Adaptive Differential Pulse Code Modulation

12

12.1 OVERVIEW

Pulse code modulation (PCM) samples an input signal using a fixed quantizer to produce a digital representation. This technique, although simple to implement, does not take advantage of any of the redundancies in speech signals. The value of the current input sample does not have an effect on the coding of future samples. Adaptive differential PCM (ADPCM), on the other hand, uses an adaptive predictor, one that adjusts according to the value of each input sample, and thereby reduces the number of bits required to represent the data sample from eight (non-adaptive PCM) to four.

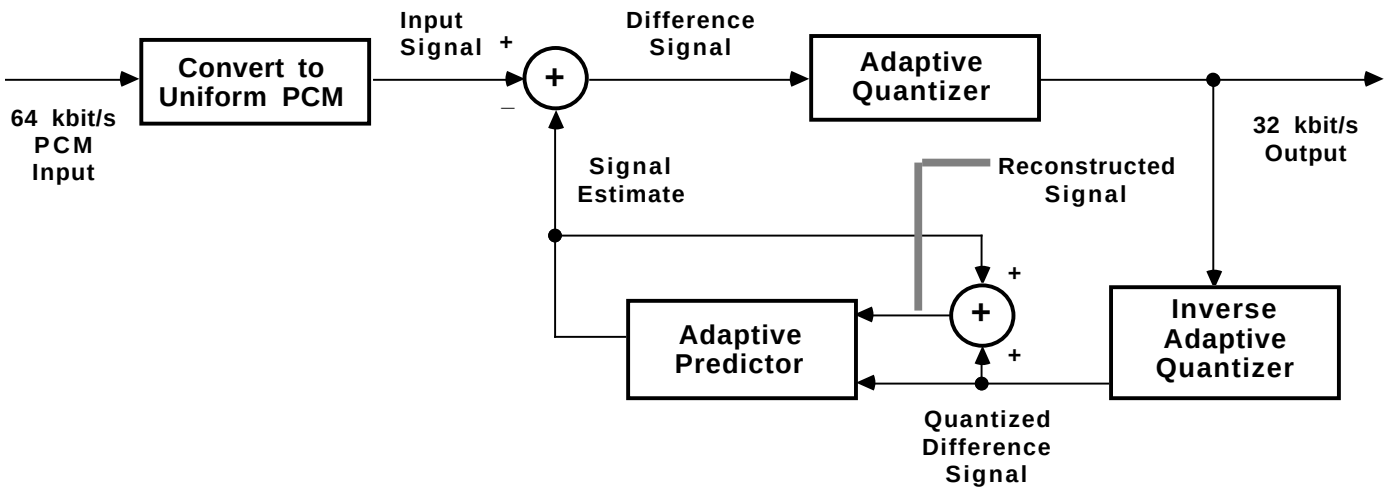
ADPCM does not transmit the value of the speech sample, but rather the difference between a predicted value and the actual sample value. Typically, an ADPCM transcoder is inserted into a PCM system to increase its voice channel capacity. Therefore, the ADPCM encoder accepts PCM values as input, and the ADPCM decoder outputs PCM values. For a complete description of PCM implementation on the ADSP-2100, see Chapter 11, *Pulse Code Modulation.*

12.2 ADPCM ALGORITHM

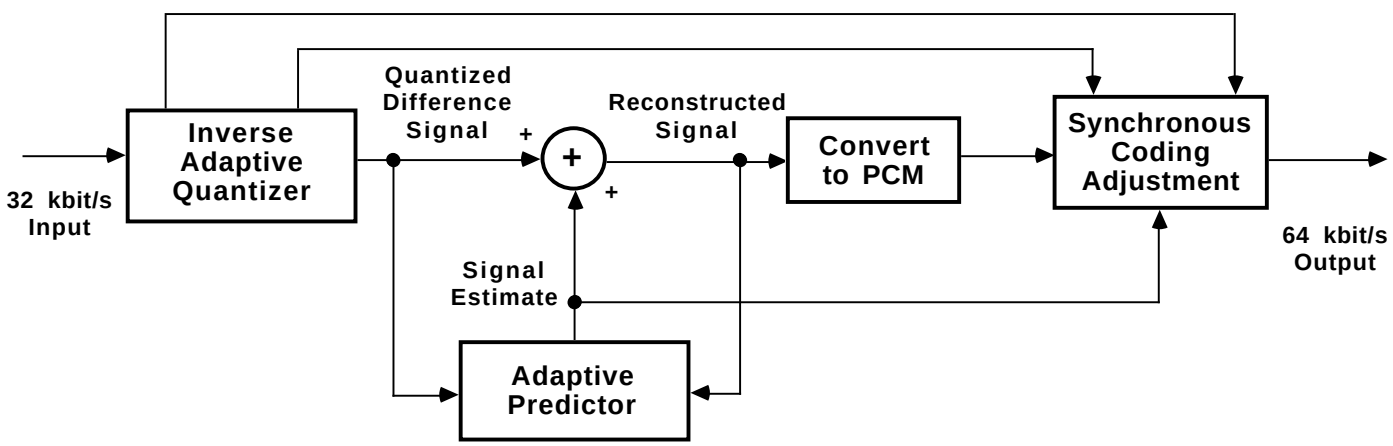
The ADSP-2100 implementation of ADPCM presented in this chapter is based on the CCITT recommendation G.721 and the identical ANSI recommendation T1.301-1987. The routines have been checked with the digital test sequences provided for the standards and are fully compatible with the recommendation. The terms and equations used in the recommendation are too numerous to be explained fully in this chapter. You should refer to the complete text of the recommendation, which can be obtained from the CCITT or ANSI (see *References* at the end of this chapter).

Figure 12.1, on the following page, shows a block diagram of the ADPCM algorithm. An 8-bit PCM value is input and converted to a 14-bit linear format. The predicted value is subtracted from this linear value to generate a difference signal. Adaptive quantization is performed on this difference, producing the 4-bit ADPCM value to be transmitted.

12 ADPCM



ENCODER



DECODER

Figure 12.1 ADPCM Block Diagram

Both the encoder and decoder update their internal variables based on only the generated ADPCM value. This ensures that the encoder and decoder operate in synchronization without the need to send any additional or sideband data. A full decoder is embedded within the encoder to ensure that all variables are updated based on the same data.

In the receiving decoder as well as the decoder embedded in the encoder, the transmitted ADPCM value is used to update the inverse adaptive quantizer, which produces a dequantized version of the difference signal. This dequantized value is added to the value generated by the adaptive predictor to produce the reconstructed speech sample. This value is the output of the decoder.

The adaptive predictor computes a weighted average of the last six dequantized difference values and the last two predicted values. The coefficients of the filter are updated based on their previous values, the current difference value, and other derived values.

The ADPCM transcoder program is presented in the listings at the end of this chapter. Two versions are provided: one that conforms fully to the recommendation and a faster version that does not conform fully. This program has two sections, *adpcm_encode* and *adpcm_decode*. Both routines update and maintain the required variables and can be called independently. The program listings indicate the registers that must be initialized before calling each routine.

The code presented in this chapter executes the ADPCM algorithm on both the ADSP-2100 and the ADSP-2101. The routines duplicate the variable and function names indicated in G.721 whenever possible, making it easy to locate the code that implements each functional block.

The format of many of the variables specified in the standard are sign-extended to the full 16-bit word size of the ADSP-2100. This data size works efficiently with the ADSP-2100 and does not affect the ADPCM algorithm. In all cases, this implementation provides at least the smallest data format required.

12.3 ADPCM ENCODER

The ADPCM encoder is shown in Figure 12.2, on the next page. The 8-bit PCM input value is converted to a 14-bit linear representation in the *expand* routine using PCM decoder routines described in Chapter 11, *Pulse Code Modulation*. This linear value is stored in the data memory location *sl*.

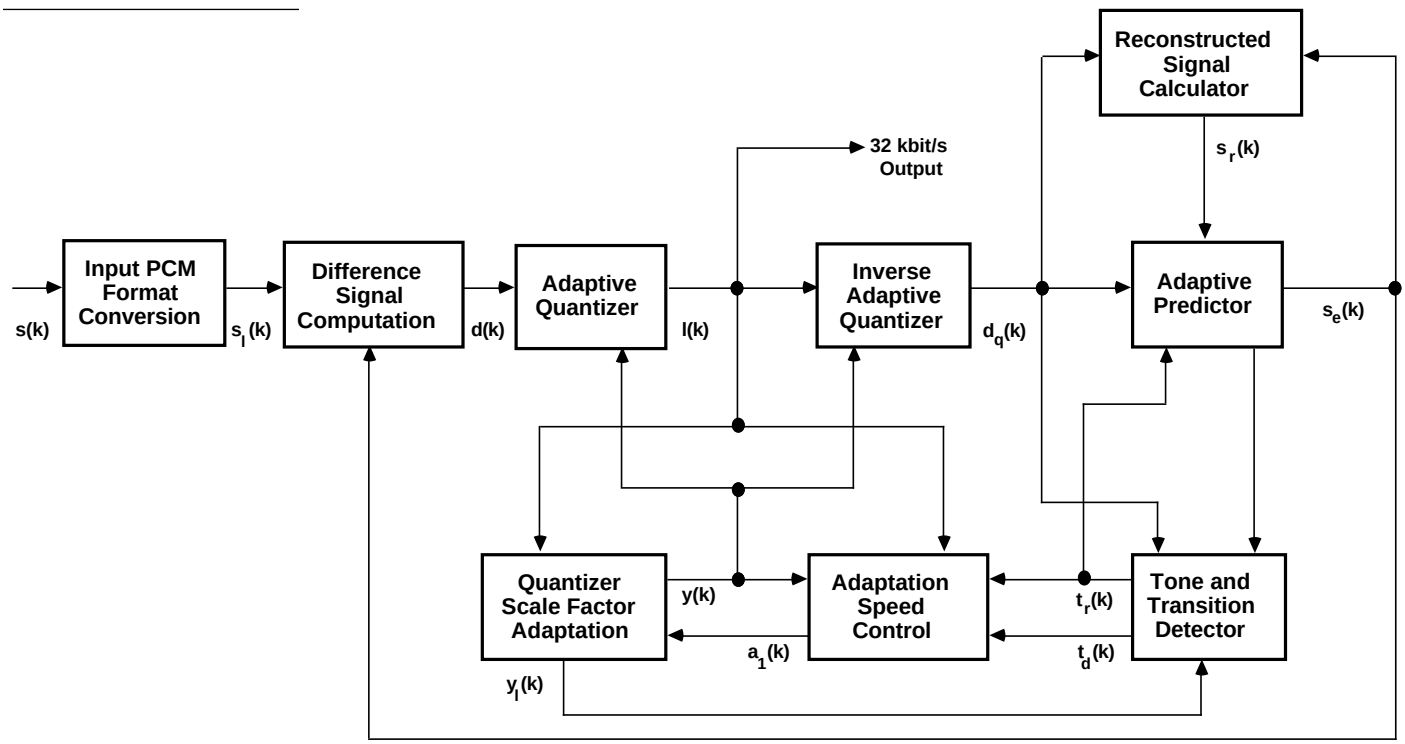


Figure 12.2 Encoder (Transmitter) Block Diagram

12.3.1 Adaptive Predictor

The predicted value of the linear sample is computed by the following equation

$$s_e(k) = \sum_{i=1}^{2} a_i(k-1)\, s_r(k-i) + s_{ez}(k)$$

where

$$s_{ez}(k) = \sum_{i=1}^{6} b_i(k-1)\, d_q(k-i)$$

The *predict* routine produces the predicted value $s_e(k)$ in the variable *s_e* and its component $s_{ez}(k)$ in the variable *sez*, implementing the FMULT and ACCUM functional blocks of the ADPCM algorithm. The running total the WAn and WBn variables is stored in the AF register during this routine. The sum of the WBns are computed first in the *sez_cmp* loop to produce *sez*, the zero component of the predicted value. The second loop, *s_e_cmp*, is identical to the first except for the address registers used; it computes the two WAns and adds them to the sum to produce the predicted sample value, *s_e*.

The implementation of the *s_r* and *dq* delay lines in the *predict* routine is slightly more efficient than the method used in the G.721 recommendation. The variables of the algorithm are in floating-point format. The three parts (sign, exponent, mantissa) of each delay line variable are not packed in a single memory location (as specified in the recommendation), but occupy three separate locations. This is a more efficient method because packing and unpacking the data is not necessary. Because the ADSP-2100 can perform a data fetch in parallel with other operations, using this method does not affect performance.

Once the floating-point components have been determined, the mantissa of the result (W-value) must be generated. During this operation, a fixed offset is added to the product before shifting. In this implementation the MR registers are loaded with the properly shifted product, then the offset is added. The MX1 and MY1 registers are used to generate the properly oriented offset value. The output is in the MR1 register, ready to be shifted by the exponent value.

12.3.2 Adaptive Quantizer

Two routines update the parameters of the adaptive quantizer. The first, *lima*, computes the new value for the speed control variable, *al*, which is used to weight the two components of the quantizer scale factor. The speed control variable varies between 0 and 1, tending towards 1 for speech signals. It is computed using a predicted value, *ap*, which can vary between 0 and 2. The relationship between these two variables is defined by:

$$a_l(k) = \begin{cases} 1, & a_p(k-1) \geq 1 \\ a_p(k-1), & a_p(k-1) \leq 1 \end{cases}$$

Because *ap* is stored in 8.8 format, it is compared to 256 (1 in 8.8 format) and capped at that value. It is shifted down twice to remove the two LSBs, then shifted up nine bits to its proper position for the *mix* routine.

