Speech Coder Filters Using the StarCore™ SC140/SC1400 Cores

By Emilian Medve

This application note describes the implementation of the Residu(), Convolve() and Syn_filt() functions on the StarCore SC140. These functions perform filtering on speech signals in the ETSI GSM EFR, ITU G.729 and other voice coding schemes. These functions are typical digital signal processing (DSP) operations and are well-suited for the StarCore DSP. G.729 uses Conjugate Structure - Algebraic Code Excited Linear Prediction (CS-ACELP) while GSM EFR uses Algebraic Code Excited Linear Prediction (ACELP). A brief description of GSM EFR, G.729 and the StarCore compiler tools is followed by a discussion of the various methods used to optimize the Residu(), Convolve() and Syn_filt() functions. Finally, some benchmarks are presented on the optimized functions.

CONTENTS
1 Speech Coder Basics ...............................................2
  1.1 GSM EFR .................................................................2
  1.2 G.729 .................................................................2
  1.3 The Residu() Function ............................................2
  1.4 The Convolve() Function ........................................3
  1.5 The Syn_filt() Function .........................................3
2 StarCore SC140/SC1400 Compiler ................................4
  2.1 StarCore Compiler ................................................4
3 Optimization Phases and Methods ..........................5
  3.1 C Optimizations ....................................................5
  3.2 Assembly .............................................................7
4 Optimization of the Residu() Function ...................7
  4.1 Assembly ............................................................9
  4.2 Results ............................................................10
5 Optimization of the Convolve() Function .............10
  5.1 Assembly ........................................................13
  5.2 Results ........................................................15
6 Optimization of the Syn_filt() Function ...............15
  6.1 Optimized C ........................................................16
  6.2 Assembly ........................................................19
  6.3 Results ...........................................................22
7 Performance Comparison ......................................22
8 Conclusions ...........................................................22
9 References ............................................................23

© Freescale Semiconductor, Inc., 2005. All rights reserved.
1 Speech Coder Basics

This section covers the major standards and functions pertaining to speech coders.

1.1 GSM EFR

The Global System for Mobile Communication Enhanced Full Rate (GSM EFR) standard is defined by the European Telecommunication Standards Institute (ETSI) and is used for coding of speech at 12.2 kbps for digital cellular communications in Europe. GSM EFR uses the ACELP coding scheme. The GSM EFR encoder is based on code excited linear prediction (CELP) which is an analysis-by-synthesis algorithm. For every 20 ms speech frame, the speech signal is analyzed to extract the parameters of the CELP model. Each speech frame is equally divided into 4 subframes of 5 ms each (40 samples) at the sampling frequency of 8 KHz. The parameters for each frame are quantized into 244 bits, resulting in a transmission rate of 12.2 kbps. The GSM EFR decoder decodes and synthesizes the speech using the same parameters as the CELP model which performs the encoding.

For additional information on GSM EFR, refer to the ETSI SMG2 ITU-T Recommendation GSM 06.60, Enhanced Full Rate Speech Transcoding [2].

1.2 G.729

G.729 is defined by the Telecommunication Standardization Sector of the International Telecommunication Union (ITU-T) and is used in multimedia digital simultaneous voice and data applications. G.729 uses the CS-ACELP algorithm, which is an analysis-by-synthesis algorithm and also belongs to the CELP class of speech coding algorithms. For every 10 ms speech frame, the speech signal is analyzed to extract the parameters of the CELP model. Each speech frame is divided into 2 equal subframes. Most parameters are determined per subframe of 5 ms (40 samples) each. These parameters are quantized into 80 bits, resulting in a transmission rate of 8 kbps. The G.729 decoder converts the digitized signal back to an analog signal using a similar approach.

For additional information on G.729, refer to the ITU Recommendation G.729, Coding of Speech at 8 kbit/s Using Conjugate Structure Algebraic Code Excited Linear Prediction (CS-ACELP) [1].

1.3 The Residu() Function

The Residu() function is used to find the residual signal r(n) which is needed to find the target vector for the adaptive codebook search in both the G.729 (see [1], Section 3.6, “Computation of the Target Signal”) and GSM EFR (see [2], Section 5.5, “Target Signal Computation”) speech coders. The linear prediction (LP) residual signal is filtered through the combination of synthesis filter $1/\hat{A}(z)$ and the weighting filter $W(z)=A(z/\gamma_1)/A(z/\gamma_2)$. After determining the excitation for the subframe, the initial states of these filters are updated by filtering the difference between the LP residual and excitation. The LP residual signal is also used in the adaptive codebook search to extend the past excitation buffer. The LP residual is given by

$$ r(n) = s(n) + \sum_{i=1}^{10} \hat{a}_i \cdot s(n-i) \quad n = 0, \ldots, 39 \quad \text{Equation 1} $$

where $s(n)$ is the pre-processed speech signal, and $\hat{a}_i$, $i = 1, \ldots, 10$, are the (quantized) LP coefficients.
In both the GSM EFR and G.729 applications, the Residu() function is represented in Q12 notation, which allows speech information for fixed-point DSPs to be handled accurately. In this notation, the decimal point is placed between bits 11 and 12. For example, “$1000” in Q12 notation represents the number 1.0. Therefore equation Equation 1 is modified as follows:

\[
r(n) = s(n) \cdot a_0 + \sum_{i=1}^{10} \hat{a}_i \cdot s(n-i) \quad n = 0, \ldots, 39
\]

Equation 2

where \( a_0 \) is equal to \$1000, and the LP coefficients \( \hat{a}_i, i = 0, \ldots, 10 \) are represented in Q12 format.

