Decision Feedback Equalizer Implementation for the SC140 DSP

Application Note

by

Ahsan Aziz

AN2072/D
Rev. 0, 12/2000

MOTOROLA

How to reach us:

USA/EUROPE/Locations Not Listed: Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado, 80217
1-303-675-2140 or 1-800-441-2447

JAPAN: Motorola Japan, Ltd.; SPS, Technical Information Center, 3-20-1, Minami-Azabu, Minato-ku,
Tokyo 106-8573 Japan. 81-3-3440-3569

ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd., Silicon Harbour Centre, 2 Dai King Street,
Tai Po Industrial Estate, Tai Po, N.T., Hong Kong. 852-26668334

HOME PAGE: http://www.motorola.com/semiconductors/

Abstract and Contents

In cellular systems, the radio waves reaching both the mobile stations and the base station undergo reflection or refraction from buildings, cars, hills, and so forth. The signal reaching a receiver antenna is said to have undergone fading, which means the receiver antenna does not receive the original signal but many different time, phase, and amplitude shifted versions of the signal. In most types of TDMA communication systems, signals undergo frequency-selective fading because of their short-symbol duration. In designing a receiver for such a mobile system, these conditions need to be accounted for to get the best performance. In communications, such as GSM/EDGE, a gaussian filter is used for spectral compaction. Both the modulation and channel impairments introduce severe inter-symbol interference (ISI). To lessen the performance impairments resulting from ISI, receivers for these types of systems use channel equalizers. An equalizer estimates the channel impulse response and uses that information to remove as much ISI as possible. In GSM/EDGE there are training bits in each burst that are used for various purposes such as frequency tracking, timing tracking, and channel estimation. The information is used later in the receiver to create an improved symbol decision. This application note explores an efficient implementation of a decision feedback equalizer (DFE) that is based on Cholesky factorization for the Motorola StarCore SC140, DSP architecture. Cholesky factorization yields a finite solution of an infinite-length spectral factorization [1]

1 Introduction

It is well known that a maximum likelihood sequence equalizer (MLSE) is the optimum equalizer for a typical intersymbol interference (ISI) channel. Unfortunately, the complexity of the MLSE equalizer increases exponentially with the channel-memory length. To reduce this computational load, many sub-optimal solutions have been developed. Decision feedback equalizers (DFE) are a good sub-optimal solution. With the introduction of faster digital signal processors (DSP), computationally-intense algorithms can now be easily implemented. The DFE can be implemented as a combination of simple FIR filters. This application note presents a method for computing the DFE coefficient and a method for implementing the algorithm on the StarCore, SC140 digital signal processor.

This document is arranged as follows: Section 2, “Decision Feedback Equalization Theory,” presents basic ideas used in the development of the DFE. Section 3, “Derivation of DFE Filter Coefficients,” presents the method used in deriving the feed forward and feedback coefficients. Section 4, “Implementation of the Algorithm,” presents implementation details of the algorithm on the SC140. Results for this implementation are presented in Section 5, “Results.” Section 6, “Conclusion,” summarizes the document. References are provided in Section 7, “References.”

2 Decision Feedback Equalization Theory

There are two major categories of a channel equalizer; linear and non-linear, as shown in Figure 1. Non-linear equalization is needed when the channel distortion is too severe for the linear equalizer to mitigate the channel impairments. An example of a linear equalizer is a zero-forcing equalizer (ZFE) and as the name implies, it forces ISI to become zero for every symbol decision. A zero-forcing equalizer enhances noise and results in performance degradation. On the other hand, a Minimizes Mean Square Error-Linear Equalizer (MMSE-LE), minimizes the error between the received symbol and the transmitted symbol without enhancing the noise. Although MMSE-LE has better performance than ZFE, its performance is not enough for channels with severe ISI. An obvious choice for channels with severe ISI is a non-linear equalizer.

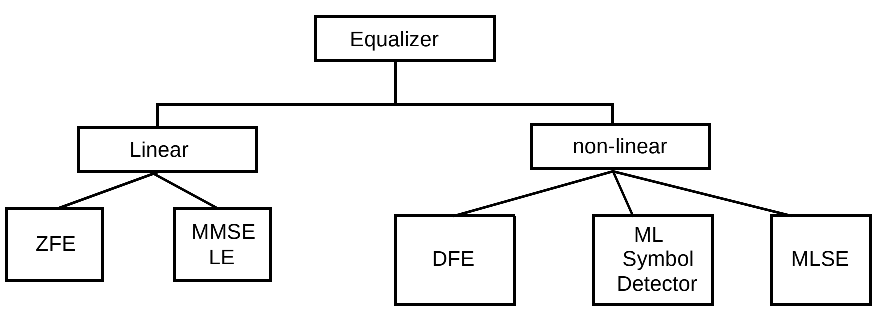

Figure 1. Linear and Non-Linear Process for Equalization of Transmission Channel

It is well know that a maximum likelihood sequence equalizer (MLSE) gives optimum performance. It tests all possible data sequences and chooses the data with the maximum probability as the output. Generally, the Viterbi algorithm provides a solution to the problem with MLSEs of a finite-state, discrete-time Markov process. However, the computational complexity of an MLSE increases with channel spread and signal constellation size. The number of states of the Viterbi decoder is expressed as M^L, where M = the number of symbols in the constellation, and L = the channel-spread length –1. Therefore, a typical 8-PSK constellation with a channel span of 5, translates to a 8^4 = 4096, state Viterbi decoder, which makes it unsuitable for cost effective implementation. In this situation, the obvious choice is to use sub-optimal solutions such as a DFE or a DFE followed by a Viterbi equalizer.

A decision feedback equalizer makes use of previous decisions in attempting to estimate the current symbol with a symbol-by-symbol detector. Any tailing ISI caused by a previous symbol is reconstructed and then subtracted. The DFE is inherently a non-linear device, but by assuming that all the previous decisions were correct, a linear analysis can be made. There are different variations of DFEs. The choice of which type of DFE to use depends on the allowable computational complexity and required performance. Considered in this application note is an MMSE-DFE (Minimum Mean Square Error) consisting of a:

- linear, anti-causal, feed forward filter, F(D)
- linear, causal, feedback filter, 1-B(D), with bo=1
- simple detector (threshold block)

The input to the feedback filter is the decision of the previous symbol (from the decision device). Figure 2 shows the layout of the DFE.

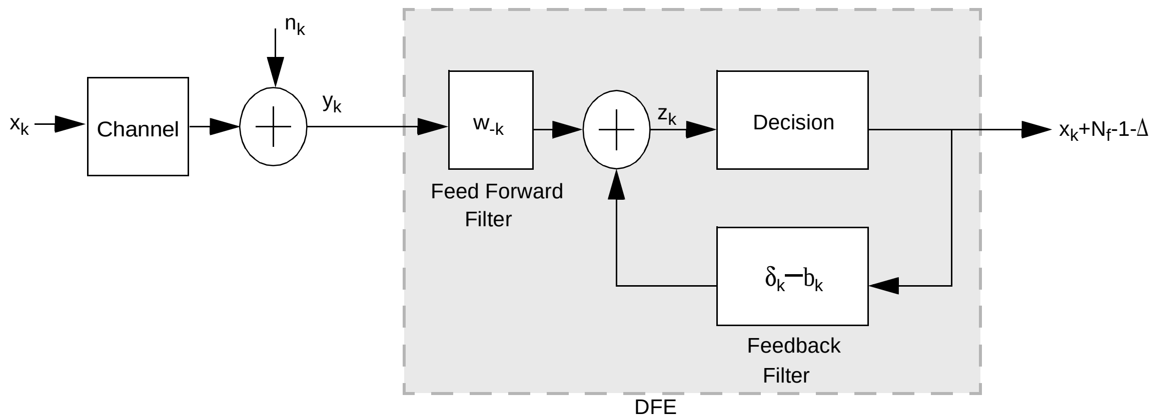

Figure 2. DFE Block Diagram

The ideal, infinite-length feed forward filter is a noise whitening filter that results in an overall response with minimum phase. Choosing an infinite-length filter eliminates the delay optimization, because the overall response, consisting of transmit, channel, receiver, sampler, and FF filters is simply a filter with minimum phase. The first tap in the overall response is the main tap. Spectral factorization is a very important concept used in deriving DFE taps. A finite version of the infinite tap DFE is derived in [1]. The Cholesky factorization theory is a useful tool in deriving the finite-length equivalent to the infinite-length DFE.

Terms used in DFE development are described as follows:

1. A filter response *F(D)*, is **canonical** if it is:
   - causal ($f_k=0$, for $k<0$)
   - monic ($f_0=1$)
   - minimum phase

2. If $F(D)$ is canonical then $F^{*}(D^{-*})$ is:

   • anti-causal

   • monic

   • maximum-phase

3. An n × n symmetric matrix, A, is **positive definite** if all diagonal elements and submatrices of A are positive definite. The definition for a positive definite matrix is expressed in Equation 1.

$$X \neq 0 \Rightarrow X^T A X = \sum_{i=1}^{n} \sum_{i=1}^{n} a_{ij} x_i x_j > 0 \qquad \textbf{Eqn. 1}$$

4. Matrices with constant entries along a diagonal are called **Toeplitz** matrices. Equation 2 expresses a Toeplitz matrix with the number three as the constant entry along the diagonal.

$$T = \begin{bmatrix} 3 & 4 & 0 \\ 2 & 3 & 4 \\ 1 & 2 & 3 \end{bmatrix}, \text{T is Toeplitz} \qquad \textbf{Eqn. 2}$$

3 Derivation of DFE Filter Coefficients

The name MMSE-DFE implies that the DFE coefficients are derived under MMSE criteria. Consider the development, where b_k and w_k are the feedback and feed forward filter coefficients derived in minimum-mean-square-sense, by making the error orthogonal to the received sequence. Equation 3 represents the received signal as, $y(t)$, the channel-input data symbols as, x_k, and the channel-impulse response as, $h(t)$; where $n(t)$ is additive-white Gaussian noise and T is the symbol duration.

$$y(t) = \sum_{m} x_m \cdot h(t - mT) + n(t) \qquad \textbf{Eqn. 3}$$

If l oversampling is used, then the sampled channel-output is expressed in Equation 4. For our implementation, l is set to 1, and channel memory is expressed as ν in the equations.

$$y_k = \sum_{m=0}^{\nu} h_m \cdot x_{k-m} + n_k \qquad \textbf{Eqn. 4}$$

Eqn. 5

$$\mathbf{y_k} = \begin{bmatrix} y\left(k + \frac{(l-1)}{l}T\right) \\ \circ \\ \circ \\ y(kT) \end{bmatrix} \equiv \begin{bmatrix} y_{l-1,k} \\ \circ \\ \circ \\ y_{0,k} \end{bmatrix} \qquad \mathbf{h_m} = \begin{bmatrix} h\left(m + \frac{(l-1)}{l}T\right) \\ \circ \\ \circ \\ h(mT) \end{bmatrix} \equiv \begin{bmatrix} h_{l-1,m} \\ \circ \\ \circ \\ h_{0,m} \end{bmatrix}$$

$$\mathbf{n_k} = \begin{bmatrix} n\left(k + \frac{(l-1)}{l}T\right) \\ \circ \\ \circ \\ n(kT) \end{bmatrix} \equiv \begin{bmatrix} n_{l-1,k} \\ \circ \\ \circ \\ n_{0,k} \end{bmatrix}$$

By combining N_f successive l-tuples of samples y_k:

$$\begin{bmatrix} y_{k+N_f-1} \\ y_{k+N_f-2} \\ \circ \\ y_k \end{bmatrix} = \begin{bmatrix} h_0 & h_1 & \dots & h_\nu & 0 & \dots & 0 \\ 0 & h_0 & h_1 & \dots & h_\nu & 0 & \dots \\ \circ & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & h_0 & h_1 & \dots & h_\nu \end{bmatrix} \cdot \begin{bmatrix} x_{k+N_f-1} \\ x_{k+N_f-2} \\ \circ \\ x_{k-\nu} \end{bmatrix} + \begin{bmatrix} n_{k+N_f-1} \\ n_{k+N_f-2} \\ \circ \\ n_k \end{bmatrix} \qquad \textbf{Eqn. 6}$$