The *mix* routine, using the speed control variable *al*, adds two factors, *yu*, the unlocked factor, and *yl*, the locked factor, to produce the quantizer scale factor variable *y*. The equation for *y* is

$$y(k) = a_l(k)\, y_u(k-1) + [1 - a_l(k)]\, y_l(k-1)$$

The scale factor *y* adapts in a different fashion for speech signals and voice band data.

The *mix* routine computes the difference between *yu* and the upper bits of *yl*. The absolute value of this difference is multiplied by *al*. The product is then added (or subtracted, based on the sign of the difference) to *yl* to produce *y*. The value *y* downshifted twice is also calculated, because this value is needed by other routines.

The difference between the linear input signal and the predicted signal is computed. This value is used by the quantizer to compute the 4-bit ADPCM value. The quantizer computes the base 2 logarithm of the difference value, then subtracts the scale factor *y*. The result is used to index a table that contains the ADPCM quantizer values. This table contains the quantizer segment endpoints and is used to determine the final ADPCM value.

The *log* routine that produces the quantized value takes the absolute value of the difference value. The exponent detector determines the number of redundant sign bits, then 14 is added to generate the exponent value. The $\log_2$ approximation is generated by normalizing the difference and masking out the two MSBs (which is equivalent to subtracting 1).

The exponent value is shifted up seven bits and ORed with the $\log_2$ approximation of the difference shifted down seven bits. The value *y* is subtracted, then a table lookup is performed to retrieve the ADPCM value. The lower end point of each quantizer range is stored in memory, and the AF register is initialized to 7. As each segment end point is checked, AF is decremented if the value is less than the lower end point. When the lower end point is less than or equal to the quantized value, AF holds the ADPCM value.

12.3.3 Inverse Adaptive Quantizer

The calculated 4-bit ADPCM value (I value) is used by the *reconst* routine. The magnitude of the I value is mapped to a data memory location that contains the value (DQLN in the recommendation) to be added to the scale factor. The inverse $\log_2$ of this sum is the quantized difference value, *dq*, which is used by the prediction and update routines.

The *reconst* routine takes the absolute value of the ADPCM input. This value is placed in the M3 register to be used in this and other routines for table lookup. The dequantized value is read from memory, and *y* is added to this value. Log-to-linear conversion is done by first isolating the exponent in SR1. The mantissa is then isolated in AR, and 1 (H#80) is added to it. The magnitude of the shift is determined by subtracting 7 from the exponent in SR1. In the last step, the sign of the dequantized difference signal is determined from the sign of the ADPCM input (I) value.

12.3.4 Adaptation Speed Control

Several of the internal variables of the algorithm are updated in one large subroutine, beginning with the scale factor components. The unlocked component, *yu*, able to adapt to a quickly changing signal, is based on the scale factor and the recently produced ADPCM value. This factor is explicitly limited to the range from 1.06 to 10.00. Other factors derived from *yu* (*y* and *yl*) are therefore implicitly limited to this same range.

The locked factor, *yl*, which adapts more slowly than *yu*, is based on the previous locked factor value as well as the current unlocked factor. The unlocked and locked factors are updated by *filtd* and *filte*, respectively. The unlocked scale factor *yu* is computed as follows:

$$y_u(k) = (1–2^{-5})\, y(k) + 2^{-5}W[I(k)]$$

The function W(I) is determined using a table lookup in the *functw* routine. The code block labeled *filtd* shifts W[I(k)] into its proper format and subtracts *y*. This double-precision remainder is downshifted five bits to accommodate the time factor. This downshifted value (gain) is added to *y* to produce *yu*.

The code block labeled *limb* limits *yu* to the range $1.06 \le yu(k) \le 10.00$. This explicit limitation on *yu* implicitly limits both *yl* and *y* to the same range. The locked factor is determined as follows:

$$y_l(k) = (1–2^{-6})\, y_l(k–1) + 2^{-6} y_u(k)$$

The update of *yl* is accomplished by first negating *yl* (in double precision) and adding the MSW of the remainder to the new *yu*. This sum is downshifted six bits and added to the original value of *yl* in double precision to produce the updated value.

The long- and short-term averages of the ADPCM value must also be updated. These values are used to compute the predicted speed control factor, *ap*, used in the next cycle of the loop. The code blocks at the *filta* and *filtb* labels update the averages, while the *filtc* code computes the new predicted weighting factor.

The short-term average (*dms*) is updated by the *filta* routine. The required F-value is determined by a table lookup. The old short-term average is subtracted from the F-value and downshifted five bits. This gain is added to the previous short-term value to generate the updated value.

$$d_{ms}(k) = (1–2^{–5})\, d_{ms}(k–1) + 2^{–5}\mathbf{F}[I(k)]$$

The long-term average is updated by the *filtb* routine.

$$d_{ml}(k) = (1–2^{–7})\, d_{ml}(k–1) + 2^{–7}\mathbf{F}[I(k)]$$

12.3.5 Predictor Coefficient Update

The *update_filter* routine has several parts, all of which are used to compute the new b and a filter coefficients. The first step is to update the b filter coefficients, whose new values are based their previous values plus a weighted product of the signs of the current quantized difference value, and the associated *dq* value contained in the filter delay line. The equation implicitly limits the coefficients to a maximum absolute value of 2.

To update the a-filter coefficients, the routine computes the sum of the quantized difference signal and the portion of the estimated signal computed by the b-coefficients. The sign of this value and the previous values are used when computing the new coefficients. Each a-value computed is also explicitly limited, to increase the stability of the filter.

The *update_filter* routine first computes the magnitude of the gain for the b-coefficients. If the current *dq* value is 0, then the gain is also 0; otherwise the magnitude of the gain is 128 ($2^{–7}$ in 2.14). The sign of the gain is the exclusive-OR of the sign of the current *dq* and the delayed *dq* associated with each coefficient. The gain is added to the old coefficient and the leak

factor ($b(k) \times 2^{-8}$) is subtracted for that sum. The new coefficient is stored in program memory and the loop is re-executed for the remaining coefficients. The relationship between the old and new b-coefficient values is defined by:

$$b_i(k) = (1-2^{-8})\, b_i(k-1) + 2^{-7}\mathrm{sgn}[d_q(k)]\ \mathrm{sgn}[d_q(k-i)]$$

After the b-coefficients are updated, the current dequantized difference value (*dq*) is placed in the delay line. This requires a fixed-to-floating-point conversion. Remember that the delay line variable used in this implementation is three separate words instead of one packed word.

$$a_2(k) = (1-2^{-7})\, a_2(k-1) + 2^{-7}\{\mathrm{sgn}[p(k)]\ \mathrm{sgn}[p(k-2)] - f[a_1(k-1)]\ \mathrm{sgn}[p(k)]\ \mathrm{sgn}[p(k-1)]\}$$

The update of the *pk* variables, which are the current and delayed sign values of the partial signal estimate, occurs in the code labeled *update_p*. Two of the MR registers are loaded with update values based on the *pk* values. This are used later in the update sections for the a-values.

$$a_1(k) = (1-2^{-8})\, a_1(k-1) + (3 \times 2^{-8})\ \mathrm{sgn}[p(k)]\ \mathrm{sgn}[p(k-1)]$$

The variable a_2 is updated first, because its value is used to limit the new value of a_1. The first step is to generate the function f, which is based on the previous value of a_1. The value for $f(a_1)$ is added to (or subtracted from) the gain, in double precision. The new value for a_2 is generated at the code labeled *upa2*. After being limited, the value of a_2 is stored and used to check for a partial band signal. The tone detector uses the a_2 value as described later in this document.

The variable a_1 is updated by the code labeled *upa1*. This final step of the update process limits a_1 based on the new value of a_2. The code labeled *limd* performs this operation.

12.3.6 Tone and Transition Detector

The last step of the algorithm checks for a tone on the input. If the tone is present, the prediction coefficients are set to 0 and the predicted weighting factor to 256 (1 in 8.8). This configuration allows rapid adaptation and improved performance with FSK modems. The *trigger_true* routine is called if the tone is detected (*tr* = 1) to set the variables to the appropriate values.

12 ADPCM

The code labeled *tone* checks whether a_2 is less than –0.71875. If it is, the variable *tdp* (tone detect) is set to 1. The *tdp* variable is used in both the *subtc* and *trans* routines.

The *trans* routine implements the transition detector. If *tdp* is a 1 and the absolute value of *dq* exceeds a predetermined threshold, the *tr* (trigger) variable is set to 1. After the filter coefficients are updated, *tr* is checked; if it is a 1, the prediction coefficients are set to zero.

12.4 ADPCM DECODER

The ADPCM decoder is shown in Figure 12.3. The core of the decoder is the same as the decoder embedded within the encoder, so most of the decoder is described in the previous encoder sections. The major difference is that the decoder routines are called with different variables (see *Program Listings* at the end of this chapter). In the decoder, all unique variables and code have an *_r* appended to their names.

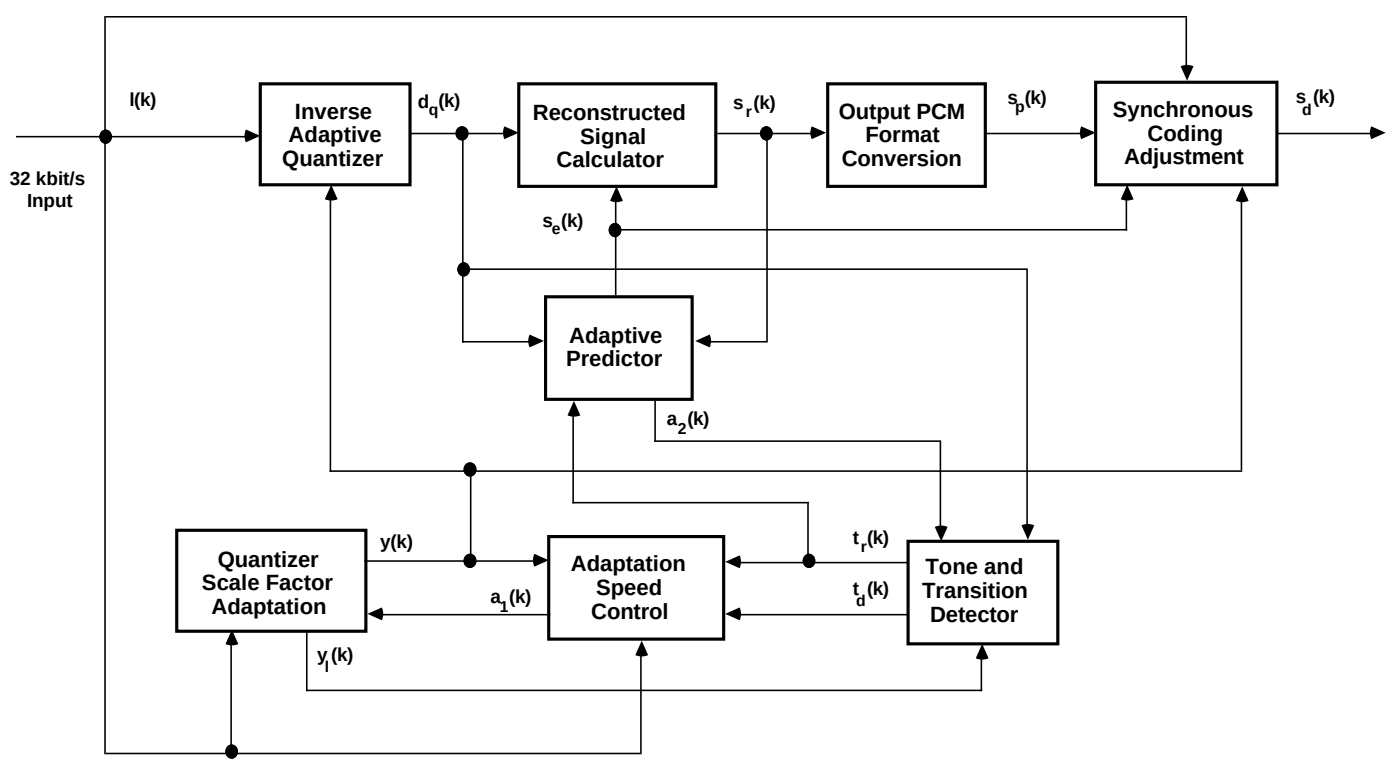


Figure 12.3 Decoder (Receiver) Block Diagram

The decoder filter update and trigger routines are the same as for the encoder, but because they use variable names within this code, it is more efficient to have separate routines rather than have the encoder and decoder call the same routines.

The decoder contains two additional routines. The *compress* routine converts a linear-PCM value to a logarithmic-PCM value. This routine operates on the reconstructed value (*s_r*).

The *sync* routine adjusts the final logarithmic-PCM output. This synchronous coding adjustment ensures no cumulative distortion of the speech signal on tandem (ADPCM-PCM-ADPCM) codings. It performs this function by determining whether another downstream ADPCM encoder would produce I values different from those of the decoder; if so, it increments or decrements the logarithmic-PCM output value by an LSB to prevent this distortion.

