UNIVERSITY OF CINCINNATI

Date: 5/25/2004

I, Mr. Swaroop Ghosh,

hereby submit this work as part of the requirements for the degree of:

Master of Science

in:

Computer Engineering

It is entitled:

"Scan Chain Fault Identification Using Weight-Based Codes for SoC Circuits"

This work and its defense approved by:

Chair: Dr. Wen-Ben Jone

Dr. Ranga Vemuri

Dr. Karen Tomko

SCAN CHAIN FAULT IDENTIFICATION USING WEIGHT-BASED CODES FOR SOC CIRCUITS

A thesis submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the Department of
Electrical and Computer Engineering and Computer Science
of the College of Engineering
May 2004

by

Swaroop Ghosh

B.E.(E.E.), University of Roorkee, Roorkee, India, May 2000

Thesis Advisor and Committee Chair: Dr. Wen Ben Jone

Dedication

To my dearest parents and brothers, Dr. Sanjay Ghosh and Manoj Kumar Ghosh

Abstract

Recently, it has been observed that embedded cores in a high-speed SoC circuit have the problem of *broken scan chains* that cannot shift properly. Also, scan chain intermittent faults caused by hold-time violations and crosstalk noises are pervasive. In this research, an efficient method is proposed to identify the faulty scan chain(s) at the core level. That is, the core where the scan chain is defective can be identified, even if the scan chain is broken. The result can be used to tune up the fabrication process or to guide the fine-grained scan cell identification process. Here, *weight-based* m-out-of-n codes, which can generate a large number of codewords, with small hardware overhead and high fault detection capability are used to generate the scan chain diagnostic patterns for permanent and intermittent faults. An efficient codeword generation method is proposed to maximize the number of codewords, minimize the aliasing probabilities and test application cost. Aliasing probabilities are analyzed thoroughly for code sequences that are shifted to the scan chain in an overlapped manner. The idea of *multiple* m-out-of-n codes is also proposed to guarantee that sufficient number of codewords are generated to perturb the scan chains and the associated combinational circuits. A test pattern broadcast architecture is incorporated to support the proposed testing method. The complexity of control signal generation and scan chain test is greatly reduced by adopting this architecture. Further, by adding a test pattern alignment scheme, it is possible to share the control signals for all checkers and to support the broadcast architecture as well. Therefore, the hardware overhead of the test architecture is also reduced. Finally, a totally self-checking checker is designed to support the application of multiple weight-based codes. Simulation results demonstrate the feasibility of the proposed method as the aliasing probability is zero in almost every code sequence.

Acknowledgements

I wish to express my sincere gratitude to my advisor, Dr. Wen-Ben Jone, for the guidance, support and constructive inputs that he provided throughout this work. He was always ready to provide his guidance and help in solving problems. This work would never have taken shape without his encouragement and expertise. He gave me complete independence to think on the research problem. However, he corrected me whenever my thinking went in the wrong direction. He gave me the opportunity to participate in various research projects which helped me understand the issues related to VLSI design and testing. Apart from guiding my thesis, he has directly and indirectly imparted many qualities in me which would certainly help me in pursuing a research oriented career. Under his guidance, I have realized that I can really make some useful contributions to this field.

I would like to thank the members of my thesis committee, Dr. Ranga Vemuri and Dr. Karen Tomko, for spending their valuable time reviewing this work, raising important issues and suggesting future research work. Special thanks are also due to my close friends, Srivatsa, Sriram, Gaurav, Kishore, Xiong, Lu Fei, Dilip, Aditi, and others, for the wonderful time I have had in their company. I would always cherish those moments. I would also like to thank my senior Vikas Vijay, who helped me during the course of my experiments. My sincere thanks to Srivatsa Kalala who guided me through the toughest period of my life and always helped me make the right decisions. Last, but not the least, I would like to express my gratitude to my parents and my brothers, Dr. Sanjay Ghosh and Manoj Kumar Ghosh, who have helped mould me into what I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advent of deep sub-micron technologies, a large number of transistors can be integrated to a single chip. Hence, system-on-chip (SoC) design is becoming a more popular and cost-effective solution than traditional PCB designs. A modern SoC may contain memories, DSP cores, baseband controllers, analog components and peripherals, which are intellectual properties (IP's) bought from different vendors. A simple SoC is shown in Fig. 1.1 where UDLs are user-defined logic blocks, DSP is a digital signal processing core, and USB is a universal serial bus core [1]. These complex core blocks facilitate design reuse which results in shorter design cycles. Testing SoC circuits through scan chains is gaining popularity because scan testing gives core users flexibility to test the internal logic of each individual core. Since a core might be deeply embedded in the SoC circuit, a test access structure called *test access mechanism* (TAM) is generally designed to wrap the core [2]. The TAM circuit is used to transport test patterns from a pattern source to the scan chains of the core. Similarly, the test responses stored in the scan chains of the core are transported to a response monitor by the TAM circuit. It has been reported in [3] that the logic circuitry associated with the scan path may occupy as much as 30% of the total area of the circuit. Hence, it is of utmost importance to ensure the proper functioning of the scan chain before proceeding to test the associated combinational logic circuits. An internal fault in the combinational circuit can get masked due to a faulty scan chain. It is also possible that desired patterns to be applied to the combinational logic can never be shifted in.

Basically, the faults that might occur at a scan chain can be categorized as *permanent* faults and *intermittent* faults. Scan chain fault diagnosis is the process of identifying the defective scan cell in a scan chain. Fault diagnosis for scan chain permanent faults assumes that the fault occurring in the scan chain is permanent, e.g., stuck-at fault, transition fault and hold-time fault. Fault diagnosis for scan chain permanent faults can be divided into *two* classes: (1) The scan chain is added extra circuits through special scan cell design or additional circuitry [4] [5] [6] [7], and (2) The scan chain is diagnosed by sequential test pattern generation or random test pattern simulation without adding any extra circuitry [3] [8] [9] [10]. In contrast to permanent fault diagnosis, intermittent fault diagnosis for scan chains are investigated only recently [11] [12]. Unlike permanent faults that can be modeled deterministically, intermittent faults occur stochastically and this makes fault diagnosis extremely difficult. According to [12], intermittent scan chain *hold-time* faults are pervasive in deep sub-micron technologies due to following reasons. *First*, in deep sub-micron technologies, it is very difficult to estimate the wire delay between the adjacent scan cells due to fabrication process variations. This might make the *hold-time margin* estimated too small. *Second*, noisy signals (at the wire between a pair of scan cells) caused by crosstalk

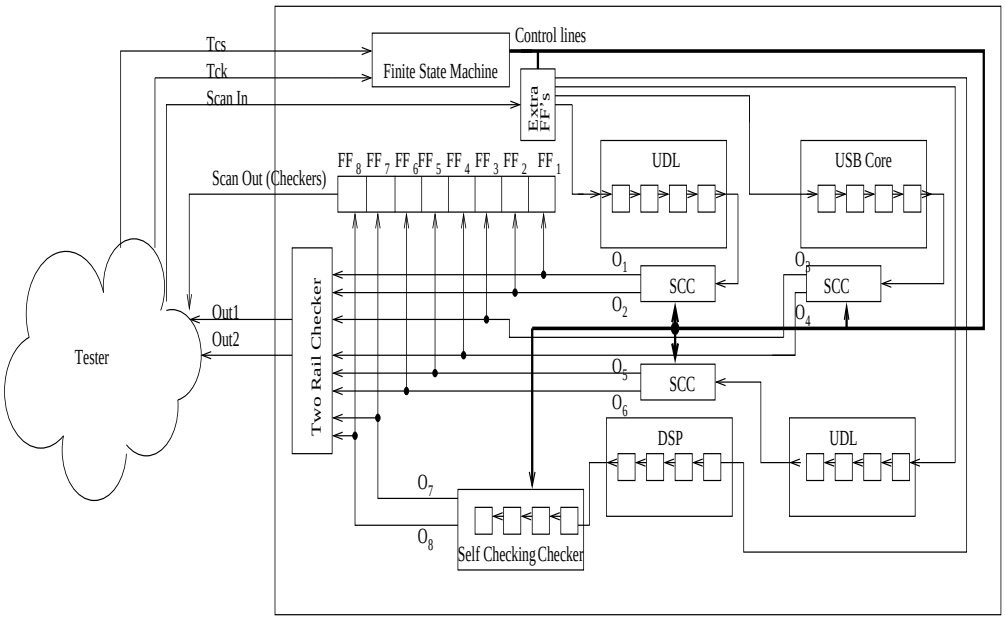


Figure 1.1: A SoC design with self-checking checkers.

or IR-drop might cause hold time fault in the scan cell especially when the hold-time margin is small. In summary, shrinking the feature size, increasing the operating speed, reducing the supply voltage etc. are the root causes of intermittent faults. In today's technologies, wire delays are becoming prominent as they are larger than gate delays. The problem with wire delay is that it is unpredictable, as it also depends on the signal running through it and through its adjacent wires. Even worse, inductive coupling between adjacent wires is taking over capacitive coupling, and its effect is hard to predict. The reason comes from the fact that the inductive coupling effect depends on the return current path, which in turn depends on the circuit topology as well as on the signal transition types (rising or falling) in victim and aggressor wires. The inductive coupling effect gives rise to overshoot/undershoot, oscillation, ringing and other transmission line effects that are hard to predict [13].

The challenging task is to activate and identify intermittent faults in the scan chain. This is possible only if proper test patterns, which can create maximum noises in the circuit, are fed to the scan chain at chip speed. Traditional permanent fault models and deterministic test methods cannot capture the uncertainties involved in the deep sub-micron scenario [12]. This work considers the fault diagnosis problem of scan chains for permanent and intermittent faults at the *core level*, instead of the scan cell level. That is, we try to identify the core whose scan chain(s) is faulty, even if the scan chain is broken and cannot shift properly. The basic idea is to add a self-checking checker to the tail of each scan chain in every core as shown in Fig. 1.1. The purpose of scan chain fault diagnosis at the core level are two-fold. First, identification of the defective scan chain in a specific core gives the manufacturer hints to tune-up the fabrication process. Since the cores of a SoC are purchased from vendors, the designer cannot modify the circuit layout. However, the manufacturer can fix some fabrication parameters such that the fabrication errors can be corrected. Second, identification of the defective scan chain in a specific core enables the test engineer to more efficiently identify the faulty scan cell. This can be achieved by restructuring the core wrappers such that scan chains of some cores can be bypassed. Thus, fine-grained scan cell identification

techniques such as [11] [12] can be applied with shorter scan chains (thus shorter diagnosis time). Traditional 01010...10 or 00110011...0011 patterns are not suitable for activating the intermittent faults as the number of valid codewords for such codes are very small. For example, the 0101.. test pattern has only two valid codewords, namely 01 and 10. Similarly, 0011... has only four valid codewords, namely 0011, 0110, 1100, 1001. The small number of codewords cannot activate the intermittent faults due to a limited number of rising/falling transitions generated. In this work, *weight-based* m-out-of-n codes, which can generate a large number of codewords, with small hardware overhead and high fault detection capability are used to generate the scan chain diagnostic patterns for permanent and intermittent faults. A codeword generation method is proposed with the objectives of (1) maximizing the number of codewords, (2) minimizing the aliasing probabilities of *bidirectional* and *multi-directional* errors caused by permanent or intermittent defects, and (3) minimize the diagnostic pattern application time by codeword overlapping. The idea of *multiple* m-out-of-n codes is also proposed to guarantee that sufficient number of codewords are generated to perturb the scan chains and the associated combinational circuits. To further reduce the hardware overhead, a broadcast test architecture is adopted to enable all checkers to share the same control signals. Simulation results demonstrate the feasibility of the proposed method which achieves the above three objectives successfully.

This thesis is organized as follows:

Chapter 2 reviews the basic concepts of weight-based m-out-of-n codes, self-checking checkers, and their applications to on-line testing such as bus crosstalk faults and power supply noises.

Chapter 3 deals with the codeword generation method.

Chapter 4 discusses the order of codeword application to the scan chains using a graph model.

Chapter 5 analyzes the aliasing probabilities of both bidirectional and multi-directional errors.

Chapter 6 discusses the merging of multiple m-out-of-n codes and the corresponding totally self-checking checker design.

Chapter 7 is dedicated to the test architecture that can support the proposed testing method. The test pattern alignment problem is also dealt with so that shared control lines can be used for all checkers and the broadcast test architecture can be incorporated.

Chapter 8 concludes the thesis with suggested future work on a more powerful test pattern alignment for broadcast test architecture, identification of fine-grained faulty scan cell, development of a built-in self-test methodology to exercise an at-speed test of the scan chain, and development of a simulator that can evaluate the power of codewords applied.

Chapter 2

Background

2.1 Weight-Based Codes

The weight-based m-out-of-n coding method is a generalization of Berger codes [14]. Weight-based codes are obtained by assigning different weights w_i to the information bits. All codewords which belong to a weighted m-out-of-n code satisfies:

$m = \sum_{i=1}^{l} w_i v_i$, and
$n = \sum_{i=1}^{l} w_i$

where $w_i \in W$ is the weight of i^{th} bit, v_i is the value of i^{th} bit, W is the the set of weights, and l is the *cone size* (i.e., the number of bits in each codeword). It should be noted that m may be larger than the n value of the ordinary m-out-of-n code, where the weight of each bit is one.

The weight-based m-out-of-n coding still possesses the property of detecting all unidirectional errors. Additionally, it can detect all bidirectional errors involving a pair of bits i and j if $w_i \neq w_j$. In fact, in the case of a bidirectional error involving such bits, we have $m_{new} = m \pm(w_i - w_j)$, so that a checker can detect the error. If all w_i values differ from each other, it is evident that all single bidirectional errors can be detected. This constraint can be achieved if $w_i = i$, but such a large weight set will increase the hardware overhead of the checker circuit. As will be shown by the analysis presented in the next section, a small set of weights can be used without deteriorating the error detection ability in our case. Moreover, the value of each w_i should be kept small to reduce the checker design complexity.

2.2 Self-Checking Checker

A self-checking checker basically verifies the correctness of the incoming code. It detects an error as soon as it occurs, and prevents the error propagation throughout the system [14]. For such a system, totally self-checking (TSC) is the main property to ensure reliable operations in the presence of faults. A TSC checker must have two outputs and, hence, four combinations (i.e., 00, 01, 10, 11). Two of these output combinations, (01,10), are considered *valid*. A *nonvalid* checker output, (00,11), indicates either a noncodeword at the input of the checker or a fault in the checker itself. The reason why a checker needs two outputs is: if there is only one output and the normal output value is 1 (0), then a stuck-at-1 (0) fault at the checker output cannot be detected during normal operation. Also, combinations (00,11) are not selected as valid outputs because a unidirectional multi-bit error may change 00 to 11 and vice versa (and cannot be detected) [15]. However, single-output TSC checkers have been proposed in [16], [17] and [18]. They generally

work with the help of a clock signal. They produce 10 or 01 in one clock period during fault-free conditions, and 00 or 11 during faulty conditions. The main advantage of a single-output checker over its counterpart is low power consumption and small hardware overhead. In the following, we describe the properties that are used to define TSC circuits. Fig. 2.1 illustrates the properties of the following definitions.

Fault-secure: A circuit is *fault-secure* with respect to a set of faults F, if for any single fault in F, the circuit never produces an incorrect output codeword for any input codeword. That is, the circuit produces the correct codeword or a non-codeword for any valid input codeword.

Self-testing: A circuit is *self-testing* with respect to a set of faults F, if for every single fault in F, the circuit produces a noncodeword at the output for atleast one input codeword.

Code-disjoint: A circuit is *code-disjoint* if it maps codewords (noncodewords) at the input to codewords (noncodewords) at the output.

Totally self-checking: A circuit is *totally self-checking* if it is both fault secure and self-testing. So, a circuit is TSC if all single faults are detectable by at least one codeword, and when a given input does not detect the fault, the output is the correct codeword (i.e., 01 or 10).

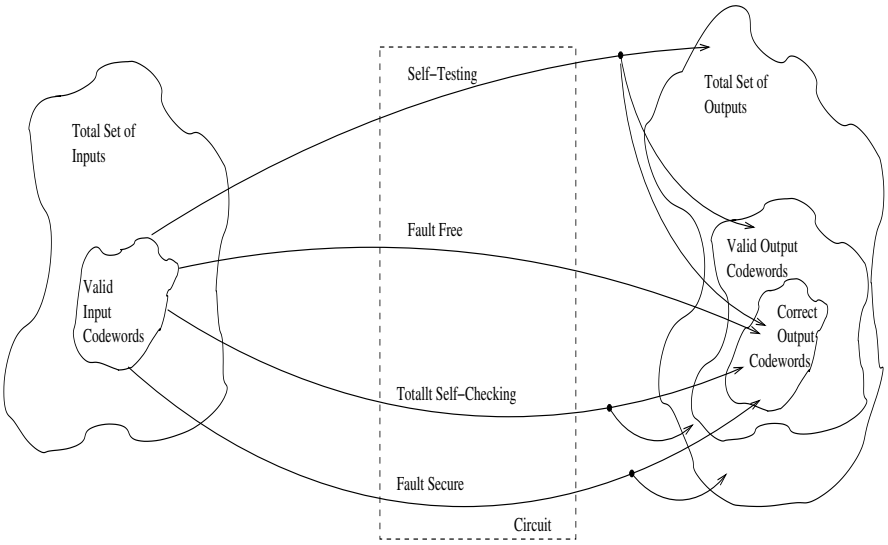


Figure 2.1: Illustration of the concepts of self-testing, fault-secure, and totally self-checking [19].

2.3 Applications of Weight-Based Codes

Weight-based codes are proposed in [20] to perform concurrent error detection of multilevel circuits. Each output bit of the multilevel combinational circuit is assigned a weight, and the check bits represent the sum of weights of the output bits which have logic value 1. Aliasing can be minimized by guidelines such as: (1) The weights should be mutually prime, and the sum of two weights should not be a multiple of the third; (2) Output bits are partitioned into clusters such that there is minimum sharing of logic between outputs in the same cluster, and outputs in the same cluster are assigned the same weight [20]. The aliasing probability is reported to be smaller than traditional error detection codes. An efficient design for the corresponding totally self-checking checker is also proposed. The idea of weight-based codes are extended to *modulo* weight-based

codes in [21] for concurrent error detection of arbitrary multilevel circuits. The major purpose of the modulo weight-based codes is to reduce the check bits required by taking the sum modulo of the weights for the output bits that have logic value 1. The algorithm to assign weights and the corresponding self-checking checker design are also presented. In [14], weight-based codes are applied to detect crosstalk faults by concurrent error detection of buses. Based on the knowledge of the layout structure, a graph can be generated to represent the most likely crosstalk faults in the bus. The problem of weight selection is then modeled as the well-known graph-coloring problem. By the weights selected, a low-cost weight-based m-out-of-n code can be found to detect the crosstalks that might cause bidirectional errors. Other self-checking detection and diagnosis schemes for different faults (e.g., transient, delay faults) and devices (e.g., buses, sequential circuits) can be found in [22] [23] [24].

Chapter 3

Codeword Generation for Scan Chain Testing

In this chapter, we will present the codeword generation methodology for weight-based m-out-of-n codes for scan chain testing.

3.1 Codeword Generation

Before proceeding with the codeword generation method, we present few common terms that will be frequently used in this work.

• **Weight set** is a set of weights by which a weighted m-out-of-n code is generated.

• **Number of codewords** denotes the total number of codewords found for a given weighted m-out-of-n coding method.

• **Cone size** (i.e., l) is the number of bits in each codeword of a given weighted m-out-of-n code.

