This is to certify that the

dissertation entitled

Design and Synthesis of On-Line
Testable Sequential Circuits

presented by

Chia-Shun Lai

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Electrical
Engineering

Major professor

Date 4-30-93

DESIGN AND SYNTHESIS OF ON-LINE TESTABLE SEQUENTIAL CIRCUITS

By

Chia-Shun Lai

A DISSERTATION

submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical Engineering

1993

ABSTRACT

DESIGN AND SYNTHESIS OF ON-LINE TESTABLE SEQUENTIAL CIRCUITS

By

Chia-Shun Lai

With ever-increasing complexity of digital applications, the issue of reliability has become very important in today's VLSI designs. Concurrent error detection schemes using redundancy design approaches have been sucessfully implemented to enhance chip yield and system reliability. Redundancy design approaches may include hardware redundancy, time redundancy, and information redundancy. The simplest concurrent error detection scheme is the duplication with comparison. Any mismatch between two identical blocks will indicate an error. However, this involves more than a 100% increase in hardware overhead cost.

Information redundancy involves the use of coding techniques that enhance circuit capability for reliable operation. Berger codes are least redundant separable codes, they have been implemented for fault-tolerant, fail-safe, and concurrent error testable digital circuits. The features of high speed and low hardware cost are highly desirable especially for a checker design. In response to these needs, a design methodology using a partitioning and folding scheme is developed for the design of fast, yet low hardware cost, Berger code checkers with self-testing capability.

Since the hardware cost is increased almost exponentially as the code length of the checker increases, the use of many smaller checkers will require much less hardware cost than that of a larger checker. In this study, an efficient output function partitioning scheme is developed to partition the set of output functions into many smaller subsets so that smaller checkers can be employed. Based on the delay constraint derived from a set of design specification, the checkers which achieve minimal hardware overhead are chosen. Experimental results show that, with the developed partitioning scheme, the hardware cost may be reduced considerably. With such a low hardware cost checker, on-line testability becomes very promising and practical.

In this research, a system, ***SOLiT***, for automated synthesis of on-line testable sequential circuits with multi-level logic implementation has been developed. The system is implemented on Sun/4 workstation in the C-language. The system receives a behavioral description of finite state machines in ***kiss2*** format and automatically generates the physical layout for a self-checking circuit.

To my wife:
Kai-Ling Hsu

ACKNOWLEDGEMENTS

I would like to thank Professor Chin-Long Wey, my thesis advisor, for his leading me not in the path of ease and comfort but under the stress and spur of difficulties and challenge. I am inspired by his valuable guidance throughout my graduate study. I would like to thank members of my thesis committee, Professors P. David Fisher, Michael Shanblatt, and Jonathan I. Hall, for their meaningful comments and remarkable ideas during the course of my dissertation research. I do very appreciate Professor Erik D. Goodman for his encouragement, support, and assistance. My learning life is richer for having crossed his. I would also like to thank Professor Kun-Mu Chen for his cordial assistance. I am benefited from his provident thinking and effective attitude.

I would like to acknowledge all the faculty member and students who gave me help during my study at Michigan State University, and many friends who showed their inspiration and benevolent feeling.

I am very grateful to my parents and my mother-in-law for their years of concern, encouragement, and support, and to my wife, Kai-Ling, for her wild imagining, significant encouragement and help, and persistent thoughtfulness. She insists in taking care of all things considerately, and keeps learning diligently even though under treatment for her cancer. My latent potentialities have encouraged by her great composure, persevering will, and endless love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

symbol	meaning
$\varnothing$	empty, or null set
$x \in A$	element x is a member of set A
$x \notin A$	element x is not a member of set A
$\exists\, x \in A$	there exists an element x in set A
$\forall\, x \in A$	for every element x in set A
$A \cup B$	union of sets A and B; $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
$A \cap B$	intersection of sets A and B; $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$
$\bigcup_{j=1}^{n} Q_j$	union of sets Q_1, Q_2, ..., and Q_n
$\bigcap_{j=1}^{n} Q_j$	intersection of sets Q_1, Q_2, ..., and Q_n
$A - B$	difference of sets A and B, $A - B = \{x \mid x \in A \text{ and } x \notin B\}$
$\lvert R \rvert$	cardinality of set R, the number of elements in set R
$f: A \rightarrow B$	f is a function mapping from A to B
$\{x \mid x \text{ is } ...\}$	set builder notation
{1,2,...}	elements in a set
$i \in I_v$	index set i = 1, 2, ..., v
$\langle f_1, f_2, ..., f_n \rangle$	ordered sequence containing f_1, f_2, ..., and f_n
Σ	logic OR operation
$\oplus$	logic exclusive OR operation

$\overline{A}$	bitwise complement of a binary vector A
S×T	concatenation of two code, S and T
(010...)	code vector
$\binom{n}{m}$	combinations of m out of n; $\binom{n}{m} = \frac{n!}{(n-m)!m!}$
m/n	m-out-of-n
$C_{m/n}$	m-out-of-n code
B(I, K)	Berger code with I information bits and K check bits
$\lfloor x \rfloor$	floor function of x; the greatest integer less than or equal to the real number x
$\lceil y \rceil$	ceiling function of x; the smallest integer greater than or equal to the real number y

CHAPTER 1

INTRODUCTION

With ever-increasing complexity of digital applications, the issues of reliability have become very important in today's VLSI designs. Reliability can be improved by sophisticated testing schemes to weed out faulty circuits [1]. However, such ***off-line*** or ***static tests*** can identify ***permanent faults***, but not ***transient faults***. It is obvious that a mechanism for ***concurrent error detection*** (CED) must be installed to detect such faults before they cause undesirable results [2].

All ***concurrent error detection*** schemes detect errors through conflicting results generated from operations on the same operands. ***Concurrent error detection*** can be achieved through space redundancy, time redundancy, and information redundancy [3]. The conflicting results are compared and errors caused by faults are detected by a hardware check circuit, i.e., the checker. Checkers can no longer be assumed to be error-free in digital systems, because they are constructed from the same types of components as the circuits that perform the operation and hence are subject to the same type of failures. This brings up to the question: ***Who is checking the checker?*** The answer is clearly a checker that is ***self-checking***.

It would be preferable for the circuits to be designed such that they will indicate any malfunction during the normal operation and will not produce an erroneous result without

an error indication. In these circuits, any fault from a specified set of faults will cause a detectable erroneous output during normal operation, and each fault must not cause erroneous outputs without also producing an error signal [4,5].

Redundancy design [3] approaches have been successfully implemented in today's VLSI designs for enhancing chip yield and system reliability [6]. However, due to speed performance degradation, time redundancy is not suitable for ***on-line*** testing. On other hand, for the hardware redundancy, the simplest ***concurrent error detection*** in multi-level circuits involves duplication of the circuit and comparing the outputs of the two blocks. Any mismatch between them will indicate an error. The equality of two sets of outputs is checked by a totally ***self-checking*** equality comparator [7,8]. This involves more than 100% redundancy (100% due to the duplicate circuitry and some extra for the checker) in the circuit. This is rather a high cost to pay for fault tolerance.

Information redundancy involves the use of coding techniques that enhance circuit capacity for reliable operation. It has been implemented for fault-tolerant, fail-safe, and ***concurrent error detection*** designs of digital circuits. The features of high speed and low hardware cost are highly desirable especially for a checker design. Therefore, this has motivated the development of fast, yet low hardware overhead cost, checker circuits and design methodologies for ***self-checking*** circuits or systems.

1.1 PREVIOUS WORK

Extensive research has shown that the errors in VLSI circuits are of a unidirectional error type [9,10]. A ***unidirectional error*** model assumes that even though both 0-to-1 and 1-to-0 errors can occur, only one type of error occurs in a particular data word [11]. Such errors have been observed in modern digital devices such as PLAs, ROMs, and compact laser disks [12]. Numerous coding techniques, such as ***m-out-of-n codes*** [13-25], ***Berger codes*** [26-32], ***Borden codes*** [12,33], ***Burst detecting codes*** [34,35,36], ***Residue codes***,

and ***AN codes*** [37,38], have been proposed to detect such errors. Among these coding techniques, ***Berger codes*** are the least redundant separable codes [26-29] for ***All-Unidirectional codes***.

A ***self-checking circuit*** [6], as shown in Figure 1.1, consists of a ***functional circuit*** and a ***self-checking checker***. The functional circuit can be either a combinational or a sequential circuit. The functional circuit is generally designed such that its primary outputs and some encoded outputs are able to produce an erroneous result in the presence of a fault. The error indicator must be designed to produce an error signal for some normal circuit inputs whenever a fault from a specified set of faults occurs within the circuit.

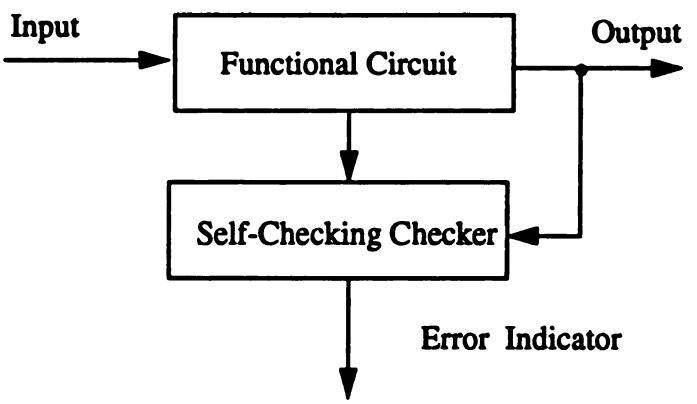


Figure 1.1 A self-checking circuit.

Considerable research efforts have been devoted to the design of ***self-checking*** combinational circuits [39-43] and ***self-checking*** sequential circuits [44-54]. The ***self-checking*** concept has been applied to microprogram control units and PLAs [9,39,52,55,56]. In addition, numerous ***self-checking checker*** designs have also been proposed [14,42,49,57,58]. Little emphasis, however, has been devoted to evaluate the practical designs in terms of area overhead and speed performance degradation.

1.2 PROBLEM STATEMENT

In general, the code length of a checker grows linearly with the number of outputs of a functional circuit, while the complexity of the checker may grow rapidly. As a result, a self-checking circuit with a large amount of outputs requires a huge checker, and the time required for the checker to detect errors may exceed the clock cycle time of the functional circuit [59]. This results in an increase in the system cycle time required to capture the error signal. To alleviate such a problem, it is highly desirable to develop a checker having the features of high speed and low hardware cost.

Since the hardware cost of the checkers increase almost exponentially as the code length increases, the use of many smaller code length checkers require less hardware cost than that of a larger code length checker. Thus, it is desirable to develop a partitioning scheme which partitions the output of a given circuit into many smaller groups so that smaller checkers can be used to reduce both hardware overhead and performance degradation. The number of smaller checkers can be generally determined based on the performance measurement AT^k, where A is the gate count, T is the gate delay, and k is a measurement parameter determined by the applications. In this thesis, the optimization is achieved by minimizing the gate count under delay constraint. In other words, based on the delay constraint derived from a set of design specifications, the checkers which achieve minimal hardware overhead are chosen.

As sophisticated techniques for the synthesis of the logic emerge, automated synthesis systems are becoming popular. The synthesis system such as *sis* [60] generates very good results for multi-level logic implementation. It targets optimizing either chip area or speed performance. However, it is also desirable to address the reliability issue, in automated synthesis system.

1.3 RESEARCH TASKS

Due to the salient feature of least redundant separable codes, ***Berger codes*** have been implemented for fault-tolerant and fail-safe designs of digital circuits to detect all unidirectional errors. The features of high speed and lower hardware cost are highly desirable especially in the checker design. Thus, the first task in this research is to develop a fast, yet low hardware cost ***Berger code*** checker. A partitioning and folding scheme is developed to design a ***self-testing checker*** for ***Berger codes*** in which both hardware overhead and speed degradation are improved significantly.

In order to reduce the code lengths of the checkers and to further reduce the hardware cost, the second task of this research is to develop an efficient output partitioning scheme. The scheme partitions the output into many smaller groups which employ small sized checkers. The chromatic partitioning of a graph is used to formulate the problem. The number of groups is equivalent to the chromatic number. In this research, a partitioning algorithm is presented. Finally, in addition to taking the area and speed as design objectives, reliability should be also a design objective of an automated synthesis system. Thus, the third task is to develop a system for automated synthesis of ***on-line*** testable sequential circuits for multi-level logic implementation.

The three tasks presented in this research are summarized as follows:

(1) develop a fast, yet low hardware cost, ***Berger code*** checker;

(2) develop an efficient output partitioning algorithm; and

(3) develop an automated synthesis system for ***on-line*** testable circuits.

1.4 THESIS ORGANIZATION

This thesis is organized as follows: Chapter 2 reviews the previous work in the design of both checkers and functional circuits. Chapter 3 describes the developed

partitioning and folding scheme for ***Berger code*** checkers. An output function partitioning algorithm is presented in Chapter 4 with some experimental results. Chapter 5 illustrates a system, namely, ***SOLiT***, for automated synthesis of ***on-line*** testable sequential circuits with multi-level logic implementation. The system takes a behavioral description as its input and produce a physical layout for chip fabrication. Finally, conclusions and future research are given in Chapter 6.

CHAPTER 2

ON-LINE SELF-TESTABLE CIRCUIT AND SYSTEM

This chapter summarizes the terminology used in this thesis, and briefly describes the characteristics of checker and functional circuits for ***on-line*** testing circuits and systems.

2.1 TERMINOLOGY

The following notation and definitions used in this study can be found in [4]:

N: input code space.

S: output code space.

x: an input vector.

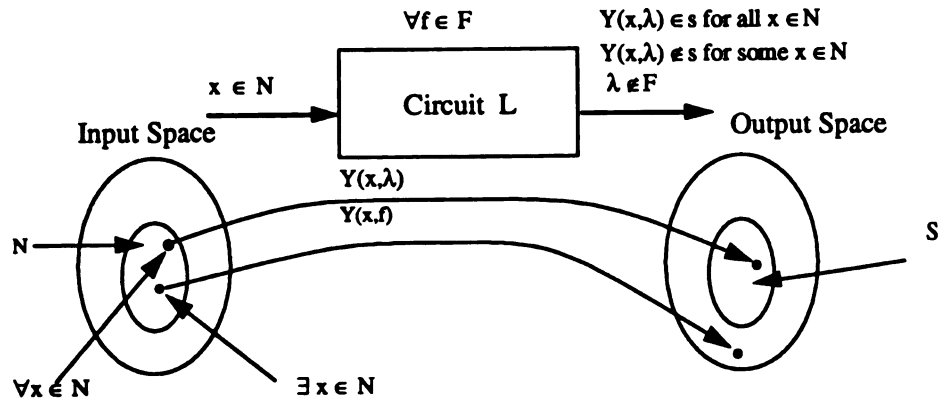
f: a fault in the fault set F.

λ: an output vector in the correct output space.

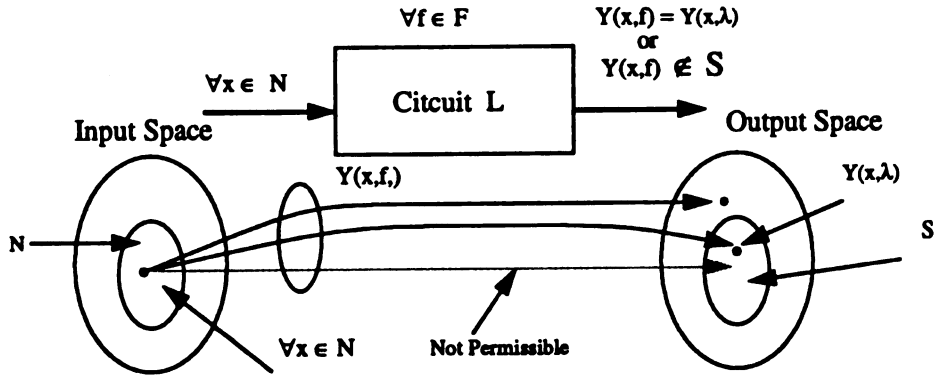
Y: mapping function of the circuit.

Self-Testing: A circuit L is ***Self-Testing*** (ST) for an input set N and a fault set F if for every fault f in F there is some input x in N such that Y(x,f) is not in S.

Fault-Secure: A circuit L is ***Fault-Secure*** (FS) for an input set N and a fault set F if for any input x in N and for any fault f in F, $Y(x,f) = Y(x,\lambda)$, or $Y(x,f) \notin S$.



(a) Self-testing circuit.



(b) Fault-secure circuit.

Figure 2.1 Self-testing and fault-secure circuits.

Totally Self-Checking: A circuit is ***Totally Self-Checking*** (TSC) if it is both ***fault-secure*** and ***self-testing***.

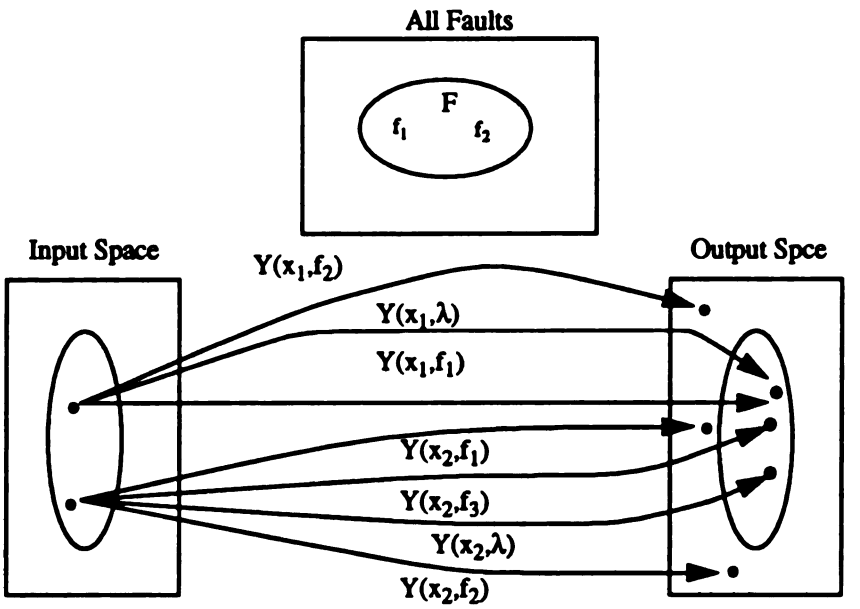


Figure 2.2 Totally self-checking circuit.

Code-Disjoint: A circuit is ***Code-Disjoint*** (CD) if the input code space maps to the output code space and the input noncode space maps to the output noncode space.

Self-Testing Checker: A checker is a ***Self-Testing Checker*** (STC) if it is ***self-testing*** and ***code-disjoint***.

Strongly Fault Secure: For a fault sequence $<f_1,f_2,...,f_n>$. let k be the smallest integer for which the combination of f_i, $1 \leq i \leq k$, does not produce the correct code output for at least one code input in a circuit G. If there is no such k, set k = n. Then G is ***Strongly Fault Secure*** (SFS).

Strongly Code Disjoint: A circuit is ***Strongly Code Disjoint*** (SCD) for a fault set F if, for every fault f in F, either

(1) the circuit is ***code-disjoint*** and ***self-testing*** for the fault f, or

(2) the circuit is ***code-disjoint*** and if f occurs then the resultant circuit is still SCD for the fault set F excluding the fault f.

Strongly-Self-Checking: A circuit is ***Strongly-Self-Checking*** (SSC) for a set of faults F if before the occurrence of any fault, the circuit is ***code-disjoint***, and for every fault in F, either:

(a) the circuit is ***self-testing*** and ***fault-secure***, or

(b) the circuit is ***fault-secure*** and always maps noncode words at the inputs to non-codewords at the outputs,

and if another fault from F occurs, then either property (a) or (b) is true for the fault sequence.

m-out-of-n (m/n) code: In a ***m-out-of-n (m/n)*** code, denoted as $C_{m/n}$, all code words have exactly m 1's and (n-m) 0's. It is also called m-hot code.

Berger code: A ***Berger code*** B(I,K) of length L has an I-bit information part and a K-bit check part, where $K = \lceil \log_2(I + 1) \rceil$ and $L = I + K$. The check part is the bit by bit complemented binary representation of the number of 1's in the information part.

Systematic: A code is ***Systematic*** if it has K check bits appended to I information bits and all the 2^I possible information I-tuple are assumed to occur.

Separable Code: If all the 2^I possible I-tuple do not occur and it is known which one is presented, a code derived for this case is referred to as a ***Separable Code***.

True Minterm: A minterm m_i for a single output function f is called a ***True Minterm*** if f= 1 when m_i is applied.

False Minterm: A minterm m_i for a multi-output function, f_i, $1 \leq i \leq n$, is called a ***False Minterm*** if all $f_i = 0$ when m_i is applied.

Complete Covering: ***Complete Covering*** requires the covering of 1-cells as usual, but in addition, each 0-cell must be covered by some product terms.

True Product Term: A product term p_i is a ***True Product Term (False Product Term)*** if all the minterms it covered are true minterms (false minterm).

2.2 SELF-TESTING CIRCUIT STRUCTURES

A general structure of a ***totally self-checking*** (TSC) circuit is shown in Figure 1.1. During the normal operation, code inputs are applied to the functional circuits and the coded outputs are produced. Both the functional circuits and the checkers should possess some special properties. This section reviews some existing STC designs for checkers and TSC design for functional circuits.

2.2.1 CHECKER DESIGN

Two coding techniques, the ***m-out-of-n (m/n) code*** and the ***Berger code***, have been successfully implemented in VLSI design for detecting all unidirectional errors. Basically, a ***m/n*** checker consists of two independent sets of subcircuits, each having a single output. The ***m/n*** code checker produces the output (0,1) or (1,0) for the application of a ***m/n*** code-word input, i.e., the number of 1's at the checker input is m, and the output (0,0) or (1,1) for non-codeword inputs. Considerable design alternatives for ***m/n*** code checkers have been studied in [13-25]. The circuits for the ***m/n*** checker may be implemented with two-level, three-level, or multi-level logic. Numerous ***self-checking checkers*** for ***Berger codes***

have been presented in [26-32]. The basic design principle for a ***Berger code*** checker is to generate the replicated check bits of binary value complementary to the original ones and to compare them using a ***two-rail*** comparator. A general structure is shown in Figure 2.3.

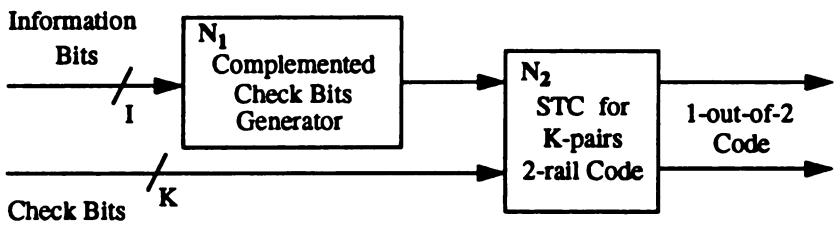


Figure 2.3 Typical structure of STC for normal Berger code checker.

A. m/n Checkers

A ***m/n*** checker design with two-level logic realization was presented in [15]. Consider a ***m-out-of-2m*** code checker, where the 2m bits are partitioned into two distinct subsets $A = \{x_i \mid 0 \le i \le m-1\}$ and $B = \{x_j \mid m \le j \le 2m-1\}$. If the two outputs of the checker are designed as

$$f = \sum_{i=p}^{q} T(k_a \ge i) \bullet T(k_b \ge m-i) \qquad i \text{ is odd,} \tag{2.1}$$

$$g = \sum_{i=p}^{q} T(k_a \ge i) \bullet T(k_b \ge m-i) \qquad i \text{ is even,} \tag{2.2}$$

where $p = \max(m-n_b, -1)$, $q = \min(n_a, m+1)$,

The

jorit

the r

has t

the v

simul

(m - i

more,

from

aratel

larly,

(m - i

k_a +

codew

the su

functi

dant i

testin

norma

n_a = the number of bits in group A,

n_b = the number of bits in group B,

m = the number of 1's in the information bits,

k_a, k_b = the number of 1's occurring in each group.

The notation Σ represents the OR operation, and • represents the AND operation. The majority function $T(k_a \geq i)$ represents the Boolean function that has the value 1 if and only if the number of 1's in subset A is greater than or equal to the value i; similarly, $T(k_b \geq m - i)$ has the value 1 if and only if the number of 1's in the subset B is greater than or equal to the value m - i.

The term $T(k_a \geq i) \bullet T(k_b \geq m - i) = 1$ is tested by making $T(k_a \geq i)$ and $T(k_b \geq m-i)$ simultaneously be equal to 1. This is done with codewords containing i 1's in subset A and (m - i) 1's in subset B. These codewords exist because $0 \leq i \leq m$ and $n_a = n_b = m$. Furthermore, $T(k_a \geq i)$ may be implemented by the AND of each different combination of i bits from the n_a available; the results are ORed together. Each combination must be tested separately, and this requires $\binom{n_a}{i} = \binom{m}{i}$ code words, each containing exactly i 1's in A. Similarly, $T(k_b \geq m - i)$ requires $\binom{n_b}{m-i} = \binom{m}{m-i} = \binom{m}{i}$ codewords, each containing exactly (m - i) 1's in B. Since the bits in A and B are independent except for the constraint that $k_a + k_b = m$ for codewords, these factors can be completely tested with the same $\binom{m}{i}$ codewords. To test all such terms for $0 \leq i \leq m$, it requires a total of 2^m test patterns, i.e., the sum of $\binom{m}{i}$ for $0 \leq i \leq m$.

If such a test set supplies all 2^m input combinations to A and B, then the majority functions, such as $T(k_a \geq i)$ or $T(k_b \geq m - i)$, are exhaustively tested, and any non-redundant implementation of them is completely diagnosed. Therefore, the checkers are ***self-testing*** for single faults or unidirectional faults if these 2^m codewords are given during normal operation.

A ***m/n*** code checker, with $n \neq 2m$, as shown in Figure 2.4, is generally realized by translating the given code to a 1-out-of-$\binom{n}{m}$ ($1/\binom{n}{m}$) code, which is then converted to a p-out-of-2p (p/2p) code via a TSC translator [15]. In this case, p should satisfy the following relation:

$$\binom{2p}{p} \geq \binom{n}{m} \geq 2^p \tag{2.3}$$

The codewords should be selected so that the left-most p bits are complementary to the right-most p bits as shown in Figure 2.5. The two-level realization obtained is indeed a minimum level TSC circuit. However, when the number of inputs become larger, the fan-in of every OR gate grows fast making it impractical for implementing.

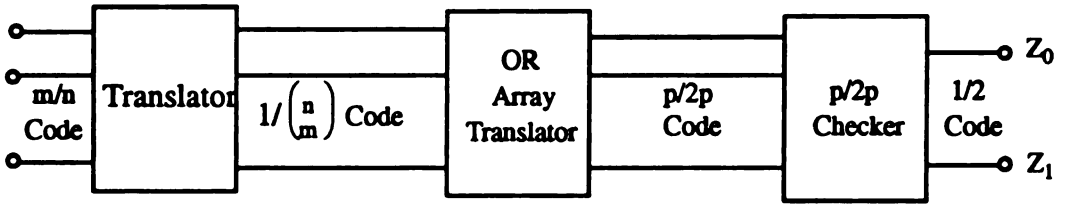


Figure 2.4 General self-testing checker for m/n checker with $n \neq 2m$.

x_0	x_1	x_2	x_3	x_4	x_5
0	0	0	1	1	1
1	0	0	0	1	1
0	1	0	1	0	1
0	0	1	1	1	0
1	1	0	0	0	1
0	1	1	1	0	0
1	0	1	0	1	0
1	1	1	0	0	0

Figure 2.5 Test patterns for m/n checker.

An alternative ***m/n*** TSC design using a 3-level realization has been presented in [17] for $n \geq 4$. For $n < 2m$ and $n > 2m$, the procedure gives AND-AND-OR and OR-OR-AND 3-level realizations, respectively, while for $n = 2m$, it results in an AND-OR (or dually OR-AND) 2-level realization which is equivalent to the design provided in [15]. The design procedure consists of the following two stages: **(1)** Construct two 2-level AND-OR (OR-AND) circuits with no shared gates, which correspond to blocks of the partition. **(2)** Provide AND (OR) gates, which are shared by two 2-level AND-OR (OR-AND) circuits, to form AND-AND-OR (OR-OR-AND) realization.

Basically, a ***m/n*** code is comprised of a set of ***m/n*** code vector x, and it can be partitioned into two disjoint subsets G_a and G_b, where k_a and k_b are defined in Equations (2.1) and (2.2).

$$G_a = \{x | x \in \textbf{\textit{m/n}}, w(k_a) \text{ is odd}\} \tag{2.4}$$

$$G_b = \{x | x \in \textbf{\textit{m/n}}, w(k_b) \text{ is even}\} \tag{2.5}$$

where $w(k_a)$ and $w(k_b)$ are the number of 1's in subset G_a and G_b, respectively. A disjoint 2-level AND-OR realization based on the partition is a pair of 2-level AND-OR circuits g_a and g_b as shown in Figure 2.6, where there exists a one-to-one correspondence between AND gates in circuit g_a and codewords in G_a, and between AND gates in circuit g_b and codewords in G_b. Exactly m bits of value 1 in codeword x are used as inputs to the AND gates for x. A disjoint two-level OR-AND realization can be defined in a dual way. Such a 3-level configuration with shared gates is illustrated in Figure 2.7. AND gates at the left-most stage are referred to as shared ANDs and those at the middle stage are called 3-level AND-AND-OR realization (a 3-level OR-OR-AND realization is defined in a dual manner).