12.5 NONSTANDARD ADPCM TRANSCODER

In some applications (voice storage, for example) not all properties of the standard ADPCM algorithms are needed. An application that codes only speech data and is not used with voice band data does not require the tone and transition detectors. The synchronous coding adjustment can be removed if the transcoder is not used in a telecommunications network in which multiple tandem codings can occur.

A version of the ADPCM code with these three sections *(trigger*, *trans*, and *sync*) removed is shown in *Program Listings*, at the end of this chapter. The nonstandard encoder and decoder routines are called *ns_adpcm_encode* and *ns_adpcm_decode*, respectively. There is no noticeable difference in the quality of the speech, and it is possible to operate two full-duplex channels on a 12.5MHz ADSP-2101.

12.6 COMPANDING TECHNIQUES

The CCITT version of ADPCM works with both the A-law and μ-law PCM companding techniques. The ADPCM algorithm itself does not change, only the PCM input and output are different. The appropriate *expand* and *compress* routines must be in the code to ensure proper operation. Both programs in this chapter are based on μ-law companding.

Changes to three routines are needed to adapt the programs in this chapter to A-law companding: *expand*, *compress* and *sync*. Chapter 11, *Pulse*

12 ADPCM

Code Modulation, contains the A-law companding routines for *expand* and *compress;* the *sync* routine for A-law values is described below. The ADSP-2101 performs both A-law and μ-law companding in hardware, so for ADSP-2101 operation, only the *sync* routine needs to be changed.

The synchronous coding adjustment (*sync* routine) reduces errors on tandem ADPCM-PCM-ADPCM codings by adjusting the logarithmic-PCM output by (at most) one LSB on output of the decoder. The A-law *sync* routine is shown in Listing 12.1.

```
sync:          AX0=DM(a_ik_r);             {Get input value of I}
               AY1=AR, AF=ABS AR;
               IF NEG AR=AY1+1;            {Convert 1s comp to 2s comp}
               AY1=AX0, AF=ABS AX0;
               IF NEG AF=AY1+1;            {Same for new I value }
               AR=AR-AF;
               AR=DM(sp_r);
               IF GT JUMP decrement_sp;    {Next most negative value}
               IF LT JUMP increment_sp;    {Next most positive value}
               AF=PASS AX0;                {Check for invalid 0 input}
               IF NE RTS;

increment_sp:  SR=LSHIFT AR BY 8 (HI);     {Get sign of PCM value}
               AY0=H#AA;
               AF=AR-AY0;                  {Check for maximum value}
               IF EQ RTS;                  {Already maximum value}
               AY0=H#55;                   {Check for sign change}
               AF=AR-AY0;
               IF NE JUMP no_pos_sgn;      {Jump if no sign change}
               AR=H#D5;
               RTS;
```

```
no_pos_sgn:    AF=ABS SR1;
               AF=AR XOR AY0;
               AR=AF-1;                    {Compute adjusted PCM value}
               IF NEG AR=AF+1;
               AR=AR XOR AY0;
               RTS;

decrement_sp:  SR=LSHIFT AR BY 8 (HI);     {Get sign of PCM value}
               AY0=H#2A;
               AF=AR-AY0;                  {Check for minimum value}
               IF EQ RTS;                  {Already minimum value}
               AY0=H#D5;
               AF=AR-AY0;
               IF NE JUMP no_sign_chn;     {If input is H#D5}
               AR=H#55;                    {New output will be H#55}
               RTS;

no_sign_chn:   AF=ABS SR1;                 {Otherwise adjust by 1}
               AY0=H#55;
               AF=AR XOR AY0;
               AR=AF+1;                    {Compute adjusted PCM value}
               IF NEG AR=AF-1;
               AR=AR XOR AY0;
               RTS;
```

Listing 12.1 A-Law Synchronous Coding Adjustment Routine

12 ADPCM

12.7 BENCHMARKS AND MEMORY REQUIREMENTS

The following tables indicate the number of cycles and execution time of both the ADPCM algorithms on each of the ADSP-210X processors. The memory requirements for the standard ADPCM algorithm are 628 words of program memory and 137 words of data memory. The nonstandard algorithm uses slightly less program memory space.

	Standard ADPCM			*Nonstandard ADPCM*		
	Cycle Count	*Time (μs)*	*Loading (%)*	*Cycle Count*	*Time (μs)*	*Loading (%)*
ADSP-2100 (8 MHz)						
Encode	467	58.3	46.7	437	54.63	43.7
Decode	514	64.2	51.4	409	51.13	40.9
Full Transcode	981	122	98.1	846	105.75	84.6
ADSP-2100A (12.5 MHz)						
Encode	467	37.3	29.8	437	34.96	27.97
Decode	514	41.1	32.8	409	32.72	26.18
Full Transcode	981	78.4	62.7	846	67.68	54.16
ADSP-2101 (12.5 MHz)						
Encode	433	34.6	27.7	403	32.24	25.80
Decode	459	36.7	29.3	375	30.00	24.00
Full Transcode	892	71.3	57.1	778	62.24	49.80

12.8 REFERENCES

American National Standards Institute, Inc. 1987. *American National Standard for Telecommunications: Digital Processing of Voice-Band Signals—Algorithm and Line Format for 32 kbit/s Adaptive Differential Pulse-Code Modulation (ADPCM)*. New York: ANSI, Inc.

International Telegraph and Telephone Consultative Committee. 1986. Study Group XVIII—Report R26(C), Recommendation G.721. *32 kbit/s Adaptive Differential Pulse-Code Modulation (ADPCM)*.

12.9 PROGRAM LISTINGS

The complete listings for both the standard and nonstandard ADPCM transcoders are presented in this section.

12.9.1 Standard ADPCM Transcoder Listing

The code below represents a full-duplex ADPCM transcoder. This program has been developed in accordance with ANSI specification T1.301-1987 and CCITT G.721 (bis). It is fully bit-compatible with the test vectors supplied by both of these organizations.

```
.MODULE       Adaptive_Differential_PCM;

{      Calling Parameters
              AR =   Companded PCM value (encoder)
                     or ADPCM I value (decoder)

              M0=3;  L0=18;
              M1=1;  L1=6;
              M2=-1
                     L3=0;
              M4=0   L4=6;
              M5=1   L5=2
              M6=-1  L6=5

       Return Values
              AR =   ADPCM I value (encoder)
                     or Companded PCM value (decoder)

       Altered Registers
              AX0, AX1, AY0, AY1, AF, AR,
              MX0, MX1, MY0, MY1, MR,
              I0, I1, I3, I4, I5, I6
              SI, SR
              M3

       Cycle Count
              467 cycles for encode
              514 cycles for decode

}
```

(listing continues on next page)

12 ADPCM

```
.ENTRY adpcm_encode, adpcm_decode;

.VAR/PM/CIRC  b_buf[6];                  {b coefficients for encode}
.VAR/PM/CIRC  a_buf[2];                  {a coefficients for encode}
.VAR/PM/CIRC  b_buf_r[6];                {b coefficients for decode}
.VAR/PM/CIRC  a_buf_r[2];                {a coefficients for decode}

.VAR/DM/CIRC  b_delay_r[18];             {dq delay for decode}
.VAR/DM/CIRC  a_delay_r[6];              {sr delay for decode}
.VAR/DM/CIRC  b_delay[18];               {dq delay for encode}
.VAR/DM/CIRC  a_delay[6];                {sr delay for encode}

.VAR/DM/CIRC  mult_data[5];              {predictor immediate data}

.VAR/DM       qn_values[10],dq_values[8]; {quantizer & dequantizer data}
.VAR/DM       f_values[12], w_values[8];  {update coefficient data}
.VAR/DM       a_data[10];

.VAR/DM       s_e,s_r,a_ik,dq,p;
.VAR/DM       sez,sl,yu,yl_h,yl_l,y,y_2,ap,p_o,p_o_o,dms,dml,tdp,tr;
.VAR/DM       a_ik_r,dq_r,p_r;
.VAR/DM       yu_r,yl_h_r,yl_l_r,ap_r,p_o_r;
.VAR/DM       p_o_o_r,dms_r,dml_r,tdp_r;

.VAR/DM       sp_r;                      {PCM code word for synchronous adj}

.VAR/DM       hld_a_t, hld_b_t, hld_a_r, hld_b_r;
```

```
.INIT  qn_values:   7, 14, H#3F80, 400, 349, 300, 246, 178, 80, H#FF84;
.INIT  dq_values :  h#F800, 4, 135, 213, 273, 323, 373, 425;
.INIT  f_values : -5, 0, 5120, 544, 0, 0, 0, 512, 512, 512, 1536, 3584;
.INIT  w_values:    65344, 288, 656, 1024, 1792, 3168, 5680, 17952;
.INIT  mult_data :  H#1FFF, H#4000, h#7E00, H#7FFF, H#FFFE;
.INIT  a_data :     H#1FFF, 2, 16384, 0, -7, 192, H#3000, H#D000,
                    H#D200,  H#3C00;

.INIT  hld_a_t : ^a_delay;
.INIT  hld_b_t : ^b_delay;
.INIT  hld_a_r : ^a_delay_r;
.INIT  hld_b_r : ^b_delay_r;

.INIT  b_buf : 0,0,0,0,0,0;                       {2.14}
.INIT  a_buf : 0,0;                               {2.14}
.INIT  b_delay : 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; {16.0, 16.0, 0.16}
.INIT  a_delay : 0,0,0,0,0,0;                     {16.0, 16.0, 0.16}
.INIT  p : 0;                                     {16.0}
.INIT  yu :0;                                     {7.9}
.INIT  yl_h : 0;                                  {7.9}
.INIT  yl_l : 0;                                  {0.16}
.INIT  ap : 0;                                    {8.8}
.INIT  p_o : 0;                                   {16.0}
.INIT  p_o_o : 0;                                 {16.0}
.INIT  dms : 0;                                   {7.9}
.INIT  dml : 0;                                   {5.11}
.INIT  tdp : 0;                                   {16.0}

.INIT  b_buf_r : 0,0,0,0,0,0;                     {2.14}
.INIT  a_buf_r : 0,0;                             {2.14}
.INIT  b_delay_r:0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; {16.0, 16.0, 0.16}
.INIT  a_delay_r : 0,0,0,0,0,0;                   {16.0, 16.0, 0.16}
.INIT  p_r : 0;                                   {16.0}
.INIT  yu_r :0;                                   {7.9}
.INIT  yl_h_r : 0;                                {7.9}
.INIT  yl_l_r : 0;                                {0.16}
.INIT  ap_r : 0;                                  {8.8}
.INIT  p_o_r : 0;                                 {16.0}
.INIT  p_o_o_r : 0;                               {16.0}
.INIT  dms_r : 0;                                 {7.9}
.INIT  dml_r : 0;                                 {5.11}
.INIT  tdp_r : 0;                                 {16.0}
```

(listing continues on next page)

```
adpcm_encode: I4=^b_buf;                {Set pointer to b-coefficients}
              I5=^a_buf;                {Set pointer to a-coefficients}
              I6=^mult_data;            {Set pointer to predictor data}
              I1=DM(hld_a_t);           {Restore pointer to s_r delay}
              I0=DM(hld_b_t);           {Restore pointer to dq delay}

              CALL expand;              {Expand 8-bit log-PCM to 12 bits}
              DM(sl)=AR;                {Store linear PCM value in sl}
              CALL predict;             {Call s_e and sez predictors}
              AX1=DM(ap);
              AY0=DM(yl_h);
              AX0=DM(yu);
              CALL lima;                {Limit ap and compute y}
              DM(y)=AR;                 {Save y for later updates}
              DM(y_2)=SR1;              {Save y>>2 for log and reconst}
              AX0=DM(sl);
              AY0=DM(s_e);
              AY1=SR1, AR=AX0-AY0;      {Compute difference signal, d}
              CALL log;                 {Determine I value from d}
              DM(a_ik)=AR;
              CALL reconst;             {Compute dq based ONLY on }
              DM(dq)=AR;
              AY0=DM(s_e);
              AR=AR+AY0;                {Compute reconstructed signal}
              DM(s_r)=AR;

              DM(I1,M1)=AR, AR=ABS AR;  {Convert s_r to floating point}
              SR1=H#4000;               {Set SR1 to minimum value}
              SE=EXP AR (HI);                    {Determine exponent adjust}
              AX0=SE, SR=SR OR NORM AR (HI);     {Normalize into SR}
              SR=LSHIFT SR1 BY -9 (HI); {Delete lower bits}
              AY0=11;                   {Base exponent}
              SR=LSHIFT SR1 BY 2 (HI);  {Adjust for ADSP-210x version}
              AR=AX0+AY0;               {Compute exponent}
              DM(I1,M1)=AR;             {Save exponent}
              DM(I1,M1)=SR1;            {Save mantissa}

              MR1=DM(tdp);
              SI=DM(yl_h);
              AX1=DM(dq);
              CALL trans;               {Compute new trigger }
              DM(tr)=AR;
```

```
AR=PASS AR;               {Check state of trigger}
IF EQ CALL update_filter; {Update filter if trigger false}

MR0=DM(ap);               {Load variables for updating}
MR1=DM(y);
MR2=DM(tdp);              {Always load MR2 after MR1!}
MY0=DM(yl_h);
MY1=DM(yl_l);
AY0=DM(y);
MX0=DM(dms);
MX1=DM(dml);
CALL functw;              {Update variables}
DM(ap)=AR;                {Store updated variables}
DM(yu)=AX1;
DM(yl_l)=MY1;
DM(yl_h)=MY0;
DM(dms)=MX0;
DM(dml)=MX1;

AY1=DM(tr);               {Load trigger }
AF=PASS AY1;              {Check state of trigger}
IF NE CALL trigger_true;  {Call only if trigger true}

AX0=DM(a_ik);             {Get I value for return}
AY0=H#F;                  {Only 4 LSBs are used}
AR=AX0 AND AY0;           {So mask redundant sign bits}

DM(hld_a_t)=I1;           {Save s_r delay pointer}
DM(hld_b_t)=I0;           {Save dq delay pointer}

RTS;                      {Return to caller}
```

