13. SELF-CHECKING DESIGN

About This Chapter

In previous chapters we have considered the problems associated with detection of faults by observation of responses to tests. In this chapter we will consider some design procedures that simplify fault diagnosis or detection: specifically design of *self-checking* systems in which faults can be automatically detected by a subcircuit called a *checker*. Such circuits imply the use of coded inputs. After a brief introduction to a few fundamental concepts of coding theory, specific kinds of codes, including parity-check codes, Berger codes, and residue codes, are derived, and self-checking designs of checkers for these codes are presented.

13.1 Basic Concepts

In some cases it may be possible to determine from the outputs of a circuit C whether a certain fault f exists within the circuit without knowing the value of the expected response. This type of testing, which can be performed "on-line" is based on checking some invariant properties of the response. In this case, it is unnecessary to test explicitly for f, and the circuit is said to be self-checking for f. Another circuit, called a *checker*, can be designed to generate an error signal whenever the outputs of C indicate the presence of a fault within C. It is desirable to design circuits, including checkers, to be self-checking to as great an extent as possible (i.e., for as many faults as possible).

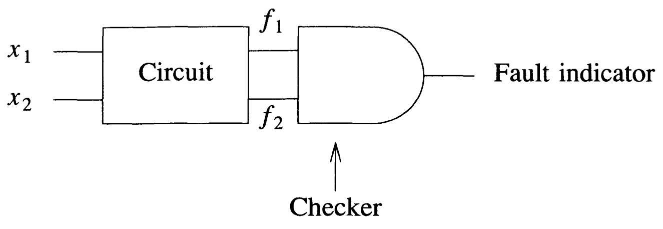
For an arbitrary combinational circuit with p inputs and q outputs, all 2^p input combinations can occur, as can all 2^q possible output combinations. If all possible output combinations can occur, it is impossible to determine whether a fault is present by just observing the outputs of the circuit, assuming no knowledge of the corresponding inputs. However, if only $k < 2^q$ output configurations can occur during normal operation, the occurrence of any of the $2^q - k$ other configurations indicates a malfunction (regardless of the corresponding input). Thus, faults that result in such a an "illegal" output can be detected by a hardware checker.

Example 13.1: Consider a circuit that realizes the combinational functions $f_1(x_1,x_2)$ and $f_2(x_1,x_2)$ described by the truth table of Figure 13.1(a). Note that the output configuration $f_1 = f_2 = 1$ never occurs. Any fault that leads to this configuration can be automatically detected by the checker shown in Figure 13.1(b), which generates a 1-output, indicating an error, if and only if $f_1 = f_2 = 1$. Note that this checker will fail to detect faults that cause an incorrect but "legal" output configuration. □

In a circuit that has output configurations that do not occur during fault-free (i.e., normal) operation, the outputs that do occur are called *(valid) code words,* and the nonoccurring configurations are called *invalid,* or *noncode words.* We will speak of both input codes and output codes in this manner. Considerable work has been done on specifying codes that are useful for detecting and correcting errors.

x_1	x_2	f_1	f_2
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	0

(a)



(b)

Figure 13.1 Automatic fault checking for Example 13.1

13.2 Application of Error-Detecting and Error-Correcting Codes

Codes are commonly classified in terms of their ability to detect or correct classes of errors that affect some fixed number of bits in a word. Thus a code is *e-error detecting* if it can detect any error affecting at most e bits. This implies that any such error does not transform one code word into another code word. Similarly a code is *e-error correcting* if it can correct any error affecting at most e bits. This implies that any two such errors e_1, e_2, affecting words w_1 and w_2, respectively, do not result in the same word.

The *Hamming distance d* of a code is the minimum number of bits in which any two code words differ. The error-detecting and error-correcting capability of a code can be expressed in terms of d as shown in the table of Figure 13.2. (The proof of these results can be found in virtually any book on coding theory).

In general, additional outputs, called *check bits*, must be generated by a circuit in order that its output word constitute a code with useful error capabilities. The *parity-check code* is the simplest such code. For this code, $d = 2$, and the number of check bits is one (independent of the number of output bits in the original circuit). There are two types of parity-check code, even and odd. For an even code the check bit is defined so that the total number of 1-bits is always even; for an odd code, the number is always odd. Consequently any error affecting a single bit causes the output word to have an incorrect number of 1-bits and hence can be detected. Note that in an arbitrary circuit a single fault may cause an error in more than one output bit due to the presence of fanout. Hence care must be taken in designing a circuit in which detection of errors is based on a code with limited error-detecting capabilities.

Example 13.2: For the functions f_1, f_2 of Example 13.1, the check bit y_e for an even parity-check code is defined so that for any input the total number of 1-output bits among f_1, f_2, and y_e is even (Figure 13.3). Thus if $x_1 = 0$, $x_2 = 1$, since $f_1 = 1$ and $f_2 = 0$, then y_e must have value 1 so that the total number of 1-bits among f_1, f_2, and y_e will be even. For an odd parity-check code, the check bit y_o would be as shown in Figure 13.3. Note that y_e and y_o are always complements of each other. □

d	Capability
1	none
2	1-error detection, 0-error correction
3	2 -error detection, 1-error correction
.	
.	
.	
$e + 1$	e-error detection, $\lceil \frac{e}{2} \rceil$ -error correction
.	
.	
.	
$2e + 1$	$2e$-error detection, e-error correction

Figure 13.2 Capability of a code with distance d

x_1	x_2	f_1	f_2	y_e	y_o
0	0	0	1	1	0
0	1	1	0	1	0
1	0	1	0	1	0
1	1	0	0	0	1

Figure 13.3

The check bits required in a code can be thought of as constituting redundancy, since they are only required for error detection. The other bits are called *information bits.* A generalized class of parity-check codes that have greater error-detecting and/or error-correcting capability can be defined. A single-error-correcting code for q information bits requires c check bits where $2^c \geq q + c + 1$. The value of c for various values of q is shown in Figure 13.4.

The c check bits and q information bits form a word with $(c+q)$ bits, $b_{c+q} \cdots b_2 b_1$. In the conventional Hamming code the check bits occur in bits b_{2^i}, $0 \leq i \leq c-1$. The values of these check bits are defined by c parity-check equations. Let p_j be the set of integers whose binary representation has a 1-value in position b_j, (i.e., $p_j = \{I \mid b_j(I) = 1\}$, where $b_j(n)$ denotes the value of the j-th bit (from the right) of a binary integer n. Then the values of the check bits are defined by the c parity-check equations of the form

$$\sum_{k \in P_i} b_k = 0 \qquad i = 1, \ldots, c$$

q	c
1	2
4	3
11	4
26	5
57	6
120	7

Figure 13.4 Values of q and c for single-error-correcting parity-check codes

where the sum is modulo 2. An error in bit b_j will result in incorrect parity for exactly those equations for which j is in p_i. Thus the erroneous bit can be computed from these c parity-check equations.

