Sub-Band ADPCM 5

5.1 OVERVIEW

Pulse Code Modulation, or PCM (CCITT Recommendation G.711), is a method of digitizing analog wave forms to transmit speech signals. This quantization scheme provides 13 bits (μ-law) or 14 bits (A-law) of dynamic range in an 8-bit value. 13 or 14-bit dynamic range is the minimum requirement to accurately reproduce the full range of speech signals, therefore, μ-law and A-law encoding are widely used in telephony.

This A/D conversion can introduce quantization noise when the analog signals are quantized to digital values. The human ear is more sensitive to quantization noise when the noise component is relatively large compared to the size of the signal. Recommendation G.711 applies a non-uniform quantization function to adjust the data size in proportion to the input signal, thus reducing noise interference. As a result, smaller signals are approximated with greater accuracy.

Adaptive Differential Pulse Code Modulation, or ADPCM (CCITT Recommendation G.721), is more efficient to transmit than PCM. ADPCM uses an adaptive predictor to take advantage of the redundancies present in speech signals. It compares a signal sample with the previous sample and transmits the difference between the two. This reduces the number of bits needed to reproduce the speech. G.721 samples speech bandwidths of 200–3400 Hz at 8 kSa/s. The inputs and outputs of a G.721-based system are still PCM values. Although G.711 and G.721 are widely used, these methods are quality and bandwidth limited. Chapters 11 and 12 of *Digital Signal Processing Applications Using the ADSP-2100 Family, Volume 1*, briefly discuss PCM and ADPCM theory, and include program examples.

To improve the overall transmission quality and add a sub-carrier frequency, CCITT developed Sub-Band ADPCM (Recommendation G.722). Recommendation G.722 is a wideband audio recommendation (50 to 7000 Hz) that splits the frequency band into two sub-bands (0 to 4000 Hz and 4000 Hz to 8000 Hz), and applies ADPCM to the sub-bands independently. G.722 operates on linear samples of speech. The auxiliary data (non-encoded) channel is available for video transmission, and is used in applications such as teleconferencing.

5 Sub-Band ADPCM

This chapter describes a method to implement the G.722 algorithm with the ADSP-2100 Family of digital signal processors. To save memory space and to clarify the implementation, the program example (Listing 5.1) at the end of this chapter is written as a collection of subroutines. This format is efficient because the higher and lower sub-bands share most of the subroutines; in fact, many subroutines (such as `filtez` and `filtep`) are also shared by the encoder and the decoder of each sub-band.

5.2 SUB-BAND ADPCM ALGORITHM

CCITT Recommendation G.722 specifies the following six parts of the algorithm (see Figure 5.1):

- Transmit quadrature mirror filter (QMF)
- Lower sub-band encoder
- Higher sub-band encoder
- Lower sub-band decoder
- Higher sub-band decoder
- Receive quadrature mirror filter

The block diagram has two halves, transmit (encoder) and receive (decoder).The implementation of the multiplexer and demultiplexer are straightforward, and are not described in this chapter.

The subroutines included at the end of this chapter were verified against digital test sequences provided by CCITT for the standards, and are fully compatible with Recommendation G.722. When possible, the names of the subroutines and variables used in the algorithm match the names specified in the recommendation.

5.3 TRANSMIT PATH

This section describes the encoder and transmit path shown in Figure 5.1. The encoder operates at 64 kbits per second, with 16 kSa/s and 14 bits.

5.3.1 Transmit Quadrature Mirror Filter

The transmit quadrature mirror filter splits the frequency band into two sub-bands, higher and lower. It also decimates the input to the encoder from 16 kHz to 8 kHz. The filter is a 24 tap Finite Impulse Response filter, or FIR. The impulse response can be approximated as a simple delay function. The transmit quadrature mirror filter shares the same coefficients and 24 tap delay line with the receive QMF. Implementation of

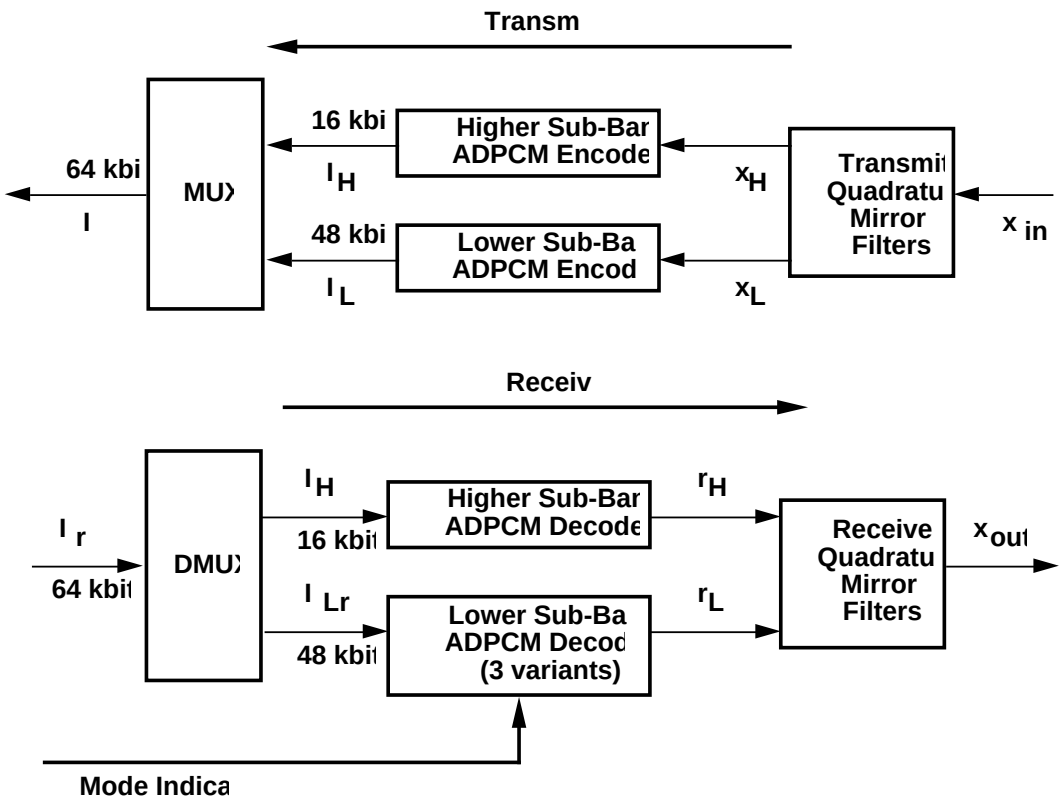

Figure 5.1 Sub-Band ADPCM Algorithm Block Diagram

QMFs in ADSP-2100 family assembly language is computationally efficient because the data adress generators, or DAGs, use indirect addressing to fetch filter coefficients and data values in the same processor cycle. Also, you can use circular buffering to represent the tapped delay lines. The output variables of the filters, `xl(n)` and `xh(n)` (lower and higher sub-band signal components), are determined by the following equations:

```
xl(n) = xa + xb
xh(n) = xa - xb
```

where

```
xa = h2i * xin(j-2i)
xb = h2i+1 * xin(j - 2i - 1)
```

5 Sub-Band ADPCM

5.3.2 Higher Sub-Band Encoder

Figure 5.2 is a functional block diagram of the higher sub-band encoder. The higher sub-band encoder operates on the differences between input signal value xh and the adaptive predictor signal estimate. After the predicted value is determined and the subtraction for the difference signal is performed, the estimate signal (el) is applied to a four level non-linear adaptive quantizer that assigns six binary digits to yield the 48 kbits/s signal, Il. Since data is not truncated from the output signal, Ih, in the feedback loop, the inverse adaptive quantizer is also 4 levels.

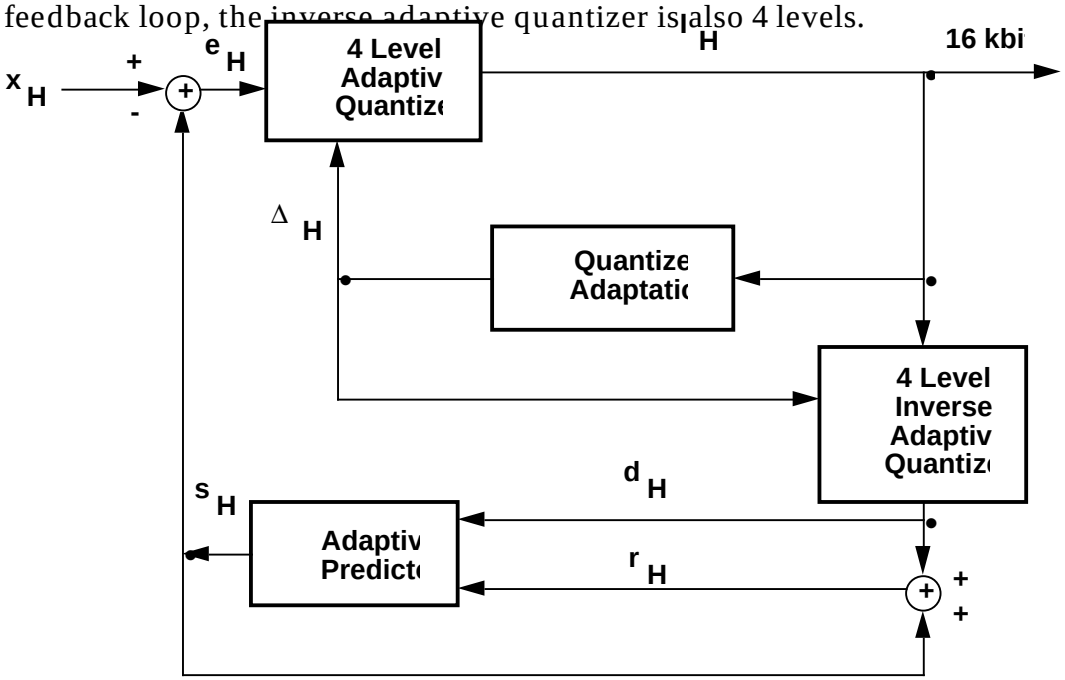

Figure 5.2 Higher Sub-Band Encoder Block Diagram

5.3.3 Lower Sub-Band Encoder

The lower sub-band encoder (shown in Figure 5.3) operates by estimating the difference in signal value between the predicted value and the actual input value. The structure of the adaptive predictor in the higher-band encoder is identical to the one in the lower sub-band encoder, but the names in memory of the adaptive predictor coefficients differ by an “l” or “h” to make the program more understandable. The number of bits required to represent the difference is smaller than the number of bits required to represent the total input signal. This difference is calculated by subtracting the predicted value from the input value:

```
el(n) = xl(n) - sl(n)
eh(n) = xh(n) - sh(n)
```