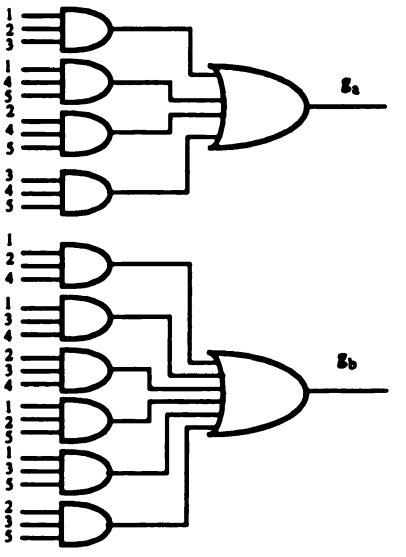


Figure 2.6 Disjoint 2-level realization for $C_{3/5}$.

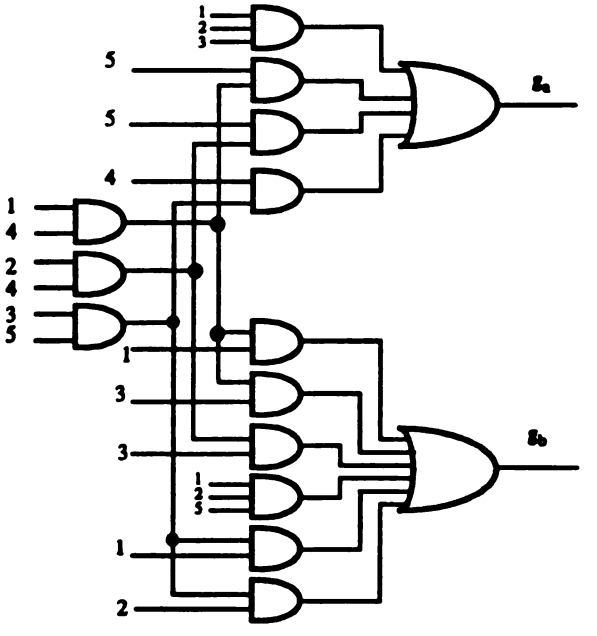


Figure 2.7 3-level realization for $C_{3/5}$.

Similar to two-level, the three-level implementation also has the large fan-in problem. In order to alleviate this problem, a cost-effective ***m/n*** checker design with multi-level logic implementation has been presented for $m \geq 3$, $4m \geq n > 2m$ [20]. Basically, any ***m/n*** code can be represented as a sum of concatenated partitioned codes i/n_a and $(m-i)/n_b$ for $0 \leq i \leq m$, and $n_a + n_b = n$. Thus, the set of all ***m/n*** codes can be formally written with the union operator as

$$C_{m/n} = \bigcup_{i=0}^{m} C_{i/n_a} \times C_{(m+i)/n_b} \tag{2.6}$$

where $\cup$ is a union representation and $\times$ is a concatenation operator. The set $C_{m/n}$ can be partitioned into the following three disjoint subsets:

$$C_{AB} = C_{k_a/n_a} \times C_{k_b/n_b}; \tag{2.7}$$

$$C_A = \bigcup_{i=0}^{k_a - 1} C_{i/n_a} \times C_{(m-i)/n_b}; \tag{2.8}$$

$$C_B = \bigcup_{i=0}^{k_b - 1} C_{(m-i)/n_a} \times C_{i/n_b}. \tag{2.9}$$

where $k_a = \lfloor m/2 \rfloor$, $k_b = \lceil m/2 \rceil$, $1 \leq k_a \leq n_a$, $1 \leq k_b \leq n_b$. Both sets C_A and C_B can be further partitioned by the same manner. With this partition scheme and algorithm presented in [20], the ***m/n*** checkers, as shown in Figure 2.8, can be obtained by using a translator H^1 of the ***m/n*** code into 2/4 code, and a TSC 2/4 checker, or H^2. The block H^2 will produce a 1/2 code as the error signal. For demonstrating this algorithm [20], an example is given below.

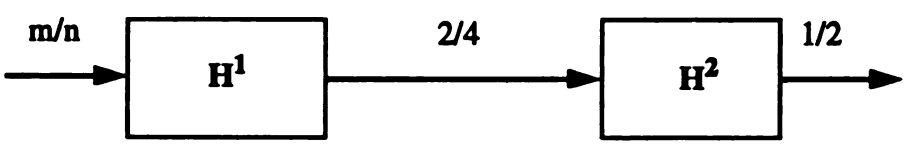


Figure 2.8 m/n code checker.

Example 2.1:

The following equations are used to design a TSC checker for 3/8 code [20].

$$h_1^1 = x_1 + (x_2 + x_3)$$
$$h_2^1 = (x_4 + x_5) + (x_6 + x_7)$$
$$h_3^1 = x_1(x_2 + x_3) + (x_4 + x_5)(x_6 + x_7)$$
$$h_4^1 = x_2x_3 + x_4x_5 + x_6x_7$$
$$h_1^2 = (h_1^1 + h_3^1)(h_2^1 + h_4^1)$$
$$h_2^2 = h_1^1 h_3^1 + h_2^1 h_4^1$$

Numerous TSC 1/n code checker designs have been developed [21,22,23]. Similar to Figure 2.8, a 1/n TSC checker, as shown in Figure 2.9, can be obtained by using a translator H^1 to translate the 1/n code into $2/n_0$ code, then into 1/2 code for indication [20]. At the first stage, n_0 should be selected such that $\binom{n_0 - 1}{2} \le n \le \binom{n_0}{2}$ and follow the algorithm in [20] to obtain the partitioned subsets h_i^0, $1 \le i \le n_0$. A $2/n_0$ code results. Then, if n_0 is large, a further reduction of n_0 will be necessary to obtain a smaller n_1, and then a 1/2 two-rail checker is appended to the last stage.

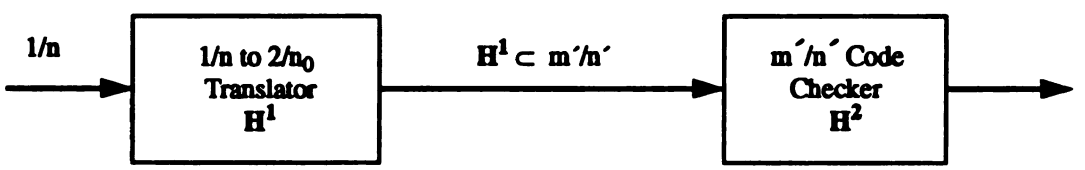


Figure 2.9 1/n code checker.

Example 2.2: [25]

Consider the design of a 1/20 code checker. It first needs a H^1 translator which translates the 1/20 code to a 2/7 code with the functions h_i^0 , $1 \leq i \leq 7$, Then the 2/7 code is translated to a 2/4 code as shown in Example 2.1. The following equations summarize the 1/20 code checker design.

$$h_1^0 = x_1 + x_7 + x_8 + x_{14} + x_{15}$$
$$h_2^0 = x_1 + x_2 + x_{11} + x_{12} + x_{19} + x_{20}$$
$$h_3^0 = x_2 + x_3 + x_8 + x_9 + x_{17} + x_{18}$$
$$h_4^0 = x_3 + x_4 + x_{12} + x_{13} + x_{15} + x_{16}$$
$$h_5^0 = x_4 + x_5 + x_9 + x_{10} + x_{20}$$
$$h_6^0 = x_5 + x_6 + x_{13} + x_{14} + x_{18} + x_{19}$$
$$h_7^0 = x_6 + x_7 + x_{10} + x_{11} + x_{16} + x_{17}$$
$$h_1^1 = h_1^0 + (h_2^0 + h_3^0)$$
$$h_2^1 = (h_4^0 + h_5^0) + (h_6^0 + h_7^0)$$
$$h_3^1 = h_1^0 (h_2^0 + h_3^0) + (h_4^0 + h_5^0)(h_6^0 + h_7^0)$$
$$h_4^1 = h_2^0 h_3^0 + h_4^0 h_5^0 + h_6^0 h_7^0$$
$$h_1^2 = (h_1^1 + h_3^1)(h_2^1 + h_4^1)$$
$$h_2^2 = h_1^1 h_3^1 + h_2^1 h_4^1$$

B. Berger Code Checkers

Figure 2.3 shows the general structure of a ***Berger code*** checker. The complemented check bits generator is used to generate the binary value corresponding to the number of 1's in the information bits. In general, a set of full adder modules that add the information bits in parallel and produce the binary number corresponding to the number of 1's in the information bits has been implemented [29]. Recently, based on a completely different

structure, a ***Berger code*** partitioning scheme has been presented for the checker design [30].

Consider a ***Berger code*** checker design [29] using a set of full adder modules for the complemented check bit generator. The design procedure is summarized as follows.

(1) Let $s = \{x_i \mid 1 \le i \le 2^k-1\}$ be the set of all information bits, and set $m = k$ and $j = 1$.

(2) Partition the set S into three groups A^j, B^j, and E^j, which respectively consist of the left-most $2^{m-1}-1$ bits, the next $2^{m-1}-1$ bits, and the right-most bit. Let $a^j = (a^j_{m-1}, a^j_{m-2}, \ldots, a^j_1)$, $b^j = (b^j_{m-1}, b^j_{m-2}, \ldots, b^j_1)$, and e^j be the binary representations of the number of ones occurring in the information bits of groups A^j, B^j, and E^j, respectively.

(3) The binary representation of the number of ones occurring in S, $g^j = (g^j_m, g^j_{m-1}, \ldots, g^j_1)$ is given by the following addition:

$$\begin{array}{cccc} & a^j_{m-1} & a^j_{m-2} \cdots & a^j_1 \\ & b^j_{m-1} & b^j_{m-2} \cdots & b^j_1 \\ + & & & e^j \\ \hline g^j_m & g^j_{m-1} & g^j_{m-2} \cdots & g^j_1 \end{array}$$

A ripple carry adder with m - 1 stages is used to generate the vector g^j.

(4) If $m > 2$, then: (i) $m = m - 1$, $L = j$; (ii) let $S = A^j$, $j = j + 1$ and repeat steps 2 and 3 to generate the vector $a^L = g^L$. The vector b^L is generated by a circuit identical to that which generates a^L.

(5) End.

Consider the ***Berger code*** checker based on partitioning scheme in [30]. The design proceeds from the idea that any ***Berger code*** can be constructed from $u = \lceil (I + 1)/2 \rceil$ ***m/n*** codes, where I is the number of information bits, $m = 2i-1$, $1 \le i \le u$, and $n = I + 1$. Table 2.1 (a) lists the B(7,3) codewords. Each subset D_i consists of seven information bits $\{x_i \mid 1 \le i \le 7\}$ and three check bits $\{x_8, x_9, x_{10}\}$. The subset D_i is a collection of ***Berger***

Table 2.1: Berger Code B(7,3).

(a)

Subset	expanded information part x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_{10}	reduced check part x_8 x_9
D_1	0 0 0 0 0 0 0 1	1 1
D_2	0 0 0 0 0 0 1 0	1 1
D_3	0 0 0 0 0 1 1 1	1 0
D_4	0 0 0 0 1 1 1 0	1 0
D_5	0 0 0 1 1 1 1 1	0 1
D_6	0 0 1 1 1 1 1 0	0 1
D_7	0 1 1 1 1 1 1 1	0 0
D_8	1 1 1 1 1 1 1 0	0 0

(b)

Subset	expanded information part x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_{10}	reduced check part x_8 x_9	partitioning
E_1	0 0 0 0 0 0 0 1	1 1	$C_{1/8}\times(1\ 1)$
E_3	0 0 0 0 0 1 1 1	1 0	$C_{3/8}\times(1\ 0)$
E_5	0 0 0 1 1 1 1 1	0 1	$C_{5/8}\times(0\ 1)$
E_7	0 1 1 1 1 1 1 1	0 0	$C_{7/8}\times(0\ 0)$

codewords in which the number of 1's in the corresponding information part is i - 1, and $D_i = C_{(i-1)/(I+1)} \times P_i$, where $\times$ is a concatenation operator, P_i is the check part of D_i. The ***Berger code*** B(7,3) can be generally represented as a union of eight ***m/n*** codes $C_{i/7}$, $0 \leq i \leq 7$, as follows

$$B(7,3) = C_{0/7} \times (111) \cup C_{1/7} \times (110) \cup C_{2/7} \times (101) \cup C_{3/7} \times (100) \cup$$
$$C_{4/7} \times (011) \cup C_{5/7} \times (010) \cup C_{6/7} \times (001) \cup C_{7/7} \times (000).$$

The ***partitioning scheme*** first expands the information part J by adding the least significant bit (LSB) of the check part. Table 2.1 (b) illustrates the ***expanded information part*** $J^* = \{x_1, x_2, ..., x_7, x_{10}\}$ and the ***reduced check part*** $P^* = \{x_8, x_9\}$. As a result, the subsets D_{2i-1} and D_{2i} not only have the same number of 1's, but also have the same reduced check part. Thus, the adjacent pair of subsets D_{2i-1} and D_{2i} are grouped and replaced by a subset E_{2i-1}, as shown in Table 2.1 (b). The ***Berger code*** is written as

$$B(7,3) = C_{1/8} \times (11) \cup C_{3/8} \times (10) \cup C_{5/8} \times (01) \cup C_{7/8} \times (00).$$

i.e., the implementation requires only four checkers for $C_{1/8}$, $C_{3/8}$, $C_{5/8}$, and $C_{7/8}$. This results in reducing the number of checkers by half with the partitioning scheme.

In [30], two design alternatives were presented: a minimal-level design and a cost-effective design. The former uses the minimal-level STCs for ***m/n*** codes to reduce the number of gate levels, while the latter employs multi-level logic implementation to reduce the gate count. Results have shown that the STC design for B(7,3), for example, requires 5 gate levels and 137 gates for minimal-level implementation and 8 levels and 87 gates for cost-effective implementation [30]. However, it needs 17 gate levels and 40 gates in [28] and 11 gate levels and 58 gates in [29]. The B(15,4) checker takes 11 gate levels and 315 gates for cost effective implementation in [30], but requires 35 gate levels and 149 gates in [28] and

17 gate levels and 150 gates in [29]. Compared to the STC designs presented in [28], the partitioning scheme offers an improvement in delay but with a great cost in hardware. However, it should be mentioned that the implementation of the partitioning scheme is available only for B(I,K) with $I = 2^K - 1$ or $2^K - 2$, i.e., the design is not STC for any other ***Berger code*** lengths.

2.2.2 FUNCTIONAL CIRCUITS

A system may consist of several subsystems which can be either combinational or sequential, interconnected with each other via certain interfaces. Two extreme approaches exist for such a system to be TSC [56].

(1) If all the subsystems are TSC and all the interfaces are monitored by checkers which are CD and ST, then the system is TSC, as shown in Figure 2.10 (a).

(2) If all the systems are CD in addition to being TSC, then the system is TSC with no checker used at the embedded interfaces except for the primary output of the system, as shown in Figure 2.10 (b).

A similar argument holds for SFS systems as shown in Figure 2.11 (a) and (b). However, in approach (1), the use of a large number of checkers for embedded partial interfaces may incur not only an intolerable increase in the amount of checker hardware, but also an unacceptable delay in error indication [52]. This is because all the checker outputs must be reduced to a single pair in a ***self-testing*** manner. On the other hand, in approach (2), it may be difficult for subsystems to be either CD, SCD. Therefore, a trade-off between these two approaches should be made.

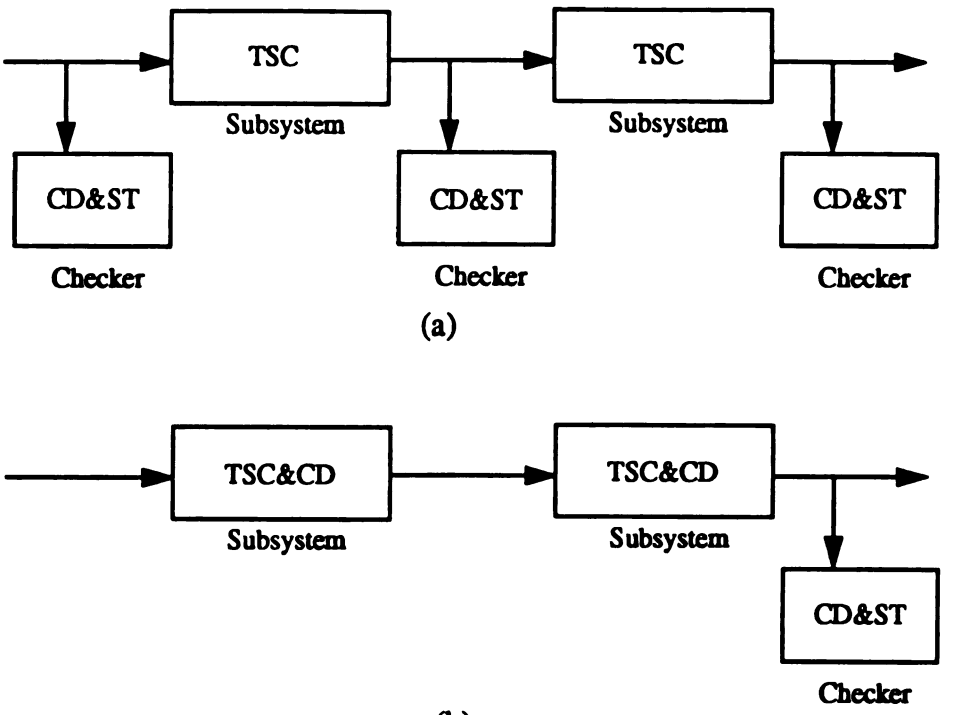


Figure 2.10 TSC systems: (a) with TSC subsystems; and (b) with TSC and CD subsystems.

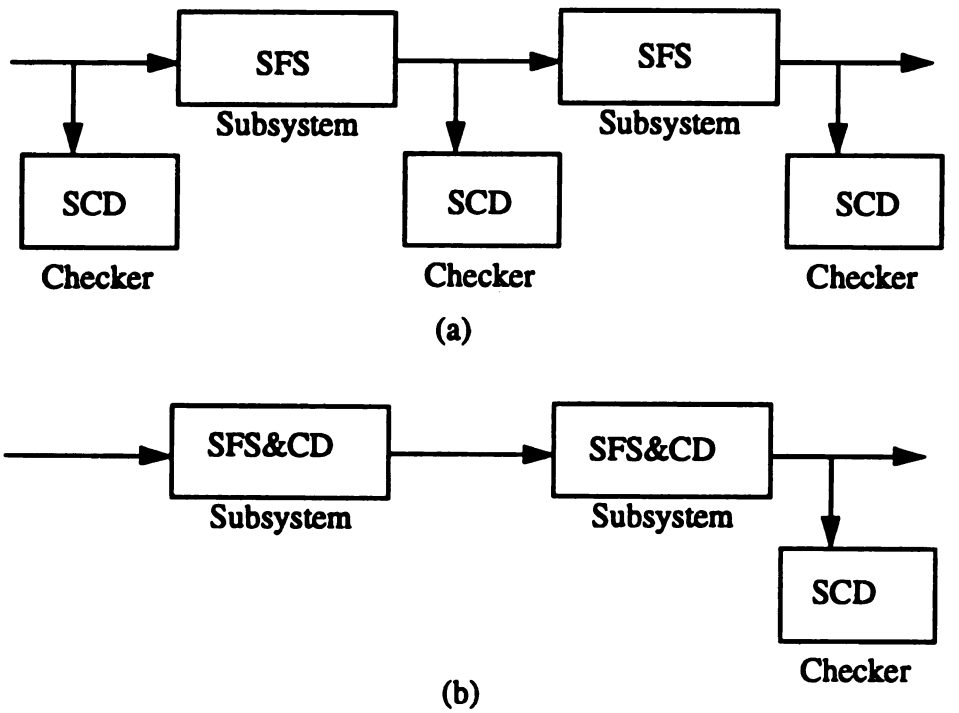


Figure 2.11 SFS systems: (a) with SFS subsystems; and (b) with SFS and CD subsystems.

A. C

Schn
self-c
[39].
ray v
done
uct t
outp
SFS
ly, th
ic fa

B. S

the
the
low
nati
exis
goe
ther

mor
will
curs
stat

A. Combinational Circuits

The concept of a dynamically checked system was first introduced by Carter and Schneider [61] and later formalized and extended by defining both totally and partially ***self-checking*** notions for combinational circuits [50]. Consider the two-level PLA design [39]. The conventional design of PLAs generates all the true product terms in the AND array without considering any false product terms, but the design of SFS PLAs should be done differently. It is proven that if a PLA includes each false minterm in some false product terms in the AND array, then any unidirectional error in the PLA will propagate to the output. If the external output is encoded in an unidirectional error-detecting code and a SFS checker is used to monitor the outputs, the error in the PLA will be detected. Similarly, this concept can be extended to the multi-level combinational circuit. With the algebraic factorization of two-level logic a SC multi-level combination circuit will be obtained.

B. Sequential Circuits

Consider a Moore type sequential machine, as shown in Figure 2.12, where Q is the set of states, Z is the set of output vectors, T is the state transition function, and G is the output function. It has been shown that a sequential machine is ST and FS if the following conditions are held [50]: **(a)** The combinational function T is SC; **(b)** The combinational function G is SC; **(c)** G is CD, even in the presence of faults in G; and **(d)** There exists a sequence i applied during normal functioning such that the next state function goes through all the transitions of the transition diagram (before a second fault occurs), then the sequential machine is SC for a fault set E_f.

Mukai and Tohma [51] have proved that, using a state assignment which leads to monotonic next state equations and output equations with respect to the primary inputs, will cause unidirectional errors at the primary output when a single fault in the circuit occurs. Any code in which the codewords have no ordering among them can be used for the state assignment.

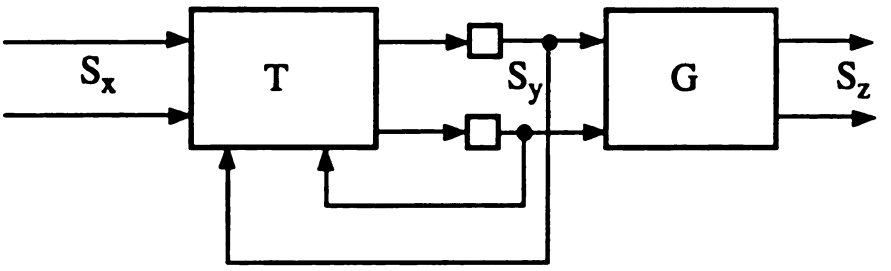


Figure 2.12 A sequential circuit (Moore type).

2.2.3 LOGIC SYNTHESIS PROCEDURE

A logic synthesis procedure for finite-state machines (FSMs) includes three major steps: state assignment, logic optimization, and logic implementation. In order to synthesize FSMs with the tools developed at the University of California at Berkeley, the standard ***kiss2*** format is used to describe the behavior of a FSM, as shown in Figure 2.13 (a), where "i" is the number of inputs; "o" is the number of outputs; "s" is the number of states, "p" is the number of product terms. The state assignment tool, ***mustang*** [62], is employed to encode internal state variables, as shown in Figure 2.13(b). The resultant logic is optimized by either ***espresso*** [63], for two-level logic minimization, ***misII*** [63] or ***sis***, for multi-level optimization. Figure 2.13(c) lists the multi-level logic optimization using ***sis***. A test pattern generation tool, ***stallion*** [63], can be used to generate test patterns to evaluate the synthesized circuits. Finally, the synthesized circuit can be implemented with either standard cells or gate arrays. Figure 2.13(d) lists the netlist generated by the technology mapper in ***sis***, where standard cells in the ***sis*** cell library are used.

```
.i  6
.o  9
.p 21
.s 14
1-----  1   3 110000000
1-----  3   2 000000000
1-----  2   5 001000000
1-----  5   7 000000000
10----  7   7 000000000
11----  7  11 100110000
1-----  11 12 100100000
1-1---  12  8 000001100
1-0---  12  8 000000100
1-0---  8   3 110000000
1-10--  8   3 110000000
1-11--  8   4 110000000
1---1-  4  13 000000010
1---0-  4  13 000000000
1-----  13 14 001000010
1-----  14  6 000000000
10----  6   6 000000000
11----  6   9 100110000
1-----  9  10 100100000
1----1  10  3 110000101
1----0  10  4 110000100
```

(a) standard *kiss2* format.

```
.p 21
#STATE ASSIGNED FINITE AUTOMATON
# 1  0010 0010
# 3  1010 1010
# 2  1000 1000
# 5  0110 0110
# 7  0111 0111
# 11 0100 0100
# 12 1011 1011
# 8  0000 0000
# 4  1111 1111
# 13 1001 1001
# 14 1100 1100
# 6  0011 0011
# 9  0101 0101
# 10 0001 0001
.i 10
.o 13
.type fr
1----- 0010 1010 110000000
1----- 1010 1000 000000000
1----- 1000 0110 001000000
1----- 0110 0111 000000000
10---- 0111 0111 000000000
11---- 0111 0100 100110000
1----- 0100 1011 100100000
1-1--- 1011 0000 000001100
1-0--- 1011 0000 000000100
1-0--- 0000 1010 110000000
1-10-- 0000 1010 110000000
1-11-- 0000 1111 110000000
1---1- 1111 1001 000000010
1---0- 1111 1001 000000000
1----- 1001 1100 001000010
1----- 1100 0011 000000000
10---- 0011 0011 000000000
11---- 0011 0101 100110000
1----- 0101 0001 100100000
1----1 0001 1010 110000101
1----0 0001 1111 110000100
```

(b) state assignment with ***mustang***.

Figure 2.13 A logic synthesis example.

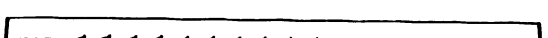


Figure 2.13 Cont'd

```
INORDER = v0 v1 v2 v3 v4 v5 v6 v7 v8 v9;
OUTORDER = v10.0 v10.1 v10.2 v10.3 v10.4 v10.5 v10.6 v10.7 v10.8 v10.9
v10.10 v10.11 v10.12;
v10.0 = v8*v10.8 + [13] + v10.5;
v10.1 = v9*!v10.0*!v10.10 + !v3*!v9*v10.5 + !v2*!v9*v10.5 + v6*v10.10
+ v7*!v10.7 + [13] + v10.12;
v10.2 = v6*v10.0*!v10.12 + v7*!v9*!v10.6 + v8*!v10.0*v10.3 +
!v10.1*v10.5;
v10.3 = !v10.8*[15] + !v7*v10.7 + v6*v9 + v10.6;
v10.4 = v10.7 + v10.5;
v10.5 = !v6*!v7*!v10.10 + !v8*v10.10;
v10.6 = v7*[13];
v10.7 = !v9*!v10.6*[15] + v6*!v7*!v8 + v10.8;
v10.8 = v1*v9*[15];
v10.9 = v2*v8*v10.10;
v10.10 = !v7*v9;
v10.11 = v4*v6*[13] + v6*v10.6;
v10.12 = v5*!v8*v10.10;
[13] = v8*!v9;
[15] = !v6*v7;
```

(c) resultant output network from *sis*.

```
.model ex4.p.esp
.inputs v0 v1 v2 v3 v4 v5 v6 v7
v8 v9
.outputs v10.0 v10.1 v10.2 v10.3
v10.4 v10.5 v10.6 v10.7 v10.8
v10.9 v10.10 v10.11 v10.12
.names v8 v10.5 v10.8 [13] v10.0
-1-- 1
---1 1
1-1- 1
.names v2 v3 v6 v7 v9 v10.0 v10.5
       v10.7 v10.10
 v10.12 [13] v10.1
---------1- 1
----------1 1
---1---0--- 1
--1-----1-- 1
0---0-1---- 1
-0--0-1---- 1
----10--0-- 1
.names v6 v7 v8 v9 v10.0 v10.1
v10.3 v10.5 v10.6 v10.12 v10.2
-----0-1-- 1
--1-0-1--- 1
-1-0----0- 1
1---1----0 1
.names v6 v7 v9 v10.6 v10.7 v10.8
[15] v10.3
---1--- 1
1-1---- 1
-0--1-- 1
-----01 1
.names v10.5 v10.7 v10.4
1- 1
-1 1
```

```
.names v6 v7 v8 v10.10 v10.5
--01 1
00-0 1
.names v7 [13] v10.6
11 1
.names v6 v7 v8 v9 v10.6 v10.8
[15] v10.7
-----1- 1
100---- 1
---00-1 1
.names v1 v9 [15] v10.8
111 1
.names v2 v8 v10.10 v10.9
111 1
.names v7 v9 v10.10
01 1
.names v4 v6 v10.6 [13] v10.11
-11- 1
11-1 1
.names v5 v8 v10.10 v10.12
101 1
.names v8 v9 [13]
10 1
.names v6 v7 [15]
01 1
.end
```

(d) netlist.

