Optimizing a Reed-Solomon Decoder for the Texas Instruments TMS320C62x DSP

by

Kamal Swamidoss

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 8, 1998

Copyright 1998 Kamal Swamidoss. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science
May 1, 1998

Certified by

V. Michael Iove, Jr.
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
ABSTRACT

Reed-Solomon is a family of block forward-error-correction codes used to facilitate robust digital communications. Reed-Solomon codes are used in many communications and storage/retrieval systems today, including the compact disc, satellites, space probes, cellular digital, asymmetric digital subscriber loops, and digital television. Reed-Solomon decoding is a computationally intense process which is generally implemented on application-specific integrated circuits (ASIC's). ASIC's provide high performance, but they are difficult and expensive to design. Digital signal processors (DSP's) provide a friendlier and more economical development platform, but they are generally slower than ASIC's. Texas Instruments recently introduced the fastest digital signal processors to date: the TMS320C62x (C62x) line. The C62x was designed for high-performance telecommunications applications. It offers an advanced instruction set architecture and powerful, user-friendly development tools. The C62x can potentially implement high-throughput Reed-Solomon decoding. This project is a series of C62x-specific optimizations of an existing C-language Reed-Solomon decoder. The goal was to improve the decoder throughput. Various difficulties were encountered and overcome while modifying the original decoder. The final modified decoder is twice as fast as the original.

Thesis Supervisor: V. Michael Bove, Jr.
Title: Principal Research Scientist, M.I.T. Media Laboratory
Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Basic Communication System</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Tables of GF(2) Arithmetic</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Tables of Addition and Subtraction in GF(2^3)</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Tables of Multiplication for Two Galois Fields of Size 2^3</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Various Representations of (Non-Zero) Elements of a GF(2^3)</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Communication System in Detail</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>Reed-Solomon Encoding</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>Reed-Solomon Decoding</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>Euclid Example Using Clark</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>Euclid Example Using Wicker</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td>C62x ADD Examples</td>
<td>18</td>
</tr>
<tr>
<td>12</td>
<td>Some C62x Assembly Instructions</td>
<td>19</td>
</tr>
<tr>
<td>13</td>
<td>Hand-Written C62x Regular Assembly Code</td>
<td>24</td>
</tr>
<tr>
<td>14</td>
<td>Hand-Written C62x Straight Assembly Code</td>
<td>25</td>
</tr>
<tr>
<td>15</td>
<td>C62x Assembly-Optimizer Output</td>
<td>27</td>
</tr>
<tr>
<td>16</td>
<td>AddExample Cycle Counts</td>
<td>29</td>
</tr>
<tr>
<td>17</td>
<td>Excerpt from RSDcodeTest Program</td>
<td>30</td>
</tr>
<tr>
<td>18</td>
<td>RSDcodeTest Program Flow</td>
<td>34</td>
</tr>
<tr>
<td>19</td>
<td>Example myrsusr.h File</td>
<td>35</td>
</tr>
<tr>
<td>20</td>
<td>Example myrssnd.h File</td>
<td>36</td>
</tr>
<tr>
<td>21</td>
<td>Example myrsrcv.h File</td>
<td>36</td>
</tr>
<tr>
<td>22</td>
<td>Example myrsend.h File</td>
<td>36</td>
</tr>
<tr>
<td>23</td>
<td>Example genrs Parameter File</td>
<td>37</td>
</tr>
<tr>
<td>24</td>
<td>Original Description of GFFourier() Function</td>
<td>40</td>
</tr>
<tr>
<td>25</td>
<td>Definition of GFFourierParameters Structure</td>
<td>40</td>
</tr>
<tr>
<td>26</td>
<td>Inner Loop of GFFourier() Function</td>
<td>41</td>
</tr>
<tr>
<td>27</td>
<td>Some C62x Assembly Instructions</td>
<td>41</td>
</tr>
<tr>
<td>28</td>
<td>GFFourier() Inner Loop Dependency Graph, 32-Bit Data</td>
<td>42</td>
</tr>
<tr>
<td>29</td>
<td>Hand-Written Software-Pipelined Regular-Assembly</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>GFFourier() Inner Loop, 32-Bit Data</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>RSDiscrepancy() Function</td>
<td>44</td>
</tr>
<tr>
<td>31</td>
<td>Some C62x Assembly Instructions</td>
<td>45</td>
</tr>
<tr>
<td>32</td>
<td>RSDiscrepancy() Loop Dependency Graph</td>
<td>46</td>
</tr>
<tr>
<td>33</td>
<td>Hand-Written Software-Pipelined Regular-Assembly</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>RSDiscrepancy() Loop</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>GFFourier() Inner Loop Dependency Graph, 16-Bit Data</td>
<td>50</td>
</tr>
<tr>
<td>35</td>
<td>Hand-Written Software-Pipelined Regular-Assembly</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>GFFourier() Inner Loop, 16-Bit Data</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>workingStorage Array</td>
<td>53</td>
</tr>
<tr>
<td>37</td>
<td>A New Data Storage Format</td>
<td>54</td>
</tr>
<tr>
<td>38</td>
<td>Reed-Solomon Code Parameters</td>
<td>56</td>
</tr>
<tr>
<td>39</td>
<td>Description of Terms</td>
<td>57</td>
</tr>
<tr>
<td>40</td>
<td>Average Cycle Counts Obtained</td>
<td>58</td>
</tr>
<tr>
<td>41</td>
<td>Rough Estimates of Throughput</td>
<td>59</td>
</tr>
</tbody>
</table>
Background

Communications
Reed-Solomon error correction is used to facilitate robust communication of digital data in radio and storage/retrieval systems. The following figure depicts the basic communication system. The basic storage/retrieval system is similar.

![Basic Communication System Diagram]

Figure 1: Basic Communication System

The sender and the receiver are connected by the communication channel. In the more general case, the user data begins as an analog signal; the digital communication system is actually the subsystem between the A/D block and the D/A block above. The analog input signal is first converted to a digital sequence by an analog-to-digital converter (A/D). The digital sequence can then be processed, e.g., compressed and/or error-correction encoded. The processed sequence is transmitted. In radio communications, this involves converting the sequence into an analog signal, modulating that signal, and transmitting it. The transmitted signal travels through the communication channel. The signal is received by the receiver. In radio communications, reception involves demodulating the received signal and converting the result into a digital sequence. The digital data can be processed, e.g., decoded and/or decompressed. If necessary, the processed sequence can be converted into an analog signal by a digital-to-analog converter (D/A). The digital data can be corrupted in any stage of communications, both in the analog and digital domains. Reed-Solomon error-correction coding is used to overcome the effect of corruption in the transmission, communication, and reception blocks, above.
In radio communications channels, corruption includes channel noise and interference from other transmissions. In storage/retrieval systems, this includes physical damage to, or deterioration of, the storage medium. Communication hardware corrupts data as well, during digital-to-analog conversion, modulation, demodulation, and analog-to-digital conversion. There are at least three ways to overcome signal corruption:

1. **Raise Signal Power**

   Raising signal power reduces the effect of channel noise. However, there are disadvantages. For example, in radio communications, if every broadcaster in a band raises the power of his/her signal, then the noise floor in the band increases from interference. The noise floor in adjacent bands can also go up, since real band-pass filters are non-ideal. In addition, the hardware required to transmit a more powerful signal is necessarily more expensive.

2. **Backward-Error-Correction**

   At the sender, an encoder computes a parity for the user data. The sequence of user data bits and parity bits is converted to an analog signal and transmitted. At the receiver, the signal is converted back to bits. A decoder uses the received parity and user bits to determine if the data was corrupted in transit. If an error is detected, the receiver requests that the data be retransmitted. Note that error detection is performed at the receiver, and that error correction is actually retransmission by the sender. The next block of bits is transmitted only when the current block is transmitted without error. Calculating the parity is relatively simple. A small number of parity bits is required for error detection, so user data throughput can be high. The downside is that backward-error-correction may not always work; if the system (transmitter-channel-receiver) is consistently noisy, then perfect transmission is impossible. In that case, the receiver continually requests retransmission, and communication fails.
3. **Forward-Error-Correction (FEC)**

Forward-error-correction is more robust. The receiver performs the error detection and correction. At the sender, an encoder computes a different kind of parity on the user data. The bits are converted and transmitted. At the receiver, the signal is converted back to bits. A decoder processes the received bits to determine if they were corrupted, and if so, the decoder attempts to remove the corruption. If the corruption is too severe, the decoder declares failure and the receiver requests retransmission. The difference between backward-error-correction and forward-error-correction is in the kind of parity information computed. FEC encoding and decoding are more computationally intense, and generally more parity bits are computed, so immediate throughput is lower. However, perfect transmission is not a requirement of FEC, so overall throughput can be acceptable in consistently noisy channels. In summary, if the right FEC scheme is chosen for a given system, robust communications can be achieved, providing high overall throughput at a moderate computing cost.

Reed-Solomon is actually a family of FEC codes. Several parameters make each RS code unique. One such parameter is the Galois field on which the code is based.

**Galois Fields**

In Reed-Solomon encoding and decoding, at an abstract level, data are not treated as collections of bits; they are treated as *Galois field* elements. Wicker states the definition of a field [27]. In practical terms, a field is a set of objects on which addition and multiplication are specially defined. Galois fields are fields with a finite number of elements. They are also called finite fields. Rowlands provides a clear description of the properties of Galois Fields.

The Galois fields most commonly used in RS are extensions of a base field. This field is denoted GF(2). It contains two elements, which can be represented as 0 and 1. Addition and subtraction of the elements of GF(2) correspond to binary XOR. Multiplication corresponds to binary AND. The non-zero element
(one) has a multiplicative inverse (itself), and division is defined as multiplication by the inverse. The following tables summarize arithmetic in GF(2):

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>-</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>×</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>+</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0+0</td>
<td>–</td>
</tr>
<tr>
<td>0+1</td>
<td>0</td>
</tr>
<tr>
<td>1+0</td>
<td>–</td>
</tr>
<tr>
<td>1+1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2: Tables of GF(2) Arithmetic

One parameter of an extension field of GF(2) is m. Extension fields of GF(2) are denoted GF(2^m). For a given m, there are many different extension fields. Each one has 2^m elements. Each of the 2^m elements of a GF(2^m) can be thought of as an (m-1)-degree binary polynomial in some dummy variable x. Each coefficient of the polynomial is one bit, which represents an element of GF(2). Thus, m-bit numbers can be thought of as elements of a GF(2^m). The following example of polynomial and binary representations are from a GF(2^4):

\[ 1011 \leftrightarrow x^3 + 0x^2 + x + 1 \] (1)

In order to generate a field, one must specify not only elements, but also arithmetic on those elements. For a given m, an extension field is uniquely defined by its arithmetic. For all extension fields, addition or subtraction is performed on elements in polynomial form; the respective coefficients are added or subtracted. This is simply addition or subtraction of elements of GF(2). Note that addition and subtraction of GF elements are closed.

The following tables depict addition and subtraction in GF(2^2). Note that the tables are identical.

<table>
<thead>
<tr>
<th>+</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>00</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>10</td>
<td>01</td>
<td>00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>-</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>00</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>10</td>
<td>01</td>
<td>00</td>
</tr>
</tbody>
</table>

Figure 3: Tables of Addition and Subtraction in GF(2^2)
Multiplication and division can be thought of as polynomial multiplication and division \textit{modulo an irreducible (in $GF(2)$) polynomial of degree $m$}. An irreducible polynomial is a polynomial which cannot be factored into smaller polynomials. The following is an example of multiplication in $GF(2^3)$.

\[
101 \times 011 \equiv (x^2 + 0x + 1) \times (0x^2 + x + 1) \% (x^3 + 0x^2 + x + 1)
\]  \hspace{1cm} (2)

The last term is an irreducible polynomial of degree 3. For a given $m$, there can be several irreducible polynomials, and each one generates a unique Galois field. Thus, the second and final parameter of an extension field is its irreducible polynomial (the first is $m$). Multiplication and division of GF elements, modulo an irreducible polynomial, are closed (except when dividing by zero). As the table to the left below shows, the product of the above multiplication is 100.

\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
\times & 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\
\hline
000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 & 000 \\
001 & 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\
010 & 000 & 010 & 100 & 101 & 110 & 111 & 001 & 011 \\
011 & 000 & 011 & 110 & 101 & 111 & 100 & 001 & 010 \\
100 & 000 & 100 & 011 & 110 & 010 & 101 & 001 & 111 \\
101 & 000 & 101 & 001 & 100 & 010 & 111 & 011 & 100 \\
110 & 000 & 110 & 111 & 001 & 101 & 011 & 010 & 100 \\
111 & 000 & 111 & 101 & 010 & 001 & 110 & 100 & 011 \\
\hline
\end{tabular}

The table on the left was generated using the irreducible polynomial $x^3 + 0x^2 + x + 1$, which is represented as 1011. The table on the right was generated using the irreducible polynomial $x^3 + x^2 + 0x + 1$, which is represented as 1101. Note that several multiplications result in different products in the two tables. The multiplicative inverse of an element can be found by identifying the multiplication which produces the identity element, 001. Tables of division can then be readily obtained.

Another useful representation of elements of $GF(2^m)$ is the power representation. Elements can be represented by integers corresponding to powers of a \textit{primitive element} of the field. The defining property of a primitive element $\alpha$ of $GF(2^m)$ is that $2^m - 1$ consecutive powers of $\alpha$ make up all non-zero elements of the field. Every extension field has at least one primitive element, so every element of any
extension field has a log. An extension field can have more than one primitive element, but one primitive element should be used consistently when taking logs and antilogs.

The following is a list of power representations of elements of $\text{GF}(2^m)$, using the irreducible polynomial $x^3 + 0x^2 + x + 1$. The primitive element used here (as shown) is 010.

<table>
<thead>
<tr>
<th>Power Representation</th>
<th>Value</th>
<th>Binary Representation</th>
<th>Polynomial Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\alpha^0$</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$\alpha^1$</td>
<td>010</td>
<td>$x$</td>
</tr>
<tr>
<td>2</td>
<td>$\alpha^2$</td>
<td>100</td>
<td>$x^2$</td>
</tr>
<tr>
<td>3</td>
<td>$\alpha^3$</td>
<td>011</td>
<td>$x + 1$</td>
</tr>
<tr>
<td>4</td>
<td>$\alpha^4$</td>
<td>110</td>
<td>$x^2 + x$</td>
</tr>
<tr>
<td>5</td>
<td>$\alpha^5$</td>
<td>111</td>
<td>$x^2 + x + 1$</td>
</tr>
<tr>
<td>6</td>
<td>$\alpha^6$</td>
<td>101</td>
<td>$x^2 + 1$</td>
</tr>
</tbody>
</table>

Figure 5: Various Representations of (Non-Zero) Elements of a $\text{GF}(2^3)$

In summary:

1. Bits can be used to represent elements of $\text{GF}(2^m)$.
2. RS encoding and decoding are performed on elements of $\text{GF}(2^m)$.
3. RS encoding and decoding can be performed on computers.
4. The error-correction capabilities of Reed-Solomon can be used in digital communication.

**Reed-Solomon**

Reed-Solomon is a family of block FEC codes. In block forward-error-correction, user data is processed as symbol-blocks; the user data bitstream is first broken into consecutive blocks of symbols, and each block is processed independently by the encoder. User data blocks are encoded into codewords at the sender, and codewords are decoded back into blocks at the receiver. Rowlands provides a clear description of Reed-Solomon.
Note the locations of the Reed-Solomon encoding and decoding blocks in relation to other blocks in Figure 6. Error-correction-encoding is the final stage of digital processing at the sender. For this reason, error-correction decoding is the first stage of digital processing at the receiver. In choosing a digital error-correction scheme, the goal is to minimize the effect of corruption in the stages between encoding and decoding (this includes corruption during transmission, communication, and reception), without sacrificing too much user data throughput.

The following parameters completely specify an RS code:

- \( m \) – The number of bits per symbol. Each symbol can be thought of as an element of a \( \text{GF}(2^m) \).
- \( t \) – The maximum number of correctable symbol errors.
  
  Note that in Reed-Solomon, corruption is modelled as symbol errors; a single bit error is considered a full symbol error, and several bit errors in the same symbol are considered one symbol error. This is because of the way RS processes symbols. At the receiver, if the number of detected symbol errors is greater than \( t \), then the codeword cannot be correctly decoded, and the data must be retransmitted.
- \( K \) – The number of symbols per user data block. \( K + 2t \) must be less than \( 2^m \).
- \( g \) – The irreducible polynomial.
  
  This polynomial is used to generate the extension Galois field on which the RS code is based.
- $\alpha$ – A primitive element of the Galois field. This parameter is used as the base for the GF log operations in the RS code.

- $m_0$ – The log of the first root of the generator polynomial $G(x)$. The significance of this value is explained below.

- $N$ – The number of symbols in the RS codeword. This number is $2^m - 1$.

Reed-Solomon is a popular FEC choice because it is easy to implement, and because it is effective in many real-world systems. RS is used in satellites, space probes, the Compact Disc, cellular digital, ADSL, and digital television.

**RS Encoding**

Before encoding, the user data bitstream is broken into blocks of symbols. Each block contains $K$ symbols, and each symbol consists of $m$ bits. Each symbol can be considered an element of $\text{GF}(2^m)$. At the sender, the encoder computes a sequence of $2t$ parity symbols for each block of user data symbols. The parity symbols and user data symbols together are called a codeword. Figure 7 depicts blocking and Reed-Solomon encoding. Each $K$-symbol user data block is encoded into a $(K+2t)$-symbol codeword.

As described above, it is sometimes useful to treat elements of an extension Galois field as binary polynomials. At a higher level, the user data block itself can be considered a polynomial, of degree $K-1$, whose coefficients are the symbols. The user data polynomial is denoted $D(x)$. The transmitted codeword, received codeword, and decoded user data block can similarly be considered polynomials.
In RS encoding, $D(x)$ is multiplied by the generator polynomial $G(x)$ to obtain the codeword polynomial $C(x)$. $G(x)$ is a parameter of the RS code. It can be written as follows:

$$G(x) = (x - \alpha^{m_0})(x - \alpha^{m_0+1})\cdots(x - \alpha^{m_0+2t-2})(x - \alpha^{m_0+2t-1})$$ (3)

The roots of $G(x)$ are $2t$ consecutive powers of $\alpha$. $\alpha^{m_0}$ is the first root of $G(x)$. $m_0$ is also a parameter of the RS code (as described above).

Thus, the encoder only generates polynomials which are multiples of $G(x)$. These are termed “correct codewords.” The sender only transmits correct codewords. If the receiver receives a codeword polynomial which is not a multiple of $G(x)$, the decoder can be sure that the polynomial was corrupted during communication, and it can begin error correction. Although it is possible for one correct codeword to be corrupted into another correct codeword during communication, the event is highly unlikely, because correct codewords are so “distant.” In fact, if the RS code is chosen properly, a corrupt codeword is hardly ever even corrected into a correct codeword that is different from the transmitted codeword.

The simplest way to satisfy the encoding criterion is to multiply $D(x)$ by $G(x)$. Because $D(x)$ has degree $K-1$ and $G(x)$ has degree $2t$, this will result in a polynomial of correct degree. However, a different formula is often implemented.

$$C(x) = D(x) \cdot x^{2t} - [D(x) \cdot x^{2t} \mod G(x)]$$ (4)

In this format, the first $K$ coefficients of $C(x)$ are the coefficients of $D(x)$, and the last $2t$ coefficients are the parity symbols. This is useful at the receiver, because it allows the user data to be obtained quite easily from the corrected codeword. RS codes which use this format are called systematic RS codes.
**RS Decoding**

At the receiver, the decoder tries to determine the transmitted codeword by correcting the received codeword. Depending on the severity of the corruption, the decoder can successfully reconstruct the transmitted codeword. Overall, this results in a reduction in (costly) retransmission. The user data is obtained from the corrected codeword.

![Diagram of Reed-Solomon Decoding](image)

Figure 8: Reed-Solomon Decoding

The received RS codeword can be treated as a polynomial, denoted $R(x)$. The relationship between the transmitted codeword, the effective digital corruption, and the received codeword is as follows:

$$R(x) = C(x) + E(x)$$  \hspace{1cm} (5)

$E(x)$ is the error polynomial. $E$ summarizes the effect of all the noise on the transmitted codeword. In order for RS decoding to work, $E$ can have at most $t$ non-zero coefficients. (That is the nature of RS error correction.)

In practice, $R(x)$ is used to obtain a syndrome polynomial $S(x)$, and the syndrome polynomial is used to determine $E(x)$. The following is a brief summary of the Petersen-Gorenstein-Zierler algorithm, the most common method of RS decoding, and the one implemented in the RS decoder modified in this project.

1. Treat the received codeword as a sequence of symbols, elements of the GF($2^m$) on which the RS code is based. The syndrome is a 2t-point Galois-field discrete Fourier transform of this sequence. The symbols in the syndrome sequence are also elements of the extension field. The Galois-field discrete Fourier transform is similar to the complex discrete Fourier transform, except that $\alpha$ is used instead of $e$. 


13
b) Treat the syndrome as a polynomial of degree 2t-1, denoted S(x). The zeroth-order coefficient of the polynomial is the first syndrome value, the first-order coefficient is the second value, and so on. The syndrome polynomial can thus be written

\[ S(x) = S_1 + S_2 x + S_3 x^2 + \ldots + S_{2t-1} x^{2t-1} \]

where the coefficients are the symbols in the syndrome sequence. Calculate an error locator polynomial \( \Lambda(x) \) using S(x). The error locator polynomial can be obtained using the Berlekamp-Massey algorithm or Euclid’s polynomial greatest-common-divisor (GCD) algorithm.

c) Find the roots of \( \Lambda(x) \).

The roots identify the locations of the symbol errors in the received codeword. The inverse Galois-field discrete Fourier transform can be used to find the roots of \( \Lambda \).

d) Calculate an error evaluator polynomial, denoted \( \Omega(x) \), using S(x) and \( \Lambda(x) \). The error evaluator polynomial can be obtained using Euclid’s algorithm.

e) Use \( \Omega \) and \( \Lambda \) to determine the magnitudes of the symbol errors.

These are the non-zero coefficients of \( E \). This is the Forney algorithm.

f) Subtract \( E(x) \) from \( R(x) \) to obtain \( C(x) \).

In systematic RS codes, the user data block can be readily obtained from \( C \).

[Rowlands, 18]

Decoding fails if there are more than \( t \) symbol errors. In that event the codeword must be retransmitted.

Reed-Solomon decoding is generally much more computationally intense than encoding.

One of the most efficient ways to find the error locator polynomial is the Berlekamp-Massey algorithm. Another way is Euclid’s algorithm, which finds not only the error locator polynomial, but also the error evaluator polynomial.

\[ S(x) = S_1 x + S_2 x^2 + S_3 x^3 + \ldots + S_{2t} x^{2t} \] [Wicker, 225].
Euclid's Algorithm
Euclid's greatest-common-divisor algorithm can be applied to polynomials whose coefficients are elements of GF(2^m). Implementations of Euclid's algorithm for RS decoding are generally less efficient than implementations of the Berlekamp-Massey algorithm, but the mechanics of Euclid are much easier to understand [Wicker, 225].

This description is based on Clark's interpretation of Euclid's algorithm, found on page 198. To obtain Λ and Ω, the algorithm is started on x^{2t} and S(x). The GCD of the two polynomials is not needed in RS decoding; the algorithm is only iterated until a special stopping condition is reached. At that point, two "intermediate values" provide Λ and Ω.

1. Set the following initial conditions:
   \[ r_{-1} = x^{2t} \]
   \[ r_0 = S(x) \]
   \[ r_{-1} = 0 \]
   \[ t_0 = 1 \]
   \[ i = 1 \]

2. Divide \( r_{i-2} \) by \( r_{i-1} \). The quotient is \( q_i \). The remainder is \( r_i \).

3. Obtain \( t_i \) using the following relation:
   \[ t_i = t_{i-2} - q_i t_{i-1} \]

4. If \( \deg(r_i) < t \) go to step 5. Otherwise increment \( i \) and go to step 2.

5. STOP.
   \[ \Lambda(x) = t_i(x) \]
   \[ \Omega(x) = r_i(x) \]
The notation may be confusing; the $t$ in step 4 is the maximum number of correctable errors, and the $t_i$ in the other steps are temporary polynomials. When the algorithm stops iterating, $\Lambda(x) = t_i(x)$ and $\Omega(x) = \tau_i(x)$.

