

# Normalized Least Mean Square for a Smart Antenna System

by Kazuhiko Enosawa/Dai Haruki/Chris Thron

The smart antenna (SA), also known as the adaptive array antenna (AAA) is an important technique for increasing wideband code division multiple access (WCDMA) system capacity [1]. In WCDMA systems, multiple access interference (MAI) is the major cause of degradation of the uplink performance, since all users communicate simultaneously in the same frequency band. AAA reduces MAI by creating receiver beam patterns that “null out” strong interference sources, thus significantly enhancing system capacity.

AAA creates these interference-suppressing beam patterns through adaptive beam forming. For this purpose, the pilot symbols are used as a reference signal to generate an error signal, and uplink antenna weights are updated based on the minimum mean squared error (MMSE) criterion. In this context, there are three major MMSE-based algorithms: least mean square (LMS), normalized least mean square (NLMS), and recursive least square (RLS). NLMS is a variant of LMS that requires additional computation but offers superior performance, particularly in fading environments. The rate of convergence for RLS is typically an order of magnitude higher than LMS and NLMS. However, RLS is far more computationally complex. NLMS is thus an intermediate solution in terms of both performance and computational complexity.

## Contents

1	Normalized Least Mean Square Algorithm . . . . .	2
2	SC3400 DSP Implementation . . . . .	3
	2.1 Conversion of Floating-Point Code to Fixed-Point Code . . . . .	3
	2.2 Optimization Techniques in the NLMS Implementation . . . . .	4
	2.3 NLMS Kernel Function Implementation . . . . .	8
3	Results . . . . .	15
	3.1 Test Vectors . . . . .	15
	3.2 Performance . . . . .	16
4	Conclusion . . . . .	21
5	References . . . . .	21
6	Document Revision History . . . . .	22

# 1 Normalized Least Mean Square Algorithm

The fundamental equations for NLMS are as follows;

$$e(n) = d(n) - w^H(n) \cdot r(n) \tag{Eqn. 1}$$

$$w(n+1) = w(n) + \frac{\mu \cdot e(n)^* \cdot r(n)}{\|r(n)\|^2} \tag{Eqn. 2}$$

where

$e(n)$ : complex error

$d(n)$ : target output

$w(n)$ :  $J \times 1$  complex array weight vector

$r(n)$ :  $J \times 1$  complex array input vector

$J$ : number of antennas in the adaptive array antenna

$n$ : time index (1, ..., N)

$N$ : maximum number of iteration

$\mu$ : convergence parameter (or step size of weight vector update)

$\|r(n)\|^2 = r^H(n) \times r(n)$ : normalizing factor

(H denotes the complex conjugate and the transposed vector)

The constant  $\mu$  is the convergence parameter that determines degree of weight updates. As shown in [Equation 1](#) and [Equation 2](#), the MMSE algorithm discriminates the desired signal from interference signals by using the waveform difference from a reference signal. The direction of arrival (DOA) information for the desired signal is not required.

[Figure 1](#) shows the AAA system in the WCDMA uplink described in this application note. The system consists of the following components:

- beam estimator block
- conventional WCDMA chip-rate processing block that performs phase, amplitude, frequency, timing estimation and maximal ratio combining
- conventional WCDMA symbol-rate processing block that handles rate matching, de-interleaving, and decoding.

The beam forming consists of an MMSE weight adjustor (in this application note, NLMS is applied) and a beam estimator that computes the target output as the reference signal for the weight adjustor.

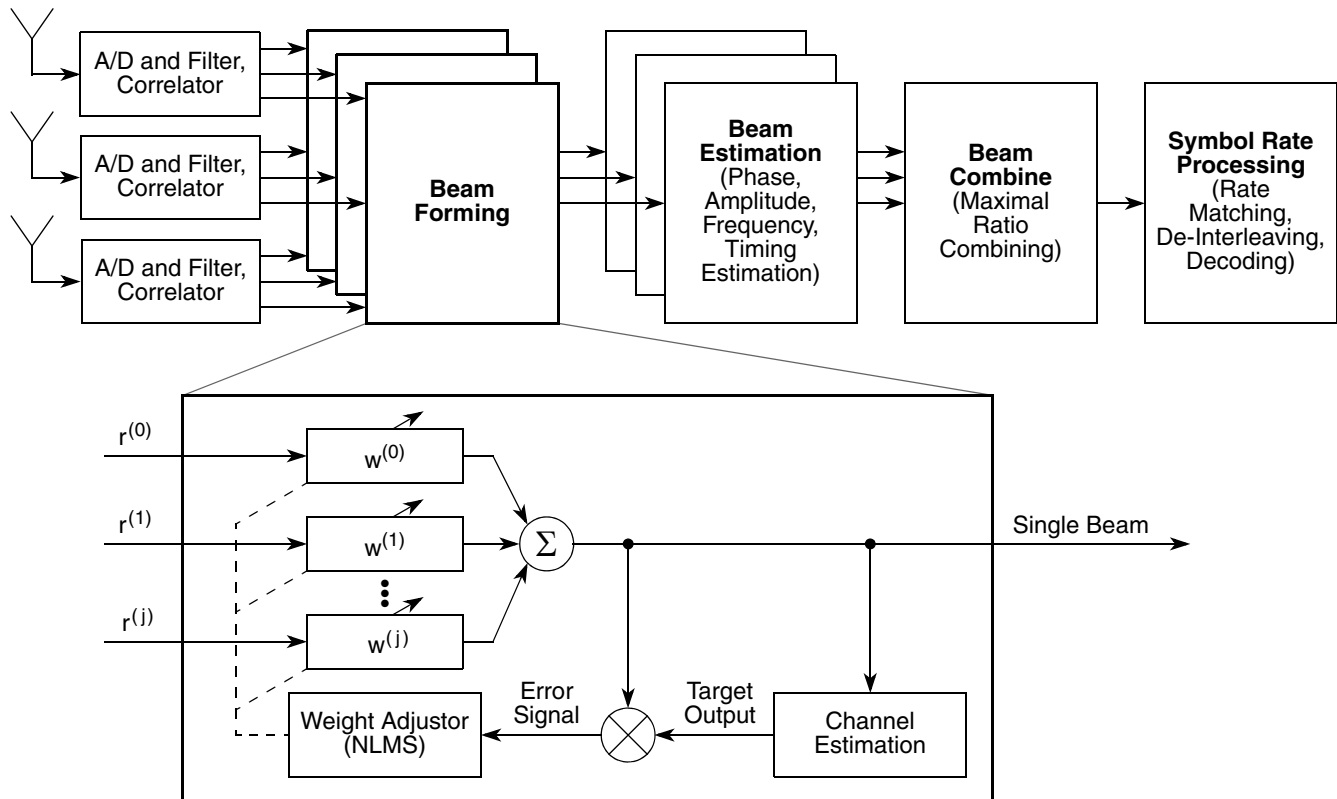


Figure 1. Adaptive Array Antenna system in WCDMA uplink

## 2 SC3400 DSP Implementation

For the SC3400 DSP core, the initial weights were chosen as  $w(1) = 1.0$  and  $w(1) = 0.0$  for  $j > 1$ , corresponding to an omni-directional initial beam. The initial condition for the average signal power is obtained by averaging the DPCCH pilot symbols that belong to two successive slots. The NLMS in this implementation is summarized as follows;

1. Initialize the weight vector.
2. Repeat steps a–b for each iteration:
  - a) Error calculation, [Equation 1](#). This routine calculates the complex error signal. The error signal is stored to calculate the complex weight-combined input vector for one slot. The beam estimator calculates the reference signal. The reference signal is given as a parameter.
  - b) Weight calculation, [Equation 2](#)). This routine updates the complex array weight vector for one slot (at least 8 DPCCH pilot symbols). The complex weight is stored in an array. Note that the buffer length of the complex array weight vector is dependent on the system. In this calculation, all values for the iteration are stored.

### 2.1 Conversion of Floating-Point Code to Fixed-Point Code

The NLMS algorithm was first coded in the MathWorks MATLAB® (64-bit floating-point representation) and then tested using test vectors generated with the channel model described in [Section 3.1](#), “Test Vectors.”