The predicted value (`sl(n)` or `sh(n)`) is produced by the adaptive predictor, which contains a second-order section to model poles, and a sixth-order section that models zeroes in the input signal. For every received sample, (`xl(n)` or `xh(n)`), `upzero` updates the six zero section coefficients of the predictor, `uppol2` calculates the second pole section coefficient, and `uppol1` calculates the first pole section coefficient.

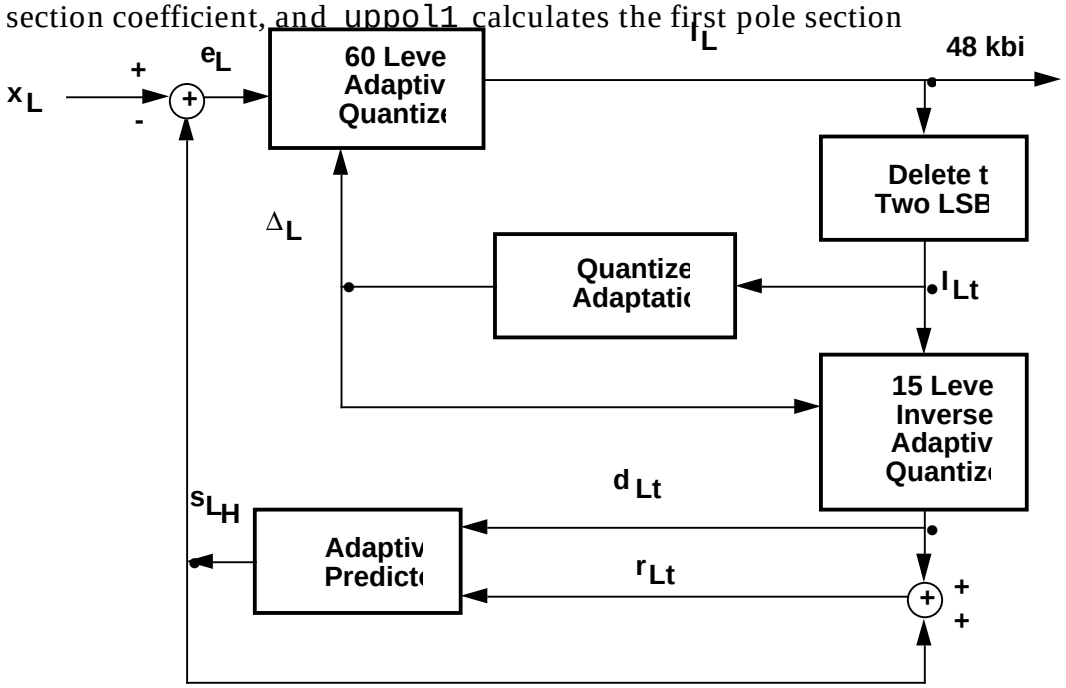

Figure 5.3 Lower Sub-Band Encoder Block Diagram

Operation is similar to the operation of the higher sub-band decoder except a 60-level adaptive quantizer is applied rather than a 4-level quantizer.

An important feature of the lower sub-band encoder is the feedback loop. The feedback loop is used for adaptation of the 60-level adaptive quantizer and to update the adaptive predictor. To do this, inside the feedback loop, the two least significant bits of `Il` are truncated to produce a 4-bit difference signal, `Ilt` . Since this value was already passed through the adaptive quantizer, an inverse adaptive quantizer produces `dlt` . This is a quantized difference signal that the adaptive predictor uses to produce `sl` (the estimate of the input signal) and update the adaptive predictor.

Four-bit operation (rather than 6-bit) leaves room for the auxiliary data

channel in the lower sub-band encoder.

5.4 RECEIVE PATH

This section describes the decoder and receive path, shown in Figure 5.1. While the encoder operates at 64 kbits/s, the decoder accepts encoded signals at 64, 56 and 48 Kbits/s. The two lower bit rates correspond to the availability of an auxiliary data channel that uses either 8 or 16 Kbits/s. The auxiliary data channel is described as a data insertion device that is totally separate from the G.722 encoder and decoder. Bits from the auxiliary data channel are simply carried over the same transmission medium as the G.722 encoded data.

The different bit rates available at the input of the decoder (depending on whether the auxiliary data channel is used) are referred to as the "modes" of operation (see Table 5.1). During operation of the algorithm on-chip, you must indicate the desired mode.

MODE	*7 kHz audio coding bit rate*	*Auxiliary data channel rate*
1	64 Kbits/s	0 Kbits/s
2	56 Kbits/s	8 Kbits/s
3	48 Kbits/s	16 Kbits/s

Table 5.1 Decoder Modes Of Operation

5.4.1 Higher Sub-Band Decoder

The higher sub-band decoder (see Figure 5.4) is the simplest element of sub-band ADPCM. There are no choices to make for inverse adaptive quantizers or mode control to indicate word truncation. Instead, the input code word, Ih, is fed into the 4-level inverse adaptive quantizer (to obtain Dh) and into the quantizer adaptation segment in parallel. The adaptive

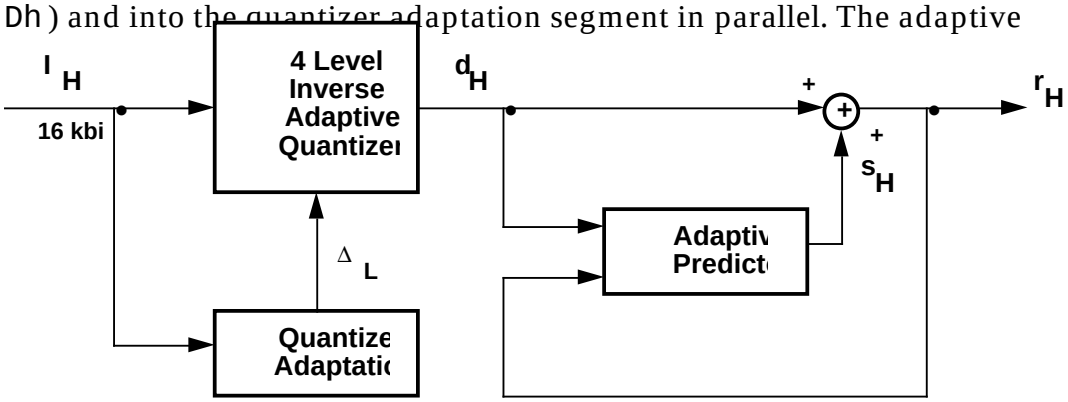

Figure 5.4 Higher Sub-Band Decoder Block Diagram

predictor generates the signal estimate Sh and adds to this the output of the inverse adaptive quantizer to generate the decoder output signal, Rh .

5.4.2 Lower Sub-Band Decoder

Figure 5.5 is a functional block diagram of the lower sub-band decoder. Generally, the higher and lower sub-band decoders and encoders share the same subroutine calls in almost the same order because they are similar in operation. In the lower sub-band decoder, however, the mode indication signal determines how many bits are truncated from the input codeword Ilr and which inverse adaptive quantizer is chosen in the