Another interpretation of Euclid’s algorithm is given by Wicker, on page 225. It starts with two polynomials different from those in Clark’s interpretation, and it specifies a different stopping condition. Both methods were implemented, and their results were compared with $\Lambda$ and $\Omega$ obtained from the unmodified RS decoder. It was determined that both methods provide $\Lambda$. However, Wicker’s implementation does not provide $\Omega$ in the form that the RS decoder expects, and it was not obvious how to transform $\Omega$ accordingly. It was decided that Clark’s method would be used in implementing Euclid’s algorithm for this project.

The following tables show a simple example of how $\Lambda$ and $\Omega$ are obtained using the two interpretations. The example comes from Wicker, 225. The parameters to Euclid’s algorithm are the two starting polynomials and the Galois field. In this example, the Galois field is a $\text{GF}(2^3)$ generated with the irreducible polynomial $x^3 + x + 1$. Symbols are shown in exponential form. The primitive element $\alpha$ is 010. The syndrome sequence is $\alpha^6, \alpha^3, \alpha^4, \alpha^3$.

<table>
<thead>
<tr>
<th>i</th>
<th>r</th>
<th>q</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>$x^{2t} = x^4$</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>$S(x) = \alpha^6 + \alpha^3 x + \alpha^4 x^2 + \alpha^3 x^3$</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$\alpha^4 + x + \alpha^6 x^2$</td>
<td>$\alpha^5 + \alpha^4 x$</td>
<td>$\alpha^5 + \alpha^4 x$</td>
</tr>
<tr>
<td>2</td>
<td>$\alpha^6 + x$</td>
<td>$\alpha^4 x$</td>
<td>$1 + \alpha^3 x + \alpha x^2$</td>
</tr>
</tbody>
</table>

Figure 9: Euclid Example Using Clark

In this case:

1. The starting polynomials are $x^{2t}$ and $S(x) = \alpha^6 + \alpha^3 x + \alpha^4 x^2 + \alpha^3 x^3$. 
2. The stopping condition is $\deg[r_1(x)] < t$ ($= 2$).

3. $\Lambda(x) = 1 + \alpha^2 x + \alpha x$

$$\Omega(x) = \alpha^6 + x$$

<table>
<thead>
<tr>
<th>i</th>
<th>$r$</th>
<th>$q$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>$x^{2t+1} = x^5$</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>$1 + S(x) = 1 + \alpha^6 x + \alpha^3 x^2 + \alpha^4 x^3 + \alpha^3 x^4$</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$\alpha^5 + x^2 + \alpha^6 x^3$</td>
<td>$\alpha^2 + \alpha^4 x$</td>
<td>$\alpha^2 + \alpha^4 x$</td>
</tr>
<tr>
<td>2</td>
<td>$1 + x + \alpha^3 x^2$</td>
<td>$\alpha^4 x$</td>
<td>$1 + \alpha^2 x + \alpha x^2$</td>
</tr>
</tbody>
</table>

Figure 10: Euclid Example Using Wicker

In this case:

1. The starting polynomials are $x^{2t+1}$ and $1 + S(x) = 1 + \alpha^6 x + \alpha^3 x^2 + \alpha^4 x^3 + \alpha^3 x^4$. Note that Wicker's polynomial representation of the syndrome sequence is different from Clark's.

2. The stopping condition is $\deg[r_1(x)] \leq t$.

3. $\Lambda(x) = 1 + \alpha^2 x + \alpha x$

$$\Omega(x) = 1 + x + \alpha^3 x^2$$

$\Lambda$ is the same in both cases. The $\Omega$ polynomials are different, and it was determined through several trials that there appears to be no simple relationship between them. It must be noted that by definition, the zeroth-degree term of $\Lambda$ must be 1, so it is sometimes necessary to scale the final $r_i$ (it was not necessary in this example).

**Texas Instruments TMS320C62x**

TMS320C62x is a family of general-purpose digital signal processors made by Texas instruments. They have a common instruction set and CPU architecture. "C62x" serves to identify any CPU in the family.
CPU
The C62x was introduced in early 1997. It is designed for use in high-throughput digital communications systems, such as cable modems, wireless base stations, and digital subscriber loops [TI_WWW]. The C62x features 1600 MIPS performance, eight independent functional units, a 32-bit address space, and powerful conditional execution. A word is 32 bits, a half-word is 16 bits, and a byte is 8 bits on the C62x.

The C62x has 32 general-purpose 32-bit registers. They are equally divided into an A side and a B side, and are labelled from 0 to 15 [TI_CIS, 2-2]. When writing C62x assembly, it is important to know that different instructions can access registers in different side combinations [TI_CIS, 2-5]. The following are some examples using the ADD instruction. The first two registers are the source registers and the third is the destination register. The semicolons begin comments, which the assembler ignores.

1  ADD  A0,A1,A2  ; valid, sources and destination from same side
2  ADD  A0,A1,B2  ; ERROR, destination from different side
3  ADD  A0,B1,B2  ; valid, second source and destination from same side
4  ADD  A0,B1,A2  ; valid, first source and destination from same side

Figure 11: C62x ADD Examples

The C62x accesses bytes using a 32-bit address. Memory can also be accessed as half-words and words. Data can be addressed indirectly, with or without an offset, from any of the 32 registers, and the address can be pre- or post-incremented or -decremented. Data can be addressed as bytes, half-words, or words. In the case of half-words, a 31-bit address is used, and in the case of words, a 30-bit address is used. [TI_CIS, 3-60]

Each C62x instruction is a 32-bit word. The CPU accesses instructions using a 30-bit address. Eight instructions are fetched from program memory at a time. Each group of fetched instructions is called a fetch packet. The instructions in each fetch packet are divided into execute packets. All the instructions in an execute packet are executed in parallel, by the different CPU functional units, and execute packets are executed in series. When all eight instructions in a fetch packet belong to the same execute packet, they are all executed in parallel [TI_CIS, 3-10]. If this is sustained, it corresponds to 1600 MIPS at a CPU.
frequency of 200 MHz. However, it is difficult to keep all eight functional units executing useful instructions at the same time, so it is often the case that during every cycle some functional units are executing NOP ("no-op," no operation).

Different instructions are executed on different functional units of the CPU. Some instructions can be executed on any of several units, allowing some programming flexibility. The programmer can either assign functional units to instructions when writing assembly, or he/she can let the C62x assembler make the assignments at assemble-time. The functional units are called .L1, .L2, .S1, .S2, .M1, .M2, .D1, and .D2. The four letters essentially correspond to different function sets, and the numbers serve to make each member of a pair uniquely identifiable. The following table (from TI_CIS, 3-5) lists some of the instructions which were used when hand-writing assembly for this project. The table also lists the functional units which can execute each instruction.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Functional Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDW/LDH</td>
<td>Load word/half-word from memory</td>
<td>.D1,.D2</td>
</tr>
<tr>
<td>MV</td>
<td>Move value from register to register</td>
<td>.L1,.L2,.S1,.S2,.D1,.D2</td>
</tr>
<tr>
<td>MVK</td>
<td>Move constant to register</td>
<td>.S1,.S2</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>.L1,.L2,.S1,.S2,.D1,.D2</td>
</tr>
<tr>
<td>ADDK</td>
<td>Add constant</td>
<td>.S1,.S2</td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
<td>.S1,.S2</td>
</tr>
<tr>
<td>CMPEQ</td>
<td>Compare equal</td>
<td>.L1,.L2</td>
</tr>
<tr>
<td>CMPGT</td>
<td>Compare greater than</td>
<td>.L1,.L2</td>
</tr>
<tr>
<td>CMPLT</td>
<td>Compare less than</td>
<td>.L1,.L2</td>
</tr>
<tr>
<td>SHL/SHR</td>
<td>Shift left/right</td>
<td>.S1,.S2</td>
</tr>
<tr>
<td>STW/STH</td>
<td>Store word/half-word to memory</td>
<td>.D1,.D2</td>
</tr>
<tr>
<td>XOR</td>
<td>Bitwise XOR</td>
<td>.L1,.L2,.S1,.S2</td>
</tr>
<tr>
<td>AND</td>
<td>Bitwise AND</td>
<td>.L1,.L2,.S1,.S2</td>
</tr>
</tbody>
</table>

Figure 12: Some C62x Assembly Instructions

Any instruction can be executed at every CPU cycle, but some instructions have latencies. Notable examples are the load and branch instructions (LDB, LDH, LDW, and B). The load instructions have four-cycle latencies, meaning data will be available in the destination register four CPU cycles after the load instruction completes. This latency is separate from the stalls that may be associated with accessing
memory; these four cycles are a CPU pipeline latency. The branch instruction has a five-cycle latency, meaning program execution branches five CPU cycles after the instruction completes. [TI_CIS, 3-9]

One way to circumvent the cost of branching is to use the conditional execution feature of the C62x. Any instruction can be executed conditionally based on the value in one of five registers: B0, B1, B2, A1, and A2. Conditional instructions can be placed inside branch delay slots. In some cases, conditional instructions can replace branches altogether. [TI_CIS, 3-13]

The effect of load and branch latencies can be diminished somewhat by efficiently using delay slots. Software-pipelining is a way of making the absolute most of delay slots. It is a method of scheduling instructions to use CPU resources optimally during every cycle in a loop. The goals of software-pipelining are to minimize load and branch latencies, and to minimize the size of the loop. To this end, instructions are placed in the execution pipeline in the most efficient order, and several instructions are executed in parallel during every CPU cycle.

Development Tools
Texas Instruments emphasizes that the C62x allows the applications engineer to focus development resources on software rather than hardware, thereby facilitating development and shortening time-to-market [TI_WWW]. To support this development style, software development tools are available for the C62x, including an ANSI C compiler-optimizer and a unique assembly optimizer.

C Compiler-Optimizer
Several C source code optimizations can be made by the C62x C compiler-optimizer. The compiler generates efficient object code, and in some simple cases, it generates optimal code. Intrinsic functions provide direct access to assembly instructions. Special preprocessor directives allow the developer to provide additional information about the source code to the compiler.
The C source code for the RS decoder in this project was compiled with the -o2 and -pm command-line options. According to the TMS320C62x Optimizing C Compiler guide, the following optimizations are made when the -o2 flag is used:

- Performs control-flow-graph simplification
- Allocates variables to registers
- Performs loop rotation
- Eliminates unused code
- Simplifies expressions and statements
- Expands calls to functions declared inline
- Performs local copy/constant propagation
- Removes unused assignments
- Eliminates local common expressions
- Performs software pipelining
- Performs loop optimizations
- Eliminates global common subexpressions
- Eliminates global unused assignments
- Converts array references in loops to incremented pointer form
- Performs loop unrolling

[TI_OCC, 3-2]

The major performance advantage comes from software-pipelining and other loop optimizations.

The -pm flag indicates that program-level optimization should be performed. When this flag is used, the compiler considers all the source files listed on the command-line at once [TI_OCC, 3-13].
Further optimizations can be made by the compiler, using the -o3 flag (which was not used in this project):

- Remove all functions that are never called
- Simplify functions with return values that are never used
- Inline calls to small functions
- Reorder function declarations so that the attributes of called functions are known when the caller is optimized
- Propagate arguments into function bodies when all calls pass the same value in the same argument position
- Identify file-level variable characteristics

[TI_OCC, 3-3]

Most of these optimizations either would not have improved cycle count, or were not applicable. One exception is the inlining of small functions. In this project, small functions (such as the GF arithmetic functions) were inlined using the C62x C inline keyword.

C62x C is a superset of ANSI C. Several special functions, called intrinsics, are recognized. Intrinsics correspond to C62x assembly instructions. They allow the C programmer to express certain operations efficiently and concisely. They operate on simple data. For example, to get the effect of the C62x assembly instruction ADD2, the function int _add2 (int src1, int src2) can be used. ADD2 adds the upper half-words and lower half-words of two words (a C int is represented in 32 bits, while a C short is represented in 16 bits); any overflow in the lower addition does not affect the upper addition. When _add2 is encountered in the C code, the compiler generates a corresponding ADD2 instruction in the output assembly. Using intrinsics in critical loops can improve the performance of code. A list of C62x intrinsics can be found in TI_OCC, 8-23.
Intrinsic functions were not used in this project. The 38 intrinsic functions were inspected and it was decided that none were readily applicable to Galois-field arithmetic, which is the processing performed in Reed-Solomon encoding and decoding.

In order to execute a software-pipelined loop, the trip count of the loop must be large enough to support the prolog. When making loop optimizations, the compiler and assembler usually generate object code for both a software-pipelined loop and a non-software-pipelined loop. The former is executed only when the trip count is large enough. A way to reduce object-code size in both C source and assembly source is to provide minimum trip count information to the compiler or assembler. The programmer writes the minimum trip count at the beginning of the loop. If this minimum is large enough to guarantee that the redundant loop will not be needed, the compiler/assembler suppresses generation of the redundant loop.

Minimum trip count information could not be provided to the compiler/assembler in this project. Because these RS functions were designed to process virtually any practical RS code, it was not possible to guarantee that any critical loop would iterate a minimum number of times. In addition, the goal of the project was to reduce the CPU cycle count of the RS decoder; object-code size was not a consideration.

**Assembly Optimizer**

The assembly optimizer is an innovative, useful tool. Normally when writing assembly, the programmer must manually schedule instructions and allocate CPU resources. This process is especially difficult when programming for machines such as the C62x, which consists of several parallel functional units.

However, in addition to a regular assembler, the C62x comes with an assembly optimizer which can assume this responsibility. The assembly optimizer accepts a unique assembly format, called straight-assembly. This is assembly without scheduling or resource allocation. Functional units need not be assigned to instructions, and latencies should be ignored. Also, names can be given to register variables.
The assembly optimizer parses the straight-assembly and outputs regular assembly source, with scheduling, register allocation, and (optionally) an assembly interface to a C environment. The assembly optimizer can thus be used to generate C-callable assembly routines. The advantage to using the assembly optimizer over the C compiler-optimizer is that it can output faster object code. Also, for small routines, straight-assembly is as easy to write as C.

This section illustrates the use of the assembly optimizer. The assembly optimizer is described in detail in TI_OCC, Ch. 4. This is a hand-written (un optimized) regular assembly routine for vector addition.

Comments begin with a semicolon.

; Assembly routine to add two vectors of size elements.
; i and j are the input vectors.
; k is the output vector.
; Call this function from C.
; C function call: AddExample(size,i,j,k)

; The following lines are assembler directives.
; They "assign" variable names to registers during assemble-time.

i .set AO ; AO contains a pointer to i
j .set BO ; BO contains a pointer to j
k .set A2 ; A2 contains a pointer to k
t1 .set A1 ; temporary values
t2 .set B1
t3 .set A3
counter .set B2 ; counter

; The program starts here.
.text
.def _AddExample ; let C code see the routine
_AddExample:

; C calling convention!
; upon entering function:
MV A4,counter ; A4 contains arg1.
MV B4,i ; B4 contains arg2.
MV A6,j ; A6 contains arg3.
MV B6,k ; B6 contains arg4

AddLoop:
LDH *i++,t1 ; two load-half-word's
||
LDH *j++,t2 ; in parallel

[ counter] ADDK -1,counter ; conditional ADDK
[ counter] B AddLoop ; conditional branch
NOP 2 ; for load latency
This routine can be called from C by calling AddExample(size, source1, source2, dest). The C calling convention specifies that the four arguments be placed in registers A4, B4, A6, and B6. Lines 25-28 move the arguments into different registers. The add loop is lines 30-40. Two load-half-word’s (LDH) are performed in parallel to obtain the inputs (one LDH is performed by the .D1 unit and the other is performed by the .D2 unit). Note that the input pointers are post-incremented within the load instructions. The counter decrement and conditional branch are placed in two of the load instructions’ four delay slots. Once the values are available in registers, they are added (ADD) and stored (STH) at the next output address. The output pointer is incremented within the store instruction. The branch instruction in line 43 tells the CPU to return from the function call. The calling convention indicates that the return address is in register B3. The ADD and STH are placed in two of the branch instruction’s five delay slots. This ordering makes some use of the load and branch latencies.

The assembly optimizer reorders instructions even more efficiently, as the following straight-assembly listing shows. Note that the straight-assembly is generally much easier to write (and to read) than regular assembly.

```assembly
_addExample: .cproc counter,i,j,k ; C arguments
        .reg t1,t2,t3 ; automatic variables
        .trip 40 ; minimum trip count
        LDH *i++,t1
        LDH *j++,t2
        ADD t1,t2,t3
        STH t3,*k++
```
Line 6 tells the assembly optimizer the minimum trip count of the loop. The programmer supplies this information. The trip count of AddLoop will always be greater than 40. This lets the assembly optimizer make an object-code size optimization. If the listed trip count is less than the minimum trip count for software-pipelining, or if no trip count information is provided, then a redundant, non-software-pipelined loop is generated.

The .cproc directive tells the assembly optimizer that the _AddExample routine is to be C-callable (the name of the C function is then AddExample). When .cproc is used, the assembly optimizer outputs assembly code which can interface with a C environment [TI_OCC, 4-15, 4-20]. The arguments to .cproc are the parameters of the C function. Line 4 defines the other variables used in the routine. The .return directive at the end of the listing instructs the assembly optimizer to insert code at that point to return from the C function. The .endproc directive indicates the end of the function. Note how this listing differs from the regular assembly listing:

1. Variable names are used instead of CPU register names. This facilitates assembly programming, and allows the assembly optimizer to efficiently allocate registers to variables.
2. The load instructions in the straight-assembly are not placed in parallel (there is no | | before the second load instruction). The assembly optimizer will automatically place the loads in parallel in the regular assembly output.
3. The straight-assembly ignores the load and branch latencies. The assembly optimizer will schedule the regular assembly instructions properly.
The assembly optimizer determines the data dependencies and resource requirements in the straight-assembly listing, performs the instruction scheduling and resource allocation, and outputs the regular assembly. The following is an excerpt from the assembly optimizer output given the above straight-assembly.

```assembly
1 ;***************************************************************
2 ;* GLOBAL FILE PARAMETERS                                 *
3 ;*                                                      *
4 ;* Architecture  : TMS320C6200                         *
5 ;* Endian       : Little                                *
6 ;* Memory Model : Small                                *
7 ;* Redundant Loops : Enabled                            *
8 ;* Pipelining   : Enabled                               *
9 ;* Debug Info   : Debug                                 *
10 ;*                                                      *
11 ;***************************************************************
12 FF .set A15
13 DF .set B14
14 SP .set B15
15 .file "adxmplsa.sa"
16 .def _AddExample
17 .sect ".text"
18 .align 32
19 .sym _AddExample, _AddExample, 36, 2, 0
20 .func 3
21 ;***************************************************************
22 ;* FUNCTION NAME: _AddExample                           *
23 ;*                                                      *
24 ;* Regs Modified  : A0,A1,A3,A4,A5,B4,B5,B6             *
25 ;* Regs Used      : A0,A1,A3,A4,A5,A6,B3,B4,B5,B6       *
26 ;***************************************************************
27 _AddExample:
28 ;** -----------------------------------------------------------*
29 ;_AddExample:       .cproc      counter,i,j,k
30 ;                  .reg        t1,t2,t3
31 ;                  .sym       counter,1,4,4,32
32 ;                  .sym       i,20,4,4,32
33 ;                  .sym       j,3,4,4,32
34 ;                  .sym       k,22,4,4,32
35 ;                  .line 1
36 ;                  MV   .L1 A4,A1
37 ;                  MV   .S1 A6,A3
38 ;                  .sym   t1,0,4,4,32
```
.sym t2,0,4,4,32
.sym t3,0,4,4,32

MVC .S2 CSR,B6
MV .L1X B6,A4
AND .L2 -2,B6,B5
MVC .S2 B5,CSR
SUB .L1 A1,3,A1

;** ---------------------------------------------
L2: ; PIPED LOOP PROLOG
  AddLoop: .trip 40
  LDH .D1 *A3++,A0 ;
  LDH .D2 *B4++,B5 ;
  ADDK .S1 0xffffffff,A1 ;
  [ A1] B .S2 L3 ;
  LDH .D1 *A3++,A0 ;
  LDH .D2 *B4++,B5 ;
  ADDK .S1 0xffffffff,A1 ;
  [ A1] B .S2 L3 ;
  LDH .D1 *A3++,A0 ;
  [ A1] B .S2 L3 ;
  LDH .D1 *A3++,A0 ;

;** ---------------------------------------------
L3: ; PIPED LOOP KERNEL
  ADD .L1X B5,A0,A5 ;
  [ A1] B .S2 L3 ;
  [ A1] B .S2 L3 ;
  LDH .D1 *A3++,A0 ;
  STH .D1 A5,*A4++ ;
  LDH .D2 *B4++,B5 ;
  ADDK .S1 0xffffffff,A1 ;

;** ---------------------------------------------
L4: ; PIPED LOOP EPILOG
  ADD .L1X B5,A0,A5 ;
  STH .D1 A5,*A4++ ;
  ADD .L1X B5,A0,A5 ;
  STH .D1 A5,*A4++ ;
  ADD .L1X B5,A0,A5 ;
  STH .D1 A5,*A4++ ;

;** ---------------------------------------------
MVC .S2 B6,CSR
.line 14
B .S1 L7
NOP 5

;** ---------------------------------------------

The straight-assembly output has more instructions than the hand-written regular assembly. As described above, only the software-pipelined loop is generated, because the listed minimum trip count was large enough to guarantee software-pipelined execution.

The loop prolog (starting at line 57) primes the software pipeline. The loop epilog (starting at line 88) executes the remaining ADD and STH operations. The loop itself is only two cycles (starting at line 77).

When we compare the loop to the eight-cycle loop of the hand-written regular assembly, we see that the assembly optimizer performed well in this example.

The following table lists some cycle counts of calls to the different implementations of the AddExample function. The important numbers are the coefficients of the \( n \) term in the complexity expressions. The assembly optimizer and compiler-optimizer were both able to bring that down to two. In this simple example, the compiler-optimized C function performed better than the straight-assembly routine. In general, the assembly optimizer will produce better results than the C compiler-optimizer.

<table>
<thead>
<tr>
<th>Cycle Counts</th>
<th>Number of Elements</th>
<th>Handwritten Assembly</th>
<th>Straight-Assembly</th>
<th>Unoptimized C</th>
<th>Compiler-Optimized C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>40</td>
<td>376</td>
<td>109</td>
<td>1374</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>736</td>
<td>189</td>
<td>2734</td>
<td>176</td>
</tr>
<tr>
<td>Complexity</td>
<td>16+9n</td>
<td>29+2n</td>
<td>14+34n</td>
<td>~16+2n</td>
<td></td>
</tr>
</tbody>
</table>

Figure 16: AddExample Cycle Counts
Software-pipelining can also be done by hand. Data dependency graphs must be drawn, and registers must be allocated to variables. The process is difficult, but it can sometimes produce better assembly than the assembly optimizer. Software-pipelining by hand was performed at various stages of this project.

**Generic C Reed-Solomon Encoder/Decoder**

In 1994 Jon Rowlands of Texas Instruments DSP Research and Development wrote a C library of functions for Reed-Solomon encoding and decoding. That source code is not publicly available. However, it was used as the basis for the work done in this project. This section describes the original source code. There are three basic programs: the encoder/decoder, an RS code generator, and a test program.

**Reed-Solomon Test Program**

The test program is used to test the validity of the RS encoding/decoding functions. It simulates communication of digital data through a noisy channel. It generates a user data block, encodes it, corrupts the codeword, decodes the corrupt codeword, and compares the final block to the original user data block.

Originally, the program randomly generated user data and randomly corrupted RS codewords; each symbol in a user data block was randomly generated, and the locations and magnitudes of the symbol errors in the received codeword were randomly determined. The errors were added to symbols in the transmitted codeword, and the result was the received codeword. Thus, the input to the RS decoder was essentially random data.

