Circuit Partitioning for Application-Dependent FPGA Testing

by

Rui Zhen Feng
B.Eng, Hefei University of Technology, 1996

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Supervisory Committee

Circuit Partitioning for Application-Dependent FPGA Testing

by

Rui Zhen Feng
B.Eng, Hefei University of Technology, 1996

Supervisory Committee

Dr. Jon C. Muzio, (Department of Computer Science)
Supervisor

Dr. Micaela Serra, (Department of Computer Science)
Co-Supervisor

Dr. Kui Wu, (Department of Computer Science)
Departmental Member

Dr. Kenneth B. Kent (Faculty of Computer Science, University of New Brunswick)
Outside Examiner

Abstract

Supervisory Committee

Dr. Jon C. Muzio, (Department of Computer Science)
Supervisor

Dr. Micaela Serra, (Department of Computer Science)
Co-Supervisor

Dr. Kui Wu, (Department of Computer Science)
Departmental Member

Dr. Kenneth B. Kent (Faculty of Computer Science, University of New Brunswick)
Outside Examiner

Application-dependent FPGA testing is performed to ensure that a particular user-defined application is implemented on fault-free areas of an FPGA. Applying this type of test technique leads to yield increases and cost reductions in the use of FPGAs.

In this thesis, we propose a novel application-dependent FPGA testing strategy, in which a recursive circuit partitioning algorithm is employed to obtain a testing configuration solution for a user-specific application. This algorithm is implemented and the experimental results are analyzed to demonstrate the effectiveness of the proposed testing strategy.

Our experimental results show that the circuit partitioning method can be used to provide a reasonable solution for an arbitrary application with significantly improved fault coverage and an approximately minimized number of cut points.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ASIC	Application-Specific Integrated Circuit
BIST	Built-In Self Test
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
CUT	Circuit Under Test
DFT	Design for Testability
FPGA	Field-Programmable Gate Array
LFSR	Linear Feedback Shift Register
LUT	Look Up Table
MTP	Manufacturing Test Procedures
ORA	Output Response Analyzer
PAL	Programmable Array Logic
PIP	Programmable Interconnect Points
PLB	Programmable logic block
TPG	Test Pattern Generator
UTP	User Test Procedures

1. Introduction

A Field-Programmable Gate Array (FPGA) is a logic device that can be programmed to implement a variety of digital circuits. FPGAs have become widely applied both in product prototyping and development because of their ability for configuration and re-configuration, the advantage of short design and implementation cycles and the low non-recurring engineering cost. .

However, with the increase in density, capability and speed, FPGAs have become more vulnerable to faults, as it is the case for all circuits. A percentage of manufactured FPGA chips are determined to be faulty after initial application-independent tests. Faulty FPGAs can also be found after delivery to users, during the system development or operation. They may be still usable for some particular application if only a portion of the circuitry is defective. *Application-dependent FPGA testing* is a process that tests the FPGA only for an implemented design, identifies the fault location, after which an application could still be implemented by avoiding the faulty part of the FPGA.

Since testability of a digital design "is *not considered in the design flow using FPGAs* [3]" based on the assumption that only fault-free FPGAs are used, Design For Testability (DFT) is an issue that needs to be considered in an application-dependent FPGA test. As it is the case for most circuits, the number of I/O blocks is much smaller compared to the number of logic blocks and interconnections. Built-In Self Test (BIST) is often used as one possible solution to this bottleneck.

Differently from application-independent FPGA testing, which tests all resources on an FPGA chip, application-dependent testing targets only the resources that are used in an application. Application-dependent testing of FPGAs has been discussed in [2] [3] [4] [7] [12] [16] [17] [18] [25] [26]. They can be categorized into three main groups.

1. One testing method is to decompose logic functions into smaller sub-functions that can be implemented on a smaller area of the FPGA being tested [16] [17] [18].
2. Another approach takes advantage of the programmability of FPGAs and modifies the configuration of interconnections or logic blocks in order to test the two parts separately [2] [3] [4] [7] [25] [26].
3. Logic blocks and interconnections can also be tested at the same time by applying one test configuration and different test sessions [12].

In this thesis, a new testing strategy for application-dependent FPGA testing is presented. In the new approach, a group of sub-circuits, which are themselves sub-components of a user defined circuit, are tested in parallel. Logic blocks and interconnections are tested at the same time as parts of the circuit being tested. No re-configuration is required to test different parts of an FPGA. A BIST structure is applied. The core of the method is a recursive circuit partitioning process, which divides a larger circuit into smaller ones and provides a configuration solution for a user specified design during the application-dependent testing. Higher fault coverage, a smaller size of each sub-circuit

and a reasonable number of partitioning points should be produced in the partitioning process.

Some background is provided in Chapter 2, with an introduction to FPGAs, to some of the fundamental concepts of digital testing, and to other relevant techniques such as DFT methods and minimum cut algorithms. In Chapter 3, application-dependent FPGA testing is discussed including its benefits and available techniques.

In Chapter 4, a novel application-dependent FPGA testing approach is presented. The work includes the design of the BIST architecture and the test configuration, together with the recursive circuit partitioning method, which is the core of the testing strategy. The circuit partitioning algorithm is the only part evaluated currently in the experiments. In Chapter 5, the experiments on the recursive circuit partitioning method are explained. In Chapter 6, the experimental results are presented and analyzed. Chapter 7 concludes the thesis with a summary of the main contributions and potential topics for future work.

2. Background

The purpose of this chapter is to review the relevant background for the research in this work, including an introduction to FPGAs, digital system testing, Built-In Self Test (BIST), Design for Testability (DFT) and the minimum cut algorithm.

2.1. Field-Programmable Gate Arrays

An FPGA was originally developed from early programmable logic devices such as Programmable Array Logic (PAL) and CPLD (Complex Programmable Logic Device). However, compared to them, FPGAs have a simpler architecture and more flexibility, but are more complex to use for design and mapping development [6].

The general FPGA architecture for this work consists of a two dimensional array of Configurable Logic Blocks (CLBs), also called Programmable Logic Blocks(PLBs), a set of configurable Input/Output blocks, referred as I/O pads, and a configurable interconnection network as shown in Figure 2.1[1].

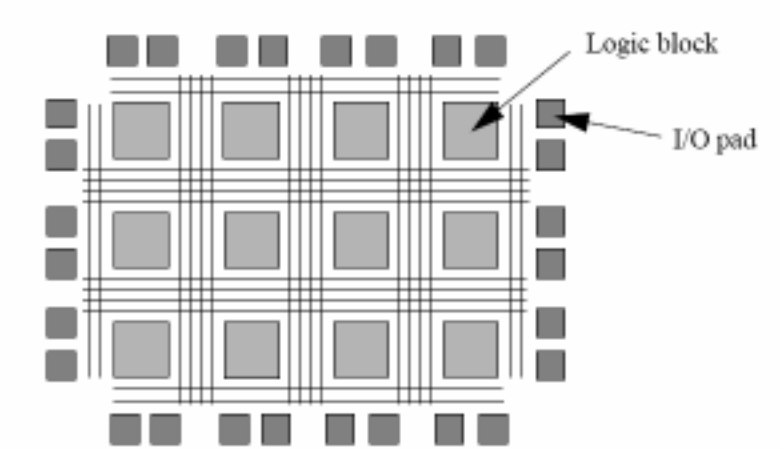


Figure 2.1 FPGA Architecture [1]

CLBs are used to implement the logic functions of a design. There are several types of implementation of logic blocks, with the variable-input Look-Up Table (LUT), for example, being the basic component of the Xilinx Virtex serial products since 1998 [42]. A CLB can implement all possible functions for *k* inputs, where *k* has normally the value of 4 or 5. A *k-input Look-Up Table (LUT)* based logic block design is illustrated in Figure 2.2[1]. With the addition of the memory element (Flip-flop) the overall CLB can realize both combinational and sequential circuits.

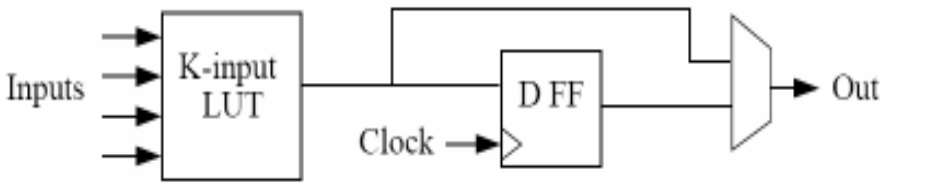


Figure 2.2 FPGA Logic Block Structure [1]

The *interconnection network* presents itself as a group of vertical and horizon wires amongst the logic blocks and I/O pads (as seen in Figure 2.1). As an example, the

segmented routing structures, used in Xilinx[6], the wire segments can be connected by switches that are in turn controlled by configuration bits. The state of the switch is either open or closed. A programmable switch matrix can be configured to connect logic blocks to other logic blocks or logic blocks to I/O blocks [6]. A simple diagram of a general interconnection is shown in Figure 2.3.

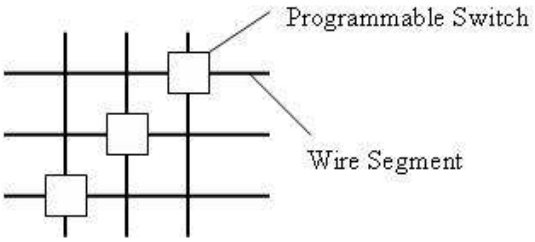


Figure 2.3 Switch Block Topology

The general design and development flow in using FPGAs follows the process briefly summarized here.

1. A circuit is designed and verified for correctness.
2. Algorithms are applied to decompose it into units of less than or equal to k inputs, such that each can be mapped into a CLB.
3. Proprietary software is used to map the units to CLBs and the interconnection routing for the wires.
4. The "personality" for the FPGA configuration, as derived by the software process, is downloaded to the hardware and execution can start.

The advances in FPGAs include the presence of higher-level embedded functions (e.g. multipliers) and memories, and the ability for dynamic and partial

re-configuration. FPGA based digital systems appear to have a shorter time to market, lower non-recurring engineering cost and the ability of reprogramming to correct design errors or update a design. However, FPGA-based applications are generally slower than an Application-Specific Integrated Circuit (ASIC) due to the special FPGA architecture, clock speed and other features.

2.2. Digital Systems Testing

Digital system testing commonly refers to a process in which "the system is exercised and its resulting response is analyzed to ascertain whether it behaved correctly [9, Ch1]." The two phases of the process are *verification*, in which a digital system is checked for behavioural correctness, and *testing*, which is performed after manufacturing in order to ensure that only fault-free products are delivered to users [11]. This work only encompasses the latter phase, testing. When faults are detected, the next step may be *fault diagnosis* to locate the fault site or determine the cause of the failure. All digital devices are tested during the manufacturing process to ensure the quality of the product. It also should be tested after the delivery and periodically during operations in order to certify the continued fault-free operation [11].

Testing techniques and methods can be classified according to many criteria. Figure 2.4 shows a brief taxonomy of testing methods, classified according to the time when the test is performed [11]. For *off-line*, testing is performed as a separate activity and the circuit enters a "test mode". For *on-line* testing, the circuit can be tested concurrently with the normal operation [11]. Off-line testing can be executed by

applying Built-In Self Test (BIST) and on-line testing can be performed by applying concurrent test, which is also a form of BIST[9,Ch1] [11].

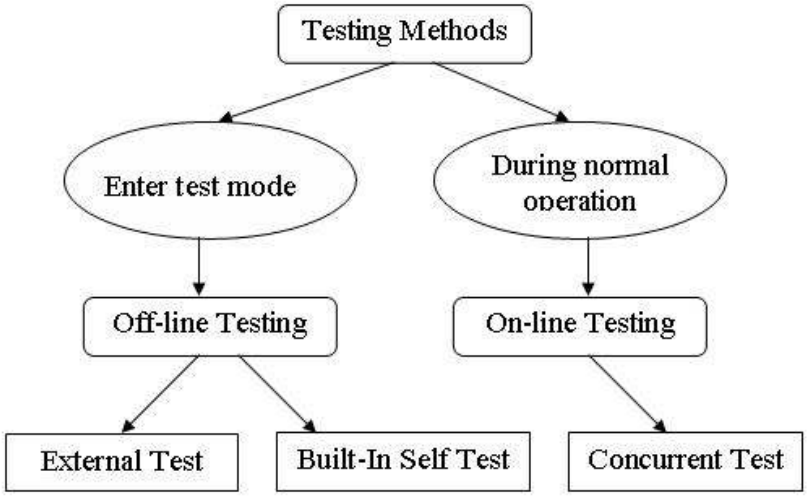


Figure 2.4 Taxonomy of Testing Methods

The testing cost is directly related to the testing complexity and to the testing process. Design for Testability (DFT) introduces the concept of improving later testability of a circuit during the design phase. Some methods of DFT can reduce the cost on some test processes down the line, but not necessarily on all processes [9]. For example, adding test points may lower the cost of test generation, but increases the number of I/O pins and area overhead.

Faults or *physical faults* refer to a collection of fabrication defects and physical failures. Faults can be classified as permanent, intermittent or transient faults according to their stability in time [9, Ch1] [11]. Here only *permanent* faults are considered.