A more compact representation of Equation 6 is expressed in Equation 7.

$$\mathbf{y}_{k+\mathbf{N}f-1,k} = \mathbf{Hx}_{k+Nf-1,k-\nu} + \mathbf{n}_{k+Nf-1,k} \qquad \textbf{Eqn. 7}$$

The equalizer output error is expressed in Equation 8.

$$\mathbf{e}_k = \tilde{\mathbf{b}}^* x_{k,k-\nu} - \mathbf{w}^* y_{k+N_f-1,k} \qquad \textbf{Eqn. 8}$$

The w^* [1] feed forward filter taps are expressed in Equation 9.

$$\mathbf{w}^* = [w^*_{-(Nf-1)} w^*_{-(Nf-2)} \dots\dots w_0] \qquad \textbf{Eqn. 9}$$

For a decision delay of Δ, the corresponding MMSE is expressed in Equation 10.

$$\mathbf{E}\left\{\left|\mathbf{e}_k\right|^{\mathbf{2}}\right\} = \mathbf{E}\{\mathbf{e}_k \cdot \mathbf{e}_k^{\mathbf{*}}\} = \mathbf{E}\left\{(\mathbf{x}_{k-\Delta} - \mathbf{wY}_k + \mathbf{bx}_{k-\Delta-1})(\mathbf{x}_{k-\Delta} - \mathbf{wY}_k + \mathbf{bx}_{k-\Delta-1})^{\mathbf{*}}\right\} \qquad \textbf{Eqn. 10}$$

Equation 11 shows where ***b*** is the vector of the coefficients for the feedback FIR filter and $x_{k-\Delta-1}$ is the vector of the data symbols in the feedback path.

$$\mathbf{b} = \begin{bmatrix} 1 & b_0 & \dots & b_{Nb} \end{bmatrix} \qquad \textbf{Eqn. 11}$$

1. $(.)^*$ is the complex conjugate transpose

Applying the orthogonal principle by making the error orthogonal to the output we get Equation 12.

$$E\{\mathbf{e}_k \cdot \mathbf{Y}_k^{*}\} = 0 \qquad \textbf{Eqn. 12}$$

In other words, the optimum error sequence is uncorrelated with the observed data. This simplifies to Equation 13, which gives the relation between the DFE feedback and feed forward filter coefficients

$$\mathbf{b}^{*}\mathbf{R}_{xy} = \mathbf{w}^{*}\mathbf{R}_{yy} \qquad \textbf{Eqn. 13}$$

The FIR MMSE-DFE autocorrelation matrix is expressed in Equation 14.

$$\mathbf{R}_{yy} = \left(\mathbf{E}\left\{\tilde{\mathbf{Y}}_{k+Nf-1,k}\mathbf{Y}^{*}_{k+Nf-1,k}\right\}\right) = \mathbf{S}_x\mathbf{H}\mathbf{H}^{*} + \mathbf{R}_{nn} \qquad \textbf{Eqn. 14}$$

Where $\mathbf{R}_{nn} = N_0\mathbf{I}_{\mathbf{Nf}}$ and where N_0 is the noise power, and where I is an identity matrix, the input-output cross-correlation matrix, where S_x is the signal power as expressed in Equation 15.

$$\mathbf{R}_{xy} = \mathbf{E}[\mathbf{x}_{k,k-\nu}\mathbf{y}^{*}_{k+Nf-1,k}] = S_x[\mathbf{0}_{(\nu+1)(Nf-1)}\ldots\mathbf{I}_{(\nu+1)}]\mathbf{H} \qquad \textbf{Eqn. 15}$$

The mean-square error is expressed in Equation 16.

$$\left(\mathbf{R}_{xx} - \mathbf{R}_{xy}\mathbf{R}^{-1}_{yy}\mathbf{R}_{yx}\right) = S_x\left[\mathbf{0}\ \mathbf{I}_{\nu+1}\right]\left[\mathbf{I}_{Nf+\nu} - \mathbf{H}^{*}\left(\mathbf{H}\mathbf{H}^{*} + \frac{1}{SNR}\mathbf{I}_{Nf+\nu}\right)^{-1}\mathbf{H}\right]\begin{bmatrix}\mathbf{0}\\ \mathbf{I}_{\nu+1}\end{bmatrix} \qquad \textbf{Eqn. 16}$$

Simplifying Equation 16 by using the matrix inversion lemma[1] results in Equation 17.

$$S_x\left[\mathbf{0}\ \mathbf{I}_{\nu+1}\right]\left[\mathbf{I}_{Nf+\nu} - \mathbf{H}^{*}\left(\mathbf{H}\mathbf{H}^{*} + \frac{1}{SNR}\mathbf{I}_{Nf+\nu}\right)^{-1}\mathbf{H}\right]\begin{bmatrix}\mathbf{0}\\ \mathbf{I}_{\nu+1}\end{bmatrix} = N_0\left[\mathbf{0}\ \mathbf{I}_{\nu+1}\right]\left(\left(\mathbf{H}\mathbf{H}^{*} + \frac{1}{SNR}\mathbf{I}_{Nf+\nu}\right)^{-1}\begin{bmatrix}\mathbf{0}\\ \mathbf{I}_{\nu+1}\end{bmatrix}\right) \qquad \textbf{Eqn. 17}$$

The middle term in the right-hand side of Equation 18, is defined as a Cholesky factorization, where LDL' is the Lower-Diagonal-Upper.

$$\mathbf{R}^{-1}_{xx} + \mathbf{H}^{*}\mathbf{R}^{-1}_{nn}\mathbf{H} = \left(\mathbf{R}_{xx} - \mathbf{R}_{xy}\mathbf{R}^{-1}_{yy}\mathbf{R}_{yx}\right)^{-1} = \frac{1}{SNR}\mathbf{I}_{Nf+\nu} + \mathbf{H}^{*}\mathbf{H} = \mathbf{L}\mathbf{D}\mathbf{L}^{*} \qquad \textbf{Eqn. 18}$$

1. The matrix inversion lemma:

$$H^{*}\left(HH^{*} + \frac{1}{SNR}I_{N_f}\right)^{-1}H = \left(HH^{*} + \frac{1}{SNR}I_{N_f}\right)^{-1}H^{*}H$$

Where **L** is a lower-triangular monic matrix, **D** is a diagonal matrix shown in Equation 19.

Eqn. 19

$$\mathbf{L} = \begin{bmatrix} l_0 & \cdots & l_{N_f+\nu-1} \end{bmatrix}$$

$$\mathbf{L}^{-1} = \begin{bmatrix} u_0' \\ \cdots \\ u_{N_f+\nu-1} \end{bmatrix}$$

$$\mathbf{D} = \begin{bmatrix} d_0 & \cdots & 0 \\ 0 & \cdots & 0 \\ 0 & \cdots & d_{N_f+\nu-1} \end{bmatrix}$$

Because **L** is a monic matrix, its columns constitute a basis for the (N_f+ν) dimensional vector space. It is shown in [1] that the optimal setting for ***b*** is column **L** which corresponds to the highest value of “d” in the diagonal matrix. When the feedback coefficients are located, the solution expressed in Equation 20, gives the optimal setting for **w**, the feed forward coefficient.

$$\mathbf{w}^*_{opt} = \mathbf{b}^*_{opt}\mathbf{R}_{xy}\mathbf{R}^{-1}_{yy} \qquad \textbf{Eqn. 20}$$

The first coefficient of ***b*** is always unity, therefore a simple back substitution method can be used to solve for the feed forward coefficients. This process for derivation of DFE filter coefficients, eliminates the need for the matrix inversion.

It has been seen that the:

- number of taps in the feedback filter $N_b = \nu$, $N_f > N_b$
- optimum delay Δ is between ν and $N_f + \nu$

4 Implementation of the Algorithm

This section presents an implementation of the algorithm using Matlab and the corresponding DSP implementation techniques. The Matlab implementation is made using practical requirements of a typical communication system (such as; GSM), with the ultimate intent of implementing the algorithm in the SC140. Table 1 presents some of the design parameters that are used.

Table 1. Parameter Considerations for Communication Systems

Parameter Considered	Parameter Value	Comments
channel-impulse response	N/A	Assumed available
Channel memory	channel-impulse response length 1	
Complex channel-impulse response length	5	Typically this is the length of the channel-impulse response for GSM
Estimate of the SNR	N/A	Assumed available

Table 1. Parameter Considerations for Communication Systems (Continued)

Parameter Considered	Parameter Value	Comments
Feed forward filter length	8	Chosen arbitrarily, typically enough for GSM channel. The only constraint is that the size of the convolution matrix needs to be multiple of 4.
Optimum feedback taps by delay optimization		Minimizes ISI, and maximizes SNR at the decision point
SNR	5 dB — 25 dB	Typical range of operation
Symbol spaced equalizer	N/A	Chosen over a T/l spaced case because its computational complexity is “l” times less. Performance does suffer.

The DSP software is written in such a way that these parameters can easily be tailored to the needs of the communication system where the DFE is used. Feedback and feed forward filter coefficients for the DFE are obtained as follows:

1. Generate the convolution matrix H for an N_f tap feed forward filter ($N_f = 8$) and a complex channel-impulse response memory length ν (typically obtained from a channel estimation block).

   This combination produces a convolution matrix of size C. For our example it will be 8 × 12.

2. Compute the matrix; **A = H × H** + (1/SNR_lin)***I**.

   The (1/SNR_lin)***I** creates a diagonal matrix with all the diagonal terms equal to 1/SNR_lin. (SNR_lin = SNR is expressed in linear scale).

   NOTE:

   The H × H (size 12 × 12) matrix is a positive semi-definite matrix, with N_f ($N_f = 8$) numbers of positive eiganvalues, and four, zero eiganvalues (channel memory size = 4). When the term 1/SNR_lin is added to all the diagonal entries, the matrix becomes strictly positive-definite.

3. Compute the Cholesky factorization of Matrix **A**.

   When computing the Cholesky factorization, use the “outer product version” of the factorization. This algorithm computes a lower triangular **G***, such that **A = GG***. For all i > j, **A**(i,j) is over-written by **G**(i,j).

   The Matlab code for steps 1through 3 is given in Code Listing 1.

Code Listing 1. Matlab Code to Compute the Cholesky Factorization

```
function [L,D,A,U]=outer_cholesky(A)

n=size(A,1);
for k=1:n-1
             A(k,k)=sqrt(A(k,k));
             A(k+1:n,k)=A(k+1:n,k)/A(k,k);
             for j=k+1:n
                    i=j:n;
                    A(i,j)=A(i,j)-A(i,k)*conj(A(j,k));
             end
end
```

4. Factor matrix A into **LDL***, where **L** is a lower-triangular monic matrix and **D** is a diagonal matrix.

   When complete, column of **L**, corresponding to the highest diagonal element of **D**, gives the optimum feedback filter taps (Matlab code for this is omitted here).

5. Compute the optimum feed forward taps. Because **L** is a monic matrix, back substitution is an easy way to solve for the feed forward filter taps. Code Listing 2 shows how the back substitution routine is implemented in Matlab.

   The feedback and the feed forward taps are then used in the channel equalizer to equalize all the received symbols within a frame of data (the channel is assumed to be quasi-static).