Example 13.3: Let $q = 4$. Then $2^c \geq q + c + 1 = 5 + c$, and hence $c = 3$. The single-error-correcting code defines a 7-bit word with check bits b_1, b_2, and b_4. The value of the check bits for any given value of the information bits can be computed from the three parity-check equations.

$$b_1 \oplus b_3 \oplus b_5 \oplus b_7 = 0 \quad \text{defined by } p_1 \tag{1}$$

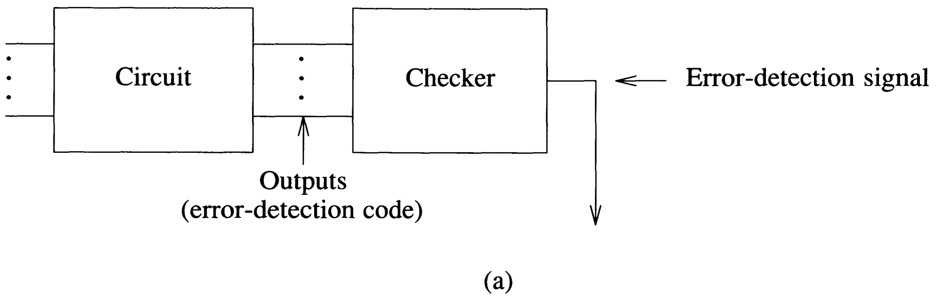
$$b_2 \oplus b_3 \oplus b_6 \oplus b_7 = 0 \quad \text{defined by } p_2 \tag{2}$$

$$b_4 \oplus b_5 \oplus b_6 \oplus b_7 = 0 \quad \text{defined by } p_3 \tag{3}$$

Thus if the information bits have the values $b_3 = 1$, $b_5 = 0$, $b_6 = 0$, and $b_7 = 1$, then the check bits will have the values $b_1 = 0$ (defined by substitution in equation 1 above), $b_2 = 0$ (from equation 2) and $b_4 = 1$ (from equation 3), and thus the encoded word is 1 0 0 1 1 0 0. If an error occurs in bit position b_4, the word becomes 1 0 0 0 1 0 0. If we recompute the three parity-check equations from this word, we derive the values 1 0 0 from equations 3, 2, and 1 respectively. The binary number formed from these three equations is 100 (binary 4) thus indicating an error in bit b_4. Thus the single-bit error can be corrected. □

The use of codes enables hardware to be designed so that certain classes of errors in the circuit can be automatically detected or corrected through the use of a hardware checker, as shown in Figure 13.5. Note that errors in the checker may not be detected. Many codes have been developed that can be used in the design of such self-checking circuits. The type of code to be used may vary depending on the type of circuit. For data-transmission busses, a parity-check code may be adequate. For other types of functions, however, we may wish to use a code for which the check bits of the result can be determined from the check bits of the operands.

Consider the design of an arithmetic adder using an even parity-check code. For the two additions illustrated in Figure 13.6, in both cases the operand check bits are 0 and 1 respectively, but the check bit of the sum $A + B_1$ is 0 and the check bit of the sum



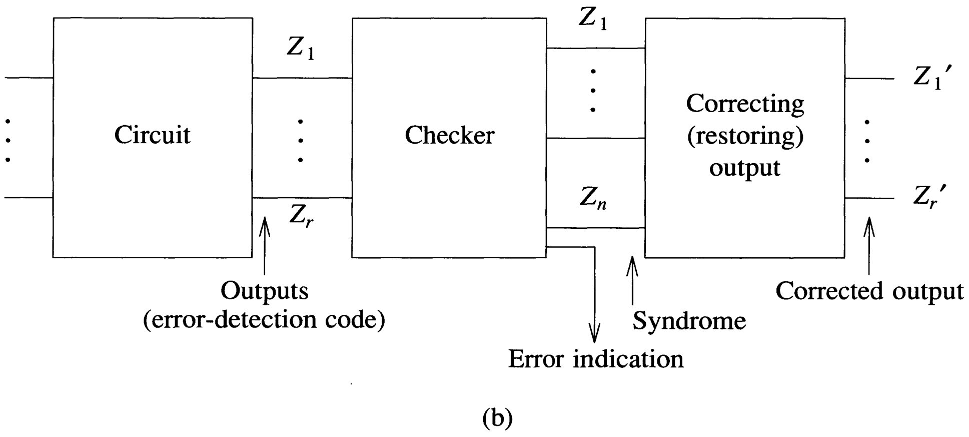


Figure 13.5 (a) Error-detecting circuit (b) Error-correcting circuit

$A + B_2$ is 1. Therefore it would be necessary to recompute the value of the check bit after each addition, and errors in the addition itself would not be detected.

Another class of codes, called *residue codes*, has the desirable property that for the arithmetic operations of addition, subtraction, and multiplication, the check bits of the result can be determined directly from the check bits of the operands. This property is called *independent checking*. Several different types of residue code have been formulated. We will consider only one such class. In this code the rightmost p bits are the check bits. The check bits define a binary number C, and the information bits define another number N. The values of the check bits are defined so that $C = (N)$ modulo m, where m is a parameter called the *residue* of the code, and the number of check bits is $p = \lceil \log_2 m \rceil$.

Example 13.4: Consider the derivation of the check bits for a residue code with three information bits I_2, I_1, I_0 and $m = 3$. Since $\lceil \log_2 m \rceil = 2$, check bits C_1 and C_0 are

$$\begin{array}{rl} A & = 0\,0\,0\,1 \\ \underline{B_1} & \underline{= 0\,1\,0\,1} \\ A + B_1 & = 0\,1\,1\,0 \end{array} \qquad \begin{array}{rl} A & = 0\,0\,0\,1 \\ \underline{B_2} & \underline{= 0\,0\,1\,1} \\ A + B_2 & = 0\,1\,0\,0 \end{array}$$

$$C(A) = 1,\ C(B_1) = C(B_2) = 0,\ C(A + B_1) = 0,\ C(A + B_2) = 1$$
$C(X)$ is the check bit of X

Figure 13.6 Parity not preserved by addition

required. Their values are defined so that the binary number $C = C_1C_0 =$ (N) modulo 3, where N is the binary number $I_2I_1I_0$, as shown in Figure 13.7. □

I_2	I_1	I_0	N	C	C_1	C_0
0	0	0	0	0	0	0
0	0	1	1	1	0	1
0	1	0	2	2	1	0
0	1	1	3	0	0	0
1	0	0	4	1	0	1
1	0	1	5	2	1	0
1	1	0	6	0	0	0
1	1	1	7	1	0	1

Figure 13.7 A 3-bit residue code with $m = 3$

We will now prove that for this type of residue code, the check bits of the sum (product) of a set of operands is equal to the sum (product) of the check bits of the operands.

Theorem 13.1: Let $\{a_i\}$ be a set of operands. The check bits of the sum are given by $(\sum a_i) \bmod m$, and the check bits of the product by $(\Pi\ a_i) \bmod m$. Then

a. The check bits of the sum are equal to the sum of the check bits of the operands modulo m.

b. The check bits of the product are equal to the product of the check bits of the operands modulo m.

Proof

a. Let $a_i = k_{i1}m + k_{i2}$ where $0 \le k_{i2} < m$. Then $(\sum a_i) \bmod m = (\sum (k_{i1}\ m + k_{i2})) \bmod m = (\sum k_{i2}) \bmod m = (\sum (a_i) \bmod m) \bmod m$ (since k_{i2} is the residue of a_i).

b. The proof of part b is similar to part a and is left as an exercise. □

Example 13.5

a. Consider the addition shown in Figure 13.8(a) using residue codes with $m = 3$. The information bits of the sum represent the number 6 and (6) mod 3 = 0. The sum of the check bits mod 3 is also 0. Thus the sum of the check bits modulo m is equal to the check bits of the sum.

b. Consider the multiplication shown in Figure 13.8(b) using residue codes with $m = 3$. The product of the check bits modulo m is equal to 2. The information bits of the product represent the number 8, and (8) mod 3 = 2. Thus the product of the check bits modulo m is equal to the check bits of the product. □

Information bits				Check bits	
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	0	1	1

(a)

Information bits				Check bits	
0	0	1	0	1	0
0	1	0	0	0	1
1	0	0	0	1	0

(b)

Figure 13.8

The use of residue codes to check addition is illustrated by the system of Figure 13.9, which computes the check bits, *C(A+B)*, of the sum $A + B$ and compares the result with the modulo m sum of the check bits, *(C(A)+C(B))* mod m. This circuit will detect an error that causes these two computations to be unequal. Let us now consider the class of errors that can be detected by this class of residue codes.