(listing continues on next page)

```
adpcm_decode: I1=DM(hld_a_r);           {Restore s_r delay pointer}
              I0=DM(hld_b_r);           {Restore dq delay pointer}
              I4=^b_buf_r;              {Set pointer to b-coefficients}
              I5=^a_buf_r;              {Set pointer to a-coefficients}
              I6=^mult_data;            {Set pointer to predictor data}

              SR=LSHIFT AR BY 12 (HI);  {Get sign of ADPCM I value here}
              SR=ASHIFT SR1 BY -12 (HI);{Sign extend ADPCM value to 16}
              DM(a_ik_r)=SR1;           {Save I value}
              CALL predict;             {Call s_e and sez predictor}
              AX1=DM(ap_r);
              AY0=DM(yl_h_r);
              AX0=DM(yu_r);
              CALL lima;                {Limit ap and compute y}
              DM(y)=AR;
              DM(y_2)=SR1;
              AY1=DM(y_2);
              AR=DM(a_ik_r);
              CALL reconst;             {Compute dq from received I}
              DM(dq_r)=AR;
              AY0=DM(s_e);
              AR=AR+AY0;                {Compute reconstructed signal}
              DM(s_r)=AR;

              DM(I1,M1)=AR, AR=ABS AR;  {Make s_r floating point}
              SR1=H#4000;               {Set SR1 to minimum value}
              SE=EXP AR (HI);                    {Determine exponent adjust}
              AX0=SE, SR=SR OR NORM AR (HI);     {Normalize value}
              SR=LSHIFT SR1 BY -9 (HI); {Remove LSBs per spec}
              AY0=11;                   {Base exponent}
              SR=LSHIFT SR1 BY 2 (HI);  {Adjust for ADSP-210x version}
              AR=AX0+AY0;               {Compute exponent}
              DM(I1,M1)=AR;             {Store exponent}
              DM(I1,M1)=SR1;            {Store mantissa}

              MR1=DM(tdp_r);
              SI=DM(yl_h_r);
              AX1=DM(dq_r);
              CALL trans;               {Compute new trigger}
              DM(tr)=AR;
```

```
AR=PASS AR;               {Check state of trigger}
IF EQ CALL update_filter_r;{Update filter if trigger false}

AY0=DM(y);                {Load variables for updating}
MY1=DM(yl_l_r);
MY0=DM(yl_h_r);
MR0=DM(ap_r);
MR1=DM(y);
MR2=DM(tdp_r);            {Always load MR2 after MR1}
MX0=DM(dms_r);
MX1=DM(dml_r);
CALL functw;              {Update variables}
DM(yu_r)=AX1;             {Stored updated variables}
DM(yl_l_r)=MY1;
DM(yl_h_r)=MY0;
DM(ap_r)=AR;
DM(dms_r)=MX0;
DM(dml_r)=MX1;

AY1=DM(tr);               {Load current trigger}
AF=PASS AY1;              {Check state of trigger}
IF NE CALL trigger_true_r;{Call only if trigger true}

CALL compress;            {Compress PCM value}
DM(sp_r)=AR;              {Save original value for sync}
CALL expand;              {Expand for sync coding adj}
AY0=DM(s_e);
AR=AR-AY0;                {Compute dx for sync coding}
AY1=DM(y_2);
CALL log;                 {Compute new dqx value}
CALL sync;                {Adjust PCM value by +1,-1, 0}

DM(hld_a_r)=I1;           {Save s_r delay pointer}
DM(hld_b_r)=I0;           {Save dq delay pointer}

RTS;
```

(listing continues on next page)

12 ADPCM

```
compress:     AR=DM(s_r);                 {Get reconstructed signal}
              AR=ABS AR;                  {Take absolute value}
              AY0=33;                     {Add offset of boundries}
              AR=AR+AY0;
              AY0=8191;                   {Maximum PCM value}
              AF=AR-AY0;                  {Cap input}
              IF GT AR=PASS AY0;          {If in excess}
              SE=EXP AR (HI);             {Find exponent adjustmet}
              AX0=SE, SR=NORM AR (LO);    {Normalize input}
              AY0=H#4000;
              AR=SR0 XOR AY0;             {Remove first significant bit}
              SR=LSHIFT AR BY -10 (LO);   {Shift position bits}
              AR=PASS AY0;
              IF POS AR=PASS 0;           {Create sign bit}
              SR=SR OR LSHIFT AR BY -7 (LO);     {Position sign bit}
              AY0=9;
              AR=AX0+AY0;                 {Compute segment}
              IF LT AR=PASS 0;
              SR=SR OR LSHIFT AR BY 4 (LO);      {Position segment bits}
              AY0=H#FF;
              AR=SR0 XOR AY0;             {Invert bits}
              RTS;

expand:       AY0=H#FF;                   {Mask unwanted bits}
              AF=AR AND AY0, AX0=AY0;
              AF=AX0 XOR AF;              {Invert bits}
              AX0=H#70;
              AR=AX0 AND AF;              {Isolate segment bits}
              SR=LSHIFT AR BY -4 (LO);    {Shift to LSBs}
              SE=SR0, AR=AR XOR AF;       {Remove segment bits}
              AY0=H#FF80;
              AF=AR+AY0;
              IF LT JUMP posval;          {Detemine sign}
              AR=PASS AF;
              AR=AR+AF;                   {Shift left by 1 bit}
              AY0=33;
              AR=AR+AY0;                  {Add segment offset}
              SR=ASHIFT AR (LO);          {Position bits}
              AR=AY0-SR0;                 {Remove segment offset}
              RTS;
```

```
posval:       AF=PASS AR;
              AR=AR+AF;                    {Shift left by 1}
              AY0=33;
              AR=AR+AY0;                   {Add segment offset}
              SR=ASHIFT AR (LO);
              AR=SR0-AY0;                  {Remove segment offset}
              RTS;

predict:      AX1=DM(I0,M2), AY1=PM(I4,M6);      {Point to dq6 and b6}
              AF=PASS 0, SI=PM(I4,M4);    {AF hold partial sum}
              AY0=DM(I6,M5);
              MX1=3;                       {This multiply will give the}
              MY1=32768;                   {+48>>4 term}
              SR=ASHIFT SI BY -2 (HI);     {Downshift b6 per spec}
              CNTR=6;                      {Loop once for each b}
              DO sez_cmp UNTIL CE;
                 AR=ABS SR1, SR1=DM(I6,M5);      {Get absolute value of b}
                 AR=AR AND AY0, AY0=DM(I6,M5);   {Mask bits per spec}
                 SE=EXP AR (HI), MY0=DM(I0,M2);  {Find exponent adjust}
                 AX0=SE, SR=SR OR NORM AR (HI);  {Compute bnMANT}
                 AR=SR1 AND AY0, AY0=DM(I0,M2);  {Mask bits per spec}
                 MR=AR*MY0 (SS), AX1=DM(I0,M2), AY1=PM(I4,M6);
                 AR=AX0+AY0, AY0=DM(I6,M5);      {Compute WbnEXP}
                 SE=AR, MR=MR+MX1*MY1 (SU);      {Compute WbnMANT}
                 SR=LSHIFT MR1 (HI), SE=DM(I6,M5);{Compute Wbn}
                 AR=SR1 AND AY0, SI=PM(I4,M4);   {Mask Wbn per spec}
                 AX0=AR, AR=AX1 XOR AY1;         {Determine sign of Wbn}
                 AR=AX0, SR=ASHIFT SI (HI);      {Downshift b(n-1) per spec}
                 IF LT AR=-AX0;                  {Negate Wbn if necessary}
sez_cmp:         AF=AR+AF, AY0=DM(I6,M5);        {Add Wbn to partial sum}
              AR=PASS AF, AX1=DM(I0,M1), AY1=PM(I5,M6);  {Get sezi}
              SR=ASHIFT AR BY -1 (HI);           {Downshift to produce sez}
              DM(sez)=SR1;
              SI=PM(I5,M4);                {Get a2}
              SR=ASHIFT SI (HI);           {Downshift a2 per spec}
              AX1=DM(I1,M2), AY1=PM(I4,M5);      {Restore bn and dqn pointers}
              CNTR=2;                     {Loop once for each a}
```

(listing continues on next page)

```
                DO s_e_cmp UNTIL CE;
                   AR=ABS SR1, SR1=DM(I6,M5);       {Get absolute value of a}
                   AR=AR AND AY0, AY0=DM(I6,M5);    {Mask bits per spec}
                   SE=EXP AR (HI), MY0=DM(I1,M2);   {Get exponent adjust for a}
                   AX0=SE, SR=SR OR NORM AR (HI);   {Compute WanMANT}
                   AR=SR1 AND AY0, AY0=DM(I1,M2);   {Mask bits per spec}
                   MR=AR*MY0(SS), AX1=DM(I1,M2), AY1=PM(I5,M6);
                   AR=AX0+AY0, AY0=DM(I6,M5);       {Compute WanEXP}
                   SE=AR, MR=MR+MX1*MY1 (SU);       {Complete WanMANT computation}
                   SR=LSHIFT MR1 (HI), SE=DM(I6,M5);        {Compute Wan}
                   AR=SR1 AND AY0, SI=PM(I5,M4);    {Mask Wan per spec}
                   AX0=AR, AR=AX1 XOR AY1;          {Determine sign of Wan}
                   AR=AX0, SR=ASHIFT SI (HI);       {Downshift a1 per spec}
                   IF LT AR=-AX0;                   {Negate Wan if necessary}
s_e_cmp:           AF=AR+AF, AY0=DM(I6,M5);         {Add Wan to partial sum}
                AR=PASS AF, AX1=DM(I1,M1), AY1=PM(I5,M5);   {Get sei}
                SR=ASHIFT AR BY -1 (HI);            {Compute se}
                DM(s_e)=SR1;
                RTS;

lima:           AY1=256;                    {Maximum value for ap}
                AR=AX1, AF=AX1-AY1;         {Cap if it exceeds}
                IF GE AR=PASS AY1;
                SR=ASHIFT AR BY -2 (HI);    {>>2 to produce al}
                SR=LSHIFT SR1 BY 9 (HI);    {Adjust for ADSP-210x version}

mix:            MY0=SR1, AR=AX0-AY0;        {MY0=al, AR=diff}
                AR=ABS AR;                  {Take absolute value of diff}
                MR=AR*MY0 (SU);             {Generate prod}
                AR=MR1+AY0;                 {Add to yu}
                IF NEG AR=AY0-MR1;          {Subtract if diff < 0}
                SR=ASHIFT AR BY -2 (HI);    {Generate y>>2}
                RTS;
```

```
log:        I3=^qn_values;              {Point to data array}
            AR=ABS AR, AX1=DM(I3,M1);   {Take absolute of d}
            SE=EXP AR (HI), AX0=DM(I3,M1);     {Determine exponent adjust}
            AY0=SE, SR=NORM AR (HI);    {Normalize}
            AR=AX0+AY0, AY0=DM(I3,M1);  {Compute exponent}
            IF LT AR=PASS 0;            {Check for exponent -1}
            SI=AR, AR=SR1 AND AY0;      {Mask mantissa bits}
            SR=LSHIFT AR BY -7 (HI);    {Position mantissa}
            SR=SR OR LSHIFT SI BY 7 (HI);      {Position exponent}

subtb:      AR=SR1-AY1, AY0=DM(I3,M1);  {Subtract y>>2 for log}
            AX0=AR, AF=PASS AX1;        {Setup for quantizing}

quan:       AR=AX0-AY0, AY0=DM(I3,M1);  {Is dl less then upper limit?}
            IF LT AF=AF-1;
            AR=AX0-AY0, AY0=DM(I3,M1);  {Continue to check for }
            IF LT AF=AF-1;
            AR=AX0-AY0, AY0=DM(I3,M1);  {where dl fits in quantizer}
            IF LT AF=AF-1;
            AR=AX0-AY0, AY0=DM(I3,M1);
            IF LT AF=AF-1;
            AR=AX0-AY0, AY0=DM(I3,M1);
            IF LT AF=AF-1;
            AR=AX0-AY0, AY0=DM(I3,M1);
            IF LT AF=AF-1;
            AR=AX0-AY0;
            IF LT AF=AF-1;
            AR=PASS AF;
            IF NEG AR=NOT AF;           {Negate value if ds negative}
            IF EQ AR=NOT AR;            {Send 15 for 0}
            RTS;
```