Code Listing 2. Matlab Back Substitution Routine

```
l=L; % Lower Triangular Monic Matrix
chan_mem=4;
h=channel_h;
%%%%%%%%%%%% BSM%%%%%%%%%%%%%
d_opt=d_opt*scale_factor^2;
sc=4;% scale factor used in DSP implementation

v = zeros(1,delay);
v(delay) = 1/sc;
temp_v=0;
l=l/sc;
%LOOP 1
for kk = delay :-1 :2;
             sum_v=0;
for jj =  kk : delay
             temp_v = -l(jj,kk-1)*v(jj);
             sum_v = sc*sc*temp_v+ sum_v;
end
             v(kk-1) = (sum_v)/sc;
end
w_opt = zeros(1,Nf);
%LOOP 2
for j = 1:Nf
    sum_w = 0;
              for k = 1 : (min(chan_mem, delay-j) + 1)
                   tw(k) = (v(k+j-1))*conj(h(k));
                   sum_w =  sum_w + tw(k);
end
             disp('writing coeff now')
   w_opt(j) = ((1/d_opt)*sum_w*sc)
                   disp('Wrote the above coeff')
end
```

4.1 DFE Implementation in the SC140

A decision feedback equalizer can be implemented on the SC140 efficiently because the SC140 has four arithmetic logic units (ALUs) and special instructions. This provides the advantage of computing complex numbers and performing matrix operations. Setting constraints on the channel-impulse response and the number of filter taps improves the efficiency of the DSP implementation on SC140. The constraints are:

- All the entries of the channel-impulse response need to be less than one.
- The length of the channel-impulse response and the length of the feed forward filter tap needs to be constrained in such a way that the size of the convolution matrix becomes a multiple of four. The size of the convolution matrix is given by ($N_f \times N_f + \nu$), where ν is the channel memory length (ν = channel-impulse length –1).

Computation of the DFE coefficients is accomplished through several assembly routines. These routines are called from a main assembly calling routine. Figure 3 illustrates the flow of the assembly program.

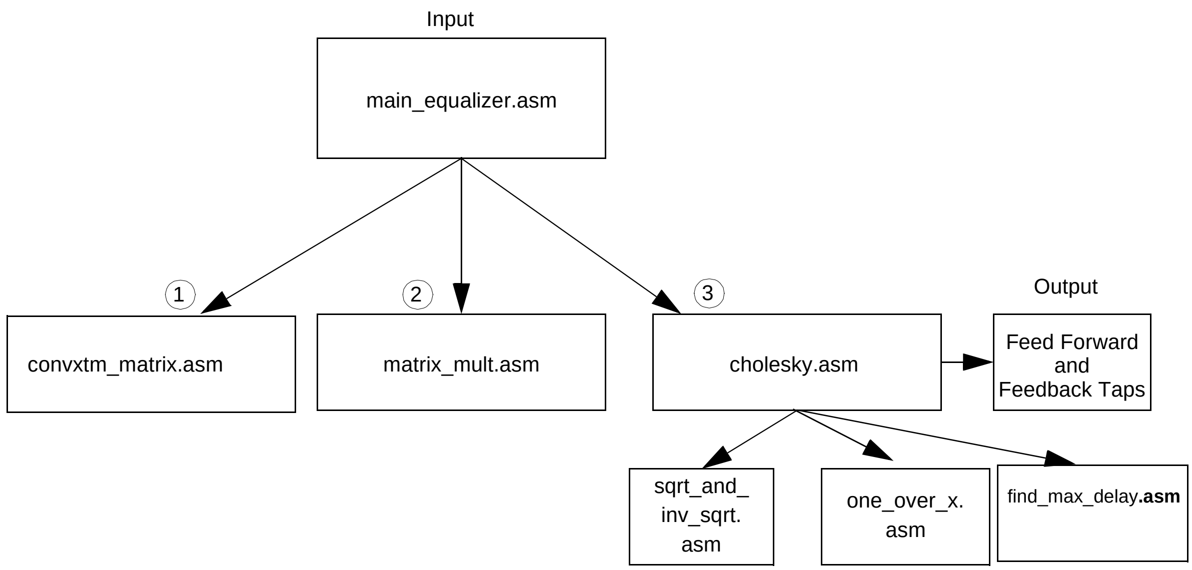

Figure 3. DSP Implementation Process Flow Chart

4.2 Main Calling Routine

The main calling routine, **main_equalizer.asm**, allocates all the memory and defines the globally used variables. This routine contains the complex-channel-impulse response and the linear-scaled value of SNR.

4.3 Forming the Matrix

When implementing this algorithm, it is important to ensure that all the data is properly scaled to be less than one. A key observations is that if the matrix, $\mathbf{H}^*\mathbf{H}+\mathbf{I}(1/\mathrm{SNR})$ has entries less than one, then the Cholesky factor will have entries less than one throughout the computation. This greatly reduces the numeric dynamic range of the algorithm and makes it easier to implement in a fixed point DSP. To achieve this, two constraints have been introduced:

1. The channel-impulse response, must be normalized.

NOTE:

Better results are achieved if normalization is done with care, by using the minimum scale factor that just allows the highest values in the channel-impulse response to be less than one. The process of normalization can be achieved by searching through the elements of the channel-impulse response generation block and normalizing with respect to the highest entry. This remains as a design parameter for the channel-impulse response and will not be addressed here.

2. The output of the matrix multiply block is scaled by right shifting each entry of the matrix multiply by ceil($\log_2$[# of rows in $(\mathbf{H}^*\mathbf{H}) + (1/\mathrm{SNR})\mathbf{I}$]). This ensures that all entries are less than one. For this specific implementation example the scale factor is 1/16.

4.3.1 Generating the Convolution Matrix

The assembly program, **convxtm_matrix.asm**, takes the complex channel-impulse response and generates the transpose of the convolution matrix (similar to using the matlab command, **transpose (H)**, where **H = convxtm(channel_h,N$_f$)**). The transpose of matrix **H** is shown in Equation 21.

$$\text{transpose}(H) = \begin{bmatrix} h_0 & 0 & \dots & 0 & 0 & \dots & 0 \\ h_1 & h_0 & 0 & \dots & 0 & 0 & \dots \\ h_2 & h_1 & h_0 & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 0 & 0 & \dots & h_\nu \end{bmatrix} \quad \textit{Eqn. 21}$$

Note that all complex numbers throughout the implementation are stored in the memory as indicated in Equation 22.

$$\begin{bmatrix} \mathrm{Re}(\circ) \\ \mathrm{Im}(\circ) \\ \dots \end{bmatrix} \quad \textit{Eqn. 22}$$

In the DSP implementation, the channel-impulse response is assumed to be stored in memory (that is; provided by the channel-impulse response block). A memory space of size (N_f*(N_f + ν)*2*2) is zeroed out to store the convolution matrix (channel-impulse response length = 5 and feed forward length = 8), so the complex-conjugate transpose of the convolution matrix will be size 12 × 8. Each entry has a real and an imaginary part and each element is stored in 16-bit memory (2 bytes). Two pointers (address registers) are used. One pointer points to "**channel_h**" and is used to read the complex channel-impulse response and the other pointer is used to point to "**conv_matrix**" and to write the data to the convolution matrix. At the start of the iteration, a pointer points to the start of the memory, where the convolution matrix will be stored, and the matrix is filled column-wise with the channel-impulse response, as shown in Equation 23. Each element of the complex channel-impulse response is read one element at time, using **move.2f** (reads data from memory to register). The program uses **moves.2f** to write each complex data from the register into memory.

Eqn. 23

$$\text{transpose}(H) = \begin{bmatrix} h_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ h_1 & h_0 & 0 & 0 & 0 & 0 & 0 & 0 \\ h_2 & h_1 & h_0 & 0 & 0 & 0 & 0 & 0 \\ h_3 & h_2 & h_1 & h_0 & 0 & 0 & 0 & 0 \\ h_4 & h_3 & h_2 & h_1 & h_0 & 0 & 0 & 0 \\ 0 & h_4 & h_3 & h_2 & h_1 & h_0 & 0 & 0 \\ 0 & 0 & h_4 & h_3 & h_2 & h_1 & h_0 & 0 \\ 0 & 0 & 0 & h_4 & h_3 & h_2 & h_1 & h_0 \\ 0 & 0 & 0 & 0 & h_4 & h_3 & h_2 & h_1 \\ 0 & 0 & 0 & 0 & 0 & h_4 & h_3 & h_2 \\ 0 & 0 & 0 & 0 & 0 & 0 & h_4 & h_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_4 \end{bmatrix}$$

Note: $h_{0\dots}h_4$ are each complex numbers, that is; h_0= Re(h_0) and Im(h_0)

The address register pointing to the memory that contains the complex-impulse response, $h_0 \dots h_4$, is set as modulo. This causes the pointer to wrap around before filling each column of the convolution matrix transpose when it reaches the last entry of the channel-impulse response in the array, "**channel_h**".

Expressions used in the assembly code shown in Code Listing 3 are defined as:

1. **r0**: **#channel_h** (Modulo 2)
2. **r1**: **#conv_matrix**
3. **r2**: holds the start location for each column in the **#conv_matrix**

Code Listing 3. Assembly Code Used to Fill the Convolution Matrix

```
       loopstart0
add_col
       doensh1 #CHANNEL_MEM                                    ; loop1 repeat
                                                                ; #CHANNEL_MEM times
       move.2f (r0)+,d0:d1                                     ; d0=Re[channel_h(1)]
                                                                ; and
                                                                ; d1=Im[channel_h(1)]
loopstart1
       move.2f (r0)+,d0:d1      moves.2f d0:d1,(r1)+n1         ; d0=Re[channel_h(1)]
                                                                ; d1=Im[channel_h(1]]
                                                                ; move d0 and d1 (r1)
                                                                ;  r1->r1+4*n1
       loopend1
       moves.2f d0:d1,(r1)+n1
       adda n2,r2                                              ;  r2->r2+$24
       tfra r2,r1                                              ;  r1->r2=next
                                                                ;  diagonal element
       loopend0
```

4.3.2 Matrix Multiply Block

The matrix multiply block generates the matrix product $\mathbf{H}^{*} \times \mathbf{H}$. This block is designed to perform this operation efficiently. The key idea to simplify this implementation is that, given a matrix **M**, if read row-wise, the result is **M** and if read column-wise, the result is $\mathbf{M}^{\mathbf{T}}$. The operation that is needed to be performed in this application is $\mathbf{H}^{*} \times \mathbf{H}$. Recall that $\mathbf{H}^{\mathbf{T}}$ was generated instead of **H**. This was done primarily to assist in this matrix multiply block. Now, looking at the problem in hand, if $\mathbf{H}^{\mathbf{T}}$ is read row-wise, the result is $\mathbf{H}^{\mathbf{T}}$ and if $\mathbf{H}^{\mathbf{T}}$ is read column-wise, the result is $(\mathbf{H}^{\mathbf{T}})^{\mathbf{T}} = \mathbf{H}$.

As an example, to multiply two matrices of size $[(N_f+\nu) \times N_f] \times [N_f \times (N_f+\nu)]$, identified as Matrix1 and Matrix2 four address registers are used in the implementation as follows:

- One address register in modulo mode, pointing to Matrix1 (moving in the direction of ptrA in Equation 24 on page 13)
- One address register in linear mode, pointing to Matrix2 (moving in the direction of ptrB in Equation 24 on page 13)
- One address register in linear mode, for intermediate address calculation
- One address register in linear mode, pointing to the output matrix **“mult_out”**

The following steps are performed by the matrix multiply block to perform the matrix multiply operation.

1. N_f **mac** operations are performed between one row of Matrix1 and one column of Matrix2 to produce one row entry of the output Matrix. Two complex values are read at a time from each matrix using the **move.4f** instruction and four **mac** operations are performed in parallel.

2. N_f **mac** operations are performed between the same previous row of the Matrix1 and the next column of Matrix2. Note here that pointer to Matrix1 needs to be modulo to wrap around the same row. The pointer to Matrix2 needs to be linear to advance to the next column in Matrix2. This is shown in the following example, where it is also shown how the first row of the output is generated.