• **Weight configuration** represents a way of (repeatedly) selecting l weights from the weight set without permutation such that the required n can be satisfied. For example, if the weight set is {1,2,3} and $n = 10$, then we have two weight configurations: [1,2,2,2,3] and [1,1,2,3,3]. However, we have only one weight configuration for weight set {1,2} and $n = 6$, namely, [1,1,1,1,2].

• **Weight sequence or distribution** denotes the weight assigned to each bit of a codeword such that required n is satisfied. Consider an example with weight set = {1,2} and cone size = 5. One possible weight distribution to get $n = 6$ is 1-2-1-1-1. Here, we can verify that $n = \sum w_i$ = 1+2+1+1+1 = 6 based on the weight configuration [1,1,1,1,2]. Note that weight distribution is different from weight configuration. A new weight distribution can be obtained simply by permuting the weights of the current weight distribution, but the similar logic is not valid for weight configuration.

• **Code sequence** denotes the order by which codewords are shifted into the scan chain. Assume we have cone size = 5, weight set = {1,2}, $n = 6$, and weight sequence = 1-1-1-2-1. A valid code sequence for $m = 2$ can be 11000 → 10001 → 00010 → 00101 → 10100 → 01001 → 01100. Note that each of the codewords has $m = \sum w_i * v_i$ =2. Generally, the codewords are ordered in such a way that there is maximum overlapping between each pair of adjacent codewords, hence all codewords can be shifted into the scan chain using the minimum number of clock cycles.

• **Shift sequence** is the number of shifts required to reach one codeword from another. Consider the codewords of the above example again. Code sequence 11000→ 10001 → 00010 → 00101 → 10100 → 01001 →01100 requires the shift sequence of 1→ 1 → 1 → 2 → 1 → 3 →1. The following illustrates how the next codeword can be obtained by shifting new bits into the current codeword. Note that the shifting operation is performed in the right-to-left fashion, while the left-to-right shifting can be discussed similarly.

(11000)-1→ (10001) − 0 → (00010) − 1 → (00101) − 00 → (10100) − 1 → (01001) − 100 → (01100) − 0 → back to first codeword.

• **Transition count** is the total number of 0 → 1 or 1 → 0 transitions generated by a particular m-out-of-n code sequence. Consider the example of 2-out-of-6 code again, the transition count for code sequence 11000→ 10001 → 00010 → 00101 → 10100 → 01001 → 01100 →11000 is 30. Note that, there is only one (left) shift between codeword 11000 & 10001 and that there are one 0→1 (rightmost bit) and one 1→0 (fourth bit from right) transitions generated when 10001 is applied after 11000. But, there are two shifts between codeword 00101 & 10100, and there are three words involved, namely, 00101, 01010 and 10100. The number of transitions generated between 00101 & 01010 is 4 and the number of transitions generated between 01010 & 10100 is again 4 giving rise to a total of 8 transitions between 00101 & 10100. The number of transitions between other codewords can be calculated similarly. The total *transition count* is the sum of transition counts generated by each pair of adjacent codewords.

• **Test application cost** denotes the total number of shifts required to apply all the codewords of a m-out-of-n code to the scan chain. In the above example for shift sequence, the test application cost is 1+1+1+2+1+3+1 = 10.

The function of the *codeword enumerator* (in software) is to select proper m, n, weight set and weight distribution, and to generate the corresponding codewords. A program is developed in C++ for finding the codespace, cost and aliasing for a specific weighted m-out-of-n coding scheme. The experimental steps are as follows:

1. Fix the weight set, e.g., {1,2} and cone size.
2. Fix the n value.
3. Find a weight configuration and its weight distributions. For example, given $n = 6$ and cone size = 5, there in only one weight configuration [1,1,1,1,2] and the weight distributions are 1-1-1-1-2, 1-1-1-2-1, 1-1-2-1-1, 1-2-1-1-1 and 2-1-1-1-1. Another example is to have weight set = {1,2,3}, cone size = 5, and $n = 10$. The possible weight configurations are [1,1,2,3,3] and [1,2,2,2,3].
4. Fix a m value.
5. For the m value, find the codewords for a specific weight distribution. Following the above example, for $m = 2$ and weight sequence 1-1-1-2-1, the codewords are 11000, 10001, 00010, 00101, 10100, 01001, 01100.
6. Arrange the codewords such that the test application cost is minimum. The aim is to apply all codewords into the scan chain by minimum number of (left) shifts. In our experiments, exhaustive search is exercised to find the arrangement corresponding to the minimum test

application cost. For example, code sequence 11000 → 10001 → 00010 → 00101 → 10100 → 01001 → 01100 gives the minimum test application cost for the above example (step 4). For a weighted m-out-of-n code with N codewords, an algorithm to accomplish the codeword ordering with complexity $O(l \cdot N^2)$ will be presented in Chapter 4. Note that l (i.e., cone size) is generally a small number compared to N.

7. Find the bidirectional and multi-directional aliasing probabilities for the weighted m-out-of-n code.

8. Rearrange the weight sequence and repeat steps 5 and 7 until all weight sequences are tried. For example, a new experiment with weight sequence 1-1-2-1-1 can be repeated.

9. Find a new m value and repeat steps 5 to 8 until all m values are tried.

10. Repeat steps 3 to 9 until all weight configurations are exhausted.

11. Among all weighted m-out-of-n codes generated, select the one which yields the maximum number of codewords, the minimum cost and aliasing probabilities. If there exists none which satisfies all requirements, try to trade-off among number of codewords, test application cost and test aliasing probabilities.

12. If necessary, try more n values and weight sets and repeat steps 3 to 11 until a satisfactory weighted m-out-of-n code is found.

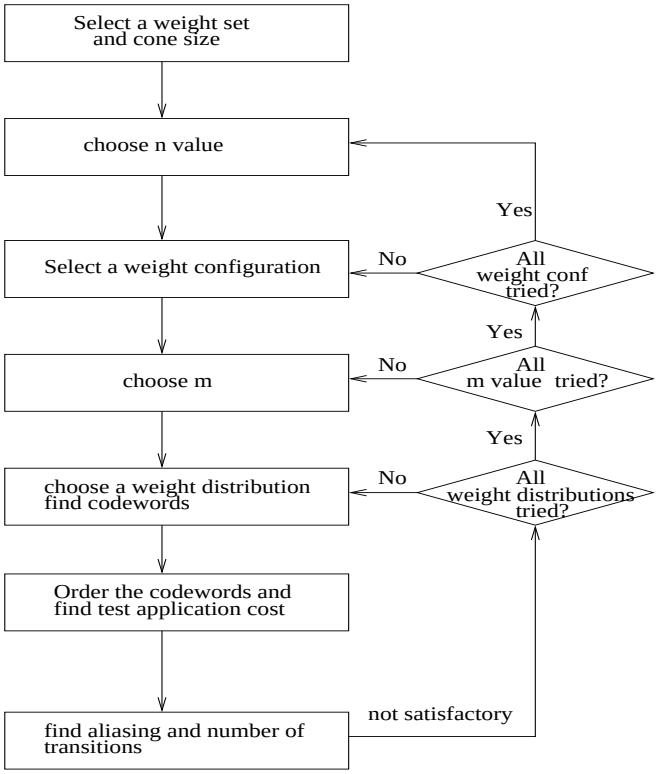


Figure 3.1: Flow for codeword enumeration.

The entire flow for the selection of m, n and weight set is depicted in Fig. 3.1. We have performed software simulation with cone size of 5 and different weight sets. The simulation results

for different cardinalities of weight sets, m and n values are presented in Tables 3.1, 3.2, 3.3 and 3.4.

Table 3.1: Results for weight set {1,2}.

m/n	# code-words	cost	Weight sequence	Code sequence	Shift sequence
1/6	4	5	1-1-1-1-2	10000 → 00010 → 00100 → 01000 →	2 → 1 → 1 → 1 →
2/6	7	10	1-1-1-2-1	11000 → 10001 → 00010 → 00101 → 10100 → 01001 → 01100 →	1 → 1 → 1 → 2 → 1 → 3 → 1 →
3/6	8	12	1-1-2-1-1	10100 → 00101 → 01011 → 01100 → 11001 → 10011 → 00110 → 11010 →	3 → 1 → 2 → 1 → 1 → 1 → 2 → 1 →
1/7	3	4	2-1-1-1-2	01000 → 00010 → 00100 →	2 → 1 → 1 →
2/7	5	6	1-2-1-2-1	01000 → 10001 → 00010 → 00101 → 10100 →	1 → 1 → 1 → 2 → 1 →
3/7	7	10	1-1-2-2-1	10100 → 00011 → 01100 → 11001 → 10010 → 00101 → 01010 →	3 → 2 → 1 → 1 → 1 → 1 → 1 →
1/8	2	4	2-1-1-2-2	01000 → 00100 →	3 → 1 →
2/8	4	4	1-2-2-2-1	01000 → 10001 → 00010 → 00100 →	1 → 1 → 1 → 1
3/8	6	10	2-1-1-2-2	11000 → 00101 → 01010 → 10100 → 01001 → 00110 →	3 → 1 → 1 → 1 → 2 → 2 →
4/8	6	10	2-1-2-1-2	10100 → 10001 → 00101 → 01011 → 01110 → 11010 →	2 → 2 → 1 → 2 → 2 → 1 →
1/9	1	3	2-2-1-2-2	00100 →	3 →
2/9	4	5	1-2-2-2-2	01000 → 00001 → 00010 → 00100 →	2 → 1 → 1 → 1 →
3/9	4	9	2-2-1-2-2	10100 → 00110 → 01100 → 00101 →	3 → 1 → 3 → 2 →
4/9	6	10	2-1-2-2-2	10100 → 10001 → 00011 → 00110 → 10010 → 00101 →	2 → 1 → 1 → 3 → 1 → 2 →

3.2 Analysis for the Number of Codewords

Let the weight set be $\{w_1, \ldots, w_r\}$, where r = cardinality of the weight set. Given a m-out-of-n code with a specific n value, we have following equation:

$$n = x_k^1 w_1 + x_k^2 w_2 + \cdots + x_k^r w_r \tag{3.1}$$

where x_k^i = integer coefficients for weight w_i. This also means that there are x_k^i number of w_i's in the weight distribution, and $k = 1$ to s where s represents the total number of possible configurations that satisfy the given n value. Similarly, we can have

$$m = y_{k,j}^1 w_1 + y_{k,j}^2 w_2 + \cdots + y_{k,j}^r w_r \tag{3.2}$$

Table 3.2: Results for weight set {2,3}.

m/n	# code-words	cost	Weight sequence	Code sequence	Shift sequence
2/11	4	5	2-2-2-2-3	10000 → 00010 → 00100 → 01000 →	2 → 1 → 1 → 1 →
3/11	1	3	2-2-3-2-2	00100 →	3 →
4/11	6	10	2-2-2-3-2	11000 → 10001 → 00101 → 10100 → 01001 → 01100 →	1 → 2 → 2 → 1 → 3 → 1 →
5/11	4	9	2-2-3-2-2	10100 → 00110 → 01100 → 00101 →	3 → 1 → 3 → 2 →
2/12	3	4	3-2-2-2-3	01000 → 00010 → 00100 →	2 → 1 → 1 →
3/12	2	4	2-2-3-3-2	00100 → 00010 →	3 → 1 →
4/12	3	6	2-3-2-3-2	10100 → 10001 → 00101 →	2 → 2 → 2 →
5/12	6	10	2-2-3-3-2	10100 → 00011 → 01100 → 10010 → 00101 → 01010 →	3 → 2 → 2 → 1 → 1 → 1 →
6/12	2	2	2-3-2-3-2	01010 → 10101 →	1 → 1 →
2/13	2	4	3-2-2-3-3	01000 → 00100 →	3 → 1 →
3/13	3	4	2-3-3-3-2	01000 → 00010 → 00100 →	2 → 1 → 1 →
4/13	1	2	3-2-3-2-3	01010 →	2 →
5/13	6	10	3-2-2-3-3	11000 → 00101 → 01010 → 10100 → 01001 → 00110 →	3 → 1 → 1 → 1 → 2 → 2 →
6/13	3	6	3-2-3-2-3	10100 → 10001 → 00101 →	2 → 2 → 2 →
2/14	1	3	3-3-2-3-3	00100 →	3 →
3/14	4	5	2-3-3-3-3	01000 → 00001 → 00010 → 00100 →	2 → 1 → 1 → 1 →
5/14	4	9	3-3-2-3-3	10100 → 00110 → 01100 → 00101 →	3 → 1 → 3 → 2 →
6/14	6	10	3-2-3-3-3	10100 → 10001 → 00011 → 00110 → 10010 → 00101 →	2 → 1 → 1 → 3 → 1 → 2 →

where $y^i_{k,j}$ = integer coefficients for weight w_i. This also means that, under weight configuration k, there are $y^i_{k,j}$ number of w_i's in each codeword, and $j = 1$ to t where t denotes the total number of possible configurations that satisfy the given m value.

Further, we have

$$x^1_k + x^2_k + \cdots + x^r_k = l \text{ (i.e., the cone size)} \tag{3.3}$$

and

$$y^i_{k,j} \le x^i_k, \quad i = 1 \; to \; r, \; j = 1 \; to \; t, \; k = 1 \; to \; s. \tag{3.4}$$

Let Q_k represent the total number of possible codewords for each weight sequence of the k^{th} configuration of the given n value. We have

$$Q_k = \textstyle\sum_{j=1}^{t} \text{ (total number of ways of choosing weights for } j^{th} \; m \text{ configuration)}$$

Assume we have weight configuration [1,1,1,2,2] (i.e., $n = 7$) and the cone size = 5. Given $m = 3$, there are two m configurations: $m = 3 \cdot w_1 + 0 \cdot w_2$ and $m = 1 \cdot w_1 + 1 \cdot w_2$ where $w_i = i$. Let

us concentrate on the second m configuration. There are three choices to assign 1 to w_1 and two choices to assign 1 to w_2. That is, the total number of codewords for the second m configuration is $C(3,1) \cdot C(2,1) = 6$, and the codewords are 10010, 10001, 01010, 01001, 00110, 00101. Thus, the total number of codewords for each weight sequence under a specific weight configuration can be represented by

$$Q_k = \sum_{j=1}^{t} (\prod_{i=1}^{r} C(x_k^i, y_{k,j}^i)) \quad , \forall\ k = 1\ to\ s. \tag{3.5}$$

Note that, for each configuration of n, we are concerned with only one weight sequence. For example, given a weight set = {1,2}, cone size = 5, and $n = 6$, there is only one weight configuration (i.e., [1,1,1,1,2]) for the specific n value. We are concerned with only one weight sequence (e.g., 1-1-1-1-2) from which a set of codewords can be generated. Note that the number of codewords generated by any other weight sequence (e.g., 2-1-1-1-1) is the same as the one selected.

Example 1:
Let us consider a 3-out-of-6 code with weight set = {1,2} (i.e., $r = 2$) and cone size = 5. From equation 3.1, we have $n = 4 \cdot 1 + 1 \cdot 2 = 6$ and this is the only weight configuration that can satisfy the n value. This leads to $x_1^1 = 4$, $x_1^2 = 1$, and $k = 1$. Thus, the weight distributions can be 1-1-1-1-2, 1-1-1-2-1, 1-1-2-1-1, 1-2-1-1-1, 2-1-1-1-1. We are interested in only (any) one of them, since all different weight distributions generated from the same weight configuration have the same number of codewords as will be proved later. From equation 3.2, there are two possible m configurations for m=3 (for any weight sequence). They are: $m = 3 \cdot 1 + 0 \cdot 2 = 3$, and $m = 1 \cdot 1 + 1 \cdot 2 = 3$. So, we have $y_{1,1}^1 = 3$, $y_{1,1}^2 = 0$, $y_{1,2}^1 = 1$, $y_{1,2}^2 = 1$. From equation 3.5, we get $Q_1 = \sum_{j=1}^{2} (\prod_{i=1}^{2} C(x_1^i, y_{1,j}^i)) = [C(4,3) \cdot C(1,0)] + [C(4,1) \cdot C(1,1)] = 4 + 4 = 8$.

Table 3.3: Results for weight set {1,2,3}.

m/n	# code-words	cost	Weight sequence	Code sequence	Shift sequence
1/8	3	4	2-1-1-1-3	01000 → 00010 → 00100 →	2 → 1 → 1 →
2/8	4	6	1-2-1-3-1	01000 → 10001 → 00101 → 10100 →	1 → 2 → 2 → 1
3/8	5	10	1-1-3-1-2	00100 → 10001 → 00011 → 11010 → 01001 →	2 → 1 → 3 → 2 → 2 →
4/8	6	12	1-1-3-2-1	10100 → 00101 → 01011 → 01100 → 10011 → 11010 →	3 → 1 → 2 → 2 → 3 → 1 →
1/9	2	4	2-1-1-1-3	01000 → 00100 →	3 → 1 →
2/9	4	6	1-2-1-3-1	01000 → 10001 → 00100 →	1 → 2 → 1 →
3/9	5	10	1-1-3-1-2	11000 → 10001 → 00010 → 00101 → 01100 →	1 → 1 → 1 → 3 → 1 →
4/9	6	12	1-1-3-2-1	10100 → 01001 → 10011 → 00110 → 11010 →	1 → 1 → 1 → 2 → 1 →
1/10	1	3	2-2-1-2-3	00100 →	3 →
2/10	3	4	1-2-2-2-3	01000 → 00010 → 00100 →	2 → 1 → 1 →
3/10	4	8	2-1-2-3-2	11000 → 00010 → 01001 → 01100 →	2 → 2 → 3 → 1 →
4/10	4	6	2-1-2-3-2	10100 → 10001 → 00101 → 01010 →	2 → 2 → 1 → 1 →
5/10	6	12	2-1-3-2-2	10100 → 00110 → 11001 → 00101 → 01011 → 11010 →	3 → 2 → 2 → 1 → 3 → 1 →
1/10	2	4	2-1-1-3-3	01000 → 00100 →	3 → 1 →
2/10	2	3	1-3-2-1-3	00100 → 10010 →	2 → 1 →
3/10	4	6	1-3-1-3-2	01000 → 10001 → 00010 → 00101 →	1 → 1 → 1 → 3 →
4/10	5	7	1-2-3-1-3	10100 → 10001 → 00011 → 00110 → 11010 →	2 → 1 → 1 → 2 → 1 →
5/10	4	4	1-3-2-3-1	01100 → 11001 → 10011 → 00110 →	1 → 1 → 1 → 1 →
1/11	1	3	2-2-1-3-3	00100 →	3 →
2/11	2	4	1-2-2-3-3	01000 → 00100 →	3 → 1 →
3/11	4	6	1-3-2-3-2	01000 → 10001 → 00010 → 10100 →	1 → 1 → 3 → 1 →
4/11	3	5	1-2-3-3-2	10100 → 01001 → 10010 →	1 → 1 → 3 →
5/11	5	7	2-1-3-2-3	10100 → 10001 → 00011 → 00110 → 11010 →	2 → 1 → 1 → 2 → 1 →
1/12	1	3	2-3-1-3-3	00100 →	3 →
2/12	1	3	1-3-2-3-3	00100 →	3 →
3/12	4	4	1-3-3-3-2	01000 → 10001 → 00010 → 00100 →	1 → 1 → 1 → 1 →
4/12	3	8	3-1-3-2-3	11000 → 01001 → 01100 →	4 → 3 → 1 →
5/12	3	8	3-1-3-2-3	10010 → 00011 → 00110 →	4 → 1 → 3 →
6/12	6	10	3-1-3-2-3	10100 → 10001 → 00101 → 01011 → 01110 → 11010 →	2 → 2 → 1 → 2 → 2 → 1-