The most classic fault model is the stuck-at fault, which refers to a signal being logically stuck at a fixed value 0 or 1[9]. The single stuck-at fault model (SSF) is widely used because it can discover other defects as well. For example, an electrical short between two wires can be viewed as a stuck-at-1 fault for a line segment connected to a power source [9].

The total number of SSFs to be explicitly analyzed is" $2 \times$ *the number of wire segments* ", but it can be reduced by fault collapsing techniques. For example, an input stuck-at-0 fault for an AND gate has the same logical effect as the output stuck-at-0 fault, and only one of the two needs to be tested [11]. Faults that have the same logical effect are called *equivalent faults* and they can be grouped into classes. Any testing method only needs to cover one fault from each class.

The *fault coverage* is defined as the percentage of detected faults over all detectable faults. "A fault *f* is detectable if there exists a test *t* that detects *f;* otherwise, *f* is an undetectable fault [9, ch4]." Undetectable faults in combinational circuits are normally caused by redundancies, while for sequential circuits, some faults are not detectable due to unreachable states. Therefore, sequential circuits are more difficult to test. In this thesis, we consider sequential circuits as the primary target for testing.

The circuit partitioning algorithm to be introduced in this thesis employs a structural model, instead of a logic model of circuits. A *structural model* of a digital system can be represented by a graph. As a result, graph theory and its algorithms can be applied to logic network analysis. Figure 2.5 shows a simple conversion from a logic circuit to a graph representation. If a signal propagates to more than one destination, we say the signal has a *fanout* [9, Ch2]. Input and fanout signals are also called *check points*. If a signal does not have fanout, we say it is fanout-free [9, Ch2]. A circuit that consists of only fanout-free signals is easily converted to a tree as in the example in Figure 2.5.

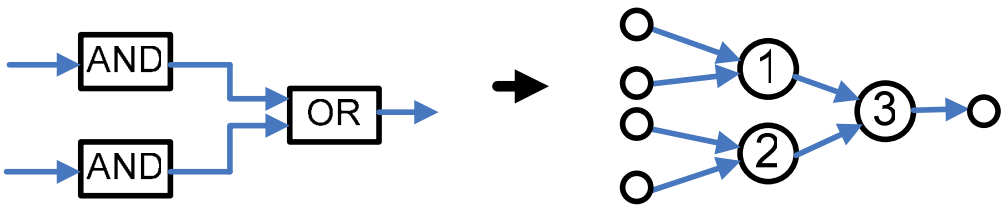


Figure 2.5 A Structural Model Example

2.2.1. Built-In Self Test

Built-In Self Test (BIST) implies that a circuit has the capability to test itself. BIST can be classified into two main categories, namely on-line and off-line BIST. On-line BIST, also called concurrent test, is performed when the system is in normal operation state whereas off-line BIST is executed when the system is not carrying its normal functions [9] [11]. Off-line BIST is the main method of the testing strategy introduced in Chapter 4.

The three main components of a general BIST architecture are: (a) the Test Pattern Generator (TPG); (b) the Circuit Under Test (CUT); and (c) the Output Response Analyzer (ORA). Figure 2.6 shows a typical BIST structure.

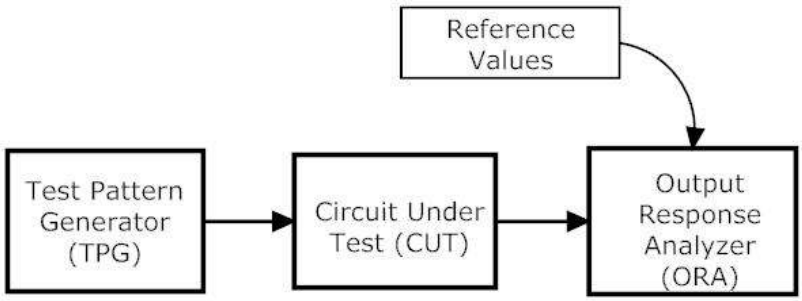


Figure 2.6 A Built-In Self Test Architecture

A TPG provides specified test patterns according to generation techniques which are usually categorized as exhaustive, pseudo-exhaustive, random, and pseudo-random methods. An n-bit counter or address generator, which produce test vectors from 0 to 2^{n-1}, is an example of an exhaustive TPG for a circuit with n inputs, while a Linear Feedback Shift Register (LFSR) is a typical pseudo-random TPG [11].

An ORA focuses on the output stream and determines the correctness of the circuit's response. An example of an ORAs is a signature analyzer. In a test design, especially BIST, it is not feasible to store all expected output sequences in order to verify the correctness of the output response [11]. A signature is a compacted version of a long sequence according to some algorithm. A signature analyzer compares the expected

good signature of an output sequence with the signature of the output obtained from a test.

2.2.2. Design for Testability Techniques

"*Testability is a design characteristic that influences various costs associated with testing* [9]." Controllability and observability are two main factors of testability. *Controllability* is the ability to access a circuit node and establish a specific signal value by setting values on the circuit inputs (test points) [9] [27]. *Observability* is the ability to observe the signal value at a node in a circuit by controlling the input and observing the output.

DFT techniques are applied to ensure that a device is testable mainly by improving the controllability and observability of the circuit [9, Ch9]. Most DFT techniques are developed by either re-synthesising an existing logic design or adding extra hardware to the design [9, Ch9]. The trade-offs of these approaches are the increase of area overhead, the number of I/O pins, and circuit delay [9, Ch9]. Test points and circuit partitioning techniques are reviewed in this section as they are directly relevant to this research.

Test points are employed to enhance the controllability and observability. There are two types of test points: control points, which are extra inputs used to improve the controllability, and observation points, which are extra outputs used to improve observability [9, Ch9].

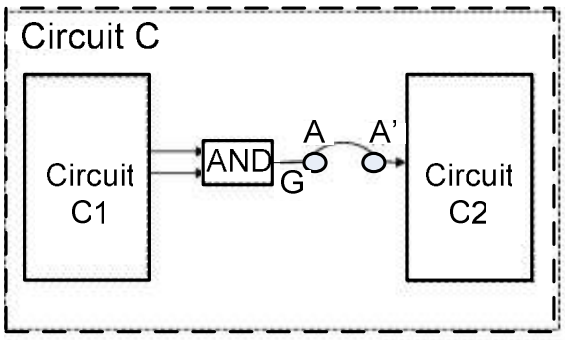


Figure 2.7 Test Points Method

Figure 2.7 shows an AND gate within a large circuit. G is the output signal from the AND gate. On a printed board implementation, G can be routed as two connected pins A and A'. A removable wire between A and A', called a "jumper" can be used to reconnect A and A' as required. A and A' acts as an observation point and a control point respectively by removing the jumper [9, Ch9]. A and A' can also be routed to an I/O pin in external testing.

Circuit partitioning is employed to reduce the test generation cost by partitioning a large circuit into smaller sub-circuits [9, Ch9]. Figure 2.8 illustrates a circuit partitioning process. Sub-circuits are viewed as black boxes. In this process, a large circuit is partitioned into smaller sub-circuits and the points being cut, called "cut points", in the partitioning process become primary inputs and outputs of the sub-circuits. This approach enhances the controllability and observability of the circuit by partitioning the connection between sub-circuit C1 and C2 into control points and observer points.

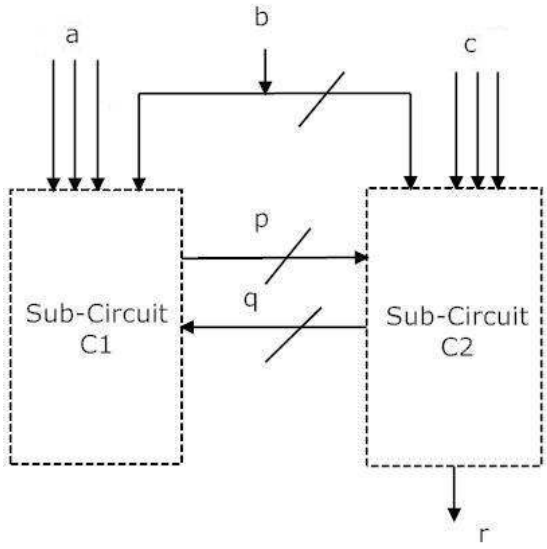


Figure 2.8 A Circuit Partitioning Method Example

This approach can also reduce the test complexity by reducing the size of test sets. In Figure 2.8, for example, one may want to test the whole circuit exhaustively. The total number of original inputs at a, b and c is 7 and the number of cut points at p and q is 2. The original circuit requires $2^7 = 128$ test vectors for an exhaustive test while two sets of $2^5 = 32$ test vectors, a total of 64, are required to test A and B individually, or only 32 test vectors are required to test C1 and C2 in parallel [9, Ch9].

2.3. FPGA Testing

A typical FPGA testing approach can be summarized by the following steps: (a) configure the device into test circuits or paths; (b) apply test patterns; and (c) analyze the output to determine if the chip is defect-free. In general, FPGA testing can be

divided into two classes: *Manufacturing Test Procedures* (MTP) and *User Test Procedures* (UTP) [37] [15].

MTP is also called *application-independent* FPGA testing and this type of tests is applied in an attempt to ship fault-free devices to users. In order to ensure the entire FPGA chip is fault-free, an exhaustive testing approach is desired for MTP, that is, all components on an FPGA chip need to be tested exhaustively. For example, a 4-input LUT test requires 16 test patterns and all logic blocks are required to be tested. All the switches and wire segments of the interconnection networks are also tested exhaustively. This type of testing is obviously very time consuming for large FPGAs.

UTP is also called *application-dependent* or application-specific FPGA testing. This type of test is performed on the users' side to ensure the user-defined circuit is implemented on fault-free areas of an FPGA chip. The detailed information of UTP is provided in chapter 3.

2.3.1. Fault Models

Because of the special architecture of FPGAs, fault models are categorized by the location and type of the fault [10][15][37][38]. Since CLBs and interconnection networks are two key parts of an FPGA, we adopt the approach and categorize FPGA fault models into two classes: *logic block fault models* and *interconnection network*

fault models. As for other digital circuits, fault models in logic blocks include stuck-at faults, delay faults and bridging faults.

As introduced in earlier section, an interconnection network consists of wire segments and programmable switches. Figure 2.9 illustrates a switch matrix and some possible connections where a Programmable Interconnect Point is either open or closed according to the configuration bit value stored in the SRAM [38].

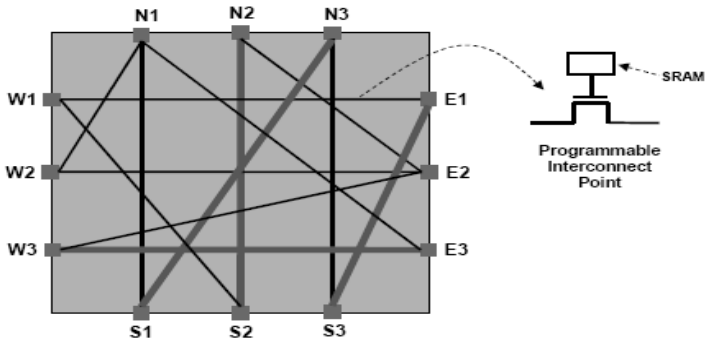


Figure 2.9 A Switch Matrix and Possible Connections [38]

An *open fault* can be a PIP (Programmable Interconnect Points) stuck-open or an open on a line segment. A PIP stuck-open fault causes the PIP to be permanently open regardless of the value of the SRAM cell controlling the PIP [38]. A short fault can be a PIP stuck-closed or a short between two routing resources [38]. Each of these resources can be a PIP or a line segment (namely, PIP-PIP, PIP-line segment or line segment-line segment). A PIP stuck-closed fault causes the PIP to be permanently closed regardless of the value of memory cell controlling it.

An open on any line surrounding a switch matrix which should be connected will cause a *line-open fault*; a bridge of any pair of lines surrounding a switch matrix will cause a *line-pair bridge fault* [10] [15]. Two interconnection wire segments incorrectly connected as either a wired-AND or a wired-OR can cause a bridge fault [37].

A stuck-at fault model can be used to cover many faults introduced above. Any open fault such as line- open or PIP-stuck-open can be modeled as a stuck-at-0 fault [15]. A short between a line and ground can also be modeled as a stuck-at-0 fault whereas a short between a line and power is a stuck-at-1 fault [15].

2.3.2. Testing Methods

Fault detection techniques for FPGAs can be divided into logic block testing and interconnection testing. Because a logic block is small and simple, it is very easy to test each PLB. For example, for 4-input CLBs, $2^4 = 16$ test patterns are required for an exhaustive test. However, in MTP, we need to consider the entire FPGA including CLBs and interconnection networks. Similarly, all the possible paths in interconnections are required to be tested. For a FPGA with thousands of logic blocks, one needs to consider the time complexity on an exhaustive test.

Both external and BIST can be applied in FPGA testing [39] [40] [41]. External testing methods apply extra hardware to provide TPG and ORA and use I/O pins for

applying test vectors and receiving outputs. BIST methods, however, take advantage of the configuration and reconfiguration ability of an FPGA chip and use the chip itself to implement testing circuits. Figure 2.10 shows an example of applying BIST for logic block testing [41]. In this example, PLBs are tested in parallel using one TPG and one ORA, which are implemented on other portions of the FPGA itself. Reconfiguration is required in order to test PLBs that are used as testing circuits. Similar BIST methods can also be used in interconnection testing.