Matrix 1 **Matrix 2**

$$
\begin{array}{rcccl}
 & & & \begin{array}{cccc} M2_{C1} & M2_{C2} & & \\ \downarrow & \downarrow & & \end{array} & \\
M1_{R1} \rightarrow & \begin{bmatrix} a(1,1) & a(2,1) & a(3,1) & a(4,1) \\ a(2,1) & a(2,2) & a(2,3) & a(2,4) \\ a(3,1) & a(3,2) & a(3,3) & a(3,4) \\ a(4,1) & a(4,2) & a(4,3) & a(4,4) \end{bmatrix} & \times & \begin{bmatrix} b(1,1) & b(2,1) & b(3,1) & b(4,1) \\ b(2,1) & b(2,2) & b(2,3) & b(2,4) \\ b(3,1) & b(3,2) & b(3,3) & b(3,4) \\ b(4,1) & b(4,2) & b(4,3) & b(4,4) \end{bmatrix} & =
\end{array}
$$

$$
\begin{bmatrix} mac(M1_{R1}, M2_{C1}) & mac(M1_{R1}, M2_{C2}) & mac(M1_{R1}, M2_{C3}) & mac(M1_{R1}, M2_{C4}) \\ \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ \end{bmatrix}
$$

where:

$M1_{R1} = Matrix1_{Row1}$ $\qquad M2_{C1} = Matrix2_{Column1}$

3. The next step is to advance the pointer to the next row in the Matrix1 and re-define the modulo address register and repeat step 2. This generates the next row of output. The final matrix is filled row-wise. Before writing the data to memory, the output of the last **mac** operation is scaled by a right shift of (ceiling $\{\log_2$[# of summation]$\}$). Therefore the amount of right shift for our implementation is four (that is, **asrr #4,dn**, where **dn** holds the data that will be written to memory).

NOTE:

An offset greater than the modulo size cannot be added to the modulo register, for this reason another intermediate register (**r4**) is used which is updated linearly to point to the next row of Matrix1. Then the value of the intermediate register is transferred to the address register (**r0**) and its corresponding base resister (**r8**) using the assembly instruction **tfra r4,r0** and **tfra r4,r8.** This register update is done in the assembly code by the hardware loop called "**NN_loop**" as shown in the Code Listing 5 on page 14.

Eqn. 24

ptrB

ptrA

$$\text{Transpose(H)}=\begin{bmatrix} h_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ h_1 & h_0 & 0 & 0 & 0 & 0 & 0 & 0 \\ h_2 & h_1 & h_0 & 0 & 0 & 0 & 0 & 0 \\ h_3 & h_2 & h_1 & h_0 & 0 & 0 & 0 & 0 \\ h_4 & h_3 & h_2 & h_1 & h_0 & 0 & 0 & 0 \\ 0 & h_4 & h_3 & h_2 & h_1 & h_0 & 0 & 0 \\ 0 & 0 & h_4 & h_3 & h_2 & h_1 & h_0 & 0 \\ 0 & 0 & 0 & h_4 & h_3 & h_2 & h_1 & h_0 \\ 0 & 0 & 0 & 0 & h_4 & h_3 & h_2 & h_1 \\ 0 & 0 & 0 & 0 & 0 & h_4 & h_3 & h_2 \\ 0 & 0 & 0 & 0 & 0 & 0 & h_4 & h_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_4 \end{bmatrix}$$

Code Listing 4. Main Kernel for Matrix Multiply Block

```
    move.4f (r0)+,d0:d1:d2:d3move.4f (r1)+,d4:d5:d6:d7              ;  load real/imag of
                                                                    ;  transpose(H) and H
    loopstart2
N_loop:
    ;load real/imag of  transpose(H) and H
        [ mac d0,d4,d8                    mac d0,d5,d9
          mac d2,d6,d10                   mac d2,d7,d11
        ]
        [ mac d1,d5,d8                    mac -d1,d4,d9
          mac d3,d7,d10                   mac -d3,d6,d11
        ]
        [ add d8,d10,d12                  add d9,d11,d13
          move.4f (r0)+,d0:d1:d2:d3     move.4f (r1)+,d4:d5:d6:d7    ;  load real/imag of
        ]                                                           ;  transpose(H) and H
          loopend2
```

Code Listing 5. Address Register Updates for Matrix Multiply Block

```
move.l #mult_out,r3
       dosetup0 NN_loop                 dosetup1 P_loop
       dosetup2 N_loop
       move.l #conv_matrix,r0
       move.l #$000008,MCTL
       doen0 #COL_CONV_MATRIX           tfra r0,r1                     ;  r1->r1 and doen0
                                                                       ;  loop0 repeat # of col times
                                                                       ;  of convolution matrix
       tfra r0,r4                                                      ;  r4->r0
       move.l #conv_matrix,b0                                          ;  b0->#conv_matrix
       move.l #ROW_CONV_MATRIX_4X,m0                                   ;  Set modulo size of r0 to # of
                                                                       ;  rows in conv_matrix*4

NN_loop:
      loopstart0
      doen1 #COL_CONV_MATRIX                                           ;  loop1 repeat # of col in
                                                                       ;  conv_matrix

P_loop:
      loopstart1
    [ doen2 #ROW_CONV_MATRIX_over2      clr d8                         ;  loop2 repeat # of row/2, d8=0
      clr d9                            clr d10                        ;  d9=0, d10=0, d11=0
      clr d11
    ]
    move.4f (r0)+,d0:d1:d2:d3           move.4f (r1)+,d4:d5:d6:d7      ;  load real/imag of
                                                                       ;  transpose(H) and H
   loopstart2
N_loop:

                                        .
                                        .
                                        .
   loopend2
   [ asrr #CONV_MATRIX_SCALE,d12        asrr #CONV_MATRIX_SCALE,d13    ;  scale the multiply
      suba #8,r0                                                       ;  output
                                                                       ;  back up r0 by 8
    ]
    [ rnd d12,d12                       rnd d13,d13                    ;  round entries before storing
      suba #8,r1                                                       ;  back up r1 by 8
    ]

    [ moves.2f d12:d13,(r3)+                                           ;  write results to memory
      clr d8                            clr d9
      clr d10                           clr d11
    ]
      loopend1

      move #conv_matrix,r1              adda #ROW_CONV_MATRIX_4X,r4,r4 ;  r1->#conv_matrix
                                                                       ;  r4->r4+32
      tfra r4,r0                        tfra r4,r8                     ;  r0->r4 and r8
           loopend0
```

At the exit from this routine all the addressing modes are set back to linear.

In this matrix multiply block 1/SNR (SNR in linear scale) is also added to all the diagonal elements of the output matrix, **mult_out**. This block completes the generation of the matrix expressed in Equation 25.

$$\mathbf{H} \times \mathbf{H} + \frac{1}{\mathrm{SNR}}\mathbf{I} \qquad \textit{Eqn. 25}$$

4.4 Cholesky Factorization and Back Substitution

Cholesky factorization is the core routine for computing the DFE coefficient. This routine computes the Cholesky factorization of the matrix illustrated in Figure 3 on page 9. The input to the routine is $\mathbf{A} = \mathbf{H} \times \mathbf{H} + (1/\mathrm{SNR\_lin})\mathbf{I}$ and the output of the routine is the feed forward and feedback filter taps.

4.4.1 Cholesky Factorization

The first step in finding the feed forward and the feedback taps is the computation of the lower triangular Cholesky factor of matrix **A** (formed by replacing the lower triangle of the original matrix **A** by its Cholesky factor). This section describes how the Cholesky factorization is implemented in StarCore. To explain the process for computing the Cholesky factorization, two iterations of the algorithm are presented.

NOTE:

Iteration refers to the outer most loop. In the following context Iteration 1 refers to all the operation as long as column 1 is the base column, **“BASE_COL”**. Similarly Iteration 2 refers to all the operation when column 2 is the base column, **“BASE_COL”**.

Iteration 1:

1. Compute sqrt(a(1,1)) and 1/sqrt(a(1,1)).

   The first step in filling up the matrix is replacing a(1,1) by sqrt(a(1,1)). Then, each element of the first column of the marix **A** is multiplied by 1/sqrt(a(1,1)) and these new values replace the existing first column of matrix **A** in DSP memory. This process is shown by the lighter-shaded column which is called the “**BASE_COL”** shown in the Figure 4.

$$A=\begin{bmatrix}
a(1,1) & a(1,2) & a(1,3) & a(1,4) & a(1,5) & a(1,6) & a(1,7) & a(1,8) & a(1,9) & a(1,10) & a(1,11) & a(1,12)\\
a(2,1) & a(2,2) & a(2,3) & a(2,4) & a(2,5) & a(2,6) & a(2,7) & a(2,8) & a(2,9) & a(2,10) & a(2,11) & a(2,12)\\
a(3,1) & a(3,2) & a(3,3) & a(3,4) & a(3,5) & a(3,6) & a(3,7) & a(3,8) & a(3,9) & a(3,10) & a(3,11) & a(3,12)\\
a(4,1) & a(4,2) & a(4,3) & a(4,4) & a(4,5) & a(4,6) & a(4,7) & a(4,8) & a(4,9) & a(4,10) & a(4,11) & a(4,12)\\
a(5,1) & a(5,2) & a(5,3) & a(5,4) & a(5,5) & a(5,6) & a(5,7) & a(5,8) & a(5,9) & a(5,10) & a(5,1(1)) & a(5,12)\\
a(6,1) & a(6,2) & a(6,3) & a(6,4) & a(6,5) & a(6,6) & a(6,7) & a(6,8) & a(6,9) & a(6,10) & a(6,11) & a(6,12)\\
a(7,1) & a(7,2) & a(7,3) & a(7,4) & a(7,5) & a(7,6) & a(7,7) & a(7,8) & a(7,9) & a(7,10) & a(7,11) & a(7,12)\\
a(8,1) & a(8,2) & a(8,3) & a(8,4) & a(8,5) & a(8,6) & a(8,7) & a(8,8) & a(8,9) & a(8,10) & a(8,11) & a(8,12)\\
a(9,1) & a(9,2) & a(9,3) & a(9,4) & a(9,5) & a(9,6) & a(9,7) & a(9,8) & a(9,9) & a(9,10) & a(9,11) & a(9,12)\\
a(10,1) & a(10,2) & a(10,3) & a(10,4) & a(10,5) & a(10,6) & a(10,7) & a(10,8) & a(10,9) & a(10,10) & a(10,11) & a(10,12)\\
a(11,1) & a(11,2) & a(11,3) & a(11,4) & a(11,5) & a(11,6) & a(11,7) & a(11,8) & a(11,9) & a(11,10) & a(11,11) & a(11,12)\\
a(12,1) & a(12,2) & a(12,3) & a(12,4) & a(12,5) & a(12,6) & a(12,7) & a(12,8) & a(12,9) & a(12,10) & a(12,11) & a(12,12)
\end{bmatrix}$$

Figure 4. First Step in Computing the Cholesky Factor

2. Perform the operation:

   `A(i,j)=A(i,j)-A(i,k)*conj(A(j,k))`

   This implies replacing the second column (starting from the diagonal dash line) with the (first iteration; k=1, j=2, and i=2) newly computed values:

   `A(2:12,2)=A(2:12,2)-A(2:12,1)*conj(A(2,1))`

   The darker-filled column in the matrix in Figure 5, shows the newly replaced columns. In the assembly code, the start location of `A(2:12,2)`is stored in a register named, **CURRENT_COL,** and the start location of `A(2:12,1)` is stored in a register named, **BASE_ROW**.