Table 3.4: Results for weight set {1,2,3,5}.

m/n	# code-words	cost	Weight sequence	Code sequence	Shift sequence
1/12	2	4	2-1-1-3-5	01000 → 00100 →	3 → 1 →
2/12	2	3	1-3-2-1-5	00100 → 10010 →	2 → 1 →
3/12	3	6	1-2-1-5-3	11000 → 00001 → 01100 →	2 → 3 → 1 →
4/12	3	6	1-2-3-1-5	10100 → 00110 → 11010 →	3 → 2 → 1 →
5/12	3	7	2-3-1-1-5	11000 → 00001 → 01110 →	2 → 3 → 2 →
6/12	4	4	2-1-5-1-3	01100 → 11001 → 10011 → 00110 →	1 → 1 → 1 → 1 →
1/13	1	3	2-2-1-3-5	00100 →	3 →
2/13	2	4	1-2-2-3-5	01000 → 00100 →	3 → 1 →
3/13	3	6	1-3-2-5-2	01000 → 10001 → 10100 →	1 → 4 → 1 →
4/13	2	3	1-2-5-3-2	10010 → 01001 →	2 → 1 →
5/13	4	8	2-1-2-5-3	11100 → 10001 → 00010 → 00101 →	2 → 1 → 1 → 4 →
6/13	3	6	1-2-5-3-2	10100 → 10011 → 11010 →	2 → 3 → 1 →
1/14	1	3	2-3-1-3-5	00100 →	3 →
2/14	1	3	1-3-2-3-5	00100 →	3 →
3/14	3	4	1-3-3-5-2	01000 → 10001 → 00100 →	1 → 2 → 1 →
4/14	2	4	2-3-1-3-5	01100 → 00110 →	3 → 1 →
5/14	3	6	1-3-2-3-5	01100 → 00001 → 00110 →	3 → 2 → 1 →
6/14	4	6	1-3-5-2-3	10100 → 01001 → 10011 → 11010 →	1 → 1 → 3 → 1 →

Chapter 4

Codeword Ordering

Once a set of codewords has been determined, the next problem is how to apply the codewords to the scan chain such that the total test application length (time) can be minimized. The key idea to reduce the test application length is to overlap consecutive codewords as much as possible. Consider two codewords 11000 and 00101 applied to the scan chain. After the first codeword 11000 has been applied (starting from bit 1), the last two bits 00 of codeword 1 can be overlapped with the first two bits (00) of the second codeword. Significant test application time can be reduced by codeword overlapping. The problem of ordering the codewords in an optimal way (i.e., to find an optimal code sequence) is the well-known *traveling salesman problem*, which is NP complete. If the number of codewords is smaller than 20, we can find the optimal solution by exhaustive searching. However, for larger number of codewords than 20, exhaustive searching is time-consuming. Hence, we propose to use the *coalesced simple paths* algorithm (CSP) for ordering the codewords.

4.1 Coalesced Simple Paths Algorithm

This algorithm uses a greedy approach to construct a tour edge-by-edge. It is similar to *Kruskal's algorithm* with the added simplicity of avoiding the complicated *union-find* data structure. The algorithm begins by first constructing a completely connected graph G1 as shown in Fig. 4.2. In this graph, the nodes represent the codewords of a given m-out-of-n code, while the weight of each edge is the cost between the the corresponding pair of nodes (i.e., the number of (left) shifts required to obtain the succeeding codeword from the current codeword). The algorithm starts with removing the minimum-cost edge (u,v) of all edges, and adding it to connect nodes u and v in graph G2 along with other previously added edges. Note that, G2 will be the optimal graph if the minimum-weight edge is always removed from G1 and added between proper nodes in G2 in each iteration. However, by repeating this process, there is a possibility that next time an edge of minimum weight could be selected from G1 and added to G2 without connecting new nodes into the tour in G2. Hence, the next task is to eliminate all such possibilities of looping that can arise with the removal of edge (u,v). To accomplish this task, all edges emerging from node u and all edges terminating to node v are removed in G1. Similarly, edge (v,u) is removed from G1 to avoid any looping in G2. Also, edge (t,s) is removed from G1, where (t,s) is the terminal and source of the *partial tour* created in G2 when edge (u,v) is added in it. For example, in Fig. 4.1, edges (D,C) and (C,B) are already removed from graph G1. The *partial tour* in graph G2 consists of edges (D,C) and (C,B). The terminal and source of this *partial tour* in G2 are B and D, respectively. If (B,D) is not removed from graph G1, then this edge will be selected during the next iteration and

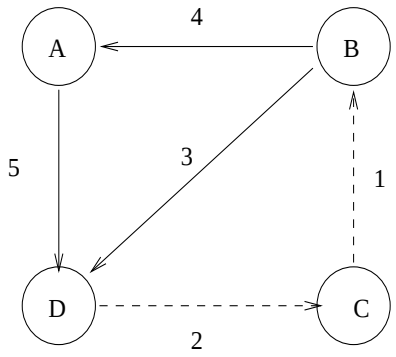


Figure 4.1: Transitive edge removal due to partial tour.

the tour in G2 will be completed without connecting node A. That is the reason for removing edge (t,s) from G1. The above process is repeated till the tour is completed in G2.

Example 2:
A complete example is shown in Fig. 4.2 for a 5-out-of-15 code with weight distribution 3-3-2-3-3. The codewords 10100, 00110, 01100 and 00101 are represented as nodes A, D, C, and B, respectively. Edge (D,C) is removed from G1 first as it has the minimum weight (step 2). It is then added to graph G2 (step 2). Now, all edges from D, namely (D,A) and (D,B), are removed from G1. Similarly, all edges to C, i.e., (B,C) and (A,C), are also removed from G1. Further, the edge (C,D) is removed from G1 to avoid looping (step 3). The algorithm proceeds repeatedly with selecting edges (B,A), (A,D) and (C,B) consecutively. The tour generated in this case is optimal.

Coalesced_Simple_Paths
Input: G1 (the complete graph for the codewords and cost for each pair of adjacent nodes).
G2 (the graph containing the codewords as nodes).
Output: G2 (a completed tour).
begin
1 Sort the edges in G1 with increasing order by weight;
do{
2 pick up an edge (u,v) from the top of sorted edge list of G1;
3 Remove all edges emerging from u from the sorted list of G1;
4 Remove all edges terminating to v from the sorted list of G1;
5 Remove edge (v,u) from the sorted list of G1;
6 Add edge (u,v) to G2;
7 Remove edge (t,s) from the sorted list of G1; //where (t,s) is the terminal and source of
//the partial tour in G2
} until a tour is generated in G2;
end

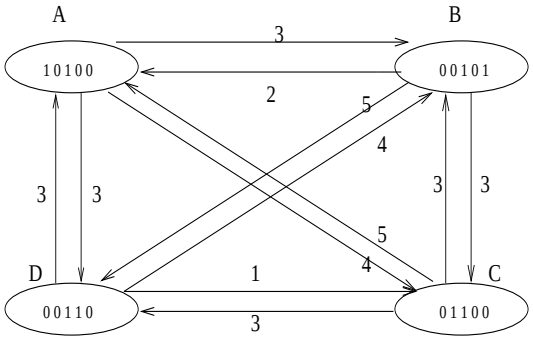
Step 1 in this algorithm is used to sort the edges in G1 with increasing order by weight. In step 2, the edge with the lowest weight is removed from G1 for process. Steps 3, 4, 5 and 7 are performed to eliminate the possibilities of edge looping. The lowest-weight edge is inserted into graph G2 in step 6 to form a tour. Steps 2 through 7 are performed repetitively till a tour is generated in G2. This algorithm is known to perform better than the *greedy algorithm* because, at

Table 4.1: Comparison of CSP over optimal ordering

Weight distribution	code	# of Nodes	cost (CSP)	cost (optimal)
2-1-1-2-2	3/8	6	10	10
3-2-3-3-3	6/14	6	10	10
2-1-3-2-2	5/10	6	12	12
2-3-5-3-3	8/16	6	12	12
1-1-1-2-1	2/6	7	10	10
1-1-2-2-1	3/7	7	10	10
1-1-2-1-1	3/6	8	12	12
1-1-1-1-1-2	2/7	11	18	18

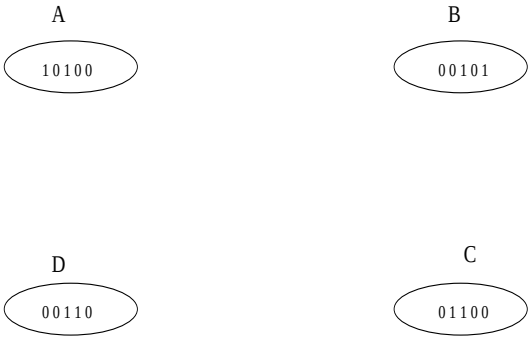
every step, it always chooses the best from all available edges in the entire graph G1. However, the greedy method chooses among edges connected to the current node only. We have experimented the algorithm for different m and n values (with the number of nodes smaller than 20). The results are depicted in Table 4.1. It can be observed that the results of the algorithm are optimal in all cases. The run time of the algorithm is $O(l \cdot N^2)$ where N is the number of codewords, because of constructing the completely connected graph. Further, l is the cone size and is taken into account because of finding the weight of the edge between each pair of nodes. To do this, one has to perform string matching (which is of complexity $O(l)$). Note that l is very small compared to N.

GRAPH G1

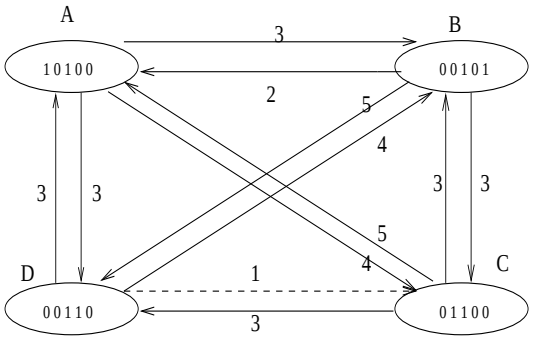


Step 1. Construct completely connected graph.

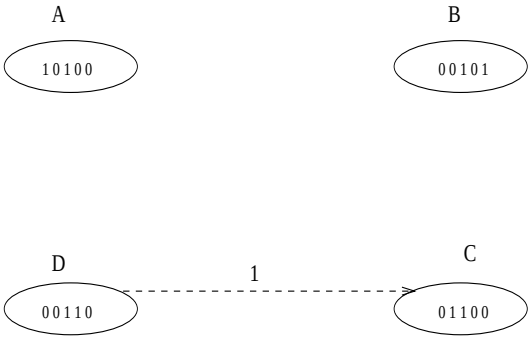
GRAPH G2



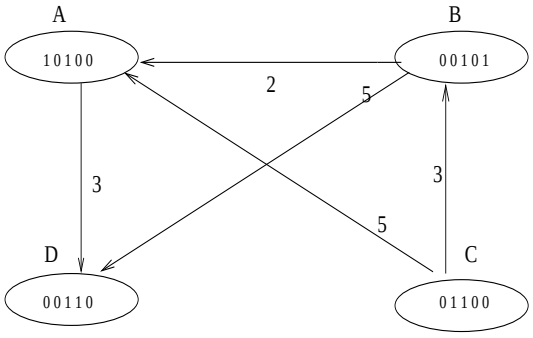
Step 1. Construct the graph without edges.



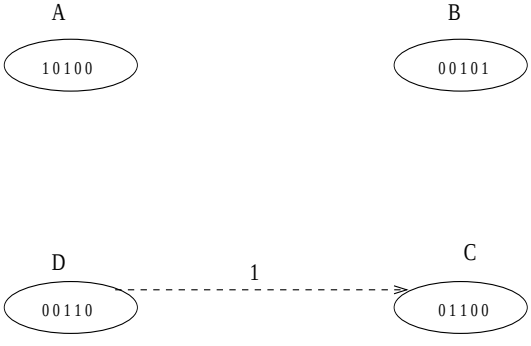
Step 2. Remove (D,C) because this edge has minimum weight



Step 2. Insert edge (D,C).



Step 3. Remove all edges to C, all edges from D, and edge (C,D).



Step 3. Do nothing.

(Continued....)

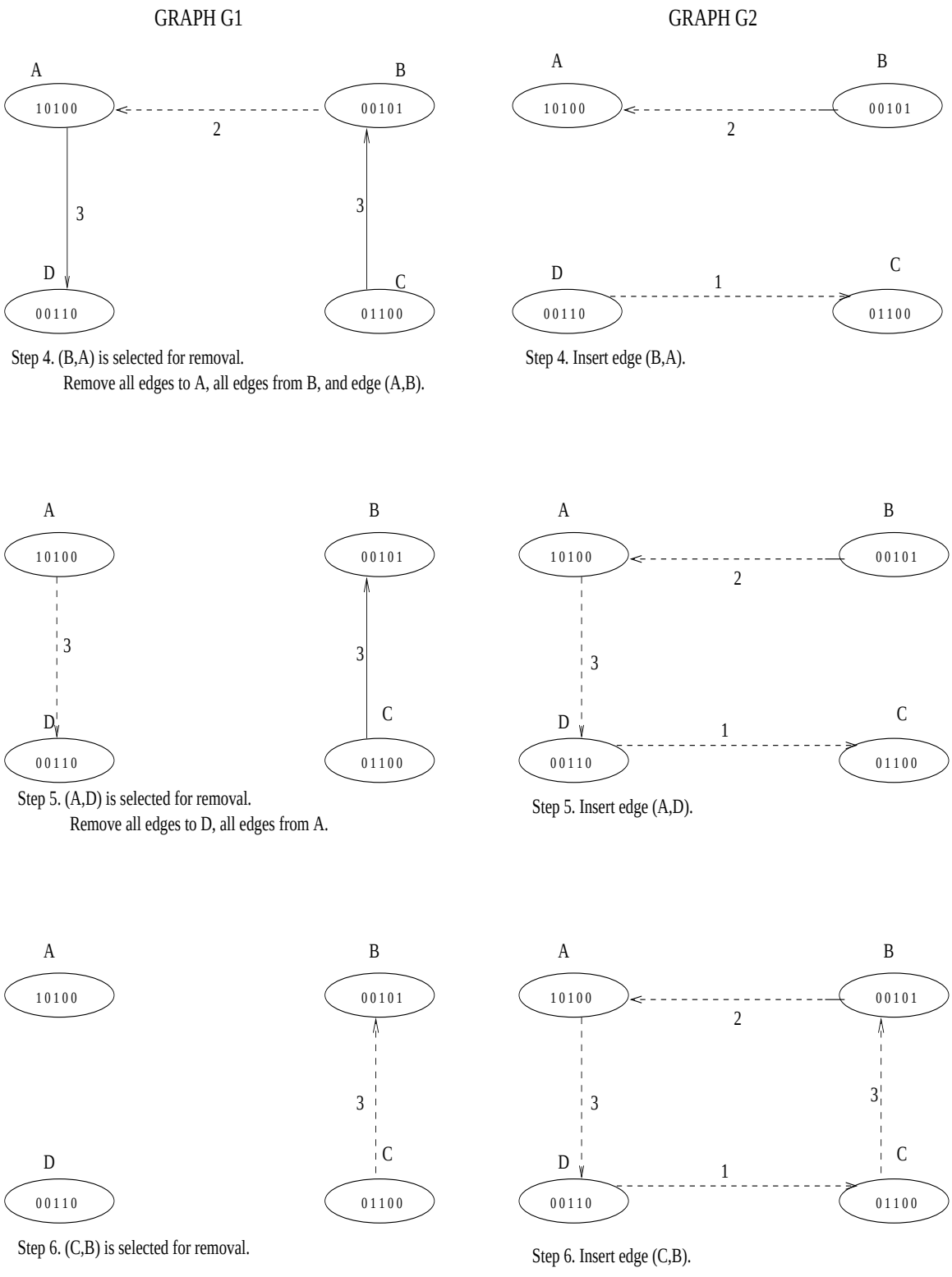


Figure 4.2: Steps involved in coalesced simple path algorithm.

Chapter 5

Aliasing Probability Analysis for Overlapped Codewords

In this chapter, we analyze the aliasing probabilities for bidirectional and multi-directional errors. Due to an error (or errors), a codeword might be changed into a non-codeword that can be easily detected by a checker. However, if a codeword is changed to another codeword, then the error cannot be detected. The aliasing probability of a codeword is the probability that the codeword is changed into another codeword (and thus the error cannot be detected). In this analysis, first, we assume that two bits in a codeword can be changed due to an error (i.e., bidirectional error). For example, given a weight sequence 1-1-1-1-2, 10001 is a codeword for the 3-out-of-6 code with cone size 5. However, if an error changes two bits such that 10001 is transferred to 01001, then this error cannot be detected. The analysis will be extended to the case of multi-directional errors. Figure 5.1 shows an example of a bidirectional error which cannot be detected. The bidirectional error results from signal delay at lines out1 and out2 due to crosstalk between both lines.

As discussed in codeword ordering, the codewords can be applied to the scan chain in an overlapped manner such that the test application time can be greatly reduced. The aliasing probability analysis for overlapped and non-overlapped codeword applications are different. In fact, the analysis for non-overlapped codeword application is much easier, and the aliasing probability derived can be used as an upper bound for the overlapped analysis. Again, consider two codewords 11001 and 01101 applied to the scan chain, and assume that the weight sequence of both codewords is 1-2-1-1-1. Obviously, both codewords are part of the 4-out-of-6 code. It can be observed that bit pattern 01 can be overlapped when 11001 and 01101 are applied. Note that 11001 is applied before 01101 and each codeword is applied from the leftmost bit. Assume bit pattern 01 in the first codeword is changed to 10 due to a scan chain error. When the first codeword is scanned out of the scan chain, the checker cannot detect this error if both codewords are applied in a non-overlapping manner. However, the checker can easily detect the error if both codewords are applied in an overlapped manner. The reason is that, though the checker cannot detect the first codeword 11010 (with $m = 4$) scanned out, the checker can find the m value of 10101 (the second codeword scanned out) equal to 3 which is incorrect. From this example, we can see that overlapping codewords for test application is more powerful than its non-overlapping counterpart in terms of test application time and aliasing probability.

In this chapter, we would present the analysis of bidirectional and multi-directional errors for overlapped codewords. Note that codeword and cone is used interchangeably in the following discussions. Further, in the following analysis, we have to get another codeword from the current

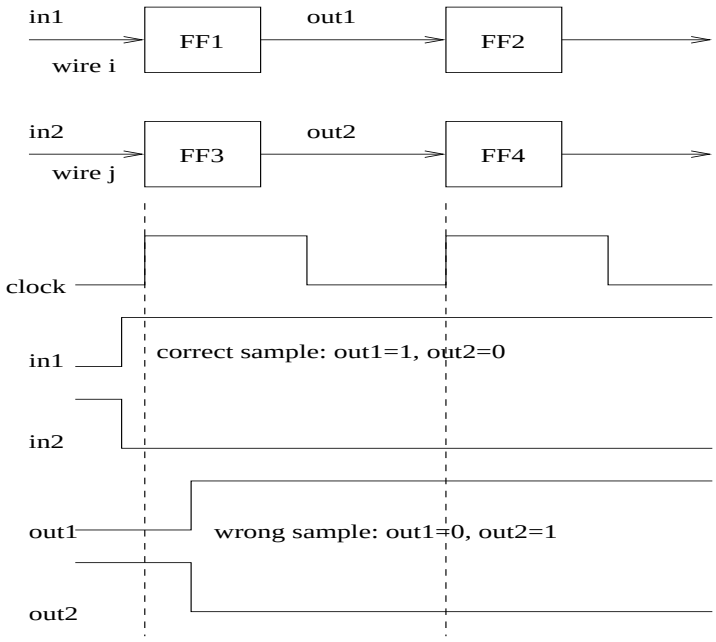


Figure 5.1: Example illustrating bidirectional error.

codeword by moving backward or forward from the current codeword. By definition, a codeword CW_1 is obtained by moving *backward* (*forward*) from the current codeword CW_2, if CW_1 is applied to the scan chain before CW_2. For example, let $CW_1 = 11001$ and $CW_2 = 01101$ and CW_1 be applied before CW_2. When both codewords are applied to the scan chain, we have the overall test pattern 11001101 (the leftmost bit is applied first). If the current codeword under consideration is 01101 (11001), then codeword 11001 (01101) can be obtained from the current codeword by moving backward (forward).

