Reed Solomon Encoder/Decoder on the StarCore™ SC140/SC1400 Cores, With Extended Examples

By Jasmin Oz and Assaf Naor

This application note describes the implementation of the Reed-Solomon error-control codes on the StarCore™ SC140 DSP core. Reed-Solomon codes are the preferred error-control coding procedures in a wide range of applications, such as ADSL, digital cellular phones, storage devices, and deep-space communications. Their popularity originates from their strong capability to correct both random and burst errors.

The current trend for improving DSP-processing speed is to place multiple processor units on a single chip with an architecture that supports parallel execution. The StarCore SC140 family of DSPs exemplifies this trend. It has four data-arithmetic units (DALUs) and two address-generation units (AGUs). Code implementation for these processors should capitalize on their capabilities. This document describes the implementation of the Reed-Solomon encoder and decoder on the SC140 core. The document begins with a basic theoretical background on the Reed-Solomon algorithm and then discusses the implementation of the encoder and decoder. Little or no background on the subject is required.

CONTENTS

freescale™ semiconductor

1 Basics of Forward Error Correction (FEC)

In an ideal communication scheme, the information received is identical to the source transmission. However, in a typical real communication scheme, the information passes through a noisy communication channel to the receiver. The information received at the destination is likely to contain errors due to the channel noise. The acceptable level of transmitted signal corruption (error level) depends on the application. Voice communication, for example, is relatively error tolerant. However, the prospect of occasionally losing a digit in communications of financial data highlights the need for error-control mechanisms.

In 1948, C.E. Shannon proved in his classic paper [1] that a communications channel can be made arbitrarily reliable by encoding the information so that a fixed fraction of the channel is used for redundant information. In the years that followed, there was a rapid development in designing FEC schemes. Today, a variety of effective coding algorithms are widely used. FEC offers a number of benefits:

- Data integrity is critical in the design of most digital communication systems and all storage devices. Along with the design trend toward increasing bandwidths and data volumes, there is a drive to decrease the allowed error rates. FEC enables a system to achieve high data reliability.
- FEC yields low error rates and performance gains for systems in which other options, such as increasing the transmitted power or installing noise-limiting components, are too expensive or impractical.
- System costs may be reduced by eliminating an expensive or sensitive component and compensating for the lost performance by a suitable FEC scheme.

For an overview of FEC schemes, consult [2] and [3].

FEC adds carefully designed information to the transmitted data and uses this redundant information to reconstruct the potentially corrupted data. **Figure 1** depicts a typical communications scheme.

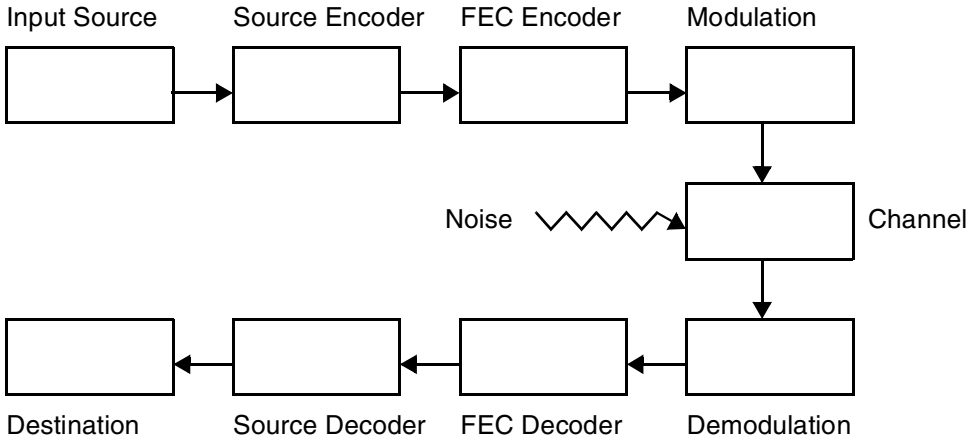


Figure 1. FEC Communications System

The two main type of error-control codes used in communications systems are as follows:

- *Convolutional codes*. Each bit depends on the current bit as well as on a number of previous bits. In this sense, the convolutional code has a memory. The most common scheme for decoding convolutional codes is the Viterbi algorithm.
- *Block codes*. A bitstream is divided into message blocks of fixed length called frames. The valid code-word block is formed from the message bitstream by adding a proper redundant part. Each code word is independent of the previous one, so the code is memory-less.

The Reed-Solomon codes are block codes. Unlike convolutional codes, Reed-Solomon codes operate on multi-bit symbols rather than on individual bits. The question of whether to choose convolutional codes or block codes depends on several variables. In low-speed, low-integrity applications, convolutional codes are the better choice, and block codes are suitable for high-speed, high-integrity applications. An example of an application suited to convolutional codes is a digitized voice communication in which a relatively high bit-error rate (about 10^{-3}) is acceptable. For blocks of machine-oriented data in which the desired bit-error rate ranges from 10^{-10} to 10^{-14}, block codes are the natural choice. Some applications use both convolutional and block codes. In such applications, concatenated codes result in strong performance by operating in two steps. The inner decoder, usually convolutional, reduces the bit-error rate to a medium-low level, and the outer decoder, usually a block type, reduces the bit-error rate further, to a very low level.

The errors introduced by the communications channel are classified into two main categories:

- *Random errors*. The bit-error probabilities are independent of each other. For example, thermal noise in communication channels typically causes random errors.
- *Burst errors*. The bit errors occur sequentially in time. Burst errors can be caused by such conditions as a fading communications channel or mechanical defects in a storage system.

When an FEC system is designed, the statistical nature of the noise environment must be considered, as well as the acceptable output bit-error rate. When the environment consists predominately of random errors, convolutional codes provide a low bit-error rate solution. However, when the environment has lower bit-error rates, long-length block codes often perform even better. In burst-error channels, Reed-Solomon codes are among the best codes because errors composed of many consecutive corrupted bits translate into only a few erroneous symbols.

2 Theory

The Reed-Solomon code was developed in 1960 by I. Reed and G. Solomon [4]. This code is an error detection and correction scheme based on the use of Galois field arithmetic. This section provides background information on binary and extended Galois fields and summarizes the essence of the Reed-Solomon codes. For details on Reed-Solomon codes, consult the literature, for example, [5] and [6].

2.1 Galois Fields

A number field has the following properties:

- Both an addition and a multiplication operation that satisfy the commutative, associative, and distributive laws.
- Closure, so that adding or multiplying elements always yields field elements as results.
- Both *zero* and *unity* elements. The zero element leaves an element unchanged under addition. The unity element leaves an element unchanged under multiplication.
- An additive/multiplicative inverse for each field element. The sole exception is the zero element, which has no multiplicative inverse.

Division is defined as the inverse of multiplication such that if $a \times b = c$, it follows that c divided by a yields b. An example of a number field is the set of real numbers together with the addition and multiplication operations. Galois fields differ from real number fields in that they have only a finite number of elements. Otherwise, they share all the properties common to number fields.

2.1.1 Binary Field, GF(2)

The simplest Galois field is GF(2). Its elements are the set {0,1} under modulo-2 algebra. Addition and subtraction in this algebra are both equivalent to the logical XOR operation. The addition and multiplication tables of GF(2) are shown in **Figure 2**.

Addition

+	0	1
0	0	1
1	1	0

Multiplication

x	0	1
0	0	0
1	0	1

Figure 2. Addition (XOR) and Multiplication Tables of GF(2)

There is a one-to-one correspondence between any binary number and a polynomial in that every binary number can be represented as a polynomial over *GF(2)*, and *vice versa*. A polynomial of degree *D* over *GF(2)* has the following general form:

$$f(x) = f_0 + f_1 x + f_2 x^2 + f_3 x^3 \ldots + f_D x^D$$

where the coefficients $f_0,\ldots, f_D$ are taken from *GF(2)*. A binary number of *(N+1)* bits can be represented as an abstract polynomial of degree *N* by taking the coefficients equal to the bits and the exponents of *x* equal to the bit locations.

For example, the binary number 100011101 is equivalent to the following polynomial:

$$100011101 \leftrightarrow 1 + x^2 + x^3 + x^4 + x^8$$

The bit at the zero position (the coefficient of x^0) is equal to 1, the bit at the first position (the coefficient of *x*) is equal to 0, the bit at the second position (the coefficient of x^2) is equal to 1, and so on. Operations on polynomials, such as addition, subtraction, multiplication and division, are performed in an analogous way to the real number field. The sole difference is that the operations on the coefficients are performed under modulo-2 algebra. For example, the multiplication of two polynomials is as follows:

$$(1 + x^2 + x^3 + x^4) \cdot (x^3 + x^5) = x^3 + x^5 + x^5 + x^6 + x^7 + x^7 + x^8 + x^9 = x^3 + x^6 + x^8 + x^9$$

This result differs from the result obtained over the real number field (the middle expression) due to the XOR operation (the + operation). The terms that appear an even number of times cancel out, so the coefficients of x^5 and x^7 are not present in the end result.

2.1.2 Extended Galois Fields GF(2^m)

A polynomial *p(x)* over GF(2) is defined as *irreducible* if it cannot be factored into non-zero polynomials over GF(2) of smaller degrees. It is further defined as *primitive* if $n = (x^n + 1)$ divided by *p(x)* and the smallest positive integer *n* equals $2^m - 1$, where *m* is the polynomial degree. An element of $GF(2^m)$ is defined as the root of a primitive polynomial *p(x)* of degree *m*. An element α is defined as primitive if

$$\alpha^{i \bmod (2^{m-1})}$$

where $i \in N$, can produce 2^{m-1} field elements (excluding the zero element). In general, extended Galois fields of class $GF(2^m)$ possess 2^m elements, where m is the symbol size, that is, the size of an element, in bits. For example, in ADSL systems, the Galois field is $GF(256)$. It is generated by the following primitive polynomial:

$$1+x^2+x^3+x^4+x^8$$

This is a degree-eight irreducible polynomial. The field elements are degree-seven polynomials. Due to the one-to-one mapping that exists between polynomials over $GF(2)$ and binary numbers, the field elements are representable as binary numbers of eight bits each, that is, as bytes. In $GF(2^m)$ fields, all elements besides the zero element can be represented in two alternative ways:

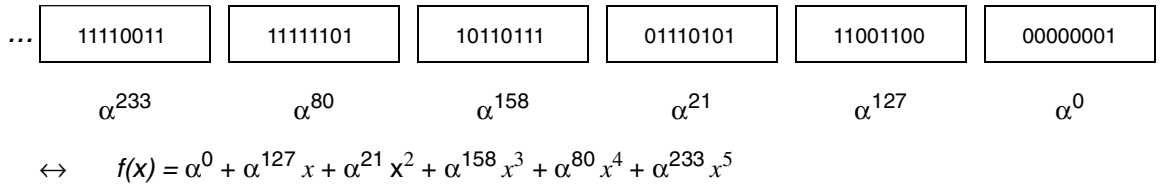
1. In binary form, as an ordinary binary number.
2. In exponential form, as α^p. It follows from these definitions that the exponent p is an integer ranging from 0 to (2^m-2). Conventionally, the primitive element is chosen as 0x02, in binary representation.

As for $GF(2)$, addition over $GF(2^m)$ is the bitwise XOR of two elements. Galois multiplication is performed in two steps: multiplying the two operands represented as polynomials and taking the remainder of the division by the primitive polynomial, all over $GF(2)$. Alternatively, multiplication can be performed by adding the exponents of the two operands. The exponent of the product is the sum of exponents, modulo 2^m-1.

Polynomials over the Galois field are of cardinal importance in the Reed-Solomon algorithm. The mapping between bitstreams and polynomials for $GF(2^m)$ is analogous to that of $GF(2)$. A polynomial of degree D over $GF(2^m)$ has the most general form:

$$f(x) = f_0 + f_1 x + f_2 x^2 + f_3 x^3 \ldots + f_D x^D$$

where the coefficients $f_0 - f_D$ are elements of $GF(2^m)$. A bitstream of $(N+1)m$ bits is mapped into an abstract polynomial of degree N by setting the coefficients equal to the symbol values and the exponents of x equal to the bit locations. The Galois field is $GF(256)$, so the bitstream is divided into symbols of eight consecutive bits each. The first symbol in the bitstream is 00000001. In exponential representation, 00000001 becomes α^0. Thus, α^0 becomes the coefficient of x^0. The second symbol is 11001100, so the coefficient of x is α^{127} and so on.



The elements are conventionally arranged in a log table so that the index equals the exponent, and the entry equals the element in its binary form. **Table 1** displays the log table for ADSL systems.

Table 1. Exponential-to-Binary Table for ADSL Systems

p	α*p*
0	0x01
1	0x02
2	0x04
3	0x08
4	0x10

Table 1. Exponential-to-Binary Table for ADSL Systems

p	αp
5	0x20
6	0x40
7	0x80
8	0x1D
9	0x3A
10	0x74
...	...
253	0x47
254	0x8E

The zero element does not appear in the table since it deserves special attention (see **Section 4.3**, *Look-up Tables*). Although multiplication is a complicated operation when performed bitwise, it is very simple if the exponential representation is used. The converse is true for addition. Therefore, two types of look-up tables are useful: a log table as shown in **Table 1** and an anti-log table that translates from binary to exponential representation.

2.2 Reed-Solomon Codes

Reed-Solomon codes are encoded and decoded within the general framework of algebraic coding theory. The main principle of algebraic coding theory is to map bitstreams into abstract polynomials on which a series of mathematical operations is performed. Reed-Solomon coding is, in essence, manipulations on polynomials over $GF(2^m)$. A block consists of information symbols and added redundant symbols. The total number of symbols is the fixed number $2^m - 1$. The two important code parameters are the symbol size m and the upper bound, T, on correctable symbols within a block. T also determines the code rate, since the number of information symbols within a block is the total number of symbols, minus $2T$. Denoting the number of errors with an unknown location as n_{errors} and the number of errors with known locations as $n_{erasures}$, the Reed-Solomon algorithm *guarantees* to correct a block, provided that the following is true: $2n_{errors} + n_{erasures} \leq 2T$, where T is configurable. The current implementation does not deal with erasures, and this document considers only error correction.

