The Implementation of G.726 Adaptive Differential Pulse Code Modulation (ADPCM) on TMS320C54x DSP

Literature Number: BPRA053
Texas Instruments Europe
July 1997

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Introducing ADPCM

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's why it's different from vocoders (voice encoders CELP, VSELP,...), that use properties of the human voice in order 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.

Let us consider a signal, which we may wish to transform in order to reduce the amount of information that it is necessary to code. Now, we must be able to reconstruct the original signal as faithfully as possible. The principle of ADPCM is to use our 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 PCM (Pulse Code Modulation).

PCM is the most direct way to code analog signals to digital. The code word 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.

Therefore we need to perform PCM before ADPCM; the aim being to decrease the number of bits for coding by passing through a PCM process before transforming to an ADPCM sample. In the G726 recommendation - which currently includes G721 and G723 recommendations of the CCITT (The International Telegraph and Telephone Consultative Committee) - 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 ADPCM word represents the prediction error of the signal, and has no significance itself.. It must be decoded (reverse transformation) in order 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.

1.2 Introducing the Software Application

As we said, ADPCM is a complete digital transcoding process. According to the CCITT standard, if the PCM input bit flow is 64 kbit/s (8 kHz sampling x 8-bit PCM word), we have to process in real-time in order to produce a 40, 32, 24, or 16 kbit/s (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 reaching doing this. In particular, the new and very efficient TMS320C54x, also called 'LEAD' (for Low-power consumption Enhanced Architecture Device), enables very rapid processing.

1.3 Software Features

- 16, 24, 32, or 40 kbit/s G726 ADPCM (including G721 / G723) 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 thanks to dynamic allocation of memory
- 8.3 - 10 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, G721 (only 32 kbit/s, while 40 kbit/s requires additional instructions) ADPCM on 'C50x requires 10.5 - 12.5 MIPS
- Around 1500 words ROM requirement (program: 680, data: 850).
- Less than a data page 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 too.

2. CCITT ADPCM Standard: Recommendation G726

This chapter is a reproduction of the G.726[1] Recommendation § 1-3 developed by the CCITT. The CCITT (the International Telegraph and Telephone Consultative Committee) is a permanent organ of the International Telecommunication Union (ITU).

This text part 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 kbit/s quantizer that takes one of 15 non-zero values (see § 2.2.3). Secondly, the transition detector equation has been corrected. In fact, the sampling index seems to be (k-1) instead of (k) for y_l and a_2 (see § 2.2.8) [2].

2.1 ADPCM principle

The characteristics below are recommended for the conversion of a 64 kbit/s A-law or μ-law Pulse Code Modulation (PCM) channel to and from a 40, 32, 24 or 16 kbit/s 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.

[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.

[2] While reporting printing errors, specification of IMAG in the COMPRES sub-block (G726/§ 4, not reproduced here), contains probably an error. In the case of LAW = 1 and IS = 1, IMAG = (IM -1)>>1, and not (IM+1)>>1

The principal application of 24 and 16 kbit/s channels is for overload channels carrying voice in Digital Circuit Equipment (DCME).

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

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

Figure 1.

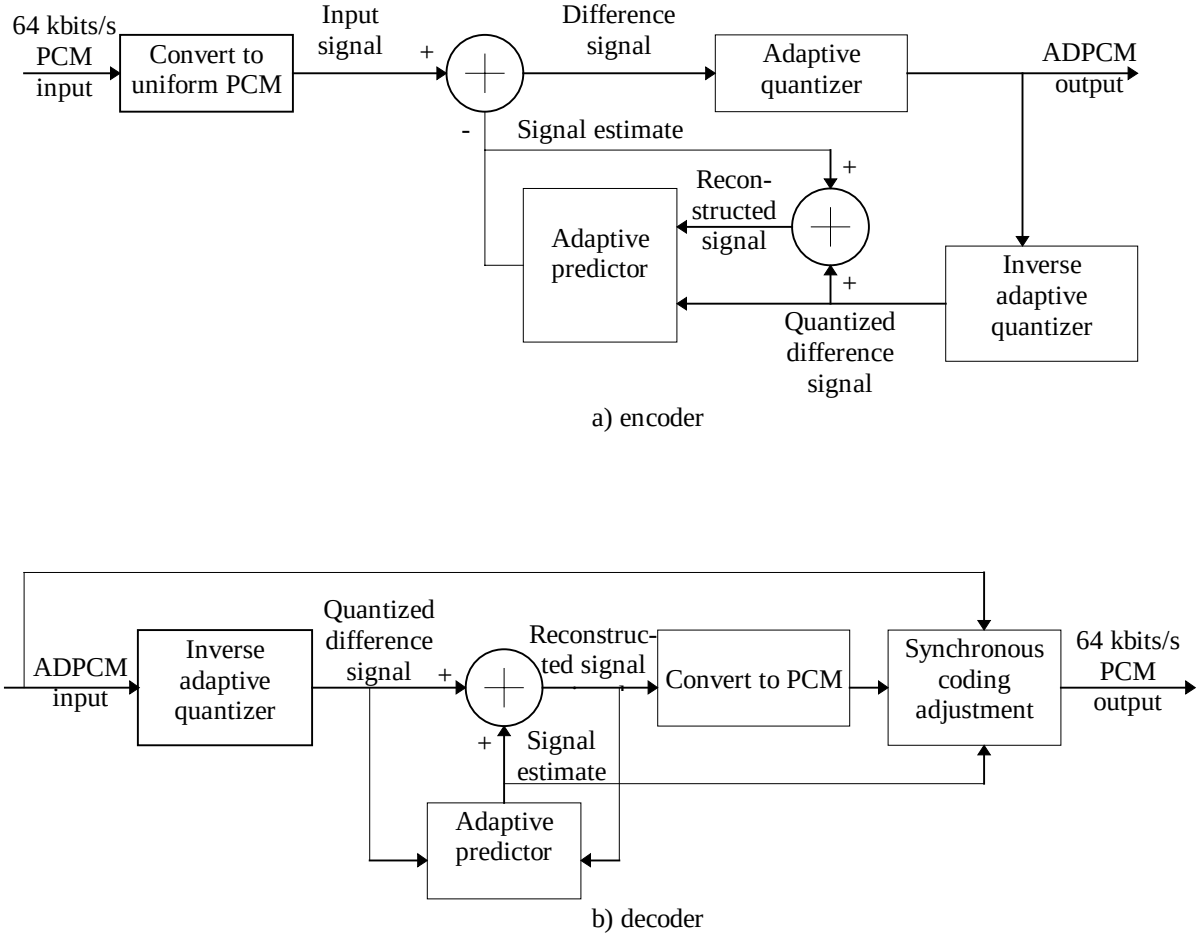

Figure 1: ADPCM Encoder and Decoder

2.1.1 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.

2.1.2 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 § 2.3.7). 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.

2.2 Encoder Description

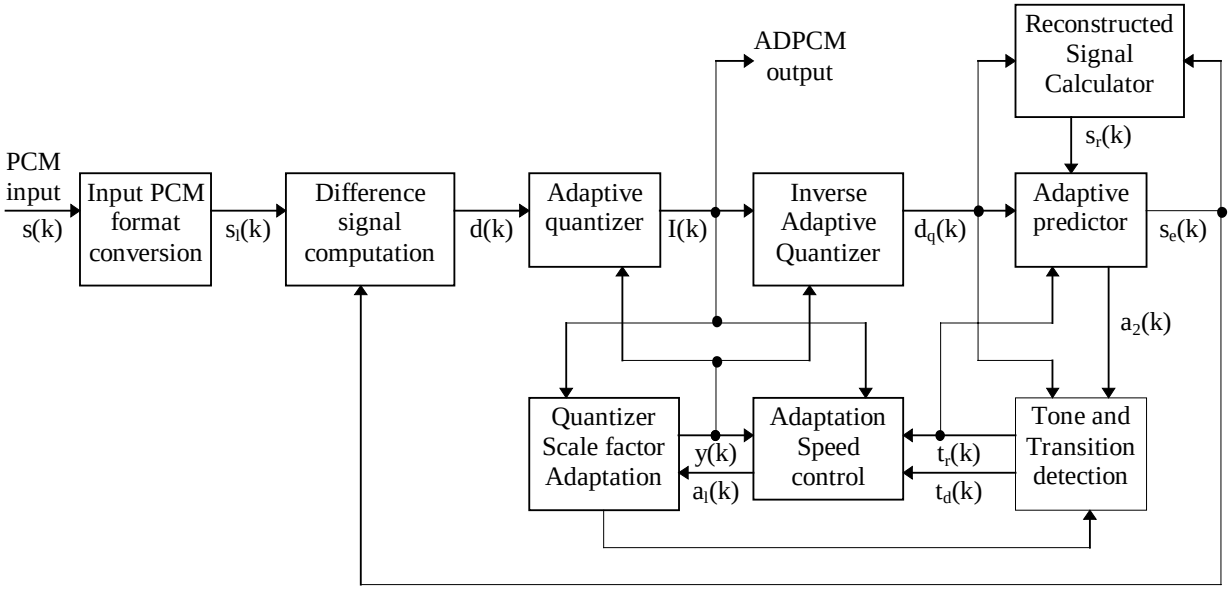

Figure 2: Encoder Block Schematic

Figure 2 is 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 §§ 2.2.1 to 2.2.8.

2.2.1 Input PCM Format Conversion

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

2.2.2 Difference Computation

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

$$d(k) = s_l(k) - s_e(k) \qquad \text{(2-1A)}$$

2.2.3 Adaptive Quantizer

A 31, 15, 7, 4-level non-uniform adaptive quantizer is used to quantize the difference signal *d(k)* for operating at 40, 32, 24 or 16 kbit/s, 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:

$$d_{\ln}(k) = \log_2(d_l(k)) - y(k) \qquad \text{(2-1B)}$$

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

2.2.3.1 Operation at 40 kbit/s

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 kbit/s 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 kbit/s 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
[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 tables 1 through 4, "[" indicates that the endpoint value is included in the range, and "(" or ")" indicates that the endpoint value is excluded from the range.

2.2.3.2 Operation at 32 kbit/s

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 kbit/s 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 kbit/s 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	-∞

2.2.3.3 Operation at 24 kbit/s

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 kbit/s 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 control 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 kbit/s 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	-∞

2.2.3.4 Operation at 16 kbit/s

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 kbit/s 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 kbit/s 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

2.2.4 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 given in Table 1 through Table 4 and then transforming the result from the logarithmic domain:

$$d_q(k) = 2^{d_{q\ln}(k)+y(k)} \qquad \text{(2-1C)}$$

2.2.5 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 parameter $a_l(k)$.

