INSTRUCTION AND HARDWARE ACCELERATIONS IN G.723.1 (6.3/5.3) AND G.729

Mikael Olausson, Dake Liu

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden
{mikol,dake}@isy.liu.se

ABSTRACT

This paper makes accelerations on instruction level based on the three speech coding algorithms G.723.1, 6.3 kbit/s and 5.3 kbit/s and G.729 8 kbit/s with hardware implementation. All these three algorithms are proposed by the H.323 standard together with G.711 64 kbit/s and G.728 16 kbit/s. The work has been done by thoroughly examining the fixed point source code from ITU, International Telecommunication Unions [1], [2]. Three hardware structures are proposed to increase the performance.

1. INTRODUCTION

The market for voice over Internet protocol, also called VoIP, has increased over the years. Voice has been a natural choice of communicating for a long long time and will continue to be so. The H.323 standard contains four different speech coders with different complexity and bit rates. The first one is G.711, which is mandatory and uses A/u-law compression at 64 kbit/s. Another coder is the G.728 Adaptive differential PCM (ADPCM) at 16 kbit/s. The last two are more interesting if we are dealing with bandwidth limited transmission channels. These are G.723.1 and G.729. While the first one have two different bit rates specified, 6.3 and 5.3 kbit/s, the last have three different, 6.4/8.0/11.8 kbit/s. These two both have parts that are common, but also parts that differ a lot. From a market point of view it is of highest interest to make the implementations of these algorithms as efficient as possible. A couple of factors may influence the choice of algorithm. For example some users want to squeeze as many channels as possible on a limited transmission channel. Then their choice is as low bit rate as possible if the speech quality is good enough. Others might use them in battery powered applications and their aim is low power consumption by reduced complexity with reduced speech quality as a tradeoff. Others might aim for high speech quality with limited bit rate. This paper will point out some factors that will influence the choice of speech codec from hardware and complexity point of view. The examination is done from a behavior approach where we are not bound to certain hardware manufacture. The quality or the robustness will not be treated.

2. GENERAL DESCRIPTION OF G.723.1 AND G.729

The first difference between the G.723.1 and the G.729 is the frame size. While the G.723.1 is based on 30 ms(240 samples), the G.729 is based on 10 ms(80 samples) frames. The delay of the algorithms are 37.5 ms and 15 ms respectively. The fixed codebook extraction is the most exhaustive part of the whole algorithm. In these two codecs there exist two different approaches. One which are used in both the lower bit rate of G.723.1 and in G.729 and is Algebraic-Code-Excited-Linear-Prediction, (ACELP). This ACELP places at most 4 non-zero pulses within a subframe. A subframe is 5 ms long in G.729 and 7.5 ms in G.723.1. The positions are determined from the codebook. The second approach is to use Multi-Pulse Maximum Likelihood Quantization (MP-MLQ). This one is used in the higher bit rate of G.723.1. In this case you have more opportunities to place the pulses more individually and not based on an algebraic codebook.

3. STATISTICS OF BASIC OPERATIONS

All the arithmetic and logic functions like add, sub, shift, multiply and so on are implemented with a standard C library. This makes it simply to do statistics over how many times different functions are used. Additional to this, the C code has been thoroughly examined and all the branch, loop and move functions have also been identified and classified. All these statistics over basic operations, branch, loop and move instructions give a good hint on where to find improvements on instruction and architecture level. The statistics are presented in table 4 in the end of the paper. The table corresponds of three columns of numbers. The statistics for each speech coder with both the average and maximum number of times each operations occurs in a frame. The stimuli used for the speech coders are the test vectors included with the C code from ITU [1], [2]. All the statistics are calculated from the encoders only, while the encoding part is the most time consuming.

3.1. Description of the operands

We can see that the most used function is the multiply-and-accumulate, L_MAC. In table 4 we have not made any distinction between accumulation with addition or subtraction. This is not significant from hardware point of view, while most DSP's incorporate both this functions. Here comes some explanation to the table.

The extension _A stands for multiplication with equal input to both operand x and y, for example when performing autocorrelation. Extension _I stands for integer multiplication, i.e. multiplication without proceeding left shift. All the other multiplications are fractional. The basic 16-bit operations like, addition, subtraction, shift, round, negate and so on are left out from the table of statistics. They are almost always present in a DSP.
The second part of the table deals with branch instructions. Except from the total number of them, some special cases has been sorted out. First of all is a distinction between the comparison statements made. We distinguish between 16-bit and 32-bit as indicated by the second part of the word in table 4, MOVE_16 and MOVE_32. The last part of the word, _COND, _CONDA and _COND_I, stands for special cases of branch instructions. _COND means conditional move, _CONDA is conditional move with absolute value of the operand before the comparison statement is executed and the last one, _COND_I, means conditional move of both operand and loop counter. Absolute value calculation of the operand before branch comparison is optional in this operation. The operation TOTAL BRANCH in table 4 is the total number of branch statements found in a frame. The third part of table 4 describes the number of loop operations. The last part of 4 is the number of data movements within a frame. This includes all data movements from clear, set update and move data. No distinction is made between 32-bit and 16-bit move.