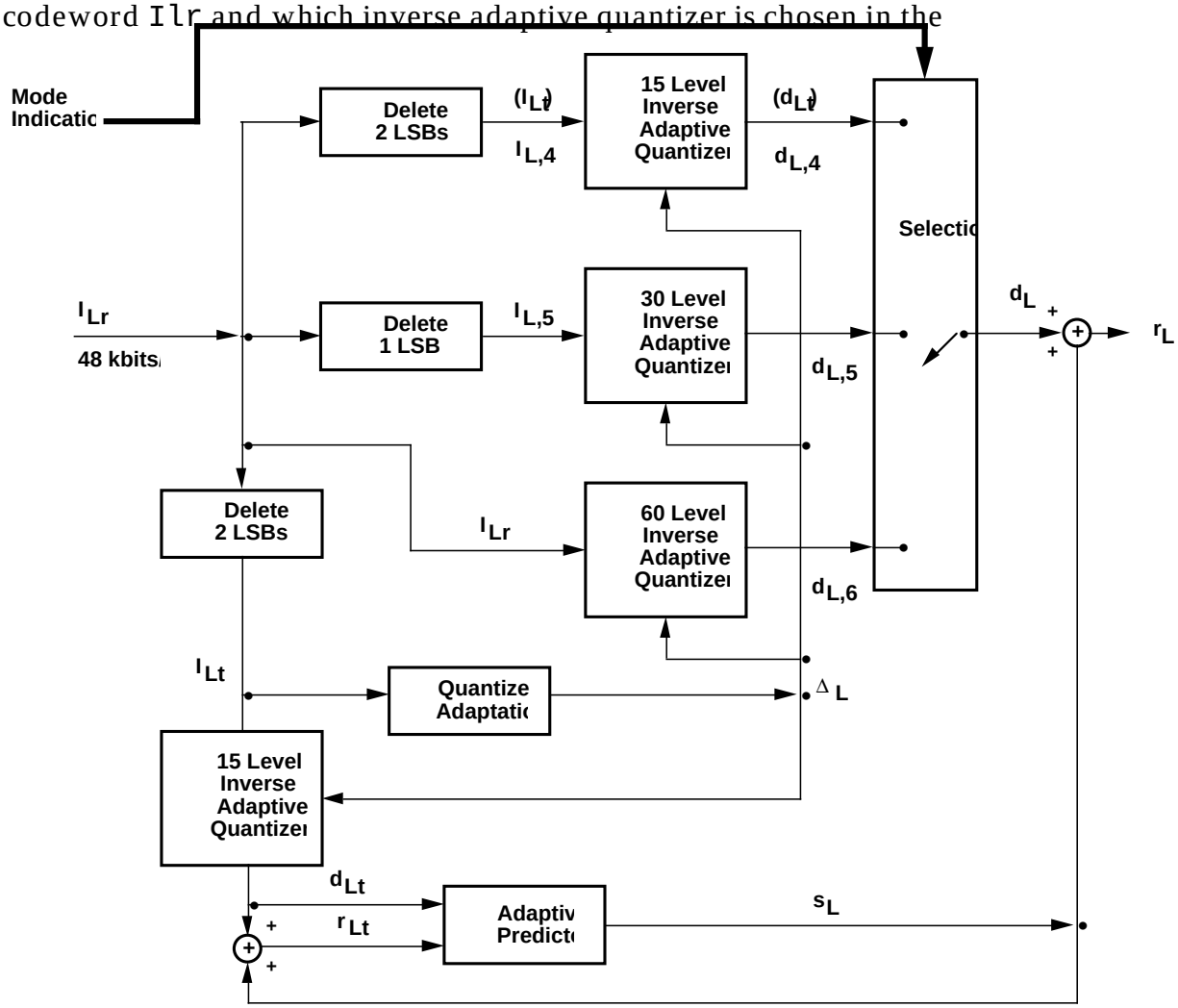

Figure 5.5 Lower Sub-Band Decoder Block Diagram

feedback loop. Table 5.2 shows you the correlation between the Mode and the number of levels for the inverse adaptive quantizer.

MODE	*Inverse adaptive quantizer levels*
1	60-level
2	30-level
3	15-level

Table 5.2 Inverse Adaptive Quantizer Modes Of Operation

In both the quantizers and inverse quantizers for the lower and higher sub-bands, indexed indirect memory access is used to read and write memory. An index of a data table is calculated (for example, see `quantl` subroutine, adaptive quantizer in the lower sub-band); this index is added to the base address of the data table and a single-cycle memory fetch is executed to obtain the desired address value.

5.4.3 Receive Quadrature Mirror Filter

The receive quadrature mirror filter interpolates the output of the decoder from 8 kHz to 16 kHz for input to the receive audio signal. The filter is a 24 tap finite impulse response filters whose impulse response can be approximated as a simple delay function.

5.5 ADSP-2100 FAMILY IMPLEMENTATION

The G.722 code implementation is a parameter set-up shell that calls subroutines corresponding to the subroutines listed in the CCITT recommendation. This is useful for the following reasons:

- Memory savings–both the encoder and decoder (and in some cases both sub-bands) use many of the same routines
- Easy transition from full-duplex to half-duplex implementation–copy the shell that includes the appropriate sub-routine calls

Circular buffering is used in the G.722 algorithm in several places: as delay lines in the receive and transmit quadrature mirror filters and in the adaptive predictor (a separate one for both encoder and decoder and both upper and lower sub-bands). The circular buffering implementation maintains only the necessary pieces of information (specific number of delay values). It does not require code to maintain the data or require

extraneous memory to store the data and time to service it.

5.6 SUBROUTINE DESCRIPTIONS

This section contains brief descriptions of the subroutines used to implement CCITT Recommendation G.722.

5.6.1 reset_mem

This subroutine initializes the state variables required for correct operation of the algorithm. You must call the `reset_mem` routine before running the encoder or decoder. The `reset_mem` routine also does the following things:

- Initializes the linear and circular buffers required by the filters, encoder, and decoder
- Sets up pointers (data memory values) to the circular buffers so the index registers do not need to be dedicated to the circular buffers
- Set up modify and length registers that will remain constant for the remainder of the algorithm

5.6.2 filtez

This subroutine computes the output of the zero section of the adaptive predictor by multiplying the zero section coefficients by the quantized difference signal buffer values. Higher and lower sub-band encoders and decoders use this subroutine.

5.6.3 filtep

This subroutine computes the output of the pole section of the adaptive predictor by multiplying the pole section coefficients by the quantized reconstructed signal buffers. Higher and lower sub-band encoders and decoders use this subroutine.

5.6.4 quantl

This subroutine calculates the encoder output codeword based on the difference in signal value and the quantizer scale factor, `detl` (calculated in `scalel` below).

This subroutine fetches data words from the look-up tables included in the G.722 recommendation. It is necessary to first compute an index that locates the magnitude of the signal difference relative to the quantizer decision levels. This is accomplished in the `ll1` loop. The decision levels (stored as a program memory data table) are multiplied by the quantizer scale factor and subtracted from the magnitude of the difference in signal value; if this value is less than zero, a flag is incremented to give the desired index. This index is then added to the address of the codeword

data table and the correct six-bit codeword is chosen. Only the lower sub-band encoder uses this subroutine.

5.6.5 invqxl

This subroutine represents the `invqal` and `invqbl` sections of the algorithm. `invqal` computes the lower sub-band quantized difference in signal value for the adaptive predictor of the encoder and decoder. `invqbl` computes the quantized difference in signal value for the decoder output in the lower sub-band decoder. Since these two routines are identical except for the presence of the "mode" signal (decoder only), they are merged to save code space. Again, this subroutine is based on an indexed table look-up, and the choice of tables depends on the mode of operation. For the encoder, you supply a constant to choose the correct table (and number of bits to be truncated). For the decoder, the shell program mathematically determines which table is chosen according to the mode you define. Similar to `quantl`, an index is calculated and added to the indicated table as an offset.

5.6.6 logscl

This subroutine updates the logarithmic scale factor in the lower sub-band encoder and decoder. It is an indexed table look-up subroutine with limits imposed on the output value, `nbpl`.

5.6.7 scalel

This subroutine computes the quantizer scale factor in the lower sub band encoder and decoder. In addition to scaling output values, this subroutine performs an indexed table look-up.

5.6.8 upzero

This subroutine determines six zero section predictor coefficients. The output values (a buffer of size six) depend on the value and signs of the quantized difference in signal value, some leakage and gain constants, the delayed difference signal values, and old zero-section predictor coefficients.

5.6.9 uppol2

This subroutine generates the second pole predictor coefficient. It is determined from the sign and value of the partially reconstructed signal `pl`, some leakage and gain constants, and the old (delayed) pole predictor coefficients.

5.6.10 uppol1

This subroutine generates the first pole predictor coefficient. It depends on the delayed first pole predictor coefficient, some leakage and gain

constants, `plt` and the old (delayed) pole predictor coefficients.

5.6.11 limit

This subroutine limits the output reconstructed signals for both lower and higher sub-band decoders.

5.6.12 quanth

This subroutine quantizes the difference in signal value in the higher sub-band encoder based on the magnitude of the signal, the higher sub-band quantizer scale factor, and a decision level indexed table look-up.

5.2.13 invqah

This subroutine computes the quantized difference in signal value in the higher sub-band encoder and decoder, based on the higher sub-band quantizer scale factor and the higher sub-band decoder output codeword, `Ih`, in indexed table look-up manner.

5.6.14 logsch

This subroutine determines the logarithmic quantizer scale factor in the higher sub-band encoder and decoder. This subroutine involves calculations for leakage and scale factors, and imposes some limits on the output signal.

Note: In addition to the routines mentioned above, several routines are performed in the shell itself. They are implemented in the shell because they are short, and because it saves two cycles (`call` and `rts`) for every execution. The following routines are implemented in the shell:

- SUBTRA
- RECONS
- PARREC

Also, the delay blocks DELAYZ, DELAYL, and DELAYA are implemented with the following two step process:

1. Variables (both single words and buffers) are given their initial value in `reset_mem` .

2. Variables are updated after processing through either the decoder or encoder with the newly computed value. They will contain the correct data for the next iteration through the system.