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

- fast for signals (i.e., speech), that produce difference signals with large fluctuations;
- slow for signals (i.e., 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 $y(k)$:

$$y_u(k) = (1-2^{-5})y(k) + 2^{-5}W|I(k)| \qquad \text{(2-2)}$$

where $y_u(k)$ is limited by:

$$1.06 \le y_u(k) \le 10.00$$

For 40 kbit/s 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 kbit/s 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 kbit/s 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 kbit/s 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 factors $y_l(k)$ is derived from $y_u(k)$ with a low pass-filter operation:

$$y_l(k) = (1-2^{-6})y_l(k-1) + 2^{-6}y_u(k) \qquad (2\text{-}3)$$

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

$$y(k) = a_l(k)y_u(k-1) + (1-a_l(k))y_l(k-1) \qquad (2\text{-}4)$$

Where

$$0 \le a_l(k) \le 1$$

2.2.6 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:

$$d_{ms}(k) = (1-2^{-5})d_{ms}(k-1) + 2^{-5}F|I(k)| \qquad (2\text{-}5)$$

and

$$d_{ml}(k) = (1-2^{-7})d_{ml}(k-1) + 2^{-7}F|I(k)| \qquad (2\text{-}6)$$

For 40 kbit/s 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 kbit/s 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 kbit/s ADPCM, *F|I(k)|* is defined by:

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

For 16 kbit/s 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:

$$a_p(k) = \begin{cases} (1-2^{-4})a_p(k-1) + 2^{-3}, \textbf{ if } |d_{ms}(k) - d_{ml}(k)| \geq 2^{-3}d_{ml}(k) \\ (1-2^{-4})a_p(k-1) + 2^{-3}, \textbf{ if } y(k) < 3 \\ (1-2^{-4})a_p(k-1) + 2^{-3}, \textbf{ if } t_d(k) = 1 \\ 1, \textbf{ if } t_r(k) = 1 \\ (1-2^{-4})a_p(k-1), \textbf{ otherwise} \end{cases} \qquad (2\text{-}7)$$

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 § 8). Note that $a_p(k)$ is set to 1 upon detection of a partial band signal transition (indicated by $t_r(k) = 1$, see § 8).

$a_p(k)$ is then limited to yield $a_l(k)$ used in equation (2-4) above:

$$a_l(k) = \begin{cases} 1, & a_p(k-1) > 1 \\ a_p(k-1), & a_p(k-1) \le 1 \end{cases} \qquad (2\text{-}8)$$

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.

2.2.7 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:

$$s_e(k) = \sum_{i=1}^{2} a_i(k-1) s_r(k-i) + s_{ez}(k), \qquad (2\text{-}9)$$

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

and the reconstructed signal is defined as

$$s_r(k-i) = s_e(k-i) + d_q(k-i)$$

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

$$a_1(k) = (1-2^{-8}) a_1(k-1) + 3 \cdot 2^{-8} \operatorname{sgn}[p(k)] \operatorname{sgn}[p(k-1)], \qquad (2\text{-}10)$$

$$a_2(k) = (1-2^{-7}) a_2(k-1) + 2^{-7} \{\operatorname{sgn}[p(k)] \operatorname{sgn}[p(k-2)] - f[a_1(k-1)] \operatorname{sgn}[p(k)] \operatorname{sgn}[p(k-1)]\}, \qquad (2\text{-}11)$$

Where

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

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

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:

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

If $t_r(k) = 1$ (see § 8), then $a_1(k) = a_2(k) = 0$.

For the sixth order predictor:

$$b_i(k) = (1 - 2^{-8})b_i(k-1) + 2^{-7}\,\mathrm{sgn}[d_q(k)]\mathrm{sgn}[d_q(k-i)]\ , \qquad \text{(2-12A)}$$

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

For 40 kbit/s coding, the adaptive predictor is changed to decrease the leak factor used for zeros coefficient operation. In this case, Equation 2.12A becomes:

$$b_i(k) = (1 - 2^{-9})b_i(k-1) + 2^{-7}\,\mathrm{sgn}[d_q(k)]\mathrm{sgn}[d_q(k-i)]\ . \qquad \text{(2-12B)}$$

If $t_r(k) = 1$ (see § 8), then $b_1(k) = b_2(k) = ... = b_6(k) = 0$.

As 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.

2.2.8 Tone and Transition Detector

In order 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 (e.g. tone) detection is invoked so that the quantizer can be driven into the fast mode of adaptation:

$$t_d(k) = \begin{cases} 1, & a_2(k) < -0.71875 \\ 0, & \textbf{otherwise} \end{cases} \qquad \text{(2-13)}$$

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:

$$t_r(k) = \begin{cases} 1, & a_2(k-1) < -0.71875 \textbf{ and } |d_q(k)| > 24 \cdot 2^{y_l(k-1)} \\ 0, & \textbf{otherwise} \end{cases} \quad (2\text{-}14)$$

2.3 Decoder Description

Figure 3 is a block schematic of the decoder. A functional description of each block is given in §§ 2.3.1 to 2.3.7 below.

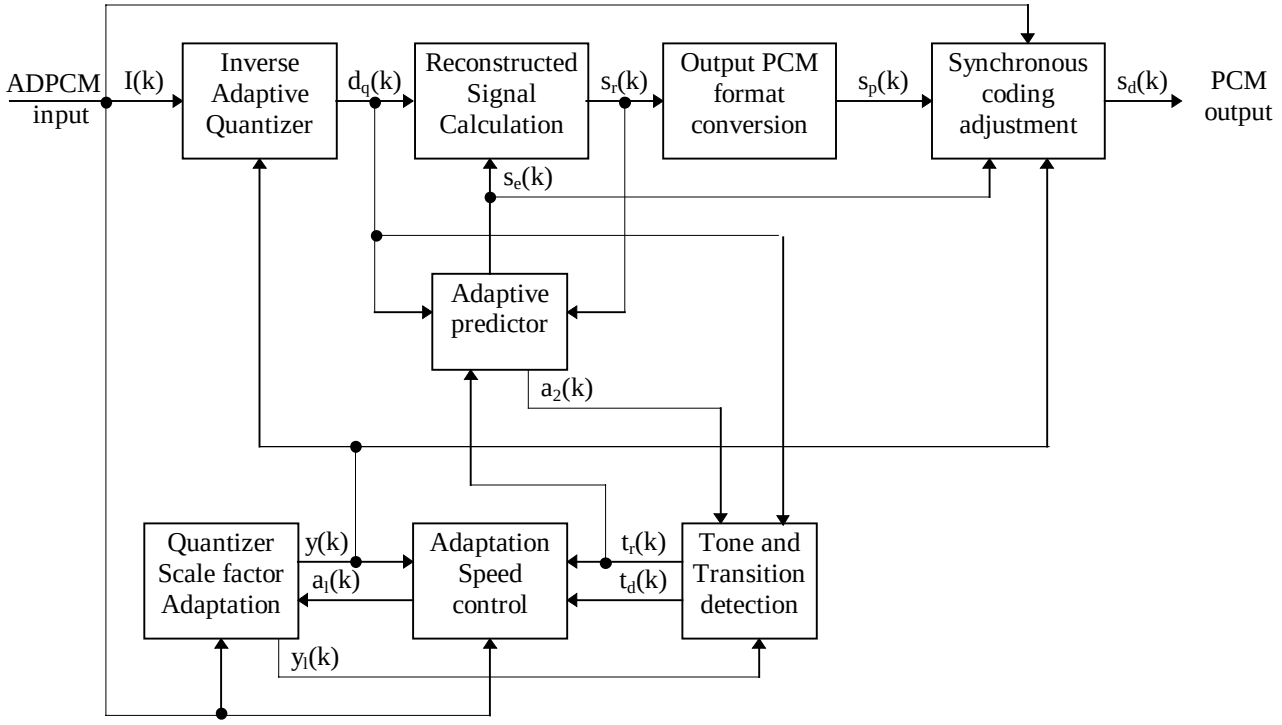

Figure 3: Decoder Block Schematic

2.3.1 Inverse Adaptive Quantizer

The function of this block is described in § 2.2.4.

2.3.2 Quantizer Scale Factor Adaptation

The function of this block is described in § 2.2.5.

2.3.3 Adaptation Speed Control

The function of this block is described in § 2.2.6.

2.3.4 Adaptive Predictor and Reconstructed Signal Calculator

The function of this block is described in § 2.2.7.

2.3.5 Tone and Transition Detector

The function of this block is described in § 2.2.8.

2.3.6 Output PCM Format Conversion

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

2.3.7 Synchronous Coding Adjustment

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

i) the transmission of the ADPCM and the intermediate 64 kbit/s PCM signals is error free, and,
ii) the ADPCM and intermediate 64 kbit/s 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 to 3400 Hz band (e.g. 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 guarantees 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 $d_x(k)$:

$$d_x(k) = s_{lx}(k) - s_e(k) \ . \qquad (3\text{-}1)$$

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:

$$s_d(k) = \begin{cases} s_p^+(k), & d_x(k) < \textbf{ lower interval boundary} \\ s_p^-(k), & d_x(k) \geq \textbf{ upper interval boundary} \\ s_p(k), & \textbf{otherwise} \end{cases} \qquad (3\text{-}2)$$

Where

$s_d(k)$ is the output PCM code word of the decoder

$s_p^+(k)$ is the PCM code word 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 code word 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. The LEAD for ADPCM Purpose

3.1 General Presentation of the Useful Features of the ‘C54x for G726 ADPCM

The typical application for LEAD 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 LEAD, 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 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 (G726 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, let us see modules whose implementation on LEAD requires some comment.

3.2 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 G711 recommendation.

However, linear/logarithmic PCM conversions are included in the CCITT ADPCM recommendation (G726) in order 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. In order to not loose precision, PCM words coming from A-law are also scaled into 14-bit words.

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 we give 'SD' as input encoder, we have also to find 'I' as output. This feature is already guaranteed for the linear PCM output word, thanks to the feedback of the ADPCM quantization error in signal estimation (the encoder also includes a decoding module). However, this property is no longer guaranteed 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.

Below, we show how these format conversions and corrections are implemented in the 'C54x for both G726 and G711 recommendations. The routines are valid for either A- or μ-law. Tables and specific variables make the distinction between the two.

3.2.1 Log-PCM Companding

This module converts a linear PCM word into the logarithmic domain. As we saw, a word coming from A-law has been one bit left shifted in order to get the maximum resolution in the ADPCM algorithm. Now we 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.

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

The logarithm calculation is quickly performed using the EXP and NORM instructions. See § 3.5 (Logarithmic Conversion) for the method. The first difference is that normalization is done in Q4 format instead of Q7. And as we compute logarithmically only for large magnitudes (≥ 32), we 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 § 3.3.1 or in § 3.5[1]

It 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 upper).