Decoding random data would have made debugging difficult. If the RS decoder (essentially) received a random codeword every time it was run, program errors could have been difficult to reproduce. The test program was modified. The following is an excerpt from the new RSDecodeTest program:

```c
1   void
2   ReadData(
3     RSCode * code,
```
int iteration
int i;

for (i=0; i<code->numberOfUserDataSymbolsInCodeword; ++i)
    userData[i] = myRSUsr[iteration][i];

for (i=0; i<code->numberOfSymbolsInCodeword; ++i) {
    transmittedMessage[i] = myRSSnd[iteration][i];
    receivedMessage[i] = myRSRcv[iteration][i];
}

int main(void) {
    int numberOfUncorrectedCodewords = 0;
    int numberOfErrorsCorrected;
    int numberOfErrorsUncorrected;
    int wasSuccessful;
    long i;

    for (i = 0; i < numberOfCodewordsToTest; i++) {
#if defined(ReadIncludeFiles)
    ReadData(
        &StandardRSCode,
        i);
#else
    GenerateUserData(
        &StandardRSCode,
        userData);
#endif
    RSEncode(
        &StandardRSCode,
        userData,
        transmittedMessage);
    CorruptMessage(
        &StandardRSCode,
        transmittedMessage,
        receivedMessage);
#endif /* if defined(ReadIncludeFiles) */
    RSDecode(
        &StandardRSCode,
        receivedMessage,
        correctedUserData,
        &numberOfErrorsCorrected,
        &numberOfErrorsUncorrected);
#if defined(WriteIncludeFiles)
    WriteData(
        &StandardRSCode,
        i,
        numberOfCodewordsToTest);
#endif /* if defined(WriteIncludeFiles) */
}
Figure 17: Excerpt from RSDecodeTest Program

1. **StandardRSCodes** is a structure containing various parameters of the RS code used (the RS encoder and decoder can be built to use any of several RS codes).

2. **GenerateUserData()** randomly generates a K-symbol user data block.

3. **CorruptMessage()** randomly corrupts the transmitted codeword into the received codeword. The locations and magnitudes of symbol errors are randomly determined.

4. **ReadData()** accesses the arrays myRSUsr, myRSSnd, and myRSRcv, which contain user data blocks, transmitted codewords, and received codewords, respectively.

5. **CompareData()** compares the decoded data to the original user data.

6. **WriteData()** writes the original user data block, the transmitted codeword, the received codeword, and the final data block to files. These files can be used in subsequent builds of the test program.

The test program has two basic modes of operation:

1. Randomly generate user data, encode it, randomly corrupt the codeword, and decode the corrupt codeword.
Decode a codeword that was read into memory from a file at compile-time. Compare
the corrected user data to a user data block that was also read into memory from a file
at compile-time.

In Mode 2, the generating, encoding, and corrupting operations are not performed. The user data,
transmitted codeword, and received codeword are placed in static arrays at compile-time. The data which
RSDecode() and CompareData() use were generated during a previous run of a different build of
the program. The data was saved to files during the previous run, and the files are included in the
program by the compiler when the program is rebuilt to run in Mode 2. The following block diagram
describes one iteration of the modified RSDecodeTest.
Two preprocessor values are used: `ReadIncludeFiles` and `WriteIncludeFiles`. They answer the questions in the block diagram. The term "ReadIncludeFiles" is misleading; the include files are not actually read at run-time; they are included at compile-time. At run-time, sections of the included data arrays are copied into `userData`, `transmittedMessage`, and `receivedMessage`, which
correspond to the user data block, the transmitted codeword, and the received codeword. There are four include files, called myrsusr.h, myrssnd.h, myrsrvc.h, and myrsend.h. Examples of these files can be found at the end of this section. They contain user data blocks, transmitted RS codewords, (corrupt) received RS codewords, and final data blocks, respectively. These files are generated by the program when the preprocessor value WriteIncludeFiles is defined. Thus, one build of the program can be used to generate reference data (the user data block, the transmitted codeword, and the received codeword), and another build can be used to perform only RS decoding on that reference data. The former mode was used on the Sun workstation and the latter was used to debug RS decoder modifications for the C62x.

Here are examples of include files generated by the test program. These arrays are presented for illustrative purposes only; they were not actually generated by the test program.

```c
RSSymbol myRSUsr[2][4] =
{ { Ox01, 0x02, 0x03, 0x04 },
  { 0x05, 0x06, 0x07, 0x08 } };
#define numberOflncludedCodewords 2;
```

Figure 19: Example myrsusr.h File

RSSymbol is a typedef in the Reed-Solomon function library. It is usually int or short. It is the data type of a Reed-Solomon symbol. myRSUsr is the name of the array of user data blocks. This example file contains two user data blocks, for an RS code in which K equals 4. The preprocessor variable numberOfIncludedCodewords is written at the end of the file myrsusr.h by the test program. It lets the compiler know that numberOfIncludedCodewords sets of data were successfully saved to the include files. Another preprocessor variable, numberOfCodewordsToTest, is defined in RSDecodeTest.c. This value specifies the number of iterations of the test program. If ReadIncludeFiles is defined, and numberOfCodewordsToTest is greater than
numberOfIncludedCodewords, the compiler exits, since there is not enough data in the include
files on which to run the test.

```
1  RSSymbol myRSSnd{2}[6] =
2    { 0x01, 0x02,
3      0x03, 0x04,
4      0x09, 0x0a },
5    { 0x05, 0x06,
6      0x07, 0x08,}
7      0x0b, 0x0c }; 
```

Figure 20: Example myrssnd.h File

This file contains data for two RS codewords. The first codeword in myRSSnd corresponds to the first
user data block in myRSUsr. Note that this (fictional) code contains two parity symbols per codeword, so
t equals one.

```
1  RSSymbol myRSRcv{2}[6] =
2    { 0x01, 0x02,
3      0x04, 0x04,
4      0x09, 0x0a },
5    { 0x05, 0x06,
6      0x07, 0x08,}
7      0x0b, 0x0d }; 
```

Figure 21: Example myrsrchv.h File

This file contains codewords which correspond to corruptions of the codewords in myrssnd.h. There is
one symbol error in each codeword in this file.

```
1  RSSymbol myRSUsr{2}[4] =
2    { 0x01, 0x02,
3      0x03, 0x04 },
4    { 0x05, 0x06,}
5      0x07, 0x08 }; 
```

Figure 22: Example myrsend.h File

The file myrsend.h can be used to manually verify that each received codeword was indeed successfully
decoded (though the compare function in RSDecodeTest also does this at each iteration).

Galois-Field Arithmetic Functions
These functions are used by the RS encoder and decoder to manipulate GF elements. Operations include
addition, subtraction, multiplication, and division. In GF(2^m), addition and subtraction of elements can
be performed by the bitwise XOR operation. In this implementation, multiplication and division are performed in the log domain. The logTable array contains GF logs and the antilogTable array contains GF antilogs. The base of the logarithm is $\alpha$, a primitive element of the Galois field and one of the parameters of the RS code.

**Reed-Solomon Encoder and Decoder**

These functions implement the encoding and decoding processes. The RS decoder is an implementation of the Petersen-Gorenstein-Zierler algorithm, described above. Different functions implement the different steps of the algorithm.

**Reed-Solomon Code Generator**

Only a part of the source code for Reed-Solomon is provided; the rest must be generated for a particular RS code. The RS code generator, genrs, generates files containing RS-code-specific source code. The input to genrs is a parameter file which completely specifies the RS code. Its output is a .h file and a .c file which complete the encoder/decoder source code for a particular RS code.

This is an example of a parameter file. It specifies the name of the RS code used, the number of bits per symbol, the maximum number of correctable errors, the number of symbols per user data block, the irreducible polynomial (in binary notation), the primitive element (also in binary notation), the log of the first root of the generator polynomial, and $N$.

```
name    = Standard
m       = 8
k       = 8
K       = 188
g       = 100011101
alpha   = 00000010
m0      = 0
N       = 255
```

*Figure 23: Example genrs Parameter File*

The source code in the output .c file does several things. It defines structures used by the Galois-field discrete Fourier transform and inverse GF DFT in the RS decoder. It makes the log and antilog tables using the $g$ and $alpha$ parameters. It defines storage arrays for use in various functions in the RS
decoder. Finally, it defines the RSCode structure. A pointer to this structure is passed to the
RSEncode() and RSDecode() functions. The RSCode structure contains the parameters of the RS
code used, pointers to the arrays logTable and antilogTable (which are the log and antilog tables),
pointers to the GFDFT and IGFDFT parameter structures, and pointers to the defined storage arrays.
Statement of Work
The C62x code profiler was used to identify the critical loops in the RS decoder. The most CPU cycles (by far) were taken by the function GFFourier(), which performs the Galois-field Discrete Fourier Transform and the IGFDFT. It was also determined that the discrepancy calculation function (RSDiscrepancy()) in the Berlekamp-Massey algorithm used a large proportion of CPU cycles.

The GFFourier() function is used twice in this implementation, once to compute the syndrome and once in the Chien search (in actuality, the inverse GF DFT is used in the Chien search, but the GFFourier() function performs this as well). Thus, it was determined that optimizing GFFourier() would significantly improve the cycle count of the decoder.

The first modification was a direct translation of the C function into regular assembly, by hand. The C calling convention was followed, and the resulting routine could be called from C source. The modification was transparent to the rest of the program. In order to obtain a performance measurement, ten randomly-generated user data blocks were encoded using a small (K = 47, m = 6, t = 8) RS code, the codewords were corrupted, and the corrupt codewords were decoded using the modified decoder. The assembly routine provided an enormous performance improvement. The modified decoder was then tested using 1000 codewords. It correctly decoded all codewords.

A similar procedure was performed with RSDiscrepancy(). The hand-written assembly for this function considerably improved the performance of the decoder, but the improvement was not as dramatic as that obtained with the first routine. The decoder with both assembly routines was tested using 1000 codewords. It correctly decoded all codewords.

Software-Pipelining
At this point, neither assembly implementation incorporated software-pipelining. The next modification was an implementation of GFFourier() with a software-pipelined inner loop.
Jon Rowlands describes the operation of GFFourier() as follows:

```c
/*
 * GFFourier
 * Calculate a number of consecutive points of the Fourier transform
 * or inverse Fourier transform of a sequence.
 * code - the description of the RS code
 * input - the input symbols, stored with element zero first.
 * output - the transformed output values, stored with the lowest
 * frequency element first.

 * The DFT equation is
 * output(j) += input(i) * alpha ^ index
 * where index =
 * startingIndex +
 * i * startingIndexStep +
 * j * indexStep +
 * i * j * indexStepStep
 */
```

Figure 24: Original Description of GFFourier() Function

The code argument of GFFourier() points to an RSCode structure, containing pointers to the logTable and antilogTable arrays, and other data, which are used by functions called by GFFourier().

As stated above, GFFourier() can be called with different parameters to take different DFT's. The structure which contains these parameters is GFFourierParameters. This is the definition of the GFFourierParameters structure:

```c
typedef struct {
    int numberOfOutputSymbols;
    RSParameter constantValue;
    RSParameter startingIndex;
    RSParameter startingIndexStep;
    RSParameter indexStep;
    RSParameter indexStepStep;
} GFFourierParameters;
```
The data types RSSymbol and RSLogSymbol are used to represent GF elements and logs, respectively, in the RS encoder and decoder. The base of the log is $\alpha$, a primitive element. This is the inner loop of GFFourier():

```c
for (j = 0; j < numberOfOutputSymbols; j++) {
    output[j] = 
    GFAdd(
        code,
        output[j],
        GFAntilog(code, index)
    );

    index = GFLogMultiplyLogLog(
        code,
        index,
        indexStep
    );
}
```

The loop can be executed by the following assembly instructions. Branch and load latencies are not considered here; this is merely a list of useful assembly instructions:

```assembly
; output1 = output2 = output
innerLoop:
    ADD index, indexStep, index
    CMPLT index.N, cond ; N is an element of RSCode
    [ cond] SUB index.N, index
    LDW *antilogTable[index], temp1
    LDW *output1++, temp2
    XOR temp1, temp2, temp2
    [ counter] ADDK -1, counter
    [ counter] B innerLoop
```

Figure 25: Definition of GFFourierParameters Structure

Figure 26: Inner Loop of GFFourier() Function

Figure 27: Some C62x Assembly Instructions
The following dependency graph was drawn for the inner loop, using the C source and assembly translation:

Figure 28: GFFourier() Inner Loop Dependency Graph, 32-Bit Data

The graph shows which instructions were used and how functional units (.L1, .L2, .S1, .S2, .D1, .D2) were allocated to instructions. CPU registers are allocated to variables, and the graph is divided into the

42
two sides of the CPU. An “X” in a functional unit allocation indicates the use of a data cross-path, from one side of the CPU to the other. The numbers show how many CPU cycles are required for the effects of instructions to occur. For example, the sum in an ADD instruction is available in the destination register at the next CPU cycle. At the top of the graph, adding indexStep and index requires one CPU cycle. The sum is placed in temp4. The loaded word in a LDW instruction is available in the destination register four CPU cycles after the instruction completes. Thus, the load-word instruction requires a total of five CPU cycles, because of the four-cycle latency.

Instructions can be scheduled such that the software-pipelined inner loop takes three cycles. This is done by placing one part of the loop path in parallel with another, independent part. Essentially, two different parts of two consecutive iterations of the loop are executed in parallel. This is the software-pipelined assembly listing of the inner loop of GFFourier():

```
1  ASMGFFourierLoop2Init:
2      MV      numberOfOutputSymbols,temp3
3      SUB     temp3,2,temp3 ; for software-pipelining
4      MV      A8,output1 ; A8 = output
5      MV      A8,output2
6
7  ASMGFFourierLoop2Prolog:
8      ADD     index,indexStep,index
9      | |       LDW  *antilogTable[index],temp1
10     | |       LDW  *output1++,temp2
11
12    CMPLT    index,N,temp4
13    |[ temp3]  ADDK  -1,temp3
14
15    |[ temp3]  B     ASMGFFourierLoop2
16    |[!temp4]  SUB     index,N,index
17
18    ADD     index,indexStep,index
19    | |       LDW  *antilogTable[index],temp1
20    | |       LDW  *output1++,temp2
21
22    CMPLT    index,N,temp4
23    |[ temp3]  ADDK  -1,temp3
24
25  ASMGFFourierLoop2:
26    XOR     temp2,temp1,temp2
27    |[ temp3]  B     ASMGFFourierLoop2
28    |[!temp4]  SUB     index,N,index
29
30    ADD     index,indexStep,index
31    | |       LDW  *antilogTable[index],temp1
```
The software-pipelining procedure was then followed for the discrepancy calculation. This is the original C RSDiscrepancy() function:

```c
STATIC RSSymbol
RSDiscrepancy(RSCode * code, int i, int errorLocatorDegree, const RSLogSymbol * logSyndrome, const RSLogSymbol * logErrorLocator
) {
RSSymbol discrepancy;
int j;
discrepancy = 0;
for (j = 0; j <= errorLocatorDegree; j++) {
    discrepancy = GFAdd(
        code,
        discrepancy,
        GFMultiplyLogLog(
            code,
            logErrorLocator[j],
            logSyndrome[i - j]
        )
    );
}
return(discrepancy);
}
```

Figure 29: Hand-Written Software-Pipelined Regular-Assembly GFFourier() Inner Loop, 32-Bit Data

Figure 30: RSDiscrepancy() Function
The for loop starting at line 14 can be software-pipelined. This is a list of useful assembly instructions.

In the assembly implementation, the logSyndrome pointer is moved forward \( j \) elements before the loop, and decremented at each iteration.

```assembly
1 ; logSyndrome = logSyndrome + i
2 ; discrepancy = 0
3 innerLoop:
4          LDW *logErrorLocator++, templ
5          LDW *logSyndrome--, temp2
6          ADD templ, temp2, temp3
7          LDW *antilogTable[temp3], temp4
8          XOR discrepancy, temp4, discrepancy
9 [ counter] ADDK -1, counter
10 [ counter] B innerLoop
```

Figure 31: Some C62x Assembly Instructions

The following dependency graph was obtained. Functional units are allocated, and the graph is divided into the two sides of the CPU:
The dependency graph shows that two log values are loaded from memory and added. The antilog of the sum is loaded from memory and XOR'ed with the discrepancy. There are three memory loads in each iteration, thus the software-pipelined loop requires at least two cycles (at most two memory loads can be performed during each CPU cycle, one by .D1 and one by .D2). Because of the two stages of memory loads, the software-pipelined discrepancy has a very large prolog and epilog. In the RS codes used to test modifications to this decoder, RSDiscrepancy() was rarely called with a trip count large enough to support software-pipeling, so the regular redundant loop was often used. In this function, the trip count is related to $t$. In the RS codes used, $t$ was always less than 8. If $t$ were 12, the software-pipelined loop...
would have been used more frequently. However, other issues (including the size of the logTable and antilogTable arrays, and the complexities of different parts of the decoder) prohibit increasing t.

This is the software-pipelined loop:

```
ASMRSDiscrepancySPLoopProlog:
   LDH *logSyndrome--, templ
   LDH *logErrorLocator++, temp2
   NOP
   LDH *logSyndrome--, templ
   LDH *logErrorLocator++, temp2
   NOP
   LDH *logSyndrome--, templ
   LDH *logErrorLocator++, temp2
   ADD templ, temp2, temp3
   [ counter] ADDK -1, counter
   || LDH *logSyndrome--, templ
   || LDH *logErrorLocator++, temp2
   ADD templ, temp2, temp3
   || [ counter] B ASMRSDiscrepancySPLoop
   || LDH '*'antilogTable[temp3], temp4
   [ counter] ADDK -1, counter
   || LDH *logSyndrome--, templ
   || LDH *logErrorLocator++, temp2
   ADD templ, temp2, temp3
   || [ counter] B ASMRSDiscrepancySPLoop
   || LDH '*'antilogTable[temp3], temp4
   [ counter] ADDK -1, counter
   || LDH *logSyndrome--, templ
   || LDH *logErrorLocator++, temp2
   ASMRSDiscrepancySPLoop:
   ADD templ, temp2, temp3
   || [ counter] B ASMRSDiscrepancySPLoop
   || LDH '*'antilogTable[temp3], temp4
   XOR discrepancy, temp4, discrepancy
   [ counter] ADDK -1, counter
   || LDH *logSyndrome--, templ
   || LDH *logErrorLocator++, temp2
   ASMRSDiscrepancySPLoopEpilog:
   ADD templ, temp2, temp3
```
Note the size of the prolog and the epilog. Note also that the loop consists of two CPU cycles.

**16-bit RSSymbol and RSLogSymbol**

To this point, 32-bit (full-word) representations of symbols had been used. However, most practical RS codes process symbols which can be represented in 16 bits (a half-word) or less. The C62x data memory could be used more efficiently by changing the representation of symbols to half-words. The necessary modifications were made and the memory benefits were seen immediately. Because memory loads have four-cycle latencies, it would be worthwhile to make the most of each memory load. Also, because the load-word instruction takes no longer to execute than the load-half-word instruction, it is possible to obtain a performance gain by using LDW to load and operate on two half-word symbols during each iteration of a loop. In order to separate the loaded word into individual half-words, the LDW instruction
should be followed by a 16-bit shift (to get the high half-word) executed in parallel with a 16-bit mask (to get the low half-word).

Because two different output values are computed at each iteration of the inner loop, essentially two separate sets of data registers must be maintained, and the program forks in the loop. Two index values must be updated, the loaded input word must be separated into two input half-words, two GF adds must be performed, and two output half-words must be stored back to memory. The following dependency graph was obtained for the inner loop of GFFourier() using double half-word loads.
Figure 34: GFFourier() Inner Loop Dependency Graph, 16-Bit Data

Note that the dependency graph of the new inner loop forks, each path being processed by one side of the CPU. In order to develop the new assembly implementation of GFFourier(), the inner loop was first written, and the rest of the routine was written around it. Because twice as many inputs are processed at
each iteration, the trip count of the inner loop was halved, but in the RS codes used in this project, 
GFFourier() was still always called with enough elements to use the software-pipelined loop.

Nevertheless, a regular redundant loop was written. Thus, with some effort, it became possible to obtain 
100% more outputs at each iteration of the inner loop of the new GFFourier(), with only 67% more 
cycles. The trade-off is register usage; many more registers must be used in the new implementation.

Two pointers to the antilogTable array and two N's are required (because of the side rules of the 
load and compare instructions). Two indexes must be maintained, as well as two input symbols and two 
conditional registers. Writing the rest of the routine to fit around this inner loop was more difficult, 
because the inner loop used so many registers.

This is the software-pipelined assembly listing of the inner loop of GFFourier() using double-half- 
word-loads.

```
1  ASMFFourierLoop2Init:
2       LDW    *+parameters[0],counter
3       MV     output,output1
4       MV     output.output2
5       NOP    2
6
7       CMPGT  counter,2,cond1
8        [!cond1] B  ASMFFourierLoop2NotSP
9        [ cond1] EEXTU  counter,31,31,cond2
10       [ cond1] MV     cond2,remainder
11       [ cond1] SHRU   counter,1,counter
12       [ cond1] ADDK   -1,counter
13     ; to count LDW in prolog
14       NOP
15
16  ASMFFourierLoop2Prolog:
17       LDW    *output1++,twoSymbols
18
19       ADD    index1,indexStepTwice,index1
20       ||      ADD    index2,indexStepTwice,index2
21       ||      LDH    *+antilogTable1[index1],antilog1
22       ||      LDH    *+antilogTable2[index2],antilog2
23       ||       CMPLT  index1,N1,cond1
24       ||       CMPLT  index2,N2,cond2
25       ||{ counter} ADDK   -1,counter
26       ||        SUB     index1,N1,index1
27       ||{!cond2} SUB     index2,N2,index2
```
One especially difficult aspect of implementing double half-word loads was the alignment of some data. The original program defined an array called `workingStorage`. This array was used by different functions to temporarily store arrays. One function which used the `workingStorage` array was...
GFFourier(). In both calls to this function in RSDecode(), the output sequences are to be placed in parts of workingStorage, and in one call, the input sequence is to be found in another part of workingStorage.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{workingStorage} & \text{syndrome} & \text{temp1} & \text{temp2} \\
\hline
\end{array}
\]

Figure 36: workingStorage Array

On the C62x, it is not possible to load just any two consecutive half-words using the load-word instruction; the half-words must be located in the same word. That is to say, the 30 most significant bits of the 31-bit addresses of the two half-words must be the same. Thus, if the arrays are not aligned properly in workingStorage, a double half-word load at the beginning or end of a sequence located within workingStorage could possibly load one invalid half-word. The program has no control over how arrays are aligned within workingStorage.

One solution is to align arrays during linking such that the first double-half-word-load always accesses two valid half-words, and to install an odd-ness check on the number of half-words to be loaded, treating a single half-word at the end of the array as a special case. This is the solution implemented for GFFourier(). The arrays of interest were defined such that they were individually alignable. Originally, functions were given pointers into workingStorage; these pointers corresponded to the beginnings of the arrays, but there was no guarantee that a pointer pointed to the beginning of a full-word. Now, functions are given pointers to the beginnings of independent arrays, which the linker automatically word-aligns.
It would not have been useful to software-pipeline `RSDiscrepancy()` with double data loads. The function was hardly ever called with a large enough trip count in the first place. Thus, in the case of `RSDiscrepancy()`, the data representations were simply changed from words to half-words. The RS decoder with the two 16-bit assembly routines was tested with 1000 codewords. All codewords were decoded correctly. The 16-bit routines actually performed slightly worse than the 32-bit routines. This was probably due to the overhead introduced in the 16-bit assembly implementation of `GFFourier()` by operating on two inputs at once.

**Euclid’s Algorithm in C**

Euclid’s greatest-common-divisor algorithm can be used to find the error locator polynomial and error evaluator polynomial in the Petersen-Gorenstein-Zierler algorithm. This algorithm was implemented, replacing the existing implementation of the Berlekamp-Massey algorithm.

First, the basic Galois-field polynomial arithmetic functions were written in C: `GFPolyXOR()`, `GFPolyMultiply()`, and `GFPolyDivide()`. Both addition and subtraction of elements of GF(2^m) correspond to bitwise XOR, so both operations are handled by the function `GFPolyXOR()`. After writing these functions, the `RSEuclid()` function was written. This function consists of some initializations, a loop with the GF polynomial arithmetic function calls listed in the right order, and a stopping condition.