5 Sub-Band ADPCM

```
.module/ram/abs=0 g722;

/* ..................... variables for filters here ..................... */
.var/ram/dm/circ        tqmf_buf[23];
.var/ram/dm             accumab_ptr;
.var/ram/pm             coefs[24];
.init coefs:            <coeffs.dat>;
.var/ram/dm             xl;
.var/ram/dm/circ        accumc[11];
.var/ram/dm/circ        accumd[11];
.var/ram/dm             accumc_ptr;
.var/ram/dm             accumd_ptr;
.var/ram/dm             xh;
.var/ram/dm             xout1;
.var/ram/dm             xout2;
.var/ram/dm             xs;

/* ................ variables for encoder (hi and lo) here ............... */
.var/ram/dm             il;
.var/ram/dm             mode;
.var/ram/dm             szl;
.var/ram/dm             spl;
.var/ram/dm             sl;
.var/ram/dm             el;
.var/ram/dm             store_this;
.var/ram/pm             code4_table[0x20];
.init code4_table:      <codword4.dat>;
.var/ram/pm             code5_table[0x40];
.init code5_table:      <codword5.dat>;
.var/ram/pm             code6_table[0x80];
.init code6_table:      <codword6.dat>;
.var/ram/pm             qq6_table[0x80];
.init qq6_table:        <quant6.dat>;
.var/ram/pm             qq5_table[0x10];
.init qq5_table:        <quant5.dat>;
.var/ram/pm             qq4_table[8];
.init qq4_table :       <quant4.dat>;
.var/ram/dm             delay_bpl[6];
.var/ram/dm             dltx_ptr;
.var/ram/dm             fbuf[6];
.var/ram/dm             tbuf[6];
.var/ram/dm/circ        delay_dltx[7];
.var/ram/dm             il4;
.var/ram/pm             wl_table[8];
.init wl_table:         <wl.dat>;
.var/ram/pm             ilb_table[32];
.init ilb_table:        <ilb.dat>;
.var/ram/dm             nbl;              /* delay line */
.var/ram/dm             al1;
.var/ram/dm             al2;
```

```
.var/ram/dm             plt;
.var/ram/dm             plt1;
.var/ram/dm             plt2;
.var/ram/dm             rs;
.var/ram/dm             dlt;
.var/ram/dm             apl1;
.var/ram/dm             apl2;
.var/ram/dm             rlt;
.var/ram/dm             rlt1;
.var/ram/dm             rlt2;
.var/ram/pm             decis_levl[29];
.init decis_levl:       <q6shft3.dat>;
.var/ram/dm             detl;
.var/ram/pm             quant26bt_pos[30];
.init quant26bt_pos:    <quant6p.dat>;
.var/ram/pm             quant26bt_neg[30];
.init quant26bt_neg:    <quant6n.dat>;
.var/ram/dm             deth;
.var/ram/dm             sh;         /* this comes from adaptive predictor */
.var/ram/dm             eh;
.var/ram/pm             bit_out2[4];
.init bit_out2:         <bit_ih2.dat>;
.var/ram/dm             dh;
.var/ram/dm             ih;
.var/ram/dm             nbh;
.var/ram/dm             szh;
.var/ram/dm             sph;
.var/ram/dm             ph;
.var/ram/dm             yh;
.var/ram/dm             rh;
.var/ram/dm/circ        delay_dhx[7];
.var/ram/dm             delay_bph[6];
.var/ram/dm             dhx_ptr;
.var/ram/dm             ah1;
.var/ram/dm             ah2;
.var/ram/dm             aph1;
.var/ram/dm             aph2;
.var/ram/dm             ph1;
.var/ram/dm             ph2;
.var/ram/dm             rh1;
.var/ram/dm             rh2;

/* ..................... variables for decoder here ..................... */
.var/ram/dm             ilr;
.var/ram/dm             yl;
.var/ram/dm             rl;
.var/ram/dm             dec_deth;
.var/ram/dm             dec_detl;
.var/ram/dm             dec_dlt;
.var/ram/dm             dec_del_bpl[6];
.var/ram/dm             dec_dltx_ptr;
```

(listing continues on next page)

```
.var/ram/dm/circ        dec_del_dltx[7];
.var/ram/dm             dec_apl1;
.var/ram/dm             dec_apl2;
.var/ram/dm             dec_plt;
.var/ram/dm             dec_plt1;
.var/ram/dm             dec_plt2;
.var/ram/dm             dec_szl;
.var/ram/dm             dec_spl;
.var/ram/dm             dec_sl;
.var/ram/dm             dec_rlt1;
.var/ram/dm             dec_rlt2;
.var/ram/dm             dec_rlt;
.var/ram/dm             dec_al1;
.var/ram/dm             dec_al2;
.var/ram/dm             dl;
.var/ram/dm             dec_nbl;
.var/ram/dm             dec_yh;
.var/ram/dm             dec_dh;
.var/ram/dm             dec_nbh;

/* ..................... variables used in filtez ..................... */
.var/ram/dm             dec_del_bph[6];
.var/ram/dm             dec_dhx_ptr;
                              /* pointer for circ buffer index - hi sb dec */
.var/ram/dm/circ        dec_del_dhx[7];
.var/ram/dm             dec_szh;

/* ..................... variables used in filtep ..................... */
.var/ram/dm             dec_rh1;
.var/ram/dm             dec_rh2;
.var/ram/dm             dec_ah1;
.var/ram/dm             dec_aph1;
.var/ram/dm             dec_ah2;
.var/ram/dm             dec_aph2;
.var/ram/dm             dec_ph;
.var/ram/dm             dec_sph;
.var/ram/dm             dec_sh;
.var/ram/dm             dec_rh;
.var/ram/dm             dec_ph1;
.var/ram/dm             dec_ph2;
.var/ram/dm             x_num;

/* ................. starting with lower sub band encoder ................. */
/* ................ if in reset, initialize required memory ............... */

/* ..... encode: put input samples in my1 and mx0(calling parameters) ..... */
/* ................. my1 = first value, mx0 = second value ................ */
/* ............... returns il and ih stored together in ax0 ............... */
```

```
/* ................ decode: calling parameters: ilr and xh ................ */
/* ... return parameters: xout1 and xout2 (in ax0 and ax1 respectively) ... */
/* ........... note: supply mode signal to decoder also (in dm) ........... */

encode:  mstat = 0x0;
         i0 = dm(accumab_ptr);
         l0 = 23;
         m0 = 2;              /* skipping through buffer with a stride of 2 */
         i5 = ^coefs;
         l5 = 0;
         m6 = 2;
         si = mx0;
         mr = 0, my0 = pm(i5,m6);
         cntr = 11;
         do e_loop until ce;

/* .... main multiply accumulate loop for even samples and coefficients .... */
         e_loop: mr = mr + mx0 * my0(ss), mx0 = dm(i0,m0), my0 = pm(i5,m6);
         dm(i0,m2) =my1, mr = mr + mx0 * my0(ss); /* final mult/accumulate */
                                                  /* and write to delay line */

/* .. save mr here, want xa (contents of mr) to be at least 24 bits wide .. */
/* ...... so start moving mr outputs into alu regs for multiprecision ...... */
         sr0 = mr1;
         ay0 = mr0;           /* for multiprecis add in hight and lowt */
         cntr = 11;
         i5 = ^coefs+1;
         mr = 0,mx0 = dm(i0,m0), my0 = pm(i5,m6);
         do o_loop until ce;

/* ......... main loop for mult/accum odd inputs and coefficients ......... */
         o_loop: mr = mr + mx0 * my0(ss), mx0 = dm(i0,m0), my0 = pm(i5,m6);
         modify(i0,m0);
         modify(i0,m0);
         dm(i0,m2) = si, mr = mr + mx0 * my0(ss); /* final mult/accumulate */
                                                  /* and write to delay line */

lowt:    ar = mr0 + ay0,  ay1 = sr0; /* add low precis word from loop first */
         ena ar_sat;
         ar = mr1 + ay1 + C;  /* need 16 bits of info, but to keep precise */
         dis ar_sat;          /* this is xl, needs to be limited */
         call chk_vals;
         dm(xl) = ar;

hight:   ar = ay0 - mr0, ay1 = sr0;
         ena ar_sat;
         ar = ay1 - mr1 + C -1;                /* subtract with borrow */
         dis ar_sat;
         call chk_vals;
         dm(xh) = ar;
         dm(accumab_ptr) = i0;
```