The test vectors were scaled down to the fixed-point range  $[-1, 1)$ . Based on the MATLAB code, fixed-point code was developed in the Freescale CodeWarrior™ C environment. The fixed-point code implementation was then tested against the MATLAB code to ensure the correct operations for NLMS. To constrain the variables, especially, the normalizing factor  $1 / \|r(n)\|^2$ , to the fixed-point range  $[-1, 1)$ , the variables were scaled to appropriate ranges as described in [Section 2.3.1, “Overflow Handling Method.”](#)

Built-in intrinsic functions in the CodeWarrior C compiler were used for the floating-point to fixed-point conversion (16-bit accuracy). These built-in intrinsic functions are optimized to simulate generic arithmetic operation on the SC3400 architecture. In addition, the optimization techniques described in the next section were used to improve performance.

## 2.2 Optimization Techniques in the NLMS Implementation

The C code optimization techniques improved performance by taking advantage of the special StarCore parallel computation capability. The SC3400 architecture has four parallel arithmetic logic units (ALUs), each performing most arithmetic operations, and two address arithmetic units (AAUs). To maximize computational performance, the four ALUs and two AAUs should operate simultaneously as much as possible.

Code optimization also accounted for the memory structure of the device. The SC3400 architecture provides a total 16 data registers D[0–15] and 16 address registers R[0–15]. If the number of active variables is greater than the number of available registers, the CodeWarrior C compiler puts the extra registers on stack memory. Therefore, to avoid performance degradation, the number of active variables was limited. The following subsections present the optimization techniques used to increase the speed of NLMS.

### 2.2.1 General Optimizations

General optimizations include complex representation, division substitution, loop unrolling, split summation, and memory alignment.

#### 2.2.1.1 Complex Representation

In the mathematical specification of NLMS, variables such as weights, target outputs and error signals are represented as complex numbers. The use of complex representation leads to simpler indexing in matrix calculations. The real and imaginary parts correspond to the In-Phase component and Quadrature component, as shown in the following code fragment;

```
// Short Complex
typedef struct tagComplex
{
    short siI;
    short siQ;
}t_COMPLEX, *pt_COMPLEX;

// Long Complex
typedef struct tagLongComplex
{
    long siI;
    long siQ;
}t_LCOMPLEX, *pt_LCOMPLEX;
```

The CodeWarrior C compiler can generate a multi-register move for each complex operation if the code and indexing are simple.

### 2.2.1.2 Division Substitution

The division substitution technique can be applied to algorithms with repeated divisions with the same denominator. This situation occurs in NLMS when the weight update vector is normalized, for each vector component is divided by the same number (see Equation 3). Computationally, division is far more expensive than multiplication, so it is advantageous to replace division with multiplication whenever possible. Division substitution was applied in the calculation of the term  $\{\mu \times e(n)^*\} / \|r(n)\|^2$ . To do this, it was necessary to ensure that the following conditions were satisfied;

- The magnitude of  $\beta = \|r(n)\|^2$  lies in the range (0.5, 1.0).
- The components of  $\mu \times e(n)^*$  all lie in the range  $[-\beta, \beta]$ .

Under these conditions, the division result always lies in the fixed-point range  $(-1.0, 1.0)$ . In actual implementation, the term was computed as follows;

**Eqn. 3**

$$\{\mu \times e(n)^*\} / \|r(n)\|^2 = \lfloor \{\mu \times e(n)^*\} \times \{0.5 / \|r(n)\|^2\} \rfloor \times 2$$

The intermediate result  $\{0.5 / \|r(n)\|^2\}$  also lies in the fixed-point range. The final multiplication by 2 is introduced to compensate for the factor of 0.5, which was introduced to put the normalizing factor within the fixed-point range.

### 2.2.1.3 Loop Unrolling

Loop unrolling involves explicitly duplicating loop operations in the code to enable the parallel use of the four ALUs. Loop unrolling is effectively used in conjunction with memory alignment (see Section 2.2.1.5, “Memory Alignment”) because parallel operation requires that the input data first be moved into the data registers D[0–15]. This move can be accomplished in a single loop step through multi-word moves if the memory is properly aligned. In the following example, loop unrolling and memory alignment ensure that four operations and address updates of the vector are executed per loop step.

```
// Overflow handling : right shift for stInput[] vector of NANTS.
pt_COMPLEX  pstR; // Pointer for Input Vector Array
#pragma align *pstR 8

short sii;

for( sii = 0; sii < 8; sii += 2 )
{
    stInput[ sii      ].siI = shr( pstR [ sii      ].siI, 2 );
    stInput[ sii      ].siQ = shr( pstR [ sii      ].siQ, 2 );
    stInput[ sii + 1 ].siI = shr( pstR [ sii + 1 ].siI, 2 );
    stInput[ sii + 1 ].siQ = shr( pstR [ sii + 1 ].siQ, 2 );
}
```

## 2.2.1.4 Split Summation

Complex vector inner products are computed frequently in the NLMS calculation. Split summation is used to parallelize the operation so that four MAC instructions execute in each cycle as shown in the following code fragment:

```
// Normalizing factor calculation for pstR vector of NANTS.

pt_COMPLEX  pstR; // Pointer for Input Vector Array
#pragma align *pstR 8

short sii0;
long liRtemp0i, liRtemp0q, liRtempli, liRtemplq;
long liRR;

//--- Normalization Factor R^H * R
liRtemp0i = liRtemp0q = liRtempli = liRtemplq = 0;
for( sii0 = 0; sii0 < NANTS; sii0 += 2)
{
    siRtemp0i = shr( pstR[sii0 ].siI, 2 );
    siRtemp0q = shr( pstR[sii0 ].siQ, 2 );
    siRtempli = shr( pstR[sii0+1].siI, 2 );
    siRtemplq = shr( pstR[sii0+1].siQ, 2 );
    liRtemp0i = L_mac( liRtemp0i, siRtemp0i, siRtemp0i);
    liRtemp0q = L_mac( liRtemp0q, siRtemp0q, siRtemp0q);
    liRtempli = L_mac( liRtempli, siRtempli, siRtempli);
    liRtemplq = L_mac( liRtemplq, siRtemplq, siRtemplq);
}
liRRtemp0i = L_add(liRtemp0i, liRtempli);
liRRtemp0q = L_add(liRtemp0q, liRtemplq);
liRR      = L_add(liRRtemp0i, liRRtemp0q); // (R^H * R)
```

Split summation makes use of the fact that addition is associative and commutative. That is, changing the order of the operations does not change the result. In the preceding example, four different temporary destination variables are assigned to the four in-loop operations. Following the loop, the four temporary variables are added together.

Like loop unrolling, memory alignment enables the use of multiple-byte bus access to move the input data into the data registers D[0–15] before the arithmetic operation. Although split summation speeds up the inner product computation by a factor of about four, the code size also increases in like proportion. Therefore, split summation should be used only if speed optimization is more important than size optimization.

## 2.2.1.5 Memory Alignment

In the SC3400 architecture, each data bus is 64 bits wide. Therefore, in the internal memory area, four short words ( $4 \times 16$ -bits) or two long words ( $2 \times 32$ -bits) can be fetched at once. However, multiple-byte bus access requires proper alignment of the memory. Multi-word moves such as `move.4f`, which moves fraction words in pairs, requires specification of the exact alignment for the objects. For example, 8-byte ( $4 \times$  short words) accessing must be aligned to the 8-byte addresses.

