AN2266
Rev. 2, 11/2004

Developing Optimized Code for Both Size and Speed on the StarCore™ SC140/SC1400 Cores

By Mihai Fecioru, Corneliu Margina, and Bogdan Costinescu

This application note describes a set of techniques for optimizing code written for the StarCore™ SC140/SC1400 DSP cores. Guidelines for using these techniques are presented, along with solutions for resolving the *size versus speed* problem. For DSPs, the challenge is to develop an application with both a short execution time and a small memory requirement. Too often, achieving a decrease in execution time requires an increase in code size. The StarCore SC140/SC1400 cores ease the development of computation-intensive communication applications by providing four parallel ALUs and two AGUs so that up to six instructions can execute in each cycle. Unfortunately, the more efficiently the code is written to maximize use of the ALUs and AGUs, the more the memory space requirement increases. This is not desirable in embedded applications. Therefore, we must make some compromises to design a product that is both fast and small. All speed optimization techniques (such as inlining, software pipelining, loop unrolling, and multisampling) must be used carefully to optimize code size. This application note assists the programmer in deciding when and how to use the speed improvement techniques that may have a heavy impact on code size. The 80 percent–20 percent rule states that 80 percent of the execution time is spent on 20 percent of the code. Thus, the speed optimizations must be applied only to time-consuming functions. The rest of the functions should be compiled for size optimizations. All the measurements presented in this document were made using Metrowerks® CodeWarrior® for StarCore.

CONTENTS

1 Speed Optimizations

Speed optimization techniques on the SC140 core are generally classified as follows:

- Loop unrolling
- Split computation
- Multisampling

1.1 Loop Unrolling

The most popular speed optimization technique, loop unrolling explicitly repeats the body of a loop with corresponding indices. As a stand-alone technique, loop unrolling increases the DALU usage per loop step. If the iterations are independent, each one is performed on a single DALU. For example, the following code unrolls the loop three times to create four operations to be executed per one loop step:

Example 1. Loop Unrolling

```
Word16 signal[SIG_LEN];
#pragma align signal 8
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);
}
```

The same effect can be obtained using the *pragma loop_unroll* compiler feature:

Example 2. Loop Unrolling Using pragma loop_unroll

```
Word16 signal[SIG_LEN];
#pragma align signal 8
for ( i = 0; i < SIG_LEN; i++ )
{
#pragma loop_unroll 4
          signal[i] = L_shr(signal[i], 2);
}
```

In this document, the *unroll-factor* refers to the number of copies of the original loop that are in the unrolled loop. For example, in **Example 1**, the unroll-factor is 4. To use this techniques, the following conditions must be met:

- The vectors for unrolling must be properly aligned to use the packed memory moves (in our case, the *signal* vector must be aligned to 8 bytes).
- The loop counter must be a multiple of the unroll-factor.

Loop unrolling usually reduces the loop execution time by the *unroll-factor* times. However, the code size increases by the same factor.

Equation 1

$$\begin{cases} N_{LU} \geq N / \ UnrollFactor \\ S_{LU} \leq S \times UnrollFactor \end{cases}$$

where:

- N_{LU} is the number of cycles consumed by the loop after the loop unroll.
- N is the number of cycles consumed by the loop before the loop unroll.
- S_{LU} is the size of the code generated for the loop after the loop unroll (in bytes).
- S is the size of the code generated for the loop before the loop unroll (in bytes).

As **Equation 1** shows, for an unroll-factor of 2 the gain is N/2. If the speed optimization continues, using an unroll-factor of 4, the speed gain is other N/4 cycles, but the code size grows an additional 2 × S bytes. Therefore, the step from an unroll-factor of 2 to 4 should be taken only if speed constraints are more important than size constraints.

The unroll-factor is usually a power of two. Sometimes the best speed performance is achieved using an unroll-factor of 8 (as in **Example 1**). Notice that the code size increase does not depend on the number of iterations of the loop. The loop unrolling technique should be applied to loops with a large loop count.

1.2 Split Computation

A frequent operation in DSP computations is to reduce one dimension of a data massive (scalars are zero-dimensional, vectors are one-dimensional, and matrices are two-dimensional). The most frequently used reductions are: energy computation of a vector, mean square error, or maximum of a vector. If the reduction operator is associative and commutative, the reduction can be performed by splitting the original data massive into several data massives (usually four on the SC140 core). The reduction is applied to the smaller massives, and the results are combined to obtain the result as shown in **Example 3**.

Example 3. Split Computation

```
/* Energy computation for the signal[] vector of */
/* size SIG_LEN (multiple of 4). */
L_e0 = L_e1 = L_e2 = L_e3 = 0;
for ( i = 0; i < SIG_LEN; i+=4 ) {
          L_e0 = L_mac(L_e0, signal[i+0], signal[i+0]);
          L_e1 = L_mac(L_e1, signal[i+1], signal[i+1]);
          L_e2 = L_mac(L_e2, signal[i+2], signal[i+2]);
          L_e3 = L_mac(L_e3, signal[i+3], signal[i+3]);
}
L_e0 = L_add(L_e0, L_e1);
L_e2 = L_add(L_e2, L_e3);
L_e0 = L_add(L_e0, L_e2);
```

The same conditions must be met as for loop unrolling (for example, the vector alignment and the loop counter). In addition, split computations are used if the operator on the given data set is associative and commutative. For example, the summation of three fractionals is not associative when the saturation mode is on (the default for C mode):

```
(-0.4  + 0.8) + 0.5 = 0.9
```

but

```
-0.4  + (0.8 + 0.5) = 0.599
```

In **Example 3**, all the summed values are positive numbers and the fractional addition is associative if the values have the same sign.

The speed and size results are similar to those produced by loop unrolling. However, there are some important differences. In the simple example discussed, we have the following before split computation:

Equation 2

$$\begin{cases} N = 1 + N_{Loop} \\ S = 1 + S_{Loop} \end{cases}$$

and the following after split computation:

Equation 3

$$\begin{cases} N_{SC} = 1 + N_{Loop} / \ SplitFactor + log_2(SplitFactor) \\ S_{SC} = SplitFactor + S_{Loop} \times SplitFactor + (SplitFactor - 1) \end{cases}$$

where:

- N is the number of cycles before the split computation.
- S is the size of the code generated before the split computation (in bytes).
- N_{SC} is the number of cycles after the split computation.
- S_{SC} is the size of the code generated after the split computation (in bytes).
- N_{Loop} is the number of cycles of the loop only before split computation.
- S_{Loop} is the size of the code generated for the loop only before split computation.
- *SplitFactor* is usually a power of two.

Notice that there is a size penalty, just as there is for the loop unrolling technique. However, the size penalty is much more serious for the split computation. Our observations of the unroll-factor value for loop unrolling also apply to the split-factor value.

1.3 Multisampling

The *multisampling* technique is frequently used in nested loops and is a combination of primitive transformations. Given a nested loop formed out of OL (outer loop) and IL (inner loop containing one or two instructions), the multisampling transformation consists of the following:

- A loop unroll applied for OL to create a new OL with four IL inside (IL0, IL1, IL2, and IL3)
- A loop merge applied for IL0, IL1, IL2, and IL3 to create a new IL that makes more efficient use of the DALU units.
- A loop unroll applied to the newly-obtained IL so that the programmer can detail the reuse of already fetched values in the computations inside the new IL.

