Error Correction Coding: Weeks 7-8

BCH Codes

Eric DeFelice
Department of Electrical Engineering
Rochester Institute of Technology
ebd9423@rit.edu

***Abstract* – The BCH codes form a class of cyclic error-correcting codes that are implemented by using many of the techniques and concepts that were introduced previously. These codes are a subset of linear codes and are able to be constructed with given distances, so that a code can be chosen based on the number of errors that should be detected or corrected at the receiver. This is a desirable feature, as many real world systems are designed based on what BER (bit error rates) and data rates are necessary for the applications to be used on that system.**

In this paper, an introduction of BCH codes will be presented, and the properties of these codes will be discussed. Before diving right into BCH codes, a brief review of some of the polynomial arithmetic concepts and linear block coding concepts will be looked at, as they are very important in the construction of BCH codes. A quick conceptual overview of BCH codes will then be looked at, to help show the purpose of this coding scheme and how it is beneficial to a communications system. Finally, the BCH encoder and algebraic decoder will be analyzed, and a couple simple examples will be used to show BCH codes in action.

I. Introduction

BCH codes were invented in 1959 by French mathematician Alexis Hocquenghem, and independently in 1960 by Raj Bose and D. K. Ray-Chaudhuri. The acronym BCH comes from the initials of these inventors' names. BCH codes are multiple error correcting codes and are a generalization of the Hamming codes. A BCH code over the field F=GF(q) of block length *n* and of a distance *d* is a cyclic code generated by a polynomial $g(x) = lcm\{m_i(x): a \leq i \leq a + d - 2\}$. The root set of this polynomial contains *d-1* distinct elements $\beta^{\alpha}, \beta^{\alpha+1}, \dots, \beta^{\alpha+d-2}$, where β is a primitive root of unity and a is an integer. This basically states that the generator for a BCH code is the least common multiple of the minimal polynomials in the given field. This turns out to be the first three odd power minimal polynomials, which will be shown later when the construction of BCH codes is discussed.

Since BCH codes are a generalization of Hamming codes and are a class of cyclic linear block codes, it will be beneficial to review the properties of polynomial arithmetic and linear block codes in general, so that we have a basis to discuss the specifics of the BCH codes. Once this is reviewed, we can begin to go over the concepts behind BCH coding and why this code works the way it does. We will also discuss what makes this particular coding scheme attractive for use in communications systems and how it can successfully correct multiple bit errors. Once the concepts behind the coding scheme are understood, we will introduce the BCH encoding scheme, followed by an algebraic decoding scheme. Finally, an example BCH code will be presented to show all of these concepts in action on a real system.

Although there are a few different BCH decoders, we will focus on an algebraic syndrome decoder because it is the closest to the actual coding structure, and does not have some of the recursive steps, like in the Berlekamp decoder with Chien searching. The recursive decoder does lend itself more to digital implementation, but it is a little more difficult to understand exactly how the coding structure allows for bit error corrections. For this reason, and because this paper is just a learning exercise, we will ignore implementation tradeoffs and just review the easiest decoder to conceptualize.

II. Polynomial Arithmetic & Linear Codes

The BCH code can correct random bit errors in a bitstream by taking chunks of the data (or blocks) and then adding redundant bits, so that the receiver can determine where the errors are located. This coding scheme is suitable to correct random bit errors, but doesn't do as good of a job when there are bursts of errors, where a Reed-Solomon code (for example) may perform better. The BCH code adds the redundant bits by multiplying the block of data by the generator, which is created by finding the least common multiple of the first three odd minimal polynomials in the Galois field of which the code is using. All of these steps require polynomial arithmetic to perform.

Firstly, since we are dealing with binary symbols (strings of ones and zeros) our field is a binary field, or the field Z_2. Using a binary field is also beneficial because it gives us 2^m elements to use as codewords, guaranteeing that we have a power of a prime as our field definition. Now that we have our field under which we will perform the arithmetic, we need to find what types of elements will be codewords in this field. To do this, we must find a primitive polynomial. A primitive polynomial is one where every element in our field can be expressed as some power of the element, p(x). The element must, by definition, be a minimal polynomial in the field. For example, if we look at the field $F(2^4)$ there are two possible choises for the primitive polynomial, $x^4 + x^3 + 1 \; and \; x^4 + x + 1$.