2.2.1 Encoding

When the encoder receives an information sequence, it creates encoded blocks consisting of $N = 2^m - 1$ symbols each. The encoder divides the information sequence into message blocks of $K \equiv N - 2T$ symbols. Each message block is equivalent to a message polynomial of degree $K-1$, denoted as $m(x)$. In systematic encoding, the encoded block is formed by simply appending $2T$ redundant symbols to the end of the K-symbols long-message block, as shown in **Figure 3**. The redundant symbols are also called parity-check symbols.

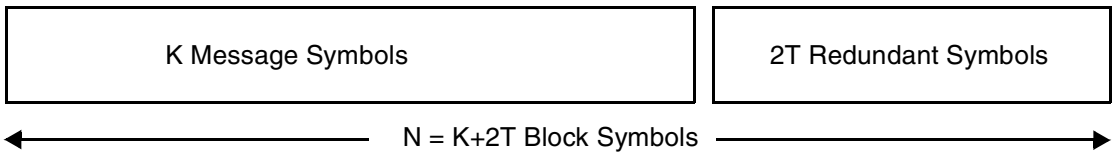


Figure 3. Block Structure

The redundant symbols are obtained from the redundant polynomial *p(x),* which is the remainder obtained by dividing $x^{2T}m(x)$ by the generator polynomial *g(x):*

$$p(x) = (x^{2T}m(x))mod g(x)$$

where is the generator polynomial. We choose the most frequently used generating polynomial:

$$g(x) = (x+\alpha^{p}1)(x+\alpha^{p}2)(x+\alpha^{p}3)\ldots(x+\alpha^{p}2T)$$

$$g(x) = (x+\alpha)(x+\alpha^{2})(x+\alpha^{3})\ldots(x+\alpha^{2T})$$

The code-word polynomial c*(x)* is defined as follows:

$$c(x) = x^{2T}m(x) + p(x)$$

Since in $GF(2^m)$ algebra, plus (+), and minus (–) are the same, the code word actually equals the polynomial $x^{2T}m(x)$ minus its remainder under division by *g(x).* It follows that c*(x)* is a multiple of g*(x).* Since there is a total of 2^{mK} different possible messages, there are 2^{mK} different valid code words at the encoder output. This set of 2^{mK} code words of length *N* is called an *(N,K) block code*.

2.2.2 Decoding

When a received block is input to the decoder for processing, the decoder first verifies whether this block appears in the dictionary of valid code words. If it does not, errors must have occurred during transmission. This part of the decoder processing is called *error detection.* The parameters necessary to reconstruct the original encoded block are available to the decoder. If errors are detected, the decoder attempts a reconstruction. This is called *error correction*. Conventionally, decoding is performed by the Petersen-Gorenstein-Zierler (PGZ) algorithm, which consists of four parts:

1. Syndromes calculation.
2. Derivation of the error-location polynomial.
3. Roots search.
4. Derivation of error values.

The error-location polynomial in this implementation is found using the Berlekamp-Massey algorithm, and the error values are obtained by the Forney algorithm. The four decoding parts are briefly outlined as follows:

1. *Syndromes calculation*: From the received block, the received polynomial is reconstructed, denoted as *c(x).* The received polynomial is the superposition of the correct code word *c(x)* and an error polynomial *e(x):*

   $$r(x) = c(x) + e(x)$$

   The error polynomial is given in its most general form by:

   $$e(x) = e_{i_0}x^{i_0} + e_{i_1}x^{i_1} + e_{i_2}x^{i_2}\ldots, e_{i_{(v-1)}}x^{i_{(v-1)}}$$

where i_0, i_1 and so on denote the error location indices, and ν the actual number of errors that have occurred. The *2T* syndromes are obtained by evaluating the received polynomial *r(x)* at the *2T* field points:

$$\alpha, \alpha^2, \alpha^3 \ldots, \alpha^{2T}$$

Since *c(x)* is a multiple of *g(x)*, it has the following general form:

$$c(x) = q(x)g(x)$$

where *q(x)* is a message-dependent polynomial. It follows from the definition of *g(x)* that the following field points:

$$\alpha, \alpha^2, \alpha^3 \ldots, \alpha^{2T}$$

are roots of *g(x)*. Hence *c(x)* vanishes at the *2T* points and the syndromes:

$$S_1, S_2, S_3 \ldots, S_{2T}$$

contain only of the part consisting of the error polynomial *e(x)*:

$$\begin{aligned} S_1 &= e(\alpha) \\ S_2 &= e(\alpha^2) \\ S_3 &= e(\alpha^3) \\ &\ldots \\ S_{2T} &= e(\alpha^{2T}) \end{aligned}$$

If all *2T* syndromes vanish, *e(x)* is either identically zero, indicating that no errors have occurred during the transmission, or an undetectable error pattern has occurred. If one or more syndromes are non-zero, errors have been detected. The next steps of the decoder are to retrieve the error locations and the error values from the syndromes. Denoting the actual number of errors as ν, α^{i_k} as X_k and the error values e^{i_k} as Y_k, the *2T* syndromes $S_1 - S_{2T}$ can then be expressed as follows:

$$\begin{aligned} S_1 &= Y_1X_1 + Y_2X_2 + Y_3X_3 \ldots Y_\nu X_\nu \\ S_2 &= Y_1(X_1)^2 + Y_2(X_2)^2 + Y_3(X_3)^2 \ldots Y_\nu(X_\nu)^2 \\ S_3 &= Y_1(X_1)^3 + Y_2(X_2)^3 + Y_3(X_3)^3 \ldots Y_\nu(X_\nu)^3 \\ &\ldots \\ S_{2T} &= Y_1(X_1)^{2T} + Y_2(X_2)^{2T} + Y_3(X_3)^{2T} \ldots Y_\nu(X_\nu)^{2T} \end{aligned}$$

Thus, there are *2T* equations to solve that are linear in the error values Y_k and non-linear in the error locations X_k.

2. *Derivation of the error-location polynomial*: The output of the Berlekamp-Massey algorithm is the error-location polynomial $\Lambda(x)$, defined as:

$$\Lambda(x) = (1 + xX_1)(1 + xX_2)(1 + xX_3)\ldots(1 + xX_\nu) \equiv 1 + \lambda_1 x + \lambda_2 x^2 + \lambda_3 x^3 + \ldots \lambda_\nu x^\nu$$

$\Lambda(x)$ has at most ν different roots. The inverses of the roots have the form α^{i_k}, where i_k is the error-location index. It can be proven [6] that the so-called Newton identity holds for the coefficients of $\Lambda(x)$ and the syndromes:

$$S_{j+\nu} + \lambda_1 S_{j+\nu-1} + \lambda_2 S_{j+\nu-2} \ldots \lambda_\nu S_j = 0$$

The Berlekamp-Massey algorithm is an iterative way to find a minimum-degree polynomial that satisfies the Newton identities for any j. If the degree of $\Lambda(x)$ obtained by the Berlekamp-Massey algorithm exceeds T, this indicates that more than T errors have occurred and the block is therefore not correctable. In this case, the decoder *detects* the occurrence of errors in the block, but no further attempt of correction is made, and the decoding procedure stops at this point for this block.

3. *Roots search*: The roots of the error-location polynomial are obtained by an exhaustive search, that is, by evaluating $\Lambda(x)$ at all Galois field elements, checking for possible zero results. The exponents of the inverses of the roots are equal to the error-location indices.

   If the number of roots is less than the degree of $\Lambda(x)$, more than T errors have occurred. In this case, errors are *detected*, but they are not corrected, and decoding stops at this point for this block.

4. *Derivation of error values*: The error values are obtained from the Forney algorithm in this implementation. Once the error locations X_k are found, the error values Y_k are found from the ν first syndromes equation by solving the following:

$$\begin{bmatrix} X_1 & X_2 & \ldots & X_\nu \\ \ldots & \ldots & \ldots & \ldots \\ X_1^\nu & X_2^\nu & \ldots & X_\nu^\nu \end{bmatrix} \begin{bmatrix} Y_1 \\ \ldots \\ Y_\nu \end{bmatrix} = - \begin{bmatrix} S_1 \\ \ldots \\ S_\nu \end{bmatrix}$$

The Forney algorithm is an efficient way to invert the matrix and solve for the errors values $Y_1 \_ Y_k$.

2.3 Error-Correcting Performance of Reed-Solomon Codes

The Reed-Solomon code ensures error *detection* and *correction* as long as the number of errors is at most T. If more than T errors occur, one of two things happen:

- An uncorrectable error is detected. No attempts are made by the decoder to correct the block.
- The received block accidentally resembles a valid code word different from the correct one and the decoder decodes the block into an incorrect code word.

Under the assumption of completely random bit errors, the bit-error rate P_b is related to the symbol-error rate P_s by the following: $P_s = 1 - (1 - P_b)^m$ On the other hand, if the symbol errors are independent, the probability of more than T errors occurring in a block is given by:

$$P_{E>T} = 1 - \sum_{i=0}^{T} \binom{N}{i} (P_s)^i (1 - P_s)^{N-i}$$

An alternative way to interpret $P_{E>T}$ is as the ratio of uncorrectable encoded blocks to the total number of received blocks, as the latter tends to infinity. A decoding error happens when an uncorrectable error is not recognized as such, and the whole block is decoded into another valid code word. The probability P_m of a miscorrection is bounded by

$$P_m \leq \frac{1}{T!} P_{E>T}$$

In typical applications with $m = 8$, the miscorrection rate P_m is about five orders of magnitude less than $P_{E>T}$. The curves in **Figure 4**, below, depict the probability of uncorrectable error for a block length N fixed to 255 and T varying from 1 to 8, as a function of the channel bit-error rate.

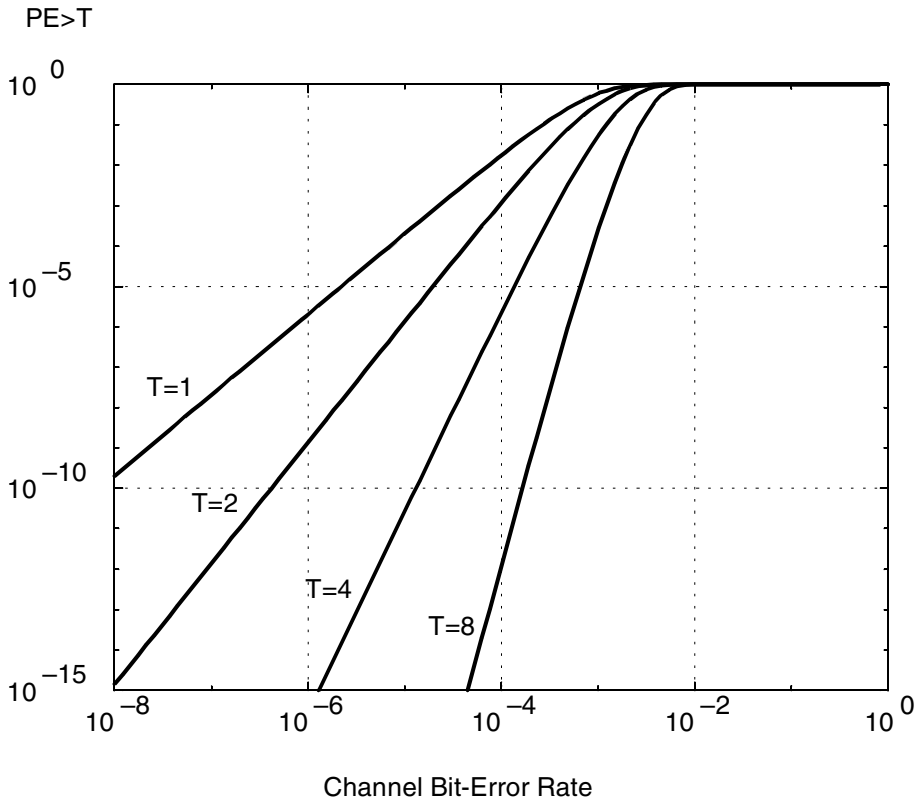


Figure 4. Probability of Uncorrectable Error Versus Bit-Error Rate

Notice that for bit-error rates below 10^{-2}, the curve exhibits a very steep slope. This is characteristic for good codes. It implies that the chances of encountering an uncorrectable error decrease drastically with only a moderately improved bit-error rate.

3 SC140 Core Overview

The SC140 core (see **Figure 5**) is a programmable high-performance DSP core that uses parallelism to execute multiple instructions in one clock cycle, running currently at 275 MHz and, eventually, at 400 MHz.

