Application Report
SPRA118 - May 2002

G.726 Adaptive Differential Pulse Code Modulation (ADPCM) on the TMS320C54x DSP

Peter Dent
Aaron Aboagye
C5000 DSP Software Applications

ABSTRACT

The G.726 is a voice-compression algorithm standard defined by the International Telecommunication Union (ITU). It can be used in many applications such as digital cordeless telephones, radio/wireless local loop, and pair-gain. The software package described in this application report is plug and play compliant with the multichannel framework using the TMS320C54x DSP.

Note: To license the code presented on this application note, please contact "MESI: Miller Engineering Services, Inc. http://www.mesi.net/

Contents

List of Figures

List of Tables

1 Introduction

Adaptive differential pulse code modulation (ADPCM) is a very efficient digital coding of waveforms. In telecommunication, the main field application is speech compression because it makes it possible to reduce the bit flow, while maintaining an acceptable quality. However, this technique applies for all waveforms, high-quality audio, image, and modem data. That is why it’s different from vocoders (voice encoders CELP, VSELP, etc.), that use properties of the human voice to reconstruct a waveform that appears very similar when it reaches the human ear, even though it is quite different from the original speech signal.

Consider a signal which you may wish to transform, to reduce the amount of information that it is necessary to code. You must be able to reconstruct the original signal as faithfully as possible. The principle of ADPCM is to use your knowledge of the signal in the past time to predict it in the future, the resulting signal being the error of this prediction. Applications of this principle are all based on digital transcoding after converting and coding analog signal to digital using pulse code modulation (PCM).

PCM is the most direct way to code analog signals to digital. The codeword in PCM is simply the quantized representation of the amplitude from the sampled signal. This word is given directly by an electronic A/D converter from the analog voltage which comprises the original signal.

You must perform PCM before ADPCM to decrease the number of bits for coding by passing through a PCM process before transforming to an ADPCM sample. In the G.726 recommendation, which currently includes G.721 and G.723 recommendations of the International Telegraph and Telephone Consultative Committee (CCITT), it is specified that an 8-bit PCM word should be reduced to a 4-bit ADPCM word, correspondingly reducing the bit flow by a factor of two.

Be aware, however, that an ADPCM word represents the prediction error of the signal, and has no significance itself.. It must be decoded (reverse transformation) to reconstruct the meaningful original waveform. For equal bit number coding, the difference between original and reconstructed signal is smaller in ADPCM than in PCM.

1.1 Introducing the Software Application

As previously stated, ADPCM is a complete digital transcoding process. According to the CCITT standard, if the PCM input bit flow is 64 kilobits per second (Kbps) (8 kHz sampling x 8-bit PCM word), you must process in real time, to produce a 40-, 32-, 24-, or 16-Kbps (8 kHz * 5, 4, 3, or 2-bit ADPCM word) output flow. A fixed-point digital signal processor (DSP) has an architecture which is capable of doing this. In particular, the new and very efficient TMS320C54x™ enables very rapid processing.

1.2 Software Features

- 16-, 24-, 32-, or 40-Kbps G.726 ADPCM (including G.721/G.723) in one single executable code, allowing rate switching during execution
- 14-bit linear PCM allowed, in addition to A-law and μ-law PCM input/output
- Possibility to implement several channels simultaneously in real-time processing (with time division sampling). This capability is possible during execution due to dynamic allocation of memory.
- 8.3–10 millions of instructions per second (MIPS) requirement (encoder + decoder), depending on program configurations (rate/PCM law choice at reset or at each sample, linear PCM or log-PCM). As a comparison, G.721 (only 32 Kbps, while 40 Kbps requires additional instructions) ADPCM on C50x requires 10.5 – 12.5 MIPS
- About 1500 words of read-only memory (ROM) requirement (program: 680, data: 850)
- Less than a data page of random-access memory (RAM) per channel (less than 100 words)
- Independent code with memory mapping
- Viable software (all CCITT test sequences successfully verified)
- Clear program organization allows configuration and optimization for a specific application and portability

2 CCITT ADPCM Standard: Recommendation G.726

This section is a reproduction of the G.726[1] Recommendation, section 1–3, developed by the International Telegraph and Telephone Consultative Committee (CCITT). The CCITT is a permanent organization of the International Telecommunication Union (ITU).

This section provides the principles and functional descriptions of the ADPCM encoding and decoding algorithms.

Two modifications have been made relating to the printed text of the recommendation. First, one detail concerning the 32-Kbps quantizer that takes one of 15 non-zero values (see section 2.4.3). Secondly, the transition detector equation has been corrected. In fact, the sampling index

[1] This recommendation completely replaces the text of Recommendation G.721 and G.723 published in Volume III.4 of the Blue Book. It should be noted that systems designed in accordance with the present recommendation will be compatible with systems designed in accordance with the Blue Book version.

seems to be (k–1), instead of (k) for y_l and a_2 (see section 2.4.12) [2].

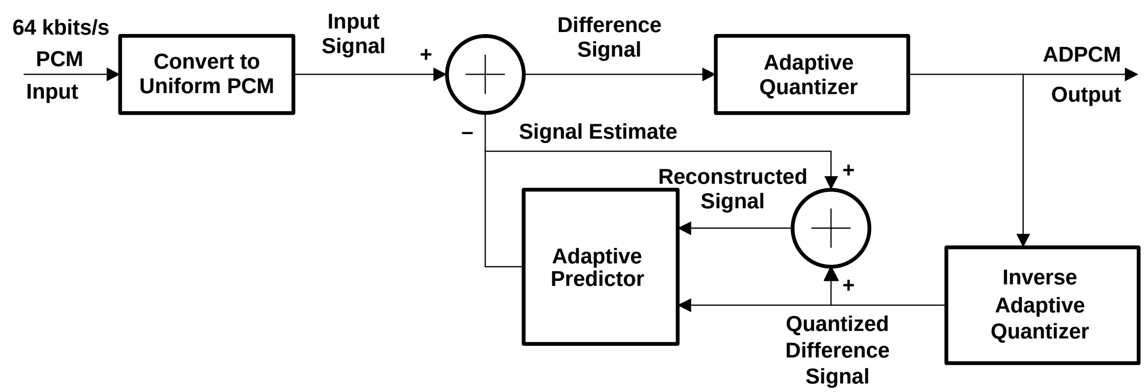
2.1 ADPCM Principle

The characteristics below are recommended for the conversion of a 64 Kbps A-law or μ-law pulse code modulation (PCM) channel to and from a 40-, 32-, 24- or 16-Kbps channel. The conversion is applied to the PCM bit stream using an ADPCM transcoding technique. The relationship between the voice frequency signals and the PCM encoding/decoding laws is fully specified in Recommendation G.711.

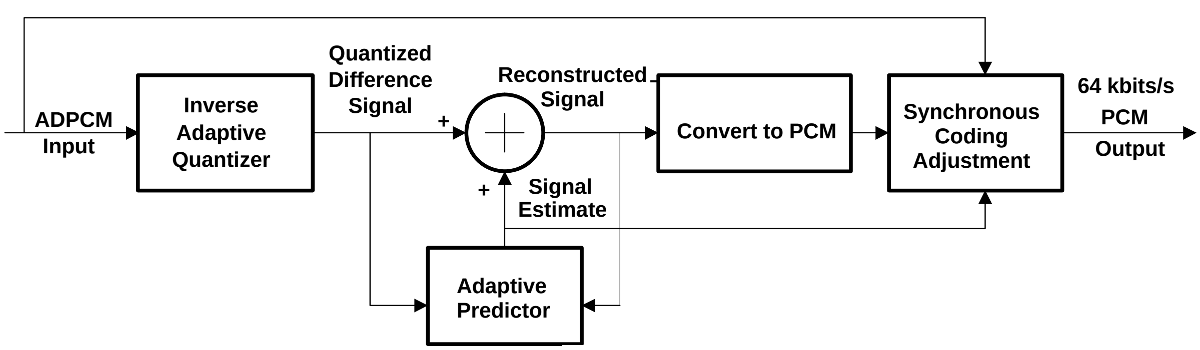
The principal application of 24- and 16-Kbps channels is for overload channels carrying voice in digital circuit equipment (DCME).

The principal application of 40-Kbps channels is to carry data modem signals in DCME, especially for modems operating at greater than 4800 Kbps.

Simplified block diagrams of both the ADPCM encoder and decoder are shown in Figure 1.



a) Encoder



b) Decoder

Figure 1. ADPCM Encoder and Decoder

[2] While reporting printing errors, specification of IMAG in the COMPRES sub-block (G.726/section 4, not reproduced here), probably contains an error. In the case of LAW = 1 and IS = 1, IMAG = (IM –1) >> 1, and not (IM+1) >> 1.

2.2 ADPCM Encoder

Subsequent to the conversion of the A-law or μ-law, PCM input signal to uniform PCM, a difference signal is obtained by subtracting an estimate of the input signal from the input signal itself.

An adaptive 31-, 15-, 7-, or 4-level quantizer is used to assign five, four, three, or two binary digits, respectively, to the value of the difference signal for transmission to the decoder. An inverse quantizer produces a quantized difference signal from these same five, four, three or two binary digits, respectively. The signal estimate is added to this quantized difference signal to produce the reconstructed version of the input signal. Both the reconstructed signal and the quantized difference signal are operated upon by an adaptive predictor, which produces the estimate of the input signal, thereby completing the feedback loop.

2.3 ADPCM Decoder

The decoder includes a structure identical to the feedback portion of the encoder, together with a uniform PCM to A-law or μ-law conversion and a synchronous coding adjustment.

The synchronous coding adjustment prevents cumulative distortion occurring on synchronous tandem coding (ADPCM, PCM, ADPCM, etc., digital connections) under certain conditions (see sectoin 2.5.7). The synchronous coding adjustment is achieved by adjusting the PCM output codes in a manner which attempts to eliminate quantizing distortion in the next ADPCM encoding stage.

2.4 Encoder Description

Figure 2 shows a block schematic for the encoder. For each variable to be described, *k* is the sampling index, and samples are taken at 125-μs intervals. A fundamental description of each block is given below in sections 2.4.1 through 2.4.12.

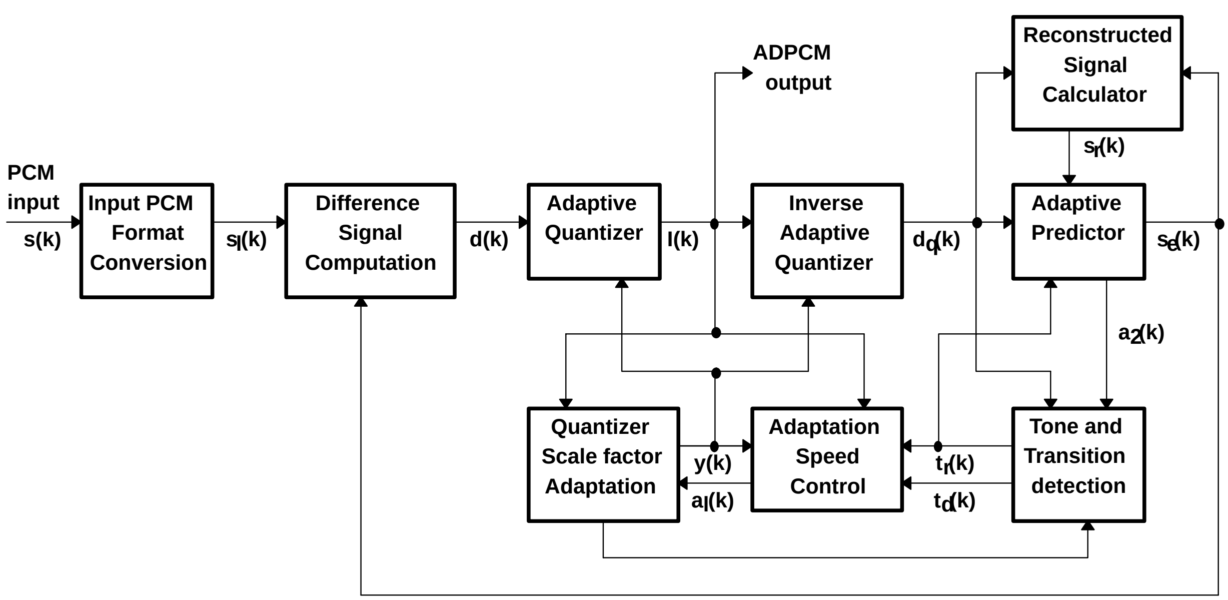


Figure 2. Encoder Block Schematic

2.4.1 Input PCM Format Conversion

This block converts the input signal *s(k)* from A-law or μ-law PCM to a uniform PCM signal, $s_l(k)$, if required.

2.4.2 Difference Computation

This block calculates the difference signal *d(k)* from the uniform PCM signal *sl(k)* and the signal estimate *se(k)*:

$$d(k) = s_l(k) - s_e(k) \tag{1}$$

2.4.3 Adaptive Quantizer

A 31-, 15-, 7-, or 4-level non-uniform adaptive quantizer is used to quantize the difference signal *d(k)* for operating at 40, 32, 24 or 16 Kbps, respectively. Prior to quantization, *d(k)* is converted to a base 2 logarithmic representation and scaled by *y(k)*, which is computed by the scale factor adaptation block:

$$d_{\ln}(k) = \log_2\left(d_l(k)\right) - y(k) \tag{2}$$

The normalized input/output characteristic (infinite precision values) of the quantizer is given in Table 1 through Table 4.

2.4.4 Operation at 40 Kbps

Five binary digits are used to specify the quantized level representing $d_{ln}(k)$ (four for the magnitude, and one for the sign in 2’s complement format). The 5-bit quantizer output *I(k)* forms the 40-Kbps output signal. *I(k)* takes on one of 31 non-zero values. *I(k)* is also fed to the inverse adaptive quantizer, the adaptation speed control and the quantizer scale factor adaptation blocks that operate on a 5-bit *I(k),* having one of 32 possible values. *I(k)* = 00000 is a legitimate input to these blocks when used in the decoder, due to transmission errors.

Table 1. Quantizer Normalized Input/Output Characteristic for 40 Kbps Operation

Normalized Quantizer Input Range $d_{ln}(k)$	|I(k)|	Normalized Quantizer Output $d_{qln}(k)$
[4.31, +∞)	15	4.42
[4.12, 4.31)	14	4.21
[3.91, 4.12)	13	4.02
[3.70, 3.91)	12	3.81
[3.47, 3.70)	11	3.59
[3.22, 3.47)	10	3.35
[2.95, 3.22)	9	3.09
[2.64, 2.95)	8	2.80
[2.32, 2.64)	7	2.48

NOTE: In Table 1 through Table 4, “[“ indicates that the endpoint value is included in the range, and “(“ or “)” indicates that the endpoint value is excluded from the range.

Normalized Quantizer Input Range $d_{ln}(k)$	|I(k)|	Normalized Quantizer Output $d_{qln}(k)$
[1.95, 2.32)	6	2.14
[1.54, 1.95)	5	1.75
[1.08, 1.54)	4	1.32
[0.52, 1.08)	3	0.81
[–0.13, 0.52)	2	0.22
[–0.96, –0.13)	1	–0.52
(–∞, –0.96)	0	–∞

NOTE: In Table 1 through Table 4, "[" indicates that the endpoint value is included in the range, and "(" or ")" indicates that the endpoint value is excluded from the range.

2.4.5 Operation at 32 Kbps

Four binary digits are used to specify the quantized level representing $d_{ln}(k)$ (three for the magnitude, and one for the sign in 2's complement format). The 4-bit quantizer output, *I(k* , forms the 32-Kbps output signal. *I(k)* takes on one of 15 non-zero values. *I(k)* is also fed to the inverse adaptive quantizer, the adaptation speed control and the quantizer scale factor adaptation blocks that operate on a 4-bit I(k), having one of 16 possible values. *I(k)* = 0000 is a legitimate input to these blocks when used in the decoder, due to transmission errors.

Table 2. Quantizer Normalized Input/Output Characteristic for 32 Kbps Operation

Normalized Quantizer Input Range $d_{ln}(k)$	|I(k)|	Normalized Quantizer Output $d_{qln}(k)$
[3.12, +∞)	7	3.32
[2.72, 3.12)	6	2.91
[2.34, 2.72)	5	2.52
[1.91, 2.34)	4	2.13
[1.38, 1.91)	3	1.66
[0.62, 1.38)	2	1.05
[–0.98, 0.62)	1	0.031
(–∞, –0.98)	0	–∞

2.4.6 Operation at 24 Kbps

Three binary digits are used to specify the quantized level representing $d_{ln}(k)$ (two for the magnitude, and one for the sign in 2's complement format). The 3-bit quantizer output *I(k)* forms the 24 Kbps output signal. *I(k)* takes on one of 7 non-zero values. *I(k)* is also fed to the inverse adaptive quantizer, the adaptation speed contro,l and the quantizer scale factor adaptation blocks that operate on a 3-bit *I(k)*, having one of 8 possible values. *I(k)* = 000 is a legitimate input to these blocks when used in the decoder, due to transmission errors.