(listing continues on next page)

```
/* ..... into regular encoder segment here - consider filters embedded ..... */
        mstat = 0x8;
        i1 = ^delay_bpl;
        i5 = dm(dltx_ptr);
        l5 = 7;
        l0 = 0;

/* ....... filtez - compute predictor output section - zero section ....... */
/* .... calling params: i1 points to delay_bpl, i2 points to delay_dltx .... */
/* ..................... return parameters: mr1 (szl) ..................... */
        call filtez;
        dm(szl) = ar;
        sr0 = dm(rlt1);
        my0 = dm(al1);
        ax0 = dm(rlt2);
        my1 = dm(al2);

/* ........ filtep - compute predictor output signal (pole section) ........ */
/* .................. calling params: sr0, my0, sr1, my1 .................. */
/* ..................... return parameters : ar (spl) ..................... */
        call filtep;

/*  predic:compute the predictor output value in the lower sub_band encoder  */
/*   not a subroutine but a small piece of code to compute predictor output  */
/* ................. adding together szl + spl to form sl ................. */
        dm(spl) = ar;
        ay0 = dm(szl);
        ar = ar + ay0;
        dm(sl) = ar;
        ay0 = dm(xl);    /* this is subtra : xl - sl = el (diff. signal) */
        ar = ay0 - ar;
        dm(el) = ar;

/* ................ quantl: quantize the difference signal ................ */
/* ........ calling params: el(ar), detl (which has value at reset) ........ */
/* ............. return parameters: il (4 bit codeword) in ax0 ............. */
        call quantl;
        dm(il) = ax0;
        my0 = dm(detl);
        ay0 = 3;                    /* this is mode for block 4l */
        mr0 = ax0;
        ay1 = ^code4_table;         /* remember, this will change w/ invqbl */

/* invqxl: does both invqal and invqbl- computes quantized difference signal */
/* .............. for invqbl, truncate by 2 lsbs, so ay0 = 3 .............. */
/* ................ calling parameters: il(mr0), detl(my0) ................ */
/* ............. and ay1(address of correct table for codeword ............. */
/* ...................... return paramters: dlt(mr1) ...................... */
        call invqxl;
        modify(i5,m7);
        m6 = 0;
```

```
        dm(i5,m6) = mr1;
        mr0 = dm(nbl);
        ar = dm(il);

/* .... logscl: updates logarithmic quant. scale factor in low sub band .... */
/* ..... calling parameters: il (ilr in decoder) - in ar , nbl in mr0 ..... */
/* ..... return parameters: nbl used next time - note - same var name ..... */
        call logscl;
        dm(nbl) = ar;
        ay1 = 8;

/* ... scalel: compute the quantizer scale factor in the lower sub band ... */
/*  calling params nbl(in ar) and 8(constant such that scalel can be scaleh  */
/* ......................... return parameter: detl ........................ */
        call scalel;
        dm(detl) = sr0;
        ax0 = dm(i5,m5);
        ay0 = dm(szl);

/* parrec - simple addition to compute recontructed signal for adaptive pred */
/* ..................... no subroutine, just in place ..................... */
/* .......... add predictor zero section + quantized diff signal .......... */
        ar = ax0 + ay0;
        dm(plt) = ar;

/* ... upzero: update zero section predictor coefficients (sixth order) ... */
/* ..... calling parameters: dlt(sr0); dlti(circ pointer for delaying ..... */
/* ..................... dlt1, dlt2, ..., dlt6 from dlt ..................... */
/* ........ bpli (linear_buffer in which all six values are delayed ........ */
/* ............... return params: updated bpli, delayed dltx ............... */
        i1 = ^delay_bpl;
        call upzero;
        ax0 = dm(al1);
        ay0 = ax0;
        mx0 = dm(al2);
        si = dm(plt);
        mr0 = dm(plt1);
        mr1 = dm(plt2);

/* . uppol2- update second predictor coefficient apl2 and delay it as al2 . */
/* ............. calling parameters: al1, al2, plt, plt1, plt2 ............. */
/* ..................... return parameters: apl2 (in ar) ..................... */
/* ..................... note: apl2 is limited to +-.75 ..................... */
        call uppol2;
        dm(apl2) = ar;
        dm(al2) = ar;
        mr0 = dm(plt1);
        mx0 = dm(al1);
        ay1 = dm(apl2);
        si = dm(plt);
```

(listing continues on next page)

```
/* .. uppol1 :update first predictor coefficient apl1 and delay it as al1 .. */
/* ............... calling parameters: al1, apl2, plt, plt1 ............... */
/* ................ note: wd3= .9375-.75 is always positive ................ */
        call uppol1;
        dm(apl1) =ar;
        dm(al1) = ar;

/* ...... recons : compute recontructed signal for adaptive predictor ...... */
/* ..................... parameters: sl(ax0), dlt(ay0) ..................... */
/* ...................... return parameters: rlt(ar) ...................... */
        ax0 = dm(sl);
        ay0 = dm(store_this);
        ar = ax0 + ay0;
        dm(rlt) = ar;

/* . done with lower sub_band encoder; now implement delays for next time . */
        modify(i5,m5);
        ax0 = dm(rlt1);
        dm(rlt2) = ax0;
        dm(rlt1) = ar;
        ax0 = dm(plt1);
        dm(plt2) = ax0;
        ax0 = dm(plt);
        dm(plt1) = ax0;
        dm(dltx_ptr) = i5;   /* save i5 in dltx_ptr, restore next time */
hi_sb_enc:i1 = ^delay_bph;
        i5 = dm(dhx_ptr);

/* ................... filtez: calling params: ax0, ax1 ................... */
/* ........................ return params: ar(szh) ........................ */
        call filtez;
        dm(szh) = ar;
        sr0 = dm(rh1);
        my0 = dm(ah1);
        ax0 = dm(rh2);
        my1 = dm(ah2);

/* ............... filtep: calling parms: sr0, my0, sr1, my1 ............... */
/* ........................ return params: ar (sph) ........................ */
        call filtep;
        dm(sph) = ar;
        ay0 = dm(szh);
        ar = ar + ay0;
        dm(sh) = ar;                    /* predic: sh = sph + szh */
        ay0 = dm(xh);
        ar = ay0 - ar;
        dm(eh) = ar;                    /* subtra: eh = xh - sh */
        my0 = dm(deth);

/* ........ quanth: calling params: eh(ar), deth (has init. value) ........ */
```

```
/* ........................... return: ih in ax0 ........................... */
        call quanth;
        dm(ih) = ax0;
        ay0 = ax0;

/*  invqah: compute the quantized difference signal in th ehigher sub_band  */
/* ............. calling parameters: ih(in ax0); deth(in my0) ............. */
/* ..................... return parameters: dh (in mr1) ..................... */
        call invqah;
        modify(i5,m7);
        m6=0;
        dm(i5,m6) = mr1;
        ay0 = dm(ih);
        my0 = 0x7f00;
        mx0 = dm(nbh);

/* ... logsch: update logarithmic quantizer scale factor in hi sub band ... */
/* ....... calling paameters: ih(ay0), nbh(mx0), my0 has a constant ....... */
/* ................. return parameters: updated nbh (in ar) ................. */
        call logsch;
        dm(nbh) = ar;
        ay1 = 0xa;

/* ..... note : scalel and scaleh use same code, different parameters ..... */
        call scalel;
        dm(deth) = sr0;

/* . parrec - add pole predictor output to quantized diff. signal(in place . */
        ax0 = dm(i5,m5);
        ay0 = dm(szh);
        ar = ax0 + ay0;
        dm(ph) = ar;

/* ... upzero: update zero section predictor coefficients (sixth order) ... */
/* .......... calling parameters: dh(sr0); dhi(circ), bphi (circ) .......... */
/* ............... return params: updated bphi, delayed dhx ............... */
        i1 = ^delay_bph;
        call upzero;

/* ...... uppol2: update second predictor coef aph2 and delay as ah2 ...... */
/* ................. calling params: ah1, ah2, ph, ph1, ph2 ................. */
/* ..................... return params:  aph2 (in ar) ..................... */
/* ..................... note: aph2 is limited to +- .75 ..................... */
        ax0 = dm(ah1);
        ay0 = ax0;
        mx0 = dm(ah2);
        si = dm(ph);
        mr0 = dm(ph1);
        mr1 = dm(ph2);
        call uppol2;
        dm(aph2) = ar;
        dm(ah2)  = ar;
```

(listing continues on next page)

```
/* .... uppol1:  update first predictor coef. aph2 and delay it as ah1 .... */
/* ..............note: wd3 = .9375 -.75 is always positive ..............  */
         mr0 = dm(ph1);
         mx0 = dm(ah1);
         ay1 = dm(aph2);
         call uppol1;
         dm(apl1) = ar;
         dm(ah1) = ar;
         ax0 = dm(sh);
         ay0 = dm(store_this);
         ar = ax0 + ay0;
         dm(yh) = ar;

/* ......limit determines the greatest and smallest magnitude of the ......  */
/* ..................... recontructed output signal .....................  */
/* .........calling params: yl (in ar); return params: rh (in ar) .........  */
/* ......  done with higher sub-band encoder, now Delay for next time ......  */
         ax0 = dm(rh1);
         dm(rh2) = ax0;
         dm(rh1) = ar;
         ax0 = dm(ph1);
         dm(ph2) = ax0;
         ax0 = dm(ph);
         dm(ph1) = ax0;
         modify(i5,m5);
         dm(dhx_ptr) = i5;

/* ............ multiplexing ih and il to get signals together ............  */
         si = dm(ih);
         ar = dm(il);
         sr = lshift si by 6(lo);
         sr = sr or lshift ar by 0(lo);
         ax0 = sr0;

/* ................. multiplexed transmission word in ax0 .................  */
         rts;            /* done with encode */

/* ........................ LOWER SUB_BAND DECODER ........................  */
/* .......expect to split transmitted word from ax0 into ilr and ih .......  */
decode: ay0 = 0x3f;
         ar = ax0 and ay0;
         dm(ilr) = ar;
         ay0 = 0xc0;
         ar = ax0 and ay0;
         sr = lshift ar by -6 (lo);
         dm(ih) = sr0;                    /* place ih in two lsb's of sr0 */

lo_sb_dec:
         mstat = 0x8;
         i1 = ^dec_del_bpl;
         i5 = dm(dec_dltx_ptr);
```

```
/* ..... filtez: compute predictor output for zero section ..... */
/* .. calling parmeters: addresses of zero section input and output bufs .. */
/* ................. return parameters: del_szl (in mr1) ................. */
        call filtez;
        dm(dec_szl) = ar;
        sr0 = dm(dec_rlt1);
        my0 = dm(dec_al1);
        ax0 = dm(dec_rlt2);
        my1 = dm(dec_al2);

/* ....... filtep: compute predictor output signal for pole section ....... */
/* ...... calling parameters: dec_rlt1, dec_rlt2, dec_al1 and dec_al2 ...... */
/* .................. return parameter: del_spl (in ar) .................. */
        call filtep;
        dm(dec_spl) = ar;
        ay0 = dm(dec_szl);
        ar = ar + ay0;
        dm(dec_sl) = ar;
        ay0 = 3;
        mr0 = dm(ilr);
        ay1 = ^code4_table;
        my0 = dm(dec_detl);

/* invqxl: compute quantized difference signal for adaptive predic in low sb */
/* ............... calling parameters: my0, mr0, ay1 , ay0 ............... */
/* .................. return parameters: mr1 (dec_dlt) .................. */
        call invqxl;
        modify(i5,m7);
        m6=0;
        dm(i5,m6) = mr1;
        ay0 = dm(mode);
        mr0 = ^code4_table;
        mr1 = ^code5_table;
        sr0 = ^code6_table;
        ax0 = 2;
        ax1 = 3;
        af = ay0 - 1;
        if eq ar = pass sr0;
        af = ay0 - ax0;
        if eq ar = pass mr1;
        af = ay0 - ax1;
        if eq ar = pass mr0;
        ay1 = ar;
        mr0 = dm(ilr);
        my0 = dm(dec_detl);
```