Once a primitive polynomial is chosen, we must now generate the elements in the field. To do this, we start with three initial elements and expand from there. The first three elements are 0, 1, and α. The rest of the elements are created by raising α to successive powers as members of the extension

field. There is one trick that is needed when doing this. Suppose we use $p(x) = x^4 + x^3 + 1$ as our primitive polynomial. When we get to the element α^4, we realize that we can't simply use that as the element, as it is out of our field, so we must reduce it somehow. Since we know $p(x) = x^4 + x^3 + 1 = 0$ we find that $\alpha^4 = \alpha^3 + 1$. This reduction operation is very important when dealing with arithmetic under our field.

Now that we know how to find all of the elements for a given coding field, we can define some useful nomenclature for talking about the elements and performing operations on them. Since we don't always want to write out the full polynomial for each element, we can define them in terms of the coefficients for each term in the polynomial. For example, we can write the element $\alpha^7 = \alpha^2 + \alpha + 1$ as 0111, because the coefficient for the α^3 term is 0, and it is 1 for the other terms. This shorthand is very useful because it also gives us a direct way to convert these polynomial codewords into a bit block, since we will be transmitting bits in the system anyways.

Using this way of representing the codewords, we can look at how to multiply two of the codewords. For this example we will use the code table shown below:

Field of 16 elements generated by $x^4 + x^3 + 1$		
Power Form	n-Tuple Form	Polynomial Form
0	0000	0
1	0001	1
α	0010	α
α^2	0100	α^2
α^3	1000	α^3
α^4	1001	$\alpha^3 + 1$
α^5	1011	$\alpha^3 + \alpha + 1$
α^6	1111	$\alpha^3 + \alpha^2 + \alpha + 1$
α^7	0111	$\alpha^2 + \alpha + 1$
α^8	1110	$\alpha^3 + \alpha^2 + \alpha$
α^9	0101	$\alpha^2 + 1$
α^{10}	1010	$\alpha^3 + \alpha$
α^{11}	1101	$\alpha^3 + \alpha^2 + 1$
α^{12}	0011	$\alpha + 1$
α^{13}	0110	$\alpha^2 + \alpha$
α^{14}	1100	$\alpha^3 + \alpha^2$

If we multiply 1011 by 1101 (or $\alpha^5 \; by \; \alpha^{11}$) we get the following:

$$(\alpha^3 + \alpha + 1)(\alpha^3 + \alpha^2 + 1) = \alpha^6 + \alpha^5 + \alpha^4 + 3\alpha^3 + \alpha^2 + \alpha + 1$$

The product of this operation is obviously not another codeword in this form, so we must reduce it in the same manner that we used before. This time we must perform the reduction for all the elements above α^3. After doing this we get:

$$\alpha^6 + \alpha^5 + \alpha^4 + 3\alpha^3 + \alpha^2 + \alpha + 1 = \alpha = 0010$$

So it is seen that the product is 0010 from this operation. It should also be noted that the operation of subtracting the generator polynomial from the product, when doing the reduction, can be implemented as the exclusive OR operation for implementation, so this arithmetic is easily realized in hardware.

III. Basis For BCH Codes

Now that we have gone over some of the arithmetic that will be used when implementing the BCH code, we can look at the basis for this code. In doing so we will describe how and why the BCH code can correct multiple bit errors and how its design lends well to this purpose.

First, we can look at linear block codes as a whole and determine how they are able to correct bit errors. Let us imagine we have a system where 4 bits are sent over the channel and 1 of those bits happens to be in error. In this scenario, we would not know which bit was in error, as there are no redundant bits giving us any extra information. Now, let us add 1 parity bit to this example. We can now determine if there was a bit error, but would not be able to locate the error.

If we keep adding extra bits to this example, we would gain more and more information as to which bits were in error, if there were any. The downside to adding all these extra bits is that we will be wasting bandwidth in the system. So the main goal is to send as few redundant bits as possible while still be able to correct the bit errors in the received codeword. To visualize this a bit more, we can think of each codeword having a little sphere around it at the receiver. If we receive a set of bits that land in one of the spheres, we then decode the received data to that codeword. With no parity bits, each sphere would only contain the codeword itself. As we add more and more extra bits, we enlarge the size of each of the spheres, which is the same as increasing the *distance* between the codewords.