Table 3. Quantizer Normalized Input/Output Characteristic for 24 Kbps Operation

Normalized Quantizer Input Range $d_{ln}(k)$	|I(k)|	Normalized Quantizer Output $d_{qln}(k)$
[2.58, +∞)	3	2.91
[1.70, 2.13)	2	2.13
[–0.06, 1.05)	1	1.05
(–∞, –0.06)	0	–∞

2.4.7 Operation at 16 Kbps

Two binary digits are used to specify the quantized level representing $d_{ln}(k)$ (one for the magnitude, and one for the sign in 2's complement format). The 2-bit quantizer output *I(k)* forms the 16 Kbps output signal. *I(k)* is also fed to the inverse adaptive quantizer, the adaptation speed control, and the quantizer scale factor adaptation blocks.

Table 4. Quantizer Normalized Input/Output Characteristic for 16 Kbps Operation

Normalized Quantizer Input Range $d_{ln}(k)$	|I(k)|	Normalized Quantizer Output $d_{qln}(k)$
[2.04, +∞)	1	2.85
[–∞, –2.04)	0	0.91

2.4.8 Inverse Adaptive Quantizer

A quantized version $d_q(k)$ of the difference signal is produced by scaling, using *y(k).* Specific values selected from the normalized quantizing characteristic are given in Table 1 through Table 4, transforming the result from the logarithmic domain:

$$d_q(k) = 2^{d_{q\,ln}(k)+y(k)} \tag{3}$$

2.4.9 Quantizer Scale Factor Adaptation

This block computes *y(k)*, the scaling factor for the quantizer and the inverse quantizer. The inputs are the 5-bit, 4-bit, 3-bit, 2-bit quantizer output, *I(k)*, and the adaptation speed control paramete,r $a_l(k)$.

The basic principle used in scaling the quantizer is bimodal adaptation:

- Fast for signals (that is, speech), that produce difference signals with large fluctuations;
- Slow for signals (that is, voiceband data tones), that produce difference signals with small fluctuations

The speed of adaptation is controlled by a combination of fast and slow scale factors.

The fast (unlocked) scale factor, $y_u(k)$, is recursively computed in the base 2 logarithmic domain from the resultant logarithmic scale factor, *y(k)*:

$$yu(k) = \left(1 - 2^{-5}\right) y(k) + 2^{-5}\, W|I(k)| \tag{4}$$

where $y_u(k)$ is limited by:

$$1.06 \le y_u(k) \le 10.00 \tag{5}$$

For 40-Kbps ADPCM, the discrete function, *W(I)*, is defined as follows (infinite precision values):

|I(k)|	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
W|I(k)|	43.50	33.06	27.50	22.38	17.50	13.69	11.19	8.81	6.25	3.63	2.56	2.50	2.44	1.50	0.88	0.88

For 32-Kbps ADPCM, the discrete function, *W(I)*, is defined as follows (infinite precision values):

|I(k)|	7	6	5	4	3	2	1	0
W|I(k)|	70.13	22.19	12.38	7.00	4.00	2.56	1.13	–0.75

For 24-Kbps ADPCM, the discrete function, *W(I)*, is defined as follows (infinite precision values):

|I(k)|	3	2	1	0
W|I(k)|	36.38	8.56	1.88	–0.25

For 16-Kbps ADPCM, the discrete function, *W(I)*, is defined as follows (infinite precision values):

|I(k)|	1	0
W|I(k)|	27.44	–1.38

The factor $(1–2^{–5})$ introduces finite memory into the adaptive process so that the states of the encoder and the decoder converge following transmission errors.

The slow (locked) scale factor, $y_l(k)$, is derived from $y_u(k)$, with a low pass-filter operation:

$$yl(k) = \left(1 - 2^{-6}\right) yl(k-1) + 2^{-6} y_u(k) \tag{6}$$

The fast and slow scale factors are then combined to form the resultant scale factor:

$$y(k) = a_l(k)\, y_u(k-1) + \left(1 - a_l(k)\right) y_l(k-1) \tag{7}$$

Where

$$0 \le a_l(k) \le 1 \tag{8}$$

2.4.10 Adaptation Speed Control

The controlling parameter, $a_l(k)$, can assume values in the range [0, 1]. It tends towards unity for speech signals, and towards zero for voiceband data signals. It is derived from a measure of the rate-of-change of the difference signal values.

Two measures of the average magnitude of $I(k)$ are computed:

$$d_{ms}(k) = \left(1 - 2^{-5}\right) d_{ms}(k-1) + 2^{-5} F\,|\,I(k)| \tag{9}$$

and

$$d_{ml}(k) = \left(1 - 2^{-7}\right) d_{ml}(k-1) + 2^{-7} F\,|\,I(k)| \tag{10}$$

For 40-Kbps ADPCM, $F|I(k)|$ is defined by:

|I(k)|	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F|I(k)|	6	6	5	4	3	2	1	1	1	1	1	0	0	0	0	0

For 32-Kbps ADPCM, $F|I(k)|$ is defined by:

|I(k)|	7	6	5	4	3	2	1	0
F|I(k)|	7	3	1	1	1	0	0	0

For 24-Kbps ADPCM, $F|I(k)|$ is defined by:

|I(k)|	3	2	1	0
F|I(k)|	7	2	1	0

For 16-Kbps ADPCM, $F|I(k)|$ is defined by:

|I(k)|	1	0
F|I(k)|	7	0

Thus, $d_{ms}(k)$ is a relatively short-term average of $F|I(k)|$, and $d_{ml}(k)$ is a relatively long-term average of $F|I(k)|$.

Using these two averages, the variable $a_p(k)$ is defined:

$$a_p(k) = \begin{cases} (1 - 2^{-4}a_p(k-1) + 2^{-3}, \mathit{if}\, |d_{ms}(k) - d_{ml}(k)| \geq 2^{-3} d_{ml}(k) \\ (1 - 2^{-4}a_p(k-1) + 2^{-3}, \mathit{if}\, y(k) < 3 \\ (1 - 2^{-4}a_p(k-1) + 2^{-3}, \mathit{if}\, t_d(k) = 1 \\ 1, \text{ if } t_r(k) = 1 \\ (1 - 2^{-4})a_p(k-1), \ \mathit{otherwise} \end{cases} \tag{11}$$

Thus, $a_p(k)$ tends towards the value 2 if the difference signal between $d_{ms}(k)$ and $d_{ml}(k)$ is large (average magnitude of $I(k)$ changing), and $a_p(k)$ tends towards the value 0 if the difference is small (average magnitude of $I(k)$ relatively constant). $a_p(k)$ also tends towards 2 for idle channel (indicated by $y(k) < 3$) or partial band signals (indicated by $t_d(k) = 1$ as described in section 2.4.12). Note that $a_p(k)$ is set to 1 upon detection of a partial band signal transition (indicated by $t_r(k) = 1$, see section 2.4.12).

$a_p(k)$ is then limited to yield $a_l(k)$ used in equation (7) above:

$$a_l(k) = \begin{cases} 1, & a_p(k-1) > 1 \\ a_p(k-1) & a_p(k-1) \leq 1 \end{cases} \tag{12}$$

This asymmetrical limiting has the effect of delaying the start of a fast-to-slow state transition, until the absolute value of *I(k)* remains constant for some time. This tends to eliminate premature transitions for pulsed input signals, such as switched carrier voiceband data.

2.4.11 Adaptive Predictor and Reconstructed Signal Calculator

The primary function of the adaptive predictor is to compute the signal estimate, $s_e(k)$, from the quantized difference signal, $d_q(k)$. Two adaptive predictor structures are used, a sixth order section that models zeros, and a second order section that models poles in the input signal. This dual structure effectively caters for the variety of input signals which might be encountered.

The signal estimate is computed by:

$$S_e(k) = \sum_{i=1}^{2} a_i(k-1)\,S_r(k-i) + S_{ez}(k), \tag{13}$$

Where

$$s_{ez}(k) = \sum_{i=1}^{6} b_i(k-1)d_q(k-i), \tag{14}$$

and the reconstructed signal is defined as:

$$S_r(k-i) = s_e(k-i) + d_q(k-i) \tag{15}$$

Both sets of predictor coefficients are updated using a simplified gradient algorithm for the second-order predictor:

$$a_l(k) = \left(1 - 2^{-8}\right)a_l(k-1) + 3.2^{-8}\, sgn[p(k)]sgn[p(k-1)], \tag{16}$$

$$a_2(k) = (1 - 2^{-7})a_2(k-1) + 2^{-7}\left\{sgn\left[p(k-2)\right] - f[a_1(k-1)\right]sgn[p(k)sgn[p(k-1)]\}, \tag{17}$$

Where

$$p(k) = d_q(k) + s_{ez}(k), \tag{18}$$

$$f(a_1) = \begin{cases} 4a_1, & |a_1| \le \frac{1}{2} \\ 2sgn(a_1), & |a_1| \prec \frac{1}{2} \end{cases}$$

and sgn[0] = 1, except sgn[*p(k–i)]* is defined to be 0 only if *p(k–i)* = 0 and i = 0, with the stability constraints:

$$|a_2(k)| \le 0.75 \textit{ and } |a_1(k) \le 1 - 2^{-4} - a_2(k) \tag{19}$$

If $t_r(k)$ = 1 (see section 2.4.12), then $a_1(k) = a_2(k) = 0$.

For the sixth-order predictor:

$$b_l(k) = \left(1 - 2^{-8}\right)b_l(k-1) + 2^{-7}\, sgn\left[d_q(k)\right]sgn\left[d_q(k-i)\right], \tag{20}$$

for i = 1, 2, ..., 6.

For 40-Kbps coding, the adaptive predictor is changed to decrease the leak factor used for zeros coefficient operation. In this case, the previous equation becomes:

$$b_I(k) = \left(1 - 2^{-9}\right)b_I(k-1) + 2^{-7}\,sgn\left[d_q(k)\right]sgn\left[d_q(k-i)\right], \tag{21}$$

If $t_r(k) = 1$ (see section 2.4.12), then $b_1(k) = b_2(k) = ... = b_6(k) = 0$.

As stated above, sgn[0] = 1, except sgn[$d_q(k–i)$] is defined to be 0 only if $d_q(k–i) = 0$ and i = 0. Note that $b_i(k)$ is implicitly limited to $\pm$ 2.

2.4.12 Tone and Transition Detector

To improve performance for signals originating from frequency shift keying (FSK) modems operating in the character mode, a two-step detection process is defined. First, partial band signal (that is, tone) detection is invoked so that the quantizer can be driven into the fast mode of adaptation:

$$t_d(k) - \begin{cases} 1, & a_2(k) < -0.71875 \\ 0, & otherwise \end{cases} \tag{22}$$

In addition, a transition from a partial band signal is defined so that the predictor coefficients can be set to zero, and the quantizer can be forced into the fast mode of adaptation:

$$t_r(k) - \begin{cases} 1, & a_2(k-1) < -0.71875 \text{ and } \left|d_q(k)\right| > 24.2^{yl(k-1)} \\ 0, & otherwise \end{cases} \tag{23}$$

2.5 Decoder Description

Figure 3 is a block schematic of the decoder. A functional description of each block is given in section 2.5.1 through section 2.5.7.

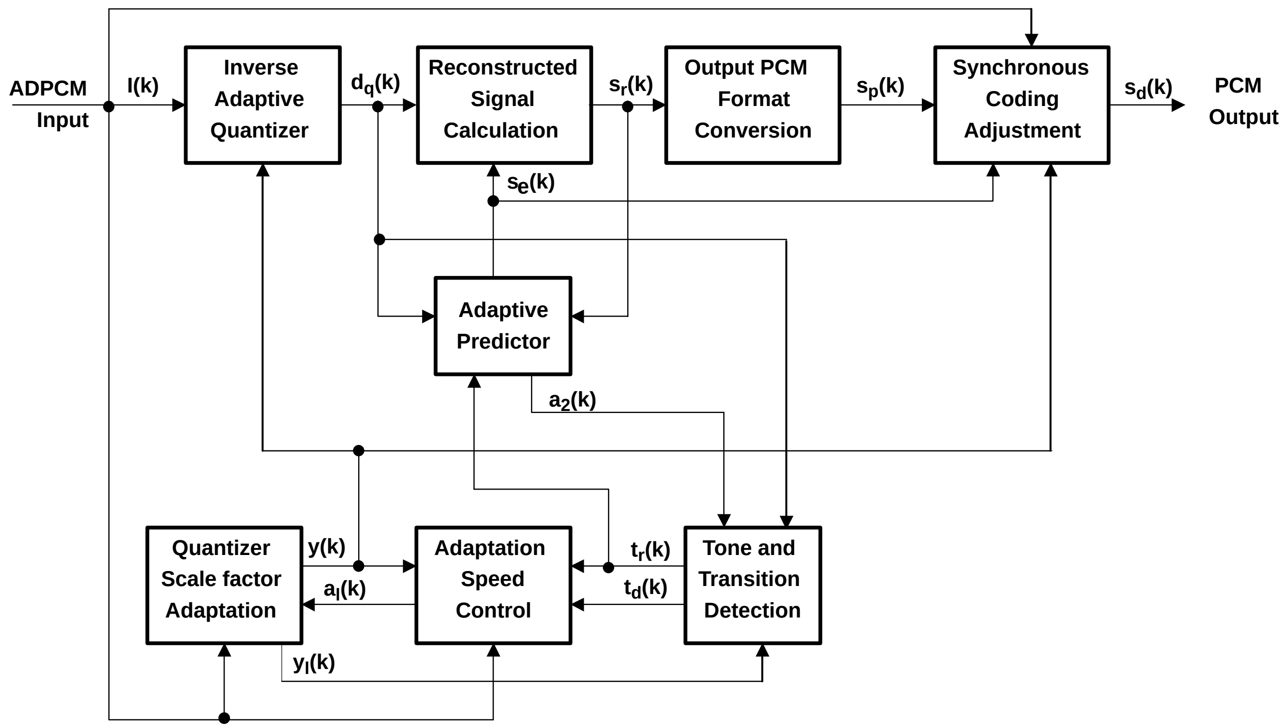


Figure 3. Decoder Block Schematic

2.5.1 Inverse Adaptive Quantizer

The function of this block is described in section 2.4.8.

2.5.2 Quantizer Scale Factor Adaptation

The function of this block is described in section 2.4.9.

2.5.3 Adaptation Speed Control

The function of this block is described in section 2.4.10.

2.5.4 Adaptive Predictor and Reconstructed Signal Calculator

The function of this block is described in section 2.4.11.

2.5.5 Tone and Transition Detector

The function of this block is described in section 2.4.12.

2.5.6 Output PCM Format Conversion

This block converts the reconstructed uniform PCM signal, *sr(k* , into an A-law or µ-law PCM signa,l *sp(k)*, as required.

2.5.7 Synchronous Coding Adjustment

The synchronous coding adjustment prevents cumulative distortion occurring on synchronous tandem codings (ADPCM, PCM, ADPCM, etc. digital connections), when:

1. The transmission of the ADPCM and the intermediate 64 Kbps PCM signals is error free, and,
2. The ADPCM and intermediate 64-Kbps PCM bit streams are not disturbed by digital signal processing devices.

If the encoder and decoder have different initial conditions (as may occur after switching, for example), then the synchronous tandeming may take time to establish. Furthermore, if this property is disturbed, or not acquired initially, then it may be recovered for those signals of sufficient level with spectra that occupy the majority of the 200 Hz to 3400 Hz band (that is, speech, 4800-bit/s voiceband data).

When a decoder is synchronously connected to an encoder, the synchronous coding adjustment block estimates quantization in the encoder. If all state variables in both the decoder and the encoder have identical values, and there are no transmission errors, the forced equivalence of both 4-bit quantizer output sequences for all values of *k* ensures the property of non-accumulation of distortion.

This is accomplished by first converting the A-law or μ-law signal, $s_p(k)$, to a uniform PCM signal, $s_{lx}(k)$, and then computing a difference signal, $d_x(k)$:

$$d_x(k) = s_{lx}(k) - s_e(k)\,, \tag{24}$$

The difference signal, $d_x(k)$, is then compared to the ADPCM quantizer decision interval, determined by *I(k)* and *y(k).* the signal $s_d(k)$ is then defined as follows:

$$s_d(k) = \begin{cases} s_p^+(k), & d_x(k) < \textit{lower interval boundary} \\ s_p^-(k), & d_x(k) \geq \textit{upper interval boundary} \\ s_p(k)\,, & \textit{otherwise} \end{cases} \tag{25}$$

Where

$s_d(k)$ is the output PCM codeword of the decoder

$s_p^-(k)$ is the PCM codeword that represents the next, more positive, PCM output level. When sp(k) represents the most positive output level, then $s_p^+(k)$ is constrained to be the value $s_p(k)$.

$s_p^-(k)$ is the PCM codeword that represents the next, more negative, PCM output level. When sp(k) represents the most negative output level, then $s_p^-(k)$ is constrained to be the value $s_p(k)$.