If a residue code defines code words with $s = p + q$ bits, then an error pattern E can be defined as an s-bit binary vector in which $e_i = 1$ if bit i is in error and $e_i = 0$ if bit i is correct. For a number N with check bits C where $C = (N)$ mod m, such an error may change N to N' and/or C to C'. Such an error will be detected, provided $C' \neq N'$ mod m. Let us first consider single-bit errors.

Theorem 13.2: In a residue code with m odd, all single-bit errors are detected.

Proof: A single-bit error can affect either the value of C or N but not both. We therefore consider two cases.

Case 1: Assume the single erroneous bit is an information bit. If bit i of the information segment is in error, then $N' = N \pm 2^i$ and $C' = C$. Then N'mod m = $(N \pm 2^i)$ mod m = N mod $m \pm (2^i)$ mod m = $C \pm (2^i)$ mod m. For m odd, (2^i) mod $m \neq 0$ and hence the error is detected.

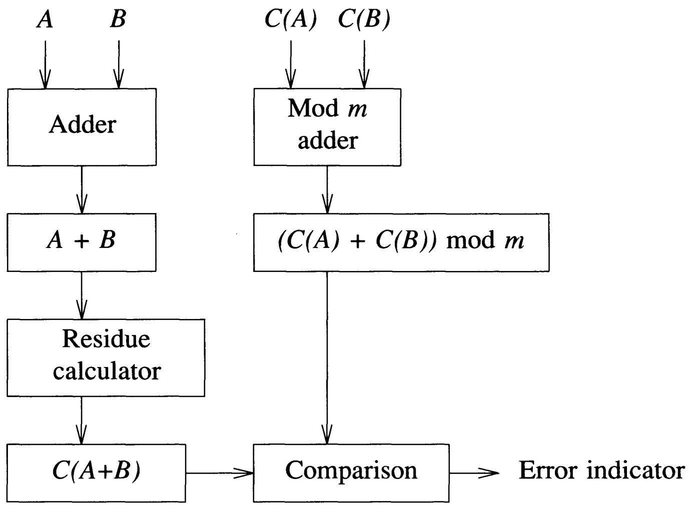


Figure 13.9

Case 2: Assume the single erroneous bit is a check bit. If bit i of the check segment is in error, then $C' = (C \pm 2^i) \bmod m \neq C$ for m odd. Hence all single-bit errors are detected. □

If m is odd, some single-bit errors may be indistinguishable. If m is even, some single-bit errors may not even be detected.

Example 13.6

a. Let $m = 3$. Consider the code word 11000 in which the rightmost two bits are check bits. For the single-bit error affecting bit 4 (from the right), the erroneous word is 10000, which is not a code word, since $N = 4$, $C = 0$, and $N \bmod 3 = 1 \neq 0$. Therefore this error will be detected. However, the same erroneous word could result from the code word 00000 due to a single-bit error pattern affecting bit 5 (from the right). Thus this code can not distinguish these two single-bit errors and hence the code is not single-error correcting.

b. Let $m = 2$. Consider the code word 1100 with the rightmost bit as the only check bit. A single-bit error affecting bit 3 results in the word 1000. Since $I' = 4$ and $C' = 0$ and $I' \bmod 2 = 0 = C'$, this is a valid code word, and the single-bit error is not detected. □

Thus the parameter m must be odd for the code to detect single errors. As m increases, the number of check bits required increases, and the error-detecting and error-correcting capabilities of the code vary with the specific value of m in a complex manner that will not be considered herein [Sellers *et al.* 1968].

13.3 Multiple-Bit Errors

The error-detecting capabilities of codes with respect to multiple-bit errors are also of interest. For a parity-check code, all errors that affect an odd number of bits will be detected, whereas even bit errors will not be. It should be noted, however, that in many technologies the most probable multiple-bit errors are not random but have some special properties associated with them such as *unidirectional errors* (all erroneous bits have the same value) and *adjacent-bit errors* (all bits affected by an error are contiguous). Some common codes are useful for detecting such multiple-bit errors.

The k/n (k-out-of-n) code consists of n-bit words in which each code word has exactly k 1-bits. These codes will detect all unidirectional errors, since a unidirectional error will either result in an increase or a decrease in the number of 1-bits in a word. This code is a *nonseparable code*, in which it is not possible to classify bits as check bits or information bits. When such codes are used, the circuitry involved must perform computations on the entire word rather than just the information bits of a word.

The *Berger codes* are separable codes. For I information bits, the number of check bits required is $C = \lceil \log_2(I + 1) \rceil$, forming a word of $n = I + C$ bits. The C check bits define a binary number corresponding to the Boolean complement of the number of information bits with value 1. Thus with three information bits, two check bits are required. If the information bits are 110, the check bits define the Boolean complement of 2 (since there are two information bits with value 1) and hence the check bits are 01. For $I = 3$ and $C = 2$, the complete code is shown in Figure 13.10.

I_1	I_2	I_3	C_1	C_0
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	0

Figure 13.10 Berger code for I=3 and C=2

Berger codes detect all unidirectional errors (Problem 13.5). Among separable codes that detect all unidirectional errors, Berger codes are optimal in requiring fewer check bits for I information bits. For large values of r, however, m-out-of-n codes require fewer bits to achieve r valid code words.

In the *modified residue code*, $m - 1$ check bits are defined so that the total number of 1-bits in a code word is a multiple of the parameter m. This code, which is a generalization of the parity-check code, detects all unidirectional errors affecting fewer than m bits. For $m = 3$, if the information bits have the value 110, the check bits have the value 01 or 10.

Given a set of kn bits arranged as k n-bit words, adding a parity-check bit to each word results in k $(n + 1)$-bit words in which any single-bit error can be detected as well as any multiple-bit error so long as no word has more than one erroneous bit. Alternatively, we can define one n-bit check word C, where for all i, $1 \leq i \leq n$, the i-th bit of C is a parity check over the i-th bit of all k words. This results in $(k + 1)$ n-bit words in which all single-bit errors are detected, all multiple-bit errors within one word are detected, and all multiple-bit errors affecting different bits of different words are detected. This technique can also be used with error-correcting codes to obtain multiple-bit error correction.

13.4 Checking Circuits and Self-Checking

The use of on-line testing (i.e., checking circuits in conjunction with coded outputs) to detect faults has the following advantages over off-line testing.

1. Intermittent faults are detected.
2. Output errors are detected immediately upon occurrence, thus preventing possible corruption of data.
3. The distribution of checkers throughout a digital system provides a good measure of the location of a fault by the location of the checker at which the error is detected.
4. The software diagnostic program is eliminated (or at least simplified substantially).

Checking by means of hardware can be combined with a general reliability strategy called *rollback*, in which the status of a system is periodically saved, and upon detection of an error the system is reconfigured to its most recent previous valid saved condition and the subsequent sequence of inputs is repeated. Repeated failures cause an interrupt. (This strategy is effective for intermittent failures of short duration).