CHAPTER 3

EFFICIENT SELF-TESTING BERGER CODE CHECKER DESIGN

This chapter presents the design of ***self-testing Berger code*** checkers with the developed partitioning and folding scheme. For a ***Berger code*** B(I,K), if $I = 2^K - 1$ or $2^K - 2$, then it is called a ***maximal length Berger*** (MLB) code. Otherwise, it is a ***non-maximal length Berger*** (NMLB) code. The designs of ***self-testing checkers*** for MLB and NMLB codes are respectively discussed in Sections 3.1 and 3.2.

3.1 MAXIMAL LENGTH BERGER (MLB) CODE CHECKER DESIGN

This first section more specifically discusses the ***Berger code*** checker design with the partitioning scheme presented in [30], and then presents an improved version with a partitioning and folding scheme for MLB codes.

3.1.1 BERGER CODE PARTITIONING SCHEME

Figure 3.1 shows the internal structure of a STC design of MLB code implemented with the partitioning scheme presented in [30]. The checker, referred to as Circuit CH, is comprised of four major blocks: Block H includes the checkers for $C_{(2i-1)/(I+1)}$, $1 \leq i \leq u$. where $u = \lceil (I+1)/2 \rceil$. Each checker takes I + 1 bits from the expanded information part J^* as its input and produces two check output lines; Block G consists of $(2^{K-1} - K)$ AND gates which have as inputs the check bits from the reduced check part P^*; Block Q consists of u - 1 pairs of AND gates; and Block S contains two u-input OR gates. The 1-out-of-2 code in the outputs of Block S indicates an error, if it exists.

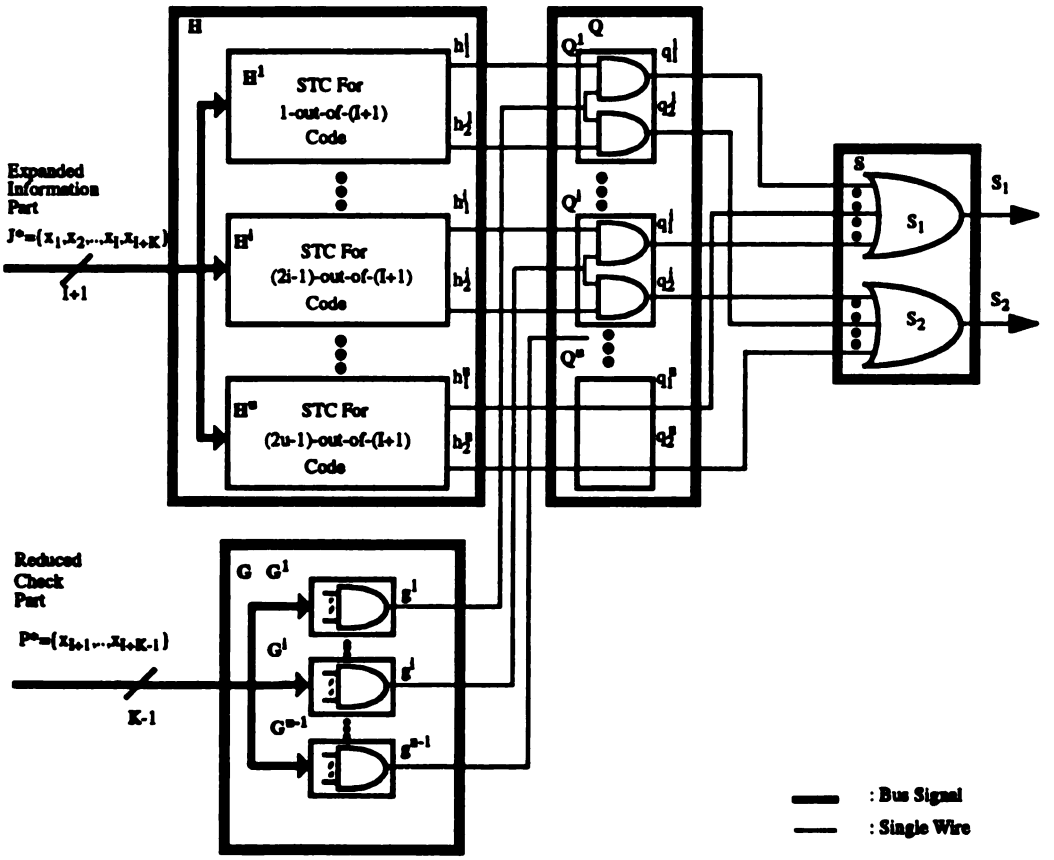


Figure 3.1 Internal structure of a STC design of MLB code B(I,K) (Circuit CH).

Theorem 3.1 Circuit **CH** is a STC Checker.

Prior to proving the properties of ***self-testing*** and ***code-disjoint*** for Theorem 3.1, the following lemmas are needed.

Lemma 3.1: The ***totally self-checking checker*** H^r produces the following code output when a code in a subset E_{2i-1} is applied to Circuit **CH.**

$$(h^r_1, h^r_2) = \begin{cases} (0\ 1) \text{ or } (1\ 0) & \text{if } i = r. \\ (0\ 0) & \text{if } i < r. \\ (1\ 1) & \text{if } i > r. \end{cases}$$

Proof: Consider the outputs h^r_1 and h^r_2 of the checker H^r. Let N_x be the number of 1's in an input codeword. Since $h^r{}_1$ and $h^r{}_2$ are the majority functions as defined in Equation (2.1) and (2.2) of the inputs, the value of (h^r_1, h^r_2) is equal to (1,1) if $N_x > r$; (1,0) or (0,1) if $N_x = r$; or (0,0) if $N_x < r$. For any code in E_{2i-1}, its number of 1's is i, i.e., $N_x = i$. Thus, the lemma results. ♦

Consider the construction of Block G which consists of 2^{K-1} - K AND gates G^i, $1 \le i \le u$. The output of G^i is denoted as g^i and its inputs are the 1-value reduced check bits in P^*_i. In case of $N_1(P^*_i) = 1$, say $x_j = 1$ in P^*_i then $g^i = x_j$ and no AND gate is necessary, where $N_1(A)$, $(N_0(A))$ denotes the number of 1's (0's) in A. For i = u, $N_1(P^*_i) = 0$, it is assumed that $g^u = 1$ [30]. Let X and Y be two N-tuple. We say that X ***covers*** Y if and only if X has 1's everywhere Y has 1's. If neither Y ***covers*** X, nor X ***covers*** Y, then we say X and Y are ***unordered.***

Lemma 3.2: (a) P^*_i never ***covers*** P^*_j if $j < i$;

(b) If $g^i = 1$ and P^*_i ***covers*** P^*_j, then $g^j = 1$; and

(c) If $g^i = 1$ and P^*_i and P^*_j are unordered, then $g^j = 0$.

Proof: (a) If $j < i$, then $N_1(J_j) < N_1(J_i)$, or $N_1(P_j) > N_1(P_i)$, i.e., P^*_j has more 1's than P^*_i. Thus, P^*_i never ***covers*** P^*_j ; (b) By definition, P^*_i ***covers*** P^*_j means that all inputs of G^j are

also inputs of G^i. Thus, if $g^i = 1$, i.e. the input variables of G^i are all 1's, then $g^j = 1$; and (c) Since P_i^* and P_j^* are unordered, there exists, at least, one input variable of G^j, say x_r, which is not an input of G^i, where $x_r = 0$ (because $g^i = 1$). Thus $x_r = 0$ causes the AND gate G^j to produces $g^j = 0$. ♦

Lemma 3.3: When a code including the reduced check part P_i^* is applied to Block G, the outputs are

$$g^j = \begin{cases} 1 & \text{for } j = i; \text{ or } j > i \text{ and } P_i^* \textit{ covers } P_j^*. \\ 0 & \text{for } j < i; \text{ or } j > i \text{ and } P_i^* \text{ and } P_j^* \text{ are unordered.} \end{cases}$$

Proof: When a code including P_i^* is applied to Circuit CH, the AND gate G^i produces $g^i = 1$, i.e., $g^j = 1$ if $j = i$. For the case of $j > i$ and P_i^* ***covers*** P_j^*, by Lemma 3.2 (b), $g^j = 1$; If $j < i$, by Lemma 3.2 (a), P_i^* never ***covers*** P_j^*, therefore $g^j = 0$. Finally, if P_i^* and P_j^* are unordered, and $j > i$, by Lemma 3.2 (c), $g^j = 0$. ♦

Lemma 3.4: Circuit CH is ***self-testing*** for all unidirectional faults.

Proof: Two types of faults are identified: stuck-at-1 (s/1) and stuck-at-0 (s/0) faults. In general, a noncode output (1,1) in Q^i results in a noncode output (1,1) in Block S which indicates an error. On the other hand, if all Q^i's produce the same noncode output (0,0), this results in a noncode output (0,0) in S that indicates an error. Thus, the application of a code in E_{2i-1} can detect not only the s/1 fault(s) at the output(s) of H^i, Q^i, and/or S, but also the s/0 faults at the output(s) of G^i, H^i, Q^i, and/or S. Finally, the s/1 faults at g^i's can be detected by applying a code in E_{2u-1}. When this code is applied to Circuit CH, all checkers produce the same noncode output (1,1), except that H^r generates a code output. Since $g^r = 0$, for $r < u$, and $g^u = 1$ for fault-free circuit, the s/1 faulty g^i will force Q^i to produce a noncode output (1,1). Thus, Circuit CH is ***self-testing*** for all unidirectional errors. ♦

Lemma 3.5: Circuit CH is ***code-disjoint***.

Proof: When a code in subset E_{2i-1} is applied to Circuit CH, the code input produces a

code output in Block S, i.e., all code inputs produce all code outputs. Consider a noncode, without loss of generality, consisting of J_q^* and P_r^*, where $r \neq q$. When the noncode input is applied to Circuit **CH**, by Lemma 3.1, the checker H^i produces a noncode (0,0), for $i > q$, or (1,1), for $i < q$. Since $g^i = 0$, for $i > r$, and $g^r = 1$, a noncode output (1,1) is produced in Q^r, if $r > q$, and further generates a noncode output in Block S. On the other hand, if $r < q$, all outputs in Block Q are (0,0)'s and result in a noncode output (0,0) in Block S. This concludes that the noncode inputs produce the noncode outputs. ♦

Proof of Theorem 3.1 Based on Lemmas 3.4 and 3.5, Circuit **CH** is a STC checker.

3.1.2 PARTITIONING AND FOLDING SCHEME

The basic concept behind the ***Berger code*** partitioning scheme is that the LSB of the check part is grouped with the information part to reduce the number of checkers used in a STC design. This section presents an alternative structure that further reduces the number of checkers. We first consider the MLB code with $I = 2^K - 1$. The same concept can be extended for $I = 2^K - 2$ and 2^{K-1}.

A. B(I,K) with $I = 2^K - 1$

As defined in Table 2.1(b), E_t, $t = 1, 3, 5$, and 7, is a subset of B(I,K) that replaces the adjacent pair of subset D_t and D_{t+1} in Table 2.1(a). In general, a B(I,K) is composed of subsets E_{2t-1}'s, $1 \leq t \leq u$. Note that $N_1(J_t) = t$ and $N_0(J_t) = 2u - t$.

Definition 3.1: Let E_s and E_t be two subsets of B(I,K), the pair of E_s and E_t is ***dual*** if and only if the bit-complement of any codeword in E_s is a codeword in E_t and vice versa. Thus, we say that $\overline{E}_s = E_t$ and $\overline{E}_t = E_s$.

Consider the MLB codes B(I,K) with $I = 2^K - 1$, i.e., $u = 2^{K-1}$. Let E_m and E_{2u-m} be any two subsets of B(I,K). Unless otherwise stated, it is assumed that $m < u$.

Lemma 3.6: The pair of subsets E_m and E_{2u-m} is dual.

Proof: For any codeword c_m in E_m, $N_1(c_m) = m$ and $N_0(c_m) = 2u - m$. Let c_p be the bit-complement of c_m, then we have $N_1(c_p) = N_0(c_m) = 2u - m$ and $N_0(c_p) = N_1(c_m) = m$. This implies that c_p is a codeword in E_{2u-m}. Similarly, the complement of any codeword in E_{2u-m} is a codeword in E_m. Thus, the pair of E_m and E_{2u-m} is dual. ♦

Since the subset J_1^* in Table 2.1(b) is a collection of 1-out-of-8 codewords and J_7^* consists of all 7-out-of-8 codewords, the pair of subsets E_1 and E_7 is dual; Similarly, the pair of E_3 and E_5 is also dual. Without loss of generality, subsets E_5 and E_7 in Table 2.1(b) can be re-written as shown in Table 3.1.

Table 3.1: Subsets of B(7,3).

Subset	expanded information part $x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_{10}$	reduced check part $x_8\ x_9$	partitioning
E_5	0 0 0 1 1 1 1 1	0 1	$C_{5/8} \times (0\ 1)$
E_7	0 1 1 1 1 1 1 1	0 0	$C_{7/8} \times (0\ 0)$

Let J_t^* and P_t^* be the expanded information part and the reduced check part of E_t, respectively, and $x_{MSB(t)}$ denotes the MSB of P_t^*. It is obvious that $x_{MSB(t)} = 1$ and $x_{MSB(2u-t)} = 0$, if $t < u$; and $x_{MSB(t)} = 0$ and $x_{MSB(2u-t)} = 1$, if $t > u$. Let A be a set and x be a bit, then we define the set $A \oplus x = \{y \mid y = z \oplus x, \text{ for all } z \in A\}$.

Lemma 3.7: If $F_t = E_{2u-t} \oplus x_{MSB(t)}$, then $F_t = F_{2u-t}$.

Proof: For $t < u$, since $x_{MSB(t)} = 1$ and $x_{MSB(2u-t)} = 0$, we have $F_{2u-t} = E_t \oplus x_{MSB(2u-t)} = E_t$ and $F_t = E_{2u-t} \oplus x_{MSB(t)} = \overline{E}_{2u-t} = E_t$, i.e., $F_{2u-t} = F_t$. On the other hand, for $t > u$, since $x_{MSB(2u-t)} = 1$ and $x_{MSB(t)} = 0$, hence, $F_t = E_{2u-t}$ and $F_{2u-t} = \overline{E}_t = E_{2u-t}$, i.e., $F_t = F_{2u-t}$. ♦

Since, by Lemma 3.6, $E_{2u-t} = \overline{E}_t$, the condition in Lemma 3.7 can be written as

$$F_t = E_{2u-t} \oplus x_{MSB(t)} = \overline{E}_t \oplus x_{MSB(t)} = \overline{E_t \oplus x_{MSB(t)}} \tag{3.1}$$

which is a XNOR (exclusive-NOR) function.

Theorem 3.2: If $m < u$, then any unidirectional errors in a subset $Z_m = \overline{E_m \oplus x_{MSB(m)}}$ can be detected by the $C_{m/(I+1)}$ checker.

Proof: Since $m < u$, $x_{MSB(m)} = 1$ implies $Z_m = \overline{E}_m$. By Lemma 3.6, $Z_m = E_{2u-m}$. Obviously, the checker $C_{(2u-m)/(I+1)}$ detects all unidirectional errors in E_{2u-m}, or Z_m. Similarly, for (b), $Z_m = E_m$, any unidirectional errors in Z_m can be detected by $C_{m/(I+1)}$ checker. ♦

Table 3.2 (a) is the same as Table 2.1 (b) except E_5 and E_7 are reordered, where $x_{MSB(1)} = x_{MSB(3)} = 1$ and $x_{MSB(5)} = x_{MSB(7)} = 0$. By Lemma 3.7, $F_1 = F_7$ and $F_3 = F_5$. As illustrated in Table 3.2 (c), Z_1 replaces the pair of identical subsets F_1 and F_7, while Z_3 substitutes the pair of F_3 and F_5. Tables 3.2(c) and 3.2(d) list two possible partitions. By Theorem 3.2, the code B(7,3) can be written as

$$B(7,3) = C_{1/8} \times (1) \cup C_{3/8} \times (0) \tag{3.2}$$

The number of ***m/n*** checkers in the STC design for B(7,3) is reduced from four to two.

Table 3.2: B(7,3) with Partitioning Scheme.

(a)

Subset	Expanded information part $x_1x_2x_3x_4x_5x_6x_7x_{10}$	Reduce check part x_8x_9	partitioning
E_1	0 0 0 0 0 0 0 1	1 1	$C_{1/8}\times(1\ 1)$
E_3	0 0 0 0 0 1 1 1	1 0	$C_{3/8}\times(1\ 0)$
E_5	1 1 1 1 1 0 0 0	0 1	$C_{5/8}\times(0\ 1)$
E_7	1 1 1 1 1 1 1 0	0 0	$C_{7/8}\times(0\ 0)$

(b)

Subset	Expanded information part $x_1x_2x_3x_4x_5x_6x_7x_{10}$	Reduce check part x_9	x_{MSB} x_8
F_1	0 0 0 0 0 0 0 1	1	1
F_7	1 1 1 1 1 1 1 0	0	0
F_3	0 0 0 0 0 1 1 1	0	1
F_5	1 1 1 1 1 0 0 0	1	0

(c)

Subset	Expanded information part $x_1x_2x_3x_4x_5x_6x_7x_{10}$	Reduce check part x_9	partitioning
Z_1	0 0 0 0 0 0 0 1	1	$C_{1/8}\times(1)$
Z_3	0 0 0 0 0 1 1 1	0	$C_{3/8}\times(0)$

(d)

Subset	Expanded information part $x_1x_2x_3x_4x_5x_6x_7x_{10}$	Reduce check part x_9	partitioning
Z_5	1 1 1 1 1 0 0 0	1	$C_{5/8}\times(1)$
Z_7	1 1 1 1 1 1 1 0	0	$C_{7/8}\times(0)$

In general, a ***Berger code*** B(I,K) implementing with the partitioning scheme can be formally written with the union operator as

$$B(I,K) = \bigcup_{i=1}^{\upsilon} C_{(2i-1)/(I+1)} \times P^{\#}_{2i-1}, \tag{3.3}$$

where $P_t^{\#} = P_t^{*} - \{x_{MSB(t)}\}$ and $\upsilon = \lceil u/2 \rceil$. (3.4)

Figure 3.2 illustrates a ***Berger code*** checker implemented with the presented partitioning scheme, referred to as circuit CS which consists of two major parts: ***Code-complementer*** and ***Checker***. The code-complementer needs (I + K - 1) XNOR gates, while the checker is comprised of Blocks H, G, Q, and S:

Block H: checkers for $C_{(2i-1)/(I+1)}$, $1 \le i \le \upsilon$;

Block G: $[2^{K-2} - (K - 1)]$ AND gates;

Block Q: $(\upsilon - 1)$ pairs of AND gates; and

Block S: two υ-input OR gates.

Compared to the STC design in [30], the hardware reduction in the presented STC design includes υ's ***m/n*** checkers for $C_{(2i-1)/(I+1)}$, $\upsilon+1 \le i \le u$, in Block H; $(2^{K-2}-1)$ AND gates in Block G; 2^{K-1} AND gates in Block Q; and the number of inputs of the OR gates in Block S is reduced by half.

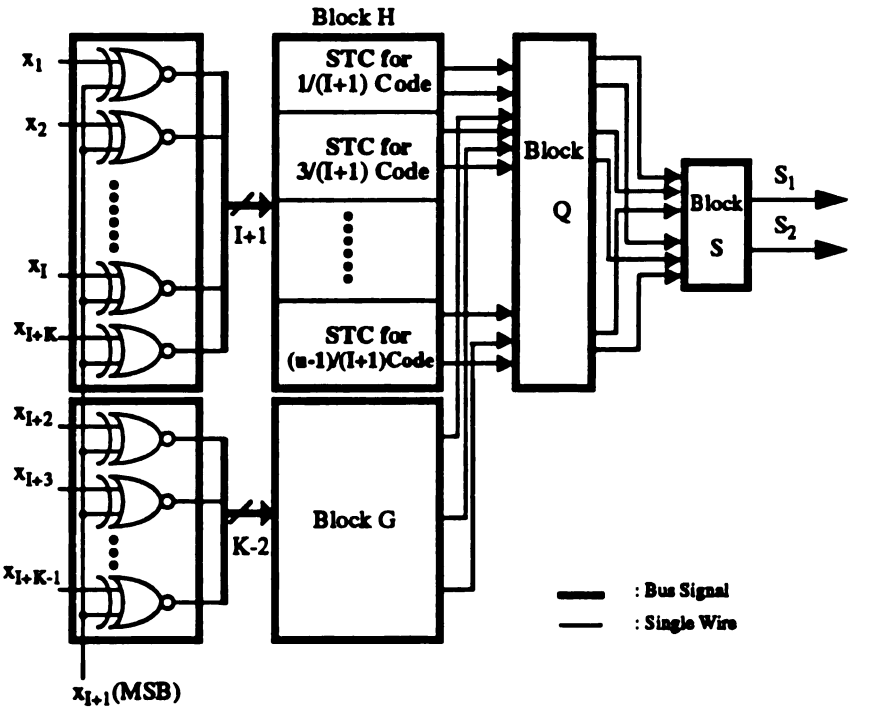
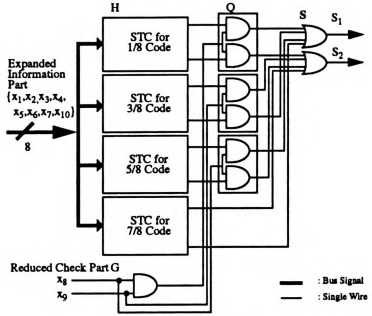
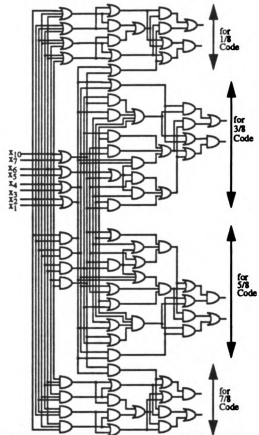


Figure 3.2 Circuit CS for B(I,K) STC design.

In order to demonstrate the effectiveness of the presented ***Berger code*** checker design, the STC design for B(7,3) implemented with the partitioning scheme in [30] is shown in Figure 3.3 (a), where the checker set $S_c = \{C_{1/8}, C_{3/8}, C_{5/8}, C_{7/8}\}$. The STC for $C_{1/8}$ can be obtained by using a translator of the 1/8 code into the 2/5 code, a translator of the 2/5 code into the 2/4 code, and a STC 2/4 checker; the STC for $C_{3/8}$ consists of a translator of 3/8 code into 2/4 code and a STC 2/4 checker [30]; and the STC for $C_{5/8}$ is comprised of a translator of 5/8 code into 2/4 code and a STC 2/4 checker. The STC for $C_{7/8}$ requires a translator of the 7/8 code into the 2/5 code, a translator of the 2/5 code into the 2/4 code, and a STC 2/4 checker. The logic expressions of the STC checkers are given in Table 3.3. Figure 3.3 (b) shows the schematic diagram of the check circuit based on the



(a) with partitioning scheme.



(b) logic implementation of block **H** in (a).

Figure 3.3 Circuit CH for B(7,3) STC design. [30].

Table 3.3: Logic Expressions for Checker in B(7,3).

$C_{1/8}$ Checker

$$h_1^{11} = (x_1 + x_2) + (x_7 + x_{10});\ h_2^{11} = x_5 + x_6 + x_{10};\ h_3^{11} = x_3 + x_4 + x_7$$

$$h_4^{11} = x_2 + x_3 + x_5;\ h_5^{11} = x_1 + x_4 + x_6$$

$$h_1^{12} = h_1^{11};\ h_2^{12} = h_2^{11} + h_3^{11};\ h_3^{12} = h_4^{11} + h_5^{11};\ h_4^{12} = h_2^{11} h_3^{11} + h_4^{11} h_5^{11}$$

$$h_1^{13} = h_1^{12} h_2^{12} + h_3^{12} h_4^{12};\ h_2^{13} = (h_1^{12} + h_2^{12})(h_3^{12} + h_4^{12})$$

$C_{3/8}$ Checker

$$h_1^{21} = (x_1 + x_2) + (x_3 + x_4)$$

$$h_2^{21} = x_5x_6 + (x_5 + x_6)(x_7 + x_{10}) + x_7x_{10} + (x_1 + x_2)(x_3x_4)$$

$$h_3^{21} = (x_5 + x_6)(x_7x_{10}) + (x_1x_2)(x_3 + x_4) + [(x_5 + x_6) + (x_7 + x_{10})]\,(x_1 + x_2)(x_3 + x_4)$$

$$h_4^{21} = (x_5x_6)(x_7 + x_{10}) + (x_3x_4)[(x_5 + x_6) + (x_7 + x_{10})] + (x_1x_2)[(x_5 + x_6) + (x_7 + x_{10})]$$

$$h_1^{23} = h_1^{22} h_2^{22} + h_3^{22} h_4^{22};\ h_2^{23} = (h_1^{22} + h_2^{22})(h_3^{22} + h_4^{22})$$

$C_{5/8}$ Checker

$$h_1^{31} = (x_1x_2)(x_3x_4)$$

$$h_2^{31} = (x_5 + x_6)[x_5x_6 + x_7x_{10}](x_7 + x_{10})[x_1x_2 + (x_3 + x_4)]$$

$$h_3^{31} = [x_5x_6 + x_7x_{10}][(x_1 + x_2) + (x_3x_4)][(x_5x_6)(x_7x_{10}) + x_1x_2 + x_3x_4]$$

$$h_4^{31} = [(x_5 + x_6) + x_7x_{10}][(x_3 + x_4) + (x_5x_6)(x_7x_{10})][(x_1 + x_2) + (x_5x_6)(x_7x_{10})]$$

$$h_1^{33} = h_1^{32} h_2^{32} + h_3^{32} h_4^{32};\ h_2^{33} = (h_1^{32} + h_2^{32})(h_3^{32} + h_4^{32})$$

$C_{7/8}$ Checker

$$h_1^{41} = (x_1x_2)(x_7x_{10});\ h_2^{41} = x_5x_6x_{10};\ h_3^{41} = x_3x_4x_7$$

$$h_4^{41} = x_2x_3x_5;\ h_5^{41} = x_1x_4x_6$$

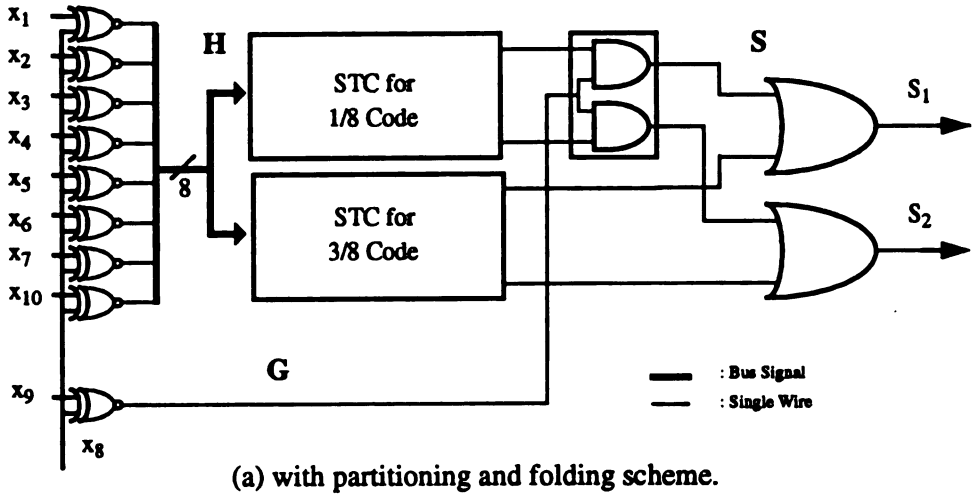
$$h_1^{42} = h_1^{41};\ h_2^{42} = h_2^{41} + h_3^{41};\ h_3^{42} = h_4^{41} + h_5^{41};\ h_4^{42} = h_2^{41} h_3^{41} + h_4^{41} h_5^{41}$$

$$h_1^{43} = h_1^{42} h_2^{42} + h_3^{42} h_4^{42};\ h_2^{43} = (h_1^{42} + h_2^{42})(h_3^{42} + h_4^{42})$$

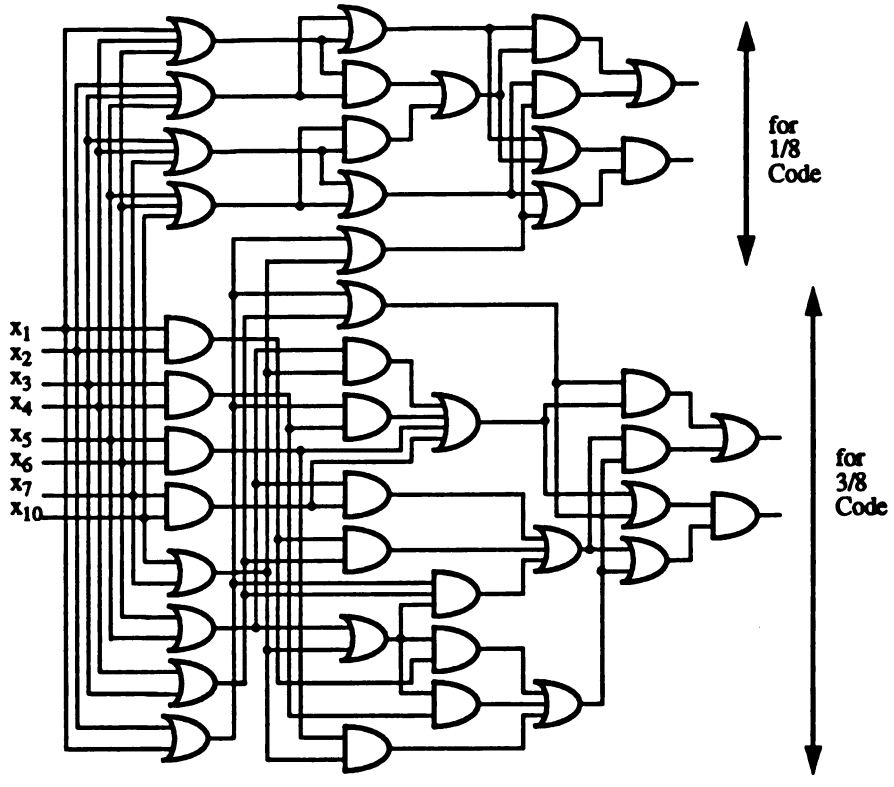
expressions. Results show that the checker circuit requires 78 gates and 6 gate levels. In addition, Blocks G, Q, and S need 1, 6, and 2 gates, respectively. Overall, the STC design for B(7,3) requires 87 gates and 8 gate levels with the cost-effective implementation [30]. Compared to 17 gate levels and 40 gates in [28] and 11 gate level and 58 gates in [29], the checker design [30] offers an improvement in delay.

