A-Law and mu-Law Companding Implementations Using the TMS320C54x

Application Note: SPRA163A

Charles W. Brokish, MTS
Michele Lewis, MTSA
SC Group Technical Marketing

Digital Signal Processing Solutions
December 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 application 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.

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

CONTACT INFORMATION

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

Contents

Abstract **7**
Introduction **8**
Human Acoustics and the Telephone Network 8
Pulse Code Modulation and Companding 10
μ-Law Companding 11
A-Law Companding 14
Implementation Using the TMS320C54X **18**
System Requirements vs. Coding Scheme 18
μ-law Compression 20
μ-law Expansion 22
A-law Compression 23
A-law Expansion 25
Summary **26**
Appendix A. Companding Examples **27**
Appendix B. Companding Code Listing **31**
Appendix C. References **36**

Figures

Figure 1. Sample Speech Signal: GOAT 9
Figure 2. μ-law Companding Curve 12
Figure 3. A-law Companding Curve 15

Tables

Table 1. μ-law Binary Encoding Table 13
Table 2. μ-law Binary Decoding Table 14
Table 3. A-law Binary Encoding Table 16
Table 4. A-law Binary Decoding Table 17
Table 5. Companding Algorithms Summary 26
Table 6. μ-law Compression: 27
Table 7. μ-law Expansion: (shown in order as produced by compression table) 28
Table 8. A-law Compression 29
Table 9. A-law Expansion (shown in order as produced by compression table) 30

A-Law and mu-Law Companding Implementations Using the TMS320C54x

Abstract

Presented in this application note is the implementation of A-law and μ-law companding routines for the TMS320C54x. Theoretical material regarding companding and speech signals is provided first, followed by thorough explanations of the algorithms. Finally, the code is benchmarked in terms of its speed and memory requirements.

Introduction

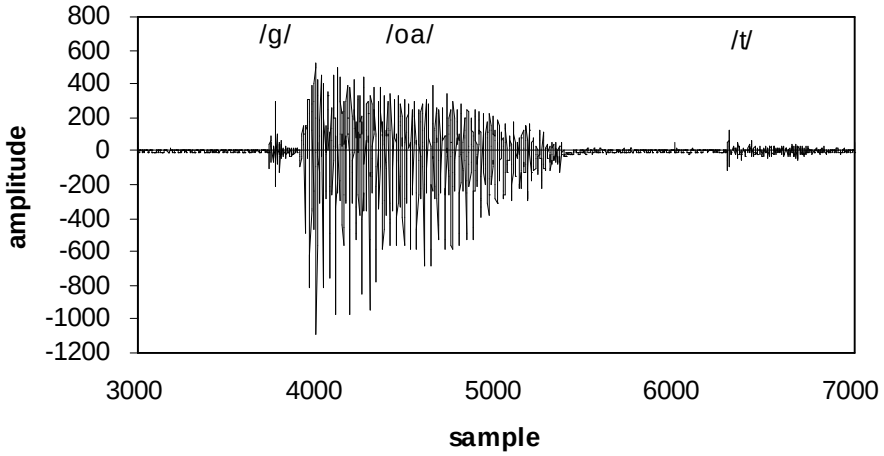
Presented in this section is a description of the components of a speech signal and their influence upon the telephone system. The tasks these components present to the telephone system may be achieved through the use of pulse code modulation and companding, also included in this discussion.

Human Acoustics and the Telephone Network

By classifying according to their mode of excitation, speech sounds can be broken into three distinct classes of phonemes, where a phoneme is defined as the smallest unit of speech that distinguishes one utterance from another. The three classes of phonemes are voiced, unvoiced, and plosives. Voiced phonemes are considered deterministic in nature. They are produced by forcing air through the glottis with the tension of the vocal cords adjusted so that they vibrate in a relaxed oscillation. This produces quasi-periodic pulses of air which excite the vocal tract.[1] Examples of voiced phonemes are the vowels, fricatives /v/, and /z/, and stop consonants /b/, /d/, and /g/. Unvoiced phonemes are generated by forming a constriction at some point in the vocal tract and forcing air through the constriction at a high enough velocity to produce turbulence. As a result, unvoiced phonemes are considered random in nature. Examples of unvoiced phonemes are the nasal consonants /m/, and /n/, fricatives /f/, and /s/, and stop consonants /p/, /t/, and /k/. Similar in nature to unvoiced sounds, plosive sounds result from making a complete closure of the vocal tract, building up pressure behind the closure, and abruptly releasing it, such as the /ch/ phoneme.

Naturally occurring speech signals are composed of combinations of voiced, unvoiced and plosive phonemes. For example, contained in Figure 1 is the speech signal ‘goat’, which contains two voiced phonemes /g/ and /oa/, followed by a partial closure of the vocal tract, and then an unvoiced phoneme, /t/. The /g/, /oa/, and /t/ occur approximately at samples 3400-3900, 3900-5400, and 6300-6900, respectively.

Figure 1. Sample Speech Signal: GOAT



Each phoneme class brings its own stress to the telephone system. In general, the peak to peak amplitude of voiced phonemes is approximately ten times that of unvoiced and plosive phonemes, as clearly illustrated in Figure 1. As a result, the telephone system must provide for a large range of signal amplitudes. Although lower in amplitude, unvoiced and plosive phonemes contain more information and thus, higher entropy then voiced phonemes. Thus, the telephone system must provide higher resolution for lower amplitude signals.

In addition to the tasks presented by the speech signal, the telephone network is also subject to bandwidth restrictions with respect to the human speech and auditory ranges. The speech bandwidth for most adults is approximately 10 kHz. In contrast, the maximum auditory range of humans is 20 kHz. This maximum auditory range is usually limited to young children; instead, the typical hearing bandwidth for most adults is 15 kHz.

Of the speech and auditory bandwidths, the telephone network restricts transmission to a 3 kHz portion, from .3 to 3.3 kHz. This frequency range is believed to coincide with the region of greatest intelligible speech, retaining only the first three formant frequencies of the sampled speech signal. This reduced bandwidth is then surrounded by unused space from 0 to .3 kHz and from 3.3 to 4 kHz. This unused space, known as the guard band, provides a buffer against conversation interference. Summing the transmission and guard bands, the telephone network has a total bandwidth of 4 kHz.

In summary, the telephone system must provide adequate quality for small amplitude signals consisting of unvoiced phonemes. Concurrently, the telephone system must provide for transmission of a wide range of signal amplitudes, due to the occasional occurrence of high energy voiced phonemes. The accomplishment of these concurrent tasks, within a limited bandwidth, may be achieved via Pulse Code Modulation and companding, as discussed in the following section.

Pulse Code Modulation and Companding

At the telephone transmitter, human speech is converted to analog signals. For digital transmission, this analog signal is converted to a digital signal, which has a fixed precision. To provide higher voice quality at a lower cost, the analog signals may be converted to digital signals using Pulse Code Modulation (PCM).