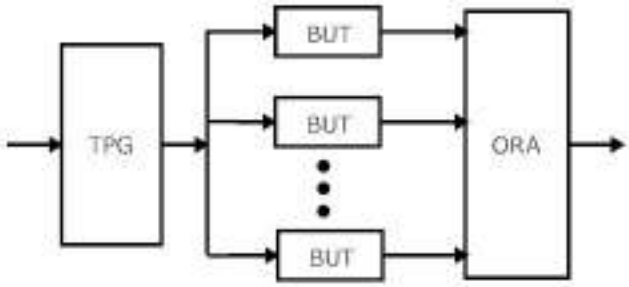


Figure 2.10 FPGA Built-In Self Test [41]

2.4. Minimum Cut Algorithms

Finding the minimum cut of an undirected edge-weighted graph is a fundamental problem found in many fields [30]. In particular, the algorithm needs to find a connection net of a graph such that its disconnection between vertices results in disconnected graphs and the cut weights, that is, the actual number of edges cut, is minimum [29].

More formally, for a graph G, a cut means a partition (S, T) of the vertex set V such that $S \bigcup T = V$, $S \bigcap T = \phi$. The min-cut problem can be presented as follows:

Given an undirected graph G= (V, E), where V is a vertex set and E is an edge set, find a min-cut, that is, minimum number of edges needed to be removed in order to make G disconnected [36].

The runtime of a min-cut algorithm varies for different approaches. For example, for the Edmonds-Karp algorithm, the runtime is $O(VE^2)$ and for Goldberg-Tarjan algorithm, it is $O(VE \log(V^2/E))$ [31]. Figure 2.11 shows an example of min-cut.

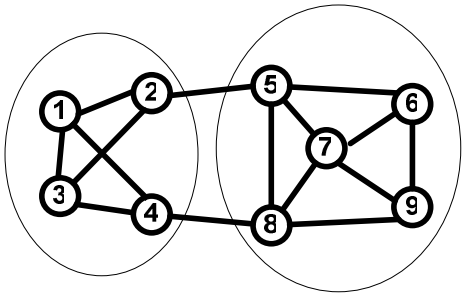


Figure 2.11 A Min-Cut Example

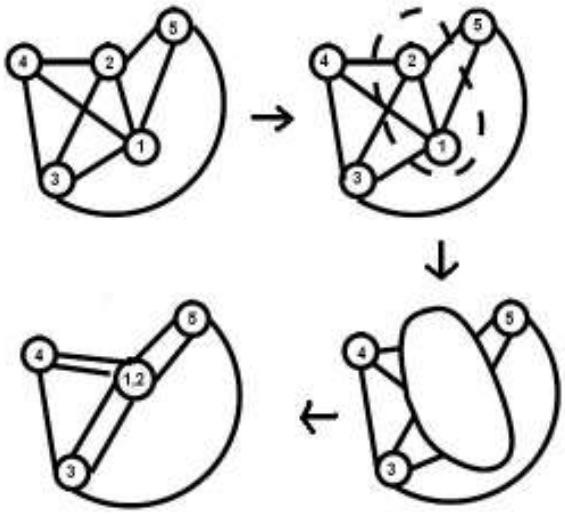


Figure 2.12 Edges Contraction Example [36]

The edge contraction algorithm is applied to obtain sub-graphs for a selected node. This process starts with the selected node *v*, then it selects an edge *e = (v,u)* and

contracts the edge to create a new vertex $\{v,u\}$. All the neighbours of $\{u,v\}$ are contracted in the same way. Figure 2.12 shows an example of the contraction where node 1 is selected as the start node and edge (1,2) is contracted to create the new vertex {1,2}.

2.5. Summary

A FPGA (Field-Programmable Gate Array) is a programmable logic device that consists of logic components, switches and interconnects. FPGA testing is performed to either ensure the quality of an FPGA chip to be delivered, or to ensure an application is implemented on the fault-free area of an FPGA. FPGA fault models are developed differently from a tradition digital circuit due to its special technology. The classic stuck-at faults are widely used in digital circuit testing including FPGA testing.

Two types of FPGA testing are MTP and UTP. Two key parts, logic blocks and interconnections are usually tested separately in a MTP. UTP or application dependent FPGA testing is introduced as the problem of the thesis in the following chapter. Other concepts such as min-cut and design for testability are also introduced in this chapter in order to provide background for the proposed testing strategy in Chapter 4.

3. Application-Dependent FPGA Testing

This chapter gives details of the application-dependent FPGA test problem in order to provide more information and background.

The programmability of Field Programmable Gate Arrays (FPGAs) results in much faster and more flexible digital system design and prototyping compared to Application-Specified Integrated Circuits (ASICs). Application-dependent FPGA testing is performed to ensure no defective area is used for user-defined circuit implementation.

The content of this chapter includes the benefits of applying application-dependent test, testing techniques and BIST usage on this particular test. Some circuit partitioning algorithms are also introduced as a potential technique that can be applied to application-dependent FPGA testing.

3.1. Benefits of Applying Application-Dependent FPGA Testing

Application-dependent FPGA testing is employed to identify the defective area on an FPGA chip for a particular application. Therefore, the functional area consisting of fault-free parts can still be used for this application. Applying this type of test benefits both FPGA manufactures and users. From the producer's aspect, a defective chip can still be delivered to certain users by disabling the faulty area to prevent it being used for some particular applications. From users' perspective, a defective or damaged FPGA chip can still be used by avoiding using the area in a FPGA mapping.

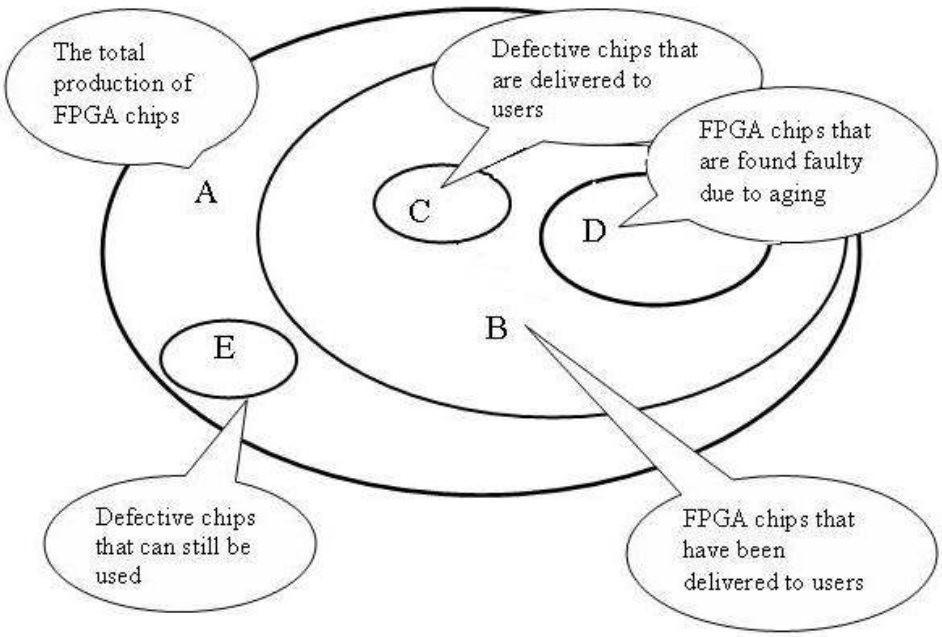


Figure 3.1 FPGA Production and Distribution Diagram

An FPGA production and distribution diagram, Figure 3.1, is presented to help show the benefits of employing application-dependent FPGA testing. Suppose set A represents the total yield of FPGA chips of a FPGA manufacturer during a period of time. A subset of A, set B, consists of chips that have passed manufacturing test and been delivered to users. Let set $F = A - B$, then F represents the chips that have failed the manufacturing test. Set E is a subset of F, which consists of chips that have a large percentage of functional parts that are still working correctly. Set C is the group of chips that are found to be defective after being delivered to users, even though they passed the tests performed by the manufacturer.

Set D consists of chips that have worn out after a period time of operation. Aging phenomena are the cause of degradation of a digital device that results in permanent faults and unusable areas of FPGAs. Depending on the design and mapping on an FPGA chip, different areas will be affected differently by aging factors such as supply voltage, current, operating temperature and switching activity [8]. Figure 3.2 is an example that shows the Mean Time To Failure (MTTF) for different wire types in FPGAs [8].

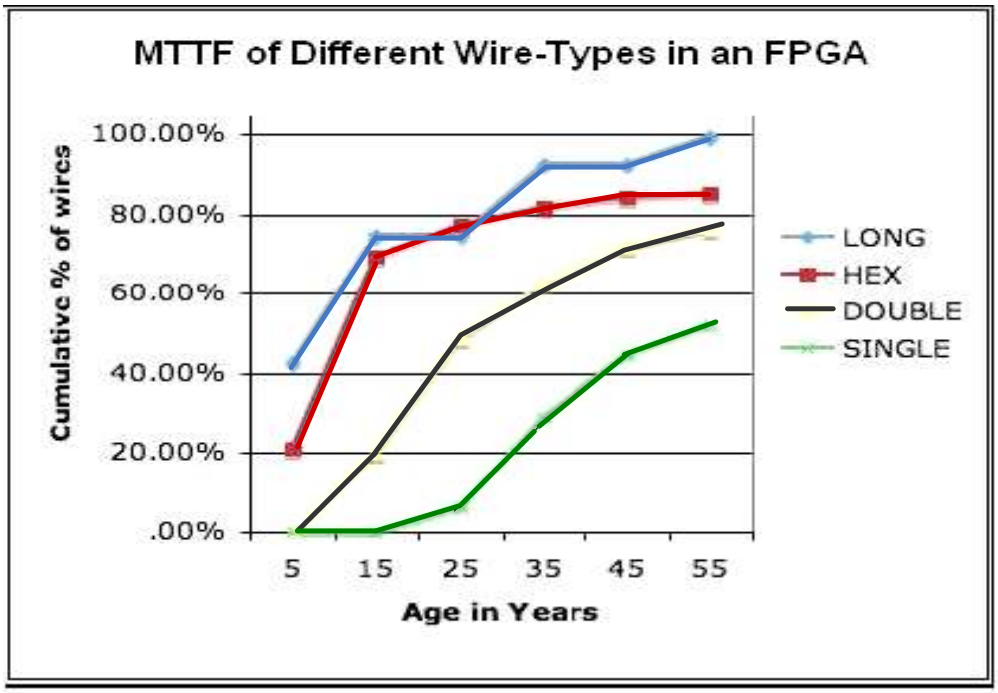


Figure 3.2 MTTF for Different Wires Type in FPGA

In the case that FPGA producers and users rely heavily on MTP, defective chips in set F, C, and D are determined as un-usable devices. However, these chips may only have defects in a small percentage of the area and a large percentage of components are still functional. Also, with the increasing density of FPGAs, the significance of

having a number of failed blocks in an FPGA decreases. For example, if a 9 X 9 FPGA has 3 defective PLBs, the percentage of failed blocks is about 4%. But if an FPGA has 3,000 PLBs, with the same number of defective blocks, the percentage of failed blocks is 0.1%.

Application-dependent testing hence can be applied for a particular design on the target FPGA. By this way, the chips in set E can still be delivered to some users; the chips in set C may still be used for some applications; and the lifetime of chips in set D is prolonged. Therefore the cost for production and cost for users are both reduced. Xilinx Easy Path Solution is an excellent example that takes advantage of the application-dependent FPGA testing to reduce the cost of end-market products and digital systems [7].

3.2. Application-Dependent FPGA Test Methods

Application-dependent testing is performed to ensure correct operation of an FPGA-based system. Instead of testing all interconnections and logic blocks of an FPGA chip, application-dependent tests only resources that are used by a particular design.

Application-specific testing is performed with an application being configured on the target FPGA chip. Simple solutions such as testing FPGA based circuits as a classic digital ASIC system have not shown good results because of the following issues [10]:

1. FPGA chips have a large number of flip-flops;
2. A mixed architecture is used in FPGA. For example, a PLB is surrounded by line segments, switches, and memory units;
3. As we introduced in Chapter 2, section 2.3, fault models in FPGAs are different from ASIC chips due to the complexity of the interconnection network.

Therefore, application-dependent FPGA testing is developed specifically for FPGA chips.

Exactly as for MTP, both external test and BIST can be employed in an application-dependent FPGA test. Logic blocks and interconnects being used in a design can be tested separately or at one testing configuration.

3.3. FPGA Application-Dependent Build-in Self-Test

FPGA Built-In Self Test (BIST) is performed by setting up the test environment on the same FPGA chip as the application. BIST is selected as a more suitable approach for application-dependent testing strategy due to several benefits.

One advantage of applying BIST is that no extra external testing devices are required for TPGs and ORAs. We can take advantage of spare areas on the same FPGA chip to implement the required testing circuit. Also, the bottleneck of the FPGA structure, the number of I/O pads is much smaller compared to the number of PLBs or interconnections. By using BIST, the number of test pins and inputs/outputs are no longer strongly restricted during a test procedure.

Generally, BIST procedure for application-dependent FPGA testing includes five steps: configure or reconfigure the FPGA, initiate the test, generate test patterns, analyze the response and read the test result [miron96]. Most application-dependent BIST methods are built on the assumption that the BIST circuit is implemented on the fault-free FPGA area. We take the same assumption for our method.