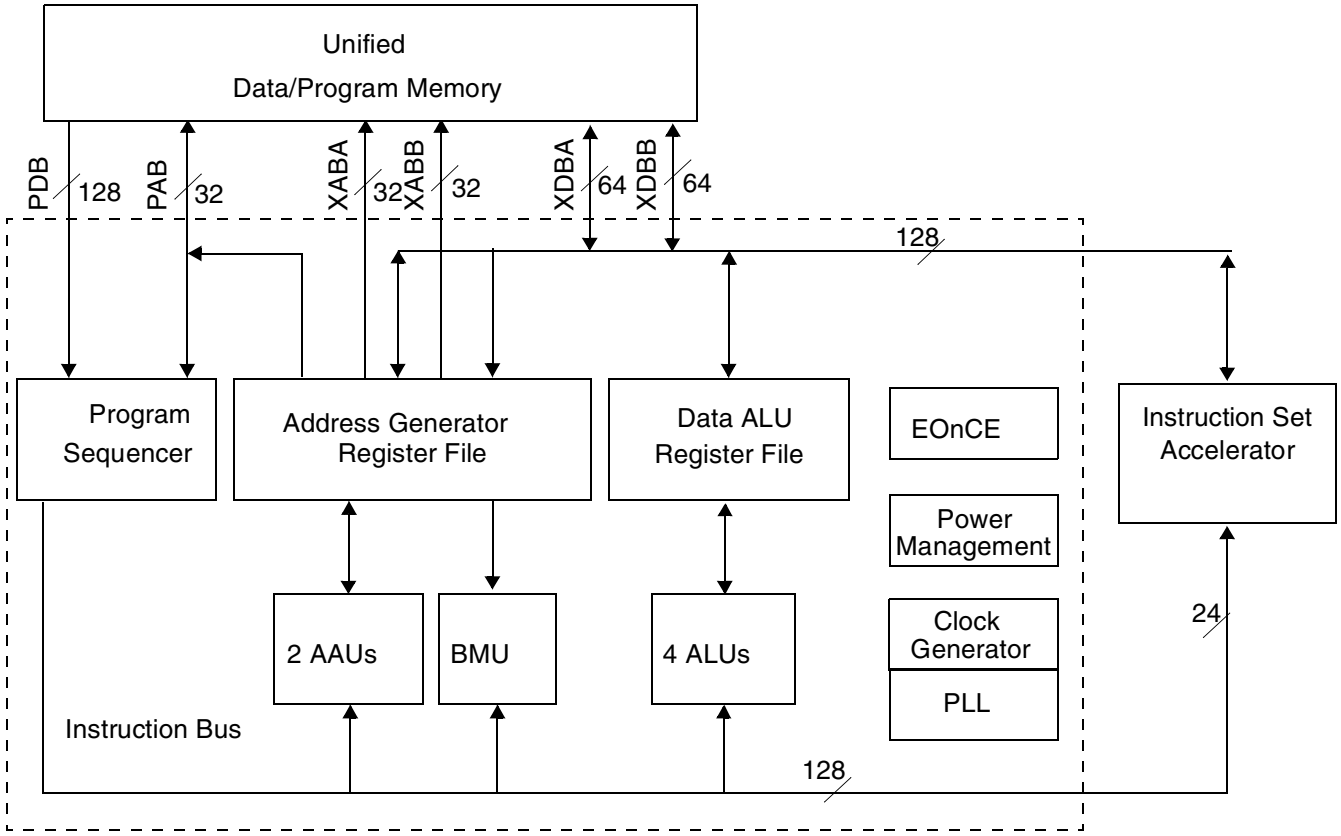


Figure 5. SC140 Core

The SC140 core provides the following main functional units:

- Data-arithmetic and logic unit (DALU) that includes four data-arithmetic and logic units (ALU) and a bank of sixteen 40-bit registers, d0 to 15.
- Address-generation unit (AGU):
  - :Sixteen 32-bit address read/write registers, r0 to r15. The contents of an address register can either point directly to data or function as an index.
  - Two AAUs, each of which can update one address register during one instruction cycle.
- Program sequencer and controller (PSEQ).
- Memory interface:
  - A 32-bit program memory address bus (PAB) and a 128-bit program memory data bus (PDB).
  - Two data memory buses (32-bit address and 64-bit data bus pairs: XABA and XDBA, XABB, and XDBB).

The SC140 core uses a unified memory space. Each address can contain either program information or data. There are no separate memory spaces for program locations and data locations. The memory is made up of a number of 32 KB groups, and each group includes eight 4 KB modules. Memory contention, which causes a one-cycle stall, can arise if two data accesses are to two different rows in the same memory module. The instruction set supports various types of move instructions that differ in access width (byte, word, long word, two long words), data type (signed, unsigned) and multiple-register moves. Integer moves from memory (byte, word, long, two long) are right aligned in the destination register, and by default are sign-extended to the left. Unsigned moves are marked with

"U" (for example, MOVEU.B), and zero extended to the destination register. **Figure 6** shows a schematic representation of some integer moves from memory to a register, used in the current implementation. The SC140 core can execute six instructions concurrently: up to *four* DALU instructions and up to *two* AGU instructions. The instructions are grouped together in an execution set and dispatched in parallel to the execution units. Chapter 6 in ref. [7] contains an overview of the instruction set, and in particular, the instruction set restrictions. Also, refer to **Section A**, *C-Codes for Decoder,* on page 20, for details on the assembly commands. For software development, StarCore offers the SC100 C compiler, assembler, simulator, and linker. The first three tools are employed in this implementation.

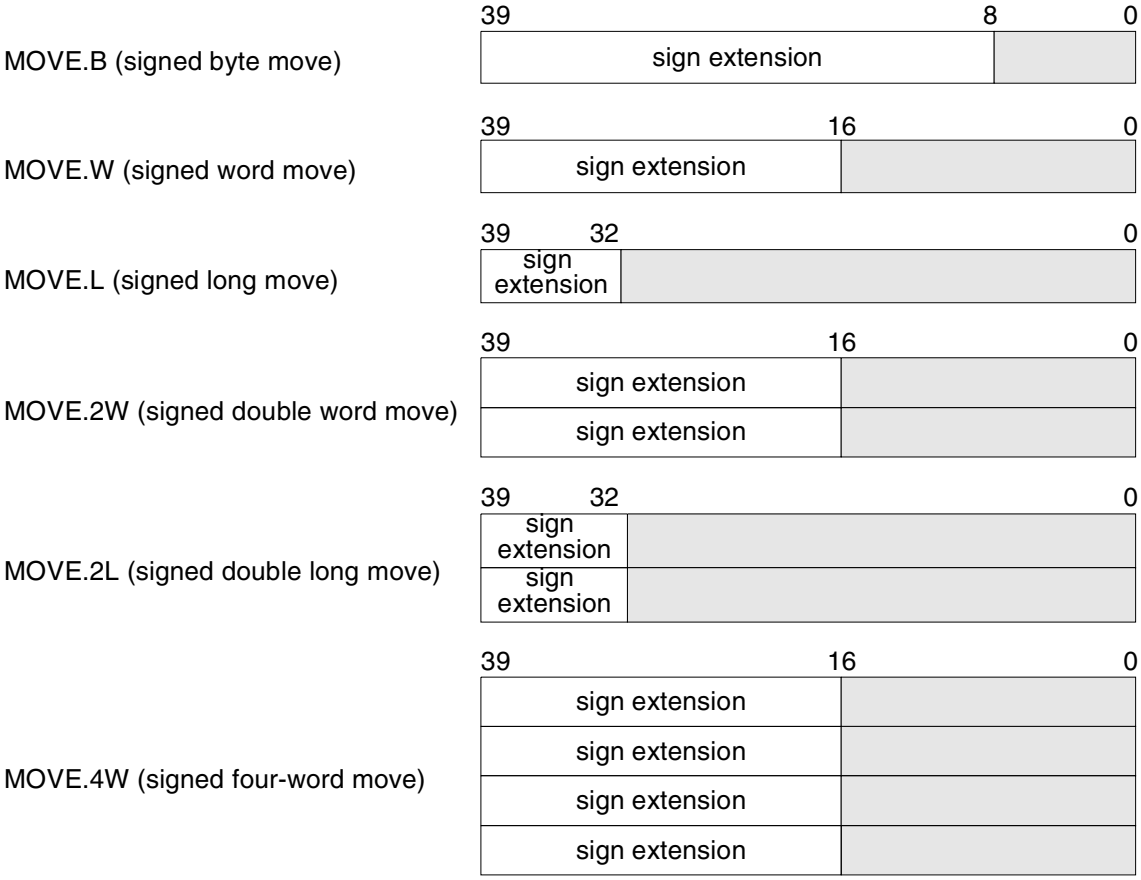


Figure 6. Integer Move Instructions

4 Implementation on the SC140 Core

The current application is in accordance with the standard for ADSL systems given in ref. [8]. The Galois field is *GF(256)* and the primitive polynomial is $1+x^2+x^3+x^4+x^8$. The block size *N* equals 255 bytes and the parameter *T* varies from 1 to 8. When the Reed-Solomon algorithm is implemented on a DSP, all routines, except for the Berlekamp-Massey algorithm, must perform polynomial evaluation in one way or another. Since most cycles of the application are spent on polynomial evaluation on a given set of field points, the major effort focused on implementing this operation as efficiently as possible under reasonable memory constraints. This section analyzes that implementation process, which used the following tools:

- Enterprise StarCore C Compiler.
- Assembler.
- Simulator.

The encoding/decoding process for which the cycle count is to be determined is summarized as follows. The encoder receives a message, a Reed-Solomon-compliant block of $K = 239$ bytes, and it produces an encoded block of 255 bytes. The encoded block is transmitted through the channel to a receiver. The received block is passed to the decoder where the four different stages of decoding are performed, as outlined in **Section 2.2**, *Reed-Solomon Codes,* on page 6. First-order estimates for the cycle counts required for this encoding/decoding process are as follows (see, for example, [5] and [6]):

1. Encoding routine $\propto 2NT$ cycles.
2. Syndromes calculation $\propto 2NT$ cycles.
3. Berlekamp-Massey algorithm $\propto T^2$ cycles.
4. Search of roots $\propto NT$ cycles.
5. Forney algorithm $\propto T^2$ cycles.

These are only initial estimates that do not account for any potential parallel processing. However, it clearly shows that encoding, syndromes calculation, and roots search consume the most cycles in the Reed-Solomon algorithm. The actual cycle counts depend both on the architecture and on the degree to which each routine can be separately implemented in a parallelized fashion. The decoder output is one of the following:

- For all-zero syndromes, the received block is identified as error-free and the program terminates.
- If the degree of the error-location polynomial exceeds T, or if the number of roots is not equal to the degree of the error-location polynomial, the received block contains more than T erroneous symbols. A flag is raised to indicate that errors are detected but are uncorrectable and the program terminates.
- In every other case, the reconstructed encoded block is returned.

4.1 Polynomial Evaluation Over GF(256)

Evaluation of a polynomial $f(x)$ of degree D at field point α^p has the most general form:

$$f(\alpha^p) = f_0 + f_1\alpha^p + f_2\alpha^{2p} + f_3\alpha^{3p} \ldots\ldots f_D\alpha^{Dp}$$

This form shows that the polynomial evaluation consists of a sequence of MAC (multiply-accumulate) operations. In Reed-Solomon codes, a polynomial is typically evaluated at a set of points. For example, let us assume that we evaluate the polynomial $f(x)$ at field points α, α^2, α^3,...., α^M. This is conveniently represented in matrix form, as multiplication of an $M \times (D+1)$ matrix. The elements, α, are raised to the appropriate powers, are multiplied by a vector:

$$\begin{bmatrix} 1 & \alpha & \alpha^2 & \alpha^3 & \ldots & \alpha^D \\ 1 & \alpha^2 & (\alpha^2)^2 & (\alpha^2)^3 & \ldots & (\alpha^2)^D \\ 1 & \alpha^3 & (\alpha^3)^2 & (\alpha^3)^3 & \ldots & (\alpha^3)^D \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & \alpha^M & (\alpha^M)^2 & (\alpha^M)^3 & \ldots & (\alpha^M)^D \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \ldots \\ f_D \end{bmatrix} = \begin{bmatrix} f(\alpha) \\ f(\alpha^2) \\ f(\alpha^3) \\ \ldots \\ f(\alpha^M) \end{bmatrix}$$

Matrices of this form are called power matrices. Thus, polynomial evaluation over a set of field points is called matrix multiplication. The basic operation is an inner vector product of the vector by the matrix row vector, which is equivalent to a sequence of MAC instructions under Galois algebra. How to efficiently implement these MAC instructions is the subject of the next section.

4.2 MAC Instructions Over Galois Fields

The two alternative ways to support Galois arithmetic, namely the binary representation and the exponential representation, are introduced in **Section 2.1.2**, *Extended Galois Fields GF(2m)*. As noted there, addition is easy in a binary representation and multiplication is easy in an exponential representation. A series of MAC instructions over a Galois field is an alternating series of multiplications and additions. Difficulties are encountered in either representation.

A first approach is to stay within the framework of the binary representation and to create a multiplication table, indexed by the multiplication operands, with entries as the product. This solution is fast but requires a large memory. For *GF(256)*, the required table size is 64 KB, which is impractical for typical DSP memories. A second approach is the extreme opposite, requiring no memory at all. In this approach, multiplication is simply performed bitwise by carry-less multiplication of the two binary operands, followed by the division by the primitive polynomial over *GF(2)*. This method is slow and inefficient. A third method is to perform addition in binary representation and multiplication in exponential representation and to perform the conversions between the two representations with the aid of look-up tables. In this particular software implementation, we chose this third approach because it offers the most reasonable trade-off between execution speed and memory conservation.

4.3 Look-up Tables

For a look-up table implementation, the following three types of tables are used:

- A binary-to-exponential conversion table with the exponent as entry and the Galois number as index.
- An exponential-to-binary conversion table with the exponent as index and the Galois number as entry.
- A power matrix of the kind introduced in **Section 4.1**.

The zero element deserves particular attention since its exponent must be defined. A suitable exponent is attributed to the zero element on the basis of the laws that such an exponent must obey if two Galois numbers, at least one of them a zero, are multiplied. The exponent of the product of two Galois numbers is the sum of their individual exponents, modulo 255 (or modulo $2^m - 1$ for a general Galois field). However, if at least one of the factors is zero, the exponent of the product must be equal to the exponent of the zero element. Following is one way to implement Galois multiplication efficiently while taking care of the zero element:

1. Associate the exponent 511 (or $2^{m+1} - 1$ for a general Galois field) to the zero element.
2. Extend the basic exponential-to-binary table whose exponents range from 0 to 254 to a table whose exponents range from 0 to 510. To accomplish this step, replicate the exponential-to-binary table entries for exponents exceeding 255 and append a zero byte at index 511.