Then, we restore the sign to the magnitude, by adding 128 for positive PCM words. Lastly, we invert bits (only even bits for A-law), in order 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 G726 highlighted in bold characters. The rest is sufficient for G711. 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             ; guarantees 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 G711)
       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
```

[1] Note that in logarithmic calculation, we use (*exponent* - 1), instead of *segment* (see § 3.5).

3.2.2 Linear PCM Expanding

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

In order to reduce the clock cycles timing, tables are used for this PCM expansion. As outputs levels are symmetrical relative to zero, we 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 we 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.

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

The routine shown here executes in 13 cycles; 2 of these cycles can be used to execute initialization instructions for following blocks. For example, if you perform G726 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 § 3.6).

```
*******************************************************************************
*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
      STLM  A, 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
```

3.2.3 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 compare 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, in order to make the comparison with *ID* possible.

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

Now, note that *SP* is a signed magnitude, and not two-complement. This does not allow us to perform normal arithmetic. For instance, SP + 1 when SP is negative 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. [1]

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 where 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+
```

[1] This subtlety is explained in 0

3.3 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, but 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 a 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.

We describe now how floating-point format can be challenged by the ‘C54x for the G726 recommendation.

3.3.1 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:

½ ≤ denormalized mantissa < 1 Mantissa is normalized with 6 bits representation, so:

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

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

To make the access faster, we choose to 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 § 3.11).

3.3.2 Floating-Point Conversion

When loaded into accumulator, the sign of a word (as defined in § 3.11) is the value of the high part of accumulator. The sign is thus extracted thanks to *STH* instruction. Then, we compute the magnitude with *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 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, we 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 low part is set to 31).

This method does not apply if the input word to convert is 0; indeed, *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 kbit/s (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)
```

3.3.3 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)
```

3.4 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 we 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 we 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-words length, while DQFLOAT has an 18-words 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 our 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), we saw an example of circular addressing in the last indirect addressing access.

3.5 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 we 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, we use the same characteristic as floating-point format (see § 3.3.1)

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

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:

$$½ \leq \text{denormalized mantissa} < 1 \Rightarrow 1 \leq (\text{denormalized mantissa}) * 2 < 2$$

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

When we take the base two logarithm, we 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 ($\log_2(1+x) \approx x$), we obtain

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

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

$128 \leq$ 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
```

3.6 3, 4, or 5-bit Quantizer

A 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). We use the block repeat capability in order to limit the number of cycles used for this loop function.

This quantizer is valid for 16, 24, 32, or 40 kbit/s 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 kbit/s coding).

The table QUANxx provides ADPCM code for the encoder, while the table SYNCxx provides *ID* (see § 3.2.3) 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 we consider 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 guarantees that this number is constant for a given rate; that is, the number of loop is 1, 2, 3, or 4 respectively for 16, 24, 32, or 40 kbit/s. So 16 kbit/s ADPCM encoding is faster than 40 kbit/s encoding. This constant number of loops makes it possible to avoid 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. 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 40 kbit/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)
```

3.7 Inverse Quantizer

From an ADPCM word, the inverse quantizer gives the quantized difference signal, in order to reconstruct the original signal. It is, of course, the basis function of the decoder. However, it is also used for the encoder in order 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 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 synchronous adjustment module, in decoder.

These values are given via the table IQUAxx (xx = 16, 24, 32, 40 for 16, 24, 32, 40 kbit/s 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.

3.8 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 G726 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, in order 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, in order to set two variables with one-cycle instruction, as we can see below. The constraint for this capability is the alignment of the long words on even boundaries.

```
*******************************************************************************
*      Reset of a_i(k), b_i(k), t_d(k), a_p(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
```

3.9 Double Precision / Dual 16-bit Arithmetic Use

The LEAD 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 G726 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 y_l(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 y_l(k) updating: double-word y_l(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 double-word arithmetic is to consider a long-word as two different variables for which a double access would be possible. We have already seen an

example with the trigger process (see § 3.8). 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 § 3.11), 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 kbit/s coding (dq coded with 15 bits) ****
**** so another solution was finally chosen to satisfy 40 kbit/s 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 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, the instruction *DSUBT*, subtract TREG from long-word; in our purpose, it makes it possible to directly compute the exponent, in floating-point conversion (see § 3.3.2).

3.10 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. We show here, 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)
```

3.11 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:

$x = |x| * sign(x)$. (1)

However this sign representation is not very significant in computing arithmetic when we use the two's complement format. In this format sign bits are non-significant leading bits: zero for positive number, one for negative numbers. As a consequence, when we load 16-bit data in the 40-bit accumulator of the 'C54x, high part of accumulator contains 16 sign bits of the data. These can be easily stored in memory thanks to *STH* instruction. We choose this representation for 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 G726 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 we 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, we have the following equivalence:

sign(x) * sign(y) $\Leftrightarrow$ sign(x) ** sign(y) in computing arithmetic,

Where ** designates logical *XOR*. Let 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
```

3.12 Coder Rate & PCM Laws Selection

The decision of coder rate (16, 24, 32, or 40 kbit/s) 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 4.1. DATA-RAM Space) 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*. In order 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). We show here, the ‘SELECT’ routine

```
*******************************************************************************
*                       CODER SELECTION AND INITIALIZATION                    *
* Initialize tables addresses of quantizer, inverse quantizer and |I|         *
* functions, depending on the flow rate of the coder (16/24/32/40 kbit/s)     *
* Initialize auxiliary registers for common usage                             *
*                                                                             *
*      INPUT:                                                                 *
* AR0           = coder/decoder selection                                     *
*               = 1 for encoder                                               *
*               = 2 for decoder                                               *
* LAW:          = LAW: PCM law selection                                      *
*               = 0 for Mu-law selection                                      *
*               = 1 for A-law selection                                       *
*               = 2 for linear selection                                      *
* RATE          = Rate flow of the coder                                      *
*               = 16 for 16 kbit/s coding                                     *
*               = 24 for 24 kbit/s coding                                     *
*               = 32 for 32 kbit/s coding                                     *
*               = 40 for 40 kbit/s coding                                     *
*                                                                             *
* ADSECOD       = Address of the coder selection table minus 16 (= SECOD-16)  *
* ADSELAW       = Address of the PCM-law selection table (= SELAW)            *
* ADTEMP        = Address of the variable N in RAM                            *
* SECOD         = Coder selection table                                       *
```

```
* SELAW         = PCM-law selection table                                      *
*                                                                             *
*      OUTPUT:                                                                *
* N             = Number of |I| levels depending on RATE value                 *
* SHIFT         = Shift value for UPB routine depending on RATE value          *
* ADITBL        = Address of the |I| table for current coding                  *
* ADQUAN        = Address of the quantizer table for current coding            *
* ADIQUA        = Address of the inverse quantizer table for current coding    *
* ADLAW         = Address of the PCM-law inverse quantizer table               *
* LAWMASK       = magnitude mask for A-law or Mu-law                           *
* LAWBIAS       = Quantization bias for A-law or Mu-law                        *
* LAWSEG        = Segment calculation constant for A-law or Mu-law             *
* AR0           = 3 = step (in words) between delayed floating-point samples   *
* AR3           = Address of the temporary variable TEMP                       *
* AR4           = Address of the temporary variable SE = Address(TEMP) - 3     *
* AR5           = Address of the temporary variable Y = Address(TEMP) - 6      *
*                                                                             *
*      CYCLES: 30                                                             *
*******************************************************************************

SELECT  STM     #0, BK          ; set BK to 0 for non circular addressing
        LDU     ADTEMP, A       ; load address of N
        STLM    A, AR3          ; AR3 points to N
        LDU     ADSECOD, A      ; load coder selection table address
        ADD     RATE, A         ;
        STLM    A, AR4          ; AR2 points to the coder variables
        LDU     ADSELAW, B      ; load coder selection table address
        ADD     LAW, 2, B       ;
        STLM    B, AR5          ; AR5 points to the law variables
        MVDD    *AR4+, *AR3+    ; initialize N
        MVDD    *AR4+, *AR3+    ; initialize RPTQUA
        MVDD    *AR4+, *AR3+    ; inititialize SHIFT
        MVDD    *AR4+, *AR3+    ; initialize ADITBL
        MVDD    *AR4+0%, *AR3+  ; initialize ADIQUA, % is for dual data-memory allowing
        MVDD    *AR4, *AR3+     ; initialize ADQUAN
        MVDD    *AR5+, *AR3+    ; initialize ADLAW
        MVDD    *AR5+, *AR3+    ; initialize LAWMASK
        MVDD    *AR5+, *AR3+    ; initialize LAWBIAS
        MVDD    *AR5, *AR3+     ; initialize LAWSEG
        STM     #3, AR0         ; AR0 = 3 for further circular addressing
        MVMM    AR3, AR5        ; AR5 points to Y
        MAR     *AR3+0%         ;
        RETD                    ;
        MVMM    AR3, AR4        ; AR4 points to SE
        MAR     *AR3+0%         ; AR3 points to TEMP
```

3.13 Channel Selection

Several channels can be simultaneously dealt with; each channel has its own memory space and all run independently. This is made it 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 so used as an address relative to the beginning of a page. We allocate a data-page per channel for memory space, so, when using DMA addressing mode, the value of DP determines which channel we want to access. To enable this, the memory RAM allocated for a channel does not exceed one page (that is 128 words).

A channel also uses data-memory constants in "ROM". 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.

But when we access to the channel RAM 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 we also use the label to designate the same variable, but for another channel (and so for another address) as we explained above. The only way to proceed is to extract DP value in order 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 RAM space of the channel (using a ROM table). Also, this address value is kept in the RAM channel, and is used later to initialize auxiliary registers (for example in the 'SELECT' routine: see 3.12). The following code shows this initialization and reset routine.

```
*******************************************************************************
*                       G726RST: CODER RESET SUB-ROUTINE                      *
*******************************************************************************
*            FUNCTION:                                                        *
* Initialize the encoder/decoder memory space for a channel                   *
* Gives to the delayed internal processing variables their reset value        *
*                                                                             *
*            INPUT:                                                           *
* DP         = Data memory page pointer (9 LSB of Status register 0: ST0)     *
*            = number of channel (must be different from zero)
*                                                                             *
*                             CYCLES: 95                                      *
*******************************************************************************

G726RST RSBX    1,OVM          ; remove overflow mode
       SSBX    1,SXM           ; set sign extension mode

**** compute address of the beginning of the page ****

       LDM     ST0, A          ;
       AND     #01FFh, A       ; A = DP (data page pointer)
       SFTA    A, 7            ; A = address of beginning of current page

**** initialize address variables depending on page number ****

       STL     A, ADDQ6        ; ADDQ6 = address of DQ6 exponent

       ADD     #SR2OF, A, B    ; add SR2 offset to beginning of page
       STL     B, ADSR2        ; ADSR2 = address of SR2 exponent

       ADD     #TEMPOF, A, B   ; add N offset to beginning of page
       STL     B, ADTEMP       ; ADTEMP = address of N

**** Initialize RAM variables using a ROM table ****

       STLM    A, AR2          ; initialize internal processing variables
       LD      #INIRAM, A
       STLM    A, AR3
       NOP
       RPT     #TEMPOF-9       ; repeat 70 times
       MVDD    *AR3+, *AR2+    ; transfers from Data ROM to Data RAM
       RET
```

4. Data Memory Organization

Data memory is shared between RAM and ROM, and includes two sections. “G726RAM” is a non-initialized section for G726 variables and constants used in direct memory access. It takes place in RAM at the beginning of a page. Each channel (for encoding or decoding) requires a RAM space equivalent to this section with a different page number. But only one such section needs to be specified for linking the program. “G726DROM” is an initialized section of constant data and takes place normally in ROM.