3.3.1. Built-In Self Test Architecture

The BIST architecture we introduced in Chapter 2, section 2.3 is originally designed for MTP and has been adopted in user's test [10] [18] [12] [3]. In application-dependent testing, multiple ORAs can be used to check different sub-circuits or components instead of considering each PLB or interconnection configuration as in an application-independent test. An example is shown in Figure 3.3[13]. Based on which component is under test, there are BUT (Block Under Test), PUT (Path Under Test) [12] and RUT (Resource Under Test) [7].

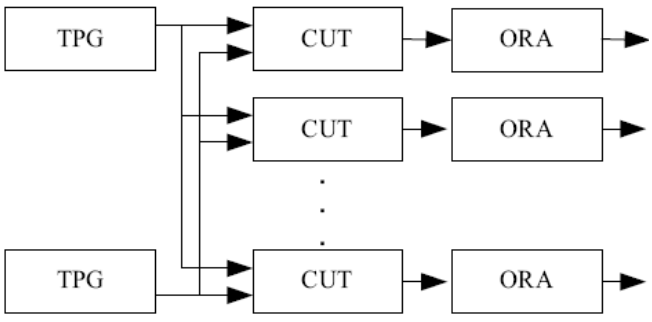


Figure 3.3 BIST Architecture with Multiple ORAs

3.3.2. FPGA Built-In Self Test Techniques

The factors that need to be considered in a FPGA application-dependent BIST method include fault models, the components under the test, the required fault coverage, and the efficiency of the test.

One implementation of FPGA application-specified BIST applies the decomposition technique on user-defined logic such that each subsection can be implemented within one logic block [16]. In this solution, multiple TPGs and ORAs are implemented temporarily for parallel testing to improve the test efficiency [16]. Another approach, called C-exhaustive testing, extends the precious decomposition procedure by combining smaller sub-circuits into groups and each group is tested simultaneously in one BIST session [17].

3.4. Fault Location and Isolation Techniques

Fault diagnosis is one of the important issues in application-dependent FPGA testing since the purpose of the test is to avoid using flawed components on a user-defined circuit. Hence a fault diagnosis algorithm is used to locate the fault location according to information extracted from one or more test sessions.

Solutions are different for application-dependent FPGA testing depending on how big the area each CUT covers. Let a “tile” be an area including one logic block and routing channels around the logic block. If a CUT covers a large number of tiles, fault diagnosis algorithms are likely more complicated in order to find a more precise

fault location. On the contrary, if a smaller number of tiles are covered by each CUT, fault location is much simpler because the faulty area is restricted to only the tiles covered by the CUT.

For an application-dependent testing approach where logic functions are decomposed to be able to be implemented on each logic block, once the fault in the logic block is detected, the fault site is also determined [16]. No extra fault diagnosis algorithm is required in this situation.

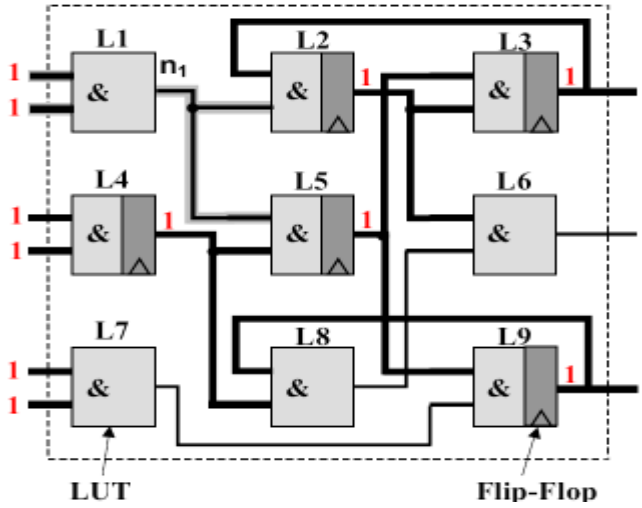


Figure 3.4 An Example of Fault Diagnosis in Interconnection Network Testing

Figure 3.4 illustrates another example of the fault diagnosis process in application-dependent FPGA interconnection network testing [25]. Figure 3.5 gives the fault diagnosis algorithm in pseudo-code of this example [25]. Two test sessions are required for a more precise fault location in this approach. Assuming that there is a stuck-at-0 fault on the network connection n_1, the fault can be observed at primary output L6 in one test session and L3, L9 in the other session. Following

the algorithm given in Figure 3.5, we can deduce that the fault may happen on the routing channel of L1. Once faults are located, the defected area can be avoided during an FPGA mapping.

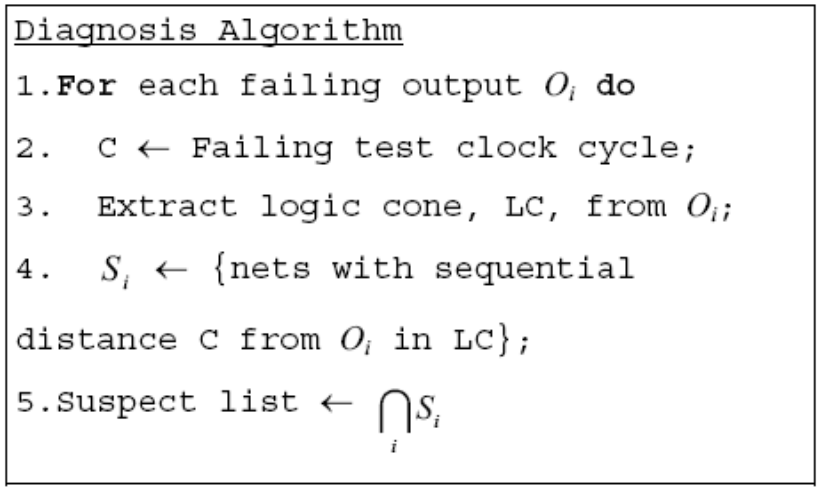


Figure 3.5 An Example of Fault Diagnosis Algorithm

3.5. Test Configuration

CLBs and interconnection network are two key parts of an FPGA chip. Because the significant structural differences between the two components, many researchers have developed different algorithms in an attempt to test FPGAs more efficiently. Nevertheless, there are also solutions developed to test both parts at the same time instead of using separate configurations and test sessions.

3.5.1. Test Model

The concept of "test model" is used in this section to explain how a test configuration environment is set up. A test configuration designed for an application-dependent

interconnection network test generally keeps the original circuit configurations for routing resources, but does not maintain the configuration in logic blocks [3]. On the other hand, the original design in CLBs remains unchanged in a test intended for application-specified PLB testing [25] [16]. In order to test CLBs and interconnections using one testing approach, either multiple configurations are required or both parts of original configurations remain unchanged entirely or partially [12].

3.5.2. Test Configuration Examples

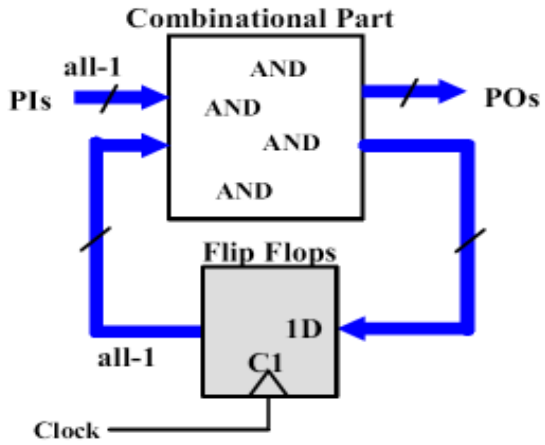
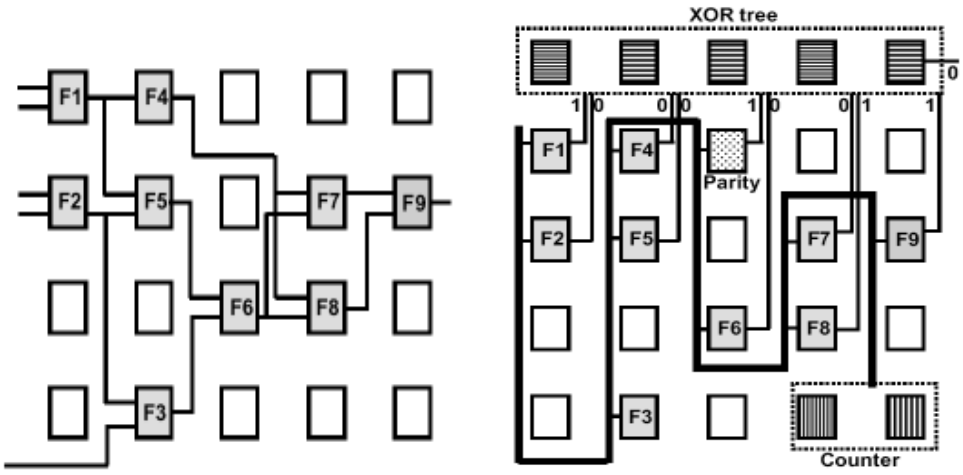


Figure 3.6 Test Model for Interconnection Test Configuration

An example of a testing configuration for application-dependent interconnection network testing is shown in Figure 3.6[26]. In this example, all the logic blocks used in the application are configured to implement logic AND function while interconnections remain as for the original FPGA mapping [26]. This test configuration is designed to detect stuck-at-0 faults, therefore all the flip-flops used in the user application are preset to value 1 and all the primary inputs are applying the

all-1 pattern test vector. Thus any stuck-at-0 fault happened in the interconnection under the test will be detected.



(a) Original Design Configuration (b) Modified Configuration for LUT Test

Figure 3.7 A Test Configuration Example for Logic Block Testing

Another example shows a logic block testing configuration in Figure 3.7[25]. The original design mapping is shown in Figure 3.7(a) [25]. The logic functions are implemented in logic blocks, which are connected to primary inputs and outputs via interconnection network. The modified configuration is shown in Figure 3.7(b) [25], in which the logic block configuration is remain unchanged while the interconnection configuration is modified specifically for the logic block testing [25]. Test pattern generator and output response analyser are also implemented using spare areas.

3.5.3. Multi-Test Strategy

There are two test models available for testing both logic blocks and routing resource. One solution is to combine two sets of configurations in one testing

algorithm and apply them in sequence. For example, the two test configurations introduced in 3.5.2 are combined in [25] to test both logic blocks and interconnections with a set of test configurations. Another solution is to keep the part of the original design both in logic blocks and routing channels. During a test session, one or more sub-circuits can be selected and other parts of the circuit can be tested in parallel or sequentially by taking advantage of the re-configuration ability of FPGA.

A test strategy for detecting delay faults is shown in Figure 3.8[12]. This approach first isolates each target Path(s) Under Test (PUT) from the rest of the circuit and selects as many parallel paths as possible in one test session, then switches to other possible paths in the next session until all the paths are tested exhaustively. Numbers of reconfigurations are required in order to test all possible paths in a circuit [12]. For example, in the first test session, paths {dAEJLy, cAEJLy, cEJLy, fBFJLY, hCGKMz, jCGKMz, nDGKMz,qHKMz} are selected and tested at the same time. A simplified test configuration for one selected path is illustrated in Figure 3.9 where sequence generator is the TPG and clock is used by the ORA [12].

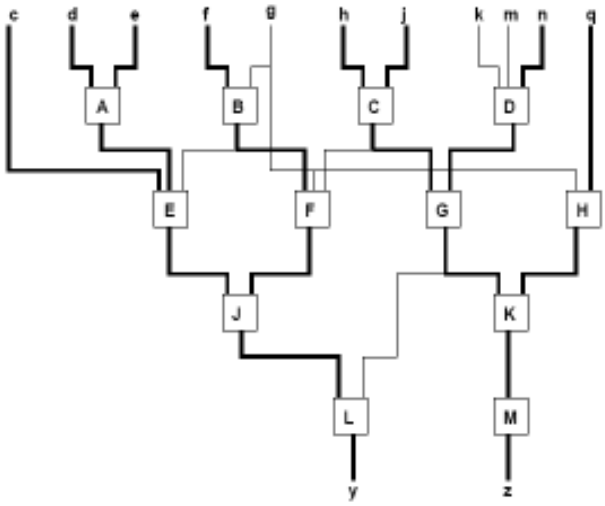


Figure 3.8 Test Configuration for Delay Fault Testing

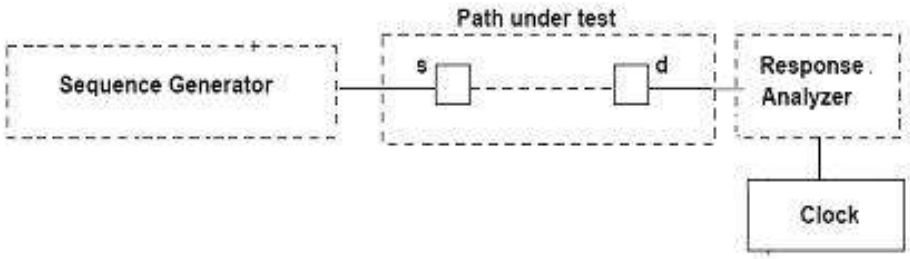


Figure 3.9 A Test Configuration for Delay Fault Testing

3.6. Circuit Partitioning Application in FPGA

Circuit partitioning techniques have been used in VLSI layout and design for many years. These techniques have been adopted in FPGA based logic design and configuration recently. Since FPGAs allow programmable configuration and reconfiguration, the focus in FPGA based circuit partitioning is no longer the consideration of modularization and module reusability.