PCM is composed of three successive steps: sampling, quantizing and coding. Sampling is the determination of a signal's amplitude at regular time intervals. Since the telephone network has a bandwidth of 4 kHz, for accurate reproduction, a voice signal must be sampled at a rate of at least 8 kHz, according to Nyquist's theorem. That is, the amplitude of the signal is sampled every 125 μs. Once the signal's amplitude is obtained, it is quantized into a discrete set of amplitude levels for representation as a digital signal. Quantization is achieved by dividing the bandwidth of the system into quantization intervals, also known as bins. All signal amplitudes falling within a bin are represented by the midpoint of that quantization interval. The quantization process introduces quantization error into the digital signal; however, the introduced error may be minimized by minimizing the width of the bins with respect to the number of bits needed to uniquely identify the quantization bins. Finally, coding of the signal is performed by converting the midpoint of each quantization level to a codeword.

In general, speech signals are composed of relatively fewer voiced phonemes than unvoiced phonemes. Unfortunately, the uniform quantizer, which has equally spaced zones, provides unneeded quality for large signals which are least likely to occur, and pronounced truncation effects for the more frequent small amplitude signals. As a result, uniform quantization does not perform as well as a quantizer with wider zones at high amplitudes and narrower zones at lower amplitudes.

Instead of employing uniform quantization, a natural non-uniform substitute is observed in the human auditory system. It is believed that the human auditory system is a logarithmic process in which high amplitude sound does not require the same resolution as low amplitude sound.

Conversion to a logarithmic scale allows quantization intervals to increase with amplitude, and it insures that low-amplitude signals are digitized with a minimal loss of fidelity. Fewer bits per sample are necessary to provide a specified signal-to-noise ratio (SNR) for small signals and an adequate dynamic range for large signals.

Non-uniform quantization may be achieved by first passing the message through a compressor, a nonlinear device which compresses the peak amplitudes. This is followed by a uniform quantizer, such that uniform zones at the output correspond to non-uniform zones at the input. At the receiving end, the compressed signal is passed through an expander, another nonlinear device used to cancel the nonlinear effect of the compressor. The combined process is known as companding.

In addition to reducing quantization error, companding decreases the required bandwidth of the system. That is, systems solely employing uniform quantization require 13-bit codewords for equivalent performance requirements of the telephone system. However, while increasing performance, systems using nonlinear companding may reduce the required codeword length to 8-bits or less.

Companding is simply a system in which information is compressed, flowed through a channel and then expanded on the other side. Companding may be accomplished in hardware via a CODEC, or in software using a look-up table approach or a real-time direct calculation. However, if hardware companding is implemented and intermediate processing of the signal is necessary, then reverse companding is required.

Two international companding standards that retain up to 5 bits of precision by encoding signal data into 8 bits are μ-law and A-law. μ-law is the accepted standard of the U.S. and Japan, while A-law is the European accepted standard. Both international standards are discussed further in the following sections.

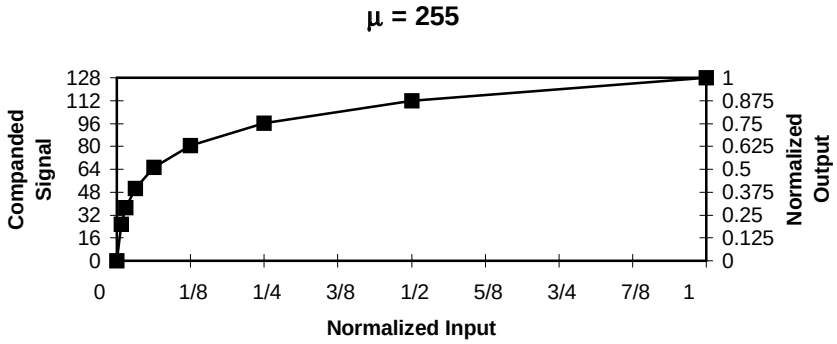
μ-Law Companding

The U.S. and Japan use μ-law companding. Limiting sample values to 13 magnitude bits, the μ-law compression portion of this standard is defined mathematically by the continuous equation:

$$F(x) = \mathrm{sgn}(x)\,\ln(1 + \mu\,|x|) / \ln(1 + \mu) \qquad \text{Equation (1)}$$
$$-1 \le x \le 1$$

where μ is the compression parameter (μ=255 for the U.S. and Japan), and x is the normalized integer to be compressed. A piece-wise linear approximation to this compression equation is illustrated in Figure 2.

Figure 2. μ-law Companding Curve



During compression, the least significant bits of large amplitude values are discarded. The number of insignificant bits deleted is encoded into a special field of the compressed code format, called the chord. Each chord of the piece-wise linear approximation is divided into equally sized quantization intervals called steps. The step size between adjacent codewords is doubled in each succeeding chord. Also encoded is the sign of the original integer. The polarity bit is set to 1 for positive integer values. Thus, an 8 bit μ-255 codeword is composed of 1 polarity bit concatenated with a 3-bit chord concatenated with a 4-bit step.

Before chord determination, the sign of the original integer is removed and a bias of 33 is added to the absolute value of the integer. Due to the bias, the magnitude of the largest valid sample is reduced to 8159 and the minimum step size is reduced to 2/8159. The added bias enables the endpoints of each chord to become powers of two, which in turn simplifies the determination of the chord and step. Chord determination may be reduced to finding the most significant 1 bit of the binary representation of the biased integer value, while the step equals the four bits following the most significant 1.

Illustrated in Table 1 is the translation from linear to μ-law compressed data. Of the compressed codeword, bits 4-6 represent the chord and bits 0-3 represent the step. The polarity bit of the compressed codeword is not shown in this table.

Table 1. μ-law Binary Encoding Table.

Biased Input Values													Compressed Code Word						
													Chord			Step			
bit:	12	11	10	9	8	7	6	5	4	3	2	1	0	bit:	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	1	a	b	c	d	x		0	0	0	a	b	c	d
	0	0	0	0	0	0	1	a	b	c	d	x	x		0	0	1	a	b	c	d
	0	0	0	0	0	1	a	b	c	d	x	x	x		0	1	0	a	b	c	d
	0	0	0	0	1	a	b	c	d	x	x	x	x		0	1	1	a	b	c	d
	0	0	0	1	a	b	c	d	x	x	x	x	x		1	0	0	a	b	c	d
	0	0	1	a	b	c	d	x	x	x	x	x	x		1	0	1	a	b	c	d
	0	1	a	b	c	d	x	x	x	x	x	x	x		1	1	0	a	b	c	d
	1	a	b	c	d	x	x	x	x	x	x	x	x		1	1	1	a	b	c	d

Finally, before transmission, the entire μ-law code is inverted. The codeword is inverted since low amplitude signals tend to be more numerous than large amplitude signals. Consequently, inverting the bits increases the density of positive pulses on the transmission line, which improves the hardware performance.

μ-law expansion is defined by the continuous inverse equation:

$$F^{-1}(y) = \text{sgn}(y)\ (1 / \mu)\ [(1+ \mu)^{|y|} - 1] \qquad \text{Equation}(2)$$
$$-1 \le y \le 1$$

Before expansion, the μ-law code is inverted again to restore the original code. During expansion, the discarded least significant bits are approximated by the median of the interval, to reduce the loss in accuracy. That is, if six of the least significant bits of the original binary integer were discarded during compression, these six least significant bits will be approximated by 100000_2 during expansion. The μ-law binary decoding table used for expansion is given in Table 2. Again, the polarity bit is not shown in this table. After decoding the μ-law code, the bias is removed and the sign of the binary integer is restored according to the polarity bit.

