AN2197
Rev. 1, 1/2005

Implementing the Levinson-Durbin Algorithm on the StarCore™ SC140/SC1400 Cores

By Corneliu Margina and Bogdan Costinescu

This application note describes a set of techniques for implementing and optimizing the Levinson-Durbin algorithm on DSPs based on the StarCore™ SC140/SC1400 cores. This algorithm is used in linear prediction coding (LPC) to compute the linear prediction filter coefficients in speech encoders. The Levinson-Durbin algorithm uses the autocorrelation method to estimate the linear prediction parameters for a segment of speech.

Linear prediction coding, also known as linear prediction analysis (LPA), is used to represent the shape of the spectrum of a segment of speech with relatively few parameters. This coding eliminates redundancy in the short-term correlation of adjacent samples, thereby providing more efficient coding.

CONTENTS

freescale™ semiconductor

1 Vocoder Recommendations

Low bit rate speech coders used in digital communications systems use audio signal compression to eliminate redundancy, thus reducing bandwidth. ITU-T has proposed several algorithms for speech signal coding at a low bit rate. This section surveys the vocoders for which the Levinson-Durbin algorithm was implemented and optimized. These include ITU-T Recommendations G.723-1, G.729, and G.729A, as well as the ETSI GSM EFR vocoder.

1.1 ITU-T Recommendation G.723-1

The G.723-1 system encodes the audio signal in frames using linear predictive analysis-by-synthesis coding. The coder uses a limited amount of complexity to represent speech at high quality. The G.723-1 coder has two associated bit rates—5.3 and 6.3 kbits per second. Although both bit rates provide good audio reproduction quality, the higher bit rate produces higher quality than the lower bit rate. It is possible to switch between the two bit rates at any 30-ms frame boundary.

In this system, the analog voice input signal is passed through a telephone bandwidth filter (Recommendation G.712 [5]), sampled at 8000 Hz, and then converted at 16-bit linear PCM to generate the encoder input. The output of the decoder is converted back to an analog signal in the same way.

Linear prediction analysis-by-synthesis is used to minimize the weighted error signal. The encoder operates on frames of 240 samples each that are equivalent to 30 ms at an 8000 Hz sample rate. A high pass filter is used to remove the DC component. Each frame is then divided into four sub-frames of 60 samples each.

A 10^{th} order LPC filter is computed for every sub-frame using the unprocessed input signal. The LPC filter for the last sub-frame is quantized using a predictive split vector quantizer. The LPC unquantized coefficients are then used to construct the perceptually weighted speech signal. Using pitch estimation, a shaping noise filter is constructed and combined with the LPC synthesis filter and the formant perceptual weighting filter to construct an impulse response. The pitch period is computed as a small differential value using pitch estimation and the impulse response to compute a closed loop pitch predictor. Both pitch period and the differential value are transmitted to the decoder.

The decoder constructs the LPC synthesis filter using the quantized LPC indices. It also decodes the adaptive codebook and fixed codebook excitations and inputs them to the LPC synthesis filter. The excitation signal for the higher rate coder is Multipulse Maximum Likelihood Quantization (MP-MLQ). For the lower rate coder the excitation signal is Algebraic-Code-Excited Linear-Prediction (ACELP).

The vocoder operates on 30 ms frames with 7.5 ms look-ahead for linear prediction analysis. The overall algorithmic delay is 37.5 ms.

1.2 ITU-T Recommendation G.729

ITU-T Recommendation G.729 proposes an algorithm for the speech signal coding at 8 kbit/s using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP). The analog voice input signal is passed through a telephone bandwidth filter, sampled at 8000 Hz, and converted at 16-bit linear PCM to generate the encoder input. The output of the decoder is converted back to an analog signal in the same way. The coder operates on 10 ms speech frames equivalent to 80 samples at a sampling rate of 8000 samples per second, with 5 ms look-ahead for linear prediction analysis. The overall algorithmic delay is 15 ms.

The encoder analyzes the speech signal to extract the CELP model parameters, which include linear prediction filter coefficients and adaptive and fixed-codebook indices and gains. These parameters are transmitted to the decoder. The decoder uses the CELP parameters to retrieve the excitation and synthesis filter parameters corresponding to a 10 ms frame. The speech is reconstructed by passing the excitation signal through a short-term synthesis filter based on a 10^{th} order linear prediction filter. During the LP analysis, the LP coefficients are converted to line spectrum pairs (LSP) and then quantized using predictive two-stage vector quantization (VQ) with 18 bits. The fixed and adaptive codebook parameters are computed for every 5 ms (40-sample) sub-frame.

The long-term synthesis filter uses the adaptive-codebook approach. A postfilter enhances the reconstructed speech.

1.3 ITU-T Recommendation G.729A

The ITU-T Recommendation G.729A proposes a reduced complexity version of the 8 kbit/s CS-ACELP speech codec (ITU-T Recommendation G.729). This version is primarily for use in simultaneous voice and data applications, although it is not limited to these applications. The coder operates on 10ms frames with a 5 ms look-ahead for linear prediction analysis. The changes applied to G.729A to reduce the codec algorithm complexity include the following:

- The perceptual weighting filter is based on the quantized LP filter coefficients. This reduces the number of filtering operations required to compute the impulse response, compute the target signal and update the filter states.
- Open-loop pitch analysis is simplified by using decimation while computing the correlations of the weighted speech.
- The adaptive codebook search is simplified by maximizing the correlation between the past excitation and the backward filtered target signal.
- The fixed algebraic codebook search is simplified by using an iterative depth-first tree search approach.
- The harmonic postfilter in the decoder is simplified by using only integer delays.

1.4 ETSI GSM EFR

The Global System for Mobile Communication Enhanced Full Rate (GSM EFR) encodes speech at 12.2 kbit/s using algebraic code excited linear prediction (ACELP). The algorithm divides the 20 ms frame into four 5 ms sub-frames. A 10th order linear prediction analysis, including an asymmetric 30 ms window, is performed twice per frame. The first window emphasizes the second sub-frame, and the second window emphasizes the fourth sub-frame.

The LP parameters are converted to line spectral pairs. The algorithm incorporates an initial open-loop search twice per frame, and a closed-loop search is repeated for each sub-frame. The pulse positions are optimized by a non-exhaustive analysis-by-synthesis search to minimize the perceptually weighted error. At the decoder, the synthesized speech is filtered with an adaptive postfilter. Decoded speech quality is high.

2 Background Theory

Linear prediction analysis characterizes the shape of the spectrum of a short segment of speech with a small number of parameters for efficient coding. Linear prediction, also referred to as linear prediction coding (LPC), predicts a time-domain speech sample based on a linearly-weighted combination of previous samples. LP analysis removes the redundancy in the short-term correlation of adjacent samples.