### 1.4 The Convolve() Function

In G.729 (see [1], Section 3.7, “Adaptive-Codebook Search”) and in GSM EFR (see [2], Section 5.6, “Adaptive Codebook Search”) the Convolve() function is used to compute the convolution of the past excitation signal with the impulse response of the weighting and synthesis filters. The impulse response \( h(n) \) is computed for each subframe by filtering a signal consisting of the coefficients of the filter \( A(z/\gamma_1) \) extended by zeros through the two filters \( 1/\hat{A}(z) \) and \( 1/A(z/\gamma_2) \). The convolution is given by

\[
y(n) = \sum_{i=0}^{n} x(i) \cdot h(n-i) \quad n = 0, \ldots, 39
\]

Equation 3

### 1.5 The Syn_filt() Function

The Syn_filt() function implements a synthesis filter that is based on a 10th order linear prediction filter in both G.729 (see [1], Section 3.2, “Linear Prediction and Quantization”) and GSM EFR (see [2], Section 4.3 “Principles of the GSM Enhanced Full Rate Speech Encoder”). The LP synthesis filter is defined as

\[
\frac{1}{\hat{A}(z)} = \frac{1}{1 + \sum_{i=1}^{10} \hat{a}_i \cdot z^{-i}}
\]

Equation 4

where \( \hat{a}_i, i = 1, \ldots, 10 \), are the (quantized) LP coefficients. Thus, the recurrence formula is

\[
y(n) = x(n) - \sum_{i=1}^{10} \hat{a}_i \cdot y(n-i) \quad n = 0, \ldots, 39
\]

Equation 5

In both GSM EFR and G.729, the Syn_filt() function is represented in Q12 notation. Therefore equation Equation 5 is modified as follows

\[
y(n) = x(n) \cdot a_0 - \sum_{i=1}^{10} \hat{a}_i \cdot y(n-i) \quad n = 0, \ldots, 39
\]

Equation 6

where \( a_0 \) is equal to \$1000. The LP coefficients \( \hat{a}_i, i = 0, \ldots, 10 \) are represented in Q12 format.
2  StarCore SC140/SC1400 Compiler

The StarCore SC140/SC1400 DSP core addresses the key market needs of next-generation DSP applications. This programmable DSP core offers exceptional performance, low power consumption, efficient compilability, and compact code density. The SC140 core efficiently deploys a variable-length execution set execution model with maximum parallelism by allowing two address generation and four data arithmetic logic units to execute multiple instructions in a single clock cycle. An optimized C compiler tool accepts C source code and produces StarCore assembly language source code. The C compiler includes a shell program, a high-level optimizer (largely platform-independent), a low-level (platform-dependent) optimizer and other utilities. The optimizers take advantage of the features specific to the StarCore architecture. General optimizations can be applied to any ANSI C code. This application note demonstrates that the StarCore compiler tools can achieve competitive performance compared to the estimated hand-coded assembly. For details on the StarCore SC140 architecture refer to the SC140 DSP Core Reference Manual [3]

2.1  StarCore Compiler

The StarCore compiler offers a variety of features to optimize the output code for speed and/or code size. These features are invoked with command line options.

2.1.1  Compiler Features

The StarCore compiler includes a powerful high-level optimizer that features aggressive loop optimizing mechanisms. These mechanisms are based primarily on an extended induction process which can handle cross-loop induction variables. They tend to move a substantially greater amount of code out of loop nests than a typical induction process, which primarily optimizes the inner-most loops.

The transformations performed by the optimizer include

- Standard optimizations (constant and copy propagation, common subexpression elimination, etc.)
- Detection and normalization of loops and hardware-mapable loops
- Invariant code motion
- Scalarization
- Pointer promotion
- Single-loop induction processes, including transformation of multi-step induction variables, composed induction variables, wrap-around variables, modulo induction, simplification of redundant induction variable, etc.
- Detection of sequences of memory accesses, access packing, and simplification of redundant memory accesses
- Cross-loop induction mechanisms, including recognition of iteration space
- Loop reordering and restructuring features, loop peeling, and loop collapse.

In addition to its extensive optimization capabilities, the compiler offers other features that make it ideal for DSP software development, including

- Compliance with the ANSI C standard
- Intrinsic function support for ITU/ETSI primitives—saturating, non-saturating, and double-precision arithmetic
Run-time libraries and environments
Easy integration of assembly code into C code

2.1.2 Command Line Options
The StarCore compiler shell offers four basic optimization levels which maintain a balance between code density and speed. Each level is invoked by a specific command line option as follows:

- Level 0 (–O0) compiles the fastest and produces the slowest output as linear code. Level 0 produces unoptimized code.
- Level 1 (–O1) takes longer to compile, applies target-independent optimizations, and produces optimized linear code.
- Level 2 (–O2, the default) compiles more slowly than Level 1, applies all Level 1 optimizations plus all target-specific optimizations, and can produce parallelized code that is faster than level 1.
- Level 3 (–O3) applies all Level 2 optimizations and uses a low level global register allocator that can produce faster, more parallelized code than level 2. This is the level selected for all functions presented in this application note.

Only one of these optimization levels can be selected for each compilation. Two supplemental optimizations are available which can be used in combination with levels 1, 2 or 3:

- Space optimization (–Os) applies the selected level of optimization while weighting the optimization process in favor of program size. Programs or modules that have been optimized for space require a smaller amount of memory but may sacrifice program speed.
- Cross-file optimization (–Og) is a complex process which requires significantly more compilation time than non-cross file optimization. With cross-file optimization, the optimizer applies the selected level of optimization across all the files in the application at the same time, and as a result produces the most efficient program code.

For a complete description of the StarCore compiler please consult the StarCore C compiler user’s manuals [4, 7].

3 Optimization Phases and Methods
There are many optimization methods that can take advantage of the SC140’s speed to increase performance. These methods can be implemented in either C or assembly. This section discusses the implementation of three of these methods—multisample, packed moves, and loop unrolling.

3.1 C Optimizations
In this first phase of code development, C code is compiled and run as is with no modifications. All ETSI/ITU standard code performs fractional data arithmetic with clearly defined math functions, which provide a standard way of completely describing the functionality for bit exact behavior. Most of these functions involve saturation,
which is not easily represented in ANSI C, so they are usually replaced by inline versions with specific machines instructions. In this phase, the original C code is modified to increase speed and/or reduce code size. The optimization techniques presented in this application note include multisample, packed moves, and loop unrolling.

### 3.1.1 Multisample

The multisample programming technique can be applied to algorithms with alignment restrictions or bit-exact requirements. This technique takes multiple samples at the input in parallel and generates multiple samples at the output simultaneously. The number of samples processed depends on the processor architecture and type of algorithm. The multisample technique reuses the operands in order to relax the alignment requirements for loading operands. This allows simpler operand addressing and effectively solves the problems of memory bus bandwidth, operand alignment, and limited algorithm parallelism when using multiple ALUs.

For more information on the multisample technique, refer to [8].

