Here,
Other example we take "ODD PARITY BIT SCHEME"
Before moving forward we must know some terms.

PARITY BITS:

The extra bit that goes in to ensure that the data given is the data recieved i-e if "a" is pressed "a" is recieved. – Senior's notes.
A simple method of detecting errors with the help of extra bit.

COMMUNICATION PROTOCOLS:

communication protocols are formal descriptions of digital message formats and rules. They are required to exchange messages in telecommunications systems and other systems because they create consistency and universality for the sending and recieving of messages.
communications protocols can cover authentication, error detection and correction, and sign~~in~~aging. They can also describe the syntax, semantics, and synchronization of analog and digital computer. communications. computer networks cannot exists without them.

Dated:

ODD PARITY BIT SCHEME:

In asynchronous communication systems odd parity refers to a parity checking mode. where each set of transmitted bits has an odd number of bits. If the total number of ones in the data plus parity bit is odd number of ones. This scheme is known as odd parity bit scheme.
If data has an odd number of ones already the value of the added parity bit is 0, otherwise it is 1 — senior's Notes.

Now, we are familiar with odd parity scheme we differiantiate it with different computational model.

* MANUAL OPERATION:

Input	0 0 1 1 0 0 1 1 1 0 0 1
Parity	1 1 0 1 1 1 0 1 0 0 0 1.

* MACHINE OPERATION:

Input =	0	0	1	1	0	0	1	1	1	0	0	1
Output =	1	1	0	1	1	1	0	1	0	0	0	1
State =	S_0	S_1	S_2	S_1	S_1	S_1	S_2	S_1	S_2	S_2	S_2	S_1

Dated:

* FINITE STATE TABLE:

state \ Input	0	1
S0	1/S1	0/S2
S1	1/S1	0/S2
S2	0/S2	1/S1

* MATHEMATICAL MODEL:

$\{S, I, O, f(x), g(x), S_{start}\}$

Where;

$S = \{S_0, S_1, S_2\}$, $I = \{1, 0\}$, $O = \{1, 0\}$, $S_{start}\{S_0\}$

for $f(x)$:-

$f(0, S_0) = S_1$ $\quad$ $f(1, S_0) = S_2$

$f(0, S_1) = S_1$ $\quad$ $f(1, S_1) = S_2$

$f(0, S_2) = S_2$ $\quad$ $f(1, S_2) = S_1$

for $g(x)$:-

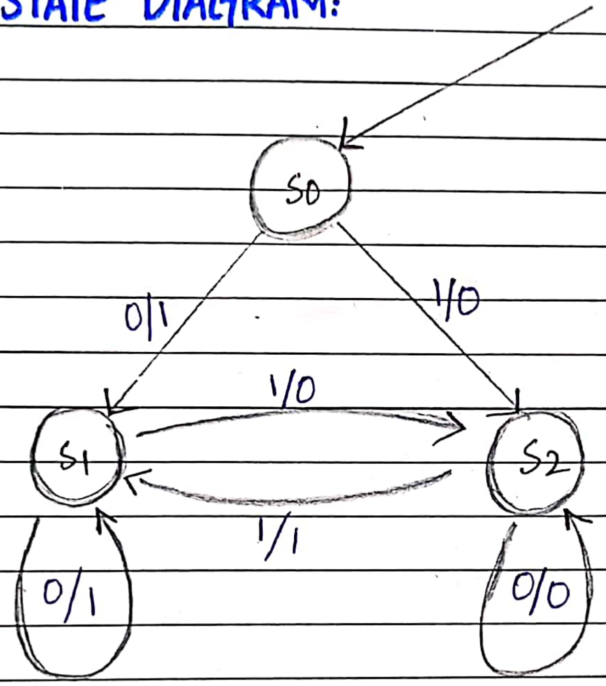
$g(0, S_0) = 1$ $\quad$ $g(1, S_0) = 0$

$g(0, S_1) = 1$ $\quad$ $g(1, S_1) = 0$

$g(0, S_2) = 0$ $\quad$ $g(1, S_2) = 1$

Dated:

* **FINITE STATE DIAGRAM:**


$$\begin{tikzcd}
 & S_0 \arrow[dl, "0/1"'] \arrow[dr, "1/0"] & \\
S_1 \arrow[rr, bend left, "1/0"] \arrow[loop below, "0/1"] & & S_2 \arrow[ll, bend left, "1/1"] \arrow[loop below, "0/0"]
\end{tikzcd}$$

Note:
In this scheme if 2 bit transfered then we cannot detect the error with this scheme this is limitation of this scheme can be correcte with good communication protocol.

