Freescale Semiconductor
Application Note

AN2094
Rev. 1, 12/2004

ITU-T G.729 Implementation on the StarCore™ SC140/SC1400 Cores

By Bogdan Costinescu, Razvan Ungureanu, Madalin Stoica, Emilian Medve, Radu Preda, Mugur Alexiu, and Costel Ilas

This application note illustrates the process of optimizing the C source code of applications written for Freescale Semiconductor DSPs based on the StarCore™ SC140/SC1400 cores while ensuring that the bit-level output of the modified software is identical to that of the original application (bit-exactness). The optimization steps and the results for this application are discussed in detail. The application chosen for this purpose is the vocoder defined by the ITU-T G.729 Recommendation [1]' but the principles illustrated can be applied to the C source code for any application.

StarCore SC140 features give powerful support and allow advanced implementations of DSP algorithms. Characteristics of the StarCore SC140/SC1400 cores include:

- Integer and fractional 16-bit data types supported in hardware
- Instruction parallelism—up to six instructions per cycle (four DALU and two AGU)
- Suitable architecture for a high-performance compiler
- Efficient support for double-precision arithmetic
- Single-cycle operation for almost all instructions, including MAC
- Zero-overhead hardware loops with up to four levels of nesting
- Modulo addressing for circular buffers—delay lines, sample buffers
- Multiple operands support via MOVE in single instructions—up to 64 bytes for aligned data

CONTENTS

- Stack optimization for improved multi-tasking
- Peak performance of 1200 DSP-MIPS at 300 MHz

This application note is written for SC140 programmers, system engineers, tool developers, and project managers.

1 G.729 Recommendation for Speech Compression

Speech compression technology is widely used in digital communication systems such as wireless systems, VoIP, and video conference technology. Speech compression reduces data redundancy and thus eases bandwidth requirements. The compression technique described in the ITU-T G.729 Recommendation is commonly employed in speech transmission systems because of the quality of the reconstructed speech signal.

1.1 Assessing Speech Quality

The Mean Opinion Score (MOS) is a commonly used test to assess speech quality. In this test, listeners rate a coded phrase based on a fixed scale [2, 3, and 4]. A MOS of four or higher is considered 'toll' quality, which means that the reconstructed speech is indistinguishable from the original speech. Tests have shown that encoding systems based on G.729 at 8 kbits/s provide toll-quality speech for most operating conditions, as shown in **Table 1** [5, 6, and 7].

Table 1. G.729 MOS Results

Test Conditions		MOS
Clean		4.125
Encoder Background Noise	10 dB MNRU[1]	3.65
	30 dB MNRU	3.975
Channel Errors	1.0% bit error	3.3
	0.1% bit error	3.95
Note: Modulated noise reference unit		

1.2 Technical Overview of ITU-T G.729

The International Telecommunications Union–Telecommunications Standardization Sector (ITU-T) G.729 Recommendation defines an algorithm for coding speech signals at 8 kbit/s using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP). In this system, an analog voice signal is passed through a 300 Hz – 3400 Hz bandpass filter and sampled at 8 kHz to yield digital data that is converted to a 16-bit linear PCM speech signal. An encoder analyzes the speech signal to extract the parameters of the CELP model. These parameters are encoded and transmitted in a bitstream. The decoder for this system uses the received parameters to retrieve the synthesis filter coefficients. The speech is then reconstructed by filtering the excitation codebook as shown in **Figure 1**. The vocoder operates on 10 ms frames with 5 ms look-ahead for linear-prediction (LP) analysis. The overall algorithmic delay is 15 ms.

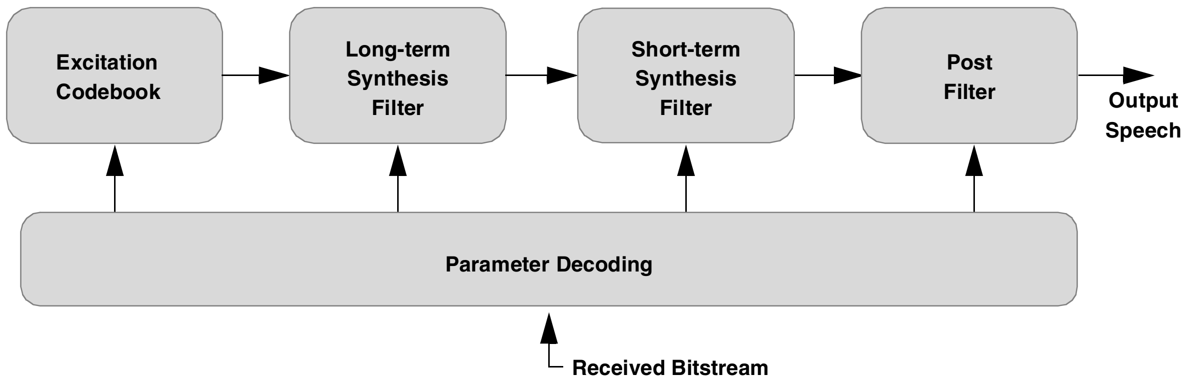

Figure 1. CELP Synthesis Model

1.2.1 Encoding

The G.729 encoding scheme is based on a code-excited linear-prediction model. In this model, the locally decoded signal is compared with the original signal. The filter parameters are then selected to minimize the mean-square weighted error between the original and reconstructed signal. The encoding principle is shown in **Figure 2**.

The input samples are passed through a 140 Hz high-pass filter, a tenth-order LP analysis is performed on the samples, and the resulting LP parameters are quantized in the line spectral pair (LSP) domain with 18 bits. The input frame is divided into two 5 ms subframes to optimize tracking of the pitch and gain parameters and reduce the complexity of the codebook searches. Interpolated LP coefficients are applied to the first subframe, and quantized and unquantized LP filter coefficients are applied to the second subframe. The excitation in each subframe is represented by both an adaptive-codebook contribution, which simulates the pitch structure of the voiced sounds, and a fixed codebook contribution, which simulates unvoiced sounds. The adaptive and fixed codebook parameters are transmitted every subframe.

The adaptive codebook component represents the periodicity in the excitation signal using a fractional pitch lag with 1/3 sample resolution. The adaptive codebook is searched using a two-step procedure. An open-loop pitch lag is estimated per frame based on a perceptually weighted speech signal. The adaptive codebook index and gain are found by a closed-loop search around the open-loop pitch lag. The signal to be matched, referred to as the target signal, is computed by filtering the LP residual through the weighted synthesis filter. The adaptive codebook index is encoded with 8 bits in the first subframe and differentially encoded with 5 bits in the second subframe. The target signal is updated by removing the adaptive codebook contribution, and this new target is used in the fixed codebook search. The fixed codebook is a an algebraic codebook with 17 bits. The gains of the adaptive and fixed codebooks are vector-quantized with 7 bits using a conjugate structure codebook.

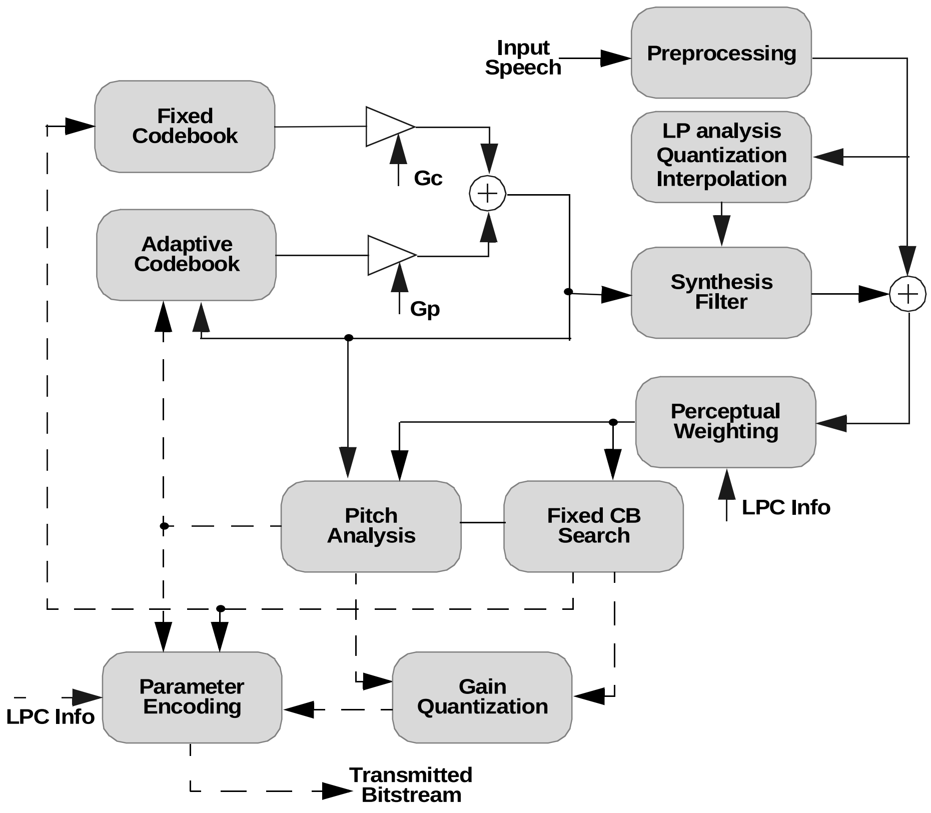

Figure 2. CS-ACELP Encoding

1.2.2 Decoding

The G.729 decoder synthesizes the output speech samples from the received bitstream as show in **Figure 3**.

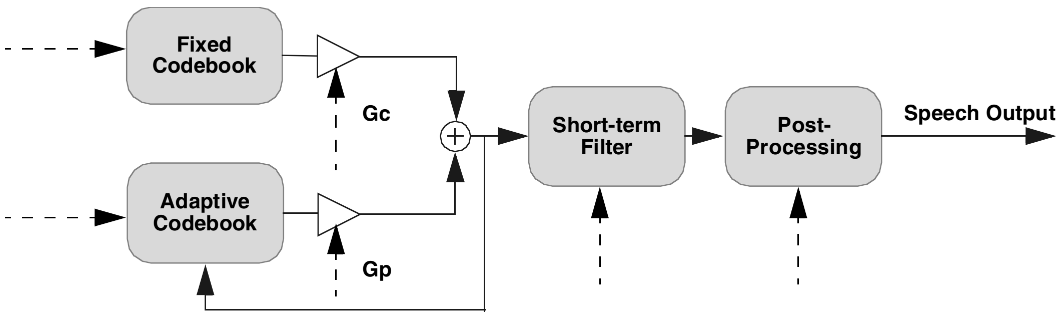

Figure 3. Principle of the CS-ACELP Decoder

First, parameter indices are extracted from the received bitstream and decoded to obtain the parameters corresponding to a 10 ms speech frame. These parameters include:

- LSP coefficients
- Two fractional pitch delays
- Two fixed codebook vectors
- Two sets of adaptive and fixed codebook gains.

The LSP coefficients are interpolated and converted to LP filter coefficients for each subframe. Then, for each 5 ms subframe, the excitation is constructed by adding the adaptive and fixed codebook vectors scaled by their gains. The speech is reconstructed by filtering the excitation through the LP synthesis filter, and the reconstructed speech signal is passed through an adaptive postfilter to enhance speech quality.

2 Optimization Process

The purpose of this section is to provide a description of the software development process used in implementing ITU G.729 Reference code on StarCore SC140. The main steps performed during this implementation process are illustrated in **Table 2** in the order that they are to be carried out. Following this process in the order given achieves high-performance code for StarCore. Additionally, this section provides the typical problems encountered and their possible solutions. Finally, programming techniques are provided to optimize the code for speed.

Table 2. Main Stages of the Software Development Process

Development Stage	Description
Porting to SC140	Data type definitions, introduction of intrinsic functions, multichannel transformations.
Project-Level Optimizations	Inlining, data alignment, StarCore adaptations.
Algorithm Changes	Platform-independent and platform-dependent changes in algorithms.
Function-Level C Optimization	C optimization techniques (multisample, loop unrolling, split summation), better use of intrinsic functions.
Function Implementation in Assembly	Implementation of selected functions in assembly for best optimization.

2.1 Test Vectors and Development Tools

All test vectors provided by the ITU-T were used to determine if the results from the ported code were the same as those obtained from the original G.729 code. These test vectors are summarized in **Table 3**.

Table 3. ITU-T Test Vectors

Encoder Inputs	Encoder Outputs and Decoder Inputs	Decoder Outputs

`algthm.in`
`fixed.in`
`lsp.in`
`pitch.in`
`speech.in`
`tame.in`
	`algthm.bit`
	`erasure.bit`
	`fixed.bit`
	`lsp.bit`
	`overflow.bit`
	`parity.bit`
	`pitch.bit`
	`speech.bit`
	`tame.bit`
		`algthm.pst`
		`erasure.pst`
		`fixed.pst`
		`lsp.pst`
		`overflow.pst`
		`parity.pst`
		`pitch.pst`
		`speech.pst`
		`tame.pst`

The development tools included the StarCore SC140 Enterprise C compiler and Metrowerks® CodeWarrior® for StarCore.

The PC compiler used to check the modifications to the C code and generate the test vectors for unit testing was Microsoft Visual C++ 6.0. The project was developed on the Windows 2000 Professional platform. Hardware tests were run on the SC140 Software Development Platform (SDP).

2.2 Porting G.729 Code to the SC140

This section describes the considerations involved in porting the fixed-point C source code distributed by the ITU-T for the G.729 Recommendation to the SC140.

2.2.1 Initial Compilation

The original C reference code was first compiled using the StarCore Enterprise C compiler with only minor changes, and the resulting code passed all ITU-T test vectors on the software simulator. This verified that the compiler is fully ANSI C compliant.

2.2.2 Initial Modifications

The first set of modifications to the original ITU-T C code was made to replace the DSP emulation instructions with SC140 intrinsic functions and accommodate multi-channel environments.

2.2.2.1 Intrinsic Functions

The DSP emulation instructions in `basic_op.c` were replaced by intrinsic functions of the SC140 Enterprise C compiler. The set of functions defined in the compiler's `prototype.h` file is substantially richer than the set provided in `basic_op.c`, so all emulation functions were easily replaced, with the exception of some minor modifications described below that were required to handle overflow and saturation. In addition, the Word16, Word32, and Flag data types used in the G.729 code were redefined in `typedef.h` to comply with SC140 architecture. For more details, refer to the StarCore *SC100 C Compiler User's Manual* [9], section 3.4.4.

2.2.2.2 Overflow

The DSP emulation instructions in `basic_op.c` simulate two processor flags, overflow and carry. The carry flag is used only inside the emulated operations, so there was no problem in removing it. However, the overflow flag is set in several functions, and tested in other functions, so it must remain. Because the overflow flag was no longer available as a global variable, two assembly routines were developed that allow C code to access the overflow flag. These routines are called `ClearOverflow()` and `GetOverflow()`.

2.2.2.3 Saturation

The G.729 code uses a `sature()` function, which saturates the lower 16 bits of a 32-bit value. This is virtually equivalent to the `sat.w` instruction, which is not implemented in the SC140. The nearest equivalent intrinsic function, `saturate()`, saturates the upper 16 bits, and is translated by the compiler to a `sat.f` instruction. One solution to this problem is to insert an inline assembly function that emulates `sature()`, as shown in **Example 1**.