Table 2. μ-law Binary Decoding Table

Compressed Code Word								Biased Output Values											
Chord				Step															
bit: 6	5	4	3	2	1	0	bit: 12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	a	b	c	d	0	0	0	0	0	0	0	1	a	b	c	d	1
0	0	1	a	b	c	d	0	0	0	0	0	0	1	a	b	c	d	1	0
0	1	0	a	b	c	d	0	0	0	0	0	1	a	b	c	d	1	0	0
0	1	1	a	b	c	d	0	0	0	0	1	a	b	c	d	1	0	0	0
1	0	0	a	b	c	d	0	0	0	1	a	b	c	d	1	0	0	0	0
1	0	1	a	b	c	d	0	0	1	a	b	c	d	1	0	0	0	0	0
1	1	0	a	b	c	d	0	1	a	b	c	d	1	0	0	0	0	0	0
1	1	1	a	b	c	d	1	a	b	c	d	1	0	0	0	0	0	0	0

The dynamic range of a compander may be defined as the difference in signal power between the lowest amplitude occupying the entire range of the first chord and the highest occurring amplitude.[2] Using this definition, the dynamic range of μ-law companding is calculated by Equation 3:

$$DR = 20 \log_{10}(8159/31) = 48.4 \text{ dB} \qquad \text{Equation (3)}$$

where 8159 is the largest amplitude possible, and 31 is the lowest amplitude spanning the first chord.

To further clarify the μ-law companding process, several sample conversions from integer values, represented in sign-magnitude form, to μ-law codewords, and vice versa, are given in Appendix A.

A-Law Companding

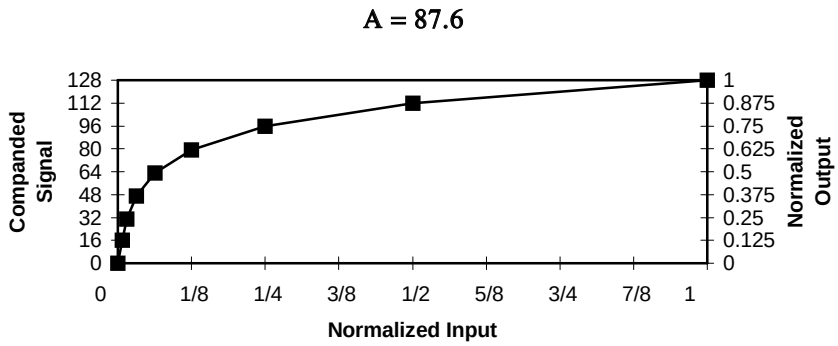
A-law is the CCITT recommended companding standard used across Europe. Limiting sample values to 12 magnitude bits, the compression portion of this standard is defined in the continuous Equation 4:

Equation (4)

$$F(x) = \text{sgn}(x)\, A\,|x| / (1 + \ln A) \qquad 0 \le |x| < 1/A$$
$$= \text{sgn}(x)\, (1+\ln A|x|) /(1 + \ln A) \qquad 1/A \le |x| \le 1$$

where A is the compression parameter (A=87.6 in Europe), and x is the normalized integer to be compressed. A piece-wise linear approximation to this compression equation is illustrated in Figure 3.

Figure 3. A-law Companding Curve



A-law companding has the same basic features and implementation advantages as μ-law companding. A-law companding is approximated by linear segments, with the first chord defined to be exactly linear. A zero-level output for the first quantization interval is not defined. Although biasing of the integer is not required before conversion, the maximum integer value is reduced to 4096. Due to the larger minimum step size of 2/4096, which yields higher quantization error, A-law companding produces small amplitude signals of lower quality than μ-law companding. However, the dynamic range of A-law companding is slightly higher than μ-law, as shown by Equation 5:

$$DR = 20 \log_{10}(4096/15) = 48.7 \text{ dB} \quad \text{Equation (5)}$$

where 15 is the lowest amplitude spanning the first chord.

The two companding standards may also be compared with respect to precision of their binary integer representations. As stated previously, retained during companding are up to 5 bits of precision: the 4-bit step, and the leading 1 (with the exception of values within chord 0). Thus, for μ-law companding, up to 8 bits of precision are lost, while a maximum of 7 bits of precision are lost for A-law companding.

Upon initial consideration, two procedures may be necessary for A-law chord determination, due to the linear definition of the first chord. For binary integers of magnitude greater than $1F_{16}$, the procedure employed in chord and step determination for μ -law compression may be implemented. For binary integers of magnitude less than or equal to $1F_{16}$, the chord is equal to 000 and the step is equal to the resulting 4 least significant bits after dividing the integer magnitude by 2.

Illustrated in Table 3 is the translation from linear to A-law compressed data. Of the compressed codeword, bits 4-6 represent the chord and bits 0-3 represent the step. Only the magnitudes of the input values and compressed codewords are shown; the sign extension of the input value and the polarity bit of the compressed codeword have been omitted.

Once the chord and step have been determined, the polarity of the original integer is determined. That is, if the original integer value is negative, the polarity bit 7 is set to 1; otherwise, the polarity bit 7 is cleared to 0.

Table 3. A-law Binary Encoding Table

Input Values												**Compressed Code Word**						
												Chord			**Step**			
bit:	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**bit:**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	0	0	0	0	0	0	0	a	b	c	d	x		0	0	0	a	b	c	d
	0	0	0	0	0	0	1	a	b	c	d	x		0	0	1	a	b	c	d
	0	0	0	0	0	1	a	b	c	d	x	x		0	1	0	a	b	c	d
	0	0	0	0	1	a	b	c	d	x	x	x		0	1	1	a	b	c	d
	0	0	0	1	a	b	c	d	x	x	x	x		1	0	0	a	b	c	d
	0	0	1	a	b	c	d	x	x	x	x	x		1	0	1	a	b	c	d
	0	1	a	b	c	d	x	x	x	x	x	x		1	1	0	a	b	c	d
	1	a	b	c	d	x	x	x	x	x	x	x		1	1	1	a	b	c	d

Again, to improve hardware performance, an inversion pattern is applied to the codeword before transmission. For A-law companding, the pattern is every other bit starting with bit 0, where bit 0 is the rightmost bit as illustrated in Table 3.

A-law expansion is defined by the continuous inverse equation:

Equation (6)

$$F^{-1}(y) = \operatorname{sgn}(y)\,|y|\,[1+\ln(A)]/A, \qquad 0 \le |y| \le 1/(1+\ln(A))$$
$$= \operatorname{sgn}(y)\,e^{(|y|[1+\ln(A)]-1)}/[A+A\ln(A)], \qquad 1/(1+\ln(A)) \le |y| \le 1$$

Before expansion, the inversion pattern is reapplied to the A-law code to restore the original code. As in μ-law expansion, the least significant bits discarded during compression are approximated by the median of the interval, to reduce loss in accuracy. The A-law binary decoding table used for expansion is given in Table 4. The polarity bit is not shown in this table. After decoding the A-law code, the sign of the integer is restored according to the polarity bit.

