SENG440 Embedded Systems

– Lesson 104: Audio Compression –

Mihai SIMA

msima@ece.uvic.ca

Academic Course

Copyright © 2019 Mihai SIMA

Disclaimer

The purpose of this course is to present general techniques and concepts for the analysis, design, and utilization of embedded systems. The requirements of any real embedded system can be intimately connected with the environment in which the embedded system is deployed. The presented design examples should not be used as the full design for any real embedded system.

Lesson 104: Audio Compression

Motivation

- Audio compression is used in a large number of applications:
  - Digital telephony
  - Voice recording
  - Digital music
  - Automatic speech recognition
- Audio compression is implemented with a logarithm-like function
- Audio expansion is implemented with an exponential-like function
- Efficient implementation of audio compression and expansion in fixed-point arithmetic is critical on embedded systems

Characteristics of speech signal I

- **Phonemes** = basic distinctive units of speech that make up utterances
  - Basic classification of phonemes: voiced and unvoiced
- **Voiced phonemes**
  - Vowels /a/, /o/, /u/, and /i/
  - Fricatives /v/ and /z/
- **Unvoiced phonemes**
  - Nasal consonants /m/ and /n/
  - Fricatives /f/ and /s/
  - Stop consonants /p/, /t/, and /k/
- Voiced phonemes have a much larger amplitude ($\approx 10\times$) and a lower probability of occurrence than the unvoiced phonemes
- Unvoiced phonemes contain more information than voiced phonemes

Characteristics of speech signal II

- Opposite requirements for the telephone system:
  - Transmission of signals with a wide range of amplitudes $> 30\,\mathrm{dB}$
  - High(er) resolution for low(er) amplitude signals
- Uniform Pulse Code Modulation (PCM)
  - The quantization intervals have equal length
  - The codewords are linearly related with the values of the analog samples
- For large signals, which have a low(er) probability of occurence anyway, the uniform PCM provides a higher quality than is actually needed
- The human auditory system has a logarithmic transfer function
- **Logarithmic (non-uniform) PCM is an interesting idea!**

Audio compression – general considerations

- Uniform Pulse Code Modulation (PCM) is an encoding technique where the quantizer values are uniformly spaced
- With equal quantization steps, the relative quantization error is larger for small signal levels than for large signal levels
- **Idea**: the maximum relative quantization error remains the same if:
  - the quantization step for large signal levels is increased
  - the quantization step for small signals levels is kept as it is
- A larger quantization step would lower the number of bits to represent larger signals, that is, compression is achieved
- **Logarithmic PCM** allows 8 bits per sample to represent the same range of values that would be encoded with 14 bits per sample uniform PCM
- This translates into a compression ratio of 1.75 : 1 (original amount of information : compressed amount of information)

Compression laws – μ law and A law

- North American μ-law quantizer (that is, PCM-to-μ-law):

$$y = \operatorname{sgn}(x)\,\frac{\ln(1+\mu|x|)}{\ln(1+\mu)}$$

where the argument $0 \le |x| \le 1$ and μ is a parameter which ranges from 0 (no compression) to 255 (maximum compression).

- European A-law quantizer (that is, PCM-to-A-law):

$$y = \begin{cases} \operatorname{sgn}(x)\,\dfrac{A|x|}{1+\ln A} & \text{for } 0 \le |x| \le \dfrac{1}{A} \\[2ex] \operatorname{sgn}(x)\,\dfrac{1+\ln(A|x|)}{1+\ln A} & \text{for } \dfrac{1}{A} \le |x| \le 1 \end{cases}$$

where $A = 87.6$ and x is the normalized integer to be compressed.

- In the past the nonlinear functions were implemented with analog hardware (e.g., diodes). Today, they are implemented in software.