If both operands differ from zero, the sum of their exponents is less than 511 and is a valid index to the table. If one or more of the original operands is zero, the sum of exponents exceeds 511. Since the exponent of the end result must be 511, multiplication is correctly performed by taking the *minimum* between the sum of the exponents and 511. The tables use in this implementation are as follows:

- *bin_2_exp*. Binary-to-exponential table, 256 words long. Indices are the Galois numbers in binary form and entries are their corresponding exponents. The first entry is equal to 511.

- *exp_2_bin_extended*. Exponential-to-binary table, 511 bytes long. Indices are the exponents and entries are the corresponding Galois numbers in binary form. The last entry is equal to 0.
- *exp_table_for_syndrome*. Power matrix for polynomial evaluation, 16 × 256 words in size. It has the typical form presented in **Section 4.1**, *Polynomial Evaluation Over GF(256),* on page 13. *M* is at most 2T and is thus chosen to be 16.

4.4 Lowest Cycle Count Limit for Polynomial Evaluation

The most general polynomial evaluation has the form presented in **Section 4.1**, *Polynomial Evaluation Over GF(256),* on page 13. We assume that the entries of the input vector are represented as binary and the power matrix is stored in exponential form. For a vector of length *D*+1 and field points α, α^2, α^3,...., α^M, the C-code for polynomial evaluation is then given by the following example:

Example 1. C Code for Matrix Multiplication

```
for (i=0; i<M; i++)
{
acc = 0;
for (j=0; j<=D; j++)
  {
  x_power = bin_2_exp[vector[j]];
  y_power = exp_table_for_syndrome[i][j];
  power = MIN((x_power + y_power),2*N+1);
  acc ^= exp_2_bin_extended[power];
  }
  result[i] = acc;
}
```

For each field element, two table look-ups are needed for each term. The first table look-up converts the polynomial from binary to exponential form. To save cycles, this conversion is performed separately, prior to entering the polynomial evaluation routine. This binary-to-exponential conversion contributes a small overhead to the total cycle count of the routine. It is implemented in the following steps:

1. Get the current vector element in binary form.

   This requires one MOVEU.B (rx) instruction, where rx denotes a general AGU register.

2. Add this byte to the table basic address and transfer the resulting address into an AGU register.

   This requires a MOVE.L command.

3. Read the table entry via a MOVE.W (rx) command.

A maximum of two move instructions can execute in one cycle. Thus, assuming full parallelization, for every polynomial term, a minimum of two cycles is needed in the conversion routine.

If the degree of the polynomial is *D* and the number of field points is *M*, a gross estimate for the number of cycles, denoted as $C_{conversion}$ required, is given by $C_{conversion} \approx 2(D+1)$. The code shown in **Example 1** involves the import of data and table look-ups, which are both AGU-based operations. The execution sets therefore are filled by AGU instructions rather than by DALU instructions. Thus, as in the binary-to-exponential conversion routine, the number of AGU operations is the critical factor in determining the cycle count.

The two methods of choice to implement polynomial evaluation in assembly code are split-summation and multi-sampling. In the split-summation method, one term in the result vector is calculated in every iteration. The inner product of each row with the input vector is divided into four partial sums by loading four matrix and four vector

terms in the same cycle and performing the MAC operations. If the number of terms in the input vector does not divide by four, the input vector is *a priori* zero-appended to bring the number of elements to an integer multiple of four.

In the multi-sample method, all result vector terms are calculated simultaneously, step by step, in each iteration. This is performed by loading the matrix elements column-wise instead of row-wise. Zero-appending is done if needed. The inner loop of code **Example 1** is implemented by the split-summation method in the following steps:

1. Load four matrix and four input vector terms.
2. From the four pairs of exponents, get the four exponents of the products. Those exponents are the offsets from the base address of the exponential-to-binary conversion table.
3. Retrieve the four table entries at those four offsets and XOR them with the accumulator.

The theoretical minimal number of cycles needed to realize these steps is as follows:

1. Loading the matrix and vector elements requires two MOVE.2L instructions.
2. The table offsets are addresses of four bytes. After their calculation, they must be transferred into four AGU registers. This requires four MOVE.L instructions.
3. The four table entries are bytes that are separately accumulated using four MOVEU.B (rx) instructions.

Thus, the total cycle count for the inner loop is five cycles per polynomial term, for four field points. In other words, under full parallelization, *one MAC operation* consumes 1.25 cycles. The theoretical minimal cycle count C_{min} is given by:

$$C_{min} \approx 5M\left\lceil \frac{(D+1)}{4} \right\rceil$$

where $\lceil\ \rceil$ denotes rounding up to the nearest integer. If the input polynomial is represented in binary form, the theoretical minimal cycle count becomes:

$$C_{min} \approx 5M\left\lceil \frac{(D+1)}{4} \right\rceil + 2(D+1)$$

In addition to this basic cycle count, a small overhead is required in the actual implementation.

4.5 Cycle Count of the Reed-Solomon Routines

The theory enables estimation of the cycle count required to execute the C code for each routine. Following are summary estimates for each routine. The C code for the decoder routines is presented in *Appendix A* on page 20:

- *Encoding routine*. A series of two concatenated polynomial evaluations, with M equal to $2T$ and D equal to 255 or $2T–1$, respectively. Therefore, the encoder routine requires at least 680T + 544 cycles, not including overhead.
- *Syndromes calculation*. A simple polynomial evaluation, with M equal to $2T$ and D equal to 255. Therefore, the lower bound on the cycle count is $640T + 512$ cycles, not including overhead.
- *Berlekamp-Massey Algorithm*. This algorithm is highly serial and parallelism cannot be applied. The error-location polynomial $\Lambda(x)$ is calculated iteratively and bytewise. There are $2T$ iterations during which mostly MAC operations are performed. The exact number of MACs is data-dependent, but it can be approximated by $T + n_{errors}\,(2/T)$. In this implementation, optimization was performed on the compiled code. Since it is not possible to perform these MAC operations with inputs *a priori* in exponential form, an order of 10 cycles is needed for each MAC instruction.

- *Roots Search.* An evaluation of $\Lambda(x)$ on all field points. Each of the 256 results is subsequently screened for zeroes. M in this kernel is equal to 256 and D is equal to T+1. If the split-summation method is applied similarly to the syndromes-calculation kernel, a second power matrix, namely the transpose of the *exp_table_for_syndrome* matrix, is required. To conserve memory, we simply invert the order of the loops in code **Example 1** and apply the multi-sample method instead of split-summation. This is conceptually equivalent to interchanging the roles of M and D. Furthermore, unless $T + 1$ is a multiple of four, the split-summation method wastes many cycles on useless calculations. Therefore, an additional benefit gained by choosing the multi-sample method is that the cycle count becomes directly proportional to T. The price of using the multi-sample method is that there are not enough registers to complete the inner loop in five cycles. In this case, the inner loop requires six cycles instead of five. The lower bound on the cycle count is $384T + 384$ cycles, not counting cycles required for zero-screening. The zero-screening procedure is performed by reading the results byte-wise and testing for zero. This is performed two cycles per byte and thus requires 512 additional cycles. Summarizing these factors, the lower bound on the cycle count is $384T + 896$ cycles, not including overhead.
- *Forney algorithm.* A series of three concatenated polynomial evaluations. However, the entries into the power table are the error locations, which are randomly distributed. Thus, the power table is not read continuously from the power table. This adds two cycles to the inner loop. The cycle count, not including overhead, is at least:

$$7(n_{errors} + T)\left\lceil \frac{T}{4} \right\rceil + 7n_{errors}\left\lceil \frac{n_{errors}}{4} \right\rceil + 6T$$

5 Results

This section presents the cycle count simulation results for the various decoder routines. All results in this section are for the case of $T = 8$.

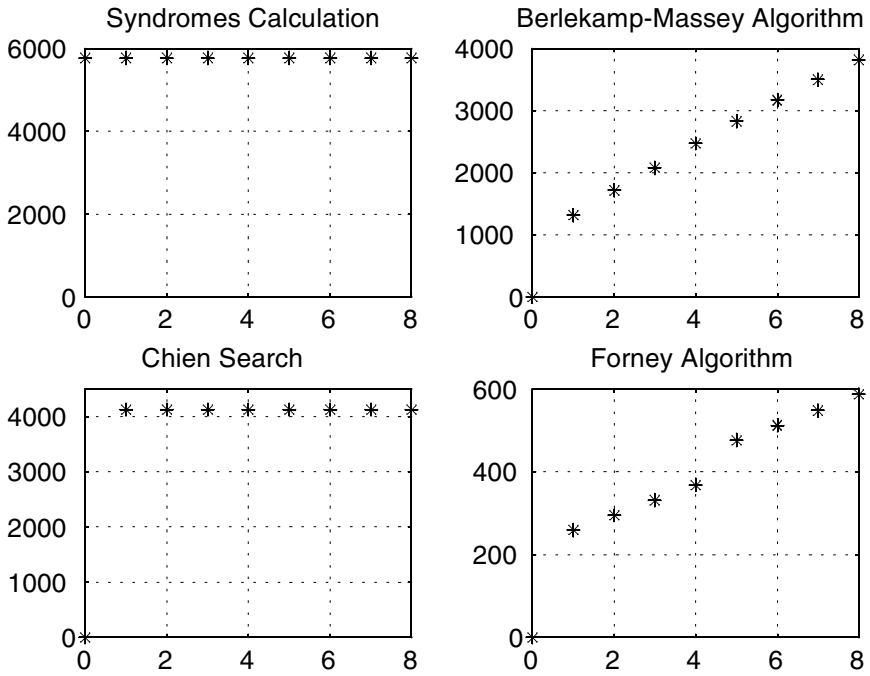


Figure 7. Average Cycle Count of Decoder Routines, as a Function of n_{errors}

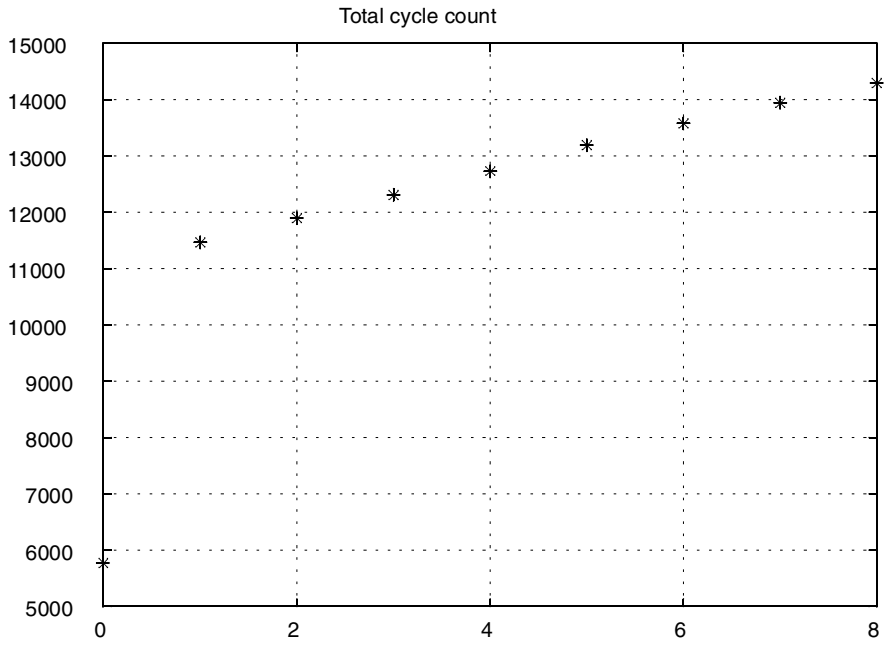


Figure 8. Average Cycle Count Sum of Decoder Routines, as a Function of n_{errors}

The assembly code for matrix multiplication contains the potential for one contention in one of the five cycles of the inner loop, since the table *exp_2_bin_extended* is accessed twice in the same cycle. The probability per contention is 1/16. **Table 2** and **Table 3** depict the results obtained for the ADSL application. The symbol time is 246.4 µsec. Assuming that the DSP runs at 300 MHz, the full processor load is 73,920 cycles. The results are obtained in the worst case, in which both T and n_{errors} equal 8.

Table 2. Encoder Routine

Encoder	Average Cycle Count	Worst Case Cycle Count	MCPS@300 MHz
Encoding routine	6359	6500	24.0
TOTAL	6359	6500	24.0

Table 3. Decoder Routine

Decoder	Average Cycle Count	Worst Case Cycle Count	MCPS@300 MHz
Syndromes calculation	5772	5894	21.8
Berlekamp-Massey	3810	3816	14.5
Roots search	4128	4128	15.6
Forney	587	590	2.1
TOTAL	14,298	14,428	54.0

Notice the syndromes calculation and the root search. The theoretical lower bounds for syndromes calculation and root search for $T = 8$ are 5664 and 3968 cycles, respectively. As **Table 2** and **Table 3** show, the actual cycle count in this implementation is very close to the lowest theoretical bound.

6 Summary

This application note demonstrates the implementation of the Reed-Solomon encoder and decoder on the SC140 core. The implementation is in accordance with the standard for ADSL systems, using Galois field GF(256) and T equal to 8. Encoding and decoding is performed by the Petersen-Gorenstein-Zierler (PGZ) algorithm. The algorithm consists mainly of polynomial evaluations over a set of Galois field points, which is equivalent to a sequence of MAC operations in the framework of Galois arithmetic. The emphasis is on efficient implementation of MAC operations on the DSP, under the constraint of reasonable memory requirements. The two most cycle-intensive routines, syndromes calculation and roots search, perform straightforward polynomial evaluations for which full parallelization is possible. Thus, we conclude that since most of the cycle-intensive algorithms can be parallelized, the architecture of the SC140 can be very efficiently employed in the Reed-Solomon application.

7 References

[1] C. E. Shannon, “A Mathematical Theory of Communication,” *Bell System Technical Journal*, vol. 27, 1948, pp. 379–423 and pp. 623–656.

