Accelerating VoIP with MIPS® CorExtend® Instructions

Document Number: MD00302
Revision 01.01
June 30, 2008

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction and Background on Voice Over IP

1.1 Introduction

Voice over IP (VoIP) or IP telephony is the technique of using IP networks like the internet to transmit packetized voice and speech data. The main advantage of VoIP is reduced cost of long-distance phone calls, since the connection to the internet is typically just a local call. Other advantages include a more overall streamlined network configuration as the network is IP-based. This allows the call management center to be more automated and support more features. The disadvantage of VoIP is that IP networks do not guarantee the delivery of packets within a specified time limit, and hence some packets may be delayed enough to cause a noticeable degradation in the speech quality. Over the years many speech compression algorithms and standards have emerged in an effort to reduce the amount of data that needs to be transmitted.

Algorithms that process digitized voice and speech signals are computation intensive. Traditionally, these encode and decode algorithms have been executed on dedicated DSP processors. But as the main embedded processors get faster, it has become possible to migrate voice processing to the main processor. If this can be done successfully, then VoIP phones and gateways can eliminate the DSP processor entirely from its SOC (System on Chip) solution. This lowers the overall cost of the product, which is a critical factor for consumer electronics manufacturers.

To extract the best possible performance from such application codes, MIPS Technologies cores offer the CorExtend® capability, which allows the addition of special User Defined Instructions (UDI) to the standard MIPS32® architecture [4]. In this paper, we will use VoIP application code to illustrate how UDIs can be used to speedup application execution on the main processor.

The rest of the paper is organized as follows. This chapter provides a brief introduction to voice compression standards from ITU-T (International Telecommunication Union, Telecom Standardization) [1], and describes one of these standards, G.729, which is the focus of this paper. Chapter 2 illustrates the type of code analysis needed to define the new instructions for increased application performance. Chapter 3 lists the new instructions and describes the hardware needed to implement them. Chapter 4 describes the performance improvement obtained with the new instructions and summarizes the result.

1.2 Voice Compression Standards

Several speech coding standards have been published by the International Telecommunication Union (ITU). These include for example, G.711, G.723, G.723.1, G.726, G.729, and G.729AB. The primary differences between the different standards are the sampling rates, the compression algorithms (hence compression ratios), and the resulting quality of speech or voice signals. Table 1.1 shows a few compression standards.

MOS (Mean Opinion Score) is a subjective voice quality specification, where 5 is excellent and 1 is bad. A score value of 4.0 or higher is considered toll quality. It is not uncommon for many VoIP phones to operate in the 3.5 to 4.0 range using speech codecs such as G.729A.

The G.729 coder is computation intensive, and needs a significant amount of C code to implement. Hence, we use it as an example throughout this application note to illustrate the process of designing CorExtend instructions. The principles used here are equally applicable to other VoIP standards, as well as other application areas.

Table 1.1 Voice Compression Standards

Algorithm	Bandwidth	Frame Size	Compression Ratio	Mean Opinion Score (MOS)
G.711	64 Kbps	0.125 ms	1:2	4.1
G.729	8 Kbps	10 ms	1:16	3.9
G.729A	8 Kbps	10 ms	1:16	3.7
G.723.1[1]	6.3 Kbps	30 ms	1:2	3.4
G.726[2]	32 Kbps	0.125 ms	up to 1:8	3.3

[1] The other bit rate 5.3 Kbps is not shown.

[2] The other bit rates 40, 24, and 16 Kbps are not shown.

1.3 G.729 and G.729 - Annex A

The G.729 coder is based on the Conjugate-Structure Algebraic-Code-Excited Linear-Predictive (CS-ACELP) coding model. The coder operates on speech frames of 10 ms corresponding to 80 16-bit samples at a sampling rate of 8000 samples per second. For every 10 ms frame, the audio signal is analyzed to extract the speech parameters, modeled after the human vocal tract. These parameters are encoded into a 80 bit output frame, thus reducing the original data size by a factor of 16.

The encoder requires many filter passes on the data, as well as a fixed code-book search. The decoding process is significantly faster, because it does not require as many filtering stages and no searches need to be performed.

The Annex A of the G.729 standard is a method for reducing the computational complexity of the standard encoder for the price of a slight loss in fidelity. One of the main speed enhancements is due to a much reduced fixed codebook search. The original G.729 fixed code book search does a brute force search of up to 8192 combinations of codes, while the G.729A search only considers a subset of 256 codes. Thus the codebook search does not find the optimal solution, but in practice the solution found is "good enough". This, in addition to a few other changes, reduces the CPU load of the G.729A encoder by a factor of roughly 2 in most implementations. The decoding process is the same for both G.729 and G.729A.

1.4 Description of CorExtend®

The MIPS32 architecture reserves 16 opcodes under the SPECIAL2 main opcode for the use of User Defined Instructions. This is shown in Figure 1.1, where bits 0 through 3 are available to encode the user’s 16 instruction opcodes. Note that this instruction format has 20 other available bits (6 to 25) for the use of the instruction. Hence, it is possible to encode more than 16 instructions using some of the other bits.

Figure 1.1 Basic UDI Instruction Format

31 26	25 21	20 16	15 6	5 0
SPECIAL2 011100	rs (optional)	rt (optional)	user-interpretable	UDI opcode 01xxxx
6	5	5	10	6

Here are some brief highlights on the flexibility and restrictions imposed by the main core on an UDI block:

- Only fixed integer instructions are allowed. No jumps, branches, loads, or stores are allowed.
- When the main core pipeline decodes to a UDI, two source operands are read using bits *rs* and *rt* as register numbers. These register contents are available on the interface to the UDI block. The UDI block can choose to ignore these if it uses only one or none of the sources for its instruction.
- The destination register value can be derived from any of the bits of the instruction, or independently chosen by the UDI block. The destination could be a GPR or an UDI block internal register.
- The UDI can have a single cycle or multiple cycle latency. All single cycle instructions and multiple cycle instructions that don’t write back to a GPR will not stall the core’s pipeline.
- All multi-cycle instructions that write to a GPR will stall the pipeline. If a UDI block does not allow an instruction to be issued back-to-back, then consecutive issuing of that instruction will also stall the pipeline for the necessary number of cycles.

Chapter 2

Code Analysis to Define New Instructions

2.1 Factors that Influenced the Definition of the New Instructions

This section describes the basic rationale behind the new instructions, describing how they enhance application performance. Original code samples are shown in this chapter to illustrate how new instructions were defined. Code samples rewritten with the new instructions are also shown. The full list of all the user defined instructions recommended for voice applications is listed in the next chapter (see Table 3.2 on page 22).

There are three fundamental factors that dictated the design of the new instructions.