Example 1. Assembly Emulation of Sature()

```
asll          #<8,d0
sat.l         d0
asll          #<8,d0
sat.f         d0,d0
```

This routine requires four cycles to execute. However, this approach forces the compiler to use a dedicated DALU register. A more acceptable solution is to use the intrinsic functions provided by the C compiler, as shown in **Example 2**.

Example 2. C Emulation of Sature()

```
A= _sat(L_shl(L_shl(L_a, 8), 8));
```

Although the code generated by this statement requires 5 cycles instead of 4 cycles, the compiler uses any DALU register for the operation. This was the solution chosen for the vocoder project.

Note: Beginning with version Beta 1.0 of Metrowerks CodeWarrior for StarCore, the `saturate()` intrinsic function saturates the lower 16 bits of a 32-bit value, just as the `sature()` function from G.729 code does.

2.2.3 Code Modifications for Multi-Channel Environments

To accommodate multi-channel systems, global variables and static data in each C module were eliminated. For each file that contained static or global data, the following steps were taken:

1. A data structure containing the static data variables was defined in the `g729codec.h` file.
2. The generated data structure in `g729codec.h` was added to the general encoder and decoder channel information.
3. Former static data initialization was performed *explicitly* in the encoder/decoder initialization function, `g729_encode_initialize()/g729_decode_initialize()`.
4. For each function in the original file that used static data:
   a. A pointer to the data structure in `g729codec.h` was added to the parameter list.
   b. All static variables referenced in the function body were replaced with variables stored in the data structure whose addresses are received as parameters.

      To improve compiler performance, it is recommended that computations be made on local copies of the ex-global data structures, passing the results back to the channel data.
5. For each function that called any of the functions modified in step 4, a pointer to the data structure in `g729codec.h` was added to its parameter list. This process was repeated recursively up to the main encoder/decoder function, `g729_encode()/g729_decode()`.

Vocoder performance at the end of this phase is listed in **Table 4**.

Table 4. Performance Characteristics After Initial Porting Phase

Speed	Program Size	Tables	Channel Data	Stack Size
29.29 MCPS[1]	37.9 KB	6.43 KB	3.18 KB	3.16 KB
Note: Processing load in millions of cycles per second				

2.3 Project-Level Optimizations

Two optimization techniques, function inlining and data alignment, were applied to the entire project. In general, these techniques improve the performance of both C and assembly code.

2.3.1 Function Inlining

Function inlining (also referred to as ‘inline expansion’ or simply ‘inlining’), is the process of replacing a function call with the body of the function itself. This optimization technique improves execution time by eliminating function-call overhead at the expense of larger code size.

Profiler information is used to determine whether a function should be inlined. Small, frequently-called functions are the best candidates for inlining. However, other factors should also be considered, including the number and type (input or output) of parameters the function passes, how the results are returned, and data alignment. In some cases, the alignment property of a vector parameter is lost when the function is inlined. For example, a procedure that calculates the energy of a signal represented in an array of 80 data samples takes 23 cycles when the all data start addresses are multiples of eight. The same procedure on unaligned data takes 42 cycles.

In general, functions that take fewer than 25 cycles offer the most speed savings by inlining. The overhead to call such a function is typically 20 percent or greater. Functions that take 25–35 cycles and pass several parameters are also worth inlining. Function inlining is done in one of three ways:

- Implicitly, allowing the compiler to select the functions to be inlined. This is done in the Enterprise C compiler by setting the –Og compiler option.
- Explicitly, using the `#pragma inline` C statement [9]. To inline a function in several files, the function should be placed in a header file, and the `static` keyword should be used in each file to prevent the linker from generating duplicate global symbols.
- Manually replacing a function call with the body of the function body. The manual inlining technique is illustrated in Code Example 3, taken from the `Autocorr()` function.

Example 3. Manual Inlining Technique

```
/**** L_Extract is defined as ****/
/*
  void L_Extract(Word32 L_32, Word16 *hi, Word16 *lo)
  {
    *hi  = extract_h(L_32);
    *lo  = extract_l( L_msu( L_shr(L_32, 1) , *hi, 16384));
    return;
  }
*/

/* next four L_Extract() calls were manually replaced with L_Extract() body*/

// L_Extract(sum0, &r_h[i], &r_l[i]);
// L_Extract(sum1, &r_h[i+1], &r_l[i+1]);
// L_Extract(sum2, &r_h[i+2], &r_l[i+2]);
// L_Extract(sum3, &r_h[i+3], &r_l[i+3]);

r_h[i]   = extract_h(sum0);
r_h[i+1] = extract_h(sum1);
r_h[i+2] = extract_h(sum2);
r_h[i+3] = extract_h(sum3);

r_l[i]   = extract_l(L_msu( L_shr(sum0, 1), r_h[i], 16384));
r_l[i+1] = extract_l(L_msu( L_shr(sum1, 1), r_h[i+1], 16384));
r_l[i+2] = extract_l(L_msu( L_shr(sum2, 1), r_h[i+2], 16384));
r_l[i+3] = extract_l(L_msu( L_shr(sum3, 1), r_h[i+3], 16384));
```

Implicit and explicit inlining requires no knowledge of the code and is performed without examining code sections because the compiler automatically tries to interleave consecutive calls. These techniques are recommended when there is only one call to the inlined function.

Manual inlining is recommended when several identical calls are made in sequence, and enables the programmer to fully exploit potential parallelism. By replacing a function call with the C statements of the function, the programmer places in parallel, by hand, the actions in the function by allocating different variables and performing several consecutive identical calls on phases. This technique requires a working knowledge of the code to identify where a small function is repeatedly called and which actions are executed in parallel. Manual inlining is also used in assembly code.

Based on these rules, five functions in the vocoder project were inlined: `Mpy32_16()`, `Mpy_32()`, `Div_32()`, `L_Extract()`, and `L_Comp()`. In most places the functions were inlined explicitly using the `static` and `#pragma inline` C statements. As a result, every call of these functions was replaced by inline code, and the linkage generated no conflicts. In some places the functions were manually inlined.

In most cases, function inlining improves execution speed, and was the major source of speed optimization in this phase of the project, yielding a reduction in required processing power of 4.6 million cycles per second (MCPS).

Typically, function inlining increases the code size. However, inlining actually decreased the code size in this project, primarily because in many cases the code that placed the parameter on the stack could be removed.

2.3.2 Data Alignment

Data flow analysis and previous experience indicated that aligning frequently-accessed data structures would improve performance. Data was aligned on all arrays to facilitate parallel data moves. Most StarCore-specific optimization techniques require the use of multiple move operations.

For function parameters the alignment is declared using the `#pragma align *ptr` statement. In addition, the `assert` statement is used to prevent various problems caused by misaligned data. The use of these statements is illustrated in Code Example 4.

Example 4. Parameter Alignment With Assert

```
void fct(Word16 *x)
{
#pragma align *x 8
  assert(((int) x & 7) == 0);
}
```

The assert statement acts like a fuse by forcing the application to stop when parameters do not conform to special alignment needs. It also generates the code line number where the assertion violation occurred, which streamlines the debugging process.

In G.729, most vectors are composed of two parts—the historical part, which contains values computed in previous steps, and the current part, which contains current values. The two parts are allocated as a single vector that is accessed with two pointers—`old_ptr` points to the first value in the historical part, which is also the start address of the vector, and `new_ptr` (or simply `ptr`) points to the first new value in the array. The second pointer is often referred to as an *internal pointer*. Functions that operate on new values require that the data they use is aligned on the internal pointer. The channel data structures were aligned so that each data structure and each vector in the data structures were aligned on an 8-byte boundary. To align internal pointers, one of the following two techniques was used:

- The relative start address of the vector in the data structure was displaced by padding or rearranging the data.
- Larger vectors were defined containing extra values that shift the data so that the internal pointer is aligned.

When internal pointers are used, conflicts can arise when two different functions require the same array to be aligned differently. Tests should be run to identify the type of alignment that gives better overall performance. When a compromise is difficult, the recommended approach is to make a local copy of the array in one of the functions which contains the required alignment.

2.3.3 32-Bit DPF Format and Operations

Many vocoders including the ITU-T G.729 use a non-standard representation of 32-bit double-precision numbers known as double precision format (DPF). The equation defining this representation is

Equation 1

$$L\_32 = hi<<16 + lo<<1$$

where L_32 is a 32-bit signed integer, and hi and lo are 16-bit signed integers. The range of values for L_32 is

Equation 2

$$\$8000000 <= L\_32 <= \$7ffffffe$$

The fact that the lower as well as the upper portion of the 32-bit value is signed speeds up multiplication operations. The DPF format and the operations based on it are defined in the G.729 `oper_32b.c` file. The principal operations defined for DPF include:

- `Mpy_32()`—multiplication of two 32-bit DPF values
- `Mpy_32_16()`—multiplication of a 32-bit DPF value with a signed 16-bit value
- `Div_32()`—division of two 32-bit DPF values

The other two operations convert a 32-bit value to two 16-bit values and *vice versa*. The 32-bit DPF format was designed for 16-bit processors that do not support 32-bit operations. Thus, although StarCore processors are 32-bit processors that support 32-bit operations, the 32-bit operations had to be implemented in DPF format to maintain bit-exactness with the original ITU-T implementation.

However, there was one optimization that could be performed to take advantage of the processor's 32-bit architecture without corrupting the DPF format. The two 16-bit portions, which were originally processed in two separate DALU registers, were combined into a single 32-bit value using only one DALU register. Thus, functions that originally received two pointers to two 16-bit arrays could now operate with one pointer to a 32-bit array. This optimization step also reduced the number of memory moves required. However, the least significant bit of partial or final computation results had to be reset to maintain bit-exactness with G.729. Again, although C replacement functions were not always as fast as their original counterparts, the benefits of custom assembly implementation and reduced data transfers resulted in an efficiency gain. For details on assembly implementation of 16-bit and 32-bit multiplication, refer to **Appendix B** and the *SC140 DSP Core Reference Manual*.

2.3.4 Results

The vocoder performance characteristics after project-level optimizations are shown in **Table 5**.

Table 5. Performance Characteristics After Project-Level Optimizations

Speed	Program Size	Tables	Channel Data	Stack Size
24.7 MCPS	37.6 KB	6.41 KB	3.16 KB	2.83 KB

Profile data was obtained after the project-level optimizations were implemented, focusing on the total number of cycles per application step. The new profile data served as a baseline for all further analysis.

2.4 Function-Level C Optimization

The primary focus of this phase is to take full advantage of the parallel architecture of the SC140. The optimization techniques presented here can be performed without global knowledge of the algorithms and without detailed analysis of data and control flow. The general approach for optimizing each function includes the following steps:

1. Establish the function interface and separate the function from the rest of the code to facilitate analysis.
2. Add test code that saves the function input and output contexts before and after each function call.
3. Optimize the function and monitor the output context to ensure that it remains the same as the corresponding reference output context.
4. Integrate the optimized function with the rest of the program and run the program to see if the vocoder passes all ITU-T test vectors.

2.4.1 Selecting the Functions to be Optimized

The selection of functions for optimization was based on profiler data and experience with similar applications, focusing on the most time-consuming functions. The functions selected are collectively referred to as ‘G1’ functions. The main criterion used to select the functions to be optimized was the profiler data obtained after the project-level optimizations. Information provided by the profiler includes:

- *Number of calls per application step*. Based on this information, the developer decides if a small, frequently-called function should be inlined. This information is most useful during project-level optimization.
- *Number of cycles per call*. This information is necessary to predict the speed improvement gained by optimizing a function. However, the number of cycles per call is not very useful by itself, because there are cases in which a function with a small number of cycles per call is called several times per application step.
- *Total number of cycles per application step* (in G.729, a frame). This is the most useful information for selecting the functions to optimize.

2.4.2 Predicting Speed Improvement

Optimizing a C function yielded speed improvements ranging from 1.08 to 4 times. Improvements were greatly affected by such function characteristics as the number of variables and pointers, the number and dimension of loops, data alignment, data dependencies, and the number of internal calls to other functions. For this reason it was difficult to estimate the speed improvement in advance. For example, the speed improvement is much easier to predict when multi-sample techniques are employed. On the other hand, the speed improvement for a function containing many calls to other functions or extensive control code, is not only difficult to predict but also difficult to achieve. Typical DSP code without data dependencies (for example, correlation or energy) easily yields a four time improvement, while DSP code with data dependencies generally yields a two to three time improvement.

2.4.3 General Optimization Techniques

Several optimization techniques were employed, including multisample, split summation, loop unrolling, loop merging, and loop splitting. Most optimizations employing these techniques also require data alignment, but these alignments rarely cause conflicts or degrade performance.

2.4.3.1 Multisample

Multisampling is a pipelining technique to process multiple samples simultaneously. It takes full advantage of the SC140 multiple-ALU architecture to maximize parallel operation of the execution units. In addition, this technique preserves bit-exactness and reduces the number of memory operations. This technique is most efficient when the number of output samples is a multiple of four. For a more complete description, refer to the Freescale Semiconductor application note, *StarCore Multisample Programming Technique* [12].

2.4.3.2 Split Summation

Split summation involves splitting a sum into four partial sums, using four variables and one-fourth the number of iterations. A final summation of the four partial sums is performed at the end of the process. This technique changes the sequence of operations, so special care must be taken when applying it to algorithms in which bit-exactness must be preserved. Split summation is often used to compute signal energy, as illustrated in Code Example 5. Bit-exactness of the algorithm is maintained because all of the terms of the sum are positive values (the samples are squared).

Example 5. Split Summation

```
/* Compute the energy of the signal stored in
 * the signal[] vector of size SIG_LEN (multiple of 4).
 * e0, e1, e2, e3 are partial sums.
 * The final result is stored in e0.
 */
for ( i = 0; i < SIG_LEN; i+=4 )
{
    e0 = L_mac(e0, signal[i+0], signal[i+0]);
    e1 = L_mac(e1, signal[i+1], signal[i+1]);
    e2 = L_mac(e2, signal[i+2], signal[i+2]);
    e3 = L_mac(e3, signal[i+3], signal[i+3]);
}
e0 = L_add(e0, e1); /* final summation of partial sums */
e2 = L_add(e2, e3);
e0 = L_add(e0, e2);
```

2.4.3.3 Loop Unrolling

Loop unrolling explicitly repeats the body of a loop with corresponding indices. As a stand-alone technique, loop unrolling is used to increase the ALU usage per loop step, as is illustrated in Code Example 6.

Example 6. Loop Unrolling

```
/* Scale all the values in the signal[] vector of size SIG_LEN (multiple of 4). */

for ( i = 0; i < SIG_LEN; i+=4 )
{
    signal[i+0] = L_shr(signal[i+0], 2);
    signal[i+1] = L_shr(signal[i+1], 2);
    signal[i+2] = L_shr(signal[i+2], 2);
    signal[i+3] = L_shr(signal[i+3], 2);
}
```

Loop unrolling is also performed in conjunction with the multisample technique to reuse variables that are already fetched, thus reducing memory bandwidth and alignment requirements.

2.4.3.4 Loop Merging

Combining two or more loops into a single loop loads the ALUs more efficiently and reduces the number of AGU operations, as illustrated in Code Example 7. If a merged loop still does not use all available ALU units it can be combined with other techniques described in this section.