(listing continues on next page)

5 Sub-Band ADPCM

```
/* invqxl: compute quantized difference signal for decoder output in low sb */
/* ..............calling parameters: my0, mr0, ay1 , ay0 ..............  */
/* ..................... return parameters: mr1( dl) ..................... */
        call invqxl;
        dm(dl) = mr1;
        ay0 = dm(dec_sl);
        ar = mr1 + ay0;
        dm(yl) = ar;

/* ................... limit: calling parameters yl (ar) ................... */
/* ..................... return parameters: rl (ar) ..................... */
        call limit;
        dm(rl) = ar;

/* .... logscl: quantizer scale factor adaptation in the lower sub-band .... */
/* ... calling parameters: dec_nbl (in mr1, dm(il4) calculated in invqxl ... */
        mr0 = dm(dec_nbl);
        ar = dm(ilr);
        call logscl;
        dm(dec_nbl) = ar;
        ay1 = 8;

/* ..... scalel: computes quantizer scale factor in the lower sub band ..... */
/* .. calling params:  updated dec_nbl, and ay1 for integer part scaling .. */
        call scalel;
        dm(dec_detl) = sr0;

/* . parrec - add pole predictor output to quantized diff. signal(in place . */
/* .................. for partially reconstructed signal .................. */
        ax0 = dm(i5,m5);
        ay0 = dm(dec_szl);
        ar = ax0 + ay0;
        dm(dec_plt) = ar;
        i1 = ^dec_del_bpl;

/* .......... upzero: update zero section predictor coefficients .......... */
/*calling params: dec_dlt(sr0),dec_dlti(circ buffer),dec_bpli(linear buffer)*/
/* ......... return parameters: updated dec_bpli, delayed dec_dlti ......... */
/* ..... note: am saving the index(i) register for circ buffers to mem ..... */
        call upzero;
        ax0 = dm(dec_al1);
        ay0 = ax0;
        mx0 = dm(dec_al2);
        si = dm(dec_plt);
        mr0 = dm(dec_plt1);
        mr1 = dm(dec_plt2);

/* . uppol2: update second predictor coefficient apl2 and delay it as al2 . */
/* . calling parameters: al1(ax0), al2(mx0), plt(si), plt1(mr0), plt2(mr1) . */
```

```
/* ..................... return parameters: apl2 (in ar) ..................... */
        call uppol2;
        dm(dec_apl2) = ar;
        dm(dec_al2) = ar;
        mr0 = dm(dec_plt1);
        mx0 = dm(dec_al1);
        ay1 = dm(dec_apl2);
        si = dm(dec_plt);

/* .......... uppol1: update first predictor coef. (pole setion) .......... */
/* calling params: dec_plt1 (mr0), dec_plt(si), dec_al1(mx0), dec_apl2(ay1) */
/* ..................... return parameter: apl1 (in ar) ..................... */
        call uppol1;
        dm(dec_apl1) = ar;
        dm(dec_al1)  = ar;
        ax0 = dm(dec_sl);
        ay0 = dm(store_this);

/* ...... recons : compute recontructed signal for adaptive predictor ...... */
/* .............. adding together  dec_sl(ax0), dec_dlt(ay0) .............. */
        ar = ax0 + ay0;
        dm(dec_rlt) = ar;

/* ... done with lower sub band decoder, implement delays for next time ... */
        modify(i5,m5);
        ax0 = dm(dec_rlt1);
        dm(dec_rlt2) = ax0;
        dm(dec_rlt1) = ar;
        ax0 = dm(dec_plt1);
        dm(dec_plt2) = ax0;
        ax0 = dm(dec_plt);
        dm(dec_plt1) = ax0;
        dm(dec_dltx_ptr) = i5;

/* ......................... HIGH SUB-BAND DECODER ......................... */
hi_sb_dec:
        i1 = ^dec_del_bph;
        i5 = dm(dec_dhx_ptr);

/* ........... filtez: compute predictor output for zero section ........... */
/* .. calling parameters: addresses of zero section input and output bufs .. */
/* .................. return parameters: dec_shl (in mr1) .................. */
        call filtez;
        dm(dec_szh) = ar;
        sr0 = dm(dec_rh1);
        my0 = dm(dec_ah1);
        ax0 = dm(dec_rh2);
        my1 = dm(dec_ah2);
```

(listing continues on next page)

5 Sub-Band ADPCM

```
/* ....... filtep: compute predictor output signal for pole section ....... */
/* ....... calling parameters: dec_rh1, dec_rh2, dec_ah1 and dec_ah2 ....... */
/* .................. return parameter: dec_sph (in ar) .................. */
        call filtep;
        dm(dec_sph) = ar;

/* predic:compute the predictor output value in the higher sub_band decoder */
/* ............... adding dec_szh and dec_sph to form dec_sh ............... */
        ay0 = dm(dec_szh);
        ar = ar + ay0;
        dm(dec_sh) = ar;
        ax0 = dm(ih);
        ay0 = ax0;
        my0 = dm(dec_deth);

/*  invqah: compute the quantized difference signal in th ehigher sub_band  */
/* ............. calling parameters: ih(in ax0); deth(in my0) ............. */
/* .................. return parameters: dec_dh (in mr1) .................. */
        call invqah;
        modify(i5,m7);
        m6=0;
        dm(i5,m6) = mr1;
        ay0 = dm(ih);
        my0 = 0x7f00;
        mx0 = dm(dec_nbh);

/* ... logsch: update logarithmic quantizer scale factor in hi sub band ... */
/* ..... calling parameters: ih(ay0), dec_nbh(mx0), my0 has a constant ..... */
/* .............. return parameters: updated dec_nbh (in ar) .............. */
        call logsch;
        dm(dec_nbh) = ar;
        ay1 = 0xa;

/* ... scalel: compute the quantizer scale factor in the higher sub band ... */
/* calling params: dec_nbl(in ar) and 10(constant so that scalel is re-used */
/* .................. return parameter: dec_deth(in sr0) .................. */
        call scalel;
        dm(dec_deth) = sr0;
        ax0 = dm(i5,m5);
        ay0 = dm(dec_szh);

/* ............. parrec: compute partially recontructed signal ............. */
/* ................ add together ax0(dec_dh), ay0 (dec_szh) ................ */
        ar = ax0 + ay0;
        dm(dec_ph) =ar;
        i1 = ^dec_del_bph;

/* .......... upzero: update zero section predictor coefficients .......... */
/*  calling params: dec_dh (sr0), dec_dhi(circ buffer), dec_bph(linear buf  */
/* .......... return parameters: updated dec_bph, delayed dec_dhi .......... */
```

```
/* ..... note: am saving the index(i) register for circ buffers to mem ..... */
        call upzero;
        ax0 = dm(dec_ah1);
        ay0 = ax0;
        mx0 = dm(dec_ah2);
        si = dm(dec_ph);
        mr0 = dm(dec_ph1);
        mr1 = dm(dec_ph2);

/* . uppol2: update second predictor coefficient aph2 and delay it as ah2 . */
/* ........ calling parameters:dec_ah1(ax0),dec_ah2(mx0),dec_ph(si) ........ */
/* ....................... dec_ph1(mr0),dec_ph2(mr1 ....................... */
/* ..................... return parameters: aph2 (in ar) ..................... */
        call uppol2;
        dm(dec_aph2) = ar;
        dm(dec_ah2)  = ar;
        mr0 = dm(dec_ph1);
        mx0 = dm(dec_ah1);
        ay1 = dm(dec_aph2);

/* .......... uppol1: update first predictor coef. (pole setion) .......... */
/*calling parameters: dec_ph1 (mr0), dec_ph(si), dec_ah1(mx0), dec_aph2(ay1)*/
/* ..................... return parameter: aph1 (in ar) ..................... */
        call uppol1;
        dm(dec_aph1) = ar;
        dm(dec_ah1)  = ar;
        ax0 = dm(dec_sh);
        ay0 = dm(store_this);

/* ...... recons : compute recontructed signal for adaptive predictor ...... */
/* ............... add  parameters: dec_sh(ax0), dec_dh(ay0) ............... */
/* ..................... to get parameters: dec_yh(ar) ..................... */
        ar = ax0 + ay0;
        dm(dec_yh) = ar;

/* ................ implementing delays for next time here ................ */
        ax0 = dm(dec_rh1);
        dm(dec_rh2) = ax0;
        dm(dec_rh1) = ar;                       /* ar has dec_yh */
        ax0 = dm(dec_ph1);
        dm(dec_ph2) = ax0;
        ax0 = dm(dec_ph);
        dm(dec_ph1) = ax0;
        modify(i5,m5);
        dm(dec_dhx_ptr) = i5;

/* ............ limit: limiting the output reconstructed signal ............ */
/* ..................... calling params:dec_yh(in ar) ..................... */
/* ..................... return parameters: dec_rh(in ar) ..................... */
        call limit;
        dm(rh) = ar;
```