5.1 Bidirectional Aliasing Analysis

Before proceeding with the bidirectional aliasing analysis, we define several related terms as follows.

1. **Current double error cone (CDC)** This is the codeword which is under consideration for bidirectional error analysis and contains two erroneous bits i and j. All bidirectional aliasing cases will be analyzed based on the CDC.

2. **Left singular error cone (LSC)** This is the codeword which is obtained from the CDC by moving backwards from the CDC and contains the left one (i.e., i) of the erroneous bits.

3. **Right singular error cone (RSC)** This is the codeword which is obtained from the CDC by moving forwards from the CDC and contains the right one (i.e., j) of the erroneous bits.

4. **Left double error cone (LDC)** This is the codeword which is obtained from the CDC by moving backwards from the CDC and contains both (i.e., i and j) erroneous bits.

5. **Right double error cone (RDC)** This is the codeword which is obtained from the CDC by moving forwards from the CDC and contains both (i.e., i and j) erroneous bits.

Note that there can be many LSC's, RSC's, LDC's and RDC's for a particular CDC. Actually, all cones applied before the CDC are potential LSC's or LDC's whereas all cones applied after the CDC are potential RSC's or RDC's. Consider a 2-out-of-6 code with weight set = {1,2} and weight sequence = 1-1-1-2-1. The code sequence is given by 11000 → 10001 → 00010 → 00101 → 10100 → 01001 → 01100. The entire sequence is depicted in Fig. 5.2. The codewords are shifted into the scan chain in a right-to-left fashion. From now on, unless otherwise stated, the shifting is always performed in a right-to-left fashion (bit 1 is applied first in Fig. 5.2). Let C1 be the CDC. For the bidirectional error at locations shown by $x1$ and $x2$, C7 is a LSC because it is at the left side of the CDC and contains bit $x1$. Each of C3 and C4 is a RSC because it is at the right side of the CDC and contains bit $x2$. C2 is a RDC because it contains both $x1$ and $x2$. Now, assume C4 is the new CDC with a bidirectional error at locations $x2$ and $x3$. In this case, C1 is a LSC, C5 is a RSC and C2 is a LDC.

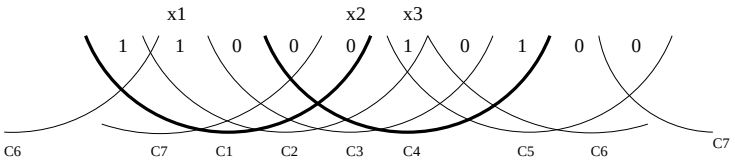


Figure 5.2: 2-out-of-6 code sequence.

The power of applying overlapped codewords lies in the fact that most of errors of a CDC can be detected by its neighboring codewords (if they cannot be detected by the CDC itself), namely, by LSC, RSC, LDC, or RDC. In the following discussions, we describe six possible cases that can arise while analyzing an *undetectable* bidirectional error for a particular CDC.

Case 1. For a bidirectional error in the CDC, there exists one or more LSC. As will be proved in Theorem 1, the error can always be detected in this case.

Case 2. For a bidirectional error in the CDC, there exists one or more RSC. Again, it will be proved in Theorem 1 that the error is always detected in this case.

Case 3. For a bidirectional error in the CDC, there exists one LDC where the weights of the erroneous bits are different. In this case, the error can always be detected (refer to Theorem 2).

Case 4. For a bidirectional error in the CDC, there exists one RDC where the weights of the erroneous bits are different. In this case, the error can always be detected (refer to Theorem 2).

Case 5. For a bidirectional error in the CDC, there exists one or more LDC and/or RDC but the the error still cannot be detected.

Case 6. For a bidirectional error in the CDC, there is no LSC, RSC, LDC or RDC. The error gets no chance of being detected as it is not contained in any neighboring cone.

For a given CDC, in Theorem 1, we prove that the bidirectional error of a CDC will be detected by a LSC or RSC (if it exists) of the CDC. This theorem will, therefore, cover Cases 1 and 2 discussed above.

Theorem 1. A bidirectional error of a CDC can be detected if one of the erroneous bits involved in the error is contained in a LSC or RSC.

Proof: Suppose bits i and j of a cone c_p are susceptible to a bidirectional error and both have weight w. Also, bit i is contained by another cone c_q. Assume that cone c_q is a LSC. This is illustrated in Fig. 5.3. Note that erroneous bits are marked by "x" and that the bits are numbered in a left-to-right manner. Now, if i (j) changes from $0 \rightarrow 1$ ($1 \rightarrow 0$) or $1 \rightarrow 0$ ($0 \rightarrow 1$), then this error will be masked in c_p; but, c_q will undergo a weight increment (decrement). This is a *unidirectional error* in cone c_q. Similar logic is true when c_q is a RSC and contains bit j as shown in Fig. 5.4. **Q.E.D**

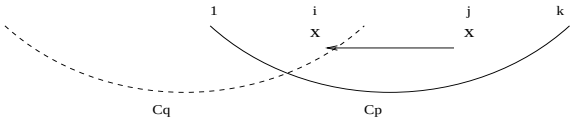


Figure 5.3: Bidirectional error detected by cone c_q (LSC).

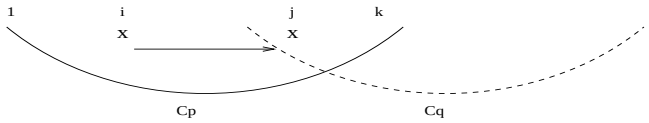


Figure 5.4: Bidirectional error detected by cone c_q (RSC).

In Theorem 2, we would prove that the bidirectional error of a CDC can be detected by the LDC or RDC, if it exists. The detection of this error depends on the weights of the erroneous bits in the LDC or RDC. This theorem covers Cases 3 and 4 as explained previously.

Theorem 2. A bidirectional error can be detected if both bits involved in the error is contained by a LDC or RDC and the weights of both bits are different.
Proof: Suppose bits i and j of cone c_p are susceptible to a bidirectional error and both have weight w. Also, bits i and j are contained in LDC c_q which is s shifts away from cone c_p. This is illustrated in Fig. 5.5. The weights of i and j in cone c_q will be $w[i-s]$ and $w[j-s]$ respectively if cone c_q is a LDC; but $w[i+s]$ and $w[k+s]$ respectively if the cone c_q is a RDC (Fig. 5.6). If their weights are not the same in c_q, then the bidirectional error will contribute a weight change of $(w[i-s]-w[j-s])$ $((w[i+s]-w[j+s]))$, if the cone c_q is a LDC (RDC), which will be detected immediately. **Q.E.D**

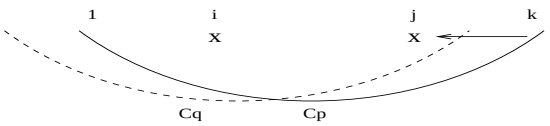


Figure 5.5: Bidirectional error detected by cone c_q (LDC).

In the following theorem, we prove that a bidirectional error cannot be detected if it is not contained by any cone or it is contained by a neighboring cone (LDC or RDC) where the weights of both erroneous bits are the same. This theorem covers Cases 5 and 6.

Theorem 3. A bidirectional error is masked, if it is not contained by any other cone or contained by a cone where the weights of both bits are the same.

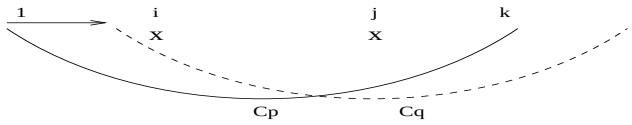


Figure 5.6: Bidirectional error detected by cone c_q (RDC).

Proof: Suppose bits i and j of cone c_p are susceptible to a bidirectional error and both have weight w. If i and j are not contained by any other cone, the error will only cause a weight change of zero in c_p which is not detectable. This case is illustrated in Fig. 5.7. Assume both bits are contained in another cone c_q (see Fig. 5.5) and the weights of both bits are $w[i-s]$ and $w[j-s]$ ($w[i+s]$ and $w[j+s]$) respectively, when cone c_q is a LDC (RDC). If the weights $w[i-s]$ ($w[i+s]$) and $w[j-s]$ ($w[j+s]$) are the same, then the bidirectional error between i and j will not cause any weight change in c_q either, and the error will be masked. **Q.E.D**

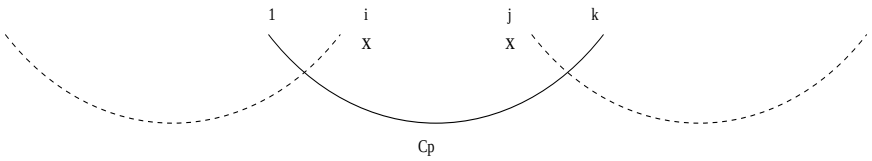


Figure 5.7: Bidirectional error not covered by a neighboring cone.

5.1.1 Total Number of Undetected Bidirectional Errors

Based on Theorems 1, 2 and 3, we present an algorithm to enumerate the bidirectional aliasing. There are totally four search regions for each error at the CDC as illustrated in Fig. 5.8. Region 1 (region 2) is searched for LSC (RSC), while region 3 (region 4) is searched for LDC (RDC). In Fig. 5.8, k is the cone size and i & j are the bit numbers which are susceptible to a bidirectional error. Note that bits are numbered from left to right. If a LSC or RSC exists for a bidirectional error in the CDC, then the error is immediately detected. Otherwise, the algorithm checks if a LDC or RDC exists for the error and if the error can be detected. If not, the error is counted as a bidirectional aliasing. In following paragraphs, we present an example to illustrate various steps of the algorithm.

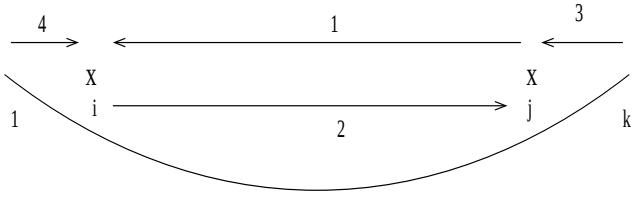


Figure 5.8: Different regions for searching an adjacent cone.

Example 1. Let us consider an unoptimized codeword ordering of a 3-out-of-8 code sequence with weight set {1,2} and cone size of 5 to illustrate the operation of algorithm *BidirectionalDetect()*.

The codewords, bit numbers and the weight distribution for the 3-out-of-8 code are illustrated in Fig. 5.9. The corresponding bidirectional graph G is constructed where the nodes are the

codewords of the given 3-out-of-8 code. The clockwise directed edges are positive and represent the number of bits required to shift to the succeeding codeword. The anticlockwise directed edges are negative and represent the number of shifts required to reach the preceding codeword from the current codeword. By traveling forwards, we can reach a RSC or RDC; whereas, by traveling backwards we can reach a LSC or LDC. The bidirectional graph G for this example is shown in Fig. 5.10.

	weight distribution				
Cone	1	1	2	2	2
C1	1	0	1	0	0
C2	0	1	1	0	0
C3	1	0	0	1	0
C4	0	1	0	1	0
C5	1	0	0	0	1
C6	0	1	0	0	1
bit numbers	0	1	2	3	4

Figure 5.9: Codewords of a 3-out-of-8 code.

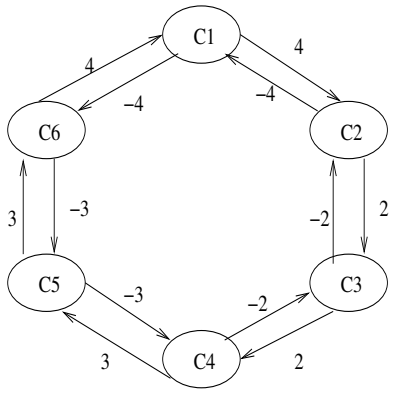


Figure 5.10: The corresponding bidirectional graph G.

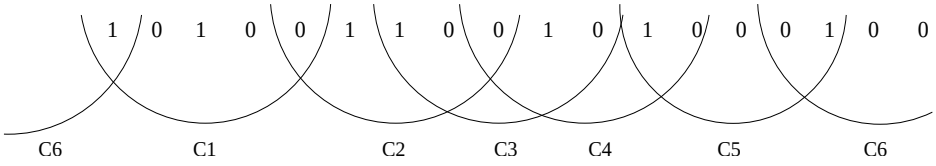


Figure 5.11: Equivalent errors in two cones.

Now, for each codeword, the potential bidirectional aliasing error locations are identified. A bit pair is susceptible to a bidirectional aliasing if they are of same weight, and their bit values are complementary to each other. All error locations for each codeword in this example are given as follows.

C1 - (0,1), (2,3), (2,4)
C2 - (0,1), (2,3), (2,4)
C3 - (0,1), (2,3), (3,4)
C4 - (0,1), (2,3), (3,4)
C5 - (0,1), (2,4), (3,4)
C6 - (0,1), (2,4), (3,4)

Totally, the number of errors equal 18.

It can be observed that, when the codewords are shifted in an overlapped manner, some of the errors are shared by two or more cones (i.e. LDC's or RDC's). Such errors are equivalent to each other. It is, therefore, necessary to remove the redundant error at cone x if the error has already been accounted in another cone y. For example, error in bits (2,3) in CDC C2 is also error in bits (0,1) in RDC C3 as shown in Fig. 5.11. Therefore, error (2,3) is dropped from C2 because it will be accounted as an error (0,1) in C3 or vice versa. After removing all equivalent errors, the reduced error list is

C1 - (0,1), (2,3), (2,4)
C2 - (0,1), (2,4)
C3 - (0,1), (3,4)
C4 - (0,1), (2,3)
C5 - (0,1), (2,4)
C6 - (0,1), (2,4), (3,4)

Totally, the number of errors after dropping equals 14.

Table 5.1: Error detection for a 3-out-of-8 code sequence.

codeword	error	detected	case
C1	(0,1)	yes	1
	(2,3)	no	5
	(2,4)	yes	2
C2	(0,1)	yes	1
	(2,4)	yes	2
C3	(2,3)	no	6
	(3,4)	yes	4
C4	(0,1)	yes	1
	(2,3)	yes	1
C5	(0,1)	no	6
	(2,4)	yes	2
C6	(0,1)	no	6
	(2,4)	yes	2
	(3,4)	yes	2

Each error at every codeword (after error dropping) is analyzed to check whether it can be detected by a possibly existent LSC, RSC, LDC or RDC. For example, let C1 is the CDC as shown

in Fig. 5.11. We try to analyze error (0,1) from the reduced error list. Using graph G to travel backwards, we can observe that C6 is the LSC which is 4 shifts away to the left side of C1 and contains the 0^{th} error bit. This is clearly illustrated in Fig. 5.11. Therefore, this error is detectable. Further, let us consider error (2,4) in CDC C1. Using graph G, we go backwards to see if any LSC exists. Since no LSC covers the error at the 2^{nd} bit, we travel forwards in graph G to see if any RSC exists. From Fig. 5.11, it is obvious that C2 is a RSC and contains the 2^{nd} erroneous (i.e., bit 4) bit of C1. Therefore, error (2,4) is detectable. Similarly, assume C3 is the CDC and we analyze error (3,4). We can observe that, by going backwards (forwards) from C3 in graph G, we cannot get any LSC (RSC). Next, by moving backwards from C3 in G, we cannot find any LDC. Finally, we move forwards and find RDC C4 where the weights of erroneous bits are 1 and 2, respectively. Hence, the error is detectable. From Table 5.1, it is evident that out of 14 reduced errors, a total of 10 errors are detected. Hence, the total number of errors that are masked is 4.

The function *GenerateErrorList()* in algorithm *BidirectionalDetect()* picks one cone at a time and finds out each pair of potential locations (in the cone) that are susceptible to a bidirectional error. Each pair of such locations must have equal weight and the bit values of both locations must be complementary to each other.

The function *Drop()* in algorithm *BidirectionalDetect()* picks up one cone at a time, and for each of its errors checks if the error will be analyzed by the succeeding cone or not. If it will be analyzed by the succeeding cone, then the current error is redundant and can be dropped from the list. For example, in Fig. 5.5, error pair (i, j) is already analyzed in CDC c_p but it is also covered in LDC c_q. So, it will be analyzed again when c_q arrives. Hence, this error can be kept in one of the covering cones and dropped from the other cone.

Once the reduced error list is available, the next task is to check if each error can be detected by any of its neighboring cones. Since Cases 1 and 2 are computationally least expensive, the algorithm first checks for a LSC or RSC for the error in the CDC. If such a cone exists, then the error is detectable, and next error is picked for analysis. Otherwise, the algorithm proceeds with detecting the error by LDC's or RDC's of the CDC (which contains the error). Case 1 through 6 are shown in algorithm *BidirectionalDetect()*.

Algorithm BidirectionalDetect ()
begin
Input: A cyclic graph G where nodes are the cone (codeword) numbers and edges represent number of shifts;
Output: Number of bidirectional aliasing cases;
GenerateErrorList();
Drop();
for (each error from the reduced error list)
do
  begin
    move backwards in region 1 of graph G; //see Fig. 5.8
    if (cone exist)
      then print "bidirectional error is detectable" and exit; // Case-1
    move forwards in region 2 of graph G;
    if (cone exist)

```
        then print "bidirectional error is detectable" and exit;// Case-2
    move backwards in region 3 of graph G;
    if (no cone found)
        then no_overlap_flag1 = 1;
    else
        for (each cone a found above)
        do
            begin
                x = weight[i + shift]_a;
                y = weight[j + shift]_a;
                if (x != y)
                    then print "bidirectional error is detectable" and exit; // Case-3
            end
    move backwards in region 4 of graph G;
    if (no cone found)
        then no_overlap_flag2 = 1;
    else
        for (each cone b found above)
        do
            begin
                x = weight[i - shift]_b;
                y = weight[j - shift]_b;
                if (x != y)
                    then print "bidirectional error is detectable" and exit; // Case-4
            end
    if (no_overlap_flag1 == 1 && no_overlap_flag2 == 1)
            then print "no cone overlaps these affected bits" and
                    increment bidirectional_count; // Case-6
    else
            print "more than one cone overlaps these affected bits" and
            increment bidirectional_count;// Case-5
  end
end
```

5.1.2 Total Number of Two-Bit Errors

The algorithm *BidirectionalDetect()* enumerates the number of bidirectional aliasing cases. To find the bidirectional probability, we must calculate the total number of possible two-bit error cases in the particular m-out-of-n code sequence. In the case of a non-overlapped code sequence, the total number of possible two-bit errors are simply Q x C(l,2) where Q is the number of codewords and l is the cone size. However, this is not true in the case of an overlapped code sequence because some of the two bit errors are counted multiple times due to overlap of codewords. We must, therefore, use the inclusion-exclusion principle to eliminate such cases and calculate the exact number of two-bit error cases.