In **Example 4**, the nested loop computes the maximum absolute value of the correlations between X[] and h[]:

Example 4. Code Before Multisampling

```
L_max = 0;
for (i = L_SUBFR-1; i >= 0; i--) {
      L_s = 0;
      for (j = i; j <  L_SUBFR; j++)
            L_s = L_mac(L_s, X[j], h[j-i]);

      y32[i] = L_s;
      L_s = L_abs(L_s);
      if( L_s > L_max ) {
      L_max = L_s;
      }
}
```

Example 5 shows the result of applying the multisampling technique. The speed and size estimations are not as obvious as they are for loop unrolling and split computation. We have the following before multisampling:

Equation 4

$$\begin{cases} N \approx \dfrac{L(L-1)}{2} \\ S = SO \end{cases}$$

Example 5. Code After Multisampling

```
L_max0 = L_max1 = L_max2 = L_max3 = 0;
for (i = L_SUBFR-4; i >= 0; i-=4)
{
      Word16 x_curr = X[i];
      L_s0 = L_s1 = L_s2 = L_s3 = 0;

      h0 = h[0];
      h1 = h2 = h3 = 0;
      for (j = i; j < L_SUBFR; j+=4)
      {
            L_s0 = L_mac(L_s0, x_curr, h0);
            L_s1 = L_mac(L_s1, x_curr, h1);
            L_s2 = L_mac(L_s2, x_curr, h2);
            L_s3 = L_mac(L_s3, x_curr, h3);
            h3 = h[j+1-i]; x_curr = X[j+1];

            L_s0 = L_mac(L_s0, x_curr, h3);
            L_s1 = L_mac(L_s1, x_curr, h0);
            L_s2 = L_mac(L_s2, x_curr, h1);
            L_s3 = L_mac(L_s3, x_curr, h2);
            h2 = h[j+2-i]; x_curr = X[j+2];

            L_s0 = L_mac(L_s0, x_curr, h2);
            L_s1 = L_mac(L_s1, x_curr, h3);
            L_s2 = L_mac(L_s2, x_curr, h0);
            L_s3 = L_mac(L_s3, x_curr, h1);
            h1 = h[j+3-i]; x_curr = X[j+3];
```

```
            L_s0 = L_mac(L_s0, x_curr, h1);
            L_s1 = L_mac(L_s1, x_curr, h2);
            L_s2 = L_mac(L_s2, x_curr, h3);
            L_s3 = L_mac(L_s3, x_curr, h0);
            h0 = h[j+4-i]; x_curr = X[j+4];
        }

        L_y[i  ] = L_s0;
        L_y[i+1] = L_s1;
        L_y[i+2] = L_s2;
        L_y[i+3] = L_s3;

        L_s0 = L_abs(L_s0);
        L_s1 = L_abs(L_s1);
        L_s2 = L_abs(L_s2);
        L_s3 = L_abs(L_s3);

        L_max0 = L_max(L_max0, L_s0);
        L_max1 = L_max(L_max1, L_s1);
        L_max2 = L_max(L_max2, L_s2);
        L_max3 = L_max(L_max3, L_s3);
    }

    L_max0 = L_max(L_max0, L_max1);
    L_max1 = L_max(L_max2, L_max3);
    L_max0 = L_max(L_max0, L_max1);
```

We have the following after multisampling:

Equation 5

$$\begin{cases} N_{MS} \approx \sum_{i=0}^{L/(SF)} (L - SF \times i) = \frac{L(L/SF - 1)}{2} \approx \frac{N}{SF} \\ S_{MS} \approx (SO - SI) \times SF + (SF + 2) \times SI \end{cases}$$

where:

- L is `L_SUBFR` from the example.
- SF comes from *SampleFactor*.
- SO is the size of the outer loop before multisampling.
- SI is the size of the inner loop before multisampling.
- N/S is the number of cycles/code size before multisampling.
- N_{MS}/S_{MS} is the number of cycles/code size after multisampling.

The speed increases by sample-factor times, but the code size also increases significantly. Therefore, multisampling should be used only if the speed constraints are much more important than the size constraints. An advantage of multisampling is that no requirements are imposed on vector alignments. In several cases, multisampling can be used with split computation in the inner loop (for example, in the correlation operation described in **Section 4.1**, *Inline Speed Optimizations With No Code Restructuring*, on page 11).

2 Size Optimizations

A size optimization technique applied at the project level reuses code in different locations in a project. Instead of rewriting a section of code several times in the project, we write a function that contains the code and then we call the function. This function can be optimized for speed or size. For example, in vocoder projects, there are often sections of code that compute the energy of a vector or scale the elements of a vector with a certain value. Instead, special functions can be written and then optimized for speed.

3 Size and Speed Optimizations

So far, techniques for achieving size optimizations and speed optimizations have been considered separately. This section covers techniques that optimize code for both size and speed.

3.1 Loop Merging

Loop merging combines multiple loops into a single loop, reducing the size of the generated code and increasing instruction-level parallelism, thus increasing speed. Note that the two candidate loops must have the same number of loop iterations. **Example 6** shows a section of code before loop merging, and **Example 7** shows a section of code after loop merging.

Example 6. Before Loop Merging

```
/* scaling loop */
for ( i = 0; i < SIG_LEN; i++)
{
     y[i] = shr(y[i], 2);
}
/* energy computation */
L_e = 0;
for ( i = 0; i < SIG_LEN; i++)
{
     L_e = L_mac(L_e, y[i], y[i]);
}
```

As **Example 7** shows, the speed increases because the merged loop usually executes at the speed of the slowest of the original loops merged. Here, the scaling loop requires two cycles per iteration and the energy computation requires one cycle per iteration. The loop obtained from the merging requires two cycles per iteration, and no extra time is required for computing the energy. Therefore, we *merge two, and get one for free.* The code size decreases because the content of the merged result equals the sum of the contents of the original loops (in some cases even smaller, if the variables are reused), but the loop overhead (loop prologue) appears only once in the merged loop.

Example 7. After Loop Merging

```
/* Compute in the same time the energy of the */
/* scaled windowed signal */
L_e = 0;
for (i = 0; i < SIG_LEN; i++)
{
          Word16 temp;

          temp = shr(y[i], 2);
          L_e = L_mac(L_e, temp, temp);
          y[i] = temp;
}
```

After the merge occurs, the resulting loop can be further optimized for speed as shown in **Example 8**.

Example 8. Optimizing the Result of Loop Merging

```
L_e0 = L_e1 = L_e2 = L_e3 = 0;
for (i = 0; i < SIG_LEN; i += 4)
{
     Word16 temp0, temp1, temp2, temp3;

     temp0 = shr(y[i+0], 2);
     temp1 = shr(y[i+1], 2);
     temp2 = shr(y[i+2], 2);
     temp3 = shr(y[i+3], 2);

     L_e0 = L_mac(L_e0, temp0, temp0);
     L_e1 = L_mac(L_e1, temp1, temp1);
     L_e2 = L_mac(L_e2, temp2, temp2);
     L_e3 = L_mac(L_e3, temp3, temp3);

     y[i+0] = temp0;
     y[i+1] = temp1;
     y[i+2] = temp2;
     y[i+3] = temp3;
}
L_e0 = L_add(L_e0, L_e1);
L_e1 = L_add(L_e2, L_e3);
L_e0 = L_add(L_e0, L_e1);
```

Here, a loop unrolling technique is applied to the first part of the loop and a split computation to the second part (*UnrollFactor*=*SplitFactor*=*Factor*=*4*). The resulting loop is *Factor* times faster, but the code size is *Factor* times bigger, so this step should be applied only if the speed constraints require it. Take care when optimizing the results of loop merging The previous example is a fortunate one: *Factor* is 4 and the loop is 4 times faster. However, there are cases in which the optimization from *Factor*=2 to *Factor*=4 has no effect on speed, and the only effect is an increase in code size (see **Example 9** through **Example 12**).

Example 9. Merging Energy and Correlation

```
L_energy = 0; L_corr = 0;
for (j = 0; j < SIG_LEN; j++)
{
     L_energy = L_mac(L_energy, signal[j], signal[j]);
     L_corr = L_mac(L_corr, vector[j], signal[j]);
}
```

Example 10. Assembly Code for Merging Energy and Correlation

```
      loopstart3
PL001
      [
           mac      d3,d3,d2
           mac      d4,d3,d0
           move.f   (r5)+,d4
           move.f   (r1)+,d3
      ]
      loopend3
```

Example 10 merges energy and correlation loops. If the optimization of the merging result has *Factor*=2, then the assembly code is as shown in **Example 11**:

Example 11. Assembly Code for Merging Energy and Correlation (Factor = 2)

```
      loopstart3
PL001
      [
           mac      d2,d2,d6
           mac      d3,d3,d4
           mac      d0,d2,d5
           mac      d1,d3,d7
           move.2f  (r5)+,d2:d3
           move.2f  (r7)+,d0:d1
      ]
      loopend3
```

Notice that the speed increases by a factor of 2 and the code size also increases by a factor of 2. If the optimization goes further and *Factor*=4, the generated assembly code is as shown in **Example 12**.