LPC determines the coefficients of a FIR filter that predicts the next value in a sequence from current and the previous inputs. This type of filter is also known as a one-step forward linear predictor. LP analysis is based on the all-pole filter described in **Equation 1**:

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 - \sum_{k=1}^{p} a_k \cdot z^{-k}} \quad \textbf{Equation 1}$$

where $\{a_k | (1 \leq k \leq p)\}$ are the predictor coefficients and p is the order of the filter.

Transforming **Equation 1** to the time-domain, as shown in **Equation 2**, predicts a speech sample based on a sum of weighted past samples.

$$s'(n) = \sum_{k=1}^{p} a_k \cdot s(n-k) \quad \textbf{Equation 2}$$

where $s'(n)$ is the predicted value based on the previous values of the speech signal $s(n)$.

LP analysis requires estimating the LP parameters for a segment of speech. The idea is to find $a_k \cdot s$ so that **Equation 2** provides the closest approximation to the speech samples. This means that $s'(n)$ is closest to $s(n)$ for all values of n in the segment. The spectral shape of $s(n)$ is assumed to be stationary across the frame, or a short segment of speech.

The error, e, between the predicted value and the actual value is

$$(n) = s(n) - s'(n \quad \textbf{Equation 3}$$

The summed squared error, E, over a finite window of length N is

$$E = \sum_{n} e^2(n) \quad \textbf{Equation 4}$$

where $0 \leq n \leq N + p - 1$.

The minimum value of E occurs when the derivative is zero with respect to each of the parameters a_k. By setting the partial derivatives of E, a set of p equations are obtained. The matrix form of these equations is

$$\begin{bmatrix} r(0) & r(1) & r(p-1) \\ r(1) & r(0) & r(p-2) \\ \cdots & \cdots & \cdots \\ (p-1) & r(p-2) & r(0) \end{bmatrix} \times \begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_p \end{bmatrix} = \begin{bmatrix} r(1 \\ r(2 \\ \cdots \\ r(p \end{bmatrix}$$

Equation 5

where $r(i)$ is the autocorrelation of lag i computed as

$$r(i) = \sum_{m=0}^{N-1-i} s(m) \cdot s(m+i)$$

Equation 6

and N is the length of the speech segment $s(n)$.

The Levinson-Durbin algorithm solves the nth order system of linear equations

$$R \cdot a = b$$

Equation 7

for the particular case where R is a Hermitian, positive-definite, Toeplitz matrix and b is identical to the first column of R shifted by one element.

The autocorrelation coefficients $r(k)$ are used to compute the LP filter coefficients a_i, $i = 1..p$, by solving the set of equations

$$\sum_{i=1}^{p} a_i \cdot r(|i-k|) = r(k)$$

Equation 8

where $k = 1..p$.

This set of equations is solved using the Levinson-Durbin recursion, **Equation 9** through **Equation 13**.

$$E(0) = r(0)$$

Equation 9

$$k_i = \frac{r(i) - \sum_{j=1}^{i-1} a_j^{i-1} \cdot r(i-j)}{E(i-1)}$$

Equation 10

$$a_i^{(i)} = k_i$$

Equation 11

$$a_j^{(i)} = a_j^{i-1} - k_i \cdot a_{i-j}^{(i-1}$$

Equation 12

$$E(i) = (1 - k_i^2) \cdot E(i-1$$

Equation 13

where $1 \leq j \leq i-1$ and $1 \leq i \leq p$.

The parameters k_i are known as the reflection parameters. If the condition $|k_i| \leq 1$ where $1 \leq i \leq p$ is satisfied, the roots of the polynomial predictor all lie within the unit circle in the z-plane, and the all-pole filter is stable.

3 Levinson-Durbin Implementation on the SC140 Core

This section presents the implementation of Levinson-Durbin algorithm for StarCore processors following the ITU-T G.729A Recommendation. The optimization techniques to increase execution speed are also described. Two implementations were made, one using C language and other using assembly language. The assembly implementation is based on the C implementation, but special techniques are applied to increase the execution speed based on StarCore processor architecture. The source code for the C implementation is listed in Appendix A, and the code for the assembly implementation is listed in Appendix B.

In G.729A, linear prediction analysis is performed once per speech frame using the autocorrelation method with a 30 ms asymmetric window. The autocorrelation coefficients of windowed speech are computed and converted to LP coefficients using the Levinson algorithm every 80 samples (10ms). The LP coefficients are then transformed to line spectrum pairs for quantization and interpolation. The interpolated quantized and unquantized filters are converted back to the LP filter coefficients to construct the synthesis and weighting filters for each subframe.

Both the C and assembly implementations resolve Equation 7 on page 5 using the Levinson-Durbin recursion by implementing the algorithm described in **Equation 9** through **Equation 13**. The algorithm is summarized in the pseudo-code shown in Code Listing 1.

Code Listing 1. Pseudo-Code of Levinson-Durbin Recursion

```
/* input data */
R[i]          - autocorrelation coefficients
A[i]          - filter coefficients
K             - reflection coefficients
Alpha         - prediction gain

/* initialization */
A[0] = 1
K = -R[1]/R[0]
A[1] = K
Alpha = R[0]* (1-K^2)                                                    (1)
For i = 2 To M
              S = SUM(R[j]*A[i-j];j=1,i-1) + R[i]                        (2)
              K = -S/Alpha
              An[j] = A[j] + K*A[i-j] /*for j=1 to i-1 where An[i] = new A[i]*/ (3)
              An[i] = K
              Alpha = Alpha * (1-K^2)
End
```

If the filter proves to be unstable when the algorithm executes, the filter coefficients from the previous execution are stored as the new values and the execution terminates. Because the algorithm contains several data dependencies, implementation on multiple execution units is difficult. However, the optimization techniques described in the following sections demonstrate how to take the most advantage of the StarCore architecture.

3.1 C Implementation

This section presents the optimization techniques applied to the original ITU-T G.729A Recommendation code of the Levinson-Durbin algorithm using the Metrowerks® CodeWarrior® for StarCore, Release 1.0. The C-optimized implementation is listed in Appendix A. The original C implementation of the Levinson-Durbin algorithm from the ITU-T G.729A Recommendation follows the pseudo-code in **Code Listing 1**. Operands used in several functions are represented in double precision format (DPF). For details on these functions and the way they are implemented, see Appendix C.