### 3.1.2 Packed Moves

Packed moves are used to optimize data flow bandwidth. The StarCore architecture enables multiple moves; that is, several moves from/to registers can be performed in one instruction. In the optimization of the functions described in this document, double-word (32 bit) and quad-word (64 bit) accesses to the memory are performed instead of single word data accesses. With two address generation units, StarCore can load/store up to eight 16-bit operands in a single cycle. The StarCore compiler generates assembly code for packed moves if the following constraints are met:

- Accesses occur in a loop.
- Memory accesses are in sequence with a certain stride (typically in arrays and structures).
- The objects to be addressed meet certain alignment requirements.

In assembly, packed moves can be used everywhere, including within loops, outside of loops, in arrays, structures and ordinary variables that are located at consecutive addresses in memory. The only restriction that applies concerns the alignment of the accessed memory group. Objects are usually aligned according to their size. The `#pragma align` compiler directive can be used to force the alignment of arrays and structures to meet the specific alignment requirements.

For more information on memory alignment, refer to the StarCore C compiler user’s manuals [4, 7].

### 3.1.3 Loop Unrolling

Loop unrolling replaces the body of a loop with several copies of the body and adjusts the loop-control code accordingly. Loop unrolling reduces the overhead of executing an indexed loop and can improve the effectiveness of other optimizations such as instruction scheduling and software pipelining. As a stand-alone technique, loop unrolling is used to increase the ALU usage per loop step. If the iterations are independent, they can be performed in parallel, each on a separate ALU. Often, loop unrolling is used in conjunction with other optimizations, such as multisample. The compiler offers a mean to automatically unroll loops by using `#pragma loop_unroll` compiler directive. For a complete description, refer to the StarCore C compiler user’s manual [4, 7].
3.2 Assembly

In general, writing functions in assembly increases speed and reduces code size. However, as C compilers become more efficient, the trend is to keep most code in C and optimized C and implement fewer functions in assembly. All the C optimization techniques can also be used in assembly. More aggressive optimizations, such as the use of special processor features and instructions, can also be performed. Writing in assembly offers greater flexibility for the optimizations performed, but it is more difficult to debug. In addition, certain details must be considered when writing assembly code, including processor restrictions, data alignment, hardware loop alignment, and nesting order.

4 Optimization of the Residu() Function

The classic (unmodified) C code for the Residu() function is presented in Code Listing 1.

**Code Listing 1. Residu() Function in Classic C Code**

```c
#include <prototype.h>
#include "constants.h"

void Residu(
    Word16 a[], /* (i) Q12: Prediction coefficients. */
    Word16 x[], /* (i) : Speech (values x[-m..-1] are needed) */
    Word16 y[]  /* (o) : Residual signal. */
) {
    Word16 i, j;
    Word32 L_s;
    for(i = 0; i < L_SUBFR; i++)
    {
        L_s = L_mult(x[i], a[0]);
        for (j = 1; j <= M; j++)
        {
            L_s = L_mac(L_s, a[j], x[i-j]);
        }
        L_s = L_shl(L_s, 3);
        y[i] = round(L_s);
    }
}
```

The optimization techniques applied to the Residu() function are multisample and packed moves. The outer loop executes 40 times, which is a multiple of 4, so the multisample technique was applied to the outer loop, 4 samples being computed in parallel. Implementation is straightforward because there are no dependencies between the computed values in each iteration. The compiler optimizes the inner loop by unrolling it completely. The rolled inner loop has 10 iterations, so the unrolled inner loop has two iterations plus two computations performed after the loop.

The code was written so that packed moves are performed at the beginning of the outer loop when loading the speech samples $x[i]$, $x[i+1]$, $x[i+2]$, and $x[i+3]$, and at the end of the loop when storing the residual speech signal samples $y[i]$, $y[i+1]$, $y[i+2]$, and $y[i+3]$. Both the speech and residual speech signals are 16-bit values. To enable quad-operand packed moves, the target arrays ($x[]$ and $y[]$) were aligned at 8-byte boundaries in the caller. The first LP parameter ($a[0]$) is fetched outside the outer loop so that the compiler retains it in the registers while the loop executes.
At the end of the outer loop the residual signal is scaled by shifting the samples 3 bits to the left using the \texttt{L\_shl()} intrinsic. The code generated by \texttt{L\_shl()} is an \texttt{asll} instruction followed by a \texttt{sat\_l} instruction. The \texttt{sat\_l} instruction saturates its operand at a 32-bit value.

The scaled values are rounded using the \texttt{round()} intrinsic, then the residual speech is saved in array \texttt{y[]}. The code generated by the compiler for \texttt{round()} is a single \texttt{rnd} instruction. The \texttt{rnd} instruction saturates its source operand according to the current saturation mode. The \texttt{L\_shl()} and \texttt{round()} intrinsics both operate in 32-bit saturation mode and saturate their operands. Thus the saturation operation is performed twice on each value.

To eliminate the \texttt{sat\_l} instruction from the generated code, 40-bit intrinsics are used. The \texttt{X\_extend()} intrinsic acts as a cast operator by converting a 32-bit value to a 40-bit value. The \texttt{X\_shl()} intrinsic does not saturate its result, so the \texttt{sat\_l} instruction is eliminated. The \texttt{X\_rnd()} intrinsic saturates its result according to the current saturation mode, which is 32-bit saturation mode for both GSM EFR and G.729. The optimized C code for the \texttt{Residu()} function is presented in \textbf{Code Listing 2}.

