Efficient Implementation of G.729 Annex A Speech Codec
On a Fixed-Point DSP

MOJTABA VALINATAJ, BEHJAT FORUZANDEH
Department of Electrical and Computer Engineering
University of Tehran
North Karegar Ave., Tehran
IRAN

Abstract: This paper describes the details of bit-exact and real-time implementation of ITU-T G.729 annex A speech codec on a fixed-point DSP processor. The DSP used for implementation is Texas Instruments’ high performance TMS320C6211. In addition to developed new DSPs with lower power consumption at higher clock speeds, the software development tools and compilers have also improved. Thus, the optimization and development process of the signal processing algorithms can be done with the reduced cost and time. In this paper we present some helpful methods for optimizing the G.729A speech codec mostly in high level programming. These techniques are applicable to any speech codec algorithms. Furthermore, the methods that are hardware independent can be used on other DSP processor platforms.

Key-Words: Speech coding, G.729 standard, fixed-point DSP, real-time implementation

1 Introduction

International Telecommunication Union (ITU-T) recommendation G.729 [1] is a toll-quality low bit-rate speech coding standard that has been standardized by Study Group 15 of the ITU-T. The main applications of this coder are personal communications systems, digital satellite systems and internet telephony.

The proposed algorithm in G.729 is CS-ACELP. The quality of reconstructed speech by CS-ACELP algorithm is equivalent to or better than that of 32 Kbps ADPCM technique, which means toll-quality. G.729 annex A recommendation (G.729A) [2] describes a reduced complexity version of G.729 speech codec. Annex B [3] describes a silence compression scheme including a Voice Activity Detection (VAD) and Comfort Noise Generator (CNG) to reduce average transmission bit-rate. All of these recommendations are accompanied by ANSI C code simulation corresponding to a 16-bit fixed-point arithmetic implementation of the codec and a set of test vectors (a set of files), which comprise input and output signals that can be used to verify code being developed. Furthermore, G.729 annex C [4] has been introduced for the floating-point ANSI C code simulation for the main G.729 standard and G.729A.

In the real-time implementation of a speech coding algorithm, the cost of implementation and amount of power consumption are two important factors that need special consideration in the product development. Besides, in many applications a large number of voice channels have to be fit into one DSP processor. Therefore, optimizing a speech codec for efficient implementation in both high level programming and low level assembly language is necessary.

Over the recent past years the programmable DSP and processing hardware have had a tremendous advance in decreasing power consumption while increasing clock speeds. Furthermore, some companies such as TI have developed the software tools and compilers to faster and easily development of signal processing algorithms. Using these software tools certainly leads to reduction in time-to-market.

In this paper we explain several optimization methods and the results of using them in the real-time implementation of G.729A speech codec on TMS320C6211 DSP processor. The organization of the paper is as follows. In section 2, we describe the G.729 speech coding algorithm and the distinctions between G.729 and G.729A. Then, in sections 3 the code optimization techniques mainly applicable to the reference ANSI C code available from the standard body are presented. The bit-exact and real-time implementation by using the previously introduced techniques and their results are explained in section 4. Finally the conclusions are suggested in section 5.
2 G.729 Speech codec

2.1 G.729

Fig.1 and Fig.2 show the block diagrams of CS-ACELP (G.729) encoder and decoder respectively. The coder is designed to operate with an appropriately band limited signal sampled at 8000Hz. The overall bit-rate is 8 Kbps. It processes input signals on a frame-by-frame and subframe-by-subframe basis. The frame length is 10 ms and consists of two 5 ms subframes. The use of subframes allows better tracking of the pitch and the gain parameters and reduces the complexity of the codebooks searches.

The first important operation is LP analysis that is done to estimate the spectral envelope characteristics. In this way, the speech signal is expressed in terms of the computed linear prediction coefficients. The quantized LP coefficients are used in the synthesis filter of this coder. The excitation of this filter consists of two parts: The first one is an adaptive codebook vector to represent the periodic component in the excitation signal (pitch structure of the voiced sounds); and the other is a fixed-codebook vector that represents unvoiced sounds. These two codebook vectors scaled by the respective gains are added to construct the excitation of the synthesis filter. The synthesized speech is constructed in such a way that the minimum distortion relative to the original speech is produced while having the best searched codebook vectors.