There are also several disadvantages associated with on-line testing by means of hardware (i.e., checking circuits).

1. More hardware is required, including a hardware checker.
2. The additional hardware must be checked or tested (*checking the checker problem*). In general, some faults cannot be automatically detected.

Thus the use of hardware for testing raises the problem of how to handle faults in the checking unit. This has led to the study of *totally self-checking circuits* (*and checkers*) [Carter and Schneider 1968]. A circuit is *fault secure* for a set of faults F, if for any fault in F, and any valid (code) input, the output is a noncode word or the correct code word, never an invalid code word. A circuit is *self-testing* for a set of faults F if for any fault f in F, there exists an valid (code) input that detects f. A *totally self-checking circuit* is both fault secure and self-testing for all faults (in the set of faults of interest). Fault secureness insures that the circuit is operating properly if the output is a code word. Self-testing insures that it is possible for any fault to be detected during normal operation. Hence a totally self-checking circuit need not have a special diagnostic program but can be completely tested by means of hardware*. One intrinsic difficulty associated with such a method is the possibility of faults within the checker. For the system to be totally self-checking, both the circuit and the checker must also be self-testing and fault secure. The part of the circuit containing faults that cannot be detected in this manner must be tested by other means and is sometimes referred to as *hardcore*. It is desirable to design the checking circuit so that the hardcore is localized and/or minimized.

13.5 Self-Checking Checkers

Most of the research of self-checking has been related to the design of self-checking checkers. A checker (for a specific code) is usually defined as a single-output circuit which assumes the value 0 for any input corresponding to a code word and assumes the value 1 for any input corresponding to a noncode word. This output is used as an indication of error. Since the output is 0 for all code inputs, a checker cannot be designed in this manner so as to be self-testing for a *s-a-*0 fault on its output, since this fault can only be detected by applying an input corresponding to a noncode word, and this will never occur during normal operation. Signals that have only one possible value during normal operation are called *passive*. One approach to the design of self-checking checkers is to replace all passive signals by a pair of signals each of which, during normal operation, could assume both logical values, 0 and 1. Thus a checker could be designed, as shown in Figure 13.11, where the two outputs of C_1 take the values (0,1) or (1,0) for code inputs and (0,0) or (1,1) for noncode inputs. If we assume that a single signal is necessary to indicate error, additional logic is required to generate this single signal z from the two nonpassive signals. In Figure 13.11 the subcircuit C_2 generates z as the exclusive-OR of the two signals generated by C_1. However, since z is a passive signal, C_2 is always non-self-testing. Thus the hardcore (i.e., the logic that is not self-testable) has actually not been eliminated but has been localized and minimized.

The concept of realizing a passive function as a pair of nonpassive functions can be formalized. Consider a binary function f with values 0,1, which are mapped into two binary functions z_1, z_2 in which (z_1,z_2) = (0,0) or (1,1) corresponds to $f = 1$ and (z_1,z_2) = (0,1) or (1,0) corresponds to $f = 0$. The set of functions (z_1,z_2) is called a *morphic function* corresponding to f. It is possible to design some morphic functions so as to be totally self-checking.

* If a circuit is totally self-testing for a set of faults F, then it is possible for any fault in F to be detected during normal operation by hardware checking. Of significance is the probability that the fault will be detected within t units of time from its occurrence. The expected value of t has been referred to as the *error latency* of the circuit [Shedletsky and McCluskey 1975]. If F is the set of single stuck faults, hardware detection is only useful if the error latency of the circuit is much less than the expected time for a second fault to occur so that a multiple fault does not exist.

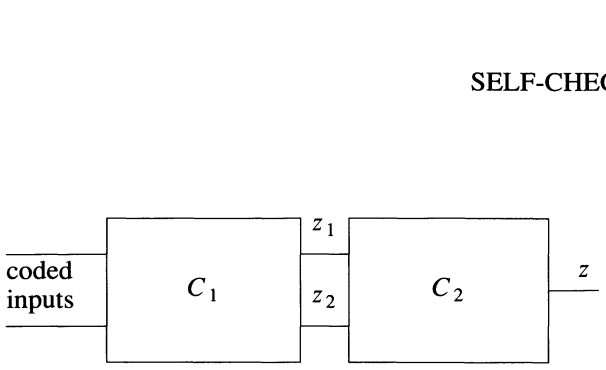


Figure 13.11

13.6 Parity-Check Function

Consider the 3-bit parity-check function represented by the Karnaugh map of Figure 13.12. The corresponding morphic function is shown in Figure 13.13, where each a_i can be 0 or 1.

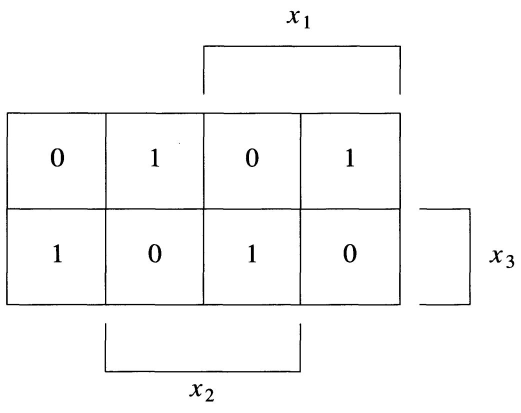


Figure 13.12 Karnaugh map of 3-input parity-check function

In general for an n-variable function there are 2^n variables a_i defined in the corresponding morphic function. The question remains as to whether these values of a_i can be selected so that both z_1 and z_2 are self-testing and hence self-checking. For the parity-check function of Figure 13.13, the choice $a_i = 0$, $i \leq 3$, $a_i = 1$, $i \geq 4$ leads to the circuit shown in Figure 13.14(a), which can be verified to be totally self-checking, where the code inputs correspond to all inputs with an even number of 1-bits. This type of realization can be generalized for an n-input parity-check function. In the totally self-checking circuit, the set of n variables is partitioned into two disjoint sets A_i, B_i, each with at least one variable, and parity-check functions of these two variable sets are realized. The resultant circuit, shown in Figure 13.14(b), is totally self-checking. (An inverter on one of the outputs is required to produce outputs of (0,1) or (1,0) for even parity inputs.) This morphic function realization can be shown

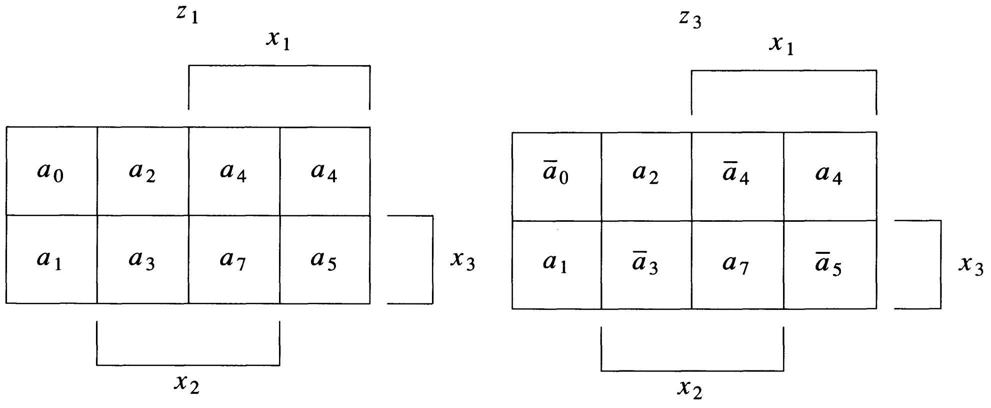


Figure 13.13 Karnaugh maps representing morphic function corresponding to 3-input parity-check function

to be a minor modification of a normal checker realization shown in Figure 13.15. Thus the hardcore of the normal circuit has now been placed with the decision logic (which interprets the error-identification signals (0,0) and (1,1) and produces a passive error signal). The hardcore has been *localized* rather than eliminated.