Example 12. Assembly Code for Merging Energy and Correlation (Factor = 4)

```
      loopstart3
PL001
      [
           mac      d0,d4,d14
           mac      d1,d5,d13
           mac      d2,d6,d12
           mac      d3,d7,d9
           move.4f  (r6)+,d4:d5:d6:d7
      move.4f  (r1)+,d0:d1:d2:d3
      ]
      [
           mac      d4,d4,d10
           mac      d5,d5,d8
           mac      d6,d6,d15
           mac      d7,d7,d11
      ]
      loopend3
```

Now the speed remains the same as for the *Factor*=2 case, but the code size almost doubles. Obviously, we should not perform the second optimization step.

3.2 Splitting Functions

The splitting functions technique is usually applied to functions associated with easily identifiable sections of code that can be optimized for size or for speed. Instead of optimizing the function, it is preferable to create and optimize another function that is called by the original function. This technique can produce both increases in speed and reductions in code size.

4 Speed and Size Optimizations on Vocoder Projects

This section presents two scenarios for developing optimized code on the SC140 core and applying the optimizations in vocoder projects, one with and one without code restructuring (see **Figure 1**).

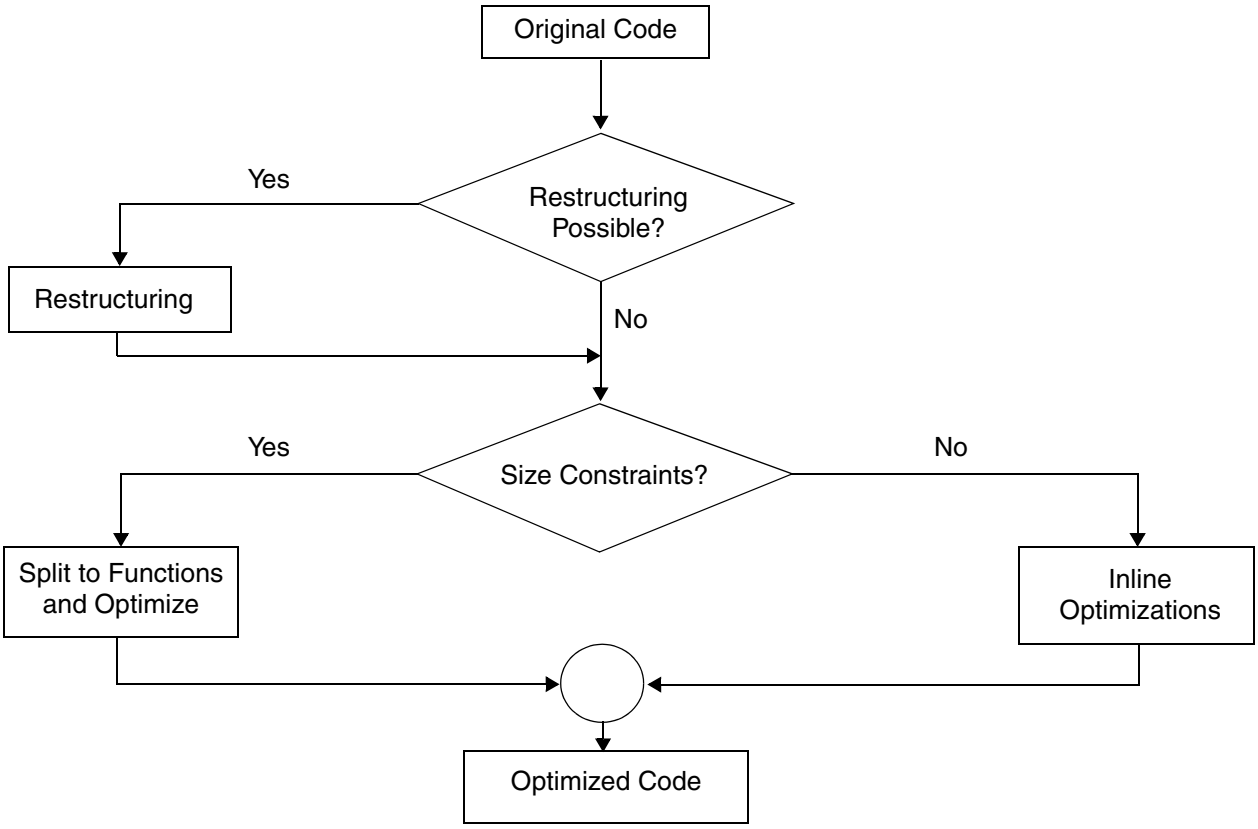

Figure 1. Scenarios for Optimizing Speed and Size on the SC140 Core

The first test, whether restructuring is possible, specifies the scenario type—for example, the possibility of loop merging. If loop merging is possible, then the necessary code restructuring is performed. The next step is to perform speed or size optimizations. If size constraints are important, a possible option is code reuse in which repeated sections of code are moved into functions and then optimized for speed or size. Tests are performed on vocoder functions that require optimization, such as Autocorr(), Norm_corr(), and Comp_corr(). The original form of these three functions is presented in **Appendix A**, *Original Code*, on page 19. The results of the tests performed on the original versions of these functions are shown in **Section 5**, *Results*, on page 16.

4.1 Inline Speed Optimizations With No Code Restructuring

First every loop is optimized for speed. Then, if the size constraints are important, we reuse the sections of code that repeat. After the Autocorr(), Norm_corr(), and Comp_corr() functions are optimized for speed, we can easily see that some sections of code repeat (energy loops, scaling loops, or correlation loops). We extract these sections of code into functions to reduce the size of the entire project. After the replacement is complete, the original functions are mainly calls to other small functions optimized for speed (written in assembly) so that they can be optimized for size. **Appendix C**, *Code Reuse*, on page 27 shows the code listing after code reuse.

4.1.1 Autocorr()

The Autocorr() function contains four loops that can be optimized for speed:

- A windowing loop to be optimized using loop unrolling by four. Four samples are computed every two cycles. Four samples cannot be computer each cycle because every loop iteration requires three memory moves (see **Example 13**).
- An energy loop to be optimized using split computation by four so that four samples are computed each cycle (see **Example 14**).
- A scaling loop to be optimized using loop unrolling by four so that four samples are computed every cycle. The shr (shift right) function is replaced with a multiply function that is more flexible for pipelining. The formula used is: *shr(x, 2) = mult(x, (1<<(15-2)))*. (See **Example 15**.)
- A correlation loop for which both the inner and outer loops are to be optimized using multisampling by two (see **Example 16**).

Example 13. Windowing Loop Optimized with Loop Unrolling

```
for (i = 0; i < L_WINDOW; i += 4)
{
     y[i+0] = mult_r (x[i+0], wind[i+0]);
     y[i+1] = mult_r (x[i+1], wind[i+1]);
     y[i+2] = mult_r (x[i+2], wind[i+2]);
     y[i+3] = mult_r (x[i+3], wind[i+3]);
}
```

Example 14. Energy Loop Optimized With Split Computation

```
L_sum0 = L_sum1 = L_sum2 = L_sum3 = 0L;
for (i = 0; i < L_WINDOW; i += 4)
{
     L_sum0 = L_mac(L_sum0, y[i1+0], y[i1+0]);
     L_sum1 = L_mac(L_sum1, y[i1+1], y[i1+1]);
     L_sum2 = L_mac(L_sum2, y[i1+2], y[i1+2]);
     L_sum3 = L_mac(L_sum3, y[i1+3], y[i1+3]);
}
L_sum = L_add(L_add(L_sum0, L_sum1), L_add(L_sum2, L_sum3));
```

Example 15. Scaling Loop Optimized With Loop Unrolling

```
for (i = 0; i < L_WINDOW; i += 4)
{
     y0 = mult(y[i+0], (1<<(15-2)));
     y1 = mult(y[i+1], (1<<(15-2)));
     y2 = mult(y[i+2], (1<<(15-2)));
     y3 = mult(y[i+3], (1<<(15-2)));
```

```
        y[i+0] = y0;
        y[i+1] = y1;
        y[i+2] = y2;
        y[i+3] = y3;
}
```