(listing continues on next page)

```
/* ..................... end of higher sub_band decoder ..................... */
/* ............. start with receive quadrature mirror filters ............. */
recv_qmf: mstat = 0x0;
         i5 = ^coefs;
         l5 = 0;
         i0 = dm(accumc_ptr);
         m0 = 0;
         l0 = 11;
         i1 = dm(accumd_ptr);
         l1 = 11;
         m6 = 2;
         ena ar_sat;
         ax0 = dm(rl);
         ay0 = dm(rh);
         ar = ax0 + ay0;                  /* xs in af */
         dm(xs) = ar;
         ar = ax0 - ay0;                  /* xd in ar */
         dis ar_sat;
         mx0 = ar;
         si = ar;
         cntr = 11;
         mr = 0, my0 = pm(i5,m6);
         do accumc_loop until ce;
                  accumc_loop: mr = mr + mx0 * my0(ss), mx0 = dm(i0,m3),
                  my0 = pm(i5,m6);
         mr = mr + mx0 * my0 (ss);
         modify(i0,m1);
         sr = ashift mr1 by -15(hi);
         sr = sr or lshift mr0 by -15(lo);
         dm(i0,m2) = si;
         ar = pass sr0;
         call chk_vals;
         dm(xout1) = ar;                  /* could leave this in a register */
         i5 = ^coefs +1;
         mr = 0, my0 = pm(i5,m6);
         cntr = 11;
         mx0 = dm(xs);
         si = mx0;
         do accumd_loop until ce;
                  accumd_loop: mr = mr + mx0 * my0(ss), mx0 = dm(i1,m3),
                  my0 = pm(i5,m6);
         mr = mr + mx0 * my0(ss);
         modify(i1,m1);
         sr = ashift mr1 by -15(hi);
         sr = sr or lshift mr0 by -15(lo);
         dm(i1,m2) = si;
         ar =pass sr0;
         call chk_vals;
         dm(xout2) = ar;
         dm(accumc_ptr) = i0;
         dm(accumd_ptr) = i1;
         rts;
```

```
reset_mem: ax0 = 1;
          dm(rs) = ax0;
          ax0 = 0x8;
          dm(deth) = ax0;
          dm(dec_deth) = ax0;
          ax0 = 0x20;
          dm(detl) = ax0;
          dm(dec_detl) = ax0;
          ax0 = 0;
          dm(nbl)  = ax0;
          dm(al1)  = ax0;
          dm(al2)  = ax0;
          dm(plt1) = ax0;
          dm(plt2) = ax0;
          dm(rlt1) = ax0;
          dm(rlt2) = ax0;
          dm(nbh)  = ax0;
          dm(ah1)  = ax0;
          dm(ah2)  = ax0;
          dm(ph1)  = ax0;
          dm(ph2)  = ax0;
          dm(rh1)  = ax0;
          dm(rh2)  = ax0;
          dm(dec_rlt1) = ax0;
          dm(dec_rlt2) = ax0;
          dm(dec_al1)  = ax0;
          dm(dec_al2)  = ax0;
          dm(dec_nbl)  = ax0;
          dm(dec_plt1) = ax0;
          dm(dec_plt2) = ax0;
          dm(dec_rh1)  = ax0;
          dm(dec_rh2)  = ax0;
          dm(dec_ah1)  = ax0;
          dm(dec_ah2)  = ax0;
          dm(dec_nbh)  = ax0;
          dm(dec_ph1)  = ax0;
          dm(dec_ph2)  = ax0;
```

(listing continues on next page)

```
/* .................. reserved regs for c run-time model .................. */
        m1 = 1;
        m5 = 1;
        m3 = -1;
        m7 = -1;
        m2 = 0;
        i5 = ^delay_dltx;
        l5 = 7;
        cntr = 7;
        do init_circ0 until ce;
                init_circ0: dm(i5,m5) = 0;
        i5 = ^delay_dhx;
        cntr = 7;
        do init_circ1 until ce;
                init_circ1: dm(i5,m5) = 0;
        i5 = ^dec_del_dltx;
        cntr = 7;
        do init_circ2 until ce;
                init_circ2: dm(i5,m5) = 0;
        i5 = ^dec_del_dhx;
        cntr = 7;
        do init_circ3 until ce;
                init_circ3: dm(i5,m5) = 0;
        i0 = ^delay_bpl;
        l0 =0;
        i1 = ^delay_bph;
        l1 = 0;
        i5 = ^dec_del_bpl;
        l5 = 0;
        i6 = ^dec_del_bph;
        l6 = 0;
        cntr = 0x6;
        do init_lin until ce;
                dm(i0,m1) = 0;
                dm(i1,m1) = 0;
                dm(i5,m5) = 0;
        init_lin: dm(i6,m5) = 0;
        i0 = ^tbuf;
        i1 = ^fbuf;
        cntr = 6;
        do init_temp_bufs until ce;
                dm(i0,m1) = 0;
        init_temp_bufs: dm(i1,m1) = 0;   /* initialize temporary buffers */
```

```
/* .... save circ buffer index ptrs in mem, may need them in the future .... */
/* .... set up permanent length and index registers for encoder/decoder .... */
/* set up pointers for circular buffers, restore at the end of encode/decode */
        ax0 = ^delay_dltx;
        dm(dltx_ptr) = ax0;
        ax0 = ^delay_dhx;
        dm(dhx_ptr) = ax0;
        ax0 = ^dec_del_dltx;
        dm(dec_dltx_ptr) = ax0;
        ax0 = ^dec_del_dhx;
        dm(dec_dhx_ptr) = ax0;

/* ............. set up pointers for circ. buffers in filters ............. */
/* ..................... initialize circ buffers in mem ..................... */
        i0 = ^tqmf_buf;
        l0=0;
        cntr = 23;
        do init_tqmf until ce;
                init_tqmf: dm(i0,m1) = 0;
        i1 = ^accumc;
        l1=0;
        i5 = ^accumd;
        l5=0;
        cntr = 11;
        do init_fil until ce;
                dm(i5,m5) = 0;
                init_fil: dm(i1,m1) = 0;

        ax0 = ^tqmf_buf;     /* these  are input values */
        dm(accumab_ptr) = ax0;
        ax0 = ^accumc;
        dm(accumc_ptr) = ax0;
        ax0 = ^accumd;
        dm(accumd_ptr) = ax0;
        l5 = 7;                     /* final set up for length register */
        m1 = 1;
        m5 = 1;
        m2 = 0;
        m3 = -1;
        m7 = -1;
        l1 = 0;
        rts;
```

(listing continues on next page)

```
filtez: mr = 0, ay0 = dm(i5,m5);                /* i1 points to bpl */
                        /* i2 points to delay buffer(dltx) */
         i0 = ^fbuf;     /* io is temp buffer for adding delay line values */
         ar = pass ay0, my0 = dm(i1,m1);
         ar = ar + ay0;
         cntr = 6;
         do m_loop until ce;
                  mr = ar * my0(ss), ay0 = dm(i5,m5);
                  dm(i0,m1) = mr1;
                  ar = pass ay0, my0 = dm(i1,m1);
         m_loop: ar = ar + ay0;

         i0 = ^fbuf;
         ar = pass 0;
         cntr = 6;
         do filt1 until ce;
                  ay0 = dm(i0,m1);
         filt1: ar = ar + ay0;
         rts;

filtep:  ay1 = sr0;
         ar = sr0 + ay1;                  /* add rlt1 + rlt1 */
         mr = ar * my0(ss), ay1 = ax0;    /* multiply by al1 */
         ay0 = mr1;                       /* save wd1 in ay0 */
         ar = ax0 + ay1, ay0 = mr1;       /* ar = rlt2 + rlt2,wd1 in ay0 */
         mr = ar  * my1(ss);              /* wd2 * al2 in mr1 */
         ar = mr1 + ay0;                  /* wd1 + wd2 */
         rts;

quantl:  sr = ashift ar by -15(lo);
         af = pass sr0;
         if eq jump cont;
         ay0 = 0x7fff;
         af = ay0 - ar, ax0 = ay0; ar = ax0 and af;

cont:    i6 = ^decis_levl;
         ay1 = 0;
         ay0 = ar;
         my0 = dm(detl);
         af = pass 0, mx0 = pm(i6,m5);
         cntr = 0x1d;
         do ll1 until ce;
                  mr = mx0 * my0(ss), mx0 = pm(i6,m5);
                  ar = ay0 - mr1;
                  if lt af = af + 1;
         ll1: ar = pass af;
         /* i0 has mil starting from 1 */
         ay1 = 0x1e;                      /* process i0 now from ay1 */
         ar = ay1 - ar;
         af = pass af;                    /* if el is greater than table values */
```

```
         if eq ar = pass ay1;             /* mil gets 30 */
         i6 = ar;
         ar = ^quant26bt_neg;
         ay0 = ^quant26bt_pos;
         af = pass sr0;
         if eq ar = pass ay0;
         af = pass ar;
         ar = af - 1;         /* offset by 1 to start addressing from 0 */
         m6 = ar;
         modify(i6,m6);
         ax0 = pm(i6,m6);
         rts;

         /* invqxl is either invqbl or invqal depending on params passed */
invqxl:  ar = ay0 - 1;                    /* ay0 is passed in to indicate */
         ar = -ar;                        /* how many bits to shift by */
         se = ar;
         sr = lshift mr0(lo);
         sr = ashift sr0 by 1(lo);
         ar = sr0 + ay1;
         i6 = ar;
         mr1 = ^qq6_table;
         mr0 = ^qq5_table;
         sr0 = ^qq4_table;
         mr2 = 2;
         sr1 = 3;
         af = ay0 - 1,ax0 = pm(i6,m5);    /* save value from table here */
         if eq ar = pass mr1;
         af = ay0 - mr2,ax1 = pm(i6,m5);  /* save sign from table here */
         if eq ar = pass mr0;
         af = ay0 - sr1;
         if eq jump offset_0;
         af = pass ax0;
         ar = ar + af;
         af = pass ar;
                                          /* work around here; qq4 starts */
                                          /* at offset 0, qq5 & qq6 at 1 */
         ar = af -1;                      /* need this for qq5 & qq6, not 4 */
         jump get_sign;

offset_0: ar = pass sr0;
         af = pass ax0;
         ar = ar + af;              /* no offset for qq4, values start at 0 */

get_sign: i6 = ar;
         ar = pm(i6,m5);
         sr = ashift ar by 3(lo);         /* now add sign */
         af = pass ax1, ar= sr0;          /* if its neg, negate value */
         if lt ar = -ar;
         mr = ar * my0(ss);               /* round off here, check it out */
         rts;
```