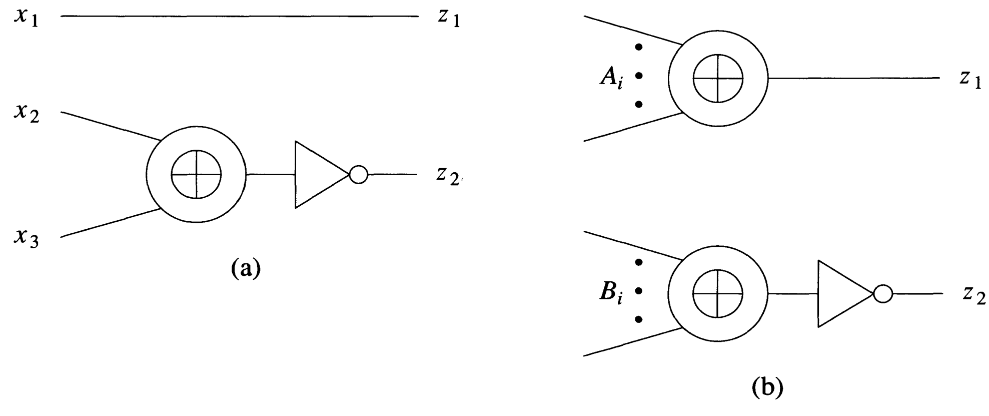


Figure 13.14 (a) Self-checking 3-bit parity checker (b) General self-checking parity checker

13.7 Totally Self-Checking m/n Code Checkers

A totally self-checking *k*/2*k* code checker can also be designed [Anderson and Metze 1973]. The checker has two outputs, *f* and *g*. The 2*k* inputs are partitioned into two

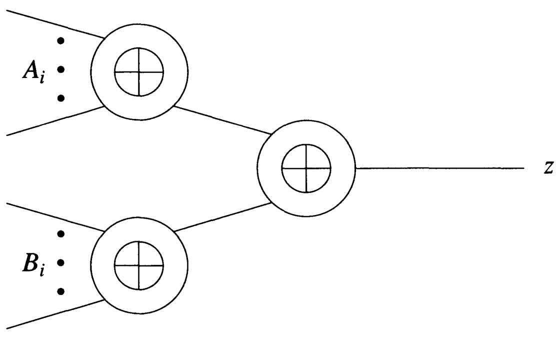


Figure 13.15 Parity-check circuit

disjoint sets of k inputs each, x_A and x_B. The function f is defined to have the value 1 if and only if i or more of the variables in x_A have the value 1 and $k - i$ or more of the variables in x_B have the value 1, for i odd. Similarly g is defined to have the value 1 if i or more variables in x_A and $k - i$ or more variables in x_B have the value 1, for i even. For code inputs (i.e., exactly k of the $2k$ variables have the value 1) then $f = 1$ and $g = 0$ or $g = 1$ and $f = 0$, while for noncode inputs with fewer than k 1-bits, $f = 0$ and $g = 0$, and for noncode inputs with more than k 1-bits, $f = g = 1$. The general form of the circuit is as shown in Figure 13.16, where T_{Aodd} has outputs for each odd value of i, and the ith output assumes the value 1 if i or more inputs assume the value 1. The subcircuits T_{Aeven}, T_{Bodd}, and T_{Beven} are similarly defined. It can be shown that this circuit is both self-checking and fault secure, and hence totally self-checking, for all single stuck faults. Other realizations of this checker have a large number of faults that cannot be tested by code word inputs.

The self-checking checker for $k/2k$ codes can also be used to realize a general k/n checker where $n \neq 2k$ [Marouf and Friedman 1978]. The general form of the circuit realization is shown in Figure 13.17. The AND array consists of a single level of k-input AND gates, one for each subset of k variables, which generate all possible products of k of the n variables. Thus, for a code word input, the output of this array will have exactly one 1-signal, while for noncode inputs with more than k 1-bits, two or more of the output signals will have the value 1, and for noncode inputs with fewer than k 1-bits, none of the output signals will have the value 1. The OR array consists of a single level of OR gates, which converts the $1/\begin{bmatrix} n \\ k \end{bmatrix}$ code on the z signals to a $p/2p$ code, where p must satisfy the constraint

$$2p \leq \begin{bmatrix} n \\ k \end{bmatrix} \leq \begin{bmatrix} 2p \\ p \end{bmatrix}.$$

The k/n checker design represented in Figure 13.17 might be considered inefficient, since it requires a great amount of hardware and *all* valid code words as tests in order