[2] E. R. Berlekamp, R.E. Peile and S.P. Pope, “The Application of Error Control to Communications,” *IEEE Communications Magazine*, vol.25, no 4, 1987, pp.44–57.

[3] V. Bhargava, “Forward Error Correction Schemes for Digitized Communications,” I*EEE Communications Magazine*, vol.21, no 1, 1983, pp.11–19.

[4] I.S. Reed and G. Solomon, *Polynomial Codes Over Certain Finite Fields,* Journal of Soc. Ind. Appl. Math, vol.8, p.300-304 and Math. Rev., vol23B, p.510, 1960.

[5] S. Lin and D.J. Costello, *Error Control Coding: Fundamentals and Applications*, Prentice Hall: Englewood Cliffs, NJ, 1983.

[6] R.E. Blahut, *Theory and Practice of Error Control Codes*, Addison Wesley: Reading, MA, 1984.

[7] *SC140 DSP Core Reference Manual*, Freescale Semiconductor, (MNSC140CORE/D).

[8] ITU standard G.992.1.

Appendix A: C-Codes for Decoder

Example 2. C-Code for Syndromes Calculation

```
void calculate_syndrome(BYTE *received_block, BYTE *syndromes)
{
int i, j;
BYTE acc;
WORD x_power, y_power, power;

for (i=0; i<2*T; i++)
    {
    acc = 0;
    for (j=0; j<255; j++)
    {
      x_power = bin_2_exp[received_block[j]];
      y_power = exp_table_for_syndrome[i][j];
      power = MIN((x_power + y_power),2*N+1);
      acc ^= exp_2_bin_extended[power];
      }
      syndromes[i] = acc;
    }
}
```

Example 3. C-Code for Berlekamp-Massey Algorithm

```
void berlekamp(BYTE *s, WORD *error_loc_poly, BYTE *error_loc_bin)
{
WORD x_power, y_power, power;
BYTE temp_sigma[2*T+2][2*T], (*sigma)[2*T], *ptr1, *ptr2;
int temp_l_mu[2*T+2], *l_mu;
int temp_mu_l_mu[2*T+2], *mu_l_mu;
BYTE temp_d[2*T+2], *d;
int p, max_p_lp, mu;
BYTE inv, factor,sum;
int i, j, shift;

    /* Initialize */
sigma = &temp_sigma[1];
d = &temp_d[1];
l_mu = &temp_l_mu[1];
mu_l_mu = &temp_mu_l_mu[1];

for (j=0; j<2*T; j++)
    {

    sigma[i][j] = 0;
    }

sigma[-1][0] = 1;
d[-1] = 1;
l_mu[-1] = 0;
mu_l_mu[-1] = -1;
inv = 1;
sigma[0][0] = 1;
d[0] = s[0];
l_mu[0] = 0;
mu_l_mu[0] = 0;
p = -1;
max_p_lp = -1;
```

```
for (mu=0; mu<2*T; mu++)
    {
    /* assume sigma[mu] = sigma[mu-1] */
    for (i=0; i<=l_mu[mu]; i++)
      {
      sigma[mu+1][i] = sigma[mu][i];
      }
    l_mu[mu+1] = l_mu[mu];
    mu_l_mu[mu+1] = mu+1 - l_mu[mu+1];

    if (d[mu] != 0)
    {
    /* if discrepancy is not zero, correct sigma[mu] */
    x_power = bin_2_exp[d[mu]];
    y_power = bin_2_exp[inv];
    power = MIN((x_power + y_power),2*N+1);
    factor = exp_2_bin_extended[power];

     for (i=0; i<=l_mu[p]; i++)
      {
      x_power = bin_2_exp[factor];
      y_power = bin_2_exp[sigma[p][i]];
      power = MIN((x_power + y_power),2*N+1);
      sigma[mu+1][mu - p +i] ^= exp_2_bin_extended[power];
      }
      shift = l_mu[p]+ mu-p;
      l_mu[mu+1] = MAX(l_mu[mu],shift);
      mu_l_mu[mu+1] = mu+1 - l_mu[mu+1];
      /* update p */
      if (mu_l_mu[mu] >= max_p_lp)
             {
             p = mu;
             max_p_lp = mu_l_mu[mu];
             power = N - bin_2_exp[d[p]];
             inv = exp_2_bin_extended[power];
             }

      }

      /* calculate d[mu+1] */
      sum = s[(mu+2)-1];
      for (i=1; i<=l_mu[mu+1]; i++)
             {
             x_power = bin_2_exp[sigma[mu+1][i]];
             y_power = bin_2_exp[s[(mu+2-i)-1]];
             power = MIN((x_power + y_power),2*N+1);
             sum ^= exp_2_bin_extended[power];
             }
      d[mu+1] = sum;
    }

    for (i=0; i<2*T; i++)
    {
      error_loc_poly[i] = bin_2_exp[sigma[2*T][i]];
    }

}
```

Example 4. C Code for Roots Search

```
int roots_search(BYTE *roots, WORD *error_loc_poly, WORD *roots_poly, BYTE*
error_locations)
{
int i,j;
BYTE acc;
WORD x_power, y_power, power;
int k = 0;
int n_roots = 0;

for (i=0; i<256; i++)
{
    roots[i] = 0x00;
}

for (i=0; i<(T+1); i++)
    {
    x_power = error_loc_poly[i];
    for (j=0; j<256; j++)
    {

    y_power = exp_table_for_syndrome[i][j];
    power = MIN((x_power + y_power),2*N+1);
    roots[j] ^= (exp_2_bin_extended[power]);
    }
}

for (i=0; i<256; i++)
    {
      if (roots[i] == 0)
      {
      n_roots++;
      roots_poly[k] = i;
      error_locations[k] = (N-i) % N;
      k++;
      }
    }
    return n_roots;
}
```

Example 5. C Code for Forney Algorithm

```
void forney(BYTE *s, WORD *elp, BYTE *el, DWORD n, BYTE *err_locs, WORD *roots,
BYTE *block)
{
WORD z[T+1];
BYTE temp, nom, denom;
WORD  x_power, y_power, power;
int i,j;

/* Determine the error evaluator polynomial Z(x) */

temp = s[0] ^ el[1];
z[0] = bin_2_exp[temp];
```

```
for (i=1; i<n; i++)
    {
    temp = s[i] ^ el[i+1];
    for (j=0; j<i; j++)
      {
      y_power =  bin_2_exp[s[i-j-1]];
      power = MIN((elp[j+1] + y_power),2*N+1);
      temp ^= exp_2_bin_extended[power];
      }
    z[i] = bin_2_exp[temp];
    }

for (i=0; i<n; i++)
    {
    nom = 0x01;
    denom = 0x00;
    for (j=0; j<n; j++)
      {
      x_power = exp_table_for_syndrome[j][roots[i]];
      power = MIN(x_power + z[j],2*N+1);
      nom ^=  exp_2_bin_extended[power];
      }
    for (j=1; j<=n; j+=2)
      {
      x_power = exp_table_for_syndrome[j-1][roots[i]];
      power = MIN(x_power + elp[j],2*N+1);
      denom ^=  exp_2_bin_extended[power];

       }

    x_power = bin_2_exp[nom];
    y_power = N - bin_2_exp[denom];
    power = MIN(x_power + y_power,2*N+1);
    block[err_locs[i]] ^= exp_2_bin_extended[power];
    }
}
```

Appendix B: Assembly Codes for Decoder

Example 6. Assembly Code for Syndromes Calculation

```
;;*****************************************************************************
;*                                                                            *
;*  Reed-Solomon error correction algorithm                                   *
;*                                                                            *
;* STARCORE 140 ASSEMBLY                                                 *
;*                                                                            *
;*****************************************************************************
;*                                                                            *
;*  Module Name:  calculate_syndrome.asm                                      *
;*                                                                            *
;*****************************************************************************
;*                                                                            *
;* Calling convention from C: calculate_syndrome(received_block, syndromes)    *
;*                                                                       *
;*****************************************************************************
;*                                                                            *
;*  INPUT: r0 : BYTE received_block[N+1]                                      *
;*              Received block of 256 bytes, zero-extended by one byte        *
;*                                                                            *
```

```
;*  OUTPUT: r1 : BYTE syndromes[2*T]                                          *
;*               2T syndromes                                                 *
;*****************************************************************************
;*                                                                           *
;*  FUNCTION : Evaluates 2T syndromes defined as evaluating the              *
;*             received block viewed as a polynomial over GF(256) at field   *
;*             points alpha, alpha^2 ... alpha^2T.                           *
;*                                                                           *
;*                                                                           *
;*  PERFORMANCE: Average cycle count  = 5772                                 *
;*               Worst case cycle count = 5894                               *
;*                                                                           *
;*  ALIGNMENT REQUIREMENTS:                                                  *
;*                                                                           *
;*               &received_block[0] should be aligned 8                      *
;*               &syndromes[0]  should be aligned 8                          *
;*                                                                           *
;*****************************************************************************

section .text local
TextStart_calculate_syndrome

; Define macros

N equ 256
TT equ 16

; Allocate space for the local variables on stack pointer

allocation   equ 536

block_off    equ allocation-0
temp2_off    equ block_off-2*N
temp3_off    equ temp2_off-4
temp4_off    equ temp3_off-4
temp5_off    equ temp4_off-4
temp6_off    equ temp5_off-4
temp7_off    equ temp6_off-4
temp8_off    equ temp7_off-4

    global _calculate_syndrome

    align 16
    opt lpa

_calculate_syndrome typefunc

; Overhead

START_SYNDROME

    push d6               push d7
    push r6               push r7

    adda #allocation,sp,r6
    tfra r6,sp

    adda #-block_off,sp,r10
    adda #-temp2_off,sp,r2
    adda #-temp3_off,sp,r3
    adda #-temp4_off,sp,r4
    adda #-temp5_off,sp,r5
    adda #-temp6_off,sp,r6
    adda #-temp7_off,sp,r7
    adda #-temp8_off,sp,r8
```

```
    doen3 #N/4                                      ; N = length of vector
    dosetup3  LOOP_ON_BLOCK

    move.l #_bin_2_exp,d0

[   clr d1                clr d3
    clr d5                clr d7
    tfra r10,r8           move.l (r0)+,d2         first 4 vector entries -> d2
]

[   asrr #8,d2            insert #8,#1,d2,d1      ; 2 x 1-st vector entry -> d1
    tfr d0,d14
]

[   asrr #8,d2            insert #8,#1,d2,d3      ; 2 x 2-nd vector entry -> d3
    add d0,d1,d1                                  ; 1-st table index ->d1
]

[   asrr #8,d2            insert #8,#1,d2,d5      ; 2 x 3-rd vector entry -> d5
    add d14,d3,d3                                 ; 2-nd table index ->d3
    move.l d1,r3                                  ; 1-st table index -> r3
]

[   add d14,d5,d5         insert #8,#1,d2,d7      ; 2 x 4-th vector entry -> d7
    move.l d3,r4                                  ; 2-nd-st table index -> r4
]

[   add d14,d7,d7                                 ; 4-th table index -> d7
    move.l d5,r5                                  ; 3-rd table index -> r5
]

; Main loop. Here, the table lookups are done."i" is the index of the iteration,
; ranging from 0 to #N/4-1.

    loopstart3
LOOP_ON_BLOCK

    [ clr d1              clr d3
      clr d5              clr d7
      move.l (r0)+,d2     move.l d7,r6
    ]

; clear d1,d3,d5,d7
; vector entries [5,6,7,8+i]-> d2    table index[4+i] -> r6

    [ tfr d14,d0
      asrr #8,d2          insert #8,#1,d2,d1
      move.w (r3),d4      move.f (r4),d6
    ]

; copy #_bin_2_exp to d0
; vector entry [6+i] -> d2 2 x vector entry[5+i] -> d1

; result[1+i] -> d4       result[2+i] -> upper portion of d6

    [ asrr #8,d2          insert #8,#1,d2,d3
      add d0,d1,d1        eor d6,d4
      move.w (r5),d0      move.f (r6),d6
    ]

; vector entry [7+i] -> d2 2 x vector entry[6+i] -> d3
; table index[5+i] ->d1   unify result[1+i] and result[2+i] into d4
; result[3+i] -> d0       result[4+i] -> upper portion of d6
```

```
    [ asrr #8,d2           insert #8,#1,d2,d5
      add d14,d3,d3        eor d0,d6
      move.l d1,r3
    ]

; vector entry [8+i] -> d2 2 x vector entry[7+i] -> d5
; table index[6+i] ->d3    unify result[3+i] and result[4+i] into d6
; table index[4+i] -> r3

    [ add d14,d5,d5        insert #8,#1,d2,d7
      move.l d3,r4         move.l d4,(r10)+
    ]

; table index[7+i] ->d5    2 x vector entry[8+i] -> d7
; table index[6+i] -> r4   write results[1+i,2+i] into r10

    [ add d14,d7,d7
      move.l d5,r5         move.l d6,(r10)+
    ]

; table index[8+i] ->d7
; table index[7+i] -> r5   write results[3+i,4+i] into r10

      loopend3

      doen2 #TT
      dosetup2 LOOP_ON_ALPHA

      move.l #_exp_2_bin_extended,d14
      move.l #_exp_table_for_syndrome,r2
      tfra r8,r10
      move.w #$1ff,d0      move.w #$1ff,d1
      dosetup3 LOOP_ON_BLOCK_POLY
      tfra r10,r8

      loopstart2
LOOP_ON_ALPHA

; For the software pipeline, d2,d3,d4 and d8 are prepared to 1-st iteration
; while d5,d6,d7,d13 are cleared (= i.e. prepared to the 0-th iteration)

    [ clr d15              doen3 #N/4              ; clear accumulator d15
      tfra r8,r10                                  ; reset ptr to vector start
    ]

    [ move.2l (r10)+,d10:d11move.2l (r2)+,d8:d9    ; table entries [1..4]->
d8:d9
    ]                                              ; vector entries [1..4] ->
d10:d11

    [ add2 d8,d10          add2 d9,d11             ; 4 sum of exponents ->
d10:d11
      tfr d1,d12                                   ; (to be separated later)
    ]                                              ; 511 in d1 and d12

    [ zxt.w d10,d5         zxt.w d11,d3            ; 1-st sum of exponents -> d5
      tfr d1,d7            asrw d10,d4             ; 3-rd sum of exponents -> d3
    ]                                              ; 2-nd sum of exponents -> d4

    [ min d1,d5            min d3,d7               ; 1-st,3-rd offset -> d5,d7
      min d0,d4            asrw d11,d8             ; 2-nd offset ->d4
    ]                                              ; 4-th sum of exponents -> d8

; d4,d8 are now initialized for the inner loop
```

```
    [ add d14,d5,d2        add d14,d7,d3           ; 1-st,3-rd table index -
>d2,d3
    ]

; d2,d3 are now initialized for the inner loop

    [ clr d7               clr d13                 ; results[-3,-2,-1,0]=0
      clr d5               clr d6
      move.2l (r10)+,d10:d11                       ; vector terms [5,6,7,8]->
d10:d11
    ]

; Main loop. Here the MACs over GF(256) are done."i" is the index of the
iteration,
; ranging from 0 to #N/4-1.Results with indices -3 ... 0 -> results were
initialized ; to zero.

    loopstart3
LOOP_ON_BLOCK_POLY

    [ add d14,d4,d4        eor d7,d15
      tfr d1,d7            min d8,d12
      move.l d3,r4         move.2l (r2)+,d8:d9
    ]
; table index[2+i] -> d4   result[1+4(i-1)] is added to accumulator
; copy 511 into d7         offset[4+i] -> d12
; table entries[5+i..8+i]->d8:d9table index[3+i] -> r4

    [ add2 d8,d10          eor d5,d15
      add d14,d12,d5       add2 d9,d11
      move.l d4,r3         move.l d2,r6
    ]

; sum of exponents[5,6+i] -> d10    result[1+4(i-1)] is added to accumulator
; table index[4+i] -> d5            sum of exponents[7,8+i] -> d11
; table index[2+i] -> r3            table index[1+i] -> r6

    [ zxt.w d10,d5                 eor d6,d15
      zxt.w d11,d3                 asrw d10,d4
      move.l d5,r5                 moveu.b (r4),d6
    ]

; sum of exponents [5+i] -> d5      result[3+4(i-1)] is added to accumulator
; sum of exponents [6+i]-> d4       sum of exponents [7+i] -> d3
; table index[4+i] -> r5            result[3+i] -> d6

    [ min d1,d5                    eor d13,d15
      min d3,d7                    asrw d11,d8
      moveu.b (r3),d13             move.2l (r10)+,d10:d11
    ]

; offset[5+i] -> d5                 result[2+4(i-1)] is added to accumulator
; offset[7+i] -> d7                 sum of exponents[8+i] -> d8
; result[2+i] -> d13                vector terms [9+i..12+i]-> d10:d11

    [ add d14,d5,d2                min d0,d4
      add d14,d7,d3                tfr d1,d12
      moveu.b (r6),d7              moveu.b (r5),d5
    ]

; table index[5+i] -> d2            offset[6+i] -> d4
; table index[4+i] -> d3            copy 511 into d12
; result[1+i] -> d7                 result[4+i] -> d5

      loopend3
```

```
; Sum up the last results to accumulator and write the result into AGU register
r1.
END_LOOP_ON_BLOCK_POLY

      eor d7,d15
      eor d5,d15
      eor d13,d15
      eor d6,d15
      move.b d15,(r1)+              suba #8,r2

      loopend2

      adda #-allocation,sp,r6
      tfra r6,sp

      pop r6                        pop r7
      pop d6                        pop d7

END_SYNDROME
      rts

    global Fcalculate_syndrome_end
Fcalculate_syndrome_end

TextEnd_calculate_syndrome
      endsec
```