In decoder first, the parameters are extracted from the received bit-stream. Then, for each 5ms subframe, the excitation is constructed by adding the adaptive and fixed-codebook vectors scaled by their respective gains. The speech signal is constructed by filtering the excitation through the LP synthesis filter (short-term filter). The reconstructed speech is passed through the post-filters to enhance the perceptual quality. Finally, the post-processing is performed on the reconstructed speech.

2.2 G.729 Annex A

G.729A is a reduced complexity version of the CS-ACELP speech coder at the expense of a slight degradation in speech quality and was designed explicitly for simultaneous voice and data applications that are prevalent in low bit-rate multimedia communications. The general aspects of G.729A are as above; but several changes were made in annex A to reduce the overall complexity. In G.729A the perceptual weighting filter, the open-loop pitch analysis, computation of impulse response of the weighted synthesis filter, the search of adaptive codebook and fixed-codebook, and the post-filtering were simplified.

More details and the reasons behind certain design choices are addressed in [1], [2], [5] and [6].

3 Code Optimization Methods

In this section we describe several useful methods for faster processing and reducing the overall code size; but there is a trade-off between faster processing and reducing the code size while using some of the methods. These optimization methods can be divided in two types. The first type includes the processor independent methods and the other includes the processor dependent methods. In the following these two types include the more helpful methods are explained.
3.1 Processor Independent Methods

3.1.1 Using Registers Optimally
An efficient optimization method is to dedicate registers for temporary or local variables instead of using the memory spaces. By using this method, frequent memory accesses are avoided and considerable amount of cycles are saved [7].

Because of limitation on the number of general-purpose registers, the functions should be written by using of less of temporary variables inside routines. The C language “Register” keyword should be used to provide the additional information to the compiler to dedicate important variables to registers.

3.1.2 Delayed Branching
In pipelined hardware often one or more cycles last before jumping to the target position in the conditional and unconditional jumps and subroutine calls. For conditional branches jump is unpredictable. Therefore, unconditional jumps and subroutine calls can be replaced with delayed branches wherein the processor finishes one or more instructions after the branch instruction and then makes the jump. By modifying the assembly code and using delayed branches or subroutine calls, jump or calling overhead can be reduced.

3.1.3 Loop Unrolling
Unrolling a loop reduces the overhead associated with adjusting the counters and the data pointers. This method is especially useful for inner nested loops and the loops with a large iteration number to reduce the processing time; but loop unrolling causes the code size to be increased. Therefore, complete unrolling is not usually used. It must be noted that loop unrolling when a DSP processor with several parallel execution lines is used (such as C62x), is more helpful.

3.1.4 Using Pointers Instead of Indexed Arrays
This method causes to improve the performance of the used C compiler.

3.1.5 Loop Merging
In some cases in a program, there are some similar loops (have the same length) that are independent. Combining these loops together to perform the operations under one loop reduces the loops overheads proportional to the number of the merged loops.

3.1.6 Code Inlining
Overhead for a function call is high since it leads to push and pop the variables off the stack, and increase the cache misses. Code inlining is removing the functions and directly inserting them inside the main program. Using this method is so helpful to speed optimization especially for the functions with too many calls and smaller functions, although it leads to increase the code size. The function inlining is usually used for functions that are called in the nested loops or in the loops with a large iteration number.

3.2 Processor Independent Methods

3.2.1 Using Intrinsic Functions instead of Basic Operations
In the software tools provided by TI for C5x and C6x DSP families, there is a set of intrinsic functions which are interpreted to a single assembly instruction. In the reference C code, all of the basic math operations such as "add", "sub", "mult", "shl", "shr", etc. have been defined as basic functions. These functions can be replaced and inlined by the intrinsic functions using macro definitions. Using this method leads to a lot of reduction in the processing time by saving function calls overheads and performing operations with fewer instructions.