Table 4. A-law Binary Decoding Table

Compressed Code Word								Biased Output Values											
Chord				Step															
bit:	6	5	4	3	2	1	0	bit:	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	a	b	c	d		0	0	0	0	0	0	0	a	b	c	d	1
	0	0	1	a	b	c	d		0	0	0	0	0	0	1	a	b	c	d	1
	0	1	0	a	b	c	d		0	0	0	0	0	1	a	b	c	d	1	0
	0	1	1	a	b	c	d		0	0	0	0	1	a	b	c	d	1	0	0
	1	0	0	a	b	c	d		0	0	0	1	a	b	c	d	1	0	0	0
	1	0	1	a	b	c	d		0	0	1	a	b	c	d	1	0	0	0	0
	1	1	0	a	b	c	d		0	1	a	b	c	d	1	0	0	0	0	0
	1	1	1	a	b	c	d		1	a	b	c	d	1	0	0	0	0	0	0

To further clarify the A-law companding process, several sample conversions from integer values to A-law codewords and from A-law codewords back to integer values are given in Appendix A.

Implementation Using the TMS320C54X

The TMS320C54x implementations make use of the EXP and NORM instructions. These instructions allow the extraction of the most significant bits without requiring a look-up table, thus saving memory. The EXP instruction computes the exponent value, which is in the range of -8 to +31, and stores the result in the T register. This is based on the most significant bit in the accumulator. The exponent is determined by subtracting 8 from the number of leading bits in the 40-bit source accumulator (with the exception of the sign bit).

The NORM instruction performs a single-cycle normalization of the accumulator, based on the 6 least significant bits of the T register, interpreted as a 2's complement number. Since the shifter is loaded with the contents of the T register during the read phase, there must be at least 1 cycle between the EXP and NORM instructions, for proper normalization and avoidance of a pipeline conflict.

Further discussion of system requirements with respect to coding schemes and the TSM320C54x is presented in the following section. Also presented are the actual coding schemes implemented for μ-law and A-law companding.

System Requirements vs. Coding Scheme

The major issues involved in coding companding routines are program overhead, required memory, and speed. The effect of program overhead, consisting of context saving and restoring, depends upon how the routines are invoked, and which registers are accessed by the other procedures involved in the system. If the initialized registers are not used by the other system procedures, initialization is performed only once; otherwise, the registers involved must be saved before and restored after execution of the companding routine. Included in the required memory is the program code, any initialization code resulting in program overhead, and any necessary data variables. The goal is to obtain routines with minimum overhead and minimum required memory that execute at maximum speed.

Companding may be performed using a look-up table method, or by direct implementation. Each approach has its advantages and disadvantages with respect to the previously mentioned issues. The look-up table method requires minimum timing (3 cycles), but it is memory intensive. To perform compression and expansion, in addition to the program memory, two 256-word tables are necessary. Coding consists of loading the starting table address into a register, adding the data sample as an offset into the table, and retrieving the codeword or data at the offset address. In addition, significant overhead is required for building the tables in memory. However, this overhead is reduced for those TMS320C54x devices with on-chip ROM containing the expansion tables (‘C542).

Companding by direct implementation may be facilitated using mathematical equations, often called direct encoding, or by invoking simplified algorithms. Either method of direct implementation limits the required memory to that of the coded program, while program overhead is reduced to register initializations. However, most often, memory and overhead reductions are achieved at the expense of increased cycle times.

The choice of companding approaches ultimately depends upon the system’s performance requirements. The method selected for this application note is the direct implementation approach using a simplified algorithm. As will be shown, an attempt is made to minimize the tradeoff between required memory and cycle time, that is, companding by direct implementation is achieved using 16 to 21 words requiring only 10 to 13 cycles.

Reduced overhead or increased performance may be achieved by combining the companding routines with other necessary functions of the overall system. As a result, the described routines only contain the necessary code for conversion; polling of the samples is not performed.

When implementing the described companding routines, the following assumptions and suggestions should be considered. It is assumed that the incoming samples are scaled appropriately, prior to execution, and the sign extension mode is selected. For optimum performance, suggested is the use of the memory-mapped ports for the incoming samples. In addition, the program and data words should be placed in single-cycle memory.

The following sections contain detailed discussions of each routine; the TMS320C54x assembly source code files are located in Appendix B.

μ-law Compression

μ-law compression may be defined by Equation 7.

$$\mu\text{-code} = \mu\text{sgn} + \mu\text{chd} + \mu\text{step} \qquad \text{Equation (7)}$$

As stated earlier, the sample values are limited to 13 magnitude bits, and sign extension mode is selected. When an integer sample is loaded into the A accumulator, the sign bit is extended through the upper accumulator and guard bits. Letting **AH** represent bits 16-31 of the upper A accumulator, the sign of the sample may be positioned into bit 7 of the μ-code by the following equation:

$$\begin{aligned}\mu\text{sgn} &= (AH * -1) << 7 = (AH * -1) * 128 \\ &= AH * (-1 * 128) = AH * (-1 << 7)\end{aligned} \qquad \text{Equation (8)}$$

Substituting the 2's complement of -1 and simplifying, Equation 8 becomes:

$$\mu\text{sgn} = AH * (FFFF_{16} << 7) = AH * FF80_{16} \qquad \text{Equation (9)}$$

The chord is determined after performing the EXP instruction on the magnitude of the biased sample integer. Since the biased sample values allow magnitudes up to 13-bits, excluding the sign bit, the EXP instruction will return values in the range of 18-25, corresponding to the largest and smallest values, respectively. By subtracting the result of the EXP instruction from the maximum possible, 25, the correct chord value is obtained. The chord value is then positioned into bits 4-6, by left shifting it by 4 (or multiplying by 16). Letting $\mathbf{T|_{EXP}}$ represent the contents of the T register after the execution of the EXP instruction, the μchd equation becomes:

Equation (10)

$$\begin{aligned}\mu\text{chd} &= (19_{16} - T|_{EXP})<<4 = 190_{16} - (T|_{EXP})*16 \\ &= 180_{16} - (T|_{EXP})*16 + 10_{16}\end{aligned}$$

As stated earlier, the four bits following the most significant 1 become the step value. The step is determined after normalizing the biased integer by the contents of the T register after the execution of the EXP instruction. After executing the NORM instruction, the step plus the leading 1 are located in bits 27-31 of the source accumulator. The biased step is then positioned into bits 0-4, by right shifting the source accumulator by 26. For true step calculation, the leading 1 located in bit 4 is removed, as seen in Equation 11:

Equation (11)

$$\mu\text{step} = ([(|int| + 33) << T|_{EXP}] >> 26) - 10_{16}$$

Finally, μ-code of Equation 7 is obtained by combining Equations 9, 10 and 11:

Equation (12)

$$\mu\text{–code} = AH * FF80_{16} + 190_{16} - (T|_{EXP})*16 + ([(|int| + 33) << T|_{EXP}\ ] >> 26) - 10_{16}$$

Evident from Equation 12 is that the removal of the leading 1 in the μstep calculation is unnecessary if a biased μchd* equation is implemented. Thus the implemented μchd* and μstep* equations become:

$$\mu chd^* = 180_{16} - (T|_{EXP})*16 \qquad \text{Equation (13)}$$

and

$$\mu step^* = (\ [\ (|int| + 33) << T|_{EXP}\ ] >> 26) \qquad \text{Equation (14)}$$

Another simplification implemented in this algorithm involves the inversion of the 8-bit μ–code for transmission. Inversion of an 8-bit μ–code is equivalent to:

Equation (15)

$$\mu\text{-code}' = FF_{16} - \mu\text{-code} = FF_{16} - \mu sgn - \mu chd - \mu step$$

Substituting Equations 9, 10 and 11, Equation 7 becomes:

Equation (16)

$$\mu\text{-code}' = FF_{16} - AH * FF80_{16} - 180_{16} + (T|_{EXP})*16 - (\ [(|int| + 33) << T|_{EXP}\ ] >> 26)$$

The final result of the μ–law compression algorithm is stored in the lower 7 bits of the B accumulator.