So from that example, we would only have decoding errors if one of the received codewords has so many errors that it caused the receiver to select the incorrect symbol. This means that there were more than *d* errors in the received symbol. With this example system, we would be able to correct *t* errors in the received symbol, where *t=(d-1)/2*. This is the idea behind BCH codes.

In the BCH coding scheme, the parity bits are chosen in such a way that the receiver is able to determine the location of the errors using syndrome decoding. This means that the symbols contain the original data as well as side information about that data. The receiver can then determine if the received symbol is a valid *syndrome* and if not, it can find the location of the errors. We can now look at the specifics of how the BCH code works.

IV. BCH Code Encoding

With the motivation behind the BCH coding scheme explained, we can begin to look at the nuts and bolts of the codes. BCH codes are a generalization of Hamming codes and have a defined structure. There are a few parameters that define the BCH codes. They are as follows:

Block Length: $n = 2^m - 1$

Information Bits: $k = 2^m - m - 1$

Correctable Errors = t

Parity Check Bits: $n - k \leq mt$

Minimum distance: $d \geq 2t - 1$

The codewords of the BCH code are formed by taking the remainder after dividing a polynomial representing the information bits by the generator polynomial. If we remember from earlier, the generator polynomial is formed by taking the least common multiple of the minimal polynomials of the prime polynomial defined for the particular code. With this definition, it is seen that all of the codewords are multiples of the generator polynomial.

Let us look a little further into the construction of the generator polynomial. As stated above, the generator for BCH codes differs from the generator of Hamming codes, for example, because it is a combination of multiple minimal polynomials. The generator is technically the least common multiple of several powers of a primitive element in the codeword field. Since all of the even powers are not minimal, they can be removed from the least common multiple operation, leaving the odd powers to be combined.

To delve into the generator polynomial construction a bit more, let us look at a simple example. Suppose we have a BCH defined as BCH(31,16). This code has 31 codeword bits, 16 of which are information bits and 15 being check bits. It can correct three errors (t=3), and has a minimum distance between codewords of 7 bits or more. Therefore, we need 2t-1, or 5, minimal polynomials of the first five powers of a primitive element in the field GF(2^5). This field is used because we have 31 codeword bits.

As stated before however, we will actually only need the odd powers' minimal polynomials, as the even powers' polynomials can just be reduced into the odd ones. We define the primitive polynomial for this field as p(x) and the minimal polynomials are $m_1(x)$, $m_2(x)$, … , $m_{2t-1}(x)$. For this example, we will use the primitive polynomial, $p(x) = x^5 + x^2 + 1$. From this polynomial, we can define the minimal polynomials with odd powers as follows:

$$\alpha\text{: } m_1(x) = x^5 + x^2 + 1$$

$$\alpha^3\text{: } m_3(x) = x^5 + x^4 + x^3 + x^2 + 1$$

$$\alpha^5\text{: } m_5(x) = x^5 + x^4 + x^2 + x + 1$$

With these minimal polynomials, we can now find the generator, g(x), by using the formula $g(x) = lcm\big(m_1(x), m_3(x), m_5(x)\big) = m_1(x), m_3(x), m_5(x)$. We just multiply these three polynomials because they are irreducible, hence being minimal polynomials. Performing this multiplication gives us the following:

$$g(x) = \ (\ x^5 + x^2 + 1)(\ x^5 + x^4 + x^3 + x^2 + 1)(\ x^5 + x^4 + x^2 + x + 1) = x^{15} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^5 + x^3 + x^2 + x + 1$$

This generator is very important as it is used for both encoding and decoding. In a practical system, this would be determined beforehand, and would be known by both the encoder and decoder. To encode the symbol and append the check bits, we must first take the current block of information bits (16-bits) and append 15 '0's to the end. This is where the check bits will go. Next, we divide the whole symbol by the generator polynomial using binary arithmetic (which was reviewed earlier). The quotient is not used at all, so it can just be thrown away, which makes the implementation a bit simpler. We are just interested in the remainder, which becomes the 15 check bits.

This type of encoding is called *systematic encoding* because the information bits and check bits are kept separate in the symbol, so they can easily be recognized by the receiver. Nonsystematic encoding scrambles all of the bit positions of the information bits as well as the check bits. This is also called *puncturing*, which is used more on convolutional codes, which will be looked at in subsequent weeks.