Example 16. Correlation Loop Optimized for Speed

```
for (i = 1; i <= m; i += 2)
{
        #pragma loop_count(5, 5, 1)
        L_sum0 = L_sum1 = L_sum2 = L_sum3 = 0L;

        t0 = y[i];
        for (j = 0; j < L_WINDOW - i; j += 4)
        {
        #pragma loop_count(55, 60, 1)

             t1 = y[j + i + 1];
             t2 = y[j + i + 2];

             L_sum0 = L_mac (L_sum0, y[j + 0], t0);
             L_sum1 = L_mac (L_sum1, y[j + 0], t1);
             L_sum2 = L_mac (L_sum2, y[j + 1], t1);
             L_sum3 = L_mac (L_sum3, y[j + 1], t2);

             t1 = y[j + i + 3];
             t0 = y[j + i + 4];

             L_sum0 = L_mac (L_sum0, y[j + 2], t2);
             L_sum1 = L_mac (L_sum1, y[j + 2], t1);
             L_sum2 = L_mac (L_sum2, y[j + 3], t1);
             L_sum3 = L_mac (L_sum3, y[j + 3], t0);
        }
        L_sumA = L_add(L_sum0, L_sum2);
        L_sumB = L_add(L_sum1, L_sum3);

        r[i] = L_shl_nosat (L_sumA, norm);
        r[i + 1] = L_shl_nosat (L_sumB, norm);
}
```

After all optimizations are complete, the resulting code is compiled for speed. **Table 1** shows the results:

Table 1. Autocorr() Inline Optimizations

Function	Compiler Option	Speed (Cycles)	Size (Bytes)
Autocorr	–O3	1004	566

4.1.2 Norm_corr()

The Norm_corr() function contains five loops to be optimized for speed:

- A simple correlation loop to be optimized using split computation by four so that four samples are computed every cycle (see **Example 17**).
- A scaling loop to be optimized with loop unrolling by four, just as for Autocorr().

- Two energy loops to be optimized with split computation by four, just as for Autocorr().
- A filter excitation loop to be optimized using loop unrolling by four (see **Example 18**).

Example 17. Simple Correlation Loop Optimized With Split Computation

```
L_s0 = L_s1 = L_s2 = L_s3 = 0;
for (j2 = 0; j2 < L_SUBFR; j2+=4) {
     L_s0 = L_mac (L_s0, xn[j2  ], s_excf[j2  ]);
     L_s1 = L_mac (L_s1, xn[j2+1], s_excf[j2+1]);
     L_s2 = L_mac (L_s2, xn[j2+2], s_excf[j2+2]);
     L_s3 = L_mac (L_s3, xn[j2+3], s_excf[j2+3]);
}
L_s0 = L_add(L_s0,L_s1);
L_s2 = L_add(L_s2,L_s3);
corr = L_add(L_s0,L_s2);
```

Example 18. Filter Excitation Loop Optimized With Loop Unrolling

```
s_old = 0;
for (j = 0; j < L_SUBFR ; j+=4)
{
     s0 = h[j  ];
     s1 = h[j+1];
     s2 = h[j+2];
     s3 = h[j+3];

     L_s0 = L_mult (k_exc, s0);
     L_s1 = L_mult (k_exc, s1);
     L_s2 = L_mult (k_exc, s2);
     L_s3 = L_mult (k_exc, s3);

     s0 = s_excf[j  ];
     s1 = s_excf[j+1];
     s2 = s_excf[j+2];
     s3 = s_excf[j+3];

     L_s0 = L_shl_nosat(L_s0, h_fac);
     L_s1 = L_shl_nosat(L_s1, h_fac);
     L_s2 = L_shl_nosat(L_s2, h_fac);
     L_s3 = L_shl_nosat(L_s3, h_fac);

     s_excf[j  ] = add (extract_h (L_s0), s_old);
     s_excf[j+1] = add (extract_h (L_s1), s0);
     s_excf[j+2] = add (extract_h (L_s2), s1);
     s_excf[j+3] = add (extract_h (L_s3), s2);

     s_old = s3;
}
```

After all optimizations are complete, the resulting code is compiled for speed. **Table 2** shows the results:

Table 2. Norm_corr() Inline Optimizations

Function	Compiler Option	Speed (Cycles)	Size (Bytes)
Norm_corr	–O3	6211	748

4.1.3 Comp_corr()

The Comp_corr() function contains a loop that is very similar to the Autocorr() correlation loop. In fact, Autocorr() can reuse the Comp_corr() code. The same optimization technique that applies to Autocorr() also applies to Comp_corr(). **Table 3** shows the results.

Table 3. Comp_corr() Inline Optimizations

Function	Compiler Option	Speed (Cycles)	Size (Bytes)
Comp_corr	–O3	5848	210

4.2 Inline Speed Optimizations With Code Restructuring

Optimization with code restructuring merges every possible couple of loops, first without optimizing the resulting loops and then by optimizing them.

4.2.1 Autocorr()

Some code restructuring is required simply to maximize the merging capability. A DO-WHILE loop (**Example 19**) is replaced with a WHILE loop (**Example 20**).

Example 19. Original Autocorr() Function

```
Windowing_loop;

do{
     Energy_loop;

     if ( . . .)
     {
          Scaling_loop;
     }
}while (. . .);

Correlation_loop;
```

Example 20. Restructuring Autocorr() for Better Loop Merging

```
Windowing_loop;
Energy_loop;

While(. . .)
{
     Scaling_loop;
     Energy_loop;
}

Correlation_loop;
```

The two outer loops (windowing and energy) and the two inner loops from the WHILE loop (scale and energy) are merged. Measurement results are listed in **Section 5** on page 16. Next, we optimize the resulting loops for speed. The first loop is optimized by a factor of four (see **Example 21**), and the second by a factor of two (a factor of four results in only a size increase, as noted in **Section 3.1**, *Loop Merging*, on page 7).

Example 21. Loop Merging Optimized for Speed, Windowing and Energy

```
L_sum0 = L_sum1 = L_sum2 = L_sum3 = 0L;
for (i = 0; i < L_WINDOW; i += 4)
{
      y[i+0] = mult_r (x[i+0], wind[i+0]);
      y[i+1] = mult_r (x[i+1], wind[i+1]);
      y[i+2] = mult_r (x[i+2], wind[i+2]);
      y[i+3] = mult_r (x[i+3], wind[i+3]);

      L_sum0 = L_mac(L_sum0, y[i+0], y[i+0]);
      L_sum1 = L_mac(L_sum1, y[i+1], y[i+1]);
      L_sum2 = L_mac(L_sum2, y[i+2], y[i+2]);
      L_sum3 = L_mac(L_sum3, y[i+3], y[i+3]);
}
L_sum = L_add(L_add(L_sum0, L_sum1), L_add(L_sum2, L_sum3));
```

Example 22. Loop Merging Optimized for Speed, Scaling and Energy

```
L_sum0 = L_sum1 = 0L;
for (i = 0; i < L_WINDOW; i += 2)
{
      t0 = mult(y[i+0], (1 << (15 - 2)));
      t1 = mult(y[i+1], (1 << (15 - 2)));

      L_sum0 = L_mac(L_sum0, t0, t0);
      L_sum1 = L_mac(L_sum1, t1, t1);

      y[i+0] = t0; y[i+1] = t1;
}
L_sum = L_add(L_sum0, L_sum1);
```

4.2.2 Norm_corr()

The first two loops (scaling and energy) and the first two inner loops (energy and simple correlation) were merged. The measurement results are listed in **Section 5**, *Results*, on page 16. Next, we optimize the resulting loops for speed—by a factor of two because a factor of four results only in a size increase, as noted in **Section 3.1**, *Loop Merging*, on page 7).