The μ–law compression algorithm requires 17 words of memory: 4 words of data, 10 words of program and 3 words for program overhead. The overhead of this algorithm consists of loading the data page pointer with the page containing the data words and linking the AR0 pointer to the input sample. Assuming optimum conditions, the program contains 10 single-cycle instructions. For a 50 MIPS device, this results in a 200 Nsec execution time. Assuming an 8 kHz sampling rate, this algorithm requires only .08 MIPS.

μ-law Expansion

The μ–law expansion algorithm implements the following equation:

Equation (17)

$$INTNUM = [(2 * \mu\text{–step} + 33) * 2^{\mu\text{–chd}} - 33] * sgn(\mu\text{–sgn})$$
$$= [(2 * \mu\text{–step} + 33) << \mu\text{–chd} - 33] * sgn(\mu\text{–sgn})$$

Since the μ–law code is inverted for transmission, what is received is μ–code'. Inversion of the 8-bit μ–code', extended to 15 bits, is equivalent to Equation 18.

The 15-bit extension is used for easy μ–sgn removal. After the μ–sgn is removed, the chord is saved to the T register, the step is isolated, and the magnitude of INTNUM is determined. Again, for proper normalization, there must be at least one cycle between the loading of the chord into the T register and the normalization of the step.

The sign of the integer is restored using the BIT, XC, and NEG instructions. In this algorithm, the XC instruction allows for the conditional execution of the NEG instruction based upon the results of the BIT instruction. The BIT instruction copies the polarity bit of μ-code into the TC bit of status register ST0. Thus, if the TC bit is 0 (NTC), then the integer represented by μ-code is negative, and the XC instruction allows the NEG instruction to be performed. Since conditions tested by the XC instruction are sampled two cycles prior to its invocation, to avoid pipeline conflict, the BIT instruction must be performed at least 2 cycles prior to the XC instruction. After sign restoration, the expanded integer is stored in the high B accumulator B[31:16].

The μ–law expansion algorithm requires 21 words of memory: 3 words of data, 10 words of program, and 8 words for program overhead. As in the previous algorithm, the overhead consists of loading the data page pointer with the page containing the dataword BIAS and linking the AR0 pointer to the input sample. Additional overhead required is the loading of ASM with the required offset, and the loading of AR2 and AR3 with the addresses of the MASK data and the T register, respectively. This additional overhead is required for implementation of the dual data-memory operand versions of the SUB and ST||LD instructions. Assuming optimum conditions, the program contains 10 single-cycle instructions. For a 50 MIPS device, this results in a 200 Nsec execution time. Assuming an 8 kHz sampling rate, this algorithm requires only .08 MIPS.

A-law Compression

A-law compression may be described by the following equation:

$$acode = asgn + achd + astep. \quad \text{Equation(19)}$$

A-law compression requires the concurrent determination of chord and step. After loading the integer into the B accumulator, the 6 least significant bits are removed; the result is placed in the A accumulator. This removal facilitates a simplified routine for chord determination. Next, the EXP instruction is applied to the A accumulator, after which the T register will contain a value between 25 and 31. NOTE, however, that if the value in the accumulator is zero, the EXP yields NO redundant sign bits. This results in a value of 0 in the T register. The accumulator is checked for a value of 0, in which case 1Fh (31) is written to the T register. The T register value is subtracted from the maximum possible, 31, yielding chord values in the range of 0 to 6. However, the correct chord range is 0-7 and may be obtained by adding the leading 1 prior to the 4 step bits, if present, to this result.

In determining the step, the sign of the original integer is removed and stored to bit 7 of the A accumulator. The magnitude of the integer is normalized by the updated contents of the T register, after which, the step is contained in bits 32-35, and the leading 1, if present, is contained in bit 36, of the B accumulator. For true step determination, bits 32-35 are isolated and the chord calculation is completed by adding bit 36 of the B accumulator to the previous chord result.

Letting $\mathbf{T|_{EXP}}$ represent the contents of T after the EXP execution, and $\mathbf{B36|_{NORM}}$ represent bit 36 of the B accumulator after the NORM execution, the chord, step and sign calculations are described by Equations 20, 21 and 22 respectively:

Equation (20)

$$achd = (1F_{16} - T|_{EXP} + B36|_{NORM})<<4$$
$$= 1F0_{16} - (T|_{EXP})*16 + (B36|_{NORM})<<4$$

Equation (21)

$$astep = (|int| << T|_{EXP}) - (B36|_{NORM} << 4)$$

Equation (22)

$$asgn = (int>>6) \& 80_{16}$$

Substituting Equations 20, 21 and 22 into Equation 19, A-law compression is reduced to Equation 23:

Equation (23)

$$acode = (int>>6)\&80_{16} + 1F0_{16} - (T|_{EXP})*16 + |int|<< T|_{EXP}$$

Finally, the even bits of the A-law codeword are inverted before transmission:

Equation (24)

$$acode^* = acode \text{ XOR } 55_{16}$$

and the compressed codeword is located in the lower 8 bits of accumulator A.

The A–law compression algorithm requires 20 words of memory: 5 words of data, 12 words of program and 3 words for program overhead. The overhead consists of loading the data page pointer with the page containing the dataword MASK, and linking the AR0 pointer to the input sample. Assuming optimum conditions, the program contains 12 single-cycle instructions. For a 50 MIPS device, this results in a 240 Nsec execution time, and assuming an 8 kHz sampling rate, this algorithm requires only .096 MIPS.

A-law Expansion

A-law expansion is defined by Equation 25:

Equation (25)

$$INTNUM = [(2*astep + 33)*2^{achd} - 32*\delta(achd)] *sgn(asgn)$$
$$= [(2*astep + 33)<<achd - 32*\delta(achd)]*sgn(asgn)$$

Where:

Equation (26)

$$\delta(achd) = 1 \qquad achd = 0$$
$$= 0 \qquad achd \neq 0$$

The techniques used in the implementation of Equation 25 are similar to those discussed in the previous algorithms. Therefore, specific details of the A-law expansion routine are limited to the coding comments found in Appendix B.

The A-law expansion algorithm requires 21 words of memory: 5 words of data, 13 words of program, and 3 words for program overhead. The additional 3 words of program are due to the required special handling of the first chord (chord 0). The overhead consists of loading the data page pointer with the page containing the data word MASK1, and linking the AR0 pointer to the input sample. Assuming optimum conditions, the program contains 13 single-cycle instructions. For a 50 MIPS device, this results in a 260 Nsec execution time, and assuming an 8 kHz sampling rate, this algorithm requires .104 MIPS.

Summary

Presented in this application note were companding routines written for the TMS320C54x digital signal processor. Theoretical material regarding companding, as well as detailed discussions of each algorithm, were included.