To enable the CodeWarrior C compiler to use multi-word moves, the pragma `#pragma align` can be used to provide the compiler with specific alignment information about pointers to objects. To inform the CodeWarrior C compiler that the address of an array is aligned as required for `move.4f`, we specify

#pragma align, followed by the pointer to the object and 8 for 8-byte alignment, as shown in following example.

```

        pt_COMPLEX  pstW;    // [IN/OUT] Pointer of Weight Vector
        pt_COMPLEX  pstE;    // [OUT] Pointer for Error Vector
        pt_COMPLEX  pstR;    // [IN] Pointer for Input Vector Array
        pt_COMPLEX  pstD;    // [IN] Pointer for Target (real part)
#pragma align *pstW 8
#pragma align *pstE 8
#pragma align *pstR 8
#pragma align *pstD 8
    
```

## 2.2.2 Efficient Fixed-Point Mathematical Algorithms

The built-in intrinsic function `div_s` in `prototype.h` is inefficient and may require 30 cycles to perform fractional integer division (depending on the compiler version) [2]. To calculate fractional integer divisions efficiently, we use the Newton-Raphson algorithm. In our implementation, each iteration of the Newton-Raphson formula for division “ $U = 1/V$ ” can be expressed as follows:

**Eqn. 4**

$$U_{i+1} = U_i + 1 - (V \cdot U_i) \cdot U_i$$

To implement the Newton-Raphson algorithm for division in the fixed-point range, the division is approximately written as follows:

**Eqn. 5**

$$\begin{aligned}
 \frac{1}{2} \times \frac{1}{V} &= \frac{U_0}{2} + (1 - V \cdot U_0) \cdot \frac{U_0}{2} + (1 - V \cdot U_0)^2 \cdot \frac{U_0}{2} + (1 - V \cdot U_0)^3 \cdot \frac{U_0}{2} + \dots + (1 - V \cdot U_0)^{15} \cdot \frac{U_0}{2} \\
 &= a + a \cdot (1 - V \cdot U_0)^2 \\
 &\quad + \{a + a \cdot (1 - V \cdot U_0)^2\} \cdot (1 - V \cdot U_0)^4 \\
 &\quad + \{a + a \cdot (1 - V \cdot U_0)^2 + \{a + a \cdot (1 - V \cdot U_0)^2\} \cdot (1 - V \cdot U_0)^4\} \cdot (1 - V \cdot U_0)^8
 \end{aligned}$$

Where:

**Eqn. 6**

$$\begin{aligned}
 a &= \frac{U_0}{2} + (1 - V \cdot U_0) \cdot \frac{U_0}{2} \\
 &= \frac{U_0}{2} + \left\{ \left( \frac{1}{2} - V \cdot \frac{U_0}{2} \right) + \frac{1}{2} - \left( V \cdot \frac{U_0}{2} \right) \right\} \cdot \frac{U_0}{2}
 \end{aligned}$$

Equation 6 is valid as long as  $V$  lies in fixed-point range  $[-1, 1)$  as shown in Figure 2. The initial value “ $U_0$ ” was evaluated as “ $U_0/2 = 0x55F3$ ” in the fixed-point representation. The fifteenth-order expansion was used to achieve a suitable balance between speed and accuracy. In our implementation, each division calculation requires only 13 cycles, as compared to about 30 for the built-in intrinsic function.

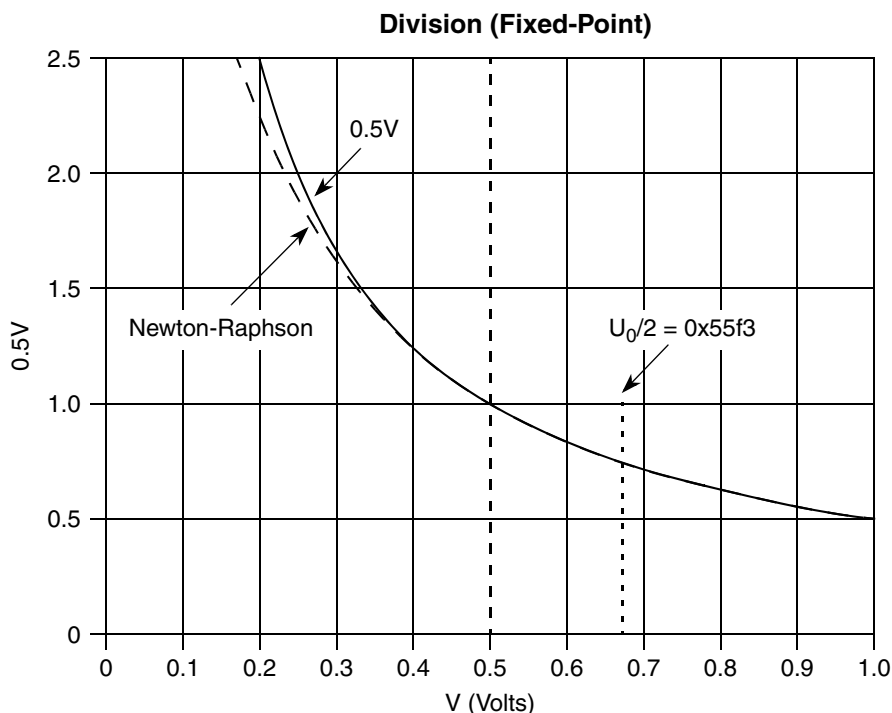


Figure 2. Newton-Raphson Method for Division in Fixed-Point Range

### 2.3 NLMS Kernel Function Implementation

The concrete implementation of the NLMS kernel function consists of the error signal calculation Equation 1 and the normalized weight estimate update Equation 2. To evaluate the efficiency of the optimization, the number of mathematical operations required for the implementation must be estimated based on the knowledge of the SC3400 instruction set. First, we compute the floating-point operations count and then we use these results to find the corresponding fixed-point operations count.

The floating-point operation counts for the NLMS kernel function are given in Table 1 (note that divide is counted as a single mathematical operation). The table makes use of the fact that the input data, weights, target outputs and error signals are represented as complex numbers while the convergence parameter  $\mu$  is a real number. For example, Step 3 (complex J vector inner product) listed in Table 1 represents the calculation of the weight-combined input  $w^H(n).r(n)$ , and is calculated as follows.

Eqn. 7

$$A^H \times B = \sum_{j=1}^J \{(A_j^r \cdot B_j^r + A_j^i \cdot B_j^i) + (A_j^r \cdot B_j^i - A_j^i \cdot B_j^r)j\}$$

$$A^H = \begin{bmatrix} A_1^r - A_1^i j \\ A_j^r - A_j^i j \end{bmatrix}$$

$$B = (B_1^r + B_1^i j, \dots, B_j^r + B_j^i j)$$



Therefore,  $4 * J$  #MACC (multiply-accumulate) operations are needed to calculate the complex J vector inner product. The steps in [Table 1](#) are given in order of execution in our implementation. Note that steps 1 and 2 are related to the calculation of the normalizing factor, which distinguishes NLMS from LMS.

**Table 1. Mathematical Operation Counts**

Step	Equation	Operation description	#MACC	#Multi	#Ad	#Div
1	Equation 2	Complex J vector inner product	$2 * J$			
2	Equation 2	Complex scalar inverse	2	2		1
3	Equation 1	Complex J vector inner product	$4 * J$			
4	Equation 1	Complex add			2	
5	Equation 2	Complex / real scalar multiply		2		
6	Equation 2	Complex scalar / complex J vector multiply & add	$4 * J$			

#MACC : Multiply-Accumulate operation

#Multi : Multiply operation

#Add : Add operation

#Div : Division operation

[Table 1](#) shows the floating-point implementation. Steps 1 and 2 are related to the calculation of normalizing factor of NLMS. Our theoretical estimate for the number of fixed-point operations counts per NLMS iteration is given in [Table 2](#), which makes use of the operations counts in [Table 1](#). Our implementation requires conversion of floating-point representation to fixed-point representation. The division operation indicated in step 2 of [Table 2](#) was performed using the Newton-Raphson algorithm (fifteenth-order expansion), which requires 3 multiplies, 1 add, and 5 MACCs per division. In addition, overflow handling is required for the fixed-point implementation so that  $2 * J$  shift operations are needed for both real and imaginary parts of the input data. Finally, 1 shift operation is introduced to compensate for the shift of input data.

By summing the operations counts in [Table 2](#), we find that  $12 * J + 18$  total operations (MACC, multiply, add, or shift) are required per NLMS iteration in fixed-point implementation, which corresponds to 114 operations per iteration for 8 antennas.