Assume there are Q codewords in a m-out-of-n code (k_1, k_2, ..., k_Q) which are optimally

ordered for overlapping. Using the inclusion-exclusion principle, the total number of ways of picking two bits for bidirectional error is given by,

$$N(k_1 \cup k_2 \cup ... \cup k_Q)_2 = S_{12} - S_{22} + ... + (-1)^{i+1} S_{i2} + ... + (-1)^{Q+1} S_{Q2} \tag{5.1}$$

where,

S_{12}= N$(k_1)_2$ + N$(k_2)_2$ + ... + N$(k_Q)_2$
= $\binom{k}{2}$ + $\binom{k}{2}$ + ... + $\binom{k}{2}$
= Q x $\binom{k}{2}$, where k is the cone size.
S_{22}= N$(k_1 \cap k_2)_2$ + N$(k_1 \cap k_3)_2$ + ... + N$(k_{Q-1} \cap k_Q)_2$
= $\binom{k_1 \cap k_2}{2}$ + $\binom{k_1 \cap k_3}{2}$ + .. + $\binom{k_{Q-1} \cap k_Q}{2}$
S_{32}= N$(k_1 \cap k_2 \cap k_3)_2$ + N$(k_1 \cap k_2 \cap k_4)_2$ + ... + N$(k_{Q-2} \cap k_{Q-1} \cap k_Q)_2$
= $\binom{k_1 \cap k_2 \cap k_3}{2}$ + $\binom{k_1 \cap k_2 \cap k_4}{2}$ + .. + $\binom{k_{Q-2} \cap k_{Q-1} \cap k_Q}{2}$
.
.
.
S_{Q2}= N$(k_1 \cap k_2 \cap ... \cap k_Q)_2$
= $\binom{k_1 \cap k_2 \cap ... \cap k_Q}{2}$.

It is evident from equation 5.1 that the total number of terms are very large (actually, exponential w.r.t. the number of codewords). However, it was found that most of the terms can be simply eliminated because they can never exist in a given a m-out-of-n code. Further, the equation can be simplified since each term containing intersection of three or more cones can be reduced to a term containing intersection of two cones. We present two theorems (4 and 5) to simplify equation 5.1.

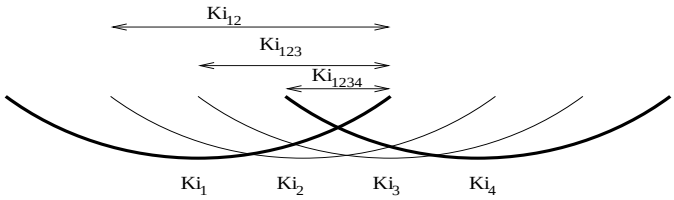


Figure 5.12: Four overlapping cones.

In Theorem 4, we prove that a term containing intersection of three or more cones can be reduced to a term containing intersection of two cones. Consider four overlapped cones $k_{i_1}, k_{i_2}, k_{i_3}$ and k_{i_4} as shown in Fig. 5.12. Let $k_{i_{12}}$ be the common bits between cones k_{i_1} and k_{i_2}, $k_{i_{123}}$ be the common bits between cones k_{i_1}, k_{i_2}, and k_{i_3}, while $k_{i_{1234}}$ be the common bits between $k_{i_1}, k_{i_2}, k_{i_3}$, and k_{i_4}. It can be observed from Fig. 5.12 that $k_{i_{1234}}$ is equal to $k_{i_{14}}(k_{i_1} \& k_{i_4})$ regardless of the amount of overlap involved by k_{i_2} and k_{i_3}.

Theorem 4: For a code sequence $k_1 \rightarrow k_2 \rightarrow ... \rightarrow k_Q$ of a m-out-of-n code, we have $(k_{i_1} \cap k_{i_2} \cap ... \cap k_{i_p}) = (k_{i_1} \cap k_{i_p})$ where $k_{i_1}, k_{i_2}, ..., k_{i_p}$ are any combination of p cones (from left to right) out of Q available cones, and k_{i_1} is the leftmost cone in the code sequence while k_{i_p} is the rightmost cone.

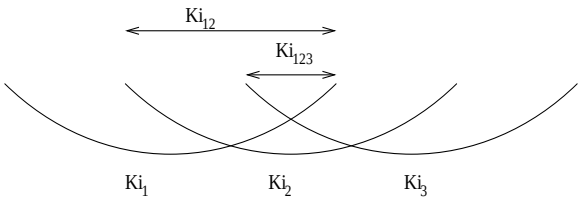


Figure 5.13: Overlapping cones.

Proof. This theorem can be proved using mathematical induction. The equation is valid for two cones. Now, consider three cones, k_{i_1}, k_{i_2} and k_{i_3} which overlap each other as shown in Fig. 5.13.

$$(k_{i_1} \cap k_{i_2} \cap k_{i_3}) = ((k_{i_1} \cap k_{i_2}) \cap k_{i_3}) = (k_{i_{12}} \cap k_{i_3}) = (k_{i_{123}}) = (k_{i_1} \cap k_{i_3}) \tag{5.2}$$

Let us assume that the proposition is true for t overlapping cones. Therefore,

$$(k_{i_1} \cap k_{i_2} \cap ... \cap k_{i_t}) = (k_{i_1} \cap k_{i_t}) \tag{5.3}$$

We now prove that the proposition is true for $(t+1)$ cones.

$(k_{i_1} \cap k_{i_2} \cap ... \cap k_{i_t} \cap k_{i_{t+1}}) = (k_{i_1} \cap k_{i_t} \cap k_{i_{t+1}})$ (from equation 5.3)

$= (k_{i_1} \cap k_{i_{t+1}})$ (from the equation 5.2)

Since the proposition is valid for $(t+1)$ number of overlapped cones, it is valid for any number of overlapped cones. Therefore,

$$(k_{i_1} \cap k_{i_2} \cap ... \cap k_{i_p}) = (k_{i_1} \cap k_{i_p}) \tag{5.4}$$

Q.E.D.

Theorem 5: For a code sequence $k_1 \rightarrow k_2 \rightarrow ... \rightarrow k_Q$ of a m-out-of-n code, we have $(k_{i_1} \cap k_{i_2} \cap ... \cap k_{i_p}) = 0$, if cones k_{i_1} and k_{i_p} are l shifts away from each other where l is the cone size.
Proof. From Theorem 1, we have $(k_{i_1} \cap k_{i_2} \cap ... \cap k_{i_p}) = (k_{i_1} \cap k_{i_p})$. If two cones k_{i_1} and k_{i_p} are more than l shifts away from each other, then there is no overlap between them. Hence, $(k_{i_1} \cap k_{i_p}) = 0$. **Q.E.D.**

Using Theorems 1 and 2, it is possible to simplify the calculation of two-bit error cases as most of the terms will either cancel out or turn out to be zero.

Example 2. Consider a 5-out-of-12 code with weight set {1,2,5} and weight distribution of 2-1-2-2-5. The code sequence is shown in Fig. 5.14.

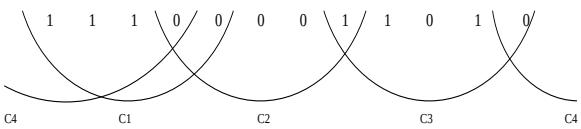


Figure 5.14: A 5-out-of-12 code sequence.

We have the following equations.
$N(k_1)_2 = N(k_2)_2 = N(k_3)_2 = N(k_4)_2 = C(5,2) = 10$,

$(k_1 \cap k_2) = 2$,
$(k_1 \cap k_3) = 0$ (from theorem 5),
$(k_1 \cap k_4) = 0$ (from theorem 5),
$(k_2 \cap k_3) = 1$,
$(k_2 \cap k_4) = 0$, (from theorem 5)
$(k_3 \cap k_4) = 1$,
$(k_1 \cap k_2 \cap k_3) = (k_1 \cap k_3) = 0$ (from theorem 4),
$(k_1 \cap k_2 \cap k_4) = (k_1 \cap k_4) = 0$ (from theorem 4),
$(k_1 \cap k_3 \cap k_4) = (k_1 \cap k_4) = 0$ (from theorem 4),
$(k_2 \cap k_3 \cap k_4) = (k_2 \cap k_4) = 0$ (from theorem 4),
$(k_1 \cap k_2 \cap k_3 \cap k_4) = (k_1 \cap k_4) = 0$ (from theorem 4),

From equation 5.1, we have
$N(k_1 \cup k_1 \cup ... \cup k_4)_2 = [N(k_1)_2 + N(k_2)_2 + N(k_3)_2 + N(k_4)_2]$ - $N(k_1 \cap k_2)_2$ - $N(k_2 \cap k_3)_2$ - $(k_3 \cap k_4)_2$
= 4 x C(5,2) - C(2,2) - C(1,2) - C(1,2) = 40 - 1 - 0 - 0 = 39.

5.1.3 Bidirectional Aliasing Probability

Based on the above discussions, the aliasing probability for bidirectional errors of a code sequence can be represented by

$$p^b = \frac{\text{Total number of undetectable bidirectional error cases}}{\text{Total number of ways of picking two bits at a time}}.$$

Example 3. Consider Example 1 again. We have the total number of bidirectional aliasing cases = 4, while the total number of two-bit error cases = 52. Therefore, the aliasing probability is $p^b = \frac{4}{52} = 0.08$.

5.1.4 Results

The bidirectional aliasing probabilities for a few minimum cost code sequences, corresponding to different m-out-of-n codes and weight sets, are presented in Tables 5.2, 5.3, 5.4 and 5.5. The first column in the tables shows the m and n values for each particular m-out-of-n code. The second column shows the sum of all undetectable bidirectional errors in each codeword that exist for the given m-out-of-n code before and after performing the *Drop()* function. For example, the sum of all undetectable bidirectional errors for the 3-out-of-8 code of Example 1 is 18 (14) before (after) *Drop()* function. The third column depicts the total number of errors detected by LSC's or RSC's (i.e., Cases 1 and 2). The fourth column shows the total number of errors detected by LDC's or RDC's (i.e., Cases 3 and 4). The fifth column shows the total number of errors that are undetectable because none of the neighboring cones contains them (Case 6). The sixth column shows the total number of errors that are contained in other cones but are still undetectable (Case 5). The seventh column contains the total number of bidirectional errors that are finally undetectable. It is obvious that this column is the summation of columns 5 and 6. The eighth column shows the total number of two-bit errors for each m-out-of-n code sequence. This value is calculated using equation 5.1. From Table 5.6, it is evident that a large proportion of bidirectional errors are detectable by LSC's and RSC's. Therefore, significant amount of computation can be

saved by eliminating the computation of errors that are detectable by LDC's and RDC's. Note that we do not calculate the total number of two-bit errors for those m-out-of-n codes which has the number of undetected errors equal 0. The reason is that the aliasing probability p_b each of those codes is 0 anyways. From Tables 5.2 to 5.5, it can be observed that the aliasing probabilities of most m-out-of-n codes are 0, and this demonstrates that the proposed method is very powerful in identifying erroneous scan chains.

Table 5.2: Results for weight set {1,2}.

m/n	Errors (before/after dropping)	Case 1 & Case 2	Case 3 & Case 4	Case 6	Case 5	Undetected Errors	Total number of two-bit errors	p^b
1/6	6/6	5	1	0	0	0	-	0
2/6	20/20	20	0	0	0	0	-	0
3/6	18/18	18	0	0	0	0	-	0
1/7	4/4	4	0	0	0	0	-	0
2/7	4/4	4	0	0	0	0	-	0
3/7	14/14	14	0	0	0	0	-	0
1/8	2/2	1	1	0	0	0	-	0
2/8	6/4	4	0	0	0	0	-	0
3/8	18/14	12	1	1	0	1	35	0.03
4/8	12/8	8	0	0	0	0	-	0
1/9	0/0	0	0	0	0	0	-	0
2/9	12/6	5	1	0	0	0	-	0
3/9	12/12	10	2	0	0	0	-	0
4/9	24/20	18	2	0	0	0	-	0

5.2 Multidirectional Aliasing Analysis

In this analysis, first, we assume that more than two bits in a codeword can be changed due to an error. For example, given a weight sequence 1-1-1-1-2, 10001 is a codeword for the 3-out-of-6 code with cone size 5. However, if an error changes three bits such that 10001 is transferred to 11100, then this error cannot be detected. The following condition is *necessary and sufficient* for the occurrence of a multi-directional error

$$\sum_{r=1}^{p} w_r(-1)^{v_r} = 0 \tag{5.5}$$

where v_r is the value of the r^{th} erroneous bit, w_r is the weight of the r^{th} erroneous bit, and p (the total number of erroneous bits) ≥ 3. In the above 3-out-of-6 code example, the weight of each erroneous bit is $w_1 = 1, w_2 = 1$, and $w_3 = 2$, and their values are $v_1 = 0, v_2 = 0$, and $v_3 = 1$, respectively. Therefore, the L.H.S of equation 5.5 is $w_1(-1)^{v_1} + w_2(-1)^{v_2} + w_3(-1)^{v_3} = 1 + 1 - 2 = 0$. So, the error is masked.

Before proceeding with the multi-directional aliasing analysis, we redefine LSC and RSC and add three more terms as follows

1. **Current multiple error cone (CMC)** This is the codeword which is under consideration

Table 5.3: Results for weight set {2,3}.

m/n	Errors (before/after dropping)	Case 1 & Case 2	Case 3 & Case 4	Case 6	Case 5	Undetected Errors	Total number of two-bit errors	p^b
2/11	6/6	5	1	0	0	0	-	0
3/11	0/0	0	0	0	0	0	-	0
4/11	20/20	18	2	0	0	0	-	0
5/11	12/12	10	2	0	0	0	-	0
2/12	4/4	4	0	0	0	0	-	0
3/12	2/2	1	1	0	0	0	-	0
4/12	4/4	4	0	0	0	0	-	0
5/12	18/14	12	1	0	1	1	35	0.03
6/12	0/0	0	0	0	0	0	-	0
2/13	2/2	1	1	0	0	0	-	0
3/13	4/4	4	0	0	0	0	-	0
4/13	0/0	0	0	0	0	0	-	0
5/13	18/14	12	1	0	1	1	35	0.03
6/13	6/4	4	0	0	0	0	-	0
2/14	0/0	0	0	0	0	0	-	0
3/14	12/6	5	1	0	0	0	-	0
5/14	12/12	10	2	0	0	0	-	0
6/14	24/20	18	2	0	0	0	-	0

for multi-directional error analysis, and contains more than two erroneous bits. All multi-directional aliasing cases will be analyzed based on the CMC.

2. **Left singular error cone (LSC)** This is the codeword which is obtained from the code sequence by moving backwards from the CMC, and contains the leftmost erroneous bit of the CMC.

3. **Right singular error cone (RSC)** This is the codeword which is obtained from the code sequence by moving forwards from the CMC, and contains the rightmost erroneous bit of the CMC.

4. **Left multiple error cone (LMC)** This is the codeword which is obtained from the code sequence by moving backwards from the CMC, and contains two or more consecutive erroneous bits.

5. **Right multiple error cone (RMC)** This is the codeword which is obtained from the code sequence by moving forwards from the CMC, and contains two or more consecutive erroneous bits.

Consider the 2-out-of-6 code with weight set = {1,2} and weight sequence = 1-1-1-2-1. The code sequence is given by 11000 → 10001 → 00010 → 00101 → 10100 → 01001 → 01100. The entire sequence is depicted in Fig. 5.2. Let C1 be the CMC. A multi-directional error at locations shown by x1, x2 and x3 would change codeword 11000 to 00010 (C3). For this error, C6 is a LSC because it is at the left side of the CMC and contains bit x1. C7 is a LMC because it is at the left side of the CMC and contains bits x1, x2 and x3. On the other hand, C4 and C3 are RSCs because

Table 5.4: Results for weight set {1,2,3}.

m/n	Errors (before/after dropping)	Case 1 & Case 2	Case 3 & Case 4	Case 6	Case 5	Undetected Errors	Total number of two-bit errors	p^b
1/8	6/4	4	0	0	0	0	-	0
2/8	6/4	4	0	0	0	0	-	0
3/8	6/6	5	1	0	0	0	-	0
4/8	12/12	11	1	0	0	0	-	0
1/9	2/2	1	1	0	0	0	-	0
2/9	2/2	2	0	0	0	0	-	0
3/9	8/8	8	0	0	0	0	-	0
4/9	4/2	2	0	0	0	0	-	0
1/10	0/0	0	0	0	0	0	-	0
2/10	6/4	4	0	0	0	0	-	0
3/10	6/6	6	0	0	0	0	-	0
4/10	6/4	4	0	0	0	0	-	0
5/10	12/12	11	1	0	0	0	-	0
1/10	2/2	1	1	0	0	0	-	0
2/10	0/0	0	0	0	0	0	-	0
3/10	4/2	2	0	0	0	0	-	0
4/10	8/8	8	0	0	0	0	-	0
5/10	4/4	4	0	0	0	0	-	0
1/11	0/0	0	0	0	0	0	-	0
2/11	2/2	1	1	0	0	0	-	0
3/11	4/2	2	0	0	0	0	-	0
4/11	2/2	1	1	0	0	0	-	0
5/11	8/8	8	0	0	0	0	-	0
1/12	0/0	0	0	0	0	0	-	0
2/12	0/0	0	0	0	0	0	-	0
3/12	6/4	4	0	0	0	0	-	0
4/12	6/6	5	1	0	0	0	-	0
5/12	6/6	5	1	0	0	0	-	0
6/12	12/8	8	0	0	0	0	-	0

they are at the right side of the CMC and contains bit x3. C2 is RMC because it is at the right side of the CMC and contains bits x2 and x3.

Similar to bidirectional aliasing analysis, for multi-directional errors, six possible cases can arise which are described below.

Case 1. For an undetected multi-directional error in the CMC, there exists one or more LSC. As will be proved in Theorem 6, this error is always detected.

Case 2. For an undetected multi-directional error in the CMC, there exists one or more RSC. Again, it will be proved in Theorem 6, that this error is always detected.

Case 3. For an undetected multi-directional error in the CMC, there exists one or more LMC. The error can be detected in this case if the weights of all erroneous bits are such that equation 5.5

Table 5.5: Results for weight set {1,2,3,5}.

m/n	Errors (before/after dropping)	Case 1 & Case 2	Case 3 & Case 4	Case 6	Case 5	Undetected Errors	Total number of two-bit errors	p^b
1/12	2/2	1	1	0	0	0	-	0
2/12	0/0	0	0	0	0	0	-	0
3/12	2/2	2	0	0	0	0	-	0
4/12	2/2	2	0	0	0	0	-	0
5/12	0/0	0	0	0	0	0	-	0
6/12	4/4	4	0	0	0	0	-	0
1/13	0/0	0	0	0	0	0	-	0
2/13	2/2	1	1	0	0	0	-	0
3/13	2/2	1	1	0	0	0	-	0
4/13	0/0	0	0	0	0	0	-	0
5/13	2/2	2	0	0	0	0	-	0
6/13	2/2	2	0	0	0	0	-	0
1/14	0/0	0	0	0	0	0	-	0
2/14	0/0	0	0	0	0	0	-	0
3/14	2/2	2	0	0	0	0	-	0
4/14	2/2	2	0	0	0	0	-	0
5/14	2/2	2	0	0	0	0	-	0
6/14	2/2	2	0	0	0	0	-	0

Table 5.6: Percentage of bidirectional errors detected by Cases 1 and 2.