When strictly dedicated to μ-law companding, the TMS320C54x can compress or expand 5 million words per second, assuming the samples for conversion are available in memory. For A-law companding, compression at 4.16 million words per second is possible, but expansion is reduced to 3.84 million words per second. Possibly of greatest importance is the simultaneous decrease in MIPS and memory requirements achieved by these algorithms, as discussed previously and summarized in Table 5.

Table 5. Companding Algorithms Summary

	Memory Requirements				
	Total	Data	Program	Overhead	MIPS
μ-law compression	17	4	10	3	.080
μ-law expansion	21	3	10	8	.080
A-law compression	20	5	12	3	.096
A-law expansion	21	5	13	3	.104

Appendix A. Companding Examples

Table 6. μ-law Compression:

Integer		Biased Integer		μ-code		μ-code'
-2460_{10} = $F664_{16}$ = $-99C_{16}$	→	$-(99C_{16}+21_{16})$ = $-9BD_{16}$ = (1)0 1001 1011 1101_2	→	(1)110 0011_2 = $E3_{16}$	→	$1C_{16}$
-1505_{10} = $FA1F_{16}$ = $-5E1_{16}$	→	$-(5E1_{16}+21_{16})$ = -602_{16} = (1)0 0110 0000 0010_2	→	(1)101 1000_2 = $D8_{16}$	→	27_{16}
-650_{10} = $FD76_{16}$ = $-28A_{16}$	→	$-(28A_{16}+21_{16})$ = $-2AB_{16}$ = (1)0 0010 1010 1011_2	→	(1)100 0101_2 = $C5_{16}$	→	$3A_{16}$
-338_{10} = $FEAE_{16}$ = -152_{16}	→	$-(152_{16}+21_{16})$ = -173_{16} = (1)0 0001 0111 0011_2	→	(1)011 0111_2 = $B7_{16}$	→	48_{16}
-90_{10} = $FFA6_{16}$ = $-5A_{16}$	→	$-(5A_{16}+21_{16})$ = $-7B_{16}$ = (1)0 0000 0111 1011_2	→	(1)001 1110_2 = $9E_{16}$	→	61_{16}
-1_{10} = $FFFF_{16}$ = -1_{16}	→	$-(1_{16}+21_{16})$ = -22_{16} = (1)0 0000 0010 0010_2	→	(1)000 0001_2 = 81_{16}	→	$7E_{16}$
$+102_{10}$ = 0066_{16} = $+66_{16}$	→	$+(66_{16}+21_{16})$ = $+87_{16}$ = (0)0 0000 1000 0111_2	→	(0)010 0000_2 = 20_{16}	→	DF_{16}
$+169_{10}$ = $00A9_{16}$ = $+A9_{16}$	→	$+(A9_{16}+21_{16})$ = $+CA_{16}$ = (0)0 0000 1100 1010_2	→	(0)010 1001_2 = 29_{16}	→	$D6_{16}$
$+420_{10}$ = $01A4_{16}$ = $+1A4_{16}$	→	$+(1A4_{16}+21_{16})$ = $+1C5_{16}$ = (0)0 0001 1100 0101_2	→	(0)011 1100_2 = $3C_{16}$	→	$C3_{16}$
$+499_{10}$ = $01F3_{16}$ = $+1F3_{16}$	→	$+(1F3_{16}+21_{16})$ = $+214_{16}$ = (0)0 0010 0001 0100_2	→	(0)100 0000_2 = 40_{16}	→	BF_{16}
$+980_{10}$ = $03D4_{16}$ = $+3D4_{16}$	→	$+(3D4_{16}+21_{16})$ = $+3F5_{16}$ = (0)0 0011 1111 0101_2	→	(0)100 1111_2 = $4F_{16}$	→	$B0_{16}$
$+7000_{10}$ = $1B58_{16}$ = $+1B58_{16}$	→	$+(1B58_{16}+21_{16})$ = $+1B79_{16}$ = (0)1 1011 0111 1001_2	→	(0)111 1011_2 = $7B_{16}$	→	84_{16}

Table 7. μ-law Expansion: (shown in order as produced by compression table)

μ-code'		μ-code		Biased Integer		Expanded Integer		
$1C_{16}$	→	$E3_{16}$ = (1) 110 0011_2	→	(1) 0 1001 1100 0000_2 = $-9C0_{16}$	→	$-(9C0_{16} - 21_{16})$	= $-99F_{16}$ = $F661_{16}$	= -2463_{10}
27_{16}	→	$D8_{16}$ = (1) 101 1000_2	→	(1) 0 0110 0010 0000_2 = -620_{16}	→	$-(620_{16} - 21_{16})$	= $-5FF_{16}$ = $FA01_{16}$	= -1535_{10}
$3A_{16}$	→	$C5_{16}$ = (1) 100 0101_2	→	(1) 0 0010 1011 0000_2 = $-2B0_{16}$	→	$-(2B0_{16} - 21_{16})$	= $-28F_{16}$ = $FD71_{16}$	= -655_{10}
48_{16}	→	$B7_{16}$ = (1) 011 0111_2	→	(1) 0 0001 0111 1000_2 = -178_{16}	→	$-(178_{16} - 21_{16})$	= -157_{16} = $FEA9_{16}$	= -343_{10}
61_{16}	→	$9E_{16}$ = (1) 001 1110_2	→	(1) 0 0000 0111 1010_2 = $-7A_{16}$	→	$-(7A_{16} - 21_{16})$	= -59_{16} = $FFA7_{16}$	= -89_{10}
$7E_{16}$	→	81_{16} = (1) 000 0001_2	→	(1) 0 0000 0010 0011_2 = -23_{16}	→	$-(23_{16} - 21_{16})$	= -2_{16} = $FFFE_{16}$	= -2_{10}
DF_{16}	→	20_{16} = (0) 010 0000_2	→	(0) 0 0000 1000 0100_2 = $+84_{16}$	→	$+(84_{16} - 21_{16})$	= $+63_{16}$ = 0063_{16}	= $+99_{10}$
$D6_{16}$	→	29_{16} = (0) 010 1001_2	→	(0) 0 0000 1100 1100_2 = $+CC_{16}$	→	$+(CC_{16} - 21_{16})$	= $+AB_{16}$ = $00AB_{16}$	= $+171_{10}$
$C3_{16}$	→	$3C_{16}$ = (0) 011 1100_2	→	(0) 0 0001 1100 1000_2 = $+1C8_{16}$	→	$+(1C8_{16} - 21_{16})$	= $+1A7_{16}$ = $01A7_{16}$	= $+423_{10}$
BF_{16}	→	40_{16} = (0) 100 0000_2	→	(0) 0 0010 0001 0000_2 = $+210_{16}$	→	$+(210_{16} - 21_{16})$	= $+1EF_{16}$ = $01EF_{16}$	= $+495_{10}$
$B0_{16}$	→	$4F_{16}$ = (0) 100 1111_2	→	(0) 0 0011 1111 0000_2 = $+3F0_{16}$	→	$+(3F0_{16} - 21_{16})$	= $+3CF_{16}$ = $03CF_{16}$	= $+975_{10}$
84_{16}	→	$7B_{16}$ = (0) 111 1011_2	→	(0) 1 1011 1000 0000_2 = $+1B80_{16}$	→	$+(1B80_{16} - 21_{16})$	= $+1B5F_{16}$ = $1B5F_{16}$	= $+7007_{10}$