3 Useful Features of the C54x for G.726 ADPCM

The typical application for the C54x is for vocoders that deal with a large number of samples at the same time. Application-oriented instructions, such as LMS, FIRS, SQUR, CMPS, or instruction with parallel load/store, do not take place naturally in the ADPCM algorithm. On the other hand, instructions, such as EXP, NORM, MIN, MAX, are often very useful for this purpose. More generally, the ADPCM algorithms benefit from the enhanced architecture of the C54x, which also provides advantages in general purpose applications. The following list sums up the principal features of the C54x™ used for the CCITT ADPCM algorithm:

- The two accumulators often make it possible to perform parallel treatments and decrease the number of memory accesses (for temporary storage).
- The eight auxiliary registers, which are all simultaneously active, simplify the use of indirect addressing.
- The 40-bit ALU makes it possible to avoid overflow when shifting the accumulator (used in floating-point multiplication when scaling the result).
- Dual data-memory access, using the two or three data buses, makes some calculations faster (used in quantization routine). Also, dual data-memory operand (when used in indirect addressing) allows some instructions to have a one-word length instead of two (in particular load, store, add, sub with left shift), which makes them one-cycle instructions.
- Circular addressing is easy to use. In fact, circular addressing is specified in the instruction word. Moreover, the corresponding buffer is automatically determined (using its memory location), simply by specifying its size (value of the BK register). Two circular buffers would be implemented for the delayed variables d_q(k-i) and s_r(k-i).
- Long-word arithmetic capability will be used for the variable y_l(k) (that requires more precision). It will be used as dual 16-bit operand, when two adjacent variables are calculated (for example, initialization of predictor coefficients, if a transition is detected).
- On-chip data-ROM capability, allows the storage of large tables of constant values, giving the possibility of data addressing.
- The integrated compare unit provides two particularly useful instructions, MIN and MAX. These instructions allow the limitation of the different coefficients, with a minimum of cycles.
- The EXP instruction makes it unnecessary to perform a *iterative* search for the most significant bit. It is used for floating-point conversion (G.726 ADPCM requires floating-point multiplication for the predictor filters), as well as for log-conversion (before quantizing, and for log-PCM compression). The NORM instruction is often associated with EXP to normalize a variable.

Now, you will see modules whose implementation on the C54x requires some comment.

3.1 Input/Output PCM Format Conversions

The ADPCM algorithm works with actual linear PCM inputs/outputs, while the standard format for digital telephony is either A or µ-law, which are logarithmic laws of quantization. The CCITT gives these conversion laws in the G.711 recommendation. However, linear/logarithmic PCM conversions are included in the CCITT ADPCM recommendation (G.726) to make the PCM inputs/outputs *consistent* with the algorithm. There are two reasons for this:

First, a word converted from A-law PCM has only 13 bits, while one word produced from µ-law is a 14-bit word. The ADPCM algorithm works with a resolution of 14 bits for PCM input words. To avoid the loss of precision, PCM words coming from A-law are also scaled into 14-bit words.

C54x is a trademark of Texas Instruments.

Secondly, a synchronous coding adjustment module is included at the output of the decoder. It finds its origin in the non-reciprocity between linear PCM word (14 bits) and logarithmic PCM word (8 bits). The problem, for the decoder, is to choose a log-PCM output word, SD, that is actually representative of the ADPCM input word I. This means that if you give SD as the input encoder, you also have to find I as output. This feature is already ensured for the linear PCM output word, because of the feedback of the ADPCM quantization error in signal estimation (the encoder also includes a decoding module). However, this property is no longer ensured after converting linear PCM into log-PCM, due to the error of this logarithmic quantization. The synchronous coding adjustment module ensures that this feature is maintained. This allows multiple encoding-decoding-encoding without adding distortion.

As shown below, these format conversions and corrections are implemented in the C54x for both G.726 and G.711 recommendations. The routines are valid for either A-law or μ-law. Tables and specific variables make the distinction between the two.

3.1.1 Log-PCM Companding

This module converts a linear PCM word into the logarithmic domain. As you saw, a word coming from A-law has been one bit left-shifted, to get the maximum resolution in the ADPCM algorithm. Now you have to do the reverse transformation before converting it to A-law. This transformation includes rounding for negative words by subtracting one from the magnitude before right-shifting. When considering that for small signals (< 64) a linear quantization is required for A-law simply by dividing sample magnitude by two, the total right shift to apply for A-law is thus 2 bits. For A-law large signals, this 2-bit right shift is also applied, and then compensated further.

Use the variable LAWBIAS (= 33 for μ-law, = 0 for A-law). Added to the linear PCM word, this variable allows you to use powers of two as quantizer decision values. For μ-law also, it avoids the need to branch to linear quantization, which is required for magnitudes smaller than 32.

The logarithm calculation is quickly performed using the EXP and NORM instructions. See § () for the method. The first difference is that normalization is done in Q4 format instead of Q7. As you compute logarithmically only for large magnitudes (≥ 32), you decrease the dynamic by storing only the segment of the word. This segment is defined by:

segment = (exponent – 1) – segment offset,
where *exponent* is defined in section 3.2.1 or in section 3.4[3]

This makes it possible to code it with only three bits instead of four. The variable LAWSEG allows us to complete the logarithm transformation, including the segment offset (4 for A-law, 5 for μ-law) and the compensation left shift for A-law (see above).

Restore the sign to the magnitude by adding 128 for positive PCM words. Lastly, invert bits (only even bits for A-law), to satisfy transmission practices; this is done by means of the variable LAWMASK (0x55 for A-law, 0x7F for μ-law).

The companding routine is shown below, with the section of code added specifically to meet the requirements of G.726 highlighted in bold characters. The rest is sufficient for G.711. When code shown in italic characters is suppressed, the routine performs μ-law compression in only 13 cycles.

```
*********************************************************************************
* Converts signal from uniform PCM to a A-law or Mu-law PCM signal with format    *
* correction                                                                       *
*                                                                                  *
*      INPUT:                                                                      *
* A          = SR(k) Reconstructed signal                                         *
* LAW        = LAW (0 for Mu-law, 1 for A-law)                                    *
* LAWBIAS    = Bias constant (=0 for A-law, =33 for Mu-law)                       *
* LAWSEG     = Constant in order to compute the segment of the PCM word           *
* LAWMASK    = Magnitude mask for A-law or Mu-law PCM word                        *
*                                                                                  *
*     OUTPUT:                                                                      *
* SD = A     = SP(k) A-law or Mu-law PCM reconstructed signal                     *
*                                                                                  *
*     CYCLES:             Min: 20, Max: 26                                         *
*                         only G711: 21                                            *
*                         only µ-law: 13                                           *
*********************************************************************************
SYNC  STH   A, *AR5       ; Store sign of SR
      BIT   *AR5, 0       ; TC = 0 if SR positive
      LD    LAW, B        ; If LAW = 1: A-law
      ABS   A             ; A = |SR| = IM
      AND   C32767, A     ; If RATE = 40, SR = 8000 can occur: overflow
      XC    1, ALT        ;
      SUB   LAW, A        ; If SR < 0, subtract 1 to IM for A-law
      ABS   A             ; Ensures A positive (case SR = 8000)
      XC    1, BNEQ       ; A-law: one shift for 12-bit unsigned word,
      SFTA  A, -2         ; and one shift for linear quantization (SFTA A, -1 for G.711)
      ADD   LAWBIAS, A    ; Add Bias
      SUB   C32, A, B     ;
      BC    ECOMP, BLT    ; Linear quantization for A-law if IMAG < 32
      EXP   A             ; TREG = 31 – EXP
      LD    LAWSEG, B     ; LAWSEG =24*2^4 for Mu-law and TREG latency
      NORM  A             ; A = (SR << (15-EXP)) << 16
      SFTA  A, –10        ; A = (SR << (4–(EXP–1))) << 16
      MAS   C16, B        ; B = 24*2^4 – (TREG*2^4) = (EXP–7) << 4
      ADD   A, –16, B ; B = |SP|
      LD    C127, A       ; Load SPmax
      MIN   A             ; Saturate if |SR| out of range
ECOMP XC    1, NTC        ; Test if SR positive
```

```
ADD   C128, A       ; Add bit sign if positive
XOR   LAWMASK, A    ; Apply law mask
STL   A, SD         ; SD = SP = A-law or Mu-law PCM word
```

Note that in logarithmic calculation, you use (*exponent* – 1), instead of *segment* (see section 3.4).

3.1.2 Linear PCM Expanding

This module converts a PCM word from logarithmic domain to linear domain.

To reduce the clock cycles timing, tables are used for this PCM expansion. As outputs levels are symmetrical relative to zero, use the magnitude of the log-PCM word as an offset table. This property limits the table to 128 words. One table is used for the A-law PCM (ALAW), and another is used for μ-law PCM (MULAW). As you saw above, the A-law table directly gives the magnitude of the linear PCM word multiplied by two, making it a 13-bit unsigned number as required.

The original sign is then introduced, so that these samples are now in 14-bit, two-complement format.

The routine shown here executes in 13 cycles; two of these cycles can be used to execute initialization instructions for the following blocks. For example, if you perform G.726 ADPCM without linear capability, this routine is always performed, so you can, for example, replace the NOP instructions by the initialization of the block repeat counter BRC, used for the ADPCM quantization (see section 3.5).

```
***************************************************************************************
*Converts signal from A-law or Mu-law PCM to uniform PCM signal with format correction*
*                                                                                     *
*     INPUT:                                                                          *
* A          = S(k) input signal (encoding)                                           *
*            = SP(k) A-law or Mu-law reconstructed signal (decoding)                  *
* LAWMASK    = Magnitude mask for A-law or Mu-law PCM word                            *
* ADLAW      = Law table address in Data-ROM                                          *
* xLAW (*AR2)= Law inverse quantizing table (x = MU or A)                             *
*                                                                                     *
*     OUTPUT:                                                                         *
* A          = SL(k) linear input signal (encoding)                                   *
*            = SLX(k) linear output signal (decoding)                                 *
*                                                                                     *
*     CYCLES: 13(actually 11 if replacing NOP latency by instructions of other blocks)*
***********************************************************************************
EXPAN XOR    LAWMASK, A   ; Invert even bits if A-law
      SFTA   A, –7, B     ; B = 1 if SP positive, 0 if SP negative
      AND    C127, A      ; A = unsigned magnitude
      ADDS   ADLAW, A     ; ADLAW = address of inverse log quantizer
      STLMA, AR2          ; AR2 = address of linear PCM word
```

```
NOP                 ; AR2 update latency. If this routine is always performed,
NOP                 ; replace NOP by instructions of other blocks
                    ; (e.g: initializations)
LD *AR2, A          ; A = linear PCM magnitude
RETD                ;
XC 1, BEQ           ; Convert A in two-complement,
NEG A               ; depending on original sign of A
```

3.1.3 Synchronous Coding Adjustment

The synchronous coding adjustment is performed at the end of the decoder module by re-encoding the output PCM word *SP* and comparing the new code *ID* with the original ADPCM word *I*.

This re-encoding is performed by using the same code routines as those used for the encoder (PCM expanding, difference signal calculation, logarithm conversion, adding quantizer scale factor, then quantization). *I* format is two-complement, but only with the assigned bits for it (2, 3, 4, or 5), and without sign extension along the 16 bits of the processor. So this format is changed, by using a table, to make the comparison with *ID* possible.

- If $ID > I$, then *SP* is overestimated, so choose *SP* – 1 as new value of *SP*.
- If $ID < I$, then *SP* is underestimated, so choose *SP* + 1 as new value of *SP*.
- Otherwise, *SP* is correctly estimated, and keeps its value.

Note that *SP* is a signed magnitude, and not two-complement. This does not allow you to perform normal arithmetic. For instance, SP + 1 when SP is negative and is obtained with *SP* – 1.

Another point is that log-PCM format includes two different values of zero: positive 0 (0^+), and negative zero (0^-). As a consequence, *SP* – 1 for *SP* = 0^+ is 0^-, and *SP* + 1 when *SP* = 0^- is 0^+. That is only true for A-law; for μ-law, *SP* – 1 for *SP* = 0^+ is –1, and *SP* + 1 when *SP* = 0^- is +1. [4]

Lastly, overflows must be avoided. This means that if *SP* = 0x7F, then *SP* + 1 is 0x7F, too. The following code shows a solution for this module (at the time when ID is already calculated).

```
**********************************************************************************
*      Perform reconstructed signal adjustment                                   *
*                                                                                *
*      INPUT:                                                                    *
* B             = ID(k) ADPCM code from re-encoded output PCM sample             *
* IQUAxx (*AR1) = Inverse quantizer table, gives here the magnitude of I (IM)    *
* AR1           = Address of IM in IQUAxx table                                  *
* SD            = SP(k) A-law or Mu-law PCM reconstructed signal                 *
* LAWMASK       = Magnitude mask for A-law or Mu-law PCM word                    *
*                                                                                *
*     OUTPUT:                                                                    *
```

```
* A             = SD(k) Decoder PCM output word                                    *
*                                                                                  *
*     CYCLES: Min: 6 (usual), Max: 25                                              *
************************************************************************************

      SUB     *AR1, B       ; B = ID – IM
      RCD     BEQ           ; If IM = ID, SD = SP and do not change SD
      LD      SD, A         ; Load output PCM word
      STH     A, 9, *AR5    ; *AR5 = sign variable = 1 if SP positive, 0 else
      XOR     LAWMASK, A    ; Apply law mask to obtain magnitude of SP
      AND     C127, A       ; A = unsigned magnitude of SP (C127 = 127)
      BC      SP0, AEQ      ; |SP| = 0 is a special case: go to SP0
      BIT     *AR5, 15      ; Test original sign of SP
      LD      #1, B         ; Load gain to prepare SP adjustment
      XC      1, BGT        ; If ID > IM, case SD = SP–
      NEG     B             ; In this case, negate the gain
      ADD     B, A          ; Add the gain if SP positive
      XC      1, NTC        ; TC = 0 if SP negative
      SUB     B, 1, A       ; In this case, subtract the gain
      LD      C127, B       ; A = |SD| is compared to |SDmax|
      MIN     A             ; Saturate the result if |SD| > 127
      RETD                  ; Return from subprogram with A = SD
      ADD     *AR5, 7, A    ; Add sign extension
      XOR     LAWMASK, A    ; Invert bits depending on the law
SP0   LD      SIGN, A       ; Case |SP| = 0
      AND     LAW, A        ; If Mu-Law, LSB of SP is 0
      XOR     C1, A         ; SP–(0+) = –1 for Mu-law, 0– else; and
      XOR     LAWMASK, A    ; SP–(0–) = –1 for both laws
      LD      C128, B       ; Load positive sign for SP+ case
      XC      2, BLT        ; case SD = SP+: LSB = 1, only if SIGN = 0
      ADD     LAW, B        ; SP+(0–) = +1 for Mu-law, 0+ else; and
      XOR     B, A          ; SP+(0+) = +1 for both laws
      RET                   ; A = SD, NB: B = ID – IM < 0 for SD = SP+
```

This subtlety is explained in 0.

3.2 Floating-Point Features: Conversion, Storage, and Multiplication

The floating-point module concerns the predictor that computes an estimation of the signal by applying adaptive filters to delayed variables. Coefficients of the predictor filters (a_1, a_2, b_i for $i = 1, ..., 6$), are between –2 and 2 in Q14 format, while variables (delayed reconstructed signals $s_r(k–i)$ and delayed quantized difference $d_q(k–i)$) are between –32768 and 32767 in Q0 format. All of these variables use a maximum resolution of 16 bits; however, there is a large difference of scale between coefficients and variables.

A fixed-point multiplication would be inadequate (too much precision for high levels, not enough for low levels). The floating-point format permits better management of the dynamic gaps. Here, the resolution is limited to 6 bits (6 bits mantissa), but the dynamic is greater (0 is coded as 32 (1/2) for mantissa and 0 for exponent). As a consequence, if a variable has a zero value, the floating-point product with the corresponding coefficient would not always be zero, which is always the case with a fixed-point multiplication.

It is described now how floating-point format can be challenged by the C54x for the G.726 recommendation.

3.2.1 Floating-Point Format Storage

Floating-point number characteristics are as follows: sign, exponent, mantissa. These features are defined by:

|number| = denormalized mantissa * 2^{exponent}, and sign(number) = sign

If number = 0, this equality is not verified (0 value is coded in floating-point format, as it was actually ½). The following inequalities are verified:

$\frac{1}{2} \leq$ denormalized mantissa < 1. Mantissa is normalized with 6 bits representation, so:

$32 \leq$ mantissa < 64, where mantissa represents the 6 most significant bits of the fixed-point number, except for zero. As for exponent, you have:

$2^{\text{exponent}-1} \leq |\text{number}| < 2^{\text{exponent}}$. In practice, the number of the most significant bit plus one (when LSB number is zero).

To make the access faster, use three (successive) words, instead of one, to code a floating-point number: one word for the exponent (Q0 with 4 significant bits), one word for the mantissa (Q6 with 6 significant bits), and one more for the sign (Q0 with 0 significant bits). The sign is coded as an arithmetic sign, and is 0 for positive and null values, –1 for negative values (see section 3.10).