4.1 DATA-RAM Space

The following tables display the map of the required RAM space for a G726 channel. For the purposes of RAM space optimization, we make the distinction between static and dynamic variables.

Static variables and constants are initialized each time you reset the coder by running the G726RST sub-routine. Otherwise they keep their value between two successive ‘calls’ of the sub-routines G726COD or G726DEC. It means that the calling program must not write to these locations (the first 72 words of the dedicated page of a channel).

The five global variables are visible from the calling program. They count inputs/outputs for both encoder G726COD and decoder G726DEC and they keep their values while running G726COD or G726DEC. Thus, there is no need to set the input values again if you want to keep these unchanged. In particular, you may set the variables ‘LAW’ and ‘RATE’ at the time you reset a channel with G726RST.

The dynamic variables are initialized each time G726COD or G726DEC is called. It includes the local variable TEMP used for intermediate calculations inside a function. So the 17 last variables can be used as temporary variables for the main program (or other sub-routines) without disturbing the coder RAM space.

Globally, each channel uses 94 variables in a page; thus the 34 left variables are always available.

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’ (for data memory address) means that the direct addressing mode is used. The numbers of the routines (as specified in 0) that use the variable are between brackets.
- ‘Reset value’ gives the assigned value of the variable by the coder reset routine G726RST.

- ‘Description’ gives a short description of the variable.

Table 5: Static Variables: 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* (Q25 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 SR2* 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*
40	YU	DMA (4, 32)	544	Designates YU*

* See description of the variable in Table 30.

Table 6: Constants

Addr	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: Static Variables: Address Variables

Addr	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)

Table 8: Global Variables: G726 Commands, Input & Output Signals

Addr	Name	Access and routine	Format	Description
72	RATE	DMA (0)	possible values: 16, 24, 32, or 40	Coder rate select: 16 for 16 kbit/s coding, 24 for 24 kbit/s coding, 32 for 32 kbit/s coding, 40 for 40 kbit/s coding
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 kbit/s coding 3 bits (LSB) for 24 kbit/s coding 4 bits (LSB) for 32 kbit/s coding 5 bits (LSB) for 40 kbit/s 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: Dynamic Variables: Coder Rate and PCM-Law Selection

Addr	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: Dynamic Variables: Temporary Internal Processing Variables

Addr	Name	Access	Description
87	Y	DMA (4, 8, 11, 28, 31)	designates Y*
88	SEZ	DMA (2, 14) / AR5 (2)	designates SEZ*
89	SE	DMA (2, 6, 13) / AR4 (2)	designates SE*
90	DQ	DMA (12, 18, 25, 34) / AR4 (9)	designates DQ*, or D*
91	PK0	DMA (14, 21, 23) / AR4 (9)	designates PK0*, or lower interval limit of iterative search in (9)

Table 11: Local Variables: Temporary Variables

Addr	Name	Access	Description
92	SIGN	DMA (...) / AR5 (...)	designates DS*, DSX*, or DQS
93	TEMP	AR3 (...)	General use temporary variable for intermediate calculation

4.2 DATA-ROM Space

This data space contains tables of constants that are combined in the section called “G726DROM” (as G726 DROM tables). So this space expects to take place in ROM, using Data-ROM capability of the LEAD when DROM bit is set to one. With this functionality, the 8 or 16 kilo-words ROM memory between E000h or C000h and FF00h addresses are visible from both data and program space. When configuring your memory space, be careful of not to overlap program and data in ROM.

This section “G726DROM” has a 849 words length, and contains six tables. The variables of the involved variables have their format described in Table 30. These tables are:

* See description of the variable in Table 30.

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 “G726RST”. This table corresponds to the RAM variable of Table 5 and Table 6.
2. |I| tables for all rates . Each table contains successively:
   - the lowest level of input quantizer interval QS|I| (first value of column two, Table 12 up to Table 15),
   - the output level of the inverse quantizer DQLN|I| (column three, Table 16 up to Table 19),
   - the rate-of-change weighting function F|I|, (Table 20 up to Table 23)
   - the scale-factor multipliers W|I| (Table 24 up to 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 up to 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 up to 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 up to Table 19). It also gives the sign of this quantized difference (column two, Table 16 up to 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 up to Table 19) which has to be compared with ID (see previous § and 3.2.3)
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 3.2.2)
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 kbit/s coding choice. The relevant variables are those of Table 9.

Table 12: Quantizer Definition for 40 kbit/s 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
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 kbit/s 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 kbit/s 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 kbit/s 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 kbit/s 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
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

Note 1 - The I values are received starting with bit 1.
Note 2 - It is possible for the decoder to receive the code word 00000 because of transmission disturbances (eg, line bit errors)

Table 17: Quantizer Output Levels for 32 kbit/s 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
1010	1	323	2
1001	1	373	1
1000	1	425	0

Note 1 - The I values are received starting with bit 1.
Note 2 - It is possible for the decoder to receive the code word 0000 because of transmission disturbances (e.g., line bit errors)

Table 18: Quantizer Output Levels for 24 kbit/s 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

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

Table 19: Quantizer Output Levels for 16 kbit/s 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 kbit/s 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 kbit/s 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 kbit/s ADPCM

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

Table 23: Map Quantizer Output for 16 kbit/s ADPCM

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

Table 24: Quantizer Scale Factor Multipliers W|I| for 40 kbit/s 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 kbit/s 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 kbit/s ADPCM

|I|	3	2	1	0
W|I|	582	137	30	-4

Table 27: Quantizer Scale Factor Multipliers W|I| for 16 kbit/s ADPCM

|I|	1	0
W|I|	439	-22

5. Program Organization

Numbers of routines refer to the numbers they are attributed in § 0.

5.1 Channel Initialization Routine: "G726RST"

The sequence of the initialization routine is:

- Initialization of status registers (see 8.2)
- Compute absolute address of channel in order to initialize address variables for indirect addressing mode (see 3.13)
- Transfers reset values and constants into channel RAM space

5.2 Encoder Routine: "G726COD"

The sequence of the encoder routine is summarized in the following table:

Table 28: Encoder Sequence (578-605 Cycles)

N°	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 kbit/s 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
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 preparing	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: "G726DEC"

The sequence of the decoder routine is summarized in the following table:

Table 29: Decoder Sequence (606-633 Cycles)

N°	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 kbit/s 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
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 preparing	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 shift left 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 G721 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 the following table. 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 G726 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, we describe these 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[a],A2[a]	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[a], ..., B6[a]	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[a]	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[a]	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[a],..., DQ6[a] exponents	4	Q0 UM	DQFLOAT	Quantized difference signal exponent with delays 0 to 6
DQ0, DQ1[a],..., DQ6[a] mantissas	6	Q6 UM	DQFLOAT	Quantized difference signal mantissa with delays 0 to 6
DQ0, DQ1[a],..., DQ6[a] 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)
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[a], PK2[a]	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[a], SR2[a] exponents	4	Q0 UM	SRFLOAT	Reconstructed signal exponent with delays 0 to 2

SR0, SR1[a)], SR2[a)] mantissas	6	Q6 UM	SRFLOAT	Reconstructed signal mantissa with delays 0 to 2
SR0, SR1[a)], SR2[a)] 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[a)]	19	Q15 UM	YL	Delayed slow quantized scale factor
YLP	19	Q15 UM	YL	Slow quantized scale factor
YU[a)]	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

[a)] 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: Static Variables: Internal Processing Delayed Variables)

5.4.1 FMULT

Function: Multiply predictor coefficients with corresponding quantized difference signal or reconstructed signal: $\mathbf{wb_i(k) = b_i(k-1) * d_q(k-i)}$ **for i = 1, ..., 6; and** $\mathbf{wa_i(k) = a_i(k-1) * s_r(k-i)}$ **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
Notes: The multiplication is performed in floating point format. It implies for the fixed-point processor of the 'C54x that we multiply the mantissas, add the exponents and compute the result sign. First, we divide Ai or Bi by 4 in order 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. We 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: $\mathbf{s_{ez}(k) = wb_1(k) + wb_2(k) + wb_3(k) + wb_4(k) + wb_5(k) + wb_6(k)}$, $\mathbf{s_e(k) = s_{ez}(k) + wa_1(k) + wa_2(k)}$
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

Notes: This routine is partially executed after each call of FMULT (1) in order to avoid using extra variables for WBi and Wai. Also, it allows overflows to occur as specified in G726, so that the accumulation is automatically limited to a 16-bit signed word in Q1 format. Then, 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 $a_p(k-1) \leq 1$, $a_l(k) = 1$, otherwise
Input: AP
Output: B (AL)
Cycles: 4
Notes: 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**:**

$$\mathbf{y(k) = a_l(k).y_u(k-1) + (1-a_l(k)).y_l(k-1)}$$

Inputs: YU, YL (high word), B (AL)
Output: Y
Cycles: 14
Notes: 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: $\mathbf{s(k) \rightarrow s_l(k)}$
Input: A (S or SP in decoder)
Output: A (SL or SLX in decoder)
Cycles: 13
Notes: 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**:** $\mathbf{d(k) = s_l(k) - s_e(k)}$
Inputs: A (SL), SE
Output: A (D)
Cycles: 1
Notes: 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)** $\rightarrow$ **$\{d_l(k) = \log_2(|d(k)|), sign[d(k)]\}$**
Input: A (D or DX in decoder)
Outputs: DS (DSX in decoder), B (DL or DLX in decoder)
Cycles: Min: 6, Max: 11
Notes: for this calculation, we use properties of exponent EXP and mantissa MANT of |d| such as $|d| = 2*MANT*2^{EXP-1}$, where $1 \leq 2*MANT < 2$ and $2^{EXP-1} \leq |d| < 2^{EXP}$. Note that this EXP corresponds to the actual exponent and has not the same definition as the one defined in G726. The word (2*MANT) has a Q7 format with 8 bits, and EXP has a Q0 format with 4 bits. Then, we 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
Notes: 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: **$\{d_{ln}(k), sign[d(k)]\}$** $\rightarrow$ **I(k)**
Input: A (DLN), SIGN (DS)
Output: A (I)
Cycles: Min: 35 (16 kbit/s), 52 (24 kbit/s), 67 (32 kbit/s), Max: 82 (40 kbit/s)
Notes: 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)** $\rightarrow$ **$\{d_{qln}(|I(k)|, sign[d_q(k)]\}$**
Input: A (I)
Outputs: SIGN (DQS), B (DQLN)
Cycles: 10

