5.2 ERROR CORRECTION

A semiconductor memory system is subject to errors. These can be categorized as hard failures and soft errors. A **hard failure** is a permanent physical defect so that the memory cell or cells affected cannot reliably store data but become stuck at 0 or 1 or switch erratically between 0 and 1. Hard errors can be caused by harsh environmental abuse, manufacturing defects, and wear. A **soft error** is a random, nondestructive event that alters the contents of one or more memory cells without damaging the memory. Soft errors can be caused by power supply problems or alpha particles. These particles result from radioactive decay and are distressingly common because radioactive nuclei are found in small quantities in nearly all materials. Both hard and soft errors are clearly undesirable, and most modern main memory systems include logic for both detecting and correcting errors.

Figure 5.7 illustrates in general terms how the process is carried out. When data are to be written into memory, a calculation, depicted as a function f, is performed on the data to produce a code. Both the code and the data are stored. Thus, if an M-bit word of data is to be stored and the code is of length K bits, then the actual size of the stored word is $M + K$ bits.

When the previously stored word is read out, the code is used to detect and possibly correct errors. A new set of K code bits is generated from the M data bits and compared with the fetched code bits. The comparison yields one of three results:

- No errors are detected. The fetched data bits are sent out.
- An error is detected, and it is possible to correct the error. The data bits plus **error correction** bits are fed into a corrector, which produces a corrected set of M bits to be sent out.
- An error is detected, but it is not possible to correct it. This condition is reported.

Codes that operate in this fashion are referred to as **error-correcting codes**. A code is characterized by the number of bit errors in a word that it can correct and detect.

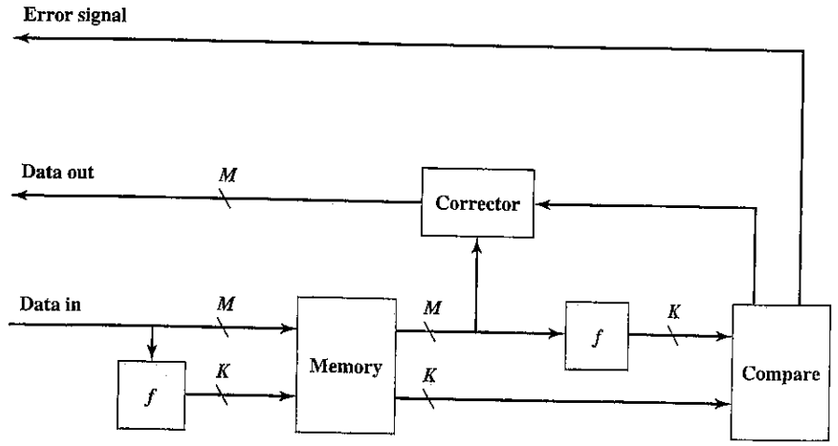

Figure 5.7 Error-Correcting Code Function

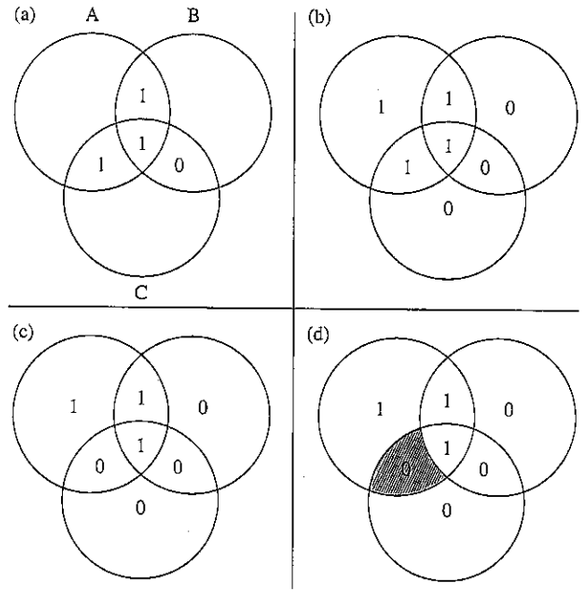

Figure 5.8 Hamming Error-Correcting Code

The simplest of the error-correcting codes is the **Hamming code** devised by Richard Hamming at Bell Laboratories. Figure 5.8 uses Venn diagrams to illustrate the use of this code on 4-bit words ($M = 4$). With three intersecting circles, there are seven compartments. We assign the 4 data bits to the inner compartments (Figure5.8a). The remaining compartments are filled with what are called *parity bits*. Each parity bit is chosen so that the total number of 1s in its circle is even (Figure5.8b). Thus, because circle A includes three data 1s, the parity bit in that circle is set to 1. Now, if an error changes one of the data bits (Figure 5.8c), it is easily found. By checking the parity bits, discrepancies are found in circle A and circle C but not in circle B. Only one of the seven compartments is in A and C but not B. The error can therefore be corrected by changing that bit.