Example 7. Loop Merging

```
/* initial loops */
for(i=0; i<L_WINDOW; i++)
{
    y[i] = mult_r(x[i], hamwindow[i]);
}

for(i=0; i<L_WINDOW; i++)
{
    e = L_mac(y[i], y[i]);
}

/* loops merged */
for(i=0; i<L_WINDOW; i++)
{
    y[i] = mult_r(x[i], hamwindow[i]);
    e    = L_mac(y[i], y[i]);
}
```

2.4.3.5 Loop Splitting

Loop splitting refers to the process of breaking a large loop with several variables or pointers into two or more shorter loops and saving the results of the partial computations in local vectors. This technique enables the compiler to allocate registers more efficiently, resulting in substantial performance improvement, especially when combined with other optimization techniques.

2.4.4 Programming Tips

The following is a summary of programming tips based on our experience in optimizing C functions. They are described in detail in *Efficient Programming Techniques for the SC140* [13].

- Declare variables as close as possible to their area of use (using C blocks) to help the compiler identify their life cycles. This improves register allocation but may require more stack memory.
- Use the `#pragma loop_count` statement, to declare that the minimum number of cycles is greater than zero, which helps the compiler to eliminate a test.
- Perform loop unrolling by rolling and reusing the values that come from the unaligned vectors.
- Use the >> operator in a variable shift displacement to prevent the compiler from translating the operation into a function call.
- Reverse the iteration order in a loop to obtain a more useful sequence of values.
- Evaluate the effect of multisample without loop unrolling to determine if the speed improvement in the unrolled case is worth the additional memory consumption.
- Add internal pointers to arrays that are already aligned to improve both alignment and clarity.
- When the data alignment property of a vector is not recognized in a code sequence, create a new function with that vector as a parameter and use the `#pragma align` directive to specify the alignment.
- Use the << operator instead of the L_shl() function to prevent the compiler from inserting a function call if overflow or underflow does not occur after a left-shift operation.

- Manually inline small functions, such as `L_Extract()`, that involve multisample operations.
- Do not apply the C operators (*, +) to operations on fractional values that only employ intrinsic functions (`L_mult()`, `L_add()`). Also, do not use intrinsic functions for integer values (for example, use only +, -, and * on indices).

Note: Some of these programming tips may not apply to future versions of the StarCore SC140 development tools.

2.4.5 Procedural Notes

Because of the lack of experience in optimizing large functions, the development team first focused on optimizing some of the smaller functions typically found in digital signal processing, such as `Syn_filt()`, `Convolve()`, `Residu()`, `Cor_h_X()`, `Autocorr()`, and `Norm_Corr()`.

The experience gained by optimizing the simple functions enabled the team to take on more difficult constructs, including `Cor_h()`, `Lsp_pre_select()`, `Levinson()`, `D4i40_17()`, `Pre_Process()`, `Post_Process()`, `Inv_sqrt()`, `Az_lsp()`, `Chebps_11()`, `Chebps_10()`, `Lag_max()`, `Pred_lt_3()`, `Get_lsp_pol()`, `Qua_gain()`, `scale_st()`, `filt_mu()`, and `search_del()`. To improve analysis, the function-level C optimization phase was extended to functions taking up to 95% of the encoder cycle count and up to 88% of the decoder cycle count.

2.4.6 Results

The vocoder performance characteristics after the function-level C optimization is shown in **Table 6**.

Table 6. Performance Characteristics After Function-Level C Optimizations

Speed	Program Size	Tables	Channel Data	Stack Size
16.6 MCPS	42.2 KB	6.75 KB	3.19 KB	2.84 KB

These results are expected to improve with compiler evolution and should not be regarded as maximum optimization values.

2.5 Algorithm Changes

If the function-level C optimization phase does not provide adequate speed improvement, the programmer can change certain algorithms or implement the existing parallelism in different ways. Algorithmic changes are classified in two major categories: platform-independent changes and platform-specific changes. In many cases algorithmic changes are performed only locally, within a particular function. But there are also cases in which data input from another function or output to another function are processed faster in another format, and the time saving is greater than the time required to convert the data. Thus, it is worthwhile to determine if frequently-accessed data structures can be changed so that:

- Functions access data sequentially
- Lengths of data arrays are multiples of four.

It is also useful to examine the relationship between module results and internal computed values. Often the number of values a module computes and stores is significantly larger than the number of values returned. For example, the output of a function might be the offset of a single value, while numerous values are computed and stored internally to compute it. In this case, it may be worthwhile to examine the relationships of the output and the internal variables to determine if the output is obtained directly or with fewer internal variables.

Special care should be taken to maintain bit-exactness when necessary. This is particularly true for algorithms in which the order of operations is changed. In these cases, it is wise to mathematically verify that bit-exactness is preserved. Other operations, such as search algorithms which return the position of a value with a given property, are more flexible and may not require such rigorous initial verification. In all cases, algorithm changes should be verified by implementing them in C rather than another high-level language.

2.5.1 Identifying Algorithms to Change

Determining which algorithms to optimize is a process that is difficult to characterize precisely. It generally involves a search of G1 functions tempered by experience. The following two guidelines reduce the scope of this search to a manageable range:

1. Use profile data to select only the most time-consuming functions from the G1 set.
2. Add the functions that provide inputs to and receive outputs from the functions chosen in step 1.

Although the functions in step 2 tend to be small and not included in the G1 function set, they are usually easily modified so that the data they provide to the functions selected in step 1 is accessed and manipulated more efficiently. In our project we selected the three most time-consuming modules, which together accounted for more than 60% of the vocoder's total execution time after the function-level C optimization phase. In the ported version these modules are also the most time consuming, each taking more than 5% of the total execution time.

Ideally, total development time would be decreased by selecting the algorithms to modify before implementing the function-level C optimizations. However, this is possible only if the improvement in execution time after function-level C optimization or assembly implementation is predicted fairly accurately, which cannot be done without a great deal of experience with the algorithms and optimization techniques.

After the algorithm changes are complete and the final C implementation passes all ITU-T test vectors, it is recommended that another round of function-level C optimizations be performed on the changed functions. The result should provide the fastest possible C implementation of the vocoder. This version also serves as excellent reference code for the assembly implementation. The split summation is already tested, concerns about bit-exactness have been addressed, and debugging of the assembly version is performed very easily by comparing it to the C version.

2.5.2 Platform-Independent Changes

Platform-independent changes to algorithms are improvements in code flow which apply to all processor types and C compilers. The following are general guidelines for implementing platform-independent changes.

- Replace the time-consuming operations `div` and `log` with multiplications.
- Remove repeated computations of the same value.
- Reorder computations to avoid repeated fetches of the same value.
- Reduce the number of tests.

2.5.3 Platform-Dependent Changes

Platform-dependent changes to algorithms are those that reorder and restructure data or reorder and regroup computation blocks to take advantage of the parallel architecture of a particular processor. Such changes, if they are done, prove to be even more effective than platform-independent changes. The following are general guidelines for performing platform-dependent changes to a system based on the SC140:

- Data structure addressing is sequential (linear), using indices rather than pointers.
- Internal pointers are initialized on multiples of four.
- Searches based on interval splitting use interval division by four, rather than two.
- Computations are adapted to pipelines with four computation units.
- DPF formats are translated to native 32-bit representation.
- Vector lengths are multiples of four. The time penalty (the difference between the time it takes to compute $(N \times 4 + 1)$ samples and $(N + 1) \times 4$ samples) is negligible, and the resulting code is substantially smaller and more clear.
- Sequential, identical computations are grouped together.

2.5.4 Procedural Notes

Because our project had an important investigative aspect, we performed algorithmic changes after implementing the function-level C optimizations. Although this resulted in some duplication of effort, we were able to demonstrate the limits of the implementation without global knowledge of the application. In a real-life application we recommend performing algorithmic changes in the first stages of the project. We performed the algorithmic changes focusing on modules that grouped several functions, including:

- `D4i40_17()` + `Cor_h()` + `Cor_h_X()`
- `Az_lsp()` + `Chebps()`
- `Lag_max()` + `Pitch_ol()`
- `pst_ltp()` + `search_del()`

2.5.5 Results

The results of the algorithmic changes proved to be quite rewarding, as shown in **Table 7**.

Table 7. Performance Characteristics After Algorithmic Changes and C Reoptimization

Speed	Program Size	Tables	Channel Data	Stack Size
12.8 MCPS	42.7 KB	6.95 KB	3.18 KB	3.05 KB

2.6 Function Implementation in Assembly

In general, rewriting C functions in assembly increases speed and reduces code size. However, because of improvements in C compiler performance, the trend is to keep most code in C and optimized C and to implement fewer functions in assembly. In our project, implementing selected functions in assembly was performed in parallel with final function-level C optimization. The functions to be implemented in assembly were selected based on profiling data and experience.

2.6.1 Selecting Functions for Assembly Implementation

At this stage of the project, speed increase is the highest priority, although code size can also decrease. The entire project is profiled, and the most time-consuming functions are identified for the G1 group. Estimates of ideal execution times for G1 functions are compared with the C optimization results. Functions compiled near-optimal are left in the optimized C version; the remainder are candidates to be implemented in assembly.

At the assembly implementation stage, functions must have a fixed interface because interface changes to an assembly function are not direct. Any modification to a C call parameter list requires modification of the corresponding assembly code where parameters are primarily accessed through the stack.

2.6.2 Implementation Approaches

There are two basic approaches to implementing assembly code—modifying the compiler output and coding directly in assembly. The first approach is most useful for relatively simple functions for which the compiled code is close to the optimal version but does not take full advantage of parallel architecture. For example, registers can be more optimally allocated, or the number of pointers needed to fetch data can be reduced.

A more frequent approach is direct assembly coding of functions that are more complex or do not perform as well as expected. The optimized C code is used as a reference for testing to ensure that bit-exactness is maintained after platform-dependent optimizations. In some instances the optimized C code is not optimal for assembly implementation due to compiler behavior. In these cases, assembly implementation is based on another model, perhaps another C version or suitable pseudo-code. In any event, it is best to use the C code as a reference, regardless of compiler performance or the techniques employed.

2.6.3 Implementation Details

Details that must be considered when writing assembly code include processor restrictions, data alignment and hardware loops alignment, and nesting order. StarCore restrictions (see the *SC140 Core Reference Manual*, section 6.4, Instruction Set Restrictions) and function calling conventions (see the *SC100 Application Binary Interface Reference Manual*, section 2.3, Function Calling Conventions) must be considered with care. For multiple-register move operations to and from memory, the memory addresses must meet certain alignment restrictions. There is no assembly equivalent to the C `assert()` function, so alignment must be checked manually. If a hardware loop starting address is not aligned (that is, the first execution set is spread over two fetch sets), one stall cycle is added to the loop execution at each iteration. In intermediate development phases, loop alignment is specified by FALIGN or OPT LPA assembler directives; in the final version of the code, instructions are rearranged or dummy instructions are manually inserted to ensure alignment. The order in which loops are nested is important because the processor prioritizes hardware loops (loop3 is the highest and loop0 is the lowest priority) and executes the active loop with higher priority. Therefore the proper approach is to nest hardware loops in reverse order of their indices.

2.6.4 Programming Tips

The following programming tips for assembly optimization are valid for all versions of the development tools; some of these ideas also apply to C optimizations [13].

- Perform multisample with less than four samples computed in parallel to reduce the number of variables or inner loops.
- Translate ratio comparisons into product comparisons because multiplication requires less cycles than division. Combine 16-bit numerators and denominators into a single 32-bit word, pairing `MPYUS` with `MPYSU`.

- Use `MOVE.L` for data register transfers when four or more `TFR` instructions are required in parallel.
- Perform a shift operation with a multiply instruction to make more registers available for parallel operations when possible.
- Improve code parallelism in loops containing comparisons by using `if` with one branch.
- Compile AGU-bound computations with DALU operations by merging loops when possible.
- Use two partial maxima to optimally find the maximum from a string of values.

Each function in our implementation is fully ABI-compliant (although speed could be improved by changing some functions that would take them out of ABI compliance). In our implementation most C-optimized functions are also implemented in assembly. There are 18 such functions and their C equivalent execution accounted for 86% of the total time of the initial ported code. Implementation strategies discussed in **Section 5** are used to reduce the number of functions implemented in assembly code.

2.6.5 Results

The vocoder performance characteristics after function implementation in assembly are shown in **Table 8**. This version is also referred to as 'mixed implementation' because it contains both assembly and C functions. All code was tested on both the simulator and SDP boards and passed the bit-exactness tests for all ITU-T test vectors.

Table 8. Performance Characteristics After Assembly Implementation

Speed	Program Size	Tables	Channel Data	Stack Size
8.44 MCPS	36.2 KB	6.56 KB	3.19 KB	2.95 KB

3 Results

Optimization activities must be measured to assess the quality of the optimizations and decide where to invest effort in subsequent phases. This section presents the results obtained during the development cycle of our project and the methods used to measure them.

3.1 Measurement Techniques

Various tools and techniques were used to measure the execution time, code size, and data size of each function.

3.1.1 Execution Time

The execution time of functions and modules were evaluated using the number of simulated cycles spent in the measured unit. Although this measurement technique does not count stall cycles (for example, generated by memory contention), it is a good approximation of actual processing time. Three tools were used to gather execution time information: the SC140 simulator (simsc100), the SC140 executable runner (runsc100), and the CodeWarrior Profiler (integrated in the CW IDE).

The most flexible tool is the SC140 simulator. The data provided by the status breakpoints is logged and parsed after execution to extract information. The simulator was used to take worst-case measurements of the encoder and decoder because it is the only tool that provides exact measurements for each function call. Examples of the simulator script to generate the log and the Perl script to analyze the log are presented in **Appendix C**.

The -t option of the SC140 executable runner is a useful tool for providing qualitative data. This tool was used extensively in the optimization phases to monitor performance and bit-exactness. An important advantage of `runsc100` is that it is the fastest of all tools.

The CodeWarrior profiler provides information on the average time spent in each function. This information is useful for quickly establishing the relationship between the functions in an application. The profiler provides the overall number of cycles spent in each function with and without descendants (in absolute and percentage values) as well as the number of times the function was called. In this project the profiler was used to identify the most time-consuming functions and where to direct further optimization efforts.

In choosing the most appropriate measurement techniques, consider the types of programs that each measurement tool can execute. All tools simulate the file I/O operations, but the profiler does not support console input. Command line parameters are passed only in profiler and executable runner. The simulator is only stopped by a halt breakpoint, while the other tools end execution at the `exit()` command.

All tools report the number of cycles to execute each function. For the vocoder project, a simple computation was used to convert the number of cycles to the processing load, measured in million cycles per second (MCPS). The number of MCPS required to encode or decode a frame is obtained by multiplying the measured number of cycles by the number of frames to be processed per second (in G.729, 100 frames of 10 ms each must be processed per second), and dividing the result by 1,000,000. For example, if it takes 220,000 cycles to encode or decode a frame, the processing power required is (220,000 × 100) / 1,000,000 = 22 MCPS.

3.1.2 Code Size

Code size is measured in two ways. In the first method, the linker generates a map file which describes the memory map of the entire application, including summary data (for example, for object files) and detail data (for example, for global functions and data). This technique is useful for obtaining data for individual functions, such as size and placement. An excerpt from a map file and notes on how to use it are provided in **Appendix C**. The second method to measure code size is based on the `sc100-size.exe` program provided with the StarCore support tools. The program analyzes an ELF object file and reports the size of the classical C segments `.text, .rodata, .data,` and `.bss`. This method is good for providing summary data.

3.1.3 Data Size

In the vocoder application, data size has three components—tables and global data, per-channel data, and variables on the stack. The first component is fixed for the entire application, while the other two have fixed values per channel. In other documents, tables are typically considered ROM data, and the equivalent of channel data is static memory. Table and overhead data are reported by the `.data` segment of the `sc100-size.exe` program. Table placement information is extracted from the map file. Channel data is obtained by running a simple C program which shows the size of the channel data structures.