Notes: 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 gives also 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: $\mathbf{d_{ql}(k) = d_{qln}(k) + y(k)}$
Inputs: B (DQLN), Y
Output: B (DQL)
Cycles: 2
Notes: 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: $\mathbf{d_q(k) = 2^{dql(k)} * sign[d_q(k)]}$
Input: B (DQL), SIGN (DQS)
Output: DQ in two-complement format, A (DQ)
Cycles: Min: 6, Max: 13
Notes: 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'complement. This method does not apply for DQL < 0, in this case DQ is zero. According to G726 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 kbit/s operation), or with 16 bits (for 16, 24, 32, or 40 kbit/s operation). For our 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: $\mathbf{s_r(k) = s_e(k) + d_q(k)}$
Inputs: A (DQ), *AR4 (SE)
Output: SD (SR)
Cycles: 2
Notes: 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 kbit/s coding where DQ < 2^{14}. Nevertheless, in 40 kbit/s coding, SR is limited to be 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
Notes: PK0 gives computing sign (0 for positive, -1 for negative), real sign is given by p(k) (variable $PK0_{low}$ = V1). The real sign worth 1 if p(k) positive, -1 if p(k) 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
Notes: We 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
Notes: We 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
Notes: At the end of this routine, either we perform TRIGA and TRIGB, or we continue the program normally. Note that G726 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>0⇔ tr =1)
Outputs: AP
Cycles: Min: 0 (usual), Max: 2
Notes: 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
Notes: 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 in fact 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)
Notes: $PK0_{low}$ 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: $\mathbf{|a_2(k)| \leq 0.75}$
Input: A (unlimited A2)
Output: A2
Cycles: Min: 0, Max: 6 (usual)
Notes: 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)
Notes: $PK0_{low}$ permits to give the real sign of p(k). Indeed 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: $\mathbf{|a_1(k)| \leq 1 - 2^{-4} - a_2(k)}$
Input: A (A1T), A2 (A2P)
Output: A1 (A1P)
Cycles: Min: 0, Max: 7 (usual)
Notes: 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: $\mathbf{U_i(k) = sign\ (d_q(k))\ **\ sign\ (d_q(k-i))}$
Inputs: DQ1 sign, ..., DQ6 sign, SIGN (DQS), AR7 (points to DQ6 sign)
Output: B (Ui)
Cycles: Min: 0, Max: 12 (usual)
Notes: 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 kbit/s coding; $b_i(k) = (1-2^{-9})b_i(k-1) + 2^{-7}\ U_i(k)$ for 40 kbit/s coding
Inputs: B (Ui), Bi, DQ, SHIFT
Outputs: B1 (B1P), ..., B6 (B6P)
Cycles: Min: 0, Max: 40 (usual)
Notes: If DQ = 0, then Ui is forced to be 0. SHIFT is -8 for 16, 24, 32 kbit/s, and is -9 for 40 kbit/s 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

Notes: 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
Notes: 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)
Notes: 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
Notes: We 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
Notes: in order to prepare FILTE (33), operations are carried out using high part of accumulator.

5.4.32 LIMB

Function: Limit quantizer scale factor: **1.06** ≤ **y(k)** ≤ **10**
Input: BH (YUT)
Output: YU (YUP), AH (YUP)
Cycles: 5
Notes: 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
Notes: -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 G726 recommendation.

5.4.34 FLOATA

Function: Convert 16-bit 2's complement to floating-point. DQ → (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
Notes: exponent EXP of DQ is defined by: $2^{EXP-1} \leq |DQ| < 2^{EXP}$. Mantissa MANT of DQ is obtained with the 6 MSB (Most Significant Bits) of DQ. MANT has the following limits: 0 ≤ MANT < 1, which implies that MANT format is Q6. Sign of DQ is 0 if DQ positive, -1 if DQ negative. If DQ ≠ 0, we find exponent of DQ by means of the instructions EXP and DSUBT instruction, 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 → (SR0 mantissa, SR0 exponent, SR0 sign)
Input: SD (SR), AR6 (points to SR2 sign)
Output: SR0, AR5 (points to SR1 exponent)
Cycles: 14
Notes: See FLOATA for definitions of exponent, mantissa, and sign; but contrary to DQ, SR = 0 always implies that 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
Notes: 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 for 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 thanks 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: $\mathbf{s_r(k) \rightarrow s_p(k)}$
Input: A (SR), LAW, LAWBIAS, LAWSEG, LAWMASK
Output: A (SP)
Cycles: Min: 20, Max: 26
Notes: This generic routine is used for both A and μ-law thanks to the use of the variables LAWxxxx that make the discrimination between laws for this quantization. First, the A-law PCM word is re-converted in 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, we substact one from it before dividing in order to perform a correct truncation. Note that G726 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 outpout PCM sample in decoder for synchronous tandem coding: $\mathbf{s_p(k) \rightarrow s_d(k)}$
Input: B (ID), *AR1 (IM), SD (SP), LAWMASK
Output: A (SD)
Cycles: Min: 6 (usual), Med: 21, Max: 25
Notes: After using the routine EXPAND (), SUBTA (), LOG (), SUBTB (), again in order to perform this synchronous adjustment, the routine QUAN () is also used to re-encode the PCM output word. The new coded word is thus 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. System Cycle Times and Memory Requirement

6.1 Memory

Table 31: Memory Requirement (16-bit Words)

Program ROM	Data ROM	Data RAM
0x2aa (682)	0x351 (849)	0x5e (94)

6.2 Cycles

Table 32: Clock Cycles Requirement for Encoder

	Linear			A-law / μ-law		
Rate	Min.	Nom.	Max.	Min.	Nom.	Max.
16 kbit/s	451	567	584	467	583	600
24 kbit/s	460	576	593	476	592	609
32 kbit/s	469	585	602	485	601	618
40 kbit/s	478	594	611	494	610	627

Table 33: Clock Cycles Requirement for Decoder

	Linear			A-law / μ-law		
Rate	Min.	Nom.	Max.	Min.	Nom.	Max.
16 kbit/s	413	524	541	486	608	644
24 kbit/s	413	524	541	495	617	653
32 kbit/s	413	524	541	504	626	662
40 kbit/s	413	524	541	513	635	671

Table 34: Clock Cycles Requirement for Encoder + Decoder

	Linear			A-law / μ-law		
Rate	Min.	Nom.	Max.	Min.	Nom.	Max.
16 kbit/s	864	1091	1125	953	1191	1244
24 kbit/s	873	1100	1134	971	1209	1262
32 kbit/s	882	1109	1143	989	1227	1280
40 kbit/s	891	1118	1152	1007	1245	1298

7. Conformance of the Present Algorithm with CCITT G726 ADPCM

7.1 Practical Audio Test on ‘C54x Evaluation Module (EVM)

The TMS320C54x evaluation module (EVM) is a PC/AT plug-in card that lets you evaluate certain characteristics of the ‘C54x digital signal processor (DSP) to see if the DSP meets your application requirements. To simplify code development and shorten debugging time, a graphical, window-oriented debugger is also included.

For our application, we use the analog interface connected to speaker and microphone via the RCA connectors. The signal is linearly 14-bits quantized in the digital converter (TLC320AC01) included in the EVM, so we use the linear capability of the software for this application.

Because output signal at the encoder does not represent the amplitude of the speech signal, we cannot directly test the ADPCM signal so we have to code and then decode the output signal. One way to test the ADPCM is to observe the quality degradation of the signal through the encoder and decoder: the difference should be very slight. To do so, we must have a signal reference that is only digitized then converted again to analog. A second program will ensure this function of direct loopback.

To use EVM, the main program first has to initialize the analog interface circuit included on the EVM card. That configures the A/D converter. Then, we can receive and transmit the digital signal via the first serial port of the DSP.

7.1.1 Initialization of the Analog Interface Circuit (TLC320AC01)

The configuration of the analog interface circuit (AIC) is determined by the values of 8 registers. First, resetting the AIC gives to these registers default configuration values and makes it possible to begin communications via the DSP serial port 1. There are two types of communications: primary serial communications and secondary serial communications.

Primary communications are normal communications where the 16-bits data transmitted comports on MSB the 14-bits digital sample, and on LSB 2 control bits. If these two bits are set to 11, a secondary serial communication is demanded, otherwise, when set to 00, primary communication continues.

Secondary serial communication is used for configuring AIC through the 8 registers, including registers A and B. The essential functions are: choice of the sampling frequency and of the low-pass filter bandwidth.

The sampling (conversion) frequency is derived from the master clock (MCLK) input by the following equation:

$$f_s = \text{Sampling frequency} = \frac{\text{MCLK}}{(\text{A register value}) * (\text{B register value}) * 2}$$

The filter clock (FCLK) is an internal clock signal that is used to determine the filter band-pass frequency and is the B counter clock. The frequency of the filter clock is derived by the following equation:

$$FCLK = \frac{MCLK}{(\text{A register value}) * 2}$$

The low-pass filter -3 dB corner is derived by:

$$f(LP) = \frac{FCLK}{40} = \frac{MCLK}{40 * (\text{A register value}) * 2}$$

The master clock value is 10368 kHz, with A = 36 and B = 18, we obtain:

f_s = 8 kHz and f(LP) = 3600 Hz, those are the values required.

For the values of the other registers, see the listing below.

Before transmitting these values using secondary serial communications, you must reset the AIC by applying a '0' on the pin AICRST during at least a master clock period. A means of doing that is to write 7FFFh on port at address 14h during at least 100 ns (i.e. repeat the port write instruction at least 4 times).

So, the AIC initialization procedure is:

- disable all interrupts during all the AIC initialization,
- configure the serial port 1 (frame synchronization pulse required, 16 bits mode, external clock),
- reset the AIC,
- transmit values to the AIC configuration registers via secondary serial communications.

7.1.2 The ADPCM Test Routine

The principle is to code the data received on serial port, then to decode the ADPCM data obtained, and to loopback the result on the output of the serial port. First, you have to initialize encoder and encoder with the routine G726RST, then the infinite process of coding-decoding can begin.

Only one interruption is allowed: the receive interrupt of the serial port 1. Indeed, the time of ADPCM processing (≈ 25 μs) is much shorter than the sampling period (125 μs), and the AIC has enough time to read the word transmitted by the DSP. So coding and decoding are real-time processed.

Because only the 14 MSB of a word are used by the AIC for the conversion, the data received by the DSP must be right shifted 2 times and the data transmitted must be left shifted 2 times. Once shifted, this format exactly corresponds with that required for linear ADPCM performing (14 bits 2's complement).