According to the available research, circuit partitioning techniques related to FPGA are used in VLSI prototyping using FPGA [20] [22], dynamic reconfiguration procedure in FPGA [21] and FPGA mapping for large VLSI hierarchical blocks [23].The general purpose in FPGA circuit partitioning is to 1) overcome the FPGA mapping bottleneck of Input/Output pin number limits, 2) reuse logic modules in a way that either time or area complexity can be reduced during an operation, and 3) enable small FPGA usage on big VLSI design.

An example of a graph model used in FPGA mapping is shown in Figure 3.10[20]. In this example, a large circuit needs to be partitioned into small sub-circuits such that each sub-circuit can be implemented on a single FPGA [20]. I/O restrictions of FPGA and routing requirements of the large circuit are considered in this algorithm and the graph model reflects the requirement of the partitioning. The min-cut algorithm is applied in this approach to eliminate the connection points between FPGAs or Field Programmable Integrated Circuits (FPICs).

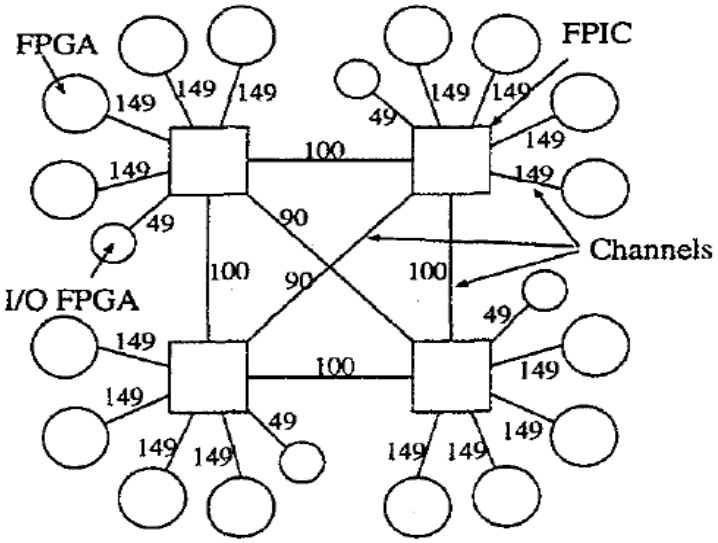
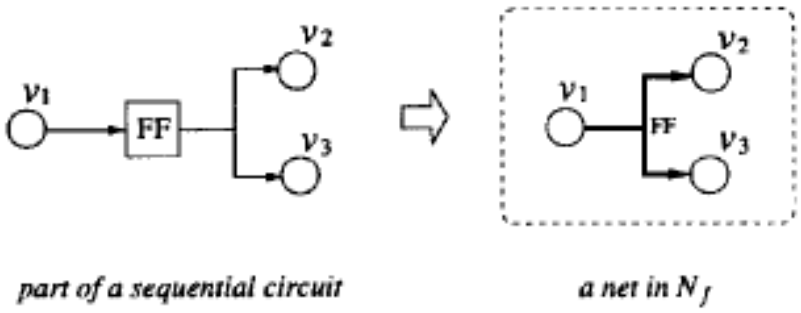
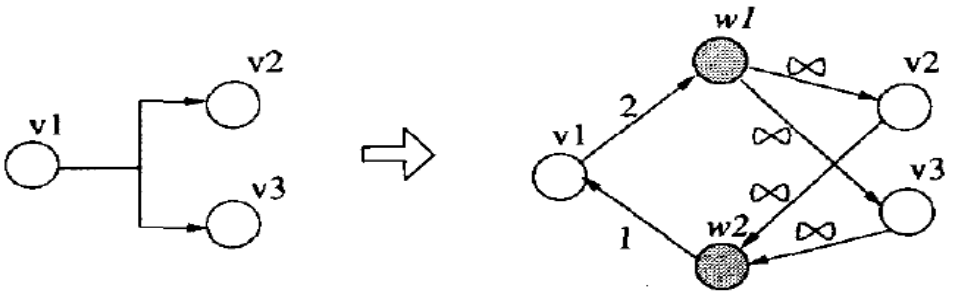


Figure 3.10 Topology Graph for Large circuit FPGA Implementation

Minimizing cut points has always been an important issue in circuit partitioning and the min-cut algorithm is often adopted by circuit partitioning methods as a typical algorithm. A partitioning algorithm is developed for dynamic reconfigurable FPGAs to reduce the time complexity of reconfiguration [21]. Figure 3.11(a) shows a graph model for a circuit network and Figure 3.11(b) shows the advanced weighted graph model for the sequential circuit, where vertex v_i represents gates, w_i represents Memory Elements(ME) used by sequential circuits in the FPGA[21]. The later graph model is used in the partitioning algorithm for the dynamic reconfigurable FPGA. The weight of edges is calculated by memory capacity [21].



(a) Graph Model of A Simple Circuit Network



(b) Circuit Graph Model with Added nodes and Weight

Figure 3.11 Graph Model of Circuit Network

Figure 3.12 shows an example of a biparitioning where min-cut technique is applied in the partitioning algorithm to minimize the connection set as well as the MEs used in each sub-graph [21].

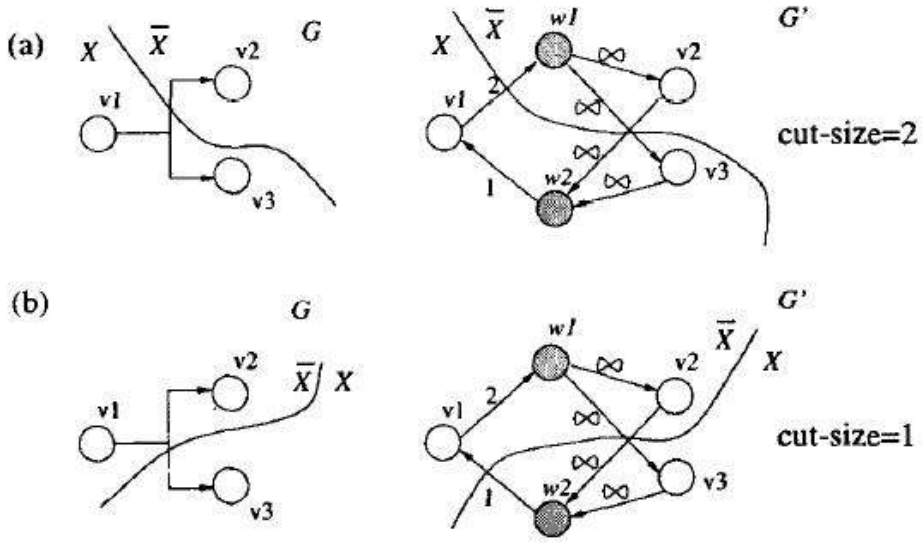


Figure 3.12 Example of the Min Cut Partitioning Algorithm

3.7. Summary

The purpose of application-dependent FPGA testing is to identify the defective or damaged parts of an FPGA chip. By doing so, the cost of FPGA manufacturing and applications can be reduced and the lifetime of an FPGA chip may be prolonged. Due to the particular architecture of an FPGA, application dependent FPGA testing approaches are developed differently from those for ASICs.

Application-dependent FPGA BIST methods have adopted the architecture from MTP. Multiple sub-circuits, interconnections or logic blocks are tested in parallel for the sake of the testing efficiency. Large circuits can be decomposed into small functions or units for the purpose of high fault coverage testing and fault location. Testing configurations are developed to test logic block and interconnections separately or both at the same time.

Partitioning techniques have been used in digital design and testing for many years. Its applications in FPGAs cover many areas from floor planning to dynamic re-configuration. This technique is also adopted in the proposed algorithm that is described in the following chapter.

4. A Novel Application-Dependent FPGA Built-In Self Test Strategy

This chapter presents a new application-dependent FPGA BIST strategy, in which a recursive circuit partitioning technique is employed. The purpose of the strategy is to provide an FPGA configuration solution for an arbitrary user-defined circuit. This goal is achieved by identifying unusable blocks of an FPGA chip and avoiding using them during the FPGA configuration.

This chapter describes the proposed BIST structure, the procedure of this particular testing method and a recursive partitioning algorithm. The BIST architecture is presented in section 4.2. Section 4.3 explains the test configuration, which shows how the test strategy works. The main focus of this chapter is to present the recursive circuit partitioning method in detail. A structural model and un-weighted directed graph are applied to perform the partitioning process. Improving the fault coverage with approximately minimum cut points is the attempted goal of the algorithm, which is presented in section 4.4.

4.1. Built-In Self Test Architecture

BIST strategy is selected as an appropriate test method for the proposed approach because the flexible programming ability of an FPGA allows the BIST circuit being implemented on unused area of an FPGA chip. By using BIST, this test requires no

extra hardware. For an arbitrary user-defined circuit, different TPGs and ORAs can be designed or selected for the particular application.

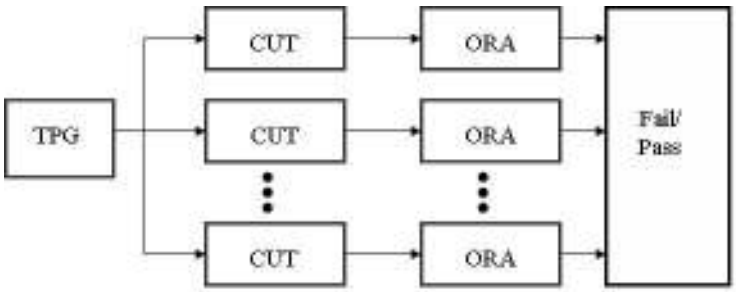


Figure 4.1 BIST Architecture for the Application-Dependent FPGA Testing

Figure 4.1 shows the BIST architecture design for the application-dependent FPGA testing strategy. In this BIST structure, only one TPG is used to provide test vectors for all sub-components. According to the test vectors required, the TPG can be implemented by an exhaustive generator such as an address generator or a random or pseudo-random test pattern generator such as an LFSR. The size of the TPG depends on the size of the selected test set. Each CUT is a sub-circuit derived from the circuit partitioning algorithm introduced in section 4.4.

The ORA can be designed as normal ASIC test output analyzer such as a signature comparator. The output of the ORA is either a fail or a pass signal, which we use to determine whether the area covered by the test is defective. With a small size CUT, a small area of FPGA can be reasonably labelled unusable and thus be avoided in an application. All the sub-circuits can be tested in parallel and the test result indicates the areas that fail the test. In order to minimize the configuration area and complexity

for each sub-circuit, maximum input and output numbers have to be decided in the circuit partitioning process.

4.2. Test Configuration

This test strategy is based on the following assumptions:

- The FPGA area used for the BIST circuit implementation is defect-free.
- The user-specified circuit is well designed with correct logic implementation and mapping.
- All input and output pins on the target FPGA are in a functional state.
- During the partitioning process, assume no fault is added to the partition node.
- If faults are detected within an area, the area is avoided in the future programming and routing. No further fault location techniques are required.

Time related faults are not considered at this stage. As we discussed in Chapter 2, section 2.3, many open, short and bridging faults can be modeled by stuck-at faults. In this test strategy, all applicable open, short, stuck-at and bridging faults are subject to be tested as stuck-at faults. Each area that a sub-circuit built on is called a “block.” Each block covers a small number of tiles, the notation introduced in section 3.4.

Figure 4.2 shows the original and modified configuration during the test. Sub-circuits are obtained by circuit partitioning graph and mapped on the FPGA chip where each sub-circuit occupies one block. Sub-circuits are viewed as a black box and are eventually connected by the completion of the test. Connection lines are indicated by

dashed lines. If the test gets an all-pass result, the original design can be reconnected by reconfiguration or partial configuration.

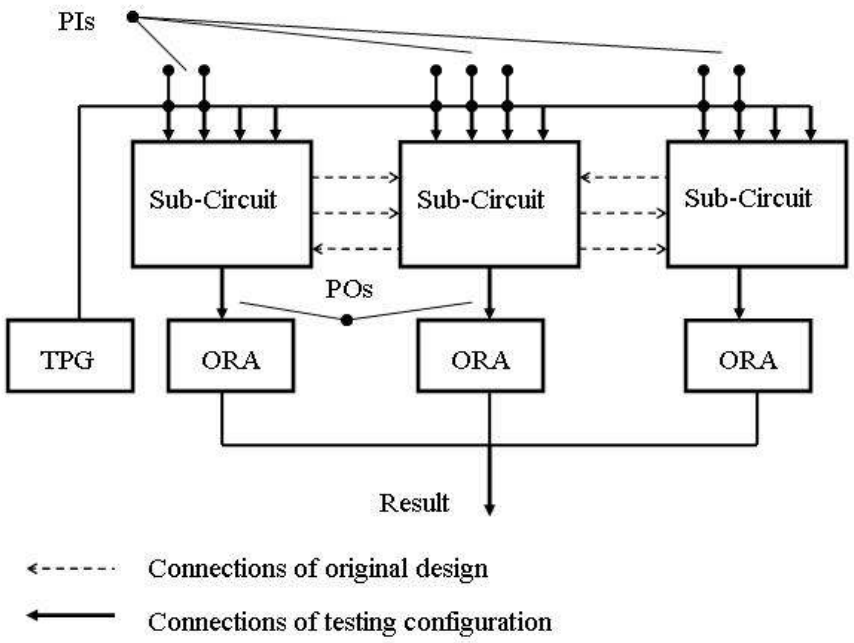


Figure 4.2 Test Configuration for Application-Dependent FPGATesting

In the case that a "bad" block is detected, the sub-circuit programmed on the block is re-programmed on a free block. Another test session is carried out on the revised configuration. This case is shown in Figure 4.3. Suppose faults are detected in block **A,** where sub-circuit a is implemented on, sub-circuit a', which is a duplicate of a, is implemented on block **E** and connected to BIST circuit, shown in dashed line. Sub-circuit a' can be tested in another test session. The re-configuration applied to connect the original design is performed once all blocks used in the application pass the test.