Measuring the stack size is more complicated. The basic idea is to measure the stack when the program starts (at the beginning of the `g729_encode()` function) and capture every change of the stack pointer without interrupting execution. The simulator provides all the features required to create a log containing every value received by the stack pointer during the execution of the coder. The stack pointer log is generated using only the status breakpoint combined with the display of the stack pointer, as illustrated in **Appendix C**. The monitored stack pointer is represented by the variable `esp`, which the C code generated by the SC140 compiler uses as an ordinary stack pointer. The generated logs (one each for the encoder and the decoder) are analyzed with a Perl script that shows the maximum difference between stack top and stack base. An example simulator script along with the Perl script that performs the analysis is presented in **Appendix C**. For stack size measurements, it is important to note that the encoder and decoder do not run concurrently. Thus, the stack figure is the maximum of the individual stack consumptions of the encoder and decoder.

3.2 Performance Estimation

The performance for the SC140 implementation of G.729 was estimated to be between 10 and 11.5 MCPS. This estimate was based on G.729 implementations on mono ALU processors and a reported optimization factor of 2.4 for certain SC140 applications compared to mono ALU architectures [15]. Based on these estimations and our particular implementation goals, a target value of 10.7 MCPS was chosen for our implementation. This is a worst-case figure, based on the maximum processing time of both the encoder and decoder for one frame. The final worst-case processor load was measured as 8.44 MCPS. It is important to note that different implementation goals result in different performance targets than the one chosen. These results are summarized in **Table 9**.

Table 9. G.729 Implementation Target and Result

	Worst-Case Processing Load (MCPS)
Project Target	10.7
Results	8.44

3.3 Project Milestones

The following sections present the results of porting the G.729 code to the SC140. The primary performance data (MCPS and program memory) is presented in comparison to the effort expended to achieve that performance. The evolution of data size is also presented. The major milestones shown on all graphs include code versions after each of the following steps:

- Initial porting to the SC140 core, including multichannel transformations.
- Project-level optimization, including inlining of DPF functions.
- Initial function-level C optimization, before algorithmic changes.
- The final C version, after algorithmic changes and reoptimization.
- The final mixed implementation, including selected functions in assembly.

3.4 Execution Time

The evolution in execution speed through the different versions of the project are summarized in **Figure 4**.

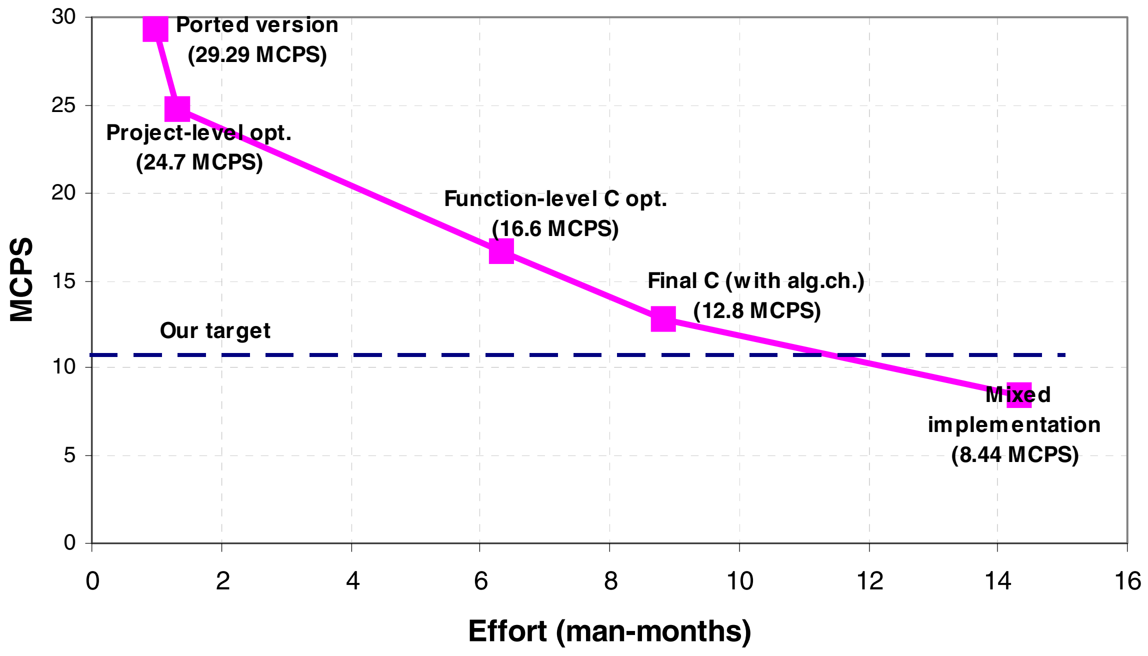

Figure 4. MCPS Versus Effort

The X axis represents the manpower used to achieve each milestone, and the Y axis represents the number of MCPS for each milestone. The project-level optimizations, especially inlining small DPF functions, proved to be beneficial for both execution time and code size. The vocoder time reduction due to this modification was more than 4.6 MCPS. The code size also decreased slightly, primarily due to the removal of actions performed before and after function calls which were no longer required.

The function-level C optimizations reduced execution time by an additional 8.1 MCPS. Further improvement from these optimizations is possible with compiler improvements and increased SC140 programming experience.

Algorithmic changes reduced execution time by another 3.8 MCPS.

The phases after project-level optimization included some redundant work because time-consuming functions were optimized both before and after algorithmic changes. Given the substantial performance improvement resulting from algorithmic changes, we recommend performing them in the first stages of the project.

Although the assembly implementation is substantially faster (4.3 MCPS less) than the best C-only version, we found the C version to be quite satisfactory. Compiler improvements help reduce the gap between the best C version and the assembly implementation, considering that the assembly version cannot be improved significantly.

For an SC140 processor running at 300 MHz (300 MCPS), the number of channels that can be processed simultaneously is (300 ÷ 8.44) = 35 channels. The C version only processes 23 channels, but with a significant reduction in development time.

The fact that the final execution time of 8.44 MCPS is substantially better than our original target of 10.7 MCPS suggests that the development effort can be reduced. Strategies for reducing the implementation effort are discussed in **Section 5**, *Implementation Strategies,* on page 30.

3.5 Code Size

The evolution in code size as the project progressed is summarized in **Figure 5**. Again, the X axis represents the manpower used to achieve each milestone. The Y axis represents the code size in KB at each milestone. The initial ported version of the G.729 to the SC140 resulted in a code size of 37.9 KB (encoder + decoder). All code was generated from C with the compiler optimized for speed (`-Ot2` option). For the initial version it was difficult to apply time and size compiler options differentially because the functions were not separated into units which could be compiled individually.

Project-level optimization resulted in a slight reduction in code size. However, code size increased to 42.2 KB after function-level C optimizations. The increase of almost 5 KB (12.5 percent) was due primarily to multisample and loop unrolling (see **Section 2.4.3**, *General Optimization Techniques,* on page 11). Applying multisample to a simple loop increases its code size by up to four times. Loop unrolling and compiler-generated software pipelining increases the code size in specific areas even further.

Algorithmic changes plus reoptimizations in C resulted in a negligible increase in code size compared to the previous C optimization phase, reducing execution time by a factor of 1.29 at a cost of only a 1.2 percent increase in code size. Two major factors reduce the amount by which code size increases without significant speed penalty. In some cases, multisample by four is replaced by multisample by two. Also, a less stringent compiler option for speed is chosen. Time optimization should be applied aggressively only on the most time-consuming functions.

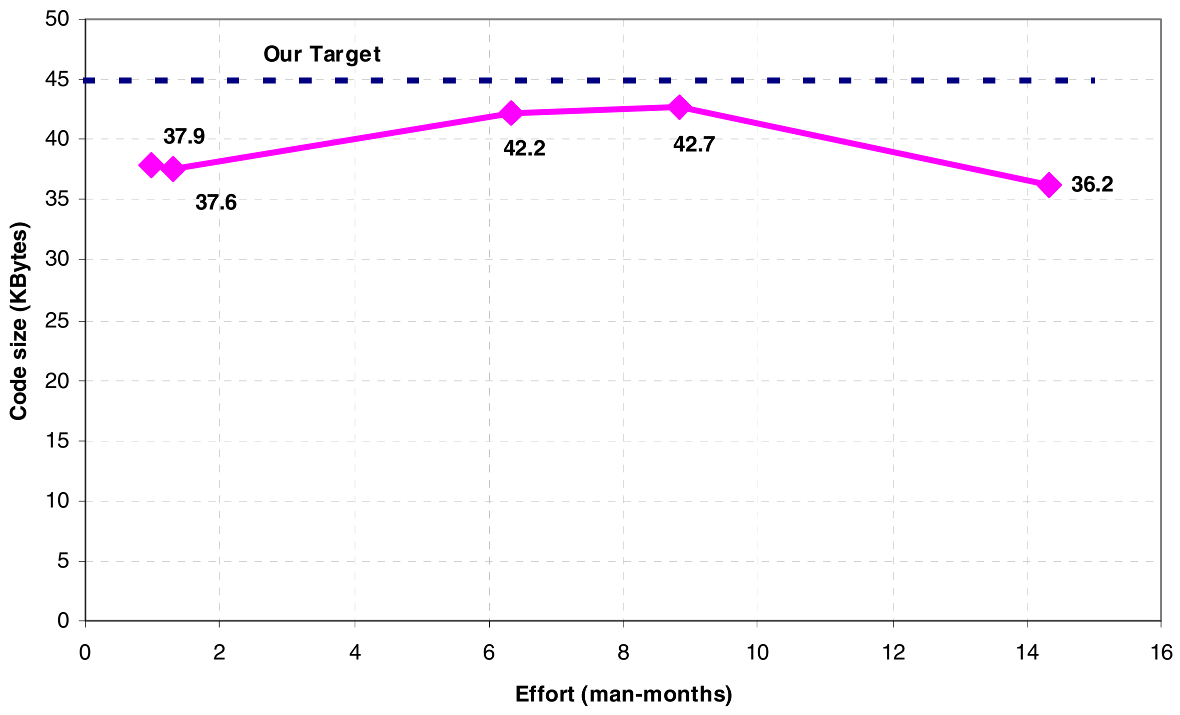

Figure 5. Code Size Versus Effort

Improvements in compiler technology should provide more satisfactory trade-offs between speed and size. Compiling the unoptimized code with a size optimization feature provides some compensation for the code increase resulting from the time optimization applied to time-consuming functions.

The final implementation in assembly resulted in a substantial decrease in code size. Although only a few functions were implemented in assembly, the code size was reduced to 36.2 KB, which is actually less than the initial code size. This confirmed that manually-written code is both faster and smaller than compiler-generated code. A future compiler version offering a size reduction option may generate sufficiently small code size without the manpower required for assembly coding.

3.6 Data Size

The data size refers to all the data handled by the vocoder. It includes read-only tables (which can be stored in ROM), stack data for transient variables, and channel data structures to store persistent channel data. **Figure 6** is a bar graph showing the size of each of these components for each phase of the project. Note that the optimizations performed in each phase, which were geared primarily to increase speed, had no substantial effect on total data size.

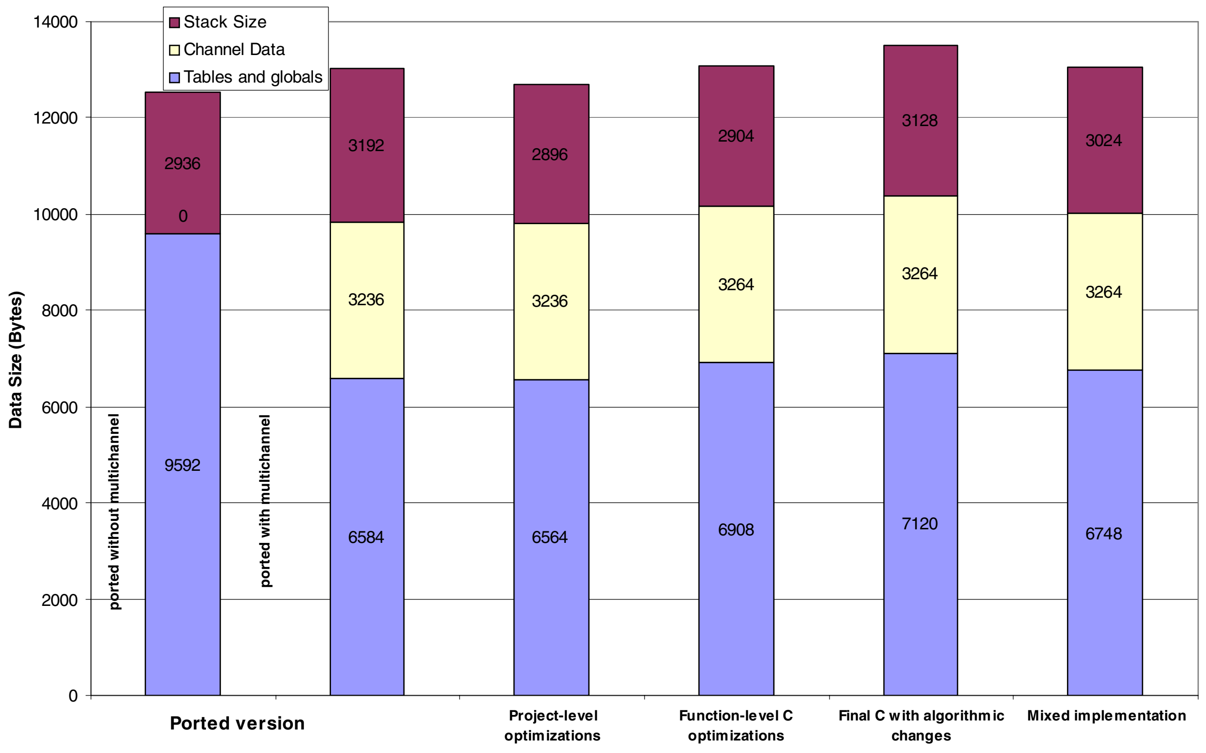

Figure 6. Data Size Versus Project Phase

3.6.1 Tables and Globals

The tables and globals component of data memory includes all data placed in the data segment. This includes all tables, most of which are defined in `tab_ld8k.c`, and compiler-generated overhead data. The compiler generates a kind of software stack of 10–20 bytes for each module. The difference of 536 bytes between the final C version and the initial ported version is mostly due to this overhead data.

3.6.2 Channel Data

The channel data size remained virtually unchanged throughout the project. The one change affecting channel data size was the insertion of extra padding to align certain vectors in the data structure.

One way to decrease the channel data size is to remove the storage area allocated to hold new samples. For example, the speech data structure contains two main parts—old speech, which contains previous frames, and new speech, which contains the current frame. Only the old speech portion is required to process the next frame. However, eliminating the unneeded data requires modifying the vocoder either to use two vectors (for old speech and new speech) or to copy the old values into a local array. Thus, reducing channel data in this way significantly increases the stack and code sizes.

3.6.3 Stack Size

The greatest demand on stack size in our application came from the fixed-codebook search. The maximum top of the stack occurs in the `D4i40_17()` function, primarily due to the large data structures stored on the `ACELP_Codebook()` stack frame. The algorithmic changes and reoptimizations that were performed on `D4i40_17()` created supplementary data structures which increased the stack size.