\textbf{Code Listing 2.} The \texttt{Residu()} Function in Optimized C Code

```c
void Residu(
    Word16 a[], /* (i) Q12: Prediction coefficients. */
    Word16 x[], /* (i) : Speech (values x[-m..-1] are needed) */
    Word16 y[]  /* (o) : Residual signal. */
){
    #pragma align * x 8
    #pragma align * y 8
    Word16 i, j;
    Word16 a0;
    Word16 x0, x1, x2, x3;
    Word32 L_s0, L_s1, L_s2, L_s3;
    a0 = a[0];
    for (i = 0; i < L_SUBFR; i+=4)
    {
        L_s0 = L_mult(x[i ], a0);
        L_s1 = L_mult(x[i+1], a0);
        L_s2 = L_mult(x[i+2], a0);
        L_s3 = L_mult(x[i+3], a0);
        x0 = x[i ];
        x1 = x[i+1];
        x2 = x[i+2];
        x3 = x[i-1];
        for (j = 1; j <= M - 2; j+=4)
        {
            L_s0 = L_mac(L_s0, a[j], x3);
            L_s1 = L_mac(L_s1, a[j], x0);
            L_s2 = L_mac(L_s2, a[j], x1);
            L_s3 = L_mac(L_s3, a[j], x2);
            x2 = x[i-j-1];
            L_s0 = L_mac(L_s0, a[j+1], x2);
            L_s1 = L_mac(L_s1, a[j+1], x3);
            L_s2 = L_mac(L_s2, a[j+1], x0);
            L_s3 = L_mac(L_s3, a[j+1], x1);
            x1 = x[i-j-2];
            L_s0 = L_mac(L_s0, a[j+2], x1);
            L_s1 = L_mac(L_s1, a[j+2], x2);
            L_s2 = L_mac(L_s2, a[j+2], x3);
            L_s3 = L_mac(L_s3, a[j+2], x0);
        }
    }
}
```
x0 = x[i-j-3];
L_s0 = L_mac(L_s0, a[j+3], x0);
L_s1 = L_mac(L_s1, a[j+3], x1);
L_s2 = L_mac(L_s2, a[j+3], x2);
L_s3 = L_mac(L_s3, a[j+3], x3);
x3 = x[i-j-4];
}
L_s0 = L_mac(L_s0, a[M-1], x3);
L_s1 = L_mac(L_s1, a[M-1], x0);
L_s2 = L_mac(L_s2, a[M-1], x1);
L_s3 = L_mac(L_s3, a[M-1], x2);
x2 = x[i-M];
L_s0 = L_mac(L_s0, a[M], x2);
L_s1 = L_mac(L_s1, a[M], x3);
L_s2 = L_mac(L_s2, a[M], x0);
L_s3 = L_mac(L_s3, a[M], x1);
}
y[i ] = X_round(X_shl(X_extend(L_s0), 3));
y[i+1] = X_round(X_shl(X_extend(L_s1), 3));
y[i+2] = X_round(X_shl(X_extend(L_s2), 3));
y[i+3] = X_round(X_shl(X_extend(L_s3), 3));
}

Note: In the development phase, assert statements can be used to verify the alignment property of the parameters. 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.

4.1 Assembly

The optimized C code was used as a reference for writing the assembly version of Residu(). To reduce the size of the function, the inner loop in the assembly version was not fully unrolled as the compiler (automatically) did with the optimized C version. Fully unrolling the function can gain about 20 cycles but increases the size by about 48 bytes. The FALIGN directive was used to instruct the assembler to align the first execution set of the hardware loops. For additional information about loop alignment refer to the SC100 Assembly Language Tools User's Manual [5]. Code Listing 3 is the assembly version of the Residu() function.

Code Listing 3. The Residu() Function—Assembly Code

INCLUDE 'constants.inc'

SECTION .text LOCAL
OPT LEA
GLOBAL Residu
ALIGN 16
_Residu TYPE FUNC
push d6

doen2 #<(L_SUBFR>>2)
dosetup2 L0_0
move.l (sp-20),r2
tfra r0,r4
move.f (r4)+,d8
adda #<-2,r1,r3
dosetup3 L1_0
FALIGN
LOOPSTART2
L0_0
[clr d0
clr d2
doen3 #<(M-1)>>2)
mov4.4f (r1)+,d4:d5::d6:d7
]
; for(j = 0; j < M-2; j += 4)
4.2 Results

Table 1 summarizes the benchmark comparisons for the classic C, optimized C, and assembly implementations of the Residu() function:

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Speed (Cycles)</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic C</td>
<td>698</td>
<td>80</td>
</tr>
<tr>
<td>Optimized C</td>
<td>174</td>
<td>296</td>
</tr>
<tr>
<td>Assembly</td>
<td>164</td>
<td>196</td>
</tr>
</tbody>
</table>

5 Optimization of the Convolve() Function

The classic (unmodified) C code for the Convolve() function is presented in Code Listing 4.
#include <prototype.h>
#include "constants.h"

void Convolve(
    Word16 x[], /* (i)    : Input vector */
    Word16 h[], /* (i) Q12: Impulse response */
    Word16 y[]  /* (o)    : Output vector */
)
{
    Word16 i, n;
    Word32 L_s;

    for (n = 0; n < L_SUBFR; n++)
    {
        L_s = 0;
        for (i = 0; i <= n; i++)
        {
            L_s = L_mac(L_s, x[i], h[n-i]);
        }
        L_s = L_shl(L_s, 3);    /* h is in Q12 and saturation */
        y[n] = extract_h(L_s);
    }
}

The optimization techniques applied to Convolve() are multisample and packed moves. The outer loop executes 40 times, which is a multiple of 4, so multisample technique was applied for the outer loop, 4 samples being computed in parallel. Its implementation is very easy since no dependencies exist between the computed values.

The new outer loop iterates 10 times, but the inner loop is executed only 9 times (it is not executed in the first iteration of the outer loop). To avoid a compare and jump sequence in the outer loop, the first iteration of the loop was moved outside (before) the loop.

The code is written so that packed moves are performed at the beginning of the outer loop when loading the speech samples $x[i], \ldots, x[i+3]$ and at the end of the loop when storing the convolved speech signal samples $y[i], \ldots, y[i+3]$. Both the speech and convolved speech signals are 16-bit values. To enable quad-operand packed moves, the target arrays ($x[]$ and $y[]$) were aligned at 8-byte boundaries in the caller.

At the end of the outer loop the convolved signal is scaled by shifting the samples 3 bits to the left using the L_shl() intrinsic. The code generated by L_shl() is an asll instruction followed by a sat.l instruction. The sat.l instruction saturates its operand at a 32-bit value.