Just like odd scheme we also have even parity bit scheme:-

EVEN PARITY BIT SCHEME:
If the total number of ones in the data is even then it is called even parity bit scheme. It is absolute vice versa of odd scheme.
- senior notes.

1.9 Communication Errors

When information is transferred back and forth among the various parts of a computer, or transmitted from the earth to the moon and back, or, for that matter, merely left in storage, a chance exists that the bit pattern ultimately retrieved may not be identical to the original one. Particles of dirt or grease on a magnetic recording surface or a malfunctioning circuit may cause data to be incorrectly recorded or read. Static on a transmission path may corrupt portions of the data. And, in the case of some technologies, normal background radiation can alter patterns stored in a machine's main memory.

To resolve such problems, a variety of encoding techniques have been developed to allow the detection and even the correction of errors. Today, because these techniques are largely built into the internal components of a computer system, they are not apparent to the personnel using the machine. Nonetheless, their presence is important and represents a significant contribution to scientific research. It is fitting, therefore, that we investigate some of these techniques that lie behind the reliability of today's equipment.

Parity Bits

A simple method of detecting errors is based on the principle that if each bit pattern being manipulated has an odd number of 1s and a pattern with an even number of 1s is encountered, an error must have occurred. To use this principle, we need an encoding system in which each pattern contains an odd number of 1s. This is easily obtained by first adding an additional bit, called a **parity bit,** to each pattern in an encoding system already available (perhaps at the high-order end). In each case, we assign the value 1 or 0 to this new bit

so that the entire resulting pattern has an odd number of 1s. Once our encoding system has been modified in this way, a pattern with an even number of 1s indicates that an error has occurred and that the pattern being manipulated is incorrect.

Figure 1.28 demonstrates how parity bits could be added to the ASCII codes for the letters A and F. Note that the code for A becomes 101000001 (parity bit 1) and the ASCII for F becomes 001000110 (parity bit 0). Although the original 8-bit pattern for A has an even number of 1s and the original 8-bit pattern for F has an odd number of 1s, both the 9-bit patterns have an odd number of 1s. If this technique were applied to all the 8-bit ASCII patterns, we would obtain a 9-bit encoding system in which an error would be indicated by any 9-bit pattern with an even number of 1s.

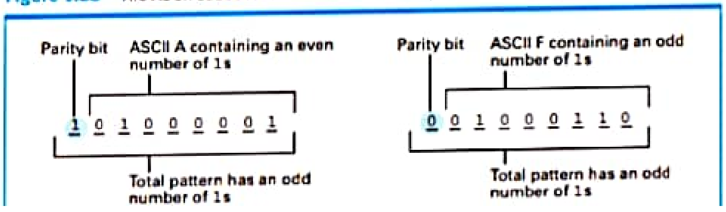
The parity system just described is called **odd parity,** because we designed our system so that each correct pattern contains an odd number of 1s. Another technique is called **even parity.** In an even parity system, each pattern is designed to contain an even number of 1s, and thus an error is signaled by the occurrence of a pattern with an odd number of 1s.

Today it is not unusual to find parity bits being used in a computer's main memory. Although we envision these machines as having memory cells of 8-bit capacity, in reality each has a capacity of 9 bits, 1 bit of which is used as a parity bit. Each time an 8-bit pattern is given to the memory circuitry for storage, the circuitry adds a parity bit and stores the resulting 9-bit pattern. When the pattern is later retrieved, the circuitry checks the parity of the 9-bit pattern. If this does not indicate an error, then the memory removes the parity bit and confidently returns the remaining 8-bit pattern. Otherwise, the memory returns the 8 data bits with a warning that the pattern being returned may not be the same pattern that was originally entrusted to memory.

The straightforward use of parity bits is simple but it has its limitations. If a pattern originally has an odd number of 1s and suffers two errors, it will still have an odd number of 1s, and thus the parity system will not detect the errors. In fact, straightforward applications of parity bits fail to detect any even number of errors within a pattern.

One means of minimizing this problem is sometimes applied to long bit patterns, such as the string of bits recorded in a sector on a magnetic disk. In this case the pattern is accompanied by a collection of parity bits making up a **checkbyte.** Each bit within the checkbyte is a parity bit associated with a particular collection of bits scattered throughout the pattern. For instance, one parity bit may be associated with every eighth bit in the pattern starting

Figure 1.28 The ASCII codes for the letters A and F adjusted for odd parity


Parity bit
ASCII A containing an even number of 1s
1 0 1 0 0 0 0 0 1
Total pattern has an odd number of 1s
Parity bit
ASCII F containing an odd number of 1s
0 0 1 0 0 0 1 1 0
Total pattern has an odd number of 1s

with the first bit, while another may be associated with every eighth bit starting with the second bit. In this manner, a collection of errors concentrated in one area of the original pattern is more likely to be detected, since it will be in the scope of several parity bits. Variations of this checkbyte concept lead to error detection schemes known as **checksums** and **cyclic redundancy checks (CRC).**

Error-Correcting Codes

Although the use of a parity bit allows the detection of an error, it does not provide the information needed to correct the error. Many people are surprised that **error-correcting codes** can be designed so that errors can be not only detected but also corrected. After all, intuition says that we cannot correct errors in a received message unless we already know the information in the message. However, a simple code with such a corrective property is presented in Figure 1.29.