The Euclid implementation was incorporated into the RS decoder. The program was tested on 1000 codewords. All codewords were correctly decoded.
**Assembly GF Polynomial Arithmetic**
The C implementation of Euclid’s algorithm was significantly slower than the implementation of the Berlekamp-Massey algorithm, even though the Euclid version computes both $\Lambda$ and $\Omega$. The functions were rewritten in regular assembly. Because these functions were always called with a small trip count, it was decided not to software-pipeline the loops. After much debugging, the assembly implementation of Euclid’s algorithm was verified.

**Straight-Assembly**
The normal development flow for the C62x is: ANSI C to C62x C to straight-assembly to regular assembly. When ANSI C functions are too slow, they are optimized with C62x intrinsics and trip count information. When C62x C functions are too slow, they are rewritten in straight-assembly which is given to the assembly optimizer. Only when assembly optimizer output is too slow should the developer start hand-writing regular assembly. The flow (generally) goes from most simple to implement to most difficult to implement, and from most inefficient code to most efficient code. Sometimes the performance improvement gained by hand-writing regular assembly is far outweighed by the difficulty of writing assembly. The assembly optimizer outputs very efficient code, and straight-assembly is relatively simple to write, so straight-assembly provides a near-ideal solution to writing assembly for the C62x.

Straight-assembly routines were written for GFFourier(), RSDiscrepancy(), GFPolyXOR(), GFPolyMultiply(), and GFPolyDivide(). (Note that {RSDiscrepancy()} and {GFPolyXOR(),GFPolyMultiply(),GFPolyDivide()} are mutually exclusive, because the first is used in Berlekamp’s algorithm and the others are used in Euclid’s algorithm.) Sometimes, the assembly optimizer generated more efficient regular assembly, given the same program flow. In these cases the output of the assembly optimizer was considered an upper limit on the performance improvement available from this optimization strategy.

The operation of the RS decoder was verified on two different RS codes. Several parameters are different among the RS codes. It was decided that the modifications were correct.
Observations
This section lists the cycle counts of different versions of the RS decoder. The versions are differentiated by their implementations of different stages. The C source was always compiled with the -o and -pm compiler flags (see Background). Unless otherwise noted, each version used 16-bit symbols and logs. In C, the data types RSSymbol and RSLogSymbol could be defined as short’s (16 bits) or as int’s (32 bits). The GFFourier assembly routines have software-pipelined inner loops. For the most part, the same ten sets of data were decoded by each version (see Background), and the cycle counts listed below are averages. However, ten sets of data could not be loaded into C62x memory to run with the versions using 32-bit symbols and logs; the data took too much memory. In those cases, the first five sets of test data were loaded and used. Each corrupt RS codeword had 8 symbol errors, the maximum number of correctable symbol errors for the RS code used. The locations and magnitudes of the errors were randomly-determined. These are the parameters for the RS code used:

- m = 8
- K = 188
- t = 8
- g = 100011101
- α = 00000010
- m₀ = 0
- N = 255

Figure 38: Reed-Solomon Code Parameters

Note that g and α are listed in the binary polynomial representation, with the highest-degree coefficients listed first. The rest of the parameters are in decimal notation. The numbers listed below are averages of the sums of the cycle counts for the following RS decoding operations, over ten (or five) codewords:

1. Calculating S(x).
2. Calculating Λ(x) and Ω(x).
3. Finding the roots of Λ(x).
4. Calculating the formal derivative of Ω(x).

This operation is used in finding the magnitudes of the symbol errors.
5. Finding the magnitudes of the symbol errors and subtracting the symbol errors from the received RS codeword.

The following operations are part of the RS decoder implementation, but their cycle counts are not included in the numbers listed below:

1. Copying the first K symbols of the (corrupt) received RS codeword to the `correctedUserData` array. Because a systematic RS code was used, the first K symbols of the received codeword form the basis of the corrected user data block. This copy operation is performed once, at the beginning of the RS decoding process.

2. Filling an array with the GF logs of the coefficients of S(x).

These logs are used in certain GF multiplication and division operations. The `logSyndrome` array is filled once, after S(x) is computed. This is done by looking up the GF log of each element in the `syndrome` array in the `logTable` array, and writing that value into the `logSyndrome` array.

This is a description of the terms used in this section.

<table>
<thead>
<tr>
<th>C GFFourier</th>
<th>Original C GFFourier() function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASM GFFourier32</td>
<td>Hand-written, hand-software-pipelined, C-callable C62x assembly routine, using 32-bit symbols and logs. This routine is a functional equivalent of GFFourier().</td>
</tr>
<tr>
<td>ASM GFFourier16</td>
<td>Hand-written, hand-software-pipelined, C-callable assembly routine using 16-bit symbols and logs, performing double half-word loads. This routine is a functional equivalent of GFFourier().</td>
</tr>
<tr>
<td>SA GFFourier</td>
<td>C-callable assembly output of assembly-optimizer, given hand-written straight-assembly. This routine is a functional equivalent of GFFourier().</td>
</tr>
<tr>
<td>C RSDS</td>
<td>C RSDiscrepancy() function.</td>
</tr>
<tr>
<td>ASM RSDS32</td>
<td>Hand-written, hand-software-pipelined, C-callable assembly routine using 32-bit symbols and logs. This routine is a functional equivalent of RSDiscrepancy().</td>
</tr>
<tr>
<td>ASM RSDS16</td>
<td>Hand-written, hand-software-pipelined, C-callable assembly routine using 16-bit symbols and logs. This routine is a functional equivalent of RSDiscrepancy().</td>
</tr>
<tr>
<td>SA RSDS</td>
<td>C-callable assembly output of assembly-optimizer, given hand-written straight-assembly. This routine is a functional equivalent of RSDiscrepancy().</td>
</tr>
<tr>
<td>C RSEuclid</td>
<td>C RSEuclid() function calling C functions for Galois-field polynomial arithmetic (XOR, multiply, divide).</td>
</tr>
<tr>
<td>ASM RSEuclid</td>
<td>C RSEuclid() function calling hand-written assembly routines for GF polynomial arithmetic.</td>
</tr>
<tr>
<td>SA RSEuclid</td>
<td>C RSEuclid() function calling C-callable assembly output of assembly-optimizer, given straight-assembly routines for GF polynomial arithmetic.</td>
</tr>
</tbody>
</table>
Each cell contains the average cycle count corresponding to a unique version of the RS decoder. Each version computes the Galois-field discrete Fourier transform using one of the four implementations listed above. The GFdft is used twice in the Petersen-Gorenstein-Zierler algorithm, once to compute the syndrome and once to find the roots of the error locator. In versions of the decoder using the Berlekamp-Massey algorithm (to find the error locator polynomial), the discrepancy is calculated using one of four implementations. In versions using Euclid’s algorithm, one of three sets of Galois-field polynomial arithmetic routines is used.

Cells along the row headed by ASM GFFourier32 and down the column headed by ASM RSDS32 contain cycle counts for the decoder using 32-bit representations of symbols and logs. Note the six cells without cycle counts; it is not possible to build versions of the decoder with certain combinations, because the assembly routines using 32-bit data are not compatible with the assembly routines using 16-bit data, and the C environment treats the data as either 16 bits or 32 bits.

The following table lists user data throughputs calculated for each program executing on a C62x running at 200 MHz. They describe the amount of user data decoded, in megabits per second. The numbers were obtained assuming that one CPU cycle corresponds to one clock cycle. This is not a valid assumption in practice, since CPU stalls are inevitable when accessing real memory (in these tests the cycle counts were obtained using C62x simulation software). The sequence of instructions, the storage of data in memory,
and the type of memory used all affect the performance of a program. Thus, the following throughputs are overly optimistic.

<table>
<thead>
<tr>
<th></th>
<th>Berlekamp-Massey</th>
<th></th>
<th>Euclid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C RSDS</td>
<td>ASM RSDS32</td>
<td>ASM RSDS16</td>
</tr>
<tr>
<td>C GFFourier</td>
<td>5.00</td>
<td>3.25</td>
<td>5.04</td>
</tr>
<tr>
<td>ASM GFFourier32</td>
<td>8.13</td>
<td>8.19</td>
<td></td>
</tr>
<tr>
<td>ASM GFFourier16</td>
<td>8.07</td>
<td></td>
<td>8.16</td>
</tr>
<tr>
<td>SA GFFourier</td>
<td>8.43</td>
<td></td>
<td>8.54</td>
</tr>
</tbody>
</table>

Figure 41: Rough Estimates of Throughput

Conclusions
The results from the previous section can be used to make certain conclusions about the performance of the different versions of the RS decoder:

1. Almost every modification made to the existing all-C RS decoder resulted in a reduction in cycle count. The exceptions are the combination of C GFFourier and ASM RSDS32 and the combination of C GFFourier and ASM RSEuclid.

2. The highest-performance combination (SA GFFourier and ASM RSDS16) provides user data throughput of about 8.5 megabits per second, or about 9.3 Mb/s total throughput. The all-C compiler-optimized implementation using the Berlekamp-Massey algorithm (C GFFourier and C RSDS) provides about 5.0 Mb/s user data throughput, or about 5.4 Mb/s total throughput. The all-C compiler-optimized implementation using Euclid’s algorithm (C GFFourier and RSEuclid) provides about 4.8 Mb/s user data throughput, or about 5.2 Mb/s total throughput. One set of modifications provide about 70% higher throughput than the fastest all C compiler-optimized code.

3. Apparently, the C compiler makes object code that handles 32-bit symbols and logs very inefficiently. The C GFFourier() function with 32-bit symbols and logs is the worst performer. This could be confirmed by comparing results from more test cases using this RS code, and by testing cases using larger RS codes.
4. Comparing numbers in any given row, it appears that ASM RSDS16 is the fastest discrepancy calculation. It also appears that SA RSEuclid contains the fastest set of GF polynomial arithmetic routines.

5. Comparing numbers in any given column, it appears that SA GFFourier is the fastest implementation of the GF DFT.

6. The version using ASM GFFourier16 and ASM RSDS16 is about 5% slower than the version using SA GFFourier and SA RSDS. From this, it seems that the hand-written regular assembly routines are highly efficient implementations of the GF DFT and discrepancy calculation. The software-pipelined loop of SA RSDS is two cycles, as is the software-pipelined loop of ASM RSDS16. The software-pipelined inner loop of SA GFFourier is three cycles. The software-pipelined inner loop of ASM GFFourier16 is five cycles, but it processes two inputs at each iteration. From this comparison, one would assume ASM GFFourier16 is generally faster. The inconsistency may reside elsewhere in the routine. Note that in ASM GFFourier16, the software-pipelined loop is inside another, non-software-pipelined loop.

7. C RSDS is apparently more efficient than SA RSDS. This is probably because RSDiscrepancy() is such a simple function that the C compiler had no trouble optimizing it. Note that C RSDS is almost as fast as ASM RSDS16.

8. The fastest version implementing Euclid's algorithm, the combination of SA GFFourier and SA RSEuclid, is only 2.5% slower than the fastest version implementing the Berlekamp-Massey algorithm, the version using SA GFFourier and ASM RSDS16. This is somewhat surprising. Based on Wicker's information, it was expected that the implementation of Euclid's algorithm would be much slower. Euclid's algorithm was easy to understand and straight-forward to implement, and in this test it performed almost as well as the Berlekamp-Massey algorithm.

9. Unfortunately, ASM RSEuclid did not perform as well as SA RSEuclid. In fact, it seems it would generally be better to use the C version of RSEuclid than the hand-written assembly
version. The version using ASM GFFourier16 and ASM RSEuclid is 14% slower than the
version using ASM GFFourier16 and ASM RSDS16, and 16% slower than the version using
SA GFFourier and SA RSEuclid. The critical loops in ASM RSEuclid were not software-
pipelined, because the trip counts, using most practical RS codes, are often too small to use
a software-pipelined loop. However, the assembly-optimizer and compiler-optimizer always
generate a software-pipelined inner loop, and this is one reason why the versions using SA
RSEuclid are faster than the versions using ASM RSEuclid.

10. The most dramatic differences in cycle count are seen when comparing versions using the
C GFFourier() function and versions using SA GFFourier. The GFDFT can still be
optimized much further, for any particular set of input and output sequence lengths, by
implementing a kind of fast Fourier transform.

11. The cycle-count for ASM GFFourier16 and ASM RSDS16 is slightly slower than the cycle-
count for its 32-bit counterpart. The reason is probably the overhead involved in computing
for two input values in ASM GFFourier 16.

12. No program combination can sustain the throughput necessary for decoding a digital
television stream, as described by the US HDTV standard. That throughput is approximately
20 Mb/s [Spectrum, 37]. Interestingly, US HDTV does use a Reed Solomon code
[Spectrum, 43].

13. No program combination can sustain the throughput necessary for decoding a DVD-Video
stream. The throughput to the error correction decoder in a DVD-Video player is just over
13 Mb/s, which corresponds to approximately 11 Mb/s user data [DVD, §3-4].

14. Most program combinations provide similar throughputs. For example, analogous
combinations using Euclid vary only slightly in performance, and analogous combinations
using Belekamp-Massey vary only slightly in performance.* The reason is that the compiler,
assembly-optimizer, and human assembly programmer all use similar criteria and

61
Euclid, elementary-school polynomial multiplication, etc.), they output object code which vary only slightly in performance. From this, one may conclude that the algorithms used in the decoder were implemented (about) optimally. Note, however, this is not the same as saying the decoder is optimal; using more efficient algorithms would have resulted in a better-performing decoder.

The fact that combinations implementing Euclid and combinations implementing Berlekamp-Massey performed roughly equivalently evidences that the two algorithms are, from a CPU perspective, similar.

* The exception is C GFFourier, which consistently performed worse than the other GFFourier implementations. The reason is that the C compiler-optimizer is not yet able to make the kinds of optimizations made by the assembly-optimizer and the human programmer. This has probably already been remedied.

In conclusion, the optimizations made in this project were not sufficient to allow the use of the RS decoder in high-throughput multimedia applications. However, at 1600 MIPS, the C62x is definitely capable of performing high-throughput processing, and although several modifications were made, the final RS decoder is by no means optimal.

**Further Work**
The modified RS decoder can be improved significantly. As stated above, a kind of fast Fourier transform can be implemented for use with one particular set of RS code parameters. The trade-off is the versatility gained by using a generic Fourier transform function. Perhaps the compiler could conditionally compile the FFT function when the special RS code is used, and in other cases compile the generic GFFourier() function. Because the Fourier transform is critical, implementing an FFT would vastly improve the performance of the decoder. In some program combinations using an FFT, it may be possible to achieve the user data throughput necessary for DVD-Video.
Another area for improvement is the implementation of the GF polynomial arithmetic, in C and in assembly. Elementary-school multiplication and division were implemented. While simple to understand and implement, these algorithms are not efficient. Multiplication of two polynomials of N coefficients each requires about \( N^2 \) coefficient multiplications and additions. The polynomial division is similarly complex. Because the Euclid implementation is currently only slightly slower than the Berlekamp-Massey implementation, more efficient algorithms could make Euclid slightly faster than Berlekamp-Massey.

Finally, it could be possible to improve the method by which the roots of the error locator are found. Huber presents one alternative to the Chien search.
References

[Clark] Clark, George C.
Error-correction coding for digital communications
Plenum Press, 1981

[Dillon] Dillon, Thomas J., Jr.
The Use of Software Pipelining in Developing DSP Algorithms for the TMS320C6x
TMS320C62x Applications Engineering Group, Texas Instruments
November 1997

[DVD] Taylor, Jim
DVD FAQ
http://www.videodiscovery.com/vdyweb/dvd/dvdfaq.html
(Accessed March 1998)

[Huber] Huber, Klaus
"Solving Equations in Finite Fields and Some Results Concerning the Structure of GF( p^m )"

[Rowlands] Rowlands, Jon
Reed-Solomon Error Correcting Codes Tutorial
Integrated Systems Laboratory, DSP Technology Branch, Texas Instruments
8 September 1993

"The US HDTV Standard"
IEEE Spectrum, Vol. 32, No. 4, April 1995

[TI_CIS] Texas Instruments
TMS320C62xx CPU and Instruction Set Reference Guide
July 1997

[TI_OCC] Texas Instruments
TMS320C6x Optimizing C Compiler User’s Guide (Preliminary)
July 1997

[TI_WWW] Texas Instruments
(Accessed December 1997)

[Wicker] Wicker, Stephen B.
Error Control Systems for Digital Communication and Storage
Prentice Hall, 1995
The appendices contain the source code written for this project; describe modifications made to the source code for the original RS decoder; and include useful data files. The source code for the original C RS decoder is Texas Instruments internal data; it cannot be published with this paper. Throughout these appendices, modifications to that source code, as they pertain to the optimization of the RS decoder, are thoroughly described, and excerpts from the modified source code are presented.

Appendix A – C Implementation of Euclid’s Algorithm

This section lists the C functions which were written to implement Euclid’s algorithm in the RS decoder. The input is the syndrome and RS code parameters. The outputs are the error locator and the error evaluator.

GF PolyAdd, GF PolySubtract

```c
/*GF Polynomial Arithmetic Functions
 * Kamal Swamidoss
 * November 1997
 */

#ifdef(UseMyRSEuclid)
UseMyRSEuclid is a preprocessor value. It can be defined in the file modefile.h (see Appendix E – Modefile). If UseMyRSEuclid is defined, then the C code for Euclid’s algorithm is compiled.

#ifndef(!defined(UseASMGFPolyXOR) && !defined(UseSAGFPolyXOR))
At most one of UseASMGFPolyXOR and UseSAGFPolyXOR can be defined in modefile.h. The former indicates to the compiler that an assembly routine will perform the GF polynomial addition and subtraction operations. The latter indicates that a straight-assembly routine will perform the operations. The C functions GF PolyAdd() and GF PolySubtract() are compiled only if neither preprocessor value is defined.

void GF PolyAdd(RSCode *code,
const RSSymbol *firstPolynomial,
const RSSymbol *secondPolynomial,
RSSymbol *sum,
int firstPolynomialDegree,
int secondPolynomialDegree,
int *sumDegree)
{
int i;
RSSymbol *holdPolynomial;
int holdDegree;

if (firstPolynomialDegree < secondPolynomialDegree) {
    holdPolynomial = *(RSSymbol *) firstPolynomial;
    firstPolynomial = secondPolynomial;
    secondPolynomial = *(const RSSymbol *) holdPolynomial;
    holdDegree = firstPolynomialDegree;
    firstPolynomialDegree = secondPolynomialDegree;
    secondPolynomialDegree = holdDegree;
}

for (i=0; i<=secondPolynomialDegree; ++i)
    *sum++ = GF Add(code, *firstPolynomial++, *secondPolynomial++);

for (i=0; i<firstPolynomialDegree-secondPolynomialDegree; ++i)
```
32  *sum++ = *firstPolynomial++;  
33  
34  holdPolynomial = --sum;  
35  holdDegree = firstPolynomialDegree;  
36  
37  while ((holdDegree > 0) && (*holdPolynomial-- == 0))  
38    --holdDegree;  
39  
40  *sumDegree = holdDegree;  
41 }  
42  
43 INLINE  
44 void GFPolySubtract(RSCode *code,  
45 const RSSymbol *firstPolynomial,  
46 const RSSymbol *secondPolynomial,  
47 RSSymbol *difference,  
48 int firstPolynomialDegree,  
49 int secondPolynomialDegree,  
50 int *differenceDegree) {  
51  
52 ASMGFPolyXOR(firstPolynomial,  
53 secondPolynomial,  
54 difference,  
55 firstPolynomialDegree,  
56 secondPolynomialDegree,  
57 sum,  
58 sumDegree);  
59 }  
60 
61 INLINE  
62 void GFPolyAdd(RSCode *code,  
63 RSSymbol *firstPolynomial,  
64 RSSymbol *secondPolynomial,  
65 RSSymbol *sum,  
66 int firstPolynomialDegree,  
67 int secondPolynomialDegree,  
68 int *sumDegree) {  
69  
70 ASMGFPolyXOR(firstPolynomial,  
71 secondPolynomial,  
72 sum,  
73 sumDegree);  
74 }  
75 
76 INLINE  
77 void GFPolySubtract(RSCode *code,  
78 RSSymbol *firstPolynomial,  
79 RSSymbol *secondPolynomial,  
80 RSSymbol *difference,  
81 int firstPolynomialDegree,  
82 int secondPolynomialDegree,  
83 int *differenceDegree) {  
84  
85 ASMGFPolyXOR(firstPolynomial,  
86 secondPolynomial,  
87 firstPolynomialDegree,  
88 secondPolynomialDegree,  
89  
90 The following section of C code is compiled if ASMGFPolyXOR, a regular assembly routine, is to be used to perform the GF polynomial addition and subtraction. The assembly routine can be assembled and linked into the decoder. Note that the different definitions of GFPolyAdd() and GFPolySubtract() are mutually exclusive; that is, exactly one set of functions is defined in any build. RSEuclid() calls the functions GFPolyAdd(), GFPolySubtract(), GFPolyMultiply(), and GFPolyDivide().
The following section is compiled when the straight-assembly routine SAGFPolyXOR is to be used.

```c
#elif defined(UseSAGFPolyXOR)

INLINE void GFPolyADD(RSCode *code,
    RSSymbol *firstPolynomial,
    RSSymbol *secondPolynomial,
    RSSymbol *sum,
    int firstPolynomialDegree,
    int secondPolynomialDegree,
    int *sumDegree) {
    SAGFPolyXOR(firstPolynomial,
        secondPolynomial,
        firstPolynomialDegree,
        secondPolynomialDegree,
        sum,
        sumDegree);}

INLINE void GFPolySubtract(RSCode *code,
    RSSymbol *firstPolynomial,
    RSSymbol *secondPolynomial,
    RSSymbol *difference,
    int firstPolynomialDegree,
    int secondPolynomialDegree,
    int *differenceDegree) {
    SAGFPolyXOR(firstPolynomial,
        secondPolynomial,
        firstPolynomialDegree,
        secondPolynomialDegree,
        difference,
        differenceDegree);
}
#endif
```

GFPolyMultiply

The GF polynomial multiply operation is performed as elementary-school polynomial multiplication, except in this case, it is performed on GF elements. A more efficient polynomial multiplication algorithm would yield significantly better performance. The regular assembly routine is called ASMGFPolyMultiply, and the straight-assembly routine is called SAGFPolyMultiply.

```c
#if (!defined(UseASMGFPolyMultiply) && !defined(UseSAGFPolyMultiply))
void GFPolyMultiply(
    RSCode *code,
    const RSSymbol *firstPolynomial,
    const RSSymbol *secondPolynomial,
    RSSymbol *product,
    int firstPolynomialDegree,
    int secondPolynomialDegree,
    int *productDegree) {
    int i;
    RSSymbol *holdPolynomial;
    int holdDegree;
    const RSSymbol *ptr1,*ptr2;
    RSSymbol *ptr3;
    RSSymbol *productProgress;
    int counter1,counter2;
```
if (firstPolynomialDegree < secondPolynomialDegree) {
    holdPolynomial = (RSSymbol *) firstPolynomial;
    firstPolynomial = secondPolynomial;
    secondPolynomial = (const RSSymbol *) holdPolynomial;
    holdDegree = firstPolynomialDegree;
    firstPolynomialDegree = secondPolynomialDegree;
    secondPolynomialDegree = holdDegree;
}

*productDegree = firstPolynomialDegree + secondPolynomialDegree;
for (i=0;i<*productDegree;++i)
    product[i] = 0;

counter2 = secondPolynomialDegree+1;
ptr2 = secondPolynomial;
productProgress = product;
ptr3 = productProgress++;
while (counter2-- > 0) {
    counter1 = firstPolynomialDegree+1;
    ptr1 = firstPolynomial;
    while (counter1-- > 0) {
        *ptr3 = GFAdd(code,
            *ptr3,
            GFMultiply(code,
                *ptr1,
                *ptr2));
        ++ptr1;
        ++ptr3;
    }
    ++ptr2;
    ptr1 = firstPolynomial;
    ptr3 = productProgress++;
}
holdDegree = *productDegree;
holdPolynomial = &(product[holdDegree]);
while ((holdDegree > 0) && (*holdPolynomial-- == 0))
    --holdDegree;

*productDegree = holdDegree;
}
#endif defined(UseASMGFPolyMultiply)