Figure 3.4 illustrates the schematic diagram of the presented STC design for B(7,3). The checker subset $\{C_{1/8},C_{3/8}\}$ requires 43 gates and 6 gate levels and Blocks G, Q, and S use only 4 gates. Thus, the B(7,3) checker needs 47 gates and 8 gates levels for the checker circuit and 9 XNOR gates for the code-complementer. Since the number of transistors and the delay in a transmission gate XNOR gate is almost the same as those in a AND/OR gate in CMOS technology [64], it is reasonable to assume that a XNOR gate takes one unit of gate count with one unit of delay. In other words, the B(7,3) checker requires only a total of 56 gates. Comparing to 87 gates for the STC design in [30], the reduction in gate count is nearly 35%. For the minimal-level implementation, the partitioning scheme requires 137 gates to realize the STC design [30], while the presented design requires only 77 gates. For B(15,4), the STC design in [30] requires 315 gates for cost-effective implementation, while our scheme requires only 174 gates. The reduction is can be 45%. As the ***Berger code*** length is increased, it is predicted that the reduction is as much as 50%

In order to show the ***self-testing*** property of the presented checker design, it is necessary to show that both the code-complementer circuit and the checker circuit are ***self-testing***. As shown in Figure 3.2, the code-complementer circuit computes in parallel the XNOR functions. It has been shown that the circuit that computes in parallel the XOR (or XNOR) functions is a TSC checker for all codeword inputs under single faults [4]. Thereby, the following lemma results.



(a) with partitioning and folding scheme.



(b) logic implementation of block H in (a).

Figure 3.4 Circuit CS for B(7,3) STC design.

Lemma 3.8: The code-complementer circuit is a TSC circuit for all single faults.

Proof: Consider the code-complementer in Figure 3.2, where XNOR gates are connected in parallel. Let A_i and B_i be two inputs of the a XNOR gate and C_i be its output, where $A_i = x_i$, $B_i = x_{I+1}$, and $C_i = \overline{A_i \oplus B_i} = \overline{x_i \oplus x_{I+1}}$, $1 \leq i \leq I+k$. Since $(A_i,B_i) = (1,0)$ and $(0,1)$ can detect any single stuck-at faults at the inputs and outputs of an XNOR gate, the code-complementer circuit is ***self-testing***. With the properties of FS and ST, the circuit is TSC for all single faults. ♦

In fact, some multiple unidirectional faults in the code-complementer circuit are still detectable. More specifically, if given a codeword input, the multiple unidirectional faults do not cause the code-complementer circuit to produce a codeword output, then those faults are detectable.

Lemma 3.9: When a code in subset E_{2i-1} is applied to Block H of Circuit CS, the ***totally self-checking*** (TSC) checker H^r produces a code output (0,1) or (1,0), if $r = i$; or generates a noncode (0,0), if $r > i$, or (1,1), if $r < i$.

Proof: This can be easily seen from the construction of TSC checkers as proof in Lemma 3.1. ♦

Recall that the checkers in Circuit CS are the TSC checkers that detect all unidirectional faults. When a code in E_{2i-1} is applied to the fault-free Circuit CH, by Lemma 3.9, H^i produces a code output (0,1) or (1,0), and H^r generates a noncode output (0,0), if $r > i$, or (1,1), if $r < i$. By Lemma 3.1, $g^r = 0$, for $r < i$, and $g^i = 1$. Thus, only Q^i produces a code output and others generate the noncode output (0,0).

Theorem 3.3: Circuit CS is a STC checker.

Proof: Similar to the proof of Lemma 3.4 and Lemma 3.5, Circuit CS can be shown as ***self-testing*** for all unidirectional faults and ***code-disjoint***. Thus, it is a STC checker. ♦

The patterns required to test Circuit CS are mainly determined by those used for testing the TSC checkers in Block H. For example, consider the STC design of B(7,3) shown in Figure 3.4. where Block H consists of the STC checkers for 1/8 and 3/8 codes. Thus, it requires the following test patterns:

for 1/8 codes	for 3/8 codes
$(x_1x_2x_3x_4x_5x_6x_7x_{10}\text{-}x_8x_9)$	$(x_1x_2x_3x_4x_5x_6x_7x_{10}\text{-}x_8x_9)$
1 0 0 0 0 0 0 0 1 1	1 1 1 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 1 1	0 1 1 1 0 0 0 0 1 0
0 0 1 0 0 0 0 0 1 1	0 0 1 1 1 0 0 0 1 0
0 0 0 1 0 0 0 0 1 1	0 0 0 1 1 1 0 0 1 0
0 0 0 0 1 0 0 0 1 1	0 0 0 0 1 1 1 0 1 0
0 0 0 0 0 1 0 0 1 1	0 0 0 0 0 1 1 1 1 0
0 0 0 0 0 0 1 0 1 1	1 0 0 0 0 0 1 1 1 0
0 0 0 0 0 0 0 1 1 1	1 1 0 0 0 0 0 1 1 0
1 1 1 1 1 1 1 0 0 0	0 0 0 1 1 1 1 1 0 1

B. B(I,K) with $I = 2^K - 2$

Similar to the partitioning procedure discussed in Table 2.1, Table 3.4 shows the B(6,3) with the partitioning scheme. The ***Berger code*** can be represented by

$$B(6,3) = C_{1/7} \times (11) \cup C_{3/7} \times (10) \cup C_{5/7} \times (01) \cup C_{7/7} \times (00). \qquad (3.5)$$

i.e., the implementation requires four checkers for $C_{1/7}$, $C_{3/7}$, $C_{5/7}$, and $C_{7/7}$. However, since neither the pair of E_1 and E_7, nor the pair of E_3 and E_5, is dual, the partitioning and folding scheme cannot be applied directly. Moreover, the problem can be resolved by adding an extra bit, $x_0 = 0$, as the MSB of the information part, i.e., to modify a given B(I,K) with $I = 2^K - 2$ to as a B(I,K) with $I = 2^K - 1$ so that the STC design presented in the previous discussion can be applied. As such, each E_t in Table 3.4(d) is comprised of all $t/(I+2)$ codes,

Table 3.4: Berger Code B(6,3).

(a)

Subset	information part $x_1\ x_2\ x_3\ x_4\ x_5\ x_6$	check part $x_7\ x_8\ x_9$
D_1	0 0 0 0 0 0	1 1 1
D_2	0 0 0 0 0 1	1 1 0
D_3	0 0 0 0 1 1	1 0 1
D_4	0 0 0 1 1 1	1 0 0
D_5	0 0 1 1 1 1	0 1 1
D_6	0 1 1 1 1 1	0 1 0
D_7	1 1 1 1 1 1	0 0 1

(b)

Subset	expanded information part $x_1x_2x_3x_4x_5x_6x_9$	reduced check Part $x_7\ x_8$
D_1	0 0 0 0 0 0 0 1	1 1
D_2	0 0 0 0 0 0 1 0	1 1
D_3	0 0 0 0 1 1 1	1 0
D_4	0 0 0 1 1 1 0	1 0
D_5	0 0 1 1 1 1 1	0 1
D_6	0 1 1 1 1 1 0	0 1
D_7	1 1 1 1 1 1 1	0 0

(c)

Subset	expanded information part $x_1x_2x_3x_4x_5x_6x_9$	reduced check part $x_7\ x_8$
E_1	0 0 0 0 0 0 1	1 1
E_3	0 0 0 0 1 1 1	1 0
E_5	0 0 1 1 1 1 1	0 1
E_7	1 1 1 1 1 1 1	0 0

(d)

Subset	expanded information part $x_0x_1x_2x_3x_4x_5x_6x_9$	reduced check part $x_7\ x_8$
E_1	0 0 0 0 0 0 0 1	1 1
E_3	0 0 0 0 0 1 1 1	1 0
E_5	0 0 0 1 1 1 1 1	0 1
E_7	0 1 1 1 1 1 1 1	0 0

(e)

Subset	expanded information part $x_0x_1x_2x_3x_4x_5x_6x_9$	reduced check part x_8	partitioning
Z_1	0 0 0 0 0 0 0 1	1	$C_{1/8}\times(1)$
Z_3	0 0 0 0 0 1 1 1	0	$C_{3/8}\times(0)$

and the pair of subsets E_t and E_{2u-t} is dual. Therefore, Equation (3.3) can be re-written as follows for $u = 2^{K-1}$, i.e., $I = 2^K - 1$, or $2^K - 2$,

$$B(I,K) = \bigcup_{i=1}^{v} C_{(2i-1)/I+2} \times P^{\#}_{2i-1} \tag{3.6}$$

Thus, the ***Berger code*** B(6,3), as shown in Table 3.4(e), can be written as

$$B(6,3) = C_{1/8} \times (1) \cup C_{3/8} \times (0) \tag{3.7}$$

The number of ***m/n*** checkers is reduced by half.

C. B(I,K) with $I = 2^{K-1}$

The ***Berger code*** B(I,K) with $I = 2^{K-1}$ can be the concatenation of a MLB code with the 1/2 code [30]. The 1/2 codewords occur on x_1, the LSB of the information part, and x_{I+K}, the MSB of the check part. On the bits $\{x_i | 2 \leq i \leq I+K-1\}$ occur the words of the MLB code. Figure 3.5 shows a schematic diagram of a STC design of B(I,K) with $I = 2^{K-1}$. The STC design requires only 6 gates more than that for $I = 2^K - 1$ in case (A).

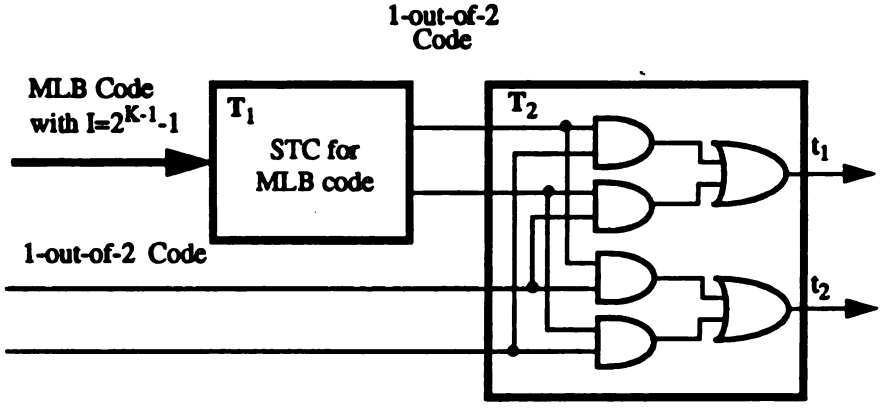


Figure 3.5 STC for modified Berger code with $I=2^{K-1}$.

3.2 NON-MAXIMAL LENGTH BERGER CODE (NMLB) CHECKER DESIGN

The partitioning scheme in [30] offers an improvement in delay. However, the implementation is available only for MLB codes. This section presents an extension to the NMLB codes. We first show that a NMLB code B(I,K) with the partitioning scheme in [30] is not a STC checker, and then present a STC checker design for NMLB codes. In addition, the partitioning and folding scheme presented in the previous section can also be applied to further reduce hardware cost.

3.2.1 STC DESIGN WITH PARTITIONING SCHEME

Consider Circuit CK for the NMLB code B(I,K) with $u < 2^{K-1}$. For the reduced check parts, let SPC = $\{P^*_{2i-1} \mid 1 \le i \le u-1\}$ be the set of code vectors and SPNC = $\{P^*_{2i-1} \mid u \le i \le 2^{K-2}\}$ be the non-codewords. Consider a noncode input which is comprised of J_{2u-1} and P^*_r, where $P^*_r \in$ SPNC. By Lemma 3.1, the checker H^i, $i < u$, produces a noncode output (1,1); and the checker H^u generates a code output. Since $g^i = 0$, for all $i < u$, and $g^u = 1$, all Q^i's have the non-code output (0,0), except that Q^u produces a code output. As a result, Block S produces a code output. In other words, a noncode input may result in a code output. Thus, circuit CK is not ***code-disjoint***. This implies that circuit CK is not a STC.

The code-disjointness of circuit CK can be improved by adding some circuits that detect the noncode $P^*_r \in$ SPNC and then connecting the output signal of these circuits to both OR gates S^1 and S^2. If $P^*_r \in$ SPC, then $g^r = 0$ does not affect the outputs of the OR gates. On the other hand, if $P^*_r \in$ SPNC, then $g^r = 1$ results in a noncode output (1,1) in S, i.e., a noncode output results when a noncode input is applied. However, such circuits cannot be realized to be ***code-disjoint***. Thus, circuit CK is not a STC. Moreover, the problem can be resolved by the folding encoding scheme, where the check part P_t is the binary encoding of $(I - N_1(J_t))$. A ***Berger code*** B(I,K) can be encoded as either

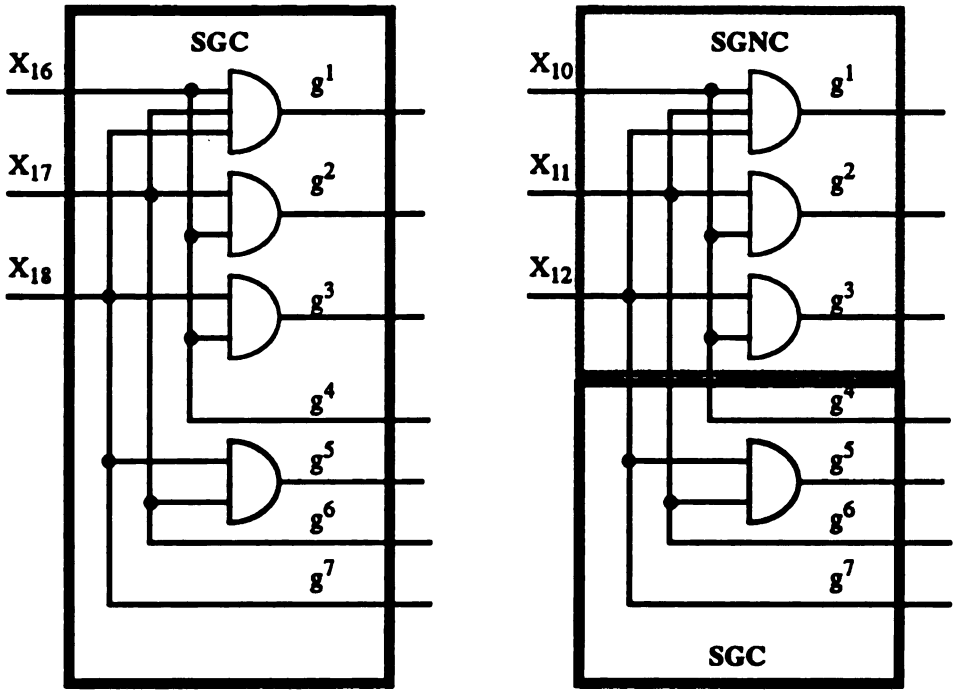
$P_t = (2^K - 1) - N_1(J_t)$, or $P_t = I - N_1(J_t)$, where $1 \leq t \leq I+1$. Both encoding schemes are to map the (I + 1) codes for the check parts into some of the 2^K K-bit codes. Let a K-bit code sequence $<2^K-1, 2^K-2, ..., 0>$ where 0 and $2^K - 1$ are the ***rightmost*** and ***leftmost*** codes of the sequence, respectively. For simplicity, the former encoding scheme is referred to as ***left-justified*** mapping approach, while the later scheme as ***right-justified.*** Since all K-bit codes are included in a MLB code, i.e., $I = 2^K - 1$, both ***left-justified*** and ***right-justified*** schemes produce the same MLB codes.

Table 3.5 shows the ***Berger codes*** B(15,4) and B(9,4) that are encoded by the ***right-justified*** scheme. Let $v = 2^K - u$ and $I_v = \{1,2,...,v\}$ be an index set. Based on the ***right-justified*** scheme, the reduced check part have SPNC = $\{P^*_{Ni} \mid i \in I_v\}$ and SPC = $\{P_{2j-1} \mid j = i - v$ and $v+1 \leq i \leq u\}$. Note that g^i is the output of the AND gate G^i which is constructed from P^*_{Ni} or P^*_i, for simplicity, let SGNC = $\{g^i \mid i \in I_v\}$ and SGC = $\{g^i \mid v+1 \leq i \leq u-1\}$. Figure 3.6 illustrates the circuit for Block G for B(15,4) and B(9,4), where SGNC = $\{g^1, g^2, g^3\}$ and SGC = $\{g^4, g^5, g^6, g^7\}$ in B(9,4).

Table 3.5: Left-justified Encoding Scheme.

B(15,4)	
$N_1(J_t^*)$	P_t^*
1	1 1 1
3	1 1 0
5	1 0 1
7	1 0 0
9	0 1 1
11	0 1 0
13	0 0 1
15	0 0 0

B(9,4)	
$N_1(J_t^*)$	P_t^*
1	1 0 0
3	0 1 1
5	0 1 0
7	0 0 1
9	0 0 0



(a) B(15,4) (b) B(9,4)

Figure 3.6 Function G for B(15,4) and B(9,4) STC design.

Figure 3.7 shows the internal structure, referred to as Circuit CW, of a STC design of ***Berger code*** B(I,K) with any code length. The structure is similar to that in Figure 3.1. Block H includes checkers for $C_{(2i-1)/(I+1)}$, $1 \leq i \leq u$; Block G consists of 2^{K-1}-K AND gates; Block Q consists of u-1 pairs of AND gates; and Block S contains two (2^{K-1})-input OR gates. Figure 3.8 illustrates the STC design for B(9,4).

It should be mentioned that, for $u < 2^{K-1}$, SGNC is not a null set. As shown in Figure 3.7, every $g^i \in$ SGC is connected to Block Q, while every $g^j \in$ SGNC is simultaneously connected to the OR gates S^1 and S^2.

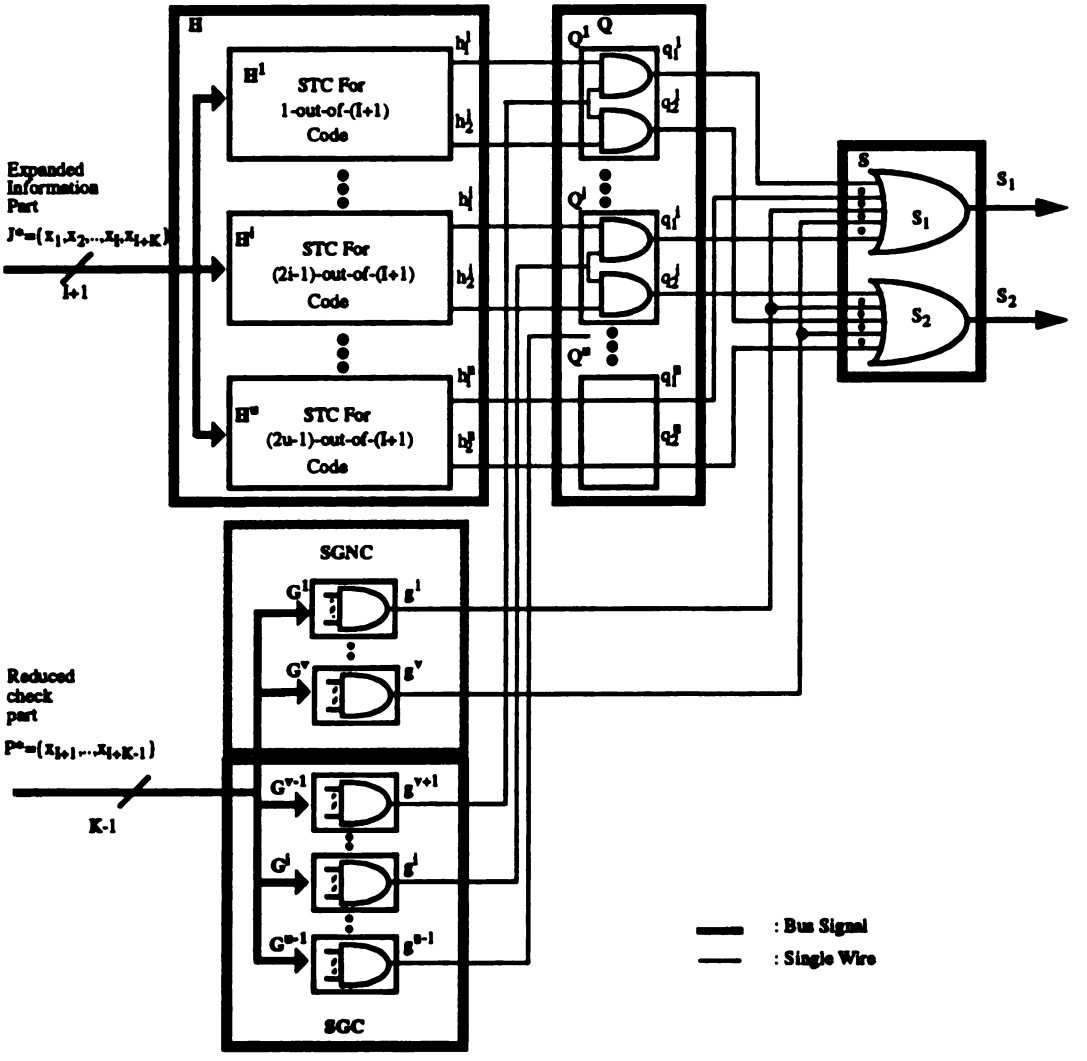


Figure 3.7 Circuit CW for B(I,K) STC design.

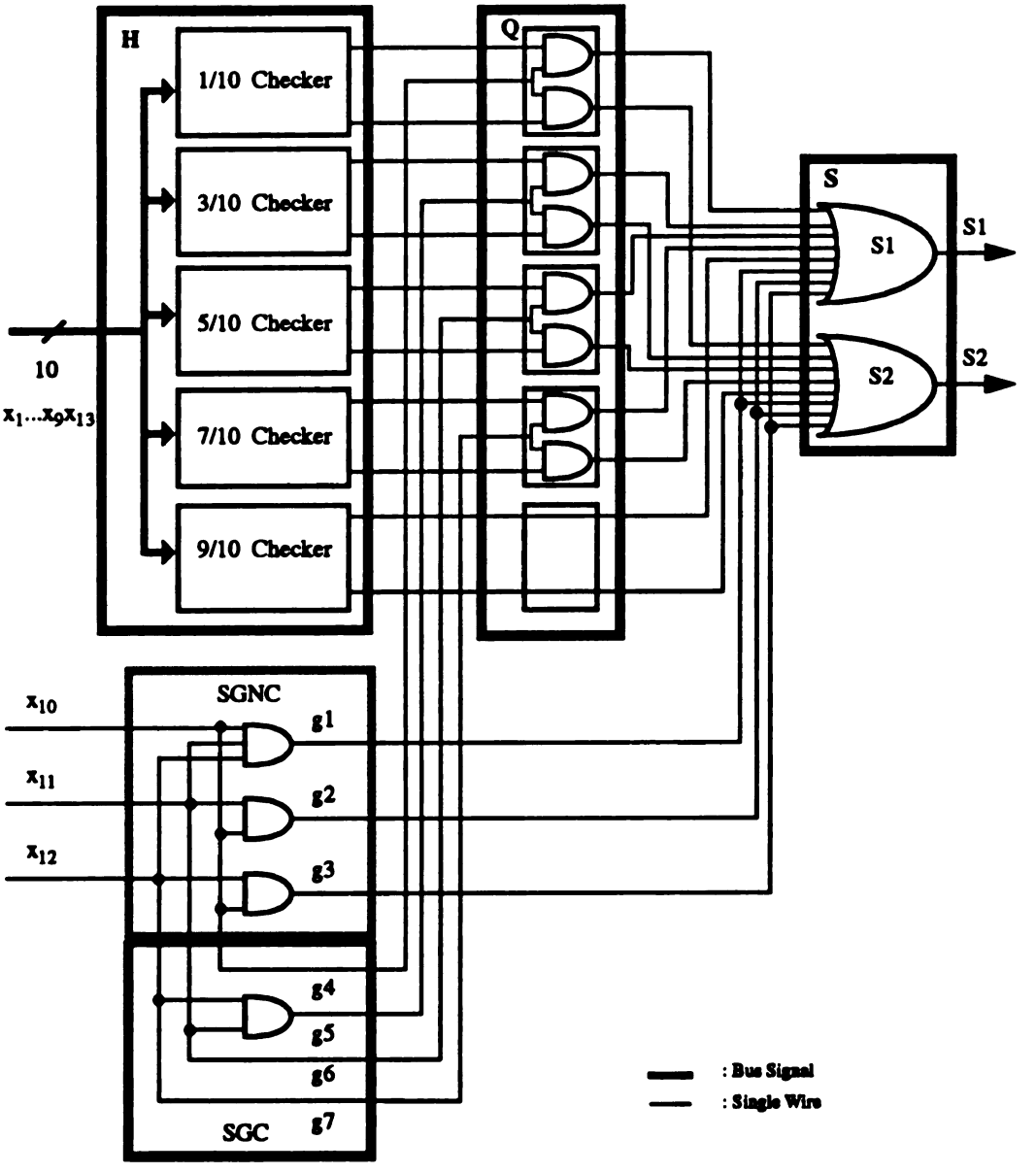


Figure 3.8 Circuit CW for B(9,4) STC design.

Lemma 3.10: Circuit CW for $u = 2^{K-1}$ is a STC.

Proof: Circuit CW with $u = 2^{K-1}$ is virtually the same as Circuit CH. By Theorem 3.3, the circuit is a STC. ♦

Lemma 3.11: Circuit CW for $u < 2^{K-1}$ is ***code-disjoint.***

Proof: When a code in subset E_{2i-1} is applied to Circuit CW, the code input produces a code output in Block S. Consider a noncode which consists of J_q^* and P_r^*, where $r \neq t$. If $P_r^* \in$ SPC, similar to the proof in Lemma 3.5, the application of such a code produces a noncode output in Block S. On the other hand, for $P_r^* \in$ SPNC, since $g^r = 1$, where $g^r \in$ SGNC, is simultaneously connected to both S^1 and S^2, a noncode output (1,1) in S results. Thus, the circuit is ***code-disjoint.*** ♦

Similar to the proof in Lemma 3.4, all unidirectional faults in Blocks H, Q, S, and all $g^i \in$ SGC in circuit CW are detectable. Since $g^i \in$ SGNC is connected to both S^1 and S^2, a s/1 fault at g^i forces Block S to produce a noncode output (1,1) with the application of any code input. For s/0 faults at all $g^i \in$ SGNC, they are redundant, but will not affect the circuit performance. More specifically, in a fault-free circuit, $g^i = 0$, for every $g^i \in$ SGNC. On the other hand, in the presence of s/0 faults, or 1-to-0 faults, the reduced check part $P_i^* \in$ SPC will never be in SPNC. In other words, applying either a code or noncode input to the circuit will never produce a non-codeword output. Therefore, if we consider a fault set F which consists of all multiple unidirectional faults except the redundant fault, the following lemma results.

Lemma 3.12: Circuit CW for $u < 2^{K-1}$ is ***self-testing*** for the fault set F. ♦

Lemma 3.13: Circuit CW for $u < 2^{K-1}$ is a STC checker.

Proof: By Lemma 3.11 and Lemma 3.12, the circuit is a STC checker. ♦

Theorem 3.4: Circuit CW for any ***Berger code*** length is a STC checker.