3.2.2 Using Word Memory Access to Operate on Smaller Data
Some of DSP processors such as C6x DSP family have a set of instructions that can operate on 8-bit or 16-bit data that stored in 32-bit registers. When a procedure must be done on a structure such as an array with data length smaller than 32-bit (word size), the word memory access reduces the overall required time.

3.2.3 Software Pipelining
Software pipelining is a technique used to schedule instructions from a loop so that multiple iterations of the loop execute in parallel [8]. This technique can automatically be done in C5x and C6x DSP families. Of course a programmer can use the compiler directives facilitated with the tools, to directly transmit the loops information to the compiler.

4 Implementation of G.729A on TMS320C62x DSP Processor

4.1 The Implementation Process of a DSP Algorithm
In general, the implementation of an algorithm on a DSP processor should be accomplished in three
In the first stage, a system simulation should be done by using MATLAB or a programming language. In this stage the correctness of the designed system must be investigated.

In the second stage, the optimization methods (introduced in section 3) should be used for optimizing the critical parts of the codes which are typically inner nested loops, in high level programming. The critical parts or non-efficient sections can be found by using the profiling tools. Such tools for C5x and C6x DSP families are provided by TI. If the performance improvement of the used methods in this stage is not enough, stage three must be done.

In the third stage, the usable optimization methods are applied on the low level assembly language. Therefore, converting high level codes to assembly codes according to the used DSP processor is necessary. Such a conversion can be done automatically or by-hand. If automatic conversion is used, the more time-consuming parts of high level codes must be directly rewritten by assembly language. Then, applicable optimization methods must be applied on these codes. If by-hand conversion is used, the optimization methods used in stage two must be applied on the whole assembly codes.

4.2 Implementation Details

4.2.1 ITU-T C Code
As we stated earlier, G.729A standard is accompanied by ANSI C code corresponding to a 16-bit fixed-point arithmetic implementation of the codec. This code was written in a manner to aid porting to different platforms. In order to achieve this, the basic operations were implemented as separate functions. This was done so that these operations can be rewritten in the most optimized style for a specific DSP architecture.

In the system simulation stage, introduced in 4.1 as the first stage, ITU-T C reference code was used. Then, optimization process was done to lay emphasis on the proposed second stage and methods applicable to the high level C codes. For convenience, the optimization methods were separately applied on encoder and decoder parts as follows.

4.2.2 G.729A Encoder
First, by using the software tool provided for C6x DSP family and executing the encoder codes these results were obtained: The required processing power is 275 Millions Cycles Per Second (MCPS), and the size of the overall program memory and data memory is 105.6 KB. As known before, about 30% of the program size consumes 90% of the overall processing time. These time-critical codes include the nested loops and the functions with too many calls. These functions themselves include the basic functions that perform the basic math operations defined in basic_op.c file, and the functions that perform double precision (32-bit) operations.

Optimization process in high level was done in two stages as follows: In the first optimization stage basic functions were optimized by using intrinsic functions and macro definitions which is easily applicable to C5x and C6x DSP families. Since the working reference code is based on a 16-bit fractional arithmetic in order to simulate the DSP hardware, there are lots of overheads in basic operation computations. These overheads are due to function calls overhead and their non-optimized bodies. When a basic function is called in a loop, the cumulative overhead is too high. Weighted Millions Operations Per Second (WMOPS) gives a good estimation for the looping overhead associated with the specific routines as explained in [9].

Most of the basic math functions were replaced by one, two or three intrinsic functions in macro definitions provided by C6x software tool. The basic functions which inlined by using macro definitions are listed in Table 1.