The 32-bit DPF format is designed for 16-bit processors that do not support 32-bit operations. Thus, although StarCore processors are 16-bit processors that support 32-bit operations, the 32-bit operations must be implemented in DPF format to maintain bit-exactness with the original ITU-T implementation. The two 16-bit portions, originally processed in two separate DALU registers, are combined into a single 32-bit value using only one DALU register. Thus, functions that originally received two pointers to two 16-bit arrays can now operate with one pointer to a 32-bit array. This optimization step also reduces the required number of memory moves. The least significant bit of partial or final computation results must be reset to maintain bit-exactness with G.729A.

The DPF representations of the algorithm input parameters (for example., the R[ ] vector of autocorrelation coefficients and the A[ ] vector of LPC coefficients) are changed. Therefore, the `L_Comp()` function, which composes a DPF 32-bit integer from two 16-bit integers, and `L_Extract()`, which extracts two 16-bit integers from a DPF 32-bit integer, are no longer used.

The code for the intrinsic function `div_s()` is inlined in the `Div_32()` function to eliminate the extra cycles required to enter and exit the subroutine and implement a hardware loop. The compiler does not inline the code for `div_s()`. The inlined `div_s()` intrinsic consists of a loop that executes `div_iter()` intrinsic function sixteen times. Code Listing 2 shows the source code for `Div_32()` with an inlined `div_s()` function.

Because the compiler does not inline the code for intrinsic function `div_s()`, the code for `div_s()` was inlined in the `Div_32()` function to eliminate the extra cycles required to enter and exit the subroutine and to allow the loop that contains `div_s()` to be transformed in a hardware loop. The inlined `div_s()` intrinsic consists of a loop that executes the `div_iter()` intrinsic function sixteen times. **Code Listing 2** shows the source code for `Div_32()` with an inlined `div_s()` function.

Code Listing 2. `Div_32()` with Inlined `div_s()`

```
static Word32 Div_32(Word32 L_num, Word32 denom)
{
#pragma inline
        Word16 approx;
        Word32 L_32;

#ifdef _SCC_METROWERKS_
{
    int i;
    clearSRbit(1);
    L_32 = 0x3fff0000;
    for ( i = 0; i < 16; i++ ){
      L_32 = div_iter(L_32, extract_h(denom));
    }
    approx = extract_l(L_32);
}
#else
    approx = div_s((Word16)0x3fff, extract_h(denom));
#endif

    L_32 = Mpy_32_16(denom, approx);            /* result in Q30    */
    L_32 = L_sub((Word32)0x7ffffffeL, L_32);    /* result in Q30    */
    L_32 = Mpy_32_16(L_32, approx);             /* 1/L_denom in Q29 */
    /* L_num * (1/L_denom) */
    L_32 = Mpy_32(L_num, L_32);                 /* result in Q29    */
    return  L_shl(L_32, 2);                     /* From Q29 to Q31  */
}
```

The original implementation uses a vector to store new values of the reflection coefficients. However, these values are no longer used in the G.729A vocoder, so the storing phase of the reflection coefficients is no longer done. As a result, the input vector parameter for reflection coefficients and the phase of storing first two values of reflection coefficients in case of unstable filter were eliminated from implementation, thus reducing the occupied size and the number of execution cycles.

Referring to the pseudo-code in **Code Listing 1**, another optimization involves loop merging the summation part of (2) with the equation in (3), thus making better use of the registers (refer to the section of code marked 'E1' in Appendix A). In addition, the first calculation of *Alpha* (the `alp` variable) is removed from the initialization phase of the algorithm and inserted into the outer loop.

The use of DPF representation in the first inner loop, which copies the filter coefficients values used to recompute the backwards coefficients, results in half the number of cycles of the original implementation.

When `L_shl()` or `L_shr()` is used and the shift offset is variable, the compiler is forced to insert a function call that checks the sign and magnitude of the offset, providing the proper offset handling and saturation. However, when `alp and` the reflection coefficient `K` are calculated, the obtained offset is part of a normalization process, so no overflow can occur. Thus, in these cases, the 32-bit fractional shift left intrinsic `L_shl` can be replaced with the 40-bit intrinsic `X_shl()`, which does not perform checking and is thus translated into a single SC140 instruction, `asll()`.

3.2 Assembly Implementation

This section presents the optimization techniques used in the assembly implementation of the Levinson-Durbin algorithm to reduce code size and the number execution cycles. The assembly-optimized implementation is listed in Appendix B.

The outer loop of the algorithm (`FIRST_LOOP`) contains two inner loops (`LOOP_1` and `LOOP_2`) that are not executed in the first pass through the outer loop. This resulted in several jump and branch instructions in the assembly code generated from the C implementation. The assembly implementation of Levinson-Durbin algorithm optimizes the execution of loops in the optimized C implementation by reducing the number of jump and branch instructions, using registers more efficiently, minimizing the number of values stored on the stack (only the filter coefficients are stored), and reordering certain DPF operations. Using the assembly implementations of the DPF operations `Mpy_32()`, `Mpy_32_16()`, and `Div_32()` (Appendix C) enables them to be mixed with other independent operations, reducing the number of execution cycles.

3.2.1 Dedicated Registers

The first optimization was to store values that are used throughout the algorithm in dedicated registers at initialization so that they do not have to be restored after various operations that use them. They are used during DPF operations and for storing the *Alpha* (`alp`) and *Alpha Exponent* (`alp_exp`) variables.

3.2.2 Software Pipelining

A software pipelining technique reduces the number of execution sets in loops that use the `Mpy_32()` function. These loops include the outer loop (`FIRST_LOOP`), the second inner loop (`LOOP_2`), and the loop that stores the filter coefficients (`LFINAL`). Refer to the areas marked *E1*, *E2* and *E3* in Appendix B. In this technique,

- the first `Mpy_32()` operation is performed before the loop
- the succeeding `Mpy_32()` operations are performed at the end of the loop in parallel with storing the results of the previous `Mpy_32()` operation

This technique is illustrated in **Code Listing 3**.

Code Listing 3. Software Pipelining

```
        move.l (r0)+,d0move.l (r1)+,d1
        mpyus d0,d1,d2 mpysu d0,d1,d3
FALIGN
LOOPSTART3
Label
        dmacss d0,d1,d2asrw d3,d3
        and d11,d2     and d11,d3
        add d2,d3,d4
        move.l (r0)+,d0move.l (r1)+,d1
        [
         mpyus d0,d1,d2mpysu d0,d1,d3
         moves.f d4,(r2)+
        ]
LOOPEND3
```

In addition, the pipelining technique provides execution sets that can be used to fulfill the delay required after a `break` or `skipls` instruction.