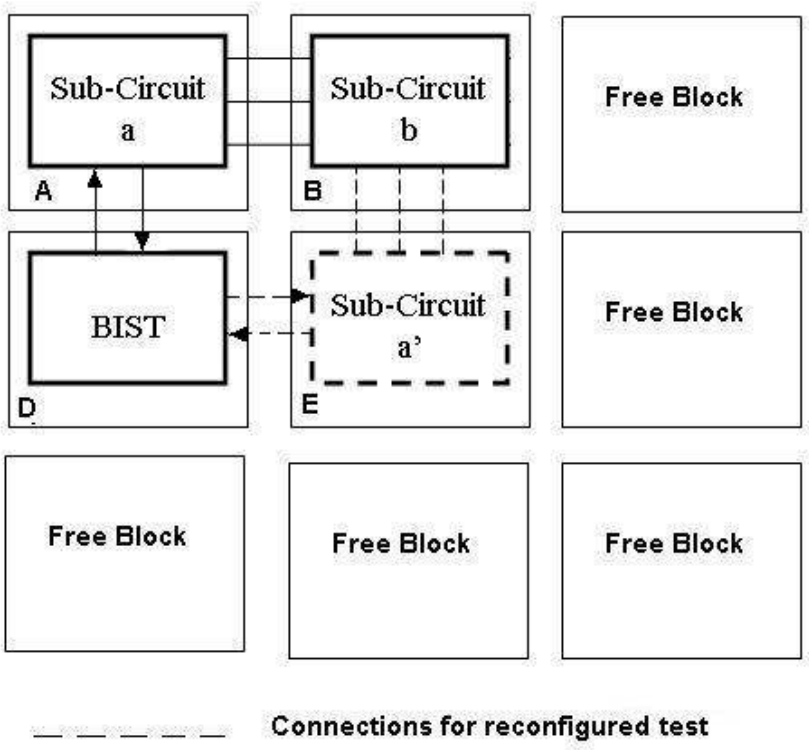


Figure 4.3 Modified Configuration of a Sub-Circuit

4.3. Recursive Circuit Partitioning Algorithm

Testability and modularity are generally considered in ASIC circuit partitioning. Circuit partitioning is also restricted by considerations of module reusability and area overhead. However, an FPGA is known to be a flexible device that can be reconfigured easily by software programming. Hence, circuit partitioning on an FPGA can be considered without worrying about module reusability. Also, with the capacity of an FPGA, which is rapidly growing with the development of IC technology, a medium sized FPGA chip is sufficient to accommodate a reasonably sized digital system and a BIST circuit. Therefore, considering a small or a moderate

design, the restriction of the Circuit partitioning for testing is relaxed to only testability in our new algorithm.

The purpose of the recursive partitioning is to produce a set of sub-circuits. The solution of the algorithm is used in FPGA implementation for BIST test configuration. This goal is accomplished by finding appropriate cut points that can be also used to improve the controllability and observability in a circuit testing. Minimizing the size of the connection set is also an important issue in that test complexity is related to the size of the sub-circuit and the number of inputs and outputs of the sub-circuit. This circuit partitioning algorithm focuses on improving fault coverage of an arbitrary circuit by finding approximately optimized cut points and minimizing the number of cut points between sub-circuit pairs at the same time.

4.3.1. Network Modelling

The testability of a digital circuit is directly related to the difficulty of controlling and observing the logical values of internal nodes. A set of signal points can be selected and their values can be checked to improve controllability and observerbility of a digital circuit. Therefore, fault coverage of a user-defined circuit is increased by selecting appropriate test points for fault detection. A structural model of the circuit network is adopted by this algorithm in order to search for appropriate signal points. An un-weighted directed graph model is selected for our partitioning algorithm. Figure 4.4 illustrates the network model used in the recursive circuit partitioning algorithm.

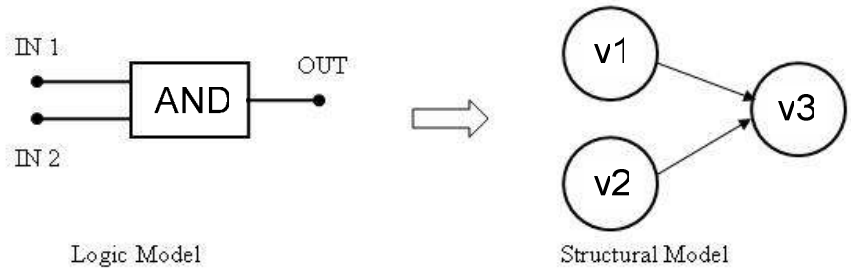


Figure 4.4 Structural Model of a Simple Circuit Network

As illustrated in Figure 4.4, a simple circuit network with one AND gate and three signals, two inputs, and one output. The structural model takes signals and transfers them to nodes in a graph and the relationships among those nodes are represented by edges. For example, in the graph structure in Figure 4.4, IN1, IN2 and OUT are transferred to three nodes v_1, v_2 and v_3 where v_1 and v_2 are essential to create node v_3. We call v_1 and v_2 are dependent signals of v_3. Figure 4.5 shows a graph constructed from a circuit design. Internal signals, those labelled starting with "S", are transferred to graph nodes such that all internal signal nodes have both incoming and outgoing edges similar to nodes v_4 and v_5, which are transferred from signals S1 and S2.

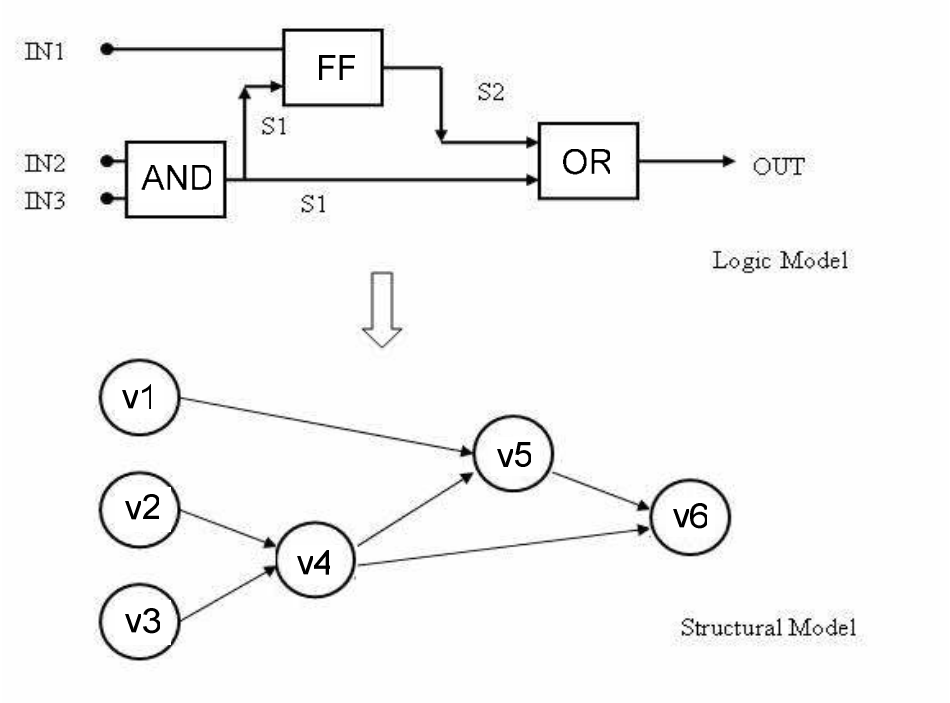


Figure 4.5 Structural Model Example of a Digital Circuit

The naming convention of circuit graphs follows the principles:

- "G" or "g" is used referring graphs or sub-graphs;
- "v" represents nodes or vertices and "V" is a set of "v";
- "e" represents edges and "E" is a set of "e";
- "p" represents directed paths from one node to another one, or from one graph to another one, and "P" is a set of "p";
- "c" is used to name cut points, and "C" is a set of "c".

Due to the nature of the graph presentation of digital circuits, there are several special characteristics of the graph model that is introduced, as compared to a regular network graph.

- Circuit input and output signals are indicated as node attributes. These nodes are treated differently from other nodes that represent internal signals.
- Only internal signals have both incoming and outgoing edges.
- Dependent nodes of a node cannot be partitioned into different sub-graphs in that the operation causes the change of original logic functions.

4.3.2. Problem Statement

A circuit can be represented by a graph $G = (V, E)$, where V is a set of nodes and E is a set of edges that connect nodes in V. Each node v in V has a set of attributes that describes the node:

1) Node identification uniquely labels each node.
2) Node type indicates the signal type of the node such as input, output, and intermediate signal.
3) A set of next nodes specifies fan-out signals propagated from this node.
4) A set of the in-coming nodes consists of all nodes that the current node depends on.

Each graph object also has a set of fields to provide information about the circuit graph:

1) Input and output numbers;
2) Graph size and circuit size, which are the number of total nodes and the number of gates respectively;

3) Fault coverage, which can be different for different sets of test patterns.

The problem of recursive circuit partitioning is to recursively partition a circuit graph $G = (V, E)$ into multiple small sized sub-circuit graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$,…, $G_n = (V_n, E_n)$ where $G_i \cup G_j = \phi, for(i \neq j)$. A path p is defined as $p = (V)$ where V is set of nodes that belong to the directed path p.

The objectives of the partitioning are to:

1) Maximize the final total fault coverage of the original circuit graph and each sub-circuit;
2) Minimize the number of cut points which connect sub-graphs to each other;
3) Balance the size of each sub-circuit graph such that each circuit block, which accommodates one sub-circuit, has a reasonable number of tiles.

The above requirements help obtain a better solution of the FPGA configuration where the number of detectable faults and fault coverage can be effectively increased if possible, test complexity can be minimized, and fault location can be identified. All three constraints should be considered simultaneously as they affect each other. The main evaluation factors for each resulting sub-circuit graph are the size of the circuit and the fault coverage of the circuit. As this algorithm is designed to improve the fault coverage without decomposing logic functions, our main concern is with structural factors.

4.3.3. Recursive Partitioning

The purpose of the partitioning is to find appropriate cut points in order to divide an applicable graph into two or more separate parts. The partitioning process continues recursively until the solution reaches the final state. The network min-cut technique is well known and widely applied for finding minimum cut in network graph partitioning. The recursive partitioning algorithm adopts the basic concept of minimum cut and combined with fault coverage evaluation in a circuit graph partitioning. However, structural factors are considered instead of edge weight, which is the main factor in network min-cut algorithm, during the selection of cut points with the goal of improving the overall fault coverage and minimizing the number of cut points.

The strategy of this algorithm is to continue the partitioning process recursively until the solution reaches the final state. Figure 4.6 shows the process of the algorithm. During each partitioning process within the recursive block, an optimal result based on certain constraints is selected from the number of candidates produced by performing iterative partitioning processes. The iterative procedure produces a set of result candidates by selecting different sets of cut points. Each solution is evaluated by checking the fault coverage, the number of cut points and the size of each sub-circuit graph. A solution is chosen from the number of candidate solutions as a final solution of the partitioning process.

A final state must be set for sub-circuits. In this algorithm, each resulting sub-circuit graph is evaluated for the size of the circuit and the fault coverage. If a circuit reaches the final state, it is collected by a solution set and the recursive step returns, otherwise, the recursive partitioning continues until the base case is reached.