1. When processing digital signals, arithmetic operations that overflow and underflow values by wrapping around are often not useful. Hence, much of the computation in digital speech processing involves saturation, i.e., clamping values to the largest positive or negative representable value with an operation overflow or underflow, respectively.

   Saturation can be handled in two ways. Wide accumulators with 16 guard bits can be used for all the accumulation operations, with overflow and clamping done once, at the end. This is a cleaner implementation and avoids frequent saturation operations. But it is a little more expensive in hardware since wider accumulators have an area cost. Another approach is to do the overflow check and saturation after every operation. The latter approach is mandated by some standards, for example, VoIP standards like G.723.1, G.729, etc. Hence, this method is adopted more frequently.

   To implement operation-level saturation using general microprocessor instructions adds many cycles to the basic arithmetic operation and is a considerable overhead. Hence, the most significant improvement is obtained by defining arithmetic operation instructions that automatically check for overflow and saturate the results.

1. Since speech/voice data can be represented using 16 bits, it is possible to put two data values in the 32 bits of a MIPS32 processor register. These values can be processed in parallel using a single instruction in a SIMD (Single Instruction Multiple Data) register-type of operation. This type of parallelism is very effective in many of the critical routines of the application and helped speedup those routines by a factor of two on top of the other improvements.

2. The third factor that improved application performance was the ability to define new instructions that used more accumulator registers. Since the UDI block is not part of the main integer core, it was possible to assume that this block could implement more than one accumulator so that the instructions specified which one to use, and this eliminated the bottleneck of the single hi/lo register pair of the main architecture. Defining new state has the disadvantage that it makes the code non-reentrant since these registers would not be saved on a context-switch (without OS modifications).

The above list provides some background and will help the reader understand the reasoning behind the new instructions that will be used in this chapter in the code samples.

2.2 G.729 Code Analysis

The G.729 algorithm was analyzed and the performance improved using the following methodology:

1. First, all obvious and trivial optimizations that could be done in C using the compiler were done. This includes forced inlining of all the ITU basic_op and oper_32b macros, as well as using the standard set of optimizations flags for best speed performance. (The ITU source code for the G.729 and G.723.1 codecs share a common set of basic signal processing operations. These are the ones being referred to here as the basic_ops. These include primitives such as 16 bit and 32 bit addition, subtraction, multiply, multiply-accumulate, rounding, saturation, and normalization.)

3. Once the basic compiler optimizations were done, a profile was generated of the entire application. The profiling helped isolate the most critical and important routines. It was also possible to use this information to generate upper bounds on the possible speedups.

4. The most critical routines were examined for their underlying algorithms to determine what functionality must be defined for the new instructions.

5. The previous step went hand-in-hand with the rewriting in assembly of these most important routines using the new UDIs.

It is not possible within the scope of this paper to show all the converted routines, but we will attempt to illustrate a flavor of this process. The application profile discussed in step two is shown in Section 2.2.1 “Initial Application Profile”. Two example routines are discussed in Section 2.2.2 “Example: D4i40_17” and Section 2.2.3 “Example: pred_lt_3”. All the analysis and code rewriting was done for the G.729 encoder. This is more computation-intensive than the G.729 decoder. The decoder runs in about 20-25% of the encoder’s execution time.

2.2.1 Initial Application Profile

The profiling of the application is shown in **Table 2.1**. Values do not sum to 100% because not all routines are shown. But the table shows all the routines that have at least a minimal impact on the total performance of the application. (The total run-time shown in the table sums to about 88%).

Table 2.1 G.729 Encoder Subroutine Run-time Profile

Percent	Function
24.9	D4i40_17
11.4	Lag_max
10.2	Norm_Corr
7.5	Syn_filt
4.7	Lsp_pre_select
4.4	Convolve
3.7	Residu
3.6	Autocorr
3.3	Chebps_11
2.2	Cor_h_X
2.1	Cor_h
1.9	Pred_lt_3
1.5	overhead
1.2	Levinson
1.2	Qua_gain
0.9	Lsp_select_2
0.9	Lsp_select_1
0.8	Pre_Process
0.7	Coder_ld8k
0.6	Az_lsp

2.2.2 Example: D4i40_17

D4i40_17, also known as the fixed code book search, is the most computation intensive subroutine of the G.729 algorithm. This is because it is searching a large space of codes, to find the best match for the unvoiced (pitch-less) component of the speech frame. The unvoiced component is modeled as four random signed impulses in time, convolved with the impulse response. Because this is done on a sub-frame (40 samples), there are 40*39*38*37/4*3*2*1 = 91390 possible sets of four impulse positions. To reduce this combinatorial space, each of the four impulses is constrained to be in its own unique “track”. Table 2.2 shows the allowed positions for each index.

Table 2.2 Fixed-Code Book Search Index Encoding

Impulses	Possible Positions	Bits to encode (without sign of impulse)	Bits to encode (with sign of impulse)
i0	0,5,10,15,20,25,30,35	3	4
i1	1,6,11,16,21,26,31,36	3	4
i2	2,7,12,17,22,27,32,37	3	4
i3	3,4,8,9,13,14,18,19,23,24,28 29,33,34,38,39	4	5
Total	0 through 39	13	17

Each of the impulses has eight possible positions and a sign, requiring 3+1=4 bits to encode, except for the fourth impulse which requires 5 bits to encode. All the impulse positions and signs can be encoded using 17 bits, which is exactly the same 17 bits that are written to the 80 bit output frame. If the signs are omitted and not encoded, only 13 bits are needed to encode the set of impulse positions. The fact that 13 bits will fit in a half-word of 16 bits is used in the optimized algorithm, which is discussed below.

The G.729 algorithm loops over all possible impulses with four nested loops, resulting in a space of 8*8*8*16 = 8192 possible combinations. Only the positions of the impulses are searched; the signs of the impulses can be determined after the search. As a shortcut, the fourth (innermost) loop of i3 is not done if the mean-square error due to the first 3 impulses is above a threshold. Because of the conditional nature of this threshold and the requirement that the optimized code be bit exact with the ITU reference code, it is not efficient to attempt to vectorize the entire nest of loops. However, the inner most loop when executed, (passing the threshold condition), can be vectorized, because it always loops over the 16 values of i3. Since the data values can be represented in 16 bits, with register SIMD parallelization, the inner most loop can be vectorized by a factor of 2 (with MIPS32 data size and registers).

The original algorithm attempts to minimize the mean-squared error by maximizing a ratio of Q15 values (the details are omitted here for clarity). Because division is computationally expensive, two ratios are not compared by dividing numerator by denominator, but instead by cross-multiplication of denominators with numerators. The best result is stored in pieces, as the numerator, the denominator and the 4 indices that give this result. Each new combination of indices is compared by cross-multiplying the proposed numerator and denominator with the best numerator and denominator.