Operation	Explanation
ABS_S	16 bit absolute value.
MULT	16-bit multiplication with 16-bit result.
L_MULT	16-bit multiplication.
L_MAC	16-bit multiplication and accumulation.
MULT_R	16-bit multiplication and rounding.
L_SHL	32-bit left shift.
L_SHR	32-bit right shift.
L_ABS	32-bit absolute value.
DIV_S	16 by 16 bit division.
L_MLS	32 by 16-bit multiplication.
DIV_32	32 by 16-bit division.

Table 1: Explanation of some of the operations in table 4.

3.2. Investigation of the statistics

If we look at the L_MAC operations of table 4 more deeply, we will find that around 20% of all MAC operations actually are integer multiplication in the 6.3 kbit/s of G.723.1 and over 33% in the 5.3 kbit/s case. When we look at G.729, there is no need at all for integer multiplication. This means that the multiplier unit must have two modes of operation, fractional and integer in the G.723.1 case. Also from table 4, around

2%-8% of the MAC operations are of the type auto-correlation, this means that the same word must be fed into both the x and y operand of the multiplier.

If we look closer into the branch operations we can see that a large amount of them actually just are conditional moves. The branch condition can be both 16- and 32-bit and be using absolute value or not. For 16-bit branch instructions this corresponds to the row MOVE_16_CONDA in table 4. The C code for this 16-bit branch instruction together with the move instruction looks like the following:

```
a16 = abs_s(a16);
if ( a16 > b16 )
   b16 = a16;
```

This kind of operation will be merged into one instruction, amax a16, b16. The instruction takes the absolute value of a16 and compares it with b16. The biggest of the two are then stored in b16. An extension to the amax instruction is needed for the lower bit rate of G.723.1. The absolute value of operand a16 must be optional. We will call this new instruction max, max a16, b16, and it compares a16 and b16 and stores the biggest value in b16.

For 32-bit branch instructions it gets more complicated. There are conditional moves with and without absolute value, but they are not so many. Instead, a large amount of the 32-bit branch instructions are of the form named
MOVE_32_COND_I in table 4. In this case we do not just perform a move instruction if the branch condition is true, we also have to store the loop counter value. While around 40-45% of the branch related instructions of 6.3 kbit/s of G.723.1 are of the type conditional move, we will design a hardware structure that merge this into one instruction. The hardware is shown in figure 1. This will also reduce the number of branch jumps needed. The branch instructions can be cumbersome if the pipeline is deep. For the other two speech coders, this conditional move is not so pronounced. Especially not in the case of G.729.

Even though division is not used very often, around 60 times per frame for both long and short division, it will be ineffective and time consuming to implement this instruction in software. With hardware support, the clock cycles can be at the range as the number of bits required in the quote.

Normalization, or count the leading one's or zero's of a signed number is another important instruction to incorpate in hardware. Doing it in software will be time consuming.

4. ASSEMBLY INSTRUCTION AND HARDWARE SPECIFICATION FOR IMPROVEMENTS

In this section we will present a couple of hardware architectures to improve the performance of these speech coding algorithms, especially the 6.3 kbit/s implementation of G.723.1.

4.1. 32-bit conditional move with loop index

As we saw from table 4 the 32-bit compare together with conditional move and storage of the loop counter will occur up to 10000 times per frame in the 6.3 kbit/s implementation of G.723.1. This sequence may also include an 32-bit absolute value calculation before the comparison. The pseudo C code looks something like this:

```
for-loop with index i
basic operations
.
.
.
a32 = L_abs(a32); Optional
if ( a32 > b32 )
   b32 = a32;
   store index i;
end of if-statement
end of for-loop
```

These five instructions will be merged into one instruction. A propose of the architecture is shown in figure 1. To perform this operation 3-8 clock cycles would have been required, whether the branch expression is true or not and depending on the hardware support for 32-bit operations. Now, with this hardware improvement, it is reduced down to 1 clock cycle. The 32-bit full adder (FA) on the left in the figure is used for absolute value calculations and is fed from register ACR1. The result from absolute calculation is then compared to the value in register ACR2, the reference register. The biggest of these values can then be stored in either ACR1 or ACR2 by data driven control, MSB, of the result of the compare. We do not need to perform the absolute calculation, instead we can perform the comparison directly between ACR1 and ACR2. In addition to this, a control signal must be sent to the register file if a new loop counter value has to be stored. The path delay from ACR1 via the