To understand how this code works, we first define the term **Hamming distance,** which is named after R. W. Hamming who pioneered the search for error-correcting codes after becoming frustrated with the lack of reliability of the early relay machines of the 1940s. The hamming distance between two bit patterns is the number of bits in which the patterns differ. For example, the Hamming distance between the patterns representing A and B in the code in Figure 1.29 is four, and the Hamming distance between B and C is three. The important feature of the code in Figure 1.29 is that any two patterns are separated by a Hamming distance of at least three.

If a single bit is modified in a pattern from Figure 1.29, the error can be detected since the result will not be a legal pattern. (We must change at least 3 bits in any pattern before it will look like another legal pattern.) Moreover, we can also figure out what the original pattern was. After all, the modified pattern will be a Hamming distance of only one from its original form but at least two from any of the other legal patterns.

Thus, to decode a message that was originally encoded using Figure 1.29, we simply compare each received pattern with the patterns in the code until we find one that is within a distance of one from the received pattern. We consider this to be the correct symbol for decoding. For example, if we received the bit pattern 010100 and compared this pattern to the patterns in the code, we would obtain

Figure 1.29 An error-correcting code

Symbol	Code
A	000000
B	001111
C	010011
D	011100
E	100110
F	101001
G	110101
H	111010

Figure 1.30 Decoding the pattern 010100 using the code in Figure 1.29

Character	Code	Pattern received	Distance between received pattern and code	
A	0 0 0 0 0 0	0 1 0 1 0 0	2	
B	0 0 1 1 1 1	0 1 0 1 0 0	4	
C	0 1 0 0 1 1	0 1 0 1 0 0	3	
D	0 1 1 1 0 0	0 1 0 1 0 0	1	Smallest distance
E	1 0 0 1 1 0	0 1 0 1 0 0	3	
F	1 0 1 0 0 1	0 1 0 1 0 0	5	
G	1 1 0 1 0 1	0 1 0 1 0 0	2	
H	1 1 1 0 1 0	0 1 0 1 0 0	4	

the table in Figure 1.30. Thus, we would conclude that the character transmitted must have been a D because this is the closest match.

You will observe that using this technique with the code in Figure 1.29 actually allows us to detect up to two errors per pattern and to correct one error. If we designed the code so that each pattern was a Hamming distance of at least five from each of the others, we would be able to detect up to four errors per pattern and correct up to two. Of course, the design of efficient codes associated with large Hamming distances is not a straightforward task. In fact, it constitutes a part of the branch of mathematics called algebraic coding theory, which is a subject within the fields of linear algebra and matrix theory.

Error-correcting techniques are used extensively to increase the reliability of computing equipment. For example, they are often used in high-capacity magnetic disk drives to reduce the possibility that flaws in the magnetic surface will corrupt data. Moreover, a major distinction between the original CD format used for audio disks and the later format used for computer data storage is in the degree of error correction involved. CD-DA format incorporates error-correcting features that reduce the error rate to only one error for two CDs. This is quite adequate for audio recordings, but a company using CDs to supply software to customers would find that flaws in 50 percent of the disks would be intolerable. Thus, additional error-correcting features are employed in CDs used for data storage, reducing the probability of error to one in 20,000 disks.

Questions & Exercises

1. The following bytes were originally encoded using odd parity. In which of them do you know that an error has occurred?

   a. 100101101 b. 100000001 c. 000000000
   d. 111000000 e. 011111111

2. Could errors have occurred in a byte from Question 1 without your knowing it? Explain your answer.

3. How would your answers to Questions 1 and 2 change if you were told that even parity had been used instead of odd?
4. Encode these sentences in ASCII using odd parity by adding a parity bit at the high-order end of each character code:

   a. "Stop!" Cheryl shouted. b. Does 2 + 3 = 5?

5. Using the error-correcting code presented in Figure 1.29, decode the following messages:

   a. 001111 100100 001100 b. 010001 000000 001011
   c. 011010 110110 100000 011100

6. Construct a code for the characters A, B, C, and D using bit patterns of length five so that the Hamming distance between any two patterns is at least three.