In the original algorithm, four integer variables hold the four indices of the best ratio. As seen in the table above, all of the indices can be packed into a 13 bit code, which will fit in a halfword.

The original C code of the first inner i3 loop of D4i40_17 is shown in Table 2.3. L_SUBFR is 40 and STEP is 5, resulting in 8 iterations of this loop. The L_mac operation is implemented using the MAQS_PH instruction. This does a SIMD multiply and accumulates the results to two accumulators. The operation is done on fractional data format and saturated.

The cross product comparison is done with the L_mult(ps3c,alpha) and subsequent L_Msu(L_temp,psc, alp) instructions, with the sign of the L_temp variable giving the result of the ratio comparison. If L_temp is positive, the “best” values of psc and alpha are updated with the new values of ps3c and alp respectively.

The cross-multiplication and comparison can be vectorized using the MULQ (fractional SIMD multiply) instruction, the C.LT comparison instruction, and finally the PICK instruction (see Table 3.2 on page 22). However, because of the parallelization, the “best” numerator and denominator are also stored as vectors (a pair of PH for MIPS32+UDI, see Figure 3.3   on page 21). In addition, the “best” index code is also stored in a PH vector. At the end of the code book search index loops, the two values in the “best” numerator vector and denominator vector need to also be reduced to a single “best” result, and a single “best” index code. This coded index (13bits) is re-encoded as 17 bits by inserting the sign bits, and this is the value that is written to the final output frame.

Table 2.3 First Inner Loop of D4i40_17 Fixed Codebook Search, C-implementation

```
for (i3 = 3; i3 < L_SUBFR; i3 += STEP)      /* 4th pulse loop */
     {
           ps3 = add(ps2, Dn[i3]);
           alp3 = L_mac(alp2, *ptr_ri3i3++, 1);
           alp3 = L_mac(alp3, *ptr_ri0i3++, 2);
           alp3 = L_mac(alp3, *ptr_ri1i3++, 2);
           alp3 = L_mac(alp3, *ptr_ri2i3++, 2);
           alp  = extract_l(L_shr(alp3, 5));
           ps3c = mult(ps3, ps3);
           L_temp = L_mult(ps3c, alpha);
           L_temp = L_msu(L_temp, psc, alp);
           if( L_temp > 0L ) {
             psc = ps3c;
             alpha = alp;
             ip0 = i0;
             ip1 = i1;
             ip2 = i2;
             ip3 = i3;
     }
}  /*  end of for i3 = */
```

Table 2.4 shows a snippet of the first inner i3 loop of D4i40_17. The register names have been predefined using cpp macros to make it easier to follow. The inner i3 loop of 8 iterations is flattened into four passes including 2 iterations each. There is no branching since the entire loop is unrolled. The original C-code is intermixed in the comments to illustrate the new code. The snippet only shows the first of the four passes. The others are very similar and hence are not shown.

The comparison magic happens due to these 3 instructions:

```
C_LT_W(temp2,temp1,1)
...
C_LT_W(temp2,temp1,0)
...
PICK_PH(psc,ps3c,psc)
```

The two C_LT_W instructions write one condition bit each into the internal Vector Condition (VC) UDI register (see Table 3.1 on page 21). The third argument to the compare instruction specifies the offset position of the bit in the VC register. The final PICK_PH instruction uses the results of the VC register to decide how to update the psc register. A 1 means to take the element from the ps3c register, while a 0 means to retain the old value from the psc register. The PICK instruction is repeated twice more for the alpha register and the index register.

For the sake of clarity, this sample does not show the best possible code schedule. For example, to prevent a stall between the MAQS_PH instruction and the first MFUS, it is possible to move a non-dependent instruction from below up between the two.

Table 2.4 First Inner Loop of D4i40_17 Fixed Codebook Search, MIPS32+UDI Implementation

```
// flatten two iterations of the loop at a time. First do 3,8
   lh temp2,  3*2(Dn)
   lh ps3,    8*2(Dn)
   ins ps3,temp2,16,16

/*      ps3 = add(ps2, Dn[i3]); */
   ADDS_PH(ps3,ps2,ps3)

   // replicate alp2 into the accumulators
   MTU(alp2,VoUDI_Acc1)
   MTU(alp2,VoUDI_Acc0)

//    alp3 = L_mac(alp2, *ptr_ri3i3++, 1);
   lw temp1,0(ptr_ri3i3)
   lw temp2,0(ptr_ri0i3)
   MAQS_PH(temp1,one)

//    alp3 = L_mac(alp3, *ptr_ri0i3++, 2);
   MAQS_PH(temp2,two)

//    alp3 = L_mac(alp3, *ptr_ri1i3++, 2);
   lw temp1,0(ptr_ri1i3)
   lw temp2,0(ptr_ri2i3)
   MAQS_PH(temp1,two)

//    alp3 = L_mac(alp3, *ptr_ri2i3++, 2);
   MAQS_PH(temp2,two)

//    alp  = extract_l(L_shr(alp3, 5));
   MFUS(temp_h,VoUDI_Acc1,0)
   MFUS(temp_l,VoUDI_Acc0,0)
   SRA_W(temp_h,temp_h,5)
   SRA_W(temp_l,temp_l,5)
   CVTS_PH_W(alp,temp_h,temp_l)

//           ps3c = ((long)ps3 *ps3)>>15;
   MULQ_PH(ps3c,ps3,ps3)

//          if(ps3c*alpha > psc*alp) { /* there is a factor of 2, but
it doesn’t matter here */
   MULQ_W_PHH(temp1,ps3c,alpha)
   MULQ_W_PHH(temp2,psc ,alp)
   C_LT_W(temp2,temp1,1)
   MULQ_W_PHL(temp1,ps3c,alpha)
   MULQ_W_PHL(temp2,psc ,alp)
   C_LT_W(temp2,temp1,0)

//             psc = ps3c;
//             alpha = alp;
   PICK_PH(psc,  ps3c,     psc)
   PICK_PH(alpha,alp,      alpha)
   PICK_PH(ip,  ip_const,ip)
   ADDS_PH(ip_const,ip_const,two)
   ADDS_PH(ip_const,ip_const,two) /* ready for next time */
...
```

2.2.3 Example: pred_lt_3

This routine is more typical of the other routines in G.729. We will demonstrate how the MIPS32+UDIs are used to speedup the code in this subroutine. Table 2.5 shows the original C implementation. This uses the ITU arithmetic macros such as *negate()*, *add()*, *sub()*, *L_mac()* and *round()*. These ITU macros are like their normal C counterparts, except that they do saturating arithmetic. The function *round()* converts a Q31 value to Q15 value.