3.2.2.1 The break Instruction

When the filter is determined to be unstable, the outer loop must be terminated and the previous filter coefficients maintained. This is done with a `break` instruction that jumps to a loop at the end of the program that copies the filter coefficients. (The loop counter is cleared as well.) The `break` instruction must be placed at least three execution sets before the last execution set in the loop. In the original code, there is no gap between `break` and the end of the loop; in the optimized code, the required additional execution sets are provided by the software pipeline technique, as illustrated in Code Example 4.

Code Listing 4. Using the `break` Instruction With Software Pipelining

```
FALIGN
LOOPSTART2
FIRST_LOOP
...
        [
         mpysu d0,d1,d3mpyus d0,d1,d14       ;Mpy_32(K,K)
         cmpgt d4,d5                         ;test unstable filter
        ]
        [
         dmacss d0,d1,d3asrw d14,d14
         inc d8                              ;increment loop counter
        ]
        [
        ift
         break L_LOOP                        ;unstable filter
        ]
        [
         and d11,d3    and d11,d14
         tfr d9,d0                           ;alp=d9=d0
        ]
         add d3,d14,d3                       ;t0=d3=Mpy_32(K,K)
         sub d3,d12,d1                       ;L_sub(0x7ffffffe,t0)
LOOPEND2
```

3.2.2.2 The skipls Instruction

The second inner loop (LOOP_2) cannot be performed during the first pass through the outer loop because the first computed reflection coefficient would be altered and the final results would be incorrect. The `skipls` instruction is used to bypass the second inner loop. This instruction tests the value of the loop counter and jumps past the loop if the loop counter is less than or equal to zero. The `skipls` instruction requires four execution cycles. The additional execution sets are provided by the software pipelining technique (refer to the area marked E6 in Appendix B). Although the first inner loop (LOOP_1) is executed the first time through the outer loop, it does not affect the final results because the values are rewritten in the next pass through the loop. The reason for executing LOOP_1 despite the fact that it is not needed is to avoid using the `skipls` instruction, which would have caused two cycles penalty per outer loop iteration.

In two instances, a loop is reduced to a single cycle by reading the initial value for the loop before the loop is entered. This technique, illustrated in Code Example 5, is applied to `LOOP_1` (refer to the section of code marked *E5* Appendix B) and `THIRD_LOOP`, which is performed near the end of the routine if the filter is unstable.

Code Listing 5. Single-Cycle Loop

```
        move.l (r0)+,d0
LOOPSTART3
LOOP_1
        move.l d0,(r1)+move.l (r0)+,d0
LOOPEND3
```

3.2.3 Relocating a Calculation

The calculation of the $(i-1)^{th}$ order filter coefficient was moved before the first inner loop because its value depends only on the reflection coefficient (`K`), which has already been computed at the beginning of the outer loop (refer area marked *E4* in Appendix B). The necessary instructions were inserted into the `alp` and `alp_exp` execution sets to reduce the of number execution sets in the code.

3.2.4 Combining Execution Sets

The `div_s` intrinsic executes immediately after the end of the second inner loop. (Refer to the area marked *E6* page 18 in Appendix B and the `Div_32()` function on page 21 in Appendix C.) This instruction cannot be placed earlier in the program because the `bmclr` instruction changes the SR register, and the modified SR required by the `div` instruction is available only after two additional cycles. Therefore, the execution sets of the `Mpy_32()` and `Mpy_32_16()` operations can be combined with the execution sets that compute the new reflection coefficient `K`. In addition, the first `div` operation executes before the loop for `div_s`, thus reducing the number of iterations of the loop. The result is more condensed code and fewer execution cycles. The code is shown in **Code Listing 6**.

Code Listing 6. Mixing Mpy_32() and Mpy_32_16() Execution Sets

```
  bmclr #<1,sr.l              doensh3 #<15
  move.l (r10),d7             move.l (r3)+,d6             ; An[1]=d7 R[i-1]=d6
  mpysu d6,d7,d1             mpyus d6,d7,d3             ; compute Mpy_32(R[i-1],An[1])
  dmacss d6,d7,d1             div d9,d15
LOOPSTART3
  div d9,d15
LOOPEND3
[
  and d11,d1                  tfr d9,d6                   ; alp=d9=d6
  aslw d15,d7                 asrw d3,d3                  ; approx=d7
]
```

```
        [
          and d11,d3                    add d1,d5,d1
          mpyus d6,d7,d15
        ]
        [
          dmacss d6,d7,d15              add d1,d3,d1                    ; d1=t0=Mpy_32(R[i-1],An[1]
          move.l #$7ffffffe,d12
```

3.2.5 Split Summation

A split summation technique is applied to the final loop of the algorithm, `LFINAL`, reducing the number of iterations almost in half, from (M–1) to M/2 iterations. Although the loop computes the M^{th} order filter coefficient, the value is computed again outside the loop without affecting the final results. Moving the calculation of the M^{th} order filter coefficient outside the loop would reduce the number of iterations to (M/2 – 1), but would require more execution cycles because the $(M–1)^{th}$ order coefficient would also have to be computed outside the loop. The `LFINAL` code is shown in **Code Listing 7**.

Code Listing 7. Split Summation

```
        [
         tfr d2,d14                         ;K=d2=d14
         mpysu d0,d1,d6mpyus d0,d1,d7       ;Mpy_32(K,An[M-1])
         move.l (r5)-,d2moves.f d15,(r11)+  ;d2=An[M-2]
                                            ;store lpc_status_oldA[0]
        ]
        [
         asrr #<4,d14
         mpysu d2,d3,d8mpyus d2,d3,d9       ;Mpy_32(K,An[M-2])
         move.l (r10)+,d4                   ;d4=An[1]
                                            ;d14=L_shr(K,4)
        ]
FALIGN
LOOPSTART3
LFINAL
        [
         dmacss d0,d1,d6dmacss d2,d3,d8
         asrw d7,d7    asrw d9,d9
         move.l (r10)+,d5                   ;d5=An[i]
        ]
        [
         and d11,d6    and d11,d7
         and d11,d8    and d11,d9
        ]
         add d6,d7,d6  add d8,d9,d8         ;d6=Mpy_32(K,An[M-i])
                                            ;d8=Mpy_32(K,An[M-i-1])
        [
         add d4,d6,d6  add d5,d8,d8         ;d6=L_add(An[i],d6)
         move.l (r5)-,d0                    ;d8=L_add(An[i+1],d8)
        ]
         asl d6,d6     asl d8,d8
        [
         rnd d6,d12    rnd d8,d13           ;temp=d12 temp=d13
         move.l (r5)-,d2 move.l (r10)+,d4   ;d2=An[M-i-2]
        ]                                   ;d4=An[i+2]
        [
```

```
        mpysu d0,d1,d6mpyus d0,d1,d7        ;Mpy_32(K,An[M-i-2])
        moves.f d12,(r1)+ moves.f d12,(r11)+;store A[i]
       ]
       [
        mpysu d2,d3,d8mpyus d2,d3,d9        ;Mpy_32(K,An[M-i-3])
        moves.f d13,(r1)+ moves.f d13,(r11)+;store A[i+1]
       ]
LOOPEND3
       [
        asl d14,d14                         ;d14=L_shl(d14,1)
        suba #<2,r1   suba #<2,r11          ;r1=&lpc->oldA[M]
                                            ;r11=&A[M]
       ]
       [
        rnd d14,d3                          ;d3=round(d14)
        jmpd L_END                          ;go to end
       ]
        moves.f d3,(r11) moves.f d3,(r1)    ;store A[M] value
```

4 Implementation Differences

This section presents the differences between various implementations of the Levinson-Durbin algorithm on different vocoders and the implementation from the ITU-T G.729A Recommendation. The differences are based on the ITU-T reference code.