**Table 2. Mathematical Operations Counts for NLMS Function**

Step	Equation	Operation description	#MACC	#Multi	#Add	#Shift
1	Equation 2	Overflow handling				$2 * J + 1$
		Complex J Vector inner product	$2 * J$	3	1	
2	Equation 2	Real inverse (Newton-Raphson)	5	2		
		Complex Scalar product	2			
3	Equation 1	Complex J vector inner product	$4 * J$			
4	Equation 1	Complex add			2	
5	Equation 2	Complex/real scalar multiply		2		

**Table 2. Mathematical Operations Counts for NLMS Function (continued)**

Step	Equation	Operation description	#MACC	#Multi	#Add	#Shift
6	Equation 2	Complex scalar / complex J vector multiply & add	4*J			
#MACC: Multiply-Accumulate operation #Multi: Multiply operation #Add: Add operation #Shift: Shift operation						

### 2.3.1 Overflow Handling Method

In the normalizing factor calculation, the repeated multiply-accumulate operations may cause overflow. Therefore, a method for handling overflow is specified, as follows:

1. Each input data  $r(n)$  is constrained to be less than  $siShiftInput = 2^{-2} = 0.25$  by appropriate downscaling. The value of 0.25 corresponds to the choice of 8 antennas.
2. The squared length of input data for all antenna  $\beta = \|r(n)\|^2$  is calculated. Note that both real and imaginary parts of the input data are constrained to be less than 0.25. Conditions 1) guarantees that the result is less than 1.0.
3. The value of  $siShiftAmount$  needed to normalize the 32-bit  $\beta$  is calculated by using “norm\_l( $\beta$ )”. This built-in intrinsic function produces the number of left shifts needed to normalize the 32-bit variable so that  $\beta$  lies in the range (0.5, 1.0).
4. The intermediate result  $\gamma = 0.5/\beta = 0.5/\|r(n)\|^2$  is calculated and lies in the range (0.5, 1.0).
5. The intermediate result  $\gamma$  is left shifted by  $(siShiftAmount - (siShiftInput + 1))$  to compensate for all the scalings introduced to put intermediate results within fixed-point range. Note that “+1” is introduced to compensate for the “0.5” in the intermediate result  $\gamma$ .

### 2.3.2 NLMS Kernel Function Code

```

/* =====
$Algorithm: Normalized Least Mean Square (NLMS)
  Error signal: e = d - W'*R
  Weight update: W(i+1) = W(i) + myu * e'*R / (R^H * R)

  R   : Input Data
  d   : Target
  W   : Weight Vector for Adaptive Processing
  e   : Error between the Target and the Corrected Input Data
  myu : Convergence Parameter
  i   : Time Index

===== */

void NLMS(
    pt_NLMSARG pstArg
)
{
    #pragma align *pstArg 8

```

```

        pt_COMPLEX  pstW;    // [IN/OUT] Pointer of Weight Vector
        pt_COMPLEX  pstE;    // [   OUT] Pointer for Error Vector
        pt_COMPLEX  pstR;    // [IN   ] Pointer for Input Vector Array
        pt_COMPLEX  pstD;    // [IN   ] Pointer for Target (real part)
#pragma align *pstW 8
#pragma align *pstE 8
#pragma align *pstR 8
#pragma align *pstD 8

        short siIter;        // [IN   ] Iteration
        short siMyu;         // [IN   ] Convergence parameter
        short siNumOfAntennas = NANTS;

        short sii, siil;     // For Loop counter
        short sij;          // For Loop counter

        // Computation buffer for Weight Vector
        long  liWtemp0i;
        long  liWtemp0q;
        long  liWtemp1i;
        long  liWtemp1q;
        long  liWtemp2i;
        long  liWtemp2q;
        long  liWtemp3i;
        long  liWtemp3q;

        // Computation buffer for Error Vector
        short siEtemp0i;
        short siEtemp0q;
        long  liEtemp0i;
        long  liEtemp0q;

        // Computation buffer for Input Vector Array
        short siRtemp0i;
        short siRtemp0q;
        short siRtemp1i;
        short siRtemp1q;
        short siRtemp2i;
        short siRtemp2q;
        short siRtemp3i;
        short siRtemp3q;

        short sii0;
        short siShiftAmount;
        const short siShiftInput = 2; // For overflow measure

        // Computation buffer for R^H * R
        long  liRtemp0i;
        long  liRtemp0q;
        long  liRtemp1i;
        long  liRtemp1q;
        long  liRRtemp1;
        long  liRRtempq;

        long  liRR;           // R^H * R
        long  liRRtemp0;
        long  liCoef1;
        long  liCoef2;
        long  liDivTemp1;
    
```

```

// Pointer setting and parameters input
pstW  = pstArg->pstW ;
pstE  = pstArg->pstE ;
pstR  = pstArg->pstR ;
pstD  = pstArg->pstD ;
siMyu = pstArg->siMyu ;
siIter = pstArg->siIter;

//-----
// Step 1: Overflow handling
//      Complex J vector inner product
//-----

//--- Loop for the Iteration
for( sij = 0; sij < siIter; sij++, pstR += siNumOfAntennas,
    pstW += siNumOfAntennas )
{
    //--- Normalization Factor R^H * R
    liRtemp0i = liRtemp0q = liRtempli = liRtemplq = 0;
    for( sii0 = 0; sii0 < NANTS; sii0 += 2)
    {
        siRtemp0i = shr( pstR[sii0 ].siI, siShiftInput ); // 1/4
        siRtemp0q = shr( pstR[sii0 ].siQ, siShiftInput ); // 1/4
        siRtempli = shr( pstR[sii0+1].siI, siShiftInput ); // 1/4
        siRtemplq = shr( pstR[sii0+1].siQ, siShiftInput ); // 1/4
        liRtemp0i = L_mac( liRtemp0i, siRtemp0i, siRtemp0i);
        liRtemp0q = L_mac( liRtemp0q, siRtemp0q, siRtemp0q);
        liRtempli = L_mac( liRtempli, siRtempli, siRtempli);
        liRtemplq = L_mac( liRtemplq, siRtemplq, siRtemplq);
    }
    liRRtemp0i = L_add(liRtemp0i, liRtempli);
    liRRtemp0q = L_add(liRtemp0q, liRtemplq);
    liRR      = L_add(liRRtemp0i, liRRtemp0q); // (R^H * R)

//-----
// Step 2: Real inverse (Newton-Raphson)
//      Complex scalar product
//-----

//-- 1/|R|^2
siShiftAmount = norm_l(liRR);
liRRtemp0      = L_shl_nosat( liRR, siShiftAmount );
                // There is enough head room.

// Division : Newton-Raphson method
liCoef1 = L_msu( L_deposit_h(0x4000), 0x55f3, extract_h(liRRtemp0) );
                // 1/2 - V*U0/2
liCoef1 = L_add( liCoef1, liCoef1 );
                // (1/2 - V*U0/2) + (1/2 - V*U0/2)
liDivTemp1 = L_mult( extract_h(liCoef1), extract_h(liCoef1) );
                // {(1/2 - V*U0/2) + (1/2 - V*U0/2)}^2
liCoef2 = L_mac( L_deposit_h( 0x55f3 ), extract_h(liCoef1), 0x55f3 );
                // a = U0/2 + {(1/2 - V*U0/2) + (1/2 - V*U0/2)}*U0/2
liCoef2 = L_mac( liCoef2, extract_h(liDivTemp1), extract_h(liCoef2);
                // a + a*(1 - V*U0)^2
liCoef1 = L_mult( extract_h(liDivTemp1), extract_h(liDivTemp1) );
                // (1 - V*U0)^4

liCoef2 = L_mac( liCoef2, extract_h(liCoef1), extract_h(liCoef2) );
                // a + a*(1 - V*U0)^2 + {a + a*(1 - V*U0)^2}*(1 - V*U0)^4