Table 2.5 Original C code for pred_lt_3

```
void Pred_lt_3(
    Word16   exc[],       /* in/out: excitation buffer */
    Word16   T0,          /* input : integer pitch lag */
    Word16   frac,        /* input : fraction of lag   */
    Word16   L_subfr)     /* input : subframe size     */
{
    Word16   i, j, k;
    Word16   *x0, *x1, *x2;
    const Word16 *c1, *c2;
    Word32  s;

    x0 = &exc[-T0];

    frac = negate(frac);
    if (frac < 0) {
        frac = add(frac, UP_SAMP);
        x0--;
    }

    for (j=0; j<L_subfr; j++) {
        x1 = x0++;
        x2 = x0;
        c1 = &inter_3l[frac];
        c2 = &inter_3l[sub(UP_SAMP,frac)];

        s = 0;
        for(i=0, k=0; i< L_INTER10; i++, k+=UP_SAMP) {
            s = L_mac(s, x1[-i], c1[k]);
            s = L_mac(s, x2[i],  c2[k]);
        }
        exc[j] = round(s);
    }
    return;
}
```

2.2.3.1 Note on the optimization of pred_lt_3

The optimized version of the code is shown in Table 2.6. The inner loop over i which runs from 0 to 9 adds the dot product of c1 with x0[-9 .. 0] and the dot product of c2 with x0[1 .. 10], or equivalently, the dot product of the combined c1||c2 array with x0[-9 .. 10], which contains 20 terms. Because the 20 terms of c1||c2 are constant, these values are stored in 10 registers as PH values. To efficiently compute the dot-product, we define the DPAQS (DP--dot product, A--accumulate, Q--fractional, S--saturating) instruction (see Table 3.2 on page 22).

The outer loop over j is unrolled twice, such that exc[j] and exc[j+1] are calculated on each pass. The loop unroll saves 5 cycles due to loop overhead and allows one word store to substitute 2 halfword stores. The MIPS32 instructions LWL (load word left) and LWR (load word right) require an extra cycle before the result can be consumed. By using extra registers and interleaving the LWL and LWR instructions, we can fill the load delay slots so that there are no stalls. In addition, we can interleave a LWL instruction between the DPAQS instruction to reduce the repeat rate of the DPAQS instruction to every other cycle.

Table 2.6 Optimized MIPS32+UDI Assembly Code for pred_lt_3

```
#define exc   $4
#define T0    $5
#define frac  $6

#define x0    $7

#define temp1 $14
#define temp2 $15
#define x0_end $2

# define c_0    $8
# define c_1    $9
# define c_2   $10
# define c_3   $11
# define c_4   $12
# define c_5   $13
# define c_6   T0  /* reused */
# define c_7   frac /* reused */
# define c_8   $3
# define c_9   $1

#define L_subfr 40
#define UP_SAMP 3
#define INTER10 10

   sll x0,T0,1   /* short alignment */
   sub x0,exc,x0

   sub frac,$0,frac
   bgez frac,1f
   nop
   addi frac,frac,UP_SAMP
   addi x0,x0,-2
1:
/* the inter_31 array has been transmogrified into an easier to load
 * array, called inter_31_voudi, which is indexed by frac*20 */

   la temp2, inter_31_voudi
   li temp1,20*2
// mul frac,frac,temp1
   mult frac,temp1
   addi x0_end,x0,40*2
   mflo frac
   add temp1,temp2,frac
```

```
   lw c_0, 0(temp1)
   lw c_1, 4(temp1)
   lw c_2, 8(temp1)
   lw c_3,12(temp1)
   lw c_4,16(temp1)
   lw c_5,20(temp1)
   lw c_6,24(temp1)
   lw c_7,28(temp1)
   lw c_8,32(temp1)
   lw c_9,36(temp1)

2:
   MTU($0,VoUDI_Acc1)
   MTU($0,VoUDI_Acc0)


/* unroll #0 */
   lwl temp1,-9*2  (x0)
   lwl temp2,-7*2  (x0)
   lwr temp1,-9*2+3(x0)
   lwr temp2,-7*2+3(x0)
   DPAQS_PH(VoUDI_Acc1,temp1,c_0)
   lwl temp1,-5*2  (x0)
   DPAQS_PH(VoUDI_Acc1,temp2,c_1)
   lwl temp2,-3*2  (x0)
   lwr temp1,-5*2+3(x0)
   lwr temp2,-3*2+3(x0)
   DPAQS_PH(VoUDI_Acc1,temp1,c_2)
   lwl temp1,-1*2  (x0)
   DPAQS_PH(VoUDI_Acc1,temp2,c_3)
   lwl temp2, 1*2  (x0)
   lwr temp1,-1*2+3(x0)
   lwr temp2, 1*2+3(x0)
   DPAQS_PH(VoUDI_Acc1,temp1,c_4)
   lwl temp1, 3*2  (x0)
   DPAQS_PH(VoUDI_Acc1,temp2,c_5)
   lwl temp2, 5*2  (x0)
   lwr temp1, 3*2+3(x0)
   lwr temp2, 5*2+3(x0)
   DPAQS_PH(VoUDI_Acc1,temp1,c_6)
   lwl temp1, 7*2  (x0)
   DPAQS_PH(VoUDI_Acc1,temp2,c_7)
   lwl temp2, 9*2  (x0)
   lwr temp1, 7*2+3(x0)
   lwr temp2, 9*2+3(x0)
   DPAQS_PH(VoUDI_Acc1,temp1,c_8)

/* unroll #1 */
   lwl temp1,-9*2  +2(x0)
   DPAQS_PH(VoUDI_Acc1,temp2,c_9)
   lwl temp2,-7*2  +2(x0)
   lwr temp1,-9*2+3+2(x0)
   lwr temp2,-7*2+3+2(x0)
   DPAQS_PH(VoUDI_Acc0,temp1,c_0)
   lwl temp1,-5*2  +2(x0)
   DPAQS_PH(VoUDI_Acc0,temp2,c_1)
   lwl temp2,-3*2  +2(x0)
   lwr temp1,-5*2+3+2(x0)
   lwr temp2,-3*2+3+2(x0)
   DPAQS_PH(VoUDI_Acc0,temp1,c_2)
   lwl temp1,-1*2  +2(x0)
   DPAQS_PH(VoUDI_Acc0,temp2,c_3)
   lwl temp2, 1*2  +2(x0)
   lwr temp1,-1*2+3+2(x0)
   lwr temp2, 1*2+3+2(x0)
   DPAQS_PH(VoUDI_Acc0,temp1,c_4)
   lwl temp1, 3*2  +2(x0)
   DPAQS_PH(VoUDI_Acc0,temp2,c_5)
   lwl temp2, 5*2  +2(x0)
   lwr temp1, 3*2+3+2(x0)
   lwr temp2, 5*2+3+2(x0)
   DPAQS_PH(VoUDI_Acc0,temp1,c_6)
   lwl temp1, 7*2  +2(x0)
   DPAQS_PH(VoUDI_Acc0,temp2,c_7)
   lwl temp2, 9*2  +2(x0)
   lwr temp1, 7*2+3+2(x0)
   lwr temp2, 9*2+3+2(x0)
   DPAQS_PH(VoUDI_Acc0,temp1,c_8)
   addi x0,x0,4
   DPAQS_PH(VoUDI_Acc0,temp2,c_9)

   MFUS(temp2,VoUDI_Acc1,0)
   MFUS(temp1,VoUDI_Acc0,0)
   CVTQR_PH_W(temp1,temp2,temp1)
   sw temp1,0(exc)

   bne x0,x0_end,2b
   addi exc,exc,4

   jr $ra
   nop
```