So another way of defining the encoding operation is as follows:

$$f(x)x^n = q(x)g(x) + r(x)$$

$$f(x)x^n \textit{ is the information bits shifted by } n \textit{ bits}$$

$$q(x) \textit{is the quotient}, g(x) \textit{ is the generator}, r(x) \textit{ is the remainder}$$

This representation is convenient because we can then express the received symbol in a similar fashion, while including the error vector. This is seen below:

$$f(x)x^n + r(x) + e(x) = y(x)$$

So our received symbol, *y(x)*, is just the transmitted symbol but with an error vector. It is seen from the above equation that the original symbol can be fully recovered if we can determine the values in the error vector. This is the key to BCH codes, as they provide a structure to be able to determine the error vector.

V. BCH Code Decoding

Once the receiver gets the blocks, the next step is to decode and correct any errors. There are a few different BCH decoding algorithms, so of which we won't go into much depth on here. The Peterson-Gorenstein-Zierler (PGZ) decoding algorithm and the Berlekamp-Massey Algorithm (BMA) are both recursive algorithms that are designed to allow efficient implementation of the decoders. The Euclidean method is an algebraic method that is more closely related to the core coding concepts of the BCH code, so we will look into that decoding method. All of these decoding algorithms follow the same basic steps. Those steps are shown below:

1) *Find the syndrome of the received codeword.*
2) *Find the error location polynomial from the set of equations derived from the syndrome.*
3) *Use the error locator polynomial to identify the errors and correct them.*

These basic steps are shared between all of the decoding algorithms and this main process is called *syndrome decoding*.

The first step of this process is to find the syndrome polynomial. This is a fairly simple process. We must take the received symbol and create syndromes from that in following manner:

$$S_1 = r(\alpha)$$
$$S_2 = r(\alpha^2)$$
$$S_3 = r(\alpha^3)$$
$$S_4 = r(\alpha^4)$$
$$S_5 = r(\alpha^5)$$

From these equations we can get the syndrome polynomial, which is defined as:

$$S(x) = \sum_{j=1}^{n-k} S_j x^{j-1}$$

With this syndrome polynomial and our syndromes, we can begin to find the error locator vector. To do this, we set up a system of equations where the syndromes are augmented by the error vectors. If there are no errors, then the error vectors are zero. The system of equations is seen below:

$$\begin{bmatrix} S_1 & S_2 \\ S_2 & S_3 \end{bmatrix} \begin{bmatrix} \epsilon_2 \\ \epsilon_1 \end{bmatrix} = \begin{bmatrix} S_3 \\ S_4 \end{bmatrix}$$

This system will correct up to 2 errors as we only have 2 elements in our error vector. To correct 3 errors, we would need to use all five of the syndromes and use a 3x3 syndrome matrix. After solving for the error vector, we should now have the locations of the errors, but how can we determine where they are in the received symbol.

We can construct the error-locator polynomial using the following equation:

$$S(x) = \epsilon_3 x^3 + \epsilon_2 x^2 + \epsilon x + 1$$

If we take this error-locator polynomial and evaluate it over all α^i, the inverse of the location of the error bits are where this polynomial equals zero. With the location of the error bits in hand, we can now correct the received symbol. The BCH code has done its job and we are able to detect and correct random bit errors in a fairly simple fashion. We can now look at an example to see exactly how the math looks when decoding the BCH code.

VI. BCH Coding Example

In this example, we will look at a symbol that has some bit-errors introduced, and see how to discover where they are located. To do this we will be using the algebraic decoding scheme described in the previous section. First we start with a generator polynomial. This allows us to encode the original data bits. In this example we will use a BCH(15,7,5) code and the generator polynomial is seen below:

$$g(x) = x^8 + x^7 + x^6 + x^4 + 1$$

With this generator, we then encode the data stream, giving us the following codeword (with errors added):

$$c(x) = x^{10} + x^9 + x^6 + x^5 + x + 1$$

There are 2 errors added in the codeword, which we will find using the decoding process. The first step is to find the syndromes so that we can use them to find the error locator vector. The four syndromes are seen below:

$$S_1 = a^{10} + a^9 + a^6 + a^5 + a + 1 = \boldsymbol{a^2}$$
$$S_2 = \boldsymbol{a^4}$$
$$S_3 = a^{30} + a^{27} + a^{18} + a^{15} + a^3 + 1 = \boldsymbol{a^{11}}$$
$$S_4 = \boldsymbol{a^8}$$