```

```

liDivTemp1    = L_mult( extract_h(liCoef1), extract_h(liCoef1) );
                // (1 - V*U0)^8
liCoef2      = L_mac( liCoef2, extract_h(liDivTemp1), extract_h(liCoef2) );
                // a + a*(1 - V*U0)^2 + {a + a*(1 - V*U0)^2}*(1 - V*U0)^4
// + [a + a*(1 - V*U0)^2 + {a + a*(1 - V*U0)^2}*(1 - V*U0)^4]*(1 - V*U0)^8

// myu / |R|^2
liCoef2      = L_shl_nosat( L_mult( siMyu, extract_h( liCoef2 ) ),
                            siShiftAmount - (siShiftInput + 1) );

//-----
// Step 3: Complex 8 vector inner product
//-----

    //--- Target setting
    liWtemp0i = L_deposit_h(pstD->siI);
    liWtemp0q = L_deposit_h(pstD->siQ);
    pstD++;

    liWtempli = liWtemplq = 0;

    //--- Error signal: e = d - W'*R
    for( sii = 0; sii < siNumOfAntennas; sii += 2 )
    {
        // Register Scheduling
        // 4 for psiR
        // 4 for liWtemp0i-li
        // 4 for pstW
        // 4 for siRtemp0i-li
        //-- Preload
        siRtemp0i = pstR[sii ].siI;
        siRtemp0q = pstR[sii ].siQ;
        siRtempli = pstR[sii+1].siI;
        siRtemplq = pstR[sii+1].siQ;
        //-- Real Part
        liWtemp0i = L_msu( liWtemp0i, pstW[sii ].siI, siRtemp0i );
        liWtemp0i = L_msu( liWtemp0i, pstW[sii ].siQ, siRtemp0q );
        liWtempli = L_msu( liWtempli, pstW[sii+1].siI, siRtempli );
        liWtempli = L_msu( liWtempli, pstW[sii+1].siQ, siRtemplq );
        //-- Imaginary Part
        liWtemp0q = L_msu( liWtemp0q, pstW[sii ].siI, siRtemp0q );
        liWtemp0q = L_mac( liWtemp0q, pstW[sii ].siQ, siRtemp0i );
        liWtemplq = L_msu( liWtemplq, pstW[sii+1].siI, siRtemplq );
        liWtemplq = L_mac( liWtemplq, pstW[sii+1].siQ, siRtempli );
    }

//-----
// Step 4: Complex add
//-----

    //--- Error update
    liEtemp0i = L_add( liWtemp0i, liWtempli );
    liEtemp0q = L_add( liWtemp0q, liWtemplq );

//-----
// Step 5: Complex / real scalar multiply
//-----

    siEtemp0i = mult_r(extract_h(liEtemp0i), extract_h(liCoef2));
                // Real: myu * e' / (R^H * R)

```

```

        siEtemp0q = mult_r(extract_h(liEtemp0q), extract_h(liCoef2));
                        // Imag: myu * e' / (R^H * R)

//-----
// Step 6: Complex scalar / complex J vector multiply & add
//-----

//--- Weight vector: W(i+1) = W(i) + myu * e'*R / (R^H * R)
// Loop unrolled two times for the speed optimization
for( siil = 0; siil < siNumOfAntennas; siil += 4 )
{
    // Register Scheduling
    // 4 for psiR          (1st set)
    // 2 for siEtemp0i,q   (1st set)
    // 4 for siRtemp0i-lq (1st set)
    // 4 for psiR          (2nd set)
    // 2 for siEtemp0i,q   (2nd set)
    // 4 for siRtemp0i-lq (2nd set)

    // Input (1st set)
    siRtemp0i = pstR[siil ].siI;
    siRtemp0q = pstR[siil ].siQ;
    siRtemp1i = pstR[siil+1].siI;
    siRtemp1q = pstR[siil+1].siQ;

    // Complex multiplication (1st set)
    liWtemp0i = L_mac( L_deposit_h( pstW[siil ].siI ), siEtemp0i, siRtemp0i );
    liWtemp1i = L_mac( L_deposit_h( pstW[siil ].siQ ), siEtemp0i, siRtemp0q );
    liWtemp0q = L_mac( L_deposit_h( pstW[siil+1].siI ), siEtemp0i, siRtemp1i );
    liWtemp1q = L_mac( L_deposit_h( pstW[siil+1].siQ ), siEtemp0i, siRtemp1q );

    // Input (2nd set)
    siRtemp2i = pstR[siil+2].siI;
    siRtemp2q = pstR[siil+2].siQ;
    siRtemp3i = pstR[siil+3].siI;
    siRtemp3q = pstR[siil+3].siQ;

    // Complex multiplication and store (1st set)
    pstW[NANTS+siil ].siI = mac_r( liWtemp0i, siEtemp0q, siRtemp0q );
    pstW[NANTS+siil ].siQ = msu_r( liWtemp1i, siEtemp0q, siRtemp0i );
    pstW[NANTS+siil+1].siI = mac_r( liWtemp0q, siEtemp0q, siRtemp1q );
    pstW[NANTS+siil+1].siQ = msu_r( liWtemp1q, siEtemp0q, siRtemp1i );

    // Complex multiplication (2nd set)
    liWtemp2i = L_mac( L_deposit_h( pstW[siil+2].siI ), siEtemp0i, siRtemp2i );
    liWtemp2q = L_mac( L_deposit_h( pstW[siil+2].siQ ), siEtemp0i, siRtemp2q );
    liWtemp3i = L_mac( L_deposit_h( pstW[siil+3].siI ), siEtemp0i, siRtemp3i );
    liWtemp3q = L_mac( L_deposit_h( pstW[siil+3].siQ ), siEtemp0i, siRtemp3q );

    // Complex multiplication and store (2nd set)
    pstW[NANTS+siil+2].siI = mac_r( liWtemp2i, siEtemp0q, siRtemp2q );
    pstW[NANTS+siil+2].siQ = msu_r( liWtemp2q, siEtemp0q, siRtemp2i );
    pstW[NANTS+siil+3].siI = mac_r( liWtemp3i, siEtemp0q, siRtemp3q );
    pstW[NANTS+siil+3].siQ = msu_r( liWtemp3q, siEtemp0q, siRtemp3i );

}
} // End of iteration loop for( sij = 0

```

```

// Store last error
pstE->siI = extract_h(liEtemp0i);
pstE->siQ = extract_h(liEtemp0q);
} // End of NLMS

```

### 3 Results

The results report in this section covers test vectors and overall performance in terms of cycle count measurements and data ALU parallel usage.

#### 3.1 Test Vectors

Test vectors were generated by MathWorks MATLAB in floating-point representation and then scaled to fixed-point range  $[-1, 1)$  and quantized to 8 bits. The simulation scenario is illustrated in Figure 3. The simulated receiver antenna has an 8-element linear array with half-wavelength spacing between antennas. A single plane-wave source is located at angle  $\theta_s$ , and plane-wave interferers is located at angles  $\theta_i$ . Isotropic background noise is also included. The simulation scenario is static, that is, we do not consider fading effects. Because sources are assumed to be plane-wave sources, angle spread effects are also not included. The parameters in the simulation are listed in Table 3. The values are default in the simulation.

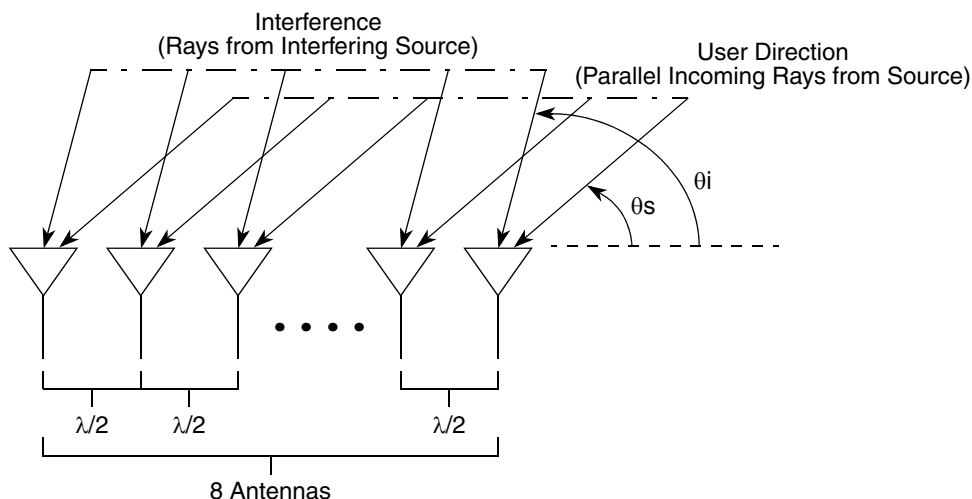


Figure 3. Simulation Scenario

Table 3. System Parameters in Simulation

Parameter	Value
Number of antennas	8
Number of users	1
Number of fingers	1
Number of iterations in simulation	100
Antenna spacing (in wavelength units)	0.5

**Table 3. System Parameters in Simulation (continued)**

Parameter	Value
Source signal incidence angle (radian)	$\pi/3$
Number of interferences	6
Interferer incident angles (radian)	$[0 \ 1*\pi/6 \ 1*\pi/2 \ 3*\pi/4 \ 5*\pi/6 \ \pi]$
Interference/source power per antenna [db]	$[0 \ 1 \ 2 \ 3 \ -1 \ -2]$
SNR (Signal to Noise Ratio) per antenna	-5

The input data was scaled within the fixed-point range  $[-1, 1)$  according to a crest factor relative to the standard deviation of the input. The formula used is

$$\text{scaling\_factor} = \text{sqrt}(\text{test\_vec\_pwr}) * \text{crest\_factor};$$

The `test_vec_pwr` variable was determined by finding the test vector average power and then dividing by  $2 \times n\_ants$ . The 2 is for in-phase and quadrature-phase, and `n_ants` is the number of antennas. The `crest_factor` chosen was 8.4 dB, which corresponds to a normal value of 2.6. This means the full data range is  $\pm 2.6$  standard deviations. The input data was quantized to 8 bits in this simulation. The received symbol data was quantized to 8 bits.