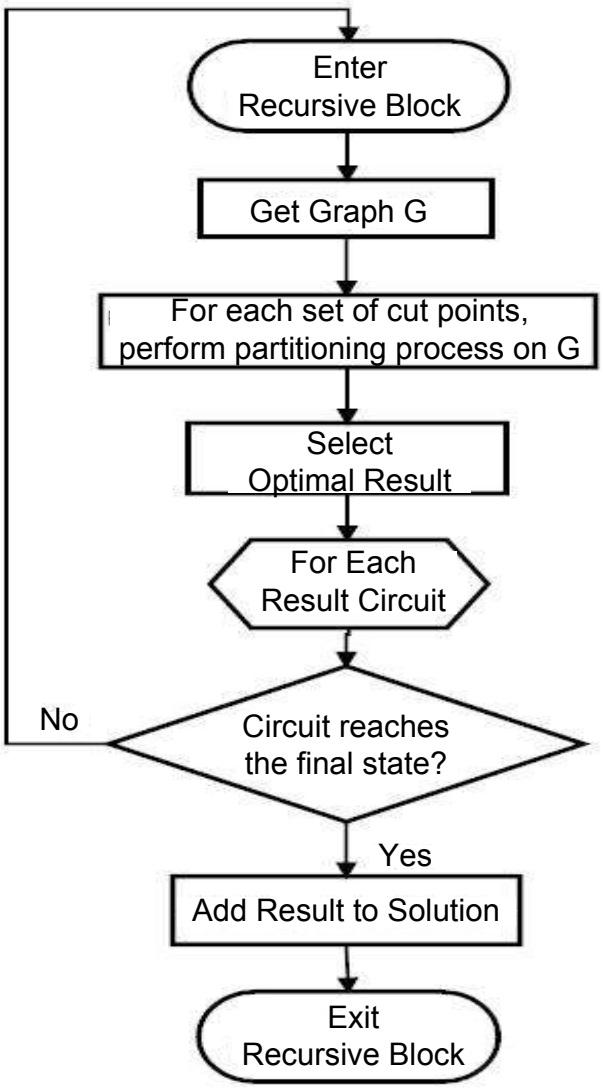


Figure 4.6 The Recursive Circuit Graph Partitioning Process

Two methods are applied in the partitioning process. One method, “check point cut”, is applied on sub-circuit graphs that are within the restricted size, but with lower fault

coverage. Another method, “bipartitioning cut”, is performed while a circuit exceeds the size restriction.

The result of a check point cut is a circuit with additional test points produced by the “Test Points” method, which is introduced in Chapter 2, section 2.2.3. An example of the check point cut method is illustrated by Figure 4.7.

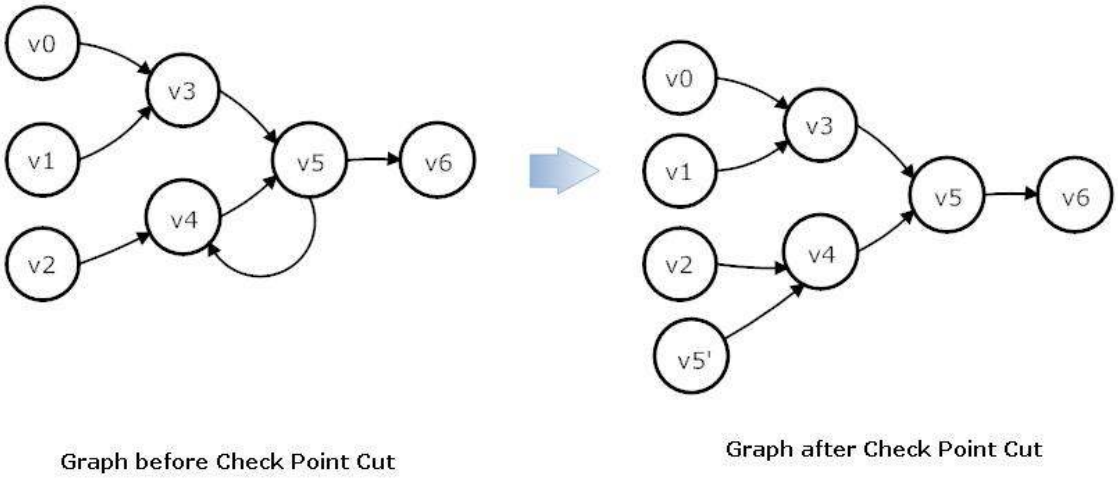
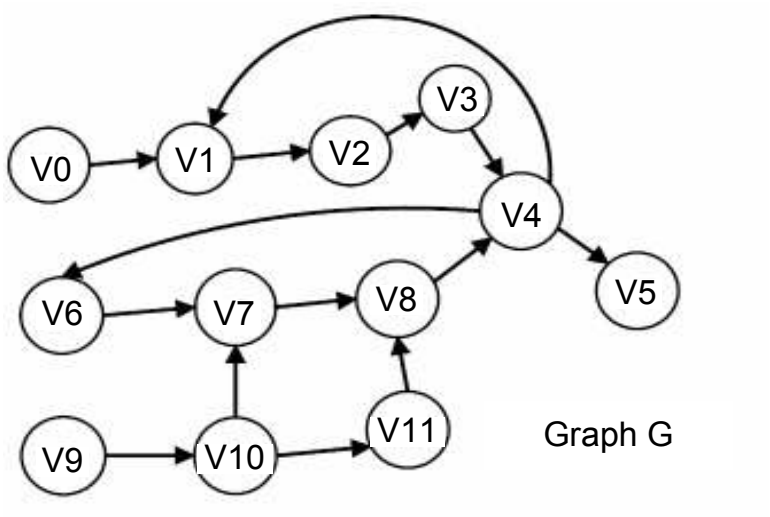


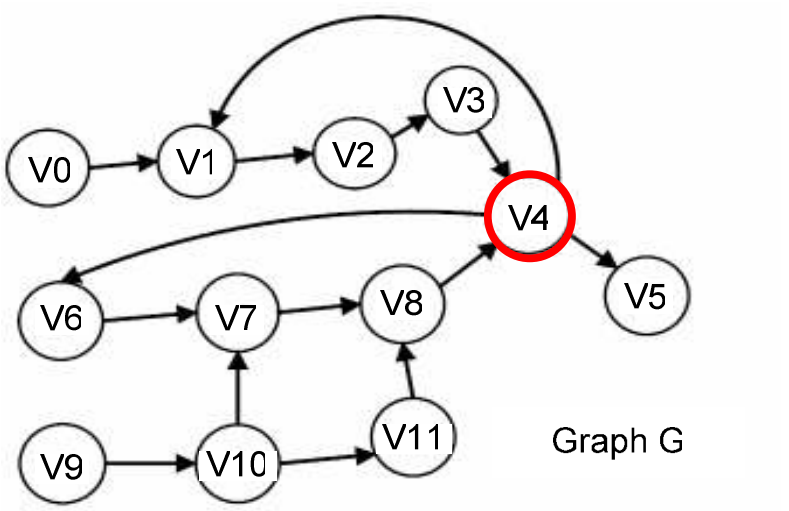
Figure 4.7 An Example of Check Point Cut

Recall that input signals and fanouts are check points (Chpater2, section 2.2.1). In this check point cut method, only fanouts are considered as candidate cut points. The node that has the larger number of fanouts has a higher priority during the cut point selection. The final result of a check point cut is selected from a set of results by choosing a set of candidate cut points. In Figure 4.7, v5 has two fanouts and one node is divided from v5 to produce a new node v5’. The new circuit graph now has 4 input nodes where v5’ is tested to improve the controllability of the testing.

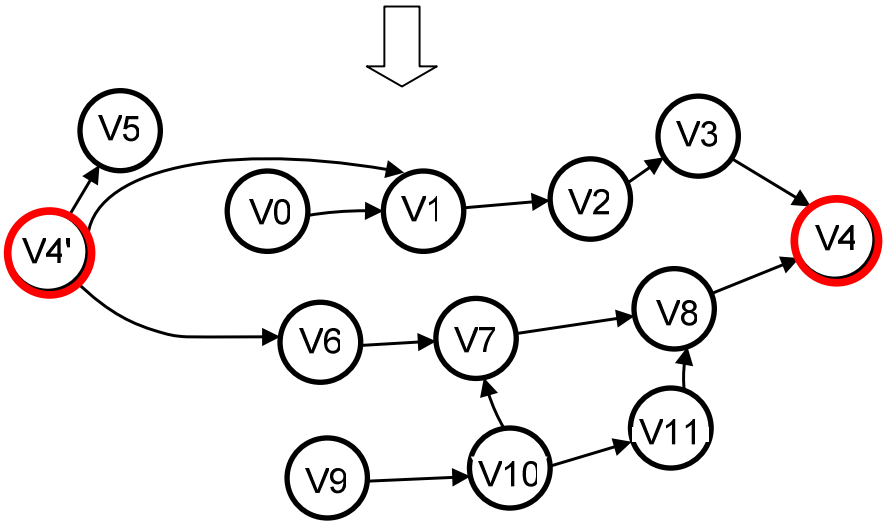
The result of the bipartitioning cut is two sub-graphs. A set of candidate results can be produced by selecting different first cut points and an optimal solution with smaller number of cut points is selected by first evaluating the number of cut points and then comparing the fault coverage. An example of the bipartitioning designed for this algorithm is illustrated in Figure 4.8.



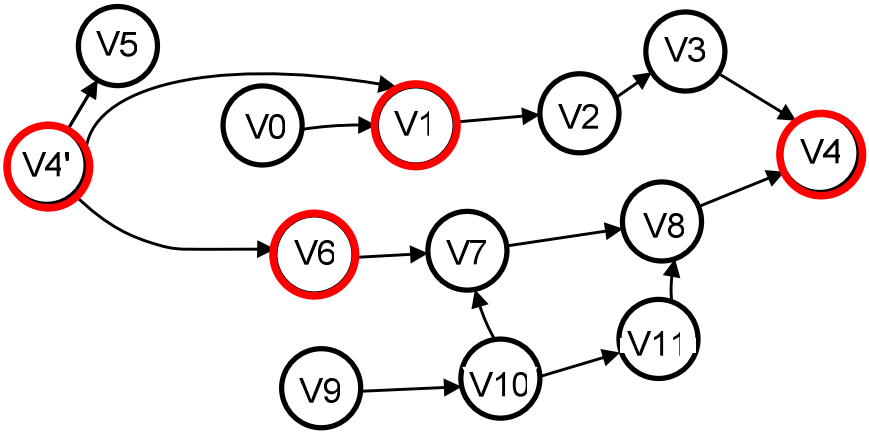
(a) An Original Graph G



(b) Decide the First Cut Point

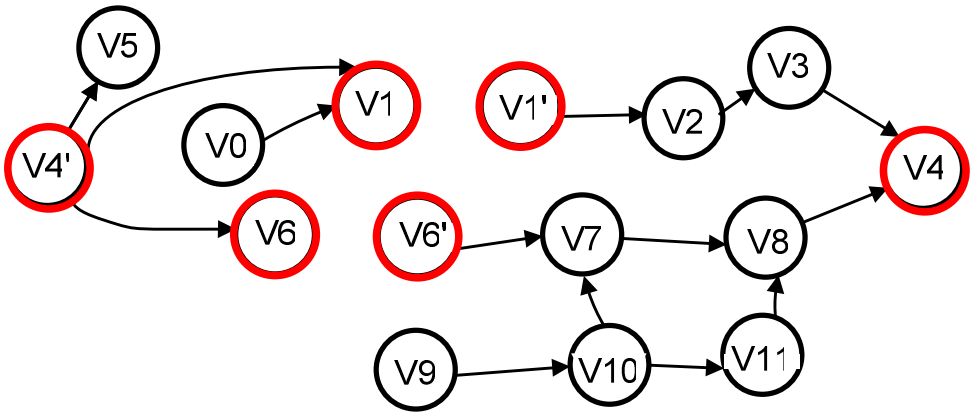


(c) A Node Partitioning



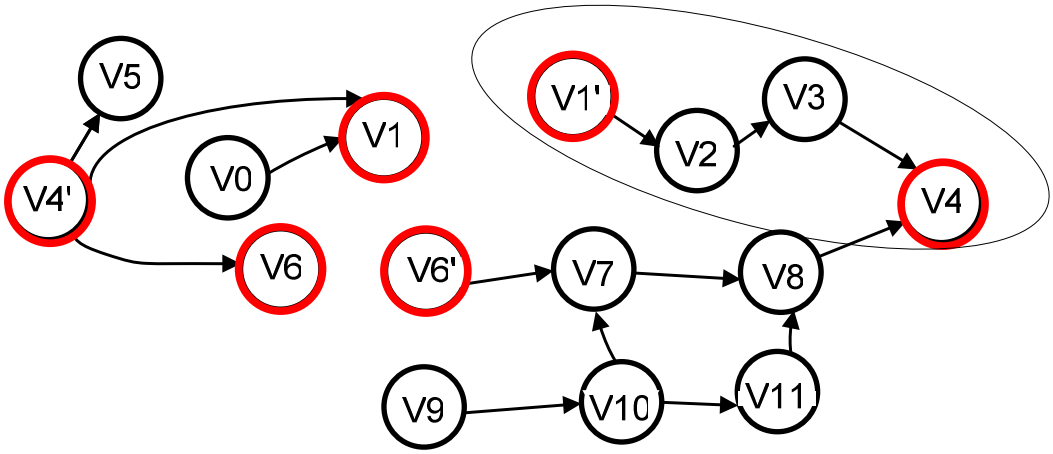
Cut Point Set C={ v4}

(d) Directed Path Cut Point



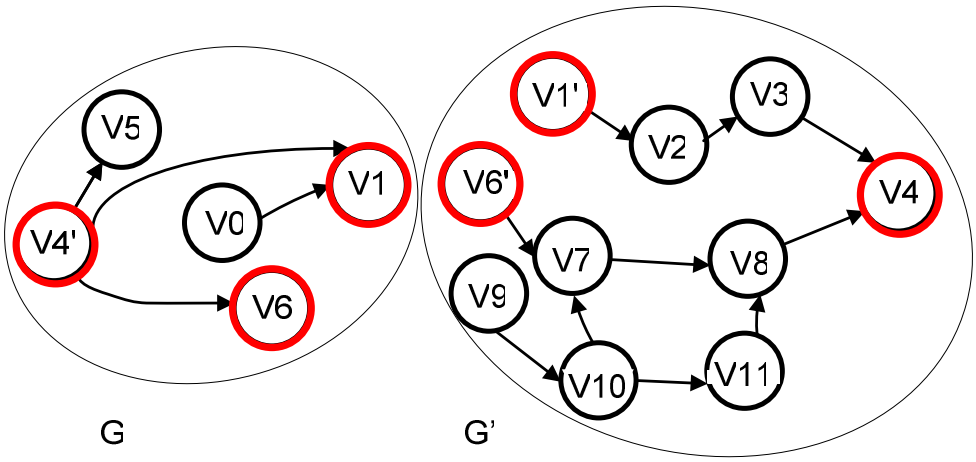
Cut Point Set C={ v4, v1, v6}

(e) Directed Path Partitioning



Cut Point Set C={ v4, v1, v6}

(f) Contraction Process



G=GUG' Cut Point Set C={ v4, v1, v6}

(g) Two Disconnected Sub-Graphs and One Cut Set

Figure 4.8 Example of a Bipartitioning

The bipartitioning process has the following five major steps.