Example 23. Loop Merging Optimized for Speed, Correlation and Energy

```
L_s0 = L_s1 = L_s2 = L_s3 = 0;
for (j = 0; j < L_SUBFR; j+=2) {
      L_s0 = L_mac (L_s0, s_excf[j+0], s_excf[j+0]);
      L_s1 = L_mac (L_s1, s_excf[j+1], s_excf[j+1]);
      L_s2 = L_mac (L_s2, xn[j+0], s_excf[j+0]);
      L_s3 = L_mac (L_s3, xn[j+1], s_excf[j+1]);
}
norm = L_add(L_s0,L_s1);
corr = L_add(L_s2,L_s3);
```

4.2.3 Comp_corr()

No loop merging is possible for Comp_corr() (see **Appendix D**, *Loop Merging, Optimized for Speed*, on page 29).

5 Results

This section summarizes the results of optimizing functions with/without code structuring and with/without speed optimizations.

Table 4. Original Version of Functions

Function	Compiler Option	Speed (Cycles)	Size (Bytes)
Autocorr	–O3	3997	336
	–O3 –Os	7379	220
Norm_corr	–O3	11522	666
	–O3 –Os	16141	398
Comp_corr	–O3	23640	108
	–O3 –Os	43747	76

Table 5. Inline Optimizations With No Code Restructuring

Function	Compiler Option	Speed (Cycles)	Size (Bytes)
Autocorr	–O3	1004	566
Norm_corr	–O3	6211	748
Comp_corr	–O3	5848	210
Total		13063	1560

Table 6. Code Reuse with No Code Restructuring

Function	Compiler Option	Speed (Cycles)	Size (Bytes)
Autocorr	–O3 –Os	1104	244
Norm_corr	–O3 –Os	7769	320
Comp_corr	ASM	5485	146
Energy4x	ASM	—	44
ScaleRight4x	ASM	—	64
Windowing4x	ASM	—	48
Correlation4x	ASM	—	52
Filter_excitation	–O3	—	110
Total		14358	1028

Table 7. Loop Merging With No Speed Optimizations

Function	Compiler Option	Speed (Cycles)	Size (Bytes)
Autocorr	–O3	3214	316
	–O3–Os	6301	218
Norm_corr	–O3	9236	430
	–O3 –Os	13029	296

Table 8. Loop Merging With Speed Optimizations

Function	Compiler Option	Speed (Cycles)	Size (Bytes)
Autocorr	–O3	930	534
Norm_corr	–O3	5648	568
Comp_corr (no loop merging)	–O3	5848	210
Total		12426	1312

6 Conclusions

Loop merging should be used as much as possible because it has a positive impact on both speed and size. Loop merging should be applied before any other transformations. Attention is required when the results of loop merging are optimized for speed. Optimization by a factor of four does not necessarily increase speed more than optimization by a factor of two (see **Section 3.1**, *Loop Merging*, on page 7.)

Code reuse is an efficient technique if the size constraints are important. As **Table 3** shows, there can be a significant size increase without a corresponding decrease in speed. Code reuse is not easy to achieve after loop merging. For example, the Autocorr() and Norm_corr() functions both have energy and scaling loops that make them promising reuse candidates. However, when the loop merging is performed, both functions contain a loop that results from merging the energy and scaling loops. For Autocorr(), the energy operation is performed on the scaled vector, and for Norm_corr(), the energy is performed on the original vector. Thus, this section of code cannot be reused.

The multisampling technique works with four samples in parallel. Using only two samples in parallel obtains similar, or almost similar, speed results yet reduces the code size by half. The key is to use a multisample by two combined with split-computation in the inner loop. **Table 9** summarizes the measurement results and compares the optimization techniques in terms of size and speed. All comparison figures reference the first row of the table. Since for vocoder projects, efficiency is defined by the number of channels divided by the project code size, and since the number of channels is inversely proportional to the number of cycles, we define a performance factor (F), as follows:

Equation 6

$$F = \frac{1}{speed \bullet size}$$

Speed is measured in millions of cycles, and size is measured in number of KB. The best method in terms of both speed and size is the one with the highest F value.

Table 9. Optimization Techniques Compared

Optimization Type	Speed		Size		F
	Cycles	Speed-up	Bytes	Gain	
Original form, compiled for speed	39159	1	1110	0	23
Inline speed optimization	13063	2.99	1560	-40.5%	49
Code reuse and speed optimization	14358	2.73	1028	+7.4%	68

Table 9. Optimization Techniques Compared

Optimization Type	Speed		Size		F
	Cycles	Speed-up	Bytes	Gain	
Loop merging and speed optimization	12426	3.15	1312	–18%	61

Increased speed is defined as the number of times the resulting code is faster than the original code. Increased size is defined as the percentage of the original code size gained by the resulting code. If the gain is a negative number, then the resulting code is greater in size than the original code.

Table 9 can be interpreted in different ways. If the interest is speed only, then the speed column determines the best method to use. If the interest is size only, then we refer to the size column. However, if we compile the original project only for size, the size results may very good while the speed results are catastrophic. If speed and size are both important, then the *F* column yields the best optimization method.

7 References

[1] *SC140 DSP Core Reference Manual*, order number (MNSC140CORE).

[2] *SC100 Assembly Language Tools User's Manual* (MNSC100ALT).

[3] *SC100 Application Binary Interface Reference Manual (*MNSC100ABI).

[4] *Metrowerks Enterprise C Compiler User's Manual*. (Available at the Metrowerks web site.)

[5] *Speed and Code-Size Trade-off with the StarCore SC140* (AN1838).

Appendix A Original Code

Example 24. Autocorr

```
Word16 Autocorr(Word16 x[], Word16 m, Word32 r[], const Word16 wind[])
{
      Word16 i, j, norm, overfl, overfl_shft;
      Word16 y[L_WINDOW];
      Word32 sum;

      /* Windowing of signal */
      for (i = 0; i < L_WINDOW; i++)
      {
           y[i] = mult_r (x[i], wind[i]);
      }

      overfl_shft = 0;
      do {
           overfl = 0;
           sum = 0L;

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

           if (L_sub (sum, MAX_32) == 0L)
           {
                overfl_shft = add (overfl_shft, 4);
                overfl = 1;

                for (i = 0; i < L_WINDOW; i++)
                {
                     y[i] = shr (y[i], 2);
                }
           }
      } while (overfl != 0);

      /* Avoid the case of all zeros */
      sum = L_add (sum, 1L);

      /* Normalization of r[0] */
      norm = norm_l (sum);
      r[0] = L_shl (sum, norm) & -2;

      for (i = 1; i <= m; i++)
      {
           sum = 0;
           for (j = 0; j < L_WINDOW - i; j++)
           {
                sum = L_mac (sum, y[j], y[j + i]);
           }
           r[i] = L_shl (sum, norm);
      }

      norm = sub (norm, overfl_shft);
      return norm;
}
```

Example 25. Norm_corr()

```
void Norm_Corr (Word16 exc[], Word16 xn[], Word16 h[], Word16 L_subfr,
                Word16 t_min, Word16 t_max, Word16 corr_norm[])
{
     Word16 i, j, k;
     Word32 corr, norm, s;
     Word16 scaling, h_fac, *s_excf, scaled_excf[L_SUBFR], excf[L_SUBFR];

     k = -t_min;

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

     /* scale "excf[]" to avoid overflow */
     for (j = 0; j < L_subfr; j++)
     {
          scaled_excf[j] = shr (excf[j], 2);
     }

     /* Compute 1/sqrt(energy of excf[]) */
     s = 0;
     for (j = 0; j < L_subfr; j++)
     {
          s = L_mac (s, excf[j], excf[j]);
     }

     /* if (s <= 2^26) */
     if (L_sub (s, 67108864L) <= 0)
     {
          s_excf = excf;
          h_fac = 15 - 12;
          scaling = 0;
     }
     else
     {
          s_excf = scaled_excf;
          h_fac = 15 - 12 - 2;
          scaling = 2;
     }

     for (i = t_min; i <= t_max; i++)
     {
          /* Compute 1/sqrt(energy of excf[]) */
          norm = 0;
          for (j = 0; j < L_subfr; j++)
          {
               norm = L_mac (norm, s_excf[j], s_excf[j]);
          }
          norm = Inv_sqrt (norm) & -2;

          /* Compute correlation between xn[] and excf[] */
          corr = 0;
          for (j = 0; j < L_subfr; j++)
          {
               corr = L_mac (corr, xn[j], s_excf[j]);
          }
```

```
            /* Normalize correlation = correlation * (1/sqrt(energy)) */
            s = Mpy_32 (corr, norm);
            corr_norm[i] = extract_h (L_shl (s, 16));

            /* modify the filtered excitation excf[] for the next iteration
*/
            if (sub (i, t_max) != 0)
            {

                  k--;
                  for (j = L_subfr - 1; j > 0; j--)
                  {
                        s = L_mult (exc[k], h[j]);
                        s = L_shl (s, h_fac);
                        s_excf[j] = add (extract_h (s), s_excf[j - 1]);
                  }
                  s_excf[0] = shr (exc[k], scaling);
            }
      }
      return;
}
```