Implementing the logarithm I

- **How to implement the logarithm using integer arithmetic while achieving a low computing time?**
- Since multiplications and divisions by 2 are simple shift operations: would it be better to implement $\log_2$ rather than natural logarithm?
  - The answer is likely YES
  - Recall that the difference between logarithms in different bases is a factor of scale ($\log_N A = \log_N M \cdot \log_M A$)
- Brute force solution: Look-Up Table (LUT) with 14 inputs and 8 outputs storing the logarithmic function – quite expensive in terms of resources, as the size of the LUT is 16KB.
- To reduce the LUT size: divide the input interval into subintervals and provide a smaller LUT per subinterval – conceptually, the problem is only forwarded to subinterval level

Implementing the logarithm II

- **Taylor series expansion** about a point – approximation good for 1 point

$$\ln(1+a) = a - \frac{1}{2}a^2 + \frac{1}{3}a^3 - \frac{1}{4}a^4 + \dots$$

(recall from Mathematics: this is Taylor series expansion about $a_0 = 0$)

- **Chebyshev expansion** – approximation good for an interval
  - Chebyshev polynomials are used
  - Taylor and Chebyshev methods exhibit similar computational difficulty
- **Piecewise linear approximation**
  - A particular case of series expansion
  - Easy to implement but precision may be an issue

The logarithm: Taylor series expansion

- The formula (expansion about $a_0 = 0$):

$$\ln(1+a) = a - \frac{1}{2}a^2 + \frac{1}{3}a^3 - \frac{1}{4}a^4 + \ldots$$

- Taylor series expansion is an expensive approach in terms of the type of operations (multiplications, divisions), the number of operations, and the wordlength needed to achive the desired precision.
- Can we approximate the logarithm using only the first (linear) term?

$$\ln(1+a) \approx a \qquad \text{about } a_0 = 0$$

- It is possible if the precision is adequate for our task (and, fortunately, it is adequate according to the μ-law standard).
- Recall that the linear approximation is good only about $a_0 = 0$.

The logarithm: piecewise linear approximation I

- Linear approximation:

$$\ln(1+a) \approx a \qquad \text{about } a_0 = 0$$

- In doer to approximate over a large range of values the Taylor series is expanded about more points. In this case multiple linear segments are considered

- This is referred to as **piecewise linear approximation**:

$$\ln(1+a) \approx \ln(1+a_0) + \frac{a-a_0}{1+a_0} \qquad \text{about } x_0$$

The logarithm: piecewise linear approximation II

- Example with four segments:

$$
\begin{aligned}
\ln(1+a) &\approx a && \text{about } a_0 = 0 \\
\ln(1+a) &\approx 1 + \frac{a-(e-1)}{e} && \text{about } a_0 = e-1 \approx 1.72 \\
\ln(1+a) &\approx 2 + \frac{a-(e^2-1)}{e^2} && \text{about } a_0 = e^2-1 \approx 7.39 \\
\ln(1+a) &\approx 3 + \frac{a-(e^3-1)}{e^3} && \text{about } a_0 = e^3-1 \approx 19.09 \\
&\ldots
\end{aligned}
$$

- Piecewise linear approximation is used today, as it is described below

Implementing the μ law I

- Consider the μ-law quantizer:

$$y = \operatorname{sgn}(x) \frac{\ln(1+\mu|x|)}{\ln(1+\mu)}$$

  where the argument $0 \leq |x| \leq 1$ and μ is a parameter which ranges from 0 (no compression) to 255 (maximum compression).

- Analog hardware era $\rightarrow$ the compression was implemented for $\mu = 100$
- In the digital / software era, the factor μ can be easily changed
  - The compression factor μ can be chosen to simplify the conversion
- x is represented on a 14-bit signed integer
  - 1.0 maps to 8192 (and -1 maps to -8192)
  - The scale factor is 2^{13}

Implementing the μ law II

- Example: For $\mu = 15$ binary logarithm needs to be calculated

$$\frac{\ln(1+\mu|x|)}{\ln(1+\mu)} = \frac{\ln(1+15|x|)}{\ln(16)} = \frac{\ln(1+15|x|)}{4\ln(2)} = \frac{1}{4}\log_2(1+15|x|)$$

- Example: For $\mu = 255$ binary logarithm also needs to be calculated

$$\frac{\ln(1+\mu|x|)}{\ln(1+\mu)} = \frac{\ln(1+255|x|)}{\ln(256)} = \frac{\ln(1+255|x|)}{8\ln(2)} = \frac{1}{8}\log_2(1+255|x|)$$

- Computing $\log_2$ should be easier than ln, since multiplications and divisions by 2 are simple shift operations.
  - This is an important observation for the implementation of the compression (or expansion) of speech signals

The μ law in the digital / software era

- Nowadays the standard implementation uses $\mu = 255$
- The compression characteristic can be approximated by:
  - Eight straight-line segments (the **chords**)
  - The slope of each chord is one-half the slope of the previous chord
- The signal amplitude ranges and the chord slopes are:

Range	Chord slope	Range	Chord slope
$[\ \ 0\ldots\ \ 31]$	1	$[\ 479\ldots\ \ 991]$	$1/16$
$[\ 31\ldots\ \ 95]$	$1/2$	$[\ 991\ldots 2015]$	$1/32$
$[\ 95\ldots 223]$	$1/4$	$[2015\ldots 4063]$	$1/64$
$[223\ldots 479]$	$1/8$	$[4063\ldots 8159]$	$1/128$

- **Implementation problem**: how to simply determine the thresholds 31, 95, 223, 479, 991, 2015, 4063, and 8159 (and, therefore, in what interval the input signal is located)

Calculating the compressed codeword I

- The input signal is first converted to a sign-magnitude representation
  - For integers represented in 2'complement, this operation is equivalent to calculating the absolute value and preserving the sign
- A bias of 33 is then added to the magnitude of the input sample
  - This bias enables the thresholds to become powers of 2

$$\begin{aligned} 31 + 33 &= 64 & 991 + 33 &= 1024 \\ 95 + 33 &= 128 & 2015 + 33 &= 2048 \\ 223 + 33 &= 256 & 4063 + 33 &= 4096 \\ 479 + 33 &= 512 & 8159 + 33 &= 8192 \end{aligned}$$

  - The largest valid speech sample is $8192 - 33 = 8159$

Calculating the compressed codeword II

- The position of the most significant bit of 1 in the magnitude gives the **chord**, as shown in the μ-law binary encoding table (next slide)
- The four bits following the most significant bit of 1 give the **step**
- The sign bit of the compressed codeword is:
  - '1' for positive samples
  - '0' for negative samples
- Encoding examples:
  - **0000001010110101** is encoded as **10100110**
    (sign = **1**, chord = **010**, step = **0110**, bits discarded = **101**)
  - **1001110110110110** is encoded as **01011101**
    (sign = **0**, chord = **010**, step = **1101**, discarded = **110110**)
- Before transmission the μ-law codeword is inverted
  - **10100110** becomes **01011001**, **01011101** becomes **10100010**, ...

μ-law binary encoding table

(Adapted from C. Brokish and M. Lewis, *A-Law and mu-Law Companding Implementations Using the TMS320C54x*, Application Note SPRA163A, Texas Instruments, December 1997)

Biased Input Values in Signed-Magnitude Representation (sign bit, 13 magnitude bits)														Compressed Code Word (sign bit, 3 chord bits, 4 step bits)								
s	12	11	10	9	8	7	6	5	4	3	2	1	0	s	6	5	4	3	2	1	0	Chord
0/1	0	0	0	0	0	0	0	1	a	b	c	d	×	1/0	0	0	0	a	b	c	d	1st
0/1	0	0	0	0	0	0	1	a	b	c	d	×	×	1/0	0	0	1	a	b	c	d	2nd
0/1	0	0	0	0	0	1	a	b	c	d	×	×	×	1/0	0	1	0	a	b	c	d	3rd
0/1	0	0	0	0	1	a	b	c	d	×	×	×	×	1/0	0	1	1	a	b	c	d	4th
0/1	0	0	0	1	a	b	c	d	×	×	×	×	×	1/0	1	0	0	a	b	c	d	5th
0/1	0	0	1	a	b	c	d	×	×	×	×	×	×	1/0	1	0	1	a	b	c	d	6th
0/1	0	1	a	b	c	d	×	×	×	×	×	×	×	1/0	1	1	0	a	b	c	d	7th
0/1	1	a	b	c	d	×	×	×	×	×	×	×	×	1/0	1	1	1	a	b	c	d	8th

μ-law binary decoding table

(Adapted from C. Brokish and M. Lewis, *A-Law and mu-Law Companding Implementations Using the TMS320C54x*, Application Note SPRA163A, Texas Instruments, December 1997)