   The **“BASE_ROW”** contains the starting row location in the **“BASE_COL”** as the algorithm proceeds.

$$
A = \begin{array}{r|cccccccccccc|}
 & a(1,1) & a(1,2) & a(1,3) & a(1,4) & a(1,5) & a(1,6) & a(1,7) & a(1,8) & a(1,9) & a(1,10) & a(1,11) & a(1,12) \\
\text{BASE\_ROW} \rightarrow & a(2,1) & a(2,2) & a(2,3) & a(2,4) & a(2,5) & a(2,6) & a(2,7) & a(2,8) & a(2,9) & a(2,10) & a(2,11) & a(2,12) \\
 & a(3,1) & a(3,2) & a(3,3) & a(3,4) & a(3,5) & a(3,6) & a(3,7) & a(3,8) & a(3,9) & a(3,10) & a(3,11) & a(3,12) \\
 & a(4,1) & a(4,2) & a(4,3) & a(4,4) & a(4,5) & a(4,6) & a(4,7) & a(4,8) & a(4,9) & a(4,10) & a(4,11) & a(4,12) \\
 & a(5,1) & a(5,2) & a(5,3) & a(5,4) & a(5,5) & a(5,6) & a(5,7) & a(5,8) & a(5,9) & a(5,10) & a(5,1(1)) & a(5,12) \\
 & a(6,1) & a(6,2) & a(6,3) & a(6,4) & a(6,5) & a(6,6) & a(6,7) & a(6,8) & a(6,9) & a(6,10) & a(6,11) & a(6,12) \\
 & a(7,1) & a(7,2) & a(7,3) & a(7,4) & a(7,5) & a(7,6) & a(7,7) & a(7,8) & a(7,9) & a(7,10) & a(7,11) & a(7,12) \\
 & a(8,1) & a(8,2) & a(8,3) & a(8,4) & a(8,5) & a(8,6) & a(8,7) & a(8,8) & a(8,9) & a(8,10) & a(8,11) & a(8,12) \\
 & a(9,1) & a(9,2) & a(9,3) & a(9,4) & a(9,5) & a(9,6) & a(9,7) & a(9,8) & a(9,9) & a(9,10) & a(9,11) & a(9,12) \\
 & a(10,1) & a(10,2) & a(10,3) & a(10,4) & a(10,5) & a(10,6) & a(10,7) & a(10,8) & a(10,9) & a(10,10) & a(10,11) & a(10,12) \\
 & a(11,1) & a(11,2) & a(11,3) & a(11,4) & a(11,5) & a(11,6) & a(11,7) & a(11,8) & a(11,9) & a(11,10) & a(11,11) & a(11,12) \\
 & a(12,1) & a(12,2) & a(12,3) & a(12,4) & a(12,5) & a(12,6) & a(12,7) & a(12,8) & a(12,9) & a(12,10) & a(12,11) & a(12,12)
\end{array}
$$

Figure 5. Second Step in Computing the Cholesky Factor

3. Perform the same set of operations as in step 2 between the first **“base column”** and the third column and this time replace the content of the third column with the results of the computation. This is represented in the matrix in Figure 6 on page 16 by the darker-filled column. This processes continues until all the columns in the 12 × 12 matrix have been replaced with the results of the computation (under the dashed-diagonal line).

Iteration 2: (In this step the second column becomes the **“BASE_COL”)**

1. Perform the same operation as step 2 of Iteration one between the second and third columns continuing until all the columns in the 12 × 12 matrix are replaced.

   This process is shown in Figure 7. Just as in step 1 of Iteration 1, the first objective is to compute 1/sqrt(a(2,2)) and sqrt(a(2,2)). The diagonal element is then replaced by sqrt(a(2,2)) and all the contents of the second column (the new **“base column”**) starting at the element below the diagonal line are multiplied by 1/sqrt(a(2,2)) and replaced in memory by the new results. This is the updated **“base column”** that will be used in the second iteration of the algorithm.

   These address changes are made in **loop_d** of the assembly code.

   After N_f –1 iteration the whole matrix (below the dashed-diagonal line) is replaced by its Cholesky factor. This process requires N^3 complex operations. Where N × N is the size of the square matrix.

$$
A = \begin{array}{r|cccccccccccc|}
 & a(1,1) & a(1,2) & a(1,3) & a(1,4) & a(1,5) & a(1,6) & a(1,7) & a(1,8) & a(1,9) & a(1,10) & a(1,11) & a(1,12) \\
 & a(2,1) & a(2,2) & a(2,3) & a(2,4) & a(2,5) & a(2,6) & a(2,7) & a(2,8) & a(2,9) & a(2,10) & a(2,11) & a(2,12) \\
\text{BASE\_ROW} \rightarrow & a(3,1) & a(3,2) & a(3,3) & a(3,4) & a(3,5) & a(3,6) & a(3,7) & a(3,8) & a(3,9) & a(3,10) & a(3,11) & a(3,12) \\
 & a(4,1) & a(4,2) & a(4,3) & a(4,4) & a(4,5) & a(4,6) & a(4,7) & a(4,8) & a(4,9) & a(4,10) & a(4,11) & a(4,12) \\
 & a(5,1) & a(5,2) & a(5,3) & a(5,4) & a(5,5) & a(5,6) & a(5,7) & a(5,8) & a(5,9) & a(5,10) & a(5,1(1)) & a(5,12) \\
 & a(6,1) & a(6,2) & a(6,3) & a(6,4) & a(6,5) & a(6,6) & a(6,7) & a(6,8) & a(6,9) & a(6,10) & a(6,11) & a(6,12) \\
 & a(7,1) & a(7,2) & a(7,3) & a(7,4) & a(7,5) & a(7,6) & a(7,7) & a(7,8) & a(7,9) & a(7,10) & a(7,11) & a(7,12) \\
 & a(8,1) & a(8,2) & a(8,3) & a(8,4) & a(8,5) & a(8,6) & a(8,7) & a(8,8) & a(8,9) & a(8,10) & a(8,11) & a(8,12) \\
 & a(9,1) & a(9,2) & a(9,3) & a(9,4) & a(9,5) & a(9,6) & a(9,7) & a(9,8) & a(9,9) & a(9,10) & a(9,11) & a(9,12) \\
 & a(10,1) & a(10,2) & a(10,3) & a(10,4) & a(10,5) & a(10,6) & a(10,7) & a(10,8) & a(10,9) & a(10,10) & a(10,11) & a(10,12) \\
 & a(11,1) & a(11,2) & a(11,3) & a(11,4) & a(11,5) & a(11,6) & a(11,7) & a(11,8) & a(11,9) & a(11,10) & a(11,11) & a(11,12) \\
 & a(12,1) & a(12,2) & a(12,3) & a(12,4) & a(12,5) & a(12,6) & a(12,7) & a(12,8) & a(12,9) & a(12,10) & a(12,11) & a(12,12)
\end{array}
$$

Figure 6. Third Step in Computing the Cholesky Factorization

NOTE:

In **loop_a** the **BASE_ROW** is moved to the next column, where as in **loop_c**, it is moved down by one row, without changing the column location. In the assembly code all loops and the address calculation have been highlighted.

$$A=\begin{bmatrix}
a(1,1) & a(1,2) & a(1,3) & a(1,4) & a(1,5) & a(1,6) & a(1,7) & a(1,8) & a(1,9) & a(1,10) & a(1,11) & a(1,12)\\
a(2,1) & a(2,2) & a(2,3) & a(2,4) & a(2,5) & a(2,6) & a(2,7) & a(2,8) & a(2,9) & a(2,10) & a(2,11) & a(2,12)\\
a(3,1) & a(3,2) & a(3,3) & a(3,4) & a(3,5) & a(3,6) & a(3,7) & a(3,8) & a(3,9) & a(3,10) & a(3,11) & a(3,12)\\
a(4,1) & a(4,2) & a(4,3) & a(4,4) & a(4,5) & a(4,6) & a(4,7) & a(4,8) & a(4,9) & a(4,10) & a(4,11) & a(4,12)\\
a(5,1) & a(5,2) & a(5,3) & a(5,4) & a(5,5) & a(5,6) & a(5,7) & a(5,8) & a(5,9) & a(5,10) & a(5,1(1)) & a(5,12)\\
a(6,1) & a(6,2) & a(6,3) & a(6,4) & a(6,5) & a(6,6) & a(6,7) & a(6,8) & a(6,9) & a(6,10) & a(6,11) & a(6,12)\\
a(7,1) & a(7,2) & a(7,3) & a(7,4) & a(7,5) & a(7,6) & a(7,7) & a(7,8) & a(7,9) & a(7,10) & a(7,11) & a(7,12)\\
a(8,1) & a(8,2) & a(8,3) & a(8,4) & a(8,5) & a(8,6) & a(8,7) & a(8,8) & a(8,9) & a(8,10) & a(8,11) & a(8,12)\\
a(9,1) & a(9,2) & a(9,3) & a(9,4) & a(9,5) & a(9,6) & a(9,7) & a(9,8) & a(9,9) & a(9,10) & a(9,11) & a(9,12)\\
a(10,1) & a(10,2) & a(10,3) & a(10,4) & a(10,5) & a(10,6) & a(10,7) & a(10,8) & a(10,9) & a(10,10) & a(10,11) & a(10,12)\\
a(11,1) & a(11,2) & a(11,3) & a(11,4) & a(11,5) & a(11,6) & a(11,7) & a(11,8) & a(11,9) & a(11,10) & a(11,11) & a(11,12)\\
a(12,1) & a(12,2) & a(12,3) & a(12,4) & a(12,5) & a(12,6) & a(12,7) & a(12,8) & a(12,9) & a(12,10) & a(12,11) & a(12,12)
\end{bmatrix}$$