The audio results are very satisfying, with very little difference with direct AIC loopback, for 24, 32, and 40 kbit/s coding. 16 kbit/s coding-decoding makes a little background noise, but does not actually affect voice quality. The software compliance seems not to be involved, because CCITT test sequences have been successfully verified. Rather, it is the fact that each sample has been coded on only two bits before being decoded. We now show the main file (main726e.asm) for this audio test on EVM, with its corresponding EVM initialization file (evminit.cmd).

```
*************************************************************************
*               C54x G726 coder/decoder test program on EVM             *
*************************************************************************

        .mmregs
        .version 541
        .global         G726RST,G726COD,G726DEC ; G726 Subroutines
        .global         RATE, LAW,S,I,SD                ; G726 Variables
        .global         entry,reset,IDLE,CODEC  ;

DP0     .SET    0 ; 0000h -> 0080h
DPMAIN  .SET    3 ; 0180h -> 0200h
DPCOD   .SET    4 ; 0200h -> 0280h
DPDEC   .SET    6 ; 0300h -> 0400h
flow    .SET    32

pile    .usect  "Stack",64

        .sect   "Reset"

* Reset and interrupt vectors

entry   ORM      0020h,PMST ; enable on chip DARAM program
        B       reset       ; reset vector
        .space 21*4*16

RINT    B       CODEC       ; serial port 1 receive interrupt
        B       IDLE

        .text

* Initialize TMS320C54x

reset   SSBX    INTM        ; disable all interrupts
        STM     #200h,SP
        LD      #0,A
        STLM    A,SWWSR     ; 0 waits in pgm and data memory
        STM     #0008h,SPC1 ; frame synchronization pulse required,16
                            ; bits mode with
        STM     #00C8h,SPC1 ; external CLKX
        LD      #0,A
        STLM    A,DXR1; clear first int (RRDY = 0)
        STLM    A,DRR1; clear first int (RRDY = 0)

* Reset AIC (TLC320AC01) by writing 0 on EVM AICRST at address 14h
```

```
        LD      #DPMAIN,DP  ;
        PORTW   RST,14h     ; Pulse AIC reset by setting it low
        RPT     #1000       ; during at least 1 MCLK=10.368MHz
        NOP                 ; (>100 ns)
        PORTW   CFFFF,14h   ; end of AIC reset and allow target control
        PORTR   14h,CFFFF

* Initialize AIC for a 8 kHz sample rate with a gain setting of 1

        LD      #DP0,DP
        STM     #1C0h,AR6   ; point to beginning of buffer
        LD      #AICCFG,A   ; load control word start address
        RPT     #7          ; move 8 configuration words in buffer
        READA   *AR6+       ; by reading in TLC320AC01 Initialization
                            ; table
        STM     #1C8h,AR0   ; load end of buffer
        STM     #1C0h,AR6   ; and beginning of buffer (AR6 = buffer
                            ; pointer)
AICWAIT BITF    SPC1,0800h  ;
        BC      AICWAIT,NTC ; wait AIC until XRDY goes high
        LD      *AR6+,A     ; read next control word
        STLM    A,DXR1 ; and send to AIC
        CMPR    00,AR6 ; check for end of buffer
        BC      AICWAIT,NTC ; if pointer != end (AR0), wait AIC

* Initialize coder and decoder

        LD      #DPCOD,DP   ; load channel for encoder
        CALL    G726RST     ; initialize encoder
        LD      #DPDEC,DP   ; load channel for decoder
        CALL    G726RST     ; initialize decoder

* Enable receive interrupt

        STM     #0040h,IMR  ; turn on receive and transmit interrupts
        RSBX    INTM        ; enable all unmasked interrupts

* Wait for interrupts (replace normal program)

IDLE    B       IDLE        ; wait for an interrupt

CODEC   LD      #DP0, DP
        LD      DRR1,-2, A  ; get PCM sample from serial port
        LD      #DPCOD,DP   ; load channel for encoder
        STL     A,S         ; S = input PCM word
        ST      #2, LAW     ; LAW = 2 for linear
        ST      #flow, RATE ; choose 32 kbit/s coding
        CALL    G726COD     ; encode this sample
        LD      I,A         ; I = ADPCM output word
        LD      #DPDEC,DP   ; load channel for decoder
        ST      #2, LAW     ; LAW = 2 for linear
        ST      #flow, RATE ; choose 32 kbit/s decoding
        STL     A,I         ; I = ADPCM input word
        CALL    G726DEC     ; decode this sample
        LD      SD, A       ; SD = PCM output word
        SFTA    A, 2        ; shift with setting 2 control bit to zero
```

```
        STLM    A,DXR1 ; and send 14-bit PCM sample into serial port
        RETE           ; return from interrupt

*************************************************************************
*               TLC320AC01 initialization table
*************************************************************************

                                ; register 0 (no op) is unchanged

AICCFG  .word   0000000000000011b ; initialize register 1 (register A)

;               CCWAAAAADDDDDDDD  ; C=Controls bits; W=Read/Write- ;
                                  ; A=Address; D=Data
        .word   0000000100100100b ; TA = 36; Controls bits = Phase;
                                  ; shift = 00
        .word   0000000000000011b ; initialize register 2 (register B)

;           CCWAAAAADDDDDDDD      ; C=controls bits; W=Read/Write-;
                                  ; A=address; D=data
        .word   0000001000010010b ; TB = 18 for fe = 8 kHz with
                                  ; MCLK=10.368MHz

        .word   0000000000000011b ; initialize register 4 (Amplifier gain
                                  ; select)

;               CCWAAAAAXXMMIIOO  ; C, W, A: see upper; X=Not used;
                                  ; (sq=squelch)
        .word   0000010000011001b ; M=Monitor output gain (0=sq,1=0dB, 2=-
                                ; 8dB, 3=-18dB)
                                ; I=Analog input gain (0=sq,1=0dB, 2=+6dB,
                                ; 3=+12dB)
                                ; O=Analog output gain (0=sq,1=0dB, 2=-
                                ; 6dB, 3=-12dB)

        .word   0000000000000011b ; initialize register 5 (Analog
                                ; Configuration)

;                CCWAAAAAXXXXEHLL ; C, W, A, X: see upper;
CFGEND  .word   0000010100000101b ; E=Echo (must be set to 0: Echo
                                  ; off)
                                  ; H=High-pass filter (0=disabled,
                                  ; 1=enabled)
                                  ; LL:00=analog loopback, 01=normal
                                  ; input (IN+&IN-)
                                  ; 10=auxiliary input (AUXIN&AUXIN-),
                                ; 11=both
                                ; register 6 (digital configuration)
                                ; is unchanged
                                ; register 7 (frame sync delay) is
                                ; unchanged
                                ; register 8 (frame sync number) is
                                ; unchanged

*************************************************************************
*                         Main Program data
*************************************************************************
```

```
        .bss PCMIN, 1       ; PCM input value
        .bss ADPCMI, 1      ; ADPCM input value
        .bss ADPCMO, 1      ; ADPCM output value
        .bss PCMOUT, 1      ; PCM output value

        .data
RST     .word   7FFFh       ; low reset for TLC320AC01
CFFFF   .word   0FFFFh      ; high reset for TLC320AC01
```

file evminit.cmd:

```
;TMS320C541 MEMORY MAP
MR
;
MA 0x0000, 1, 0x002A, RAM  ; MMRs
MA 0x0030, 1, 0x0003, RAM  ;
MA 0x0060, 1, 0x0020, RAM  ; SCRATCH PAD
MA 0x0080, 1, 0x1380, RAM  ; INTERNAL DATA RAM
MA 0x0400, 0, 0x1000, RAM  ; INTERNAL PROGRAM RAM
MA 0x9000, 0, 0x7000, ROM  ; INTERNAL ROM

ma 0x1400, 0, 0xec00, ram  ; external ram
ma 0x1400, 1, 0xec00, ram  ; external ram

;ma 0x0000, 2, 0x15, ioport ; i/o space

map on
;
;Define reset alias to set PMST for MC mode
;
alias myreset, "e pmst = 0xff80; reset "
e pmst = 0xffe0  ; MP mode, OVLY, DROM off CLKOUT on
e hbpenbl = 0x0000

e *0x28 = 0x2000 ; two wait states on i/o, none for memory
e *0x29 = 0x0000 ; no bank switching necessary

; dasm pc
echo Loaded TMS320C54x evminit.cmd
load c:\edo\opti\g726c54e.out
mem 0x247
mem2 0x347
```

7.2 CCITT Digital Test Sequences on ‘C54x Simulator

Digital sequences are used to verify the conformance of an implementation to a digital transcoding algorithm. The sequences are chosen to exercise the major arithmetic components and thus give a reasonable level of confidence in the compliance of an implementation with this recommendation. Note that with a limited number of test sequences it is not possible to demonstrate 100% coverage of all states of the implementation.

The verification testing procedure consists of applying an input sequence to an ADPCM implementation, and verifying that the output sequence is the same sequence in the output file for the same test condition (Flow rate coding, PCM coding law, type of input, initial state of the implementation).

There are three types of input sequences. The first type consists of various sinusoidal PCM inputs that are representative of the signal expected in normal operation. These are called “normal inputs”. A second group of input sequences is the test of “overload inputs” that contain PCM signals of very large amplitudes. The third group is the set of ADPCM sequences that can exercise the algorithm in a manner that is not possible with any PCM input sequence. They test the arithmetic and algorithmic performance of the ADPCM decoder by driving it to states that are unreachable by PCM signals under normal conditions. For example, these states may result from errors on the transmission line. These intermediate sequences will be denoted as “i-inputs”.

The initial state of the implementation may be either the reset state of the algorithm defined in Table 30, or a well-defined initial state that follows the application of an initialization (or homing) sequence. Accordingly, there are two types of test sequences, reset sequences and homing sequences.

For each initial state, both PCM laws (A-law and μ-law) are considered. To allow for the interpolation of both PCM laws, four possible combinations must be considered (A-law → A-law, μ-law → μ-law, A-law → μ-law, and μ-law → A-law).

Last each coder rate (16, 24, 32, 40 kbit/s) is tested for all these combinations.

In all, 80 tests must be performed: for each of the four possible rates, there are two laws, three sequence types (normal, overload, and i-inputs), and two different initial states (reset and homing), (4*2*3*2 = 48), plus two transcoding laws with two sequences types (normal and overload), and two different initial states (reset and homing), (4*2*2*2 = 32).

Note that for normal and overload sequences, each test demands two output sequences to verify: intermediate output ADPCM sequence, and output PCM sequence.

But this is not actually true for law transcoding (A-law → μ-law, or μ-law → A-law), because intermediate sequences are already tested in the other tests (A-law → A-law, or μ-law → μ-law).