<table>
<thead>
<tr>
<th>Table 1: The basic functions replaced by intrinsic functions by using macro definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define L_deposit_l(a) (a)</td>
</tr>
<tr>
<td>#define L_deposit_h(a) (_sshl(a,16))</td>
</tr>
<tr>
<td>#define L_abs(a) (abs(a))</td>
</tr>
<tr>
<td>#define extract_l(a) (Word16) (a)</td>
</tr>
<tr>
<td>#define extract_h(a) (((a)) &gt;&gt; 16)</td>
</tr>
<tr>
<td>#define norm_s(a) (_norm(a) - 16)</td>
</tr>
<tr>
<td>#define norm_l(a) (_norm(a))</td>
</tr>
<tr>
<td>#define round(a) (_sadd(a,(Word32)0x00008000) &gt;&gt; 16)</td>
</tr>
<tr>
<td>#define negate(a) (((a) == MIN_16) ? MAX_16 : -(a))</td>
</tr>
<tr>
<td>#define saturate(a) (sat((long)(_sahl((a),16))) &gt;&gt; 16)</td>
</tr>
<tr>
<td>#define add(a,b) (sature(a+b))</td>
</tr>
<tr>
<td>#define sub(a,b) (sature(a-b))</td>
</tr>
<tr>
<td>#define mult(a,b) (_smpy((a),(b)))</td>
</tr>
<tr>
<td>#define mult_r(a,b) (_sadd(_smpy((a),(b)),0x00008000) &gt;&gt; 16)</td>
</tr>
<tr>
<td>#define L_add(a,b) (_sadd((a),(b)))</td>
</tr>
<tr>
<td>#define L_mult(a,b) (_smpy((a),(b)))</td>
</tr>
<tr>
<td>#define L_mac(a,b,c) (_sadd((a),(_smpy((b),(c)))))</td>
</tr>
<tr>
<td>#define L_sub(a,b) (_ssub((a),(b)))</td>
</tr>
<tr>
<td>#define L_msu(a,b,c) (_ssub((a),(_smpy((b),(c)))))</td>
</tr>
<tr>
<td>#define L_shl(a,b) (((b) &lt; 0) ? (a) &gt;&gt; (-(b)) : _sshl((a),(b)))</td>
</tr>
<tr>
<td>#define L_shr(a,b) (((b) &gt;= 0) ? (a) &gt;&gt; (b) : _sshl((a),-(b)))</td>
</tr>
</tbody>
</table>
Another change was to separate the basic functions in which the "Overflow" flag is modified and the G.729A algorithm is influenced by this modification, from the other basic functions. In the reference code, most of the basic functions can change the "Overflow" flag, however the G.729A algorithm is influenced by this change only in four functions which are 32-bit "L_add" and "L_sub" and 16-bit "L_mult" and "sature". These four functions were renamed and their optimizations were not done by intrinsic functions, instead by directly inlining them in the next stage. Replacing basic functions by intrinsic functions and inlining them by using macro definitions and changing the functions that modify the "Overflow" flag led to 85% reduction in the required processing power. In another word, the maximum computational complexity for encoder was reduced to 42.3 MCPS.

The second optimization stage concentrated on the function inlining in the nested loops. First, two basic functions 16-bit "add" and 16-bit "sub" were indirectly inlined by intrinsic functions. In other words, these two functions inlined by using "sature" function which itself was replaced by three intrinsic functions. In this way add(a,b) and sub(a,b) were replaced by sature(a+b) and sature(a-b) respectively.

Then, all the functions in which the "Overflow" flag is modified and the G.729A algorithm is influenced by this modification were inlined by hand in several loops. Afterwards, all of the functions that perform double precision operations were inlined in the caller loops.

At the end, some modifications were done in the C codes such as:
- Replacing the shr(data,N) (16-bit shift of data to the right by N locations) by "data>>N" where N is a constant
- Converting if(sub(a,b)>0) to if(a>b).