To clarify the concepts involved, we will develop a code that can detect and correct single-bit errors in 8-bit words.

To start, let us determine how long the code must be. Referring to Figure 5.7, the comparison logic receives as input two K-bit values. A bit-by-bit comparison is done by taking the exclusive-OR of the two inputs. The result is called the *syndrome word*. Thus, each bit of the **syndrome** is 0 or 1 according to if there is or is not a match in that bit position for the two inputs.

The syndrome word is therefore K bits wide and has a range between 0 and $2^K - 1$. The value 0 indicates that no error was detected, leaving $2^K - 1$ values to indicate, if there is an error, which bit was in error. Now, because an error could occur on any of the M data bits or K check bits, we must have

$$2^K - 1 \geq M + K$$

Table 5.2 Increase in Word Length with Error Correction

	Single-Error Correction		Single-Error Correction/ Double-Error Detection	
Data Bits	**Check Bits**	**% Increase**	**Check Bits**	**% Increase**
8	4	50	5	62.5
16	5	31.25	6	37.5
32	6	18.75	7	21.875
64	7	10.94	8	12.5
128	8	6.25	9	7.03
256	9	3.52	10	3.91

This inequality gives the number of bits needed to correct a single bit error in a word containing M data bits. For example, for a word of 8 data bits ($M = 8$), we have

- $K = 3: 2^3 - 1 < 8 + 3$
- $K = 4: 2^4 - 1 > 8 + 4$

Thus, eight data bits require four check bits. The first three columns of Table 5.2 lists the number of check bits required for various data word lengths.

For convenience, we would like to generate a 4-bit syndrome for an 8-bit data word with the following characteristics:

- If the syndrome contains all 0s, no error has been detected.
- If the syndrome contains one and only one bit set to 1, then an error has occurred in one of the 4 check bits. No correction is needed.
- If the syndrome contains more than one bit set to 1, then the numerical value of the syndrome indicates the position of the data bit in error. This data bit is inverted for correction.

To achieve these characteristics, the data and check bits are arranged into a 12-bit word as depicted in Figure 5.9. The bit positions are numbered from 1 to 12. Those bit positions whose position numbers are powers of 2 are designated as check bits. The check bits are calculated as follows, where the symbol $\oplus$ designates the exclusive-OR operation:

$$C1 = D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7$$
$$C2 = D1 \oplus D3 \oplus D4 \oplus D6 \oplus D7$$
$$C4 = D2 \oplus D3 \oplus D4 \oplus D8$$
$$C8 = D5 \oplus D6 \oplus D7 \oplus D8$$

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1

Figure 5.9 Layout of Data Bits and Check Bits

Table 5.2 Increase in Word Length with Error Correction

	Single-Error Correction		Single-Error Correction/ Double-Error Detection	
Data Bits	**Check Bits**	**% Increase**	**Check Bits**	**% Increase**
8	4	50	5	62.5
16	5	31.25	6	37.5
32	6	18.75	7	21.875
64	7	10.94	8	12.5
128	8	6.25	9	7.03
256	9	3.52	10	3.91

This inequality gives the number of bits needed to correct a single bit error in a word containing M data bits. For example, for a word of 8 data bits ($M = 8$), we have

- $K = 3: 2^3 - 1 < 8 + 3$
- $K = 4: 2^4 - 1 > 8 + 4$

Thus, eight data bits require four check bits. The first three columns of Table 5.2 lists the number of check bits required for various data word lengths.

For convenience, we would like to generate a 4-bit syndrome for an 8-bit data word with the following characteristics:

- If the syndrome contains all 0s, no error has been detected.
- If the syndrome contains one and only one bit set to 1, then an error has occurred in one of the 4 check bits. No correction is needed.
- If the syndrome contains more than one bit set to 1, then the numerical value of the syndrome indicates the position of the data bit in error. This data bit is inverted for correction.

To achieve these characteristics, the data and check bits are arranged into a 12-bit word as depicted in Figure 5.9. The bit positions are numbered from 1 to 12. Those bit positions whose position numbers are powers of 2 are designated as check bits. The check bits are calculated as follows, where the symbol $\oplus$ designates the exclusive-OR operation:

$$
\begin{aligned}
C1 &= D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7 \\
C2 &= D1 \oplus D3 \oplus D4 \oplus D6 \oplus D7 \\
C4 &= D2 \oplus D3 \oplus D4 \oplus D8 \\
C8 &= D5 \oplus D6 \oplus D7 \oplus D8
\end{aligned}
$$

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1

Figure 5.9 Layout of Data Bits and Check Bits

Each check bit operates on every data bit whose position number contains a 1 in the same bit position as the position number of that check bit. Thus, data bit positions 3, 5, 7, 9, and 11 (D1, D2, D4, D5, D7) all contain a 1 in the least significant bit of their position number as does C1; bit positions 3, 6, 7, 10, and 11 all contain a 1 in the second bit position, as does C2; and so on. Looked at another way, bit position n is checked by those bits C_i such that $\Sigma_i = n$. For example, position 7 is checked by bits in position 4, 2, and 1; and $7 = 4 + 2 + 1$.