The compiler time-optimized some functions by always storing variables in registers rather than the stack frame. However, the unused stack frames were not removed until the assembly phase. Directly implementing `D4i40_17()` in assembly accounts for most of the stack reduction after this phase.

One way to reduce stack size is to examine the life cycle of variables (especially large arrays) to collapse arrays with disjointed life cycles into one data structure. Allocating blocks for variables may help improve the management of local variables, even in large functions.

3.7 Testing

Our test procedures included both unit testing and integration tests. All tests were run on the SC140 simulator, and several tests were also run on the reference C compiler. One complete test on the reference C compiler took about 10 minutes, while a worst-case analysis on the simulator (which also served as an integration test for bit-exactness) took 6 to 10 hours, depending on the optimization stage. Tests on the SDP board were faster (less than 1 hour), but did not provide cycle counts.

All builds were tested on the simulator to verify bit-exactness. The SDP board tests were especially useful after the assembly phase.

4 Details of Selected Functions

This section presents details of the optimization process for three functions, `Norm_Corr()`, `ACELP_Codebook()`, and `Lag_max()`. The function-level, algorithmic, and assembly changes for each function are presented, as well as the effect of these changes on the efficiency of code generated by the compiler.

4.1 Optimizations in Norm_Corr()

The `Norm_Corr()` function finds the normalized correlation (correlation divided by the square root of the energy of filtered excitation) between the target vector and the filtered past excitation. The main steps of this function in the reference C code include the following:

1. Compute the filtered excitation for the minimum delay.
2. Scale the excitation vector to avoid overflow.
3. Compute the energy of the filtered excitation to check overflow.

4. Scale the filtered excitation to avoid overflow and compute the energy of the scaled filtered excitation.
5. For every possible delay between minimum and maximum, compute the normalized correlation vector and modify the excitation for the next iteration.

4.1.1 Function-Level C Optimizations

The C optimizations applied to `Norm_Corr()` to speed up the function include the following:

- Align the input and local vectors to allow parallel data moves.
- Unroll the scaling excitation loop.
- Apply split summation to the energy and correlation computation loops.
- Use multisample to compute the excitation for next iteration.
- Replace the tests that use subtraction with direct comparisons.
  for example, replace *if (sub(a,b)>0)* with *if (a>b)*.
- Replace functions calls with operators when integer values are used.
  for example, replace *i= sub(i,1)* with *i--*
- Replace the `L_shl()` function call with the << operator.
- Replace unmodified variables with constants defined in the G.729 reference code.

The initial code for computing energy and correlation was inefficient because the compiler did not take advantage of input vector alignment. The compiler could not use the registers efficiently when dealing with large loops, which contain several inner loops. To increase this efficiency, the two major loops that performed correlation and energy computation were moved to separate functions in separate files to avoid function inlining. This incurred a calling overhead of up to 6 cycles, but the performance gain was about 10 cycles. This gain was significant because the functions are called 14 times. After these function-level optimizations, the `Norm_Corr()` function ran 2.22 times faster than the initial version.

4.1.2 Algorithmic Changes

The major modifications to the algorithms in `Norm_Corr()` included the following:

- Compute the scaled filtered excitation vector only when overflow occurs. In the ITU-T `speech.bit` test vector, overflow occurs in only 205 out of 3750 frames.
- Compute the factors that affect the new scaled filtered excitations in a separate loop and store the results in a separate vector. This modification enables the use of multisample to use the registers more efficiently.
- Exploit the 32-bit capabilities of the processor by using 32-bit variables instead of the DPF format defined in the G.729 standard, and replace the `Mpy_32()` function with a new function which works with native 32-bit data but preserves bit-exactness.
- Eliminate the *else* branch of the *if()…else…* statement to reduce the number of branch-like instructions.
- Pipeline the main loop to avoid the unnecessary scaling of filtered excitation for the final iteration.
- Compute the energy in the same loop which computes the new scaled filtered excitation values to avoid extra memory moves.

Note: This modification did not work as intended due to suboptimal register utilization, but it was implemented in assembly.

These modifications were first applied to the unoptimized C code to verify bit-exactness. The function was then reoptimized in C, resulting in a speed improvement of 2.8 over the unoptimized version.

4.1.3 Assembly Implementation

Assembly implementation was applied to the algorithm-modified code. Loops which had been extracted to separate functions were re-inlined, and techniques such as software-pipelining and split summation were applied. The result was an impressive 6.1 factor in speed improvement over the initial version.

4.1.4 Summary

Table 10 lists the `Norm_Corr()` cycle count and the code size for each version.

Table 10. Norm_Corr() Performance Summary

Version	Cycle Count	Size (Bytes)
Initial version	9235	706
Function-level C optimization	4156	772
Algorithmic changes	3291	1084
Assembly implementation	1512	692

Appendix A lists the prototype and pseudocode for `Norm_Corr()`.

4.2 Optimizations in ACELP_Codebook()

The codebook search procedure is the most time-consuming part of the G.729 vocoder. In the reference C code provided with the G.729 standard, this procedure is performed in the `ACELP_Codebook()` function, which consists of the following basic steps:

1. Pre-filter the selected codebook vector to enhance the harmonic components to improve the quality of reconstructed speech.
2. Call `Cor_h()`, which computes the correlation matrix of the impulse response *h(n)*.
3. Call `Cor_h_X()`, which computes the correlation vector of the target signal *x'(n)* with the impulse response *h(n)*.
4. Call `D4i40_17()` to perform an exhaustive search for the four pulses. This function effectively generates all possible code words and chooses the one that maximizes the ratio between the correlation squared and the energy.

4.2.1 Function-Level C Optimizations

The following C optimization procedures were applied to the initial G.729 version:

- Align the input data and local data on the stack.
- Initialize pointers and define local variables where they are used to minimize stack allocation and execution time.
- Apply multisample and software pipelining techniques to loops.
- Replace the tests that use subtraction with direct comparisons. For example, replace *if (sub(a,b)>0)* with *if (a>b)*.

- Replace function calls with operators when integer values are used. For example, replace *time= sub(time,1)* with *time--*
- Merge two lower-level loops into a single loop.

After these optimizations, the `ACELP_Codebook()` function ran 1.2 times faster than the initial version but now consumed 45 percent of optimized C encoder time instead of the initial 31 percent. Thus, the decision was made to apply algorithmic changes directly to the original `ACELP_Codebook()` function and its descendants.

4.2.2 Algorithmic Changes

The original code contained many pointers, indices, and variables. For example, in the original search block there were 19 pointers and at least 18 variables in four-nested loops which required a great deal of time-consuming save and restore operations in both C and assembly. Thus, reducing the number of variables became a primary focus of the algorithm change phase. The major modifications to the algorithms were as follows:

- Include the sign information in the correlation matrix at the time it is built. In the original code, the sign information is not introduced until the `D4i40_17()`function. We modified `Cor_h_X()` to compute the sign information and reversed the sequence of the first two functions calls so that `Cor_h_X()` is called first. Thus, the sign information is now computed in `Cor_h_X()` and passed to `Cor_h()` through a separate vector. The function `Cor_h()` now includes the sign information as it builds the correlation matrix, so the sign computations in `D4i40_17()` are no longer necessary.
- Rearrange the correlation matrix and correlation vector so that the correlation vector is addressed sequentially using the same pointer, thus reducing the number pointers required from 19 to 13.
- Combine the two 16-bit values *E* and *C* in the innermost loop into a single 32-bit value, *E:C*. In this way the two pairs of 16-bit values are stored in two registers instead of four, and computations are performed more efficiently using 32-bit multiplication instructions.

After these changes, the `ACELP_Codebook()` function was 1.43 times faster than the initial version.

4.2.3 Reoptimizing C After Algorithmic Changes

Because the algorithmic changes were applied to the reference G.729 C code, a new function-level C optimization was needed. The following optimizations were included in both the C and assembly versions:

- All vectors were aligned on 8-byte boundaries to enable multiple data transfers.
- The `max` instruction was used instead of the classic compare function.
- Split summation and multisample were applied to inner loops.
- Two or more data variables were combined into a single variable.
- Multiplication was performed by right-shifting.
- The *else* branch was removed from *if(…)…else* statements.

The C optimizations after algorithmic changes focused on primarily on `D4i40_17()` because it was by far the most time-consuming function. Programming tips used to speed up the ‘C’ implementation included:

- Multisample by two instead of four in all loops.
- Use variables instead of constants to force the C compiler to use all four ALUs. For example, in the initial C code, the use of the constant 1 in certain instructions, for example, *L = L_mac(L, v[i], 1),* results in assembly statements such as *mac #1,Dx,Dy* which cannot be grouped into a single instruction set due to instruction size. Replacing the constant 1 with the variable *one*, initialized with the value 1, enables the use of a statement such as *L = L_mac(L, v[i], one)*.

- Replace the `L_shl()` function with the << operator.
- Replace the `mult()` instruction with `L_mult()` combined with a 16-bit right shift.

After function-level reoptimization the encoder ran 1.77 times faster than the initial version.

4.2.4 Assembly Implementation

The assembly version is similar to the reoptimized C version, but optimizes register usage. The final version of the `ACELP_Codebook()` function is 2.8 times faster and 1.1 times smaller than the initial version.

4.2.5 Summary

Table 11 lists the `ACELP_Codebook()` cycle count and code size after each optimization step

Table 11. ACELP_Codebook() Performance Summary

Version	Cycle Count	Size (Bytes)
Initial version	38713	4804
Initial function-level C optimizations	32102	4828
Algorithmic changes[1]	26994	6456
C optimizations after algorithm changes	21770	6600
Assembly implementation	13842	4372
Note: Applied to initial version.		

4.3 Optimizations in Lag_max()

The `Lag_max()` function is called by the `Pitch_ol()` function to compute the open-loop pitch estimation. This computation is done in the following steps:

1. Compute the autocorrelation of the input signal for all possible time lags between minimum and maximum lag.
2. Determine which lag corresponds to the maximum autocorrelation value.
3. Compute the normalized correlation for the selected lag.

4.3.1 Function-Level C Optimizations

The following optimizations were applied to the original reference code:

- Align the input and local vectors to allow parallel data moves.
- Use multisample to compute correlations.
- Replace unmodified variables with constants.

Analysis of the original function revealed that some parameters are constant. The maximum lag and the minimum lag parameters of the `Lag_max()` function are always specific values, and the number of correlations computed is always a multiple of four. This suggests the use of multisample to take advantage of the four ALU architecture of the SC140. Also, the *scal_sig[-PIT_MAX]* and *signal[-PIT_MAX]* pointers, which are used to compute correlations in the in `Pitch_ol()` function, are aligned on 8-byte boundaries to generate more efficient code.

The reference C code uses ‘greater or equal’ to compare values in its search for the maximum correlation, but there is no such instruction for the SC140. The ‘greater or equal’ comparison was replaced with ‘greater’ so that the compiler would generate more efficient code, and the comparison order was reversed to retain the same comparison bias as the original code. (The original code applied the ‘greater or equal’ comparison from maximum to minimum lag values, thus favoring the minimum value. Applying the ‘greater’ comparison to the values in order from minimum to maximum also favors the minimum value.)

The `Lag_max()` code resulting from these function-level changes is listed in **Appendix A**. Because four correlations are computed in parallel, four comparisons must be performed inside the outer loop to determine the maximum value. The compiler generated efficient code for the inner loop (four *MAC*s and two moves in the same execution set), but did not generate the best possible code for comparisons, so this became the focus of assembly optimization.

4.3.2 Assembly Implementation

The initial assembly version was developed from the optimized C version, without optimizing the four sequential comparison blocks. Less than 100 cycles was gained, but the code was quite similar to the code generated by the compiler, which verified that the compiler generated nearly optimum code.

The comparison block, which initially used one variable to track the maximum value, required 8 cycles. By using two variables, the cycle count can theoretically be reduced to 5 cycles. Software pipelining reduces the effective cycle count to closer to four. An additional cycle is also required to compare the two variables to each other. This technique does not compile well from C, so it was implemented in assembly, as shown in Code Example 8.

Example 8. Lag_max() Comparison Fragment Using Two Maxima

```
cmpgt d0,d7
[
 ift   tfr d7,d0     add d12,d2,d15
 ifa   cmpgt d1,d6   sub #2,d2
]
[
 ift   tfr d6,d1     add d12,d2,d14
 ifa   cmpgt d0,d5   sub #2,d2
]
[
 ift   tfr d5,d0     add d12,d2,d15
 ifa   cmpgt d1,d4   sub #2,d2
]
[
 ift   tfr d4,d1     add d12,d2,d14
 ifa   sub #2,d2
]
```

The final assembly code for the `Lag_max()` function is listed in **Appendix B**. In this version, the final comparison is moved to just after the `doen3` instruction to fulfill the 2-instruction minimum requirement between `doen3` and `loopstart3`. Also note the use of the fake comparison of `d0` and `d1` to initialize the T bit to FALSE (`d0` and `d1` are not equal), which gains 1 cycle in the outer loop.

4.3.3 Summary

Table 12 lists the `Lag_max()` cycle count and the code size for each version.

Table 12. Lag_max() Performance Summary

Version	Cycles per Frame[1]	Size
Initial C version	12756	324
Optimized C version	3625	574
Final assembly version	3247	362
Note: Includes three calls.		

These results show that the C compiler generates efficient code when optimization techniques are used. Code generated for the inner loop is especially efficient—four macs with two moves appear in the same execution set. However, the compiler does not create the best code if two maximum values are used. Code generated for comparisons is similar to the first assembly version, taking eight cycles to compute all four comparisons. However, even if the generated code is not optimum, it is very efficient and performs well.

5 Implementation Strategies

This chapter discusses different SC140 implementation strategies for complex DSP applications. Theoretical approaches are examined which suggest a limit for the percentage of functions to be optimized in either C or assembly; this limit is established at either 80 or 94 percent of application execution time. More practical approaches are also discussed, such as optimizing only those functions which are strictly necessary to meet the project performance target, or implementing a larger set of functions to improve the performance parameters.

The discussions in this section are based only on the encoder portion of the vocoder. The code generated from the optimized C version of the encoder required improvement from assembly implementation, primarily in the fixed codebook search module, to meet the performance target. The decoder met the performance target using *only* optimized C.

All implementation strategies should start with an analysis of the information provided by the profile data. This information is used to indicate, among other things, which functions are the most time consuming, and which functions may be equivalent (require the same number of cycles to execute). Optimizing C code is well worth the effort. Even if the optimizations are not always well-reflected in the compiler output, the optimized C code serves as a reference point for starting the assembly implementation in the early stages of the project. The optimized C maintains the bit-exactness and serves as an implementation pattern for assembly programming.

One way to determine when to begin the assembly implementation is to use a graph of performance versus effort. This graph is viewed as an asymptotic curve, especially when functions are optimized in descending order of execution time. Given a deadline date, assembly implementation should begin at the point where the tangent to the curve crosses the project performance target *after* the effort point corresponding to the project deadline.

5.1 Theoretical Background

The basic formula that links the performance of the group of functions targeted for optimization (G1) to the overall application performance is shown in **Equation 3**.

$$S = \frac{1}{1 - \frac{P(f-1)}{f}} = \frac{f}{P(1-f)+f}$$ **Equation 3**

where S is the application performance improvement
P is the percentage of the application run-time taken by the G1 functions, and
f is the optimization factor.

