AN2488
Rev. 1, 12/2004

Implementation of ITU-T G.711 on the StarCore™ SC140/SC1400 Cores

By Priyadarshan Kolte

G.711 is an ITU-T standard [1] for compressing linear narrowband audio data at a sample rate of 8000 samples per second to a data rate of 64 kbps using logarithmic quantization. G.711 is a simple algorithm that uses one of two coding laws: µ-law (U.S. standard) or A-law (European standard). This application note describes two implementations of G.711 for Freescale Semiconductor DSPs based on the StarCore™ SC140/SC1400 cores. The first implementation minimizes memory usage, and the second uses data parallelism to minimize core processing load. This document discusses each of these implementations in terms of performance measurements and optimization techniques.

CONTENTS

1 External Interface 2
2 StarCore Implementation 2
2.1 G.711 Encoding Basics 2
2.2 G.711 µ-law Encoding 3
2.3 G.711 A-law Encoding 3
2.4 G.711 Decoding 3
3 Optimization Techniques 4
4 Performance 5
5 References 5

1 External Interface

The external interface to the G.711 implementation consists of eight Starcore application binary interface (ABI)-compliant C functions. The low memory implementation consists of the following four functions:

```
void mot_g711_mu_decode (MOT_UINT8 *input, MOT_INT16 *output, MOT_UINT32 count);
void mot_g711_a_decode (MOT_UINT8 *input, MOT_INT16 *output, MOT_UINT32 count);
void mot_g711_mu_encode (MOT_INT16 *input, MOT_UINT8 *output, MOT_UINT32 count);
void mot_g711_a_encode (MOT_INT16 *input, MOT_UINT8 *output, MOT_UINT32 count);
```

- MOT_UINT8 is an array of count 8-bit unsigned integers.
- MOT_INT16 is an array of count 16-bit signed fractions.

The interface to the low-CPU implementation consists of the following functions:

```
void mot_g711_mu_decode_4 (MOT_UINT8 *input, MOT_INT16*output, MOT_UINT32 count);
void mot_g711_a_decode_4 (MOT_UINT8 *input, MOT_INT16 *output, MOT_UINT32 count);
void mot_g711_mu_encode_4 (MOT_INT16 *input, MOT_UINT8*output, MOT_UINT32 count);
void mot_g711_a_encode_4 (MOT_INT16 *input, MOT_UINT8 *output, MOT_UINT32 count);
```

- MOT_UINT8 is an 8-byte aligned array of count 8-bit unsigned integers.
- MOT_INT16 is an 8-byte aligned array of count 16-bit signed fractions.
- Count is a multiple of 4.
- Each input array can be read for four array elements beyond input[count].

The relationship between 14-bit integers and the 16-bit signed fractions for the μ-law encoders and decoders is that the fraction is obtained by shifting the 14-bit integer to the left by 2 bits. Similarly, the relationship between 13-bit integers and the 16-bit signed fractions for A-law is that the fraction is obtained by shifting the 13-bit integer to the left by 3 bits.

2 StarCore Implementation

This section describes the implementation of the G.711 encoders and decoders on the SC140 core. It also describes the optimization techniques and the reasons for limiting the use of the low-CPU procedures.

2.1 G.711 Encoding Basics

G.711 encoders take a 16-bit speech sample as an input and produce an 8-bit encoded PCM value as an output. The encoded value consists of three parts: a sign bit, three bits for the segment, and four bits for the interval within the segment. The μ-law and A-law coding laws have different rules for computing these three parts. The high-level control flow of an encoder is:

```
for each sample
     load 16-bit linear value
     compute 1-bit sign
     compute 3-bit segment
     compute 4-bit interval
     assemble 8-bit value from sign, segment, interval
     apply XOR mask to 8-bit value
     store 8-bit PCM encoded value
```

2.2 G.711 μ-law Encoding

The computation for encoding a single sample is performed as follows:

```
linear = load ()
sign = (linear >> 15) & 0x1
abs = abs (linear)
clipped = min (32632, abs)
biased = clipped + 132
clb = clb (biased)
segment = (clb + 23) & 0x7
shift = clb + 26
interval = (biased >> shift) & 0xf
mu_val = (sign << 7) | (segment << 4) | interval
pcm = mu_val ^ 0xff
store(pcm)
```

The computation of a μ-law PCM value is straightforward except for the computation involving the count leading bits (CLB). The CLB is an SC140 assembler instruction that takes a 40-bit value and returns 9 minus the number of leading zeros. Since *biased* contains just 16 bits, we know that the higher 24 bits have a value of zero. Since the μ-law encoding specifies that this segment is equal to 8 minus the number of leading zeros in *biased*, we compute segment = 8 – (9 – (clb – 24)) which yields segment = clb + 23.

2.3 G.711 A-law Encoding

The computation for encoding a single sample is performed as follows:

```
linear = load ()
sign = (linear >> 15) & 0x1
abs = max (linear, ~linear)
biased = abs | 0x80
clb = clb (biased)
segment = clb + 23
pre_shift = clb + 26
shift = max (pre_shift, 4)
interval = (abs >> shift) & 0xf
a_val = (sign << 7) | (segment << 4) | interval
pcm = a_val ^ 0xd5
store (pcm)
```

The A-law PCM is different from the μ-law in a number of computations:

- The computation of *abs* uses a *max* instruction instead of an *abs*.
- The computation of *biased* sets the bit denoting 128 instead of adding 132.
- The value of *shift* is never less than 4.
- The computation of *interval* shifts *abs* instead of *biased.*
- The value of the XOR mask is 0xD5 instead of 0xFF.