Proof: By Lemma 3.10 and Lemma 3.13, Circuit CW is a STC checker. ♦

3.2.2 STC DESIGN WITH PARTITIONING AND FOLDING SCHEME

Recall that the necessary condition for implementing the partitioning and folding scheme is that ***every pair of subsets E_m and E_{2u-m} must be dual.*** Consider a ***non-maximal-length Berger*** (NMLB) code B(11,4), as listed in Table 3.6. Table 3.6(b) shows that the expanded information parts J_1^* and J_{11}^* are dual, but the corresponding reduced check parts P_1^* and P_{11}^* are not dual. Thus, the pair of subsets E_1 and E_{11} are not dual. This motivates the development of the following new encoding scheme that makes all pairs of subsets to be dual. To distinguish the previous ***left-justified*** and ***right-justified*** mapping schemes, the scheme presented here is referred to as ***center-justified mapping scheme.***

For simplicity, we first consider the ***Berger codes*** B(I,K) with any odd number I and the even number u. The ***center-justified*** encoding scheme encodes the check part as:

$$P_t = (2^{K-1} + 2\upsilon - 1) - N_1(J_t), \text{ where } 1 \le t \le I+1,$$

i.e.,

$$P_t = [2^{K-1} + (I + 1)/2 - 1] - N_1(J_t) = 2^{K-1} + (I + 1)/2 - t,$$

$$P_1 = 2^{K-1} + 2\upsilon - 1 = 2^{K-1} + (I + 1)/2 - 1, \text{ and}$$

$$P_{I+1} = 2^{K-1} - 2\upsilon = 2^{K-1} - (I + 1)/2.$$

Table 3.7 shows the ***Berger code*** B(11,4) encoded by three encoding schemes. Since all K-bit codes are used to encode the check parts of the MLB codes, the encoding schemes produce the same MLB code.

For the case where both I and u are even, as discussed in Section 3.1.2 (B), an extra bit, $x_0 = 0$, is added as the MSB of the information part to make I as an odd number. Thus, the above encoding scheme applies.

Table 3.6: Berger Code B(11,4).

(a)

Subset	Information Part (J) $x_1\,x_2\,x_3\,x_4\,x_5\,x_6\,x_7\,x_8\,x_9\,x_{10}\,x_{11}$	Check Part(P) $x_{12}x_{13}x_{14}x_{15}$
D_1	0 0 0 0 0 0 0 0 0 0 0	1 1 1 1
D_2	0 0 0 0 0 0 0 0 0 0 1	1 1 1 0
D_3	0 0 0 0 0 0 0 0 0 1 1	1 1 0 1
D_4	0 0 0 0 0 0 0 0 1 1 1	1 1 0 0
D_5	0 0 0 0 0 0 0 1 1 1 1	1 0 1 1
D_6	0 0 0 0 0 0 1 1 1 1 1	1 0 1 0
D_7	0 0 0 0 0 1 1 1 1 1 1	0 1 1 1
D_8	0 0 0 0 1 1 1 1 1 1 1	0 1 1 0
D_9	0 0 0 1 1 1 1 1 1 1 1	0 1 0 1
D_{10}	0 0 1 1 1 1 1 1 1 1 1	0 1 0 0
D_{11}	0 1 1 1 1 1 1 1 1 1 1	0 1 1 1
D_{12}	1 1 1 1 1 1 1 1 1 1 1	0 1 1 0

(b)

Subset	Expended Information Part (J^*) $x_1\,x_2\,x_3\,x_4\,x_5\,x_6\,x_7\,x_8\,x_9\,x_{10}\,x_{11}x_{15}$	Reduced Check Part(P^*) $x_{12}x_{13}x_{14}$
E_1	0 0 0 0 0 0 0 0 0 0 0 1	1 1 1
E_3	0 0 0 0 0 0 0 0 0 1 1 1	1 1 0
E_5	0 0 0 0 0 0 0 1 1 1 1 1	1 0 1
E_7	0 0 0 0 0 1 1 1 1 1 1 1	1 0 0
E_9	0 0 0 1 1 1 1 1 1 1 1 1	0 1 1
E_{11}	0 1 1 1 1 1 1 1 1 1 1 1	0 1 0

(c)

Subset	Expended Information Part (J^*) $x_1\,x_2\,x_3\,x_4\,x_5\,x_6\,x_7\,x_8\,x_9\,x_{10}\,x_{11}x_{15}$	Reduced Check Part(P^*) $x_{12}x_{13}x_{14}$
E_1	0 0 0 0 0 0 0 0 0 0 0 1	1 1 0
E_3	0 0 0 0 0 0 0 0 0 1 1 1	1 0 1
E_5	0 0 0 0 0 0 0 1 1 1 1 1	1 0 0
E_7	0 0 0 0 0 1 1 1 1 1 1 1	0 1 1
E_9	0 0 0 1 1 1 1 1 1 1 1 1	0 1 0
E_{11}	0 1 1 1 1 1 1 1 1 1 1 1	0 0 1

(d)

Subset	Expended Information Part (J^*) $x_1\,x_2\,x_3\,x_4\,x_5\,x_6\,x_7\,x_8\,x_9\,x_{10}\,x_{11}x_{15}$	Reduced Check Part(P^*) $x_{12}x_{13}x_{14}$	Partitioning
Z_1	0 0 0 0 0 0 0 0 0 0 0 1	1 1 1	$C_{1/12}\times(1\,1\,1)$
Z_3	0 0 0 0 0 0 0 0 0 1 1 1	1 1 0	$C_{3/12}\times(1\,1\,0)$
Z_5	0 0 0 0 0 0 0 1 1 1 1 1	1 0 1	$C_{5/12}\times(1\,0\,1)$

Table 3.7: Berger Code Encoding Scheme for B(11,4).

B(11,4)	Left-justfied	Right-justfied	Center-justfied
P_1	1111	1011	1101
P_2	1110	1010	1100
P_3	1101	1001	1011
P_4	1100	1000	1010
P_5	1011	0111	1001
P_6	1010	0110	1000
P_7	1001	0101	0111
P_8	1000	0100	0110
P_9	0111	0011	0101
P_{10}	0110	0010	0100
P_{11}	0101	0001	0011
P_{12}	0100	0000	0010

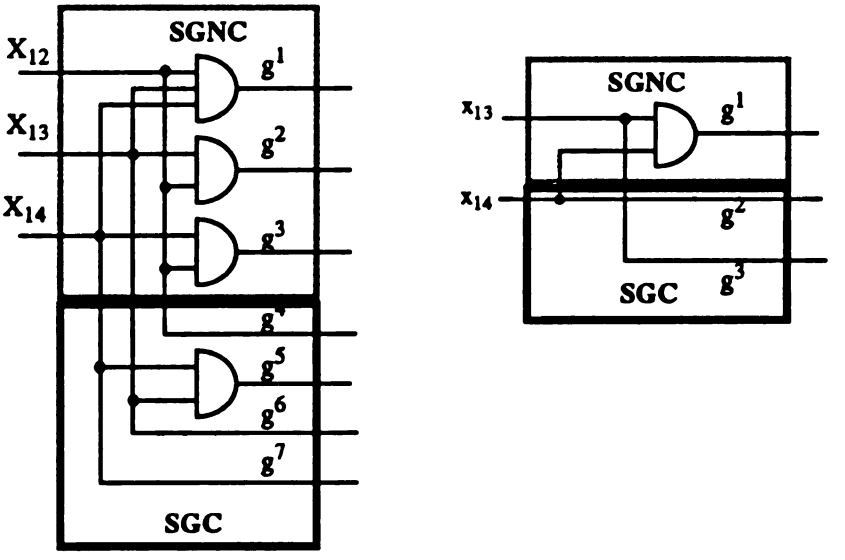
Based on this encoding scheme and partitioning and folding scheme, a ***Berger code*** B(11,4), as shown in Table 3.6, can be constructed by

$$B(11,4) = C_{1/12}\times(10) \cup C_{3/12}\times(01) \cup C_{5/12}\times(00), \tag{3.8}$$

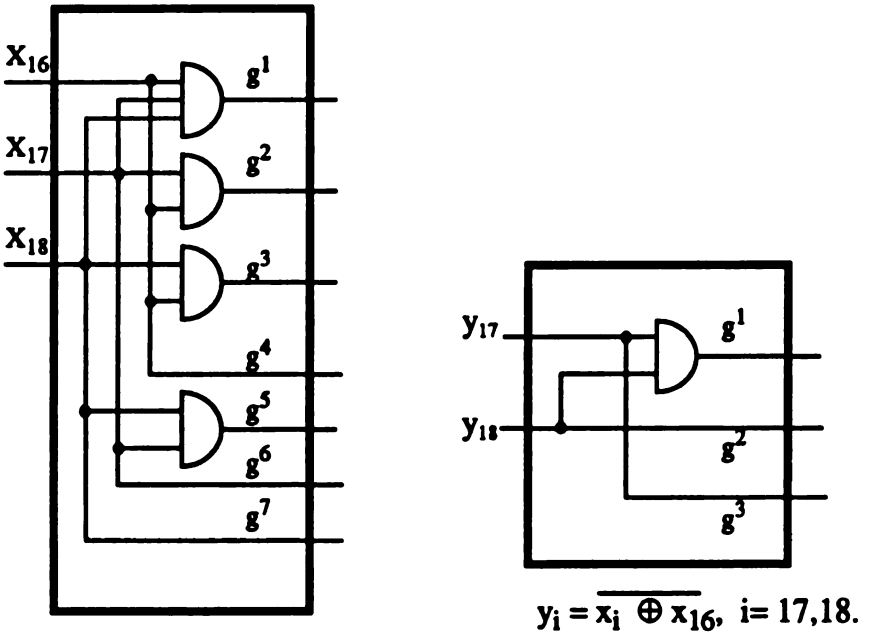
where $u = 6$ and $\upsilon = 3$. In a general case, a ***Berger code*** B(I,K) with any even number u can be constructed by υ's $(2m-1)/(I+1)$ codes, where $1 \le m \le \upsilon$.

Let $v = 2^{K-2} - \upsilon$. For the reduced check parts, the set of non-codewords SPNC = $\{P_i^* \mid i \in I_v\}$ and the set of codewords SPC = $\{P_i^* \mid v+1 \le i \le \upsilon\}$. Since g^i is the output of the AND gate G^i which is constructed from P_i^*, for simplicity, let SGNC = $\{g^i \mid i \in I_v\}$ and SGC = $\{g^i \mid v+1 \le i \le \upsilon-1\}$. Figure 3.9 illustrates the circuit for Block G for B(15,4) and B(11,4), where SGNC = $\{g^1\}$ and SGC = $\{g^2, g^3\}$ in B(11,4).

Figure 3.10 shows the internal structure, referred to as Circuit CL, of a STC design for B(I,K) with any even number u. Circuit CL consists of a code-complementer and a checker. The checker is comprised of the following four blocks:



(a)



(b)

Figure 3.9 Function G for (a) B(11,4); and (b) B(15,4) STC design.

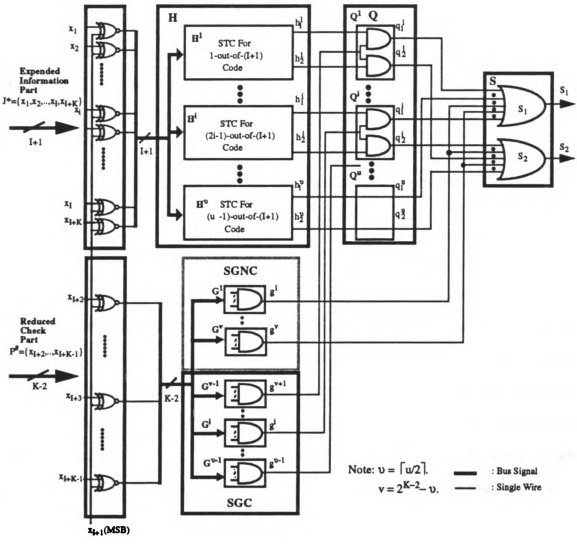


Figure 3.10 Circuit CL for B(I,K) STC design with even u.

Block H: Checkers for $C_{(2i-1)/(I+1)}$, $1 \leq i \leq \upsilon$;

Block G: 2^{K-2} - (K - 1) AND gates;

Block Q: (υ - 1) pairs of AND gates; and

Block S: Two 2^{K-2}-input OR gates.

Circuit CL is identical to Circuit CS for the ***Berger codes*** with $u = 2^{K-1}$. The same Blocks G and S are used for all STC designs for B(I,K), where $2^{K-1} \leq I \leq 2^K-1$. However, the structure in Blocks H and Q depends on the length of I. Since SGNC is not a null set for any even number $u < 2^{K-1}$, every $g^i \in$ SGC is connected to Block Q, while every $g^j \in$ SGNC is simultaneously connected to the OR gates S^1 and S^2. Figure 3.11 illustrates the STC design for B(11,4).

Lemma 3.14: Circuit CL is ***code-disjoint.***

Proof: When a code in subset E_{2i-1} is applied to Circuit CL, the code input produces a code output in Block S. Consider the application of a noncode which consists of J_q^* and P_r^*, where $r \neq t$, to Circuit CL. If $P_r^* \in$ SPC, by Lemma 3.3, the checker H^i produces a noncode (0,0), for $i > q$, or (1,1), for $i < q$. Since $g^i = 0$, for $i > r$, and $g^r = 1$, a noncode output (1,1) is produced in Q^r, if $r > q$, and further generates a noncode output in Block S. On the other hand, for $P_r^* \in$ SPNC, since $g^r = 1$, where g^r is connected to both S^1 and S^2, and $g^r = 1$ and $g^r \in$ SGNC a noncode output (1,1) in S results. Thus, the circuit is ***code-disjoint.*** ♦

Similar to the proofs of Lemma 3.11 and Lemma 3.12, Circuit CL can be shown as ***code-disjoint*** and ***self-testing*** for the fault set F. Thus, it is a STC checker.

The previous discussion assumes that u is even. For the odd number u, it is necessary to convert it to a even number. More specifically, as shown in Table 3.7 for B(11,4), where u = 6, the check parts $P_5 = (1001)$, $P_6 = (1000)$, $P_7 = (0111)$, and $P_8 = (0110)$. In general, for an even number u, the check parts $P_{u-1} = 2^{K-1} + 1$, $P_u = 2^{K-1}$,

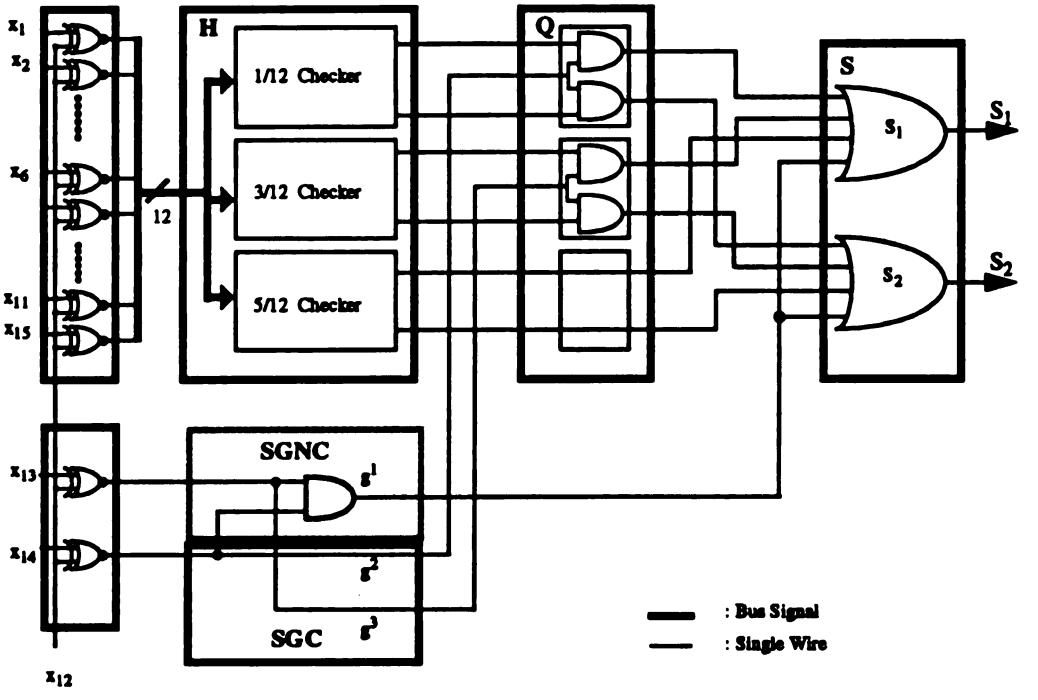


Figure 3.11 Circuit CL for B(11,4) STC design.

$P_{u+1} = 2^{K-1} - 1$, and $P_{u+2} = 2^{K-1} - 2$. The pairs (P_{u-1},P_u) and (P_{u+1},P_{u+2}) are referred to as ***center pairs***. If u is an odd number, then $2\upsilon = u + 1$. For example, for B(9,4), u = 5 and $\upsilon = 3$, there exist 5 reduced check parts P_1^*, P_3^*, P_5^*, P_7^*, and P_9^*. Among them, P_1^* can be paired with P_9^*, and P_3^* can be paired with P_7^*, but P_5^* cannot be paired with anyone. In general, for any odd number u, P_u^* will not be paired with any other reduced check parts available in a B(I,K). Thus, the codes for the center pair (P_{u+1},P_{u+2}) are not assigned to any check part, or they are defined as noncode. Mathematically, the check parts are encoded as follows: $P_t = (2^{K-1} + 2\upsilon - 1) - N_1(J_t) + W$, where W = 2, if $t \geq u + 2$; W = 0, otherwise. Thus, the presented encoding scheme can be re-stated as follows:

Modified Berger Code Encoding Scheme: For a ***Berger code*** B(I,K), the check part P_t is the binary encoding of $(2^{K-1} + 2\upsilon - 1) - N_1(J_t) + W$, where

$$W = \begin{cases} 0 & \text{if u is even, or u is odd and } t < u + 2; \\ 2 & \text{if u is odd and } t \geq u + 2. \end{cases}$$

Similar to the previous discussion, a ***Berger code*** B(9,4) can be expressed as

$$B(9,4) = C_{1/10} \times (10) \cup C_{3/10} \times (01) \cup C_{5/10} \times (00). \tag{3.9}$$

Similar to the previous discussion, if $v = 2^{K-1} + 2\upsilon - 1$, for the reduced check parts, the set of non-codewords SPNC = $\{P_i^* \mid i \in I_v\}$ and the set of codewords SPC = $\{P_i^* \mid v+1 \leq i \leq u\}$. Also, SGNC = $\{g^i \mid i \in I_v\}$ and SGC = $\{g^i \mid v+1 \leq i \leq u-1\}$. For example, SGNC = $\{g^1\}$ and SGC = $\{g^2,g^3\}$ in B(9,4). This shows that the STC structure for B(I,K) with an even number u is almost identical to that with an odd number u, except an AND gate is added, as shown in Figure 3.12, for detecting the noncodes, $2^{K-1} - 1$ and $2^{K-1} - 2$ for the center pair (P_{u+1},P_{u+2}). More specifically, consider the ***Berger code*** B(9,4), for example, the codewords for the reduced check parts are {(110),(101),(100), (010),(001)} and the non-codewords are {(111),(011),(000)}. For a noncode which consists of J_5^* and a

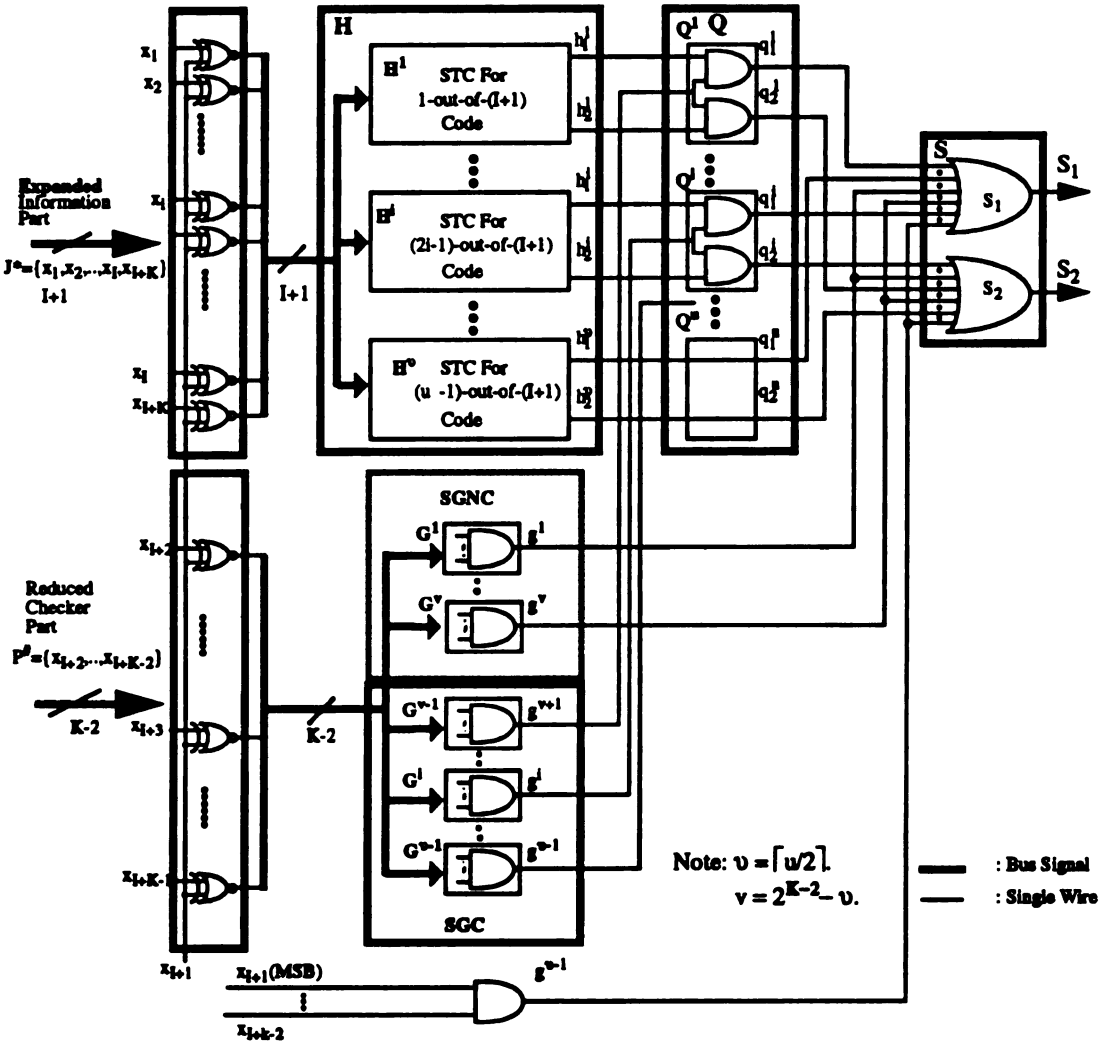


Figure 3.12 Circuit CL for B(I,K) STC design with odd u.

reduced check part (011), since $x_{MSB(t)} = 0$ and $P_t^{\#} = (11)$, the code-complementer will produce an alternative code J_5^* to Block H and a code (00) to Block G. Thus, a code output results in Block S. This implies that, without the extra AND gate, the circuit in Figure 3.13 is not ***code-disjoint***. Since the code $(x_I,x_{I+1},...,x_{I+K-1}) = (11...1)$ is always a non-codeword in a B(I,K) with an odd number u, and the code (011...1) is assigned as a non-codeword, hence, combining these two non-codewords as $(x_{I+1},x_{I+2},...,x_{I+K-1}) = (11...1)$, where x_I is a "don't care" term. Thus, an AND gate which takes $(x_{I+1},x_{I+2},...,x_{I+K-1})$ as its input can detect the non-codewords and makes the circuit in Figure 3.12 be ***code-disjoint*** and further be a STC checker.

In summary, the STC structure in Figure 3.10 is implemented for the ***Berger codes*** with an even number u, while the STC structure in Figure 3.12 is employed for the ***Berger codes*** with an odd number u. The only difference is in the extra AND gate. Thus, the presented STC structure can be implemented with any ***Berger code*** length.

3.3 Design Alternative

Previous sections have presented the STC design for both MLB and NMLB with partitioning and folding scheme. Results show that the designs need only half of the checkers required in [30]. The significant reduction is achieved by XNORing each input bit with the MSB of the reduced check part P^*, as indicated in Figure 3.2. As shown in Table 3.2, there are two ways to fold the subsets E_{2i-1}'s, i.e. we may fold the subsets either upward, i.e., choose the pair (Z_1,Z_3), or downward, i.e., choose the pair (Z_5,Z_7). Circuit CS, as shown in Figure 3.2, was generated based on the upward folding. Thus, it is also possible to generate an alternative design using the downward folding.

Similar to Theorem 3.2, the following Corollary results.

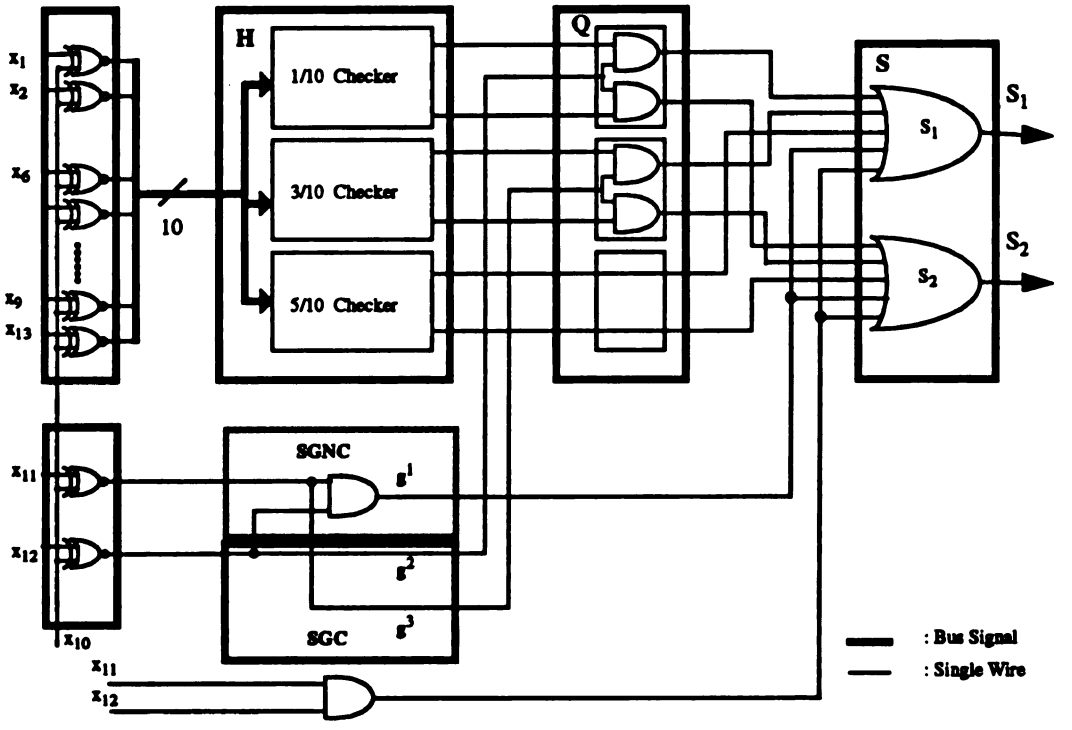


Figure 3.13 Circuit CL for B(9,4) STC design.

Corollary: If m < u, then any unidirectional errors in a subset $Z_m = E_m \oplus x_{MSB(m)}$ can be detected by the checker $C_{(2u-m)/(I+1)}$.

Therefore, in Table 3.2(d) if the pair (Z_5,Z_7) is chosen, then the ***Berger code*** B(7,3) can be expressed as

$$B(7,3) = C_{5/8} \times (01) \cup C_{7/8} \times (00) \tag{3.10}$$

In general, if the downward folding is chosen, a ***Berger code*** B(I,K) can be expressed as

$$B(I,K) = \bigcup_{i=u/2}^{u} C_{(2i-1)/(I+1)} \times P^{\#}_{2i-1} \tag{3.11}$$

Figure 3.14 shows the internal structure of Circuit CS with the downward folding scheme, where the code complementer is comprised of XOR gates.

The above results conclude that, based on the MSB of P*, the subsets E_{2i-1}'s can be folded by choosing either the upper half or lower half of these subsets. Both implementations have similar structures as shown in Figures 3.2 and 3.14. The former uses XNOR gates while the latter employs XOR gates.

In practice, however, any bit of p* can also be used to fold the E subsets. More specifically, consider $P^* = (x_{I+1},x_{I+2},...,x_{I+k-1})$. Let $y_i = x_{I+i+1}$, for simplicity, i.e., $P^* = (y_0,y_1,...,y_{k-2})$, where y_0 and y_{k-2} are the MSB and LSB of P^*, respectively.

Definition 3.2: Let S_E be a collection of subsets E_t's in B(I,K). $S_{e1}(p)$ and $S_{e0}(p)$ are the two y_p-***partitioned subsets*** of S_E if $S_{e1}(p)$ ($S_{e0}(p)$) is a collection of $E_t \in S_E$ whose y_p-bit of P^*_t is 1 (0).

Definition 3.3: Let S_c be the checker set for B(I,K), where $S_c = \{C_{(2t-1)/(I+1)} \mid 1 \le t \le u\}$. $S_{c1}(p)$ and $S_{c0}(p)$ are the ***y_p-partitioned checker subsets*** of S_c if $S_{c1}(p) = \{C_{t/(I+1)} \mid E_t \in S_{e1}(p)\}$, $S_{c0}(p) = \{C_{t/(I+1)} \mid E_t \in S_{e0}(p)\}$, and $S_{c1}(p) \cup S_{c0}(p) = S_c$.