2.3 G.729A

We have not optimized the G.729A algorithm, but based on optimizations done by others, we estimate that the G.729A can be done in half the cycles of the G.729.

The best StarCore SC140 implementation shows a ratio of G729A cycles to G729 cycles of 4.7:8.4 = 56% [2] [3]. We will use this value to estimate G.729A results from our study of G.729.

Chapter 3

Proposed User Defined Instructions for Voice Applications

3.1 Data Types

The data types used for speech processing are primarily 16 bit values, with 32 bit values used as intermediates. The encoding of values are either signed integers (-32768 to +32767), fixed point fractions (-1.0000 to +0.9999) or encoded in bit fields (G.729 output frame). These data types are typically supported by DSP hardware, but fixed point fractions are usually not handled by general purpose microprocessors. In addition, DSP hardware offers the ability to saturate a result if an overflow or underflow occurs due to an arithmetic operation, while general purpose microprocessors usually return results where the most significant bits are lost, and the remnant is leftover noise from the lower bits.

Fixed point fractions are notated using the Q format, where the number of bits used for the mantissa is given after the letter Q. In G.729, all operations are done using Q15 (a 16 bit value with one sign bit and 15 bits of mantissa, show in Figure 3.1) or Q31 (a 32 bit value with one sign bit and 31 bits of mantissa, shown in Figure 3.2). Arithmetically, they are identical to plain integer values, meaning that addition, subtraction and multiplication are done using the same algorithm, except that in multiplication, the result is renormalized to fit the destination register. For example, a Q15 value times a Q15 value results in 30 bits of mantissa. To store this value in a Q31 register, the result is shifted to the left by one and an extra 0 is padded on the right. If the result is to be stored in a Q15 register, the result is shifted to the right by 15 (30-15 = 15).

Figure 3.1 GPR in PH Data Format with Q15 Data Type

31	30 16	15	14 0
sign	mantissa	sign	mantissa

Figure 3.2 GPR in W Data Format with Q31 Data Type

31	30 0
sign	mantissa

3.2 Data Formats

The Voice UDI introduces two new data formats in the GPR register set.

For MIPS32, the *pair half (PH)* and single *word (W)* are used.

- In *pair half* format, a 32-bit GPR is interpreted as a vector of two ("paired") signed 16-bit integers. See Figure 3.3.
- In single *word* format, a 32-bit GPR is interpreted as a single signed 32-bit integer. See Figure 3.4.

Positions within a vector are denoted as v[i], where i=0 for the least significant position in the vector register. For instructions that refer to the *PHH* data format, this refers to the "high" vector element in the *PH* format, that is, v[1]. Similarly, *PHL* refers to the "low" vector element in *PH*, that is, v[0].

Figure 3.3 A MIPS32 GPR in PH Data Format

31 16	15 0
v[1]	v[0]

Figure 3.4 A MIPS32 GPR in W Data Format

31 0
v[0]

3.3 CorExtend Registers

The CorExtend block uses six internal registers, four accumulators that are 64 bits each, a vector condition code register (VC), and an overflow indicator register (OF). These are listed in Table 3.1.

Table 3.1 Registers in the Voice UDI block

Register Number	Register Name	Register Size in bits	Description	Instructions that can write to the register	Instructions that can read the register
0	Acc0	64	Accumulator 0	DPAQS, DPSQS, MAQS, MTU	DPAQS, DPSQS, MAQS, MFU
1	Acc1	64	Accumulator 1	DPAQS, DPSQS, MAQS, MTU	DPAQS, DPSQS, MAQS, MFU
2	Acc2	64	Accumulator 2	DPAQS, DPSQS, MAQS, MTU	DPAQS, DPSQS, MAQS, MFU
3	Acc3	64	Accumulator 3	DPAQS, DPSQS, MAQS, MTU	DPAQS, DPSQS, MAQS, MFU
4	VC	2	Vector condition code register	C.cond.fmt, MTU	PICK, MFU
5	OF	1	Overflow indicator register	All instructions that saturate, MTU	MFU

3.4 Instruction Description

Table 3.2 provides a listing of all the instructions used by the G.729 codec. The instructions are categorized into their types, for example, Arithmetic, Shift, etc. The instruction mnemonic is shown along with a brief description of its functionality. The table indicates which specific ITU macro uses that instruction, if applicable. Overall, the G.729 uses nineteen instructions. The relative importance of these instructions during execution, and the criteria that can be used to implement a smaller number is discussed in the next chapter (see Section 4.2.1, "Relative Importance of UDI instructions" on page 28).

Table 3.2 Instructions in the VoIP CorExtend Block

#	Type	Instruction	Description	Used in ITU Code/Macro
1	Arithmetic	ADDS.PH rd, rs, rt	Integer vector saturating addition.	add,L_add
2		CRLS.PH rd, rs	Count leading sign bits	norm_s, norm_l
3		DPAQS.PH ud, rs, rt	Fractional vector saturating dot-product with accumulate.	L_mac
4		DPSQS.PH ud, rs, rt	Fractional vector saturating dot-product with subtract.	
5		MAQS.PH rs, rt	Fractional vector saturating multiplies accumulating to separate accumulators	(code-book)
6		MULQ.PH rd, rs, rt	Fractional vector multiply with same-size products.	mult, L_mult
7 8		MULQ.W.PHH rd, rs, rt MULQ.W.PHL rd, rs, rt	Fractional vector multiply with full-size products.	mult, L_mult
9		SUBS.PH rd, rs, rt	Integer vector saturating subtraction.	sub, L_sub
10	Shifts	SLAS.W rd, rs, shift	Vector saturating arithmetic left shift.	shl, L_shl
11		SLAVS.W rd, rs, rt	Vector saturating arithmetic variable left shift	
12		SRA.W rd, rs, shift	Vector right shift.	shr, L_shr
13		SRAV.W rd, rs, rt	Vector variable right shift.	
14	Compare	C.LT.PH rs, rt	Condition vector compare instructions.	
15	Register Move	MFU rd, us, shift, sat	Copy a UDI register value to a GPR, with optional 32bit saturation.	
16		MTU ud, rs	Copy a GPR value to an UDI register.	
17		PICK.PH rd, rs, rt	Selectively pick vector elements from two registers.	
18	Format Conversion	CVTQR.PH.W rd, rs, rt	Fractional vector conversion to reduced precision with rounding.	round
19		CVTS.PH.W rd, rs, rt	Integer vector saturating conversion to reduced precision.	extract_l