2.4 G.711 Decoding

G.711 decoders take an 8-bit PCM value as an input and produce a 16-bit decoded value as output. Since there are just 256 possible inputs to the decoder, a table look-up is the preferred way to implement the decoder. There is a table of 256 16-bit values for μ-law decoding and a separate table of 256 16-bit values for A-law decoding. The high-level control flow of a decoder is:

```
for each pcm value
load 8-bit value
look up 16-bit decoded value
store 16-bit linear value
```

3 Optimization Techniques

The low-memory version of the G.711 encoder and decoder implements the code sequences shown in **Section 2** in assembler language. For the decoders that use table look-up, the look-up table arrays are aligned on 512-byte boundaries. This alignment simplifies the computation of the expression pcm_to_linear[pcm_value] because the SC140 **insert** instruction can be used to insert the bits of the pcm_value directly into the register containing the pcm_to_linear value. This optimization conserves program memory space and reduces CPU usage. In comparison, the usual sequence of accessing a uint16 array consists of shifting the bits of the index by 1 and then adding to the base of the array.

The only difference between the A-law and μ-law decoders is the base address of the table used. The program memory for the A-law decoder is reduced by loading the base address for the alaw_to_linear array into a register and branching to the code in the μ-law decoder.

The low-CPU version of the G.711 encoder and decoder uses the parallelism of the SC140 core to reduce the CPU requirements. Since G.711 encoding is performed independently on each 16-bit input sample, multiple samples can be encoded in parallel. Similarly, since G.711 decoding of an 8-bit PCM value occurs independently of any other PCM value, multiple PCM values can be decoded in parallel. The SC140 provides four DALUs, so we encode and decode four inputs in parallel. The high-level control flow for the low-CPU encoders and decoders is as follows:

```
for count/4 iterations
     load 4 inputs
     compute the 4 outputs in parallel
     store 4 outputs
```

If the number of inputs is not a multiple of 4, a “clean-up” loop must follow the main loop. The clean-up loop processes the 1, 2, or 3 inputs that remain after the main loop finishes processing all blocks of 4 inputs. To conserve program memory, we require that the number of inputs to the low-CPU version be a multiple of 4. If the number of inputs to be processed is not known to be a multiple of 4, the low-memory version can be used as the clean-up loop.

We require that the input and output arrays be aligned on 8-byte boundaries to permit the use of SC140 **move.4w** instructions, which provide fast access to the memory.

We use a technique known as software pipelining to overlap operations from one iteration of the loop with operations from a subsequent iteration. This reduces the number of CPU cycles by executing operations from consecutive iterations in parallel. The high-level control flow is as follows:

```
load 4 inputs
compute first half of 4 outputs
for (count/4)-1 iterations
     compute second half of 4 outputs for this iteration
     load the 4 inputs for next iteration
     compute first half of 4 outputs for next iteration
     store the 4 outputs for this iteration
compute second half of 4 outputs for last iteration
store the 4 outputs for last iteration
```

To conserve program memory in the software-pipelined version, we fold the computation of the last iteration in with the computation in the loop that performs unnecessary computations in the last iteration. The unnecessary work in the last iteration has negligible impact on the CPU time. However, for safety we require that the input array must be readable for 4 items beyond the number of items in the array. The high-level control flow of this modified software-pipelined version is as follows:

```
load 4 inputs
compute first half of 4 outputs
for count/4 iterations
     compute second half of 4 outputs for this iteration
     load 4 inputs for next iteration
     compute first half of 4 outputs for next iteration
     store 4 outputs for this iteration
```

4 Performance

This section presents the measured CPU and memory requirements of the G.711 implementations. **Table 1** shows the requirements of the low memory implementation, and **Table 2** shows the requirements of the low-CPU implementation. In both tables, *N* indicates the number of channels. The million cycles per second (MCPS) is a measure of CPU cycles required for processing *N* channels. The MCPS measurements assume that the count has a large value, such as 80. Smaller values of count increase the MCPS because the fixed overhead of the procedure call and loop set-up is amortized over fewer inputs.

Table 1. Performance of Low-Memory Implementation

G.711 Function	Program Memory (Bytes)	Table Memory (Bytes)	Stack Memory (Bytes)	Processing Load (MCPS)
mot_g711_mu_encode	74	0	0	N × 0.08
mot_g711_mu_decode	44	512	0	N × 0.04
mot_g711_a_encode	98	0	0	N × 0.10
mot_g711_a_decode	14	512	0	N × 0.04
TOTAL	230	1024	0	

Table 2. Performance of Low-CPU Implementation

G.711 Function	Program Memory (Bytes)	Table Memory (Bytes)	Stack Memory (Bytes)	Processing Load (MCPS)
mot_g711_mu_encode_4	260	8	8	N × 0.03
mot_g711_mu_decode_4	114	512	0	N × 0.01
mot_g711_a_encode_4	268	0	16	N × 0.03
mot_g711_a_decode_4	14	512	0	N × 0.01
TOTAL	656	1032	16	

5 References

[1] *ITU-T Recommendation G.711*, Pulse Code Modulation (PCM) of Voice Frequencies, 1972.

NOTES:

NOTES:

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Order No.: AN2488
Rev. 1
12/2004

freescale™
semiconductor