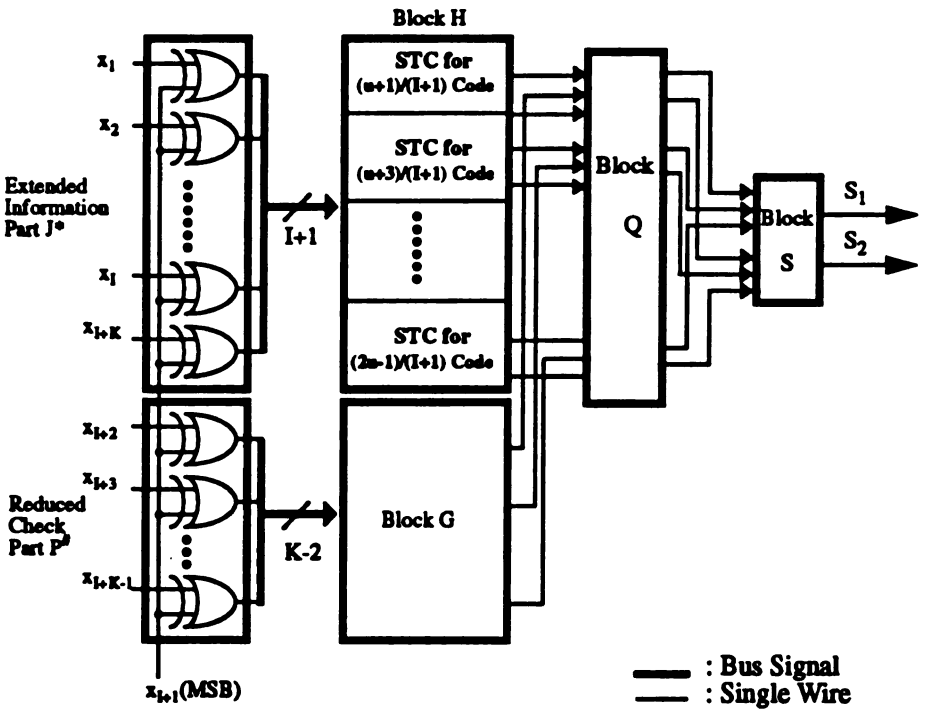


Figure 3.14 Circuit CS with XOR code complementer.

Definition 3.4: Let $I_{c0}(p)$ ($I_{c1}(p)$) be the ***index set*** of the y_p-partitioned subset $S_{c0}(p)$ ($S_{c1}(p)$). Then,

$$I_{c1}(p) = \{2^{K-p-1}j + (2t - 1) \mid 0 \le j \le 2^p-1 \text{ and } 1 \le t \le 2^{K-p-2} \}; \text{ and}$$
$$I_{c0}(p) = I_E - I_{c1}(p). \tag{3.12}$$

Table 3.8(a) lists the subsets E_{2t-1} and the reduced check part of B(31,5) and the corresponding index sets are shown in Table 3.8(b). It shows that there exist eight possible STC designs of ***Berger code*** B(31,5). The choice is determined by which checker subset requires the least gate count.

Definition 3.5: Let M_{ip} be a ***measure*** of the implementation with the y_p-partitioned subset $S_{ci}(p)$, where i = 0 or 1, and $0 \le p \le K-2$. $M_{ip} << M_{jq}$ means the implementation with the y_p-partitioned subset $S_{ci}(p)$ is better than that with $S_{cj}(q)$.

The ***measure*** in Definition 3.5 can be either the speed performance and/or the total gate count for a y_p partitioned checker subset. However, since it has been shown that the partitioning scheme offers an improvement in speed performance, the *measurement* here, thus, concentrates on the total gate count.

Property 1: Let $M_{ct}(q) = \min\{M_{ci}(p) \mid i = 0 \text{ or } 1, \text{ and } 0 \le p \le K-2\}$ be the best measure among $M_{ci}(p)$'s. A ***Berger code*** B(I,K) can be implemented by

$$B(I,K) = \bigcup_{i=1}^{v} C_{(2i-1)/(I+1)} \times P^{\#}_{2i-1} \tag{3.13}$$

which requires least gate count, where $P_t^{\#} = P_t^{*} - \{y_q\}$.

Table 3.8: Reduced Check Part for B(31,5).

(a)

Subsets	$y_0y_1y_2y_3$
E_1	1 1 1 1
E_3	1 1 1 0
E_5	1 1 0 1
E_7	1 1 0 0
E_9	1 0 1 1
E_{11}	1 0 1 0
E_{13}	1 0 0 1
E_{15}	1 0 0 0
E_{17}	0 1 1 1
E_{19}	0 1 1 0
E_{21}	0 1 0 1
E_{23}	0 1 0 0
E_{25}	0 0 1 1
E_{27}	0 0 1 0
E_{29}	0 0 0 1
E_{31}	0 0 0 0

(b)

Reduced Check Bit (p)	$I_{c1}(p)$	$I_{c0}(p)$
0	{1,3,5,7,9,11,13,15}	{17,19,21,23,25,27,29,31}
1	{1,3,5,7,17,19,21,23}	{9,11,13,15,25,27,29,31}
2	{1,3,9,11,17,19,25,27}	{5,7,13,15,21,23,29,31}
3	{1,5,9,13,17,21,25,29}	{3,7,11,15,19,23,27,31}

Theoretically speaking, any ***Berger code*** STC checker can be designed either with the partitioning scheme, or the partitioning and folding scheme. Note that the STC design with either scheme requires a set of ***m/n*** code checkers. In practice, not all ***m/n*** code checkers can be efficiently designed with the TSC property. This implies that not all ***Berger code*** checkers can be designed efficiently with the partitioning scheme. However, among the various alternative designs presented in this section, it is possible to find a folding which employs those existing ***m/n*** code checkers to achieve the best solution.

3.4 SUMMARY

Based on the partitioning scheme presented in [30], both information part and check part of a ***Berger code*** are expanded and reduced, respectively. The scheme offers a significant improvement in speed performance, but requires a great hardware cost. This chapter presents a partitioning and folding scheme to improve the hardware cost and speed performance as well. Results have shown that the partitioning and folding scheme reduces the hardware cost nearly by half compared to the scheme in [30].

In order to demonstrate the effectiveness of the presented scheme, Table 3.9 summarizes various existing design schemes. Results show that the partitioning and folding scheme indeed provides an efficient, yet low hardware cost, STC design for ***Berger codes***. However, the table also shows that the code length of the checker grows linearly with the number of outputs of the functional circuit, but the complexity of the checker grows rapidly and the hardware cost increases nearly exponentially as the code length increases. Interestingly, the table shows that the use of many smaller code checkers may take much less hardware cost than that of a larger checker for the same total information bits. Therefore, if the output bits of a given functional circuit can be partitioned into smaller groups which employ smaller checkers, then the hardware cost can be further reduced. This has motivated the development of an output partitioning algorithm that achieves this objective in the next chapter.

Table 3.9: Comparisons for Various Checker Designs.

B(I,K)	Gate Count						Gate Level					
	28	29	30_{CE}	30_{ML}	31	*	28	29	30_{CE}	30_{ML}	31	*
(6,3)	31	52	61	72	67	54	15	13	8	5	10	9
(7,3)	40	58	87	137	76	56	17	11	8	5	10	9
(8,4)	58	72	93	143	101	61	21	19	10	7	12	11
(14,4)	136	142	261	$>2^{15}$	208	173	33	19	11	5	12	12
(15,4)	149	150	315	$>2^{16}$	227	174	35	17	11	5	12	12
(16,4)	181	166	321	$>2^{16}$	260	180	39	27	13	7	15	14
(31,5)	436	346	1279	$>2^{31}$	720	673	69	23	13	5	15	14
(32,6)	488	356	1285	$>2^{31}$	824	679	73	33	15	7	17	16

CE : cost-effective design
ML : minimal-level design
* : with the developed partitioning and folding scheme

CHAPTER 4

OUTPUT PARTITIONING ALGORITHM

This chapter describes an efficient output partitioning algorithm. The algorithm partitions the output of a given functional circuit into many smaller groups and each group employs a smaller checker. We first describe the problem to be solved. Then, the problem is formulated in Section 4.2, and an algorithm is given in Section 4.3. Section 4.4 discusses some practical design considerations. Experimental results on MCNC benchmark finite-state machines (FSMs) are shown in Section 4.5.

4.1 PROBLEM STATEMENT

Consider the following matrix, referred to as an *essential output matrix*, which represents all possible outputs of a given circuit,

$$
\begin{array}{l}
1\,2\,3\,4\,5\,6\,7\,8\,9 \leftarrow \text{column index} \\
0\,0\,0\,0\,0\,0\,0\,0\,0 \\
1\,0\,0\,1\,1\,0\,0\,0\,0 \\
0\,0\,0\,0\,0\,1\,1\,0\,0 \\
0\,0\,1\,0\,0\,0\,0\,1\,0 \\
1\,1\,0\,0\,0\,0\,1\,0\,1
\end{array}
\qquad (4.1)
$$

A 4/13 code checker is generally chosen for a ***self-checking*** circuit design. The checker circuit can be realized either by 721 gates with three-level implementation [17], or by 105 gates and 7 levels with the cost-effective implementation [20].

The output matrix may be partitioned into either the following two submatrices

$$\begin{array}{c|c} 1\,2\,3\,4\,8 & 5\,6\,7\,9 \\ 0\,0\,0\,0\,0 & 0\,0\,0\,0 \\ 1\,0\,0\,1\,0 & 1\,0\,0\,0 \\ 0\,0\,0\,0\,0 & 0\,1\,1\,0 \\ 0\,0\,1\,0\,1 & 0\,0\,0\,0 \\ 1\,1\,0\,0\,0 & 0\,0\,1\,1 \end{array} \tag{4.2}$$

or the following three submatrices

$$\begin{array}{c|c|c} 2\,3\,4\,6 & 5\,8\,9 & 1\,7 \\ 0\,0\,0\,0 & 0\,0\,0 & 0\,0 \\ 0\,0\,1\,0 & 1\,0\,0 & 1\,0 \\ 0\,0\,0\,1 & 0\,0\,0 & 0\,1 \\ 0\,1\,0\,0 & 0\,1\,0 & 0\,0 \\ 1\,0\,0\,0 & 0\,0\,1 & 1\,1 \end{array} \tag{4.3}$$

The former requires two checkers for $C_{2/7}$ and $C_{2/6}$, which can be respectively realized by 26 and 20 gates with 3-level implementation [17], or 46 gates and 3 levels in total excluding a checker for the final result. (In general, a two-rail checker with 6 gates and 2 levels can be used as the final checker.) On the other hand, the circuit for Matrix (4.3) needs three checkers for $C_{1/5}$, $C_{1/4}$, and $C_{2/4}$, which can be respectively realized by 10, 8, and 6 gates with 3-level implementation.

The above example demonstrates that the use of a larger code checker may require a larger gate count and gate level than that of many smaller ones in total. Table 3.9 also endorses the above finding. Now, the question that naturally arises is ***how to partition the outputs of a given circuit so that smaller code checkers can be employed to reduce hardware overhead and increase speed performance.***

4.2 PROBLEM FORMULATION

Consider an output matrix $C = [c_{ij}]_{nxm}$, where the entry c_{ij} is either 0 or 1, and its *column index set* $\Gamma = \{1,2,..., m\}$. Let R_i denote the set of column indices in the i-th row with $c_{ij} = 1$, i.e., $R_i = \{j \mid c_{ij} = 1\}$, and $|R_i|$ be its cardinality. A matrix is called a ***k-set matrix*** if it has at most k 1's in its rows, i.e., $|R_{max}| = \max\{|R_i| \; 1 \le i \le n\} = k$. Suppose that the k-set matrix C is partitioned into t submatrices whose column index sets are denoted as G_i, $1 \le i \le t$, where $\bigcup_{i=1}^{t} G_i = \Gamma$, $\bigcap_{i=1}^{t} G_i = \varnothing$, and $|G_i| \le k$. For simplicity, we will first assume that all submatrices are r-sets. (The constraint will be relaxed later.) Therefore, our problem is to develop an efficient algorithm that partitions a given k-set output matrix into a minimal number of r-set submatrices, where $r \le k$. This problem can be formulated as follows.

Problem P: To minimize t

Subject to $\bigcup_{i=1}^{t} G_i = \Gamma$, $\bigcap_{i=1}^{t} G_i = \varnothing$, and $|G_i \cap R_j| \le r$, for all i and j.

The condition, $|G_i \cap R_j| \le r$ for all i and j, implies that a set G_i cannot contain more than r elements in a set R_j. In other words, the set G_i should be constructed such that any (r + 1) elements of a set R_j cannot be in the same G_i. Thus, the theorem defines the lower bound on the number of the partitioned submatrices.

Theorem 4.1: $\lfloor |R_{max}|/r \rfloor \le t$.

Proof: Since $|R_{max}| = \max\{|R_i| \; 1 \le i \le n\}$ and at most r elements in R_{max} can be in the same G_i, this implies that t will not be less than $\lfloor |R_{max}|/r \rfloor$, i.e., $\lfloor |R_{max}|/r \rfloor \le t$, where the equality possibly holds only when $|R_{max}| = rt$. ♦

Note that, for $|R_{max}| = m$ or $|R_{max}| \leq r$, the given output matrix can be arbitrarily partitioned into any $\lceil m/r \rceil$ r-set submatrices. Therefore, no partition is needed in our implementation.

As mentioned, we are dealing with the output matrix C which consists of all possible outputs of a given circuit. The question is whether or not we should deal with all possible outputs in that matrix. $R_s \subseteq R_j$ denotes that the j-th row ***covers*** the s-th row.

Theorem 4.2: Let G_i's be some sets that satisfy the conditions $\bigcup_{i=1}^{t} G_i = \Gamma$, $\bigcap_{i=1}^{t} G_i = \emptyset$, and $|G_i \cap R_j| \leq r$, for $1 \leq i \leq t$. If $R_s \subseteq R_j$, then $|G_i \cap R_s| \leq r$ for $1 \leq i \leq t$.

Proof: Since $R_s \subseteq R_j$, we have $G_i \cap R_s \subseteq G_i \cap R_j$. Thus, $|G_i \cap R_s| \leq |G_i \cap R_j| \leq r$. ♦

A row is *essential* if it is not ***covered*** by any other row. Thus, the ***covered*** outputs and the duplicated outputs are not necessarily included. In other words, we are only dealing with the matrix, referred to as *reduced matrix*, which consists of all essential rows. Consider the MCNC benchmark FSM, *ex4.kiss2*, as shown in Figure 4.1(a), and its output matrix, Figure 4.1(b). There exist four essential rows which form the reduced matrix in Figure 4.1(c). The reduction in dimension is significant. Based on Theorem 2, Problem P can be re-formulated as follows.

Problem: To minimize t •

subject to $\bigcup_{i=1}^{t} G_i = \Gamma$, $\bigcap_{i=1}^{t} G_i = \emptyset$, and $|G_i \cap R_j| \leq r$, for all i and essential rows j.

```
.i 6
.o 9
.p 21
.s 14
1-----   1   3 110000000
1-----   3   2 000000000
1-----   2   5 001000000
1-----   5   7 000000000
10----   7   7 000000000
11----   7  11 100110000
1----- 11  12 100100000
1-1--- 12   8 000001100
1-0--- 12   8 000000100
1-0---   8   3 110000000
1-10--   8   3 110000000
1-11--   8   4 110000000
1---1-   4  13 000000010
1---0-   4  13 000000000
1----- 13  14 001000010
1----- 14   6 000000000
10----   6   6 000000000
11----   6   9 100110000
1-----   9  10 100100000
1----1 10   3 110000101
1----0 10   4 110000100
```

(a) the original *kiss2* file.

	Column Index								
Row No.	1	2	3	4	5	6	7	8	9
1	1	1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	1	0	0	1	1	0	0	0	0
7	1	0	0	1	0	0	0	0	0
8	0	0	0	0	0	1	1	0	0
9	0	0	0	0	0	0	1	0	0
10	1	1	0	0	0	0	0	0	0
11	1	1	0	0	0	0	0	0	0
12	1	1	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	1	0
14	0	0	0	0	0	0	0	0	0
15	0	0	1	0	0	0	0	1	0
16	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0
18	1	0	0	1	1	0	0	0	0
19	1	0	0	1	0	0	0	0	0
20	1	1	0	0	0	0	1	0	1
21	1	1	0	0	0	0	1	0	0

(b) output matrix.

	Column Index								
Row No.	1	2	3	4	5	6	7	8	9
6	1	0	0	1	1	0	0	0	0
8	0	0	0	0	0	1	1	0	0
15	0	0	1	0	0	0	0	1	0
20	1	1	0	0	0	0	1	0	1

(c) reduced output.

Figure 4.1 A FSM, *ex4.kiss2*.

4.3 ALGORITHMS

For simplicity, we first present the algorithm for r = 1, and then for r = 2 and other r.

A. Case of r = 1

For r = 1, by Property 1, a G_i contains at most one element of a R_j set. The question that arises is which column should be selected first when a G_i set is constructed. In this development, a simple rule is presented to establish the selection priority.

We first record the *column count*, i.e., the number of 1's in each column, of the reduced matrix. A column with a higher count means that it most likely belongs to more R_j sets and it will be more difficult to satisfy the condition $|G_i \cap R_j| \leq 1$. Thus, we apply a heuristic that the column with a higher count should have a lower selection priority, and if two columns have the same count, the lower column index is defined to have the higher priority. Let $\Gamma_p = <\gamma_1, \gamma_2, ..., \gamma_m>$, a permutation of Γ, denote the descending priority set, i.e., the priority of γ_i is higher than that of γ_j if $i < j$. The priority set Γ_p is used to construct G_i's.

Once the priority set Γ_p is generated, a selection process follows. Let S_c be the ordered set that consists of the column indices to be selected, where $S_c = \Gamma_p = <\gamma_1, \gamma_2, ..., \gamma_m>$ initially. We start constructing the set G_1 by selecting the leading element, γ_s, of the set S_c, where $\gamma_s = \gamma_1$ and $G_1 = \{\gamma_1\}$. Since each R_j contains at most one element in a G_1, those R_j sets which contain γ_1 must be excluded from Sc, i.e., $S_c = S_c - \cup \{R_j \mid \gamma_1 \in R_j\}$. The same process is carried out repeatedly by including the leading element of the set S_c into the G_1 set until $S_c = \varnothing$. Once G_1 is generated, the construction of G_2 proceeds. In order to keep the disjointedness, G_1 must be excluded from S_c, i.e., $S_c = S_c - G_1$. The above procedure is then repeatedly applied for generating the G_2 set and other G_i sets. The process is finally terminated when all m columns are selected, i.e., $\bigcup_{i=1}^{t} G_i = \Gamma$. Of course, all G_i's are disjoint, i.e., $\bigcap_{i=1}^{t} G_i = \varnothing$.

The above procedure is summarized in Algorithm I.

Algorithm I. (r = 1)

Step 1. Generate the reduced output matrix and derive the R_j sets.

Step 2. If $|R_{max}| = \max\{|R_i| \; 1 \le i \le n\} = m$ or $|R_{max}| \le r$, then STOP;

/* No partition is needed. */

Step 3. Establish the selection priority and generate the set $\Gamma_p = <\gamma_1,\gamma_2,...,\gamma_m>$.

Step 4. Selection Processes

$u = 0$;

Repeat

4.1. $u = u + 1$; $G_u = \varnothing$; $S_c = \Gamma_p$;

Repeat γ_s = the leading element of S_c;

$G_u = G_u \cup \{\gamma_s\}$;

$S_c = S_c - \cup \{R_j \mid \gamma_s \in R_j\}$;

Until $S_c = \varnothing$;

4.2. $\Gamma_p = \Gamma_p - G_u$;

Until $\Gamma_p = \varnothing$;

Example 4.1:

Consider the reduced output matrix of Figure 4.1(b). The R_j-sets are generated as follows: $R_6 = \{1,4,5\}$, $R_8 = \{6,7\}$, $R_{15} = \{3,8\}$, and $R_{20} = \{1,2,7,9\}$. Since Column 1 and 7 have 2-count, while the remaining columns have only 1-count, the priority set is $\Gamma_p = <2,3,4,5,6,8,9,1,7>$. By Algorithm I, we first select $\gamma_s = 2$, where $2 \in R_{20} = \{1,2,7,9\}$. Thus, $S_c = S_c - R_{20} = <3,4,5,6,8>$ and $G_1 = \{2\}$. This is followed by selecting $\gamma_s = 3$. This results in $G_1 = \{2,3\}$ and $S_c = S_c - R_{15} = \{4,5,6\}$. After subsequently selecting 4 and 6, we obtain $G_1 = \{2,3,4,6\}$ and $S_c = \varnothing$. Thus, the construction of G_1 set completes. For constructing the G_2 set, $S_c = \Gamma_p = \Gamma_p - G_1 = <5,8,9,1,7>$. Repeating Step 4.1 we obtain $G_2 = \{5,8,9\}$, $G_3 = \{1\}$, and $G_4 = \{7\}$.

Example 4.2:

Consider the MCNC benchmark FSM, *ex1.kiss2*, consisting of 9 inputs, 19 outputs, 20 states, and 138 product terms. Figure 4.2 shows the reduced output matrix which contains only 10 essential rows. Based on the R_j sets listed in Figure 4.2, the priority set C_p= <6,8,11,14,15,16,17,18,19,1,12,2,3,9,13,7,5,10,4>. Repeating Step 4 of Algorithm I, we obtain

$G_1 = \{6,8,11,14,16,17,19,3\}$; $G_2 = \{15,18,1\}$; $G_3 = \{12,2\}$;

$G_4 = \{9\}$; $G_5 = \{13\}$; $G_6 = \{7\}$; $G_7 = \{5\}$; $G_8 = \{10\}$; $G_9 = \{4\}$.

Row Index	Column Index 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9	R_j-Sets
11	1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 2 3 4 5
3	1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0	1 6 7
9	0 1 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0	2 3 4 5 7 9
20	0 1 0 1 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0	2 4 5 8 10
22	0 0 0 1 1 0 1 0 0 1 1 1 1 0 0 0 0 0 0	4 5 7 10 11 12 13
32	0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 0	10 14 15
39	0 0 0 1 1 0 1 0 1 1 0 0 1 0 0 1 0 0 0	4 5 7 9 10 13 16
53	0 0 0 1 1 0 1 0 1 1 0 1 1 0 0 0 1 1 0	4 5 7 9 10 12 13 17 18
99	0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1	4 10 13 19
127	0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0	3 4 5 10

Figure 4.2 The output matrix and R_j-sets for *ex1.kiss2*.

B. Case of $r \geq 2$

For r = 2, the condition, $|G_i \cap R_j| \leq 2$ for all i and essential rows j, implies that a G_i cannot include more than two elements of R_j. The priority set is generated as follows. Consider a pair set Q_j which consists of all pairs of the elements in R_j, i.e., $Q_j = \{(u_1,u_2)|$

all $u_1,u_2 \in R_j\}$. Based on these pair sets, the descending priority set $\Gamma_p = <\gamma_1,\gamma_2,...,\gamma_d>$, where each γ_q is a pair of elements in a R_j and $d = \left|\bigcup_{j=1}^{n} Q_j\right|$ is the number of distinct pairs. Without loss of generality, let $\gamma_q = (u_{q1},u_{q2})$ and $\gamma_p = (u_{p1},u_{p2})$. The priority of γ_q is higher than that of γ_p if (1) the number of occurrences of γ_p is higher than that of γ_q, or (2) when both have the same number of occurrences, $u_{q1} < u_{p1}$, or $u_{q1} = u_{p1}$ and $u_{q2} < u_{p2}$. Note that $Q_j = R_j$ for r = 1.

The same concept can be extended for any r, where the r-tuple sets $Q_j = \{(u_1,u_2,...,u_r) | \text{ all } u_1,u_2,...,u_r \in R_j\}$, and the priority set $\Gamma_p = <\gamma_1,\gamma_2,...,\gamma_d>$, where each γ_q is a n-tuple elements of a R_j and $d=\left|\bigcup_{j=1}^{n} Q_j\right|$. The priority is defined in the same manner as for r = 2.

Similar to Algorithm I, Algorithm W is employed for $r \geq 2$.

Algorithm W

Step 1. Generate the reduced output matrix, derive the R_j sets, and the r-tuple sets $Q_j = \{(u_1,u_2,...,u_r) | \text{ all } u_1,u_2,...,u_r \in R_j\}$, for all $1 \leq j \leq n$.

Step 2. If $|R_{max}| = \max\{|R_i| \; 1 \leq i \leq n\} = m$ or $|R_{max}| \leq r$, then STOP;

/* No partition is needed. */

Step 3. Establish the selection priority and generate the set $\Gamma_p = <\gamma_1,\gamma_2,...,\gamma_d>$, where $d = \left|\bigcup_{j=1}^{n} Q_j\right|$.

Step 4. Selection Processes

u = 0;

Repeat

4.1. $u = u + 1$; $G_u = \emptyset$; $S_c = \Gamma_p$;

Repeat γ_s = the first element of S_c;

$G_u = G_u \cup \{\gamma_s\}$;

$S_c = S_c - \cup \{Q_j \mid \gamma_s \in Q_j\}$;

Until $S_c = \emptyset$;

4.2. $\Gamma_p = \Gamma_p - G_u$;

Until $\Gamma_p = \emptyset$;

Example 4.3:

Consider the reduced matrix of Figure 4.1(c). For r = 2, the Q_j sets are generated as follows: $Q_1 = \{(1,4),(1,5),(4,5)\}$;

$Q_2 = \{(6,7)\}$;

$Q_3 = \{(3,8)\}$; and

$Q_4 = \{(1,2),(1,7),(1,9),(2,7),(2,9),(7,9)\}$.

Thus, the priority set $C_p = <(1,2),(1,4),(1,5),(1,7),(1,9),(2,7),(2,9),(3,8),(4,5),(6,7),(7,9)>$. Similar to Example 4.1, we obtain $G_1 = \{1,2,3,4,8\}$ and $G_2 = \{5,6,7,9\}$.

Similarly, following Algorithm W, the following five 2-set submatrices are generated for *ex1.kiss2* of Figure 4.2 in Example 4.2

$G_1 = \{1,2,6,8,9,11,12,14,15,16,19\}$; $G_2 = \{3,7,10\}$;

$G_3 = \{4,17\}$; $G_4 = \{5,18\}$; and $G_5 = \{13\}$.

4.4 DISCUSSION

The presented algorithm partitions a given k-set matrix C into t submatrices which were assumed to be r-sets, i.e., each submatrix has the same dimension. The question that arised is whether or not this assumption is necessary.

First, a r-set submatrix can be obtained by combining r 1-set submatrices. For example, consider the 1-set submatrices in Example 4.1, the submatrices G_1 and G_3 can be combined as a 2-set submatrix, so do G_2 and G_4. Secondly, the total hardware cost of $1/p_1$, $1/p_2$, ..., $1/p_t$ code checkers is generally much less than that of $t/(p_1+p_2+...+p_t)$. For example, the gate counts for $C_{1/2}$, $C_{1/4}$, and $C_{1/4}$ checkers are respectively 2, 8, and 8, with 3-level implementation. The total is 18 gates which is much less than 127 gates for a 3/10 (=(1+1+1)/(2+4+4)) code checker with the same 3-level implementation. Finally, the most important fact is that Algorithm I for generating 1-set submatrices is simple, yet efficient. Based on the above observation, generating 1-set submatrices for such an implementation is adequate. In other words, we should implement with all 1-set submatrices.

However, implementing many 1-set submatrices is not free of penalty as far as the overall hardware cost is concerned. Consider the structure of a ***self-checking*** sequential circuit, where the next state functions are encoded in the ***m/n*** code and the output functions are encoded in either ***m/n*** or ***Berger code*** [54]. Two ***self-checking*** checkers are employed to monitor the next state functions and the output functions. The outputs of these two checkers are then fed to a two-rail, or $C_{2/4}$, checker to produce the final checker output. Suppose that the output functions have been partitioned into t 1-set submatrices, where each submatrix needs a code checker. Then, the outputs of these t checkers and the outputs of the next state function checker are fed to a (t+1)/2(t+1) code checker to produce the final checker output. (Such an implementation generally results in less performance degradation.) Apparently, the hardware cost, for (t+1)/2(t+1) code checker, increases as t increases. For example, *ex1.kiss2* in Example 4.2 requires a $C_{9/28}$ checker for its output functions and a two-rail checker for the final checker output. With the presented partitioning algorithm, however, the output functions are partitioned into 9 1-set submatrices and the circuit needs 3 checkers for $C_{1/9}$, $C_{1/4}$, and $C_{1/3}$, for $|G_1| = 8$, $|G_2| = 3$, and $|G_3| = 2$, respectively, no checkers for $|G_i| = 1$, $4 \leq i \leq 9$, and a $C_{10/20}$ checker for the final checker output.

In summary, each 1-set submatrix needs a very low cost checker to implement with, but the use of many 1-set submatrices may require a larger checker for the final checker output. Thus, in order to attain the optimal result, it is necessary to generate all 1-set submatrices. However, some of these submatrices with lower cardinality $|G_i|$ should be combined into a larger r-set submatrix to reduce the size of the checker for the final checker output. Experimental results presented in the next section will discuss the trade-off.

4.5 EXPERIMENTAL RESULTS

This section describes the implementation of the partitioning algorithm in the design of ***self-checking*** circuits for reducing hardware overhead and performance degradation. In order to demonstrate the effectiveness of such an implementation, experimental results on MCNC benchmark FSMs are presented.