INLINE void GFPMultiply(
    RSCode     *code,
    RSSymbol   *firstPolynomial,
    RSSymbol   *secondPolynomial,
    RSSymbol   *product,
    int        firstPolynomialDegree,
    int        secondPolynomialDegree,
    int        *productDegree) {
    ASMGFPolyMultiply(
        firstPolynomial,
        secondPolynomial,
        firstPolynomialDegree,
        secondPolynomialDegree,
        product,
        productDegree);
}
#endif defined(UseSAGFPolyMultiply)
INLINE
void GFPolyMultiply(
  RSCode  *code,
  RSSymbol *firstPolynomial,
  RSSymbol *secondPolynomial,
  RSSymbol *product,
  int firstPolynomialDegree,
  int secondPolynomialDegree,
  int *productDegree) {
  SAGFPolyMultiply(
    firstPolynomial,
    secondPolynomial,
    firstPolynomialDegree,
    secondPolynomialDegree,
    product,
    productDegree);
}

#if (!defined(UseASMGFPolyMultiply) && !defined(UseSAGFPolyMultiply))
GFPolyDivide
The GF polynomial divide operation is similar to elementary-school polynomial division. Again, a better algorithm would produce better results. The regular assembly routine is called ASMGFPolyDivide, and the straight-assembly routine is called SAGFPolyDivide. The MyPrint...() functions allow the developer to see the contents of arrays during run-time (for debugging purposes).
#endif /* #if (!defined(UseASMGFPolyDivide) && !defined(UseSAGFPolyDivide)) */

#if (!defined(UseASMGFPolyDivide) && !defined(UseSAGFPolyDivide))
void GFPolyDivide(
  RSCode  *code,
  const RSSymbol *numerator,
  const RSSymbol *denominator,
  RSSymbol *quotient,
  RSSymbol *remainder,
  int numeratorDegree,
  int denominatorDegree,
  int *quotientDegree,
  int *remainderDegree) {
  int counterl;
  const RSSymbol *ptrl;
  RSSymbol *ptr2;
  RSSymbol div.prod;
  int quotientIndex,remainderIndex;
  int i;
  counterl = numeratorDegree+1;
  ptr2 = remainder;
  ptrl = numerator;
  while (counterl-- > 0)
    *ptr2++ = *ptrl++;
  *remainderDegree = numeratorDegree;
  if (numeratorDegree < denominatorDegree) {
    *quotientDegree = 0;
    quotient[0] = 0;
    return;
  }
  *quotientDegree = numeratorDegree - denominatorDegree;
  quotientIndex = *quotientDegree;

while (*remainderDegree >= denominatorDegree) {
    div = GFDivide(code, remainder[*remainderDegree], denominator[denominatorDegree]);
    quotient[quotientIndex--] = div;
    remainderIndex = *remainderDegree;
    for (i=denominatorDegree; i>=0; --i) {
        prod = GFMultiply(code, div, denominator[i]);
        remainder[remainderIndex] = GFSubtract(code, remainder[remainderIndex], prod);
        --remainderIndex;
    }
    --*remainderDegree;
}
#endif /* if defined(EnableConsoleOutput) */
ptr2 = &quotient[quotientIndex];
while (quotientIndex-- >= 0)
    *ptr2-- = 0;
quotientIndex = *quotientDegree;
ptr2 = &quotient[quotientIndex];
while ((quotientIndex > 0) && (*ptr2-- == 0))
    --quotientIndex;
*quotientDegree = quotientIndex;
remainderIndex = numeratorDegree;
ptr2 = &quotient[remainderIndex];
while ((remainderIndex > 0) && (*ptr2-- == 0))
    --remainderIndex;
*remainderDegree = remainderIndex;
#elif defined(UseASMGFPolyDivide)
INLINE
void GFPolyDivide(
    RSCode   *code,
    RSSymbol  *numerator,
    RSSymbol  *denominator,
    RSSymbol  *quotient,
    RSSymbol  *remainder,
    int       numeratorDegree,
    int       denominatorDegree,
    int       *quotientDegree,
    int       *remainderDegree)
{
    ASMGFPolyDivide(
        numerator,
        denominator,
        quotient,
Euclid

This function is called to obtain the error locator and error evaluator polynomials. The error evaluator is given in log form because the rest of the decoder uses it in that form. As stated before, Clark's interpretation of Euclid's algorithm is implemented here.

```c
void RSEuclid(RSCode *code, const RSSymbol *syndrome, RSSymbol *errorLocator, int errorLocatorDegree, RSLogSymbol *logErrorEvaluator, int *errorEvaluatorDegree) {
    int i;
    RSSymbol *q;
    RSSymbol *r, *rp, *rpp;
    RSSymbol *t, *tp, *tpp;
    RSSymbol *im;
    RSSymbol *hold, *hold2;
    RSLogSymbol *logHold;
    RSSymbol temp;
    int qDegree;
    int rDegree,rpDegree,rppDegree;
    int tDegree, tpDegree, tppDegree;
    int imDegree;
    int holdDegree;
    int tCopy;

    tCopy = code->numberOfCorrectableErrors;

    /* address memory */
    q = code->euclid0;
    r = code->euclid1;
    rp = code->euclid2;
```
28   rpp = code->euclid3;
29   t   = code->euclid4;
30   tp  = code->euclid5;
31   tpp = code->euclid6;
32   im  = code->euclid7;

The code->euclid? pointers point to temporary storage arrays (see Appendix H – Modifications to RSCode). The next few lines initialize the state polynomials.

33   /* initialize polynomials */
34   for (i=0;i<2*tCopy;++i)
35     *rpp++ = 0;
36     *rpp++ = 1;
37     *rpp++ = 0;
38     rpp = code->euclid3;
39     rppDegree = 2*tCopy;
40   hold = (RSSymbol *)syndrome;
41   for (i=0;i<2*tCopy;++i)
42     *rp++ = *hold++;
43     *rp++ = 0;
44     *rp = 0;
45     rp--; 
46     rp--; 
47     rpDegree = 2*tCopy-1;
48   while ((*rp == 0) && (rp > code->euclid2)) {
49     --rp;
50     --rpDegree;
51   }
52   rp = code->euclid2;
53   for (i=0;i<2*tCopy+2;++i) {
54     *tpp++ = 0;
55     *tp++ = 0;
56     *t++  = 0;
57     *r++  = 0;
58     *q++  = 0;
59     *im++ = 0;
60   }
61   tpp = code->euclid6;
62   tp  = code->euclid5;
63   t   = code->euclid4;
64   r   = code->euclid1;
65   q   = code->euclid0;
66   im  = code->euclid7;
67   tp[0] = 1;
68   tpDegree = 0;
69   tppDegree = 0;
70   rDegree = -1;
71   qDegree = -1;
72   tDegree = -1;
73   imDegree = -1;
74   *errorLocatorDegree = -1;
This is the main loop of Euclid's algorithm. Several calls to MyPrint...() functions, used during debugging, have been commented out.

```c
    do {
        /* Get q and r */
        GFPolyDivide(code,
                    rpp,
                    rp,
                    q,
                    r,
                    rppDegree,
                    rpDegree,
                    &qDegree,
                    &rDegree);

        #if defined(EnableConsoleOutput)
        /*
        */
        puts("After Divide (logs):");
        MyPrintRSSymbolArrayLog(code," rpp: ",rpp,rppDegree+l);
        MyPrintRSSymbolArrayLog(code," rp: ",rp,rpDegree+l);
        MyPrintRSSymbolArrayLog(code," q: ",q,qDegree+l);
        MyPrintRSSymbolArrayLog(code," r: ",r,rDegree+l);
        #endif /* #if defined(EnableConsoleOutput) */

        /* Get im = q*tp */
        GFPolyMultiply(code,
                        q,
                        tp,
                        im,
                        /* im gets product */
                        qDegree,
                        tpDegree,
                        &imDegree);

        #if defined(EnableConsoleOutput)
        /*
        */
        puts("After Multiply (logs):");
        MyPrintRSSymbolArrayLog(code," q: ",q,qDegree+l);
        MyPrintRSSymbolArrayLog(code," tp: ",tp,tpDegree+l);
        MyPrintRSSymbolArrayLog(code," im: ",im,imDegree+l);
        #endif /* #if defined(EnableConsoleOutput) */

        /* Subtract im (= q*tp) from tpp */
        GFPolySubtract(code,
                        tpp,
                        im,
                        t,
                        tppDegree,
                        imDegree,
                        &tDegree
                        );
        #if defined(EnableConsoleOutput)
```
puts("After Subtract (logs):");
MyPrintRSSymbolArrayLog(code," tpp: ",tpp,tppDegree+l);
MyPrintRSSymbolArrayLog(code," im: ",im,imDegree+l);
MyPrintRSSymbolArrayLog(code," t: ",t,tDegree+l);
puts("After Subtract:");
MyPrintRSSymbolArray(" tpp: ",tpp,tppDegree+l);
MyPrintRSSymbolArray(" im: ",im,imDegree+l);
MyPrintRSSymbolArray(" t: ",t,tDegree+l);

The following lines update the state polynomials as the algorithm iterates. Note that only the pointers are
updated; the array elements are not moved.

hold = tpp;
holdDegree = tppDegree;
tpp = tp;
tppDegree = tpDegree;
t = tp;
tDegree = tDegree;
hold = t;
tDegree = holdDegree;

hold = rpp;
holdDegree = rppDegree;
rpp = rp;
rppDegree = rpDegree;
r = r;
rDegree = rDegree;
r = hold;
rDegree = holdDegree;

while (rDegree >= tCopy);

The loop is executed until the stopping condition is satisfied. At that point, the error locator is scaled (if
necessary), and the error evaluator is converted to log form.

/* We're copying tp and not t
 * because of the shift at the end of the iteration.
 * Also note that tp[tpDegree] is never zero, because of the
 * construction of the error locator polynomial.
 */
hold = errorLocator;
hold2 = tp;

if (tp[0] != 1) {
temp = tp[0];
for (i=0;i<=tpDegree;++i)
 *hold++ = GFDivide(code,*hold2++,temp);
logHold = logErrorEvaluator;
hold2 = rp;
for (i=0;i<=rpDegree;++i)
 *logHold++ = GFLog(code,GFDivide(code,*hold2++,temp));
} else {
for (i=0;i<=tpDegree;++i)
 *hold++ = *hold2++;
logHold = logErrorEvaluator;
hold2 = rp;

for (i=0; i<=rpDegree; ++i)
    *logHold++ = GFLog(code,*hold2++);

#errorLocatorDegree = tpDegree;
#errorEvaluatorDegree = rpDegree;

#if defined(EnableConsoleOutput)
    /*
     * MyPrintRSSymbolArrayLog(code, 
     *    "logSyndrome: ", 
     *    (RSSymbol *) syndrome, 
     *    2*code->numberOfCorrectableErrors);
     * MyPrintRSSymbolArrayLog(code, 
     *    "logErrorLocator: ", 
     *    errorLocator, 
     *    "errorLocatorDegree+1);
     * MyPrintRSLogSymbolArrayLog("logErrorEvaluator: ", 
     *    logErrorEvaluator, 
     *    "errorEvaluatorDegree+1);
     */
#endif /* #if defined(EnableConsoleOutput) */

return;

#endif /* #if defined(UseMyRSEuclid) */
Appendix B – Regular Assembly Files

This section lists six of the seven regular assembly files. The seventh file, containing the 32-bit implementation of RSDiscrepancy(), is virtually identical to the 16-bit implementation listed here (the same algorithm is implemented in the same way; the only difference is that symbols are accessed as 32-bit values). In general, these routines are direct implementations of the corresponding C functions.

ASMGFFourier32 – 32-bit GFFourier()

This routine can be used when symbols and logs are represented in 32 bits. The header describes how this function can be called from C. As per the C calling convention, upon entering the routine from C, the first argument to the C function is found in register A4, the second in B4, the third in A6, the fourth in B6, and the fifth in A8.

```c
void ASMGFFourier32() {
    RCode *code,
    GFFourierParameters *parameters,
    int numberOfInputSymbols,
    RSSymbol input[],
    RSSymbol output[];
}
```

The following assembler directives are useful for printing the assembly file to paper.

```assembly
.MYTABSIZE .set 8
.MYPAGewidth .set 78
.MYPAGELENGTH .set 75
.tab .MYTABSIZE
.width .MYPAGewidth
.length .MYPAGELENGTH
.FP .set B5
.DP .set B14
.SP .set B15
.align 32
.global _logTable, _antilogTable
.global _ASMGFFourier32
.text
_ASMGFFourier32:
ASMGFFourierEnter:
```

The following lines align the object code (in program memory) on a 32-bit boundary, and define _logTable, _antilogTable, and _ASMGFFourier32 as global variables. _logTable and _antilogTable correspond to the C pointers logTable and antilogTable. _ASMGFFourier32 is the name of the assembly routine.
The following assembler directives assign registers to assembly variables. While defining variables and choosing assignments, it was necessary to consider several factors, including the side rules of various instructions (see Background) and the initial locations of the function's arguments. Some variables were defined while software-pipelining the inner loop; the remaining variables were defined as the rest of the assembly code was written around the inner loop.

The following lines load values from the RSCode and GFFourierParameters structures.

The following lines move the addresses of the \texttt{log} and \texttt{antilog} arrays into the appropriate registers.

The following lines move the function's arguments to the appropriate registers.

The counter test is used to determine if the number of inputs is large enough to use the software-pipelined loop. Note that this routine returns to the calling function if the number is too small. The 16-bit version (listed later in this section) contains a redundant, non-software-pipelined loop which is used when the
number of inputs is too small to use the software-pipelined loop. In the RS codes used in this project, the number of input symbols was always large enough to use the software-pipelined loop.

```
79  ASMGFFourierTest:               MV    numberOfOutputSymbols, temp3
80  SUB    temp3, 2, temp3
81 ; for software-pipelining
82  CMPGT temp3, 0, temp4
83  [{temp4}  B    ASMGFFourierExit
84  NOP    5
85
86  ASMGFFourierInitOutput:
87  LDW    *B4[1], temp3
88 ; temp3 = constantValue
89  MV    output1, temp2
90  MV    numberOfOutputSymbols, temp4
91  NOP    3
92
93  ASMGFFourierInitOutputLoop:
94  [{temp4}  B    ASMGFFourierInitOutputLoop
95  [{temp4}  STW    temp3, *temp2++
96  [{temp4}  ADDK    -1, temp4
97  NOP    3
98
99  ASMGFFourierLoop1:
100  [{numberOfInputSymbols}  LDW    *input++, temp3
101  NOP    4
102  [{temp3}  B    ASMGFFourierLoop1Continue
103  NOP    5
104
105  ASMGFFourierLoop2Init:
106  MV    numberOfOutputSymbols, temp3
107  SUB    temp3, 2, temp3
108  MV    A8, output1
109  MV    A8, output2
110
111  ASMGFFourierLoop2Prolog:
112  ADD    index, indexStep, index
113  ||    LDW    *-antilogTable[index], temp1
114  ||    LDW    *output1++, temp2
115  ||    CMPLT    index, N, temp3
116  ||    ADDK    -1, temp3
117  ||    [{temp3}  B    ASMGFFourierLoop2
118  ||    [{temp4}  SUB    index, N, index
119  ||    ||    ADD    index, indexStep, index
120  ||    ||    LDW    *-antilogTable[index], temp1
121  ||    ||    LDW    *output1++, temp2
122  ||    ||    CMPLT    index, N, temp4
123  ||    ||    ADDK    -1, temp3
124
125  ASMGFFourierLoop2:
126  XOR    temp2, temp1, temp2
127  ||    [{temp3}  B    ASMGFFourierLoop2
128  ||    [{temp4}  SUB    index, N, index
129```
ASMFFourier - 16-bit GFFourier

This routine can be used when symbols and logs are represented in 16 bits. The following file contains the register assignments for this routine. Again, several factors were considered when making these assignments. In fact, register assignment was one of the more difficult aspects of writing assembly in this project. Every C62x register was used in this routine. The value in every register must be saved to the software stack before the register is used, and the values must be restored before the routine returns. As described previously, this implementation of the software-pipelined inner loop required many variables.

Register Allocation

```
* Register Allocation
* for ASMFFourier() in gffrl6.asm
* Kamal Swamidoss
* December 1997
.output1 .set A14
.output2 .set B14
.twoSymbols .set A3
```

79
Instructions

******************************************************************************

* ASMGFFourier() in TMS320C6201 Scheduled Assembly
* C-callable
* 16-bit RSSymbol
* 16-bit RSLogSymbol
*
* Written by: Kamal Swamidoss
* 16 October 1997
*
* Based on: C Code from Jon Rowlands
*
* void
* GFFourier(
* RSCode *code,
* GFFourierParameters *parameters,
* int numberOfInputSymbols,
* RSSymbol input[],
* RSSymbol output[]
* );
*
******************************************************************************
.width MYPAGEWIDTH
.length MYPAGELENGTH
.align 32
.global _ASMGFFourier
.global _antilogTable,_logTable
.include gffrl6.inc ; include register assignments
STK_SIZE .set 14
.text

_ASMGFFourier:
   ADDK -STK_SIZE*4,SP
   STW A10-A15,B10-B15
   STW A10.+SP[1]
   STW A11.+SP[2]
   STW A12.+SP[3]
   STW A13.+SP[4]
   STW A14.+SP[5]
   STW A15.+SP[6]
   STW B10.+SP[7]
   STW B11.+SP[8]
   STW B12.+SP[9]
   STW B13.+SP[10]
   STW B15.+SP[12]
   STW B1.+SP[13]

ASMGFFourierInit:
   LDW *+parameters[0],counter
   LDH *+parameters[5],indexStepTwice
   LDW *+code[1],N1
   MVK _antilogTable,antilogTable1
   MVKH _antilogTable,antilogTable1
   MVK _antilogTable,antilogTable2
   MVKH _antilogTable,antilogTable2
   MVK _logTable,logTable
   MVKH _logTable,logTable
   MV output,output1
   MV output,output2
   SHL indexStepTwice,1,indexStepTwice
   EXTU counter,31,31,cond2
   MV cond2,remainder
   SHRU counter,1,counter
   MVK 0xffff,lowMask
   MVKH 0x0000,lowMask
   MV N1,N2
   LDH *+parameters[2],constantValue
   LDH *+parameters[3],startingIndex
   LDH *+parameters[4],startingIndexStep
   LDH *+parameters[5],indexStep
   LDH *+parameters[6],indexStepStep
ASMGFFourierInitOutputLoop:
  [ counter] B
  [ counter] STH
  [ counter] STH
  [ counter] ADDK
  NOP
  [ remainder] STH
ASMGFFourierInitOutputLoopDone:
ASMGFFourierLoop1Init:
  MV
ASMGFFourierLoop1:
  LDH
  NOP
  [!cond1] B
  [ cond1] LDH
  NOP
  ADD
  CMPLT
  [!cond1] SUB
ASMGFFourierLoop2Init:
  LDW
  MV
  CMPGT
  [!cond1] B
  [ cond1] EXTU
  [ cond1] MV
  [ cond1] SHRU
  [ cond1] ADDK
; to count LDW in prolog
  NOP
ASMGFFourierLoop2Prolog:
  LDW
  ADD
  ||
  ADD
  ||
  LDH
  ||
  LDH
  ||
  CMPLT
  ||
  CMPLT
  ||
  ADDK
  [!cond1] SUB
  ||[cond2] SUB
  ||[counter] B
ASMGFFourierLoop2:
The non-software-pipelined inner loop begins here. Note that while the software-pipelined loop requires five cycles, the non-software-pipelined loop requires nine.
ASMFFourierLoop: Continue:

```
[!cond1] SUB startingIndex, startingIndex
ADD startingIndex, startingIndex
CMPLT startingIndex
[!cond1] SUB startingIndex
ADD startingIndex
CMPLT startingIndex
[!cond2] SUB startingIndex
```

ASMFFourierExit:

```
LDW A10, B10
LDW B11
LDW B12
LDW B13
LDW B14
LDW B15
LDW B3
LDW SP
ADDK STR_SIZE*4, SP
```

**ASMRSDiscrepancy – 16-bit RDiscrepancy**

**Register Allocation**

```
antilogTable .set A3
temp1 .set A7
temp2 .set B7
temp3 .set A9
temp4 .set A2
discrepancy .set B9
counter .set B2
code .set A4
i .set B4
errorLocatorDegree .set A6
logSyndrome .set B6
logErrorLocator .set A8
```

**Instructions**

```

...............................................................
```
ASMRSDiscrepancy() in TMS320C6201 Scheduled Assembly

* C Callable
* 16-bit RSSymbol
* 16-bit RSLogSymbol

* Ported/
* Written by: Kamal Swamidoss
* 21 October 1997

* from Code by: Kamal Swamidoss
* September 1997
* (version for 32-bit data)

* Based on: C Code from Jon Rowlands

* RSSymbol
* ASMRSDiscrepancy()

* RSCode *code,
19  int i,
20  * int errorLocatorDegree,
21  * RSLogSymbol *logSyndrome,
22  * RSLogSymbol *logErrorLocator
23
24  *
25
26  ******************************************************************************
27
28 MYTABSIZE .set 8
29 MYPAGEWIDTH .set 78
30 MYPAGELENGTH .set 75
31
32 FP .set B5
33 DP .set B14
34 SP .set B15
35
36 .tab MYTABSIZE
37 .width MYPAGEWIDTH
38 .length MYPAGELENGTH
39 .align 32
40
41 .global _ASMRSDiscrepancy
42 .global _antilogTable
43
44 .include rsds.inc
45
46 .text
47
48 _ASMRSDiscrepancy:
49 ASMRSDiscrepancyTest:
50     ADD       errorLocatorDegree,1,counter
51     CMPGT     counter,6,Al
52     [!Al] B   ASMRSDiscrepancyNotSP
53     [ A1] SUB  counter,6,counter
54     MVK       _antilogTable,antilogTable
55     MVK       _antilogTable,antilogTable
56     MVK       0,discrepancy
57     ADDAH     logSyndrome,i,logSyndrome
58
59  * Software-Pipelined Version
60  ASMRSDiscrepancySPLoopProlog:
61     LDH      *logSyndrome--,templ1
62     ||
63     LDH      *logErrorLocator++,temp2
64     NOP
65
66     LDH      *logSyndrome--,templ1

85
*logErrorLocator++, temp2

*logSyndrome--, temp1

*logErrorLocator++, temp2

ADD temp1, temp2, temp3

[ counter] ADDK -1, counter

*logSyndrome--, temp1

*logErrorLocator++, temp2

ADD temp1, temp2, temp3

B ASMRSDiscrepancySLoop

LDH *+antilogTable[temp3], temp4

[ counter] ADDK -1, counter

*logSyndrome--, temp1

*logErrorLocator++, temp2

B ASMRSDiscrepancySLoop

LDH *+antilogTable[temp3], temp4

ASMRSDiscrepancySLoop:

ADD temp1, temp2, temp3

[ counter] B ASMRSDiscrepancySLoop

LDH *+antilogTable[temp3], temp4

XOR discrepancy, temp4, discrepancy

[ counter] ADDK -1, counter

*logSyndrome--, temp1

*logErrorLocator++, temp2

ASMRSDiscrepancySLoopEpilog:

ADD temp1, temp2, temp3

[ counter] B ASMRSDiscrepancySLoop

LDH *+antilogTable[temp3], temp4

XOR discrepancy, temp4, discrepancy

ADD temp1, temp2, temp3

*+antilogTable[temp3], temp4

XOR discrepancy, temp4, discrepancy

ADD temp1, temp2, temp3

*+antilogTable[temp3], temp4

XOR discrepancy, temp4, discrepancy

ADD temp1, temp2, temp3

*+antilogTable[temp3], temp4

XOR discrepancy, temp4, discrepancy

ADD temp1, temp2, temp3

*+antilogTable[temp3], temp4

XOR discrepancy, temp4, discrepancy

ADD temp1, temp2, temp3

*+antilogTable[temp3], temp4

XOR discrepancy, temp4, discrepancy

B B3

XOR discrepancy, temp4, discrepancy

NOP
ASMRSDiscrepancyNotSP:
138  LDH  *logErrorLocator++,templ
139  LDH  *logSyndrome--,temp2
140  NOP  4
141
142  ASMRSDiscrepancyNotSPLoop:
143  ADD  templ,temp2,temp3
144  LDH  ^=antilogTable[temp3].temp4
145  [ counter] ADDK  -1,counter
146  [!counter] B  B3
147  [ counter] B  ASMRSDiscrepancyNotSPLoop
148  [ counter] LDH  *logErrorLocator++,templ
149  [!counter] LDH  *logSyndrome--,temp2
150  NOP
151  XOR  discrepancy,temp4,discrepancy
152  [!counter] MV  discrepancy,A4
153  NOP

ASMGFPolyXOR

Register Allocation

1   a  .set A4
2   b  .set B4
3   aD  .set A6
4   bD  .set B6
5   x  .set A8
6   XD  .set B8
7
8   counter .set B0
9   cond1 .set A1
10  k  .set A2
11
12  temp1 .set A5
13  temp2 .set B5
14  i  .set A7
15  j  .set B7

Instructions

In C, the first two arguments are pointers to the input polynomials. The next two arguments indicate the degrees of the input polynomials. The last two arguments are the pointer to the output polynomial and a pointer to its degree.

1  ******************************************************
2  *
3  * ASMGFPolyXOR() in TMS320C6201 Scheduled Assembly
4  * C-callable

87
**Written by:** Kamal Swamidoss  
**November 1997**

```assembly
; ASMGFPolyXOR

void ASMGFPolyXOR( short *a, short *b, int aD, int bD, short *x, int *xD);

畢竟

MYTABSIZE .set 8
MYPAGEWIDTH .set 78
MYPAGELENGTH .set 75

FP .set B5
DP .set B14
SP .set B15

.tab MYTABSIZE
.width MYPAGEWIDTH
.length MYPAGELENGTH
.align 32

.def _ASMGFPolyXOR
.include gfplxr16.inc

STK_SIZE .set 0

.text

副市长

 ASMGFPolyXOR:
       CMPLT aD, bD, cond1
     [ cond1] MV a, templ
     [ cond1] MV bD, temp2
     [ cond1] MV b, a
     [ cond1] MV aD, bD
     [ cond1] MV temp1, b
     [ cond1] MV temp2, aD
     ADD bD, 1, counter

 ASMGFPolyXORLoop1:
     LDM *a++, i
     LDM *b++, j
     [ counter] ADDK -1, counter
     [ counter] B ASMGFPolyXORLoop1
     NOP 2
     XOM i, j, k
     STH k, *x++
     NOP
     SUB aD, bD, counter
     [counter] B ASMGFPolyXORContinuel

 ASMGFPolyXORLoop2:
```

---

88
ASMGF Poly XOR

ASMGF Poly XOR Continuel:

ASMGFPolyXORLoop3:

ASMGF Poly Multiply

Register Allocation
**Instructions**

In C, the first two arguments are pointers to the input polynomials. The next two arguments indicate the degrees of the input polynomials. The last two arguments are the pointer to the product polynomial and a pointer to its degree.