3.2.2 Floating-Point Conversion

When loaded into the accumulator, the sign of a word (as defined in section 3.10) is the value of the high part of accumulator. The sign is thus extracted, due to the *STH* instruction. Then, compute the magnitude with the *ABS* instruction. Now the instructions *EXP* and *NORM* are useful for computing exponent and mantissa. *EXP* calculates the number of non-significant bits relative to the first 32 bits of the accumulator, and stores the result in the temporary register TREG. The wanted value of exponent is thus:

exponent = 31 – TREG for a word different from zero.

When associated with *DSUBT* that subtracts TREG from a variable that is here set to 31, you directly obtain the value of the exponent (note that *DSUBT* actually uses a long-word; the high part of this word must be at an even address, while the low part is set to 31).

This method does not apply if the input word to convert is 0. *EXP* set TREG to zero in this case.

Note also that *DSUBT* needs one cycle latency to use the TREG value computed by *EXP*. This feature has been underlined with the evaluation module; however, this latency was unnecessary for the simulator.

Mantissa is calculated by means of the *NORM* instruction that uses the TREG value (31 – exponent) to normalize the word in accumulator (TREG value left-shift). Then, a 9-bit right-shift gives the wanted 6-bit mantissa in the high part of the accumulator (bits 16 to 21).

The following code gives the floating-point conversion for the reconstructed signal $s_r(k)$. It takes 12 cycles to execute. One instruction (in bold characters) has been added to satisfy 40 Kbps (possibility of overflow on SR).

```
**********************************************************************************
*      Convert fixed-point number to floating-point format                      *
*                                                                                *
*      INPUT:                                                                    *
* SD          = SR(k): Reconstructed signal in two-complement format            *
* AR6         = Address of SR(k–2) sign                                         *
*                                                                                *
*                        OUTPUT:                                                 *
* SRFLOAT (*AR6)= SR((k+1)–1) exponent, mantissa and sign                       *
*                                                                                *
*      CYCLES: 13                                                                *
**********************************************************************************
      LD       SD, B          ; Load reconstructed signal
      ABS      B, A           ; A = |SR|
      AND      C32767, A      ; exp <=15, for RATE=40, SR=8000 is overflow
      EXP      A              ; TREG = 31 – EXP(|SR|)
      STH      B, *AR6–       ; Store sign of SR (0 if >= 0, –1 if negative)
      NORM     A              ; A = (|SR| mantissa) << 9
      DSUBT    C31–1, B       ; BL = 31 – TREG = EXP(|SR|)
      XC       2, AEQ         ; if SR = 0
      LD       C16384, 16, A  ; then normalize A to obtain mantissa(SR)=32
      LD       #0, B          ; and set B to zero to obtain EXP(|SR|) = 0
      STH      A, –9, *AR6–   ; Store |SR| mantissa (6 bits)
      STL      B, *AR6+0%     ; Stores EXP of |SR| (4 bits)
```

3.2.3 Floating-Point Multiplication

This routine has a crucial importance in CCITT ADPCM timing. The sixth-order FIR filter and the second-order IIR filter are concerned so that eight floating-point multiplications have to be performed. With 25 clock cycles per routine, it totals 200 clock cycles, equivalent to one-third of the global coding process.

The routine also includes floating-point conversion of predictor coefficients (truncated in Q12 format). The principle of this conversion is the same as explained above. Mantissas of the two operands (Q6 format) are multiplied to form the product mantissa (Q12), which is then truncated in Q8 format. Exponents of the two operands are added to form the exponent of the product. The result is immediately converted into two-complement format, including an 11-bit right-shift, for scaling it in Q1 format, and sign calculation. In fact, accumulation of these products is executed in fixed-point format.

Each partial product is limited to 16 bits, which means that the contribution of each one to forming the signal estimate is limited to half of the greatest possible value for the reconstructed signal $s_r(k)$. This property is also true for the global signal estimate. However, signal estimate value could reach twice the greatest input PCM value.

The code bellow shows one of the eight floating-point multiplications.

```
*************************************************************************************
* Compute a2(k) * sr(k–2) in floating-point format                                  *
*                                                                                   *
*     INPUT:                                                                        *
* SRFLOAT (*AR6)          = SR(k–2) exponent, mantissa and sign                     *
* A2                      = A2(k–1)                                                 *
* AR6                     = SR(k–2) address                                         *
*                                                                                   *
*                         OUTPUT:                                                   *
* B                       = WA2(k) = A2(k) * SR(k–2)                                *
* AR6                     = SR(k–1) address                                         *
*                                                                                   *
*     CYCLES: 24                                                                    *
*************************************************************************************
      LD      A2, –2, B     ; truncate A2 (Q12 2’comp: S,0,...–12)
      ABS     B, A          ; A = |A2|, (|A2>>2| in fact)
      EXP     A             ; If A > 0, TREG = 31 – EXP(|A2|)
      STH     B, SIGN       ; SIGN = sign(A2) (0 if >= 0, –1 if negative)
      DSUBT   C15–1, B      ; BL = 15 – TREG = EXP(|An|) – 16
      NORM    A             ; AH =(|A2| mantissa) << 9
      XC      2, AEQ        ; If A = 0, TREG = 0 and AH = 0 then
      LD      C16384, 16, A ; normalize A mantissa = 32 << 9 for A = 0
      LD      M16, B        ; and set B to –16, to obtain B = EXP – 16
      ADD     *AR6+, B      ; B = EXP(SR2) + EXP(A2) – 16 = WA2EXP – 16,
      STL     B, *AR3       ; save WA2EXP – 16 in *AR3
      SFTA    A, –9         ; AH = A2MANT = mantissa of A2 (6 bits)
      MPYA    *AR6+         ; B = A2MANT*SR2MANT (SR2MANT = SR2 mantissa)
      ADD     C48, B        ; Add 48 for preparing rounding
      LD      *AR3, T       ; T  = WA2EXP – 16
      LDU     *AR6+%, A     ; A = sign(SR2)
      XOR     SIGN, A       ; A = sign(SR2) * * sign(A2)
      SFTA    B, –4         ; B = mantissa of product = WA2MANT (Q8)
      NORM    B             ; WA2MANT<<(EXP–16)=WA2 magnitude (WA2MAG)<<3
      SFTA    B, –3         ; Complete scaling (–8 –3 = –11) for WA2MAG
      AND     C32767, B     ; Avoid 16- or 17-bit results for WA2EXP > 26
```

```
XC      1, ANEQ       ; Introduce sign of product
NEG     B             ; B = WA2 = A2 * SR2 (Q1 2’comp:S,13,...,–1)
```

3.3 Delayed Variables Management, Use of Circular Buffers

Among the variables that have to be delayed, the partial product estimate p(k), the floating-point versions of the quantized difference $d_q(k)$, and the reconstructed signal $s_r(k)$ have to be delayed several times. So, one location in memory is insufficient for all these variables. p(k) has to be delayed twice, so choose two successive locations for it. The instruction *DELAY* of the C54x immediately delays p(k–1) to p(k–2). $S_r(k)$ has also to be delayed twice but, as you saw, 3 successive locations memory are used for one sample, as well as for $d_q(k)$, which has to be delayed six times.

Two circular buffers are therefore used for these two variables: SRFLOAT for $s_r(k)$, and DQFLOAT for $d_q(k)$.

SRFLOAT has a 6-word length, while DQFLOAT has an 18-word length. These values have to be stored in BK (circular buffer size) before using these buffers. After that, the access of the buffer is executed using circular addressing (noted *ARx%). In this addressing mode, it is only necessary to note the location of the new (or the oldest) delayed variable; physical limits of the buffer are automatically managed. For this, the CPU assumes the physical buffer begins on a k-bit boundary (that is, the k LSB bits of the address are 0, with k verifying $2^k > BK$), inside [ARx% – (BK–1), ARx%] address interval.

For this purpose, DQFLOAT must be implemented in an address whose 5 LSB are 0, and SRFLOAT must be implemented in an address whose 3 LSB are 0. In the floating-point multiplication routine (see the code above), you saw an example of circular addressing in the last indirect addressing access.

3.4 Logarithmic Conversion

The logarithm conversion is applied to the difference signal (that is, input signal minus signal estimate), before scaling it by the quantizer scale factor. Going to logarithmic domain makes it possible to reduce the signal dynamic, favoring low levels, and it has to obtain a more uniform signal-to-noise ratio in the quantization. It also means that you subtract the scale factor from the difference signal.

Note that the logarithm signal gives only a level of the difference signal magnitude, for further quantization. The sign of the difference signal must be saved to obtain the signed ADPCM code.

To convert a number to logarithm, use the same characteristic as floating-point format (see section 3.2.2)

$|\text{number}| = [(\text{denormalized mantissa}) * 2] * 2^{\text{exponent}-1}$

If number = 0, this equality is not verified: $\log_2(0)$ is defined here to be 0, so the following method does not apply for 0.

The following inequalities are verified:

$\frac{1}{2} \leq \text{denormalized mantissa} < 1 \Rightarrow 1 \leq (\text{denormalized mantissa}) * 2 < 2$

and $2^{\text{exponent}-1} \leq |\text{number}| < 2^{\text{exponent}}$

When you take the base two logarithm, you have:

$\log_g(|\text{number}|) = (\text{exponent} - 1) + \log_2[(\text{denormalized mantissa}) * 2]$

Using linear approximation of the logarithm in the vicinity of 1 ($\log_2(1+x) \approx x$), you obtain

$\log_g(|\text{number}|) = (\text{exponent} - 2) + [(\text{denormalized mantissa}) * 2]$

(denormalized mantissa * 2) is normalized with 8 bits representation (Q7 format), so:

$128 \leq \text{mantissa} < 256$, and mantissa represents the 8 most significant bits of the fixed-point number.

Addition of the two words is performed by scaling (exponent – 2) in Q7 format, to form a 11-bit word (Q7).

Computing implementation of this logarithmic conversion uses the instructions EXP and NORM (as for floating-point conversion). Moreover, the instruction MAS allows the completion of the exponent calculation, while scaling it (by multiplication of powers of two). The program code below shows that this procedure is short and executes quickly (10 cycles max).

```
***********************************************************************************
*     Convert difference signal to logarithmic domain for further quantization:   *
*                                                                                 *
*     INPUT:                                                                      *
* A          = D(k) linear input PCM signal (Encoder)                             *
*            = DX(k) Quantized reconstructed signal (Decoder)                     *
*                                                                                 *
*     OUTPUT:                                                                     *
* B          = DL(k) logarithmic difference signal                                *
*                                                                                 *
*     CYCLES: Min: 5, Max: 10 (usual)                                             *
***********************************************************************************
      BC      SUBTB, AEQ  ; If A = 0, DL = 0
      ABS     A, B        ; B = magnitude of D = DQM
      EXP     B           ; TREG = 31 – EXP
      LD      C3712, A    ; C3712 =29*2^7 for EXP computing and scaling
      NORM    B           ; BH = DQM << (14–(EXP–1))
      SFTA    B, –7       ; BH = 2*MANT (Q7 format)
      MAS     C128, A     ; A = 29*2^7 – (TREG*2^7) = (EXP–2) << 7
      ADD     B, –16, A   ; Add scaled(EXP–2) to 2*MANT to form DL
```

3.5 3-, 4-, or 5-Bit Quantizer

An iterative search is used for the quantizer, where the difference signal (quantizer input) is compared to the low levels of decision corresponding to a quantized level (ADPCM code). Use the block repeat capability to limit the number of cycles used for this loop function.

This quantizer is valid for 16-, 24-, 32-, or 40-Kbps coding to yield ADPCM code, as it is for synchronous coding adjustment function in the decoder. For this, it uses several variables and tables.

First, the table ITBLxx provides low levels of decisions values, depending on the chosen rate (xx = 16, 24, 32, or 40 to designate 16-, 24-, 32-, or 40-Kbps coding).

The table QUANxx provides ADPCM code for the encoder, while the table SYNCxx provides *ID* (see section 3.1.3) code for the synchronous coding adjustment in the decoder (xx also designates the coding rate).

As for the variables, *N*, which is the number of |I| levels, is use to initialize the research interval. In fact, there are actually *N* levels of decision values that correspond to 2 * N quantization levels when you consider the sign of difference signal. The other variable is *RPTQUA*; this is the number of repeats for the iterative search block. As *N* is an even value, it ensures that this number is constant for a given rate; that is, the number of loops is 1, 2, 3, or 4, respectively, for 16-, 24-, 32-, or 40-Kbps. So, 16-Kbps ADPCM encoding is faster than 40-Kbps encoding. This constant number of loops makes it possible to avoid the usual test of algorithm termination.

The instruction *SACCD* allows the storage of the current middle level of quantization, while comparing the corresponding decision value with the difference signal. The instructions *SUB* and *ADD*, used with dual data-memory access, allow addition and subtraction on two memory operands to be performed in one clock cycle. Details of this routine are shown in the program code below, where loop block for iterative search is in bold characters.

```
*************************************************************************************
*  16, 24, 32, or 40Kbit/s ADPCM quantizer                                          *
*                                                                                   *
*     INPUT:                                                                        *
* A   = DLN(k): Log2(Difference signal) with quantizer scale factor normalization *
* SIGN       = sign(D): Difference signal sign (=0 if positive, =–1 else)           *
* N          = Number of |I| levels                                                 *
* ADQUAN     = DROM address of quantizer QUANxx                                     *
* QUANxx (*AR2)          = Quantizer table (xx = 16, 24, 32, or 40)                 *
* ADITBL                 = DROM address of |I| table ITBLxx                         *
* ITBLxx (*AR2)          = QSi: quantizer levels (xx = 16, 24, 32, or 40)           *
*                                                                                   *
*     OUTPUT:                                                                       *
* A          = I(k) ADPCM code *
*                                                                                   *
*     CYCLES:                                                                       *
* 32 (16 Kbit/s)                                                                    *
* 41 (24 Kbit/s)                                                                    *
* 50 (32 Kbit/s)                                                                    *
* 59 (40 Kbit/s)                                                                    *
*************************************************************************************
      STL     A, *AR4+     ; DQ = DLN
      LD      #0, A        ;
      STL     A, *AR4–     ; Initialize a = lower step of quantization
      LD      N, 16, A     ;
```

```
      STH     A, *AR3       ; Initialize b = upper step of quantization
      SFTA    A, –15, B     ;
      MVDK    RPTQUA, BRC   ; Initialize block repeat counter
      RPTB    EQUAN         ; Repeat loop for iterative search
      ADDS    ADITBL, B     ; Add origin address of quantization table
      STLM    B, AR2        ; AR2 = address of QS[M]
      SFTA    A, –1         ; A = M = middle step of quantization
      LD      #0, ASM       ; ASM = 0 for SACCD and AR2 update latency
      SUB     *AR4+, *AR2, B ; A = (D-QS[M]) << 16
      SACCD   A, *AR4, BGEQ ; a = M if D >= QS[M]
      SACCD   A, *AR3, BLT ; b = M if D < QS[M]
      ADD     *AR3, *AR4–, A ; A = b+a << 16  = 2*M << 16
EQUAN SFTA    A, –15, B     ; B = offset table (4*M)
      LD      SIGN, A       ; Load sign of difference signal
      SFTA    B, –2         ; B = M = level of quantization
      ADDS    ADQUAN, B     ; Add table quantization address
      XC      1, ANEQ       ;
      ADD     N, B          ; Add offset if D was negative
      STLM    B, AR2        ; *AR2 = address of ADPCM code
      RETD                  ;
      LD      *AR2, B       ;
      LD      *AR2, A       ; A = ADPCM code (or = ID for SYNC routine)
```

3.6 Inverse Quantizer

From an ADPCM word, the inverse quantizer gives the quantized difference signal,to reconstruct the original signal. It is, of course, the basis function of the decoder. However, it is also used for the encoder to estimate the next sample signal, which means that quantization error is re-introduced into the proper input signal. At the level of the encoding process, the difference signal is calculated between input PCM sample and a signal estimate that would be the same as that calculated by the decoder. Thus, the decoding process does not diverge relative to the encoding process. In other words, there is no accumulation of quantization error.

From the inverse quantizer, there begins a whole process of algorithm adaptation that is common to the encoder and decoder. It includes quantizer scale-factor adaptation, speed control parameter adaptation, and predictor adaptation, as well as tone and transition detection.

To allow these adaptations, the inverse quantizer provides:

- Quantized difference signal: $d_{qln}(I)$ (normalized and in logarithmic domain)
- The functions F(I) (rate of change weighting function), and W(I) (scale-factor multipliers)
- Sign of I (ADPCM code), which was the difference signal sign, and which will be the quantized difference signal sign.
- IM: Magnitude of I in the sense where IM positive but order relation in I is kept for IM. This value is useful for the synchronous adjustment module, in decoder.

These values are given via the table IQUAxx (xx = 16, 24, 32, 40 for 16, 24, 32, 40 Kbps coding). In fact, as dqln(I), F(I), and W(I) only depends on |I| value, this table gives the address where these functions are located, depending only on |I|. This makes it possible to save memory.