Weight set	% errors detected by Cases 1 and 2	Weight set	% errors detected by Cases 1 and 2
{1,2}	93	{2,3}	87
{1,2,3}	92	{1,2,3,5}	88

is not satisfied (refer to Theorem 7).

Case 4. For an undetected multi-directional error in the CMC, there exists one or more RMC. Again the error can be detected in this case if the weights of all erroneous bits are such that equation 5.5 is not satisfied (refer to Theorem 7).

Case 5. For an undetected multi-directional error in the CMC, there exists one or more LMC and/or RMC but the error still cannot be detected.

Case 6. For an undetected multi-directional error in the CMC, there is no LSC, RSC, LMC or RMC. The error in the CMC has no chance of being detected as it is not contained by neighboring cones.

In Theorem 6, we prove that a multi-directional error of the CMC will be detected by a LSC or RSC, if it exists. This theorem will, therefore, cover Cases 1 and 2 discussed above.

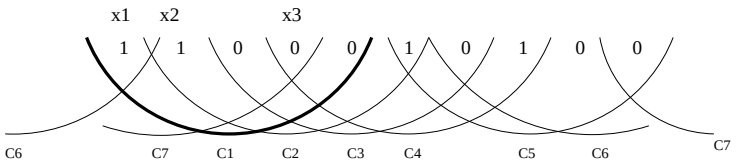


Figure 5.15: 2-out-of-6 code sequence.

Theorem 6. An undetected multi-directional error of the CMC will be detected by another cone (codeword), if one of the corner bits involved in the error is contained by a LSC or RSC.
Proof: Let us assume that four bits, namely i, j, p and q are involved in a multi-directional error in cone c_a as shown in Fig. 5.16. Now, if a LSC contains bit i, then the error will be detected for the reason similar to that of Theorem 1. Similarly, if bit q is covered by a RSC, the error will be detected. **Q.E.D**

In Theorem 7, we prove that a multi-directional error of the CMC can be detected by a LMC or RMC, if it exists. The detection of the error depends on the weights of the erroneous bits at the LMC or RMC. This theorem covers Cases 3 and 4 as explained previously.

Theorem 7. An undetected multi-directional error of the CMC can be detected by another cone (codeword), if two or more bits involved in the error are contained by LMCs or RMCs and those covered bits in each LMC/RMC do not satisfy equation 5.5 in the new cone.
Proof: Referring to Fig. 5.16 again. Let us assume that a neighboring cone (LMC or RMC) covers more than one erroneous bits. Now, if equation 5.5 is not satisfied for the erroneous bits in the new cone, the resulting positive or negative imbalance in weight will cause a change in the overall weight of the new cone and the error will be detected. **Q.E.D**

In the following theorem, we would prove that a multi-directional error cannot be detected if it is not contained by any cone or it is contained by neighboring cones (LMC or RMC) where the weights of the erroneous bits in each new cone still satisfy equation 5.5. This situation is covered by Cases 5 and 6 respectively.

Theorem 8. An undetected multi-directional error of the CMC cannot be detected, if it is not contained in any other cone or two or more bits involved in the error are contained by LMCs or RMCs, but those covered bits satisfy equation 5.5 in each new cone.
Proof: The first part of this theorem is very similar to Theorem 3 and can be proved accordingly. Now, if the erroneous bits, covered by each LMC or RMC still satisfy equation 5.5, then the total change in weight of the LMC or RMC is zero, which means that the error is undetectable. **Q.E.D**

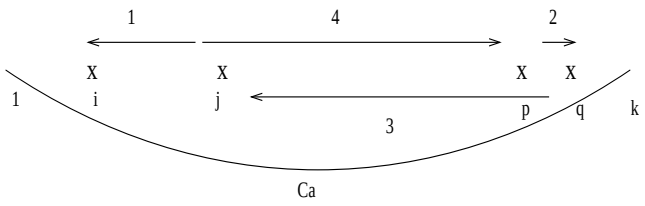


Figure 5.16: Different regions for searching a multi-directional error.

5.2.1 Total Number of Undetected Multi-directional Errors

Based on Theorems 6, 7 and 8, we present an algorithm to calculate the multi-directional aliasing probabilities. Again, there are total four search regions for each error in the CMC as illustrated in Fig. 5.16. Region 1 (region 2) is searched for a LSC (RSC) while region 3 (region 4) is searched for LMCs (RMCs). Here, k is the cone size and i, j, p and q are the bit numbers which are susceptible to an error. If a LSC or RSC exists for a multi-directional error in the CMC, then the error is immediately detected. Otherwise, the algorithm checks if a LMCs or RMCs exist for the error and that if the error can be detected by them. If not, the error is undetectable and counts as multi-directional aliasing. In the following paragraphs, we present an example to illustrate various steps of the algorithm.

Example 4. Let us consider an un-optimized codeword ordering of a 3-out-of-8 code sequence with weight set {1,2} and cone size of 5 to illustrate the operation of algorithm *MultidirectionalDetect ()*. The codewords, bit numbers, and the weight distribution are illustrated in Fig. 5.9. The corresponding graph G is shown in Fig. 5.10. The error list generated by *GenerateErrorList()* is as follows:

C1 - (0,1,2,3), (0,1,2,4)
C2 - (0,1,2,3), (0,1,2,4)
C3 - (0,1,2,3), (0,1,3,4)
C4 - (0,1,2,3), (0,1,3,4)
C5 - (0,1,2,4), (0,1,3,4)
C6 - (0,1,2,4), (0,1,3,4)

The total number of errors before dropping = 12, and the total number of errors after dropping = 12. Again, we emphasize that the errors in each cone is undetected by the CMC.

Table 5.7: Summary of multi-directional aliasing example.

codeword	error	detected	case
C1	(0,1,2,3)	yes	1
	(0,1,2,4)	yes	1
C2	(0,1,2,3)	yes	1
	(0,1,2,4)	yes	1
C3	(0,1,2,3)	yes	3
	(0,1,3,4)	yes	4
C4	(0,1,2,3)	yes	1
	(0,1,3,4)	yes	1
C5	(0,1,2,4)	yes	2
	(0,1,3,4)	no	5
C6	(0,1,2,4)	yes	2
	(0,1,3,4)	yes	2

Each of the errors from the reduced error list for each codeword is analyzed to check if it can be detected by a neighboring LSC, RSC, LMC or RMC. For example, let C1 be the CMC. We try to analyze error (0,1,2,3) from the reduced error list. Using the graph in Fig. 5.10 to travel

backwards, we can observe that C6 is the LSC which is 4 shifts away to the left side of C1 and contains the 0^{th} erroneous bit. This is clearly illustrated in Fig. 5.11. Therefore, this error is detectable. Further, let us consider error (0,1,2,3) in CMC C3. Using Fig. 5.10, we go backwards to see if any LSC exists. Since no LSC covers error on 0^{th} bit, we travel forwards in graph G to find if any RSC exists. From Fig. 5.11, it is obvious that no RSC covers error at the 3^{rd} bit alone. Next, by moving backwards in G, we find a LMC C2 which covers error bits (0,1,2) of C3 with the weights of erroneous bits (2,2,2) at C2. Hence, the total change of weight in C2 due to error (0,1,2) at C3 is -2+2+2 = 2 which is non-zero. Therefore, the error will be detected. From Table 5.7, it is evident that out of 12 reduced errors, a total of 11 errors are detected. Hence, the total number of errors that are masked is 1 only.

Based on the above example, we now present algorithm *MultidirectionalDetect ()* to analyze the number of aliasing cases. This algorithm is very similar to algorithm *BidirectionalDetect ()*. The function *GenerateErrorList()* picks one cone at a time, and finds out potential locations that are susceptible to a multi-directional error in the cone. All such locations must satisfy equation 5.5. The function *Drop()* picks up one cone at a time and, for each of its errors, checks if the error will be analyzed by the succeeding cone or not. This function is same as *Drop()* that is presented in algorithm *BidirectionalDetect ()* for eliminating the possibility of over-counting.

Once the reduced error list is available, the next task is to check if each error can be detected by any of its neighboring cones. Since Cases 1 and 2 are computationally least expensive, the algorithm again starts with checking LSC or RSC for an error in the CMC. If such a cone exists, then the error is detectable and the next error is picked for analysis. Otherwise, the algorithm proceeds with checking if a LMC exist for the error in the CMC. If so, the new weights of all erroneous bits covered by the LMC are checked. If the new weights do not satisfy equation 5.5, then the error is detectable in the LMC. Otherwise, the error is undetectable and the algorithm looks for the next LMC. If there is no more LMC, then the algorithm searches for a RMC and the above procedure is repeated. If no LMC or RMC exists for the given error in the CMC, then the error is undetectable (Case 5). If the error has one or more LMC or RMC but the new weights of each LMC/RMC satisfies equation 5.5, then it is still undetectable (case 6). Cases 1 through 6 are shown in algorithm *MultidirectionalDetect ()* below.

Algorithm MultidirectionalDetect ()
begin
Input: A cyclic graph G where nodes are the cone numbers and edge represents shifts;
Output: Number of multi-directional aliasing cases;
GenerateErrorList();
Drop();
for (each error from the reduced error list)
do
    begin
        move backwards in region 1 of graph G; //see Fig. 5.8
        if (cone exist)
            then print "multi-directional error is detectable" and exit; // Case-1
        move forwards in region 2 of graph G;
        if (cone exist)

```
        then print "multi-directional error is detectable" and exit;// Case-2
    move backwards in region 3 of graph G;
    if (no cone found)
        then no_overlap_flag1 = 1;
    else
        for (each cone found above)
        do
            begin
                w_r = weight[r + shift], ∀ r = erroneous bits covered by new cone;
                if (∑_r w_r x (−1)^{v_r} == 0)
                    then print "multi-directional error is detectable" and exit; // Case-3
            end
    move forwards in region 4 of graph G;
    if (no cone found)
        then no_overlap_flag2 = 1;
    else
        for (each cone found above)
        do
            begin
                w_r = weight[r - shift], ∀ r = erroneous bits covered by new cone;
                if (∑_r w_r x (−1)^{v_r} == 0)
                    then print "multi-directional error is detectable" and exit; // Case-4
            end
    if (no_overlap_flag1 == 1 and no_overlap_flag2 == 1)
            then print "no cone overlaps these affected bits" and
                increment multidirectional_count; // Case-6
    else
            print "more than one cone overlaps these affected bits" and
            increment multidirectional_count;// Case-5
  end
end
```

5.2.2 Total Number of Multi-Bit Errors

The algorithm *MultidirectionalDetect ()* enumerates the number of multi-directional aliasing cases. To find the multi-directional probability, we must calculate the number of all possible multi-bit errors in the particular m-out-of-n code sequence. In the case of non-overlapped codeword applications, the number of all possible multi-bit errors is Q x [C(l,3) + C(l,4) + ...+ C(l,l)] where Q is the number of codewords and l is the cone size. However, this is not true in the case of overlapped code sequence because some of the multi-bit errors are counted several times due to overlap of codewords. We must, therefore, use the inclusion-exclusion principle to eliminate duplicated cases and derive the exact number of multi-bit error cases.

For a given m-out-of-n code, assume there are Q codewords (k_1, k_2, ..., k_Q) which are optimally ordered for maximum overlapping. Using the inclusion-exclusion principle, the total number to ways to pick multiple bits for multi-directional errors is given by

$$N(k_1 \cup k_2 \cup ... \cup k_Q)_3 + N(k_1 \cup k_2 \cup ... \cup k_Q)_4 + ... N(k_1 \cup k_2 \cup ... \cup k_Q)_j + ... + N(k_1 \cup k_2 \cup ... \cup k_Q)_k \quad (5.6)$$

where $N(k_1 \cup k_2 \cup ... \cup k_Q)_j$ stands for the number of ways in which j bits at a time can be erroneous in $k_1 \cup k_2 \cup ... \cup k_Q$. Note that $k_1 \cup k_2 \cup ... \cup k_Q$ is in fact the entire overlapped codeword sequence. Thus, we have

$N(k_1 \cup k_2 \cup ... \cup k_Q)_j = S_{1j} - S_{2j} + ... + (-1)^{i+1} S_{ij} + ... + (-1)^{Q+1} S_{Qj}$
where

S_{1j}= N$(k_1)_j$ + N$(k_2)_j$ + ... + N$(k_Q)_j$
= $\binom{k}{j}$ + $\binom{k}{j}$ + ... + $\binom{k}{j}$
= Q x $\binom{k}{j}$,
S_{2j}= N$(k_1 \cap k_2)_j$ + N$(k_1 \cap k_3)_j$ + ... + N$(k_{Q-1} \cap k_Q)_j$
= $\binom{k_1 \cap k_2}{j}$ + $\binom{k_1 \cap k_3}{j}$ + .. + $\binom{k_{Q-1} \cap k_Q}{j}$,
.
.
.
S_{Qj}= N$(k_1 \cap k_2 \cap ... \cap k_Q)_j$
= $\binom{k_1 \cap k_2 \cap ... \cap k_Q}{j}$.

Hence, total number of ways of picking multiple (≥ 3) bits equals

$$\sum_{j=3}^{k} S_{1j} - \sum_{j=3}^{k} S_{2j} + ... + (-1)^{i+1} \sum_{j=3}^{k} S_{ij} + ... + (-1)^{Q+1} \sum_{j=3}^{k} S_{Qj} \quad (5.7)$$

It is evident from equation 5.7 that the total number of terms is again very large. However, as proved in Theorem 5, most of the terms can never exist in a given m-out-of-n code, if they involve cones which are more than l (i.e., cone size) shifts away from each other. Further, the equation can be simplified as all terms containing intersection of three or more cones can be reduced to terms containing intersection of two cones (Theorem 4). Hence, many terms cancel out among themselves. Using Theorems 4 and 5, it is possible to simplify the calculation of multi-bit error cases, as most of the terms will either cancel out or turn out to be zero. Let us solve Example 2 again for multi-bit error enumeration.

Example 5. Consider the 5-out-of-12 code with weight set {1,2,5} and weight distribution of 2-1-2-2-5. The code sequence is shown in Fig. 5.14. Once again, we have

N$(k_1)_3$ = N$(k_2)_3$ = N$(k_3)_3$ = N$(k_4)_3$ = C(5,3) = 10,
N$(k_1)_4$ = N$(k_2)_4$ = N$(k_3)_4$ = N$(k_4)_4$ = C(5,4) = 5,
N$(k_1)_5$ = N$(k_2)_5$ = N$(k_3)_5$ = N$(k_4)_5$ = C(5,5) = 1,
$(k_1 \cap k_2) = 2$,
$(k_1 \cap k_3) = 0$ (from Theorem 5),
$(k_1 \cap k_4) = 0$ (from Theorem 5),
$(k_2 \cap k_3) = 1$,
$(k_2 \cap k_4) = 0$ (from Theorem 5),

$(k_3 \cap k_4) = 1$,
$(k_1 \cap k_2 \cap k_3) = (k_1 \cap k_3) = 0$ (from Theorem 4),
$(k_1 \cap k_2 \cap k_4) = (k_1 \cap k_4) = 0$ (from Theorem 4),
$(k_1 \cap k_3 \cap k_4) = (k_1 \cap k_4) = 0$ (from Theorem 4),
$(k_2 \cap k_3 \cap k_4) = (k_2 \cap k_4) = 0$ (from Theorem 4),
$(k_1 \cap k_2 \cap k_3 \cap k_4) = (k_1 \cap k_4) = 0$ (from Theorem 4).

From equations 5.6 and 5.7, we have
total number of multi-directional cases

$= \mathrm{N}(k_1 \cup k_2 \cup ... \cup k_4)_3 + \mathrm{N}(k_1 \cup k_2 \cup ... \cup k_4)_4 + \mathrm{N}(k_1 \cup k_2 \cup ... \cup k_4)_5$
$= \sum_{j=3}^{k} S_{1j} - \sum_{j=3}^{k} S_{2j} + \sum_{j=3}^{k} S_{3j} - \sum_{j=3}^{k} S_{4j}$
$= \sum_{j=3}^{5}[\mathrm{N}(k_1)_j + \mathrm{N}(k_2)_j + \mathrm{N}(k_3)_j + \mathrm{N}(k_4)_j]$ -$[\mathrm{N}(k_1 \cap k_2)_3 + \mathrm{N}(k_1 \cap k_2)_4 + \mathrm{N}(k_1 \cap k_2)_5]$ -$[\mathrm{N}(k_2 \cap k_3)_3$
$+ \mathrm{N}(k_2 \cap k_3)_4 + \mathrm{N}(k_2 \cap k_3)_5]$ -$[\mathrm{N}(k_3 \cap k_4)_3 + \mathrm{N}(k_3 \cap k_4)_4 + \mathrm{N}(k_3 \cap k_4)_5]$
= [4 C(5,3) + 4 C(5,4) + 4 C(5,5)] - [C(2,3) + C(2,4) + C(2,5)] - 2 [C(1,3) + C(1,4) + C(1,5)]
= 4 x[10 + 5 + 1] - 0 - 0
= 64.

5.2.3 Multi-directional Aliasing Probability

Based on the above discussions, the aliasing probability for multi-directional errors of a code sequence can be represented by

$$p^m = \frac{\text{Total number of undetectable multi-directional error cases}}{\text{Total number of ways of picking multiple } (\geq 3) \text{ bits at a time}}.$$

Example 6. Let us consider Example 4 again. We have the total number of multi-directional aliasing cases = 1, while the total number of multi-bit error cases = 94. Therefore, the aliasing probability for multi-directional errors is $p^m = \frac{1}{94} = 0.01$.

5.2.4 Results

The multi-directional aliasing probabilities for the minimum-cost codeword sequences corresponding to different m-out-of-n codes and weight sets are presented in Tables 5.8, 5.9, 5.10 and 5.11. Note that, the first column in the tables shows the m and n values for each particular m-out-of-n code. The second column shows the sum of all undetectable multi-directional errors in each codeword that exist for the given m-out-of-n code, before and after performing the *Drop()* function. For example, the sum of all undetectable multi-directional errors for the 3-out-of-8 code in Example 4 is 12 (12) before (after) the *Drop()* function. The third column gives the total number of errors detected by LSC's and RSC's (i.e., Cases 1 and 2). The fourth column shows the total number of errors detected by LMC's and RMC's (i.e., Cases 3 and 4). The fifth column shows the total number of errors that are undetectable, because none of the neighboring cones contains them (Case 6). The sixth column shows the total number of errors that are contained in other cones, but still cannot be detected (Case 5). The seventh column contains the total number of multi-directional errors that are finally undetectable. It is obvious that this column is the summation of columns 5 and 6. The eighth column shows the total number of multi-bit errors for each given m-out-of-n code sequence. This value is calculated using equation 5.6. It is surprising that the aliasing probability

of multi-directional errors in each m-out-of-n code is zero. This demonstrate that the proposed method is ideal and powerful. The power of the overlapped test application method comes from that most undetected multi-directional errors in the CMC can be detected by its neighboring LSC or RSC as unidirectional errors. This is shown in Table 5.12. Therefore, significant amount of computing time can be saved since it is extremely easy to check Cases 1 and 2.