Example 26. Comp_corr()

```
void comp_corr (Word16 s[], Word16 scal_sig[], Word16 L_frame,
                Word16 nr_loop, Word32 corr[])
{
      Word16 i, j;
      Word32 L_t;

      for (i = 0; i <= nr_loop; i++)
      {
            L_t = 0;
            for (j = 0; j < L_frame; j++)
            {
                  L_t = L_mac (L_t, scal_sig[j], s[i+j+1]);
            }
            corr[i]   = L_t;
      }
      return;
}
```

Appendix B Inline Optimizations for Speed

Example 27. Autocorr

```
Word16 Autocorr (Word16 x[], Word16 m, Word32 r[], const Word16 wind[])
{
      Word16 i, i1, j, norm, t0, t1, t2;
      Word16 y[L_WINDOW + 4];
#pragma align y 8

      Word16 y0, y1, y2, y3;
      Word32 sum, sum0, sum1, sum2, sum3, sumA, sumB;
      Word16 overfl, overfl_shft;
```

```
     /* necessary for the loop unrolling technique */
#pragma align * x 8
#pragma align * r 8
#pragma align * wind 8

     y[L_WINDOW + 0] = 0;
     y[L_WINDOW + 1] = 0;
     y[L_WINDOW + 2] = 0;
     y[L_WINDOW + 3] = 0;

     /* Windowing of signal */
     /* loop unrolling */
     for (i = 0; i < L_WINDOW; i += 4)
     {
          y[i+0] = mult_r (x[i+0], wind[i+0]);
          y[i+1] = mult_r (x[i+1], wind[i+1]);
          y[i+2] = mult_r (x[i+2], wind[i+2]);
          y[i+3] = mult_r (x[i+3], wind[i+3]);
     }

     /* Compute r[0] and test for overflow */
     overfl_shft = 0;
     do {
          overfl = 0;
          sum0 = sum1 = sum2 = sum3 = 0L;
          /* split summation */
          for (i1 = 0; i1 < L_WINDOW; i1 += 4)
          {
               sum0 = L_mac(sum0, y[i1+0], y[i1+0]);
               sum1 = L_mac(sum1, y[i1+1], y[i1+1]);
               sum2 = L_mac(sum2, y[i1+2], y[i1+2]);
               sum3 = L_mac(sum3, y[i1+3], y[i1+3]);
          }
          sum = L_add(L_add(sum0, sum1), L_add(sum2, sum3));

          /* If overflow divide y[] by 4 */
          if (L_sub (sum, MAX_32) == 0L)
          {
               overfl_shft = add (overfl_shft, 4);
               /* Set the overflow flag */
               overfl = 1;

               /* loop unrolling */
               for (i = 0; i < L_WINDOW; i += 4))
               {
                    /* y0 = shr(y[i], 2) */
                    y0 = mult(y[i+0], (1<<(15-2)));
                    y1 = mult(y[i+1], (1<<(15-2)));
                    y2 = mult(y[i+2], (1<<(15-2)));
                    y3 = mult(y[i+3], (1<<(15-2)));
```

```
                    y[i+0] = y0;
                    y[i+1] = y1;
                    y[i+2] = y2;
                    y[i+3] = y3;
               }
          }
     } while (overfl != 0);

     /* Avoid the case of all zeros */
     sum = L_add (sum, 1L);

     /* Normalization of r[0] */
     norm = norm_l (sum);
     r[0] = L_shl_nosat (sum, norm) & -2;

     /* r[1] to r[m] */
     for (i = 1; i <= m; i += 2)
     {
          sum0 = sum1 = sum2 = sum3 = 0L;
          t0 = y[i];
          for (j = 0; j < L_WINDOW - i; j += 4)
          {
          #pragma loop_count(55, 60, 1)

               t1 = y[j + i + 1];
               t2 = y[j + i + 2];

               sum0 = L_mac (sum0, y[j + 0], t0);
               sum1 = L_mac (sum1, y[j + 0], t1);
               sum2 = L_mac (sum2, y[j + 1], t1);
               sum3 = L_mac (sum3, y[j + 1], t2);

               t1 = y[j + i + 3];
               t0 = y[j + i + 4];

               sum0 = L_mac (sum0, y[j + 2], t2);
               sum1 = L_mac (sum1, y[j + 2], t1);
               sum2 = L_mac (sum2, y[j + 3], t1);
               sum3 = L_mac (sum3, y[j + 3], t0);
          }
          sumA = L_add(sum0, sum2);
          sumB = L_add(sum1, sum3);

          r[i] = L_shl_nosat (sumA, norm);
          r[i + 1] = L_shl_nosat (sumB, norm);
     }

     norm = sub (norm, overfl_shft);

     return norm;
}
```

Example 28. Norm_corr

```
void Norm_Corr (Word16 exc[], Word16 xn[], Word16 h[], Word16 t_min,
                Word16 t_max, Word16 corr_norm[])
{
#pragma align *exc 8
#pragma align *xn 8
#pragma align *h 8
     Word16 i, k, j1, j2, j3,j;
     Word32 corr, norm;
     Word32 s;

     /* Usally dynamic allocation of (L_subfr) */
     Word16 excf[L_SUBFR];
#pragma align excf 8
     Word16 scaling, h_fac;
     Word16 *s_excf;
#pragma align *s_excf 8
     Word16 scaled_excf[L_SUBFR];
#pragma align scaled_excf 8
     Word32 L_s0,L_s1,L_s2,L_s3;
     Word16 k_exc;
     Word16 s0,s1,s2,s3,s_old;

     k = -t_min;

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

     /* scale "excf[]" to avoid overflow */
     for (j = 0; j < L_SUBFR; j += 4)
     {
          scaled_excf[j  ] = mult (excf[j  ], (1<<(15-2)));
          scaled_excf[j+1] = mult (excf[j+1], (1<<(15-2)));
          scaled_excf[j+2] = mult (excf[j+2], (1<<(15-2)));
          scaled_excf[j+3] = mult (excf[j+3], (1<<(15-2)));
     }

     /* Compute 1/sqrt(energy of excf[]) */
     L_s0 = L_s1 = L_s2 = L_s3 = 0;
     for (j = 0; j < L_SUBFR; j += 4)
     {
          L_s0 = L_mac (L_s0, excf[j  ], excf[j  ]);
          L_s1 = L_mac (L_s1, excf[j+1], excf[j+1]);
          L_s2 = L_mac (L_s2, excf[j+2], excf[j+2]);
          L_s3 = L_mac (L_s3, excf[j+3], excf[j+3]);
     }
     L_s0 = L_add(L_s0,L_s1);
     L_s2 = L_add(L_s2,L_s3);
     s = L_add(L_s0,L_s2);

                    s_excf = excf;

                    h_fac = 3;
                    scaling = 0;
```

```
                /* if (s <= 2^26) */
                if (L_sub (s, 67108864L) > 0)
                {
                     s_excf = scaled_excf;
                     h_fac = 1;
                     scaling = 2;
                }

           /* loop for every possible period */
           for (i = t_min; i <= t_max; i++)
           {
                /* Compute 1/sqrt(energy of excf[]) */
                L_s0 = L_s1 = L_s2 = L_s3 = 0;
                for (j1 = 0; j1 < L_SUBFR; j1 += 4)
                {
                     L_s0 = L_mac (L_s0, s_excf[j1  ], s_excf[j1  ]);
                     L_s1 = L_mac (L_s1, s_excf[j1+1], s_excf[j1+1]);
                     L_s2 = L_mac (L_s2, s_excf[j1+2], s_excf[j1+2]);
                     L_s3 = L_mac (L_s3, s_excf[j1+3], s_excf[j1+3]);
                }
                L_s0 = L_add(L_s0,L_s1);
                L_s2 = L_add(L_s2,L_s3);
                norm = L_add(L_s0,L_s2);

                norm = Inv_sqrt (norm) & -2;

                /* Compute correlation between xn[] and excf[] */
                L_s0 = L_s1 = L_s2 = L_s3 = 0;
                for (j2 = 0; j2 < L_SUBFR; j2 += 4)
                {
                     L_s0 = L_mac (L_s0, xn[j2  ], s_excf[j2  ]);
                     L_s1 = L_mac (L_s1, xn[j2+1], s_excf[j2+1]);
                     L_s2 = L_mac (L_s2, xn[j2+2], s_excf[j2+2]);
                     L_s3 = L_mac (L_s3, xn[j2+3], s_excf[j2+3]);
                }
                L_s0 = L_add(L_s0,L_s1);
                L_s2 = L_add(L_s2,L_s3);
                corr = L_add(L_s0,L_s2);

                /* Normalize correlation = correlation * (1/sqrt(energy)) */
                s = Mpy_32 (corr, norm);
                corr_norm[i] = extract_h (L_shl (s, 16));

                /* modify the filtered excitation excf[] for the next iteration
*/
                k--;
                k_exc = exc[k];
                s_old = 0;
                for (j3 = 0; j3 < L_SUBFR ; j3 += 4)
                {
                     s0 = h[j3  ];
                     s1 = h[j3+1];
                     s2 = h[j3+2];
                     s3 = h[j3+3];
```

```
                L_s0 = L_mult (k_exc, s0);
                L_s1 = L_mult (k_exc, s1);
                L_s2 = L_mult (k_exc, s2);
                L_s3 = L_mult (k_exc, s3);

                s0 = s_excf[j3  ];
                s1 = s_excf[j3+1];
                s2 = s_excf[j3+2];
                s3 = s_excf[j3+3];

                L_s0 = L_shl_nosat(L_s0, h_fac);
                L_s1 = L_shl_nosat(L_s1, h_fac);
                L_s2 = L_shl_nosat(L_s2, h_fac);
                L_s3 = L_shl_nosat(L_s3, h_fac);

                s_excf[j3  ] = add (extract_h (L_s0), s_old);
                s_excf[j3+1] = add (extract_h (L_s1), s0);
                s_excf[j3+2] = add (extract_h (L_s2), s1);
                s_excf[j3+3] = add (extract_h (L_s3), s2);

                s_old = s3;
            }
            s_excf[0] = shr_nosat(k_exc, scaling);
        }

        return;
}
```