3.7 Transition Detector and Trigger Process

This implementation of the ADPCM algorithm has been conceived to follow a linear progress with a minimum of *branch* instructions for a maximum of clarity, and with regard to the G.726 recommendation. However, this module constitutes an exception. When a transition is detected, predictor coefficients and tone-detection variables take their reset value (0), while the speed control parameter is set to one, to go into fast adaptation mode. When a transition is detected, the chosen solution is to reset the variables concerned, then to skip the adaptation process where they are implied. In the opposite case, it avoids testing a transition variable at the end of the adaptation process.

The reset of the coefficients is performed using long-word instructions, to set two variables with one-cycle instruction, as shown below. The constraint for this capability is the alignment of the long words on even boundaries.

```
**********************************************************************************
*     Reset of ai(k), bi(k), td(k), ap(k) (to one) in case of transition detect    *
*
*
*           CYCLES: 9                                                             *
**********************************************************************************
      LD    C256, 16, A      ; Load 0100 0000 in A
      DST     A, AP          ; AP(k) = 256 (1) and TD(k) is set to 0
      LD      #0, A          ; Reset of all predictor coefficients
      DST     A, A1          ; A1(k) = A2(k) = 0
      DST     A, B1          ; B1(k) = B2(k) = 0,
      BD      ADAPTY         ; then go directly to routine ADAPTY: skip
                             ; adaptation process
      DST     A, B3          ; B3(k) = B4(k) = 0
      DST     A, B5          ; B5(k) = B6(k) = 0
```

3.8 Double Precision/Dual 16-Bit Arithmetic Use

The TMS320C54x DSP is a 16-bit processor, but its two read data buses allow it to perform dual data-memory access. Some long-word (32-bits) instructions are thus available, making possible 32-bit arithmetic. For these instructions, the long-word operand has to be aligned on an even word address in memory.

The first utilization of this feature is the double-precision requirement. In G.726 recommendation, all variables can be implemented on 16-bit words, except the locked quantizer scale factor, $y_l(k)$, that needs a 19-bit resolution in Q15 format. The chosen solution is to implement it in a long-word as Q25 format with 29-bit resolution. That makes the format of the high-word compatible with the format of the other scale factors ($y_u(k)$ and $y(k)$ in Q9 format). To respect the required resolution, the 10 LSB of the low word must have been set to zero. The code below illustrates the possibility of using $y_l(k)$, depending on the required format:

```
**** Quantizer scale factor y(k) calculation: single-word yl(k–1) use (Q9 format) ****
      STLM B, T            ; T = AL for multiplication
      LD YU, A             ;
      SUB  YL, A           ; Here, YL = YL(high word) = (Q9)
      ABS  A, B            ; A = YU – YL
      STL  B, TEMP         ; B = |YU – YL|
      MPY  TEMP, B         ; Multiply unsigned magnitudes: |YU–YL| * AL
      SFTA B, –6           ; Scale the result to obtain (Q9)
      XC 1, ALT            ; Convert magnitude to two’s complement
      NEG  B               ; Negate if YU – YL was negative
      RETD                 ;
      ADD  YL, B           ; B = YL + AL * (YU–YL)
      STL  B, Y            ; Store Y(k) (Q9)
(...)

** Locked quantizer scale factor yl(k) updating: double-word yl(k–1) use (Q25 format,
** actually Q15) **
      LD YU, 16, A         ; Scale YU with YL
      DSUB YL, A           ; A = YU – YL = 19-bit word
      STH  A, *AR3         ; Truncate the 6 LSB of –YL to limit to Q15
      DLD  YL, B           ; B = YL
      ADD  *AR3, 10, B     ; B = YL + (YU–YL) >> 6 (Q15 format)
      DST B, YL            ; Store long-word YL
```

Another possibility of doubleword arithmetic is to consider a long-word as two different variables for which a double access would be possible. You have already seen an example with the trigger process (see section 3.7). Another case is for the variable p(k), that is the partial signal reconstructed signal (sum of partial signal estimate $s_{ez}(k)$ and quantized difference $d_q(k)$). The physical long-word associated is the PK0 variable. High-word is the sign of p(k) (in definition of section 3.10), and low-word is p(k) itself. The following code shows how long-word PK0 can be used, depending on the required information:

```
**** partial signal reconstructed calculation ****
**** works only for 16, 24, 32 Kbps coding (dq coded with 15 bits) ****
**** so another solution was finally chosen to satisfy 40 Kbps coding also ****
      ADD    SEZ, A        ;
      DST    A, PK0        ; PK0high = sign(DQ+SEZ), PK0low = DQ + SEZ = P(k)
(...)
**** predictor coefficient a1(k) updating ****
      LDU    PK1, A ; A = PK1
      XOR    PK0, A ; A = PK0 ** PK1 (signs): single word access for PK0
      LD     C192, B       ; 192 = 3 * 2^–8 in Ai scale
      DLD    PK0, A ; test P(k) = 0 : double-word access for PK0
      XC     1, ANEQ       ; Test PK0 ** PK1 sign
```

```
NEG    B              ; B = 3 * 2^–8 * PK0 * PK1
XC 1,  AEQ            ;
LD     #0, B          ; If P(k) = 0, then sign(P(k))= 0, so B is 0
LD     A1, A          ;
SUB    A, –8          ; A = A1 – A1 >> 8
ADD    B, A           ; A = A1 – A1 >> 8 + 3 * (PK0 * PK1) >> 8
```

Lastly, special instructions for dual 16-bit arithmetic are available. Dual 16-bit arithmetic can be chosen by setting the C16 bit of ST1.

In this case, the ALU considers the long-word as two separates 16-bit words. However, when using only the low-word for these special instructions, this bit need not be set. For instance, for the instruction *DSUBT*, subtract TREG from the long-word. For your purpose, this makes it possible to directly compute the exponent, in floating-point conversion (see section 3.2.2).

3.9 Limitation of Coefficients Using Compare Unit

To ensure that the filters do no diverge, some variables and adapted coefficients are limited. That is the case for $y_u(k)$, $a_l(k)$, $a_1(k)$, $a_2(k)$, while the others are implicitly limited.

For these limitations, the *MIN* and *MAX* instructions of the C54x are very useful. Shown here is a typical example of coefficient limitation:

```
**** Limit predictor coefficient a2(k) **** 5 cycles

LD     C12288, B      ; B = 12288 (0.75) = upper limit of A2
MIN    A              ; A = A2 <= 12288
NEG    B              ; B = –12288 (–0.75) = lower limit
MAX    A              ; –12288 <= A <= 12288
STL    A, A2          ; Store A2(k)
```

3.10 Sign Representation

Sign of a word is normally defined as:

sign(x) = +1 if $x \geq 0$, else sign(x) = –1

This definition allows to use the property:

$$x = |x| * sign(x) \tag{26}$$

This sign representation is not very significant in computing arithmetic when using he two's complement format. In this format, sign bits are non-significant leading bits: zero for a positive number, one for negative numbers. As a consequence, when loading 16-bit data in the 40-bit accumulator of the C54x, the high part of accumulator contains 16 sign bits of the data. These can be easily stored in memory due to the *STH* instruction. This representation is chosen for its sign distinction. So, this sign has the value:

sign = 0x0000 = 0 for positive data

sign = 0xFFFF = –1 for negative data

The temporary variable, SIGN, is often used to store these signs. Note that G.726 recommendation defines computing sign with only one bit, that is 0 for positive values, 1 for negative values. In fact, it is similar, when considering that you always use sign extension (arithmetical approach), while they do not (logical approach).

Such a representation of sign allows easy sign calculation, storage, and test. But equality (1) is no longer valid. For instance equation (2–11) cannot be implemented with simple sign multiplication.

In fact, you have the following equivalence:

sign(x) * sign(y) ⇔ sign(x) ** sign(y) in computing arithmetic,

Where ** designates logical *XOR*. You will see how to implement this feature with predictor coefficient $a_2(k)$ adaptation (sign arithmetic in bold characters):

```
*********************************************************************************
* Update a2(k) predictor coefficients:                                          *
* A2 = (1–2^–7)*A2 + 2^–7*[PK0*PK2–F(A1)*PK0*PK1] where PK0 is 0 if P(k) = 0     *
*                                                                               *
*                               INPUT:                                          *
* A2          = A1(k–1), A2(k–1): 2nd order IIR filter coefficients             *
* PK0 (high) = sgn(P(k))                                                        *
* PK0 (low)  = DQSEZ = P(k) = DQ(k) + SEZ(k): Partial reconstructed signal      *
* PK1, PK2   = sgn(P(k–1)), sgn(P(k–2))                                         *
*                                                                               *
*                               OUTPUT:                                         *
* A2          = unlimited A2(k)                                                 *
*                                                                               *
*                               CYCLES: 23                                      *
*********************************************************************************
      LDU     PK1, A        ; A = PK1
      XOR     PK0, A        ; A = PK0 ** PK1
      LD      A1, B         ; B = A1
      XOR     B, –16, A     ; AL = PK0 ** PK1 ** sign(A1)
      STL     A, *AR5       ; *AR5 = PK0 ** PK1 ** sign(A1)
      ABS     B             ; B = |A1|
      BIT     *AR5+, 0      ; Test sign of PK0 ** PK1 ** sign(A1)
      LD      C8191, A      ; Perform f(A1): compare |A1| with 1/2
      MIN     B             ; and saturate if |A1| > 1/2
      SFTA    B, 2          ; |f(A1)| = 4 * |A1|
      XC      1, NTC        ; B = |f(A1)|
      NEG     B             ; B = –f(A1)*PK0*PK1
      LDU     PK0, A        ; A = PK0
      XOR     PK2, A        ; A = PK0 ** PK2
```

```
SUB    C16384, B    ; B = –f(A1)*PK0*PK1 – |PK0*PK2|
DLD    PK0, A       ; Test P(k) = 0
XC     1, AEQ       ; Test PK0 ** PK2 sign
ADDS   C32768, B    ; B = –f(A1)*PK0*PK1 + PK0*PK2
XC     1, AEQ       ;
LD     #0, B        ; If P(k) = 0, then sign(P(k))= 0, so B is 0
LD     A2, A        ;
SUB    A, –7        ; A = A2 – A2 >> 7
ADD    B, –7, A     ; A = A2 – A2>>7 +(–f(A1)*PK0*PK1+PK0*PK2)>>7
```

3.11 Coder Rate and PCM Laws Selection

The decision of coder rate (16, 24, 32, or 40 Kbps) and PCM law (A-law, μ-law, or linear PCM) is performed during the execution of the program. The choice is made by the main program, or calling program, by setting some variables (RATE, LAW), and is then managed by the SELECT routine. This routine is currently called by both the encoder and decoder so that the choice is estimated at each new sample.

Ten variables (see Table 9) allow a distinction to be made between the coder rates and the PCM laws. These variables are set according to the values of the global variables *RATE* and *LAW*. To estimate these variables more quickly, two initializations tables are used, thereby avoiding multiple tests. These tables directly give the correct values of the ten relevant variables; for this, the *MVDD* instruction is used. It allows data memory transfer in one clock cycle. Also, auxiliary registers are initialized using *MVMM* (memory-mapped register transfer).

3.12 Channel Selection

Several channels can be simultaneously dealt with; each channel has its own data memory space (context), and runs independently. This is made possible by direct addressing, using data-memory address (DMA). In this mode, the absolute address is obtained with the DMA address (7 bits used as address LSB), and the value of the data-page pointer (DP), 9 bits used as address MSB). DMA is also used as an address relative to the beginning of a page. You allocate a data-page per channel for memory space, so, when using DMA addressing mode, the value of DP determines which channel you want to access. To enable this, the context memory allocated for a channel does not exceed one page (that is, 128 words).

A channel also uses data-memory constants in program space. These data are used by each channel and are held at a precise location, depending on program linking. They are accessed by indirect addressing, and do not cause problems because these locations have constant addresses.

When you access the channel context (dataP) via indirect addressing, the address contained in the auxiliary register must be correctly set. This is an actual absolute address, and cannot be set using the absolute address of the label used with direct addressing. In fact, each variable is allocated only one time, at a precise address when linking. The label continues to designate this location, even if you also use the label to designate the same variable, but for another channel (and so for another address), as explained above. The only way to proceed is to extract DP value, to calculate the absolute address of the concerned variable. This is done in the initialization routine when implementing a channel, for a specific variable. The initialization routine (shown below) uses this address to initialize the context (data) space of the channel (using a program table). Also, this address value is kept in the context channel, and is used later to initialize auxiliary registers (for example, in the SELECT routine). See section 3.11.

4 Data Memory Organization

Table 5 through Table 8 display the map of the required RAM space for a G.726 channel.

The tables includes these different fields:

- “Address” is the relative address (in decimal format) relating the beginning of a data page.
- “Name” is the label of the variable address. When the label designates a location of several words as for DQFLOAT, YL, and SRFLOAT, the index in the bracket is for the number of the word for this location. For example, DQFLOAT(5) designates the fifth word from DQFLOAT location.
- “Access and routine” is for the addressing mode. When indicated ARx, it means that the indirect addressing mode is used with the auxiliary register number x. “DMA” means that the direct-addressing mode is used. The numbers of the routines (see Table 28) that use the variable are between brackets.
- “Reset value” gives the assigned value of the variable by the reset routine, _G726ENC_TI_reset or _G726DEC_TI_reset.
- “Description” gives a short description of the variable.

Table 5. Internal Processing Delayed Variables

Address	Name	Access and Routine	Reset Values	Description
0	DQFLOAT(1)	AR7 (1, 34)	0	Designates either DQ1†, ..., or DQ6† exponent
1	DQFLOAT(2)	AR7 (1, 34)	32	Designates either DQ1†, ..., or DQ6† mantissa
2	DQFLOAT(3)	AR7 (1, 25, 34)	0	Designates either DQ1†, ..., or DQ6† sign
...	... (4*3 variables)	...	...	...
15	DQFLOAT(16)	AR7 (1, 34)	0	Designates either DQ1†, ..., or DQ6† exponent
16	DQFLOAT(17)	AR7 (1, 34)	32	Designates either DQ1†, ..., or DQ6† mantissa
17	DQFLOAT(18)	AR7 (1, 25, 34)	0	Designates either DQ1†, ..., or DQ6† sign
18	YL(high word)	DMA (4, 18, 33)	544	Designates YL† format with 10
19	YL(low word)	DMA (33)	0	LSB set to 0 making it an actual Q15 format)
20	AP	DMA (3, 19, 29)	0	Designates AP†
21	TD	DMA (18, 20, 28)	0	Designates TD†
22	PK1	DMA (21, 36)	0	Designates PK1†
23	PK2	DMA (23, 36)	0	Designates PK2†
24	SRFLOAT(1)	AR6 (1, 35)	0	Designates either SR1† or SR2† exponent
25	SRFLOAT(2)	AR6 (1, 35)	0	Designates either SR1† or SR2† mantissa
26	SRFLOAT(3)	AR6 (1, 35)	0	Designates either SR1† or SR2† sign
27	SRFLOAT(4)	AR6 (1, 35)	0	Designates either SR1† or SR2† exponent
28	SRFLOAT(5)	AR6 (1, 35)	0	Designates either SR1† or SR2† mantissa
29	SRFLOAT(6)	AR6 (1, 35)	0	Designates either SR1† or SR2v sign
30	A1	DMA (1, 20, 23, 24)	0	Designates A1†
31	A2	DMA (1, 20, 21, 22)	0	Designates A2†
32	B1	DMA (1, 20, 26)	0	Designates B1†
33	B2	DMA (1, 20, 26)	0	Designates B2†
34	B3	DMA (1, 20, 26)	0	Designates B3†
35	B4	DMA (1, 20, 26)	0	Designates B4†
36	B5	DMA (1, 20, 26)	0	Designates B5†
37	B6	DMA (1, 20, 26)	0	Designates B6†
38	DMS	DMA (16, 28)	0	Designates DMS†
39	DML	DMA (17, 28)	0	Designates DML†

Table 5. Internal Processing Delayed Variables (Continued)

Address	Name	Access and Routine	Reset Values	Description
40	YU	DMA (4, 32)	544	Designates YU†

† See the description of this variable in Table 30.