(listing continues on next page)

```
reconst:    AF=ABS AR;
            IF NEG AR=NOT AR;          {Find absolute value}
            M3=AR;                     {Use this for table lookup}
            I3=^dq_values;             {Point to dq table}
            MODIFY(I3,M3);             {Set pointer to proper spot}
            AX1=DM(I3,M1);             {Read dq from table}

adda:       AR=AX1+AY1;                {Add y>>2 to dq}

antilog:    SR=ASHIFT AR BY 9 (LO);    {Get antilog of dq}
            AY1=127;                   {Mask mantisa}
            AX0=SR1, AR=AR AND AY1;    {Save sign of DQ+Y in AX0}
            AY1=128;                   {Add 1 to mantissa}
            AR=AR+AY1;
            AY0=-7;                    {Compute magnitude of shift}
            SI=AR, AR=SR1+AY0;
            SE=AR;
            SR=ASHIFT SI (HI);         {Shift mantissa }
            AR=SR1, AF=PASS AX0;
            IF LT AR=PASS 0;           {If DQ+Y <0, set to zero}
            IF NEG AR=-SR1;            {Negate DQ if I value negative}
            RTS;

trans:      SR=ASHIFT SI BY -9 (HI);   {Get integer of yl}
            SE=SR1;                    {Save for shift}
            SR=LSHIFT SR0 BY -11 (HI); {Get 5 MSBs of fraction of yl}
            AY0=32;
            AR=SR1+AY0;                {Add one to fractional part}
            AX0=SE, SR=LSHIFT AR (HI); {Shift into proper format}
            AY0=8;
            AR=H#3E00;                 {Maximum value}
            AF=AX0-AY0;
            IF LE AR=PASS SR1;         {Cap at maximum value}
            AF=ABS AX1, AY0=AR;        {Get absolute value of dq}
            SR=LSHIFT AR BY -1 (HI);
            AR=SR1+AY0;
            SR=LSHIFT AR BY -1 (HI);
            AF=SR1-AF, AR=MR1;         {tdp must be set for tr true}
            IF GE AR=PASS 0;           {If dq exceeds threshold no tr}
            RTS;
```

```
functw:     I3=^w_values;              {Get scale factor multiplier}
            MODIFY(I3,M3);             {Based on I value}
            AF=PASS 0, SI=DM(I3,M1);
            I3=^f_values;

filtd:      SR=ASHIFT SI BY 1 (LO);    {Update fast quantizer factor}
            AR=SR0-AY0, SE=DM(I3,M1);  {Compute difference}
            SI=AR, AR=SR1-AF+C-1;      {in double precision}
            SR=ASHIFT AR (HI), AX0=DM(I3,M1); {Time constant is 1/32}
            SR=SR OR LSHIFT SI (LO), AY1=DM(I3,M1);
            AR=SR0+AY0, AY0=DM(I3,M1);         {Add gain}

limb:       AF=AR-AY1, SI=DM(I3,M3);   {Limit fast scale factor}
            IF GT AR=PASS AY1;         {Upper limit 10}
            AF=AR-AY0, AY1=MY1;
            IF LT AR=PASS AY0;         {Lower limit 1.06}

filte:      AF=AX0-AY1, AY0=MY0;       {Update quantizer slow factor}
            AF=AX0-AY0+C-1, AX0=DM(I3,M1);     {Compute difference}
            AX1=AR, AR=AR+AF;
            SR=ASHIFT AR BY -6 (HI);   {Time constant is 1/64}
            AR=SR0+AY1, AY1=MX0;       {Add gain}
            MY1=AR, AR=SR1+AY0+C;      {in double precision}

filta:      MY0=AR, AR=AX0-AY1;        {Update short term I average}
            SR=ASHIFT AR (HI), SI=AX0; {Time constant is 1/32}
            AR=SR1+AY1, AY0=MX1;       {Add gain}

filtb:      SR=LSHIFT SI BY 2 (HI);    {Update long term I average}
            MX0=AR, AR=SR1-AY0;
            SR=ASHIFT AR BY -7 (HI);   {Time constant is 1/128}
            AR=SR1+AY0, SI=MX0;        {Add gain}

subtc:      SR=ASHIFT AR BY -3 (HI);   {Compute difference of long}
            AF=PASS AR, AX0=SR1;       {and short term I averages}
            SR=ASHIFT SI BY 2 (HI);
            MX1=AR, AR=SR1-AF;
            AF=ABS AR;
            AR=MR2, AF=AX0-AF;         {tdp must be true for ax 0}
            IF LE AR=PASS 1;
            AY0=1536;
            AF=MR1-AY0, AY0=MR0;
            IF LT AR=PASS 1;           {Y>3 for ax to be 0}
```

(listing continues on next page)

```
filtc:          SR=ASHIFT AR BY 9 (HI);     {Update speed control}
                AR=SR1-AY0;                 {Compute difference}
                SR=ASHIFT AR BY -4 (HI);    {Time constant is 1/16}
                AR=SR1+AY0;                 {Add gain}
                RTS;

trigger_true:   CNTR=6;                     {Only called when trigger true}
                AX0=0;
                DO trigger UNTIL CE;
trigger:           PM(I4,M5)=AX0;           {Set all b-coefficients to 0}
                AX1=DM(dq);

                DM(tdp)=AX0;                {Set tdp to 0}

                PM(I5,M5)=AX0;              {Set a2 to 0}
                PM(I5,M5)=AX0;              {Set a1 to 0}

                AR=ABS AX1;                 {Add dq to delay line}
                SE=EXP AR (HI);
                AX0=SE, SR=NORM AR (HI);
                SR=LSHIFT SR1 BY -9 (HI);
                AY0=11;
                AY1=32;
                AR=SR1 OR AY1;
                AY1=DM(a_ik);
                SR=LSHIFT AR BY 2 (HI);
                AR=AX0+AY0, DM(I0,M1)=AY1;
                DM(I0,M1)=AR;
                DM(I0,M1)=SR1;

                AY0=DM(sez);                {Compute new p values}
                AR=AX1+AY0;
                AX0=DM(p);
                AY0=DM(p_o);
                DM(p)=AR;
                DM(p_o)=AX0;
                DM(p_o_o)=AY0;

                AR=256;
                DM(ap)=AR;                  {Set ap to triggered value}

                RTS;
```

```
update_filter: AX0=DM(dq);                {Get value of current dq}
               AR=128;
               AF=PASS AX0, AY1=DM(I0,M0); {Read sign of dq(6)}
               IF EQ AR=PASS 0;           {If dq 0 then gain 0}
               SE=-8;                     {Time constant is 1/256}
               AX1=AR;
               CNTR=6;
               DO update_b UNTIL CE;      {Update all b-coefficients}
                  AF=AX0 XOR AY1, AY0=PM(I4,M4);  {Get sign of update}
                  IF LT AR=-AX1;
                  AF=AR+AY0, SI=AY0;      {Add update to original b}
                  SR=ASHIFT SI (HI), AY1=DM(I0,M0); {Get next dq(k)}
                   AR=AF-SR1;                      {Subtract leak factor}
update_b:         PM(I4,M5)=AR, AR=PASS AX1;      {Write out new b-coefficient}

place_dq:      AR=ABS AX0, AY0=DM(I0,M2);  {Take absolute value of dq}
               SE=EXP AR (HI);             {Determine exponent adjust}
               SR1=H#4000;                 {Set minimum value into SR1}
               AX1=SE, SR=SR OR NORM AR (HI);     {Normalize dq}
               AY0=11;                     {Used for exponent adjustment}
               SR=LSHIFT SR1 BY -9 (HI);   {Remove lower bits}
               SR=LSHIFT SR1 BY 2 (HI);    {Adjust for ADSP-210x version}
               DM(I0,M2)=SR1, AR=AX1+AY0;  {Save mantisa, compute exp.}
               DM(I0,M2)=AR;               {Save exponent}
               AX1=DM(a_ik);               {Use sign of I, not dq}
               DM(I0,M0)=AX1;              {Save sign}

update_p:      AY0=DM(sez);                {Get result of predictor}
               AR=AX0+AY0;                 {Use dq from above}
               AY1=DM(p);                  {Delay all old p's by 1}
               AY0=DM(p_o);
               DM(p)=AR;
               DM(p_o)=AY1;
               DM(p_o_o)=AY0;
               AX1=AR, AR=AR XOR AY0;      {Compute p xor poo}
               MR1=AR, AR=AX1 XOR AY1;     {Compute p xor po}
               MR0=AR;
```

(listing continues on next page)

```
upa2:       I3=^a_data;
            SI=PM(I5,M5);               {Hold a2 for later}
            AR=PM(I5,M5);               {Get a1 for computation of f}
            AR=ABS AR, AY0=DM(I3,M1);   {Cap magnitude of a1 at 1/2}
            AF=AR-AY0, SE=DM(I3,M1);
            IF GT AR=PASS AY0;
            IF NEG AR=-AR;              {Restore sign}
            SR=ASHIFT AR (LO), AY0=DM(I3,M1);
            AF=ABS MR0, AY1=DM(I3,M1);  {If p xor po = 0 negate f}
            AR=SR0, AF=PASS SR1;
            IF POS AR=AY1-SR0;          {Double precision}
            IF POS AF=AY1-SR1+C-1;
            SR0=AR, AR=PASS AF;
            SR1=AR, AF=ABS MR1;         {If p xor poo = 1 subtract}
            AR=SR0+AY0, SE=DM(I3,M1);
            AF=SR1+AY1+C, AX0=DM(I3,M1);
            IF NEG AR=SR0-AY0;
            IF NEG AF=SR1-AY1+C-1;
            SR=LSHIFT AR (LO);
            AR=PASS AF;
            SR=SR OR ASHIFT AR (HI), AY0=SI;
            AY1=SR0, SR=ASHIFT SI (HI); {Downshift a2 for adjustment}
            AR=AY0-SR1, AY0=DM(I3,M1);
            AF=PASS AX1;
            IF NE AR=AR+AY1;            {If sigpk = 1, no gain}

limc:       AF=AR-AY0, AY1=DM(I3,M1);   {Limit a2 to .75 max}
            IF GT AR=PASS AY0;
            AF=AR-AY1, AY0=DM(I3,M1);   {Limit a2 to -.75 min}
            IF LT AR=PASS AY1;
            PM(I5,M5)=AR;               {Store new a2}

tone:       AF=AR-AY0, AY1=AR;          {If a2 < .71, tone = 1}
            AR=0;
            IF LT AR=PASS 1;
            DM(tdp)=AR;                 {Store new tdp value (for ap)}
```

```
upa1:           AR=AX0, AF=PASS MR0;
                IF LT AR=-AX0;
                AF=PASS AX1, SI=PM(I5,M4);
                IF EQ AR=PASS 0;
                SR=ASHIFT SI BY -8 (HI);    {Leak Factor = 1/256}
                AF=PASS AR, AR=SI;
                AF=AF-SR1;
                AR=AR+AF, AX1=DM(I3,M1);

limd:           AX0=AR, AR=AX1-AY1;         {Limit a1 based on a2}
                AY0=AR, AR=AY1-AX1;
                AY1=AR, AR=PASS AX0;
                AF=AR-AY0;
                IF GT AR=PASS AY0;          {Upper limit 1 - 2^-4 - a2}
                AF=AR-AY1;
                IF LT AR=PASS AY1;          {Lower limit a2 - 1 + 2^-4}
                PM(I5,M5)=AR;               {Store new a1}

                RTS;

trigger_true_r: CNTR=6;                     {Here only if trigger true}
                AX0=0;
                DO trigger_r UNTIL CE;
trigger_r:      PM(I4,M5)=AX0;              {Set all b-coefficients to 0}
                AX1=DM(dq_r);

                DM(tdp_r)=AX0;              {Set tdp to 0}

                PM(I5,M5)=AX0;              {Set a2 to 0}
                PM(I5,M5)=AX0;              {Set a1 to 0}

                AR=ABS AX1;                 {Add dq_r to delay line}
                SE=EXP AR (HI);
                AX0=SE, SR=NORM AR (HI);
                SR=LSHIFT SR1 BY -9 (HI);
                AY0=11;
                AY1=32;
                AR=SR1 OR AY1;
                AY1=DM(a_ik_r);
                SR=LSHIFT AR BY 2 (HI);
                AR=AX0+AY0, DM(I0,M1)=AY1;
                DM(I0,M1)=AR;
                DM(I0,M1)=SR1;
```