Because overflow cannot occur (the filter coefficients $h[]$ are in Q12 format) the sat.l instruction was eliminated from the generated code by replacing the L_shl() intrinsic with the regular C left-shift operator $\ll$. The optimized C code for the Convolve() function is presented in Code Listing 5.
Code Listing 5. The Convolve() Function—Optimized C Code

```c
#include <prototype.h>
#include "constants.h"

void Convolve(
    Word16 x[], /* (j)    : L_SUBFR Input vector */
    Word16 h[], /* (j) Q12: L_SUBFR Impulse response */
    Word16 y[]  /* (o)    : L_SUBFR Output vector */
)
{
    #pragma align * h 8
    #pragma align * y 8

    Word16 i, j;
    Word32 L_s0, L_s1, L_s2, L_s3;
    Word16 x0, x1, x2, x3;
    x0 = x[0];
    x1 = x[1];
    x2 = x[2];
    x3 = x[3];

    L_s0 = L_mult(x0, h[0]);
    L_s1 = L_mult(x0, h[1]);
    L_s2 = L_mult(x0, h[2]);
    L_s3 = L_mult(x0, h[3]);

    L_s1 = L_mac(L_s1, x1, h[0]);
    L_s2 = L_mac(L_s2, x1, h[1]);
    L_s3 = L_mac(L_s3, x1, h[2]);

    L_s2 = L_mac(L_s2, x2, h[0]);
    L_s3 = L_mac(L_s3, x2, h[1]);

    L_s3 = L_mac(L_s3, x3, h[0]);
    L_s0 <<= 3;  /* h is in Q12 */
    L_s1 <<= 3;  /* h is in Q12 */
    L_s2 <<= 3;  /* h is in Q12 */
    L_s3 <<= 3;  /* h is in Q12 */

    y[0] = extract_h(L_s0);
    y[1] = extract_h(L_s1);
    y[2] = extract_h(L_s2);
    y[3] = extract_h(L_s3);  

    for (i = 4; i < L_SUBFR; i+=4) 
    {
        x0 = x[i];
        x1 = x[i+1];
        x2 = x[i+2];
        x3 = x[i+3];
        L_s0 = L_mult(x0, h[i]);
        L_s1 = L_mult(x0, h[i+1]);
        L_s2 = L_mult(x0, h[i+2]);
        L_s3 = L_mult(x0, h[i+3]);

        L_s1 = L_mac(L_s1, x1, h[i]);
        L_s2 = L_mac(L_s2, x1, h[i+1]);
        L_s3 = L_mac(L_s3, x1, h[i+2]);

        L_s2 = L_mac(L_s2, x2, h[i]);
        L_s3 = L_mac(L_s3, x2, h[i+1]);
```

Speech Coder Filters Using the StarCore™ SC140/SC1400 Cores, Rev. 1
L_s3 = L_mac(L_s3, x3, h[i]);

for (j = 4; j <= i; j += 4)
{
    x0 = x[j];
    L_s0 = L_mac(L_s0, x1, h[i-j+3]);
    L_s1 = L_mac(L_s1, x2, h[i-j+3]);
    L_s2 = L_mac(L_s2, x3, h[i-j+3]);
    L_s3 = L_mac(L_s3, x0, h[i-j+3]);
    x1 = x[j+1];
    L_s0 = L_mac(L_s0, x2, h[i-j+2]);
    L_s1 = L_mac(L_s1, x3, h[i-j+2]);
    L_s2 = L_mac(L_s2, x0, h[i-j+2]);
    L_s3 = L_mac(L_s3, x1, h[i-j+2]);
    x2 = x[j+2];
    L_s0 = L_mac(L_s0, x3, h[i-j+1]);
    L_s1 = L_mac(L_s1, x0, h[i-j+1]);
    L_s2 = L_mac(L_s2, x1, h[i-j+1]);
    L_s3 = L_mac(L_s3, x2, h[i-j+1]);
    x3 = x[j+3];
    L_s0 = L_mac(L_s0, x0, h[i-j]);
    L_s1 = L_mac(L_s1, x1, h[i-j]);
    L_s2 = L_mac(L_s2, x2, h[i-j]);
    L_s3 = L_mac(L_s3, x3, h[i-j]);
}

L_s0 <<= 3; /* h is in Q12 */
L_s1 <<= 3; /* h is in Q12 */
L_s2 <<= 3; /* h is in Q12 */
L_s3 <<= 3; /* h is in Q12 */

y[i ] = extract_h(L_s0);
y[i+1] = extract_h(L_s1);
y[i+2] = extract_h(L_s2);
y[i+3] = extract_h(L_s3);
}

5.1 Assembly

The optimized C code was used as a reference for writing the assembly version of Convolve(). The FALIGN directive, which was used to align hardware loops, generated several NOPs. Dummy clr instructions were added to remove them. Code Listing 6 is the assembly version of the Convolve() function.

Code Listing 6. The Convolve() Function—Assembly Code

INCLUDE 'constants.inc'

SECTION .text LOCAL

GLOBAL _Convolve
ALIGN 16

_Convolve TYPE FUNC

push d6
  tfra r0,r4
  move.f (r4)+,d7
[
  mpy d8,d7,d6
  mpy d9,d7,d6
  mpy d10,d7,d2
  move.f (r4)+,d6
][
  asll #<3,d0
  mac d8,d6,d11
  mac d9,d6,d11
  move.f (r4)+,d5
push d7
  tfra r1,r3
  move.4f (r3),d8:d9:d10:d11
  load x[0], load h[0..3]
  ; y[0] = h[0] * x[0]
  ; load x[1], r2 = &y[0]
  ; y[0] << 3
  ; y[1] * x[1]
  ; y[2] * x[2]
  ; y[3] * x[3]
Optimization of the Convolve() Function

```assembly
; load x[2]
[ asll #<3,d1  mac d8,d5,d2   ; y[2] << 3
  mac d9,d5,d3  ; y[3] == h[0] * x[2]
  move.f (r4)+,d4  ; load x[3]
]
[ asll #<3,d2  ; y[2] << 3
doen2 #<(L_SUBFR)>>2)-1  ; y[3] == h[0] * x[3]
]
[ asll #<3,d3  ; y[3] << 3
  adda #<6,r4  ; r4 = 4x[7]
]
[ clr d0
  moves.4f d0:d1:d2:d3,(r2)+
] FALIGN
LOOPSTART2
L0_0
[
  sub #<6,d13
  move.4f (r3)+,d8:d9:d10:d11
  movf.4f (r4)-,d4
] FALIGN
LOOPSTART3
L1_0
[
  mac d9,d4,d0  ; y[n] == h[i] * x[n-i]
  mac d11,d4,d2  ; y[n+1] == h[i+1] * x[n-i-1]
  move.f (r4)-,d5  ; load x[n-i-1], load h[i-1]
]
[ mac d10,d5,d0  ; y[n+2] == h[i+2] * x[n-i-2]
  mac d8,d5,d2  ; y[n+3] == h[i+3] * x[n-i-3]
  move.f (r4)-,d6  ; load x[n-i-2], load h[i]
]
[ mac d11,d6,d0  ; y[n+3] == h[i+4] * x[n-i-2]
  mac d9,d6,d2  ; y[n+2] == h[i+5] * x[n-i-3]
  move.f (r4)-,d7  ; load x[n-i-2], load h[i-1]
]
[ mac d8,d7,d0  ; y[n] == h[i+4] * x[n-i-3]
  mac d10,d7,d2  ; y[n+1] == h[i+5] * x[n-i-3]
  move.f (r4)-,d8  ; load x[n-i-4], load h[i+1]
] LOOPEND3
[ asll #<3,d0  ; y[n] << 3
  asll #<3,d2  ; y[n+1] << 3
tfra r0,r4  ; y[n+2] << 3
]
[ moves.4f d0:d1:d2:d3,(r2)+
tfra r1,r3  ; r4 = 4x[8]
]
LOOPEND2
pop d6
rts
SIZE _Convolve,*-_Convolve
ENDSEC
END
```
5.2 Results