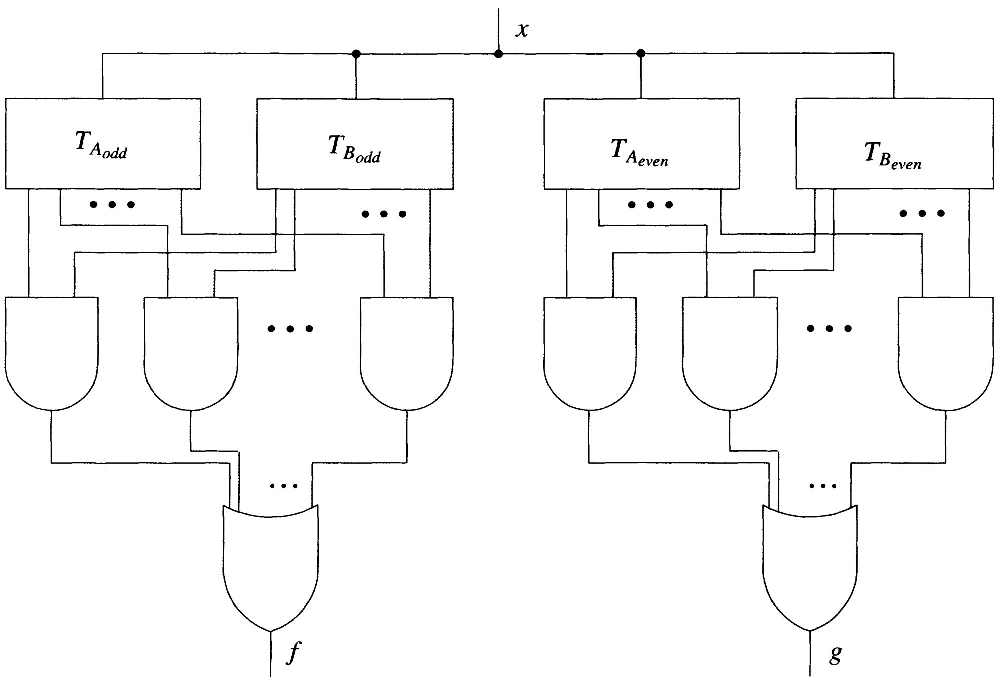


Figure 13.16 Totally self-checking *k*/2*k* checker

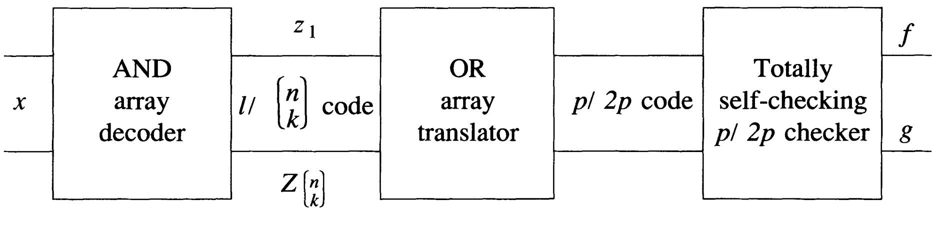


Figure 13.17 A *k*/*n* checker design

to detect all single stuck faults. Subsequent research was related to attempts to finding improved realizations that were totally self-checking and required either a simpler circuit design and/or fewer tests to detect all single stuck faults. A more efficient checker can be designed, as shown in Figure 13.18, consisting of three subcircuits, C_1, C_2, and C_3. The circuit C_1 has n inputs and Z outputs where $Z = 4$ for $m/(2m+1)$ codes, $Z = 5$ for $2/n$ codes, and $Z = 6$ for any other m/n code. In normal operation, C_1 receives m/n code words as inputs and produces a $1/Z$ code on its outputs. The circuit

C_2 translates the 1/Z code on its inputs to a 2/4 code on its outputs, and C_3 is a 2/4 code checker. All three of the subcircuits are designed as totally self-checking circuits. This type of realization can be shown in general to require a much smaller set of tests to detect all single stuck faults compared to the design shown in Figure 13.16.

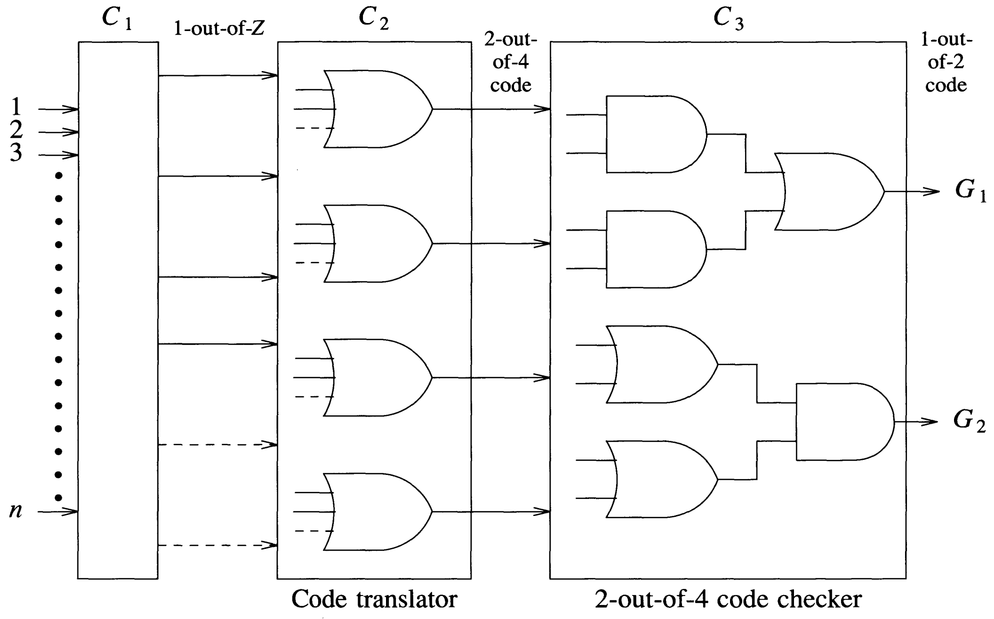


Figure 13.18 m-out-of-n code checker

13.8 Totally Self-Checking Equality Checkers

Totally self-checking equality checkers have also been designed [Anderson 1971]. These circuits test two k-bit words to determine if they are identical. For $k = 2$, the circuit of Figure 13.19 is a totally self-checking equality checker for the two words (a_1,a_2) and (b_1,b_2). This can be generalized for arbitrary values of k. Such an equality checker can be used as the basis for checkers for various operations and codes as illustrated previously in the design of checkers for addition using residue codes.

13.9 Self-Checking Berger Code Checkers

A general structure for a totally self-checking checker for separable codes is shown in Figure 13.20 [Marouf and Friedman 1978b]. The circuit N_1 generates check bits from the information bits. The equality checker, N_2, compares the check bits of the input word with the check bits generated by N_1.

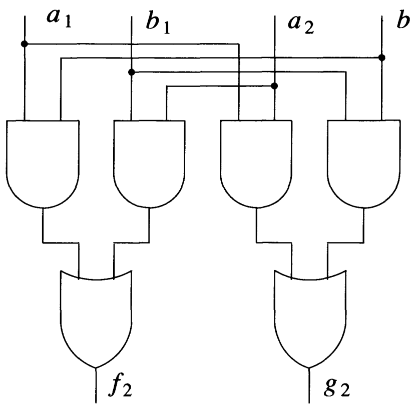


Figure 13.19 Totally self-checking equality checker

For the Berger code, the generator circuit N_1 can be constructed from full-adder modules, as shown in Figure 13.21, for the case $I = 7$, $C = 3$. A set of four test vectors is shown.

This type of design is easily generalized for the case where I, the number of information bits, is $2^k - 1$. The generator circuit, N_1, is easily tested with eight inputs sufficient to detect all single stuck faults within N_1 as well as all multiple faults occurring within a single full-adder module within N_1.

For the case where $I = 2^k - 1$, all of the 2^k possible combinations of check-bit values occur in some code words. Therefore a two-level realization of the equality checker, N_2, will be totally self-checking. However, such a realization is very inefficient both in terms of the number of gates required and the number of tests required to detect all single stuck faults. A much more efficient realization is a tree circuit formed as an interconnection of one-bit comparator modules. Although somewhat more complex, relatively efficient designs of both N_1 and N_2 can be derived for the cases where $I \neq 2^k - 1$ [Marouf and Friedman 1978b].

A significant amount of research has been directed toward the development of more efficient totally self-checking circuits for many of the more common codes, some of which have been considered herein [Jha 1989, Gastanis and Halatsis 1983, Nikolos *et al.* 1988], as well as to specific technologies such as PLAs [Mak *et al.* 1982]. Little has been done, however, in the way of a general theory for the design of totally self-checking circuits for arbitrary combinational or sequential functions.

13.10 Toward a General Theory of Self-Checking Combinational Circuits

We will now consider the general problem of self-checking circuit design for arbitrary functions. It is easily shown that the inputs of a self-checking circuit must be coded in a distance d code where d is at least 2 in order for the circuit to be fault secure with