```c
void ASMGFPolyMultiply( short *a, short *b, int aD, int bD, short *p, int *pD);
```

---

```
MYTABSIZE .set 8
MYPAGEWIDTH .set 78
MYPAGELENGTH .set 75
FP .set B5
DP .set B14
SP .set B15
.tab MYTABSIZE
.width MYPAGEWIDTH
.length MYPAGELENGTH
.align 32
.def _ASMGFPolyMultiply
.ref _antilogTable,_logTable
.include gfplml16.inc
STK_SIZE .set 3
.text

_SUBAW SP,STK_SIZE,SP
STW A10,=*SP[1]
STW B10,=*SP[2]
STW A11,=*SP[3]
MVK _antilogTable,_antilogTable
MVKH _antilogTable,_antilogTable
MVK _logTable,logTable
MVKH _logTable,logTable
CMPLT aD,bD,cond1
MV a,templ1
||| cond1 MV bD,temp2
|| cond1 MV b,a
||| cond1 MV aD,bD
|| cond1 MV templ1,b
||| cond1 MV temp2,aD
```
60       ADD      aD,bD,temp3
61       STW      temp3,*pD
62
63       ADD      temp3,1,counter
64       ZERO     temp1
65       MV        p,temp2
66
67       STH       tmpl,*temp2--
68       [ counter] ADDK     -1,counter
69
70       ASMGFPolyMultiplyInitLoop:
71       [ counter] B      ASMGFPolyMultiplyInitLoop
72       [ counter] STH    temp1,*temp2--
73       [ counter] ADDK   -1,counter
74       NOP        3
75
76       ADD      bD,1,counter2
77       MV        b,ptr2
78       MV        p,ptr3
79       ADDAH     p,1,productProgress
80
81       ASMGFPolyMultiplyLoop1:
82       ADD      aD,1,counter1
83       MV        a,ptr1
84
85       ASMGFPolyMultiplyLoop1A:
86       LDH      *ptr1,temp1
87       ||       LDH      *ptr2,temp2
88
89       NOP        4
90
91       LDH      *+logTable[temp2],temp2
92       MV        temp1,temp2
93       LDH      *+logTable[temp2],temp1
94
95       NOP        4
96
97       ADD      temp1,temp2,temp4
98       LDH      *ptr3,temp3
99       ||       LDH      *+antilogTable[temp4],temp4
100
101       NOP        4
102
103       XOR      temp3,temp4,temp3
104       STH       temp3,*ptr3
105
106       ADDAH     ptr1,1,ptr1
107       ADDAH     ptr3,1,ptr3
108
109       [ counter1] ADDK   -1,counter1
110       [ counter1] B      ASMGFPolyMultiplyLoop1A
111       NOP        5
112
113       ADDAH     ptr2,1,ptr2
114       MV        a,ptr1
115       MV        productProgress,ptr3
116       ADDAH     productProgress,1,productProgress
117
118       [ counter2] ADDK   -1,counter2
119       [ counter2] B      ASMGFPolyMultiplyLoop1
120
121       LDW      *pD,temp2
122
123       NOP        4
124
ADDAH

ASMGFPolyMultiplyLoop2:
LDH     *ptr3--, temp3
CMPGT   temp2.0, condl
NOP      3
CMPGEQ  temp3.0, condl
AND     condl, condl, condl

[ condl] B     ASMGFPolyMultiplyLoop2
[ condl] ADDK  -1, temp2
NOP      4
STW     temp2, *pD
LDW     *+SP[1], A10
B       B3
LDW     *+SP[2], B10
LDW     *+SP[3], A11
ADDAW   SP, STK_SIZE, SP
NOP      3

**Register Allocation**

<table>
<thead>
<tr>
<th>n</th>
<th>.set</th>
<th>A4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>.set</td>
<td>B4</td>
</tr>
<tr>
<td>q</td>
<td>.set</td>
<td>A6</td>
</tr>
<tr>
<td>r</td>
<td>.set</td>
<td>B6</td>
</tr>
<tr>
<td>nD</td>
<td>.set</td>
<td>A8</td>
</tr>
<tr>
<td>dD</td>
<td>.set</td>
<td>B8</td>
</tr>
<tr>
<td>qD</td>
<td>.set</td>
<td>A10</td>
</tr>
<tr>
<td>rD</td>
<td>.set</td>
<td>B10</td>
</tr>
<tr>
<td>code</td>
<td>.set</td>
<td>A12</td>
</tr>
<tr>
<td>logTable</td>
<td>.set</td>
<td>A0</td>
</tr>
<tr>
<td>antilogTable</td>
<td>.set</td>
<td>A1</td>
</tr>
<tr>
<td>dCurrent</td>
<td>.set</td>
<td>A2</td>
</tr>
<tr>
<td>qCurrent</td>
<td>.set</td>
<td>B0</td>
</tr>
<tr>
<td>rCurrent</td>
<td>.set</td>
<td>A3</td>
</tr>
<tr>
<td>p1</td>
<td>.set</td>
<td>B1</td>
</tr>
<tr>
<td>p2</td>
<td>.set</td>
<td>B5</td>
</tr>
<tr>
<td>temp1</td>
<td>.set</td>
<td>B7</td>
</tr>
<tr>
<td>temp2</td>
<td>.set</td>
<td>A5</td>
</tr>
<tr>
<td>cond1</td>
<td>.set</td>
<td>B2</td>
</tr>
<tr>
<td>tempSide</td>
<td>.set</td>
<td>B9</td>
</tr>
</tbody>
</table>

**Instructions**

```
 125 ADDAH     p, temp2, ptr3
 126 ASMGFPolyMultiplyLoop2:
 127     LDH     *ptr3--, temp3
 129     CMPGT   temp2.0, condl
 131     NOP      3
 133     CMPGEQ  temp3.0, condl
 134     AND     condl, condl, condl
 136     [ condl] B     ASMGFPolyMultiplyLoop2
 137     [ condl] ADDK  -1, temp2
 138     NOP      4
 139     STW     temp2, *pD
 141     LDW     *+SP[1], A10
 142     B       B3
 143     LDW     *+SP[2], B10
 144     LDW     *+SP[3], A11
 145     ADDAW   SP, STK_SIZE, SP
 146     NOP      3

**Register Allocation**

<table>
<thead>
<tr>
<th>n</th>
<th>.set</th>
<th>A4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>.set</td>
<td>B4</td>
</tr>
<tr>
<td>q</td>
<td>.set</td>
<td>A6</td>
</tr>
<tr>
<td>r</td>
<td>.set</td>
<td>B6</td>
</tr>
<tr>
<td>nD</td>
<td>.set</td>
<td>A8</td>
</tr>
<tr>
<td>dD</td>
<td>.set</td>
<td>B8</td>
</tr>
<tr>
<td>qD</td>
<td>.set</td>
<td>A10</td>
</tr>
<tr>
<td>rD</td>
<td>.set</td>
<td>B10</td>
</tr>
<tr>
<td>code</td>
<td>.set</td>
<td>A12</td>
</tr>
<tr>
<td>logTable</td>
<td>.set</td>
<td>A0</td>
</tr>
<tr>
<td>antilogTable</td>
<td>.set</td>
<td>A1</td>
</tr>
<tr>
<td>dCurrent</td>
<td>.set</td>
<td>A2</td>
</tr>
<tr>
<td>qCurrent</td>
<td>.set</td>
<td>B0</td>
</tr>
<tr>
<td>rCurrent</td>
<td>.set</td>
<td>A3</td>
</tr>
<tr>
<td>p1</td>
<td>.set</td>
<td>B1</td>
</tr>
<tr>
<td>p2</td>
<td>.set</td>
<td>B5</td>
</tr>
<tr>
<td>temp1</td>
<td>.set</td>
<td>B7</td>
</tr>
<tr>
<td>temp2</td>
<td>.set</td>
<td>A5</td>
</tr>
<tr>
<td>cond1</td>
<td>.set</td>
<td>B2</td>
</tr>
<tr>
<td>tempSide</td>
<td>.set</td>
<td>B9</td>
</tr>
</tbody>
</table>
```
* Written by: Kamal Swamidoss
* November 1997

void

ASMGFPolyDivide(RSSymbol *numerator,
                 RSSymbol *denominator,
                 RSSymbol *quotient,
                 RSSymbol *remainder,
                 int numeratorDegree,
                 int denominatorDegree,
                 int *quotientDegree,
                 int *remainderDegree,
                 RSCode *code) {

    MYTABSIZE .set 8
    MYPAGEWIDTH .set 78
    MYPAGELENGTH .set 75

    FP .set B5
    DP .set B14
    SP .set B15

    .tab MYTABSIZE
    .width MYPAGEWIDTH
    .length MYPAGELENGTH
    .align 32

    .def _ASMGFPolyDivide
    .ref _logTable,_antilogTable

    .include gfpldv16.inc

    STK_SIZE .set 0

    .text

    _ASMGFPolyDivide:
    ASMGFPolyDivideInit:
        MVK _antilogTable,antilogTable
        MVKH _antilogTable,antilogTable
        MVK _logTable,logTable
        MVKH _logTable,logTable

        ADD nD,l,cond1
        MV n,p1
        MV r,p2

    ASMGFPolyDivideLoop1:
    *** BGN OF ASMGFPolyDivideLoop1
    *** This loop copies n to r.

        LDH *p1++,templ1
        [ cond1] ADDK -1,cond1
        [ cond1] B ASMGFPolyDivideLoop1
        NOP 2

        STH templ1,*p2++
        NOP 2

    *** END OF ASMGFPolyDivideLoop1

    STW nD,*rD
ASMGFPolyDivideLoop2:

*** BGN OF ASMGFPolyDivideLoop2

*** This is the main loop.

CMPLT nD,dD,condl
B3

[ condl] MVK 0.temp1
[ condl] MVXH 0.temp1
[ condl] STW templ.*qD
[ condl] STH templ.*q

NOP

SUB nD,dD,temp1
STW templ.*qD
MV q.tempSide
ADDAH tempSide,temp1.qCurrent

LDW *RD.temp1

NOP 4

ASMGFPolyDivideLoop2Continue

4

4

** t* This loop makes the new remainder.

ASMGFPolyDivideLoop2A:

*** BGN OF ASMGFPolyDivideLoop2A

*** This loop makes the new remainder.

CMPLT templ,dD,condl

[ condl] B ASMGFPolyDivideLoop2Continue

[!condl] ADDAH r.temp1.pl
[!condl] ADDAH d.dd,p2
[!condl] LDH *pl.rCurrent
[!condl] LDH *p2.dCurrent

NOP 4

LDW **code[1].temp2 ; get code->N

LDH **logTable[dCurrent].dCurrent
LDH **logTable[rCurrent].rCurrent

NOP 3

SUB temp2.dCurrent,dCurrent

ADD rCurrent.dCurrent,rCurrent
LDH **antilogTable[rCurrent].temp1

NOP 4

MV temp1.rCurrent

LDH **logTable[rCurrent].temp1

: obtain log(div) in the correct interval

STH rCurrent.*qCurrent--
ADD dD,1,condl

ASMGFPolyDivideLoop2A:

*** BGN OF ASMGFPolyDivideLoop2A

*** This loop makes the new remainder.

LDM *p2--,temp2

NOP 4

LDM **logTable[temp2],temp2

NOP 4

ADD templ,temp2,temp2

LDM **antilogTable[temp2],temp2
**ASMGFPolyDivideLoop3:**

*** This loop reduces the degree of q.

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>138</td>
<td>LDH &quot;p1,rCurrent&quot;</td>
<td></td>
</tr>
<tr>
<td>139</td>
<td>[ cond1] ADDK -1,cond1</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>[ cond1] B</td>
<td>ASMGFPolyDivideLoop2A</td>
</tr>
<tr>
<td>141</td>
<td>NOP 2</td>
<td></td>
</tr>
<tr>
<td>142</td>
<td>XOR rCurrent,temp2,temp2</td>
<td></td>
</tr>
<tr>
<td>143</td>
<td>STH temp2,&quot;p1&quot;-</td>
<td></td>
</tr>
<tr>
<td>144</td>
<td>NOP</td>
<td></td>
</tr>
</tbody>
</table>

*** END OF ASMGFPolyDivideLoop2A

**ASMGFPolyDivideLoop2Continue:**

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>148</td>
<td>LDW &quot;rD,templ&quot;</td>
<td></td>
</tr>
<tr>
<td>149</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>B</td>
<td>ASMGFPolyDivideLoop2</td>
</tr>
<tr>
<td>151</td>
<td>NOP 2</td>
<td></td>
</tr>
<tr>
<td>152</td>
<td>ADDK -1,templ</td>
<td></td>
</tr>
<tr>
<td>153</td>
<td>CMPLT temp1,0,cond1</td>
<td></td>
</tr>
<tr>
<td>154</td>
<td>STW temp1,&quot;rD&quot;</td>
<td></td>
</tr>
</tbody>
</table>

*** END OF ASMGFPolyDivideLoop2

**ASMGFPolyDivideLoop3:**

*** This loop reduces the remaining coefficients of q.

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>155</td>
<td>[!cond1] CMPLT qCurrent.q,cond1</td>
<td></td>
</tr>
<tr>
<td>156</td>
<td>[!cond1] B</td>
<td>ASMGFPolyDivideLoop3</td>
</tr>
<tr>
<td>157</td>
<td>[!cond1] STH temp1,&quot;qCurrent--&quot;</td>
<td></td>
</tr>
<tr>
<td>158</td>
<td>NOP 4</td>
<td></td>
</tr>
</tbody>
</table>

*** END OF ASMGFPolyDivideLoop3

**ASMGFPolyDivideLoop4:**

*** This loop reduces the degree of q.

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>159</td>
<td>LDH &quot;qCurrent--,temp2&quot;</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>NOP 3</td>
<td></td>
</tr>
<tr>
<td>161</td>
<td>MV q,tempSide</td>
<td></td>
</tr>
<tr>
<td>162</td>
<td>ADDAH tempSide,templ,qCurrent</td>
<td></td>
</tr>
</tbody>
</table>

*** END OF ASMGFPolyDivideLoop4

**ASMGFPolyDivideLoop4Continue:**

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>163</td>
<td></td>
<td></td>
</tr>
<tr>
<td>164</td>
<td></td>
<td></td>
</tr>
<tr>
<td>165</td>
<td></td>
<td></td>
</tr>
<tr>
<td>166</td>
<td></td>
<td></td>
</tr>
<tr>
<td>167</td>
<td></td>
<td></td>
</tr>
<tr>
<td>168</td>
<td></td>
<td></td>
</tr>
<tr>
<td>169</td>
<td>CMPLT qCurrent.q,cond1</td>
<td></td>
</tr>
<tr>
<td>170</td>
<td>[!cond1] B</td>
<td>ASMGFPolyDivideLoop4</td>
</tr>
<tr>
<td>171</td>
<td>[!cond1] STH temp2,0,cond1</td>
<td></td>
</tr>
<tr>
<td>172</td>
<td>B</td>
<td>ASMGFPolyDivideLoop4</td>
</tr>
</tbody>
</table>

*** END OF ASMGFPolyDivideLoop4

**ASMGFPolyDivideLoop4Continue:**

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>173</td>
<td></td>
<td></td>
</tr>
<tr>
<td>174</td>
<td></td>
<td></td>
</tr>
<tr>
<td>175</td>
<td></td>
<td></td>
</tr>
<tr>
<td>176</td>
<td></td>
<td></td>
</tr>
<tr>
<td>177</td>
<td></td>
<td></td>
</tr>
<tr>
<td>178</td>
<td></td>
<td></td>
</tr>
<tr>
<td>179</td>
<td></td>
<td></td>
</tr>
<tr>
<td>180</td>
<td></td>
<td></td>
</tr>
<tr>
<td>181</td>
<td></td>
<td></td>
</tr>
<tr>
<td>182</td>
<td></td>
<td></td>
</tr>
<tr>
<td>183</td>
<td></td>
<td></td>
</tr>
<tr>
<td>184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>185</td>
<td></td>
<td></td>
</tr>
<tr>
<td>186</td>
<td></td>
<td></td>
</tr>
<tr>
<td>187</td>
<td>LDH &quot;qCurrent--,temp2&quot;</td>
<td></td>
</tr>
<tr>
<td>188</td>
<td>NOP 3</td>
<td></td>
</tr>
<tr>
<td>189</td>
<td>CMPGT templ,0,cond1</td>
<td></td>
</tr>
<tr>
<td>190</td>
<td>[ cond1] CMPEQ temp2,0,cond1</td>
<td></td>
</tr>
<tr>
<td>191</td>
<td>[ cond1] B</td>
<td>ASMGFPolyDivideLoop4</td>
</tr>
<tr>
<td>192</td>
<td>[ cond1] ADDK -1,templ</td>
<td></td>
</tr>
<tr>
<td>193</td>
<td>NOP 4</td>
<td></td>
</tr>
</tbody>
</table>

*** END OF ASMGFPolyDivideLoop4

**ASMGFPolyDivideLoop4Continue:**

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>194</td>
<td></td>
<td></td>
</tr>
<tr>
<td>195</td>
<td></td>
<td></td>
</tr>
<tr>
<td>196</td>
<td></td>
<td></td>
</tr>
<tr>
<td>197</td>
<td></td>
<td></td>
</tr>
<tr>
<td>198</td>
<td></td>
<td></td>
</tr>
<tr>
<td>199</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>201</td>
<td></td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>STW templ,&quot;qD&quot;</td>
<td></td>
</tr>
</tbody>
</table>
ASMGFPolyDivideLoop5:
*** BGN OF ASMGFPolyDivideLoop5
*** This loop reduces the degree of r.

203    NOP     3
204    ADDAH   r,templ,tempSide
205    MV       tempSide,rCurrent
206
207    ASMGFPolyDivideLoop5:
208    *** BGN OF ASMGFPolyDivideLoop5
209    *** This loop reduces the degree of r.
210
211
212    LDH       *rCurrent--,temp2
213
214    NOP      3
215    CMPGT     templ,0,cond1
216    [ cond1] CMPEQ   temp2,0,cond1
217    [ cond1] B ASMGFPolyDivideLoop5
218
219    [ cond1] ADDK    -1,templ
220    NOP     4
221
222    *** END OF ASMGFPolyDivideLoop5
223
224    ASMGFPolyDivideExit:
225    B B3
226
227    STW       templ,*rD
228    NOP     4
Appendix C – Straight Assembly Files

This section includes the straight-assembly files that were written for this project. In general, they are direct implementations of the corresponding C functions. These files were written after the regular assembly routines were written. This is contrary to the normal C62x development flow, in which regular assembly is written only after straight-assembly has been written and proven lacking. In this project, the straight-assembly files were written to obtain an upper limit on the performance of the assembly implementations, and to measure the performance of the regular assembly routines.

Note that the straight-assembly files are much more readable than their regular assembly counterparts (they were much easier to write, as well). Note also the similarities in program flow.

SAGFFourier – 16-bit Straight-Assembly GFFourier