Table 2 summarizes the benchmark comparisons for the classic C, optimized C, and assembly implementations of the Convolve() function:

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Speed (Cycles)</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic C</td>
<td>1237</td>
<td>96</td>
</tr>
<tr>
<td>Optimized C</td>
<td>351</td>
<td>474</td>
</tr>
<tr>
<td>Assembly</td>
<td>269</td>
<td>258</td>
</tr>
</tbody>
</table>

6 Optimization of the Syn_filt() Function

The classic (unmodified) C code for the Syn_filt() function is presented in Code Listing 7.

Code Listing 7. Syn_filt() Function in Classic Code

```c
#include <prototype.h>
#include "constants.h"

void Syn_filt(
    Word16 a[],  /* (i) Q12: Prediction coefficients */
    Word16 x[],  /* (i) : Input signal */
    Word16 y[],  /* (o) : Output signal */
    Word16 lg,   /* (i) : Size of filtering */
    Word16 mem[], /* (i/o) : Memory associated with this filtering. */
    Word16 update /* (i) : 0=no update, 1=update of memory. */
) {
    Word16 i, j;
    Word32 L_s;
    Word16 tmp[L_SUBFR + M]; /* This is usually done by memory allocation (lg+M) */
    /* lg is L_SUBFR or L_H. L_SUBFR has greater value */
    Word16 * yy; /* Copy mem[] to yy[] */
    yy = tmp;
    for(i = 0; i < M; i++) {
        *yy++ = mem[i];
    }
    /* Do the filtering. */
    for (i = 0; i < lg; i++) {
        L_s = L_mult(x[i], a[0]);
        for(j = 1; j <= M; j++) {
            L_s = L_msu(L_s, a[j], yy[-j]);
        }
        L_s = L_shl(L_s, 3);
        *yy++ = round(L_s);
    }
}
```
6.1 Optimized C

The optimization techniques applied to the `Syn_filt()` function are multisample, loop unrolling and packed moves. Because of the dependency that exists when computing the filtered signal samples (`tmp[i]` depends on `tmp[i-1]`), the inner loop was reversed so that the multisample technique could be used. The outer loop executes 40 times, which is a multiple of 4, so multisample technique was applied to the outer loop, 4 samples being computed in parallel. The inner loop was completely unrolled.

In the copy loops (the first and third loops) the `#pragma loop_unroll` directive was used to instruct the compiler to automatically unroll the loops. The first loop was unrolled four times and the third loop two times. This was done so that packed moves can be generated.

The code is written so that packed moves are performed at the beginning of the outer loop when loading the speech samples `x[i],...,x[i+3]` and at the end of the loop when storing the convolved speech signal samples `y[i],...,y[i+3]`. Both the speech and convolved speech signals are 16-bit values. To enable quad-operand packed moves, the target arrays (`x[]` and `y[]`) were aligned at 8-byte boundaries in the caller.

The code was written so that packed moves are performed in the first copy loop on `tmp[i],...,tmp[i+3]` and `mem[i],...,mem[i+3]`. Packed moves are also performed at the beginning of the filtering loop when loading the speech samples `x[i],...,x[i+3]` and the filtered speech samples `tmp[i],...,tmp[i+3]`, and at the end of the loop when storing the filtered speech signal samples `y[i],...,y[i+3]` and `tmp[i],...,tmp[i+3]`. Both the filter memory and filtered speech signals are 16-bit values. In order to perform quad-operand packed moves, the target arrays (`x[]`, `y[]`, `mem[]` and `tmp[]`) were aligned at 8-byte boundaries.

The first LP parameter (`a[0]`) is fetched outside the outer loop so that the compiler retains it in the registers while the loop executes.

At the end of the outer loop the filtered signal is scaled by shifting the samples 3 bits to the left using the `L_shl()` intrinsic. The code generated by `L_shl()` is an `asll` instruction followed by a `sat.l` instruction. The `sat.l` instruction saturates its operand at a 32-bit value.

The scaled values are rounded using the `round()` intrinsic, then the residual speech is saved in array `y[]`. The code generated by the compiler for `round()` is a single `rnd` instruction. The `rnd` instruction saturates its source operand according to the current saturation mode. The `L_shl()` and `round()` intrinsics both operate in 32-bit saturation mode and saturate their operands. Thus the saturation operation is performed twice on each value.

To eliminate the `sat.l` instruction from the generated code, 40-bit intrinsics are used. The `X_extend()` intrinsic acts as a cast operator by converting a 32-bit value to a 40-bit value. The `X_shl()` intrinsic does not saturate its result, so the `sat.l` instruction is eliminated. The `X_rnd()` intrinsic saturates its result according to
the current saturation mode, which is 32-bit saturation mode for both GSM EFR and G.729. The optimized C code for the Syn_filt() function is presented in Code Listing 8.