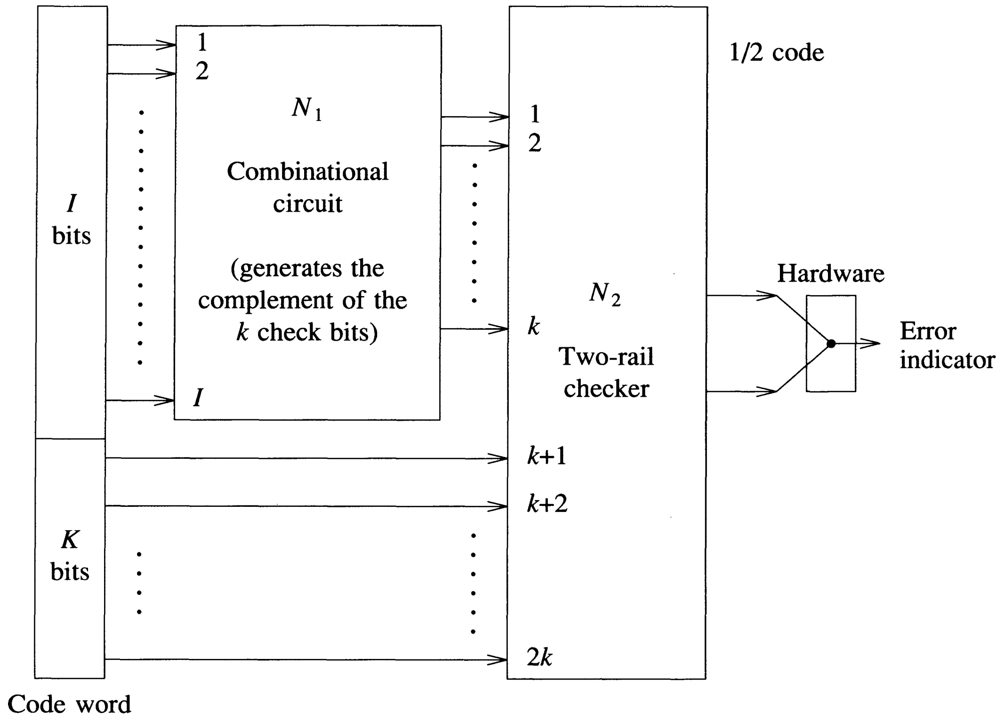


Figure 13.20 Totally self-checking checker for separable codes

respect to single stuck input faults. The outputs must be similarly coded to be fault secure with respect to single stuck output faults. If it is assumed that there are no input faults and the inputs are uncoded, then self-testing is automatically achieved and the circuit must only be designed to be fault secure. If the outputs of the circuit are defined to be of even parity, they define a distance 2 code. However, it is possible for a single fault to affect two output bits, as illustrated by the fault *a* *s-a*-0 with input $(x_1,x_2) = (0,1)$ in the circuit of Figure 13.22(b). In this case the circuit is not fault secure, since the normal output $(f_1,f_2,f_3) = (1,1,0)$ and the faulty output $(f_1,f_2,f_3) = (0,0,0)$ are both valid possible circuit outputs for some input value. The circuit can be redesigned to prevent such an error pattern, as shown in Figure 13.22(c), wherein duplicated signals have been generated to ensure odd parity errors. Note that each of the gates G_1 and G_2 of Figure 13.22(b) have been replaced by two gates, and it must be further assumed that no input fault can affect both of these gates. Thus if no input faults are considered, it is possible to design fault-secure realizations of arbitrary combinational functions.

As we have seen previously, if we assume that a single passive signal is required for error indication design of totally self-checking circuits cannot completely eliminate the hardcore. Referring to Figure 13.11, if C_1 is to be totally self-checking and C_2 is to represent the hardcore, the question arises whether the logic in C_1 can be substantially

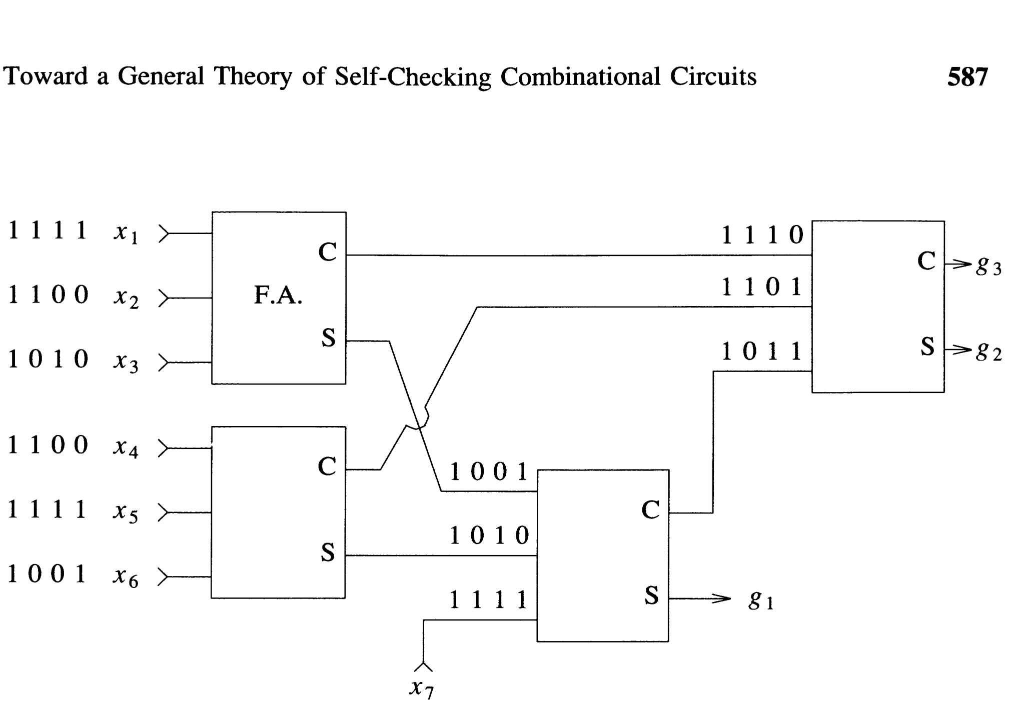


Figure 13.21 Self-checking Berger code checker, I=7, C=3

simplified if C_2 is made somewhat more complex. It appears likely that a general theory of self-checking circuits would not be restricted to two-output morphic functions and would explicitly recognize the trade-off between the hardcore and the degree of self-checking. In this connection a class of circuits that are totally fault secure but only partially self-testing, and a class of circuits that are totally self-testing but are only fault secure for a subset of all possible inputs may be considered. However, no general design procedures for such circuits have yet been developed.

13.11 Self-Checking Sequential Circuits

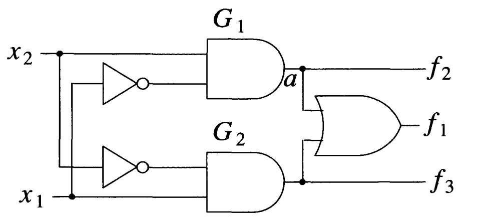
The use of codes and the concepts of self-checking design are also applicable to the design of self-checking sequential circuits. Consider the sequential-circuit model of Figure 13.23, in which the outputs and state variables are coded. Self-checking can be obtained by designing the combinational logic so that

1. For any fault internal to C* and for any input, either the output and the next state are correct or the output and/or the next state is a noncode word.
2. For any state corresponding to a noncode word resulting from a fault f in C, and for any input, the next state generated by C with fault f is a noncode word and the output is also a noncode word.

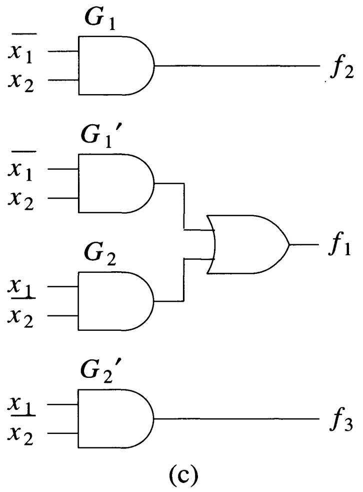
* If input faults are also to be considered, the inputs must be coded.

x_1	x_2	f_1	f_2	f_3
0	0	0	0	0
0	1	1	1	0
1	0	1	0	1
1	1	0	0	0

(a)



(b)



(c)

Figure 13.22 Redesign of a circuit to restrict effect of single stuck faults

The net result from these conditions is that C serves as a checker on the coded state variables.