(listing continues on next page)

```
                AY0=DM(sez);                {Compute new p_r's}
                AR=AX1+AY0;
                AX0=DM(p_r);
                AY0=DM(p_o_r);
                DM(p_r)=AR;
                DM(p_o_r)=AX0;
                DM(p_o_o_r)=AY0;

                AR=256;
                DM(ap_r)=AR;                {Set ap_r to triggered value}

                RTS;

update_filter_r: AX0=DM(dq_r);              {Get dq_r}
                AR=128;                     {Set possible gain}
                AF=PASS AX0, AY1=DM(I0,M0); {Get sign of dq(6)}
                IF EQ AR=PASS 0;            {If dq_r 0, gain 0}
                SE=-8;                      {Leak factor 1/256}
                AX1=AR;
                CNTR=6;
                DO update_b_r UNTIL CE;     {Update all b-coefficients}
                   AF=AX0 XOR AY1, AY0=PM(I4,M4);  {Get sign of gain}
                   IF LT AR=-AX1;
                   AF=AR+AY0, SI=AY0;               {Add gain to original b}
                   SR=ASHIFT SI (HI);               {Time constant is 1/256}
                   AR=AF-SR1, AY1=DM(I0,M0);        {Compute new b-value}
update_b_r:        PM(I4,M5)=AR, AR=PASS AX1;       {Store new b-value}

place_dq_r:     AR=ABS AX0, AY0=DM(I0,M2);          {Get absolute value fo dq_r}
                SE=EXP AR (HI);             {Determine exponent adjustment}
                SR1=H#4000;                 {Set SR to minimum value}
                AX1=SE, SR=SR OR NORM AR (HI);      {Normalize dq_r}
                AY0=11;                     {Used for exponent adjust}
                SR=LSHIFT SR1 BY -9 (HI);   {Remove lower bits}
                SR=LSHIFT SR1 BY 2 (HI);    {Adjust for ADSP-210x version}
                DM(I0,M2)=SR1, AR=AX1+AY0;  {Store mantissa, compute exp}
                AX1=DM(a_ik_r);             {Use sign of I, not dq}
                DM(I0,M2)=AR;               {Store exponent}
                DM(I0,M0)=AX1;              {Store sign}
```

```
update_p_r:    AY0=DM(sez);                {Compute new p}
               AR=AX0+AY0;                 {Use dq_r from above}
               AY1=DM(p_r);                {Delay old p's by 1}
               AY0=DM(p_o_r);
               DM(p_r)=AR;
               DM(p_o_r)=AY1;
               DM(p_o_o_r)=AY0;
               AX1=AR, AR=AR XOR AY0;      {Compute p and poo}
               MR1=AR, AR=AX1 XOR AY1;     {Compute p and po}
               MR0=AR;

upa2_r:        I3=^a_data;
               SI=PM(I5,M5);               {Hold a2 for later}
               AR=PM(I5,M5);               {Get a1 for computation of f}
               AR=ABS AR, AY0=DM(I3,M1);   {Cap magnitude of a1 to 1/2}
               AF=AR-AY0, SE=DM(I3,M1);
               IF GT AR=PASS AY0;
               IF NEG AR=-AR;              {Restore sign of f}
               SR=ASHIFT AR (LO), AY0=DM(I3,M1);
               AF=ABS MR0, AY1=DM(I3,M1);  {If p_r xor poo_r =1 subtract}
               AR=SR0, AF=PASS SR1;
               IF POS AR=AY1-SR0;
               IF POS AF=AY1-SR1+C-1;
               SR0=AR, AR=PASS AF;
               SR1=AR, AF=ABS MR1;
               AR=SR0+AY0, SE=DM(I3,M1);
               AF=SR1+AY1+C, AX0=DM(I3,M1);
               IF NEG AR=SR0-AY0;
               IF NEG AF=SR1-AY1+C-1;
               SR=LSHIFT AR (LO);
               AR=PASS AF;
               SR=SR OR ASHIFT AR (HI), AY0=SI;
               AY1=SR0, SR=ASHIFT SI (HI); {Leak factor of 1/128}
               AR=AY0-SR1, AY0=DM(I3,M1);
               AF=PASS AX1;
               IF NE AR=AR+AY1;            {If sigpk = 1 , no gain}

limc_r:        AF=AR-AY0, AY1=DM(I3,M1);   {Limit a2 to .75 max}
               IF GT AR=PASS AY0;
               AF=AR-AY1, AY0=DM(I3,M1);   {Limit a2 to -.75 min}
               IF LT AR=PASS AY1;
               PM(I5,M5)=AR;               {Store new a2}
```

(listing continues on next page)

```
tone_r:     AF=AR-AY0, AY1=AR;
            AR=0;
            IF LT AR=PASS 1;          {If a2 < .71, tdp = 1}
            DM(tdp_r)=AR;

upa1_r:     AR=AX0, AF=PASS MR0;
            IF LT AR=-AX0;
            AF=PASS AX1, SI=PM(I5,M4);
            IF EQ AR=PASS 0;
            SR=ASHIFT SI BY -8 (HI);  {Leak Factor = 1/256}
            AF=PASS AR, AR=SI;
            AF=AF-SR1;
            AR=AR+AF, AX1=DM(I3,M1);

limd_r:     AX0=AR, AR=AX1-AY1;       {Limit a1 based on a2}
            AY0=AR, AR=AY1-AX1;
            AY1=AR, AR=PASS AX0;
            AF=AR-AY0;
            IF GT AR=PASS AY0;        {Upper limit 1 - 2^-4 -a2}
            AF=AR-AY1;
            IF LT AR=PASS AY1;        {Lower limit a2 - 1 + 2^-4}
            PM(I5,M5)=AR;             {Store new a1}

            RTS;

sync:       AX0=DM(a_ik_r);           {Get input value of I}
            AY1=AR, AF=ABS AR;
            IF NEG AR=AY1+1;          {Convert 1's comp to 2's comp}
            AY1=AX0, AF=ABS AX0;
            IF NEG AF=AY1+1;          {Same for new I value }
            AR=AR-AF;
            AR=DM(sp_r);
            IF GT JUMP decrement_sp;  {Next most negative value}
            IF LT JUMP increment_sp;  {Next most positive value}
            AF=PASS AX0;              {Check for invalid 0 input}
            IF NE RTS;
```

```
increment_sp:  SR=LSHIFT AR BY 8 (HI);     {Get sign of PCM value}
               AY0=H#80;
               AF=AR-AY0;
               IF EQ RTS;                  {Already maximum value}
               AF=ABS SR1;
               AF=PASS 1;
               IF NEG AF=PASS -1;          {If negative, subtract 1}
               AR=AR+AF;                   {Compute adjusted PCM value}
               RTS;

decrement_sp:  SR=LSHIFT AR BY 8 (HI);     {Get sign of PCM value}
               AR=PASS AR;
               IF EQ RTS;                  {Already minimum value}
               AY0=H#FF;
               AF=AR-AY0;
               IF NE JUMP no_sign_chn;     {If input is H#FF}
               AR=H#7E;                    {New output will be h#7E}
               RTS;

no_sign_chn:   AF=ABS SR1;                 {Otherwise adjust by 1}
               AF=PASS -1;
               IF NEG AF=PASS 1;           {Add 1 for negative values}
               AR=AR+AF;                   {Compute adjusted PCM value}
               RTS;

.ENDMOD;
```

Listing 12.2 Standard ADPCM Transcoder Routine

12.9.2 Nonstandard ADPCM Transcoder Listing

The code below represents a full-duplex ADPCM transcoder. Although developed in accordance with ANSI specification T1.301-1987 and CCITT G.721 (bis), it has been modified to improve its speed. The modifications include the removal of the synchronous coding adjustment and the tone and transition detectors. These deletions do not noticeably affect speech-only coding.

```
.MODULE       Adaptive_Differential_PCM;

{     Calling Parameters
              AR = Companded PCM value (encoder)
                     or ADPCM I value (decoder)

              M0=3;  L0=18;
              M1=1;  L1=6;
              M2=-1
                     L3=0;
              M4=0   L4=6;
              M5=1   L5=2
              M6=-1  L6=5

       Return Values
              AR = ADPCM I value (encoder)
                     or Companded PCM value (decoder)

       Altered Registers
              AX0, AX1, AY0, AY1, AF, AR,
              MX0, MX1, MY0, MY1, MR,
              I0, I1, I3, I4, I5, I6
              SI, SR
              M3

       Cycle Count
              437 cycles for encode
              409 cycles for decode

}
```

```
.ENTRY        ns_adpcm_encode, ns_adpcm_decode;

.VAR/PM/CIRC  b_buf[6];                  {b coefficients for encode}
.VAR/PM/CIRC  a_buf[2];                  {a coefficients for encode}
.VAR/PM/CIRC  b_buf_r[6];                {b coefficients for decode}
.VAR/PM/CIRC  a_buf_r[2];                {a coefficients for decode}

.VAR/DM/CIRC  b_delay_r[18];             {dq delay for decode}
.VAR/DM/CIRC  a_delay_r[6];              {sr delay for decode}
.VAR/DM/CIRC  b_delay[18];               {dq delay for encode}
.VAR/DM/CIRC  a_delay[6];                {sr delay for encode}

.VAR/DM/CIRC  mult_data[5];              {Predictor immediate data}

.VAR/DM       qn_values[10],dq_values[8]; {quantizer & dequantizer data}
.VAR/DM       f_values[12], w_values[8];  {Update coefficient data}
.VAR/DM       a_data[10];

.VAR/DM       s_e,s_r,a_ik,dq,p;
.VAR/DM       sez,sl,yu,yl_h,yl_l,y,y_2,ap,p_o,p_o_o,dms,dml,tdp,tr;
.VAR/DM       a_ik_r,dq_r,p_r;
.VAR/DM       yu_r,yl_h_r,yl_l_r,ap_r,p_o_r;
.VAR/DM       p_o_o_r,dms_r,dml_r,tdp_r;

.VAR/DM       sp_r;                      {PCM code word for synchronous adj}

.VAR/DM       hld_a_t, hld_b_t, hld_a_r, hld_b_r;

.INIT  qn_values:    7, 14, H#3F80, 400, 349, 300, 246, 178, 80, H#FF84;
.INIT  dq_values :   h#F800, 4, 135, 213, 273, 323, 373, 425;
.INIT  f_values :    -5, 0, 5120, 544, 0, 0, 0, 512, 512, 512, 1536, 3584;
.INIT  w_values:     65344, 288, 656, 1024, 1792, 3168, 5680, 17952;
.INIT  mult_data :   H#1FFF, H#4000, h#7E00, H#7FFF, H#FFFE;
.INIT  a_data :      H#1FFF, 2, 16384, 0, -7, 192, H#3000, H#D000,
                     H#D200, H#3C00;

.INIT         hld_a_t : ^a_delay;
.INIT         hld_b_t : ^b_delay;
.INIT         hld_a_r : ^a_delay_r;
.INIT         hld_b_r : ^b_delay_r;
```

(listing continues on next page)

12 ADPCM

```
.INIT         b_buf : 0,0,0,0,0,0;                {2.14}
.INIT         a_buf : 0,0;                        {2.14}
.INIT         b_delay : 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
                                                  {16.0, 16.0, 0.16}
.INIT         a_delay : 0,0,0,0,0,0;              {16.0, 16.0, 0.16}
.INIT         p : 0;                              {16.0}
.INIT         yu :0;                              {7.9}
.INIT         yl_h : 0;                           {7.9}
.INIT         yl_l : 0;                           {0.16}
.INIT         ap : 0;                             {8.8}
.INIT         p_o : 0;                            {16.0}
.INIT         p_o_o : 0;                          {16.0}
.INIT         dms : 0;                            {7.9}
.INIT         dml : 0;                            {5.11}
.INIT         tdp : 0;                            {16.0}

.INIT         b_buf_r : 0,0,0,0,0,0;              {2.14}
.INIT         a_buf_r : 0,0;                      {2.14}
.INIT         b_delay_r : 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
                                                  {16.0, 16.0, 0.16}
.INIT         a_delay_r : 0,0,0,0,0,0;            {16.0, 16.0, 0.16}
.INIT         p_r : 0;                            {16.0}
.INIT         yu_r :0;                            {7.9}
.INIT         yl_h_r : 0;                         {7.9}
.INIT         yl_l_r : 0;                         {0.16}
.INIT         ap_r : 0;                           {8.8}
.INIT         p_o_r : 0;                          {16.0}
.INIT         p_o_o_r : 0;                        {16.0}
.INIT         dms_r : 0;                          {7.9}
.INIT         dml_r : 0;                          {5.11}
.INIT         tdp_r : 0;                          {16.0}

ns_adpcm_encode: I4=^b_buf;                       {Set pointer to b-coefficients}
              I5=^a_buf;                          {Set pointer to a-coefficients}
              I6=^mult_data;                      {Set pointer to predictor data}
              I1=DM(hld_a_t);                     {Restore pointer to s_r delay}
              I0=DM(hld_b_t);                     {Restore pointer to dq delay}

              CALL expand;                        {Expand 8-bit log-PCM to12 bits}
              DM(sl)=AR;                          {Store linear PCM value in sl}
              CALL predict;                       {Call s_e and sez predictors}
```

```
AX1=DM(ap);
AY0=DM(yl_h);
AX0=DM(yu);
CALL lima;                          {Limit ap and compute y}
DM(y)=AR;                           {Save y for later updates}
DM(y_2)=SR1;                        {Save y>>2 for log and reconst}
AX0=DM(sl);
AY0=DM(s_e);
AY1=SR1, AR=AX0-AY0;                {Compute difference signal, d}
CALL log;                           {Determine I value from d}
DM(a_ik)=AR;
CALL reconst;                       {Compute dq based ONLY on }
DM(dq)=AR;
AY0=DM(s_e);
AR=AR+AY0;                          {Compute reconstructed signal}
DM(s_r)=AR;

DM(I1,M1)=AR, AR=ABS AR;            {Convert s_r to floating point}
SR1=H#4000;                         {Set SR1 to minimum value}
SE=EXP AR (HI);                     {Determine exponent adjust}
AX0=SE, SR=SR OR NORM AR (HI);      {Normalize into SR}
SR=LSHIFT SR1 BY -9 (HI);           {Delete lower bits}
AY0=11;                             {Base exponent}
SR=LSHIFT SR1 BY 2 (HI);            {Adjust for ADSP-210x version}
AR=AX0+AY0;                         {Compute exponent}
DM(I1,M1)=AR;                       {Save exponent}
DM(I1,M1)=SR1;                      {Save mantissa}

CALL update_filter;                 {Update filter if trigger false}

MR0=DM(ap);                         {Load variables for updating}
MR1=DM(y);
MR2=DM(tdp);                        {Always load MR2 after MR1!}
MY0=DM(yl_h);
MY1=DM(yl_l);
AY0=DM(y);
MX0=DM(dms);
MX1=DM(dml);
CALL functw;                        {Update variables}
DM(ap)=AR;                          {Store updated variables}
DM(yu)=AX1;
DM(yl_l)=MY1;
DM(yl_h)=MY0;
```