With these syndromes we can use the following equation to find the two error locator values, which are the coefficients in the error locator polynomial:

$$\begin{bmatrix} S_1 & S_2 \\ S_2 & S_3 \end{bmatrix} \begin{bmatrix} \epsilon_2 \\ \epsilon_1 \end{bmatrix} = \begin{bmatrix} S_3 \\ S_4 \end{bmatrix} \rightarrow \begin{bmatrix} a^2 & a^4 \\ a^4 & a^{11} \end{bmatrix} \begin{bmatrix} \epsilon_2 \\ \epsilon_1 \end{bmatrix} = \begin{bmatrix} a^{11} \\ a^8 \end{bmatrix}$$
$$\begin{bmatrix} \epsilon_2 \\ \epsilon_1 \end{bmatrix} = \begin{bmatrix} a^{14} \\ a^2 \end{bmatrix}$$

After finding the coefficients, we now have the error locator polynomial. This is shown below:

$$E(x) = a^{14} x^2 + a^2 x + 1$$

We can evaluate this polynomial over all a^i with $i \in \{0 \dots 14\}$. Doing this gives us the two error locations, which are a^5 and a^{11}, or bits 5 and 11 of the received message. Using this information to fix the errors gives us the following received codeword:

$$r(x) = x^{11} + x^{10} + x^9 + x^6 + x + 1$$

VII. Conclusion

In this paper, BCH codes were introduced and their operation was described. A brief review of polynomial arithmetic was looked at, as well as a review of linear block coding concepts. A conceptual overview of BCH codes was then presented, and the encoding and decoding algorithms were analyzed. This was followed by a functional example showing all of these concepts in action. The BCH code is a powerful linear block code capable of correcting random bit errors with minimal redundant bits.

The next couple of weeks will be spent by looking into Reed-Solomon codes, which are similar to BCH codes. The investigation into this specific coding scheme will be more Matlab based, as the theory behind its operation is similar to the block codes already discussed. The purpose of focusing on Matlab examples is to get a better sense of the performance of the Reed-Solomon code, and how it would affect a communications system.