A conceptually similar design procedure is shown in Figure 13.24. The outputs of the sequential circuit C_1 are encoded in a k/n code, which is input to a k/n checker. The checker generates a signal R_s, which is required for the next clock pulse (or in the case of asynchronous circuits, for the next input signal) to be generated. Thus for many faults the system will stop as soon as an erroneous output is generated.

A considerable amount of research has been focused on the application of the basic concepts of self-checking to the design of more complex systems including microprogrammed control units and microprocessors [Nicolaidis 1987, Paschalis *et al.* 1987].

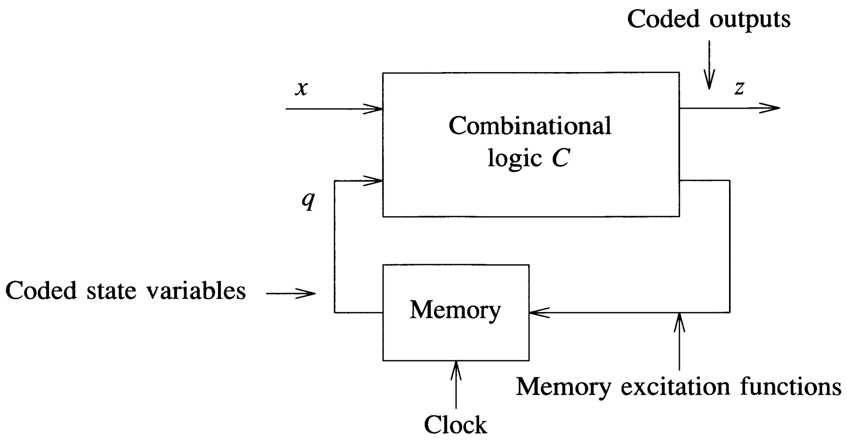


Figure 13.23 Sequential circuit model

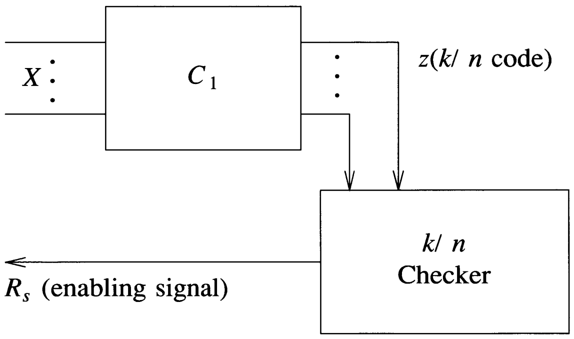


Figure 13.24 Use of codes for self-checking in sequential circuits

REFERENCES

[Anderson 1971] D. A. Anderson, "Design of Self-Checking Digital Networks Using Coding Technique," Univ. of Illinois CSL Report R-527, September, 1972.

[Anderson and Metze 1978] D. A. Anderson and G. Metze, "Design of Totally Self-Checking Check Circuits for *m*-out-of-*n* Codes," *IEEE Trans. on Computers*, Vol. C-22, No. 3, pp. 263-269, March, 1973.

[Carter and Schneider 1968] W. C. Carter and P. R. Schneider, "Design of Dynamically Checked Computers," *IFIP Proceedings*, Vol. 2, pp. 873-883, 1968.

[Gastanis and Halatsis 1983] N. Gastanis and C. Halatsis, "A New Design Method for *m-out-of-n* TSC Checkers," *IEEE Trans. on Computers*, Vol. C-32, No. 3, pp. 273-283, March, 1983.

[Jha 1989] N. K. Jha, "A Totally Self-Checking Checker for Borden's Code," *IEEE Trans. on Computer-Aided Design,* Vol. 8, No. 7, pp. 731-736, July, 1989.

[Mak *et al.* 1982] G. P. Mak, J. A. Abraham, and E. S. Davidson, "The Design of PLAs with Concurrent Error Detection," *Proc. 12th Annual Intn'l. Symp. Fault-Tolerant Computing*, pp. 300-310, June, 1982.

[Marouf and Friedman 1978a] M. Marouf and A. D. Friedman, "Efficient Design of Self-Checking Checkers for Any *m*-out-of-*n* Code," *IEEE Trans. on Computers*, Vol. C-27, No. 6, pp. 482-490, June, 1978.

[Marouf and Friedman 1978b] M. Marouf and A. D. Friedman, "Design of Self-Checking Checkers for Berger Codes," *Digest of Papers 8th Annual Intn'l. Conf. on Fault-Tolerant Computing*, pp. 179-184, June, 1978.

[Nicolaidis 1987] M. Nicolaidis, "Evaluation of a Self-Checking Version of the MC 68000 Microprocessor," *Microprocessing and Microprogramming*, Vol. 20, pp. 235-247, 1987.

[Nikolos *et al.* 1988] D. Nikolos, A. M. Paschalis, and G. Philokyprou, "Efficient Design of Totally Self-Checking Checkers For All Low-Cost Arithmetic Codes," *IEEE Trans. on Computers*, Vol. 37, pp. 807-814, July, 1988.

[Paschalis *et al.* 1987] A. M. Paschalis, C. Halatsis, and G. Philokyprou, "Concurrently Totally Self-Checking Microprogram Control Unit with Duplication of Microprogram Sequencer," *Microprocessing and Microprogramming*, Vol. 20, pp. 271-281, 1987.

[Sellers *et al.* 1968] F. F. Sellers, M. Y. Hsiao, and L. W. Bearnson, *Error Detecting Logic for Digital Computers,* McGraw-Hill, New York, New York, 1968.

[Shedletsky and McCluskey 1975] J. J. Shedletsky and E. J. McCluskey, "The Error Latency of a Fault in a Combinational Digital Circuit," *Digest of Papers 1975 Intn'l. Symp. on Fault-Tolerant Computing*, pp. 210-214, June, 1975.

PROBLEMS

13.1.

a. Consider a 6-bit residue code in which $m = 3$ with the rightmost two bits being check bits. For each of the following, assuming at most a single-bit error, determine if such an error is present and if so which bits might be erroneous.

010110, 011110, 011001

b. Consider a 7-bit Hamming single-error correction code. For each of the following, assuming at most a single-bit error, determine the erroneous bit if any.

0101100, 0101101, 0111101

13.2. Consider a residue code and an error that results in the interchange of two successive bits. Prove that all such errors are detected, or present a counterexample to this conjecture.

13.3. Prove that the $k/2k$ checker of Figure 13.16 is totally self-checking for all single stuck faults.

13.4. Consider a residue code and a burst error that results in unidirectional errors in a sequence of k successive bits. Will all such errors be detected?

13.5. Prove that Berger codes detect all unidirectional errors.

13.6. Design a 2/4 code checker in the form of Figure 13.16, and prove that the circuit is totally self-checking for all single stuck faults.

13.7. Consider the following design of a k-bit equality checker ($k \geq 3$) to determine the equivalence of two words $(a_1,a_2,...,a_k)$ and $(a_1',a_2',...,a_k')$. The circuit has two outputs f_k and g_k defined by the recursive equations

$$f_k = f_{k-1}b_k + g_{k-1}a_k$$

$$g_k = f_{k-1}a_k + g_{k-1}b_k$$

where $b_k = \overline{a}_k'$ and f_2 and g_2 are as defined in Figure 13.19. Verify that this circuit is totally self-checking for all single stuck faults.