Figure 7. First Step of Iteration 2

Code Listing 6. Cholesky

```
        move.l #mult_out,r0                                          ; r0->#mult_out, d8=11 (counter)
        move.l #d_array,r5                                           ;  r5->#d_array
        move.w #ROW_COL_MULT_MATRIX_SUB1,d8                          ;  d8=counter for outermost loop loop_a
        dosetup0 loop_a           doen0 #ROW_COL_MULT_MATRIX_SUB1    ;  repeat loop_a Matrix size-1 times
        move.l #ROW_COL_MULT_MATRIX_ADD1,n1   tfra r0,CURRENT_COL    ;  n1=Matrix size+1,
                                                                     ;  CURRENT_COL->r0
        tfra r0,BASE_ROW          move.l #ROW_COL_MULT_MATRIX,n0     ;  BASE_RW->r0 and n0->size Matrix
        move.f (r0),d15                                              ;  Read element A(1,1) from memory
        loopstart                                                    ;  start for loop_a
loop_a:                                                              ;  loop_a  k=1:Nf+nu-1
         dosetup1 loop_b
         tfr d15,d0               doen1 d8                           ; d0=real(diagonal) , loop_b 1:Nf+nu-1

         jsr sqrt_and_invsqrt                                        ;  d0=sqrt(real(diagonal),
         clr d1
                                                                     ;  d4=1/sqrt(real(A(1,1))
       [ move.f d0,(r5)+          moves.2f d0:d1,(r0)+n0             ;  d_array(i)=sqrt(real(diagonal)
         clb d4,d0                                                   ;  r0->a(i,i)=sqrt(Re(diagonal)),
       ]                                                             ;  r0->r0+#ROW_COL_MULT_MATRIX($30),
                                                                     ;  d0=scale for
                                                                     ;fromsqrt_and_invsqrtroutine
        asrr d0,d4                                                   ;  repeat loop_b for l=k+1:n
                                                                     ;  d4 = d4*scale
        loopstart1
loop_b:                                                              ;  start for loop (loop_b)
        ;---------------------------------------------------------------------------;
        ;This loop will multiply each entry of k th col by 1/sqrt(A(k,k)) & will replace each ;
        ;entry of the k th col by the results                                        ;
        ;---------------------------------------------------------------------------;
move.2f (r0),d2:d3                                                   ;  d2=Re[(A(k+1:n,k)]
                                                                     ;  d3=Im[A(k+1:n,k)]
         mpy d4,d2,d2             mpy d4,d3,d3                       ;  d2=Re[A(k+1:n,k)/A(k,k)]
                                                                     ;  d3=Im[A(k+1:n,k)/A(k,k)]
         asll d0,d2               asll d0,d3                         ;  scale d2 and d3 by scale for
                                                                     ;  1/sqrt(A(k,k))
         moves.2f d2:d3,(r0)+n0                                      ;  move d2 to memory (real result)
                                                                     ;  move d3 to memory (imag result)
                                                                     ;  pointer r0->r0+$30, point to
                                                                     ;  next element in the column
         loopend1                                                    ;  end loop_b
;****End loop_b*****
end_loop_b:

         tfra BASE_ROW,r0                                            ;  r0->prevoius diagonal address
         adda #N_1,r0,r0          dosetup1 loop_c                    ;  r0->r0+$34 address for current
                                                                     ;  diagonal
         tfra r0,r1               tfra r0,CURRENT_COL                ;  save loaction of the current
                                                                     ;  diagonal element in CURRENT_COL
       [ suba #4,r1              tfr d8,d9
         dosetup2 loop_d                                             ;  r1->(CURRENT_COL-$4)=
       ]                                                             ;  previous col=BASE_COL
                                                                     ;  d9=d8 (counter)
         tfra r1,BASE_COL   doen1 d9                                 ;  BASE_COL->r1
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

         move.2f (r1),d0:d1                                  ;  for j=k+1:n  d0=real(A(j,k)) d1=imag(A(j,k))
       [ move.2f (r1),d4:d5      tfra r0,r4
         clr d6 clr d7                                               ;  d4=Re([A(j:n,k)] &
                                                                     ;  d5=Im(A(j:n)
       ]                                                             ;  d6=0 and d7=0
                                                                     ;  r4->r0
       [ mac d0,d4,d6             mac d0,d5,d7
       ]
       [ mac d1,d5,d6             mac -d1,d4,d7
move.2f (r0),d2:d3                                                   ;  d6 & d7 = Re & Im
       ]
        ;  [A(i,k)*conj(A(j,k))]
       [ sub d6,d2,d10            sub d7,d3,d11

       ]                                                             ;  d10 & d11 = Re & Im [A(i,j)]
       tfr d10,d15
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                                                                     ;  base col.), loop_c for l=k+1:n
         loopstart1
loop_c:
         doen2 d9   move.2f (r1),d0:d1                     ;  for j=k+1:n  d0=real(A(j,k)) d1=imag(A(j,k))
       [ move.2f (r1)+n0,d4:d5    tfra r0,r4
         clr d6 clr d7

                                                                     ;  d5=Im(A(j:n,k))
       ]                                                             ;  d6=0 and d7=0
                                                                     ;  r4->r0

         mac d0,d4,d6             mac d0,d5,d7
       [ mac d1,d5,d6             mac -d1,d4,d7
         move.2f (r1)+n0,d4:d5    move.2f (r0)+n0,d2:d3              ;  d6 & d7 = Re & Im
       ]
                                                                     ;  [A(i,k)*conj(A(j,k))]
       [ sub d6,d2,d10            sub d7,d3,d11
```

```
        move.2f (r0)+n0,d2:d3        clr d6 clr d7
       ]                                                            ;  d10 & d11 = Re & Im [A(i,j)]



        loopstart2
loop_d:

       ;---------------------------------------------------------------------------
                                                             ;  d4=real(A(j:n,k)) &
                                                             ;  d5=imag(A(j:n,k))
       rnd d10,d10               rnd d11,d11                 ;  d6=0 d7=0
      [ mac d0,d4,d6              mac d0,d5,d7
        moves.2f d10:d11,(r4)+n0                              ;  d6 & d7 = Re & Im
      ]
        mac d1,d5,d6              mac -d1,d4,d7               ;  [A(i,k)*conj(A(j,k))]
                                                              ;  d10=Re(A(i,j))
                                                              ;  d11=Im(A(i,j))
                                                              ;  move d10,d11 in to memory
      [ sub d6,d2,d10            sub d7,d3,d11
        move.2f (r0)+n0,d2:d3    move.2f (r1)+n0,d4:d5
        clr d6 clr d7;r0->r0+$30 and r1->r1+$30
      ]

       loopend2;  end of for loop j
;        mac d1,d5,d6                mac -d1,d4,d7                  ;  NOTE Taking -d1
                                                                    ;  implements the conj operation
;*****End loop_d
;       moves.2f d10:d11,(r4)+n0
end_loop_d:

        adda #N_1,CURRENT_COL,CURRENT_COL  adda #N_1,CURRENT_COL,r0
                                                                    ;  CURRENT_COL=CURRENT_COL+$34
        adda #N_0,BASE_COL,r1                                       adda #N_0,BASE_COL,BASE_COL;
k_COL=BASE_COL+$30
                                                                    ;  r1->BASE_COL , r0->CURRENT_COL

        loopend1
;****End loop_c******
end_loop_c:

        sub #1,d8  adda #N_1,BASE_ROW,r0      adda #N_1,BASE_ROW,BASE_ROW
                                                                    ;  r0->r0+$34,BASE_ROW->
                                                                    ;  BASE_ROW+$34
                                                                    ;  d8=d8-1 (counter) r0->BASE_ROW
                                                                    ;  (r0 points to the next col. and
                                                                    ;  BASE_ROW contains the start
                                                                    ;  address for that col.
        loopend0                                                    ;  end loop_a
;****End loop_a
```

4.4.2 Back Substitution Algorithm

Back substitution is implemented as part of the Cholesky factorization routine. This routine takes the result of the Cholesky factored matrix and performs a back substitution solving a set of linear equations to get the final solutions for the optimum feed forward and feedback taps. This section explains how the back substitution was implemented on the SC140 by making a comparison of the assembly implementation to its corresponding Matlab implementation presented in Code Listing 2 on page 8.

LOOP 1 (Refers to LOOP 1 of Matlab code in Code Listing 2 on page 8)

1. The column corresponding to the maximum value in the diagonal of Matrix **A** (the lower triangular Cholesky factor) is located.

   This gives the location of the optimum feedback filter coefficients and its index gives the optimum decision delay. Locating the index with the maximum diagonal value in the Cholesky factored matrix is done by the assembly routine **find_max.asm** and will be explained Section 4.4.3, “Finding the Optimum Delay.”

   In the Matlab code, the monic matrix **L** (**L_matrix)** of the **LDL*** Cholesky factor is generated by multiplying each column of matrix **A** by the reciprocal of the diagonal entry for that column. In the assembly routine, a new array called **L_matrix** is formed in the same way as described above but only a subset of the matrix **L** is generated in the DSP implementation. To clarify this point further, Figure 8 on page 20 depicts an example which shows the elements of matrix **A** that are used in forming the **L_matrix** (here the optimum delay is assumed to be 8) and the direction of the arrow shows how the contents of **L_matrix** are arranged in memory. For this example the **L_matrix** array is filled with elements l(8,7), l(7,6), l(8,6), l(6,5), l(7,5), l(8,5)…where l(i,j)=a(i,j)/a(j,j).

$$A=\begin{bmatrix}
a(1,1) & a(1,2) & a(1,3) & a(1,4) & a(1,5) & a(1,6) & a(1,7) & a(1,8) & a(1,9) & a(1,10) & a(1,11) & a(1,12)\\
a(2,1) & a(2,2) & a(2,3) & a(2,4) & a(2,5) & a(2,6) & a(2,7) & a(2,8) & a(2,9) & a(2,10) & a(2,11) & a(2,12)\\
a(3,1) & a(3,2) & a(3,3) & a(3,4) & a(3,5) & a(3,6) & a(3,7) & a(3,8) & a(3,9) & a(3,10) & a(3,11) & a(3,12)\\
a(4,1) & a(4,2) & a(4,3) & a(4,4) & a(4,5) & a(4,6) & a(4,7) & a(4,8) & a(4,9) & a(4,10) & a(4,11) & a(4,12)\\
a(5,1) & a(5,2) & a(5,3) & a(5,4) & a(5,5) & a(5,6) & a(5,7) & a(5,8) & a(5,9) & a(5,10) & a(5,1(1)) & a(5,12)\\
a(6,1) & a(6,2) & a(6,3) & a(6,4) & a(6,5) & a(6,6) & a(6,7) & a(6,8) & a(6,9) & a(6,10) & a(6,11) & a(6,12)\\
a(7,1) & a(7,2) & a(7,3) & a(7,4) & a(7,5) & a(7,6) & a(7,7) & a(7,8) & a(7,9) & a(7,10) & a(7,11) & a(7,12)\\
a(8,1) & a(8,2) & a(8,3) & a(8,4) & a(8,5) & a(8,6) & a(8,7) & a(8,8) & a(8,9) & a(8,10) & a(8,11) & a(8,12)\\
a(9,1) & a(9,2) & a(9,3) & a(9,4) & a(9,5) & a(9,6) & a(9,7) & a(9,8) & a(9,9) & a(9,10) & a(9,11) & a(9,12)\\
a(10,1) & a(10,2) & a(10,3) & a(10,4) & a(10,5) & a(10,6) & a(10,7) & a(10,8) & a(10,9) & a(10,10) & a(10,11) & a(10,12)\\
a(11,1) & a(11,2) & a(11,3) & a(11,4) & a(11,5) & a(11,6) & a(11,7) & a(11,8) & a(11,9) & a(11,10) & a(11,11) & a(11,12)\\
a(12,1) & a(12,2) & a(12,3) & a(12,4) & a(12,5) & a(12,6) & a(12,7) & a(12,8) & a(12,9) & a(12,10) & a(12,11) & a(12,12)
\end{bmatrix}$$

Figure 8. Generating L_Matrix for Back Substitution

2. The entries of **L_matrix** are processed and the results are stored in an intermediate array, **v** (same notation as the Matlab code).

   One very important point to mention here is that, in the matlab code, the array labeled **v** is filled from the back (that is; the first element computed is the last entry in **v**). In the DSP routine, **v** array is filled in the opposite way, that is, the first value computed is the first element in array **v**. This way, at the end of filling the array **v**, the pointer points to the last element of the array **v**. Filling the array in this direction is helpful because the next loop, the loop that computes **w_opt,** can read the entries from array **v** by simply decrementing the address register by one each time.

Code Listing 7. DSP Code that Generates the L_Matrix

```
;**************************************************************************

        loopstart0                                  ;  forloop for moving pointer to
                                                    ;  previous col. in the cholesky
                                                    ;  matrix
loop_address_setting:
        move.f (r0),d1   doen1 d8                   ;  d1=diagonal elenemt, d8=counter

        jsr one_over_x                              ;  d0=1/diagonal element, d2=scale
                                                    ;  factor


        loopstart1                                  ;  start for loop to fill up
                                                    ;  L_matrix
loop_compute:
        adda n0,r0                                  ;  r0->next element in the col.
        move.2f (r0),d4:d5                          ;  d4=real and d5=image od the
                                                    ;  elements down the col.
        mpy d4,d0,d6     mpy d5,d0,d7               ;  d6=real(1/diagonal)*elements in
                                                    ;  the same col.
                                                    ;  d7=imag(1/diagonal)*elements in
                                                    ;  the same col.
        asrr d2,d6       asrr d2,d7                 ;  scale d6 and d7 by the devide
                                                    ;  routine scale factor

        asrr #2,d6       asrr #2,d7                 ;  scale the entry of L_matrix
                                                    ;  before saving to memory to
                                                    ;  prevent overflow

        moves.2f d6:d7,(r1)+                        ;  write results to L_matrix
        loopend1                                    ;  end forloop L_matrix
;End loop_compute

        add #1,d8        suba n1,BASE_ROW           ;  d8=d8+1,  BASE_ROW=BASE_ROW-$34
        tfra BASE_ROW,r0 ;  r0->BASE_ROW
        loopend0                                    ;  end forloop for moving pointer
                                                    ;  to previous col. in the cholesky
                                                    ;  matrix
;****End loop_address_setting
```

LOOP 2

1. In this part of the algorithm, data is read from two different locations in memory:
   a) "**v**" (starting at the location of the last entry in **v**)
   b) "**channel_h**" (starting at the first element of the impulse response)

   The **v** matrix is arranged in Matlab as shown in Equation 26 (for our example with, an optimum delay = 8 and chan_mem = 4).

$$\mathbf{V} = \begin{bmatrix} v(8) & v(7) & v(6) & v(5) & v(4) & v(3) & v(2) & v(1) \end{bmatrix} \quad \textit{Eqn. 26}$$

During the execution of LOOP 2 (following the same notation as the Matlab routine) data is read out as shown in Table 2. Note that the number of elements read in each iteration depends on the min(chan_mem, jj)+1, where jj is decremented by one from a value of N_f every iteration.