Example 7. Assembly Code for Berlekamp-Massey Algorithm

```
;*****************************************************************************
;*                                                                           *
;*                                                                           *
;*  Reed-Solomon error correction algorithm                                  *
;*                                                                           *
;* SC140 ASSEMBLY                                                      *
;*                                                                           *
;*****************************************************************************
;*                                                                           *
;*  Module Name:     berlekamp.asm                                           *
;*                                                                           *
;*****************************************************************************
;*                                                                           *
;* Calling convention from C:                                                *
;* berlekamp(syndromes, error_loc_poly, error_loc_poly_bin)                  *
;*                                                                           *
;*****************************************************************************
;*                                                                           *
;*  INPUT: r0 :      BYTE syndromes[2*T]                                     *
;*                   2T syndromes                                            *
;*                                                                           *
;*  OUTPUT: r1       : WORD error_loc_poly[2*T]                              *
;*                     Error location polynomial in exponential form         *
;*          (sp-588) : BYTE error_loc_poly_bin[2*T]                          *
;*                     Error location polynomial in binary form              *
;*                                                                           *
;*****************************************************************************
;*                                                                           *
;*  FUNCTION : Deriving the error location polynomial by Berlekamp's iterative *
;*             algorithm. This is a compiled code which has been slightly    *
;*             modified  by applying software pipelining applied and efficient *
;*             register allocation.                                          *
;*                                                                           *
;*  PERFORMANCE: Cycle count:                                                *
```

```
;*                                                                              *
;*  #errors = 0 -> 0 cycles                                                     *
;*  #errors = 1 -> 1317 cycles                                                  *
;*  #errors = 2 -> 1716 cycles                                                  *
;*  #errors = 3 -> 2082 cycles                                                  *
;*  #errors = 4 -> 2472 cycles                                                  *
;*  #errors = 5 -> 2829 cycles                                                  *
;*  #errors = 6 -> 3172 cycles                                                  *
;*  #errors = 7 -> 3501 cycles                                                  *
;*  #errors = 8 -> 2816 cycles                                                  *
;*                                                                              *
;*                                                                              *
;*  ALIGNMENT REQUIREMENTS:                                                     *
*                                                                          *
;*               &syndromes[0] should be aligned 8                              *
;*               &error_loc_poly[0] should be aligned 8                         *
;*               &error_loc_poly_bin[0] should be aligned 8                     *
;*                                                                              *
;********************************************************************************

    section .data local
    align    8

F__MemAllocArea
    ds       13056         ; gap

Dexternal_aliased
    ds       4             ; offset = 13056

Dsoft_stack
    ds       4             ; offset = 13060

Dasm_3
    ds       4             ; offset = 13064

Dasm_2
    ds       4             ; offset = 13068

Dasm_1
    ds       4             ; offset = 13072
    ds       1             ; gap

Dasm_4
    ds       1             ; offset = 13077

    align4

    endsec

    section .text local
TextStart_berlekamp

bb_cs_offset__berlekampequ0; At _berlekamp sp = 0
bb_cs_offset_DW_2equ2      ; At DW_2 sp = 2
bb_cs_offset_DW_3equ4      ; At DW_3 sp = 4
bb_cs_offset_DW_5equ144    ; At DW_5 sp = 144
bb_cs_offset_DW_162equ2    ; At DW_162 sp = 2
bb_cs_offset_DW_163equ0    ; At DW_163 sp = 0

     global _berlekamp

     align 16

     opt lpa
```

```
_berlekamptypefunc
      push d6               push d7
DW_2
      push r6               push r7
DW_3
      adda #>560,sp,r6
    [ clr d1
      tfra r6,sp
    ]
DW_5

       adda #>-232,sp,r4    move.w #288,d6
      move.l r1,(sp-556)    doensh3 d6
      adda #>-536,sp,r2     adda #>-259,sp,r3
      adda #>-156,sp,r5     adda #>-552,sp,r6
      move.l r0,(sp-560)    move.l r4,(sp-164)

loopstart3

L43
      move.b   d1,(r6)+
      loopend3

      doen2 #<16
      dosetup2 L42
    [ clr d6                clr d2
      clr d3                clr d4
      moveu.b (r0),d5
    ]

      move.l d1,(sp-236)

    [ inc d2                inc d3
      clr d1
      move.l d6,(r4)
    ]

    [ inc d1
      move.w #<-1,d4        move.b d2,(sp-552)
    ]

      move.b d3,(sp-260)    adda #>-259,sp,r7
      move.b d5,(r3)        move.l d4,(sp-160)
      move.w #<1,r6         move.l d6,(r5)
      suba r11,r11          adda #>-232,sp,r12
      move.b d1,(r2)        adda  #>1,r0,r3

    [ clr d5
      adda #>-258,sp,r8     adda #>-152,sp,r9
    ]

    [ inc d5
      adda #>-228,sp,r10    move.w #<-1,d7
    ]

    loopstart2

L42


    [ tfr d6,d8             addnc.w #<1,d6,d10
      move.l (r12),r0       move.l (r12),d11
    ]
```

```
      nop
      tstgea   r0
      bf       <L8

    [ addnc.w #<1,d11,d9   asll #<4,d8
      asll #<4,d10
    ]

      sub #1,d9            move.l d8,r14
      doensh3 d9           move.l d10,r13
      adda r2,r14
      adda r2,r13          moveu.b (r14)+,d8

      loopstart3
L41
      moveu.b (r14)+,d8    move.b d8,(r13)+

      loopend3

      move.b d8,(r13)+

L30
L8

      move.l (r12),d9      moveu.b (r7),d11

    [ sub d9,d5,d10          tsteq d11
      move.l d9,(r10)        move.l #_bin_2_exp,r14
    ]

    btd      L9

    move.l   d10,(r9)        tfra     r6,r15

    [ tfr d5,d11
      asla r15               moveu.b (r7),r13
    ]

      adda r14,r15 move.l    #_exp_2_bin_extended,r0

    [ asll #<4,d11
      moveu.w (r15),d9       asla r13
    ]

      adda r14,r13           move.w #511,d14

    [ sxt.l d11
      moveu.w (r13),d8       move.l d4,r14
    ]

      move.l r13,(sp-36)     move.l (sp-164),r4

    [ add  d8,d9,d10
      asl2a r14
    ]

    [ min d10,d14            tfr d4,d8
      adda r4,r14
    ]

    [ sub d4,d6,d10
      move.l d14,(sp-28)
    ]

    [ asll #<4,d8
      moveu.w (sp-28),r15    move.l (r14),r1
    ]
```

```
        nop

        adda r0,r15             tstgea r1

        moveu.b (r15),r15       move.l (r14),d12

      bf       L13

        asla r15                move.l #_bin_2_exp,r1

        dosetup3 L39

      [ addnc.w  #<1,d12,d9
        adda r1,r15             move.l r2,d12
      ]

      [ add d12,d11,d13
        doen3 d9                move.l d10,r13
      ]

        move.l d13,r0           moveu.w  (r15),r15
        move.l d8,r1            move.w r15,(sp-32)

        adda r13,r0            moveu.w (sp-32),d9
        adda r2,r1

        falign
        loopstart3

L39     moveu.b (r1)+,r13               move.l #_bin_2_exp,r4
        move.w #511,d14                 asla r13
        moveu.b (r0),d15                adda r4,r13
        moveu.w  (r13),d8               move.l #_exp_2_bin_extended,r4
        add d9,d8,d10
        min d10,d14
        move.l d14,r13
        nop
        adda r4,r13
        moveu.b (r13),d8
        eor d15,d8
        move.b d8,(r0)+
        loopend3

L32
L13

         move.l (r14),d12               move.l (r12),d8
      [ add d12,d6,d9
        tfra r11,r0                     move.w #255,d11
      ]

      [ sub d4,d9,d12
        adda r5,r0                      move.l #_exp_2_bin_extended,r4
      ]

        max d8,d12
      [ sub d12,d5,d9
        move.l d12,(r10)
      ]

         move.l d9,(r9)                 move.l (r0),d10
        cmpgt d10,d7
        bt       <L9
```

```
    [ tfr d6,d4
      move.l (r0),d7                move.l (sp-36),r13
    ]

       nop
      moveu.w (r13),d9
      sub d9,d11,d11
      move.l d11,r0
      nop
      zxta.w r0
      adda r4,r0
      moveu.b (r0),r6

L9

    [ tfr d5,d11
      move.l (r10),d9               moveu.b (r3)+,d8
    ]

    [ cmpgt.w #<0,d9
      move.l (r10),d10              move.l d6,r1
    ]

    bf L20
    dosetup3 L40

    [ asll #<4,d11
      doen3 d10                     move.l (sp-560),r4
    ]


      move.l d11,r0                 move.l #_bin_2_exp,r14

      adda r4,r1                    adda r2,r0
      adda #<1,r0                   move.l #_exp_2_bin_extended,r15
      falign
      loopstart3

L40

      moveu.b (r0)+,r13             move.w #511,d15
      nop
      asla r13
      adda r14,r13
      moveu.w  (r13),d9             moveu.b  (r1),r13
      asla r13                      deceqa r1
      adda r14,r13

      moveu.w (r13),d10
      add d9,d10,d11
      min d11,d15
      move.l d15,(sp-32)
      moveu.w (sp-32),r13
      nop
      adda r15,r13
      moveu.b (r13),d9
      eor d9,d8
      loopend3

L34
L20
```

```
    [ inc d6                        inc d5
      move.b d8,(r8)+               adda #<4,r12
    ]

      adda #<4,r11                  adda #<4,r10
      adda #<4,r9                   adda #<1,r7
      loopend2

      doen3 #15
      dosetup3 L44
      adda #>-280,sp,r2
      moveu.b (r2),d4               move.l (sp-588),r0
      move.l #_bin_2_exp,r14        move.b d4,(r0)+
      moveu.b (r2)+,r3              move.l (sp-556),r1
      asla r3
      adda r14,r3
      moveu.w (r3),d5
      loopstart3

L44

      moveu.b (r2),d4               move.w d5,(r1)+
      moveu.b (r2)+,r3
      move.b d4,(r0)+
      asla r3
      adda r14,r3
      moveu.w (r3),d5

      loopend3

      adda #>-560,sp,r6
      tfra r6,sp                    move.w d5,(r1)+
      DW_161
      pop r6                        pop r7
      DW_162
      pop d6                        pop d7
      DW_163
      rts

      global Fberlekamp_end
Fberlekamp_end

TextEnd_berlekamp
      endsec
```