Thus, in all, 112 different output sequences must be verified. To perform these tests, the sequences can be supplied as MS-DOS text files that contain at each line the hexadecimal value corresponding to the sample. There are 18 input files and thus 112 output files as references for verification.

A typical main file (main726s.asm) for one of this test (A-law, 32 kbit/s, normal input, homing sequence) is shown below with the corresponding initialization file of the simulator (siminit.cmd):

```
*************************************************************************
*
*               C54X G726 Encoder/Decoder test program on Simulator
*
*************************************************************************

        .mmregs  ; defines names for memory-mapped registers
        .global  G726RST,G726COD,G726DEC    ; G726 subroutines
        .global  LAW, RATE, I,S, SD         ; G726 global variables
        .global  entry                      ; start of program

DPMAIN  .SET    5
DPCOD   .SET    4
DPDEC   .SET    6
law     .SET    1
flow    .SET    32

pile    .usect  "Stack", 64      ; reserve space for stack

        .sect   "Reset"
*************************************************************************
*                        Reset vector
*************************************************************************

entry   ORM     #0088h,PMST      ; enable on-chip program rom (MC = 0)
        B       reset            ; and data ROM (DROM = 1)

*************************************************************************
*                        Main program
*
* This program successively codes and decodes samples extracted from test
* vectors available on input test files supplied by the CCITT. The ADPCM
* output samples and PCM output samples are stored into output files. Then
* these files are compared with theoretical results.
*************************************************************************

        .text

reset   STM     #0200h,SP   ; top of stack at 200h address
        LD      #DPCOD,DP   ; choose a channel for encoder
        ST      #flow, RATE ; choose rate coding
        ST      #law, LAW   ; choose law PCM
        CALL    G726RST     ; initialize decoder
        LD      #DPDEC,DP   ; choose a channel for decoding
        ST      #flow, RATE ; choose rate coding
        ST      #law, LAW   ; choose law PCM
        CALL    G726RST     ; initialize decoder
        LD      #DPMAIN, DP ;
        ST      #3496, CNTR ; Loop 3496 times for homing sequence
debut   LD      #DPMAIN,DP  ; load page for main program
        PORTR   051h,PCMIN  ; read PCM sample from homing input file
        LD      PCMIN,A     ; load input PCM
        LD      #DPCOD, DP  ; load channel for encoder
        STL     A, S        ; store input PCM in S
        CALL    G726COD     ; code this sample
        LD      I, A        ;  I = intermediate ADPCM sample
```

```
        LD      #DPDEC,DP   ; load channel for decoding
        STL     A,I         ; I = input ADPCM sample
        CALL    G726DEC     ; decode this sample
        LD      #DPMAIN, DP ;
        LD      CNTR, A     ;
        SUB     #1, A       ; decrement counter
        STL     A, CNTR     ;
        BC      debut, AGT  ; loop while counter > 0
        LD      #DPMAIN, DP ; load page for main variable access
        ST      #16384, CNTR ; Loop 16384 times: number of test samples
debut2  LD      #DPMAIN,DP  ; load page for main program
        PORTR   050h,PCMIN  ; read PCM sample from input file
        LD      PCMIN,A     ; load PCM input
        LD      #DPCOD, DP  ; load channel for encoder
        STL     A, S        ; store PCM input into S
        CALL    G726COD     ; code this sample into I
        LD      I, A        ; load I = intermediate output ADPCM
        LD      #DPMAIN, DP ; load page for main variable access
        STL     A, ADPCMO   ; store ADPCM sample
        PORTW   ADPMO, 055h ; and write it into intermediate ADPCM file:
                            ; 54adpcm.i
        LD      #DPDEC,DP   ; load channel for decoding
        STL     A,I         ; I = input ADPCM sample
        CALL    G726DEC     ; decode this sample
        LD      SD,A        ; SD = output PCM code
        LD      #DPMAIN,DP  ; load page for main variables access
        STL     A,PCMOUT    ; store PCM sample
        PORTW   PCMOUT,060h ; and write it into PCM output file: 54pcm.o
        LD  CNTR, A         ; load counter variable
        SUB #1, A           ; decrement counter
        STL A, CNTR         ; and store it
        BC  debut2, AGT     ; loop if counter superior to zero
fin     B       debut2      ; put a breakpoint at this line

        .bss PCMIN, 1       ; PCM input value
        .bss ADPCMI, 1      ; ADPCM input value
        .bss ADPCMO, 1      ; ADPCM output value
        .bss PCMOUT, 1      ; PCM output value
        .bss CNTR, 1        ; Loop repeat counter
```

file siminit.cmd:

```
;       LEAD Simulator startup command file for G726 ADPCM
;       ************************************************

mr
e pmst = 0xff88

ma 0xFF80,0, 0x0080, R|W
ma 0xFF80,0, 0x0080, R
ma 0x9000,0, 0x300, R|W
ma 0x9000,0, 0x300, R


ma 0x0000,1, 0x0060, R|W
```

```
ma 0x0060,1, 0x0020, R|W
ma 0x0080,1, 0x1380, R|W
ma 0xE000,1, 0x1000, R|W
ma 0xE000,1, 0x1000, R

ma 0x50,2, 0x1, R|P             ; config input port
mc 0x50,2,0x1, C:\edo\test\allvect\i40,R  ; connect file to input port

;ma 0x51,2, 0x1, R|P             ; config input port
;mc 0x51,2,0x1, C:\edo\test\allvect\i_ini_40.m,R  ; connect file to
                                                  ; input port

ma 0x55,2, 0x1, W|P             ; config input port
mc 0x55,2,0x1, C:\edo\opti\54adpcm.i,W   ; connect file to output port

ma 0x60,2,0x1, W|P              ; config output port
mc 0x60,2,0x1,C:\edo\opti\54pcm.o,W      ; connect file to output port

dasm pc
load c:\edo\opti\g726c54s.out
mem 0x0247
mem2 0x0347
move 0,29
ba debut
ba fin
wa *0x284,CNTR,d
move 0,25
```

In this example the intermediate ADPCM output file **54adpcm.i** had to be compared with **hn32fa.i**, and the output PCM file **54pcm.o** had to be compared with **hn32fa.o**. A test like this takes six to eight hours running on a 80486 computer.

Debugging this software shows particular points to respect if you want to be in conformance with G726 specification. The major point concerns the overflows, and appears especially in 40 kbit/s coding when the $d_q(k)$ variable is assigned to be a 16-bit word instead of 15-bit word. In most of the cases, the arithmetic used in G726 is 16-bit format, even for the ALU, which means that overflows occur as soon as an accumulation is 17-bit word or more (including sign bit). That is obviously different in the ‘C54x, where accumulator has a 40-bit format. The main consequence of this, is that when we add two positive 16-bit numbers (including sign bit), the overflow occurs and the result becomes negative, while for the ‘C54x, the word remains positive because of the sign-extension.

The concerned variables by this feature are: $wa_i(k)$, $wb_i(k)$, $s_e(k)$, $s_{ez}(k)$, and in 40 kbit/s coding: $s_r(k)$, and p(k) (see Table 30: Internal Processing Variables). The result could sometimes appear non-significant, but the overflow is not a bad solution for these variables, which should be as small as possible when the prediction is correct. It seems to be an implicit means of resetting when the algorithms tend to diverge.

Another point is due to the asymmetry between positive and negative numbers. The most negative value is -32768, and after taking from it the absolute value, it makes a word whose exponent is 16 (16-bit unsigned value). On the other hand, exponents of

positive numbers are limited to 15 (15-bit unsigned value or 16-bit signed value). This feature occurs when converting numbers to floating-point (see § 3.3.2). G726 specification truncates the words to 15 unsigned bits, so that |-32768| becomes 0. The reason is probably the same as explained upper for overflow.

The version of the ADPCM software for TMS320C54x, that we supply with this report has been implemented, then “debugged” to be in conformance with the G726 recommendation. All the test sequences (112) have been successfully verified. The resulting code makes it possible to deal with more types of signals than when we just code speech. Indeed, the audio test of § 7.1 is much easier to verify, and is less fine that digital test, all the more so since it skips the PCM interface. Lastly, this conformance with G726 recommendation makes the ADPCM code produced by this software compatible with other G726 systems, on other architectures for example.