Given S, f is computed as

$$f = \frac{SP}{1 + SP - S} \qquad \textbf{Equation 4}$$

A common rule of thumb for software performance is the so-called 80-20 rule, which states that 80 percent of run time is spent in 20 percent of the code. This suggests that the optimization effort should focus on the most time-consuming functions which account for 80 percent of the total execution time. With a performance improvement factor of four (based on four ALUs), the run time for our optimized code should be [(80 ÷ 4) + 20]=40 percent of the original run time. This amount of improvement is not always achieved with only C optimization. Another way to interpret the 80-20 rule is to apply it to the optimized software. In other words, if 20 percent of the optimized software accounts for 80 percent of the run time, what percent P of the original run time should be the focus for optimization?

Let P = percent of original ported code to be optimized
T = total original run time
$t1$ = run time of optimized portion after optimization
= $PT/4$ with an optimization factor of four.
$t2$ = run time of the unoptimized portion
= $T(1 - P)$.

Then

$$\frac{t1}{t2} = \frac{80}{20} = \frac{PT/4}{T(1-P)} = \frac{P/4}{(1-P)} \qquad \textbf{Equation 5}$$

and P = 94 percent. The final application execution time is estimated as (P/4 + (1 – P)) = 1 – 3P/4, which is about 29.5 percent of the original time.

To achieve target performance with the least amount of development time, the number of functions implemented in assembly should be minimized. Performance prediction is very important here—the analyst must decide which combination of functions will achieve target performance with the least amount of development time. The best candidates are time-consuming functions for which the compiler does not produce optimum code.

5.2 Project Implementation

The G.729 vocoder optimization was the first StarCore project for our team. After running the first profile, the first functions we selected for C optimization were general DSP functions such as `Autocorr()`, `Convolve()`, `Cor_h_X()`, `Syn_filt()`, `Residu()`, and `Lag_max()`. We also spent some time on less important functions, such as `Inv_sqrt()` and `Get_lsp_pol()`, mostly for training purposes. The experience gained with the C compiler from these functions helped in addressing the more important functions `D4i40_17()`, `Cor_h()`, `Norm_Corr()`, `Az_lsp()`, `Chebps()`, `Levinson()`, and `search_del()`.

At the end of the C optimization phase, we decided to follow two directions in parallel. The first group implemented algorithmic changes for the most time-consuming functions, after which they were reoptimized. This provided a C source reference for assembly implementation. The second group rewrote certain functions in assembly which were not the target of algorithmic changes, including `Syn_filt()`, `Residu()`, `Convolve()`, and `Autocorr()`. The experience gained by the second group was successfully applied to the assembly implementation of the optimized C code produced by the first group. The assembly optimization phase ended when the gain versus effort curve saturated.

Compared to the ported version, we achieved an application improvement of 2.28 times for optimized C and 3.47 times for the best assembly implementation. These values translate into a StarCore optimization factor of 2.44 for the optimized C code and 3.99 for the assembly. Thus, in assembly the performance improvement approaches the ideal limit of four for a system with four ALUs.

The primary reason for this dramatic improvement is that the assembly programmer fully exploits the potential of the architecture—the registers are more fully used, and memory moves are performed in parallel with other computations. These improvements cannot be reliably implemented by the compiler. The performance margin between our results and the project target allowed us to study several hypothetical scenarios and to explore ways to reduce the efforts in future implementation.

5.3 Project Results

Figure 7 presents the results of the various implementation strategies for the G.729 vocoder and the effort needed to achieve these results.

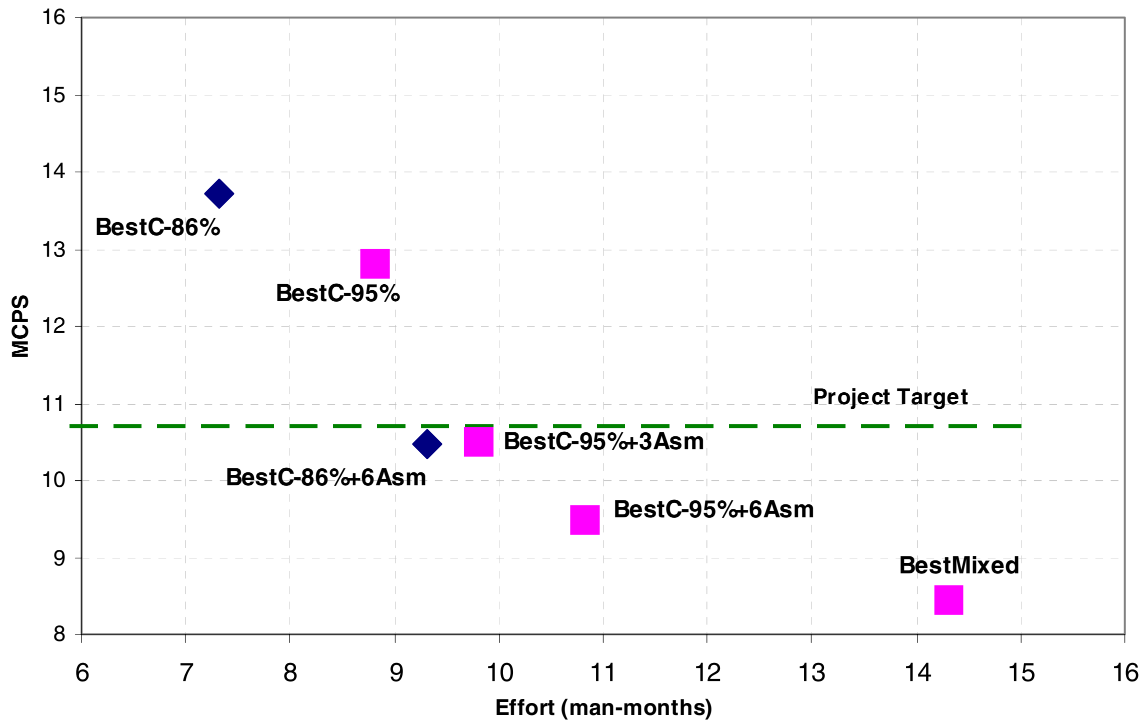

Figure 7. Performance Versus Effort for Various G.729 Vocoder Implementations

The squares in **Figure 7** represent steps in the actual implementation. First, code representing 95 percent of the original run time was optimized in C, with no assembly optimization (the ‘BestC-95 percent’ point on the graph). Then, writing only three functions in assembly (`D4i40_17()`, `Cor_h()`, and `Norm_Corr()`), a performance of 10.49 MCPS was achieved. Thus, about 9.5 percent of the code is in assembly and 90.5 percent is optimized C code. The effort required to achieve this target was only 10 man-months. In the next phase, three additional functions (`Syn_filt()`, `Az_lsp()` and `Chebps()`) were implemented in assembly for a total of 6 assembly functions, resulting in a run time of 9.47 MCPS after a total effort of about 11 man-months. In the final phase, represented by the ‘BestMixed’ point on the graph, 18 functions are implemented in assembly. These results are summarized in **Table 13**.

Table 13. Performance Versus Number of Assembly-Implemented Functions

Number of Functions	MCPS	Improvement Factor	Additional Man-Months
0	12.8	1	0
3	10.49	1.22	1
6	9.47	1.35	2
18	8.44	1.52	5

The diamonds in **Figure 7** represent a hypothetical variation based on our actual results. In this variation, only those functions which are not eventually implemented in assembly are optimized in C; this amounts to code representing 86 percent of the original run time. The 'BestC-86 percent' point represents the performance after this portion of the code is optimized, with no assembly optimizations. BestC-86 percent+6Asm represents the performance after an additional six (non-C-optimized) functions are implemented in assembly. These results confirm that the SC140 has a compiler-friendly architecture and that more emphasis should be placed on C development and less on assembly. Our project demonstrated the importance of developing code in C and implementing in assembly only those few functions for which the compiler does not produce optimum code.

6 Conclusions

The approach used and recommended has the following key components:

- C source optimizations for selected functions, with or without algorithmic changes.
- Assembly implementation of a restricted number of time-critical functions.

Selected C functions are optimized at both the project and function level. These optimizations can be performed without specific knowledge of the code algorithms. Application profiling helps to identify the most time-consuming functions. Previous experience with a similar application further refines information gathered through profiling.

The recommended procedure for implementing project-level and function-level C optimization includes profiling the initially ported code, inlining frequently-called functions, and optimizing the C code so that the compiler produces code that is better adapted to the SC140 architecture. In the C optimization step we employed several techniques, including multisample, loop unrolling, split summation, and loop merging. On the G.729 project, these optimizations reduced the program run time by a factor of 1.76, increasing code size by only 11 percent. The development time for this phase of the project was 5.5 man-months. The programming team had broad experience with C but not with the StarCore platform.

Further run-time reduction is achieved by applying algorithmic changes to critical functions grouped in modules. This activity involves advanced understanding of the algorithms and algorithm design. Four such modules were chosen for the vocoder project. The algorithm-modified functions were then reoptimized in C code.

Profiler data assists in identifying functions on which to implement algorithmic changes. Each of the functions initially selected accounted for more than 5 percent of the total execution time of the ported version of the vocoder, and collectively took more than 60 percent of total execution time of the optimized C version. This initial set of functions was expanded to include functions that provide inputs to and receive outputs from the functions in the initial set.

The algorithmic changes employed on the vocoder project included both architecture-independent modifications and StarCore-specific adaptations. The latter algorithmic changes would not have been necessary if the algorithms were developed specifically for a StarCore implementation from the beginning. Basic algorithmic changes implemented included modifying vector sizes and internal pointer offsets to be multiples of four, sequential array addressing, searches with interval division by four, and use of native 32-bit data format.

Algorithmic changes reduced run time by a factor of 1.3, from 16.6 MCPS to 12.8 MCPS, which is 2.29 times faster than the original version. The development time for this phase was 2.5 man-months. On this project, these improvements were performed after the initial C optimization, which resulted in some functions being C-optimized twice, both before and after the algorithmic changes. Time is gained by avoiding this duplication of effort if the development team has sufficient experience with the algorithms involved to identify the key functions before the first optimization. The time gained in our project would have been around 1 man-month.

Assembly implementation should be the final phase of development because it is very difficult to change the structure of an implementation in assembly, especially after algorithmic changes. Functions targeted for assembly implementation should be primarily those for which the C compiler cannot produce effective code, and those which require a large number of cycles. Particular emphasis should be placed on loops, where each cycle saved results in a significant reduction in execution time.

The final phase of the vocoder project was the implementation of a restricted number of critical functions in assembly. The optimized C implementation proved to be very useful as a reference for the assembly implementation. It confirmed that the intended modifications do not violate bit-exactness requirements and served as pseudo-code for algorithm description. Selection of the functions for this phase was based primarily on the difference between the performance prediction and actual performance of the generated code for each function.

Target performance was achieved by optimizing only three functions in assembly, with a development time for this phase of one man-month.

We investigated two approaches for writing assembly code. The first one was to write optimized C code and then manually correct the compiler-generated assembly. This approach is most useful when the compiled code is close to optimum, requiring only minor improvement. If the generated assembly of the optimized code is not close to optimum, direct implementation in assembly based on optimized C is recommended.

Implementing 18 functions in assembly reduced run time from 12.8 MCPS to 8.44 MCPS, a factor of 1.52, with an effort of five man-months. This version is 3.47 times faster than the initial ported version. Code size was reduced to 36.2 KB compared to the initial ported code size, 37.9 KB. This version handles up to 35 channels simultaneously, with a total memory requirement (vocoder software plus data for 35 channels) of less than 260 KB.

These results confirm that the SC140 core offers a compiler-friendly architecture and that development should focus more on C development than assembly. The decoder implementation meets the target using only optimized C. With the equivalent of less than 10 percent of code in assembly and 90 percent in optimized C code, we can meet the project target. Actually, the three most time-consuming functions implemented in assembly account for only 9.5 percent of the total number of lines in the final optimized C version. Our project revealed the importance of developing code in C, and only a few functions (for which the compiler does not generate the best code) should be implemented in assembly.

7 References

[1] *International Telectommunications Union, ITU-T G.729 Recommendation: Coding of Speech at 8kbit/s using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP), (1996).*

[2] *CCITT Blue Book*. The International Telegraph and Telephone Consultative Committee. CCITT, (Geneva: 1989).

[3] *A Practical Handbook of Speech Coders*, R. G. Goldberg and L. Riek (CRC Press: 2000). ISBN 0-8493-8525-3.

[4] *Vocoder Intelligibility and Quality Test Methods*, J. Tardelli and E. Kreamer. IEEE International Conference Acoustics. Sp. Signal Processing, 1996, pp.1145-1148.

[5] "The Implementation of G.729 Speech Coder on a 16-bit DSP Chip for the CDMA IMT_2000 System," J. Kim et al. I*EEE Trans. on Consumer Electronics*, vol. 45, no. 2, May 1999, pp. 443-448.

[6] "A New Low Bit Rate Low Delay Algebraic CELP (ACELP) Coder," R. El-Kouatly and S. H. El-Ramly. *Seventeenth National Radio Science Conference*, Feb. 22–24, 2000, Minufiya University, Egypt.

[7] *ITU-T P.810 Recommendation*. Modulated noise reference unit (MNRU), (1996).

[8] *SC140 DSP Core Reference Manual* (order number, MNSC140CORE/D).

[9] *SC100 C Compiler User's Manual* (order number, MNSC100CC/D).

[10] *SC100 Assembly Language Tools User's Manual* (order number, MNSC100ALT/D).

[11] *SC100 Application Binary Interface Reference Manual* (order number, MNSC100ABI/D).

[12] *StarCore Multisample Programming Technique Application Note* (order number, STCR140MLAN/D)

[13] *GSM EFR Vocoder on StarCore 140*, Dror Halahmi, Sharon Ronen, Yariv Mishlovsky, Assaf Naor, Shlomo Malka, Amit Gur, Haim Rizi (ICSPAT: 1999).