The target output was determined according to the signal-to-noise ratio (SNR) of the final combined signal. The target SNR was chosen to be 6dB, and the target power per antenna was chosen so that a noise value of less than 2.6 noise standard deviations at 6dB SNR does not take the output out of range. To monitor the performance of the algorithm, the following quantities were computed for each iteration, stored and plotted:

- Squared difference between weight-combined inputs and the target output;

$$|w^H(n) \cdot r(n) - d|^2$$

- Antenna coverage pattern.

## 3.2 Performance

Performance is measured in terms of the antenna coverage pattern and weight combined inputs, cycle counts, and data ALU parallel usage.

### 3.2.1 Antenna Coverage Pattern and Weight-Combined Inputs

Figure 4 shows the corresponding MATLAB (floating-point) and the SC3400 (fixed-point) results for the error signal of NLMS. As the figure indicates, the SC3400 and the MATLAB results are virtually identical. Figure 5 shows the antenna coverage pattern for NLMS, which was calculated by the SC3400 fixed-point code running on the MSC8144ADS. The plots correspond to the 200 iterations. DOA of the desired user is 60 degrees. The figure shows that the response is successfully steered towards the desired signal source (DOA = 60 degrees) and away from the strongest interfering sources.



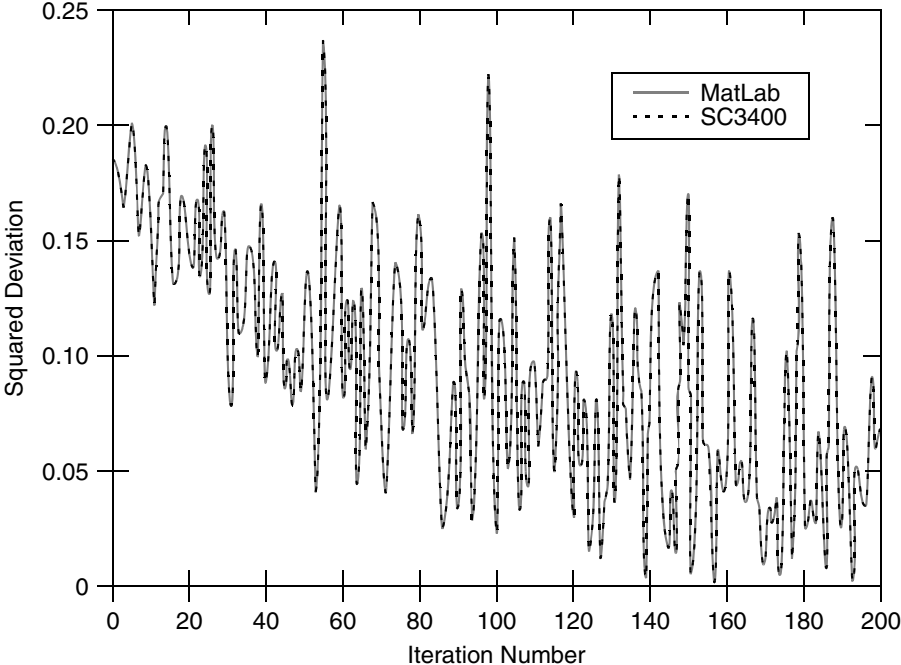


Figure 4. NLMS Squared Deviations Versus Iteration Number

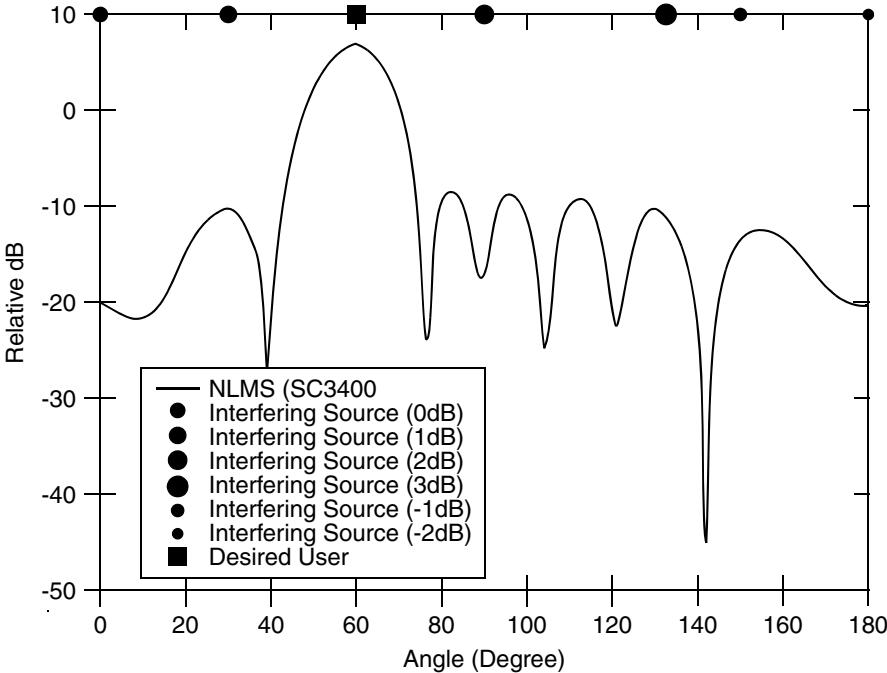


Figure 5. NLMS Antenna Coverage Pattern

## 3.2.2 Cycle Count Measurements

We measured the real-time cycle counts of the implemented NLMS using the on-chip-emulation (OCE30) and the debug and profiling unit (DPU) of the MSC8144 DSP. The conditions for the measurements are described in the following subsections.

### 3.2.2.1 MSC8144ADS

The MSC8144ADS board includes an MSC8144 DSP that is the first device to use the SC3400 architecture from Freescale Semiconductor, Inc. For this project, the MSC8144 was set to the following conditions:

- SC3400 core frequency of 800 MHz.
- L1 data cache and instruction cache were enabled.
- L1 data cache write-back policy was the cacheable write-back.
- L2 instruction cache was enabled.
- Input data and output data were stored in M2 memory.
- Program was put into M2 memory.

### 3.2.2.2 Metrowerks CodeWarrior R3.0

StarCore optimization level 3 (-O3) performs all optimizations, producing the highest-performance code possible with cross-file optimization. Outputs optimize non-linear assembly code.

### 3.2.2.3 Number of Antennas per Sector

The number of antennas is fixed to eight for the loop optimization.

### 3.2.2.4 Number of DPCCH Pilot Symbols per Slot

The slot rate in the UMTS standard is 1,500 slots per second, and there are at most 8 pilot symbols per slot [3]. The cycle count of the NLMS kernel function per 8 pilot symbols is 525 cycles. Therefore, the cycle count per user is calculated as  $525 \text{ cycles} \times 1,500 \text{ slots} / \text{sec} = 0.79 \text{ mega cycles per second (MCPS)}$ . Note that the finger (beam) per user is assumed to be one finger (beam) per user. [Table 4](#) lists the performance results for the modules of the NLMS kernel function ([Equation 1](#) and [Equation 2](#)). The cycle counts in [Table 4](#) also include the overhead. For example, the NLMS kernel function code listed in [Section 2.3.2](#), “NLMS Kernel Function Code,” includes the initial pointer setting, the pointer updates, the variable initialization, storing of the complex error, and the function calls.