Table 4.1 shows the experimental results for MCNC benchmark FSMs without implementing the developed partitioning algorithm, where only those FSMs with more than 5 output variables are tested. The columns "#i", "#p", "#s", and "#o" represent the number of inputs, product terms, state variables, and outputs, respectively. The column "max #1's" lists the maximal number of 1's, k, in the rows of each output matrix, i.e., the output matrix is a k-set matrix. The output matrix may be encoded in either ***m/n*** codes or ***Berger codes***.

Since none of existing checker design approaches can be universally, optimally implemented for any code length, the experimental results listed in Table 4.1 are obtained by implementing the design approaches which are selected with the following priority: For ***m/n*** code checkers, (1) if $m \geq 3$, $4m \geq n > 2m$, the efficient checker design [20] is applied; (2) if n is small, the checker design with 3-level implementation [17] or 2-level implementation is employed [15]; (3) if $n = 2m$ or $n = 2m \pm 1$, the efficient checker designs in [14,15] are applied, respectively; and (4) the checker design in [13] is implemented for the remaining cases. On the other hand, for ***Berger code*** checkers, among the design approaches [28-31] the choice is determined by whichever provides the optimal results. Note that ***m/n*** code checker is generally realized with less gate level than the ***Berger code*** checker, but it is achieved at the cost of increasing gate count. However, the ***Berger code*** checker requires much less check bits in the function circuit than the ***m/n*** code checker. Thus, the ***m/n*** code checker is implemented when gate level is of concern.

Table 4.1: Experimental Results.

Benchmark	#i	#p	#s	#o	max. #1's	Checker					
						m/n Checker			Berger		
						code	gate	level	code	gate	level
bbsse.kiss2	7	16	56	7	3	3/10	$127_{\$}$	3	B(7,3)	76_{u}	10
cse.kiss2	7	16	91	7	3	3/10	$127_{\$}$	3	B(7,3)	76_{u}	10
ex1.kiss2	9	20	138	19	9	9/28	$2043_{!}$	10	B(19,5)	$241_{\#}$	20
ex4.kiss2	6	14	21	9	4	4/13	$105_{!}$	7	B(9,4)	$120_{\#}$	14
ex6.kiss2	5	8	34	8	5	5/13	$110_{!}$	7	B(8,4)	$93_{\&}$	12
mark1.kiss2	5	15	22	16	5	5/21	2486_{*}	7	B(16,4)	260_{u}	15
planet.kiss2	7	48	115	19	9	9/28	$2043_{!}$	10	B(19,5)	$241_{\#}$	20
scf.kiss2	27	121	166	56	13	13/69	$-2.0\times10^{9}_{*}$	10	B(56,6)	$687_{\#}$	24
styr.kiss2	9	30	166	10	5	5/15	$283_{!}$	11	B(10,4)	$132_{\#}$	14
opus.kiss2	5	22	10	6	3	3/9	$91_{\$}$	3	B(6,3)	$61_{\&}$	8
s1.kiss2	8	107	20	6	3	3/9	$91_{\$}$	3	B(6,3)	$61_{\&}$	8
sand.kiss2	11	184	32	9	5	5/14	$212_{!}$	10	B(9,4)	$120_{\#}$	14
sse.kiss2	7	56	16	7	3	3/10	$127_{\$}$	3	B(7,3)	76_{u}	10

%: [14]; +: [15]; $: [17]; *: [18]; !: [20]; @: [25]; #: [29]; &: [30]; u : [31].

Table 4.2 lists the experimental results for FSMs with the partitioning algorithm. In this implementation, we first apply Algorithm I, for r = 1, to generate the partitioned1-set submatrices. For instance, the output matrix of *ex4.kiss2*, as shown in Example 4.1, is partitioned into four 1-set submatrices, where $|G_1| = 4$, $|G_2| = 3$, and $|G_3| = |G_4| = 1$. Thus, "4,3,2*" is recorded in the column "Groups" of Table 4.2 for *ex4.kiss2*, where the asterisk indicates the number of 1-set submatrices with $|G_i| = 1$. Once all 1-set submatrices are generated, the submatrices with $|G_i| \leq 2$ are combined and formed into a larger submatrix. Thus, the checkers required for *ex4.kiss2* are $C_{1/5}$, $C_{1/4}$, and $C_{2/4}$. As indicated in Table 4.2, based on the design approaches in [15,17], the checkers need 23 gates and 3 levels.

Consider the FSM, *ex1.kiss2*, as discussed in Example 4.2, its output matrix is partitioned into nine 1-set submatrices, where $|G_1| = 8$, $|G_2| = 3$, $|G_3| = 2$, and $|G_4| = |G_5| = |G_6| = |G_7| = |G_8| = |G_9| = 1$. Thus, "8,3,2,6*" is recorded in column "Groups". The checkers for G_1 and G_2 are $C_{1/9}$ and $C_{1/4}$, respectively. Combining submatrices G_3 through G_9 into a larger submatrix, one may need either a B(8,4) or $C_{7/15}$ checker. As indicated in Table 4.2, the implementation of the checkers for $C_{1/9}$, $C_{1/4}$, and B(8,4) requires 116 gates with 12 levels, while the checkers for $C_{1/9}$, $C_{1/4}$, and $C_{7/15}$ need 304 gates with 3 levels. It should be mentioned that the former implementation requires only 6 check bits in the functional circuit, while the latter one needs 9 check bits. It is obvious that the hardware reduction in the former implementation is achieved at the cost of requiring more gate levels.

In our implementation, an ***m/n*** checker with $n \geq 10$ should be partitioned further in order to eliminate the fan-in constraint for the efficient 2- or 3-level implementation. For example, the 7/15 code in *ex1.kiss2* is partitioned into 3/7 and 4/8 codes, where the efficient ***m/n*** checker design with $n = 2m$ [15] and $n = 2m \pm 1$ [14] can be employed. As indicated in Table 4.2, the implementation of $C_{1/9}$, $C_{1/4}$, $C_{3/7}$, and $C_{4/8}$ checkers requires 73 gates and 3 levels. Compared with 116 gates and 12 levels for $C_{1/9}$, $C_{1/4}$, and B(8,4) checkers and with 304 gates and 3 levels for $C_{1/9}$, $C_{1/4}$, and $C_{7/15}$ checkers, the reductions in both gate count and level are significant.

Table 4.2: Experiment Results with Output Partitioning Scheme.

Benchmark	#i	#p	#s	#o	Groups	Checker	gate	level
bbsse.kiss2	7	16	56	7	5,2*	1/6,2/4	$17_{\$}$	3
cse.kiss2	7	16	91	7	3,2,2	1/4,2/6	$28_{\$}$	3
ex1.kiss2	9	20	138	19	8,3,2,6*	1/9,1/4,B(8,4)	$116_{\$\#}$	12
						1/9,1/4,7/15	$304_{\$\%}$	3
						1/9,1/4,3/7,4/8	$73_{\$+}$	3
ex4.kiss2	6	14	21	9	4,3,2*	1/5,1/4,2/4	$23_{\$+}$	3
ex6.kiss2	5	8	34	8	3,5*	1/4,B(5,3)	$80_{\$\#}$	10
						1/4,5/10	$68_{\$+}$	3
						1/4,2/4,3/6	$28_{\$+}$	3
mark1.kiss2	5	15	22	16	9,3,2,2*	1/10,1/4,3/5	$39_{\$\%}$	3
planet.kiss2	7	48	115	19	6,3,2,2,6*	1/7,1/4,4/9,4/9	$110_{\$\%}$	3
scf.kiss2	27	121	166	56	17,9,5,4,4,3, 2,2,2,2,6*	1/18,1/10,1/6,1/5,1/5,1/4,B(14,4)	$296_{\$\#}$	14
						1/18,1/10,1/6,1/5,1/5,1/4,10/24	$2189_{\$*}$	6
						1/18,1/10,1/6,1/5,1/5,1/4,2/4,4x(2/5)	$141_{@\$}$	6
styr.kiss2	9	30	166	10	5,2,3*	1/6,4/9	$56_{\$\%}$	3
opus.kiss2	5	22	10	6	3,3*	1/4,3/6	$22_{\$+}$	3
s1.kiss2	8	107	20	6	4,2*	1/5,2/4	$15_{\$}$	3
sand.kiss2	11	184	32	9	4,5*	1/5,5/10	$69_{\$+}$	3
sse.kiss2	7	56	16	7	5,2*	1/6,2/4	$17_{\$}$	3

*: All are 1-set.

Similarly, in addition to the checkers for $C_{1/18}$, $C_{1/10}$, $C_{1/6}$, $C_{1/5}$, $C_{1/5}$, and $C_{1/4}$, the *scf.kiss2* may be implemented with either a B(14,4) or $C_{10/24}$ checker. However, due to the property of "2,2,2,2,6*" in the partitioned submatrices, the $C_{10/24}$ can be partitioned into a $C_{2/4}$ and four $C_{2/5}$'s. Compared with 2110 gates and 6 levels for a $C_{10/24}$ checker, it requires only 62 gates and 3 levels for the smaller checkers. As a result, the implementation is even better than that with ***Berger code*** in both gate count and level.

In order to demonstrate the effectiveness of the partitioning algorithm, Table 4.3 compares the experimental results for MCNC benchmark FSMs with and without implementing the partitioning algorithm. As mentioned, without the partitioning algorithm, a 2/4 code checker, realized by 6 gates and 2 levels, is needed for the final checker output, while a (t+1)/2(t+1) code checker is required for implementing with the partitioning algorithm, where t is the number of the smaller checkers which constitute the ***self-checking*** checker. Thus, a $C_{3/6}$ checker, realized by 20 gates and 2 levels, is employed for those MCNC benchmark FSMs, *bbsse*, *cse*, *styr*, *opus*, *s1*, *sand*, and *sse*. For *scf*, it requires a $C_{12/24}$ checker for the final checker output. In order to reduce the hardware cost, while still keeping less gate levels, the $C_{12/24}$ is partitioned into four $C_{3/6}$'s and their outputs are checked by a $C_{4/8}$ checker. As a result, these smaller checkers requires only 85 gates and 6 levels, or 160 gates and 4 levels.

For any circuit realization, there exists an inverse proposition relationship between area and delay. The reduction in gate count is generally achieved at the cost of increasing gate level, and vice versa. In Table 4.3, the columns "***m/n*** code" and "***Berger*** code" list the gate counts and gate levels of various circuits without applying the partitioning scheme. The m/n code checker takes less gate level than the corresponding Berger code checker, but it requires more gate count. The remaining columns indicate the experimental results of the circuits with the partitioning scheme. In our partitioning scheme, the optimization may be achieved by optimizing gate count, or gate level, or gate count under gate level constraint. The gate count optimization attempts to obtain the minimal gate count regardless of gate

Table 4.3: Comparisons.

Benchmark	m/n Code		Berger Code		Algorithm I					
					gate count optimal		gate level optimal		with Berger code	
	gate	level	gate	level	gate	level	gate	level	gate	level
bbsse.kiss2	133	5	82	12	29	7	39	5	-	-
cse.kiss2	133	5	82	12	40	7	50	5	-	-
ex1.kiss2	2049	12	247	22	83	9	327 133	5 6	189	14
ex4.kiss2	111	9	126	16	41	7	95	5	-	-
ex6.kiss2	116	9	99	14	46	7	100	5	100	12
mark1.kiss2	2492	9	266	17	64	7	118	6	-	-
planet.kiss2	2049	12	247	22	104	9	364 170	5 6	-	-
scf.kiss2	$\sim 2 \times 10^9$	12	693	26	179	11	241	10	408	18
styr.kiss2	289	13	138	16	49	7	78	5	-	-
opus.kiss2	91	5	67	10	36	6	44	5	-	-
s1.kiss2	91	5	67	10	29	6	37	5	-	-
sand.kiss2	212	12	126	16	47	7	91	5	-	-
sse.kiss2	127	5	82	12	31	6	39	5	-	-

level. Similarly, the gate level optimization targets to reduce gate level without considering gate count. Both optimization results are listed in Table 4.3. Interestingly, the gate count and gate level in both optimizations are much better than those without the partitioning scheme listed in the second and third columns. It should be mentioned that the use of Berger code generally requires less check bits than that of m/n code. This implies that the reduction in gate level of the checker circuits may be achieved at the cost of increasing the number of check bits, and, thus, increasing both gate level and gate count in the functional circuit. Therefore, for a practical design, the circuit should be synthesized such that the total gate count in both functional circuit and checker circuits is optimized under gate level constraint. This leads to the development of an automatic synthesis system which optimizes the gate count for both functional circuit and checker circuit and keeps a reasonably low gate level.

4.6 SUMMARY

This chapter has presented an efficient output matrix partitioning algorithm for $r = 1$ and its extension for $r \geq 2$. The presented algorithm has been implemented in the design of self-checking circuits and systems for MCNC Benchmark FSMs for reducing both hardware overhead and performance degradation of the required checkers. The significant reduction has been illustrated from the experimental results. The algorithm particularly benefits those circuits with more output functions.

It should be mentioned that the problem of partitioning a k-set output matrix into a minimal number of 1-set submatrices is similar to the problem of *chromatic partitioning of a graph*, where $|R_{max}| = k$ is the *chromatic number*. Since the chromatic partitioning problem is NP-complete [65], any efficient heuristic algorithm developed for the chromatic partitioning problem can also be implemented to improve our experimental results.

However, since not all ***m/n*** and ***Berger code*** checkers have been efficiently designed for any code length, the size of the partitioned submatrices should be determined by the availability of the checker design.

As shown in Figure 4.1, or in Example 1, the output portion of a FSM is used as the output matrix for generating 1-set submatrices. A ***self-checking*** FSM is generally synthesized in such a way that each state is assigned a code, referred to as codeword, and the outputs corresponding to the non-codewords are assigned to all 0's. Based on this state assignment, any two product terms will be disjoint. In other words, only one product term is enabled at a time when an input is applied. Thus, the output portion of a FSM can be used as the output matrix.

CHAPTER 5

SOLiT: A SYSTEM FOR AUTOMATED SYNTHESIS OF ON-LINE TESTABLE SEQUENTIAL CIRCUITS

This chapter presents a system, namely, ***SOLiT***, for automated Synthesis of ***On-Line*** Testable sequential circuits with multi-level logic implementation. A ***self-checking*** circuit is comprised of a functional circuit and a ***self-checking*** checker. The functional circuit, either a combinational or a sequential circuit, is generally designed such that its primary outputs and some encoded outputs are able to produce an erroneous result in the presence of fault(s). The ***self-checking*** checker is designed to produce an error signal for some normal circuit inputs whenever a fault from a specified set of faults occurs within the circuit. As described in the previous chapters, numerous designs of ***self-checking*** checkers have been presented. The synthesis of ***self-testing*** functional circuits is presented in Section 5.1. Section 5.2 describes the major components in ***SOLiT*** and its implementation is discussed in Section 5.3. In order to demonstrate the automated synthesis system, a complete example from design specification to the physical layout is presented in Section 5.4.

5.1 SYNTHESIS OF SELF-CHECKING FUNCTIONAL CIRCUITS

Consider a Mealy-type sequential circuit, as shown in Figure 5.1. (The same concept can be easily extended for Moore-type sequential circuits.) The combinational part is realized with multi-level logic implementation. The state is encoded in the ***m/n*** code, while the output functions are encoded in either ***Berger codes*** or ***m/n*** codes for detecting unidirectional errors. It has been shown that, with the above encoding, the circuit output and the next state output functions are unate in state variables [53]. Thus, unidirectional errors can be detected. Note that the inputs need not be encoded if they are obtained from a preceding functional circuit. As shown in Figure 5.1, the circuit requires two TSC checkers to monitor the circuits outputs: one checker (***m/n*** checker) checks the next state functions, and the other (either ***m/n*** checker or ***Berger code*** checker) checks the next state functions. The outputs of these two checkers are then fed to a two-rail checker to produce the final checker output.

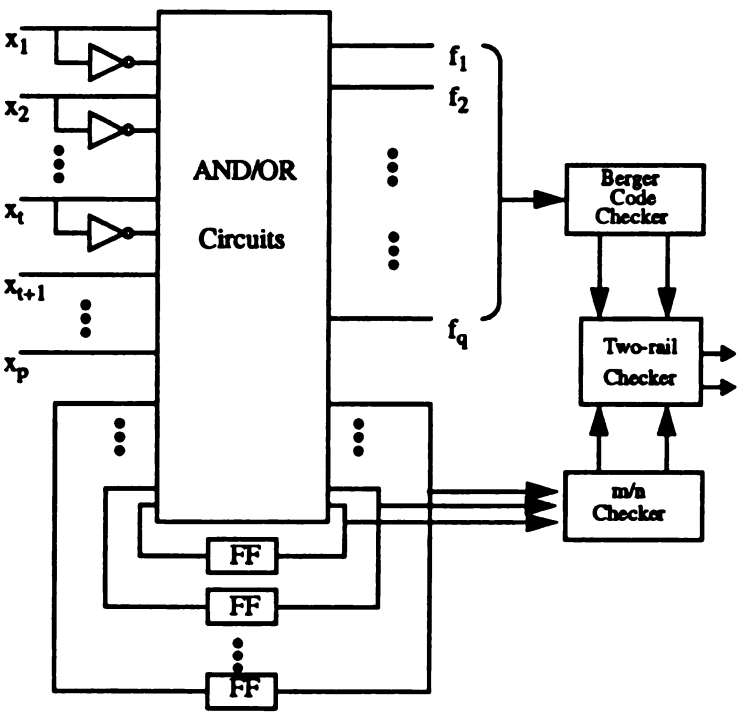


Figure 5.1 A Moore-type self-checking sequential circuit.

For detecting multiple unidirectional errors, the encoded output functions of the functional circuit must be realized with inverter-free circuits. The combinational part must be inverter-free, i.e., with AND and OR gates, and the only inverters allowed are at the primary inputs. Based on this realization, the output functions are binate in terms of inputs x_i, $1 \leq i \leq t$, and unate in terms of inputs x_j, $t+1 \leq j \leq p$. If the primary inputs are connected from previous stages which are encoded and have been detected as fault-free, then the multiple unidirectional errors can be detected. Otherwise, the circuit can be guaranteed only to detect single stuck-at faults instead of multiple unidirectional stuck-at faults.

5.2 THE AUTOMATED SYNTHESIS SYSTEM *SOLiT*

The conventional synthesis procedure for sequential circuits includes the following three steps: (1) state assignment (or encoding); (2) logic minimization (two-level or multi-level); and (3) technology mapping. The state assignment process is to encode each state with a binary vector. Once the states are encoded, logic synthesis including logic minimization and technology mapping are used to optimally realize the circuit with either gate arrays or standard cells available in a cell library.

SOLiT is a system for automated synthesis of ***on-line*** testable sequential circuits, with multi-level logic implementation. Figure 5.2 illustrates the flow chart of ***SOLiT*** which is comprised of the following five major components: (1) output function partitioning and encoding; (2) state encoding; (3) logic minimization and technology mapping; (4) checker library; and (5) layout synthesis. The output function partitioning procedure partitions the output functions of a circuit during synthesizing into many smaller sub-functions and the resultant output sub-functions are encoded with either ***m/n*** or ***Berger codes***. The checker library is comprised of all possible TSC ***m/n*** code checkers and ST ***Berger code*** checkers. The layout synthesis includes the placement and routing of the functional circuits, checkers, and interconnections.

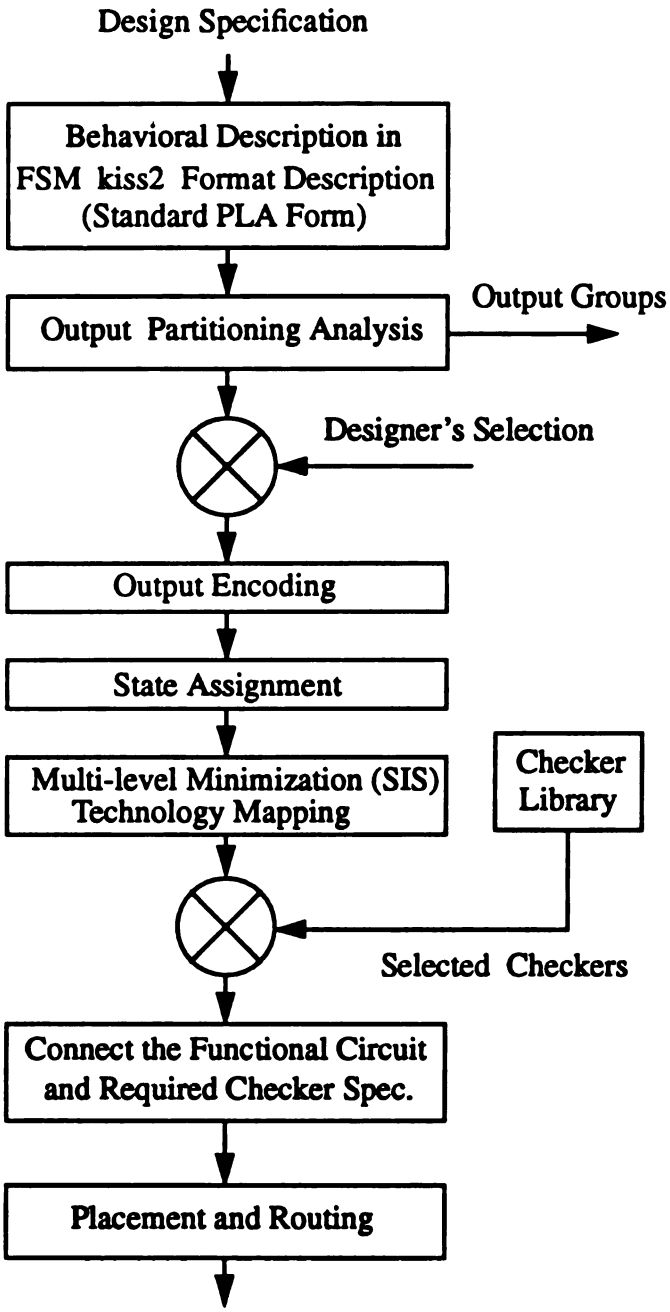


Figure 5.2 Flow chart for *SOLiT*.

5.3 IMPLEMENTATION

SOLiT has been implemented on Sun/4 workstation in the C-language. Since the finite state machines (FSMs) in the MCNC benchmarks are represented in either ***kiss2*** or ***blif*** format, the system takes the FSMs described in ***kiss2*** format as input. However, it can be easily modified for any other format.

Output Partitioning and Encoding

The output portion of the ***kiss2*** file is partitioned by the algorithms developed in Chapter 4. A procedure, namely, ***out_part***, is developed to partition the set of output functions into many independent subsets. Each independent set requires a checker. Thus, the number of independent sets determines the number of checkers required in a realized circuit. As described in Section 4.4, there exists a trade-off between the size of the independent sets and the hardware cost. The strategy presented in Section 4.5 has been taken into account in this program development. The experimental results have been shown in Table 4.2 (in Section 4.5).

Once the set of output functions are partitioned, each subset is encoded individually. The output functions are encoded in either Berger or ***m/n*** codes depending upon whichever provides an optimal solution. (Based on the designer's choice in either area-optimum or time-optimum.) The procedure, namely, ***out_encode***, receives the input information which includes the partitioned groups in the output functions, selected encoding scheme, and area- or time-optimum, and produces the appropriate encoded output sub-functions for that circuit.

State Encoding

The VLSI design tool, ***mustang***, is generally used for state encoding targeting multi-level logic implementation. However, the program only provides 1/n codes, but not

for ***m/n*** codes. In order to assign ***m/n*** codes to the next state functions, a program, namely ***assigner***, is developed in which each state can be assigned sequentially or randomly. In this implementation, the states are assigned sequentially, for example, for a 2/5 code, State 1 is assigned as (00011), State 2 as (00101), State 3 as (00110), and etc. In order to make the next state functions unate, the present states are modified by changing "0" to "-" ("don't care"). For example, if the present state is (00101), it is modified as (--1-1). The resultant next state functions are referred to as *modified encoded state functions*. Note that, as shown in Figure 5.1, the combinational part of the circuit must be realized with AND and OR gates only.

Logic Synthesis - Logic Minimization and Technology Mapping

The functional circuit is realized with multi-level logic. In order to keep the output functions and the next state functions unate, the functions are optimized by a multi-level logic minimizer, ***sis***, with algebraic decompositions. Then, the technology mapper in ***sis*** is used to map the resultant circuit network to AND and OR gates in a cell library which is generated from the cell library in ***sis***.

Checker Library

In order for checkers to check their own faults, they must also be inverter-free circuits. As discussed in Chapter 2, numerous ***Berger*** and ***m/n*** code checkers have been developed, but not for all possible code lengths. The checker library is a collection of modules which implement the existing checker designs with AND and OR gates. The checker library provides the information of the size and delays for each checker. For example, a gate-level 2/7 code checker is shown in Example 2.2, and the cell-level is illustrated in Figure 5.3. The modules are designed such that it has the same height as the AND and OR cells.

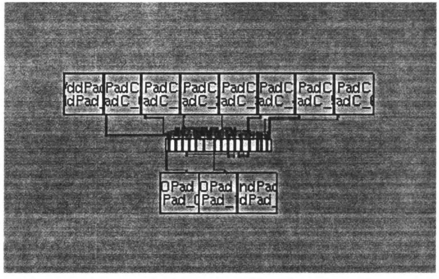


Figure 5.3 Cell-level implementation of 2/7 code checker.

Layout Synthesis - Placement and Routing

Once the net-list of the ***self-checking*** circuit is available, the VLSI tool implementing Timberwolf placement algorithm, ***twp*** [66], is employed for placing the components in the functional circuits and the checkers. Note that the placement of the functional circuits and the checker circuits are performed simultaneously, where the checker circuits are pre-designed as macro modules. The program generates a ***cif*** file for routing. where a software program, namely, ***aisce*** [67], a modified version of the layout editor, ***magic***, is employed.

5.4 SYNTHESIS EXAMPLE

In order to demonstrate the developed synthesis procedure in ***SOLiT***, the MCNC benchmark FSM, *mark1.kiss2*, as shown in Figure 5.4, is generated. First, based on the

```
.i 5
.o 16
.p 22
.s 15
0---- *      state1  -11---1-00------
1---- state1 state3  -11---1-00------
1---- state2 state0  -11---1-00------
1---- state3 state4  101---1-01------
1-111 state4 state13 -11---1-00------
1-110 state4 state10 -11---1-00------
1-10- state4 state9  -11---1-00------
1-011 state4 state8  -11---1-00------
1-010 state4 state7  -11---1-00------
1-001 state4 state6  -11---1-00------
1-000 state4 state5  -11---1-00------
1---- state5 state14 0011--1-00------
1---- state6 state14 00100-0-00000011
1---- state7 state14 001---1100------
1---- state8 state14 010---1-00------
1---- state9 state14 001---1010000101
1---- state10 state11 -11---1-00100000
10--- state11 state13 -11---1-00------
11--- state11 state12 -11---1-00------
1---- state12 state13 -110110-00------
1---- state13 state14 -11---1-00------
1---- state14 state3  -110110-00------
```

Figure 5.4 A FSM example - *mark1.kiss2*.

output portion of *mark1.kiss2*, the output partitioning procedure, ***out_part*** produces three partitioned independent sets, as shown in Figure 5.5, where the algorithm first generates five independent sets which contain 9, 3, 2, 1, and 1 column(s), respectively. The procedure suggests three groupings as shown. In addition, the procedure also generates an output file, as shown in Figure 5.6, which is used as an input for procedure ***out_encode*** for the output function encoding. The first line in Figure 5.6 indicates the number of groupings and area-optimum option, where the options "a" and "t" indicate area-optimum and time-optimum, respectively. Each group is described by two lines with the information of the encoding scheme ("m" for ***m/n*** code and "b" for ***Berger code***), m and n for ***m/n*** code or I and K for ***Berger code***, the number of columns, and the column indices. As the second line indicates in Figure 5.6, the first group is encoded in ***m/n*** code, where m = 1 and n = 10, and contains 9 columns which are 1, 4, 5, 8, 9, 11, 12, 13, and 15. The second group is encoded in 1/4 code which contains three columns, 6, 10, and 14; Finally, the third group is encoded in 1/5 code and has four columns, 2, 3, 7, and 16. This concludes that the output sub-functions are encoded in 1/10, 1/4, and 3/5 codes, respectively and require three extra columns for the check part. In this procedure, the designer is allowed to modify the selected code scheme and grouping scheme. The input file (in Figure 5.6) is applied to Procedure ***out_encode*** to encode the output function, as the output portion shown in Figure 5.7.