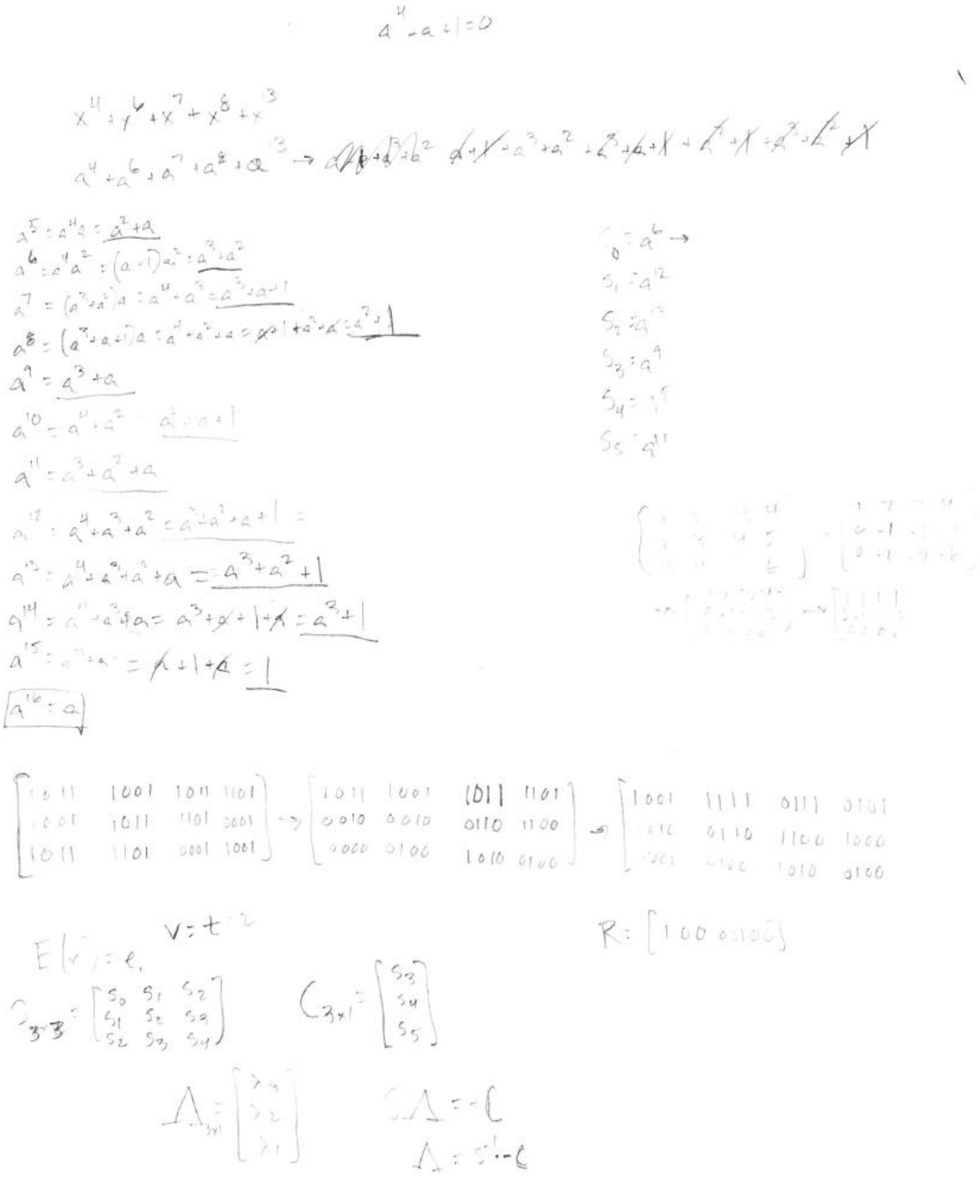
References

[1] San Ling, Chaoping Xing, *Coding Theory: A First Course*. Cambridge University Press, 2004

[2] Andre Neubauer, Jurgen Freudenberger, Volker Kuhn, *Coding Theory: Algorithms, Architectures, and Applications*. John Wiley & Sons Ltd., 2007

[3] Scott A. Vanstone, Paul C. van Oorschot, *An Introduction to Error Correcting Codes with Applications*. Kluwer Academic Publishers., 1989