8. Software Configuration and Implementation

8.1 Adapt and Optimize Encoder & Decoder Program for Specific Application

The supplied version of ADPCM is mainly intended for general use and test. It includes the maximum number of possibilities, thereby causing time penalties. Perhaps certain features will not be used in a specific application: linear PCM capability, Rate switching between samples without resetting the encoder or decoder. As these functions are just sub-programs, and they are called (with CALL instruction) in the first layer of encoder or decoder, it is therefore easy to move or suppress them. The first layer of the encoder and decoder programs, which can be modified, is shown here:

```
*************************************************************************
*
*              G726COD: ENCODER SUB-ROUTINE
*
*************************************************************************
*
*                FUNCTION:
*
* Encode a PCM sample into an ADPCM word
*
*                INPUT:
*
* S             = S(k): input A-law or Mu-law PCM sample
* LAW           = LAW: PCM law selection
*               = 0 for Mu-law selection
*               = 1 for A-law selection
*               = 2 for linear selection
* RATE          = Flow rate of coding
*               = 16 for 16 kbit/s coding
*               = 24 for 24 kbit/s coding
*               = 32 for 32 kbit/s coding
*               = 40 for 40 kbit/s coding
* DP            = Data memory page pointer (9 LSB of Status register 0: ST0)
*               = number of channel (must be different from zero)
*
*                OUTPUT:
*
* I             = I(k): output ADPCM word
*               = 2-bit word for 16 kbit/s coding
*               = 3-bit word for 24 kbit/s coding
*               = 4-bit word for 32 kbit/s coding
*               = 5-bit word for 40 kbit/s coding
*
*                CYCLES: (usual values)
*
* 567 (uniform PCM) 583 (log-PCM) at 16 kbit/s
* 576 (uniform PCM) 592 (log-PCM) at 24 kbit/s
* 585 (uniform PCM) 601 (log-PCM) at 32 kbit/s
* 594 (uniform PCM) 610 (log PCM) at 40 kbit/s
*
*                                SUB-ROUTINES:
```

```
*
* SELECT  PCM-Law and flow rate selection
* SECOMP  Estimation of signal
* AQUAN   Adaptive quantizer of error prediction
* IAQUAN  Inverse adaptive quantizer of error; adaptation of predictor
*
*********************************************************************

** Select encoder and law **** 34 cycles

G726COD CALLD SELECT
        LD      #1, A
        STLM    A, AR0      ; AR0 = 1 for coding from PCM to ADPCM

** Compute signal estimate SE and quantizer scale factor Y **** 248 cycles

        CALL    SECOMP;

** Load input PCM sample S **** 2 cycles

        LD      LAWMASK, B  ; if L = 2 (linear PCM), LAWMASK = 0
        LD      S, A        ; load input PCM sample

** and convert it to linear **** 20 cycles

        NOP             ; XC latency
        XC       2, BNEQ         ; test PCM law, if linear, bypass EXPAND
        CALL     EXPAND     ; convert to linear if A-law or Mu-law

** Call adaptive quantizer **** 44-53-62-71 cycles max (16/24/32/40 kbit/s)

        CALL    AQUAN ;

** Store output ADPCM sample I **** 1 cycle

        STL     A, I        ;
** quantizer and predictor adaptation **** Cycles: Min: Med: 234, Max:

        CALL    IAQUAN;
        RET             ;









*********************************************************************
*
*               G726DEC: DECODER SUB-ROUTINE
*
*********************************************************************
```

```
*
*                FUNCTION:
*
* Decode an ADPCM word into a PCM sample
*
*                INPUT:
*
* I             = I(k): ADPCM word
*               = 2-bit word for 16 kbit/s coding
*               = 3-bit word for 24 kbit/s coding
*               = 4-bit word for 32 kbit/s coding
*               = 5-bit word for 40 kbit/s coding
* LAW           = LAW: PCM law selection
*               = 0 for Mu-law selection
*               = 1 for A-law selection
*               = 2 for linear selection
* RATE          = Flow rate of decoding
*               = 16 for 16 kbit/s coding
*               = 24 for 24 kbit/s coding
*               = 32 for 32 kbit/s coding
*               = 40 for 40 kbit/s coding
* DP            = Data memory page pointer (9 LSB of Status register 0: ST0)
*               = number of channel (must be different from zero)
*
*                OUTPUT:
*
* SD            = SD(k): Output A-law or Mu-law PCM word
*
*                CYCLES: (usual values)
*
* 524 (uniform PCM)
* 608 (log-PCM at 16 kbit/s)
* 617 (log-PCM at 24 kbit/s)
* 626 (log PCM at 40 kbit/s)
* 635 (log PCM at 40 kbit/s)
*
*
*                SUB-ROUTINES:
*
* SELECT        PCM-Law and flow rate selection
* SECOMP        Estimation of signal
* IAQUAN        Inverse adaptive quantizer of error; adaptation of predictor
* SYNC          Synchronous coding adjustment
*
*************************************************************************

** Select decoder and law **** 34 cycles

G726DEC CALLD   SELECT      ;
        LD      #2, A       ;
        STLM    A, AR0      ; AR0 = 2 for decoding

** Compute signal estimate SE and quantizer scale factor Y **** 248 cycles

        CALL    SECOMP;
```

```
** Load input ADPCM sample I **** 1 cycle

       LD      I, A        ;

** inverse quantizer, then adapt quantizer and predictor **** 229 cycles

       CALL    IAQUAN;

** load reconstructed signal ****  2 cycles

       LD      LAWMASK, B  ; if L = 2 (linear PCM), LAWMASK = 0
       LD      SD, A       ; load reconstructed signal (SD = SR)

** convert it to log PCM with synchronous adjustment **** 88-115 cycles

       NOP             ; XC latency
       XC       2, BNEQ          ; test PCM law, if linear, bypass SYNC
       CALL    SYNC        ;

** Store output PCM sample and return to main program **** 6 cycles

       STL     A, SD       ;
       RET             ;
```

Linear capability can be suppressed with the following procedure: suppress the conditional executing *XC* (for both encoder and decoder), and call together EXPAND and AQUAN functions with only one *CALL* and one *RET* (for encoder and decoder in routine SYNC too). In this last case, it is possible to optimize the EXPAND function, by running initialization instructions instead of NOP instructions (see § 3.2.2.Linear PCM Expanding).

Rate/PCM law switching function can be moved into the reset routine G726RST just by moving the *CALL SELECT*, so that selection of the PCM law and the rate flow is only done when resetting the coder. In this way, the time gain is 0.5 MIPS. Otherwise, if you want to keep this capability, you could call together the SELECT and SECOMP functions with only one CALL and one RET.

It is also possible to separate the SELECT routine in two parts: one for rate selecting, and the other one for PCM law selecting. One part (rate selecting) would take place in the reset routine, and the law selection would stay in the encoder or decoder routine.

As there is one initialization table for each selection type, this separation could be easily done.

8.2 Integrate G726 ADPCM on Your Device

Implementing this version of G726 ADPCM onto C54x mapping would not bring about problems because the code never refers to an absolute address. Nevertheless, there is a rule to respect for the RAM. For each channel, the RAM space is located at the beginning of a page. When linking the program, you only need to implement the

'G726RAM' section (that refers to one channel) at the beginning of a page. But, each time you implement another channel, by running the G726RST routine (with correct DP value), the beginning of the new page would be reserved for this channel, even if these locations were not reserved or were reserved for another use at the link moment (see § 4.1.DATA-RAM Space). So be careful that you do not overwrite data from other modules when you implement a new channel.

This software will give its best results using the 'C541 in microcomputer mode. With this configuration, you use on-chip Data-ROM for the 'G726DROM' section by setting DROM bit to one. This avoids external accesses. For maximum efficiency, the program code and the Data-ROM section must not reside in the same ROM block, the Data-ROM space being divided in two 4 K-word blocks. The figure bellow shows an example of memory implementation.

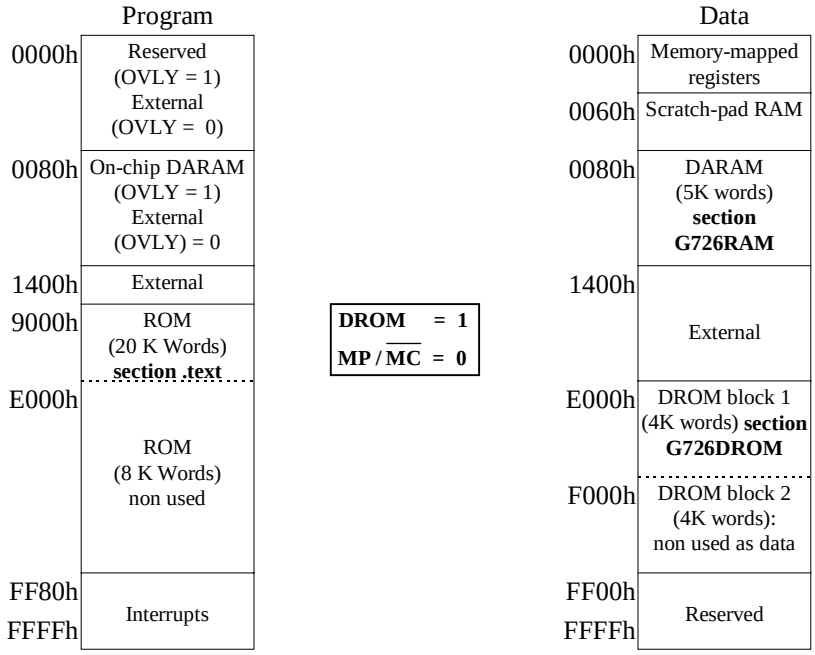

Figure 4: C541 Memory Map. Example of Configuration

As for the status and control registers (ST0, ST1, PMST) configuration, this is done by the reset routine. The bits concerned are C16, SXM, and CPL for which the values directly count on program working. The following values are assigned to these bits:

- C16 = 0 for double precision mode (reset value),
- SXM = 1 for sign extension mode (reset value),
- CPL = 0 for relative direct-addressing using data page pointer (reset value).

Note that G726 specification does not consider that ALU integrates the sign extension. But, in practice, this characteristic makes the calculations lighter. Double precision is used (see § 3.9), and DP is constantly used as the address segment for the channel

RAM (see § 3.13). That is why these bits must be properly set before running G726COD or G726DEC, if you have modified ST1 since channel initialization. Since these values are usual, they are currently not saved in the encoder or decoder routine.

The values of the other registers have no importance, otherwise the registers are initialized in the encoder or decoder routine. That is especially the case for the auxiliary registers (AR0-AR7) that are initialized and saved in G726COD and G726DEC.

8.3 Use of the G726 ADPCM Software

For the user, this ADPCM software consists of 3 sub-programs (called with *CALL* instruction), 5 variables (S, I, SD, LAW, RATE), and the value of DP assigned to a channel. The 3 routines are:

- reset sub-program: G726RST to initialize the channel and reset it.
- encoder sub-program: G726COD to compute the output ADPCM sample I from the input PCM sample S.
- decoder sub-program: G726DEC to compute the output PCM sample SD from the input ADPCM sample I.

LAW is for the PCM law selection: 0 for μ-law, 1 for A-law, and 2 for linear PCM. RATE is for the rate flow choice: 16, 24, 32, or 40 for respectively 16, 24, 32, or 40 kbit/s coding. Only these values give a coherent result: 20 value for RATE will not code to 20 kbit/s! The DP value determines, for each of these variables, which channel is concerned when calling one of the three sub-programs. You cannot choose DP = 0 as a channel, because it corresponds to the memory-mapped registers.

RATE and LAW are currently estimated at each sample, but could be set just one time if wanted, when initializing the channel for example (see § 4.1). Examples of use are shown in § 7.1

8.4 Software Package

Files Description:

G726C54X.ASM	'C54x G.726 asm source
G726C54X.DOC	Application note documentation
EVMINIT.CMD	Initialization file for 'C54x EVM
G726C54E.CMD	Linker command file for 'C54x EVM version
MAIN726E.ASM	Main file for EVM version
MAKE726E.ASM	Batch file for EVM version
G726C54S.CMD	Linker command file for simulator version
MAIN726S.ASM	Main file for simulator version
MAKE726S.BAT	Batch file for simulator version

SIMINIT.CMD	Initialization file for ’C54x simulator.

References

1. 40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM) Recommendation G726. General aspects of digital transmission systems; terminal equipments. ITU (International telecommunication union). CCITT (the international telegraph and telephone consultative committee). Geneva 1990
2. Digital Coding of Speech Waveforms: PCM, DPCM, and DM Quantizers, by Nuggehally S. JAYANT. Proceedings of the IEE, vol 62, No. 5, May 1974, pp 612-633
3. A simple approach to digital signal processing, by Craig MARVEN & Gillian EWERS. TI MENTORS. Texas Instrument. 1993
4. TMS320C54x User's guide. Digital Signal Processing Products. 1995, Texas Instruments
5. Digital Coding of Waveforms, Jayant, N.S. and Noll, P. - Prenctice 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, by Habibi, A.. IEEE Transactions on communications, December 1971, pp 948-956.
8. Digital Signal Processing with the TMS320 Family, Volume 1, Lin, Kun-Shan. Prenctice Hall, Englewood Cliffs, NJ. 1987
9. Texas Instruments World Wide Web internet server, http://www.ti.com/