4.1 ITU-T G.729

The Levinson-Durbin algorithm implementations of the ITU-T G.729 and ITU-T G.729A Recommendations are identical.

4.2 ITU-T G.723-1

The differences between the ITU-T G.723-1 and ITU-T G.729A implementations of the Levinson-Durbin algorithm are substantial:

- The operations are used only on 16-bit integers.
- For an unstable filter, only the prediction error is returned.
- The autocorrelation coefficients, which are used to compute the LPC filer coefficients, are obtained differently. (Hamming window and cosine function cycles in G.729A; Hamming window and a correction factor of 1025/1024 in G.723-1). Thus, many of the operations are different.

4.3 ETSI GSM EFR

The Levinson-Durbin algorithm implementation from ESTI GSM EFR is identical with the implementation from ITU-T G.729A Recommendation except that for an unstable filter, the first four reflection coefficients are saved for the next execution of the algorithm.

5 Conclusions

The optimized C implementation of the Levinson-Durbin algorithm provided several advantages for the assembly implementation. The loop merging technique was used to take advantage of the StarCore architecture to reduce the number of cycles in the `Mpy_32()` functions. The inlining technique reduced the number of cycles in the `Div_32()` function. These changes can be also applied to the C implementation.

The major changes applied in the assembly implementation included split summation and software pipelining.

Table 1 lists the results of executing both implementations. Notice that the assembly implementation consists of many more optimization techniques than the C implementation.

Table 1. Execution Results Comparison

Implementation	Number of Cycles	Size
Reference C Model (compiled with –O3)	2447	1432
Optimized C (compiled with –O3)	1438	1360
Assembly	808	828

6 References

[1] ITU-T Recommendation G.723.1—*Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 and 6.3 kbit/s*, March 1996.

[2] ITU-T Recommendation G.729—*Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP)*, March 1996.

[3] ITU-T Recommendation G.729/Annex A—*Reduced Complexity 8 kbit/s CS-ACELP Speech Codec*, November 1996.

[4] ETSI SMG2 ITU-T Recommendation GSM 6.60 EFR, January 1996.

[5] ITU-T Recommendation G.712—*Transmission Performance Characteristics of Pulse Code Modulation*, September 1992.

[6] *Digital Speech - Coding for Low Bit Rate Communications Systems*, A.M.Kondoz (John Wiley & Sons Ltd.: 1994).

[7] *A Practical Handbook of Speech Coders*, Randy Goldberg and Lance Riek. (CRC Press LLC: 2000).

Appendix A
C Implementation

```
;***************************************;
;* COPYRIGHT 2000 - 2005 Freescale Semiconductor, Inc.*;
;***************************************;
;*                                     *;
;* FILE NAME: levinson.c               *;
;* TARGET PROCESSOR: StarCore SC140    *;
;***************************************;

;************************************************************ *;
;* NOTE:                                                     *;
;* L_shl_nosat() and L_shr_nosat() are available starting with   *;
;* Metrowerks CodeWarrior for StarCore Release 1.5           *;
;************************************************************ *;

#if defined(_SC140_)
#define L_shl_nosat(a,b)        X_trunc(X_shl(X_extend(a),b))
#define L_shr_nosat(a,b)        X_trunc(X_shr(X_extend(a),b))
#define L_shl_nosat_r(a,b)      X_round(X_shl(X_extend(a),b))
#else
#define L_shl_nosat(a,b)        L_shl(a,b)
#define L_shr_nosat(a,b)        L_shr(a,b)
#define L_shl_nosat_r(a,b)      round(L_shl(a,b))
#endif

void Levinson(
  Word32 R[],                                   /* autocorrelations coefs. */
  Word16 A[],                                   /* LPC coefficients */
  G729A_LPC_STATUS_T * lpc_status               /* lpc status */
)
{
  Word16 i, j, temp,flag=0,approx;              /* temporary variables */
  Word32 K;                                     /* reflection coefficient */
  Word16 alp_exp = 0;
  Word32 alp = R[0];                            /* prediction gain */
  Word32 Ac[M+1];                               /* LPC coefs in DPF */
  Word32 An[M+1];                               /* LPC coefs. new in DPF */
  Word32 t0,t1,L_32;                            /* temporary variables */

                                                /* K=A[1]=-R[1]/R[0] */
  K = L_abs(R[1]);
  K = Div_32(K,R[0]);                           /* R[1]/R[0] in Q31 */
  if(R[1] > 0){
    K = L_negate(K);                            /* -R[1]/R[0] */
  }                                             /* K in DPF */

                                                /* Alpha = Alpha * (1-K^2) */
  t1 = Mpy_32(K,K);                             /* K*K in Q31 */
  t1 = L_sub((Word32)0x7ffffffeL,t1);           /* 1-K*K in Q31 */
  t1 = Mpy_32(alp,t1);                          /* Alpha in Q31 */

  for(i = 2; i <= M; i++){
    temp = neg_norm_l(t1);
    alp = L_shr_nosat(t1,temp);
    alp_exp = alp_exp - temp;                   /* Add normalization to alp_exp */
    approx = div_s((Word16)0x3fff, extract_h(alp));
    An[i-1] = L_shr(K, 4) & -2;                 /* convert K to Q27 */
                                                /* An[i] in Q27 */
    for( j = 0; j < i-1; j++){
      Ac[j] = An[j];
    }
    t0 = 0;                                     /* E1 */
    for( j = 1; j < i-1; j++){
      Word32 temp1,temp2;
      temp1 = Mpy_32(K,Ac[j]);
      temp2 = Mpy_32(R[j],An[i-j]);
      An[i-1-j] = L_add(Ac[i-1-j],temp1);
      t0 = L_add(t0,temp2);
    }

    t0 = L_add(t0,Mpy_32(R[i-1], An[1]));
    t0 = L_shl(t0,4);                           /* convert t0 to Q31 */
    t0 = L_add(t0,R[i]);                        /* add R[i] in Q31 */
    K = L_abs(t0);

                                                /*1/L_denom=approx*(2.0-L_denom*approx)*/
    L_32=Mpy_32_16(alp, approx);                /* result in Q30 */
    L_32 = L_sub((Word32)0x7ffffffeL,L_32);     /* result in Q30 */
```

```
    L_32 = Mpy_32_16(L_32,approx);                      /* 1/L_denom in Q29 */
                                                        /* L_num*(1/L_denom) */
    L_32 = Mpy_32(K, L_32);                             /* result in Q29 */
    K = L_shl(L_32, 2);                                 /* from Q29 to Q31 */

    if(t0 > 0){
      K = L_negate(K);                                  /* K =-t0/Alpha */
    }
    K = L_shl_nosat(K,alp_exp);

                                          /* Test for unstable filter. If unstable keep old A(z) */
                                                        /*  Alpha = Alpha*(1-K^2) */
    t1 = Mpy_32(K, K);                                  /* K*K in Q31 */
    t1 = L_sub((Word32)0x7ffffffeL, t1);                /* 1-K*K in Q31 */
    t1 = Mpy_32(alp, t1);                               /* Alpha in Q31 */

    if (abs_s(extract_h(K)) > 32750){
      flag=1;
      break;
    }
  }
  if(flag==1){
    for(j=0; j<=M; j++){
      A[j] = lpc_status->old_A[j];
    }
  }
  else{
    A[0] = 4096;
    for(i=1; i<M; i+=2){
      Word32 temp1,temp2,temp3,temp4;
      temp3=Mpy_32(K,An[M-i]);
      temp4=Mpy_32(K,An[M-i-1]);
      temp1 = L_add(An[i],temp3);
      temp2 = L_add(An[i+1],temp4);
      lpc_status->old_A[i] = A[i] = L_shl_nosat_r(temp1,1);
      lpc_status->old_A[i+1] = A[i+1] = L_shl_nosat_r(temp2,1);
  }
  lpc_status->old_A[M] = A[M] = L_shl_nosat_r(L_shr_nosat(K,4),1);
 }
}
```