Compressed Code Word (sign bit, 3 chord bits, 4 step bits)								Biased Output Values in Signed-Magnitude Representation (sign bit, 13 magnitude bits)														
s	6	5	4	3	2	1	0	s	12	11	10	9	8	7	6	5	4	3	2	1	0	Chord
1/0	0	0	0	a	b	c	d	0/1	0	0	0	0	0	0	0	1	a	b	c	d	1	1st
1/0	0	0	1	a	b	c	d	0/1	0	0	0	0	0	0	1	a	b	c	d	1	0	2nd
1/0	0	1	0	a	b	c	d	0/1	0	0	0	0	0	1	a	b	c	d	1	0	0	3rd
1/0	0	1	1	a	b	c	d	0/1	0	0	0	0	1	a	b	c	d	1	0	0	0	4th
1/0	1	0	0	a	b	c	d	0/1	0	0	0	1	a	b	c	d	1	0	0	0	0	5th
1/0	1	0	1	a	b	c	d	0/1	0	0	1	a	b	c	d	1	0	0	0	0	0	6th
1/0	1	1	0	a	b	c	d	0/1	0	1	a	b	c	d	1	0	0	0	0	0	0	7th
1/0	1	1	1	a	b	c	d	0/1	1	a	b	c	d	1	0	0	0	0	0	0	0	8th

Implementing the A law

- Consider the A-law quantizer:

$$y = \begin{cases} \operatorname{sgn}(x) \dfrac{A|x|}{1+\ln A} & \text{for } 0 \le |x| \le \dfrac{1}{A} \\ \operatorname{sgn}(x) \dfrac{1+\ln(A|x|)}{1+\ln A} & \text{for } \dfrac{1}{A} \le |x| \le 1 \end{cases}$$

where $A = 87.6$ and x is the normalized integer to be compressed.

- The magnitude values are limited to 12 bits
- x is represented on a 13-bit signed integer
  - 1.0 maps to 4096 (and -1 maps to -4096)
  - The scale factor is 2^{12}
- Biasing is not needed any longer

A-law binary encoding table

(Adapted from C. Brokish and M. Lewis, *A-Law and mu-Law Companding Implementations Using the TMS320C54x*, Application Note SPRA163A, Texas Instruments, December 1997)

Biased Input Values in Signed-Magnitude Representation (sign bit, 13 magnitude bits)													Compressed Code Word (sign bit, 3 chord bits, 4 step bits)								
s	11	10	9	8	7	6	5	4	3	2	1	0	s	6	5	4	3	2	1	0	Chord
0/1	0	0	0	0	0	0	0	a	b	c	d	×	1/0	0	0	0	a	b	c	d	1st
0/1	0	0	0	0	0	0	1	a	b	c	d	×	1/0	0	0	1	a	b	c	d	2nd
0/1	0	0	0	0	0	1	a	b	c	d	×	×	1/0	0	1	0	a	b	c	d	3rd
0/1	0	0	0	0	1	a	b	c	d	×	×	×	1/0	0	1	1	a	b	c	d	4th
0/1	0	0	0	1	a	b	c	d	×	×	×	×	1/0	1	0	0	a	b	c	d	5th
0/1	0	0	1	a	b	c	d	×	×	×	×	×	1/0	1	0	1	a	b	c	d	6th
0/1	0	1	a	b	c	d	×	×	×	×	×	×	1/0	1	1	0	a	b	c	d	7th
0/1	1	a	b	c	d	×	×	×	×	×	×	×	1/0	1	1	1	a	b	c	d	8th

A-law binary decoding table

(Adapted from C. Brokish and M. Lewis, *A-Law and mu-Law Companding Implementations Using the TMS320C54x*, Application Note SPRA163A, Texas Instruments, December 1997)

Compressed Code Word (sign bit, 3 chord bits, 4 step bits)								Biased Output Values in Signed-Magnitude Representation (sign bit, 13 magnitude bits)														
s	6	5	4	3	2	1	0	s	11	10	9	8	7	6	5	4	3	2	1	0	Chord
1/0	0	0	0	a	b	c	d	0/1	0	0	0	0	0	0	0	a	b	c	d	1	1st
1/0	0	0	1	a	b	c	d	0/1	0	0	0	0	0	0	1	a	b	c	d	1	2nd
1/0	0	1	0	a	b	c	d	0/1	0	0	0	0	0	1	a	b	c	d	1	0	3rd
1/0	0	1	1	a	b	c	d	0/1	0	0	0	0	1	a	b	c	d	1	0	0	4th
1/0	1	0	0	a	b	c	d	0/1	0	0	0	1	a	b	c	d	1	0	0	0	5th
1/0	1	0	1	a	b	c	d	0/1	0	0	1	a	b	c	d	1	0	0	0	0	6th
1/0	1	1	0	a	b	c	d	0/1	0	1	a	b	c	d	1	0	0	0	0	0	7th
1/0	1	1	1	a	b	c	d	0/1	1	a	b	c	d	1	0	0	0	0	0	0	8th