**Table 4. Performance Result on Cycle Count**

Module Name	Cycle Count (@ 8 Pilot Symbols)	Memory Usage (MCPS)	Cycle Count per User (Bytes)
NLMS kernel function	525	0.79	488

In addition to the overhead, cache usage can significantly affect overall performance of an application. [Table 5](#) shows the breakdown of the cycles for the NLMS kernel function. To reduce overhead, these counts were measured within 200 iterations. The application cycles are refined as follows:

1. Cycles when there is no bubble in the pipeline (total number of execution set).
2. Cycles when the bubble in the pipeline is due to a change-of-flow (COF) or interrupt.
3. Cycles when the bubble in the pipeline is due to the core resource conflict.
4. Cycles when the bubble in the pipeline is due to program starvation (no instructions in prefetch/dispatch buffer).
5. Cycles when the bubble in the pipeline is due to other internal core reasons (such as program memory fault).
6. Cycles when the bubble in the pipeline is due to data memory holds.

A bubble cycle is a cycle in which there is no dispatch of instructions due to COF or interrupt, program starvation, core resource conflict, or data memory holds. The net application cycles are calculated as the sum of the application cycle refinements in the preceding numbered list (1 + 2 + 3 + 4 + 5 + 6), which in this case yields the following count:  $10,223 + 45 + 1 + 1,803 + 1,042 = 13,114$  cycles. From this result, we can get the cycle counts per the 8 pilot symbols:  $13114 \text{ cycles} / 200 \text{ iterations} = 525 \text{ cycles @ } 8 \text{ pilot symbols}$ .

As [Table 5](#) shows, the main reasons to reduce the performance are the bubble due to core resource conflicts and the bubble due to data memory holds. Note that the program was put into the L1 instruction cache before cycle counts and the cache effect were measured. Note that the cycle counts include the overhead of the module. To reduce overhead, the cycle counts in [Table 5](#) were measured within 200 iterations.

**Table 5. L1 Effect of L1 Instruction Cache and Data Cache**

Event		Counts
Instruction Cache	Misses	0
	Hits	6,461
	Prefetch Hits	0
Data Cache	Misses	26
	Hits	5,191
	Prefetch Hits	0
Application Cycles		13.125 cycles
No Bubble		10,223 cycles
Bubble due to change-of-flow		45 cycles
Bubble due to starvation (no instructions in the prefetch/dispatch buffer)		1 cycle
Bubble due to core resource conflicts		1,803 cycles
Bubble due to data memory holds		1,042 cycles
<sup>1</sup> The application cycles are calculated as $10,223 + 45 + 1 + 1,803 + 1,042 = 13,114$ cycles. <sup>2</sup> $13114 \text{ cycles} / 200 \text{ iterations} = 525 \text{ cycles @ } 8 \text{ pilot symbols}$		

### 3.2.3 Data ALU Parallel Usage

With its four ALUs, the SC3400 core can execute up to four data ALU instructions in a single clock cycle. Data ALU instructions perform operations on the data registers D[0–15]. This section evaluates the instruction-level parallelism in the data ALUs for the NLMS kernel function code. Recall that 114 arithmetic operations are required per iteration for the NLMS kernel function (represented by Equation 1 and Equation 2) with 8 antennas in fixed-point. When these formulas are ported to the C code, additional overhead is incurred, such as variable initialization for the vector product. The CodeWarrior C compiler compiles the code into 130 SC3400 Data ALU instructions, so that 16 extra instructions are caused due to overhead. Appendix A shows the assembly code transformed by the CodeWarrior C compiler, which displays the data ALU instructions used in this implementation (**add, mac, macr, mpy, mpyr, neg, sub, tfr, asll, asrr, clb**).

We compare this instruction count to the actual SC3400 cycle counts. Table 4 shows 525 cycles per slot for the case of 8 pilot symbols per slot, which corresponds to 59 cycles per iteration for 8 loop iterations, plus 57 cycles for overhead from the initialization. (The cycle count per loop iteration can be evaluated from the SC3400 instruction sets between LOOPSTART0 and LOOPEND0 indicated in Appendix A). It follows that the 130 data ALU instructions for the NLMS kernel function execute in 59 cycles, which translates to an average data ALU usage of 2.20 (= 130 data ALU instructions/59 SC3400 execution cycles). This corresponds to an average of 55 percent loading on the SC3400 data ALUs because the SC3400 has four ALUs.

We further investigate the parallelism of the inner loop of the NLMS kernel function (represented by Equation 1 and Equation 2), which we refer to as the “loop kernel.” This part of the code is the most important for optimizing cycle counts because it executes the most frequently. As shown in Appendix A, cycle counts for operations within the NLMS loop kernel are evaluated as shown in Table 6.

**Table 6. Cycle Counts for Operations in the NLMS Loop Kernel**

Normalizing factor in Equation (2)	36 Data ALU instruction/10 SC3400 execution cycles
Weight-combines input vector in Equation (1)	34 Data ALU instructions/10 SC3400 execution cycles
Weight-update in Equation (2)	32 Data ALU instructions/14 SC3400 execution cycles

Combining this result gives an average loop kernel data ALU usage of 3.00 (out of 4 total data ALUs), which corresponds to the average of 75 percent loading on SC3400 data ALUs. The data ALU usage indicates that there is still room for optimization of both the NLMS kernel and its loop kernel. Table 7 summarizes the data ALU usage in this implementation.

**Table 7. Data ALU Parallel Usage for NLMS Kernel Function**

Arithmetic Operations in Mathematical Formula	SC3400 Data ALU Instruction	SC3400 Execution Cycle Counts
114	130	59
Average Data ALU Usage		Loop Kernel Data ALU Usage
2.20 (55% loaded)		3.00 (75% loaded)

## 4 Conclusion

The NLMS algorithm was implemented on the StarCore SC3400 DSP core, and the performance was evaluated on MSC8144 DSP. Under our static model, the antenna coverage pattern shows that the response is successfully steered towards the desired signal source and away from the strongest interfering sources. In addition, the SC3400 fixed-point code incurs very little loss of accuracy compared with the floating-point MATLAB results.

Table 8 shows UMTS single-user resource requirements for NLMS kernel function implemented on the SC3400 core for various update scenarios. These measurements are based on DPCCH pilot symbols per slot from 3 to 8 [3]. Note that the finger (beam) per user is assumed to be one finger. The resulting optimized cycle counts of NLMS kernel function is 0.79 MCPS for 8 pilot symbols per slot, including overhead such as the initial pointer setting, the pointer updates, the variables initialization, storing of the complex error, the function calls, and the cache effect.

**Table 8. Input Data and Measured cycle Counts for NLMS Kernel Function**

DPCCH Pilot Symbols per Slot	Input Data Rate [ksps]	Cycle Counts per User [MCPS]
8	192	0.79
7	168	0.69
6	144	0.59
5	120	0.49
4	96	0.39
3	72	0.30

**Note:** Slot rate is 1,500 slots per second. For the worst case of one update per pilot symbol with 8 pilot symbol per slot, the required input data is  $2 I/Q \times 8 \text{ antennas} \times 8 \text{ pilot symbols} \times 1,500 \text{ slots per second} = 192 \text{ ksps}$ .

Table 8 shows the input data rates and the measured cycle counts for the NLMS kernel function for various weight update rates. In our implementation of the NLMS kernel function, the 130 data ALU instructions execute in 59 cycles. An average data ALU usage is 2.20, which corresponds to an average of 55 percent loading on the SC3400 data ALUs because the SC3400 core has four ALUs. The key to the speed optimization is to increase the instruction-level parallelism in a loop. The average loop kernel data ALU usage in our implementation is 3.00 (out of 4 total data ALUs), which corresponds to the average of 75 percent loading on the SC3400 data ALUs. The optimization techniques described in this application note produced the speed optimization in the instruction-level parallelism in data ALU as shown in Table 9.

**Table 9. Data ALU Parallel Usage for NLMS Kernel Function**

Average Data ALU Usage	Loop Kernel Data ALU Usage
2.20 (55% loaded)	3.00 (75% loaded)

## 5 References

1. S.Tanaka, M.Sawahashi, F.Adachi, "Pilot Symbol-Assisted Decision-Directed Coherent Adaptive Array Diversity for DS-CDMA Mobile Radio Reverse Link," *IEICE Trans. Fundamentals*, Vol. E80-A, No.12, December 1997.

**Document Revision History**

2. Freescale *Enterprise C Compiler User's Guide*. Revised August 2006.
3. *3GPP TS 25.211 V3.10.0 (2002-03) Technical Specification*, 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Physical channels and mapping of transport channels onto physical channels (FDD).

## 6 Document Revision History

Table 10 provides a revision history for this application note.

