               ; Initialize repeat counter
          RPTB      LMS                 ; Do i = 1, N-1
          MPYF3     *--AR0(1),R7,R1     ; R1 = e(n) * u * x(n-i+1)
       || ADDF3     *--AR1(1),R1,R2     ; R2 = wi(n) + e(n) * u * x(n-i)
          LDF       *AR0,R6             ; Get x((n+i-N+1)
       || STF       R2,*AR1             ; wi(n+1) = wi(n) + e(n) * u * x(n-i)
LMS       STF       R6,*+AR0(1)         ; Shift x[]
          ADDF3     *--AR1(1),R1,R2     ; R2 = wi(n) + e(n) * u * x(n)
          STF       R2,*AR1             ; Update last w
*
          POPF      R7
          POPF      R6
          POP       R4
          POPF      R2
          POP       R2
          POPF      R1
          POP       R1
          POP       AR2
          POP       AR1
          POP       AR0
          POP       FP
          RETS
*
          .end
```