**Code Listing 8. Syn_filt() Function in Optimized C Code**

```c
#include <prototype.h>
#include "constants.h"

void Syn_filt(
    Word16 a[],     /* (i) Q12 : a[m+1] Prediction coefficients   (m=10) */
    Word16 x[],     /* (i) : Input signal                      */
    Word16 y[],     /* (o) : Output signal                     */
    Word16 lg,      /* (i) : Size of filtering                  */
    Word16 mem[],   /* (i/o) : Memory associated with this filtering. */
    Word16 update   /* (i) : 0=no update, 1=update of memory.     */
){
    #pragma align * x 8
    #pragma align * y 8
    #pragma align * mem 8
    Word16 i, j;
    Word16 L_s0, L_s1, L_s2, L_s3;
    Word16 s0, s1, s2, s3;
    Word16 a0, a1, a2, a3;
    Word16 tmp[M+L_SUBFR];  /* This is usually done by memory allocation (lg*M) */
    /* lg is L_SUBFR or L_H. L_SUBFR has greater value */
    #pragma align tmp 8
    /*
    Copy mem[] to tmp[].
    Extended the loop with two more iterations, so the number of iterations is a multiple of 4.
    This was done so #pragma loop_unroll 4 will work.
    */
    for(i = 0; i < M + 2; i++)
    {
        #pragma loop_unroll 4
        tmp[i] = mem[i];
    }
    a0 = a[0];

    for (i = 0, j = M; i < lg; i += 4, j += M)
    {
        s0 = tmp[i ];
        s1 = tmp[i+1];
        s2 = tmp[i+2];
        s3 = tmp[i+3];
        L_s0 = L_mult(x[i ], a0);
        L_s1 = L_mult(x[i+1], a0);
        L_s2 = L_mult(x[i+2], a0);
        L_s3 = L_mult(x[i+3], a0);
        L_s0 = L_msu(L_s0, a[j ], s0);
        L_s1 = L_msu(L_s1, a[j], s1);
        L_s2 = L_msu(L_s2, a[j], s2);
        L_s3 = L_msu(L_s3, a[j], s3);
        s0 = tmp[i+4];
        j--;
        L_s0 = L_msu(L_s0, a[j], s1);
        L_s1 = L_msu(L_s1, a[j], s2);
        L_s2 = L_msu(L_s2, a[j], s3);
        L_s3 = L_msu(L_s3, a[j], s0);
        s1 = tmp[i+5];
        j--;
        L_s0 = L_msu(L_s0, a[j], s2);
        L_s1 = L_msu(L_s1, a[j], s3);
        L_s2 = L_msu(L_s2, a[j], s0);
        L_s3 = L_msu(L_s3, a[j], s1);
        s2 = tmp[i+6];
        j--;
        L_s0 = L_msu(L_s0, a[j], s3);
        L_s1 = L_msu(L_s1, a[j], s0);
    }
}
```

Speech Coder Filters Using the StarCore™ SC140/SC1400 Cores, Rev. 1
Optimization of the Syn_filt() Function

L_s2 = L_msu(L_s2, a[j], s1);
L_s3 = L_msu(L_s3, a[j], s2);

s3 = tmp[i+7];
j--;

L_s0 = L_msu(L_s0, a[j], s0);
L_s1 = L_msu(L_s1, a[j], s1);
L_s2 = L_msu(L_s2, a[j], s2);
L_s3 = L_msu(L_s3, a[j], s3);
s0 = tmp[i+8];
j--;

L_s0 = L_msu(L_s0, a[j], s0);
L_s1 = L_msu(L_s1, a[j], s1);
L_s2 = L_msu(L_s2, a[j], s2);
L_s3 = L_msu(L_s3, a[j], s3);
s1 = tmp[i+9];
j--;

L_s0 = L_msu(L_s0, a[j], s0);
L_s1 = L_msu(L_s1, a[j], s1);
L_s2 = L_msu(L_s2, a[j], s2);
L_s3 = L_msu(L_s3, a[j], s3);
s2 = tmp[i+10];
j--;

L_s0 = L_msu(L_s0, a[j], s0);
L_s1 = L_msu(L_s1, a[j], s1);
L_s2 = L_msu(L_s2, a[j], s2);
L_s3 = L_msu(L_s3, a[j], s3);
s3 = tmp[i+11];
j--;

L_s0 = L_msu(L_s0, a[j], s0);
L_s1 = L_msu(L_s1, a[j], s1);
L_s2 = L_msu(L_s2, a[j], s2);
L_s3 = L_msu(L_s3, a[j], s3);
s0 = X_round(X_shl(X_extend(L_s0), 3));

a1 = a[j+1];
a2 = a[j+2];
a3 = a[j+3];

L_s1 = L_msu(L_s1, a1, s0);
L_s2 = L_msu(L_s2, a2, s0);
L_s3 = L_msu(L_s3, a3, s0);
s1 = X_round(X_shl(X_extend(L_s1), 3));

L_s2 = L_msu(L_s2, a1, s1);
L_s3 = L_msu(L_s3, a2, s1);
s2 = X_round(X_shl(X_extend(L_s2), 3));

L_s3 = L_msu(L_s3, a1, s2);
s3 = X_round(X_shl(X_extend(L_s3), 3));

y[i + ] = tmp[M+i + ] = s0;
y[i+1] = tmp[M+i+1] = s1;
y[i+2] = tmp[M+i+2] = s2;
y[i+3] = tmp[M+i+3] = s3;

} /* Update of memory if(update == 1) */
if(update != 0)
{
/* Packed loads can’t be performed because lg is a variable. This can be done in assembly. */
for (i = 0; i < M; i++)
{
#pragma loop_unroll 2
  mem[i] = tmp[lg+i];
}
}
### 6.2 Assembly

The optimized C code was used as a reference for writing the assembly version of `Syn_filt()`. In addition, the filtering loop was software-pipelined to improve performance. The modulo addressing mode was used in the filtering loop to access the LP coefficients (`a[]`) to optimize both the speed and the size of the function. The alignment requirements on the incoming parameters for the assembly version of `Syn_filt()` are that `&a[1]` and `y[]` must be aligned at 4-byte boundaries while `x[]` and `mem[]` must be aligned at 8-byte boundaries. A symbolic stack management method was used to facilitate changing the number and the order of the incoming parameters and the local variables (the `tmp[]` array) that must be stored on the stack.