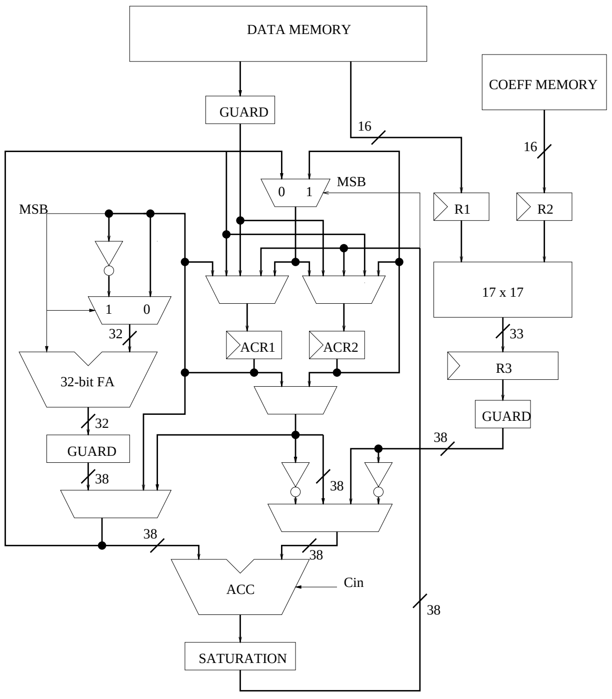

Figure 1: A 32-bit conditional move with absolute value and loop counter move.

32-bit full adder (FA) and the accumulator to the reference register (ACR2) is less the path delay through the multiplier. This extra hardware will not add extra delay to the system.

4.2. Hardware improvements in G.723.1 6.3 kbit/s

By examining the C code for the G.723.1 6.3 kbit/s we found one loop where this 32-bit conditional move with loop index and extra hardware could give dramaticly improvements in the performance. The loop is the search for the pulse positions in the fixed codebook excitation. This loop also uses a complex scheme for the pointer update, when fetching data together with the 32-bit compare and absolute value presented in the previous section. The C code for the loop look like the following:

```
for ( l = 0 ; l < 60 ; l += 2 ) {

   if ( OccPos[l] != (Word16) 0 ){
      continue ;
   }

   Acc0 = WrkBlk[l] ;
   Acc0 = L_msu( Acc0, Temp.Pamp[j-1],
      ImrCorr[abs_s((Word16)(l-Temp.Ploc[j-1]))] ) ;
   WrkBlk[l] = Acc0 ;
   Acc0 = L_abs( Acc0 ) ;

   if ( Acc0 > Acc1 ) {
      Acc1 = Acc0 ;
      Temp.Ploc[j] = (Word16) l ;
   }
}
```

This loop will in the worst case be entered 288 times. The last part of this loop, from the L_abs instruction, is covered by our hardware proposal from the previous section. To keep the performance efficient, we have to make the fetch of the variable ImrCorr in one clock cycle. We can not use an ordinary pointer and just post increment after data fetch due to the absolute value. The solution is instead segmentation addressing with offset. The principle of this addressing and the offset calculation is shown in figure 2. The value stored in Temp.Ploc[j-1] is constant during the whole loop and will be stored in a register, REG in figure 2. To get an efficient calculation of the offset, we have to use a loop counter with variable step size. In this case the step size needs to be two. A second problem is to make it bit exact with the C implementation above. This problem will rise from the fact that the loop in the C code is increasing the loop index l, while the loop counter in hardware is decreasing it's value. This will give a different result in the stored loop index if two or more calculations on Acc0 will give the same result. The solution can be to implement a loop counter that counts in the same direction as the for loop. A better solution is to keep the hardware of the loop counter and instead change the branch option from '>' to '>='.

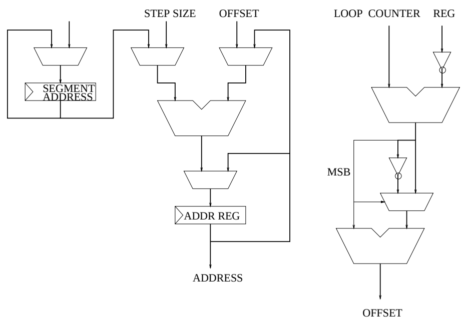

Figure 2: Offset calculation with loop counter and absolute value operation.