All the UDIs in Table 3.2 are encoded into two UDI opcodes, UDI0 and UDI1. The encoding of the UDI instructions in Table 3.2 into UDI0 (with opcode bits [3:0] = 0000) is shown in Table 3.3, and the encoding into UDI1 (with opcode bits [3:0] = 0001) is shown in Table 3.4.

Note that there is a lot of flexibility in encoding these instructions, they could all have been put in a single UDI opcode if desired. We chose this encoding so that UDI0 encodes all the instructions that use only a subset of registers *rs*, *rt*, and *rd* of the GPR and no UDI registers or immediate values. (The exception are the SLAVS and the SRAV instructions, which were grouped with the other shift operations in the UDI1 encoding).

3.5 Instruction Bit Encoding

Figure 3.5 shows the basic UDI format.

Figure 3.5 Basic UDI Format

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs (optional)	rt (optional)	rd/ud/imm	op xxxxx	UDI opcode 01xxxx
6	5	5	5	5	6

Note that bits 0 to 3 are used to encode the UDI, and the other bits from 6 to 25 are free for the UDI block to use in any way that it likes. Note that however, if GPR source register values are required, then they must be specified in the fields denoted *rs* and *rt*. That is, the main processor core always supplies the source values of the registers that are encoded in bits 21-25 and in bits 16-20. Hence, a random use of these bits might lead to a register dependency detection and pipeline stall (which is a false dependency that does not really exist). Hence we have attempted an encoding of the instructions that is aware of this issue. If the instruction also specifies the destination register, then this is done in the field denoted *rd*.

Table 3.3 UDI Encoding of the op field for UDI0 (bits [3:0] = 0000)

op		*bits 8..6*							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00	ADDS.PH	-	-	-	-	-	-	-
1	01	ΣΥΒΣ.ΠΗ	–	-	-	CRLS.PH	-	-	-
2	10	-	-	-	-	-	-	–	-
3	11	MULQ.PH	-	MULQ.W.PHH	MULQ.W.PHL	-	-	ΧςΤΘ.ΠΗ.Ω	ΧςΤΘΡ.ΠΗ.Ω

Table 3.4 UDI Encoding of the op field for UDI1 (bits [3:0] = 0001)

op		*bits 8..6*							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00	-	SLAS.W	-	SLAVS.W	-	-	-	-
1	01	–	ΣΡΑ.Ω	-	SRAV.W	PICK.PH	-	MFU	MTU
2	10	C.LT.PH	-	-	-	-	-	-	-
3	11	DPAQS.PH	DPSQS.PH	MAQS.PH	-	-	-	-	-

3.6 Estimated Gate Counts

Table 3.5 shows a gate count estimate for a MIPS32 implementation of G.729 on a MIPS32 processor core, 4KEc™. Note that the 4KEc is about 180 K gates (assuming 16 K 4-way caches, no RAMS, MIPS16e™, memBIST, EJTAG TAP, COP2 interface, and no PDtrace™ in TSMC 0.18g process). The proposed UDI block adds about 23% to the total core area. Note that the multiplier is the largest component of the UDI block.

Table 3.5 Gate Count of the Proposed UDI Block

Unit	Gates/Unit	# Units	Net	% of 180 K (core size without CorExtend)
16x16 Multiplier	15 K	2 (1)	30 K (15 K)	
16+16 Adders	1.5 K	2	3 K	
Accumulator Registers	0.5 K	2	1 K	
Shifters	0.5 K	2	1 K	
Overhead	7 K	1	7 K	
Total			42 K (27 K)	23% (15%)

If size is a critical factor, then only one multiplier can be implemented, with a little performance penalty. To execute the SIMD multiply-accumulate or the dot-product instructions, the single 16x16 multiplier would have to be double-pumped. The multiplier is assumed to be fully pipelined, so that when it is double-pumped for the two different multiplies of the 2-way SIMD instruction, the repeat-rate of the instruction is two. That is, the dot-product or the multiply instruction can be issued only every other cycle. This is typically not a scheduling problem in the code since at least one (load) instruction is available to be inserted between two dot-product instructions. We estimate a maximum of 10% performance penalty with only one multiplier. The total gate count with one multiplier would be 27K which is about 15% of the total core size.

The assumed latency and repeat rates of the UDI instructions is shown in Table 3.6.

Table 3.6 UDI Latency and Repeat Rates

	2 multipliers (1 multiplier)	
Instruction	Latency	Repeat Rate
ADDS, CRLS, SUBS	1	1
MULQ	2 (3)	1 (2)
DPAQS, DPSQS, MAQS	3 (4)	1 (2)
CVTQR, CVTQ	1	1
C.LT	1	1
SLAS, SLAVS, SRA, SRAV	1	1
MFU, MTU, PICK	1	1

3.7 The Feasibility of a CorExtend Block for General VoIP Processing

Since G.729 is only one of the many VoIP standards, we examine some of the other applications to determine how they might impact the number of instructions needed and the size of the UDI block. Other common (and popular) codecs that are used are G.711, G.726, and G.723.1.

G.711 comprises a A-law and µ-law encoder and decoder. The G.711 application is not very computation intensive. To process 64 K samples takes about 0.5 MHz of the processor. And the entire encoding and decoding can be done using a single instruction or table look-up function, if needed. Hence, this codec is not discussed any further.

The G.726 codec is based on ADPCM (Adaptive Differential Pulse Code Modulation) and is of medium complexity. The processing in the adaptive quantizer and the inverse adaptive quantizer involves log2() and antilog() operations. The VoIP UDI does not need a LOG2 instruction since the ITU standard defines a log2() approximation. The bulk of the processing in this codec is in computing the signal estimate in the adaptive predictor which uses floating point in the predictor filters. Thus, it is necessary to manage exponents and mantissas efficiently. These are performed effectively using the CRLS instruction to compute the exponents, and the SLAVS instruction to compute the mantissas using normalization operations.