Example 8. Assembly Code for Roots Search

```
;*****************************************************************************
;*                                                                           *
;*  Reed-Solomon error correction algorithm                                  *
;*                                                                           *
;* SC140 ASSEMBLY                                                      *
;*                                                                           *
;*****************************************************************************
;*                                                                           *
;*  Module Name:  chien_search.asm                                           *
;*                                                                           *
;*****************************************************************************
;*                                                                           *
;* Calling convention from C:                                                *
;* n_roots = chien_search(roots, error_loc_poly, roots_poly, error_locations)  *
;*                                                                           *
;*****************************************************************************
;*                                                                           *
```

```
;*  INPUT: r0 :      BYTE roots[N+1]                                          *
;*                   Table elements -> root candidates                        *
;*         r1 :      WORD error_loc_poly[2*T]                                 *
;*                   Error location polynomial in exponential form            *
;*                                                                            *
;*  OUTPUT: (sp-28) : WORD roots_poly[2*T]                                    *
;*                    Exponents of the roots                                  *
;*          (sp-32) : BYTE error_locations[T]                                 *
;*                    Error locations                                         *
;*          d0      : DWORD n_roots                                           *
;*                    Number of roots                                         *
;*                                                                            *
;******************************************************************************
;*                                                                            *
;*  FUNCTION : Find the error locations by evaluating the error location      *
;*             polynomial at all field points. The error locations are derived *
;*             from the exponents of the field elements which are the roots   *
;*             of the error location polynomial.                              *
;*                                                                            *
;*  PERFORMANCE: Cycle count  = 4128                                          *
;*                                                                            *
;*  ALIGNMENT REQUIREMENTS:                                                   *
;*                                                                            *
;*               &roots[0]  should be aligned 8                               *
;*               &error_loc_poly[0] should be aligned 8                       *
;*               &roots_poly[0]  should be aligned 8                          *
;*               &error_locations[0]  should be aligned 8                     *
;*                                                                            *
;******************************************************************************

    section .text local
TextStart_chien_search

; Define macros

BYTE_SIZE equ 8
N equ 256
T equ 8

; Allocate space for the local variables on stack pointer and calculate offsets
of function arguments

allocation   equ 24

roots_off    equ allocation+28
err_locs_off equ allocation+32

temp2_off    equ allocation-0
temp3_off    equ temp2_off-4
temp4_off    equ temp3_off-4
temp5_off    equ temp4_off-4
temp6_off    equ temp5_off-4
temp7_off    equ temp6_off-4
temp8_off    equ temp7_off-4

    global _chien_search

    align 16
    opt lpa

_chien_search typefunc

; Overhead
```

```
      push r6                                 push r7
      push d6                                 push d7

      adda #allocation,sp,r6
      tfra r6,sp

      move.l (sp-roots_off),r13               move.l (sp-err_locs_off),r14
      adda #-temp2_off,sp,r2
      adda #-temp3_off,sp,r3
      adda #-temp4_off,sp,r4
      adda #-temp5_off,sp,r5
      adda #-temp6_off,sp,r6
      adda #-temp7_off,sp,r7
      adda #-temp8_off,sp,r8

START_CHIEN_SEARCH
      doen2 #T+1              clr d15
      dosetup2 LOOP_ON_T

      move.l #_exp_2_bin_extended,d14
      move.l #_exp_table_for_syndrome,r2



      move.w #$1ff,d0         move.w #$1ff,d1
    [ dosetup3 LOOP_ON_N
      tfra r0,r8              move.w (r1)+,d11        ; 1-st term of error_loc_poly
->
      d11
    ]

    aslw d11,d10

; For the software pipeline, d2,d3,d4 and d8 are prepared to 1-st iteration
; while d5,d6,d7,d13 are prepared to the 0-th iteration)

    loopstart2
LOOP_ON_T

    [ doen3 #N/4              eor d11,d10             ; 1-st term of error_loc_poly
in
      tfr d1,d12                                      ; upper and lower portion of
d10
      tfra r8,r0              move.2l (r2)+,d8:d9
    ]

    [ add2 d10,d8             add2 d10,d9
    ]

    [ zxt.w d8,d5             zxt.w d9,d3
      tfr d1,d7               asrw d8,d4
    ]

    [ min d1,d5               min d3,d7
      min d0,d4               asrw d9,d8
    ]

    [ add d14,d5,d2           add d14,d7,d3
      add d14,d4,d4           min d8,d12
    ]

    [ add d14,d12,d5
      move.l d4,r3            move.l d3,r4
    ]
```

```
    [ move.l d5,r5          move.2l (r2)+,d8:d9
    ]

    [ moveu.b (r3),d13      move.l d2,r6
    ]

    [ zxt.w d8,d5           asrw d8,d4
      moveu.b (r4),d6       moveu.b (r5),d11
    ]

    [ min d1,d5             min d0,d4
      asll #8,d13
      moveu.b (r6),d7
    ]

    [ add d14,d5,d2         zxt.w d9,d5
      asrw d9,d8            tfr d11,d9
    ]

    [  min d1,d5            asll #24,d9
    ]

    [ add d14,d5,d3         tfr d9,d5
      tfr d1,d12            asll #16,d6
      move.l (r0),d15
    ]

; Main loop. Here the MACs over GF(256) are done."i" is the index of the
iteration,
; ranging from 0 to #N/4-1.Results with indices -3 ... 0 -> results were
initialized to zero.

; Main loop

    loopstart3
LOOP_ON_N

    [ add d14,d4,d4                 eor d7,d15
      tfr d1,d7                     min d8,d12
      move.l d3,r4                  move.2l (r2)+,d8:d9
    ]

; table index[2+i] -> d4            result[1+4(i-1)] is added to accumulator
; copy 511 into d7                  offset[4+i] -> d12
; table entries[5+i..8+i]->d8:d9    table index[3+i] -> r4

    [ add2 d10,d8                   eor d5,d15
      add d14,d12,d5                add2 d10,d9
      move.l d4,r3                  move.l d2,r6
    ]

; sum of exponents[5,6+i] -> d8     result[1+4(i-1)] is added to accumulator
; table index[4+i] -> d5            sum of exponents[7,8+i] -> d9
; table index[2+i] -> r3            table index[1+i] -> r6


    [ zxt.w d8,d5                   eor d6,d15
      zxt.w d9,d3                   asrw d8,d4
      move.l d5,r5                  moveu.b (r4),d6
    ]

; sum of exponents [5+i] -> d5      result[3+4(i-1)] is added to accumulator
; sum of exponents [6+i]-> d4       sum of exponents [7+i] -> d3
; table index[4+i] -> r5            result[3+i] -> d6
```

```
    [ min d1,d5                     eor d13,d15
      min d3,d7                     min d0,d4
      moveu.b (r3),d13
    ]

; offset[5+i] -> d5                result[2+4(i-1)] is added to accumulator
; offset[7+i] -> d7                offset[6+i] -> d4
; result[2+i] -> d13

    [ add d14,d5,d2                 asrw d9,d8
      add d14,d7,d3                 tfr d1,d12
      move.l d15,(r0)+              moveu.b (r5),d5
    ]

; table index[5+i] -> d2           sum of exponents[8+i] -> d8
; table index[4+i] -> d3           copy 511 into d12
; result[1+i] -> d7                result[4+i] -> d5

    [ asll #24,d5                   asll #16,d6
      asll #8,d13
      moveu.b (r6),d7               move.l (r0),d15
    ]



; result[4+i] -> bits [31:24] of d5 result[3+i] -> bits[23:16] of d6
; result[2+i] -> bits [15:8] of d13
; result[1+i] -> d7                read accumulator of i-th iteration

      loopend3

END_LOOP1

      suba #16,r2          move.w (r1)+,d11        ; i-th term of error_loc_poly
-> d11

         aslw d11,d10

      loopend2

; Root finding routine. The 256 candidate roots are scanned for zeroes. The
exponent ; of the root is written into r13 and the error location indices into
r14.

      dosetup3 FIND_ZEROES
      tfra r8,r0                                ; r0 -> first table element

    [ doen3 #N/2-2
      clr d0               clr d1               ; roots counter d0 cleared
      moveu.b (r0)+,d2                          ; root exponent d1 cleared
    ]

    [ tsteq d2
      moveu.b (r0)+,d2


    [ ift

      inc d0
      move.w d1,(r13)+     move.b d1,(r14)+     ; special case: zero element
    ]

    [ tsteq d2
      inc d1
      moveu.b (r0)+,d4
    ]
```

```
      move.l #N-2,d3

    [ ift
      inc d0
      move.b d3,(r14)+     move.w d1,(r13)+        ; error location index -> d3
    ]

FIND_ZEROES
    loopstart3

start_loop

    [ tsteq d4
      inc d1               sub #1,d3
      moveu.b (r0)+,d2
    ]

    [ ift
      inc d0
      move.b d3,(r14)+     move.w d1,(r13)+
    ]

    [ tsteq d2
      inc d1               sub #1,d3
      moveu.b (r0)+,d4
    ]

    [ ift
      inc d0
      move.b d3,(r14)+     move.w d1,(r13)+
    ]

end_loop

     loopend3

      adda #-allocation,sp,r6
      tfra r6,sp

      pop d6               pop d7
      pop r6               pop r7

END_CHIEN_SEARCH
      rts

      global Fchien_search_end
Fchien_search_end

TextEnd_chien_search
      endsec
```