Let us verify that this scheme works with an example. Assume that the 8-bit input word is 00111001, with data bit D1 in the rightmost position. The calculations are as follows:

$$
\begin{aligned}
C1 &= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1 \\
C2 &= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1 \\
C4 &= 0 \oplus 0 \oplus 1 \oplus 0 = 1 \\
C8 &= 1 \oplus 1 \oplus 0 \oplus 0 = 0
\end{aligned}
$$

Suppose now that data bit 3 sustains an error and is changed from 0 to 1. When the check bits are recalculated, we have

$$
\begin{aligned}
C1 &= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1 \\
C2 &= 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0 \\
C4 &= 0 \oplus 1 \oplus 1 \oplus 0 = 0 \\
C8 &= 1 \oplus 1 \oplus 0 \oplus 0 = 0
\end{aligned}
$$

When the new check bits are compared with the old check bits, the syndrome word is formed:

	C8	C4	C2	C1
	0	1	1	1
$\oplus$	0	0	0	1
	0	1	1	0

The result is 0110, indicating that bit position 6, which contains data bit 3, is in error.

Figure 5.10 illustrates the preceding calculation. The data and check bits are positioned properly in the 12-bit word. Four of the data bits have a value 1 (shaded in the table), and their bit position values are XORed to produce the Hamming code 0111, which forms the four check digits. The entire block that is stored is

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1
Word stored as	0	0	1	1	0	1	0	0	1	1	1	1
Word fetched as	0	0	1	1	0	1	1	0	1	1	1	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Check bit					0				0		0	1

Figure 5.10 Check Bit Calculation

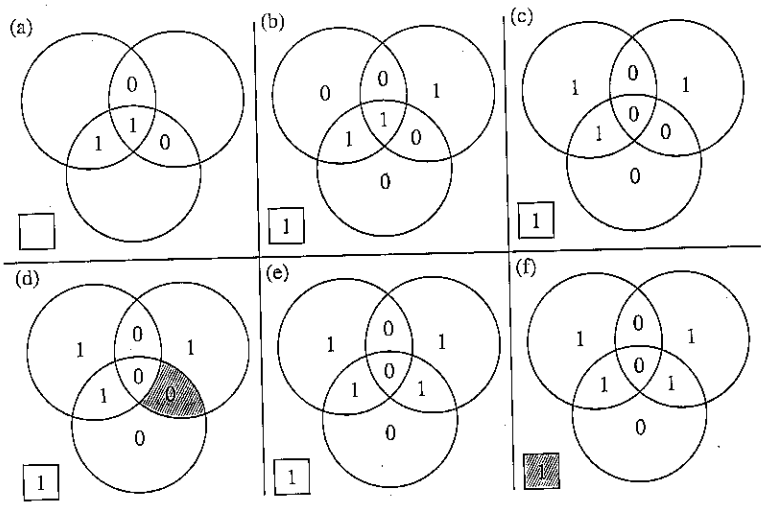

Figure 5.11 Hamming SEC-DEC Code

001101001111. Suppose now that data bit 3, in bit position 6, sustains an error and is changed from 0 to 1. The resulting block is 001101101111, with a Hamming code of 0111. An XOR of the Hamming code and all of the bit position values for nonzero data bits results in 0110. The nonzero result detects an error and indicates that the error is in bit position 6.

The code just described is known as a **single-error-correcting (SEC) code**. More commonly, semiconductor memory is equipped with a **single-error-correcting, double-error-detecting (SEC-DED) code**. As Table 5.2 shows, such codes require one additional bit compared with SEC codes.

Figure 5.11 illustrates how such a code works, again with a 4-bit data word. The sequence shows that if two errors occur (Figure 5.11c), the checking procedure goes astray (d) and worsens the problem by creating a third error (e). To overcome the problem, an eighth bit is added that is set so that the total number of 1s in the diagram is even. The extra parity bit catches the error (f).

An error-correcting code enhances the reliability of the memory at the cost of added complexity. With a 1-bit-per-chip organization, an SEC-DED code is generally considered adequate. For example, the IBM 30xx implementations used an 8-bit SEC-DED code for each 64 bits of data in main memory. Thus, the size of main memory is actually about 12% larger than is apparent to the user. The VAX computers used a 7-bit SEC-DED for each 32 bits of memory, for a 22% overhead. A number of contemporary DRAMs use 9 check bits for each 128 bits of data, for a 7% overhead [SHAR97].

5.3 ADVANCED DRAM ORGANIZATION

As discussed in Chapter 2, one of the most critical system bottlenecks when using high-performance processors is the interface to main internal memory. This interface is the most important pathway in the entire computer system. The basic building block of main memory remains the DRAM chip, as it has for decades; until