Appendix A
Selected C and Pseudocode Listings

A.1 Norm_Corr() prototype

Example 9. Norm_Corr() prototype

```
void Norm_Corr(Word16 exc[], Word16 xn[], Word16 h[], Word16 L_subfr,
               Word16 t_min, Word16 t_max, Word16 corr_norm[])
{
#pragma align *exc 8
#pragma align *xn 8
#pragma align *h 8

  Word16 i,j,k,exv;
  Word32 s0, s1;
  Word32 sa, sb, sc;

  Word16 excf[L_SUBFR];
#pragma align excf 8
  Word16 scaling, h_fac, *s_excf, scaled_excf[L_SUBFR];
#pragma align scaled_excf 8
#pragma align *s_excf 8
  Word16 factor[L_SUBFR];
#pragma align factor 8

…
}
```

A.2 Norm_Corr() pseudocode

Example 10. Norm_Corr() pseudo-code

```
{
 k=-t_min;

 /* compute the filtered excitation for the first delay t_min */
 Convolve(&exc[k],h,excf);

 /* Compute energy of excf[] for danger of overflow */
 E=0;
 for(j=0;j<40;j++)
        E = E + excf[j] * excf[j];

 h_fac=3;
 scaling=0;
 s_excf=excf;

 if(E>2^26)
 {
  h_fac-=2;
  scaling+=2;
  s_excf=scaled_excf;
```

```
      /*compute scaled_excf and recompute energy*/
      E=0;
      for(j=0;j<40;j++)
      {
        s_excf[j]=excf[j] >> 2;
        E = E + s_excf[j] * s_excf[j];
      }
    }

   /*loop for every possible period*/
    for(i=t_min; i<t_max; i++)
    {
      /*compute 1/sqrt[E]}*/
      E1 = Inv_sqrt(E);

      /* Compute correlation between xn[] and excf[] */
      C=0;
      for(j=0; j<40; j++)
           C = C + xn[j] + s_excf[j];

      /* Normalize correlation = correlation * (1/sqrt(energy)) */
      corr_norm[i] = E1 * C;      /*32bit multiplication*/

      /* modify the filtered excitation excf[] for the next iteration */
      k--;
      for(j=0; j<40; j++)
           factor[j] = (exc[k] * h[j]) << h_fac;

      /*energy and new excf[]*/
      s_excf[0] = exc[k] >> scaling;
      E = s_excf[0] * s_excf[0];
      for(j=1; j<40; j++)
      {
        s_excf[j] += factor[j];
        E = E + s_excf[j] * s_excf[j];
      }
    }

    /*compute 1/sqrt[E]}*/
    E1 = Inv_sqrt(E);

    /* Compute correlation between xn[] and excf[] */
    C=0;
    for(j=0; j<40; j++)
           C = C + xn[j] + s_excf[j];

    /* Normalize correlation = correlation * (1/sqrt(energy)) */
    corr_norm[i] = E1 * C;      /*32bit multiplication*/

   }
```

A.3 Lag_max() - C code

Example 11. Lag_max() C code

```
Word16 Lag_max(        /* output: lag found                                 */
  Word16 signal[],     /* input : signal used to compute the open loop pitch */
  Word16 L_frame,      /* input : length of frame to compute pitch          */
  Word16 lag_max,      /* input : maximum lag                               */
  Word16 lag_min,      /* input : minimum lag                               */
  Word16 *cor_max)     /* output: normalized correlation of selected lag    */
{
#pragma align *signal 8
/* in the original version, signal pointed to scaled_signal[PIT_MAX]
 * now, it points to scaled_signal[0]
 */
  Word16  i, j; /* loop indexes */
  Word16  *p, *ref_signal; /* pointers in the signal array */
  Word32  max; /* maximum corelation and its position */
  Word16  p_max;
  Word32  s0, s1; /* partial sums used in split-summation energy computation */

  /* check if the signal array is 8-byte aligned */
  assert (((int) signal & 7) == 0);

  /* initialize the maximum value and the position of the maximum */
  max = MIN_32;
  p_max = 0;

  /* initialize the reference signal pointer used in correlations */
  ref_signal = &signal[lag_max];

  /* check the necessary condition for 4 samples in parallel */
  assert (((lag_max - lag_min + 1) & 3) == 0);

  for (i = lag_max - lag_min - 3; i >= 0; i-=4)
  {
    /*
     * this loop computes four correlations in parallel
     * to create the framework for better ALU usage
     */

    /*
     * declare the variables as close as possible to their usage
     */
    Word32 c0, c1, c2, c3;         /* correlations */
    Word16 rs;                     /* current value from the reference signal */
    Word16 sig0, sig1, sig2, sig3; /* four values from past signals */

    /* initialize the correlations */
    c0 = 0;
    c1 = 0;
    c2 = 0;
    c3 = 0;

    /* fetch needed values for first correlations */
    rs = ref_signal[0];
    sig0 = signal[i+0];
    sig1 = signal[i+1];
    sig2 = signal[i+2];
    sig3 = signal[i+3];

    /*
     * Unroll next loop into four iterations to reuse previously fetched values.
     * The reference signal is not aligned, so one reference sample is used
     * for all computations in a phase.
     */
    for (j=0; j < L_FRAME; j+=4)
    {
      /* in each phase:
       * - compute a step in the correlations, and
       * - fetch next values
       */
      c0 = L_mac(c0, rs, sig0);
      c1 = L_mac(c1, rs, sig1);
      c2 = L_mac(c2, rs, sig2);
      c3 = L_mac(c3, rs, sig3);
      rs = ref_signal[j+1];
      sig0 = signal[i+j+4];

      c0 = L_mac(c0, rs, sig1);
      c1 = L_mac(c1, rs, sig2);
```

```
        c2 = L_mac(c2, rs, sig3);
        c3 = L_mac(c3, rs, sig0);
        rs = ref_signal[j+2];
        sig1 = signal[i+j+5];

        c0 = L_mac(c0, rs, sig2);
        c1 = L_mac(c1, rs, sig3);
        c2 = L_mac(c2, rs, sig0);
        c3 = L_mac(c3, rs, sig1);
        rs = ref_signal[j+3];
        sig2 = signal[i+j+6];

        c0 = L_mac(c0, rs, sig3);
        c1 = L_mac(c1, rs, sig0);
        c2 = L_mac(c2, rs, sig1);
        c3 = L_mac(c3, rs, sig2);
        rs = ref_signal[j+4];
        sig3 = signal[i+j+7];
      }
      /* The function must favor the correlation with the maximum lag */
      /* Test if the correlation computed is greater than previous maximum */
      if (c3 > max)
      {
        max   = c3;
        p_max = i+3;
      }
      if (c2 > max)
      {
        max   = c2;
        p_max = i+2;
      }
      if (c1 > max)
      {
        max   = c1;
        p_max = i+1;
      }
      if (c0 > max)
      {
        max   = c0;
        p_max = i;
      }
    }

    /* Compute energy with split-summation.
     * Split-summation is performed with 2 partial sums because the array
     * is not aligned.
     */

    s0 = 0;
    s1 = 0;
    p = &signal[p_max];
    for (i=0; i < L_frame; i+=2)
    {
      s0 = L_mac(s0, p[i+0], p[i+0]);
      s1 = L_mac(s1, p[i+1], p[i+1]);
    }
    s0 = L_add(s0, s1);

    /* compute 1/sqrt(energy), with the result in Q30 */

    s0 = Inv_sqrt(s0);

    /* compute max = max/sqrt(energy)         */
    /* This result will always be 16 bit-aligned. */

    s0 = Mpy_32_new(max, s0);
    *cor_max = extract_l(s0);

    return (lag_max - p_max);
  }
```

Appendix B Selected Assembler Operations

B.1 32-Bit DPF Operations

Multiplying a 16-bit integer by a 32-bit DPF can be viewed as the multiplication of a Q31 number by a Q15 number, with the result in Q31 format:

$$L\_32 = (hi1 \times lo2) << 1 + [(lo1 \times lo2) >> 15] << 1$$

To perform this operation most efficiently using StarCore instructions, we first analyze the operation

$$(a >> p) << q$$

where $q < p$

After the initial shift right, a loses the least significant p bits. After the second shift, a receives q zeros in the least significant q bits. This is equivalent to zeroing the p least significant bits of a, then right-shifting the result by $(p\text{-}q)$ bits:

$$[(a >> p)] << q \equiv (a \ \& \ \underbrace{\underbrace{1...1}_{n-p\ bits}\underbrace{0...0}_{p\ bits}}_{n\ bits}) >> (p - q)$$

where

$$\underbrace{\underbrace{1...1}_{n-p\ bits}\underbrace{0...0}_{p\ bits}}_{n\ bits} = [2^n - (2^p - 1) - 1] \bmod 2^n = -2^p$$

With this, our formula becomes:

$$L\_32 = (hi1 \times lo2) << 1 + [(lo1 \times lo2) \ \& \ \$fffe0000] >> 14$$

However, in StarCore's registers the lower half of an ITU 32-bit DPF word is shifted left one bit, and the result of a StarCore multiply operation is shifted left one bit. Thus, in a StarCore implementation the formula becomes

$$L\_32 = (hi1 \times lo2)_{SC} + \{[((lo1_{SC} >> 1) \times lo2)_{SC} >> 1] \ \& \ \$fffe0000\} >> 14$$

$$= (hi1 \times lo2)_{SC} + [(lo1_{SC} \times lo2)_{SC} \ \& \ \$fffe0000] >> 16$$

where *SC* denotes StarCore-specific operations or value representations.

Example 12 illustrates how this formula can be applied to the multiplication of a 32-bit DPF and a 16-bit integer in StarCore. In this example, *d0* contains the 32-bit DPF, *d1.h* contains the 16-bit integer, *d3* contains -2^{15}=\$fffe0000, and the result is stored in *d2*.

Example 12. StarCore Multiplication of 32-Bit DPF and 16-Bit Integer

```
…
mpyus d0,d1,d2
and d3,d2
dmacss d0,d1,d2
…
```

Multiplying two 32-bit DPF numbers can be viewed as the multiplication of two Q31 numbers, with the result in Q31 format:

$$L\_32 = (hi1 \times hi2) << 1 + [(hi1 \times lo2) >> 15 + (lo1 \times hi2) >> 15] << 1$$

Code Example 13 illustrates how this formula can be applied to the multiplication of two 32-bit DPF numbers in StarCore. In this example, *d0* and *d1* contain the 32-bit DPF values, *d4* contains -2, and the result is stored in *d2*.

Example 13. StarCore Multiplication of Two 32-Bit DPF Values

```
…
mpysu d0,d1,d2                  mpyus d0,d1,d3
asrw d2                         asrw d3
and d4,d2                       and d4,d3
add d2,d3,d2
mac d0,d1,d2
…
```

Note: The least significant bit of a DPF value is always zero. To ensure that the result of the operation described here conforms to DPF format, the least significant bit should be zeroed by using a bit wise *and* or *bmclr* instruction that uses the bit mask $2^{31} - 1 = -2$.

B.2 Lag_max() ASM code

Example 14. Lag_max() ASM Code

```
;* ******************************************* PURPOSE *************************************** *;
;*                                                                                              *;
;* PURPOSE:                                                                                     *;
;*          This is the open-loop pitch estimation                                              *;
;*                                                                                              *;
;* ***************************************** INPUT AND OUTPUT ******************************** *;
;*                                                                                              *;
;* INPUT:                                                                                       *;
;*      r0 - input : signal used to compute the open loop pitch (Word16 * signal)               *;
;*                   signal[-pit_max] to signal[-1] should be known                             *;
;*      d1 - input : length of frame to compute pitch (Word16 L_frame)                          *;
;*   sp-10 - input : maximum lag (Word16 pit_max)                                               *;
;*   sp-12 - input : minimum (Word16 pit_min)                                                   *;
;* OUTPUT:                                                                                      *;
;*      d0 - output : lag found(Word16)                                                         *;
;*    sp-12 - output : normalized correlation of selected lag                                   *;
;* CALLED BY:  pitch_ol() function                                                              *;
;*                                                                                              *;
;* ******************************************* RESOURCES *********************************** *;
;*                                                                                              *;
;* REGISTERS USED:                                                                              *;
;*      d1-d15;                                                                                 *;
;*      r0-r3;                                                                                  *;
;*      n3;                                                                                     *;
;* CYCLE COUNT:                                                                                 *;
;*     Typical=  1115                                                                           *;
;*     Maximum=  1835                                                                           *;
;* CODE SIZE:   670                                                                             *;
;*                                                                                              *;
;* ASSEMBLER SYNTAX:                                                                            *;
;* Star*Core 100 Assembler  Version 6.3.58 spec 0.63                                            *;
;* asmsc100 -q -l nul -s all -o elf -b Lag_max.eln Lag_max.asm                                  *;
;*                                                                                              *;
;* ******************************************************************************************* *;
        section .text local
TextStart_lag_max
        global _Lag_max
        align   16

_Lag_max   type   func

    move.w (sp-10),d1         move.w (sp-12),d2

;* function is ABI compliant so r6,r7 and d6,d7 must be saved *;
    [
     push r6                  push r7
     sub d2,d1,d1             asl d1,d2
    ]
```

```
;* lag_mag-lag_min is multiplied by 2 because variables used are Word16 *;
    [
     push d6                 push d7
     sub #3,d1
    ]
;* two maxima are used                          *;
;* initialized with MIN_32 value (-2147483648) *;
    [
     move.l #-2147483648,d0   move.l d2,r1
     asl d1,d1                tfr d1,d2
    ]
    [
    move.l d1,r2              asrr #2,d2
    dosetup2 L2start
    ]
    [
     adda r0,r1               inc d2
     move.l #2,n3
    ]
;* loop count is (lag_max-lag_min+1)/4         *;
    [
     adda r0,r2               sub #1,d2
;* Fake comparison between d0 and d1 to initialize the T bit to FALSE for first loop2 iteration *;
     doen2 d2                 cmpeq d0,d1
    ]
    [
     move.w #8,d12            tfr d0,d1
     asll #3,d2               dosetup3 L3start
    ]
    falign
    loopstart2
L2start
;************************************************************************************* *;
;* There must be a minimum of 2 instruction sets between doen3 and loopstart3           *;
;* so the final comparison of the max corr computation has been moved here.             *;
;* This section uses a loop unrolling technique.                                        *;
;* ************************************************************************************ *;
    [
     doen3 #20                tfra r1,r3
     tfr d8,d8                tfr d9,d9                    ; tfr d8,d8 and tfr d9,d9 - dummy
                                                           ; instruction for code alignment
;* T bit has been initialized to FALSE for first iteration. *;
    [
     ift
        tfr d4,d1
        add d12,d2,d14
     ifa
        sub #2,d2
        tfr d8,d8                           ; dummy instruction for code alignment
        move.f (r2),d4                      ; dummy instruction for code alignment
    ]
;************************************************************************************* *;
;* Compute correlations                                                                 *;
;* ************************************************************************************ *;

    [
     clr d4  clr d5  clr d6  clr d7
     move.4f (r2)+,d8:d9:d10:d11  move.f (r3)+,d3
    ]
    falign
    loopstart3
L3start
    [
     mac d3,d8,d4
     mac d3,d9,d5
     mac d3,d10,d6
     mac d3,d11,d7
     move.f (r2)+,d8
     move.f (r3)+,d3
    ]
    [
     mac d3,d9,d4
     mac d3,d10,d5
     mac d3,d11,d6
     mac d3,d8,d7
     move.f (r2)+,d9
     move.f (r3)+,d3
    ]
    [
     mac d3,d10,d4
     mac d3,d11,d5
     mac d3,d8,d6
     mac d3,d9,d7
     move.f (r2)+,d10
     move.f (r3)+,d3
    ]
    [
     mac d3,d11,d4
     mac d3,d8,d5
     mac d3,d9,d6
     mac d3,d10,d7
     move.f (r2)+,d11
     move.f (r3)+,d3
    ]
    loopend3
;* ****************************************************************************************** *;
;* Determine maximal correlation using two values—one for even and one for odd.               *;
;* Four correlations are computed in parallel using multisample technique.                    *;
;* ****************************************************************************************** *;
    cmpgt d0,d7      adda #-176,r2,r2
    [
     ift
        tfr d7,d0
        add d12,d2,d15
     ifa
        cmpgt d1,d6
        sub #2,d2
    ]
    [
     ift
        tfr d6,d1
        add d12,d2,d14
     ifa
        cmpgt d0,d5
        sub #2,d2
    ]
   [
     ift
        tfr d5,d0
        add d12,d2,d15
     ifa
       cmpgt d1,d4
       sub #2,d2
    ]
loopend2
;* final comparison
    [
     ift
        tfr d4,d1
        add d12,d2,d14
    ]
;* ****************************************************************************************** *;
;* Determine maximal correlation between the two suboptimal values.                           *;
;* Index of maximal correlation is also stored.                                               *;
;* ****************************************************************************************** *;
    cmpgt d0,d1    move.l #2,r3  tfr d15,d6   doensh3 #40
    [
     ift
        tfr d1,d0
        tfr d14,d6
     ifa
        cmpeq d0,d1
        tfr d14,d2
    ]
    [
     ift
        max d2,d6
     ifa
        tfr d0,d7                                                        ; max in d7   p_max in d6
    ]
    move.l d6,r2    adda r0,r3
;******************************************************************************************* *;
;* Compute energy for index of maximal correlation.                                           *;
;* Data is not aligned; therefore only 2 parallel macs can be done.                           *;
;******************************************************************************************* *;
    clr d0    clr d2
    adda r0,r2       adda r2,r3
    move.f (r2)+n3,d5     move.f (r3)+n3,d4
    loopstart3
    [
     mac d5,d5,d0
     mac d4,d4,d2
     move.f (r2)+n3,d5
     move.f (r3)+n3,d4
    ]
    loopend3
    add d2,d0,d0

;* 1/sqrt(energy)

    jsr _Inv_sqrt
    tfr d7,d1

;* ******************************************************************************************* *;
;* max = max/sqrt(energy)                                                                       *;
;* This result will always be a 16-bit value!                                                   *;
;* ******************************************************************************************* *;

    mpysu d0,d1,d3          mpyus d0,d1,d4
    and #$fffe0000,d3,d3     and #$fffe0000,d4,d4  asr d6,d6
    add d3,d4,d3            move.l (sp-32),r2      move.w (sp-26),d2
    dmacss d0,d1,d3         sub d6,d2,d0
    move.w d3,(r2)

Lrts
    pop  d6   pop  d7
    pop  r6   pop  r7

    rts

        global FLag_max_end
FLag_max_end
TextEnd_lag_max
        endsec
```