Once the output functions are encoded, the program, ***assigner***, is executed for encoding states. As shown in Figure 5.7, the states are encoded in 3/6 codes, where the "0"s in the present state are changed to "-"s. The encoded states and output functions are then optimized by the multi-level optimizer, ***sis***, and the resultant network is shown in Figure 5.8, where only AND and OR gates are realized in the combinational part and the inverters appear only in the primary inputs. Thus, the output functions are unate in terms of the inputs v1, v5, V6, and v9, and binate in terms of the remaining primary inputs. According to Figure 5.9, the inverters are also needed for the functions v11.1 and 11.3 which are the output of the state variables. Since a D flip-flop provides Q and $\overline{Q}$ outputs, and v11.1 and v11.3 are the outputs of flip-flops, the inverting functions can be absorbed by using the $\overline{Q}$ outputs.

```
The product term under consideration
Row#                              # of 1's
  4 1010001001000000                   4
 12 0011001000000000                   3
 13 0010000000000011                   3
 14 0010001100000000                   3
 16 0010001010000101                   5
 17 0110001000100000                   4
 20 0110110000000000                   4

Occurence Table
Occ.  Column numbers
  1    1  4 5 6 8 9 10 11 14 15
  2    2 16
  5    7
  7    3

Selected Indepentent Sets:

 1 4 5 8 9 11 12 13 15
 6 10 14
 2 16
 7
 3

Suggested Groupings
1 4 5 8 9 11 12 13 15
6 10 14
2 3 7 16
```

Figure 5.5 Output of *Procedure out_part.*

```
3 a
m 1 10 9
1 4 5 8 9 11 12 13 15
m 1 4 3
6 10 14
m 1 5 4
2  3  7 16
```

Figure 5.6 Input file to *Procedure out_encode.*

```
.i 5
.o 19
.p 22
.s 15
0---- *       state1  01100010000000000110
1---- state1  state3  01100010000000000110
1---- state2  state0  01100010000000000110
1---- state3  state4  10100010010000000001
1-111 state4  state13 01100010000000000110
1-110 state4  state10 01100010000000000110
1-10- state4  state9  01100010000000000110
1-011 state4  state8  01100010000000000110
1-010 state4  state7  01100010000000000110
1-001 state4  state6  01100010000000000110
1-000 state4  state5  01100010000000000110
1---- state5  state14 00110010000000000011
1---- state6  state14 00100000000000011011
1---- state7  state14 00100011000000000011
1---- state8  state14 01000010000000000111
1---- state9  state14 00100010100001010000
1---- state10 state11 01100010001000000010
10--- state11 state13 01100010000000000110
11--- state11 state12 01100010000000000110
1---- state12 state13 01101100000000000001
1---- state13 state14 01100010000000000110
1---- state14 state3  01101100000000000001
```

Figure 5.7 Encoded output functions.

```
.p 22
# state1  ---111 000111
# state3  --1-11 001011
# state2  --11-1 001101
# state0  --111- 001110
# state4  -1--11 010011
# state13 -1-1-1 010101
# state10 -1-11- 010110
# state9  -11--1 011001
# state8  -11-1- 011010
# state7  -111-- 011100
# state6  1---11 100011
# state5  1--1-1 100101
# state14 1--11- 100110
# state11 1-1--1 101001
# state12 1-1-1- 101010
.i 11
.o 25
.type fr
0---- ------ 000111 01100010000000000110
1---- ---111 001011 01100010000000000110
1---- --11-1 001110 01100010000000000110
1---- --1-11 010011 10100010010000000001
1-111 -1--11 010101 01100010000000000110
1-110 -1--11 010110 01100010000000000110
1-10- -1--11 011001 01100010000000000110
1-011 -1--11 011010 01100010000000000110
1-010 -1--11 011100 01100010000000000110
1-001 -1--11 100011 01100010000000000110
1-000 -1--11 100101 01100010000000000110
1---- 1--1-1 100110 00110010000000000011
1---- 1---11 100110 00100000000000011011
1---- -111-- 100110 00100011000000000011
1---- -11-1- 100110 01000010000000000111
1---- -11--1 100110 00100010100000101000
1---- -1-11- 101001 01100010001000000010
10--- 1-1--1 010101 01100010000000000110
11--- 1-1--1 101010 01100010000000000110
1---- 1-1-1- 010101 01101100000000000001
1---- -1-1-1 100110 01100010000000000110
1---- 1--11- 001011 01101100000000000001
```

Figure 5.8 Resultant state encoding with *assigner*.

```
INORDER = v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10;
OUTORDER = v11.0 v11.1 v11.2 v11.3 v11.4 v11.5 v11.6 v11.7 v11.8
v11.9 v11.10 v11.11 v11.12 v11.13 v11.14 v11.15 v11.16 v11.17
v11.18 v11.19 v11.20 v11.21 v11.22 v11.23 v11.24;
v11.0 = v0*v11.22*[151] + v0*!v8*v11.22 + v0*!v10*!v11.1 + [163] +
[162];
v11.1 = [149]*[152]*[155] + !v4*[149]*[155];
v11.2 = v4*v11.1*!v11.3 + v0*!v5*v10 + v9*[162] + v0*[151] +
v2*v11.1 + [160];
v11.3 = !v2*!v3*v4*v11.1 + v8*!v11.1*[155] + !v7*v10*[150] +
v3*!v4*v11.1 + [157] + [10];
v11.4 = v2*v4*v8*[155] + !v4*v11.1*[152] + !v1*!v8*[150] +
v9*[151] + v8*[10] + [157] + v11.16;
v11.5 = v8*v9*[154] + !v2*v11.1*!v11.3 + !v3*v11.1*v11.3 +
v1*!v8*v11.3 + v7* [10] + [160] + [148];
v11.6 = [15];
v11.7 = [158] + [10];
v11.8 = [160] + [157] + [150] + v11.16 + [10];
v11.9 = v9*v10*[163];
[10] = v0*v5*[151] + v10*[162];
v11.10 = [10];
v11.11 = [10];
v11.12 = [158] + [153] + [15] + [14];
v11.13 = v9*[164];
[14] = [155]*[163];
v11.14 = [14];
[15] = [149]*[151];
v11.15 = [15];
v11.16 = v7*[164];
v11.17 = 0;
v11.18 = 0;
v11.19 = [14];
v11.20 = v10*[164];
v11.21 = v11.20 + [14];
v11.22 = v6*[151] + [150] + !v0;
v11.23 = [158] + [153] + v11.20;
v11.24 = 0;
[129] = v8*[154] + v5*v10 + v7*v8 + [151];
[148] = v6*v7*[129] + !v0;
[149] = v0*v8;
[150] = v9*[129];
[151] = v7*v10;
[152] = !v3 + !v2;
[153] = v11.13 + v11.9;
[154] = v10 + v5;
[155] = v7*v9;
[157] = [15] + !v0;
[158] = [150] + [148];
[160] = [153] + v11.21;
[162] = v5*[149];
[163] = v0*v6;
[164] = v6*[149];
```

Figure 5.9 Resultant optimized network.

The ***sis*** technology mapper is employed to map the resultant optimized network (in Figure 5.9) to the AND and OR gates of the modified cell library. Figure 5.10 shows the partial netlist generated from the ***sis*** technology mapper.

Finally, the placement program, ***twp***, and the routing program, ***aisce***, are executed to generate the physical layout for the functional circuits and the checker circuit which are shown in Figures 5.11(a) and 5.11(b), respectively. The functional circuit has a core dimension of 616x1455 (=896280)λ^2, while the checker circuit takes the area of 368x1550(=570400) λ^2. Figure 5.12 illustrates the physical layout of the complete example. The complete circuit is comprised of the encoded functional circuit and the checker circuits. In this example, the 1/10, 1/4, and 3/5 code checkers are used for the output functions. The states are encoded in 3/6 codes and thus need a 3/6 code checker, referred to as the state checker. The outputs of the state checker and the output function checker are fed to three 2/4 checkers as a final checker. The total dimension is 896280+570400(=1466680)λ^2. For the purpose of comparison, Figure 5.13 illustrates the layout of the same FSM which is generated by using the conventional synthesis procedure. It takes the area of 551x1350(=743850)λ^2 which is smaller than the area of the layout in Figure 5.12 with the ***on-line*** testable capability. If the circuit in Figure 5.13 is to achieve the same degree of testability and reliability, one may apply the duplication with comparison approach described in Chapter 1. This implies that the circuit requires a area of 2x551x1350(=1487700)λ^2 excluding the area of the ***self-checking*** comparator. In general, the ***self-checking*** comparator takes a considerably large amount of chip area. Thus, the total area of the circuit presented in Figure 5.12 is much smaller than the circuit in Figure 5.13, the duplication with comparison scheme.

```
.model mark1
.inputs v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10
.outputs v11.0 v11.1 v11.2 v11.3 v11.4 v11.5 v11.6 v11.7 v11.8
v11.9 v11.10 v11.11 v11.12 v11.13 v11.14 v11.15 v11.16 v11.17
v11.18 v11.19 v11.20 v11.21 v11.22 v11.23 v11.24
.default_input_arrival 0.00 0.00
.default_output_required 0.00 0.00
.default_input_drive 0.10 0.10
.default_output_load 2.00
.latch v5 v11.0 1
.latch v6 v11.1 0
.latch v7 v11.2 1
.latch v8 v11.3 0
.latch v9 v11.4 0
.latch v10 v11.5 1
.names v5 [655]
0 1
.names v0 [656]
0 1
.names v8 [657]
0 1
.names [656] [657] [658]
1- 1
-1 1
.names [655] [658] [659]
1- 1
-1 1
.names [659] [954]
0 1
```

•

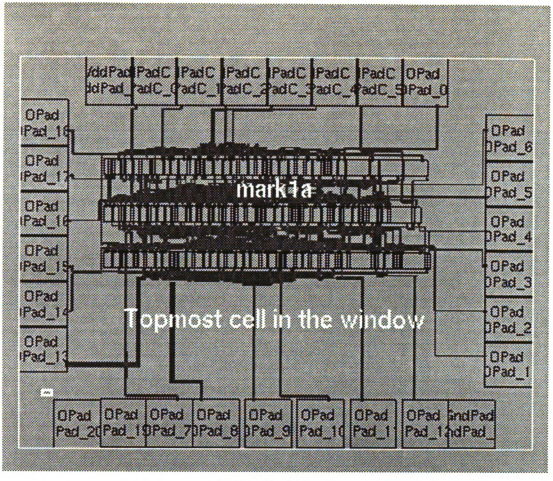
•

•

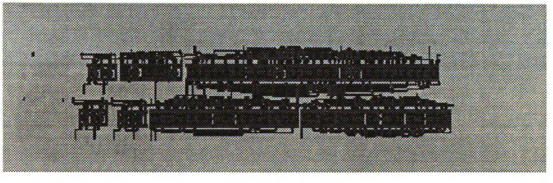
```
.names v0 v5 v6 v7 v8 v9.18
0---- 1
-0--- 1
---1- 1
--1-1 1
.names v0 v5 v6 v7 v8 v9.19
0---- 1
-0--- 1
---1- 1
--1-1 1
.end
```

Figure 5.10 Netlist generated by *sis* technology mapper.

(Only partial list is shown).



(a) functional circuit.



(b) checker circuit.

Figure 5.11 Physical layouts.

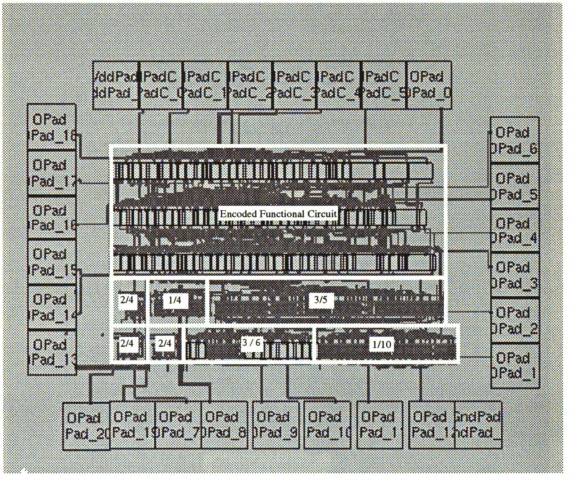


Figure 5.12 Layout of an on-line testable sequential circuit, *mark1.kiss2*.

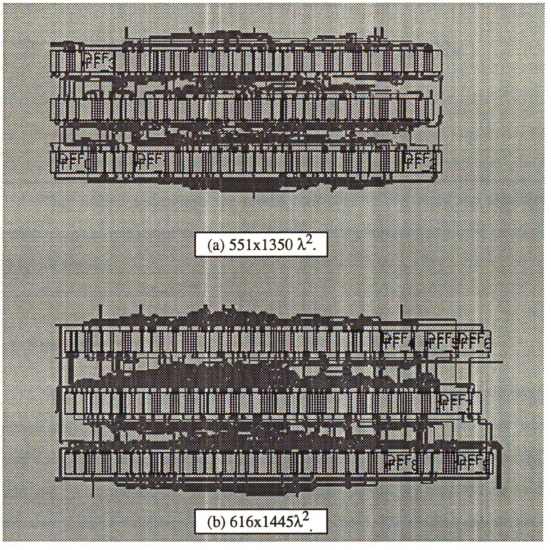


Figure 5.13 Layout of *mark1.kiss2* with (a) the conventional synthesis procedure; and (b) encoded functional circuit.

CHAPTER 6

SUMMARY AND CONCLUSION

This chapter summarizes the results concluded in this study and identifies the contributions and impacts of this research. Finally, some interesting problems are identified for future research.

6.1 Summary

With ever-increasing complexity of digital applications, the issue of reliability has become very important in today's VLSI designs. ***Concurrent error detection*** schemes using redundancy design approach have been successfully implemented to enhance chip yield and system reliability. The redundancy design approaches may include hardware redundancy, time redundancy, and information redundancy. The simplest ***concurrent error detection*** scheme involves duplication of circuit and comparing the outputs of the two blocks, referred to as duplication with comparison scheme. Any mismatch between them will indicate an error. However, this involves more than 100% hardware.

Information redundancy involves the use of coding techniques that enhance circuit capacity for reliable operation. Due to the salient feature of least redundant separable codes, ***Berger codes*** have been implemented for fault-tolerant, fail-safe, and ***concurrent error detection*** designs of digital circuits. The features of high speed and low hardware cost are

highly desirable especially for a checker design. In response to these needs, a design methodology using the partitioning and folding scheme is developed for the design of fast, yet low hardware cost ***Berger code*** checkers with ***self-testing*** capability. Table 3.9 has summarized the significance of the developed checker design comparing it with existing designs.

As shown in Table 3.9, the code length of a checker grows linearly with the number of outputs in a functional circuit, while the complexity of the checker may grow rapidly. As a result, a functional circuit may be optimized in its hardware overhead at the cost of requiring a larger checker. As such, the overall hardware cost for the entire ***self-checking*** circuit is not reduced through the optimization procedure. In addition, if the speed of the larger checker exceeds the clock cycle time for the functional circuit, then the system cycle time must be increased to capture the error signals. As shown in Table 3.9, the hardware cost of the checkers increases almost exponentially as the code length increases. Thus, the use of many smaller code checkers may take much less hardware cost and delay time than those of a larger checker with the same information bits. Thus, in Chapter 4, an efficient output function partitioning scheme is developed to partition the set of output functions to many smaller subsets so that smaller checkers can be employed. Experimental results listed in Table 4.3 have shown that, with the developed partitioning scheme, the hardware cost can be reduced considerably. With such low hardware cost checkers, ***on-line*** testable design becomes very promising and practical.

In this research, a system, ***SOLiT***, for automated synthesis of ***on-line*** testable sequential circuits with multi-level implementation has been developed and illustrated in Figure 5.2. The system is implemented on Sun/4 workstation in the C-language. The system receives a behavioral description of finite state machines in ***kiss2*** format, and automatically generates a physical layout for a ***self-checking*** circuit. Results show that the circuits generated by this system take much smaller chip area and delay time than those with the duplication with comparison scheme.

6.2 CONTRIBUTIONS AND FUTURE RESEARCH

The contributions and impacts of this research can be summarized as follows:

(1) developed a fast, yet low hardware cost ***Berger code*** checker;

(2) developed an efficient output partitioning algorithm for reducing hardware cost and speed degradation of checker circuits; and

(3) developed an automated synthesis system for ***on-line*** testable sequential circuits with multi-level logic implementation.

Based on the results of this study, in addition to hardware cost and delay time, reliability now can also be one of the major design objectives. Several improvements that can be done for upgrading the synthesis system, ***SOLiT***, are summarized as follows:

(1) Design Specification

In ***SOLiT***, it receives the input file of a FSM in ***kiss2*** format. This is mainly because the existing benchmarks are in such a format. The system is readily adapted for the specification in any other format.

(2) Output Partitioning Algorithm

In procedure ***out_part*** of ***SOLiT***, the independent sets of a given output function are generated. The program provides the suggested output groupings. The optimal choice of output groupings should be measured by the overall hardware cost and delay time. Thus, developing an efficient decision algorithm for global optimization can further improve the results.

(3) State Encoding

As mentioned in Section 5.3, the state variables are encoded sequentially. It did not take the circuit behavior, not structure, into account. The synthesized results would be improved significantly if an efficient state assignment for multi-level logic implementation is developed.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] B. W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Co., Reading, Mass., 1989.

[2] C. L. Wey, "Concurrent Error Detection in Array Dividers by Alternating Input Data," *IEE Proceedings-E*, Vol. 139, pp.123-130, March 1992.

[3] P. K. Lala, *Fault Tolerant & Fault Testable Hardware Design*, Prentice-Hall, Englewood Ciffs, N. J., 1985.

[4] T. R. N. Rao and E. Fujiwara, *Error-Control Coding for Computer Systems*, Prentice-Hall, Englewood Ciffs, N. J., 1989.

[5] M. M. Yen, W. K. Fuchs, and J. A. Abraham, "Designing for Concurrent Error Detection in VLSI: Application to Microprogram Control Unit," *IEEE Journal of Solid-State Circuits*, Vol. SC-22, pp.595-605, August 1987.

[6] C. L. Wey, "On Yield Consideration for the Design of Redundant Programmable Logic Arrays," *IEEE Trans. on Computer-Aided-Design*, Vol. 7, pp.528-535, April 1988.

[7] Y. Tamir and C. H. Sequin, "Design and Application of Self-Testing Comparators Implemented with MOS PLA's," *IEEE Trans. on Computers*, Vol. C-33, pp.493-506, June 1984.

[8] J. L. A. Hughes, E. J. McCluskey, and D. J. Lu, "Design of Totally Self-Checking Computers with an Arbitrary Number of Inputs," *IEEE Trans. on Computers*, Vol. C-33, pp.546-550, June 1984.

[9] R. W. Cook, W. H. Sisson, T. F. Storey, and W. N. Toy, "Design of a Self-Checking Microprogram Control," *IEEE Trans. on Computers*, Vol. C-22, pp.255-262, March 1973.

[10] D. K. Pradhan and J. J. Stiffler, "Error-Correcting Codes and Self-Checking Circuits," *IEEE Trans. on Computers*, Vol. 13, pp.27-37, March 1980.

[11] N. K. Jha and M. B. Vora, "A Systematic Code for Detecting t-Unidirectional Errors," *Proc. International Symp. on Fault-Tolerant Computing*, pp.96-101, July 1987.

[12] S. J. Piestrak, "Design of a Self-Testing Checker for Borden Code," *Proc. International Conference on Computer Design: VLSI in Computers*, pp.582-585, October 1991.

[13] M. A. Marouf and A. D. Friedman, "Efficient Design of Self-Checking Checker for Any m-out-of-n Code," *IEEE Trans. on Computers*, Vol. C-27, pp.482-490, June 1978.

[14] C. Halatsis, N. Gaitanis, and M. Sigala, "Fast and Efficient Totally Self-Checking Checkers for m-out-of-(2m±1) Codes," *IEEE Trans. on Computers*, Vol. C-32, pp.507-511, May 1983.

[15] 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, pp.263-269, March 1973.

[16] S. M. Reddy, "A Note on Self-Checking Checkers," *IEEE Trans. on Computers*, Vol. C-23, pp.1100-1102, October 1974.

[17] T. Nanya and Y. Tohma, "A 3-Level Realization of Totally Self-Checking Checkers for m-out-of-n Codes," *Proc. International Symp. on Fault-Tolerant Computing*, pp.173-176, 1983.

[18] M. A. Marouf and A. D. Friedman, "Efficient Design of Self-Checking Checker for Any m-out-of-n Code," *IEEE Trans. on Computers*, Vol. C-27, pp.482-490, June 1978.

[19] S. M. Reddy and J. R. Wilson, "Easily Testable Cellular Realizations for the (exactly p)-out-of-n and (p or more)-out-of-n Logic Functions," *IEEE Trans. on Computers*, Vol. C-23, pp.98-100, January 1974.

[20] S. J. Piestrak, "Design of Self-Testing Checkers for m-out-of-n Codes," *Proc. International Symp. on Fault-Tolerant Computing*, pp.173-176, June 1983.

[21] J. Khakbaz, "Totally Self-Checking Checker for 1-out-of-n Code Using Two-Rail Codes," *IEEE Trans. on Computers*, Vol. C-31, pp.677-681, July 1982.

[22] G. Laskaris, T. Haniotakis, A. Paschalis, and D. Nikolos, "New Design Method for Low-Cost TSC Checkers for 1-out-of-n and (n-1)-out-of-n Codes in MOS Implementation," *International Journal of Electonics*, Vol. 69, pp.805-817, 1990.

[23] M. Kotocova, "Design of Totally Self-Testing Check Circuits for Some 1/n Codes," *Proc. International Conference on Fault-Tolerant System Diagnostics*, pp.241-245, September 1981.

[24] G. Laskaris, T. Haniotakis, A. Paschalis, and D. Nikolos, "Efficient Design of TSC Checker for 1-out-of-n Codes in MOS Transistor Implementation," *Proc. International Conference on Fault-Tolerant System Diagnostics*, pp.805-817, September 1989.

[25] S. J. Piestrak, "Design of Self-Testing Checkers for 1-out-of-n Codes," *Proc. International Conference on Fault-Tolerant System Diagnostics*, pp.57-63, September 1983.

[26] H. Dong, "Modified Berger Codes for Detection of Unidirectional Errors," *Proc. International Symp. on Fault-Tolerant Computing*, pp.317-320, June 1982.

[27] N. K. Jha, "Separable Codes for Detecting Unidirectional Errors," *IEEE Trans. on Computer-Aided-Design*, Vol. 8, pp.571-574, May 1989.

[28] M. J. Ashjaee and S. M. Reddy, "On Totally Self-Checking Checkers for Separable Codes," *IEEE Trans. on Computers*, Vol. C-26, pp.737-744, August 1977.

[29] M. A. Marouf and A. D. Friedman, "Design of Self-Checking Checkers for Berger Codes," *Proc. International Symp. on Fault-Tolerant Computing*, pp.179-184, June 1978.

[30] S. J. Piestrak, "Design of Fast Self-Testing Checkers for a Class of Berger Codes," *IEEE Trans. on Computers*, Vol. C-36, pp.629-634, May 1987.

[31] J. C. Lo and S. Thanawastien, "The Design of Fast Totally Self-Checking Berger Code Checkers Based on Berger Code Partitioning," *Proc. International Symp. on Fault-Tolerant Computing*, pp.226-231, 1988.

[32] M. J. Ashjaee and S. M. Reddy, "Totally Self-Checking Checkers for a Class of Separable Codes," *Proc. Allerton Conference Circuits System Theory*, pp.238-242, 1974.

[33] N. K. Jha, "A Totally Self-Checking Checker for Borden's Code," *IEEE Trans. on Computer-Aided-Design*, Vol. 8, pp.731-736, July 1989.

[34] B. Bose and D. J. Lin, "Systematic Unidirectional Error-Detecting Codes," *IEEE Trans. on Computers*, Vol. C-34, pp.1026-1032, November 1985.

[35] B. Bose, "Burst Unidirectional Error Detecting Codes," *IEEE Trans. on Computers*, Vol. C-35, pp.350-353, April 1986.

[36] M. Blaum, "On Systematic Burst Unidirectional Error Detecting Codes," *IEEE Trans. on Computers*, Vol. C-37, pp.453-457, April 1988.

[37] Y. Tohma, "Coding Techniques in Fault-Tolerant, Self-Checking and Fail-Safe Circuits," in *Fault-tolerant Computing: Theory and Techniques*, edited by D. K. Pradhan, Prentice-Hall, 1986.

[38] D. K. Prahan,"A New Class of Error Correcting/Detecting Codes for Fault Tolerant Applications," *IEEE Trans. on Computers*, Vol. C-29, pp.471-481, June 1980.

[39] G. P. Mak, J. A. Abraham, and E. S. Davidson, "The Design of PLAs with Concurrent Error Detection," *Proc. International Symp. on Fault-Tolerant Computing*, pp.303-310, June 1982.

[40] S. L. Wang and A. Avizienis, "The Design of Totally Self-Checking Circuits Using Programmable Logic Arrays," *Proc. International Symp. on Fault-Tolerant Computing*, pp.173-180, June 1979.

[41] J. E. Smith and G. Metze, "The Design of Totally Self-Checking Combinational Circuits," *Proc. International Symp. on Fault-Tolerant Computing*, pp.130-134, June 1977.

[42] J. F. Wakerly, "Partially Self-Checking Circuits and Their Use in Performing Logical Operations," *IEEE Trans. on Computers*, Vol. C-23, pp.658-666, July 1974.

[43] J. E. Smith and G. Metze, "Strongly Fault Secure Logic Networks," *IEEE Trans. on Computers*, Vol. C-27, pp.491-499, June 1978.

[44] M. Diaz, "Design of Totally Self-Checking Asynchronous Sequential Machines," *Proc. International Symp. on Fault-Tolerant Computing*, pp.3-9 to 3-24, June 1974.

[45] F. Ozguner, "Design of Totally Self-Checking Asynchronous Sequential Machines," *Proc. International Symp. on Fault-Tolerant Computing*, pp.124-129, June 1977.

[46] J. Viaud and R. David, "Sequential Self-Checking Circuits," *Proc. International Symp. on Fault-Tolerant Computing*, pp.263-268, June 1980.

[47] S. Devadas, H. K. T. Ma, A. R. Newton and, A. L. Sangiovanni-Vincentelli, "Synthesis and Optimization Procedure for Fully and Easily Testable Sequential Machines," *Proc. International Test Conference*, pp.621-630, September, 1988.

[48] H. K. T. Ma, S. Devada, A. R. Newton, and A. L. Sangiovanni-Vincentelli, "Test Generation for Sequential Finite State Machines," *International Conference on Computer-Aided-Design*, pp.288-291, November 1987.

[49] J. E. Smith and P. Lam, "A Theory of Totally Self-Checking System Design," *IEEE Trans. on Computers*, Vol. C-32, pp.831-843, September 1983.

[50] M. Diaz, P. Azema, and J. M. Ayache, "Unified Design of Self-Checking and Fail-Safe Combinational Circuits and Sequential Machine," *IEEE Trans. on Computers*, Vol. C-28, pp.276-281, March 1979.

[51] Y. Mukai and Y. Tohma, "A Method for the Realization of Fail-Safe Asynchronous Sequential Circuits," *IEEE Trans. on Computers*, Vol. C-23, pp.736-739, July 1974.

[52] D. K. Pradhan, "Asynchronous State Assignments with Unateness Properties and Fault-Secure Design," *IEEE Trans. on Computers*, Vol. C-27, pp.396-404, May 1978.

[53] G. Mago, "Monotone Function in Sequential Circuits," *IEEE Trans. on Computers* Vol. C-22, pp.928-933, October 1973.

[54] N. K. Jha and S. J. Wang, "Design and Synthesis of Self-checking VLSI Circuits and Systems," *1991 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pp.578-581, October 1991.

[55] A. L. Hopkins and T. B. Smith, "The Architectural Elements of a Symmetric Fault-Tolerant Multiprocessor," *Proc. International Symp. on Fault-Tolerant Computing*, pp.42-46, June 1974.

[56] T. Nanya, "Error Secure Propagating Concept and its Application to the Design of Strongly Fault-Secure Processor," *IEEE Trans. on Computers*, Vol. C-37, pp.14-24, January 1988.

[57] E. J. McCluskey, "Design Techniques for Testable Embedded Error Checkers," *Computer*, pp.84-88, July 1990.

[58] H. B. Bakoglu, *Circuits, Interconnections, and Packing for VLSI*, Addison-Wesley Publishing Co., Reading, Mass., 1990.

[59] W. K. Fuchs and J. A. Abraham, "A Unified Approach to Concurrent Error Detection in Highly Structured Logic Arrays," *Proc. International Symp. on Fault-Tolerant Computing*, pp.4-9, June 1984.

[60] W. Baker, "Oct. Tools Distribution,3.0 Vol. 6: Light/Oct/Vemlisp Implementation," Electronics Research Laboratory, University of California at Berkeley, 1989.

[61] W. C. Carter and P. R. Schneider, "Design of Dynamically Checked Computers," *Proc. International Federation for Information Processing*, Vol. 2, pp.878-883, August 1968.

[62] S. Devadas, H. K. Ma, and A. R. Newton, "MUSTANG: Targeting Multilevel Logic Implementations," *IEEE Trans. on Computer-Aided-Design*, Vol. 7, pp.1290-1300, December 1988.

[63] R. Spickelmier, "Oct. Tools Distribution 2.1," Electronics Reserach Laboratory, University of California at Berkeley, 1988.

[64] N. Weste and K. Eshraghian, *Principle of CMOS VLSI Design*, Addison-Wesley Publishing Co., Reading, Mass., 1985.

[65] G. Chartrand and L. Lesniak, *Graphs & Diagraphs*, Second Edition, Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, Ca., 1986.

[66] C. Sechen and A. Sangiovanni-Vincentelli, "The Timberwolf Placement and Routing Package," *IEEE Journal of Solid-State Circuits*, Vol. C-20, pp.510-522, April 1985.

[67] H. Li and W. Li, "AISCE: A Layout Synthesis System for ASIC Design," *Proc. International Conference on Circuits and Systems*, pp.419-422, June 1991.