Appendix B
Assembly Implementation

```
;****************************************;
;* COPYRIGHT 2000 - 2005 Freescale Semiconductor, Inc.*;
;****************************************;
;*                                      *;
;* FILE NAME: levinson.asm              *;
;* TARGET PROCESSOR: StarCore SC140     *;
;****************************************;

INCLUDE 'constants.inc'
     SET Lvsn_AnSize MP1<<2
     SET Lvsn_AcSize MP1<<2
     SET Lvsn_FrameSize (Lvsn_AnSize+Lvsn_AcSize+7)&$fffffff8
     SET Lvsn_AnOff Lvsn_FrameSize
     SET Lvsn_AcOff Lvsn_AnOff-Lvsn_AnSize

     SECTION .text LOCAL
     GLOBAL _Levinson
     ALIGN 16

_Levinson TYPE FUNC
       push d6                      push d7                      ; save data registers
       push r6                      push r7                      ; save address registers
     [
       adda #>Lvsn_FrameSize,sp,r3  move.2l (r0),d0:d1           ; compute stack frame size
     ]                                                           ; for Ac[] and An[] arrays
     [
       tfr d1,d6                    tfr d0,d2                    ; d1=d6=R[1] d0=d2=R[0]
       doensh3 #<16                 bmclr #<1,sr.l               ; set div_s loop
     ]
       move.w #$fffe,d11            move.f #$3fff,d15
     [
       clr d10                                                   ; alp_exp=d10=0
       adda #<4,r0
     ]
     LOOPSTART3
       div d2,d15
     LOOPEND3
     [
       abs d1                       aslw d15,d3                  ; d13=-2^15 approx=d3
       move.l #$7ffffffe,d12        tfra r3,sp
     ]
    [
       mpyus d2,d3,d4                                            ; compute Mpy_32_16(R[0],approx)
       adda #-(Lvsn_AnOff-4),sp,r10 adda #-Lvsn_FrameSize-28,sp,r11
                                                                 ; r10=&An[1] r11=&lpc_status
     ]
     [
       dmacss d2,d3,d4                                           ; d4=Mpy_32_16(R[0],approx)
       move.l (r11),r11                                          ; r11=&lpc_status_oldA[0]
     ]
       and d11,d4
       sub d4,d12,d2                                             ; L_32=d2=L_sub(7ffffffe,d4)
       mpyus d2,d3,d4                                            ; compute Mpy_32_16(L_32,approx)
       dmacss d2,d3,d4
       and d11,d4                   tfr d1,d5                    ; L_32=d4=Mpy_32_16(L_32,approx)
       mpysu d4,d5,d2               mpyus d4,d5,d3               ; compute Mpy_32(K,L_32)
     [
       dmacss d4,d5,d2              asrw  d3,d3
       adda #-(Lvsn_AnOff-4),sp,r4  adda #-(Lvsn_AcOff-4),sp,r5  ; r4=&An[1] r5=&Ac[1]
     ]
       and d11,d2                   and d11,d3
     [
       tfr d0,d9                    tstgt d6                     ; alp=d9=R[0]; test if(R[1]>0)
       add d2,d3,d2                                              ; d2 = Mpy_32(K,L_32)
       doen2 #<9                    dosetup2 FIRST_LOOP          ; set loop FOR(i=2; i <=M; i++)
     ]
       asll #<2,d2
       sat.l d2                                                  ; K=d2=L_shl(d2,2)
     [
      ift neg d2                                                 ; K=L_negate(K)
     ]
     [
       tfr d2,d0                    tfr d2,d1                    ; K=d0=d1=d2
       clr d8                                                    ; d8=0 (LOOP_1 & LOOP_2 counter)
     ]
```

```
        mpysu d0,d1,d3          mpyus d0,d1,d4          ; compute Mpy_32(K,K)          E1
        dmacss d0,d1,d3         asrw d4,d4
      [
        and d11,d3              and d11,d4
        tfr d9,d0                                       ; alp=d9=d0
      ]
        add d3,d4,d3                                    ; t0=d3=Mpy_32(K,K)             E1
        sub d3,d12,d1                                   ; t0=d1=L_sub(0x7ffffffeL,t0)
      FALIGN
      LOOPSTART2
FIRST_LOOP
      [
        mpysu d0,d1,d4          mpyus d0,d1,d5          ; compute Mpy_32(alp,t0)
        tfr d2,d3                                       ; K=d2=d3
        doensh3 d8                                      ; set LOOP_1
      ]
      [
        dmacss d0,d1,d4         asrr #<4,d3             ; d3 = L_shr(K,4)              E4
        asrw d5,d5
        adda #-(Lvsn_AnOff-4),sp,r6  adda #-(Lvsn_AcOff-4),sp,r7  ; r6=&An[1] r7=&Ac[1]
      ]
      [
        and d11,d3              and d11,d4              ; d3=An[i-1]=L_shr(K,4)&-2
        and d11,d5
        tfra r0,r3              move.f #$3fff,d15       ; r3=r0=&R[1]
      ]
      [
        add d4,d5,d4                                    ; d4=Mpy_32(alp,t0)
        move.l d3,(r4)          move.l (r6)+,d5         ; store An[i-1] d5=An[1]       E5
      ]
      LOOPSTART3
LOOP_1
        move.l d5,(r7)+         move.l (r6)+,d5         ; Ac[j]=An[j] for j=1..i-1
      LOOPEND3
      [
        tfr d4,d9               clb d4,d3               ; t0=d4=d9 d3=norm_l(t0)
        doen3 d8                dosetup3 LOOP_2         ; set LOOP_2
      ]
      [
        sub d3,d10,d10          asrr d3,d9              ; alp_exp=d10 alp=d9
        adda #-(Lvsn_AcOff-4),sp,r6  tfra r5,r7         ; r5=r6=r7=$Ac[1]
      ]
      [
        clr d5                                          ; t0=d5=0
        tfra r4,r2              skipls L02              ; r2=r4=&An[i-1]
                                                        ; go to L02 label if d8=0
      ]
      [
        clr d4                                          ; dummy for loop alignment
        move.l (r2)-,d0         move.l (r3),d1          ; d0=An[i-j] d1=R[j]
      ]
      [
        mpysu d0,d1,d7          mpyus d0,d1,d6          ; compute Mpy_32(R[j],An[i-j])  E2
        clr d5                  clr d4                  ; dummy for loop alignment
        move.l (r6)+,d3         move.l (r7)-,d12        ; d3=Ac[j] d12=Ac[i-1-j]
      ]
      [
        mpysu d2,d3,d14         mpyus d2,d3,d4          ; compute Mpy_32(K,Ac[j])
        clr d5                                          ; dummy for loop alignment
        adda #<4,r5                                     ; (r5)+
      ]
      FALIGN
      LOOPSTART3
LOOP_2
      [
        dmacss d2,d3,d14        dmacss d0,d1,d7
        asrw d4,d4              asrw d6,d6
        adda #<4,r3
      ]
      [
        and d11,d14             and d11,d4
        and d11,d7              and d11,d6
        move.l (r6)+,d3         move.l (r3),d1          ; d3=Ac[j] d1=R[j]             E2
      ]
        add d14,d4,d14          add d6,d7,d7            ; d14=Mpy_32(K,Ac[j])
                                                        ; d7=Mpy_32(R[j],An[i-j])
        add d12,d14,d0          add d5,d7,d5            ; temp=d0 t0=d5
      [
        mpysu d2,d3,d14         mpysu d0,d1,d7          ; compute Mpy_32(K,Ac[j])      E2
        mpyus d2,d3,d4          mpyus d0,d1,d6          ; and Mpy_32(R[j],An[i-j])
        move.l d0,(r2)-         move.l (r7)-,d12        ; store An[i-1-j]=temp=d0
                                                        ; get a new Ac[i-1-j]
      ]
```

```
        LOOPEND3
L02
          bmclr #<1,sr.l              doensh3 #<15                ;                              E6
          move.l (r10),d7             move.l (r3)+,d6             ; An[1]=d7 R[i-1]=d6