Table 2. Read Out of Loop 2 Data

Iteration					
First	v(8)	v(7)	v(6)	v(5)	v(4)
Second	v(7)	v(6)	v(5)	v(4)	v(3)
Third	v(6)	v(5)	v(4)	v(3)	v(2)
Fourth	v(5)	v(4)	v(3)	v(2)	v(1)
Fifth	v(4)	v(3)	v(2)	v(1)	

Table 2. Read Out of Loop 2 Data (Continued)

Iteration					
Sixth	v(3)	v(2)	v(1)		
Seventh	v(2)	v(1)			
Eighth	v(1)				

However, in the DSP code this process is done slightly different. Data is arranged in the **v** array in the DSP as shown in Equation 27. As the algorithm progresses, data is read out every iteration as shown in Table 3.

$$\mathbf{V} = \begin{bmatrix} 0 & 0 & 0 & 0 & v(1) & v(2) & v(3) & v(4) & v(5) & v(6) & v(7) & v(8) \end{bmatrix} \qquad \textit{Eqn. 27}$$

Table 3. Read Out of Loop 2 Data in the SC140

Iteration					
First	v(8)	v(7)	v(6)	v(4)	v(5)
Second	v(7)	v(6)	v(5)	v(4)	v(3)
Third	v(6)	v(5)	v(4)	v(3)	v(2)
Forth	v(5)	v(4)	v(3)	v(2)	v(1)
Fifth	v(4)	v(3)	v(2)	v(1)	0
Sixth	v(3)	v(2)	v(1)	0	0
Seventh	v(2)	v(1)	0	0	0
Eight	v(1)	0	0	0	0

Stuffing four zeros (eight complex elements for this particular example) makes the code easier to write and more optimal, because this eliminates the need for address manipulation that is required to implement min(chan_mem,jj)+1.

Code Listing 8. Back Substitution

```
      move.l #L_matrix,r1                                   ;  r1->L_matrix
      move.l #v,r2                                          ;  r2->v
      move.f #.25,d0         clr d1                         ;  d0=0.25 first entry in v, d1=0
      moves.2f d0:d1,(r2)                                   ;  move first complex result to v(1)
      move.l n3,d0           move.w #$1,d12                 ;  n3=max_delay_index-1
                                                            ;  (d12 = counter for jj_loop)

      dosetup0 kk_loop       tfra r2,BASE_ROW               ;  r2->BASE_ROW
      doen0 d0               dosetup1 jj_loop
                                                            ;  kk_loop for j=1:max_delay_index-1

      loopstart0

kk_loop
      doen1 d12

      [ move.2f (r1)+,d0:d1  move.2f (r2)-,d2:d3
        clr d4 clr d5
      ]
      loopstart1

jj_loop

cc
      mac -d0,d2,d4          mac -d0,d3,d5
      mac d1,d3,d4           mac -d1,d2,d5
      asll #4,d4             asll #4,d5
      move.2f (r1)+,d0:d1    move.2f (r2)-,d2:d3
                                                            ;  remove the scaling by 4
                                                            ;  (Note: all the elements were
                                                            ;  scaled by 1/4, therefore the
                                                            ;  product was scaled by 1/16)
      add d4,d6,d6           add d5,d7,d7
      clr d4 clr d5
      loopend1

      [ asrr #2,d6            asrr #2,d7
        adda #$4,BASE_ROW    adda #$4,BASE_ROW,r2
      ]                                                     ;  move pointer to next address to
                                                            ;  fill up the new value of v
ox2

      add #1,d12             suba #4,r1      moves.2f d6:d7,(r2)    clr d6  clr d7
      loopend0
;-----------------------------------------------------------------------------
      ;NOTE: at the end of kk_loop "start_location" contains the location for
      ;the last element in v
;-----------------------------------------------------------------------------
```

4.4.3 Finding the Optimum Delay

This section describes the assembly routine to find the optimum delay and how to implement it into the SC140. The StarCore SC140 has a special instruction, **max2**, which helps implement this type of array search very efficiently. The **max2** instruction finds the maximum number between two sets of 16-bit numbers. However, this assembly routine requires the size of the input array to be in multiples of eight. In our example there are 12 diagonal values. To resolve this problem, **d_array**, which contains all the diagonal values, is first zero padded to make the total size of the array equal to 16. The **max2** instruction then finds the maximum number between two sets of 16-bit numbers. The following description briefly explains how this routine works:

The vector **d_array** has $N_f + \nu$ elements indexed: 0,1,2,...,$(N_f + \nu + 4) - 1$.

The vector is divided into eight sets:

set #0 contains elements with index - 0,8,16,...

set #1 contains elements with index - 1,9,17,...

set #2 contains elements with index - 2,10,18,...

o
o
o

set #7 contains elements with index - 7,15,23,...

In the first loop, **Loop_1**, the local maximum value in each of the eight sets is found.

The index of each local maximum is not known at the end of the kernel.

Maximum of set #0 is at d4.l, maximum of set #1 is at d4.h,

Maximum of set #2 is at d5.l, maximum of set #3 is at d5.h,

Maximum of set #4 is at d6.l, maximum of set #5 is at d6.h,

Maximum of set #6 is at d7.l, maximum of set #7 is at d7.h.

In the second loop, **Loop_2**, the global maximum is found from the local maxima. The set where this global maxima exists is then located.

Once the set that contains the global maximum is known (it has only $(N_f+\nu+4)/8$ elements), all that remains is to scan the set and locate the index of the element that is equal to the global maximum. This is accomplished in the third loop, **Loop_3**.

Code Listing 9. Locating the Optimum Delay (find_max.asm)

```
section .data local
       global Ftemp
N       equ     16
      align 4
DUMMY  dc  0,0,0,0,0,0

Ftemp   ds 3*4
        endsec


        section .text local
        global find_max_delay
find_max_delay type func

       move.w #d_array,r0           move.w #<2,n3
       move.w #d_array+8,r1         doensh3 #<(N/8)-2
       move.w #Ftemp,r2          move.2l (r0)+n3,d4:d5
       move.w #Ftemp+4,r3        move.2l (r1)+n3,d6:d7
       move.2l (r0)+n3,d0:d1     move.2l (r1)+n3,d2:d3


Loop1_1:

     loopstart3
     [ max2 d0,d4                    max2 d1,d5
       max2 d2,d6                    max2 d3,d7
       move.2l (r0)+n3,d0:d1         move.2l (r1)+n3,d2:d3
     ]
     loopend3

     [ max2 d0,d4                    max2 d1,d5
       max2 d2,d6                    max2 d3,d7
       move.w #d_array,d2               doensh3 #<5
     ]

      ; The starting address for the subset with maximum is saved in d3
     [
     sxt.w d4,d4 asrw d4,d0 tfr d2,d3 tfr d2,d1
     move.l d5,(r2)   move.2l d6:d7,(r3)
     ]

     [
     cmpgt d4,d0  max d0,d4  add #<2,d2
     move.w (r2)+,d0   move.w #<8,n3
     ]
loop_2:
     loopstart3
     [ cmpgt d4,d0       max d0,d4
       tfrt d2,d3 add #<2,d2 move.w (r2)+,d0
     ]
     loopend3

     [ cmpgt d4,d0       max d0,d4
       tfrt d2,d3 add #<2,d2  move.w #<16,d5
     ]

     [
     ift tfr d2,d3 add d2,d5,d4
     iff add d3,d5,d4
     ]
     sub d1,d3,d7 move.l d3,r1 move.l d4,r0
     asr d7,d7    doensh3 #<(N/8)-2

     ; The best index is saved in d7l
     tfr d7,d6 move.w (r1),d4  move.w (r0)+n3,d0
     ; The maximum value is saved in d4l
     cmpgt d4,d0 add #<8,d6 max d0,d4 move.w (r0)+n3,d0

Loop_3:
       loopstart3
      [ cmpgt d4,d0   max d0,d4
       tfrt d6,d7     add #<8,d6      move.w (r0)+n3,d0
      ]
        loopend3
       tfrt d6,d7
        move.l d7,n3    ;load the delay index in to n3
       rts
       endsec
```

4.4.4 Square Root and Inverse Square Root

The square root and inverse square root routine computes the sqrt(x) and 1/sqrt(x). First, 1/sqrt(x) is computed, multiplying this by x gives sqrt(x).

Computation of 1/sqrt(x) is made by iterative quadratic method as shown in Equation 28.

$$U_{i+1} = U_i/2*(3 - V*U_i*U_i) \qquad \textit{Eqn. 28}$$

After a few iterations, the results of the iterative quadratic method reaches a fairly good approximation of 1/sqrt(V). For the purpose of implementation Equation 28 is re-written as $0.5U_{i+1} = 0.25*U_i - V*U_i$^3. Intermediate results need to be scaled to ensure that the data going into the 16-bit multiplier yields good precession. The first step is to scale the input x by 2^x (where x must be even) so that it is a number less than one (but as close to one as possible). To obtain a scale factor (2^x) for the input, x is made even by an **“and”** operation of the form “**and #$0000fffe,d2,d2**”. This instruction ensures **d2** is even. The usefulness of this will become obvious in the following example where the relationship between the scale factor and the square root of the scale factor is expressed.

Let us assume that we want to compute the square root of x. Also, assume that a scale factor after performing a dynamic scaling on x (using the instruction **“clb”**) comes out to be 16 (2^4 or left shift by 4). After performing the left shift operation the number is $x \times 2^4$. Our goal is to compute sqrt(x) which is sqrt($x \times 2^4$)/sqrt(2^4). This is exactly equal to shifting sqrt($x \times 2^4$) right by two.

Our example shows that the square root of the scale factor can be obtained simply by dividing the shifts by two (that is; **asrr #1, d2**).

Assuming that the input is scaled by ***SC*** the result that is computed is expressed in Equation 29.

$$Ui = 1/\mathrm{sqrt}(V*\boldsymbol{SC}) = 1/[\mathrm{sqrt}(V)*\mathrm{sqrt}(\boldsymbol{SC})] \qquad \textit{Eqn. 29}$$

To find 1/sqrt(V), it is necessary to multiply the result 1/sqrt(V****SC***) by sqrt(***SC***). As scale factors ***SC*** are always chosen to be even shifts of two, the procedure mentioned in the example above can be used to compute sqrt(***SC***).