Table 6. Constants

Address	Name	Access	Reset Value	Description
41	C1	DMA	1	Constant value C1 = 1
...	... (24 variables)	...	...	... (Constant value Cx = x)
64	C32768	DMA	32768	Constant value C32767 = 32768 (used as unsigned word)
65	M16	DMA	–16	Constant value M16 = –16
66	M11776	DMA	–11776	Constant value M11776 = –11776
67	ADSECOD	DMA	SECOD – 16	Address of rate coder selection table (constant = SECOD –16)
68	ADSELAW	DMA	SELAW	Address of law–PCM selection table (constant = SELAW)

Table 7. Address Variables

Address	Name	Access	Reset Value	Description
69	ADDQ6	DMA	Address of DQFLOAT(1)	Exponent Address of 6 times delayed quantized difference
70	ADSR2	DMA	Address of SRFLOAT(1)	Exponent Address of 2 times delayed reconstructed signal
71	ADTEMP	DMA	Address of N	Address of variable N (constant once initialized)
72	ADY	DMA	Address of Y	

Table 8. Global Variables: G.726 Commands, Input and Output Signals

Address	Name	Access and Routine	Format	Description
73	_LAW	DMA (0, 37)	Possible values: 0, 1, or 2	PCM law select: 0 for µ-law, 1 for A-law, 2 for linear PCM
74	S	DMA	(7 + S) bits, Q0 SM (log-PCM) (13 + S) bits, Q0 TC (linear-PCM)	PCM input word for encoder
75	I	DMA	2 bits (LSB) for 16 Kbps coding 3 bits (LSB) for 24 Kbps coding 4 bits (LSB) for 32 Kbps coding 5 bits (LSB) for 40 Kbps coding	ADPCM word (output for encoder, input for decoder)
76	SD	DMA (13, 35, 37, 38)	(7 + S) bits, Q0 SM (log-PCM) (13 + S) bits, Q0 TC (linear-PCM)	PCM output word for decoder

Table 9. Coder Rate and PCM-Law Selection

Address	Name	Access	Description
77	N	DMA (9)/AR3 (0)	Current number of |I| levels (N = 2, 4, 8,16)
78	RPTQUA	DMA (9)/AR3 (0)	Current block repeat number for quantization loop (RPTQUA = 0,1,2,3)
79	SHIFT	DMA (10)/AR3 (0)	Current shift value for Bi update
80	ADITBL	DMA (9)/AR3 (0)	Current |I| table address
81	ADIQUA	DMA (10)/AR3 (0)	Current inverse quantizer table address
82	ADQUAN	DMA (9)/AR3 (0)	Current quantizer table address
83	ADLAW	DMA (5)/AR3 (0)	Current PCM inverse quantizer table address
84	LAWMASK	DMA (5, 37, 38)/AR3 (0)	Current log-PCM magnitude mask
85	LAWBIAS	DMA (37)/AR3 (0)	Current bias for log-PCM quantizer
86	LAWSEG	DMA (37)/AR3 (0)	Current constant for segment calculation in PCM quantizer

Table 10. Temporary Internal Processing Variables

Address	Name	Access	Description
87	Y	DMA (4, 8, 11, 28, 31)	Designates Y†
88	SEZ	DMA (2, 14)/AR5 (2)	Designates SEZ†
89	SIGN	DMA (...)/AR5 (...)	Designates DS†, DSX†, or DQS
90	SE	DMA (2, 6, 13)/AR4 (2)	Designates SE†
91	DQ	DMA (12, 18, 25, 34)/AR4 (9)	Designates DQ†, or D†
92	PK0	DMA (14, 21, 23)/AR4 (9)	Designates PK0†, or lower interval limit of iterative search in (9)

† See description of the variable in Table 30.

Table 11. Temporary Variables

Address	Name	Access	Description
93	TEMP	AR3 (...)	General use temporary variable for intermediate calculation
94	ADQUAND	DMA	Holds ADQUAN at reset
95	RATE	DMA	Compression rate
96	LAW	DMA	PCM data format
97	FRLEN	DMA	Processing frame length

4.1 Algorithm Tables (Program Space)

This section has 849 words and contains six tables. The variables of the involved variables have their format described in Table 30. These tables are:

1. RAM initialization table, INIRAM. This table contains reset values of internal processing variables, and constant values that are transferred in RAM when applying the coder reset routine, _G726ENC_TI_reset or _G726DEC_TI_reset.
2. |I| tables for all rates. Each table contains successively:
   a. The lowest level of input quantizer interval, QS|I| (first value of column two, Table 12 through Table 15).
   b. The output level of the inverse quantizer, DQLN|I| (column three, Table 16 through Table 19).
   c. The rate-of-change weighting function, F|I|, (Table 20 through Table 23).
   d. The scale-factor multipliers, W|I| (Table 24 through Table 27).
3. Quantizer tables for all rates. Each table gives the output word that corresponds to a certain level of quantization. For the encoder, it gives the ADPCM word I (in two's complement, column three of Table 12 through Table 15), and for the decoder (in re-encoding routine for synchronous adjustment), it gives the word ID (in absolute value, column four of Table 12 through Table 15), which has to be compared with the original ADPCM code.
4. Inverse quantizer tables for all rates. Each table gives, from a I ADPCM code, the output level of the quantizer which corresponds to the quantized difference signal (normalized and in logarithmic format, column three, Table 16 through Table 19). It also gives the sign of this quantized difference (column two, Table 16 through Table 19). This sign, which is also the sign of I, is the sign of the original difference signal before being normalized and going into logarithmic domain. Lastly, it gives the word the magnitude of I (column four of Table 16 through Table 19), which has to be compared with ID (see Table 12 through Table 15).
5. A-law and μ-law tables for PCM expanding. This inverse quantizer table gives the linear PCM level corresponding to a logarithmic PCM code (see section 3.1.2).
6. PCM laws and coder rate selection tables. These tables permit the initialization of the coder, based on linear PCM, A-law PCM, or μ-law PCM choice, and 16, 24, 32, or 40 Kbps coding choice. The relevant variables are those of Table 9.

Table 12. Quantizer Definition for 40-Kbps ADPCM

DS/DSX	DLN/DLNX	I	ID
0	553, ..., 2047	01111	31
0	528, ..., 552	01110	30
0	502, ..., 527	01101	29
0	475, ..., 501	01100	28
0	445, ..., 474	01011	27

NOTE: The I values are transmitted with bit 1.

Table 12. Quantizer Definition for 40-Kbps ADPCM (Continued)

DS/DSX	DLN/DLNX	I	ID
0	413, ..., 444	01010	26
0	378, ..., 412	01001	25
0	339, ..., 377	01000	24
0	298, ..., 338	00111	23
0	250, ..., 297	00110	22
0	198, ..., 249	00101	21
0	139, ..., 197	00100	20
0	68, ..., 138	00011	19
0	–16, ..., 67	00010	18
0	–22, ..., –17	00001	17
0	–2048, ..., –23	11111	15
1	–2048, ..., –23	11111	15
1	–22, ..., –17	11110	14
1	–16, ..., 67	11101	13
1	68, ..., 138	11100	12
1	139, ..., 197	11011	11
1	198, ..., 249	11010	10
1	250, ..., 297	11001	9
1	298, ..., 338	11000	8
1	339, ..., 377	10111	7
1	378, ..., 412	10110	6
1	413, ..., 444	10101	5
1	445, ..., 474	10100	4
1	475, ..., 501	10011	3
1	502, ..., 527	10010	2
1	528, ..., 552	10001	1
1	553, ..., 2047	10000	0

NOTE: The I values are transmitted with bit 1.

Table 13. Quantizer Definition for 32-Kbps ADPCM

DS/DSX	DLN/DLNX	I	ID
0	400, ..., 2047	0111	15
0	349, ..., 399	0110	14
0	300, ..., 348	0101	13
0	246, ..., 299	0100	12
0	178, ..., 245	0011	11
0	80, ..., 177	0010	10
0	–124, ..., 79	0001	9
0	–2048, ..., –125	1111	7
1	–2048, ..., –125	1111	7
1	–124, ..., 79	1110	6
1	80, ..., 177	1101	5
1	178, ..., 245	1100	4
1	246, ..., 299	1011	3
1	300, ..., 348	1010	2
1	349, ..., 399	1001	1
1	400, ..., 2047	1000	0

NOTE: The I values are transmitted with bit 1.

Table 14. Quantizer Definition for 24-Kbps ADPCM

DS/DSX	DLN/DLNX	I	ID
0	331, ..., 2047	011	7
0	218, ..., 330	010	6
0	8, ..., 217	001	5
0	–2048, ..., 7	111	3
1	–2048, ..., 7	111	3
1	8, ..., 217	110	2
1	218, ..., 330	101	1
1	331, ..., 2047	100	0

NOTE: The I values are transmitted with bit 1.

Table 15. Quantizer Definition for 16-Kbps ADPCM

DS/DSX	DLN/DLNX	I	ID
0	261, ..., 2047	01	3
0	–2048, ..., 260	00	2
1	–2048, ..., 260	11	1
1	261, ..., 2047	10	0

NOTE: The I values are transmitted with bit 1.

Table 16. Quantizer Output Levels for 40-Kbps ADPCM

I	DQS	DQLN	IM
01111	0	566	31
01110	0	539	30
01101	0	514	29
01100	0	488	28
01011	0	459	27
01010	0	429	26
01001	0	395	25
01000	0	358	24
00111	0	318	23
00110	0	274	22
00101	0	224	21
00100	0	169	20
00011	0	104	19
00010	0	28	18
00001	0	–66	17
00000	0	–2048	16
11111	1	–2048	15
11110	1	–66	14
11101	1	28	13
11100	1	104	12
11011	1	169	11

NOTES:
1. The I values are received, starting with bit 1.
2. It is possible for the decoder to receive the codeword 00000 because of transmission disturbances (e.g., line bit errors).

Table 16. Quantizer Output Levels for 40-Kbps ADPCM (Continued)

I	DQS	DQLN	IM
11010	1	224	10
11001	1	274	9
11000	1	318	8
10111	1	358	7
10110	1	395	6
10101	1	429	5
10100	1	459	4
10011	1	488	3
10010	1	514	2
10001	1	539	1
10000	1	566	0

NOTES: 1. The I values are received, starting with bit 1.
2. It is possible for the decoder to receive the codeword 00000 because of transmission disturbances (e.g., line bit errors).

Table 17. Quantizer Output Levels for 32-Kbps ADPCM

I	DQS	DQLN	IM
0111	0	425	15
0110	0	373	14
0101	0	323	13
0100	0	273	12
0011	0	213	11
0010	0	135	10
0001	0	4	9
0000	0	–2048	8
1111	1	–2048	7
1110	1	4	6
1101	1	135	5
1100	1	213	4
1011	1	273	3

NOTES: 1. The I values are received, starting with bit 1.
2. It is possible for the decoder to receive the codeword 00000 because of transmission disturbances (e.g., line bit errors).

I	DQS	DQLN	IM
1010	1	323	2
1001	1	373	1
1000	1	425	0

NOTES: 1. The I values are received, starting with bit 1.
2. It is possible for the decoder to receive the codeword 00000 because of transmission disturbances (e.g., line bit errors).

Table 18. Quantizer Output Levels for 24-Kbps ADPCM

I	DQS	DQLN	IM
011	0	373	7
010	0	273	6
001	0	135	5
000	0	–2048	4
111	1	–2048	3
110	1	135	2
101	1	273	1
100	1	373	0

NOTES: 1. The I values are received, starting with bit 1.
2. It is possible for the decoder to receive the codeword 000 because of transmission disturbances (e.g., line bit errors).

Table 19. Quantizer Output Levels for 16-Kbps ADPCM

I	DQS	DQLN	IM
01	0	365	3
00	0	116	2
11	1	116	1
10	1	365	0

NOTE: The I values are received, starting with bit 1.

Table 20. Map Quantizer Output F|I| for 40-Kbps ADPCM

|I(k)|	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F|I(k)|	6	6	5	4	3	2	1	1	1	1	1	0	0	0	0	0

Table 21. Map Quantizer Output F|I| for 32 Kbps ADPCM

|I(k)|	7	6	5	4	3	2	1	0
F|I(k)|	7	3	1	1	1	0	0	0

Table 22. Map Quantizer Output for 24 Kbps ADPCM

|I(k)|	3	2	1	0
F|I(k)|	7	2	1	0

Table 23. Map Quantizer Output for 16 Kbps ADPCM

|I(k)|	1	0
F|I(k)|	7	0

Table 24. Quantizer Scale Factor Multipliers W|I| for 40 Kbps ADPCM

|I|	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
W|I|	696	529	440	358	280	219	179	141	100	58	41	40	39	24	14	14

Table 25. Quantizer Scale Factor Multipliers W|I| for 32 Kbps ADPCM

|I|	7	6	5	4	3	2	1	0
W|I|	1122	355	198	112	64	41	18	–12

Table 26. Quantizer Scale Factor Multipliers W|I| for 24 Kbps ADPCM

|I|	3	2	1	0
W|I|	582	137	30	–4

Table 27. Quantizer Scale Factor Multipliers W|I| for 16 Kbps ADPCM

|I|	1	0
W|I|	439	–22

5 Program Organization

Numbers of routines refer to the numbers they are attributed in Table 28.

5.1 Channel Initialization Routine: _G726ENC_TI_reset / _G726DEC_TL_reset

The sequence of the initialization routine is to:

- Initialize status registers
- Compute absolute address of channel to initialize address variables for indirect addressing mode (see section 3.12)
- Transfer reset values and constants into channel RAM space

5.2 Encoder Routine: G726COD

The sequence of the encoder routine is summarized in Table 28:

Table 28. Encoder Sequence (578–605 Cycles)

No.	Function	Description	Routines	Cycles
1	Select PCM law and encoder flow rate	Initialize tables, address, and variables for selecting either A-law, μ-law, or linear PCM, and for choosing either 16-, 24-, 32-, or 40-Kbps flow rate.	0	34
2	Compute signal estimate	Calculate the signal estimate $s_e(k)$ from the previous quantized difference samples $d_q(k–i)$ (i = 1, ..., 6), and reconstructed samples $s_r(k–i)$ (i = 1, 2) with filters using floating-point multiplication.	1, 2	230
3	Compute quantizer scale factor	Calculate speed control parameter $a_l(k)$ and, using it, calculate the quantizer scale factor y(k).	3, 4	18
4	Load input PCM word	Read the input PCM sample s(k).		2
5	Convert log-PCM word to linear PCM	Linearize 8-bit log-PCM sample s(k) to a 14-bit two’s complement sample $s_l(k)$.	5	20
6	Compute difference signal and convert it into logarithmic domain	Calculate the difference signal d(k) between signal estimate $s_e(k)$ and current sample $s_l(k)$. Calculate the logarithm $d_{ln}(k)$ of this difference signal.	6, 7	12
7	Adaptive quantizing of the difference signal	Scale the difference signal $d_{ln}(k)$ using quantizer scale factor y(k), and quantize the result to form the ADPCM output I(k).	8, 9	32–59
8	Store output ADPCM word	Write the ADPCM output I(k).		1
9	Adaptive inverse quantizing of the ADPCM word	Yield the output of the inverse quantizer $d_{qln}(k)$. Scale it, using y(k), to form $d_{ql}(k)$, and convert it from logarithmic domain, to linear domain to obtain the quantized difference sample $d_q(k)$.	10–12	29
10	Reconstruct the PCM signal	Calculate the reconstructed signal $s_r(k)$ from quantized difference $d_q(k)$ and signal estimate $s_e(k)$.	13, 14	4
11	Speed control parameter adaptation	Adapt short-term $d_{ms}(k)$ and long-term $d_{ml}(k)$ average magnitude of |I(k)|.	15–17	10
12	Transition detection and trigger process	Detect possible transition $t_r(k)$. If so, reset the predictor, set quantizer into the fast mode of adaptation, and bypass functions (12) to (14).	18–20	6

Table 28. Encoder Sequence (578–605 Cycles) (Continued)

No.	Function	Description	Routines	Cycles
13	Predictor adaptation	Calculate the update for the coefficients $b_i(k)$ (k = 1, ..., 6) of the FIR filter, and for the coefficients $a_i(k)$ (k = 1, 2) of the IIR filter.	21–26	102
14	Tone detection	Detect possible partial band signal (e.g. tone) $t_d(k)$.	27	3
15	Speed control parameter update	Update unlimited speed control parameter $a_p(k)$, using $t_d(k)$, $d_{ms}(k)$, $d_{ml}(k)$.	28–29	23
16	Quantizer scale factor adaptation	Update slow $y_l(k)$ and fast $y_u(k)$ quantizer scale factors.	30–33	15
17	Floating-point conversion and delays preparation	Convert quantized difference $d_q(k)$ and reconstructed signal $s_r(k)$ into floating-point, and store them in the filter buffer.	34–36	37

5.3 Decoder Routine: g726_decode1

The sequence of the decoder routine is summarized in Table 29:

Table 29. Decoder Sequence (606–633 Cycles)

No.	Function	Description	Routines	Cycles
1	Select PCM law and encoder flow rate	Initialize tables address and variables for selecting either A-law, µ-law, or linear PCM, and for choosing either 16-, 24-, 32-, or 40-Kbps flow rate.	0	34
2	Compute signal estimate	Calculate the signal estimate $s_e(k)$ from the previous quantized difference samples, $d_q(k–i)$ (i = 1, ..., 6), and reconstructed samples, $s_r(k–i)$ (i = 1, 2), with filters using floating-point multiplication.	1, 2	230
3	Compute quantizer scale factor	Calculate speed control parameter $a_l(k)$ and, using it, calculate the quantizer scale factor y(k).	3, 4	18
4	Load input ADPCM word	Read the input ADPCM sample I(k).		1
5	Adaptive inverse quantizing of the ADPCM word	Yield the output of the inverse quantizer $d_{qln}(k)$. Scale it, using y(k), to form $d_{ql}(k)$, and convert it from logarithmic domain to linear domain, to obtain the quantized difference sample $d_q(k)$.	10–12	29
6	Reconstruct the PCM signal	Calculate the reconstructed signal $s_r(k)$ from quantized difference $d_q(k)$ and signal estimate $s_e(k)$.	13, 14	4
7	Speed control parameter adaptation	Adapt short-term $d_{ms}(k)$ and long-term $d_{ml}(k)$ average magnitude of |I(k)|.	15–17	10
8	Transition detection and trigger process	Detect possible transition $t_r(k)$, if so, reset the predictor, set quantizer into the fast mode of adaptation, and bypass functions (12) to (14).	18–20	6
9	Predictor adaptation	Calculate the update for the coefficients $b_i(k)$ (k = 1, ..., 6) of the FIR filter, and for the coefficients $a_i(k)$ (k = 1, 2) of the IIR filter.	21–26	102
10	Tone detection	Detect possible partial band signal (e.g. tone) $t_d(k)$.	27	3
11	Speed control parameter update	Update unlimited speed control parameter $a_p(k)$, using $t_d(k)$, $d_{ms}(k)$, $d_{ml}(k)$.	28–29	23

Table 29. Decoder Sequence (606–633 Cycles) (Continued)

No.	Function	Description	Routines	Cycles
12	Quantizer scale factor adaptation	Update slow $y_l(k)$ and fast $y_u(k)$ quantizer scale factors.	30–33	15
13	Floating-point conversion and delays preparation	Convert quantized difference $d_q(k)$ and reconstructed signal $s_r(k)$ into floating-point, and store them in the filter buffer.	34–36	37
14	Convert linear PCM word to log-PCM	Convert the reconstructed linear PCM signal $s_r(k)$ to a log-PCM signal $s_p(k)$	37	32
15	Synchronous coding adjustment	Calculate an ADPCM signal $I_d(k)$ from $s_p(k)$, and adjust $s_p(k)$ to create $s_d(k)$ if $I_d(k)$ differs from I(k)	5–9, 38	56–83
16	Store output PCM word	Write the PCM output sample $s_d(k)$		6

5.4 Brief Functional Description of Each Sub-Block

The notations used in the sub-block descriptions are follows:

<< n denotes an n-bit left-shift operation (zero fill).