          mpysu d6,d7,d1              mpyus d6,d7,d3              ; compute Mpy_32(R[i-1],An[1])
          dmacss d6,d7,d1             div d9,d15
        LOOPSTART3
          div d9,d15
        LOOPEND3
        [
          and d11,d1                  tfr d9,d6                   ; alp=d9=d6
          aslw d15,d7                 asrw d3,d3                  ; approx=d7
        ]
        [
          and d11,d3                  add d1,d5,d1
          mpyus d6,d7,d15
        ]
        [
          dmacss d6,d7,d15            add d1,d3,d1                ; d1=t0=Mpy_32(R[i-1],An[1]
          move.l #$7ffffffe,d12
        ]
        [
          and d11,d15                 asll #<4,d1                 ; t0=L_shl(t0,4) without sat.l
          move.l (r3),d14                                         ; d14 = R[i]
        ]
        [
          add d1,d14,d2               add d1,d14,d6               ;t0=L_add(t0,R[i])=d2=d6
          sub d15,d12,d0              tfr d7,d1                   ; approx=d7=d1 L_32=d0
        ]
          mpyus d0,d1,d3                                          ; compute Mpy_32_16(L_32, approx)
          dmacss d0,d1,d3
          abs d2                      and d11,d3                  ; d3=L_32 K=d2=L_abs(t0)
          mpysu d2,d3,d0              mpyus d2,d3,d1              ; Mpy_32(K,L_32)
          dmacss d2,d3,d0             asrw d1,d1
        [
          and d11,d0                  and d11,d1
          tstgt d6                                                ; test if(t0 >0)
          move.w #$7fee,d4                                        ; d4=32750
        ]
          add d0,d1,d5                add d0,d1,d2
          asll #<2,d5                 asll #<2,d2
          sat.l d5                    sat.l d2
        [
         ift                                                      ; K=L_negate(K)=d2=d5
          neg d5
          neg d2
        ]
          asll d10,d5                 asll d10,d2                 ; K=K<<alp_exp=d2=d5
          asrw d5,d5
        [
          tfr d2,d1                   tfr d2,d0
          abs d5                                                  ; d5=abs_s(extract_h(K))
          adda #<4,r4
        ]
        [
          mpysu d0,d1,d3              mpyus d0,d1,d14             ; compute Mpy_32(K,K)           E1
          cmpgt d4,d5                                             ; test for unstable filter
        ]
        [
          dmacss d0,d1,d3             asrw d14,d14                ;                              E1
          inc d8                                                  ; increment loop counter
        ]
        [
         ift
          break L_LOOP                                            ; unstable filter go to L_LOOP
        ]
        [
          and d11,d3                  and d11,d14                                                E1
          tfr d9,d0                   clr d13                     ; alp=d9=d0
                                                                  ; clr- dummy for loop alignment
        ]
          add d3,d14,d3                                           ; t0=d3=Mpy_32(K,K)
          sub d3,d12,d1                                           ; t0=d1=L_sub(0x7ffffffe,t0)
        LOOPEND2
        [
          tfr d2,d3                   tfr d2,d1                   ; K=d2=d3=d1
          doen3 #<5                   dosetup3 LFINAL             ; set LFINAL loop
        ]
          move.f #$1000,d15           adda #>32,r10,r5            ; d15=4096 r10=&An[M-1]
          moves.f d15,(r1)+           move.l (r5)-,d0             ; store A[0] d0=An[M-1]
```

```
        [
          tfr d2,d14
          mpysu d0,d1,d6              mpyus d0,d1,d7              ; compute Mpy_32(K,An[M-1])      E3
          move.l (r5)-,d2             moves.f d15,(r11)+          ; d2=An[M-2]
                                                                  ; store lpc_status_oldA[0]
                                                                  ; K=d2=d14
        ]
        [
          clr d15                     asrr #<4,d14                ; clr-dummy for loop alignment
          mpysu d2,d3,d8              mpyus d2,d3,d9              ; compute Mpy_32(K,An[M-2])      E3
          move.l (r10)+,d4                                        ; d4=An[1]
                                                                  ; d14=L_shr(K,4)
        ]
        FALIGN
        LOOPSTART3
LFINAL
        [
          dmacss d0,d1,d6             dmacss d2,d3,d8
          asrw d7,d7                  asrw d9,d9
          move.l (r10)+,d5                                        ; d5=An[i]
        ]
        [
          and d11,d6                  and d11,d7
          and d11,d8                  and d11,d9
        ]
          add d6,d7,d6                add d8,d9,d8                ; d6=Mpy_32(K,An[M-i])
                                                                  ; d8=Mpy_32(K,An[M-i-1])
        [
          add d4,d6,d6                add d5,d8,d8                ; d6=L_add(An[i],d6)
          move.l (r5)-,d0                                         ; d8=L_add(An[i+1],d8)
        ]
          asl d6,d6                   asl d8,d8
        [
          rnd d6,d12                  rnd d8,d13                  ; temp=d12 temp=d13
          move.l (r5)-,d2             move.l (r10)+,d4            ; d2=An[M-i-2]
        ]                                                         ; d4=An[i+2]
        [
          mpysu d0,d1,d6              mpyus d0,d1,d7              ; compute Mpy_32(K,An[M-i-2])    E3
          moves.f d12,(r1)+           moves.f d12,(r11)+          ; store A[i]
        ]
        [
          mpysu d2,d3,d8              mpyus d2,d3,d9              ; compute Mpy_32(K,An[M-i-3])
          moves.f d13,(r1)+           moves.f d13,(r11)+          ; store A[i+1]
        ]
        LOOPEND3
        [
          asl d14,d14                                             ; d14=L_shl(d14,1)
          suba #<2,r1                 suba #<2,r11                ; r1=&lpc_status->oldA[M]
                                                                  ; r11=&A[M]
        ]
        [
          rnd d14,d3                                              ; d3=round(d14)
          jmpd L_END                                              ; end Levinson
        ]
          moves.f d3,(r11)            moves.f d3,(r1)             ; store A[M] value
L_LOOP
          doensh3 #<11                                            ; set unstable filter THIRD_LOOP
          nop                                                     ; dummy for loop alignment
          move.f (r11)+,d4                                        ; get lpc_status->oldA[0]
        LOOPSTART3
THIRD_LOOP
          move.f (r11)+,d4            moves.f d4,(r1)+            ; A[j]=lpc_status->oldA[j]
        LOOPEND3
L_END
          adda #-Lvsn_FrameSize,sp,r2                             ; set Levinson frame size
          tfra r2,sp                                              ; set SP to initial value
          pop r6            pop r7
          pop d6            pop d7
          rts                                                     ; end call
           ENDSEC
```