(listing continues on next page)

```
                DM(dms)=MX0;
                DM(dml)=MX1;

                AX0=DM(a_ik);                   {Get I value for return}
                AY0=H#F;                        {Only 4 LSBs are used}
                AR=AX0 AND AY0;                 {So mask redundant sign bits}

                DM(hld_a_t)=I1;                 {Save s_r delay pointer}
                DM(hld_b_t)=I0;                 {Save dq delay pointer}
                RTS;                            {Return to caller}

ns_adpcm_decode: I1=DM(hld_a_r);                {Restore s_r delay pointer}
                I0=DM(hld_b_r);                 {Restore dq delay pointer}
                I4=^b_buf_r;                    {Set pointer to b-coefficients}
                I5=^a_buf_r;                    {Set pointer to a-coefficients}
                I6=^mult_data;                  {Set pointer to predictor data}

                SR=LSHIFT AR BY 12 (HI);        {Get sign of ADPCM I value here}
                SR=ASHIFT SR1 BY -12 (HI);
                                                {Sign extend ADPCM value to 16}
                DM(a_ik_r)=SR1;                 {Save I value}
                CALL predict;                   {Call s_e and sez predictor}
                AX1=DM(ap_r);
                AY0=DM(yl_h_r);
                AX0=DM(yu_r);
                CALL lima;                      {Limit ap and compute y}
                DM(y)=AR;
                DM(y_2)=SR1;
                AY1=DM(y_2);
                AR=DM(a_ik_r);
                CALL reconst;                   {Compute dq from received I}
                DM(dq_r)=AR;
                AY0=DM(s_e);
                AR=AR+AY0;                      {Compute reconstructed signal}
                DM(s_r)=AR;

                DM(I1,M1)=AR, AR=ABS AR;        {Make s_r floating point}
                SR1=H#4000;                     {Set SR1 to minimum value}
                SE=EXP AR (HI);                 {Determine exponent adjust}
                AX0=SE, SR=SR OR NORM AR (HI);  {Normalize value}
                SR=LSHIFT SR1 BY -9 (HI);       {Remove LSBs per spec}
                AY0=11;                         {Base exponent}
```

```
                SR=LSHIFT SR1 BY 2 (HI);          {Adjust for ADSP-210x version}
                AR=AX0+AY0;                       {Compute exponent}
                DM(I1,M1)=AR;                     {Store exponent}
                DM(I1,M1)=SR1;                    {Store mantissa}

                CALL update_filter_r;             {Update filter if trigger false}

                AY0=DM(y);                        {Load variables for updating}
                MY1=DM(yl_l_r);
                MY0=DM(yl_h_r);
                MR0=DM(ap_r);
                MR1=DM(y);
                MR2=DM(tdp_r);                    {Always load MR2 after MR1!}
                MX0=DM(dms_r);
                MX1=DM(dml_r);
                CALL functw;                      {Update variables}
                DM(yu_r)=AX1;                     {Stored updated variables}
                DM(yl_l_r)=MY1;
                DM(yl_h_r)=MY0;
                DM(ap_r)=AR;
                DM(dms_r)=MX0;
                DM(dml_r)=MX1;

                CALL compress;                    {Compress PCM value}

                DM(hld_a_r)=I1;                   {Save s_r delay pointer}
                DM(hld_b_r)=I0;                   {Save dq delay pointer}
                RTS;

compress:       AR=DM(s_r);               {Get reconstructed signal}
                AR=ABS AR;                {Take absolute value}
                AY0=33;                   {Add offset of boundries}
                AR=AR+AY0;
                AY0=8191;                 {Maximum PCM value}
                AF=AR-AY0;                {Cap input}
                IF GT AR=PASS AY0;        {If in excess}
                SE=EXP AR (HI);           {Find exponent adjustmet}
                AX0=SE, SR=NORM AR (LO);  {Normalize input}
                AY0=H#4000;
                AR=SR0 XOR AY0;           {Remove first significant bit}
                SR=LSHIFT AR BY -10 (LO); {Shift position bits}
                AR=PASS AY0;
                IF POS AR=PASS 0;         {Create sign bit}
```

(listing continues on next page)

```
            SR=SR OR LSHIFT AR BY -7 (LO);     {Position sign bit}
            AY0=9;
            AR=AX0+AY0;                {Compute segment}
            IF LT AR=PASS 0;
            SR=SR OR LSHIFT AR BY 4 (LO);      {Position segment bits}
            AY0=H#FF;
            AR=SR0 XOR AY0;            {Invert bits}
            RTS;

expand:     AY0=H#FF;                  {Mask unwanted bits}
            AF=AR AND AY0, AX0=AY0;
            AF=AX0 XOR AF;             {Invert bits}
            AX0=H#70;
            AR=AX0 AND AF;             {Isolate segment bits}
            SR=LSHIFT AR BY -4 (LO);   {Shift to LSBs}
            SE=SR0, AR=AR XOR AF;      {Remove segment bits}
            AY0=H#FF80;
            AF=AR+AY0;
            IF LT JUMP posval;         {Detemine sign}
            AR=PASS AF;
            AR=AR+AF;                  {Shift left by 1 bit}
            AY0=33;
            AR=AR+AY0;                 {Add segment offset}
            SR=ASHIFT AR (LO);         {Position bits}
            AR=AY0-SR0;                {Remove segment offset}
            RTS;

posval:     AF=PASS AR;
            AR=AR+AF;                  {Shift left by 1}
            AY0=33;
            AR=AR+AY0;                 {Add segment offset}
            SR=ASHIFT AR (LO);
            AR=SR0-AY0;                {Remove segment offset}
            RTS;

predict:    AX1=DM(I0,M2), AY1=PM(I4,M6);      {Point to dq6 and b6}
            AF=PASS 0, SI=PM(I4,M4);   {AF hold partial sum}
            AY0=DM(I6,M5);
            MX1=3;                     {This multiply will give the}
            MY1=32768;                 {+48>>4 term}
            SR=ASHIFT SI BY -2 (HI);   {Downshift b6 per spec}
            CNTR=6;                    {Loop once for each b}
```

```
                DO sez_cmp UNTIL CE;
                   AR=ABS SR1, SR1=DM(I6,M5);       {Get absolute value of b}
                   AR=AR AND AY0, AY0=DM(I6,M5);    {Mask bits per spec}
                   SE=EXP AR (HI), MY0=DM(I0,M2);   {Find exponent adjust}
                   AX0=SE, SR=SR OR NORM AR (HI);   {Compute bnMANT}
                   AR=SR1 AND AY0, AY0=DM(I0,M2);   {Mask bits per spec}
                   MR=AR*MY0 (SS), AX1=DM(I0,M2), AY1=PM(I4,M6);
                   AR=AX0+AY0, AY0=DM(I6,M5);       {Compute WbEXP}
                   SE=AR, MR=MR+MX1*MY1 (SU);       {Compute WbnMANT}
                   SR=LSHIFT MR1 (HI), SE=DM(I6,M5);        {Compute Wbn}
                   AR=SR1 AND AY0, SI=PM(I4,M4);    {Mask Wbn per spec}
                   AX0=AR, AR=AX1 XOR AY1;          {Determine sign of Wbn}
                   AR=AX0, SR=ASHIFT SI (HI);       {Downshift b(n-1) per spec}
                   IF LT AR=-AX0;                   {Negate Wbn if necessary}
sez_cmp:           AF=AR+AF, AY0=DM(I6,M5);         {Add Wbn to partial sum}
                AR=PASS AF, AX1=DM(I0,M1), AY1=PM(I5,M6);   {Get sezi}
                SR=ASHIFT AR BY -1 (HI);            {Downshift to produce sez}
                DM(sez)=SR1;
                SI=PM(I5,M4);                       {Get a2}
                SR=ASHIFT SI (HI);                  {Downshift a2 per spec}
                AX1=DM(I1,M2), AY1=PM(I4,M5);       {Restore bn and dqn pointers}
                CNTR=2;                             {Loop once for each a}
                DO s_e_cmp UNTIL CE;
                   AR=ABS SR1, SR1=DM(I6,M5);       {Get absolute value of a}
                   AR=AR AND AY0, AY0=DM(I6,M5);    {Mask bits per spec}
                   SE=EXP AR (HI), MY0=DM(I1,M2);   {Get exponent adjust for a}
                   AX0=SE, SR=SR OR NORM AR (HI);   {Compute anMANT}
                   AR=SR1 AND AY0, AY0=DM(I1,M2);   {Mask bits per spec}
                   MR=AR*MY0(SS), AX1=DM(I1,M2), AY1=PM(I5,M6);
                   AR=AX0+AY0, AY0=DM(I6,M5);       {Compute WanEXP}
                   SE=AR, MR=MR+MX1*MY1 (SU);       {Complete WanMANT computation}
                   SR=LSHIFT MR1 (HI), SE=DM(I6,M5);        {Compute Wan}
                   AR=SR1 AND AY0, SI=PM(I5,M4);    {Mask Wan per spec}
                   AX0=AR, AR=AX1 XOR AY1;          {Determine sign of Wan}
                   AR=AX0, SR=ASHIFT SI (HI);       {Downshift a1 per spec}
                   IF LT AR=-AX0;                   {Negate Wan if necessary}
s_e_cmp:           AF=AR+AF, AY0=DM(I6,M5);         {Add Wan to partial sum}
                AR=PASS AF, AX1=DM(I1,M1), AY1=PM(I5,M5);   {Get sei}
                SR=ASHIFT AR BY -1 (HI);            {Compute se}
                DM(s_e)=SR1;
                RTS;
```

(listing continues on next page)

```
lima:       AY1=256;                    {Maximum value for ap}
            AR=AX1, AF=AX1-AY1;         {Cap if it exceeds}
            IF GE AR=PASS AY1;
            SR=ASHIFT AR BY -2 (HI);    {>>2 to produce al}
            SR=LSHIFT SR1 BY 9 (HI);    {Adjust for ADSP-210x version}

mix:        MY0=SR1, AR=AX0-AY0;        {MY0=al, AR=diff}
            AR=ABS AR;                  {Take absolute value of diff}
            MR=AR*MY0 (SU);             {Generate prod}
            AR=MR1+AY0;                 {Add to yu}
            IF NEG AR=AY0-MR1;          {Subtract if diff < 0}
            SR=ASHIFT AR BY -2 (HI);    {Generate y>>2}
            RTS;

log:        I3=^qn_values;              {Point to data array}
            AR=ABS AR, AX1=DM(I3,M1);   {Take absolute of d}
            SE=EXP AR (HI), AX0=DM(I3,M1);     {Determine exponent adjust}
            AY0=SE, SR=NORM AR (HI);    {Normalize}
            AR=AX0+AY0, AY0=DM(I3,M1);  {Compute exponent}
            IF LT AR=PASS 0;            {Check for exponent -1}
            SI=AR, AR=SR1 AND AY0;      {Mask mantissa bits}
            SR=LSHIFT AR BY -7 (HI);    {Position mantissa}
            SR=SR OR LSHIFT SI BY 7 (HI);      {Position exponent}

subtb:      AR=SR1-AY1, AY0=DM(I3,M1);  {Subtract y>>2 for log}
            AX0=AR, AF=PASS AX1;        {Setup for quantizing}

quan:       AR=AX0-AY0, AY0=DM(I3,M1);  {Is dl less then upper limit?}
            IF LT AF=AF-1;
            AR=AX0-AY0, AY0=DM(I3,M1);  {Continue to check for}
            IF LT AF=AF-1;
            AR=AX0-AY0, AY0=DM(I3,M1);  {where dl fits in quantizer}
            IF LT AF=AF-1;
            AR=AX0-AY0, AY0=DM(I3,M1);
            IF LT AF=AF-1;
            AR=AX0-AY0, AY0=DM(I3,M1);
            IF LT AF=AF-1;
            AR=AX0-AY0, AY0=DM(I3,M1);
            IF LT AF=AF-1;
            AR=AX0-AY0;
            IF LT AF=AF-1;
            AR=PASS AF;
            IF NEG AR=NOT AF;           {Negate value if ds negative}
            IF EQ AR=NOT AR;            {Send 15 for 0}
            RTS;
```

```
reconst:    AF=ABS AR;
            IF NEG AR=NOT AR;          {Find absolute value}
            M3=AR;                     {Use this for table lookup}
            I3=^dq_values;             {Point to dq table}
            MODIFY(I3,M3);             {Set pointer to proper spot}
            AX1=DM(I3,M1);             {Read dq from table}

adda:       AR=AX1+AY1;                {Add y>>2 to dq}

antilog:    SR=ASHIFT AR BY 9 (LO);    {Get antilog of dq}
            AY1=127;                   {Mask mantisa}
            AX0=SR1, AR=AR AND AY1;    {Save sign of DQ+Y in AX0}
            AY1=128;                   {Add 1 to mantissa}
            AR=AR+AY1;
            AY0=-7;                    {Compute magnitude of shift}
            SI=AR, AR=SR1+AY0;
            SE=AR;
            SR=ASHIFT SI (HI);         {Shift mantissa}
            AR=SR1, AF=PASS AX0;
            IF LT AR=PASS 0;           {If DQ+Y <0, set to zero}
            IF NEG AR=-SR1;            {Negate DQ if I value negative}
            RTS;

functw:     I3=^w_values;              {Get scale factor multiplier}
            MODIFY(I3,M3);             {Based on I value}
            AF=PASS 0, SI=DM(I3,M1);
            I3=^f_values;

filtd:      SR=ASHIFT SI BY 1 (LO);    {Update fast quantizer factor}
            AR=SR0-AY0, SE=DM(I3,M1);  {Compute difference}
            SI=AR, AR=SR1-AF+C-1;      {in double precision}
            SR=ASHIFT AR (HI), AX0=DM(I3,M1); {Time constant is 1/32}
            SR=SR OR LSHIFT SI (LO), AY1=DM(I3,M1);
            AR=SR0+AY0, AY0=DM(I3,M1); {Add gain}

limb:       AF=AR-AY1, SI=DM(I3,M3);   {Limit fast scale factor}
            IF GT AR=PASS AY1;         {Upper limit 10}
            AF=AR-AY0, AY1=MY1;
            IF LT AR=PASS AY0;         {Lower limit 1.06}

filte:      AF=AX0-AY1, AY0=MY0;       {Update quantizer slow factor}
            AF=AX0-AY0+C-1, AX0=DM(I3,M1);     {Compute difference}
            AX1=AR, AR=AR+AF;
            SR=ASHIFT AR BY -6 (HI);   {Time constant is 1/64}
            AR=SR0+AY1, AY1=MX0;       {Add gain}
            MY1=AR, AR=SR1+AY0+C;      {in double precision}
```