Example 29. Comp_corr

```
void comp_corr (Word16 s[], Word16 scal_sig[], Word16 L_frame, Word16
nr_loop,
          Word32 corr[])
{
#pragma align *s 8
#pragma align *scal_sig 8
#pragma align *corr 8
     Word16 i, j;
     Word32 L_t0,L_t1,L_t2,L_t3;
     Word16 s0,s1,s2;

     for (i = 0; i <= nr_loop; i+=2)
     {
     #pragma loop_count(62,63,1)

          L_t0 = L_t1 = L_t2 = L_t3= 0;
          s0 = s[i+1];
          for (j = 0; j < L_frame; j+=4)
          {
          #pragma loop_count(20,40,1)

               s1 = s[j+i+2];
               s2 = s[j+i+3];

               L_t0 = L_mac (L_t0, scal_sig[j  ], s0);
               L_t1 = L_mac (L_t1, scal_sig[j  ], s1);
               L_t2 = L_mac (L_t2, scal_sig[j+1], s1);
               L_t3 = L_mac (L_t3, scal_sig[j+1], s2);
```

```
                s1 = s[j+i+4];
                s0 = s[j+i+5];

                L_t0 = L_mac (L_t0, scal_sig[j+2], s2);
                L_t1 = L_mac (L_t1, scal_sig[j+2], s1);
                L_t2 = L_mac (L_t2, scal_sig[j+3], s1);
                L_t3 = L_mac (L_t3, scal_sig[j+3], s0);
           }

           corr[i]   = L_add(L_t0,L_t2);
           corr[i+1] = L_add(L_t1,L_t3);
      }

      return;
}
```

Appendix C Code Reuse

Example 30. Autocorr

```
Word16 Autocorr (Word16 x[], Word16 m, Word32 r[], const Word16 wind[])
{
      Word16 i, norm;
      Word16 y[L_WINDOW+10];
      Word32 sum;
      Word16 overfl, overfl_shft;

#pragma align * x 8
#pragma align * y 8
#pragma align * r 8
#pragma align * wind 8

      for (i=0; i<10; i+=2)
      {
           y[L_WINDOW+i+0] = 0;
           y[L_WINDOW+i+1] = 0;
      }

      /* Windowing of signal */
      Windowing(x, (Word16*) wind, L_WINDOW, y);

      /* Compute r[0] and test for overflow */
      overfl_shft = 0;
      do {
           overfl = 0;
           sum = 0L;
           /* Energy of a signal */
           sum = Energy(y, L_WINDOW);
```

```
            /* If overflow divide y[] by 4 */
            if (L_sub (sum, MAX_32) == 0L)
            {
                  overfl_shft = add (overfl_shft, 4);
                  /* Set the overflow flag */
                  overfl = 1;
                  /* Scaling a vector */
                  shr_with_mpy_vector(y, y, L_WINDOW, 1 << (15 - 2));
            }
      } while (overfl != 0);

      /* Avoid the case of all zeros */
      sum = L_add (sum, 1L);

      /* Normalization of r[0] */
      norm = norm_l (sum);
      sum = L_shl (sum, norm) & -2;

      comp_corr(y, y, L_WINDOW, m-1, r);

      /* m is allways 10 *
      for (i = 10; i > 0; i--)
      {
            r[i] = L_shl_nosat (r[i-1], norm);
      }

      r[0] = sum;
      norm = sub (norm, overfl_shft);

      return norm;
}
```

Example 31. Norm_corr

```
void Norm_Corr (Word16 exc[], Word16 xn[], Word16 h[], Word16 t_min,
          Word16 t_max, Word16 corr_norm[])
{
#pragma align *exc 8
#pragma align *xn 8
#pragma align *h 8
      Word16 i, k;
      Word32 corr, norm;
      Word32 s;

      /* Usally dynamic allocation of (L_subfr) */
      Word16 excf[L_SUBFR];
#pragma align excf 8
      Word16 scaling, h_fac, *s_excf;
      Word16 scaled_excf[L_SUBFR];
#pragma align scaled_excf 8

      k = -t_min;

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

      /* scale "excf[]" to avoid overflow */
      shr_with_mpy_vector(scaled_excf, excf, L_SUBFR, 1 << (15 - 2));
```

```
     /* Compute 1/sqrt(energy of excf[]) */
     s = Energy(excf,L_SUBFR);

     /* if (s <= 2^26) */
     if (L_sub (s, 67108864L) <= 0)
     {
          s_excf = excf;
          h_fac = 3;
          scaling = 0;
     }
     else
     {
          /* "excf[]" is divided by 2 */
          s_excf = scaled_excf;
          h_fac = 1;
          scaling = 2;
     }

     /* loop for every possible period */
     for (i = t_min; i <= t_max; i++)
     {
          /* Compute 1/sqrt(energy of excf[]) */
          norm = Energy(s_excf,L_SUBFR);
          norm = Inv_sqrt (norm) & -2;

          /* Compute correlation between xn[] and excf[] */
          corr = Correlation(xn,s_excf,L_SUBFR);

          /* Normalize correlation = correlation * (1/sqrt(energy)) */
          s = Mpy_32 (corr, norm);
          corr_norm[i] = extract_h (L_shl (s, 16));

          /* modify the filtered excitation excf[] for the next iteration
*/
          if (sub (i, t_max) != 0)
          {
               k--;
               Filter_Excitation(s_excf,h,exc[k],h_fac,scaling,L_SUBFR);
          }
     }
     return;
}
```