Table 8. A-law Compression

Integer			A-code			A-code*
-2460_{10}	$= F664_{16}$ $= -99C_{16}$	→	(1) 1001 1001 1100_2	→	(1) 111 0011_2 $= F3_{16}$	→	$A6_{16}$
-1505_{10}	$= FA1F_{16}$ $= -5E1_{16}$	→	(1) 0101 1110 0001_2	→	(1) 110 0111_2 $= E7_{16}$	→	$B2_{16}$
-650_{10}	$= FD76_{16}$ $= -28A_{16}$	→	(1) 0010 1000 1010_2	→	(1) 101 0100_2 $= D4_{16}$	→	81_{16}
-338_{10}	$=FEAE_{16}$ $= -152_{16}$	→	(1) 0001 0101 0010_2	→	(1) 100 0101_2 $= C5_{16}$	→	90_{16}
-90_{10}	$= FFA6_{16}$ $= -5A_{16}$	→	(1) 0000 0101 1010_2	→	(1) 010 0110_2 $= A6_{16}$	→	$F3_{16}$
-1_{10}	$= FFFF_{16}$ $= -1_{16}$	→	(1) 0000 0000 0001_2	→	(1) 000 0000_2 $= 80_{16}$	→	$D5_{16}$
$+40_{10}$	$= 0028_{16}$ $= +28_{16}$	→	(0) 0000 0010 1000_2	→	(0) 001 0100_2 $= 14_{16}$	→	41_{16}
$+102_{10}$	$= 0066_{16}$ $= +66_{16}$	→	(0) 0000 0110 0110_2	→	(0) 010 1001_2 $= 29_{16}$	→	$7C_{16}$
+169	$= 00A9_{16}$ $= +A9_{16}$	→	(0) 0000 1010 1001_2	→	(0) 011 0101_2 $= 35_{16}$	→	60_{16}
$+420_{10}$	$= 01A4_{16}$ $= +1A4_{16}$	→	(0) 0001 1010 0100_2	→	(0) 100 1010_2 $= 4A_{16}$	→	$1F_{16}$
$+499_{10}$	$= 01F3_{16}$ $= +1F3_{16}$	→	(0) 0001 1111 0011_2	→	(0) 100 1111_2 $= 4F_{16}$	→	$1A_{16}$
$+980_{10}$	$= 03D4_{16}$ $= +3D4_{16}$	→	(0) 0011 1101 0100_2	→	(0) 101 1110_2 $= 5E_{16}$	→	$0B_{16}$

Table 9. A-law Expansion (shown in order as produced by compression table)

A-code*		A-code		Expanded Integer		
$A6_{16}$	→	$F3_{16}$ = (1) 111 0011_2	→	(1) 1001 1100 0000_2	→	$-9C0_{16}$ = -2496_{10} = $F640_{16}$
$B2_{16}$	→	$E7_{16}$ = (1) 110 0111_2	→	(1) 0101 1110 0000_2	→	$-5E0_{16}$ = -1504_{10} = $FA20_{16}$
81_{16}	→	$D4_{16}$ = (1) 101 0100_2	→	(1) 0010 1001 0000_2	→	-290_{16} = -656_{10} = $FD70_{16}$
90_{16}	→	$C5_{16}$ = (1) 100 0101_2	→	(1) 0001 0101 1000_2	→	-158_{16} = -344_{10} = $FEA8_{16}$
$F3_{16}$	→	$A6_{16}$ = (1) 010 0110_2	→	(1) 0000 0101 1010_2	→	$-5A_{16}$ = -90_{10} = $FFA6_{16}$
$D5_{16}$	→	80_{16} = (1) 000 0000_2	→	(1) 0000 0000 0001_2	→	-1_{16} = -1_{10} = $FFFF_{16}$
41_{16}	→	14_{16} = (0) 001 0100_2	→	(0) 0000 0010 1001_2	→	$+29_{16}$ = $+41_{10}$ = 0029_{16}
$7C_{16}$	→	29_{16} = (0) 010 1001_2	→	(0) 0000 0110 0110_2	→	$+66_{16}$ = $+102_{10}$ = 0066_{16}
60_{16}	→	35_{16} = (0) 011 0101_2	→	(0) 0000 1010 1100_2	→	$+AC_{16}$ = $+172_{10}$ = $00AC_{16}$
$1F_{16}$	→	$4A_{16}$ = (0) 100 1010_2	→	(0) 0001 1010 1000_2	→	$+1A8_{16}$ = $+424_{10}$ = $01A8_{16}$
$1A_{16}$	→	$4F_{16}$ = (0) 100 1111_2	→	(0) 0001 1111 1000_2	→	$+1F8_{16}$ = $+504_{10}$ = $01F8_{16}$
$0B_{16}$	→	$5E_{16}$ = (0) 101 1110_2	→	(0) 0011 1101 0000_2	→	$+3D0_{16}$ = $+976_{10}$ = $03D0_{16}$

Appendix B. Companding Code Listing

μ-law Compression: int2mu.asm

```
; INT2MU.ASM
;
; Integer originally loaded into A Accumulator
;       (Q13 number is assumed to be sign-extended to 16 bits)
;
;               mucode   = musign + muchord + mustep
;
;               muchord  = (19h - T|EXP)<<4
;                  = 190h - (T|EXP)*16
;                  = 180h - (T|EXP)*16 + 10h
;
;               musign   = (AH * (-1)) <<7
;                  = (AH * FFFFh) * 128 = AH * (FFFFh * 128)
;                  = AH * (FFFFh << 7) = AH * FF80h
;
;               mustep   = ((|int| + 33) << (T|EXP)) << -26) - 10h
;
;  Inversion of 8-bit mu code is equivalent to:
;
;    mucode' = FF - mucode = FF - musign - muchord - mustep
;
;    mucode' = FF-180h-AH*FF80h+(T|EXP)*16-((|int|+33)<<(T|EXP))>>26
;
;  Final output is stored in Low accumulator B(7:0)

        .def    START
        .mmregs

        .data
BIAS1   .word   0FFh-180h
BIAS2   .word    0FF80h
BIAS3   .word    21h
BIAS4   .word    16

TABLE  .word   -2460,-1505,-650,-338,-90,-1,102,169,420,499,980,7000
;TEST VALUES

        .text
START    STM    #TABLE, AR0
         LD     #BIAS1, DP

cbeg     LD     *AR0+, A    ;LOAD INTEGER FOR CONVERSION
         LD     BIAS1, B    ;LOAD (FFh-180h)
         MASA   BIAS2, B    ;Acc B = (FFh-180h-AH*FF80h)
         ABS    A           ;A = |int|
         ADD    BIAS3, A    ;A = |int| + 33 (33 = 21H)
         EXP    A           ;# OF LEADING ZEROS -> (T|EXP)
         MAC    BIAS4, B    ;ACCB += (T|EXP)*16
```

```
NORM    A           ;A<<(T|EXP)
SFTA    A,-16       ;(A<<(T|EXP))>>16
SUB     A,-10, B    ;mucode' = B - (A<<(T|EXP))>>26
B       cbeg        ;DO IT AGAIN!!
```