The G.723.1 is a dual-rate codec with the lower bit rate based on CELP (Code Excited Linear Predictive), like the G.729 codec. The ITU source code for G.729 and G.723.1 share a common set of basic signal processing operations. These are the primitive 16-bit and 32-bit arithmetic operations in the basops.c file. The set of instructions defined for G.729 cover these basic ops well, hence G.723.1 can also be efficiently performed using the same instructions. The only instructions that would be needed in addition are a vector MIN/MAX function.

Other than the codecs, line echo cancellation is a key component of VoIP processing. The basic computation here is the LMS (Least Mean Squared) FIR Filter. The main operations in the LMS filter consist of convolution (FIR filter), followed by updating the filter coefficients. The convolution is essentially a dot product (which is covered by our DPAQS instruction), and updating the filter state (shifting the data in the delay line). The filter state update can be emulated using a CIRADD (circular add) instruction that takes a buffer base and increments the pointer through the buffer. Such an instruction has been shown in the application note that addresses the user defined instructions needed for DSP filter processing [6].

Hence, to cover the full space of VoIP processing, the G.729 CorExtend block would need three additional instructions, the MIN, MAX, and CIRADD. These instructions do not impact the gate count significantly (a 1-2% increase approximately).

Chapter 4

Performance of G.729

4.1 Methodology

The MIPSsim™ simulator was used to evaluate the performance impact of the proposed instructions on the G.729 application code. The simulator can model and simulate all processor cores available from MIPS Technologies, and can estimate cycle counts fairly accurately. For the 4KEc core, the simulator has been verified to be within 5% of the performance of the hardware. The simulator provides a CorExtend UDI interface that allows a user to describe the execution of the proposed instructions in C. This C library is then bolted onto the simulator which sends all UDI instructions across the interface to be emulated by the user-written C library code. The simulator interfaces with the GreenHills MULTI debugging environment to allow easy debugging as well as performance evaluation. The performance results provided in this chapter have been obtained using the methodology just described.

4.2 G.729 Performance

The values in Table 4.1 represent the cycle counts for some selected, important routines in G.729. Note that these cycle counts are approximate values, and in some cases vary dramatically with the speech data being processed. The Fixed Codebook Search (D4i40_17) in particular can vary significantly with the type of data being processed. Likewise, the frame totals are approximate values, based on 10 frames of algthm.in test data. The values given here are fairly typical, and hence used for reference purposes and as a basis for further performance analysis. The input voice vector used was obtained from the ITU website [1].

Since a full implementation of the G.729 encoder using MIPS64® ([5]) and 64-bit UDIs (not listed in this document) has been completed, we show this performance data. Some of the key routines have been rewritten in MIPS32+UDI (described in Table 3.2 on page 22). The speedups of these routines have been used to extrapolate results to other routines that have not yet been rewritten in MIPS32+UDI. This is possible since the type of computation in each routine is well understood. Hence, the estimated number in the last row and last column of Table 4.1 is judged to be fairly accurate.

As the reader will note, the speed ups of individual routines are considerable (ranging from about 3.7 to over 20). The speed ups are due to several factors, listed here in decreasing order of importance:

1. Saturation is done by the arithmetic operations defined by the UDI, rather than using several comparison and branch instructions.

1. The reference code for some instructions was written for portability, not efficiency. (This was particularly true for bit shifting operations.) These were rewritten more efficiently for the MIPS architecture.

2. The 2-way SIMD instructions allow the use of 32 bit registers to store two values, and compute in parallel on two values.

3. Critical routines were hand-coded to maximize use of registers and for efficient code scheduling.

4. Using UDIs often reduced the code size, which had an overall beneficial effect on the rate of instruction cache misses.

The final result was an application that can encode G729 speech data using 24.2 MHz of the bandwidth of a MIPS64 processor.

Table 4.1 Cycle Counts for Selected G.729 Subroutines

Subroutine	Compiler Optimized C implementation (MIPSsim cycle counts)	MIPS64+UDI (MIPSsim cycle counts)	Speedup of C to MIPS64+UDI	MIPS32+UDI (MIPSsim cycle counts)	Speedup of C to MIPS32+UDI
D4i40_17 [1]	172676	47570	3.6	101826	1.7
Lag_max	106756	6446	16.5		
Norm_corr	113653	8982	12.7		
Syn_Filt	10133	1163	8.7		
Lsp_pre_select	37343	1712	21.8		
Convolve	19027	1456	13	3019	6.3
Residu_rev	10994	544	20.2	950	11.6
Autocorr	58736	4037	14.5		
Cor_h_X	19039	1102	17.3		
Pred_lt_3	17099	798	21.4	1529	11.2
Lsp_select_1,2	7585	474	16		
Pre_Process	13101	2125	6.2		
Az_lsp	57578	15398	3.7		
One Frame Total [2]	1220600	242419	5	[3]420000±10 %	2.9

[1] highly dependent on frame data, this is the 4th frame of algthm.in

[2] approximate, data dependent

[3] estimated, based on MIPS64 values and partial conversion of the code into MIPS32 and overall impression of vectorization of total algorithm

To predict the application performance on a 32-bit machine (such as the 4KEc), a few key routines were rewritten using the MIPS32+UDI instruction set. By cutting the register size in half (and the SIMD data size in half), we expect that the MIPS32 implementation would require twice as many operations to get the same results as the MIPS64 implementation. Indeed, we see that the cycle counts of the critical loops increase by about a factor of 2 (1.75-2.14 in the examples of the table). We estimate that about 75% of the application will suffer from this degradation, giving a net cycle increase of about 1.75x, or about 42 MHz.

The decoding process is considerably faster than the encoding process. We have estimated that the decode requires about 20% of the cycles of the encoding process. This results in about 30 MHz for a full duplex codec running on a MIPS64 processor implementation, and about 50 MHz for a full duplex codec running on a (MIPS32) 4KEc implementation.

Figure 4.2 shows the summary of the megacycles requirement for the encoder and the decoder and also the estimated megacycles for the G.729A annex to the G.729 scheme. (See Section 2.3, "G.729A" on page 19 for a description of how the G.729A numbers were obtained). We can estimate expected full-duplex performance by comparing the optimized C-implementations of the encoder versus decoder to get the ratio of computational complexity, and then extrapolating the UDI optimized decoder speed from the UDI optimized encoder speed. We find that the C-implementation decoder requires 22.7% of the cycles of the encoder.

We see from the results that using say a 240 MHZ 4KEc processor core, and using the G.729A compression scheme, which is one of the most commonly used ones in the VoIP area, one can easily implement four channels of voice processing. Each channel takes approximately 27 MHZ of the cpu and hence a total of 108 MHZ or about 45% of the cpu can be used for voice processing. The rest is available for operating systems and other control functions.