**Code Listing 9. The `Syn_filt()` Function—Assembly Code**

```assembly
INCLUDE 'constants.inc'

;* Local variables sizes
SET Syn_filt_tmp_sz (M+L_SUBFR)<<1
;* Stack frame layout
SET Syn_filt_outgoingParameters_sz 0
SET Syn_filt_localVariables_sz Syn_filt_tmp_sz
SET Syn_filt_stackFrame_sz (Syn_filt_localVariables_sz+Syn_filt_outgoingParameters_sz+7)&$fffffff8
SET Syn_filt_savedRegisters_sz 16
SET Syn_filt_returnAddress_sz 8
SET Syn_filt_incomingParameters_off
Syn_filt_stackFrame_sz+Syn_filt_savedRegisters_sz+Syn_filt_returnAddress_sz
;* Local variables offsets
SET Syn_filt_tmp_off Syn_filt_stackFrame_sz
;* Local variables
DEFINE Syn_filt_tmp '(sp-Syn_filt_tmp_off)'

;* Incoming parameters sizes
SET Syn_filt_y_sz 4
SET Syn_filt_lg_sz 2
SET Syn_filt_mem_sz 4
SET Syn_filt_update_sz 2
;* Incoming parameters offsets
SET Syn_filt_y_off Syn_filt_incomingParameters_off+Syn_filt_y_sz
SET Syn_filt_lg_off Syn_filt_y_off+Syn_filt_lg_sz
SET Syn_filt_mem_off Syn_filt_lg_off+2+Syn_filt_mem_sz
SET Syn_filt_update_off Syn_filt_mem_off+Syn_filt_update_sz
;* Incoming parameters
DEFINE Syn_filt_y '(sp-Syn_filt_y_off)'
DEFINE Syn_filt_lg '(sp-Syn_filt_lg_off)'
DEFINE Syn_filt_mem '(sp-Syn_filt_mem_off)'
DEFINE Syn_filt_update '(sp-Syn_filt_update_off)'

SECTION .text LOCAL
OPT LPA
GLOBAL _Syn_filt
ALIGN 16
_Syn_filt TYPE FUNC
push d6                       push d7
push mctl                     push r6
tfra sp,r4                    adda #Syn_filt_stackFrame_sz,sp,r6
; Build the stack frame
; r4 = &yy[0]
tfra r6,sp
move.w #<22,m0
move.1 Syn_filt_y,r2
move.1 Syn_filt_mem,r3
move.1 Syn_filt_update,n3
movs.4f (r3)+,d0:d1:d2:d3
movs.4f d0:d1:d2:d3, (r4)+
move.2f d0:d1, (r4)
[   asll #<2,d12
   dosetup3 L0_0
   [   asll #<2,d12
       dosetup3 L0_1
   ]
   [   sub #<1,d12
       move.f (r0)+,d8
   ]
   doen3 d12
   move.4f (r1)+,d0:d1:d2:d3
   move.2f (r3)+,d0:d1
   suba #<(M-2)<<1,r4
   ; r4 = &tmp[0]
   ]
   [   sub #<1,d12
       move.f (r0)+,d8
       move.4f (r1)+,d0:d1:d2:d3
   ]
   doen3 d12
   adda #<10,r5
   ; r5 = &a[5]
   ]
```

Speech Coder Filters Using the StarCore™ SC140/SC1400 Cores, Rev. 1

Freescale Semiconductor 19
Speech Coder Filters Using the StarCore™ SC140/SC1400 Cores, Rev. 1

Optimization of the Syn_filt() Function
6.3 Results

Table 3 summarizes the benchmark comparisons for the classic C, optimized C, and assembly implementations of the `Syn_filt()` function.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Speed (Cycles)</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic C</td>
<td>797</td>
<td>224</td>
</tr>
<tr>
<td>Optimized C</td>
<td>298</td>
<td>494</td>
</tr>
<tr>
<td>Assembly</td>
<td>183</td>
<td>608</td>
</tr>
</tbody>
</table>

7 Performance Comparison

Table 4 summarizes an overall benchmark comparison for this implementations. The assembly speed-up and memory increase are relative to the optimized C figures.

<table>
<thead>
<tr>
<th>Function</th>
<th>Phase</th>
<th>Speed (Cycles)</th>
<th>Size (Bytes)</th>
<th>Speed Improvement</th>
<th>Memory Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Residu()</td>
<td>Classic C</td>
<td>698</td>
<td>80</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Optimized C</td>
<td>174</td>
<td>296</td>
<td>4.01</td>
<td>370%</td>
</tr>
<tr>
<td></td>
<td>Assembly</td>
<td>164</td>
<td>196</td>
<td>1.06</td>
<td>-33.3%</td>
</tr>
<tr>
<td>Convolve()</td>
<td>Classic C</td>
<td>1237</td>
<td>96</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Optimized C</td>
<td>351</td>
<td>474</td>
<td>3.52</td>
<td>493.7%</td>
</tr>
<tr>
<td></td>
<td>Assembly</td>
<td>269</td>
<td>258</td>
<td>1.30</td>
<td>-45.5%</td>
</tr>
<tr>
<td>Syn_filt()</td>
<td>Classic C</td>
<td>797</td>
<td>224</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Optimized C</td>
<td>298</td>
<td>494</td>
<td>2.67</td>
<td>220%</td>
</tr>
<tr>
<td></td>
<td>Assembly</td>
<td>183</td>
<td>608</td>
<td>1.63</td>
<td>134.5%</td>
</tr>
</tbody>
</table>

The `Residu()` function has a very good performance in optimized C because the computation is straightforward, with no dependencies between the computed elements. In `Convolve()` the optimized C performance is not as good because the number of iterations within the inner loop is variable. The `Syn_filt()` function is somewhat more complicated because of the dependencies between the computed elements.

8 Conclusions

This application note presents three software development flows—classic C, optimized C, and assembly—for three speech filter functions on the SC140/SC1400 cores.
Classic C has several advantages over hand-coded assembly

- No modification is necessary.
- The code is easier to debug.
- The code size is smaller.

Optimized C has the following advantages over hand-coded assembly:

- Faster development.
- Easier to write and debug.
- Portable.
- Easy to reuse.
- High-quality implementation.
- Maintainable.

Assembly has the advantage that it gives full control over processor resources. Speech applications exhibit a high degree of parallelism that can be exploited by a variable-length execution set architectures such as StarCore. This document has demonstrated that StarCore tools can achieve competitive performance with respect to estimated hand-written assembly.

9 References

[3] SC140 DSP Core Reference Manual, order number MNSC140CORE.
[5] SC100 Assembly Language Tools User’s Manual, order number MNSC100ALT.
[8] StarCore Multisample Programming Technique Application Note, order number STCR140MLTAN.
[9] ITU-T G.729 Implementation on StarCore SC140 Application Note, Bogdan Costinescu, Razvan Ungureanu, Madalin Stoica, Emilian Medve, Radu Preda, Mugur Alexiu and Costel Ilas, order number AN2094.
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

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. StarCore is a trademark of StarCore LLC. All other product or service names are the property of their respective owners.