Code Listing 10. Routine for Computing Square Root Inverse Square Root

```
section .text local
      global   sqrt_and_invsqrt
sqrt_and_invsqrt type func
      push d1                 push d8                     ;  save d1 and d8
      clb d0,d2               tfr d0,d4                   ;  scale d0=v=input, scale factor
                                                          ;  in d4
                                                          ;  save v in d4
      and #$0000fffe,d2,d2                                ;  this will make the scale factor an
                                                          ;  even integer. e.g. if d2=3 the and
                                                          ;  operation will output d2=4
      asrr d2,d0                                          ;  scale d0=input by d2
      asrr #2,d0              move.f #.75,d1              ;  asrr #2 implememts factor 0.25,
                                                          ;  d1=0.75=Ui
      tfr d1,d3               dosetup3 loop1    doen3 #6  ;  d3=0.75=scale factor for Ui,
                                                          ;  loop1 for 1=1:6
      loopstart3                                          ;  start loop1
loop1
      clb d1,d10                                          ;  d10 = scale factor for Ui
      asrr d10,d1                                         ;  scale d1 by d10
     [ mpy d1,d1,d5mpy d1,d0,d6                           ;  d5=Ui^2, d6=0.25*Ui*V, d7=0.75*Ui,
      mpy d3,d1,d7            add d10,d10,d11             ;  d11=2*scale for Ui
     ]
      mpy d5,d6,d6            inc d10                     ;  d6=0.25*Ui^3*V, d10 = scale for Ui+1
      asll d11,d6                                         ;  scale back d6 by d11
      sub d6,d7,d1                                        ;  d1=0.75*Ui-0.25*V*Ui^3
      asll d10,d1             tfr d1,d9                   ;  scale back d1 by d10, d9=d1
      loopend3                                            ;  end loop1
      asrr #1,d2              mpy d9,d4,d0                ;  d2=sqrt(scale factor),
                                                          ;  d0=1/sqrt(V)*V =sqrt(V)
      asrr d2,d1              asll d10,d0
      asrr d2,d0              tfr d1,d4
      pop d8                  pop d1
      rts
      endsec
```

5 Results

Figure 9 – Figure 12 shows the Matlab and corresponding DSP results for the feedback and feed forward filter taps for an arbitrary channel-impulse response. Channel impulse response = [(-0.5251 -0.4487i) (0.0953 -0.2673i) (-0.2129 -0.0084i) (-0.3605 -0.2713i) (0.1874 -0.3487i)] and SNR = 20 dB.

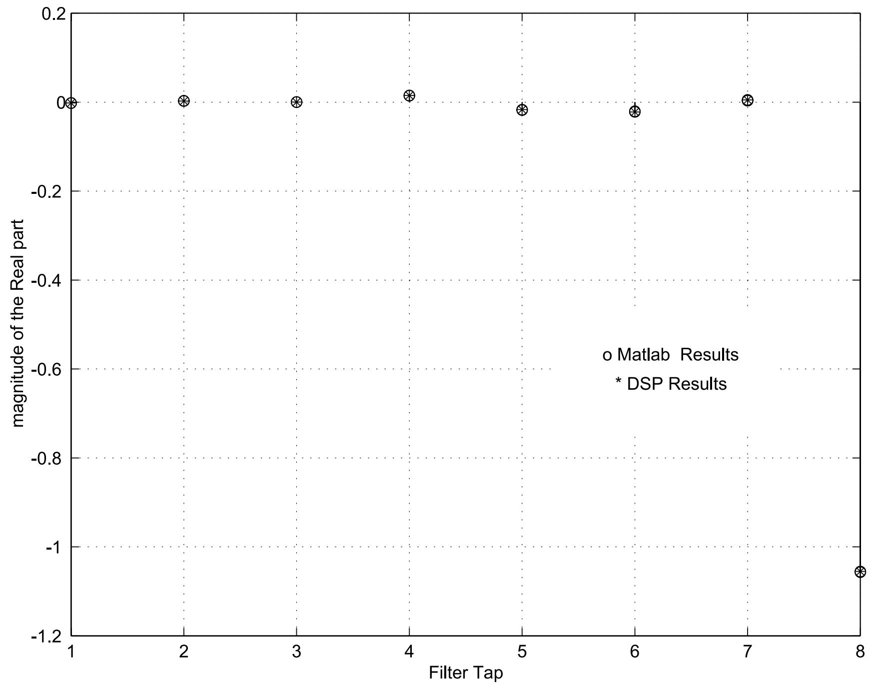

Figure 9. Real Part of the DFE Feed Forward Tap

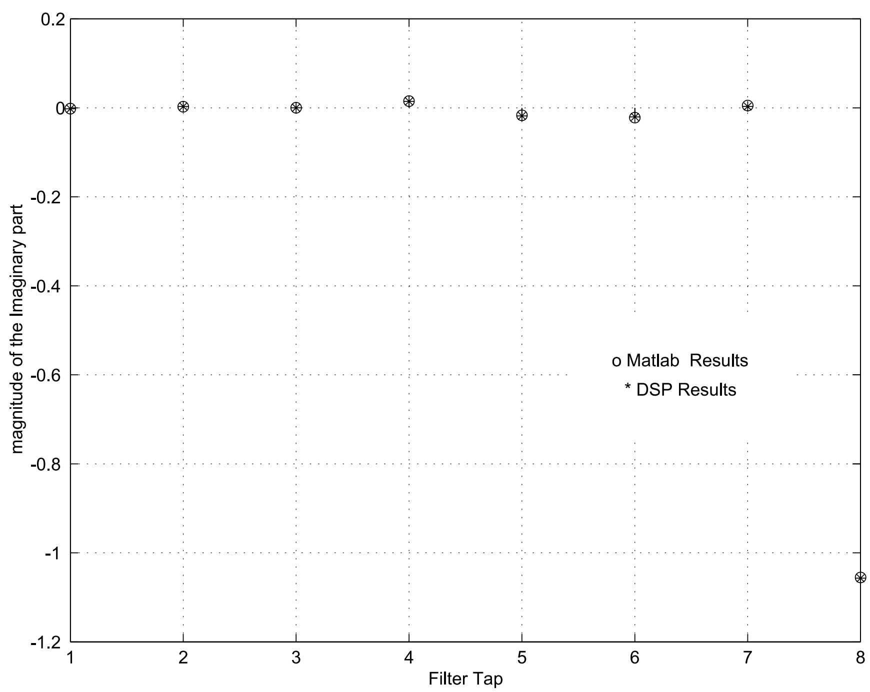

Figure 10. Imaginary Part of the DFE Feed Forward Tap

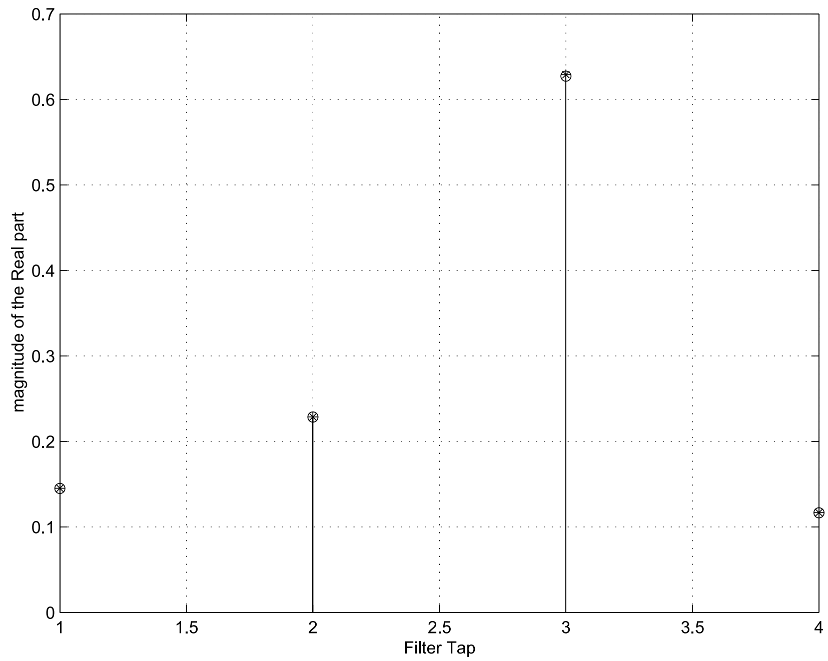

Figure 11. Real Part of the DFE Feedback Tap

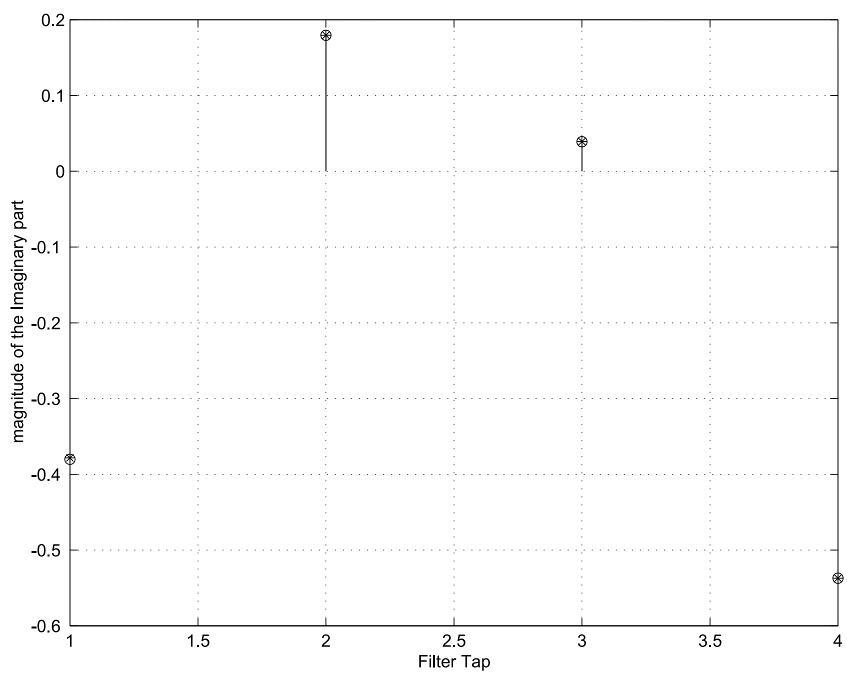

Figure 12. Imaginary Part of the DFE Feedback Tap

These results show that the DSP and the Matlab results agree within the allowable error range of 16-bit arithmetic. The “X” axis represents the filter taps and the “Y” axis represents the magnitude of each of the filter taps. It is important to note that the DSP feedback filter taps need to be conjugated before using them. The frequency response of the feedback and the feed forward filter obtained from the DSP implementation closely matches the theoretical Matlab result. Storing the intermediate result (that is, the element for which 1/sqrt() needs to be computed) in a data register instead of memory improves the accuracy of the algorithm to a great extent. This is primarily because storing small fractional numbers to memory (16-bit) increases the error, whereas keeping the result in a data register holds better precision as the data register uses 32-bit representation.

It is important to consider that because of error introduced in fixed-point calculation, the optimum delay may be more than one in some situations, but they will always appear next to each other. In this case picking any one of the columns of the L_matrix (feedback coefficient and the corresponding feed forward coefficient) that has the same optimum delay (that lies between $N_f - 1$ to $N_f - \nu$) will give good results in terms of BER of the overall communication system. Also, there are communication systems where it is safe to assume the optimal decision delay is $N_f - 1$. In such systems the computation can be further reduced by removing the routine that locates the optimal delay. The entire process of computing the DFE coefficient from the channel-impulse response required 7680 cycles for a DFE (8,4). In this implementation example the channel-impulse response length was assumed to be of length five.

6 Conclusion

Decision feedback equalizers are very useful as sub-optimal solutions when the constellation size is large and the channel memory is long. This situation is common in many communication systems, especially in many current and next generation communication systems. The fact that decision feedback equalizers are implemented as FIR filters makes them especially attractive for processors such as SC140, which have multiple ALU. This is primarily because of the fact that multi-sample programming can be used to implement FIR filters very efficiently. The DSP implementation of Cholesky based DFEs presented here is a way of finding the DFE coefficients using the concept of spectral factorization. This eliminates the need for computationally expensive complex-matrix inversion. Also, simulation results show that optimizing the decision delay improves the decision point SNR, which in turn improves BER performance of the communication receiver. The implementation presented here optimizes the decision delay. The results obtained from the real-time SC140 implementation are accurate within the precession of the 16-bit computation.

7 References

[1] *MMSE Decision-Feedback Equalizers: Finite-Length Results*, Al-Dhahir, N.; Cioffi, J.M. (Information Theory, IEEE Transactions on Volume: 41 4 , July 1995 , Page(s): 961 - 975)

[2] *Efficient computation of the delay-optimized finite length MMSE-DFE*, Al-Dhahir, N.; Cioffi, J.M, (Signal Processing, IEEE Transactions on Volume: 44 5, May 1996, Page(s): 1288 - 1292)