Appendix C
32-bit DPF Format and Operations

ITU-T G.729 and G.729A use a non-standard 32-bit double precision representation known as Double Precision Format (DPF). **Equation 14** is a mathematical representation of DPF equation representation.

$$L\_32 = hi << 16 + lo << \quad \textbf{Equation 14}$$

where `L_32` is a 32-bit signed integer and `hi` and `lo` are 16-bit signed integers. The range value of this representation is:

$$\$8000000 \leq L\_32 \leq \$7fffffe \quad \textbf{Equation 15}$$

That both the higher and lower parts are signed increases the speed of multiplication operations. This format was designed for 16-bit processors that do not support 32-bit operations. Even if the StarCore processors supported 32-bit operations, the DPF format must still be used to maintain bit-exactness of the original ITU-T implementation. The following special functions were defined to operate with this format:

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

The following sections describe the assembly implementations of these functions.

C.1 Mpy_32()

The `Mpy_32()` function multiplies two 32-bit numbers. In Code Example 8, `d0` and `d1` contain 32-bit DPF values, `d4` contains –2 ($fffffffe) and the result is stored in `d2`.

Code Listing 8. Mpy_32()

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

C.2 Mpy_32_16()

The `Mpy_32_16()` function multiplies 32-bit number and a 16-bit number. In Code Example 9, `d0` contains a 32-bit DPF value, `d1.h` contains the 16-bit integer, `d3` contains –2 ($fffffffe), and the result is stored in `d2`.

Code Listing 9. Mpy_32_16()

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

C.3 Div_32()

The `Div_32()` function divides two 32-bit numbers. In Code Example 10 `d1` and `d2` contain 32-bit DPF values, `d4` contains $3fff0000, `d5` contains $7FFFFFFE and `d6` contains –2 ($FFFFFFFE). The result is stored in `d3`. This implementation uses the `Mpy_32()` and `Mpy_32_16()` functions. The speed of this function is increased by inlining the `div_s` intrinsic function.

Code Listing 10. Div_32()

```
        doensh3 #<16
        bmclr #<1,sr.l
        nop
FALIGN
LOOPSTART3
        div d2,d4
LOOPEND3
        aslw d4,d3
                                                        ;d0 <- Mpy_32_16(d2,d3)
        mpyus d2,d3,d0
        dmacss d2,d3,d0
        and d6,d0
        sub d0,d5,d2
                                                        ;d0 <- Mpy_32_16(d2,d3);
        mpyus d2,d3,d0
        dmacss d2,d3,d0
        and d6,d0
                                                        ;d3 -< Mpy_32(d0,d1);
        mpyus d0,d1,d2          mpysu d0,d1,d3
        dmacss d0,d1,d3         asrw d2,d2
        and d6,d3               and d6,d2
        add d2,d3,d3
        asll #<2,d3
        sat.l d3
```

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

Document Order No.: AN2197
Rev. 1
1/2005

freescale™
semiconductor