Constructing an Error Correcting Code

Andrew E. Phelps
University of Wisconsin, Madison
Nov 9, 2006

1. Introduction

error

soft error

Whenever data is stored or transmitted, there is some chance that one or more bits will “flip” -- that is, will change to an incorrect value. Such incorrect values are called *errors*; they may be due to a permanent fault (broken hardware) or a transient condition. Transient or *soft* errors often occur when storing data in DRAM or transmitting a packet across a network connection. As transistors shrink, errors are becoming much more common; in a modern chip the devices are so small that cosmic rays or alpha particles can change the value of bits that are stored in SRAM or registers, or are simply moving across the die.

ECC

To counteract this problem and ensure reliable operation, error correcting codes (*ECC*) are used. Extra bits are sent or stored alongside the data bits to provide redundant information. With enough bits of carefully chosen redundant information, we can detect or correct the most probable classes of errors.

A trivial example of redundant information would be to simply send an additional copy of each bit. If there is a single error (one or the other bit is wrong), we can detect it (because the two bits no longer match). But this code has two disadvantages: it has a high overhead (100%), and it does not allow error correction because it does not tell us which of the two bits is wrong.

A commonly used code is parity. One extra bit is used with a group of data bits; this "parity bit" is set such that the total number of "1" bits is odd. (This is called "odd parity"; another code is "even parity" in which the total number of "1" bits is even.) This code has a low overhead for a reasonable size of data word, and it allows detection of any one error; but it still does not allow correction.

SECDED

One of the commonest error correcting codes is the *SECDED* code; this stands for "Single Error Correction, Double Error Detection". We shall see below how to create the logic for such a code.

2. Hamming Distance

First, we need a few definitions.

data bits

check bits

codeword

valid codeword

The *data bits* are the original bits we want to protect.
The *check bits* are the extra bits that we send or store alongside the data bits.
The *codeword* is the entire set of data bits and check bits.
A *valid codeword* is one where the check bits are correctly generated from the data bits according to the rules of the code.

Hamming distance

The *Hamming distance* between two words is the number of bits that are different between the two words. For example, "001" and "011" have a Hamming distance of one, because only the middle bit is different. "00111" and "01000" have a Hamming distance of four.

The minimum Hamming distance between any two valid codewords in a code determines how many errors may be detected or corrected using that code.

For example, if the minimum Hamming distance is one, then it is possible for a single bit error to make one valid codeword into another valid codeword, so it is not possible in general to detect or correct even a single error. But if the minimum Hamming distance is two, then a single error must turn a valid codeword into an invalid one, thus allowing the error to be detected.

If the minimum Hamming distance is three, then a single error can be corrected. Consider figure 1. Suppose “0000” is a valid codeword in a code with minimum distance three. If it is corrupted by an error to “0001”, we know it is not a valid codeword (since the Hamming distance is only one), and furthermore we know it is in the set of codewords that could only have started as “0000”. No other valid codeword is only a single bit-flip away from “0001”, since that would violate the assumed minimum Hamming distance for this code. Thus invalid codewords can be placed into disjoint sets that each correspond to at most one valid codeword.

Suppose 0000 and 0111 are valid codewords. A single error will produce a codeword that is correctable to the original valid codeword. But a double error could, for example, change 0000 into 0011, which would be falsely corrected to 0111.

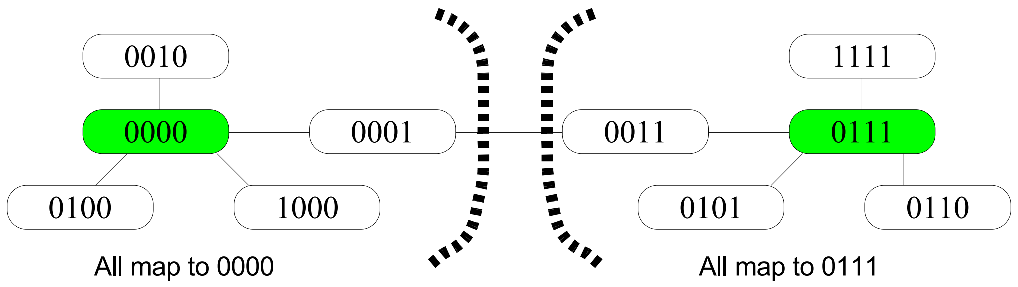

Figure 1 – Code with Minimum Hamming Distance of 3

If the minimum Hamming distance is four, then we can correct any single-bit error and detect any two-bit error. Consider figure 2. Once again, there are disjoint sets of invalid codewords that arise from single-bit errors. But a two-bit error cannot take us into a different set; it can only take us halfway to another valid codeword and thus leaves us in a no-man's-land which indicates a double-bit error.

Suppose 00000 and 01111 are valid codewords. A single error will produce a codeword that is correctable to the original valid codeword. A double error could, for example, change 00000 into 00011, which is Hamming distance two from the nearest valid codewords and thus is known to be an uncorrectable error.

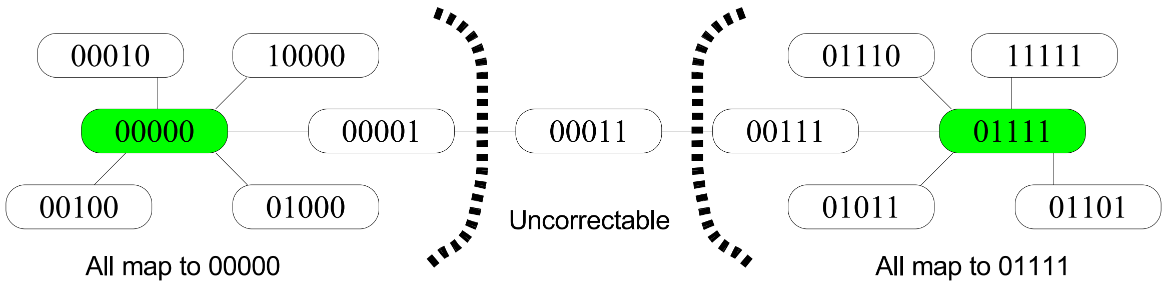

Figure 2 – Code with Minimum Hamming Distance of 4

In general, a code that can correct *n* errors and detect *m* errors ($m \geq n$) must have a minimum Hamming distance of $n+m+1$.

In order to implement an error correcting code, we need three things:

- Logic to create the check bits with the desired minimum Hamming distance
- Logic on the receiving end to check the check bits and determine if there was an error
- Logic to correct the data, if correctable

3. A Simple Error-Correcting Code

First, we will consider a single-error-correcting code with a minimum Hamming distance of three.

name

To create the check bits, we will choose a pattern of check bits associated with each data bit. Let's call this the *name* for that data bit.[1] The check bits for an entire word will be the exclusive-OR of the names for each data bit whose value is one. For example, if we choose the name "0 0 0 1 1" to go with data bit zero, and "0 0 1 1 0" to go with data bit one, then a codeword with just data bits zero and one set would have check bits "0 0 1 0 1".

But how do we choose names so that the minimum Hamming distance is three? This is actually easier than it may sound. We may choose any names we like such that:

- Each data bit has a unique name;
- Each name has at least two bits set.

To see that we have achieved our minimum distance, consider any two valid nonidentical codewords. The data portion must differ, since two valid codewords couldn't have identical data but different checkbits. If the data portion differs in:

- A single data bit, then their check words differ in the two or more bits corresponding to the name for that data bit. So the minimum distance is three (1 data + 2 check).
- Two data bits, then their check words differ by the exclusive-OR of the two corresponding names. Since each name is unique, they must differ in at least one bit, so the minimum distance is three (2 data + 1 check).
- Three or more data bits, then we have met our minimum distance of three regardless of the check bits.

The code is often represented in a table, such as the one below. The name of each data bit may be read by looking down the column below the bit. In this case, we have assigned names in order starting with bit zero, using each possible name that meets the criterion of having two or more bits set. Thus, we used these names: 0011, 0101, 0110, 0111, 1001, 1011, 1100. We could also have used 1101, 1110, or 1111, but we only needed eight of the eleven possible names. (We could have chosen any eight, and assigned them in any order.)

	Data bits							
	7	6	5	4	3	2	1	0
Check bit 0:	0	1	0	1	1	0	1	1
Check bit 1:	0	1	1	0	1	1	0	1
Check bit 2:	1	0	0	0	1	1	1	0
Check bit 3:	1	1	1	1	0	0	0	0

Each check bit then is simply the exclusive-OR of each data bit that has a "1" in the corresponding row. The logic (in Verilog) that generates the check bits for this example is given here. (In Verilog, the "^" operator denotes exclusive-OR. Since exclusive-OR is commutative and associative, the grouping of these operations into gates is immaterial.)

```
assign checkbits[0] = d[6] ^ d[4] ^ d[3] ^ d[1] ^ d[0];
assign checkbits[1] = d[6] ^ d[5] ^ d[3] ^ d[2] ^ d[0];
assign checkbits[2] = d[7] ^ d[3] ^ d[2] ^ d[1];
assign checkbits[3] = d[7] ^ d[6] ^ d[5] ^ d[4];
```

1 This is not standard terminology, but is a term that the author believes is useful.

When the code word is received (or retrieved from storage), one or more errors may have occurred. Each error is considered to be a single bit that has changed state. (Errors such as reading the wrong location in memory are not considered here.) To determine if the received codeword is valid, we generate a new set of checkbits from the data and compare it to the received checkbits. If no error has occurred, the two versions of the checkbits must match.

syndrome

The exclusive-OR of the new and old checkbits is called the *syndrome*. The syndrome tells us what we need to know about the error. Consider what happens if a single data bit i flips from zero to one. If the original data bits were x_0, x_1, ... x_n, the original checkbits were

$$\text{name}_{x0} \wedge \text{name}_{x1} \ldots \wedge \text{name}_{xn}$$

The new checkbits will be

$$\text{name}_{x0} \wedge \text{name}_{x1} \ldots \wedge \text{name}_{i} \ldots \wedge \text{name}_{xn}$$

Recalling that exclusive-OR is commutative and associative, and its own inverse, the exclusive-OR of the above two expressions is

$$\begin{aligned} & (\text{name}_{x0} \wedge \text{name}_{x0}) \wedge (\text{name}_{x1} \wedge \text{name}_{x1}) \wedge \ldots \wedge (\text{name}_{xn} \wedge \text{name}_{xn}) \wedge \text{name}_{i} \\ = {} & 0 \wedge 0 \wedge \ldots \wedge 0 \wedge \text{name}_{i} \\ = {} & \text{name}_{i} \end{aligned}$$

Thus, in this case, the syndrome is simply the name of the bad bit. In the case where data bit i flipped from one to zero, the original checkbits were

$$\text{name}_{x0} \wedge \text{name}_{x1} \ldots \wedge \text{name}_{i} \ldots \wedge \text{name}_{xn}$$

and the new checkbits are

$$\text{name}_{x0} \wedge \text{name}_{x1} \ldots \wedge \text{name}_{xn}$$

These are the same two expressions as above, in the opposite order; the exclusive-OR is the same. Either way, the result is the name of the flipped bit.

Of course, errors can occur in check bits as well as in data bits; the cosmic rays do not care which bits are which. If there is a single bit error in a checkbit, the syndrome will have just that bit set. This is easily distinguished from a data bit name, since names have at least two bits set.

A simple decoder on the syndrome can be used to generate the signals with which to exclusive-OR the data bits to correct single-bit errors. But a decode of zero (no error), or of 1, 2, 4, 8, ... (check bit error), will indicate that the data does not need to be corrected.

This code is not designed for double errors. If there are errors in two data bits, the syndrome will equal the exclusive-OR of the two names, and this will be some arbitrary non-zero number. If that number happens to be the name of some other bit, the circuit will mistakenly "correct" that bit, causing a third error. If this situation is likely to occur, then we need a stronger code.

4. SECDED Codes

The most commonly-used error-correcting codes are SECDED codes; as stated earlier this stands for "single error correcting, double error detecting". The logic is very similar to that of the single-error correcting code we just looked at, but, we saw above, we now need a minimum Hamming distance of four.

To do this, we will add one more rule in choosing names for bits. (The first two rules are the same as in the code above.)

- Each data bit has a unique name;
- Each name has at least two bits set;
- Each name must have an odd number of bits set in it.

One way to do this is by increasing the size of the checkbit field by one, and using this bit as a parity for the checkbits. This will also guarantee that the entire codeword has consistent parity, since each additional bit set in the codeword causes an even number of bits to be flipped (an odd number in the checkbits plus the one data bit). Actually, one can equivalently think of the extra checkbit as the exclusive-OR of the rest of the codeword, and it is sometimes described this way; but it is unlikely that one would implement it that way since this would become the critical path of the logic.

Note that each name now has at least three bits set in it. Now let's re-examine the cases to see if we meet our minimum Hamming distance. Consider any two valid nonidentical codewords. If their data bits differ in:

- A single data bit, then their check words differ in the three or more bits corresponding to the name for that data bit. So the minimum distance is four (1 data + 3 check).
- Two data bits, then their check words differ by the exclusive-OR of the two corresponding names. Since each name is unique, they must differ in at least one bit. But since each name has odd parity, it is impossible for them to differ in only a single bit. So the minimum distance is four (2 data + 2 check).
- Three data bits: Since the exclusive-OR of three values with odd parity must have odd parity, it must have at least one bit set. So the minimum distance is four (3 data + 1 check).
- Four or more data bits, then we have met our minimum distance of four regardless of the check bits.

Here is the code in table form. While one approach would have been to simply choose the first eight names that meet all our criteria, we have instead chosen to define check bits 0-3 as in the previous code and simply add check bit 4 as a parity bit:

	Data bits							
	7	6	5	4	3	2	1	0
Check bit 0:	0	1	0	1	1	0	1	1
Check bit 1:	0	1	1	0	1	1	0	1
Check bit 2:	1	0	0	0	1	1	1	0
Check bit 3:	1	1	1	1	0	0	0	0
Check bit 4:	1	0	1	1	0	1	1	1

One additional improvement we can make to the code: We can invert the output of some of the exclusive-OR operations (equivalent to replacing the exclusive-OR with exclusive-NOR). With the same change made in both the sender and the receiver, this has no real effect except to change the "baseline" value of some of the checkbits. This is useful because one common type of multi-bit error is to have the entire word turned "off" in some way, so that it becomes entirely zero. It is good practice to have the all-zero codeword represent an uncorrectable error. In our example, let's pick check bits 3 and 4 to make exclusive-NOR. The resulting Verilog for generating the check bits is:

```
assign checkbits[0] =   d[6] ^ d[4] ^ d[3] ^ d[1] ^ d[0];
assign checkbits[1] =   d[6] ^ d[5] ^ d[3] ^ d[2] ^ d[0];
assign checkbits[2] =   d[7] ^ d[3] ^ d[2] ^ d[1];
assign checkbits[3] = ~(d[7] ^ d[6] ^ d[5] ^ d[4]);
assign checkbits[4] = ~(d[7] ^ d[5] ^ d[4] ^ d[2] ^ d[1] ^ d[0]);
```

On the receiving end, once again, we generate new checkbits, and exclusive-OR them with the received checkbits to form the syndrome. If there is no error, the syndrome will be zero; and if there is a single error the syndrome will equal the name of the bad bit. So far this is the same as the single-error-correcting code above. But now consider what happens when there are two errors. The syndrome will be the exclusive-OR of two names (or a name and a checkbit, or two checkbits), but it cannot possibly be equal to some valid name because the parity will not match. Whether names or checkbits, it will be the exclusive-OR of two odd-parity numbers and thus will have even parity. Parity guarantees that we will detect that an uncorrectable error has occurred.

Here is the Verilog of the check/correct block:

```
// calculate syndrome:

assign synd[0] =   c[0] ^ d[6] ^ d[4] ^ d[3] ^ d[1] ^ d[0];
assign synd[1] =   c[1] ^ d[6] ^ d[5] ^ d[3] ^ d[2] ^ d[0];
assign synd[2] =   c[2] ^ d[7] ^ d[3] ^ d[2] ^ d[1];
assign synd[3] = ~(c[3] ^ d[7] ^ d[6] ^ d[5] ^ d[4]);
assign synd[4] = ~(c[4] ^ d[7] ^ d[5] ^ d[4] ^ d[2] ^ d[1] ^ d[0]);

// Decoder for syndromes:
//   The “flip” signals will be used to correct the data bits
//   The “chkb” signals indicate a single checkbit error
//   The “dbl” signals indicate a double-bit error
//   The “badcode” signals indicate unused names and thus multibit errors

assign noerror = ~synd[4] & ~synd[3] & ~synd[2] & ~synd[1] & ~synd[0];
assign chkb[0] = ~synd[4] & ~synd[3] & ~synd[2] & ~synd[1] &  synd[0];
assign chkb[1] = ~synd[4] & ~synd[3] & ~synd[2] &  synd[1] & ~synd[0];
assign dbl[0]  = ~synd[4] & ~synd[3] & ~synd[2] &  synd[1] &  synd[0];
assign chkb[2] = ~synd[4] & ~synd[3] &  synd[2] & ~synd[1] & ~synd[0];
assign dbl[1]  = ~synd[4] & ~synd[3] &  synd[2] & ~synd[1] &  synd[0];
assign dbl[2]  = ~synd[4] & ~synd[3] &  synd[2] &  synd[1] & ~synd[0];
assign flip[3] = ~synd[4] & ~synd[3] &  synd[2] &  synd[1] &  synd[0];
assign chkb[3] = ~synd[4] &  synd[3] & ~synd[2] & ~synd[1] & ~synd[0];
assign dbl[3]  = ~synd[4] &  synd[3] & ~synd[2] & ~synd[1] &  synd[0];
assign dbl[4]  = ~synd[4] &  synd[3] & ~synd[2] &  synd[1] & ~synd[0];
assign flip[6] = ~synd[4] &  synd[3] & ~synd[2] &  synd[1] &  synd[0];
assign dbl[5]  = ~synd[4] &  synd[3] &  synd[2] & ~synd[1] & ~synd[0];
assign badcode[0]=~synd[4]&  synd[3] &  synd[2] & ~synd[1] &  synd[0];
assign badcode[1]=~synd[4]&  synd[3] &  synd[2] &  synd[1] & ~synd[0];
assign dbl[6]  = ~synd[4] &  synd[3] &  synd[2] &  synd[1] &  synd[0];
assign chkb[4] =  synd[4] & ~synd[3] & ~synd[2] & ~synd[1] & ~synd[0];
assign dbl[7]  =  synd[4] & ~synd[3] & ~synd[2] & ~synd[1] &  synd[0];
assign dbl[8]  =  synd[4] & ~synd[3] & ~synd[2] &  synd[1] & ~synd[0];
assign flip[3] =  synd[4] & ~synd[3] & ~synd[2] &  synd[1] &  synd[0];
assign dbl[9]  =  synd[4] & ~synd[3] &  synd[2] & ~synd[1] & ~synd[0];
assign flip[3] =  synd[4] & ~synd[3] &  synd[2] & ~synd[1] &  synd[0];
assign flip[3] =  synd[4] & ~synd[3] &  synd[2] &  synd[1] & ~synd[0];
assign dbl[10] =  synd[4] & ~synd[3] &  synd[2] &  synd[1] &  synd[0];
assign dbl[11] =  synd[4] &  synd[3] & ~synd[2] & ~synd[1] & ~synd[0];
assign flip[4] =  synd[4] &  synd[3] & ~synd[2] & ~synd[1] &  synd[0];
assign flip[5] =  synd[4] &  synd[3] & ~synd[2] &  synd[1] & ~synd[0];
assign dbl[12] =  synd[4] &  synd[3] & ~synd[2] &  synd[1] &  synd[0];
assign flip[7] =  synd[4] &  synd[3] &  synd[2] & ~synd[1] & ~synd[0];
assign dbl[13] =  synd[4] &  synd[3] &  synd[2] & ~synd[1] &  synd[0];
assign dbl[14] =  synd[4] &  synd[3] &  synd[2] &  synd[1] & ~synd[0];
assign badcode[2]=synd[4] &  synd[3] &  synd[2] &  synd[1] &  synd[0];

// final results:
```

```
assign corrected_data = d ^ flip;
assign uncorrectable = (| dbl[14:0])      // even number of errors
                     | (| badcode[2:0]); // 3 or more (odd) errors
                                          // that don't even LOOK like
                                          // correctable errors

// Note:  The chkb[] signals, indicating an error in the checkbits, are
// unused because no action is necessary in this case.
```

What happens if three bits get flipped? Since this is a SECDED code, it makes no guarantees. The exclusive-OR of three odd-parity names might or might not equal some other valid name, causing a mis-correction. (The few cases that don't look like a valid name trigger the “badcode” signals in the logic above.) Any even number of errors, however, will always be flagged as an uncorrectable error, since the parity of the syndrome will be same as for a two-bit error.

5. Other codes

When choosing a code, it is important to consider what types of errors are expected in the system being protected. The SECDED code above assumes that single-bit errors are important, two-bit errors are rare, and three-bit (or more) errors are so rare that they can be ignored. This is often a good assumption, but not always. Suppose you are storing data in a memory that is built from 4-bit-wide chips, and there is some possibility of an entire chip failing. What code would we use then?

We do not need to detect any arbitrary 4-bit error, but we do want to detect the case where a single chip fails causing 1, 2, 3, or 4 errors within one aligned group of four bits (one “nibble”). Our code already detects any 1, 2, or 4 bits, but what about 3?

Here is another way to state the problem: For any aligned group of four bits, their names must be chosen such that the exclusive-OR of any three of them cannot look like any valid name.

Here is one solution: Choose the names such that bits 0, 1, and 2 have these values:

Bit 0 of any nibble:	0 0 0
Bit 1 of any nibble:	0 0 1
Bit 2 of any nibble:	0 1 0
Bit 3 of any nibble:	1 0 0

The rest of the bits of the name are chosen so the names meet the same rules given before. One can see by inspection that the exclusive-OR of any three of these cannot equal a valid name, since it will have two or three bits set among bits 0, 1, and 2, while all valid names have only zero or one bits set.

This code requires an additional checkbit, which is not surprising since we are deliberately making valid codewords more sparse. Codes exist, however, for doing nibble detection with larger data words without any addition bits beyond what is needed for SECDED.

The code above corrects any one error, detects any two, and detects any error within one nibble. More complex codes can be used to correct any error within a nibble, or even to do that while correcting single additional errors as well. Adding more check bits allows making a code stronger in order to overcome more types of expected errors.