Appendix D Loop Merging, Optimized for Speed

Example 32. Autocorr

```
Word16 Autocorr ( Word16 x[], Word16 m, Word32 r[], const Word16 wind[])
{
     Word16 i, j, norm, t0, t1, t2;
     Word16 y[L_WINDOW + 4];
#pragma align y 8

     Word32 sum, sum0, sum1, sum2, sum3, sumA, sumB;
     Word16 overfl_shft;
```

```
/* necessary for the loop unrolling technique */
#pragma align * x 8
#pragma align * r 8
#pragma align * wind 8

     y[L_WINDOW + 0] = 0;
     y[L_WINDOW + 1] = 0;
     y[L_WINDOW + 2] = 0;
     y[L_WINDOW + 3] = 0;

     sum0 = sum1 = sum2 = sum3 = 0L;
     /* Merging the windowing and the energy loops witht loop
optimization. */
     for (i = 0; i < L_WINDOW; i += 4)
     {
          y[i+0] = mult_r (x[i+0], wind[i+0]);
          y[i+1] = mult_r (x[i+1], wind[i+1]);
          y[i+2] = mult_r (x[i+2], wind[i+2]);
          y[i+3] = mult_r (x[i+3], wind[i+3]);

          sum0 = L_mac(sum0, y[i+0], y[i+0]);
          sum1 = L_mac(sum1, y[i+1], y[i+1]);
          sum2 = L_mac(sum2, y[i+2], y[i+2]);
          sum3 = L_mac(sum3, y[i+3], y[i+3]);
     }
     sum = L_add(L_add(sum0, sum1), L_add(sum2, sum3));

     /* Compute r[0] and test for overflow */
     overfl_shft = 0;
     while (L_sub (sum, MAX_32) == 0L)
     {
          overfl_shft = add (overfl_shft, 4);
          sum0 = sum1 = sum2 = sum3 = 0L;
          /* Merging the scale and the energy loops with loop
optimization. */
          /* The "shr" were replaced by "mult". */
          for (i = 0; i < L_WINDOW; i += 2)
          {
               t0 = mult(y[i+0], (1 << (15 - 2)));
               t1 = mult(y[i+1], (1 << (15 - 2)));

               sum0 = L_mac(sum0, t0, t0);
               sum1 = L_mac(sum1, t1, t1);

               y[i+0] = t0;
               y[i+1] = t1;
          }
          sum = L_add(sum0, sum1);
     }

     /* Avoid the case of all zeros */
     sum = L_add (sum, 1L);

     /* Normalization of r[0] */
     norm = norm_l (sum);
     r[0] = L_shl_nosat (sum, norm) & -2;
```

```
     /* r[1] to r[m] */
     for (i = 1; i <= m; i += 2)
     {
          sum0 = sum1 = sum2 = sum3 = 0L;
          t0 = y[i];
          for (j = 0; j < L_WINDOW - i; j += 4)
          {
          #pragma loop_count(55, 60, 1)

               t1 = y[j + i + 1];
               t2 = y[j + i + 2];

               sum0 = L_mac (sum0, y[j + 0], t0);
               sum1 = L_mac (sum1, y[j + 0], t1);
               sum2 = L_mac (sum2, y[j + 1], t1);
               sum3 = L_mac (sum3, y[j + 1], t2);

               t1 = y[j + i + 3];
               t0 = y[j + i + 4];

               sum0 = L_mac (sum0, y[j + 2], t2);
               sum1 = L_mac (sum1, y[j + 2], t1);
               sum2 = L_mac (sum2, y[j + 3], t1);
               sum3 = L_mac (sum3, y[j + 3], t0);
          }
          sumA = L_add(sum0, sum2);
          sumB = L_add(sum1, sum3);

          r[i] = L_shl_nosat (sumA, norm);
          r[i + 1] = L_shl_nosat (sumB, norm);
     }

     norm = sub (norm, overfl_shft);

     return norm;
}
```

Example 33. Norm_corr

```
void Norm_Corr (Word16 exc[], Word16 xn[], Word16 h[], Word16 t_min,
          Word16 t_max, Word16 corr_norm[])
{
#pragma align *exc 8
#pragma align *xn 8
#pragma align *h 8
     Word16 i, j, j1, k;
     Word32 corr, norm;
     Word32 s;
```

```
     Word16 excf[L_SUBFR];
#pragma align excf 8
     Word16 scaling, h_fac;
     Word16 *s_excf;
#pragma align *s_excf 8
     Word16 scaled_excf[L_SUBFR];
#pragma align scaled_excf 8
     Word32 L_s0,L_s1,L_s2,L_s3;
     Word16 k_exc;
     Word16 s0,s1,s2,s3,s_old;

     k = -t_min;

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

     /* scale "excf[]" to avoid overflow */
     /* Compute 1/sqrt(energy of excf[]) */
     L_s0 = L_s1 = 0;
     for (j = 0; j < L_SUBFR; j+=2)
     {
          scaled_excf[j  ] = mult (excf[j  ], (1<<(15-2)));
          scaled_excf[j+1] = mult (excf[j+1], (1<<(15-2)));

          L_s0 = L_mac (L_s0, excf[j  ], excf[j  ]);
          L_s1 = L_mac (L_s1, excf[j+1], excf[j+1]);
     }
     s = L_add(L_s0,L_s1);

     s_excf = excf;
     h_fac = 3;
     scaling = 0;
     /* if (s <= 2^26) */
     if (L_sub (s, 67108864L) > 0)
     {
          s_excf = scaled_excf;
          h_fac = 1;
          scaling = 2;
     }

     /* loop for every possible period */
     for (i = t_min; i <= t_max; i++)
     {
          /* Compute 1/sqrt(energy of excf[]) */
          /* Compute correlation between xn[] and excf[] */
          L_s0 = L_s1 = L_s2 = L_s3 = 0;
          for (j1 = 0; j1 < L_SUBFR; j1+=2)
          {
               L_s0 = L_mac (L_s0, s_excf[j1], s_excf[j1]);
               L_s1 = L_mac (L_s1, s_excf[j1+1], s_excf[j1+1]);
               L_s2 = L_mac (L_s2, xn[j1], s_excf[j1]);
               L_s3 = L_mac (L_s3, xn[j1+1], s_excf[j1+1]);
          }

          norm = L_add(L_s0,L_s1);
          corr = L_add(L_s2,L_s3);
          norm = Inv_sqrt (norm) & -2;
```

```
                    /* Normalize correlation = correlation * (1/sqrt(energy)) */

                    s = Mpy_32 (corr, norm);

                    corr_norm[i] = extract_h (L_shl (s, 16));

                    /* modify the filtered excitation excf[] for the next iteration
            */
                    k--;
                    k_exc = exc[k];
                    s_old = 0;
                    for (j = 0; j < L_SUBFR ; j+=4)
                    {
                          s0 = h[j  ];
                          s1 = h[j+1];
                          s2 = h[j+2];
                          s3 = h[j+3];

                          L_s0 = L_mult (k_exc, s0);
                          L_s1 = L_mult (k_exc, s1);
                          L_s2 = L_mult (k_exc, s2);
                          L_s3 = L_mult (k_exc, s3);

                          s0 = s_excf[j  ];
                          s1 = s_excf[j+1];
                          s2 = s_excf[j+2];
                          s3 = s_excf[j+3];

                          L_s0 = L_shl_nosat(L_s0, h_fac);
                          L_s1 = L_shl_nosat(L_s1, h_fac);
                          L_s2 = L_shl_nosat(L_s2, h_fac);
                          L_s3 = L_shl_nosat(L_s3, h_fac);

                          s_excf[j  ] = add (extract_h (L_s0), s_old);
                          s_excf[j+1] = add (extract_h (L_s1), s0);
                          s_excf[j+2] = add (extract_h (L_s2), s1);
                          s_excf[j+3] = add (extract_h (L_s3), s2);

                        s_old = s3;
                    }
                    s_excf[0] = shr_nosat(k_exc, scaling);
            }
            return;
}
```

NOTES:

NOTES:

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

AN2266
Rev. 2
11/2004

freescale™
semiconductor