Appendix C
Script Sources

C.1 Perl Script for Generating Cycle Statistics

Example 15. cycles_analyzer.pl

```
#!c:/Perl/bin/perl
# DSP Center Romania, 2000

=head1
The script parses the log file generated by coder_worst_case.sc or
decoder_worst_case.sc simulator scripts.

The output of this script is a table with worst-case and average execution time
on each test vector described in the .ini file (g729coder.ini or g729decoder.ini).

The script assumes a fixed directory structure. There are two parameters:
- The first parameter is mandatory and represents the name of the module to be analyzed
  (coder or decoder)
- the second parameter establishes if the script also runs the simulation.

The log files for coder and decoder are similar in structure.
=cut

# analyzed module is received from command line
$module=<$ARGV[0]>;

# check if the module tester is available; end script if not found
$exec_file="../../../code/bin/".$module.".eld";
if (! -e $exec_file) {die "Binary file doesn't exist!!!";}

print "\nStatistics made for $module :\n\n";

# run the module tester without ini files to extract the build number and
# build date
system("runsc100 $exec_file > tmp.txt");
open (tmpfile,"tmp.txt") || die "Cannot create temporary files!!!";

# extract the build number and date
while (<tmpfile>)
{
    if(/.*build *(\d{4}).*/)
    {
       print "Build number : $1\n";
    }
    if(/.*time: *(.*)/)
    {
       print "$1\n\n";
    }
}

# close and remove the temporary file
close(tmpfile);
system("rm -f tmp.txt");

# change the current working directory to log files directory
$path="worst_case/".$module."/";
chdir $path;

# Check if the .ini file is available.
# This file is needed for both running the simulation and for extracting
# the names of the test vectors and the corresponding number of frames.
$ini_file_name="g729".$module.".ini";
open ( inifile, $ini_file_name) || die "Cannot open ini file : $!\n";

# check the second parameter; if the value is "exec", run the simulation
$execute = <$ARGV[1]>;
if ($execute eq "exec")
{
    # check the presence of the worst-case log generating script
    $cmd_file=$module."_worst_case.sc";
    if (! -e $cmd_file) {die "Inexistent worst-case command file for simulator!!!";}

    # run the simulation
    system("simsc100 $cmd_file");
}

# check and open the worst-case simulation log
$log_file_name=$module."_worst_case.log";
open ( fin, $log_file_name ) || die "Can’t open log file : $!\n";

# the comparison Perl module is needed to check if the test was passed
use File::Compare;

$frames[0]=0;
$name[0]="";
$count=1;

# create test name array from ini file
while ( <inifile> )
{
 if(/(\w+)\ +(\d+)/)
 {
   $name[$count] = $1;
   $frames[$count] = $frames[$count-1]+$2;
   $count++;
 }
}
close(inifile);

$max_cycles = 0;
$max_cycles_test = 0;
$count = 0;
$sum = 0;
$sum_test = 0;
$test_count = 1;

# print the head of the table
print "-" x 74; print "\n";
print "| Test     |    average |        worst-case |     frames |     sim. test |\n";
print "-" x 74;
print "\n";

# change the current working directory to test_cases root directory
chdir "../../../../test_cases/";

=head2
Example lines from the worst-case log:

Break #1 _g729_encode s ;dev:0 pc:134b6 cyc:87723
p:$000134b6 94c0 9e20 9f20 = push r6 & push r7
Break #2 _frame_end s ;dev:0 pc:17ca0 cyc:172248
p:$00017ca0 9f71 = rts
Break #1 _g729_encode s ;dev:0 pc:134b6 cyc:175396
p:$000134b6 94c0 9e20 9f20 = push r6 & push r7
Break #2 _frame_end s ;dev:0 pc:17ca0 cyc:275577
p:$00017ca0 9f71 = rts

=cut

# Start analyzing the worst-case log.
# Extract the number of cycles before and after processing a frame.
while ( <fin> )
{

# the start of frame processing breakpoint
if (/Break #1/)
        {
         $start_cyc = $_;
         $start_cyc =~ s/.*cyc:(\d+)/$1/;
        }

# the end of frame processing breakpoint
if (/Break #2/)
        {
           $end_cyc = $_;
           $end_cyc =~ s/.*cyc:(\d+)/$1/;

           # compute the number of cycles needed to process a frame
           $cycles = $end_cyc - $start_cyc;

           # compute the sums needed for averages (global and test average)
           $sum += $cycles;
           $sum_test += $cycles;

           $count++;            # one more frame processed
           # if the time per last frame is greater than current maximum, keep it
           if ($cycles > $max_cycles_test)
           {
               $max_cycles_test = $cycles;
           }

           # run summarizy routine if end of test case
           if ($count == $frames[$test_count])
           {
             $frames_in_test = $frames[$test_count]-$frames[$test_count-1];

             # print a new table line
             print "| $name[$test_count]";
             print " " x (9-length($name[$test_count]));
             $average = int($sum_test / ($frames_in_test));
             print "|";
             print " " x (11-length($average));
             print "$average";
             print " |";
             print " " x (18-length($max_cycles_test));
             print "$max_cycles_test";
             print " |";
             print " " x (11-length($frames_in_test));
             print "$frames_in_test";
             print " |";

             # form the name of the test case file depending on module
             if ($module eq "coder")
             {
                 $file_ref="reference/ref_bit/".$name[$test_count].".bit";
                 $file_result="results/".$name[$test_count].".bit";
             }
             if ($module eq "decoder")
             {
                 $file_ref="reference/ref_pst/".$name[$test_count].".pst";
                 $file_result="results/".$name[$test_count].".pst";
             }

             # compare the simulation result with the reference output
             if (compare($file_ref,$file_result)==0)
             {
                 print "       passed";
             }
             else
             {
                 print "       FAILED";
             }
             print "  | \n";

             # record the worst-case value and test name
             if ($max_cycles_test > $max_cycles)
             {
                 $max_cycles = $max_cycles_test;
                 $max_name   = $name[$test_count];
             }

             # reinitialize the counters for next test case
             $sum_test = 0;
             $max_cycles_test = 0;
             $test_count++;
           }
        }
}

# close all files
close(fin);


#########################################
# Overall statistics                    #
#########################################
$average = int($sum / $count);
print "-" x 74;
print "\nOverall Test\n"; print "-" x 20;
print "\nThe average    =";
print " " x (10-length($average)); print "$average";
print "\nThe worst_case =";
print " " x (10-length($max_cycles)); print "$max_cycles (in $max_name)\n";
```

C.2 Simulator Command Files for Worst-Case and Average Analysis

C.2.1 Command File for Coder (coder_worst_case.sc)

Example 16. coder_worst_case.sc

```
display off
break off
output off
input off
load ..\..\..\..\..\code\bin\coder.eld
radix d

break _g729_encode s
break _frame_end s
break _exit

log s coder_worst_case.log -o
go
quit
```

C.2.2 Command File for Decoder (decoder_worst_case.sc)

Example 17. decoder_worst_case.sc

```
display off
break off
output off
input off
load ..\..\..\..\..\code\bin\decoder.eld
radix d

break _g729_decode s
break _frame_end s
break _exit

log s decoder_worst_case.log -o
go
quit
```

C.3 Perl Script for Generating Stack Statisticsstack.pl

```
#!c:/Perl/bin/perl
# DSP Center Romania, 2000

=head1
The script parses the log file generated by coder_measure_stack.sc and
decoder_measure_stack.sc simulator scripts.

The output of this script are the maximum values for coder and decoder stack
size.

The script assumes a fixed directory structure. The only parameter it receives
establishes if the script runs also the simulation or not.

The log files are similar in structure, thus the same basic block is repeated
twice, once for the coder module and once for the decoder module.
=cut

# analyzed module is coder
$module="coder";

# the module needed to run the simulation and to extract build number and date
$exec_file="../../../../code/baseline/".$module.".eld";
```

```
# run the module tester to extract build number and date
# g729coder.ini has to be renamed temporary to not perform the test
system("mv g729coder.ini g729coder.tmp");
system("runsc100 $exec_file > tmp.txt");
open (tmpfile,"tmp.txt") || die "Cannot create temporary files!!!";
while (<tmpfile>)
{
    if(/.*build *(\d{4}).*/)
    {
       print "Build number : $1\n";
    }
    if(/.*time: *(.*)/)
    {
       print "$1\n\n";
    }
}
close(tmpfile);
system("rm -f tmp.txt");

system("mv g729coder.tmp g729coder.ini");

# run the simulation if parameter value is "exec"
if ( $ARGV[0] eq "exec" ) {
  system("simsc100 ".$module."_measure_stack.sc");
}

# open the log file of the module
$log_file_name=$module."_stack.log";

open ( fin, $log_file_name ) || die "Can't open log file : $!\n";

$stack_top=0;
$stack_base=0xffffffff;
$begin_flag=1;

# Parse the log file.
# - First, keep the first value of the stack pointer in the $stack_base.
# - Find the maximum of the other values => $stack_top.
# The stack pointer for the C code is esp.
# Typical display line for esp:
#                      esp={00000164264}
while (<fin>) {
  if (/.*esp.*\{(.*)\}/ ) {
    if ($begin_flag == 0) {
      if ( $1 > $stack_top ) {
        $stack_top = $1;
      }
    } else {
      $stack_base = $1;
      $begin_flag = 0;
    }
  }
}

# output stack dimension
$encoder_stack = $stack_top - $stack_base;
print "Encoder stack size = ".$encoder_stack." bytes.\n";

# analyzed module is decoder
$module = "decoder";

# run the simulation if parameter value is "exec"
if ( $ARGV[0] eq "exec" ) {
  system("simsc100 ".$module."_measure_stack.sc");
}

# open the log file of the module
$log_file_name=$module."_stack.log";

open ( fin, $log_file_name ) || die "Can't open log file : $!\n";

$stack_top=0;
$stack_base=0xffffffff;
$begin_flag=1;

# Parse the log file.
# - First, keep the first value of the stack pointer in the $stack_base.
# - Find the maximum of the other values => $stack_top.
# The stack pointer for the C code is esp.
# Typical display line for esp:
#                      esp={00000164264}
while (<fin>) {
  if (/.*esp.*\{(.*)\}/ ) {
    if ($begin_flag == 0) {
      if ( $1 > $stack_top ) {
        $stack_top = $1;
      }
    } else {
      $stack_base = $1;
      $begin_flag = 0;
    }
  }
}

# output stack dimension
$decoder_stack = $stack_top - $stack_base;
print "Decoder stack size = ".$decoder_stack." bytes.\n";
```

C.4 Simulator Command Files for Stack Analysis

Example 18. coder_measure_stack.sc

```
display off
break off
output off
input off
radix u esp

load ..\..\..\..\code\baseline\coder.eld
display on esp
break #1 _g729_encode
break #2 _frame_end s
break #3 _exit

log s coder_stack.log -o
go
break #4 w esp s
break #1 _g729_encode s
go
quit
```

C.5 Code and Data Size from Map Files

The linker can generate a map file describing the memory map of the application (encoder or decoder). This option is enabled either in the linker command line using the –M (or –Map <file>) option or from the compiler shell using the –dm <file> option. The map file provides a great deal of information about both the vocoder code and the application that runs it, and some effort is required to separate the vocoder information from the application information. The map file provides size and memory location information for all global variables and functions, as well as total memory allocated to each object file. The size of local (static) variables and functions are derived from this information. **Example 19** lists fragments from a map file generated for the encoder tester application. The line numbers on the left are manually inserted for reference.

Example 19. Fragments from the Vocoder Map File

```
         Value        Size     Symbol
…
1  0x00000200       15464  Section: .data
…
2  0x000005f0        5956      Section: .data(obj/tab_ld8k.eln)
3  0x000005f0                      _lspcb1
4  0x00000ff0                      _lspcb2
5  0x00001270                      _hamwindow
…
6  0x00010000       66592  Section: .text
…
7  0x000159b0        4638      Section: .text(obj/coder_ld8k.eln)
8  0x00016030                      _Coder_ld8k
9  0x00016bce                      FCoder_ld8k_end
10 0x00016bd0          92      Section: .text(obj/corr.eln)
11 0x00016bd0                      _Corr_40
12 0x00016c2c                      FCorr_40_end
…
13 0x000185c4           0      Section: .text(obj/tab_ld8k.eln)
…
```

The global data for all object files generated from the C source code is listed in the `.data` section, and the text code in the `.text` section. In the final executable files, the `.data` and `.text` sections are linked together. The memory start addresses of these sections are listed in the map file (lines 1 and 6 in **Example 19**).

The `.data` section contains a subsection for each object file that lists the start address of each global variable declared in that object file (lines 3, 4, and 5 in **Example 19**) and the total data size of that object file (line 2). The size of each global variable is derived by subtracting its address from the address of the next variable listed. The first line of the `.data` section also shows the total size of all global variables in all object files.

The `.text` section contains a subsection for each object file that lists all global functions declared in the file as well as the start address and total size of the code section for that file. The first line of the `.text` section indicates the total size of all code for all object files. The start and end of each global function are denoted as `_functionName` and `FfunctionName_end` respectively; the difference in their addresses indicates the size of the function. Static functions in a section are not listed in the map file, but their total size in each section is derived from the listed information. For example, the `coder_ld8k.eln` section in **Example 19** is 4638 bytes (size column in line 7). This section contains one global function, `coder_ld8k(),` which is 2974 bytes (difference in the value column of lines 8 and 9). The size of the static functions in this section is 4638 – 2974 = 1664 bytes.

The data in the map file are very detailed, but it is difficult to distinguish the size of the code generated for the vocoder by the rest of the tester.

An object file can contain both global data and global functions, global functions only, or global data only. For example, the `tab_ld8k.eln` object file contains data only (see lines 2 and 13 in **Example 19**).

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

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

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

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

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

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

Document Order No.: AN2094
Rev. 1
12/2004