```assembly
.SAGFFourier: .cproc
SAGFFourierInit:
.reg antilogTable, logTable
NVK _antilogTable,antilogTable
NVKH _antilogTable,antilogTable
NVK _logTable,logTable
NVKH _logTable,logTable
.reg N
LDW *+code[1],N
```
.reg numberOfOutputSymbols,constantValue
.reg startingIndexStep,startingIndex
.reg indexStepStep,indexStep
LDW /*parameters[0],numberOfOutputSymbols
LDH /*parameters[2],constantValue
LDH /*parameters[3],startingIndex
LDH /*parameters[4],startingIndexStep
LDH /*parameters[5],indexStep
LDH /*parameters[6],indexStepStep
.reg counter1,counter2
.reg pl
MV output.pl
MV numberOfOutputSymbols,counter1
SAGFFourierOutputInitLoop:
| STH constantValue,*pl++
| [ counter1] ADDK -1,counter1
| [ counter1] B SAGFFourierOutputInitLoop

SAGFFourierLoop1:
LDH /*input++,templ
CMPEQ templ.0,cond1
[ cond1] B SAGFFourierLoop1Continue

.reg index,temp2
LDH /*logTable[templ],templ
ADD templ.startingIndex,index
CMPLT index,N,cond1
[!cond1] SUB index,N,index
MV numberOfOutputSymbols,counter2
MV output.pl
SAGFFourierLoop1A:
LDH /*pl,templ
LDH /*antilogTable[index],temp2
XOR templ.temp2.temp1
STH templ,*pl--
ADD index,indexStep,index
CMPLT index,N,cond1
[!cond1] SUB index,N,index
[ counter2] ADDK -1.counter2
[ counter2] B SAGFFourierLoop1A

*** END OF SAGFFourierLoop1A

SAGFFourierLoop1Continue:
ADD startingIndex,startingIndexStep,startingIndex
CMPLT startingIndex,N,cond1
[!cond1] SUB startingIndex,N,startingIndex
ADD indexStep,indexStepStep,indexStep
CMPLT indexStep,N,cond1
[!cond1] SUB indexStep,N,indexStep
[ counter1] ADDK -1.counter1
[ counter1] B SAGFFourierLoop1

*** END OF SAGFFourierLoop1
SARSDiscrepancy – 16-bit RSDiscrepancy

*SARSDiscrepancy() in TMS320C5201 Straight-Assembly
*(input to assembly optimizer)
*C-Callable
*16-bit RSSymbol
*16-bit RSLogSymbol
*Written by: Kamal Swamidoss
*December 1997
*Based on: C Code from Jon Rowlands
*RSSymbol
*SARSDiscrepancy(
*RSCode *code,
*int i,
*int errorLocatorDegree,
*RSLogSymbol *logSyndrome,
*RSLogSymbol *logErrorLocator
*
*
*********************************************************************/

MYTABSIZE .set 8
MYPAGEWIDTH .set 78
MYPAGELENGTH .set 75

.tab MYTABSIZE
.width MYPAGEWIDTH
.length MYPAGELENGTH
.align 32

.def _SARSDiscrepancy
.ref _antilogTable

.text
_SARSDiscrepancy: .cproc code,i,errorLocatorDegree,logSyndrome,logErrorLocator
.reg antilogTable
.MVX _antilogTable,antilogTable
.MVXH _antilogTable,antilogTable

.ADDAH logSyndrome,i,logSyndrome
.reg counter
.ADD errorLocatorDegree,1,counter

.reg templ,temp2,temp3
.reg cond1
.reg discrepancy
.ZERO discrepancy

SARSDiscrepancyLoop1:
.LDH *logErrorLocator++,templ
.LDH *logSyndrome--,temp2
.ADD templ,temp2,temp3
.LDH *( antilogTable[temp3],temp3
.XOR discrepancy,temp3,discrepancy
{ counter} ADDK -1,counter
SAGFPolyXOR – 16-bit GFPolyXOR

*******************************************************************************
1  *
2  * SAGFPolyXOR() in TMS320C6201 Straight Assembly
3  * (input to assembly optimizer)
4  * C-callable
5  * 16-bit RSsymbol
6  * 16-bit RSLogSymbol
7  *
8  * Written by: Kamal Swamidoss
9  * December 1997
10 *
11 *
12 * void
13 * SAGFPolyXOR(short *a,
14 * short *b,
15 * int aD,
16 * int bD,
17 * short *x,
18 * int *xD);
19 *
*******************************************************************************

MYTABSIZE .set 8
MYPAGEWIDTH .set 78
MYPAGELENGTH .set 75

.tab MYTABSIZE
.width MYPAGEWIDTH
.length MYPAGELENGTH
.align 32

.def _SAGFPolyXOR

.text

_SAGFPolyXOR: .cproc a,b,aD,bD,x,xD

.reg cond1,temp1,temp2
CMPLT aD,bD,cond1

[ cond1] MV a,temp1
[ cond1] MV b,a
[ cond1] MV temp1,b
[ cond1] MV bd,temp2
[ cond1] MV aD,bD
[ cond1] MV temp2,aD

.reg counter
ADD bD,1,counter

.reg i,j,k

SAGFPolyXORLoop1:
LDH *a++,i
LDH *b++,j
XOR i,j,k
STM k,*x++

[ counter] ADDK -1,counter
[ counter] B SAGFPolyXORLoop1
SUB aD.bD,counter

SAGFPolyXORLoop2:

[ counter] LDH "a++,i
[ counter] STH i,"x++
[ counter] ADDK -1,counter
[ counter] B SAGFPolyXORLoop2

SUBAH x,1.x
MV aD.counter

[ counter] B SAGFPolyXORLoop3

STW counter,*xD
.return

SAGFPolyXORLoop3:

LDH "x--,k
CMPEQ k,0,cond1
AND cond1,counter,cond1

[ cond1] ADDK -1,counter
[ cond1] B SAGFPolyXORLoop3

STW counter,*xD
.return
.endproc

SAGFPolyMultiply – 16-bit GFPolyMultiply

******************************************************************************

* SAGFPolyMultiply() in TMS320C6201 Straight-Assembly
* (input to assembly-optimizer)
* C-callable
* 16-bit RSSymbol
* 16-bit RSLogSymbol
* Written by: Kamal Swamidoss
* December 1997
* void
* SAGFPolyMultiply( short *a,
* short *b,
* int aD,
* int bD,
* short *p,
* int *pD);

******************************************************************************

MYTABSIZE .set 8
MYPAGEWIDTH .set 78
MYPAGELENGTH .set 75
.tab MYTABSIZE
.width MYPAGEWIDTH
.length MYPAGELENGTH
.align 32
.def _SAGFPolyMultiply
.ref _antilogTable,_logTable
.text
_SAGFPolyMultiply: .cproc
.a.b,aD,bD,p,pD
.reg
antilogTable,logTable
MVK
_antiLogTable,antiLogTable
MVKH
_antiLogTable,antiLogTable
MVK
_logTable,logTable
MVKH
_logTable,logTable

[ cond1] MV a,temp1
[ cond1] MV b,a
[ cond1] MV temp1,b
[ cond1] MV bD,temp2
[ cond1] MV aD,bD
[ cond1] MV temp2,aD

.reg
.temp3,temp4
ADD aD,bD,temp3
STW temp3, tempD

.reg
counter
ADD temp3,1,counter
ZERO temp1
MV p,temp2

STH temp1,*temp2++
[ counter] ADDK -1,counter
[ counter] B SAGFPolyMultiplyInitLoop

.reg
counter1,counter2
ADD bD,1,counter2
.reg
.ptr1,ptr2,ptr3
MV b,ptr2
MV p,ptr3
.reg
.productProgress
ADDAH p,1,productProgress

ADD aD,1,counter1
MV a,ptr1

LDH *ptr1,temp1
LDH *ptr2,temp2
LDH *-logTable[temp1],temp1
LDH *-logTable[temp2],temp2
ADD templ,temp2,temp4
LDH *-antiLogTable[temp4],temp4
LDH *ptr3,temp3
XOR temp3,temp4,temp3
STH temp3, tempD
ADDAH ptr1,1,ptr1
ADDAH ptr3,1,ptr3

[ counter1] ADDK -1,counter1
[ counter1] B SAGFPolyMultiplyLoop1A
99 ADDAH ptr2,1,prr2
100 MV a.ptr1
101 MV productProgress,ptr3
102 ADDAH productProgress,1,productProgress
103 [ counter2] ADDK -1,counter2
104 [ counter2] B SAGFPolyMultiplyLoop1
105 LDW *pD,temp2
106 ADDAH p,temp2.ptr3
107 .reg cond2
108 SAGFPolyMultiplyLoop2:
109 LDH *ptr3--,temp3
110 CMPGT temp2,0,cond1
111 ADD cond1,cond2,cond1
112 [ cond1] ADDK -1,temp2
113 [ cond1] B SAGFPolyMultiplyLoop2
114 STW temp2,*pD
115 .return
116 .endproc

SAGFPolyDivide - 16-bit GFPolyDivide

******************************************************************************

* void SAGFPolyDivide(
  * RSSymbol *numerator,
  * RSSymbol *denominator,
  * RSSymbol *quotient,
  * RSSymbol *remainder,
  * int numeratorDegree,
  * int denominatorDegree,
  * int *quotientDegree,
  * int *remainderDegree,
  * RSCode *code)

******************************************************************************

MYTABSIZE .set 8
MYPAGEWIDTH .set 78
MYPAGELENGTH .set 75
.tab MYTABSIZE
.width MYPAGEWIDTH
.length MYPAGELENGTH
.align 32
.def _SAGFPolyDivide
.ref _logTable,_antilogTable
.text
_SAGFPolyDivide:
cproc n,d,q,r,nD,dD,qD,rD,code
SAGFPolyDivideInit:
.reg antilogTable,logTable
MVK _antilogTable,antilogTable
MVKH _antilogTable,antilogTable
MVK _logTable,logTable
MVKH _logTable,logTable
.reg cond1
ADD nD,1,cond1
.reg pl,p2
MV n.pl
MV r.p2
.reg temp1,temp2
SAGFPolyDivideLoop1:
*** BGN OF SAGFPolyDivideLoop1
*** This loop copies n to r.
LDH *pl++,temp1
STH temp1,*p2++
[ cond1] ADDK -1,cond1
[ cond1] B SAGFPolyDivideLoop1
*** END OF SAGFPolyDivideLoop1
STW nD,*rD
CMPLT nD,dD,cond1
[ cond1] ZERO temp1
[ cond1] STW temp1,*qD
[ cond1] STH temp1,*q
[!cond1] B SAGFPolyDivideContinuel
.return
SAGFPolyDivideContinuel:
SUB nD,dD,temp1
STW temp1,*qD
.reg qCurrent
ADDAH q,templ,qCurrent
LDW *rD,temp1
.reg N
LDW *+code[1],N
.reg rCurrent,dCurrent
SAGFPolyDivideLoop2:
*** BGN OF SAGFPolyDivideLoop2
*** This is the main loop.
CMPLT templ,dD,cond1
[ cond1] B SAGFPolyDivideLoop2Continue
ADDAH r,templ.pl
ADDAH d,dD,p2
LDH *pl,rCurrent
LDH *p2,dCurrent
LDH *+logTable[dCurrent],dCurrent
LDH *+logTable[rCurrent],rCurrent
SUB N,dCurrent,temp2
ADD rCurrent,temp2,temp1
LDH *+antilogTable[temp1],rCurrent
STH rCurrent,*qCurrent--
LDH *+logTable[rCurrent],temp1
; temp1 contains log(div) in correct interval
ADD dD,1,cond1

SAGFPolyDivideLoop2A:
*** BGN OF SAGFPolyDivideLoop2A
*** This loop makes the new remainder.
LDH *p2--,temp2
LDH *+logTable[temp2],temp2
ADD temp1,temp2,temp2
LDH *+antilogTable[temp2],temp2
LDH *pl,rCurrent
XOR rCurrent.temp2.temp2
STH temp2,*pl--
[ cond1] ADDK -1,cond1
[ cond1] B SAGFPolyDivideLoop2A
*** END OF SAGFPolyDivideLoop2A

SAGFPolyDivideLoop2Continue:
ZERO temp1
SAGFPolyDivideLoop3:
*** BGN OF SAGFPolyDivideLoop3
*** This loop zeros the remaining coefficients of q.
CMPLT temp1.0,cond1
[!cond1] STW temp1,*rD
B SAGFPolyDivideLoop2
*** END OF SAGFPolyDivideLoop2
SAGFPolyDivideLoop3:
CMPLT qCurrent.q,cond1
[!cond1] STH temp1,*qCurrent--
[!cond1] B SAGFPolyDivideLoop3
*** END OF SAGFPolyDivideLoop3

SAGFPolyDivideLoop4:
*** BGN OF SAGFPolyDivideLoop4
*** This loop reduces the degree of q.
LDW ADDAH q,templ,qCurrent
164 CMPGT template,0,condl
165 [ cond1] LDH *qCurrent--,temp2
166 [ cond1] CMPEQ temp2,0,condl
167 [ cond1] ADDK -1,templ
168 [ cond1] B SAGFPolyDivideLoop4

*** END OF SAGFPolyDivideLoop4

170 STW templ,*qD
171 LDW *rD,templ
172 ADDAH r,templ,rCurrent

SAGFPolyDivideLoop5:
*** BGN OF SAGFPolyDivideLoop5
*** This loop reduces the degree of r.

180 CMPGT template,0,condl
181 [ cond1] LDH *rCurrent--,temp2
182 [ cond1] CMPEQ temp2,0,condl
183 [ cond1] ADDK -1,templ
184 [ cond1] B SAGFPolyDivideLoop5

*** END OF SAGFPolyDivideLoop5

SAGFPolyDivideExit:

190 STW templ,*rD
191 .return
192 .endproc
Appendix D - README File for the Modified RS Decoder

This file was written to explain the modified RS decoder.

RSDecodeTest - Reed-Solomon Forward-Error-Correction Decoder Test
GFPolyArith - Galois-Field Polynomial Arithmetic Calculator
genrs - Reed-Solomon Code Generator

Kamal Swamidoss
December 1997

This directory contains source code for three programs. Two of the programs are closely-connected. The third is completely independent. genrs is the independent one. RSDecodeTest and GFPolyArith share a lot of files and functions, but they can only be compiled separately. The program which I optimized is RSDecodeTest. The other two programs are useful in their own ways. This file tries to explain all this in detail.

SECTION ONE: QUICKSTART

Here's a brief step-by-step guide to using the programs in this directory. All this is discussed in detail in this file:

1. Make the two data files for the RS code you want to use. Use the program genrs to make these two files.
   prompt% cd genrs
   Copy the appropriate RS Code parameter file to prmrs.prm.
   prompt% cp prmrs.prm.adsl prmrs.prm
   prompt% genrs prmrs.prm

2. Copy the two output files from genrs to the main directory.
   prompt% cp prmrs.c ..
   prompt% cp prmrs.h ..
   prompt% cd ..

3. Set the flags in the file modefile.h. If you're going to compile for the Sun, only set the flags in the Sun section. If you're going to compile for the c6x, only set the flags in the DSP section. Whatever you do, DON'T modify the flags after the line 'You shouldn't need to change anything below here.'
   prompt% xemacs modefile.h

4. Build the program you want.
   A. If you're building for the c6x, you can only make the RS Decode Test. You can't make the Galois-Field Polynomial Arithmetic Calculator.
      I recommend building c6x programs in the directory DSPVersion.
      prompt% cd DSPVersion
      If you need to assemble some assembly files, do this:
      prompt% cl6x file1.asm file2.asm ...
      Compile the C files.
      prompt% cl6x -o -pm -dMakeExecutable=0 ..//*.c
      prompt% lnk6x *.obj dsp.cmd -o dsp.out
      Note that the only command-line flag you have to set is MakeExecutable. Making it equal to zero indicates that you want a c6x program. That's obvious, considering you're using cl6x and not gcc, right? But setting MakeExecutable to zero sets other compiler flags in modefile.h. These flags are used to compile the program in different ways. See modefile.h for details.
Note also that this program comes with its own linker command file.

If you've enabled console output, do this:

```
prompt% load6x dsp.out
```

Otherwise, do this:

```
prompt% sim6x dsp.out
```

Note that this program also comes with its own simulator configuration files: init.clr, init.cmd, sim6x.cfg, and siminit.cmd.

B. If you're building for the Sun, you can make either the RS Decode Test or the Galois-Field Polynomial Arithmetic Calculator.

I recommend building the Sun programs in the directory SunVersion.

```
prompt% cd SunVersion
```

Compile the C files. Note the gcc flags.

```
prompt% gcc -Wall -Wformat -ansi -pedantic -c ../*.c
```

Note that you don't have to set MakeExecutable when compiling for the Sun. That's because modefile.h tells the compiler to compile the Sun version by default.

If you're making the GFPolyArith, link like this:

```
prompt% gcc *.o -o gfpa
```

If you're making the RSDecodeTest, link like this:

```
prompt% gcc *.o -o rsdt
```

Run the program.

SECTION TWO: THE DETAILS

-----------------------------

PART A: genrs

genrs is used to generate a Reed-Solomon code. What do I mean by this?
genrs reads a parameter file which specifies all the parameters of some Reed-Solomon code, and it outputs two files which contain data structures for that RS code. Take a look at the file "genrs/prmrs.prm.adsl" to see what the RS code parameters are. This particular parameter file specifies the RS code for part of ADSL, the Asymmetric Digital Subscriber Loop standard.

The two output files from genrs are compiled with the other files to make either the RSDecodeTest or the GFPolyArith calculator. genrs provides the flexibility to use one of several RS codes in those programs. You can even make your own RS code parameter file for use with genrs. I did that with the file prmrs.prm.xmpl, which is a code from the book "Error Control Systems for Digital Communication and Storage" by Stephen B. Wicker.

NOTE 1: In order to maintain compatibility with RSDecodeTest and GFPolyArith, the input parameter file for genrs MUST be called prmrs.prm. The genrs directory contains parameter files for several RS codes. In order to generate files for a particular code, copy the parameter file to prmrs.prm and run genrs. For example, to generate the MPEG RS code do this:

```
prompt% cp prmrs.prm.mpeg prmrs.prm
prompt% genrs prmrs.prm
prompt% mv prmrs.c ..
```
PART B: Include Files Generated by RSDecodeTest
Sometimes the Sun version of RSDecodeTest can write some output files. These files are myrsusr.h, myrssnd.h, myrsrvr.h, and myrsend.h. These files can be used as include files the next time you compile RSDecodeTest for either the Sun or the c6x. The files are the data which the program generates and manipulates. These files can be convenient if you don't want to generate data and RS encode it every time you run the program. Remember that these files are only valid for a particular RS code, so if you change the code, you can't use the old include files.

To generate the include files, #define WriteIncludeFiles in modefile.h. Then build RSDecodeTest on the Sun, in the SunVersion directory. The four files will be in that directory when the program finishes. Move these files to the main directory. When run, RSDecodeTest generates data, RS-encodes it, corrupts the codeword, and RS-decodes the result. Data at each stage of the process is saved to one of the four output files.

To use the include files, #define ReadIncludeFiles in modefile.h. Then build RSDecodeTest on either the Sun or the c6x. When run, RSDecodeTest won't generate data, encode it, and corrupt the codeword. At each iteration, RSDecodeTest will read a block from the data in myrsusr.h, read a block from the data in myrssnd.h, and read a block from the data in myrsrvr.h. The data in myrsusr.h is the "user data," the data to be encoded and transmitted. The data in myrssnd.h is the RS codeword. The data in myrsrvr.h is the corrupted codeword. The only operation RSDecodeTest does when ReadIncludeFiles is #define'd is the RS decoding.

NOTE 3: If you #define ReadIncludeFiles and you've set numberOfCodewordsToTest to a number larger than the number of codewords represented in the include files, then you'll get a compiler error.

PART C: modefile.h
The only other complicated thing is modefile.h. This file is included in all the main C files. It consists completely of comments and preprocessor flags. These flags are used by the compiler to build different programs. The comments in modefile.h describe what the different flags are for. I'd recommend reading modefile.h, compiling it as is, and then trying one change at a time, until you're comfortable with what it does.

SECTION THREE: THE RS DECODE ALGORITHM
-----------------------------------------
That's about it. The optimizations I made are for the c6x version of RSDecodeTest. These optimizations are various assembly routines to replace C functions. These assembly routines are:

- **ASMFFourier** This provides a good performance gain.
- **ASMFFourier12** ASMFFourier with 32-bit RSsymbol, RSLogSymbol.
- **ASMRSDiscrepancy** Also a considerable gain.
- **ASMRSDiscrepancy32** ASMRSDiscrepancy with 32-bit RSsymbol, RSLogSymbol.
- **ASMGFPolynomial** Negligible gain.
- **ASMGFPolyMultiply** A LOSS of performance from the corresponding C code!
- **ASMGFPolyDivide** A significant loss!
- **SAGFFourier** Straight-Assembly
- **SARDiscrepancy** Straight-Assembly
- **SAGGFPolynomial** Straight-Assembly
- **SAGGFPolynomialMultiply** Straight-Assembly
- **SAGGFPolynomialDivide** Straight-Assembly

prompt% my prmrs.h ...
* The 32-bit routines are incompatible with 16-bit data.

The rest of the routines are meant to run on 16-bit data.

You can set flags to include or exclude each of these routines.

If you want to understand the RSDecode algorithm, start at the function RSDecode() in rs.c. There are a few different steps in the algorithm, and each step has a corresponding function in RSDecode().

If you want to learn about RS decoding, the tutorial by TI's own Jon Rowlands is excellent. It also provides references to the authorities.

Kamal Swamidoss

December 1997
Appendix E – Modefile

This is the most important control file for the modified RS decoder. This file is included at the beginning of every C source file comprising the RS decoder. It contains all the preprocessor data needed to control the compilation of the decoder. It lets the user tell the compiler how to build the decoder.

/*
 * ModeFile
 * Kamal Swamidoss
 * November 1997
 * This file helps you make executables of the RS Decode Test for either the Sun or the c6x.
 * There are two basic MakeExecutable modes: 0 and 1.
 * 0 means make for the DSP.
 * 1 means make for the Sun.
 * Just define MakeExecutable at the command-line when you compile. This package was tested by compiling with the following commands.
 * gcc -Wall -Wformat -ansi -pedantic -c -DMakeExecutable=1 *.c
 * cl6x -g -as -o -DMakeExecutable=0 *.c
 * I recommend making the object files in the directories SunVersion and DSPVersion, respectively. It keeps the main directory clean.
 * Link the object files to create your executable. This is how I did it.
 * gcc *.o -o sun.out
 * lnk6x *.obj dsp.cmd -o dsp.out
 * If you’re making a c6x .out file, I recommend using the command file in the directory DSPVersion. That directory also contains some assembly files and some c6x simulator initialization/configuration files.
 * gffr16.asm
 * gffr16.inc This is the assembly for ASMGFFourier(), a function that works like GFFourier(), but it’s faster. These files must be assembled if UseASMGFFourier is defined below.
 * gffr32.asm Assembly for ASMGFFourier32(). 32-bit RSSymbol and RSLogSymbol. Incompatible with 16-bit program.
 * rsds.asm
 * rsds.inc This is the assembly for ASMRSDiscrepancy(), a function that works like RSDiscrepancy(), but it’s faster. These files must be assembled if UseASMRSDiscrepancy is defined below.
 * rsds32.asm Assembly for ASMRSDiscrepancy32(). 32-bit RSSymbol and RSLogSymbol. Incompatible with 16-bit program.
 * init.cmd
 * init.clr
 * simint.cmd Simulator initialization/configuration files.
 * This is a description of the flags listed in this file.
 * RunGFPArith
 * There are actually two main() functions in this directory.
 * The first is at the end of RSDecodeTest.c. That main() is used to run the RS Decode Test. The second main() is at the end of GFPArith.c, and it’s used to run the GF Polynomial Arithmetic Test. This is a little program which is designed to run only on the Sun. It allows the user to perform GF arithmetic on two polynomials at a time.
* ReadIncludeFiles
  * Read start data from include files? See RSDecodeTest.c.
  * The files myrsusr.h, myrssnd.h, and myrsrchv.h
  * are included during the compile. These files are generated
  * by this program when the WriteIncludeFiles flag is defined
  * (see below). They contain data which can be used directly.
  * This can eliminate the time involved in pseudo-randomly
  * generating user data, encoding it, and pseudo-randomly
  * corrupting it. myrsusr.h contains an array of arrays
  * contain user data symbols. myrssnd.h contains corresponding
  * arrays containing RS codewords. myrsrchv.h contains corresponding
  * arrays of "corrupted" symbols.

* WriteIncludeFiles
  * Write data to include files when done? See RSDecodeTest.c.
  * The files myrsusr.h, myrssnd.h, myrsrchv.h, and myrsend.h
  * are generated
  * by the program. They contain arrays of
  * symbol arrays.
  * "usr" stands for user, "snd" stands for send.
  * "rcv" stands for receive, and "end" stands for end.
  * myrsend.h contains an array of RS-decoded symbol arrays.

* EnableConsoleOutput
  * This lets the program write console output.

* UseMyRSEuclid
  * Use the RSEuclid library of functions? See the GF Polynomial
  * Arithmetic section, near the end of rs.c.

* UseASMGFPolyXOR
* UseASMGFPolyMultiply
* UseASMGFPolyDivide
  * Use hand-coded assembly routines for the different
  * Galois-Field polynomial arithmetic operations?
  * Don't define any of these for Sun executables.

* UseSAGFPolyXOR
* UseSAGFPolyMultiply
* UseSAGFPolyDivide
  * Use the auto-optimized c6x routines for the different
  * Galois-Field polynomial arithmetic operations?
  * Don't define any of these for Sun executables.

  * NOTE: UseMyRSEuclid must be defined if any of {UseASMGFPolyXOR,
      * UseASMGFPolyMultiply, UseASMGFPolyDivide, UseSAGFPolyXOR,
      * UseSAGFPolyMultiply, UseSAGFPolyDivide} are defined.

* UseASMGFFourier
  * Use the ASMGFFourier c6x routine? See gffr16.asm gffr16.inc.
  * Don't define this for Sun executables.

* UseASMGFFourier32
  * Use ASMGFFourier32? See gffr32.asm. Don't define for Sun.

* UseSAGFFourier
  * Use the auto-optimized GFFourier c6x routine? See gffrl6sa.sa.
  * Don't define this for Sun executables.

  * NOTE: At most one of {UseASMGFFourier, UseSAGFFourier} may be defined
      * at one time.

* UseASMRSDiscrepancy
  * Use the ASMRSDiscrepancy c6x routine? See rsds.asm and rsds.inc.
  * Don't define this for Sun executables.
UseASMRSDiscrepancy32

UseSARSDiscrepancy
Use the auto-optimized RSDiscrepancy c6x routine? See rsdsssa.asm.
Don't define this for Sun executables.

NOTE: At most one of UseASMRSDiscrepancy, UseSARSDiscrepancy may be
defined at one time.

NOTE: The RSDiscrepancy assembly routines will only be called if
UseMyRSEuclid is not defined.

UseInline
This flag is used in rs.c. Some small functions can be inlined.

UseStatic
This flag is used in rs.c. The functions are made static.

NOTE: At most one of {UseASMRSDiscrepancy, UseSARSDiscrepancy} may be
defined at one time.

UseMyRSEuclid
/* This tells the compiler to make a Sun executable by default. */
#if !defined(MakeExecutable)
#define MakeExecutable 1
#endif

/* DSP (c6x) Parameters
Some small functions can be inlined.
This flag is used in rs.c. The functions are made static.
*/

/* DSP (c6x) Parameters
Some small functions can be inlined.
This flag is used in rs.c. The functions are made static.
*/

#ifdef ReadIncludeFiles
#define WriteIncludeFiles
#undef UseASMGFFourier
/* Use 16-bit ASMGFFourier assembly routine? */
#undef UseSAGFFourier
/* Use 16-bit SAGFFourier assembly routine? */
#undef UseASMGFFourier32
/* Use 32-bit ASMGFFourier32 assembly routine? */
#undef UseASMRSDiscrepancy
/* Use 16-bit ASMRSDiscrepancy assembly routine? */
#undef UseSARSDiscrepancy
/* Use 16-bit SARSDiscrepancy assembly routine? */
#undef UseASMRSDiscrepancy32
/* Use 32-bit ASMRSDiscrepancy32 assembly routine? */
#define UseMyRSEuclid /* Use Euclid's algorithm? */
#undef UseASMGFPolyXOR /* Use 16-bit ASMGFPolyXOR assembly routine? */
#undef UseSAGFPolyXOR /* Use 16-bit SAGFPolyXOR assembly routine? */
#undef UseASMGFPolyMultiply /* Use 16-bit ASMGFPolyMultiply assembly routine? */
#undef UseSAGFPolyMultiply /* Use 16-bit SAGFPolyMultiply assembly routine? */
#undef UseASMGFPolyDivide /* Use 16-bit ASMGFPolyDivide assembly routine? */
#undef UseSAGFPolyDivide /* Use 16-bit SAGFPolyDivide assembly routine? */
#define UseInline
#define UseStatic

#elif (MakeExecutable == 0)
#define ReadIncludeFiles
#undef WriteIncludeFiles
#undef UseMyRSEuclid
#undef RunGFPolyArith
/* Compile the GF Poly. Arith. package? */
#else
#error Invalid Executable Mode
#endif
/
* You shouldn't need to change anything below here.
*/
/
* The 32-bit routines are incompatible with the 16-bit routines.
*/
#endif
#endif
#endif
#define(UseASMGFFourier) || defined(UseSAGFFourier) || \
    defined(UseASMRSDiscrepancy) || defined(UseSARSDiscrepancy) || \
    defined(UseASMGFPolyXOR) || defined(UseSAGFPolyXOR) || \
    defined(UseASMGFPolyMultiply) || defined(UseSAGFPolyMultiply) || \
    defined(UseASMGFPolyDivide) || defined(UseSAGFPolyDivide))
#undef UseASMGFFourier32
#undef UseASMRSDiscrepancy32
#endif
#endif
#if defined(UseASMGFFourier)
#undef UseSAGFFourier
#endif
#if defined(UseSAGFFourier)
#undef UseASMGFFourier
#endif
#if defined(UseASMRSDiscrepancy)
#undef UseSARSDiscrepancy
#endif
#endif
#endif
#endif
#define(UseMyRSEuclid)
#undef UseASMRSDiscrepancy
#undef UseSARSDiscrepancy
#undef UseASMRSDiscrepancy32
#else
#undef UseASMGFPolyXOR
#undef UseASMGFPolyMultiply
#undef UseASMGFPolyDivide
#undef UseSAGFPolyXOR
#undef UseSAGFPolyMultiply
#undef UseSAGFPolyDivide
#endif
#endif
#if defined(UseASMGFFourier)
#undef UseSAGFFourier
#endif
#if defined(UseSAGFFourier)
#undef UseASMGFFourier
#endif
#if defined(UseASMRSDiscrepancy)
#undef UseSARSDiscrepancy
#endif
#undef UseSARSDiscrepancy

#ifndef
#endif

#if defined(UseSARSDiscrepancy)
#endif

#idefinf(UseASMRSDiscrepancy)
#endif

#if defined(UseASMGFPolyXOR)
#endif

#if defined(UseASMGFPolyXOR)
#endif

#if defined(UseASMGFPolyMultiply)
#endif

#if defined(UseASMGFPolyMultiply)
#endif

#if defined(UseASMGFPolyDivide)
#endif

#if defined(UseASMGFPolyDivide)
#endif

#if (MakeExecutable == 0)
#include <time.h> /* For cycle-counting. */
#endif

#elif (MakeExecutable == 1)
#endif

/* The following flags allow the use of certain c6x assembly routines.
* These routines cannot be executed on the Sun.
*/

#undef UseASMGFPolyXOR
#undef UseASMGFPolyMultiply
#undef UseASMGFPolyDivide

#undef UseSAGFPolyXOR
#undef UseSAGFPolyMultiply
#undef UseSAGFPolyDivide

#undef UseASMGFPolyFourier
#undef UseSAGFPolyFourier
#undef UseASMGFPolyFourier32
#undef UseASMGFPolyFourier32

#undef UseASMRSDiscrepancy
#undef UseASMRSDiscrepancy
#undef UseASMRSDiscrepancy32
#undef UseASMRSDiscrepancy32
Appendix F – GFPolyArith Sun Program

This is a C program written for SunOS 4.1.4. It is a two-polynomial arithmetic calculator. It uses the GF arithmetic functions and the RSCode structure from the RS library. This program was written to help debug the implementation of Euclid’s algorithm in the RS decoder. The user can input two GF polynomials, coefficient by coefficient, and specify one of four operations. The program outputs the result(s), in normal and log form. It can use any RS code that can be used by the RS decoder.