Table 5.8: Results for weight set {1,2}.

m/n	Errors (before/after dropping)	Case 1 & Case 2	Case 3 & Case 4	Case 6	Case 5	Undetected Errors	Total number of multi-bit errors	p^m
1/6	0/0	0	0	0	0	0	-	0
2/6	18/18	18	0	0	0	0	-	0
3/6	32/32	30	2	0	0	0	-	0
1/7	0/0	0	0	0	0	0	-	0
2/7	12/11	11	0	0	0	0	-	0
3/7	24/22	22	0	0	0	0	-	0
1/8	0/0	0	0	0	0	0	-	0
2/8	6/6	6	0	0	0	0	-	0
3/8	12/10	10	0	0	0	0	-	0
4/8	18/18	16	2	0	0	0	-	0
1/9	0/0	0	0	0	0	0	-	0
2/9	0/0	0	0	0	0	0	-	0
3/9	0/0	0	0	0	0	0	-	0
4/9	6/6	6	0	0	0	0	-	0

Table 5.9: Results for weight set {2,3}.

m/n	Errors (before/after dropping)	Case 1 & Case 2	Case 3 & Case 4	Case 6	Case 5	Undetected Errors	Total number of multi-bit errors	p^m
2/11	0/0	0	0	0	0	0	-	0
3/11	0/0	0	0	0	0	0	-	0
4/11	6/6	6	0	0	0	0	-	0
5/11	0/0	0	0	0	0	0	-	0
2/12	0/0	0	0	0	0	0	-	0
3/12	0/0	0	0	0	0	0	-	0
4/12	0/0	0	0	0	0	0	-	0
5/12	12/10	10	0	0	0	0	-	0
6/12	2/2	2	0	0	0	0	-	0
2/13	0/0	0	0	0	0	0	-	0
3/13	0/0	0	0	0	0	0	-	0
4/13	0/0	0	0	0	0	0	-	0
5/13	12/10	10	0	0	0	0	-	0
6/13	0/0	0	0	0	0	0	-	0
2/14	0/0	0	0	0	0	0	-	0
3/14	0/0	0	0	0	0	0	-	0
5/14	0/0	0	0	0	0	0	-	0
6/14	6/6	6	0	0	0	0	-	0

Table 5.10: Results for weight set {1,2,3}.

m/n	Errors (before/after dropping)	Case 1 & Case 2	Case 3 & Case 4	Case 6	Case 5	Undetected Errors	Total number of multi-bit errors	p^m
1/8	0/0	0	0	0	0	0	-	0
2/8	6/6	5	1	0	0	0	-	0
3/8	15/15	13	2	0	0	0	-	0
4/8	18/18	16	2	0	0	0	-	0
1/9	0/0	0	0	0	0	0	-	0
2/9	4/4	4	0	0	0	0	-	0
3/9	12/12	10	2	0	0	0	-	0
4/9	16/16	15	1	0	0	0	-	0
1/10	0/0	0	0	0	0	0	-	0
2/10	0/0	0	0	0	0	0	-	0
3/10	6/6	6	0	0	0	0	-	0
4/10	6/4	3	1	0	0	0	-	0
5/10	18/18	16	2	0	0	0	-	0
1/10	0/0	0	0	0	0	0	-	0
2/10	2/2	2	0	0	0	0	-	0
3/10	8/8	8	0	0	0	0	-	0
4/10	12/12	12	0	0	0	0	-	0
5/10	8/8	8	0	0	0	0	-	0
1/11	0/0	0	0	0	0	0	-	0
2/11	0/0	0	0	0	0	0	-	0
3/11	8/8	8	0	0	0	0	-	0
4/11	4/4	4	0	0	0	0	-	0
5/11	12/12	12	0	0	0	0	-	0
1/12	0/0	0	0	0	0	0	-	0
2/12	0/0	0	0	0	0	0	-	0
3/12	6/6	6	0	0	0	0	-	0
4/12	0/0	0	0	0	0	0	-	0
5/12	0/0	0	0	0	0	0	-	0
6/12	18/18	16	2	0	0	0	-	0

Table 5.11: Results for weight set {1,2,3,5}.

m/n	Errors (before/after dropping)	Case 1 & Case 2	Case 3 & Case 4	Case 6	Case 5	Undetected Errors	Total number of multi-bit errors	p^m
1/12	0/0	0	0	0	0	0	-	0
2/12	2/2	2	0	0	0	0	-	0
3/12	4/4	4	0	0	0	0	-	0
4/12	4/4	2	2	0	0	0	-	0
5/12	6/6	4	2	0	0	0	-	0
6/12	8/8	8	0	0	0	0	-	0
1/13	0/0	0	0	0	0	0	-	0
2/13	0/0	0	0	0	0	0	-	0
3/13	4/4	2	2	0	0	0	-	0
4/13	2/2	2	0	0	0	0	-	0
5/13	10/10	7	3	0	0	0	-	0
6/13	4/4	2	2	0	0	0	-	0
1/14	0/0	0	0	0	0	0	-	0
2/14	0/0	0	0	0	0	0	-	0
3/14	4/4	4	0	0	0	0	-	0
4/14	0/0	0	0	0	0	0	-	0
5/14	4/4	3	1	0	0	0	-	0
6/14	10/10	7	3	0	0	0	-	0

Table 5.12: Percentage of multi-directional errors detected by Cases 1 and 2.

Weight set	% errors detected by Cases 1 and 2	Weight set	% errors detected by Cases 1 and 2
{1,2}	97	{2,3}	100
{1,2,3}	93	{1,2,3,5}	76

Chapter 6

Multiple Weight-Based Codes and Checker Design

Sometimes, a single set of m-out-of-n codewords applied to the scan chain may not give enough perturbation to the logic circuits. In this case, we can increase the number of perturbations by applying more than one set of codewords. Thus, the codeword generation process presented in chapter 3 can be repeated several times to obtain multiple sets of codewords. Different sets of codewords cannot be applied in a mixed manner. The reason is that different sets of codewords may have different weight configurations and m values, and the checker must be restructured to monitor each different set of codewords. There are two major problems for the application of multiple weight-based m-out-of-n codes: (1) design of checkers such that they are totally self-checking, and (2) aliasing probability analysis when many sets of codewords are applied. In this chapter, we discuss both problems in detail.

6.1 Multiple Weight-Based Code Sequences

Let us assume that we have two sets of m-out-of-n codes that are shifted into the scan chain one after the other, and the combined sequence is

$$A_1 A_2 ... A_{Q_A} B_1 B_2 ... B_{Q_B}$$

where A_i's and B_i's are codewords corresponding to two sets of m-out-of-n codes, A and B, differing in the m value and weight distribution only. Further, Q_A and Q_B are the numbers of codewords in codes A and B, respectively. The aliasing probability analysis for two or more sets of codewords will be very tedious, if an aliasing codeword (i.e., erroneous and undetectable) for A turns out to be a codeword of B. First, we will show that this is impossible for bidirectional errors. Consider any particular codeword A_i in sequence A, and the total numbers of zero's and one's with the same weight w in codeword A_i are u and v separately. It can be easily observed that an undetectable bidirectional error occurring to A_i, caused by one bit of opposite transition in the u and v bits, will form a valid codeword of sequence A. Since the m value of the erroneous codeword A_i is not changed, it is impossible that A_i be one of the codewords of sequence B (which has a different m value).

The above reasoning concludes that aliasing of codewords in sequence A (B) can only occur among codewords of A (B), and there is no case of inter-code aliasing. This observation is inde-

pendent of the number of different sets of m-out-of-n codes that are combined and of the ordering among them. Based on such an observation, as shown in Chapter 5, the total number of bidirectional aliasing cases will be given as

$$(\text{number of undetectable bidirectional errors})_A + (\text{number of undetectable bidirectional errors})_B. \tag{6.1}$$

The total number of two-bit error cases equals

$$N(k_1 \cup k_2 \cup ... \cup k_{Q_A})_2 + N(k_1 \cup k_2 \cup ... \cup k_{Q_B})_2. \tag{6.2}$$

Hence, the bidirectional aliasing probability of the combined sequence using two m-out-of-n codes equals

$$\frac{(\text{number of undetectable bidirectional errors})_A + (\text{number of undetectable bidirectional errors})_B}{N(k_1 \cup k_2 \cup ... \cup k_{Q_A})_2 + N(k_1 \cup k_2 \cup ... \cup k_{Q_B})_2}.$$

Similarly, for multi-directional aliasing, the total number of cases where multi-directional errors can occur is

$(\text{number of undetectable multi-directional errors})_A$ + $(\text{number of undetectable multi-directional errors})_B$

while the total number of multi-bit error cases equals

$$\sum_{j=3}^{l} N(k_1 \cup k_1 \cup ... \cup k_{Q_A})_j + \sum_{j=3}^{l} N(k_1 \cup k_1 \cup ... \cup k_{Q_B})_j.$$

Hence, the multi-directional aliasing probability of the combined sequence is

$$\frac{(\text{number of undetectable multi-directional errors})_A + (\text{number of undetectable multi-directional errors})_B}{\sum_{j=3}^{l} N(k_1 \cup k_1 \cup ... \cup k_{Q_A})_j + \sum_{j=3}^{l} N(k_1 \cup k_1 \cup ... \cup k_{Q_B})_j}.$$

The same discussion can be applied to the aliasing probability analysis for the case where more than two multiple m-out-of-n codes are merged as the test sequence.

Example 1:
Consider a set of 1-out-of-10 code with weight distribution 2-1-1-3-3, and another set of 2-out-of-10 code with weight distribution 1-3-2-1-3 (refer to Table 3.3). The combined code sequence, where 1-out-of-10 code is followed by 2-out-of-10, is given by:

(01000→00100)→(00100→10010)

The bidirectional aliasing probability of the combined sequence is $\frac{0+0}{(\text{all two-bit error cases})} = 0$. Similarly, the multi-directional aliasing probability of the combined sequence equals $\frac{0+0}{(\text{all multi-bit error cases})} = 0$.

6.2 Checker Implementation

After we generate the codewords that can activate the faults and can capture the errors with the minimum probability of masking, the next important task is to detect the errors with the maximum reliability. The implementation of SCC is crucial as the success of our scan chain testing scheme mainly depends on it. In this section, we present a checker design for multiple weight-based m-out-of-n codes with variable weights. In fact, it is a general design converting the incoming variable weight-based m-out-of-n codes to simple values code which are later fed to a TSC two-rail checker (TRC).

6.2.1 Variable Weight m-out-of-n Checker

The totally self-checking circuit presented in [20] can be easily extended for *multiple* sets of weight-based m-out-of-n codes by providing extra AND gates as shown in Fig. 6.1. The function of AND gates T_1 to T_{10} is to control the connection of codeword lines (e.g., O_0, O_1) to the sum blocks for different sets of weighted m-out-of-n codes. The control lines (e.g., S_{11}, S_{12}) to control the AND gates are provided by the a finite state machine driven by the tester using signals Tck (test clock) and Tcs (test control signal). Note that Tcs is very similar to the TMS signal used in the boundary scan controller IEEE 1149.1 [25]. For example, if the weight distribution is 2-3-2-3-2, then T_3 and T_7 will be transparent and codeword bits input to sum0 are $\{O_1, O_3\}$. Further, T_2, T_4, T_6, T_8 and T_{10} are transparent, so the sum1 block will receive inputs $\{O_0, O_1, O_2, O_3,$ and $O_4\}$. Now, if the weight distribution is changed to 3-2-3-2-2, then T_1 and T_5 must become transparent, so codeword bits input to the sum0 block will be changed to $\{O_0, O_2\}$.

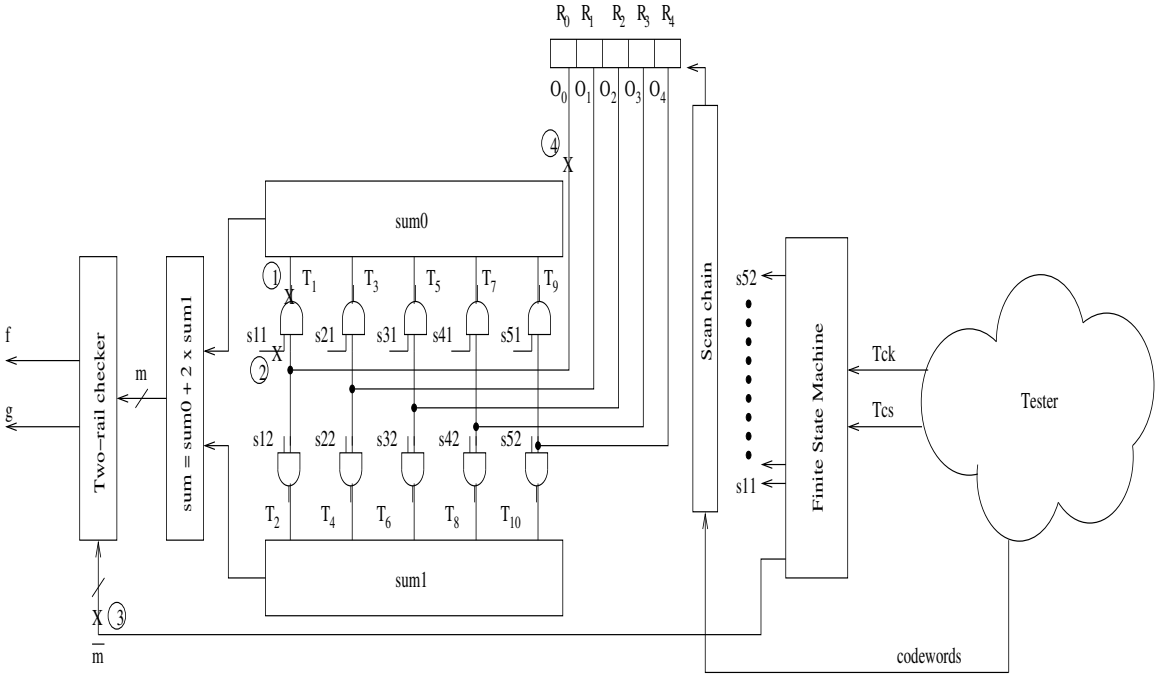


Figure 6.1: Self checking checker for variable weight line

The output of the sum block (with inputs sum0 and sum1) is input to a totally self-checking (TSC) two-rail checker (TRC) which checks the *sum* against the $\overline{m}$ value provided by the finite state machine. A TRC has two groups of inputs $(x_1, x_2, ..., x_n)$ and $(y_1, y_2, ..., y_n)$ and two outputs

f and g. The signals observed on the outputs must always be complementary, that is, a 1-out-of-2 code if and only if every pair (x_j, y_j) is also complementary for all j $(1 \leq j \leq n)$ [15]. Therefore, the finite state machine provides the complementary value of the expected m to the TRC as one set of inputs. The other set of inputs come from *sum* block. In the following discussions, we will show that inserting the AND gates to the sum0 and sum1 blocks does not change the TSC property of our design. The TSC property of the *sum* blocks and TRC checker can be found in [20] and [15] respectively. We make the following assumptions generally considered for self-checking circuits: (a) Errors occur one at a time and, and (b) The time elapsing between two consecutive errors is long enough to allow the application of all possible input codewords. The analysis is done based on the AND gates added as follows.

1. Stuck-at faults at the data input or output line of an AND gate: If the output line of a AND gate is Sa0 (Sa1), then the m value calculated by the sum block will be smaller (greater) than the expected m value for some codewords, and the fault is detected. Therefore, the TSC property of the circuit is maintained. For example, if T_1 is transparent and its input is 1 but its output (refer to the circled 1 in Fig. 6.1) is Sa0, then the fault will be detected because the calculated sum will be smaller than the expected m value. Similar arguments can be given for Sa1 at output and Sa1/Sa0 at the input. Thus, the circuit is TSC.

2. Stuck-at faults at the control input line: Suppose the control line to an AND gate is Sa0 or Sa1. The fault will be detected and the TSC property of the circuit is maintained. For example, if the control input S_{11} of T_1 is Sa1 (refer to the circled 2 in Fig. 6.1), then the AND gate will always be transparent. When O_0 is 1 and T_1 output is expected to be 0, the sum will be greater than the expected m value and the error will be detected. Similarly, if control input S_{11} of T_1 is Sa0, then the AND gate output will always be 0. When O_0 is 1 and T_1 is expected to be transparent, the sum will be smaller than the expected m value and the error is detected. Hence, the TSC property is still satisfied.

If there is a Sa0 (Sa1) fault at any bit of expected m provided by the finite state machine (refer to circled 3 in Fig. 6.1), the TRC checker will produce an error since the generated m will be smaller (greater) than the expected m. One underlying assumption in this case is that, there is no simultaneous errors in the expected m and the generated m at the same bit position. If such a situation arises, then the (incorrect) expected m equals the (incorrect) generated m, then the fault can go undetected.

If any of the $O_0, O_1, ..., O_4$ is Sa0 (Sa1) as shown by circled 4 in Fig. 6.1, whereas the original value is 1 (0), the generated m value will be smaller (more) than the expected m value and the error will be detected. For example, consider the 6-out-of-13 code with weight distribution 3-2-3-2-3 again. Suppose O_0 is Sa0, but here O_0 is 1 for codeword 10100. The generated codeword will be 00100 which has $m = 3$. Hence, the error is detected. Based on the above discussion, we are sure that the proposed checker design shown in Fig. 6.1 is TSC.

Chapter 7

Test Architecture

In this Chapter, we will first explain the test pattern alignment problem that arises due to the incorporation of broadcast architecture and checkers used in our scheme. Then, we give a test architecture that can efficiently support the proposed test method. The test architecture (implemented in the SoC circuit) includes a finite state machine (FSM) that can communicate with the external tester well. The number of pins dedicated to communication between the tester and the finite state machine must be as small as possible. We use test pattern broadcasting architecture so that test control signals can be shared by all self-checking checkers. By sharing test control signals, the design-for-testability (DFT) circuits can be greatly simplified, and the control signal routing area can be minimized. The test architecture is based on the assumption that even the shortest scan chain of the entire SoC chip is longer than the cone size of the m-out-of-n codeword sequence used for testing. In Sections 1, we will present the test pattern alignment problem and our solution to it, while in Sections 2, we will present the test architecture.

7.1 Test Pattern Alignment

The test pattern alignment problem is severe in our case because we have checker (s) for each core. Each test pattern (or codeword) must be aligned at the checkers so that a shared control signal can be used for all checkers. Further, the patterns must be aligned at the input because a single input is used to scan-in the test patterns in the broadcast test architecture. Before solving the alignment problem, we will explain the concept of *repeat pattern* by an example. Consider the case of a 2-out-of-7 code with weight set of {1,2} and weight distribution of 1-2-1-2-1. The code sequence for this code is 01000 → 10001 → 00010 → 00101 → 10100. The test application cost of this code is 6, and the total number of bits present in the code sequence is 11 (5 for shifting the first pattern and 6 for shifting the rest of patterns). The code sequence for the 2-out-of-7 code is shown in Fig. 7.1. Note that by repetitively shifting test pattern 101000 into the initial codeword (i.e., 01000), we can obtain the entire sequence of the 2-out-of-7 code. We call this a *repeat pattern* for the given 2-out-of-7 code (Fig. 7.1). By repetitively pumping the repeat pattern α times (after the very first patterns is already shifted), we can obtain the given m-out-of-n code sequence α times. For the example of 2-out-of-7 code, if we shift in the repeat pattern (i.e., 101000) two times, then we obtain the code sequence 01000 through 10100 two times.