Table 4.2 Summary Table for G729 and G729A.

Algorithm	Direction	CPU Usage (Megacycles per Second)	
		MIPS32+UDI	MIPS64+UDI
G.729	Encode	42	24
	Decode	9.5	5.5
	Total (Full Duplex)	52	30
G.729A	Encode	24	13
	Decode	2.8	3.1
	Total (Full Duplex)	27	16

4.2.1 Relative Importance of UDI instructions

Figure 4.3 shows the usage of each UDI instruction. These counts were done for a few frames of speech input. In addition, the equivalent cycles to emulate each instruction is given for comparison. Finally, the net degradation to the overall performance is estimated by multiplying the equivalent cycles with the count. The final column is the accumulated degradation, starting from the bottom of the list. The Move From and To UDI register instructions (MFU and MTU) can not be emulated, so they can not be assigned a hypothetical degradation. The accumulated degradation is roughly 400% or 5x, which is in line with the overall performance improvement of 24 MHz (for MIPS64+UDI G.729 Encode) versus 122 MHz.

Table 4.3 Profile of MIPS64+UDI Instructions in G.729 Encoder (Sorted by decreasing importance)

UDI Instruction (MIPS64 version)	Count	Percent	Approximate Equivalent Cycles	Degradation if not implemented	Accumulated Degradation (added up from last row of the table)
MTU	36368	11.5%	N/A	N/A	N/A
MFUS	8000	2.5%	N/A	N/A	N/A
DPAQS_QH	79700	25.1%	54	162.47%	376.19%
MULQ_QH	15302	4.8%	52	30.02%	213.73%
MAQS_QH	18928	6.0%	40	28.39%	183.71%
DPSQS_QH	14400	4.5%	40	21.60%	155.32%
PICK_QH	14276	4.5%	40	21.41%	133.72%
ADDS_QH	10838	3.4%	40	16.26%	112.30%
SUBS_QH	10320	3.3%	40	15.48%	96.05%
MULQ_PW_QHL	12644	4.0%	26	12.16%	80.57%
MULQ_PW_QHH	12604	4.0%	26	12.12%	68.41%
C_LT_PW	11944	3.8%	20	8.73%	56.29%
SLAVS_PW	10881	3.4%	20	7.95%	47.56%
SRA_PW	10264	3.2%	20	7.50%	39.61%
SLAS_QH	4732	1.5%	40	7.10%	32.11%
SLAS_PW	7600	2.4%	20	5.55%	25.01%
CVTQR_QH_PW	7200	2.3%	20	5.26%	19.46%
CVTS_QH_PW	5262	1.7%	20	3.85%	14.20%
PICK_PW	4340	1.4%	20	3.17%	10.35%
CVTQ_QH_PW	3500	1.1%	20	2.56%	7.18%
ADDS_PW	2080	0.7%	20	1.52%	4.62%
MULQR_QH	600	0.2%	60	1.36%	3.10%
MFAS_PW_2D	14184	4.5%	3	1.09%	1.74%
SRAV_PW	400	0.1%	20	0.29%	0.65%
ABS_PW	400	0.1%	17	0.25%	0.36%
CRLS_QH	100	0	24	0.09%	0.11%
SUBS_PW	20	0	20	0.01%	0.02%
CRLS_PW	20	0	12	0.01%	0.01%

Note that since the accumulated performance degradation for the last nine instructions (from CVTQ_QH_PW to CRLS_PW) is only about 7.18%, they were omitted from the list of user defined instructions shown in Table 3.2 on page 22. Many of these nine instructions can be implemented if needed without any significant hardware penalty since the required hardware already exists (for other instructions).

The most significant use of this table lies in the fact that it can be used to determine the top 10, or 5, or even 2 instructions needed to accelerate G.729. And the corresponding performance can be easily estimated from the given data. For example, if one wants to implement only 4 instructions, then implementing DPAQS, MULQ, MAQS, and DPSQS would result in a 133% degradation from the current performance level, which implies 56 MHZ to do encoding in MIPS64, which might be an acceptable performance level for only 4 instructions. (Note that this corresponds to a performance improvement of 2.2x over the no UDI case.)

4.3 Summary

We have shown that using User Defined Instructions on a MIPS32 Pro Series Core improves the performance of G.729 by a factor of 3. The G.729 codec consumes about 52 MHz of a MIPS32 Pro Series processor, and G.729A is about half that value, i.e., 26 MHz. At this performance level, a 4KEc core can easily support VoIP functions such as video conferencing as well as other general OS and system functions.

We have also shown that the gate count of a UDI block for VoIP is reasonably small and is estimated to be about 23%, with two multipliers, or 15%, with one multiplier, of the total gate count for a 4KEc core. The performance penalty for one multiplier (over two) is only 10%. This type of cost/performance trade-off needs to be made on a per-user basis, and knowing all the other constraints on the system.

We have also shown that the set of instructions designed for G.729 can be augmented by only three more instructions to cover the range of applications needed for VoIP. This includes other codecs such as G.726 and G.723.1, as well as echo cancellation.

The disadvantage of using new state such as the accumulator registers in the CorExtend block is that it makes the application code non-reentrant. Since there is no operating systems support to save and restore this extra architectural state, there can only be one instance of the application active at any given time in the processor. This restriction can be avoided if only new instructions are implemented using CorExtend without the addition of any new state.

Instead of using this feature to boost performance, the speedup obtained using these additional instructions can be used to save power and lower the core clock frequency. For instance, by lowering the operating frequency by only 10%, power consumption can be reduced by about 20%. This reduction in core frequency has a corollary effect in that it allows the whole core to be synthesized more for area/power and less for clock speed. This allows a saving in silicon area that would offset the area increase for the CorExtend block.

4.4 References

[1] International Telecommunications Union - Telecom Division, http://www.itu.int/ITU-T/

[2] ITU-T G.729A Implementation on StarCore SC240, AN2151/D, Motorola Application Note

[3] ITU-T G.729 Implementation on StarCore SC140, AN2094/D, Motorola Application Note

[4] MIPS3® Architecture for Programmers, Volumes I and II

[5] MIPS64® Architecture for Programmers, Volumes I and II

[6] Accelerating DSP Filter Loops with CorExtend® in MIPS32® Pro Series® Cores (MD00303)

Appendix A

Revision History

Change bars (vertical lines) in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that are more than one revision old.

This document may refer to Architecture specifications (for example, instruction set descriptions and EJTAG register definitions), and change bars in these sections indicate changes since the previous version of the relevant Architecture document.

Revision	Date	Author	Description
00.10	October 18, 2002	H/RT	First draft of application note
00.20	January 20, 2003	RT	Additional editing of document
01.01	June 30, 2008	EF	Update document title