APPENDIX



$g = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$

$M(x) = x^4 + x^3 + x + 1$

$x^{11} + x^3 + 1$ | $x^{10} + x^8 + x^5 + x^4 + x^2 + 1$

$x^{10} + x^9 + x^6$

$x^9 + x^8 + x^6 + x^5 + x^4 + x^2 + 1$

$x^9 + x^8 + x^5$

$x^6 + x^4 + x^2 + 1$

$x^6 + x^4 + x^3$

$x^3 + x^2 + 1$

$x^3 + x^2 + x$

$x + 1$

$x^{11} = x^2$

$(x^{11} + x^3 + 1)(x^6 + x^5 + x^2 + x)$

$x^{10} + x^8 + x^6 + x^5 + x^3 + x$

$x^{10} + x^8 + x^6 + x^5 + x^3 + x + 1$

x^5

$x^5 = 1$

$(15, 7, 2)$ BCH code $\quad m = (0\,1100\,11) \rightarrow x^6 + x^5 + x^2 + x$

15 bits, 7 data, $d_{min} \geq 2t+1$, so $t = 1$: 1 error correction

$g(x) = m_1(x)\, m_2(x) \rightarrow (x^4 + x + 1)(x^4 + x^3 + x^2 + x + 1)$

$m_1(x) = x^4 + x + 1$

$m_2(x) = x^4 + x^3 + x^2 + x + 1$

$x^3 + x + 1$

$x^8 + x^7 + x^6 + x^4 + x^2 + x + 1$

$= x^8 + x^7 + x^6 + x^4 + 1$

$x^8 \cdot m = x^8 m = x^{14} + x^{13} + x^{10} + x^9$

$r(x) = C \bmod g(x) = x^8 + x^7 + x^6 + x^4 + 1$

$x^{14} + x^{13} + x^{10} + x^9$

$x^{14} + x^{13} + x^{10} + x^9 + x^6$

$x^8 + x^7 + x^6$

$x^{11} + x^{10} + x^9 + x^8 + x^5$

$x^{10} + x^9 + x^7 + x^6 + x^5$

$x^9 + x^7 + x^6 + x^4 + x^3$

$x^8 + x^7 + x^6 + x^4 + 1$

$x^3 + 1$

$\alpha = (0000)$

$1 = 1 \quad (0001)$

$\alpha = \alpha \quad (0010)$

$\alpha^2 = \alpha^2 \quad (0100)$

$\alpha^3 = \alpha^3 \quad (1000)$

$\alpha^4 = \alpha + 1 \quad (0011)$

$\alpha^5 = \alpha^2 + \alpha \quad (0110)$

$\alpha^6 = \alpha^3 + \alpha^2 \quad (1100)$

$\alpha^7 = \alpha^4 + \alpha^3 = \alpha^3 + \alpha + 1 \quad (1011)$

$\alpha^8 = \alpha^4 + \alpha^2 + \alpha = \alpha^2 + 1 \quad (0101)$

$\alpha^9 = \alpha^3 + \alpha \quad (1010)$

$\alpha^{10} = \alpha^4 + \alpha^2 + 1 \quad (0111)$

$\alpha^{11} = \alpha^3 + \alpha^2 + \alpha \quad (1110)$

$\alpha^{12} = \alpha^4 + \alpha^3 + \alpha^2 = \alpha^3 + \alpha^2 + \alpha + 1 \quad (1101)$

$\alpha^{13} = \alpha^4 + \alpha^3 + \alpha = \alpha^3 + 1 \quad (1001)$

$\alpha^{14} = \alpha^4 + \alpha = 1$

parity $= (10010000)$

$C = (1100111\,|\,10010001)$

$R = (1101110\,|\,10010001)$

$R = C + E$

$S_0 = a$; $S_1 = a^2$; $S_2 = a^3$; $S_3 = a^4$, $S_4 = a^5$, $S_5 = a^6$

(0010) (0100) (1000) (0011) (0110) (1100)

BCH(15, 7, 5)

$g(x) = x^8 + x^7 + x^6 + x^4 + 1$

$r(x) = x^{10} + x^9 + x^6 + x^5 + x + 1$

$S_1 = a^{10} + a^9 + a^6 + a^5 + a + 1 = a^2$

$S_2 = a^4$

$S_3 = a^{30} + a^{27} + a^{18} + a^{15} + a^3 + 1 = (a^2+1)(a^3+1)(a^2) + (a^3+1)(a^3+a^2+1) + (a^3+1)(a+1) + (a^3+1)(a) + a^3 + 1$

$= a^9 + a^6 + a^6 + a^3 + a^3 + 1 + a^4 + a^3 + a^3 + a + a^4 + a^3 + a^2 + a$

$= a^8 + a^6 + a^4 + a^3 + a^2 + a + 1$

$= (a^3+a)(a^3+a) + a^3 + a^2 + a + 1$

$= a^6 + a^4 + a^4 + a^2 + a^3 + a + 1 = a^3 + a + a + 1 = a^3 + 1 = a^{14}$

$S_4 = a^8$

$S(x) = a^8 x^3 + a^{11} x^2 + a^4 x + a^2$

$\begin{bmatrix} S_1 & S_2 \\ S_2 & S_3 \end{bmatrix} \begin{bmatrix} \Lambda_2 \\ \Lambda_1 \end{bmatrix} = \begin{bmatrix} S_3 \\ S_4 \end{bmatrix} \rightarrow \begin{bmatrix} a^2 & a^4 \\ a^4 & a^{11} \end{bmatrix} \begin{bmatrix} \Lambda_2 \\ \Lambda_1 \end{bmatrix} = \begin{bmatrix} a^{11} \\ a^8 \end{bmatrix} \rightarrow \begin{matrix} \Lambda_1 = a^2 \\ \Lambda_2 = a^{14} \end{matrix}$

$a^2 \Lambda_2 + a^4 \Lambda_1 = a^{11} \rightarrow a^{16} + a^6 = a + a^3 + a^2 = a^{11}$

$a^4 \Lambda_2 + a^{11} \Lambda_1 = a^8 \rightarrow a^{18} + a^{13} = a^3 + a^3 + a^2 + 1 = a^8$

$\Lambda(x) = a^{14} x^2 + a^2 x + 1$ → evaluate @ a^i for all $i \in \{0 \ldots 14\}$

$r(x) = x^{10} + x^9 + x^6 + x^5 + x + 1$

$= x^2 + x + 1 + x^3 + x + x^3 + x^2 + x + x + 1 + x = x^2$