μ-law Expansion: mu2int.asm

```
; MU2INT.ASM
;
;IMPLEMENT EQUATION
;
;  mucode = musign : muchord  : mustep
;           X         XXX       XXXX
;
;  INTNUM = [ ((2 * mustep + 33) << muchord) - 33 ] * SGN(musign)
;
; NOTE:   since mucode is inverted for xmission, received is
;                   mucode' = musign' : muchord' : mustep'
;
;  Inversion of 8-bit mucode, extended to 15 bits is equivalent to
;
;  mucode = 7FFFh - mucode'
;
;  The 15 bit extension is used for easy musign removal
;
;The final output is stored in high accumulator B(31:16)

       .def    START
       .mmregs
       .data
BIAS   .word  21h
MASK   .word  7FFFh,1Fh
TABLE  .word  1CH,27H,3ah,48h,61h,7eh,0dfh,0d6h,0c3h,0bfh,0b0h,084h
;TEST VALUES

TREG   .equ   14

       .text
START   STM   #TABLE, AR4
        STM   #MASK, AR2
        STM   #TREG, AR3
        LD    #-12, ASM
        LD    #BIAS, DP

cbeg    SUB   *AR2+,*AR4, A ;7FFFh - mucode' -> AH
        AND   A,8           ;REMOVE POLARITY BIT
                            ;(A<<8 AND A)=(7F AND A(16-23))<<24
        ST    A,*AR3        ;STORE CHORD TO T:  (A<<ASM-16)-> T and
     || LD    *AR2-,B       ;LOAD MASK FOR STEP ISOLATION B=1F<<16
        AND   A,-7,B       ;ISOLATE STEP BITS (17-20): BH = 2*mustep
        ADD   BIAS,16,B     ;BH = 2*mustep + 33
        BIT   *AR4+,8       ;COPY SIGN OF mcode'->TC of ST0
        NORM  B             ;BH = (2*mustep+33)<<(muchord)
        SUB   BIAS,16,B     ;BH = (2*mustep+33)<< muchord) - 33
        XC    1,NTC         ;mucode NEGATIVE? (BIT8 == 0; TC ==0)
         NEG  B             ;IF SO NEGATE INTEGER:  BH *= -1
        B     cbeg          ;DO IT AGAIN!
```

A-law Compression: int2alaw.asm

```
; INT2ALAW.ASM
;
; Integer originally loaded into B Accumulator
;       (Q12 number is assumed to be sign-extended to 16 bits)
;
; 6 LSB are then striped off and the result is put in A
;
;  acode  = asgn + achord + astep
;
;  achord = (1Fh - T|EXP + B36|NORM)<<4
;         = 1F0h - (T|EXP)*16 + (B36|NORM)<<4
;
;  asgn   = A8<<7
;
;  astep   = |int| << (T|EXP) - (B36|NORM)<< 4
;    step is located in guard bits of B (B32-39)
;
;  acode = A8<<7 + 1F0h - (T|EXP)*16 + (B36|NORM)<<4
;          + |int|<<(T|EXP) - (B36|NORM)<< 4
;        = A8<<7 + 1F0h - (T|EXP)*16 + |int|<<(T|EXP)
;
;  Before transmission the even bits starting from bit0 are inverted
;
;        acode* = acode XOR %01010101
;
; Final output is stored in low accumulator A(0-7)

        .def    START
        .mmregs

        .sect  acompnd
MASK    .word  80h
BIAS    .word 1F0h
SHFT    .word 10h
INVT    .word 055h
CONST   .word 1Fh

TABLE   .word  -2460,-1505,-650,-338,-90,-1,40,102,169,420,499,980
;TEST VALUES

        .text
START    STM   #TABLE, AR0
         LD    #MASK, DP

cbeg     LD    *AR0+, B  ;LOAD INTEGER FOR CONVERSION
         SFTA  B, -6, A  ;STRIP OFF 6 LSBs:  A = B>>6
         EXP   A         ;# LEADING ZEROS -> (T|EXP) =(25..31)
         ABS   B         ;B = |int|
         XC    1,AEQ ;If number is between 0 and 63
          LD CONST,T ;load 1F into T(EXP of zero yields NO sign bits)
         NORM  B         ;B = |int| << T|EXP
         AND   MASK, A   ;A = asgn
         ADD   BIAS, A   ;A = asgn + 1F0h
         MAS   SHFT, A   ;A -= (T|EXP)*16
         ADD   BG, A     ;A += |int| << T|EXP
         XOR   INVT, A   ;INVERT FOR XMISSION: A =(A XOR 55h)
         B     cbeg      ;DO IT AGAIN!
```

A-law Expansion: alaw2int.asm

```
; ALAW2INT.ASM
;
;IMPLEMENT EQUATION
;
;    A-LAW = ASGN  : ACHD  : ASTEP
;              X      XXX     XXXX
;
; INTNUM = [ (2*ASTEP + 33)*2^(ACHD) - 32*DELTA(ACHD)] * SGN(ASGN)
;
; DELTA(ACHD)    = 1   ACHD == 0
;                = 0   elsewhere
;
; Final output is stored in low accumulator B

       .def    START
       .mmregs

       .sect   aexpnd
TABLE  .word   0ah,0b2h,81h,90h,0f3h,0d5h,42h,65h,6ch,19h,05h,0a6h
;TEST VALUES

MASK1  .word 0ABh;(55h<<1)+1 to account for left shift on load
MASK2  .word     0FFh
ONEF   .word     1Fh
ONE    .word     1
THREE2 .word     32

       .text
START   STM    #TABLE, AR5
        LD     #MASK1,DP

cbeg    LD     *AR5,1,B ;LOAD 2*ALAW CODE
        XOR    MASK1,B  ;INVERT AT RECEIVER
        AND    MASK2,B  ;REMOVE SIGN BIT
        SFTA   B,-5,A   ;STORE CHORD (SHIFT VAL) TO A
        SUB    ONE,A    ;A = CHD -1
        STLM   A,T      ;STORE CHD-1 TO T FOR NORMALIZATION
        AND    ONEF,B   ;ISOLATE SEGMENT
        BIT    *AR5+,8  ;CHECK SIGN OF ORIGINAL A-LAW CODE
        XC     2,AGEQ   ;IF CHD IS NOT -1 EXECUTE NEXT 2 INSTR
         ADD   THREE2,B ;ADD 32 OFFSET
         NORM  B        ;ALIGN SEGMENT, (2*ASTEP+33)*2^(ACHR)
        XC     1,TC     ;CHECK IF A-CODE WAS NEGATIVE (BIT8 == 0)
         NEG   B        ;IF SO NEGATE INTEGER
        B      cbeg     ;DO IT AGAIN!
```

Appendix C. References

Rabiner, L.R., Schafer R.W., *Digital Processing of Speech Signals*, Bell Laboratories, Inc., 1978.

Bellamy, J., *Digital Telephony, 2nd Edition*, John Wiley & Sons, Inc., New York, 1991.

Brokish, C., *μ-law Compression on the TMS320C54x*, TMS320 DSP Designer's Notebook, Texas Instruments, 1996.

Hambley, A.R., *An Introduction to Communication Systems*, Computer Science Press, New York, 1990, pp. 239-251.

Pagnucco, L., Erskine C., *Companding Routines for the TMS32010/TMS32020*, DSP Applications with the TMS320 Family, Vol. 1, Texas Instruments, 1989.

Stremler, F. G., *Introduction to Communication Systems*, 3rd Ed., Addison-Wesley Publishing Co., New York, 1990, pp.402-412, 541-547.