Example 9. Assembly Code for Forney Algorithm

```
;;*****************************************************************************
*
;;*                                                                           *
;*  Reed-Solomon error correction algorithm                                  *
;*                                                                           *
;* SC140 ASSEMBLY                                                           *
;*                                                                           *
;*****************************************************************************
;*                                                                           *
;*  Module Name:     forney.asm                                              *
;*                                                                           *
;*****************************************************************************
;*                                                                           *
```

```
;* Calling convention from C:                                                  *
;* forney(syndromes,error_loc_poly, n_roots,error_locations,                   *
;*          roots_poly,received_block)                                         *
;*                                                                             *
;*****************************************************************************
;*                                                                             *
;*  INPUT: r0 :       BYTE syndromes[2*T]                                      *
;*                    2T syndromes                                             *
;*         r1 :       WORD error_loc_poly[2*T]                                 *
;*                    Error location polynomial in exponential form            *
;*         (sp-28) :  DWORD n_roots                                            *
;*                    Number of roots                                          *
;*         (sp-32) :  BYTE error_locations[T]                                  *
;*         (sp-36) :  WORD roots_poly[2*T]                                     *
;*                    Error locations                                          *
;*                    Exponents of the roots                                   *
;*         (sp-40) :  BYTE received_block[N]                                   *
;*                    Received block                                           *
;*                                                                             *
;*  OUTPUT: (sp-40) : BYTE received_block[N]                                   *
;*                    Corrected block                                          *
;*                                                                             *
;*****************************************************************************
;*  FUNCTION : Deriving the error location polynomial by Berlekamp's iterative *
;*             algorithm. This is a compiled code which has been slightly      *
;*             modified  by applying software pipelining applied and efficient *
;*             register allocation.                                            *
;*             polynomial at all field points. The error locations are derived *
;*             from the exponents of the field elements which are the roots    *
;*             of the error location polynomial.                               *
;*                                                                             *
;*  PERFORMANCE: Cycle count:                                                  *
;*                                                                             *
;*  #errors = 0 -> 0 cycles                                                    *
;*  #errors = 1 -> 259 cycles                                                  *
;*  #errors = 2 -> 295 cycles                                                  *
;*  #errors = 3 -> 331 cycles                                                  *
;*  #errors = 4 -> 368 cycles                                                  *
;*  #errors = 5 -> 476 cycles                                                  *
;*  #errors = 6 -> 512 cycles                                                  *
;*  #errors = 7 -> 548 cycles                                                  *
;*  #errors = 8 -> 587 cycles                                                  *
;*                                                                             *
;*  ALIGNMENT REQUIREMENTS:                                                    *
;*                                                                             *
;*               &syndromes[0] should be aligned 8                             *
;*               &error_loc_poly[0] should be aligned 8                        *
;*               &error_locations should be aligned 8                          *
;*               &roots_poly[0] should be aligned 8                            *
;*               &received_block[0] should be aligned 8                        *
;*                                                                             *
;*****************************************************************************

; Define macros

N equ 255

T equ 8
```

```
allocation        equ 104
n_off             equ allocation+28
err_locs_off      equ allocation+32
rp_off            equ allocation+36
r_off             equ allocation+40

z_off             equ allocation-0
z_exp_off         equ z_off-2*T
temp_off          equ z_exp_off-2*T
t_nomden_off      equ temp_off-2*(2*T)

    section .data local

    global _forney

    align 16
    opt lpa

_forneytypefunc
      push d6              push d7
      push r6              push r7

      adda #allocation,sp,r6
      tfra r6,sp

BEGIN_FORNEY

;  fill in the temp array (r3) from the s array (r0)

      adda #-temp_off,sp,r3                                ; r3->temp[0]
      doensh3 #(T-1)       tfra r3,r9
      move.l #_bin_2_exp,r14
      move.l #2*N+1,d0
    [ clr d1
      tfra r14,r12         tfra r14,r13
    ]

      loopstart3

      move.w d0,(r3)+                                      ; temp[0]...[T-2] =
2*N+1

      loopend3

      moveu.b (r0)+,r5     doen3 #T/2                      ; r5 some temp
register
      addl1a r5,r12        dosetup3 LOOP_ON_S
      move.w d1,(r3)+      moveu.b (r0)+,r6                ; temp[T-1] = 0
      moveu.w (r12),d5                                     ; r6 some temp
register

      loopstart3

LOOP_ON_S

      tfra r14,r12         tfra r14,r13
      moveu.b (r0)+,r5     addl1a r6,r12
      move.w d5,(r3)+      addl1a r5,r13
      moveu.b (r0)+,r6     moveu.w (r12),d6
      move.w d6,(r3)+      moveu.w (r13),d5

      loopend3

      move.w d5,(r3)+
      move.w d6,(r3)       suba #(T+2),r0                  ; r0 -> s[0]

; Calculate m = (short int) (((n-1) >> 2) + 1);
```

```
      move.l (sp-n_off),d15
      sub #1,d15
      asrr #2,d15
      inc d15

; m in d15 now, elp in (sp-elp_off), temp in r3

      adda #-4,sp,r7
      move.w #0,(r7)
      adda #2*T,r9                                        ; r9 -> temp[T]
      dosetup2 OUTER_LOOP
      tfr d15,d0                                          ; m -> d0
      doen2 d0
      dosetup3 INNER_LOOP
      adda #-z_off,sp,r2                                  ; r2->z[0]
      move.l #_exp_2_bin_extended,r3
      tfra r9,r10          tfra r1,r15                    ; r10 -> temp[T]
                                                          ; r15 -> elp[0]

         moveu.w (r1)+,d7                              ; r8 ->elp[1]
         clr d3
         move.w #(2*N+1),d0
         tfr d0,d1         tfr d0,d2

      loopstart2

OUTER_LOOP
    [ clr d8               clr d9                    ; sum[0..3]=0

      clr d10              clr d11

      doen3 #(T+1)         move.4w (r10),d12:d13:d14:d15 ; load temp[T+0..3]

    ]                                                 ; denote:t0...t3 =
temp[T+0...3]

    [ add d7,d12,d4        add d7,d13,d5              ; pow0,1=x_pow+t0,1
      add d7,d14,d6                                   ; pow2=x_pow+t2
      suba #2,r10          adda #-4,sp,r7             ; r10->temp[T-1]
    ]

    loopstart3

INNER_LOOP

;1

    [ eor d3,d10           min d0,d4
      min d1,d5            add d7,d15,d15
      moveu.b (r7),d3      moveu.w (r1)+,d7
    ]

; sum2^=sum -> d10                                    pow0 = min(pow0,2*N+1) -> d4
; pow1=min(pow1,2*N+1) -> d5                          pow3=x_pow+t3 -> d15
; load bin_2_exp[pow3] -> d3                          load x_pow=elp[j+1] -> d7

;2

    [ eor d3,d11           min d2,d6
      tfr d15,d4
      move.l d4,r4         move.l d5,r5
    ]

;     sum3^=sum3 -> d11                               pow2=min(pow2,2*N+1) -> d6
;     copy d15 into d4
;     pow0 = min(pow0,2*N+1) -> r4                    pow1=min(pow1,2*N+1) -> r5
```

```
;3

     [ min d0,d4           tfr d14,d15
      tfr d13,d14          tfr d12,d13
      move.l d6,r6         moveu.w (r10)-,d12

     ]

; pow3=min(pow3,2*N+1) -> d4    t3 = t2
; t2 = t1                       t1 = t0
; load t0=temp[T-(j+1)] -> r6

;4

    [ adda r3,r4           move.l d4,r7
    ]

;5

    [ moveu.b (r4),d3      adda r3,r5
    ]

; load b[pow0] -> d3

;6

    [ eor d3,d8
      moveu.b (r5),d3      adda r3,r6
    ]

; sum0^=sum0 -> d8
; load b[pow0] -> d3

;7

    [ eor d3,d9            add d7,d12,d4
      add d7,d13,d5        add d7,d14,d6
      moveu.b (r6),d3      adda r3,r7
    ]

; sum1^=sum1 -> d9         pow0=x_pow+t0 -> d4
; pow1=x_pow+t1 -> d5      pow2=x_pow+t2 -> d6
; load b[pow2] -> d3

    loopend3

    [ eor d3,d10                                 ; sum2^=sum2 ->d10
      moveu.b (r7),d3      tfra r15,r1           ; load b[pow3]->d3, r1-
>elp[0]

    ]

    [ eor d3,d11           clr d3                ; sum3^=sum3->d11
      adda #(4*2),r9,r9    moveu.w (r1)+,d7      ; 9->temp[T+i+4],x_pow=elp[0]
    ]

    [ move.4w d8:d9:d10:d11,(r2)+                ; store z[i+0..3]=sum0..3
      tfra r9,r10
    ]

    loopend2

LLABEL

      adda #-z_off,sp,r2                         ; r2 -> z[0]
```

```
    [ clr d2               clr d3
      move.l (sp-n_off),r5 move.l (sp-n_off),d5
    ]

      asla r5              move.l #T,d4
      sub d5,d4,d5
      adda r2,r5           doensh3 d5              ; r5 -> z[n]
      adda #-z_exp_off,sp,r9                       ; r9 -> z_exp[0]

loopstart3

LLLABEL

      move.w d2,(r5)+
      loopend3

; fill in the z_exp array (r9) from the z array (r2).
; Cycle count ~ 4 + (T/2)*4 = 20 (worst case)



      tfra r14,r12         tfra r14,r13
      moveu.w (r2)+,r3     doen3 #T/2
      addl1a r3,r13        dosetup3 LOOP_ON_Z
      moveu.w (r2)+,r4
      moveu.w (r13),d3

      loopstart3

LOOP_ON_Z

      tfra r14,r12         tfra r14,r13
      moveu.w (r2)+,r3     addl1a r4,r12
      move.w d3,(r9)+      addl1a r3,r13
      moveu.w (r2)+,r4     moveu.w (r12),d4
      move.w d4,(r9)+      moveu.w (r13),d3

      loopend3

      move.w d4,(r9)+
      move.w d3,(r9)

; Forney_b

      adda #-4,sp,r6                       ; needed because of using
      move.w #0,(r6)                       ; software pipeline in kernel

      move.l #_exp_table_for_syndrome,r8   ; r8->exp_table_for_syndrome[0]

      adda #-z_exp_off,sp,r9               ; r9->z_exp[0]
      tfra r15,r14                         ; r14->elp[0]
      move.l (sp-rp_off),r11               ; r11->rp[0]
      adda #-t_nomden_off,sp,r3            ; r3->t_nomden[0]
      tfra r3,r13                          ; r3,r13 -> t_nom_den[0]
      adda #2,r14                          ; r14->elp[1]

      move.w #(N+1),n0
      move.w #2,n1

      dosetup2 LOOP1
      dosetup3 LOOP2
      move.l (sp-n_off),d0
      doen2 d0

      moveu.l #_exp_2_bin_extended,d15
      tfra r8,r0                           ; r0->syndrome[0][0]
      moveu.w (r11)+,r7                    ; r7=rp[0]
```

```
      tfra r9,r10                          ; r10 ->z_exp[0]
      tfra r15,r2                          ; r2->elp[0]
      addl1a r7,r0                         ; r0->syndrome[0][rp[0]]

      moveu.w (r0)+n0,d8                   ; load syndrom[1][rp[0]]
      move.2w (r10)+,d10:d11               ; load y0_pow=z_exp[0]
                                           ; y1_pow=z_exp[1]

      move.w #(2*N+1),d0
      tfr d0,d1            tfr d0,d2

      loopstart2

LOOP1

    [ clr d3               clr d12
      clr d13
      nom=0,den=0
      moveu.w (r0)+n0,d9   doen3 #4               ; load table[1][rp[i]]
    ]

    [ add d8,d10,d4        add d9,d11,d5          ; pow_n0=x0_pow+y0_pow
      add #1,d12                                  ; pow_n1=x1_pow+y1_pow,nom=1
      move.2w (r10)+,d10:d11 tfra r14,r2          ; load y0_pow=z_exp[2]
                                                  ; y1_pow=z_exp[3],r2->elp[1]
    ]

    [ min d0,d4            min d1,d5              ; pow_n0=min(pow_n0,2*N+1)
                                                  ; pow_n1=min(pow_n1,2*N+1)
      moveu.w (r2)+n1,d7   moveu.w (r11)+,r7      ; load y_pow=elp[1],load
rp[i+1]
    ]

    falign
    loopstart3

LOOP2

;1

    [ add d15,d4,d4        eor d3,d12             ; nom^=nom
      add d8,d7,d6                                ; pow_d=x0_pow+y_pow
      moveu.b (r6),d3      moveu.w (r0)+n0,d8     ; load bin_2_exp[pow_d]
                                                  ; load
x0_pow=table[j+2][rp[i]]
    ]

;2

    [ add d15,d5,d5        eor d3,d13             ;
                                                  ; den=den^

      min d2,d6                                   ; pow_d=min(pow_d,2*N+1)
      move.l d4,r4         moveu.w (r0)+n0,d9 ;
                                                  ; load
x1_pow=table[j+3][rp[i]]
    ]

;3

    [ add d8,d10,d4        add d15,d6,d6          ; pow_n0=x0_pow+y0_pow
      move.l d5,r5         moveu.w (r2)+n1,d7     ; load y_pow=elp[j+3]
    ]

;4
```

```
    [ min d0,d4             add d9,d11,d5           ; pow_n0=min(pow_n0,2*N+1)
      move.l d6,r6          moveu.b (r4),d3         ; pow_n1=x1_pow+y1_pow
                                                    ; load bin_2_exp[pow_n0]
    ]

;5

    [ eor d3,d12            min d1,d5               ; nom^=nom,pow_n1=min
                                                    ; (pow_n1,2*N+1)
      moveu.b (r5),d3       move.2w (r10)+,d10:d11  ; load bin_2_exp[pow_n1]
                                                    ; load y0_pow=z_exp[j+4]
    ]                                               ; y1_pow=z_exp[j+5]

    loopend3

LABEL

    [ eor d3,d12                                    ; nom^=nom
      moveu.b (r6),d3       tfra r8,r0              ; load bin_2_exp[pow_d]
                                                    ; r0->table[0][0]
    ]

    [ eor d3,d13                                    ; den=den^
      addl1a r7,r0          tfra r9,r10             ; r0->table[0][rp[i]]
                                                    ; r10->z_exp[0]
    ]

    [ move.2w d12:d13,(r3)+ moveu.w (r0)+n0,d8      ; store t_nomden[2*i+0]=nom
                                                    ; t_nomden[2*i+1]=den
                                                    ; load table[0][rp[i]]
    ]

       move.2w (r10)+,d10:d11 adda #-4,sp,r6        ; load y0_pow=z_exp[0]
                                                    ; y1_pow=z_exp[1]

    loopend2

; Forney_c

      move.l #255,d13
      move.l #511,d0
      move.l #_bin_2_exp,d12
      move.l #_exp_2_bin_extended,d14
      move.l (sp-r_off),r15
                                                    ; r13->t_nomden[0]
      move.l (sp-err_locs_off),r1
      move.l (sp-n_off),d10
      zxt.b d10,d10

; Overhead before loop

FORNEY_C

      moveu.w (r13)+,d1     tfra r15,r14            ; d1 -> nom[0]
      moveu.b (r1)+,r8      moveu.w (r13)+,d2       ; d2 -> denom[0]
      asll #1,d1            asll #1,d2

    [ add d12,d1,d1         add d12,d2,d2
      adda r8,r14
    ]

      move.l d1,r3          move.l d2,r4            ; ptr to table
      moveu.w (r13)+,d1     moveu.b (r14),d15       ; d1 -> t_nomden[1]
                                                    ; d15 -> received[err_loc[0]]
      moveu.w (r13)+,d2
```

```
    [ asll #1,d1            asll #1,d2
      moveu.w (r3),d5       moveu.w (r4),d6         ; d5 -> pwr of nom[0]
    ]                                               ; d6 -> pwr of denom[0]

    [ sub d6,d13,d6         add d12,d1,d1           ; d6 -> 255-pwr of denom[0]
      add d12,d2,d2
    ]

    [ add d5,d6,d4                                  ; pwr of error[0]
      move.l d1,r3          move.l d2,r4
    ]

      min d0,d4             dosetup3 LOOP_ON_ERRORS


    [ add d14,d4,d4                                         ; ptr to location in
table
      moveu.w (r3),d5       moveu.w (r4),d6                 ; d6 for 1-st
iteration ready
    ]

    [ sub d6,d13,d6
      move.l d4,r6                                          ; d5 for 1-st
iteration ready
    ]

      doen3 d10

      moveu.b (r6),d4                                       ; d4 for 0-iteration
ready

      loopstart3

LOOP_ON_ERRORS

    [ add d5,d6,d4          eor d4,d15
      moveu.w (r13)+,d1     moveu.b (r1)+,r8
    ]

    [ min d0,d4             asll #1,d1
      move.b d15,(r14)      moveu.w (r13)+,d2
    ]

    [ asll #1,d2
      tfra r15,r14
    ]


    [ add d12,d1,d1         add d12,d2,d2
      add d14,d4,d4
      adda r8,r14
    ]

      move.l d1,r3          move.l d2,r4
      move.l d4,r6
      moveu.w (r3),d5       moveu.w (r4),d6

    [ sub d6,d13,d6
      moveu.b (r6),d4       moveu.b (r14),d15
    ]

      loopend3

END_FORNEY
```

```
        adda #-allocation,sp,r6
        tfra r6,sp

        pop r6                  pop r7
        pop d6                  pop d7
        rts

        global Fforney_end

Fforney_end

TextEnd_forney
        endsec
```

NOTES:

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

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

Document Order No.: AN2407
Rev. 1
12/2004