>> n denotes an n-bit arithmetical shift right operation (with sign shift).

& denotes the logical “and” operation.

+ denotes arithmetic addition.

– denotes arithmetic subtraction.

***** denotes arithmetic multiplication.

****** denotes the logical “exclusive or” operation.

A denotes the accumulator A.

B denotes the accumulator B.

TEMP denotes the temporary variable (see RAM space).

ARn denotes the auxiliary register n.

For each routine to be described, a formal description of the function realized is indicated, corresponding to the specification of the G.721 recommendation of part one (1.4). The input and output variables are given by their real place in the C54x (that may be the Accumulator A for example) and into brackets, the formal name of the specification, whose description is given in Table 30. Also indicated are the number of cycles of the routine. A short note will comment on some specific details of the routine.

The following table describes the internal processing variables. It includes these fields:

- “Name” is the formal name corresponding to G.726 recommendation.
- “Bits” gives the number of significant bits among the sixteen bits of the word, and indicates if the word is signed with the “S” information.
- “Format” gives the weight of the bits. A QX number has X fractional bits, whose weights are 2^{-1}, ..., 2^{-X}. TC denotes two’s complement, SM denotes signed magnitude, and UM denotes unsigned magnitude.
- “Memory” indicates the physical location of the variable, which could be the accumulator (A or B). It is possible that this location does not exist, in the case where the formal variable is replaced by a branch. However, these are described because they are used in the notes.
- “Description” gives a short description of the variable.

Table 30. Internal Processing Variables

Name	Bits	Format	Memory	Description
A1†,A2†	15 + S	Q14 TC	A1, A2	Delayed second-order predictor coefficients
A1P, A2P	15 + S	Q14 TC	A1, A2	Second-order predictor coefficients
A1R, A2R	15 + S	Q14 TC	None (branch)	Triggered second-order predictor coefficients
A1T	15 + S	Q14 TC	Accumulator	Unlimited a_1 coefficient
A2T	15 + S	Q14 TC	Accumulator	Unlimited a_2 coefficient
AL	7	Q6 UM	Accumulator	Limited speed control parameter
AP[a)]	10	Q8 UM	AP	Delayed speed control parameter
APP	10	Q8 UM	AP	Unlimited speed control parameter
APR	10	Q8 UM	None (branch)	Triggered unlimited speed control parameter
AX	1	Q0 UM	Accumulator	Speed control parameter update
B1†, ..., B6†	15 + S	Q14 TC	B1, ..., B6	Delayed sixth order predictor coefficients
B1P, ..., B6P	15 + S	Q14 TC	B1, ..., B6	Sixth order predictor coefficients
B1R, ., B6R	15 + S	Q14 TC	none (branch)	Triggered sixth order predictor coefficients
D	15 + S	Q0 TC	Accumulator	Difference signal, only in encoder
DL	11	Q7 UM	Accumulator	Log_2 (difference signal), only in encoder
DLN	11 + S	Q7 TC	DQ	Log_2 (normalized difference), only in encoder
DLNX	11 + S	Q7 TC	DQ	Log_2 (normalized difference), only in decoder
DLX	11	Q7 UM	Accumulator	Log_2 (difference signal), only in decoder
DML†	14	Q11 UM	DML	Delayed long term average of F(I) sequence
DMLP	14	Q11 UM	DML	Long term average of F(I) sequence
DMS†	12	Q9 UM	DMS	Delayed short term average of F(I) sequence
DMSP	12	Q9 UM	DMS	Short term average of F(I) sequence
DQ	15 + S	Q0 TC	DQ	Quantized difference signal
DQ0, DQ1†,..., DQ6† exponents	4	Q0 UM	DQFLOAT	Quantized difference signal exponent with delays 0 to 6
DQ0, DQ1†,..., DQ6† mantissas	6	Q6 UM	DQFLOAT	Quantized difference signal mantissa with delays 0 to 6
DQ0, DQ1†,..., DQ6† signs	S	Q0 TC	DQFLOAT	Quantized difference signal sign with delays 0 to 6
DQL	11 + S	Q7 TC	Accumulator	Log_2 (quantized difference signal)

† Indicates variables that are set to specific values by the optional reset. When reset is invoked (by running G726RST), these variables are set to their reset value (see these values in Table 5.

Table 30. Internal Processing Variables (Continued)

Name	Bits	Format	Memory	Description
DQLN	11 + S	Q7 TC	Accumulator	Log_2 (normalized quantized difference signal)
DQS	S	Q0 TC	SIGN	Sign bit of quantized difference signal
DS	S	Q0 TC	SIGN	Sign bit of difference signal, only in encoder
DSX	S	Q0 TC	SIGN	Sign bit of difference signal, only in decoder
DX	15 + S	Q0 TC	Accumulator	Difference signal, only in decoder
FI	3	Q0 UM	DROM table	Output of F(I)
PK0	S	Q0 TC	PK0	Sign of DQ +SEZ with delay 0
PK1[†], PK2[†]	S	Q0 TC	PK1, PK2	Sign of DQ +SEZ with delays 1 and 2
PK	15 + S	Q0 TC	TEMP	DQ + SEZ
SE	14 + S	Q0 TC	SE	Signal estimate
SEZ	14 + S	Q0 TC	SEZ	Sixth-order predictor partial signal estimate
SL	13 + S	Q0 TC	Accumulator	Linear input signal, only in encoder
SLX	13 + S	Q0 TC	Accumulator	Quantized reconstructed signal, only in decoder
SP	7 + S	Q4 SM	SD	PCM reconstructed signal, only in decoder
SR	15 + S	Q0 TC	SD	Reconstructed signal
SR0, SR1[†], SR2[†] exponents	4	Q0 UM	SRFLOAT	Reconstructed signal exponent with delays 0 to 2
SR0, SR1[†], SR2[†] mantissas	6	Q6 UM	SRFLOAT	Reconstructed signal mantissa with delays 0 to 2
SR0, SR1[†], SR2[†] signs	S	Q0 TC	SRFLOAT	Reconstructed signal sign with delays 0 to 2
TD[a)]	S	Q0 TC	TD	Delayed tone detect
TDP	S	Q0 TC	TD	Tone detect
TDR	S	Q0 TC	None (branch)	Triggered tone detect
TR	S	Q0 TC	None (branch)	Transition detect
U1, ..., U6	S	Q0 TC	Accumulator	Sixth-order predictor coefficient update sign bit
WA1, WA2	15 + S	Q1 TC	SE	Partial product of signal estimate
WB1, ..., WB6	15 + S	Q1 TC	SEZ	Partial product of partial signal estimate
WI	11 + S	Q4 TC	DROM table	Quantizer multiplier
Y	13	Q9 UM	Y	Quantizer scale factor
YL[†]	19	Q15 UM	YL	Delayed slow quantized scale factor

[†] Indicates variables that are set to specific values by the optional reset. When reset is invoked (by running G726RST), these variables are set to their reset value (see these values in Table 5.

Table 30. Internal Processing Variables (Continued)

Name	Bits	Format	Memory	Description
YLP	19	Q15 UM	YL	Slow quantized scale factor
YU†	13	Q9 UM	YU	Delayed fast quantizer scale factor
YUP	13	Q9 UM	YU	Fast quantizer scale factor
YUT	13	Q9 UM	Accumulator	Unlimited quantizer scale factor

† Indicates variables that are set to specific values by the optional reset. When reset is invoked (by running G726RST), these variables are set to their reset value (see these values in Table 5.

5.4.1 FMULT

Function: Multiply predictor coefficients with corresponding quantized difference signal or reconstructed signal. $wb_i(k) = b_i(k–1) * d_q(k–i)$ for i = 1, ..., 6; and $wa_i(k) = a_i (k–1)* s_r(k–i)$ for i = 1, 2.

Input: Ai and SRi, or Bi and DQi, AR6 or AR7 (points to SRi or DQi exponent)

Output: B (WAi or WBi), AR6 or AR7 (points to DQi–1 or SRi–1 exponent, except for SR1 and DQ1 inputs: points respectively to SR2 and DQ6 sign)

Cycles: 206

NOTE: The multiplication is performed in floating-point format. It implies, for the fixed-point processor, of the C54x, that you multiply the mantissas, add the exponents and compute the result sign. First, divide Ai or Bi by 4 to truncate it, making it a Q12 format. Then, it is converted into floating-point (see FLOATA (34) for the method). The multiplication is performed by using floating-point values of SRi or DQi in the table SRFLOAT or DQFLOAT (see FLOATA (34) and FLOATB (35)). Then, the result (WAi or WBi) is converted in 2’s complement. You also have to scale the result (11 right-shift) because of the different scales between Ai (or Bi) and SRi (or DQi), making it a Q1 format. Because of the eight coefficients of the filters, this routine is performed 8 times.

5.4.2 ACCUM

Function: Addition of predictor outputs to form the partial signal estimate (from the sixth order predictor) and the signal estimate. $s_{ez}(k) = wb_1(k) + wb_2(k) + wb_3(k) + wb_4(k) + wb_5(k) + wb_6(k)$, $s_e(k) = s_{ez}(k) + wa_1(k) + wa_2(k)$

Input: B (WAi, WBi), then for the next executions, use also the partial outputs as inputs: SE (partial addition of WAi), SEZ (partial addition of WBi)

Output: SE, SEZ

Cycles: 20

NOTE: This routine is partially executed after each call of FMULT (1) to avoid using extra variables for WBi and Wai. Also, it allows overflows to occur as specified in G.726, so that the accumulation is automatically limited to a 16-bit signed word in Q1 format. The results, SE and SEZ, are divided by two, making them a Q0 format like DQ (quantized difference). This routine is performed 8 times.

5.4.3 LIMA

Function: Limit speed control parameter. $a_l(k) = a_p(k-1)$ if $a_p(k-1) \le 1$, $a_l(k) = 1$, otherwise

Input: AP

Output: B (AL)

Cycles: 4

NOTE: AP is truncated of 2 bits (format Q6) before to be limited. So the limit for AL is 64 (=1).

5.4.4 MIX

Function: Form linear combination of fast and slow quantizer scale factors
$y(k) = a_l(k).y_u(k-1) + (1-a_l(k)).y_l(k-1)$

Inputs: YU, YL (high word), B (AL)

Output: Y

Cycles: 14

NOTE: YL is 6-bit truncated at format Q9 automatically by the use of the variable YL (high word) that represents the 13 MSB of YL. The product AL*(YU–YL) is performed in absolute value, then it is scaled, and only after the sign is introduced.

5.4.5 EXPAND

Function: Convert either A-law or μ-law PCM to uniform PCM. $s(k) \rightarrow s_l(k)$

Input: A (S or SP in decoder)

Output: A (SL or SLX in decoder)

Cycles: 13

NOTE: A table (ALAW for the A-law or MULAW for the μ-law) is used to perform this inverse quantization. For A-law, the signal is multiplied by two to obtain a 14-bit 2's complement word (Q0 format) for both A and μ-law. See description of these logarithmic quantization laws in (), and of the table in ().

5.4.6 SUBTA

Function: Compute the difference between input linear PCM value and signal estimate.
$d(k) = s_l(k) - s_e(k)$

Inputs: A (SL), SE

Output: A (D)

Cycles: 1

NOTE: SL format is a 14-bit word, while SE format is a 15-bit word, making for D a 16-bit word (Q0 format).

5.4.7 LOG

Function: Convert difference signal from the linear to the logarithm domain.
$d(k) \rightarrow \{d_l(k) = \log_2(|d(k)|), \text{sign}[d(k)]\}$

Input: A (D or DX in decoder)

Outputs: DS (DSX in decoder), B (DL or DLX in decoder)

Cycles: Min: 6, Max: 11

NOTE: for this calculation, use properties of exponent EXP and mantissa MANT of |d| such as $|d| = 2 * \text{MANT} * 2^{\text{EXP}-1}$, where $1 \le 2 * \text{MANT} < 2$ and $2^{\text{EXP}-1} \le |d| < 2^{\text{EXP}}$. Note that this EXP corresponds to the actual exponent and does not have the same definition as the one defined in G.726. The word (2*MANT) has a Q7 format with 8 bits, and EXP has a Q0 format with 4 bits. Then, use linear approximation of $\log_2(|d|)$ for the mantissa: $\log_2(|d|)$ = EXP–1 + (2*MANT–1) = (EXP–2) + (2*MANT). After scaling (EXP–2), both are added to form a Q7 word of 11 bits. Note that this method doesn't apply for d = 0, where $\log_2(|d|)$ is defined to be 0.

5.4.8 SUBTB

Function: Scale logarithmic version of difference signal by subtracting scale factor.
$d_{ln}(k) = d_l(k) - y(k)$

Input: A (DL), Y

Outputs: A (DLN)

Cycles: 2

NOTE: Y is 2-bit truncated in order to have the same format as DL (Q7).

5.4.9 QUAN

Function: Quantize difference signal in logarithm domain. $\{d_{ln}(k), \text{sign}[d(k)]\} \rightarrow I(k)$

Input: A (DLN), SIGN (DS)

Output: A (I)

Cycles: Min: 35 (16 Kbps), 52 (24 Kbps), 67 (32 Kbps), Max: 82 (40 Kbps)

NOTE: The level of quantization is determined by a iterative research where DL is compared with low limits of quantization levels QSi. The values of QSi are stored in the tables ITBLxx. A quantization table (QUANxx for the encoder or SYNCxx for the decoder) is then used to give the output code (I for encoder, ID for decoder). See () for details about these tables.

5.4.10 RECONST

Function: Reconstruction of quantized difference signal in the logarithmic domain.
$I(k) \rightarrow \{d_{qln}(|I(k)|, \text{sign}[d_q(k)]\}$

Input: A (I)

Outputs: SIGN (DQS), B (DQLN)

Cycles: 10

NOTE: Two data tables are used for this function. First, a table (IQUAxx) gives the address of the second table (ITBLxx) depending on the value |I|, and also gives the sign of the original difference signal. The second table, which is pointed by AR2, gives directly DQLN(|I|), and further, will give F(|I|) and W(|I|).

5.4.11 ADDA

Function: Addition of scale factor to logarithmic version of quantized difference signal.
$d_{ql}(k) = d_{qln}(k) + y(k)$

Inputs: B (DQLN), Y

Output: B (DQL)

Cycles: 2

NOTE: Y is 2-bit truncated in order to have the same format as DQL (Q7).

5.4.12 ANTILOG

Function: Convert quantized difference signal from the logarithm to the linear domain, in two-complement. $d_q(k) = 2^{dql(k)} * sign[d_q(k)]$

Input: B (DQL), SIGN (DQS)

Output: DQ in two-complement format, A (DQ)

Cycles: Min: 6, Max: 13

NOTE: Computation of 2^{DQL} using decomposition $2^{EXP-1+MANT}$ and linear approximation of $2^{MANT} = 1 + MANT$, where MANT is the mantissa of DQL, and EXP is the exponent of DQL (see routine LOG (7)). Then DQ = (1+ MANT) * 2^{EXP-1}. The sign of DQ, which is given directly by SIGN, allows the completion of the conversion in 2's complement. This method does not apply for DQL $<$ 0, In this case, DQ is zero. According to G.726 recommendation, DQ must be a signed-magnitude word. Here it is represented in 2's complement format to make the calculations easier, with an extra variable for the sign (SIGN = DQS = sign(DQ)). The recommendation also indicates that DQ can be coded with 15 bits (for 16-, 24-, or 32-Kbps operation), or with 16 bits (for 16-, 24-, 32-, or 40-Kbps operation). For these purposes (all rates simultaneously available), a 16-bit word must be chosen; however, with 2's complement format, it makes no difference.