5. PERFORMANCE ESTIMATIONS

In order to make an estimation on the performance we have weighted the operation by how many clock cycles they consume. This is of course very hardware dependent, but to get a rough estimate it is a good starting point. It is also important when evaluating the improvements. All the operation from table 4 are grouped together with operations that consumes the same amount of clock cycles. Even the operations that are left out from the table are included in this performance estimation. The table 5 below lists all the operations and their corresponding clock cycle consumption. The branch instructions are weighted after

Cycles	**Operation**
1	add, sub abs_s, shl, shr, negate, extract_h extract_l, round, l_deposit_h, l_deposit_l norm_s and multiplications
2	l_add, l_sub, l_negate, l_shl, l_shr, norm_l
3	l_abs
18	div_s
20	div_32

Table 2: Number of clock cycles per operation

their complexity in the comparison statement. If you compare with zero, then the cycle count is 1. For two 16-bit number the cycle count is 2 and finally, when you compare two 32-bit numbers the cycle count is 3. All the initialization of loops are counted as one clock cycle. When moving data are 16-bit movement treated as 1 clock cycle and 32-bit movement treated as two clock cycles. The only exception is when data is set to zero, then are both 16-bit and 32-bit treated as 1 clock cycle. Table 5 gives the estimated performance of the three speech coders.

6. CONCLUSION

In this paper we have proposed three hardware architectures and three assembly instruction to improve the performance. We have also seen statistics over three different speech coders in terms of basic operations, add, sub, etc, branch statements, loop and move instructions. In table 6 are the estimated savings presented. Most of this work applies to the higher bit rate of G.723.1, 6.3 kbit/s. The number presented, both the cycle count and the performance improvements, are estimated from worst case scenario.

Operation	**G.723.1(6.3)**	**G.723.1(5.3)**	**G.729(8.0)**
Cycle count	6000000	431000	458000
32-bit conditional with loop index	56-90000	12-30000	9-14400
Total saving (%)	9-15	3-7	2-3

Table 3: Improvements of the different hardware and assembly proposals in the G.723.1 and G.729. Note that the figures for G.729 is normalized to 30 ms, 3 frames, in order to be comparable with G.723.1.

7. ACKNOWLEDGMENT

This work was financially supported by the Center for Industrial Information Technology at Linköping Institute of Technology (CENIIT), Sweden.

8. REFERENCES

[1] Itu-t recommendation g.723.1, dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s, 1996.

[2] Itu-t recommendation g.729, coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited-linear-prediction (cs-acelp), 1996.

	G.723 (6.3 kbit/s)	**G.723 (5.3 kbit/s)**	**G.729 (8.0 kbit/s)**
Operation	**Average/Max**	**Average/Max**	**Average/Max**
L_MAC	222373/264810	130872/141129	38734/42384
L_MAC_I	53118/69108	65343/67188	0/0
L_MAC_A	5538/5660	5908/5916	5081/5383
L_MAC_IA	684/720	713/720	0/0
L_MULT	2007/6822	5610/10339	5914/6842
L_MULT_A	119/254	123/254	9/10
MULT	329/7544	2067/7544	6564/7544
I_MULT	0/0	3280/3312	0/0
MULT_R	3308/3478	3436/3478	240/240
ABS_S	6975/8945	1189/1201	21/21
L_ADD	557/682	489/597	596/597
L_SUB	638/841	366/841	832/845
L_SHL	8550/10242	5210/5693	3068/3097
L_SHR	2625/4075	4540/6794	3150/4081
L_ABS	6937/9068	646/652	100/100
NORM_S	6/11	6/11	11/11
NORM_L	378/782	400/790	70/71
L_MLS	291/404	303/404	0/0
MPY_32_16	48/1002	9/1002	972/1002
MPY_32	8/166	1/166	166/166
DIV_S	11/24	15/24	23/24
DIV_32	47/50	49/50	10/10
TOTAL OP	343149/461334	275471/329995	111612/124163
MOVE_16_COND	28/104	2229/2256	84/104
MOVE_16_CONDA	1050/1105	1094/1105	0/0
MOVE_16_COND_I	1/23	91/92	22/23
MOVE_32_COND	4/10	4/10	7/10
MOVE_32_CONDA	337/351	586/591	80/80
MOVE_32_COND_I	8876/10869	3063/3148	508/508
TOTAL BRANCH	18534/28358	12516/20485	4585/5690
TOTAL LOOP	15333/18011	13180/13676	2618/2749
TOTAL MOVE	58633/73370	50360/56163	11592/12335

Table 4: Statistics of G.723.1 6.3 kbit/s, 5.3 kbit/s and G729 8.0 kbit/s. Note the different frame sizes between G.723.1 and G729, 30 ms and 10 ms respectively.