Implementing the μ law in software

- Consider that the input samples are represented in 2's complement
- For **compression** the following operations need to be implemented:
  1. Convert the sample to a sign-magnitude representation by storing its sign and calculating its absolute value (magnitude)
  2. To find the chord calculate the location of the most significant bit of '1' (this is equivalent to finding the number of leading zeros)
  3. Extract the four step bits through masking
  4. Assemble the sign, chord, and step bits into a compressed codeword
  5. Perform bit-wise inversion of the codeword

- Questions
  - How to determine the 14-bit sign-magnitude representation fast?
  - How to calculate the chord and the step fast?

Finding the sign and magnitude – sample software I

```
int signum( int sample) {
  if( sample < 0)
    return( 0); /* sign is '0' for negative samples */
  else
    return( 1); /* sign is '1' for positive samples */
}

int magnitude( int sample) {
  if( sample < 0) {
    sample = -sample;
  }
  return( sample);
}
```

- Would the use of predicate operations increase the computing speed?

Finding the chord and step – sample software I

- Only 13 bits of the variable **sample_magnitude** are relevant (the first 3 bits are zero)

```
char codeword_compression( unsigned int sample_magnitude,
                                        int sign) {
  int chord, step;
  int codeword_tmp;

  if( sample_magnitude & (1 << 12)) {
    chord = 0x7;
    step = (sample_magnitude >> 8) & 0xF;
    codeword_tmp = (sign << 7) & (chord << 4) & step;
    return ( (char)codeword_tmp);
  }
```

Finding the chord and step – sample software II

```
if( sample_magnitude & (1 << 11)) {
  chord = 0x6;
  step = (sample_magnitude >> 7) & 0xF;
  codeword_tmp = (sign << 7) & (chord << 4) & step;
  return ( (char)codeword_tmp);
}

if( sample_magnitude & (1 << 10)) {
  chord = 0x5;
  step = (sample_magnitude >> 6) & 0xF;
  codeword_tmp = (sign << 7) & (chord << 4) & step;
  return ( (char)codeword_tmp);
}
```

Finding the chord and step – sample software III

```
if( sample_magnitude & (1 << 9)) {
  chord = 0x4;
  step = (sample_magnitude >> 5) & 0xF;
  codeword_tmp = (sign << 7) & (chord << 4) & step;
  return ( (char)codeword_tmp);
}

if( sample_magnitude & (1 << 8)) {
  chord = 0x3;
  step = (sample_magnitude >> 4) & 0xF;
  codeword_tmp = (sign << 7) & (chord << 4) & step;
  return ( (char)codeword_tmp);
}
```

Finding the chord and step – sample software IV

```
if( sample_magnitude & (1 << 7)) {
  chord = 0x2;
  step = (sample_magnitude >> 3) & 0xF;
  codeword_tmp = (sign << 7) & (chord << 4) & step;
  return ( (char)codeword_tmp);
}

if( sample_magnitude & (1 << 6)) {
  chord = 0x1;
  step = (sample_magnitude >> 2) & 0xF;
  codeword_tmp = (sign << 7) & (chord << 4) & step;
  return ( (char)codeword_tmp);
}
```

Finding the chord and step – sample software V

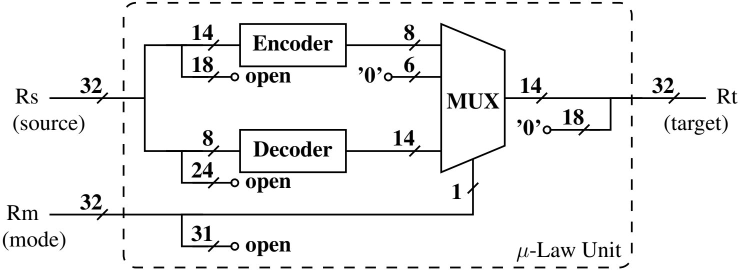
```
  if( sample_magnitude & (1 << 5)) {
    chord = 0x0;
    step = (sample_magnitude >> 1) & 0xF;
    codeword_tmp = (sign << 7) & (chord << 4) & step;
    return ( (char)codeword_tmp);
  }
}
```

Comments and questions

- Large values are the least likely to occur ⇒ do not check them first
- The code is large, as it is riddled with **`if-then-else`** statements
- Would the use of **`switch-case`** statements be a better choice?