**Table 10. Document Revision History**

Revision Number	Change(s)
0	Initial release.

# Appendix A Assembly Code of NLMS Kernel Function

Statistical results of the assembly code of the NLMS kernel function transformed by the CodeWarrior C compiler are summarized in [Table A-1](#).

**Table A-1. NMLS Kernel Function**

SC3400 Data ALU Instructions		SC3400 Execution Cycle Counts			
130		59			
Loop Kernel					
Loop Kernel (i): normalized factor in Equation (2)		Loop Kernel (ii): Weight-combined input vector in Equation (1)		Loop Kernel (ii): Weight update in Equation (2)	
Data ALU Instruction	SC3400 Execution Cycles	Data ALU Instructions	SC3400 Execution Cycles	Data ALU Instructions	SC3400 Execution Cycles
36	10	34	10	32	14

The DALU instructions used in this implementation are **add**, **mac**, **macr**, **mpy**, **mpyr**, **neg**, **sub**, **tfr**, **asll**, **asrr**, and **clb**.

```

;*****
;Function _NLMS, ; Stack frame size: 24 (0 from LLT)
;Calling Convention: 1
;Parameter pstArg passed in register r0
;*****

        GLOBAL _NLMS
        ALIGN 16
_NLMS TYPE func OPT_SPEED
        SIZE _NLMS,F_NLMS_end-_NLMS,16
;PRAGMA stack_effect _NLMS,24
        falign
_NLMSOptimized_away_inline_blocks
DW1
F_NLMS_blockstart
[
    move.l    (r0)+,r1
    push.2l  d6:d7
]
DW_2
DW2
        push.2l  r6:r7
DW_3
DW3
        adda    #8,sp
DW_4
DW4
    
```

## Assembly Code of NLMS Kernel Function

```

[
    move.l    (r0)+,r6
    tfra     r1,r2
]
DW5
[
    adda     #>40,r1,r4
    adda     #>32,r1,r8
]
DW6
    adda     #>8,r1,r9
DW7
    move.l   (r0)+,d3
DW8
[
    move.l   (r0)+,r3
    move.l   d3,r12
]
DW9
[
    move.l   d3,r5
    move.l   d3,r7
]
DW10
    adda     #>8,r12,r11
DW11
    move.f   (r0)+,d5
DW12
[
    move.w   (r0),d7
    move.l   d5,m0
]
DW13
[
    tstgt    d7
    adda     #>2,r3,r0
]
DW14

```



```

        bf      L16

DW15
[
    doen0     d7
    dosetup0  L28
]

    FALIGN
    LOOPSTART0
    falign

L28
;;;;;;;;;;;;;
;;; Loop Kernel (i) Start
;;;;;;;;;;;;;

DW16
[
    clr      d7
    clr      d6
    clr      d4
    clr      d5
    doensh3  #3
    move.4f  (r5)+,d0:d1:d2:d3
]

DW17
[
    asrr     #<2,d1
    asrr     #<2,d0
    asrr     #<2,d2
    asrr     #<2,d3
]

    FALIGN
    LOOPSTART3

DW18
[
    mac      d0,d0,d5
    mac      d1,d1,d4
    mac      d2,d2,d6
    mac      d3,d3,d7
    move.4f  (r5)+,d0:d1:d2:d3
]

DW19
[

```

**Assembly Code of NLMS Kernel Function**

```

    asrr    #<2,d0
    asrr    #<2,d1
    asrr    #<2,d2
    asrr    #<2,d3
]

        LOOPEND3

DW20
[
    move.l  #1073741824,d13
    doensh2 #3
]

DW21
[
    mac     d0,d0,d5
    mac     d2,d2,d6
    mac     d3,d3,d7
    mac     d1,d1,d4
    move.l  #1441988608,d12
]

;;;;;;;;;;;;;
;;; Loop Kernel (i) End
;;;;;;;;;;;;;

DW22
[
    add     d5,d6,d8
    add     d4,d7,d10
    tfr     d13,d14
    move.4f (r7)+,d4:d5:d6:d7      ;<- Loop kernel (ii)
    move.l  #1441988608,d13
]

DW23
[
    add     d8,d10,d15
    clr     d8
    move.f  (r3),d11
    move.f  (r0),d9
]

DW24
[
    tfr     d15,d10
    tfr     d15,d0
    clr     d15
    adda    #<4,r3
    adda    #<4,r0

```

```

]
DW25
[
    clb      d10,d10
    tfr      d0,d15
    move.l   d15,m2
    move.4f  (r2)+,d0:d1:d2:d3      ;<- Loop kernel (ii)
]

```

```

DW26
[
    neg      d10
    mac      -d2,d6,d8              ;<- Loop kernel (ii)
    mac      -d0,d4,d11            ;<- Loop kernel (ii)
    mac      -d0,d5,d9              ;<- Loop kernel (ii)
]

```

```

DW27
[
    asll     d10,d15
    sub      #<3,d10
]

```

```

DW28
    move.l   d8,m1

```

```

DW29
[
    mac      -d12,d15,d14
    move.l   m2,d15
    move.l   m0,d12
]

```

```

DW30
[
    add      d14,d14,d8
    mac      -d2,d7,d15            ;<- Loop kernel (ii)
]

```

```

DW31
[
    mac      #22003,d8,d13
    mpy      d8,d8,d2
]

```

```

DW32
    move.l   d15,m2
DW33

```

**Assembly Code of NLMS Kernel Function**

```

[
    mac      d2,d13,d13
    mpy      d2,d2,d15
]
DW34
[
    mac      d15,d13,d13
    mpy      d15,d15,d14
]
DW35
[
    mac      d14,d13,d13
    move.l   m2,d14
]
DW36
[
    mpy      d12,d13,d12
    move.l   m1,d13
]
DW37
    asll     d10,d12

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Loop Kernel (ii) Start
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    FALIGN
    LOOPSTART2
DW38
[
    mac      d1,d4,d9
    mac      d3,d6,d14
    mac      -d3,d7,d13
    mac      -d1,d5,d11
    move.4f  (r7)+,d4:d5:d6:d7
    move.4f  (r2)+,d0:d1:d2:d3
]
DW39
[
    mac      -d0,d5,d9
    mac      -d2,d7,d14
    mac      -d2,d6,d13
    mac      -d0,d4,d11
]

```

```

LOOPEND2
DW40
[
    tfr      d14,d15
    tfr      d13,d8
    mac      d1,d4,d9
    mac      -d1,d5,d11
    doen1    #<2
    dosetup1 L27
]
DW41
[
    mac      d3,d6,d15
    mac      -d3,d7,d8
    move.w   #2,n3
]
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Loop Kernel (ii) End
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DW42
[
    add      d11,d8,d11
    add      d9,d15,d9
]
DW43
[
    mpyr     d11,d12,d13
    mpyr     d9,d12,d14
    tfr      d11,d12
    tfr      d9,d15
]
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Loop Kernel (iii) Start
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    FALIGN
    LOOPSTART1
    falign
L27
DW44
[
    move.4f  (r1)+n3,d0:d1:d2:d3
    move.4f  (r12)+n3,d4:d5:d6:d7

```



```

[
    tfr      d12,d11
    tfr      d15,d9
]

LOOPEND0
DW52
[
    move.l   d9,(sp-4)
    move.l   d11,(sp-8)
]

    falign
    L16
DW53
[
    move.f   (sp-8),d6
    move.f   (sp-4),d5
]
DW54
    moves.f  d6,(r6)+
DW55
[
    suba     #8,sp
    moves.f  d5,(r6)
]
DW_56
DW56
[
    pop.2l   r6:r7
    pop.2l   d6:d7
]
DW_57
DW57
    rts
    F_NLMS_blockend
DW58
    GLOBAL F_NLMS_end
    F_NLMS_end
    FuncEnd_NLMS
    TextEnd_nlms
    ENDSEC

```

### **How to Reach Us:**

**Home Page:**

www.freescale.com

**email:**

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  
480-768-2130  
support@freescale.com

**Europe, Middle East, and Africa:**

Freescale Halbleiter Deutschland GmbH  
Technical Information Center  
Schatzbogen 7  
81829 Muenchen, 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  
+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  
support.asia@freescale.com

**For Literature Requests Only:**

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

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

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

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

© Freescale Semiconductor, Inc., 2007.

Document Number: AN3351  
Rev.0  
03/2006