(listing continues on next page)

```
filta:          MY0=AR, AR=AX0-AY1;          {Update short term I average}
                SR=ASHIFT AR (HI), SI=AX0;   {Time constant is 1/32}
                AR=SR1+AY1, AY0=MX1;         {Add gain}

filtb:          SR=LSHIFT SI BY 2 (HI);      {Update long term I average}
                MX0=AR, AR=SR1-AY0;
                SR=ASHIFT AR BY -7 (HI);     {Time constant is 1/128}
                AR=SR1+AY0, SI=MX0;          {Add gain}

subtc:          SR=ASHIFT AR BY -3 (HI);     {Compute difference of long}
                AF=PASS AR, AX0=SR1;         {and short term I averages}
                SR=ASHIFT SI BY 2 (HI);
                MX1=AR, AR=SR1-AF;
                AF=ABS AR;
                AR=MR2, AF=AX0-AF;           {tdp must be true for ax 0}
                IF LE AR=PASS 1;
                AY0=1536;
                AF=MR1-AY0, AY0=MR0;
                IF LT AR=PASS 1;             {Y>3 for ax to be 0}
filtc:          SR=ASHIFT AR BY 9 (HI);      {Update speed control}
                AR=SR1-AY0;                  {Compute difference}
                SR=ASHIFT AR BY -4 (HI);     {Time constant is 1/16}
                AR=SR1+AY0;                  {Add gain}
                RTS;

update_filter: AX0=DM(dq);                   {Get value of current dq}
                AR=128;
                AF=PASS AX0, AY1=DM(I0,M0); {Read sign of dq(6)}
                IF EQ AR=PASS 0;             {If dq 0 then gain 0}
                SE=-8;                       {Time constand is 1/256}
                AX1=AR;
                CNTR=6;
                DO update_b UNTIL CE;        {Update all b-coefficients}
                   AF=AX0 XOR AY1, AY0=PM(I4,M4);     {Get sign of update}
                   IF LT AR=-AX1;
                   AF=AR+AY0, SI=AY0;        {Add update to original b}
                   SR=ASHIFT SI (HI), AY1=DM(I0,M0);  {Get next dq(k)}
                   AR=AF-SR1;                {Subtract leak factor}
update_b:          PM(I4,M5)=AR, AR=PASS AX1;         {Write out new b-coefficient}

place_dq:       AR=ABS AX0, AY0=DM(I0,M2);   {Take absolute value of dq}
                SE=EXP AR (HI);              {Determine exponent adjust}
```

```
                SR1=H#4000;                 {Set minimum value into SR1}
                AX1=SE, SR=SR OR NORM AR (HI);       {Normalize dq}
                AY0=11;                     {Used for exponent adjustment}
                SR=LSHIFT SR1 BY -9 (HI);   {Remove lower bits}
                SR=LSHIFT SR1 BY 2 (HI);    {Adjust for ADSP-210x version}
                DM(I0,M2)=SR1, AR=AX1+AY0;  {Save mantisa, compute exp.}
                DM(I0,M2)=AR;               {Save exponent}
                AX1=DM(a_ik);               {Use sign of I, not dq}
                DM(I0,M0)=AX1;              {Save sign}

update_p:       AY0=DM(sez);                {Get result of predictor}
                AR=AX0+AY0;                 {Use dq from above}
                AY1=DM(p);                  {Delay all old p's by 1}
                AY0=DM(p_o);
                DM(p)=AR;
                DM(p_o)=AY1;
                DM(p_o_o)=AY0;
                AX1=AR, AR=AR XOR AY0;      {Compute p xor poo}
                MR1=AR, AR=AX1 XOR AY1;     {Compute p xor po}
                MR0=AR;

upa2:           I3=^a_data;
                SI=PM(I5,M5);               {Hold a2 for later}
                AR=PM(I5,M5);               {Get a1 for computation of f}
                AR=ABS AR, AY0=DM(I3,M1);   {Cap magnitude of a1 at 1/2}
                AF=AR-AY0, SE=DM(I3,M1);
                IF GT AR=PASS AY0;
                IF NEG AR=-AR;              {Restore sign}
                SR=ASHIFT AR (LO), AY0=DM(I3,M1);
                AF=ABS MR0, AY1=DM(I3,M1);  {If p xor po = 0 negate f}
                AR=SR0, AF=PASS SR1;
                IF POS AR=AY1-SR0;          {Double precision}
                IF POS AF=AY1-SR1+C-1;
                SR0=AR, AR=PASS AF;
                SR1=AR, AF=ABS MR1;         {If p xor poo = 1 subtract}
                AR=SR0+AY0, SE=DM(I3,M1);
                AF=SR1+AY1+C, AX0=DM(I3,M1);
                IF NEG AR=SR0-AY0;
                IF NEG AF=SR1-AY1+C-1;
                SR=LSHIFT AR (LO);
                AR=PASS AF;
                SR=SR OR ASHIFT AR (HI), AY0=SI;
                AY1=SR0, SR=ASHIFT SI (HI); {Downshift a2 for adjustment}
```

(listing continues on next page)

```
                AR=AY0-SR1, AY0=DM(I3,M1);
                AF=PASS AX1;
                IF NE AR=AR+AY1;            {If sigpk = 1, no gain}

limc:           AF=AR-AY0, AY1=DM(I3,M1);   {Limit a2 to .75 max}
                IF GT AR=PASS AY0;
                AF=AR-AY1, AY0=DM(I3,M1);   {Limit a2 to -.75 min}
                IF LT AR=PASS AY1;
                PM(I5,M5)=AR;               {Store new a2}

upa1:           AR=AX0, AF=PASS MR0;
                IF LT AR=-AX0;
                AF=PASS AX1, SI=PM(I5,M4);
                IF EQ AR=PASS 0;
                SR=ASHIFT SI BY -8 (HI);    {Leak Factor = 1/256}
                AF=PASS AR, AR=SI;
                AF=AF-SR1;
                AR=AR+AF, AX1=DM(I3,M1);

limd:           AX0=AR, AR=AX1-AY1;         {Limit a1 based on a2}
                AY0=AR, AR=AY1-AX1;
                AY1=AR, AR=PASS AX0;
                AF=AR-AY0;
                IF GT AR=PASS AY0;          {Upper limit 1 - 2^-4 - a2}
                AF=AR-AY1;
                IF LT AR=PASS AY1;          {Lower limit a2 - 1 + 2^-4}
                PM(I5,M5)=AR;               {Store new a1}
                RTS;

update_filter_r:AX0=DM(dq_r);               {Get dq_r}
                AR=128;                     {Set possible gain}
                AF=PASS AX0, AY1=DM(I0,M0); {Get sign of dq(6)}
                IF EQ AR=PASS 0;            {If dq_r 0, gain 0}
                SE=-8;                      {Leak factor 1/256}
                AX1=AR;
                CNTR=6;
                DO update_b_r UNTIL CE;     {Update all b-coefficients}
                   AF=AX0 XOR AY1, AY0=PM(I4,M4);  {Get sign of gain}
                   IF LT AR=-AX1;
                   AF=AR+AY0, SI=AY0;               {Add gain to original b}
                   SR=ASHIFT SI (HI);               {Time constant is 1/256}
                   AR=AF-SR1, AY1=DM(I0,M0);        {Compute new b-value}
update_b_r:        PM(I4,M5)=AR, AR=PASS AX1;       {Store new b-value}
```

```
place_dq_r:   AR=ABS AX0, AY0=DM(I0,M2);  {Get absolute value fo dq_r}
              SE=EXP AR (HI);             {Determine exponent adjustment}
              SR1=H#4000;                 {Set SR to minimum value}
              AX1=SE, SR=SR OR NORM AR (HI);     {Normalize dq_r}
              AY0=11;                     {Used for exponent adjust}
              SR=LSHIFT SR1 BY -9 (HI);   {Remove lower bits}
              SR=LSHIFT SR1 BY 2 (HI);    {Adjust for ADSP-210x version}
              DM(I0,M2)=SR1, AR=AX1+AY0;  {Store mantissa, compute exp}
              AX1=DM(a_ik_r);             {Use sign of I, not dq}
              DM(I0,M2)=AR;               {Store exponent}
              DM(I0,M0)=AX1;              {Store sign}

update_p_r:   AY0=DM(sez);                {Compute new p}
              AR=AX0+AY0;                 {Use dq_r from above}
              AY1=DM(p_r);                {Delay old p's by 1}
              AY0=DM(p_o_r);
              DM(p_r)=AR;
              DM(p_o_r)=AY1;
              DM(p_o_o_r)=AY0;
              AX1=AR, AR=AR XOR AY0;      {Compute p and poo}
              MR1=AR, AR=AX1 XOR AY1;     {Compute p and po}
              MR0=AR;

upa2_r:       I3=^a_data;
              SI=PM(I5,M5);               {Hold a2 for later}
              AR=PM(I5,M5);               {Get a1 for computation of f}
              AR=ABS AR, AY0=DM(I3,M1);   {Cap magnitude of a1 to 1/2}
              AF=AR-AY0, SE=DM(I3,M1);
              IF GT AR=PASS AY0;
              IF NEG AR=-AR;              {Restore sign of f}
              SR=ASHIFT AR (LO), AY0=DM(I3,M1);
              AF=ABS MR0, AY1=DM(I3,M1);  {If p_r xor poo_r =1 subtract}
              AR=SR0, AF=PASS SR1;
              IF POS AR=AY1-SR0;
              IF POS AF=AY1-SR1+C-1;
              SR0=AR, AR=PASS AF;
              SR1=AR, AF=ABS MR1;
              AR=SR0+AY0, SE=DM(I3,M1);
              AF=SR1+AY1+C, AX0=DM(I3,M1);
              IF NEG AR=SR0-AY0;
              IF NEG AF=SR1-AY1+C-1;
              SR=LSHIFT AR (LO);
              AR=PASS AF;
```

(listing continues on next page)

```
                SR=SR OR ASHIFT AR (HI), AY0=SI;
                AY1=SR0, SR=ASHIFT SI (HI); {Leak factor of 1/128}
                AR=AY0-SR1, AY0=DM(I3,M1);
                AF=PASS AX1;
                IF NE AR=AR+AY1;            {If sigpk = 1 , no gain}

limc_r:         AF=AR-AY0, AY1=DM(I3,M1);   {Limit a2 to .75 max}
                IF GT AR=PASS AY0;
                AF=AR-AY1, AY0=DM(I3,M1);   {Limit a2 to -.75 min}
                IF LT AR=PASS AY1;
                PM(I5,M5)=AR;               {Store new a2}

upa1_r:         AR=AX0, AF=PASS MR0;
                IF LT AR=-AX0;
                AF=PASS AX1, SI=PM(I5,M4);
                IF EQ AR=PASS 0;
                SR=ASHIFT SI BY -8 (HI);    {Leak Factor = 1/256}
                AF=PASS AR, AR=SI;
                AF=AF-SR1;
                AR=AR+AF, AX1=DM(I3,M1);

limd_r:         AX0=AR, AR=AX1-AY1;         {Limit a1 based on a2}
                AY0=AR, AR=AY1-AX1;
                AY1=AR, AR=PASS AX0;
                AF=AR-AY0;
                IF GT AR=PASS AY0;          {Upper limit 1 - 2^-4 -a2}
                AF=AR-AY1;
                IF LT AR=PASS AY1;          {Lower limit a2 - 1 + 2^-4}
                PM(I5,M5)=AR;               {Store new a1}
                RTS;

.ENDMOD;
```

Listing 12.3 Nonstandard ADPCM Transcoder Routine