The results of the first and the second optimization stages for encoder are shown in Table 2.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Average Complexity</th>
<th>Maximum Complexity</th>
<th>Program and Data size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enc.</td>
<td>18 MCPS</td>
<td>19 MCPS</td>
<td>70.7 KB</td>
</tr>
</tbody>
</table>

4.2.3 G.729A Decoder
The optimization process for G.729A decoder was done similar to that of G.729A encoder. First, by using the software tool and executing the decoder codes these results were obtained: The required processing power is 55 MCPS, and the size of the overall program and data memory is 73.7 KB. Then, like the encoder part, the basic functions were optimized by using intrinsic functions and macro definitions. This sort of optimization led to 78% reduction in the required processing power. In the other words, the maximum computational complexity for decoder was reduced to 12.2 MCPS.

The second optimization stage was done like the encoder part. The results of the whole first and second optimization stages for decoder are shown in Table 3.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Average Complexity</th>
<th>Maximum Complexity</th>
<th>Program and Data size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec.</td>
<td>6.3 MCPS</td>
<td>6.5 MCPS</td>
<td>34.1 KB</td>
</tr>
</tbody>
</table>

4.2.4 Porting to TMS320C6211 DSP
All of the performed optimizations so far, have been done using the software tool provided by TI as a simulation on the DSP, and the real DSP has not been used. Now the results of using the real DSP are presented.

As stated before, the used DSP processor is TMS320C6211. The clock rate of this DSP is 150 MHz. The on-chip memory available for both the program and the data is a unified 64 KB RAM. Since it can execute maximum eight instructions per one clock cycle, the maximum processing power of this DSP is 1200 Millions Instructions Per Second (MIPS).

Due to the consuming of some of the external RAM by C functions libraries and the fact that the external RAM is more slower compare to the internal RAM, the computational complexity of the G.729A codec will be slightly more than that of presented in Tables 2 and 3. The final results are represented in Table 4.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Average Complexity (MCPS)</th>
<th>Maximum Complexity (MCPS)</th>
<th>Program Size (KB)</th>
<th>Data size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enc.</td>
<td>23</td>
<td>25</td>
<td>44.7</td>
<td>14.39</td>
</tr>
<tr>
<td>Dec.</td>
<td>8.4</td>
<td>10.1</td>
<td>16.45</td>
<td>12.45</td>
</tr>
</tbody>
</table>

As can be seen in Table 4, the implementation results for encoder and decoder parts have separately been represented. For the real-time implementation, the overall required program and data memory size is not the summation of the numbers shown in Table 4, since the basic functions and many other functions are common between encoder and decoder. In addition, the data memory is mostly
common encoder and decoder parts. From Table 4 we can conclude that up to four full-duplex channels can be implemented on a TMS320C6211 DSP. Since the maximum processing power required for encoder and decoder is equal to 35.1 MCPS (from Table 4) and the processing power of the used DSP is 150 MCPS, the obtained result is correct.

All of the performed optimizations were done on the C reference codes. If further optimizations are done in low level assembly language, the required processing power for G.729A codec can be reduced to ¼ of that of Table 4 or about 8 MCPS, which means about 20 full-duplex channels can be implemented on a TMS320C6211 DSP.

5 Conclusion
In this paper we presented the real-time implementation of G.729 annex A speech codec on a fixed-point C62x DSP. For better understanding that leads to more optimized implementation, the best optimization methods for DSP implementation of signal processing algorithms especially speech coding have been discussed. The more helpful methods used in this work are replacing the basic math operations by the intrinsic functions using macro definitions and inlining the functions with too many calls in the nested loops or in the loops with a large iteration number. It must be noted that this implementation has been accomplished using the optimization methods in high level C codes. Therefore, further optimizations on the processing time and the code size can be done by optimization techniques in assembly language level. In addition, the introduced techniques are applicable to any speech codec algorithms, especially the methods that are hardware independent can be used on other DSP platforms.

In this implementation after each change or replacement in the codes, the output has been checked with a set of test vectors provided by ITU-T for bit-exact implementation.

One tributary result is that C5x DSP family has better instruction set compare to C62x DSP family, since in spite of C62x, the most of basic math functions can be replaced by only one C5x intrinsic function. This lead to more optimized code while using C5x DSP family.

Acknowledgment
This research was funded by the Research Council of University of Tehran.

References: