intel

APPLICATION NOTE

AP-46

Error Detecting and Correcting Codes Part 1

JOE ALTNETHER
MEMORY COMPONENTS APPLICATIONS

9800752

INTRODUCTION

Complex electronic systems require the utmost in reliability. Especially when the storage and retrieval of critical data demands faultless operation, the system designer must strive for the highest reliability possible. Extra effort must be expended to achieve this high reliability. Fortunately, not all systems must operate with these ultra reliability requirements.

The majority of systems operate in an area where system failure ranges from irritating, such as a video game failure, to a financial loss, such as a misprinted check. While these failures are not hazardous, reliability is important enough to be designed into the system.

A memory system is one of the system components for which reliability is important. Also, it is one of the few system components which can be altered to greatly enhance its reliability. The purpose of this report is to examine different methods of error encoding, especially Error Correction Codes (ECC), to increase the reliability of the memory system.

SYSTEM RELIABILITY

Individual device reliability is the foundation of memory system reliability. Reliability is expressed as mean time between failures. The mean time between failures (MTBF) of a system is a function of the number of devices and the device failure rate. Failure rate of the memory device can be obtained from the reliability report on the specific device. MTBF of the device is:

$$T_D = \frac{1}{\lambda} \qquad [1]$$

where T_D = MTBF of the *device*

λ = *device* failure rate (%/1000 hrs)

and MTBF of the *system* is approximately:

$$T_S = \frac{T_D}{D} \qquad [2]$$

where T_S = MTBF of the *system*

D = number of devices in the *system*

As the number of devices required to construct a system becomes larger, the system MTBF becomes smaller.

A plot of system MTBF as a function of the number of memory devices is shown in Figure 1 for different failure rates. Included for reference are the failure rates of the Intel® 2104A 4Kx1 RAM and the Intel® 2117 16Kx1 RAM. Using RAMs which are organized one bit wide, the amount of devices required for a system is calculated by multiplying the number of words by the word length

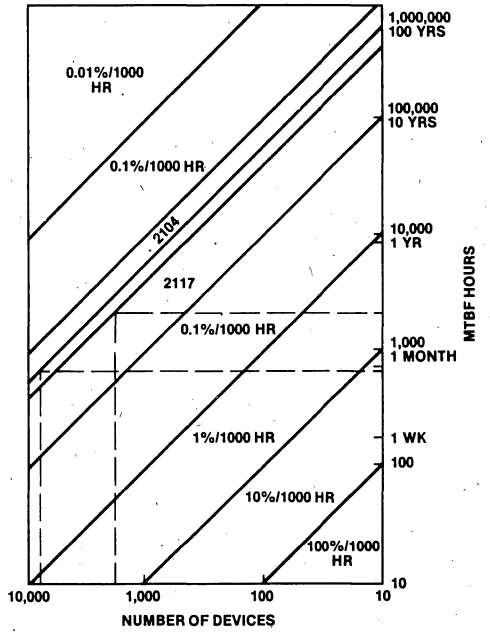

Figure 1. System Reliability vs Number of Devices

and dividing by the size of the RAM. To illustrate, assume a 1 megaword memory system with a word width of 32 bits, implemented with Intel® 2104A 4Kx1 RAMs. The number of required devices is:

$$D = \frac{1{,}048{,}576 \times 32}{4{,}096} = 8{,}192 \text{ devices}$$

Prediction of failure for this system, shown in Figure 1, is 667 hours or 28 days — assuming continuous use and worst case temperature.

Equation 2 showed that system MTBF is increased when fewer devices are used. A one megaword memory having 32 bit wide words can be constructed with Intel 2117 16K RAMs. In this case one fourth as many devices are required — 2048 devices. From Equation 2, the expected MTBF should be four times as large — 2668 hours. It is not. The failure rate from Figure 1 for this system is 2000 hours. Different device failure rates account for this difference. The failure rate of the 16K is not *yet* equal to that of the 4K. Memory device reliability is a function of time as shown in Figure 2. Reliability improvement often is a result of increased experience in manufacturing and testing. In time, the failure rate of the 16K will reach that of the 4K and one fourth as many devices will result in a system MTBF approximately four times better.

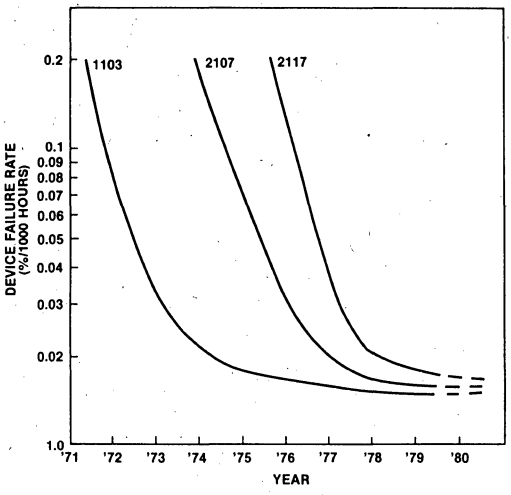

Figure 2. Device Failure Rate as a Function of Time.

The failure rate of a system without error correction will follow a similar curve over time. Indeed, in very large systems built with large numbers of devices, the *system* failure rate may be intolerable, even with very reasonable *device* failure rates. To increase the system reliability beyond the device reliability, *redundancy coding techniques* have been developed for detecting and correcting errors.

REDUNDANCY CODES

Redundancy codes add bits to the data word to provide a validity check on the entire word. These additional bits, used to detect whether or not an error has occurred, are called encoding bits. With M data bits and K encoding bits, the encoded word width is N bits. Shown in Figure 3 is the form of the encoded word.

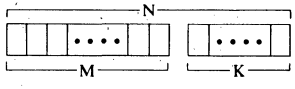

Figure 3. Encoded Word Form

Mathematically, N is related to M and K by:

$$N = M + K \qquad [3]$$

where N = number of bits in the encoded word

M = number of data bits

K = number of encoding bits

Exactly how K is related to M, and the number of required K bits depends on several factors which will be described later.

One measure of a code is its efficiency. Efficiency is the ratio of the number of bits in the encoded word to the number of bits of data:

$$E = \frac{N}{M}$$

Substituting $N = M + K$:

$$E = \frac{M + K}{M} \qquad [4]$$

where E = efficiency

All of the data are contained in the M bits. The K bits contain no data, only validity checks. To maximize the amount of data in the encoded word, the number of K bits must be minimized. Examination of Equation 4 shows that the minimum value of K is zero. With K equal to zero, the efficiency is unity. Efficiency is maximized, but the word has no encoding bits. Therefore, it has no capability to detect an error.

As an example, consider a two bit word. It can assume 2^2 or 4 states, which are:

State 1	00
State 2	01
State 3	10
State 4	11

Figure 4. All States of a Two-Bit Word

All possible states have been used as data; consequently any error will cause the error state to be identical to a valid data state.

The mechanics of the encoding bits create encoded words such that every valid encoded word has a set of error words which differ from all valid encoded words. When an error occurs, an error word is formed and this word is recognized as containing invalid data.

By adding one K bit to the two bit word error detection becomes possible. The value of the K bit will be such that the encoded word has an odd number of ONES. As will be explained later, this technique is "odd" parity.

The sum of the ONES in a word is the *weight* of the word. Parity operates by differentiating between odd and even weights. The encoded word will always have an odd weight as a result of having an odd number of ONES.

If a single bit error occurs, one bit in the encoded word will change state and the word will have an even weight. Then in this example, all encoded states with an even weight — an even number of ones — are error states.

The value of the encoding bit or parity bit is found by counting the number of ones — calculating the weight — and setting the value of K to make the weight of the encoded word odd. Referring to Figure 4, State 1 was 00,

the weight of this word is 0, so K is set to 1 and the weight of the encoded word is odd. State 2 is 01, the weight is odd already, so K is set to 0. The weight of State 3 is identical to that of State 2 so K is again set to 0. Finally, State 4 has an even weight (1 + 1 = 2), thus K is 1. The encoded states of the two bit data word are listed in Figure 5.

	Data	Encoding Bit
State 1	00	1
State 2	01	0
State 3	10	0
State 4	11	1
	M	K
	N	

Figure 5. Code Bits for All Possible States of a Two-Bit Word

To illustrate the error detection, Figure 6a lists all states of the encoded data word and all possible single bit errors. Because the encoded word is 3 bits long, there are only 3 possible single bit errors for each encoded state.

	A	B	C	D
Encoded States	001	010	100	111
Error States	000	000	000	011
	011	011	101	101
	101	110	110	110

Figure 6a. All Possible Single-Bit Errors

Notice that every error state has an even weight, while the valid encoded states have odd weights.

Converting all the values of these states to decimal equivalents makes the errors more obvious as shown in Figure 6b.

Valid States	1	2	4	7
Error States	0	0	0	
	3	3		3
	5		5	5
		6	6	6

Figure 6b. Decimal Representation of Errors

No error state is the same as any valid encoded state. Identical error states can be found in several columns. The fact that some error states are identical prevents identification of the bit in error, and hence correction is impossible. Importantly though, error detection has occurred.

Figure 6a demonstrates another property of codes. Every error state differs from its valid encoded state by one bit, whereas each of the encoded states differs from the others by two bits. Examine the encoded states labeled B and D in Figure 6a and shown in Figure 7.

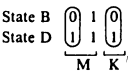

Figure 7. Bit Difference.

These two states have two bit positions which differ. This *difference* is defined as *distance* and these two states have a distance of two. Distance, then, is the number of bits that differ between two words. The encoded words have a minimum distance of two. Longer encoded words may have distances greater than two but never less than two if error detection is desired. The error states have a minimum distance of one from their valid encoded state.

A minimum distance of two between encoded states is required for error detection. A re-examination of a word with no encoding bits shows that the states have a minimum distance of 1 (see Figure 8). No error detection is possible because any single bit error will result in a valid word.

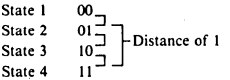

Figure 8. Minimum Distance of a Two-Bit Word

PARITY

A minimum distance of two code is implemented with Parity. Refer to previous section for an explanation. Parity is generated by exclusive-ORing all the data bits in the word, which results in a parity bit. This parity bit is the K encoding bit of the word. If the word contains M data bits, the parity bit is:

$$C = b1 \oplus b2 \oplus b3 \oplus \ldots \oplus bm$$

where C = parity bit

b = value in the bit position

The parity bit combines with the original data bits to form the encoded word as shown in Figure 9. Encoded words always have either "odd" parity, which is an odd number of 1s (an odd weight) or "even" parity which is an even number of 1s (an even weight). Odd and even parity are never intermixed, so that the encoded words all have either odd or even parity — never both.

When the encoded word is fetched, the parity bit is removed from the word and saved. A new parity bit is generated from the M bits. Comparing this new parity bit with the stored parity bit determines if a single bit error has occurred.

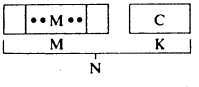

Figure 9. Encoded Word Form

Consider the two bit data word whose value is "01." Exclusive-NORing the two data bits generates a parity bit which causes the encoded word to have odd parity:

$$\overline{C} = \overline{0 \oplus 1}$$

$$\overline{C} = 0$$

The encoded word becomes:

$$\underset{\overline{0\ 1}}{M} \quad \underset{\overline{0}}{K}$$

parity

LSB of data

Assume that an error occurs and the value of the word becomes "110." Stripping off the parity bit and generating a new parity bit:

$$\text{transmitted parity} = 0$$

$$\text{transmitted word} = 11$$

$$\text{new parity of transmitted word} = \overline{1 \oplus 1} = 1$$

$$\text{generated parity} \neq \text{transmitted parity}$$

Note that the error could have occurred in the parity bit and the final result would have been the same. An error in the encoding bit as well as in the data bits can be detected.

Although parity detects the error, no correction is possible. This is because each valid word can generate the same error state. Illustration of this is shown in Figure 10.

Correct Word with Parity	Possible Single Bit Error
0 0 1	0 1 1
1 1 1	0 1 1
0 1 0	0 1 1

Figure 10. Possible Errors

Each of the errors is identical to the others and reconstruction of the original word is impossible.

Parity fails to detect an *even* number of errors occurring in the word. If a double bit error occurs, no error is detected because two bits have changed state, causing the weight of the word to remain the same.

Using the encoded word "010" one possible double bit error (DBE) is:

1 1 1

parity

Checking parity:

$$\overline{C} = \overline{1 \oplus 1} = 1$$

The transmitted parity and the regenerated parity agree. Therefore the technique of parity can detect only an *odd* number of errors.

In summary, single bit parity will detect the majority of errors, but cannot be used to correct errors. Using parity introduces a measure of confidence in the system. Should a single bit error occur, it will be detected.

ERROR CORRECTION

Classical texts on error coding contain proofs showing that a minimum distance of three between encoded words is necessary to correct errors. While this fact does not describe the code, it does give an indication of the form of the code.

Correcting errors is not as difficult as it first appears. As a result of a paper published by R. W. Hamming on error correction the most widely used type of code is the "Hamming" code. Using the same technique as parity, Hamming code generates K encoding bits and appends them to the M data bits. As shown in Figure 11, this N bit word is stored in memory.

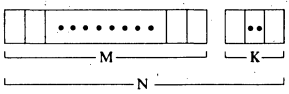

Figure 11. Encoded Word Form

Thus far the mechanism is similar to parity. The only difference is the number of K bits and how they relate to the M data bits.

When the word is read from memory, a new set of code bits (K′) is generated from the M′ data bits and compared to the fetched K encoding bits. Comparison is done by exclusive-ORing as shown in Figure 12. Like parity the result of the comparison — called the syndrome word — contains information to determine if an error has occurred. Unlike parity, the syndrome word also contains information to indicate which bit is in error.

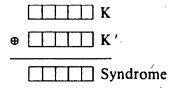

Figure 12. Syndrome Generation

The *syndrome word* is therefore K bits wide. The syndrome word has a range of 2^K values between 0 and $2^K - 1$. One of these values, usually zero, is used to indicate that no error was detected, leaving $2^K - 1$ values to indicate which of the N bits was in error. Each of these $2^K - 1$ values can be used to uniquely describe a bit in error. The range of K must be equal to or greater than N. Mathematically, the formula is:

$$2^K - 1 \geq N$$

$$\text{but } N = M + K$$

$$\text{and } 2^K - 1 \geq M + K \qquad [5]$$

Equation 5 gives the number of K bits needed to correct a single bit error in a word containing M data bits. Ranges of M for various values of K are calculated and listed in Table I.

K	Single Correct/ Single Detect ⩽ M <		Single Correct/ Double Detect ⩽ M <	
4	4	11	1	3
5	12	26	4	10
6	27	57	11	25
7	58	120	26	56
8	121	245	57	119

Table I.
Range of M for Single Correct/Single Detect or Double Detect Codes for Values of K

To detect and correct a single bit error in a 16 bit data word, five encoding bits must be used. As a result, the total number of bits in the encoded word is 21 bits.

Efficiencies of single detect — parity — and single detect/single correct codes as a function of the number of data bits are shown in Figure 13. For large values of M, the efficiency of single detect/correct is approximately equal to that of the single detect code — parity.

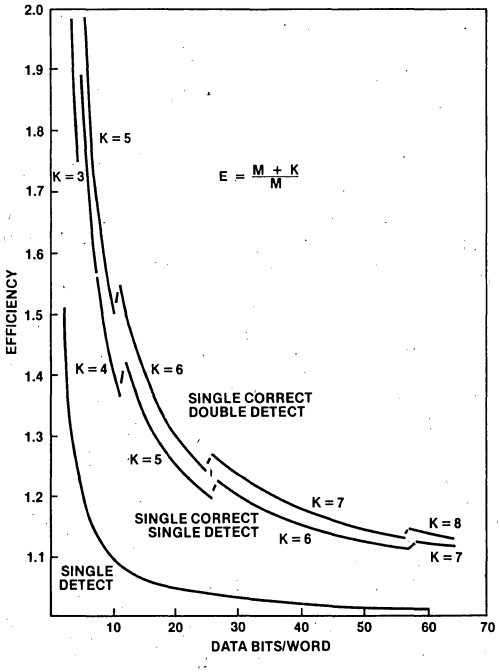

Figure 13. Code Efficiency vs Data Word Size

CODE DEVELOPMENT

Contained in the syndrome word is sufficient information to specify which bit is in error. After decoding this information, error correction is accomplished by inverting the bit in error. All bits, including the encoding bits — called check bits — are identified by their positions in the word.

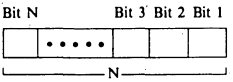

Figure 14. Positional Representation of Bits in the Word

Bits in the N bit word are organized as shown in Figure 14. Bit numbers shown in decimal form are converted to binary numbers. From equation 5, this binary number will be K bits wide. In Figure 15 is an example using a 16 bit data word. Because there are 16 data bits, M equals 16, K equals 5 and N equals 21. Shown in Figure 15 the word is binary equivalent of the position. Notice that where the M and the K bits are located is not yet specified.

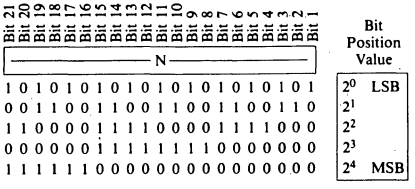

Figure 15. Binary Value of Bit Position.

The syndrome word is the difference between the fetched check bits and the regenerated check bits. Identification of the bit in error by the syndrome word is provided by the binary value of the bit position. The syndrome word is generated by exclusive-ORing the fetched check bits with the regenerated check bits. Any new check bits that differ from the old check bits will set 1s in the syndrome word. To identify bit 3 as a bit in error, the syndrome word will be 00011, which is the binary value of the bit position. Weight is determined only by the 1s in the bit position chart in Figure 15, so they are replaced with an X and the 0s are deleted. The result is shown in Figure 16.

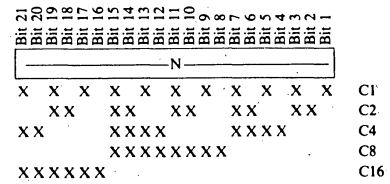

Figure 16. Relationship of Data Bits and Check Bits.

Check bit function is now defined by equating the check bits to the powers of 2 in the binary positions. Each check bit will operate on every bit position that has an X in the row shown in Figure 16. Five bit positions — 1, 2, 4, 8, and 16 — have only one X in their columns. The corresponding check bits are in these respective locations. Check bit C1 is stored in Bit Position 1, C2 is stored in Bit Position 2, and C4, C8, and C16 are stored in positions 4, 8, and 16 respectively. Because each of these positions has one X in the column, the check bits are independent of one another. If a check bit fails, the syndrome word will contain a single "1." A data bit failure will be identified by two or more "1s" in the syndrome word.

The data bits are filled in the positions between the check bits. The least significant bit (LSB) of data is located in position 3.

Data Bit 2 is stored in position 5 — position 4 is a check bit. Figure 17 shows the positions of data bits and check bits for sixteen bits of data.

When the check bits are generated for storage, bits 1, 2, 4, 8, and 16 are omitted from the generation circuitry because they do not yet exist, being the result of generation.

Parity check on the specified bits is used to generate the check bits. Each check bit is the result of exclusive-ORing the data bits marked with an "X" in Figure 18. Check bits are generated by these logic equations:

$$C1 = M1 \oplus M2 \oplus M4 \oplus M5 \oplus M7 \oplus M9 \oplus M11 \oplus M12 \oplus M14 \oplus M16$$

$$C2 = M1 \oplus M3 \oplus M4 \oplus M6 \oplus M7 \oplus M10 \oplus M11 \oplus M13 \oplus M14$$

$$C4 = M2 \oplus M3 \oplus M4 \oplus M8 \oplus M9 \oplus M10 \oplus M11 \oplus M16 \oplus M16$$

$$C8 = M5 \oplus M6 \oplus M7 \oplus M8 \oplus M9 \oplus M10 \oplus M11$$

$$C16 = M12 \oplus M13 \oplus M14 \oplus M15 \oplus M16$$

How the Hamming code corrects an error is best shown with an example. In this example, a data word will be assumed, check bits will be generated, an error will be forced, new check bits will be generated, and the syndrome word will be formed. Assuming the 16-bit data word

0101 0000 0011 1001

Check bits are generated by overlaying the data word on the Hamming Chart of Figure 16 and performing an odd parity calculation on the bits matching the "Xs."

The simplest mechanism to calculate the check bits is shown in Figure 18. The data word is aligned on the chart. Because weight and hence parity are affected only by "1s," only columns containing "1s" are circled for identification. The check bits are the result of odd parity generated on the rows. For example, the C1 row has three "Xs" circled; therefore C1 is 0 to keep the row parity odd. In this example, all other rows contain an even number of circled "Xs;" therefore the remaining check bits are "1s." These check bits are incorporated into the data word, forming the encoded word. Performing this function, the 21 bit encoded word is:

	C16		C8		C4		C2	C1
0101	0	1	000	0011	1	100	1	1	1	0

Forcing an error with bit position 7 — data bit 4:

	C16		C8		C4		C2	C1
0101	0	1	000	0011	1	000	1	1	1	0

A new set of check bits is generated on the error word as shown in Figure 18 and is:

C16	C8	C4	C2	C1
1	1	0	0	1

When the new check bits are exclusive-ORed with the old check bits, the syndrome word is formed:

	C16	C8	C4	C2	C1	
	1	1	0	0	1	New check bits
⊕	1	1	1	1	0	Old check bits
	0	0	1	1	1	

The result is 00111, indicating that bit position 7 — data bit 3 — is in error. Bit position of the error is indicated directly by the syndrome word.

While this "straight" Hamming code is simple, implementing it in hardware does present some problems. First, the number of bits exclusive-ORed to generate parity is not equal for all check bits. In the preceding example, the number of bits to be checked ranges from 10 to 5. The propagation delay of a 10 input exclusive-OR is much longer than that of a 5 input exclusive-OR. The system must wait for the longest propagation delay path, which slows the system. Equalizing the number of bits checked will optimize the speed of the encoders.

16	15	14	13	12		11	10	9	8	7	6	5		4	3	2		1			Data Bits
					C16								C8				C4		C2	C1	Check Bits
21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	Position

Figure 17. Data and Check Bit Positions in the Encoded Word.

16	15	14	13	12	C16	11	10	9	8	7	6	5	C8	4	3	2	C4	1	C2	C1	
X		X		X		X		X		X		X		X		X		X		X	C1
		X	X			X	X			X	X			X	X			X	X		C2
X	X					X	X	X	X					X	X	X	X				C4
						X	X	X	X	X	X	X	X								C8
X	X	X	X	X	X																C16

Figure 18a. Hamming Chart.

Bit Position	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1		
Data Bit	16	15	14	13	12	C16	11	10	9	8	7	6	5	C8	4	3	2	C4	1	C2	C1		
	X		X		X		X		X		X		X		X		X		X			C1	0
		X		X			X	X			X	X			X	X			X			C2	1
	X	X					X	X	X	X					X	X	X					C4	1
							X	X	X	X	X	X	X									C8	1
	X	X	X	X	X																	C16	1
Word as Stored	0	1	0	1	0		0	0	0	0	0	1	1		1	0	0		1				
Word as Fetched	0	1	0	1	0	1	0	0	0	0	0	1	1	1	0	0	0	1	1	1	0		
	X		X		X		X		X		X		X		X		X		X			C1	1
		X	X	X			X	X			X	X			X	X			X			C2	0
	X	X					X	X	X	X					X	X	X					C4	0
							X	X	X	X	X	X	X									C8	1
	X	X	X	X	X																	C16	1

Figure 18b. Check Bit Generation.

Secondly, two bits in error can cause a correct bit to be indicated as being in error. For example, if check bits C1 and C2 failed, data bit 1 would be flagged as a bit in error.

Because of these two difficulties, the Error Correction Code (ECC) most commonly used is a "modified" Hamming code is most widely used which will detect double bit errors and correct single bit errors.

SINGLE BIT CORRECT/ DOUBLE BIT DETECT CODES

Modern algebra can be used to prove that a minimum distance of four is required between encoded words to detect two errors or correct a single bit error. An excellent text on this subject is *Error Correcting Codes* by Peterson and Weldon.

One possible double bit error is two check bits. Using straight Hamming code, the circuit would "correct" the wrong bit. Double error detection techniques — modified Hamming codes — prevent this by separating the encoded words by a minimum distance of four. As a result each data bit is protected by a minimum of three check bits, so that the syndrome word always has an odd weight. Therefore, even weight syndrome words cannot be used. When two check bits fail, the syndrome word has two "1s" or an even weight. Even weight is detectable as a double bit error by performing a parity check on the syndrome word. If two data bits fail, again the syndrome word has an even weight — a detectable error.

Adding one additional check bit to the correction check bits provides the capability to detect double bit errors. The number of encoding or check bits required to detect double bit errors and correct single bit errors is:

$$2^M \leqslant \frac{2^{N-1}}{N}$$

Substituting M + K for N:

$$2^{K-1} \geqslant M + K \qquad [6]$$

Equation 6 is similar to equation 5, which describes single bit correct and detect except for the left side of the inequality, which shows one additional encoding bit is required. For single bit detect and correct the left side of the inequality was 2^K. Table I also lists the ranges of M for values of K, for a direct comparison to single bit detect and single bit correct codes.

Figure 13 includes the efficiency curve for single bit correct/double bit detect (SBC/DBD) codes for values of M. As would be expected, because of the additional encoding bit the efficiency is slightly lower. For large values of M, the efficiency of this code approaches unity like the two other curves.

Syndrome words for the SBC/DBD code are developed like the straight Hamming code, except that syndrome words do not map directly to bit positions. The syndrome word has an odd weight and does not increment like straight Hamming code. In addition, implementation considerations can impose constraints. For example, the 74S280 parity generator is a nine input device. If a check bit is generated from ten bits, extra hardware is required.

Empirical methods can be used to form the syndrome words. All possible states of the encoding bits are listed and those with an even weight are stricken from the list. Again like Hamming code, states which have a weight of one are used for syndrome words for check bits. For a sixteen bit data word, six check bits are required. Figure 19 lists the possible states of syndrome words for a 16 bit data word.

C6	C5	C4	C3	C2	C1
1	1	1	0	0	0
1	1	0	1	0	0
1	1	0	0	1	0
1	1	0	0	0	1
1	0	1	1	0	0
1	0	1	0	1	0
1	0	1	0	0	1
1	0	0	1	1	0
1	0	0	1	0	1
1	0	0	0	1	1
0	1	1	1	0	0
0	1	1	0	1	0
0	1	1	0	0	1
0	1	0	1	1	0
0	1	0	1	0	1
0	1	0	0	1	1
0	0	1	1	1	0
0	0	1	1	0	1
0	0	1	0	1	1
0	0	0	1	1	1
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	0	0	0
1	0	0	0	0	0

Figure 19. Possible Syndrome Words

In Figure 19 only twenty syndrome words for data bits are listed, because the possible words with a weight of 5 were eliminated so that every data bit would have only three bits protecting it. This simplifies the hardware implementation. If there are more than 20 data bits, states with a weight of 5 must be used. All states listed in Figure 19 are valid syndrome words, so that the problem becomes one of selecting the optimum set of syndrome words. To minimize circuit propagation delay the number of data bits checked by each encoding bit should be as close as possible to all the others.

The syndrome words can be mapped to any bit position, providing that identical code generations are done at storage and retrieval times. Syndrome word mapping may be arranged to solve system design problems. For example, in byte oriented systems the lower order syndrome bits are identical, so that the circuit design may be simplified by using these syndromes to determine which bit is in error, and the higher order syndromes to determine which byte is in error. Double bit detect/single bit correct code is implemented in hardware as a straight Hamming code would be.

DESIGN EXAMPLE

To illustrate code development, the design example uses single bit correct/double bit detect code on a 16 bit data word. In addition to the memory, the ECC system has five components: write check bit generator, read check bit generator, syndrome generator, syndrome decoder, and bit correction. Connected together as shown in Figure 20, these components comprise the basic system. Features can be added to the system to enhance its performance. Some systems include error logs as a feature. Because the address of the error and the errors are known, the address and the syndrome word are saved in a non-volatile memory. At maintenance time this error log is read and the indicated defective devices are replaced. Being a basic design, this example does not include an error log.

Write check bits are generated when data are written into the memory, while read check bits are generated when data are read from the memory. Off-the-shelf TTL is used to implement the design. Check bits are generated by performing parity on a set of data bits, so that this function is performed by 74S280 9-bit parity generators. One parity generator for each check bit is required. Because the read and write check bit generations are the same, the circuits are similar. One minor difference should be noted. In this example, the check bit will be formed from parity on eight data bits. The 74S280 parity generator has nine inputs; therefore, the write check bit generator will have the extra input grounded while the read generator has as an input the fetched check bit. Developed directly in the read check bit generator is the syndrome bit, which saves one level of gating. Figure 21 shows the identical results of generating the syndrome bit by exclusive-ORing the fetched check bit with the regenerated check bit and forming the syndrome bit in the read check bit generator.

Implementing the syndrome generator word in this way reduces the circuit propagation delay by approximately 10 nanoseconds. This implementation imposes a restriction on the code to be used — the check bit must be formed from no more than eight data bits.

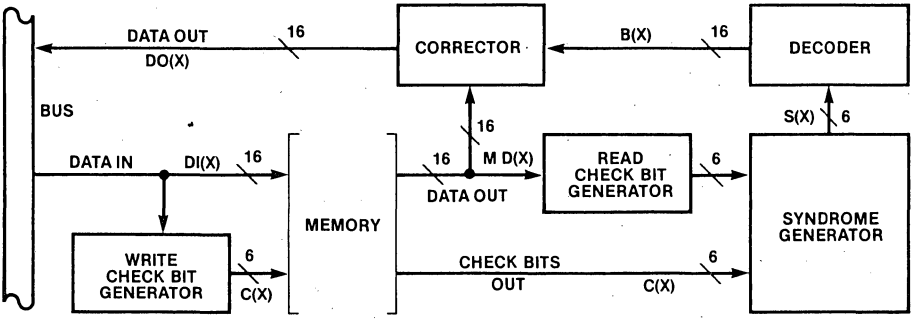

Figure 20. Block Diagram of ECC System.

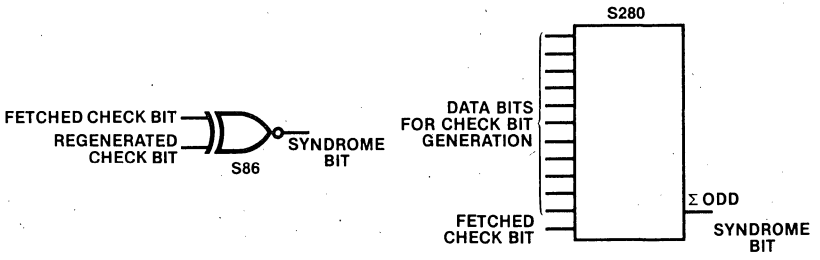

Figure 21. Syndrome Bit Generation.

Figure 19 listed the possible syndrome words for a 16 bit data word. These are relisted in Figure 22 with the syndrome words for the check bits and the zeros deleted.

```
      1    1  1 1    1  1 1  1 1 1          C1
    1    1  1  1  1  1  1  1  1  1          C2
  1    1    1 1  1    1 1  1 1    1         C3
1      1 1 1      1 1 1      1 1 1          C4
1 1 1 1             1 1 1 1 1 1            C5
1 1 1 1 1 1 1 1 1 1 1 1                    C6
```

Figure 22. Possible Syndrome Words with Three Check Bits.

While there are twenty possibilities for syndrome words, only 16 are needed. Each row contains ten "1s" and each column contains three "1s." Four columns are eliminated but in a way that each row contains eight "1s." When the columns are matched to data bits, the "1s" in each row define inputs to the 74S280 parity generators for the given check bit. Eliminating the two columns from each end results in sixteen columns with each row having eight "1s." These remaining sixteen columns which match the data bits are rearranged in Figure 23 for convenience of printed circuit board layout and assigned to the data bits. The syndrome words for check bits are also shown for complete code development.

								Data Bit												
M16	M15	M14	M13	M12	M11	M10	M9	M8	M7	M6	M5	M4	M3	M2	M1	C1	C2	C3	C4	C5	C6	
X				X	X	X		X		X			X		X	X						C1
	X	X				X	X		X		X			X	X		X					C2
X	X		X		X		X					X	X	X				X				C3
X	X	X	X	X						X	X	X							X			C4
		X	X	X	X	X	X	X	X											X		C5
								X	X	X	X	X	X	X	X						X	C6

Figure 23.

With this information the check bit generators can be designed. Figure 24 depicts write check bit generators while Figure 25 depicts read check bit generators.

Double bit error detection is accomplished by generating parity on the syndrome bits. Except for the syndrome word of 000000 — no error — even parity will be the result of a double bit error. Hardware implementation is shown in Figure 26. OR-ing the syndrome detects the zero state, which has even parity and prevents flagging this state as a double bit error.

Decoding the syndrome word must be done to invert the one bit in error. Combinational logic will decode only those syndrome states which select the one of sixteen bits for correction. Figure 28 shows the logic of the decoder.

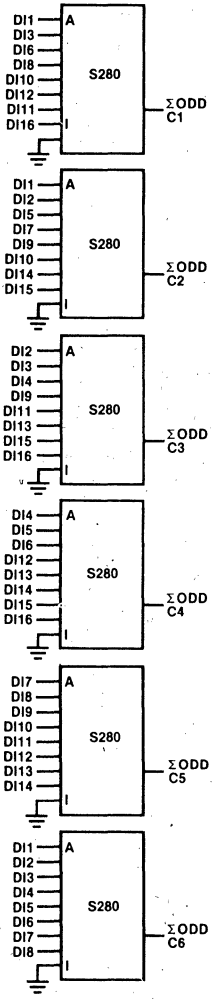

Figure 24. Write Check Bit Generators

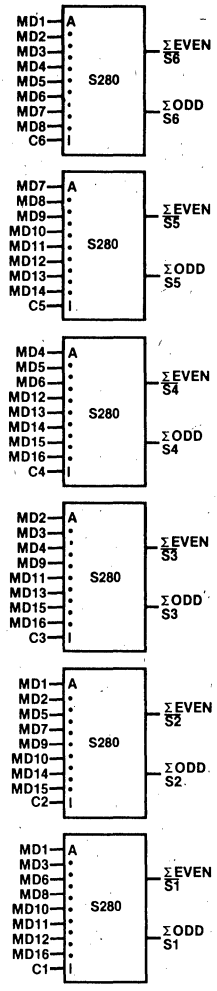

Figure 25. Read Check Bit Generators

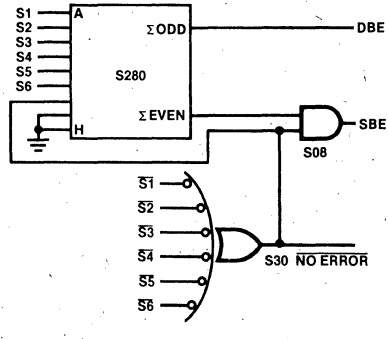

Figure 26. Double Error Decoder

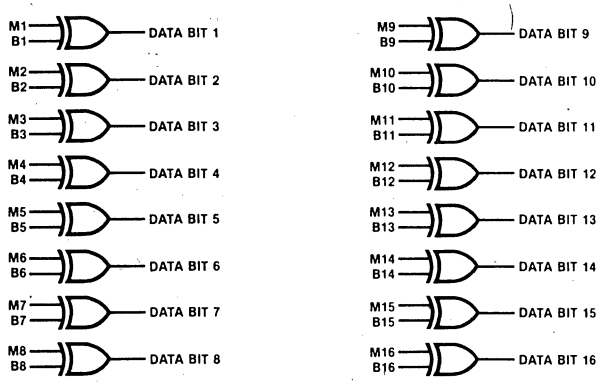

Figure 27.Correction Circuit.