Hardware support for the μ law I

- Consider a functional unit which provides architectural support for implementing the μ law (or the A law)



- The estimated latency of this unit is 3 cycles
  - Addition latency is 1 cycle
  - Multiplication latency is 3 cycles

Hardware support for the μ law II

- A new instruction **`muLaw`** to control this functional unit is defined

```
muLaw Rt, Rs, Rm
```

  - **`Rs`** = source register (only the least significant 14 or 8 bits are relevant depending on the mode of operation)
  - **`Rm`** = mode register (the least significant bit is '1' for compression, and '0' for expansion)
  - **`Rt`** = target register (only the least significant 14 or 8 bits are relevant depending on the mode of operation)
- Latency is 3 cycles
- The C code using this instruction is shown next

C code with hardware support

- The code is very simple as it simply calls the new instruction:

```
char codeword_compression( int sample) {
  const int mode_compression = 1;
  int codeword_tmp;

  __asm__ ("muLaw   %0, %1, %2"
           : "=r" (codeword_tmp)
           : "r" (sample), "r" (mode_compression));
  return ( (char)codeword_tmp);
}
```

- The assembly is obtained through compilation:

```
> arm-linux-gcc -S codeword_compression.c
```

Assembly with hardware support

```
codeword_compression:
    str     fp, [sp, #-4]!
    add     fp, sp, #0
    sub     sp, sp, #20
    str     r0, [fp, #-16]
    mov     r3, #1
    str     r3, [fp, #-12]
    ldr     r2, [fp, #-16]
    ldr     r3, [fp, #-12]
    muLaw   r3, r2, r3
    str     r3, [fp, #-8]
    ldr     r3, [fp, #-8]
    and     r3, r3, #255
    mov     r0, r3
    add     sp, fp, #0
    ldmfd   sp!, fp
    bx      lr
```

- The new instruction is used
- The code is riddled with Load and Store operations
- Would the specifier **`register`** help the programmer in removing the Load and Store operations?
- The answer is provided below

Improved C code with hardware support

- The code is very simple as it simply calls the new instruction:

```
char codeword_compression( register int sample) {
  register const int mode_compression = 1;
  register int codeword_tmp;

  __asm__ ("muLaw␣␣␣%0,␣%1,␣%2"
              : "=r" (codeword_tmp)
              : "r" (sample), "r" (mode_compression));
  return ( (char)codeword_tmp);
}
```

- The assembly is obtained through compilation:

```
> arm-linux-gcc -S codeword_compression.c
```

Improved assembly with hardware support

```
codeword_compression:
    str     fp, [sp, #-4]!
    add     fp, sp, #0
    mov     r2, r0
    mov     r3, #1
    muLaw   r3, r2, r3
    and     r3, r3, #255
    mov     r0, r3
    add     sp, fp, #0
    ldmfd   sp!, fp
    bx      lr
```

- The new instruction is used
- The code does not include any Load or Store operations
- Further improvement can be obtained by inlining the function
- The execution of the inlined function would take about 5 cycles (3 cycles for the instruction **`muLaw`** and 1 or 2 cycles for **`mov`** operations)

Audio compression – project requirements

- Record 10 seconds of your voice and save it as a wave file
- Write code to compress the speech signal with the μ law
- Write code to decompress the speech signal with the μ law
- Compare the original voice with the decompressed voice
- Optimize the C code and the assembly and determine the instruction count and processing time
- Provide architectural support for a μ law operation (that is, define a new instruction) and implement the μ law unit in hardware
- Rewrite the C code in order to use the new instruction
- Compare the software solution with the hardware solution

References

- John Bellamy, *Digital Telephony*, John Wiley & Sons, 1982.
- Bernhard E. Keiser and Eugene Strange, *Digital Telephony and Network Integration*, Van Nostrand Reinhold 1985.
- Charles W. Brokish and Michele Lewis, *A-Law and mu-Law Companding Implementations Using the TMS320C54x*, Application Note SPRA163A, Texas Instruments, December 1997.

Questions, feedback

?

Notes I

Notes II

Notes III

Notes IV

Project Specification Sheet

- **Student name:** . . . .
- **Student ID:** . . . .
- **Function to be implemented:** . . . .
- **Argument range:** . . . .
- **Wordlength:** . . . .