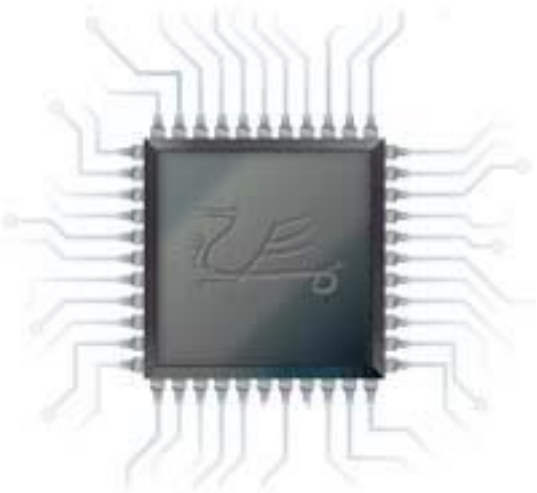
Introduction to Error Detection and Correction



Simon Southwell

September 2022

Preface

This document brings together two articles written in August and September 2022 and published on LinkedIn, that is an introduction to the detection and correction of errors in binary data within a noisy channel. It covers parity, checksums, CRC, and Hamming codes in Part 1,and Reed-Solomon Codes in Part 2.

Simon Southwell
Cambridge, UK
September 2022

Contents

Part 1: Parity to Hamming

Introduction

I first came across error correction codes (ECC), back in the 1990s, whilst working on integrated circuits for Digital Data Storage (DDS), based on digital audio tape (DAT) technology, at Hewlett Packard. I wasn't working on ECC implementation (I was doing, amongst other things, data compression logic) and it seemed like magic to me. However, I had to learn how this worked to some degree as I'd pitched an idea for a test board, using early FPGAs, to emulate the mechanism and channel for the tape drive for testing the controller silicon and embedded software. That is, it could look, from the controller point of view, like a small section of tape than could be written to and read and moved forward and back. To make this useful, the idea was to create a fully encoded set of data, with any arbitrary error types added, which was loaded into a large buffer for the tape data emulation. Therefore, I would have to be able to compile this data, including the encoding of ECC. Fortunately, I had the engineer on hand who was doing the ECC implementation, and he'd usefully written some C models for encoding and decoding which I could adapt for my purposes. This all worked well and was used in earnest for testing hardware and software in all sorts of error scenarios. Indeed, when a colleague from our partner development company was returning home, after secondment to analyse tape errors, we had the idea for his last analysis to be a 'fake' error pattern, generated by the tool, that said 'best wishes' in an error pattern.

Since then, access to information on ECC has become more readily available, with the expansion of the internet (in its infancy when I was working on DDS), but I have found that most information on this subject to be quite mathematical (necessarily when proving the abilities and limits of algorithms), with very little discussion of how this is turned into actual logic gates. In this article I want to assume that the proof of functionality is well established and concentrate on how to get from these mathematical concepts to actual gates that encode and decode data. I want to take this one step at a time so that I can map the mathematical concepts to real-world logic and hopefully dispel the idea that it's too complicated. I will start at the beginning with concepts that many of you will already be familiar with and you may skip these sections if you wish, but it is in these sections I will be introducing the logic that fits the terminology, used in the following sections, as we build to more complex ideas

Once we get to Hamming and Reed-Solomon coding there will be logic implementations of the examples that can be accessed on github, along with accompanying test benches, so that these implementations can be explored as executing logic. The logic is synthesisable, but has no state, which it would need for practical use, so it is 'behavioural' in that sense, but is just logic gates, with no high-level Verilog used. Also, there will be an Excel Spreadsheet for exploring Hamming codes where any data byte value can be input, and errors added to the channel to see what the Hamming code does to correct or detect these.

ECC has been around for some time (even before the 90s when I was introduced to it), so you may wonder how relevant it is to today's design problems. Well, Reed-Solomon codes are used, for example, in QR codes (see diagram below), now ubiquitous in our lives, and the newly launched PCI Express 6.0 specification adds forward error correction to its encoding (see my PCIe article).

So, let's get to it, and the beginning seems like a good place to start.

Basic Error Detection

In this section I want to go through two of the most basic error detection methods that you may already be familiar with—parity and checksums. I want to do this firstly for completeness (some people may be just starting out and not be as familiar with these concepts), but also to introduce some terms and concepts used later.

In all the algorithms we will discuss, though, there is a trade-off between the amount of data we have to add to the basic information (i.e., the amount of redundancy) versus the effectiveness of the encoding for error detection or

correction. The engineering choices made rely on an understanding of the channel that the data is transported through (whether wire, wireless or even magnetic or optical media) and the noise and distortion *likely* to be encountered. None of these methods is 100% and can be defeated so that errors occur. The idea is to make the observed error rate below a certain probability. Often more than one mechanism is employed. This article will not be discussing the analysis of channels and the design choices to make in choosing the ECC scheme, but we will look at the effectiveness of each method.

Parity

This is perhaps the simplest encoding for detecting errors but, as we shall see, it is the foundation of many of the more complex algorithms we will discuss.

Parity, fundamentally, is "adding up ones" to see if there are an odd number of them or an even number of them. For example, taking an 8-bit byte, if there are an even number of ones (say, 4), then it has even parity, else it is has odd parity. We can encode the data into a codeword to ensure the code has either even or odd parity by adding an extra bit and setting that bit as one or zero to make the codeword even or odd parity. We still need to 'add up' the ones from the data to know how to set this bit. This is done using *modulo 2* arithmetic. That is, we add 1s and 0s with no carry, so 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, but 1 + 1 = 0 as it is modulo 2—the equivalent in C is (x + y)%2. Note that in modulo 2 arithmetic, adding and subtracting are the same thing, so 1 – 1 is 0, just like 1 + 1. The is useful to know as I've read articles that mention subtraction and then give example where addition is being used.

You may have noticed that this is the same output as an exclusive OR gate (XOR). Therefore, we can use XOR or XNOR gates to add up the number of ones and either set or clear the added bit for even or odd parity as desired. The Verilog code to calculate the added bit is shown below for odd and even codes:

```
// Parity
wire [7:0] byte;
wire [8:0] ecode, ocode;
assign ecode = { ^byte, byte};
assign ocode = {~^byte, byte};
```

Here we see the even parity code as the byte bits all XORed together, whilst the odd parity code uses the inverse of this. The byte and parity bit code can then be transmitted, over a UART say, and the receiver performs a parity module 2 sum over the whole codeword. If the sum is 0, then no error occurred. If the sum is 1, then error is detected. This is true whether even or odd parity, so long as the same XOR or XNOR is used byte both transmitter and receiver.

This method is very limited in protection. It can't correct any errors and can only detect a single bit in error. A two-bit error will make the code 'valid' once more, and the bad data will be considered good. Therefore, it is suitable for channels where bit errors are rare, and certainly the probability of two-bit errors within the space of a codeword length (9 bits in the example) has a probability below the required rate for the system.

In the example a byte was used, so adding a parity bit has a redundancy of 12.5%. This can be reduced by encoding over a larger data size, so 16 bits has a redundancy 6.25%, but the probability of two-bit errors with the codeword now increases. This is why I said an understanding of the channel is important in making design choices. Increasing to 16-bits might be a valid choice for increasing efficiency if the raw bit error rate is low enough to meet design criteria.

Checksum

A checksum is an improvement to basic parity but is still only a detection algorithm. In this case the data is formed into words (bytes, 16-bit words, 32-bit double words etc.), and then a checksum is done over a block of these words, by adding them all up using normal arithmetic, to form a new word which is appended to the data. The checksum word will be of a fixed number of bits, and so is doing modulo 2^n summation, where n is the number of bits.

The checksum word might be bigger than the word size of data so, for example, 8-bit byte data might be being used, with a 16-bit checksum word. In all cases all the data words are added up as unsigned numbers, modulo the width of the checksum word. I.e., $(D_0 + D_1 + D_2 \ldots + D_{n-1}) \% 2^{chksum\_width}$. This sum is then two's complemented and appended to the data. On reception, the same summation is made, including the appended word and if zero, the checksum detected no errors, else an error occurred. Alternatively, the checksum is appended unmodified, and then subtracted from the data summation at the receiver to give 0 for no error—it depends where it is most convenient to do the two's complement. The diagram

below shows some code for generating a checksum, with the data buffer assumed loaded with data.

```
// Check sum (behavioural)
reg  [7:0] data [0:255];
reg [15:0] chksum = 16'h0000;
integer    i;

always (posedge clk)
begin
  for(i = 0; i < 256; i = i+1)
    chksum = chksum + data[i];
end
wire [15:0] append = -chksum;
```

Like for parity, the checksum adds redundancy, and the trade of is between size of the checksum word, the length of the data block and the probability of errors. In general, checksums perform more efficiently than parity. It comes down to the probability of corrupted data generating the same checksum value as good data. If the checksum word is narrow, say a byte, then the number of messages that can generate a 'valid' checksum is 1 in 256. This goes up as the word gets wider but drops again as the data block size is increased.

Another failing of checksums is there is no position information in the encoding. In the example above, all the data in the buffer could be re-arranged in all possible combinations and all give a valid checksum, though only one is correct. This can be somewhat alleviated using cyclic redundancy checks, which we'll deal with in the next section.

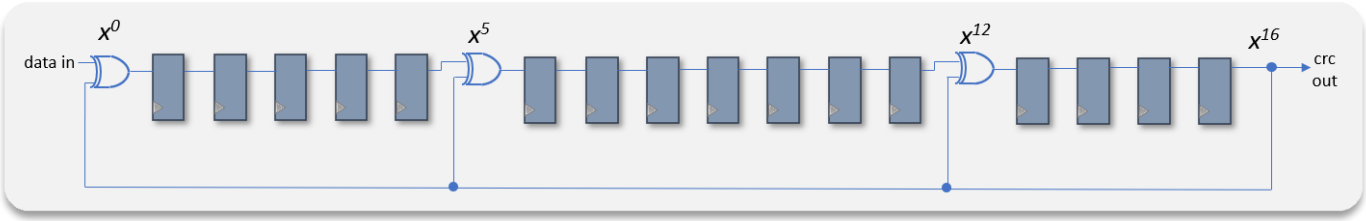
Before leaving checksums, though, I want to briefly mention MD5, which is sometimes referred to as a checksum. Actually, it is a kind of hash function using cryptographic techniques. Its original intention was for it to be infeasible to have two distinct data patterns (of the same size—512-bit blocks for MD5) that produce the same MD5 value (128 bits), which would give some indication that the data received, matching a given MD5 number, was the message intended to be sent. However, this has long since been shown not to be true for MD5, so it is not really

used for this purpose anymore. It is still seen in many places though, as it does give high confidence in detecting data being corrupted by channel noise (as opposed to malicious interference) and, in this sense, is used as a checksum. As it's not a really a checksum, I won't discuss this further here.

Cyclic Redundancy Checks

For cyclic redundancy checks we need to introduce the concept of a linear feedback shift register, or twisted ring counter. This is like a normal shift register, but the output is feedback to various stages and combined with the shifting data using an XOR gate. The useful property of this arrangement is that, if the feedback pints are chosen carefully, and the shift registers are initialised to a non-zero value and then clocked, the shift registers will the go through all possible values except 0, given the length of the shift register (i.e., modulo *n*, with *n* the number of bits in the shift register), without repeating, until it reaches the start point value again. Depending on the feedback points, the values at each clock cycle will follow a pseudo random pattern and, indeed, these types of shift register are used for pseudo-random number generation, with the start value being the 'seed'.

The set of numbers and their pattern are known as a Galois filed (after Évariste Galois, and interesting character worth looking up), or sometimes a finite field. These can (and often are) described using a polynomial. For example, if we have a 16-bit shift register and XOR the output into the shift registers bit inputs at 0, 5 and 12, with the actual output at 16, then this can be described with the polynomial $x^{16} + x^{12} + x^5 + x^0$ (note that x^0 is 1, and you will see this written as 1 quite often). By default, this will cycle through its Galois field, but if we add a data input, then this modifies the shift values dependent on that data. The diagram below shows this arrangement of the polynomial with a data input:



If we have a block of data and shift each data bit into the shift register at each clock, then the shift register will have a value which, mathematically, is the remainder when dividing the data by the polynomial (using modulo 2 arithmetic). By appending this remainder (the CRC code) to the data at transmission, then at a

reception, pushing the data, including the CRC codeword, through the same LFSR, with the same starting 'seed', should return 0—i.e., there is no remainder. If there is, then the data has been corrupted.

The polynomials used are often standardised. The example polynomial is the CRC-CCITT 16 bit standard, but $x^{16} + x^{15} + x^2 + 1$ is the CRC-16 standard. Larger width CRCs are used for better protection, such as 32-bit:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^4 + x^2 + x + 1$$

This polynomial is used in, amongst other things, Ethernet, PKZIP, and PNG. The advantage of the CRC over a checksum, even though it might be the same width, is that the CRC codeword is a function of the data *and* the order in which the data was input.

The arrangement shown in the above diagram, works well if shifting data in serial form, but often data is being processed as words (bytes, 16-bit words, double words etc.). Serialising this to calculate the CRC codeword is not practical without sacrificing bandwidth. What is required is a way to calculate the CRC in a single cycle as if all n bits of the data had been shifted through the LSFR. For a given set of data bits, from a given CRC starting point, the remainder will be fixed. Therefore, we can generate a table of all possible remainders for dividing the polynomial by the data. For 8-bit data, this is 256 remainders. The data can be pre-calculated and placed in a lookup table (LUT). The C code fragment below generates the LUT for 8-bit data for the 32-bit polynomial shown above:

```c
#include <stdio.h>
#include <stdint.h>

#define POLYNOMIAL 0x04c11db7

uint32_t LUT[256];

void calc_table(void)
{
    uint32_t rem;

    for (int byte_val = 0; byte_val < 256; byte_val++)
    {
        rem = byte_val;

        for (uint8_t bit = 0; bit < 8 ; bit++)
        {
            if (rem & 1)
                rem = (rem >> 1) ^ POLYNOMIAL;
            else
                rem = (rem >> 1);
        }

        LUT[byte_val] = rem;
    }
}
```

Note that the set bits in the POLYNOMIAL definition correspond to the polynomial terms, except for an implied x^{32}. To calculate the CRC codeword using the LUT, a 32-bit CRC state is kept and initialised (usually to `0xffffffff`, the inverse of a remainder of 0). The block of data bytes are run through the CRC, with each byte XORed with the CRC bottom 8 bits to index into the LUT. This LUT value is XORed with the CRC state shifted down by a byte, which becomes the new CRC value. When all the bytes have been processed like this, the CRC codeword is append as the inverse of the CRC state.

Note that data wider than a byte can be processed in this way but the LUT length, for the arrangement above, is a function of $2^{\text{DATA\_WIDTH}}$, so 16-bit data needs a LUT of 65536 entries and 32-bit data a LUT of length of 4294967296 entries. It can be pipelined, with intermediate CRC values being fed forward and multiple ports into the smaller LUT, or multiple copies of the LUT for each stage, at the expense of more gates and other resources. These, though, are optimisations for speed, but do not change the result of the basic CRC shift register.

It should be noted that for a given CRC polynomial, with the protection properties that it brings, has the same protection if inverted or reversed. For example, the 32-bit polynomial is represented as a value `0x04c11db7` in the LUT generation code. If this 32-bit value is inverted (`0xfb3ee248`) or bit reversed (`0xedb88320`), or both (`0x12477cdf`), then the characteristics remain the same, even if the generated values are different and often this 32-bit polynomial is seen in these alternative forms.

Error Correction

All of the above methods are for error detection only. In this section, and the next article, I want to start talking about error correction and look at a couple of methods that build on what we have looked at so far. These methods are known as forward error correction codes. That is, information is passed forward to allow the receiver to recover the data. This is in contracts with, say, a retry mechanism, where errors are detected, and a message passed back to the transmitter (a NAK, for example) to resend the data. We will look first at Hamming codes and then, in the next article, Reed-Solomon codes.

Hamming Codes

Hamming codes build upon the concept of parity and by careful placement of multiple parity can be used to *locate* the position of an error, and thus correct it. Since we are using ones and zeros, knowing the position of an error means we can correct it by simply inverting the bit. The basic principle is that data is encoded into codewords, which are a sub-set of a larger set of values, such that valid codewords do not become other valid codewords in the presence of n bits in error. The n value is known as the Hamming distance.

For example, a 4-bit code with valid codes being 0000b, 0011b, 0101b, 0110b, 1001b, 1111b, would have a Hamming distance of 2 as it would take 2 bits in error to move from one valid code to the next. All eleven other codes are invalid, and thus this example is only 37.5% efficient. With more distance these codes can be used to correct errors by moving valid codes back to the nearest valid code.

Let's look at a more practical example of Hamming code usage. To keep the example manageable, we will encode 8-bit bytes into code words for the ability to correct a single bit in error. The addition of another parity bit will allow for the detection of a second bit in error. The Hamming code will need four bits, and with a

fifth parity bits, the codeword will be 13 bits in total, giving a 62.5% redundancy. This is not so good, but after going through the example we will see how this might be scaled for encoding larger words with better efficiency. Below is a view of a spreadsheet that encodes an 8-bit byte into a 12-bit hamming code and then adds a detection parity bit, and then allows the code to be decoded, through a channel that can add errors. This spreadsheet is available on [github](), along with the Hamming and Reed-Solomon Verilog source code, so this can be experimented with.

		D0	D1	D2	D3	D4	D5	D6	D7						
	Data	0	1	0	0	1	1	1	1						
		1	2	3	4	5	6	7	8	9	10	11	12	P	
		P1	P2	D0	P4	D1	D2	D3	P8	D4	D5	D6	D7		
	D			0		1	0	0		1	1	1	1		
	P1	1		x		x		x		x		x			
	P2		0	x			x	x			x	x			
even parity	P4				0	x	x	x					x		
	P8								0	x	x	x	x		
	code	1	0	0	0	1	0	0	0	1	1	1	1	0	code parity (even)
	error	0	0	0	0	0	0	0	0	0	0	0	0	0	
	rx code	1	0	0	0	1	0	0	0	1	1	1	1	0	
		x		x		x		x		x		x			0 e0
			x	x			x	x			x	x			0 e1
					x	x	x	x					x		0 e2
									x	x	x	x	x		0 e3
															0 error index
	corrected	1	0	0	0	1	0	0	0	1	1	1	1	0	0

From the top, the data is defined as a set of 8 bits that can be set to any byte value. To construct the Hamming codeword we need to calculate 4 different parity values (even parity in this case) that calculate parity over different bits of the data. The codeword bits are numbered 1 to 12, with a P detection parity bit (more on this later). The parity bits P1, P2, P3, and P4, are positioned at codeword locations 1, 2, 4 and 8. These are the power of 2 locations. The rest of the 12 bits are filled with the data byte bits. It actually doesn't matter where the data bits are placed in the remaining spaces, so long as they are extracted in the same manner on decode but placing them in order seems like a good idea.

So, the line marked D has the values of the data byte, and the parity bits are calculated on the next four lines. P1 is calculated for even parity across D0, D1, D3, D4, and D6, as marked by the cells with a x. The other four parity values are calculated in a similar manner but choosing different data bits for inclusion, as shown. We will discuss the choice of data bit inclusion shortly.

The code line, then, contains the final codeword. Over this whole codeword the detection parity bit is calculated for even parity. The spreadsheet then has an error 'channel' where errors can be added (none in the example above), before getting the received codeword (marked rx code). To decode this, the parity is calculated for the bit positions as during encoding, giving four results marked e0 to e3 on the spreadsheet. When all zeros, as in the above example, there is no error, and the data bits can be extracted from the codeword as good.

Notice the pattern of x positions on the e0, to e3 decoding (and I left this until now, as it is easer to see, I think). Looking down the cells the x positions (if representing 1 with spaces as 0) encode values 1, 2, 3, and so on, in binary, for the 12 positions of the codeword. This is how location of an error will be detected. Let's introduce a single bit error.

		D0	D1	D2	D3	D4	D5	D6	D7							
	Data	0	1	0	0	1	1	1	1							
		1	2	3	4	5	6	7	8	9	10	11	12	P		
		P1	P2	D0	P4	D1	D2	D3	P8	D4	D5	D6	D7			
	D			0		1	0	0		1	1	1	1			
	P1	1		x		x		x		x		x				
even parity	P2		0	x			x	x			x	x				
	P4				0	x	x	x					x			
	P8								0	x	x	x	x			
	code	1	0	0	0	1	0	0	0	1	1	1	1	0	code parity (even)	
	error	0	0	0	0	1	0	0	0	0	0	0	0	0		
	rx code	1	0	0	0	0	0	0	0	1	1	1	1	0		
		x		x		x		x		x		x			1	e0
			x	x			x	x			x	x			0	e1
					x	x	x	x					x		1	e2
									x	x	x	x	x		0	e3
															5	error index
	corrected	1	0	0	0	1	0	0	0	1	1	1	1	0	0	

In the above example D1 is corrupted at position 5, and on decode e0 and e2 are set, giving a value of 5. This is the index of the corrupted bit, so now the bit can be inverted and the corrected code at the bottom is the same as that transmitted. This works for any position, including for the parity bits. So, by careful encoding and positioning of the parity bits, the codeword can move an invalid codeword back to a valid codeword in the presence of a single bit error. What if we have two errors?

		D0	D1	D2	D3	D4	D5	D6	D7							
	Data	0	1	0	0	1	1	1	1							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	P		
		P1	P2	D0	P4	D1	D2	D3	P8	D4	D5	D6	D7			
	D			0		1	0	0		1	1	1	1			
	P1	**1**		x		x		x		x		x				
even parity	P2		**0**	x			x	x			x	x				
	P4				**0**	x	x	x					x			
	P8								**0**	x	x	x	x			
	code	1	0	0	0	1	0	0	0	1	1	1	1	0	code parity (even)	
	error	0	0	0	0	1	0	1	0	0	0	0	0	0		
	rx code	1	0	0	0	0	0	1	0	1	1	1	1	0		
		x		x		x		x		x		x			0	e0
			x	x			x	x			x	x			1	e1
					x	x	x	x					x		0	e2
									x	x	x	x	x		0	e3
															2	error index
	corrected	1	1	0	0	0	0	1	0	1	1	1	1	0	1	

In the above example an additional error is introduced at position 7. The error bits now index position 2, which has no error, but it is inverted anyway as there is no indication that this isn't correct at this point. The corrected codeword is now wrong, but the parity is now odd, which is incorrect, and so the double error is detected. It can't be corrected but at least its presence is known. There is another failure mode where two errors decode to give an error index that is greater than 12—i.e., it indexes off the end of the codeword. This is also detecting a double bit error. I won't show a spreadsheet example here but get hold of the spreadsheet and try for yourself.

So, what does the Verilog look like for this? An encoder Verilog module is shown below.

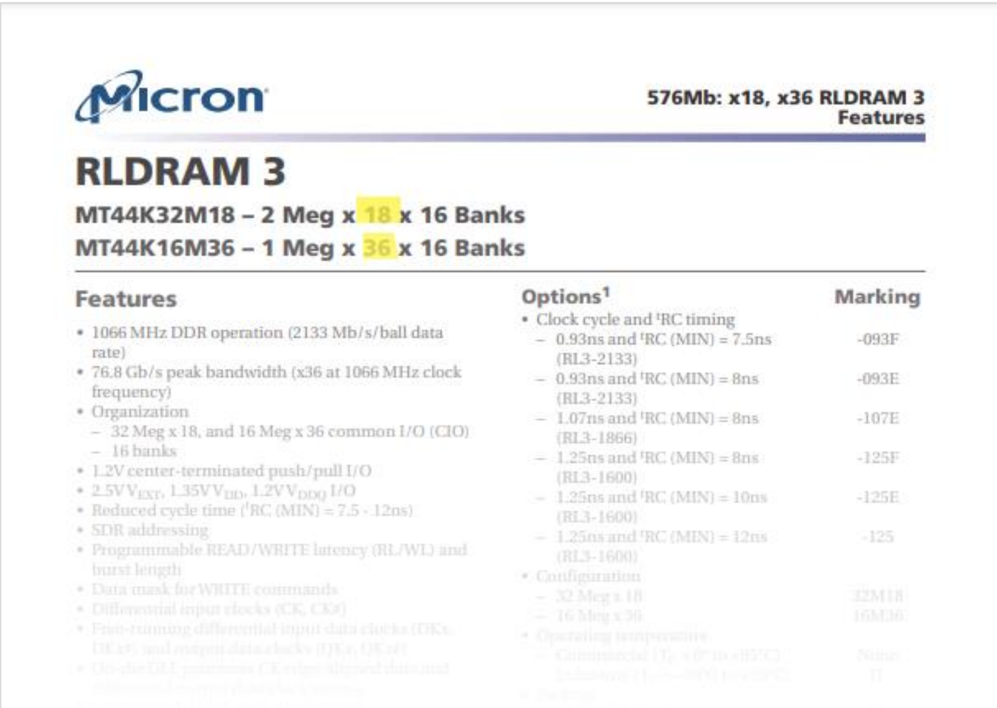
```
// ------------------------------
// Encoder
// ------------------------------
module hamming8_encoder(
  input      [7:0]      data,
  output    [12:0]      code);

wire p1               = data[0] ^ data[1] ^ data[3] ^ data[4] ^ data[6];
wire p2               = data[0] ^ data[2] ^ data[3] ^ data[5] ^ data[6];
wire p4               = data[1] ^ data[2] ^ data[3] ^ data[7];
wire p8               = data[4] ^ data[5] ^ data[6] ^ data[7];

wire [11:0] hamm_code = {data[7], data[6], data[5], data[4], p8, data[3],
                         data[2], data[1], p4,      data[0], p2, p1};

wire parity           = ^hamm_code;

assign code           = {parity, hamm_code};

endmodule
```

This code is purely combinatorial (one could register the output if required) and simply calculates the four parity bits from the relevant input data positions. The 12-bit hamming code is then constructed, placing the parity and data in the correct locations, and then the detection parity bit is calculated over the entire hamming code. The output is then the parity bit and hamming code concatenated. The diagram below shows the decoder.

```
// ------------------------------
// Decoder
// ------------------------------

module hamming8_decoder (
  input     [12:0]      code,
  output     [7:0]      data,
  output                error);

wire [3:0] e;

assign      e[0]        = code[0] ^ code[2] ^ code[4] ^ code[6]  ^ code[8] ^ code[10];
assign      e[1]        = code[1] ^ code[2] ^ code[5] ^ code[6]  ^ code[9] ^ code[10];
assign      e[2]        = code[3] ^ code[4] ^ code[5] ^ code[6]  ^ code[11];
assign      e[3]        = code[7] ^ code[8] ^ code[9] ^ code[10] ^ code[11];

wire [15:0] flip_mask   = 16'h0001 << e;

wire [11:0] corr_code   = code[11:0] ^ flip_mask[12:1];

wire        corr_parity = ^{code[12], corr_code};

assign data             = {corr_code[11], corr_code[10], corr_code[9], corr_code[8],
                           corr_code[6],  corr_code[5],  corr_code[4], corr_code[2]};

// Error if parity on a corrected code fails, or index into codeword invalid (i.e. > 12)
assign error            = (|e & corr_parity) | (|flip_mask[15:13]);

endmodule
```

In this code, the error bits are calculated from the relevant input code positions. From this error index a ‘flip mask’ is generated by shifting a 1 by the index. This is a 16-bit number to cover all possible value for e. A corrected code is then calculated by XORing it with the flip_code, bits 12 down to 1. Bit 0 isn’t used, as an error code of 0 is the no-error case. If the index is greater than 12, then this is an invalid code, which we will deal with shortly. The corrected code plus parity bit is then used to generate the parity for the entire message. The output data is extracted from the relevant positions within the code, before an error status calculated as either a parity error (when error index non-zero), or an invalid index (flip_mask bit above 12 set). And that’s all there is to it. This code is available on github, along with a simple test bench to run through all possible data and one and two bit errors.

It was stated before that encoding a byte this way is not too efficient. In more practical usage, such as protecting RAM data, 64-bit data can be encoded with 7 bits of hamming parity, and an 8^{th} detection parity bit for 72 bits in total. Hopefully it can be seen from the index pattern on the spreadsheet how this may be expanded for 64-bit data by simply following the indexing in binary, with the parity bits at the power of 2 locations. Many RAM components come in multiple of 9, 18 or 36 bits just so 72-bit words (or multiples thereof) can be constructed. For example, the Micron 576Mb RLDRAM 3 comes in 18- and 36-bit wide versions.


576Mb: x18, x36 RLDRAM 3
Features

RLDRAM 3

MT44K32M18 – 2 Meg x 18 x 16 Banks
MT44K16M36 – 1 Meg x 36 x 16 Banks

Features

- 1066 MHz DDR operation (2133 Mb/s/ball data rate)
- 76.8 Gb/s peak bandwidth (x36 at 1066 MHz clock frequency)
- Organization
  - 32 Meg x 18, and 16 Meg x 36 common I/O (CIO)
  - 16 banks
- 1.2V center-terminated push/pull I/O
- 2.5V V_{EXT}, 1.35V V_{DD}, 1.2V V_{DDQ} I/O
- Reduced cycle time (^tRC (MIN) = 7.5 - 12ns)
- SDR addressing
- Programmable READ/WRITE latency (RL/WL) and burst length

Options[1] — Marking

- Clock cycle and ^tRC timing
  - 0.93ns and ^tRC (MIN) = 7.5ns (RL3-2133) — -093F
  - 0.93ns and ^tRC (MIN) = 8ns (RL3-2133) — -093E
  - 1.07ns and ^tRC (MIN) = 8ns (RL3-1866) — -107E
  - 1.25ns and ^tRC (MIN) = 8ns (RL3-1600) — -125F
  - 1.25ns and ^tRC (MIN) = 10ns (RL3-1600) — -125E
  - 1.25ns and ^tRC (MIN) = 12ns (RL3-1600) — -125

I have implemented Hamming codecs for DDR DRAM in just this manner for memory systems in high performance computers, where error rates over the system skyrocket as the amount of RAM used becomes so much larger, with thousands of compute nodes.

Conclusions

In this article we have gone from simple parity for low level error detection, to a more useful way of employing multiple parity for error correction, with a detection fallback. I have tried to simplify the mathematical concepts into practical logic-based implementations, with example logic that can be used in real synthesisable modules. Checksums give a simple method for improving detection over parity but have no position protection. CRCs introduce the concept of polynomials and Galois fields and then reduce this to practical logic, with methods to improve encoding efficiency. For the Hamming code, a spreadsheet implementation is used for exploring the codes more immediately and visually and revealing the pattern of the encoding process to allow indexing to error locations giving the means for correction. (Verilog and spreadsheet available on github.) Each of the methods discussed trade complexity against expected channel error rates and nature, and the appropriate method depends on the design requirements.

In the next article will be explored Reed-Solomon codes, which takes parity even further, also using the Galois fields we saw for CRC, to be tolerant of burst errors as well as random (Gaussian) errors.

Part 2: Reed-Solomon

Introduction

In the first part of this document we looked at error detection, starting from parity before moving through checksums and the CRCs. This introduced the idea of modulo 2 arithmetic which maps to XOR gates, and linear feedback shift registers to implement polynomials and generate Galois fields (basically pseudo-random number sequencers). In this final article I want to cover the concepts of Reed-Solomon codes and we'll bring together the concepts of the previous article to do so. The goal though, like for most of my articles, is to get to actual synthesisable gate implementations to map the concepts to practical solutions.

We will work through a small example, so that large, unwieldy amounts of data don't obscure what's going on, and there is Verilog code, discussed in the text, implementing an encoder and decoder, along with a test bench to exercise it so that those interested can experiment. This code can be found, along with the first part's Hamming code Verilog, on github here.

Reed-Solomon codes are a form of block code. That is, they work on a fixed size block of data made up of words, in contrast to Hamming codes that work on a single multi-bit word to make up a code word. The advantage of this is that they can correct whole words so that burst errors, where consecutive data bits are corrupted for a short period, can be corrected. This might occur on, say, magnetic or optical media where there is dirt or a scratch. There may still be random noise on the channel and other schemes may need to be employed to counter these affects. Reed-Solomon codes are often used with interleaving data, and with two stage processing, as we'll see.

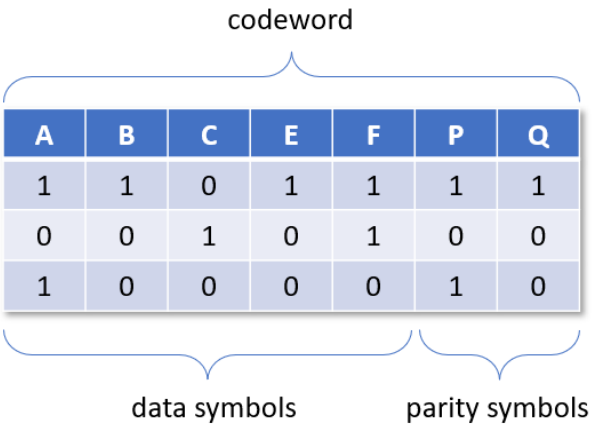
We will now look in detail of the small example. This is adapted from John Watkinson's *The Art of Digital Audio* book, with some modifications and, more importantly for this article, taken to a logic implementation, which is not done in the book. I was lucky enough to have had John Watkinson teach me some of these concepts when at Hewlett Packard (see the introduction of Part 1), as he was hired by them to teach us Digital Audio Tape concepts, which we were about to adapt into Digital Data Storage. I have had his book since then and still refer to it often. So, let's get started.

Reed-Solomon Codes

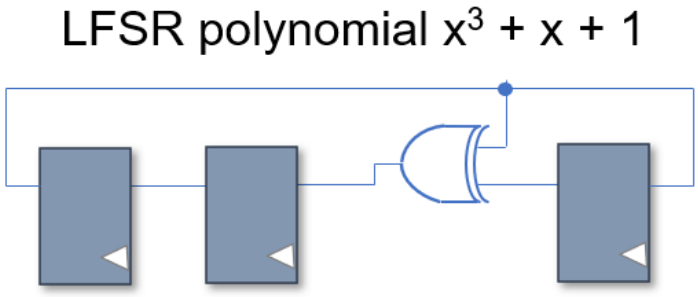
For local reference I will redefine some terms here that we will need for generating Reed-Solomon codes, along with some new ones.

- Galois Field: a pseudo-random table generated from an LFSR defined as a polynomial (as per CRC in Part 1 of the document). The table of Galois field values will be used as powers (i.e., the logarithms) of the primitive element *a* (defined below).
- The primitive element *a*, which is a constant value. In our case this is simply 2, and we will stick to modulo 2 arithmetic. This primitive element will effectively be the 'seed' to the pseudo-random number generator (PRNG), made from the LFSR.
- P and Q: the parity symbols added to the message symbols. This is the redundant information for allowing error correction. In Reed-Solomon codes these are 'appended' to the data block, rather than mixed in with the message like Hamming codes. P is used to determine the error pattern and is called the corrector, and Q is used to determine the location of the symbol in error and is called the locator.
- Generator Polynomials: polynomials calculated from a set procedure. It is these polynomials that are used to generate the parity symbols P and Q. In the example this is done by solving simultaneous equations.

In the example we'll look at we will have data symbols of 3 bits. In a practical solution this might be 8-bit bytes, but we will reduce this to 3 for clarity. The example will have 5 of these data symbols (A to E) and then the two parity symbols, (P and Q) which we will calculated using the predetermined generator polynomials (more later). The diagram below shows the example codeword.



The symbols in the example are 3 bits, so there are 8 possible values, but a value of 0, like for the Hamming codes, is the error free case and so we have a seven symbol block, allowing 5 data symbols and the two parity symbols. To generate P and Q from the data, the Galois field must be defined and each of the powers of the primitive element *a* mapped to this field. The diagram below shows the polynomial for this example, the logic used to generate it, and a table of the output values if seeded with the value of *a* (i.e., 2).



a	0	1	0
a^2	1	0	0
a^3	0	1	1
a^4	1	1	0
a^5	1	1	1
a^6	1	0	1
a^7	0	0	1

The value of *a* is 2, and we will map each subsequent step as successive powers of *a*, as shown in the table. Thus, each power of *a* has its corresponding value, as shown. That is, a^5 is 111b. These, then, are the values of *a* raised to successive powers, and the indexes into the table (starting from 1) are the logs of *a*. It is by this means, as we shall see, that we can bypass multiplications by finding the logs and doing addition.

We can find the values of P and Q for the particular data by using the following polynomials, given here but calculated from generator polynomials and solving simultaneous equations:

$$P = a^6A \oplus aB \oplus a^2C \oplus a^5D \oplus a^3E$$
$$Q = a^2A \oplus a^3B \oplus a^6C \oplus a^4D \oplus aE$$

The equations involve multiplying powers of *a* with the data but, by using the logs this becomes much simpler. For example, if a data symbol is 100b and needs to be multiplied by *a* cubed then:

$$100b = a^2 \text{ so } a^2 \times a^3 = a^{2+3} = a^5 = 111b$$

So, this multiplication is done by adding the logs. Note that, if the addition falls off the end of the table (i.e., is > 7) it wraps back to the beginning of the table so a^{11} maps to a^4. This isn't quite modulo8, as that would include the value 0, which is not part of the Galois field. It is really modulo8 + 1.

It is possible to generate simple logic circuits, using XOR gates, for multiplying data symbols by each of the powers of *a*, effectively stepping through the table the number of times to the value of the data, with the required multiplying power of *a* as the starting point in the table. This can also be done with look up tables for generating logs and antilogs and that is what we'll do in the implementation.

So now we can calculate the P and Q using the polynomials as given above. The diagram below shows this worked through:

calculating parity symbols

A	101	a^6A	111	a^2A	010
B	100	aB	011	a^3B	111
C	010	a^2C	011	a^6C	001
D	100	a^5D	001	a^4D	101
E	111	a^3E	010	aE	101
P	100	⇐	100		
Q	100	⇐	⇐	⇐	100

It happens, in this example, the P and Q values are the same, but this is a coincidence. Column 2 is the data values, with column 3 the terms for the generator polynomial for P and the fifth column that for Q. The results (columns 4 and 6) are worked through, just as for the example from before. So, C is 010b which is *a* multiplied by a^2, to give 1 + 2 = 3. So, a^3 is 011b and that's the result shown. You can go through the rest at your leisure. Having done this for data symbols A through E, the parity is then just all five values XORed together, which is where the P and Q values come from. This is the block that is then transmitted.

On reception we need to generate two syndromes S_0 and S_1. The first is just the received symbols, including the parity symbols, XORed together. The second is the received symbols multiplied by powers of a (using the log trick) and then XORed. This step of generating syndromes is effectively multiplying the received symbols with a matrix, albeit that one multiplication row (that of generating S_0) is multiplying by 1. This matrix is known as the check matrix, and the check matrix for our example is as shown below, in a form that would be the usual way of specifying this.

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ a^7 & a^6 & a^5 & a^4 & a^3 & a^2 & a \end{bmatrix}$$

Other larger Reed-Solomon codes might have more rows in such a matrix and also slightly vary in form, for instance starting at a to the power 0 (i.e., 1)—we will look at an example of a real-world specification later.

The diagram below shows the syndrome calculation where no errors have been introduced.

calculating syndromes (no errors)

A	101	a^7A	101
B	100	a^6B	010
C	010	a^5C	101
D	100	a^4D	101
E	111	a^3E	010
P	100	a^2P	110
Q	100	aQ	011
S_0	**000**	**S_1**	**000**

Since there were no errors, both the syndromes come to 0 and the message is good. Note that the table shows the power of a multiplications going from power of 7 down to 1. This could be the other way around, but this would also affect the P and Q polynomial equations, so it is about matching between these two things.

Let's now introduce an error into the message. We will corrupt symbol C, which is then C'. In order to locate which symbol is bad we need to know the power of a that the bad symbol was multiplied by. This, as yet unknown, power is k. The way things

have been set up we can find a to the power k by dividing S_1 by S_0. This is because the error will have been XORed with power of a at the corrupted location (i.e., k) to give S1, so dividing both sides by S_0 gives $a^k = S_1/S_0$.

We can use logs again, where division becomes subtraction. The diagram below shows the case where C has been corrupted (to 110b).

calculating syndromes (with errors)

A	101	a^7A	101	7
B	100	a^6B	010	6
C'	**110**	**a^5C'**	**100**	**5**
D	100	a^4D	101	4
E	111	a^3E	010	3
P	100	a^2P	110	2
Q	100	aQ	011	1
S_0	**100**	**S_1**	**001**	

$$a^k = \frac{S_1}{S_0} = \frac{1}{4} = \frac{a^7}{a^2} = a^{7-2} = a^5$$

From the example, C is now C' and is 110b. C' multiplied by a^5 yields 100b. The two syndromes are now 100b and 001b. So, 1 divided by 4 is antilog(7) divided by antilog(2), or 7 – 2 = 5. Thus, k is 5 and indicates the location of the error— at symbol C.

Now we have to correct the located symbol. Since S_0 is 000b when there are no errors, it must be the value of the corruption when there are errors, so C' xor S_0 gives C, the correct symbol value, or 110b xor 100b = 010b.

An important point to note here is that it doesn't matter what the corruption of the symbol was, with 1, 2, or 3 bits in error— S_0 would be the corruption value and the multi-bit symbol can still be corrected. This is where the burst error correction capabilities come from, and is also important for using other methods without locator parity symbols (more later).

Implementation

In this section we will examine and encoder and decoder Verilog implementation for the example. Necessarily, this will be fragments of code to highlight the operation, but the entire code is available on github, along with a test bench.

Encoder

The encoder's main purpose is to calculate the locator and corrector value P and Q. It uses tables of pre-calculated values for the Galois field and inverse Galois field. That is, it will have an antilog and log table for the primitive value *a*. In the code these are two arrays `GF_alog` and `GF_log`, with seven entries of 3 bits width. In addition, it has tables for the generator polynomials' (P and Q) powers (i.e., log values). These arrays are `p_poly` and q_poly. All these arrays are initialised in an initial block and remain unchanged, with `GF_alog` set to be like the table in the LFSR diagram above, and `GF_Log` set with the diagram values as the indexes and the powers of *a* the entries. (An initial block is fine for FPGA, but these might be better moved to a reset block or fixed as constants, such as using local parameters. This is just for ease of illustration here.)

P and Q are defined as 3-bit vectors (of the same name), and there is a seven-entry array of 3 bits with, `s`, which holds the calculated symbols. This is used to split the flat input data (`idata`—see below) into the 3-bit words A to E, assigning them to `s[0]` through `s[4]`, whilst `s[5]` and `s[6]` are assigned to `P` and `Q` results. In this implementation, the input data (`idata`) is a flat 15-bit vector, for the 15 bits of five 3-bit words A to E. Similarly, the output is a flat 21-bit vector. Two intermediate arrays, `p_data` and `q_data`, are also defined for use in calculating the P and Q values, as we shall see. Having defined the signalling (and reference the source code for the details), let's look at the functional part of the encoder:

```
always @(*)
begin
  P = 3'b000;
  Q = 3'b000;

  for (idx = 0; idx < 5; idx = idx + 1)
  begin
    p_data[idx] = p_poly[idx] + GF_log[s[idx]];
    p_data[idx] = p_data[idx][2:0] + p_data[idx][3];
    p_data[idx] = GF_alog[p_data[idx]];
    P           = P ^ p_data[idx];

    q_data[idx] = q_poly[idx] + GF_log[s[idx]];
    q_data[idx] = q_data[idx][2:0] + q_data[idx][3];
    q_data[idx] = GF_alog[q_data[idx]];
    Q           = Q ^ q_data[idx];
  end
end
```

The code loops through all five input data words (which are assigned to `s[0]` through `s[4]`), calculating P and Q. The logs of these value are looked up by indexing into `GF_log` and adding to the polynomial logs (as appropriate for P or Q), placing the result in the temporary variables (`p_data` or `q_data`). This is the main operation of 'multiplication' by adding the logarithms.

The next line takes care of overflow, where the addition might index off the end of the tables. As we saw earlier, this is modulo 8 + 1, so the logic does the modulo 8 by taking the 3 lower bits of the addition result, and then adds one if bit 3 is 1, indicating the overflow.

Once the log addition and overflow are done, the antilog of the result is taken by indexing the result into GF_alog to get the result of the data multiplied by the polynomial term. These are accumulated in P (or Q), which were set to 0 at the start of the loop, to get all the terms XORed together, to generate the parity symbols. These are already being assigned to `s[5]` and `s[6]`. The flat output signal is then generated by concatenating each entry of the `s` array.

Like for the Hamming Verilog of Part 1, this code is purely combinatorial (though synthesisable), and a practical solution might have synchronous registers, at least on the outputs as appropriate to the design, and also be better constructed. Here, I just wish to remove any extraneous code that might obfuscate the main points of an

implementation. This also applies to the decoder, which we will discuss in the next section.

Decoder

The decoder's main purpose is to generate the syndrome values and the use them to find the index (k) that indicates where an error occurred (if any). It has its own Galois Field tables, set in an initial block, just as for the encoder. In place of P and Q we have `S0` and `S1` syndrome signals, and a 4-bit `k` index. An `s` symbol array is used as for the encoder, to hold the seven 3-bit symbols, split out from the flat input vector (`icode`). There is also a temporary calculation signal, `S1_poly`. The diagram below shows the functional Verilog fragment for the decoder.

```
always @(icode)
begin
  s[0] = icode[2:0];   s[1] = icode[5:3];
  s[2] = icode[8:6];   s[3] = icode[11:9];
  s[4] = icode[14:12]; s[5] = icode[17:15];
  s[6] = icode[20:18];

  S0   = s[0] ^ s[1] ^ s[2] ^ s[3] ^ s[4] ^ s[5] ^ s[6];
  S1   = 3'b000;

  for (idx = 0; idx < 7; idx = idx + 1)
  begin
    S1_poly[idx] = (7 - idx) + GF_log[s[idx]];
    S1_poly[idx] = S1_poly[idx][2:0] + S1_poly[idx][3];
    S1_poly[idx] = s[idx] ? GF_alog[S1_poly[idx]] : 3'b000;
    S1           = S1 ^ S1_poly[idx];
  end

  if (S0 != 3'b000)
  begin
    k = GF_log[S1] - GF_log[S0];
    k = k[2:0] - k[3];
    k = (k == 4'b0000) ? GF_log[3'b001] : k;

    s[7-k] = s[7-k] ^ S0;
  end
end
```

As discussed earlier, `S0` is just all the symbols XORed together, and so this does not need to be in the loop (though it could have been and generate the same logic). `S1` is set to 0, and then a loop iterates over all seven symbols. Firstly, in the loop, the log of the power of *a* is calculated as (7 – idx), since the loop has an increasing index, but the example multiplies the symbols by decreasing powers of *a*, as shown in the

example earlier. Just like for the encoder, once adding the logs for multiplication, any wrapping over the table is dealt with. It is possible the symbol being processed is 0 but using logs to process zero doesn't really work, so this is detected on the next line and the result set to 0, as multiplying by 0 gives 0, without the need for logs. If it isn't zero, then the antilog is taken for the result for that term. These are all accumulated over the loop by XORing into `S1`, which was initialised to 0.

Now we have `S0` and `S1`, `k` needs to be calculated. This is the division of `S1` by `S0`, turned into a subtraction with logs. Thus, the log of `S0` is subtracted from the log of `S1` indexing into the `GF_log` table. The `k` value can be negative, which needs handling, so this is done by subtracting 1 if the result underflowed in much the same was as overflow was handled. Also, the result for `k` can be 0—that is a^0, or 1. Again, logarithms don't handle zero, but the result is known to be 1, so the log of 1 is looked up in this case (k is already a power of a, so is in logarithmic form already, the value of 1 also needs to be a logarithm).

To correct the error, then, the indexed value in the `s` array (reversed for earlier stated reasons) is XORed with the `S0` to get the correct result. The flat output data vector (`odata`—not shown in code above) is just `s[4]` down to `s[0]` concatenated.

Real-World Example

The example we have worked through is meant to boil down the essential operations of Reed-Solomon. There are many variants of this basic operation. For instance, if there is a means to locate the position of the errors by some other means (e.g., product codes, not discussed here), then the locator parity symbols are not needed, halving the redundancy added or using the same number of parity symbols to correct multiple errors. Since two errors are effectively XORed together, then simultaneous equations are solved to separate them. This is vastly simplified by the fact that, as we saw, the nature of the error is not important (1, 2 or n bits in error makes no difference), so the known erroneous symbols can be set to 0, thus removing the error values from the calculations. This is known as correction by erasure.

Now, I've skipped the calculations of the polynomials P and Q in this article as I wanted to keep away from complex mathematics and because they are fixed values, known up front. In general, so the theory goes, if n symbols are to be corrected (or $n/2$ corrected and $n/2$ located), then n redundant symbols are necessary. In general, there is a generator polynomial defined as:

$$G(x) = \prod_{i=0}^{i=n-1} (x - a^i)$$ NB: $(x - a^i) = (x + a^i)$ since using modulo 2 arithmetic (XOR)

Now you might think that this doesn't look much like the polynomials of the example but the values of the powers of *a* are known constants (the values of the Galois field), and one just has to multiply it all out to get a polynomial in terms of powers of *a* multiplied by the polynomial terms (powers of *x*) to get the generator polynomials. These generator polynomials are then used to solve the simultaneous equations to get the parity symbols (P and Q in the example). Most specifications I have seen give this general form and leave it up to the engineer to plug in the values. The reason I'm explaining all this is so that the example we're about to look at makes sense, and there is a mapping back to the example already discussed. There is no space to go through this here, but for those interested in the mathematics, then there is plenty of information out there.

ECC in Digital Data Storage

Let's look at a case I'm familiar with, namely DDS as mentioned in Part 1 of the document. In DDS, Reed-Solomon encoding is done as a two-stage process. Firstly, data is interleaved to improve error robustness, and formed into two rectangular arrays (a plus and minus array), with space for parity symbols. A first RS coding is done across this array, filling in parity symbols in the middle of the rows, and the down the array, filling in parity symbols at the end of the columns, though the nature and position of the parity symbols changes with iterations of the specification. So, two RS codes are added at right angles to each other over an array of interleaved data bytes.

The specification for the two RS codes is given in the specifications as follows. There is an RS(32,28) code and an RS(32,26). It uses correction by erasure, as discussed above, and has a Galois field polynomial of:

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

The primitive element *a* is 00000010b, and the generator polynomials for the two RS codes are defined as:

$$G_P(x) = \prod_{i=0}^{i=3}(x - a^i)$$
$$G_Q(x) = \prod_{i=0}^{i=5}(x - a^i)$$

Finally, the check matrices are defined as:

$$H_P = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 & 1 & 1 \\ a^{31} & a^{30} & a^{29} & \dots & a^2 & a & 1 \\ a^{62} & a^{60} & a^{58} & \dots & a^4 & a^2 & 1 \\ a^{93} & a^{90} & a^{87} & \dots & a^6 & a^3 & 1 \end{bmatrix}$$

$$H_Q = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 & 1 & 1 \\ a^{31} & a^{30} & a^{29} & \dots & a^2 & a & 1 \\ a^{62} & a^{60} & a^{58} & \dots & a^4 & a^2 & 1 \\ a^{93} & a^{90} & a^{87} & \dots & a^6 & a^3 & 1 \\ a^{124} & a^{120} & a^{116} & \dots & a^8 & a^4 & 1 \\ a^{155} & a^{145} & a^{140} & \dots & a^{10} & a^5 & 1 \end{bmatrix}$$

note: P and Q here are not the same as in example,
but are for rows and columns of the interleaved array

This, then, is what's provided in a typical specification, which then needs to be implemented as gates—or possibly as software on an embedded processor, depending on the bandwidth requirements. Hopefully, some of this now makes sense with respect to the example discussed, though we've had to take some things as given.

Conclusions

In this article, we've taken some of the concepts from Part 1 of the document and seen how they can be used in Reed-Solomon codes to generate a code with properties robust to burst errors. An example has provided a means to show how parity symbols can be generated to locate and then correct an erroneous symbol, whatever the number of bits in the symbol are in error. One needs double the amount of parity symbols for the number of symbols that can be corrected, but one can trade this for just correction if other means of error location are employed (not discussed here) and with the use of error symbol erasure.

Having worked through the example, this was mapped to an actual implementation in logic (the source code of which can be found here), which is the main purpose of these articles, to take theory to practice. A real-world example was reviewed, in the form of DDS error correction, to see what a practical specification looks like, mapping some of this back to the example that was worked through.

I've tried to minimise the mathematical theory behind this and have taken somethings as given in order to get to a practical solution and demonstrate how one gets from mathematical symbols to actual logic gates. This, of course, means that some details have been glossed over. If you ever find yourself having to implement a solution, then there will still be work to do for your particular case. The hope is that, having seen a worked example mapped to logic, you know that it is at least possible and the problem does simplify to manageable levels.