5.4.13 ADDB

Function: Addition of quantized difference signal and signal estimate to form reconstructed signal. $s_r(k) = s_e(k) + d_q(k)$

Inputs: A (DQ), *AR4 (SE)

Output: SD (SR)

Cycles: 2

NOTE: DQ format is a 16-bit word (Q0), and SE format is a 14-bit word (Q0). Overflow is always avoided in 32, 24, or 16 Kbps coding where DQ $< 2^{14}$. Nevertheless, in 40-Kbps coding, SR is limited to be a 16-bit word (Q0 format).

5.4.14 ADDC

Function: Obtain sign of addition of quantized difference signal and partial signal estimate. $p(k) = d_q(k) + s_{ez}(k)$, $sign[p(k)] = sign(dq(k) + sez(k))$

Input: A(DQ), SEZ

Output: $PK0_{high}$ (PK0 = sgn(p(k)), $PK0_{low}$ (p(K) that gives information about real p(k) sign)

Cycles: 2

NOTE: PK0 gives computing sign (0 for positive, –1 for negative), and the real sign is given by p(k) (variable $PK0_{low}$ = V1). The real sign is worth 1 if p(k) positive, –1 if p(k) is negative, and is defined to be 0 if p(k) = 0 (but, once delayed it is worth 1; see ()). In this special case, the adaptation of A1 and A2 are different.

5.4.15 FUNCTF

Function: Map quantizer output into the F(I) function. $I(k) \rightarrow F[|I(k)|]$

Input: AR2 (points to F[|I|])

Output: A (F[|I|] << 9)

Cycles: 1

NOTE: Load F[|I|] << 9 to scale it with DMS (Q9 format) for routine FILTA (16). Values of F|I| are included in the |I| table, pointed by AR2.

5.4.16 FILTA

Function: Update of short-term average of F(I). $d_{ms}(k) = (1 - 2^{-5})d_{ms}(k-1) + 2^{-5} F[|I(k)|]$

Inputs: A (F[|I|] << 9), DMS

Output: DMS

Cycles: 4

5.4.17 FILTB

Function: Update of long-term average of F(I). $d_{ml}(k) = (1 - 2^{-7})d_{ml}(k-1) + 2^{-7} F[|I(k)|]$

Inputs: AR2 (points to F[|I|]), DML

Output: DML, AR2 (points to W[|I|])

Cycles: 5

NOTE: You load F[|I|] << 11 to scale it with DML (Q11 format).

5.4.18 TRANS

Function: Transition detector. $t_r(k) = 1 \Leftrightarrow t_d(k-1) = 1$ and $|d_q(k)| > 24 * 2^{yl(k-1)}$

Inputs: TD, YL (high word), DQ

Output: branch to UPA2 () (TR = 0), branch to TRIGB (TR = 1)

Cycles: Min: 6 (usual), Max: 25

NOTE: At the end of this routine, either perform TRIGA and TRIGB, or continue the program normally. Note that G.726 recommendation indicates in the text $t_d(k)$ instead of $t_d(k-1)$, and $y_l(k)$ instead of $y_l(k-1)$, but it is in contradiction with the further block description.

5.4.19 TRIGA

Function: Speed control trigger block. $a_p(k) = a_p(k-1)$ if $t_r(k) = 0$, $a_p(k) = 1$ if $t_r(k) = 1$

Inputs: B (B $> \quad 0 \Leftrightarrow$ tr = 1)

Outputs: AP

Cycles: Min: 0 (usual), Max: 2

NOTE: By using long-word instruction, $t_d(k)$ is initialized at the same time as $a_p(k)$, performing by this way a part of TRIGB (20).

5.4.20 TRIGB

Function: Predictor trigger block. If $t_r(k) = 1$, $a_i(k) = 0$ for i = 1, 2; $b_i(k) = 0$ for i = 1, ..., 6; and $t_d(k) = 0$

Input: none

Output: A1, A2, B1, ..., B6, TD, and branch to FUNCTW() if performed

Cycles: Min: 0 (usual), Max: 7

NOTE: This routine (just as TRIGA) is executed in function of the precedent “conditional branch” at the end of the routine, TRANS (18). See TRIGA (19) for execution conditions. If tr(k) = 1 (transition is detected), TRIGB is performed: A1, A2, B1, ..., B6 are set to 0, then routines (21) to (29) are skipped to avoid Ai and Bi adaptation. Otherwise, (transition not detected), TRIGB is not performed: Ai, Bi, and TD keep their value, and are then adapted (for Ai, Bi: routines (21–23) and (25–26)), or calculated (for TD). Note that TD is initialized at the same time as AP (see TRIGA (19)).

5.4.21 UPA2

Function: Update a_2 coefficient of second order predictor. $a_2(k) = (1 - 2^{-7})a_2(k-1) + 2^{-7}\{sgn[p(k)]\ sgn[p(k-2)] - f[a_1(k-1)]\ sgn[p(k)]\ sgn[p(k-1)]\}$

Inputs: $PK0_{high}$ (PK0 = sgn(p(k)), $PK0_{low}$ (P(k)), PK1, PK2, A1, A2

Outputs: A (unlimited A2)

Cycles: Min: 0, Max: 25 (usual)

NOTE: $PK0_{low}$ makes it possible to give the real sign of p(k). In fact, it is worth 0 if p(k) = 0. Otherwise, it is given by PK0 (0/–1) and is worth +/– 1. This routine is not performed if $t_r(k) = 1$.

5.4.22 LIMC

Function: Limits on a_2 coefficient of second order predictor. $|a_2(k)| \leq 0.75$

Input: A (unlimited A2)

Output: A2

Cycles: Min: 0, Max: 6 (usual)

NOTE: Computing value for 0.75 is 12288 (Q14 format). This routine is not performed if $t_r(k) = 1$.

5.4.23 UPA1

Function: Update a_1 coefficient of second order predictor. $a_1(k) = (1 - 2^{-8})a_1(k-1) + 3.2^{-8}$ sgn[p(k)] sgn[p(k–1)]

Inputs: $PK0_{high}$ (PK0 = sgn(p(k)), $PK0_{low}$ (P(k)), PK1, A1

Outputs: A (A1T)

Cycles: Min: 6, Max: 12 (usual)

NOTE: $PK0_{low}$ permits, to give the real sign of p(k). It is 0 if p(k) = 0, otherwise, it is given by PK0 (0/–1) and is worth +/–1. This routine is not performed if $t_r(k) = 1$.

5.4.24 LIMD

Function: Limits on a1 coefficient of second order predictor. $|a_1(k)| \leq 1 - 2^{-4} - a_2(k)$

Input: A (A1T), A2 (A2P)

Output: A1 (A1P)

Cycles: Min: 0, Max: 7 (usual)

NOTE: Computing value for $1 - 2^{-4}$ is 15360 (Q14 format). This routine is not performed if $t_r(k) = 1$.

5.4.25 XOR

Function: “Exclusive or” of sign of difference signal and sign of delayed difference signal. $U_i(k)$ = sign ($d_q(k)$) ** sign ($d_q(k-i)$)

Inputs: DQ1 sign, ..., DQ6 sign, SIGN (DQS), AR7 (points to DQ6 sign)

Output: B (Ui)

Cycles: Min: 0, Max: 12 (usual)

NOTE: DQi sign is pointed by AR7 in table DQFLOAT. This routine is partially executed before each UPBi (26). This routine is not performed if $t_r(k) = 1$.

5.4.26 UPB

Function: Update for coefficients of sixth-order predictor. $b_i(k) = (1–2^{–8})b_i(k–1) + 2^{–7}\ U_i(k)$ for 16-, 24-, 32-Kbps coding; $b_i(k) = (1–2^{–9})b_i(k–1) + 2^{–7}\ U_i(k)$ for 40 Kbps coding

Inputs: B (Ui), Bi, DQ, SHIFT

Outputs: B1 (B1P), ..., B6 (B6P)

Cycles: Min: 0, Max: 40 (usual)

NOTE: If DQ = 0, then Ui is forced to be 0. SHIFT is –8 for 16, 24, 32 Kbps, and is –9 for 40 Kbps coding. It corresponds to the term $2^{–8}$ or $2^{–9}$ for Bi adaptation. This routine is not performed if $t_r(k) = 1$.

5.4.27 TONE

Function: Partial band-signal detection. $t_d(k) = 1$ if $a_2(k) < –0.71875$, $t_d(k) = 0$. Otherwise:

Input: A2 (A2P)

Output: TD (TDP)

Cycles: Min: 0, Max: 3

NOTE: Computing value for –0.71875 is –11776 (Q14 format). This routine is not performed if $t_r(k) = 1$.

5.4.28 SUBTC

Function: Compute magnitude of the difference of short- and long-term functions of quantizer output sequence, and then perform threshold comparison for quantizing speed control parameter. Ax = 0 if $y(k) \geq 3$ and $|d_{ms}(k–1) – d_{ml}(k–1)| \geq 2^{–3}\ d_{ml}(k–1)$ and $t_d(k–1) = 0$, Ax = 1. Otherwise:

Inputs: DMS, DML, TD (TDP), Y

Output: A (AX << 9)

Cycles: Min: 0 (rare), Med: 6, Max: 21

NOTE: If $t_d(k) = 1$ (TD = –1), the routine is limited to 6 execution cycles, else if $|dms(k–1) – dml(k–1)| \geq 2^{–3}\ dml(k–1)$, the routine is limited to 19 cycles. Otherwise, the cycle number is 21. If $t_r(k) = 1$, the routine is not performed.

5.4.29 FILTC

Function: Low-pass filter of speed control parameter. $a_p(k) = (1– 2^{–4})a_p(k–1) + 2^{–3}$ Ax

Inputs: A (AX << 9), AP

Output: AP (APP)

Cycles: Min: 0, Max: 4 (usual)

NOTE: This routine is not performed if $t_r(k) = 1$. AX << 9 is either 2^9 or 0. It corresponds to 2 for AP scale (Q8 format). The difference between AX << 9 and AP is computed before dividing the result by 16.

5.4.30 FUNCTW

Function: Map quantizer output into logarithmic. $I(k) \rightarrow W[|I(k)|]$

Input: AR2 (points to W[|I|])

Output: A (W[|I|] << 5

Cycles: 1

NOTE: You load W[|I|] << 5 to scale it with DMS (Q9 format) for routine FILTD (31). Values of W|I| are included in the |I| table pointed by AR2.

5.4.31 FILTD

Function: Update of fast quantizer scale factor. $y_u(k) = (1 - 2^{-5}).y(k) + 2^{-5}.W[|I(k)|]$

Inputs: A(W[|I|] << 5), Y

Output: BH (YUT)

Cycles: 3

NOTE: To prepare FILTE (33), operations are carried out using the high part of the accumulator.

5.4.32 LIMB

Function: Limit quantizer scale factor. $1.06 \leq y(k) \leq 10$

Input: BH (YUT)

Output: YU (YUP), AH (YUP)

Cycles: 5

NOTE: Computing value for 1.06 is 544, and 5120 for 10 (Q9 format).

5.4.33 FILTE

Function: Update of slow quantizer scale factor. $y_l = (1 - 2^{-6}).y_l(k-1) + 2^{-6}.y_u(k)$

Inputs: YL, AH (YUP)

Outputs: YL (YLP)

Cycles: 5

NOTE: YL must be calculated before applying the 2^{-6} factor. As the theoretical format of YL is Q25, (–YL) is truncated to obtain Q15 format, as specified in G.726 recommendation.

5.4.34 FLOATA

Function: Convert 16-bit 2's complement to floating-point. DQ $\rightarrow$ (DQ0 mantissa, DQ0 exponent, DQ0 sign)

Input: DQ, SIGN (DQS), AR7 (points to DQ6 exponent)

Output: DQ0 exponent, DQ0 mantissa, DQ0 sign (in DQFLOAT buffer), AR7 (points to DQ5 exponent)

Cycles: 13

NOTE: Exponent EXP of DQ is defined by: $2^{EXP-1} \leq |DQ|\ 2^{EXP}$. Mantissa MANT of DQ is obtained with the 6 most significant bits (MSB) of DQ. MANT has the following limits: $0 \leq$ MANT $<$ 1, which implies that MANT format is Q6. Sign of DQ is 0 if DQ positive, –1 if DQ negative. If DQ $\neq$ 0, find the exponent of DQ by means of the EXP and DSUBT instructions, and MANT via the NORM instruction. If DQ = 0, EXP is 0 and MANT is defined to be 1/2 (= 32).For this particular case, the sign is given by the variable SIGN (DQS = sign of DQ given by the inverse quantizer). Note that DQ = 0 does not imply that DQS = 0.

5.4.35 FLOATB

Function: Convert 16-bit 2's complement to floating point. SR $\rightarrow$ (SR0 mantissa, SR0 exponent, SR0 sign)

Input: SD (SR), AR6 (points to SR2 sign)

Output: SR0, AR5 (points to SR1 exponent)

Cycles: 14

NOTE: See FLOATA for definitions of exponent, mantissa, and sign; but contrary to DQ, SR = 0 always implies that the sign of SR is also 0 (this means that, in this case, SR is positive).

5.4.36 DELAY

Function: Memory block. For the input x, the output is given by y(k) = x(k–1).

Input: x

Output: y

Cycles: 10

NOTE: This routine applies for all delayed variables. For the one-time delayed variables such as DMS, DML, AP, Ai, Bi, TD, YL, YU, x(k + 1) has the same memory location as x(k). So, the delay was applied at the time these variables were updated. Thus, this routine really applies to the variables that are delayed several times, such as Pki, SRi, DQi. As for PKi, PK2 location just follows the PK1 location, so the instruction DELAY automatically realizes the PK2 update. In the case of Sri and Sri, these variables are automatically delayed, due to the use of two circular buffers (DQFLOAT for DQi and SRFLOAT for SRi). Only the addresses of the next DQ6 and the next SR2 must be saved (ADDQ6 and ADSR2).

5.4.37 COMPRESS (decoder only)

Function: Convert from uniform PCM to either A-law or μ-law PCM. $s_r(k) \rightarrow s_p(k)$

Input: A (SR), LAW, LAWBIAS, LAWSEG, LAWMASK

Output: A (SP)

Cycles: Min: 20, Max: 26

NOTE: This generic routine is used for both A-law and μ-law, due to the use of the variables, LAWxxxx, which make the discrimination between laws for this quantization. First, the A-law PCM word is re-converted into 13-bit signed word by dividing it by two. It is actually divided by four to perform linear quantization directly, but in the case of logarithmic quantization, this right-shift is then compensated. For negative A-law PCM word, subtract one from it before dividing, to perform a correct truncation. Note that G.726 recommendation indicates adding one to it, but it seems to be a printing error. The principle of the logarithm calculation is the same as in LOG (7). See () for more details about PCM companding.

5.4.38 SYNC (decoder only)

Function: Re-encode output PCM sample in decoder for synchronous tandem coding.
$s_p(k) \rightarrow s_d(k)$

Input: B (ID), *AR1 (IM), SD (SP), LAWMASK

Output: A (SD)

Cycles: Min: 6 (usual), Med: 21, Max: 25

NOTE: After using the routines EXPAND (), SUBTA (), LOG (), SUBTB () again to perform this synchronous adjustment, the routine QUAN () is also used to re-encode the PCM output word. The new coded word is ID, and is compared with the magnitude (given by *AR1) of the original I ADPCM word. In the case where ID = IM, which is the usual case, the routine takes only six clock cycles.

6 References

1. 40-, 32-, 24-, 16-Kbps Adaptive Differential Pulse Code Modulation (ADPCM) Recommendation G.726. General aspects of digital transmission systems: terminal equipments. ITU (International Telecommunication Union). CCITT (International Telegraph and Telephone Consultative committee). Geneva, 1990.
2. Digital Coding of Speech Waveforms: PCM, DPCM, and DM Quantizers. Jayant, Nuggehally S. Proceedings of the IEE, Vol. 62, No. 5, May 1974, pp 612–633.
3. *A Simple Approach to DSP* (SPRDE01).
4. *TMS320C54x DSP CPU and Peripherals Reference Set Volume 1* (SPRU131).
5. Digital Coding of Waveforms. Jayant, N.S. and Noll, P. Prentice Hall: Englewood Cliffs, NJ, 1984.
6. Digital Signal Processing Design. Bateman, A. and Yates, W. Pitman Publishing, London, U.K., 1988.
7. Comparison of Nth Order DPCM Encoder with Linear Transformations and Block Quantization Techniques. Habibi, A. IEEE Transactions on Communications, December 1971, pp. 948–956.
8. Digital Signal Processing with the TMS320 Family, Volume 1, Lin, Kun-Shan. Prentice Hall, Englewood Cliffs, NJ. 1987.

For more information, go to the Texas Instruments web site at http://www.ti.com/

IMPORTANT NOTICE

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265