In our test scheme, once the first test pattern and the last test pattern are aligned in all cores, we just keep broadcasting the repeat patterns. For example, consider three cores IP1, IP2

and IP3 with scan chain lengths of 13, 20 and 16, respectively. Assume that we shift in the 2-out-of-7 code into the scan chains. The first pattern (f in Fig. 7.2) is shifted into every IP, and then the repeat pattern r is shifted once to IP1 and IP3; whereas it is shifted twice into IP2. The remaining empty flip-flops in IP1, IP2 and IP3 are 2 (r_1), 3 (r_2) and 5 (r_3) respectively. It is easy to note that the test patterns at the checker side of each IP are fully aligned. The next task is to align the test patterns at the input side (S_i) as we want to use a single line to provide inputs to all IP's. To achieve this, first of all, we find the IP with the maximum number of empty flops. In this case, it is IP3. The scan chain of all other IP's are added extra flip-flops, so that each one of them has the equal number of empty flops (as the maximum value found above). This situation is clearly illustrated in Fig. 7.2 where the scan chains of IP1 and IP2 are added 3 and 2 flops respectively, so that the sum of empty flops and added flops is equal to five. Since five empty flops can contain only a part of the repeat pattern in this case, we pump in the first 5 bits of the repeat pattern to every scan chain. Now, our scan chains are aligned in the checker side as well as the input side. The remaining portion of the current repeat pattern and the subsequent repeat patterns can be broadcast to all IP's simultaneously. In mathematical terms, if the number of empty flops for IP1, IP2 and IP3 are given by r_1, r_2 and r_3, then r_{max} is given by

$$r_{max} = \text{Max } (r_1, r_2, r_3).$$

Note that we have
$r_i = (L_i - k)$ % r where L_i is the scan chain length of IP_i, and k is the cone size.

The number of extra flops to be added for each core is given by,

$$r'_1 = r_{max} \text{ - } r_1,$$
$$r'_2 = r_{max} \text{ - } r_2$$
$$r'_3 = r_{max} \text{ - } r_3$$

Therefore, it can be observed that the number of extra flops for each core mainly depends on the length of the repeat pattern. If the repeat pattern is such that, the maximum deviation between the r_1, r_2, r_3, $\cdots$ are high, then the number of extra flops will naturally increase. Since the scan chain lengths of the IP's are fixed, the number of extra flip-flops required can be reduced by choosing an appropriate m-out-of-n code with a suitable repeat pattern. This is an extra constraint that must be considered while choosing the m-out-of-n code for testing. Once input/output are aligned, we just keep pushing the test repeat patterns. Therefore, all control signals can be saved.

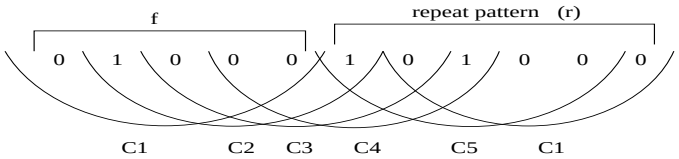


Figure 7.1: 2-out-of-7 code sequence.

7.2 Test Architecture and Control Signal Sharing

To simplify the discussion, we assume that there is a single scan chain in each core, and the scan chains of all cores receive inputs from a scan in port. This is a broadcast test architecture [1].

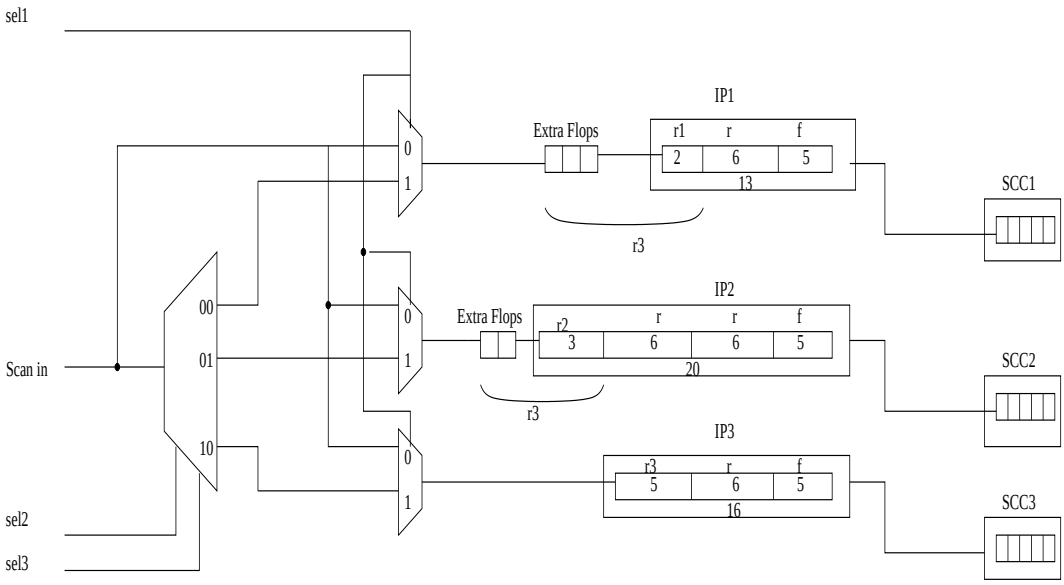


Figure 7.2: Extra hardware to support broadcast test architecture.

As shown in Fig. 7.3, the SoC chip contains four cores and each core contains one scan chain with a SCC attached to the tail of the scan chain. There is a FSM which receives inputs Tck (test clock) and Tcs (test control signal) from the tester, and outputs control signals are shared by all checkers. Note that Tcs is similar to TMS in the boundary scan controller design of IEEE 1149.1 [25]. This design enables the FSM to communicate with the tester using an extremely small number of pins. The control signals mainly consist of the expected m value, the signals to turn on some gates to sum0 and sum1 blocks, and signals to latch the SCC outputs (i.e., 01 or 10 if no error is found) into the flop-flops (e.g., FF_1 and FF_2 in Fig. 7.3) for each SCC. A pair of flip-flops is used to latch the outputs for each checker, and all flip-flops are connected as another scan chain to identify the core that contains erroneous scan chain. The outputs of each SCC also go to a (global) two rail checker (TRC), which gives a codeword (i.e., 01 or 10) if all SCC outputs are valid codewords (i.e., 01 or 10). If any of the SCC output is invalid (i.e., 00 or 11), the TRC output becomes 00 or 11. Note that the tester monitors the TRC outputs in every valid codeword cycle. Once an error is detected at the TRC output, the contents of the smaller scan chain (consisting of FF_1 to FF_8) are shifted out, and thus the faulty core can be identified. It is important to note that the cores present in the SoC circuit are provided by vendors, while the checkers and all associated test circuitries are designed by the core users. Therefore, the core users should design the FSM, checkers and test flip-flops with extra care, so that there is a minimum chance of error occurrence (in these designs) due to signal integrity. Ideally, the DFT circuitry should be isolated from the cores, so that the probability of errors occurring in the DFT design, due to high-speed switchings in cores, is minimized.

The test architecture works as follows. First, the tester gives a signal by Tcs to wake up the FSM, then the tester inputs another signal by Tcs to have the FSM send appropriate test control signals such as the expected m value, the S_{ij} signals to the checkers as shown in Fig. 7.3 and Fig. 6.1. This is to reconfigure the test architecture for a specific m-out-of-n code sequence. The next objective is to align the test patterns of each scan chain. To begin with, we assume that extra flip-flops are already added at the inputs of the scan chains for each core. Next, the $sel1$ signal is asserted 1 and $sel2$ and $sel3$ (Fig. 7.2 are also asserted by the FSM so that IP1 is selected. Now,

the first pattern (f) and repeat pattern (r) are pumped in until IP1 is full. Note that, it is possible that only a part of the repeat pattern is shifted into IP1 by the end of the last cycle. Next, the $sel2$ and $sel3$ signals are changed so that IP2 is selected. The same process is repeated again in this case. Similarly, all scan chains are selected one-by-one and f followed by r patterns are shifted into it. After this, the test architecture is ready to test the scan chains simultaneously. So, the FSM de-asserts the $sel1$ signal so that all IP's are connected into the broadcast mode, and the tester begin pumping in the remainder of the repeat pattern, if any. This is followed by continuous shifting of repeat patterns. Every time when a codeword is shifted into the R registers in each SCC as shown in Fig. 6.1, the tester sends a signal by Tcs to the FSM to generate a signal for latching the outputs of each checker. The tester also monitors the TRC outputs and in case of error (indicated by TRC), scans out the control flip-flops (e.g., FF_1 to FF_8 in Fig. 7.3) to identify the faulty core. The above process is repeated until the user is satisfied with the test results.

It can be easily observed that there is a latency involved in beginning the actual testing of the scan chain (due to test pattern alignment process). But, it is possible to save some test cycles by broadcasting the common test patterns to the IP's during the alignment process. For the above example, the first 13 test patterns can be broadcast, and then the fine alignment process can be exercised. Once the alignment is done at both ends, the repeat patterns can be broadcast as described above.

If multiple m-out-of-n code sequences are used, they *cannot* be broadcast in a sequential manner, i.e., AAA...BBBAAA..BBB, because there will be an alignment problem at the checker side when codewords of code B are applied just after the codewords of code A. Hence, we propose that the multiple codes be combined in an interleaved manner (i.e., ABABA..BA), and the repeat pattern of the combined sequence (i.e., AB) is determined. After the first pattern f of code A is shifted, the multiple m-out-of-n codewords can be obtained by simply repeating the new repeat pattern. The FSM (under the control of the tester by Tcs) must issue the expected m values and all corresponding S_{ij} values in an interleaved manner.

For example, consider a 3-out-of-11 code with weight set of {1,2,3} and weight distribution of 1-3-2-3-2, and a 4-out-of-11 code with weight distribution of 1-2-3-3-2. The code sequence for the 3-out-of-11 code is 01000 → 10001 → 00010 → 10100, while the code sequence for the 4-out-of-11 code is 01001 → 10010 → 10100. The codewords belonging to the 4-out-of-11 code are followed by the codewords belonging to the 3-out-of-11 code. The repeat pattern for the combined sequence is 10100101000 (Fig. 7.4). It can be noticed that by iterating the new repeat pattern, one can obtain the codewords of 3-out-of-11 and 4-out-of-11 code in an interleaved manner. Once the alignment is achieved (by following the same procedure as mentioned previously), the FSM only needs to switch the expected m and S_{ij}'s at proper cycles. That is, when the codewords of the 3-out-of-11 code are being shifted out, the expected m should be 3 and the weight distribution should be configured for 1-3-2-3-2 (by providing proper S_{ij}'s); while the expected m should be changed to 4 and the weight distribution should be changed to 1-2-3-3-2 by providing S_{ij}, when the codewords of the 4-out-of-11 code are shifted out. The same procedure is repeated as many times as the user wishes. The above idea can also be extended to combine more than two m-out-of-n codes.

In can be observed that the complexity of control signal generation is greatly reduced by adopting the test pattern broadcasting architecture. Also, the control signals can be easily shared by all SCC's because every checker receives a codeword at the same instance. Further, we have avoided the insertion of dummy patterns because the alignment problem is easily solved by adding a limited number of extra registers.

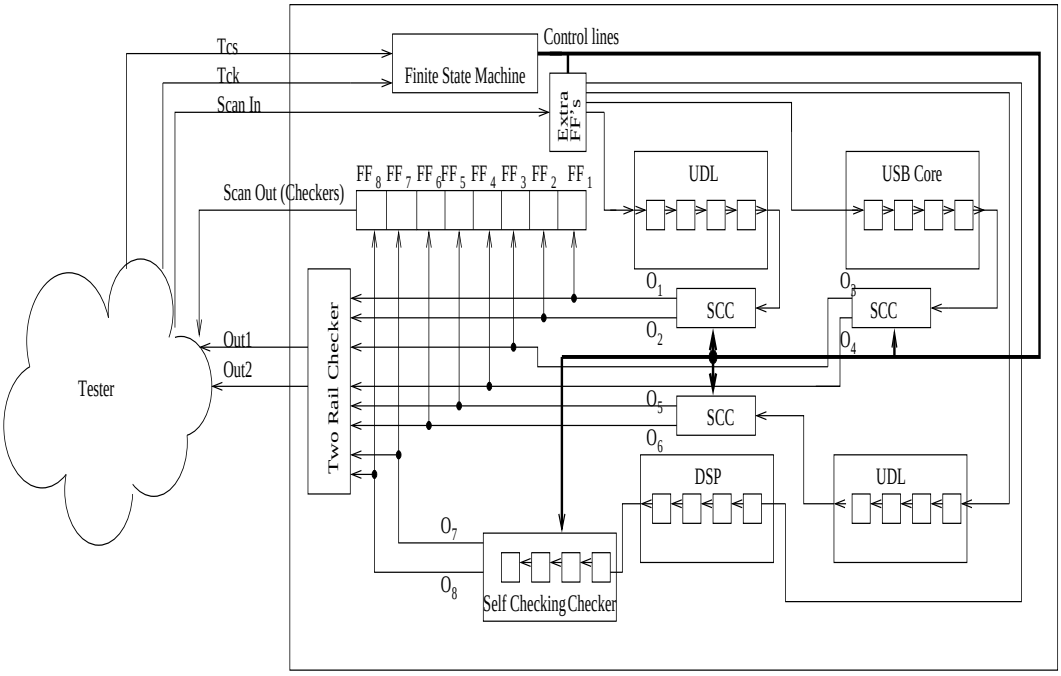


Figure 7.3: Testing scheme with shared control lines.

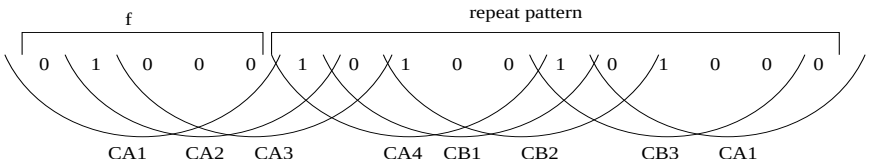


Figure 7.4: Combined 3-out-of-11 and 4-out-of-11 code sequence.

Chapter 8

Conclusions and Future Work

We have proposed the application of weight-based m-out-of-n codes for scan chain fault diagnosis at the core level of a SoC circuit. The codewords are shifted into the scan chain(s) as test patterns, and the output responses are validated using self-checking checkers. By proper code selection and/or by combining two or more m-out-of-n codes, it is possible to activate many scan chain faults and to diagnose all unidirectional as well as many bidirectional and multi-directional errors at the core level with small hardware overhead. The aliasing probabilities have been thoroughly analyzed for single and a combination of two or more weight-based m-out-of-n codes. Experimental results demonstrate that aliasing probabilities are zero in most cases. A test pattern broadcast architecture is incorporated to support the proposed testing method. By adopting this architecture, the complexity of control signal generation and scan chain test is greatly reduced. Further, by adding a test pattern alignment scheme, it was possible to use shared control lines for all checkers and to use a single scan-in signal for all scan chains (as required by the broadcast architecture). Therefore, the hardware overhead of the test architecture is greatly reduced. Totally self-checking checkers are designed to support the entire approach. The future research work are: (1) to develop a simulator that can evaluate the power of codewords applied (instead of using only the transition counts); (2) to extend the result to the domain of fine-grained scan cell identification; (3) to develop a more powerful method (e.g., to support multiple m-out-of-n codes) to solve the test pattern alignment problem by using small hardware overhead; (4) to develop a built-in self-test method for generating the efficient codewords and perform at-speed diagnosis of the scan chain.

Bibliography

[1] B. W. Johnson. *Design an analysis of fault-tolerant Digital systems*. Addison-Wesley Publishing Company, 1989.

[2] J.H. Jiang, W.B. Jone, S.C. Chang, and S.Ghosh. Embedded core test generation using test pattern broadcasting and netlist scrambling. In *IEEE Transaction on reliability*, volume 52, pages 435–443, 2003.

[3] Y. Zorian, E. J. Marinissen, and S. Dey. Testing embedded-core based system chips. In *Proceedings of International Test Conference*, pages 130–143, 1998.

[4] S. Kundu. On diagnosis of faults in a scan chain. In *Proceedings of VLSI test symposium*, pages 303–308, 1993.

[5] J. L. Schafer, F. A. Policastri, and R. J. McNulty. Partner srls for improved shift register diagnostics. In *Proceedings of VLSI Test Conference*, pages 198–201, 1992.

[6] S. Edirisooriya and G. Edirisooriya. Diagnosis of scan path failures. In *Proceedings of VLSI Test Symposium*, pages 303–308, 1993.

[7] S. Narayanan and A. Das. An efficient scheme to diagnose scan chains. In *Proceedings of International Test Conference*, pages 704–713, 1997.

[8] Y. Wu. Diagnosis of scan chain failures. In *Proceedings of international symposium on defect and fault tolerance in VLSI systems*, 2, pages 217–222, 1998.

[9] L. Cheney and N. Sheils. A method for isolating defects in scannable sequential elements. In *Proceedings of Intel Design and Test Technology Conference*, 2000.

[10] J. Hirase, N.Shindou, and K. A. Akahori. Scan chain diagnosis using iddq current measurement. In *Proceedings of Asian Test Symposium*, pages 153–157, 1999.

[11] K. Stanley. High accuracy flush-and-scan software diagnosis. In *IEEE design and test of computers*, pages 56–62, 2001.

[12] Y. Huang, W. T. Cheng, S. M. Reddy, C. J. Hseih, and Y. T. Hung. Statistical diagnosis for intermittent scan chain hold-time fault. In *International Test Conference*, pages 319–328, 2032.

[13] Y. Huang, W. T. Cheng, C. J. Hseih, H. Y. Tseng, A. Huang, and Y. T. Hung. Efficient diagnosis of multiple intermittent scan chain hold time faults. In *Proceedings of Asian test symposium*, pages 40–49, 2003.

[14] Y. Cao, X. Huang, N. H. Chang, S. Lin, O. S. Nakagawa, W. Xie, D. Sylvester, and C. Hu. Effective on-chip inductance modeling for multiple signal lines and application to repeater insertion. In *IEEE Transactions on VLSI Systems*, volume 10, 2002.

[15] M. Favalli and C. Metra. Optimization of error detecting codes for the detection of crosstalk originated errors. In *Proceedings of Conference of Design, Automation and Test in Europe*, pages 290–296, 2001.

[16] Parag K. Lala. *Self-Checking and Fault-Tolerant Digital Design*. Morgan Kaufman Publishers, 2001.

[17] X. Kavousianos and D. Nikolos. Novel single and double output tsc berger code checkers. In *Proceedings of VLSI test symposium*, pages 348–353, 1998.

[18] C. Metra, M. Favalli, and B. Ricco. 1-out-of-3 code checker with single output. In *Electronics Letters*, pages 1373–1374, 1997.

[19] C. Metra, M. Favalli, and B. Ricco. Highly testable and compact 1-out-of-n code checker with single output. In *Proceedings of Design, Automation and Test Conference*, pages 981–982, 1998.

[20] D. Das and N. Touba. Weight-based codes and their application to concurrent error detection of multilevel circuits. In *Proc. of IEEE VLSI Test Symposium*, pages 370–376, 1999.

[21] D. Das, N. Touba, M. Seuring, and M. Gossel. Low cost concurrent error detection based on modulo weight-based codes. In *IEEE On-Line Test Worksho*, pages 524–533, 1998.

[22] M. Favalli and C. Metra. Bus crosstalk fault error detection capabilities of error detecting codes for on-line testing. *IEEE Transaction on VLSI Systems*, 3(7):392–396, 1999.

[23] C. Metra, M. Favalli, and B. Ricco. Self-checking detection and diagnosis for transient , delay and crosstalk faults affecting bus lines. *IEEE Transaction on Computers*, 49(6):560–574, 2000.

[24] C. Metra, M. Favalli, and B. Ricco. On-line detection of logic errors due to crosstalk, delay and transient faults. In *Proceedings of International Test Conference*, pages 524–533, 1998.

[25] IEEE Standards Board. *IEEE Standard Test Access Port and Boundary-Scan Archirecture*. IEEE/ANSI Standard 1149.1-1994 (revision b), includes supplements 1149.1a and 1149.1b, 1994.