```c
#include "modefile.h"
static int filler=0;
#if defined(RunGFPolyArith)
# include <stdio.h>
# include <string.h>
# include <math.h>
# include "prmrs.h"

void PrintPoly(char *name,RSSymbol *poly) {
    RSSymbol *hold;
    printf("%s",name);
    hold = poly;
    while (*poly != -1)
        printf(" %d",(int) *poly++);
    puts(" ");
    printf("%s",name);
    poly = hold;
    while (*poly != -1)
        printf("a%d ",(int) GFLog(&StandardRSCode,*poly++));
    puts(" ");
}

void GFPolyArithMultiply (RSSymbol *a,RSSymbol *b,RSSymbol *p) {
    RSSymbol *pl,*p3,*c;
    int i;
    c = p;
    p3 = c;
    for (i=0;i<64;++i)
        c[i] = 0;
    while (*b != -1) {
        pl = a;
        p3 = c;
        while (*pl != -1) {
            *p3 = GFAdd(&StandardRSCode,
                        *p3,
                        GFPolyArithMultiply(&StandardRSCode,  
                        *p1,  
                        "b");
            ++pl;
            ++p3;
        }
        ++b;
        ++c;
    }
```

56 *p3 = -1;
57 p3 = p;
58 while (*p3 != -1)
59 ++p3;
60 --p3;
61 while ((*p3 == 0) && (p3 > p))
62 *p3 = -1;
63 return;
64 }
65
66 void GFPolyArithDivide(RSSymbol *n, RSSymbol *d, RSSymbol *q, RSSymbol *r) {
68 RSSymbol div, prod;
69 int nd, dd, qd, rd;
70
71 if ((*n == -1) || (*d == -1)) {
72 *q = -1;
73 *r = -1;
74 return;
75 }
76
77 nd = -1;
78 p1 = n;
79 while (*p1++ != -1)
80 ++nd;
81 dd = -1;
82 p1 = d;
83 while (*p1++ != -1)
84 ++dd;
85 qd = nd - dd;
86 rd = nd;
87 p1 = n;
88 p2 = r;
89 while (*p1 != -1)
90 *p2++ = *p1++;
91
92 if (dd > nd) {
93 *q++ = 0;
94 *q = -1;
95 return;
96 }
97
98 p3 = &q[qd];
99 while (rd >= dd) {
100 div = GF Divide (&StandardRSCode, r[rd], d[dd]);
101 p2 = &d/dd];
102 *p3-- = div;
103 p4 = &r[rd];
104 while (*p2 >= d) {
105 prod = GF Multiply (&StandardRSCode, *p2, div);
106 *p4 = GF Subtract (&StandardRSCode, *p4, prod);
107 --p2;
108 --p4;
109 }
110 }
111 }
while (p3 >= q)
    *p3-- = 0;
*p3-- = -1;
p3 = &q[qd+1];
while ((*p3 == 0) && (p3 > q))
    *p3-- = -1;
p4 = &r[nd+1];
*p4-- = -1;
while ((*p4 == 0) && (p4 > r))
    *p4-- = -1;
return;
}

void GFPolynomialXOR(RSSymbol *a, RSSymbol *b, RSSymbol *x) {
    RSSymbol *p, *c;
    c = x;
    while (*a != -1) {
        if (*b == -1)
            break;
        *c++ = GFAdd(&StandardRSCode, *a++, *b++);
    }
    if (*b == -1)
        while (*a != -1)
            *c++ = *a++;
    else if (*a == -1)
        while (*b != -1)
            *c++ = *b++;
    *c = -1;
    p = x;
    while (*p != -1)
        ++p;
    --p;
    while ((*p == 0) && (p > x))
        *p-- = -1;
    return;
}

RSSymbol upperBound = 0;

int StrToPoly(char *s, RSSymbol *p) {
    char *token;
    token = strtok(s, " ");
    while (token != NULL) {
        if (*token == 'a') {
            *p = (RSSymbol) atoi(token);
            *p = GFAdd(&StandardRSCode, *p);
        } else {
            *p = (RSSymbol) atoi(token);
        }
        if (*p >= upperBound) {
            *p = -1;
        }
Symbols must be less than 2**%d, StandardRSCode.m);
return 0;
else
  **p;
token = strtok(NULL, " ");
}
*p = -1;
return 1;
}

int IntPowIntInt(int b, int p) {
  int r = 1;
  while (p-- > 0)
    r *= b;
  return r;
}

int main(int argc, char *argv[]) {
  char commandString[512];
  RSSymbol poly1[64], poly2[64], poly3[64], poly4[64];
  int done = 0;
  puts("GF Polynomial Arithmetic");
  puts("Two Polynomials at a Time");
  puts("Kamal Swamidoss");
  puts("December 1997");
  puts("Code Parameters:");
  printf(" m: %d \n", (int) StandardRSCode.m);
  printf(" t: %d \n", (int) StandardRSCode.numberOfCorrectableErrors);
  printf(" K: %d \n", (int) StandardRSCode.numberOfUserDataSymbolsInCodeword);
  printf(" mO: %d \n", (int) StandardRSCode.mO);
  printf(" N: %d \n", (int) StandardRSCode.numberOfSymbolsInCodeword);
  upperBound = (RSSymbol) IntPowIntInt(2, StandardRSCode.m);
  puts(" symbol upperBound: %d, (int) upperBound");
  puts(" Enter Polynomials In Decimal Form");
  puts(" From Lowest-Degree-Coefficient");
  puts(" To Highest-Degree-Coefficient.");
  puts(" Type \"/exit\" to Exit.");

  while (!done) {
    printf("Enter First Polynomial: ");
    fgets(commandString, 511, stdin);
    commandString[strlen(commandString) - 1] = \0;
    if (strcmp(commandString, "/exit") == 0) {
      done = 1;
      continue;
    }

    if (!StrToPoly(commandString, poly1))
      continue;

    printf("Enter Second Polynomial: ");
    fgets(commandString, 511, stdin);
    commandString[strlen(commandString) - 1] = \0;
    if (strcmp(commandString, "/exit") == 0) {
      done = 1;
    }
    continue;
}

if (!StrToPoly(commandString,poly2))
    continue;

    printf("Enter Operation (mdx) :");
fgets(commandString,511,stdin);
commandString[strlen(commandString)-1] = '0';
if (strcmp(commandString,"/exit") == 0) {
    done = 1;
    continue;
} else if (strlen(commandString) != 1) {
    puts("Invalid Operation.");
    continue;
}

switch (*commandString) {
    case 'm':
        GFPolyArithMultiply(poly1,poly2,poly3);
        PrintPoly("Product: ",poly3);
        break;
    case 'd':
        GFPolyArithDivide(poly1,poly2,poly3,poly4);
        PrintPoly("Quotient: ",poly3);
        PrintPoly("Remainder: ",poly4);
        break;
    case 'x':
        GFPolyArithXOR(poly1,poly2,poly3);
        PrintPoly("XOR: ",poly3);
        break;
    default : puts("Invalid Operation.");
        break;
}

    puts("------------------------");
}

return 0;

#endif
Appendix G – Diagnostic Output Functions

These functions can be used by the debugger to display the contents of GF arrays at run-time.

```c
#include <stdio.h>
#include "prmrs.h"
#include "modefile.h"

static int filler=0;

#if defined(EnableConsoleOutput)
extern RSLogSymbol GFLog(RSCode *code, RSSymbol x);
extern RSSymbol GFAntilog(RSCode *code, RSLogSymbol x);

/*
 * Diagnostic Output Functions
 * Kamal Swamidoss
 * November 1997
 */

void MyPrintRSSymbolArray(char *s,RSSymbol *a,int 1) {
    int j;
    printf("%s",s);
    if (1 == 1) {
        printf("%04d\n",a[0]);
        return;
    }
    j = 0;
    while (j < 1-1) {
        printf("%04d, ",a[j++]);
        printf("%04d",GFLog(code,a[j]));
        puts("\n");
    }
}

void MyPrintRSSymbolArrayLog(RSCode *code, char *s,RSSymbol *a,int 1) {
    int j;
    printf("%s",s);
    if (1 == 1) {
        printf("%04d\n",GFLog(code,a[0]));
        return;
    }
    j = 0;
    while (j < 1-1) {
        printf("%04d, ",GFLog(code,a[j++]));
        printf("%04d",GFLog(code,a[j]));
        puts("\n");
    }
}

void MyPrintRSLogSymbolArray(RSCode *code,char *s,RSLogSymbol *a,int 1) {
    int j;
    printf("%s",s);
```
if (l == 1) {
    printf("%04d\n", GFAnitilog(code,a[0]));
    return;
}

j = 0;
while (j < l-1)
    printf("%04d, \n", GFAnitilog(code,a[j++]));
printf("%04d\n", GFAnitilog(code,a[j]));
puts("\n");

void MyPrintRSLogSymbolArrayLog(char *s, RSLogSymbol *a, int l) {
    int j;
    printf("%s\n", s);
    if (l == 1) {
        printf("%04d\n", a[0]);
        return;
    }
    j = 0;
    while (j < l-1)
        printf("%04d, \n", a[j++]);
    printf("%04d\n", a[j]);
    puts("\n");
#endif /* #if defined(EnableConsoleOutput) */
Appendix H – Modifications to RSCode

This section describes the modifications to the RSCode structure. The structure is defined differently based on whether or not the preprocessor value UseMyRSEuclid is defined (see Appendix E – Modefile). See Appendix I for a description of how RSCode is initialized.

```c
/*
 * RSCode
 * a structure containing the information defining an RS code. It also contains pointers to any tables used by the encoding and decoding functions.
 */
typedef
struct {
    /*
     * m
     * the length in bits of a symbol.
     */
    int m;
    /*
     * N
     * the internal length of the codewords in symbols, and the size of the multiplicative group of Galois field elements. This is always equal to 2^m - 1. For shortened codes, some of the symbols of the codeword are implicitly zero and are not passed through the I/O interface of the coder.
     */
    int N;
    /*
     * numberOfCorrectableErrors
     * the number of errors that can be corrected by the code.
     * For an odd minimum distance the minimum distance and number of correctable errors are related by t = (d-1)/2,
     * which is a t error correcting, t error detecting code.
     * For an even minimum distance the relationship is t = (d-2)/2,
     * and this is a t error correcting, t+1 error detecting code.
     */
    int numberOfCorrectableErrors;
    /*
     * m0
     * the power of the first of the consecutive roots of the generator polynomial of the code.
     */
    RSLogSymbol m0;
    /*
     * numberOfUserDataSymbolsInCodeword
     */
    int numberOfUserDataSymbolsInCodeword;
    /*
     * numberOfCheckSymbolsInCodeword
     */
    RSLogSymbol numberOfCheckSymbolsInCodeword;
    /*
     * numberOfSymbolsInCodeword
     */
    int numberOfSymbolsInCodeword;
}```
/*
 * log
 * a table of logarithms for the Galois field used to define
 * the RS code. The logarithm is a function which maps RSSymbols
 * to integers such that multiplication of RSSymbols is equivalent
 * to addition of their logarithms.
 */
RSLogSymbol * log;

/*
 * antilog
 * a table of inverse logarithms for the Galois field used to
 * define the RS code.
 */
RSSymbol * antilog;

/*
 * generatorLogCoefficient
 * the logarithm of the coefficients of the generator polynomial
 * used to define the code. The coefficients are stored as logs
 * to remove some operations from the linear feedback shift
 * register in the encoding process.
 */
RSLogSymbol * generatorLogCoefficient;

/*
 * generatorLogRoot
 * the logarithm of the roots of the coefficients of the generator
 * polynomial used to define the code. The roots are stored as
 * logs to remove some operations from the linear feedback shift
 * registers in the decoding process.
 */
RSLogSymbol * generatorLogRoot;

/*
 * syndromeCalculationParameters
 * parameters to the GFFourier function to perform the syndrome
 * calculation in the decoder.
 */
GFFourierParameters * syndromeCalculationParameters;

/*
 * chienSearchParameters
 * parameters to the GFFourier function to perform the Chien search
 * function in the decoder. The Chien search evaluates the error
 * locator polynomial at every possible error location.
 */
GFFourierParameters * chienSearchParameters;

/*
 * startLogErrorRoot
 * the error location value of the first codeword coefficient
 * tested during the Forney algorithm.
 */
RSLogSymbol startLogErrorRoot;

RSSymbol *syndromePtr;
RSSymbol *logSyndromePtr;
RSSymbol *logErrorLocatorPtr;
RSSymbol *errorLocatorPtr;

If UseMyRSEuclid is not defined, then four pointers are defined. These pointers reference storage
arrays needed by the implementation of Berlekamp's algorithm.
If `UseMyRSEuclid` is defined, then eight pointers are defined. These pointers reference storage arrays needed by the implementation of Euclid's algorithm.

```c
# if !defined(UseMyRSEuclid)
  RSSymbol *previousErrorLocatorPtr;
  RSSymbol *previousLogErrorLocatorPtr;
  RSSymbol *nextErrorLocatorPtr;
  RSSymbol *nextLogErrorLocatorPtr;
  #endif

  RSSymbol *logErrorEvaluatorPtr;
  RSSymbol *logErrorLocatorDerivativePtr;
  RSSymbol *chienSearchResultPtr;

# if defined(UseMyRSEuclid)
  RSSymbol *euclid0;
  RSSymbol *euclid1;
  RSSymbol *euclid2;
  RSSymbol *euclid3;
  RSSymbol *euclid4;
  RSSymbol *euclid5;
  RSSymbol *euclid6;
  RSSymbol *euclid7;
  RSSymbol *euclid8;
  #endif

  } RSCode;
```
Appendix I – Modifications to genrs

This section lists the C code added to the program genrs, used to generate the C code for various RS data structures, including the RSCode structure. The modifications allow various arrays to be conditionally defined. In addition, the modified genrs program allows the RSCode structure to be initialized in a manner consistent with its definition (see Appendix H – Modifications to RSCode).

```c
1  fprintf(dataFile,
2      "\n",
3      "\n",
4      "\n",
5      "#if defined(UseMyRSEuclid)\n",
6      "RSSymbol euclidIndex0[%ld];\n",
7      "RSSymbol euclidIndex1[%ld];\n",
8      "RSSymbol euclidIndex2[%ld];\n",
9      "RSSymbol euclidIndex3[%ld];\n",
10     "RSSymbol euclidIndex4[%ld];\n",
11     "RSSymbol euclidIndex5[%ld];\n",
12     "RSSymbol euclidIndex6[%ld];\n",
13     "RSSymbol euclidIndex7[%ld];\n",
14     "#endif\n",
15     "\n",
16     "RSSymbol mySyndromeArray[%ld];\n",
17     "RSSymbol myLogSyndromeArray[%ld];\n",
18     "RSSymbol myLogErrorLocatorArray[%ld];\n",
19     "RSSymbol myErrorLocatorArray[%ld];\n",
20     "\n",
21     "#if !defined(UseMyRSEuclid)\n",
22     "RSSymbol myPreviousErrorLocatorArray[%ld];\n",
23     "RSSymbol myPreviousLogErrorLocatorArray[%ld];\n",
24     "RSSymbol myNextErrorLocatorArray[%ld];\n",
25     "RSSymbol myNextLogErrorLocatorArray[%ld];\n",
26     "#endif\n",
27     "\n",
28     "RSSymbol myLogErrorEvaluatorArray[%ld];\n",
29     "RSSymbol myLogErrorLocatorDerivativeArray[%ld];\n",
30     "RSSymbol myChienSearchResultArray[%ld];\n",
31     "\n",
32     (long) (2*(t.value)+2),
33     (long) (2*(t.value)+2),
34     (long) (2*(t.value)+2),
35     (long) (2*(t.value)+2),
36     (long) (2*(t.value)+2),
37     (long) (2*(t.value)+2),
38     (long) (2*(t.value)+2),
```

The file modefile.h is included in the modified .c file. The modefile controls the compilation of the RS decoder; it contains several important preprocessor values (see Appendix E – Modefile). If the preprocessor value UseMyRSEuclid is defined in modefile.h, then eight arrays are defined here for use in the implementation of Euclid’s algorithm. These arrays are named euclidIndex0-euclidIndex7.

If UseMyRSEuclid is not defined, then it is implied that Berlekamp’s algorithm is to be used. The implementation of that algorithm requires four additional arrays, which are defined here if UseMyRSEuclid is not defined.
The variables \( t.\ value, K.\ value, N.\ value, m.\ value, \) and \( m0.\ value \) are the RS parameters \( t, K, \)

\( N, m, \) and \( m0. \)

The RSCode data structure has been modified accordingly. If Euclid's algorithm is to be used, then the
RSCode structure contains pointers to the eight previously defined storage arrays. Otherwise, it contains
pointers to the four additional arrays needed by the implementation of Berlekamp's algorithm. Each set
of arrays and the corresponding set of pointers are defined based on UseMyRSEuclid.

\[
\begin{align*}
39 & \quad (\text{long}) \ (2^* (t.\ value)+2), \\
40 & \quad (\text{long}) \ (2^* (t.\ value)+2), \\
41 & \quad (\text{long}) \ t.\ value * 2, \\
42 & \quad (\text{long}) \ t.\ value * 2, \\
43 & \quad (\text{long}) \ t.\ value + 1, \\
44 & \quad (\text{long}) \ t.\ value + 1, \\
45 & \quad (\text{long}) \ t.\ value + 1, \\
46 & \quad (\text{long}) \ t.\ value + 1, \\
47 & \quad (\text{long}) \ t.\ value + 1, \\
48 & \quad (\text{long}) \ t.\ value + 1, \\
49 & \quad (\text{long}) \ t.\ value + 1, \\
50 & \quad (\text{long}) \ K.\ value + 2 ^* t.\ value); \\
51 & \quad (\text{long}) \ t.\ value + 1, \\
52 & \quad (\text{long}) \ t.\ value + 1, \\
53 & \quad (\text{long}) \ t.\ value + 1, \\
54 & \quad (\text{long}) \ K.\ value + 2 ^* t.\ value); \\
\end{align*}
\]
&euclidIndex3[0],
&euclidIndex4[0],
&euclidIndex5[0],
&euclidIndex6[0],
&euclidIndex7[0]

#define
}

name.valueText,
(long) m.value,
(long) N.value,
(long) t.value,
(long) m0.value,
(long) K.value,
(long) t.value * 2,
(long) K.value + t.value * 2,
(long) N.value = 1 - K.value - 2 * t.value

};