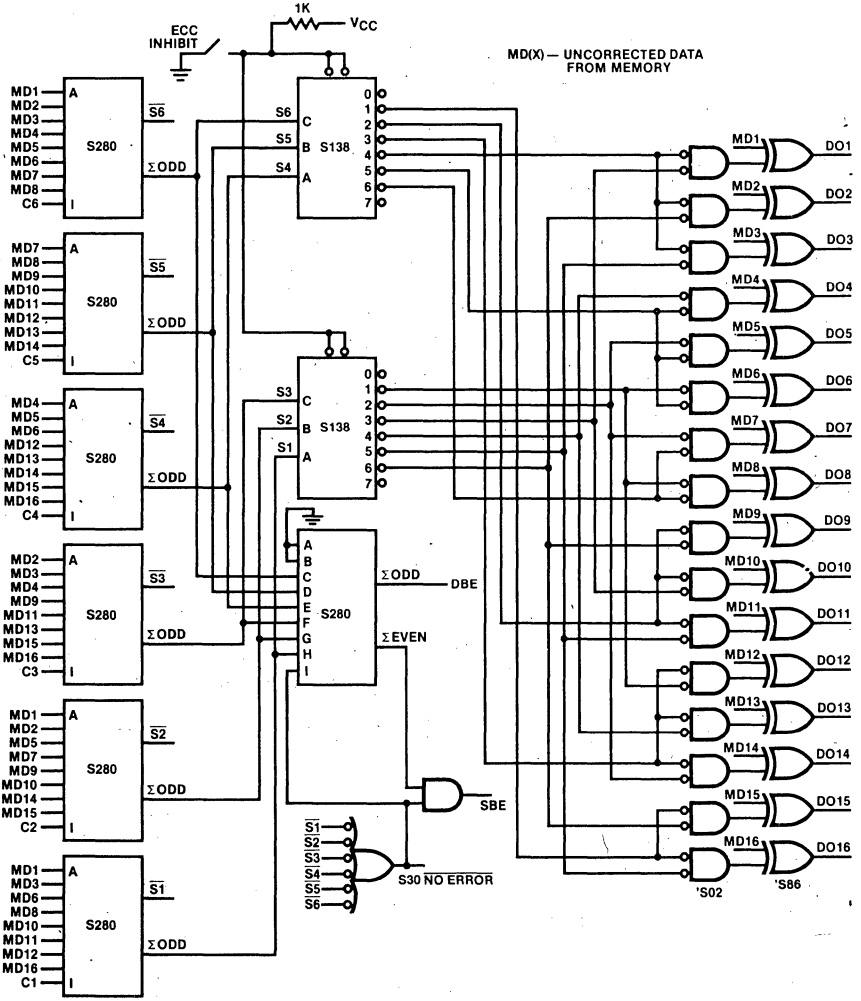

Figure 28. Complete Correction Circuit

Enabling the correction logic, the decoded B(x) signals become "high" to invert the output of the 74S86 exclusive-OR circuits. If the B(x) signals are "low" the output of the correction is the same level as the input. The correction circuit is shown in Figure 29.

Connecting the five circuits as shown in the block diagram of Figure 20 completes the error correction circuitry.

SUMMARY

An unprotected memory has a system MTBF which is approximately equal to the device MTBF divided by the number of devices. Redundancy codes are used to protect memories. While parity is a redundancy code, it only indicates that an error has occurred. A "modified" Hamming code can correct single bit errors and detect double bit errors, truly enhancing the system MTBF.

This report has laid the foundation of ECC basic concepts. Building on this foundation, the next report will address the mathematics for calculating the enhancement factor of ECC in a system environment.

REFERENCES

1. "2107A/2107B N-Channel Silicon Gate MOS 4K RAMs," Reliability Report RR-7, Intel Corporation, September, 1975.
2. "2115/2125 N-Channel Silicon Gate 1K MOS RAMs," Reliability Report RR-14, Intel Corporation, 1976.
3. "2104A N-Channel Silicon Gate 4K Dynamic RAM," Reliability Report RR-15, Intel Corporation, September, 1977.
4. "2116 N-Channel Silicon Gate 16K Dynamic RAM," Reliability Report RR-16, Intel Corporation, August, 1977.
5. R. W. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal,* Vol. 26 (April 1950), pp. 147-160.
6. Len Levine and Ware Meyers, "Semiconductor Memory Reliability with Error Detecting and Correcting Codes," *Computer,* October, 1976, pp. 43-50.
7. —, "Modern Algebra for Coding," *Electro-Technology,* April, 1965, pp. 59-66.
8. William W. Peterson and E. J. Weldon Jr., *Error Correcting Codes,* MIT Press, Cambridge, Mass., 1972.