(listing continues on next page)

```
logscl:  my0 = 0x7f00;     /* compensating for scale factor 32512 */
         mr = mr0 * my0(SS);           /* wd in mr1 */
         sr = lshift ar by -2(lo);
         sr = ashift sr0 by 1(lo);
         ay1 = ^code4_table;
         ar = sr0 + ay1;
         i6 = ar;
         m6 = 0;
         ax0 = pm(i6,m6);
         ay0 = ^wl_table;  /* use value from code4_table as index */
         ar = ax0 + ay0;   /* into wl_table */
         i6 = ar;
         ay1 = pm(i6,m6);  /* address for wl(il4) here */
         ar = mr1 + ay1;   /* nbpl here  */
         ay0 = 0x4800;
         af = pass ar;
         if lt af = pass 0;      /* limiting ar - if >18432 */
                                 /* nbpl gets 18432 */
         ar = ar - ay0;          /* if < 0 gets 0 */
         if gt af = pass ay0;
         ar = pass af;
         rts;                    /* this is new delay value */

scalel:  si = ar;
         sr = ashift ar by -6(hi);
         ay0 = 0x1f;
         ar = sr1 and ay0;             /* and with 31 - ar has WD1 */
         mr0 = ar;                     /* this is wd1 in mr0 */
         sr = ashift si by -11(lo);    /* this gives wd2 in sr0 */
         ar =  ay1 - sr0;              /* ay1 has 8 for scalel */
                                       /* and 10 scaleh */
         ar = -ar;
         se = ar;                /* se gets 8 - wd2 */
         ay1= ^ilb_table;
         ar = mr0 + ay1;
         i6 = ar;                /* use wd1 as index in ilb_table */
         ar = pm(i6,m5);
         sr = ashift ar (lo);    /* wd3 = ilb(wd1) >> (8-wd2) */
         sr = ashift sr0 by 2(lo);
         rts;

upzero:  ay0 = ax0;
         se = -15;
         mx0 = 0x7f80;
         ar = pass ay0;
         if eq jump wdi_over;
         ay0 = 0x80;
```

```
wdi_over: sr = ashift ar(lo),si = dm(i5,m5);
          ay1 = sr0;
          cntr = 6;
          do upzero_l until ce;
                 sr = ashift si(lo), si = dm(i5,m5);
                 ax1 = sr0;
                 af = pass ay0;
                 ar = ax1 - ay1, my0 = dm(i1,m2);
                 if ne af = -af;
                 mr = mx0 *my0(ss);
                 ar = mr1 + af;
          upzero_l:dm(i1,m1) = ar;
          dm(store_this) = si;
          rts;

uppol2:   ar = ax0 + ay0;    /* mx0 has al2, ay0,ax0 have al1 */
                             /* si has plt,mr0 has plt1,mr1 has plt2 */
          af = pass ar;
          ar = ar + af;
          se = -15;
          sr = ashift si(lo), ay0 = ar;       /* wd1 in ay0 */
          ay1 = sr0;                          /* sg0 in ay1 */
          sr = ashift mr0(lo);                /* sg1 in sr0 */
          ar = sr0 xor ay1;
          ar = ay0;
          if eq ar = -ay0;                    /* wd2 in ar */
          sr = ashift ar by -7(lo);
          ax0 = sr0;                          /* wd2 in ax0 */
          ax1 = 0x80;
          sr = ashift mr1(lo);                /* sg2 in sr0 */
          ar = sr0 xor ay1;
          ar = ax1;
          if ne ar = - ax1;
          af = pass ar;
          ar = ax0 + af;                      /* wd2 + wd3 = wd4 */
          my0 = 0x7f00;
          mr = mx0 * my0 (ss);
          ay1 = mr1;
          ar = ar + ay1;                      /* apl2 = wd4 + wd5 */
          ay0 = 0x3000;
          ar = abs ar;
          af = ar - ay0;
          if gt ar = pass ay0;    /* note: apl2 limited to ± .75 */
          if neg ar = -ar;
          rts;
```

(listing continues on next page)

```
uppol1:  ay0 = 0xc0;
         sr = ashift si by -15(lo);
         af = pass sr0;                        /* sg0 in af */
         sr = ashift mr0 by -15(lo);           /* sg1 in sr0 */
         ar = sr0 xor af;
         ar = ay0;
         if ne ar = - ay0;
         my0 = 0x7f80;
         mr = mx0 *  my0(ss), ay0 = ar;
         ar = mr1 + ay0;           /* apl1 before limits = wd1 + w2 */
         mr0 = ar;
         ax0 = 0x3c00;
         ar = ax0 - ay1;
         ay1 = ar;                 /* wd3 in ar, ay1 has apl2 */
         ar = abs mr0;             /* note: wd3 is always positive, */
                                   /* so abs value works */
         af = ar - ay1;
         if gt ar = pass ay1;
         if neg ar = -ar;
         rts;

limit:   ax1 = ar;
         af = pass ax1;
         ay1 = 0x3fff;
         ar = ax1 - ay1;
         ar = pass ar;
         if gt af = pass ay1;
         ay1 = 0xc000;
         ar = ax1- ay1;
         ar = pass ar;
         if lt af = pass ay1;
         ar = pass af;
         rts;

quanth:  sr = ashift ar by -15(lo);          /* sr0 has sih */
         af = pass sr0;
         if eq jump continue;
         ay0 = 0x7fff;
         ax0 = 0x7fff;
         af = ay0 - ar;
         ar = ax0 and af;

continue: af = pass 0,ay0 = ar;
         ay1 = 2;
         mx0 = 0x11a0;             /* q2(564) <<3 */
         mr = mx0 * my0 (ss);      /* this means mr1 will be 2.14 */
         ar = mr1 - ay0;           /* (q2(1)<<3*deth)-wd;if gt 0, */
                                   /* mih is 1) */
         if le af = pass 1;        /* af has mih */
         ar = pass af;
         af = pass 0, ax1 = ar;    /* clear af for processing */
```

```
                ar = pass sr0;
                if eq af = pass ay1;
                ar = ax1 + af;
                ay1 = ^bit_out2;
                ar = ar + ay1;
                i6 = ar;
                ax0 = pm(i6,m5);
                rts;

/* ...... INVQAH: inverse adaptive quantizer for the higher sub-band .....  */
        invqah:  mr0 = 0x650;
                mr1 = 0x1cf0;
                ar = pass ay0;
                if eq af = - mr1; /* save sih as flag for passing later */
                ar = ay0 -1;
                if eq af = - mr0;
                ay0 = ar;
                ar = ay0 -1;
                if eq af = pass mr1;
                ay0 = ar;
                ar = ay0 -1;
                if eq af = pass mr0;
                ar = pass af;
                mr = ar * my0 (ss);
                rts;

        logsch:  mr0 = 0xff2a;     /* these correspond to wh for 1 & 2 */
                mr1 = 0x31e;
                ar = pass ay0;
                if eq af = pass mr1;    /* save sih here as flag */
                                        /* for passing later */
                ar = ay0 -1;
                if eq af = pass mr0;
                ay0 = ar;
                ar = ay0 -1;
                if eq af = pass mr1;
                ay0 = ar;
                ar = ay0 -1;
                if eq af = pass mr0;          /* af has wh(ih2) */
                mr = mx0 * my0(ss);
                ar = mr1 + af;
                ay0 = 0x5800;
                ar = abs ar;
                af = ar - ay0;
                if gt ar = pass ay0;
                if neg ar = pass 0;
                rts;
```

(listing continues on next page)

```
chk_vals:ax1 = 0xc000;
         ax0 = 0x3fff;
         if av jump chk_ov;
         af = pass ar;
         ar = abs ar;
         if pos jump chk_pos;
         ar = ax0 + af;              /* if abs val is neg, execute this code */
         if lt af = pass ax1;
         ar = pass af;
         rts;

chk_pos: ar = af - ax0;
         if gt af = pass ax0;
         ar = pass af;
         rts;

chk_ov:  ar = pass ax0;
         if lt ar = pass ax1;
         rts;

.endmod;
```

Listing 5.1 Implementation Of The G.722 Algorithm

5.7 BENCHMARKS

Table 5.3 contains typical benchmarks for implementing Sub-band ADPCM (CCITT Recommendation G.722).

Memory Usage:	*PM RAM*	*DM RAM*
	1312 Locations	208 Locations

DSP	*Processor Speed*	*Number of Cycles*		*Execution Time*	*Processor Loading*
ADSP-2101	20 MHz	Encoder	821	41.05 µs	34.8%
		Decoder	742	37.10 µs	27.2%
		Total	1563	78.15 µs	62%
ADSP-2171	33 MHz	Encoder	821	24.63 µs	19.7%
		Decoder	742	22.26 µs	17.8%
		Total	1563	46.89 µs	37.5%

Table 5.3 Typical Benchmark Performance