1). Find the first node v as a cut point that is likely to result in a smaller number of cut points of G. If a vertex has more outgoing edges in a circuit graph, it is likely a point that leads to more cycles of the graph. Therefore, selecting this vertex as a cut point may result in fewer cut points in order to bipartition a graph.

Let $f(v)$ be the fan-out degree of node v, then for two nodes $v1, v2$, if $f(v1) > f(v2)$, then $v1$ has higher priority to be selected as a cut point during the first selection. Figure 4.8(b) illustrates the process.

2). Separate v into two nodes v and v'. Initialize G' and G'' as two target sub-graphs consist of nodes v and v' respectively. The result of the partition is that node v has only incoming edges and node v' has only outgoing edges. In

this way, one node acts as a control signal and the other one acts as an observer signal. The original node is put in a set C, where all cut points are collected. Figure 4.6 (c) illustrated the process.

3). Look for all possible paths from v' to v. If no directed paths found in this step, step 3 is omitted and the process goes to step 4 directly. Cut those paths with selected nodes, which are the optimized nodes to break the paths with minimum cut points number. In this step, nodes with higher degree of fan-outs are also examined first. The principles of node selection are discussed in the following sub-section. The cut points are also put in C for later calculation and evaluation. This process is shown in Figure 4.6 (d) and (e).

4). The contraction algorithm is applied in this step to break off all the connecting points of selected nodes v and v'. Some cases are discussed in the following sub-section. Figure 4.6 (f) and (g) illustrate a simple example of contraction with no left connections from step 3 between node v and v'.

5). Evaluation of the generated sub-graphs is done at the end of each iteration. For the first candidate solution, only the size ratio of two sub-graphs is considered to avoid the extreme situation such as too few nodes being included in one sub-graph. Otherwise, the size of sub-graphs, number of cut points and fault coverage of each sub-circuit graph are compared among candidates to ensure the quality of partitioning solutions.

In the last step, the solution selection is based on two main factors of the algorithm, the number of cut points and the fault coverage of a sub-circuit that is represented by a graph. In order to reduce the test complexity of each sub-circuit, the number of cut points is the major factor considered in this algorithm. For all sub-circuits that satisfy the fault coverage threshold, a solution with the minimum number of cut points is selected as the final solution.

4.3.4. Node Selection and Contraction Cases

With different structural layouts within a graph, different strategies need to be considered in order to achieve a better result.

During a biparitioning process, suppose a set of directed paths P is found between the two subject nodes. For a path p in P such that p does not have common parts with other paths in P, each node on p can be selected as a potential "good" cut point, as shown in Figure 4.9 (a). In some situations, nodes on the path should be examined in order to obtain an result with smaller number of cut points. As illustrated Figure 4.9 (b), path p has node v', v_1, v_2, and v''. If we choose node v_1 as a cut point, then the graph is divided into two parts at a clean cut. The cut point number is increased by one. Choosing node v_2, however, will cause the cut point number to be increased by two. In this case, node v_1 and v_2 should be checked to find further connectivity with v' and v''.

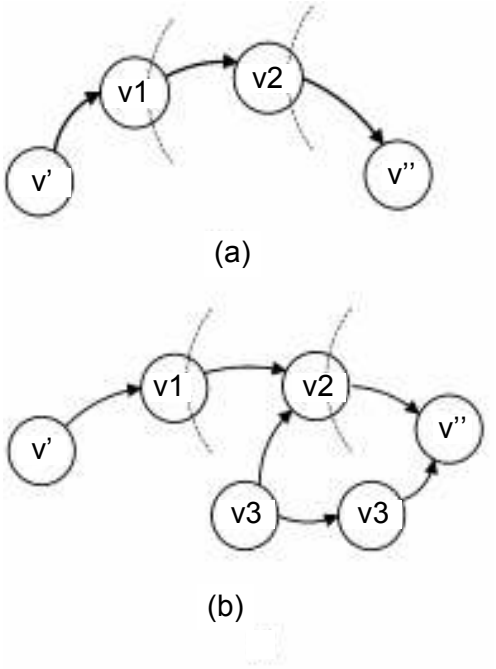


Figure 4.9 An Example of a Single Path Partition

The situation of multiple paths with common nodes is another issue that needs to be considered carefully. An appropriate node selection scheme can result in a better solution. One of multiple path cases is discussed in Lemma 1.

Lemma 1: for a graph G such that G is subject to be partitioned into two sub-graph G' and G'', there is a set of paths $P = (p1, p2, ...pk)$ where all paths in P are connections of two nodes v' and v'' such that $v' \in G', v'' \in G''$. If there is a set of nodes $V = (v1, v2, ...vk)$ where $V = p1 \bigcap p2 \bigcap ... \bigcap pk$, then each node in V can be selected to minimize the cut points to disconnect P.

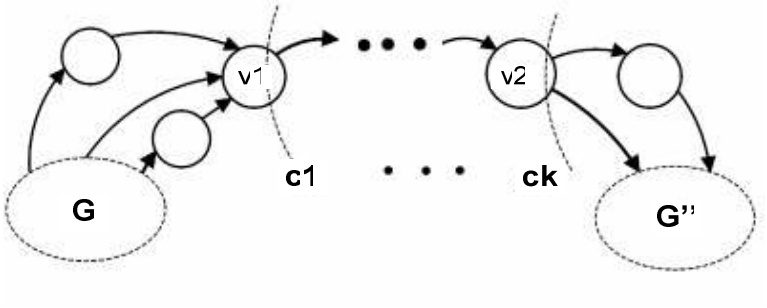


Figure 4.10 Multiple Paths with Common Nodes

As an example shown in Figure 4.10, there are multiple paths from G' to G'', so selecting each point from c1 to ck will produce only one cut point in order to disconnect the two graphs. More than one cut point will be generated if other nodes are selected.

Another multiple paths case is shown in Figure 4.11. In this example, three paths connect graph G' to G''.

Let $P_1 = (v', v_2, v_3, v'')$, $P_2 = (v', v_1, v'')$, $P_3 = (v', v_1, v_2, v_3, v'')$. P_1 and P_2 share no common node, but they both share a node with P_3. No matter which common node of P_1 and P_3 or P_2 and P_3 is selected as cut point, at least two cut points will be produced.

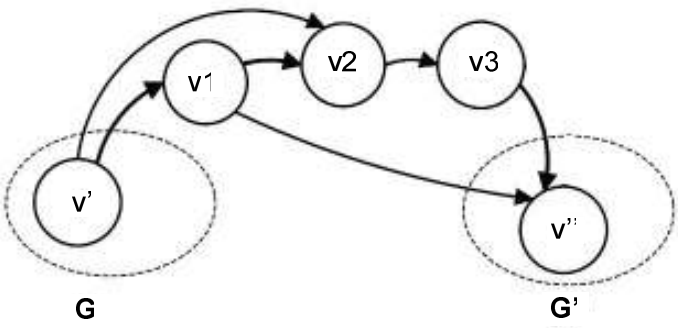


Figure 4.11 Multiple Paths with Different Common Nodes

Lemma 2: If a set of paths P consists of some paths that connect two sub-graphs, g1 and g2, such that no common node on all paths exists, but some of paths share some nodes with other paths. If no single path exists in P, let N be the number of paths, then there are at least 2 and at most N-1 cut points produced in order to cover all paths in P.

There are also patterns that can be recognized and therefore the algorithm can be simplified by avoiding considering the non-exist situations. Figure 4.12 is an example of this type of case.

Lemma 3: If a selected node v is divided into two duplicates, v' and v'', such that v' only has incoming edges and v'' only has outgoing edges, then there does not exist a node vi such that vi has one or more directed incoming edges from both v' and v'', or outgoing edges to v' and v'' respectively.

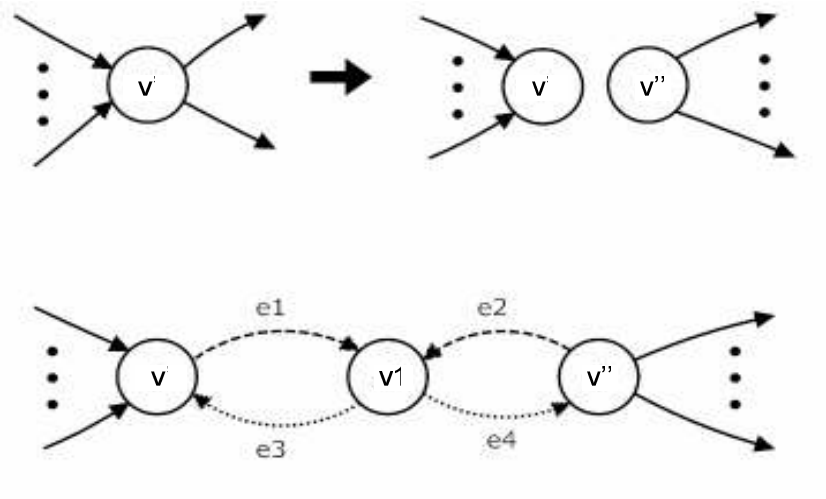


Figure 4.12 An Example of a Node Connectivity Case

As shown in Figure 4.12, node v is divided into *v'* and *v''* such that they only have edges with one direction. If *e1* and *e2* both exist in this graph, then there is a contradiction with the claim statement. Neither *e3* nor *e4* can exist at the same time.

When all the paths are disconnected in a bipartitioning, contraction is applied to form two disconnected sub-graphs. Some connections are required to be disconnected during the contraction process.

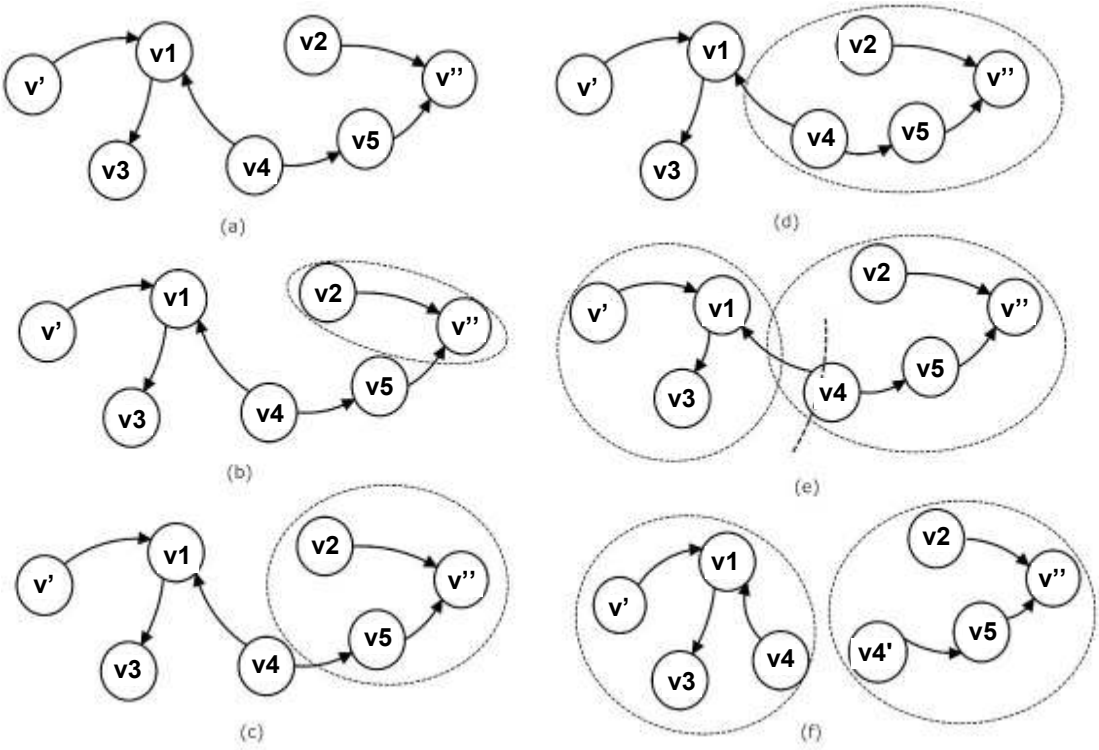


Figure 4.13 An Example of Contraction Case

Figure 4.13 describes the contraction process on a particular circuit during bipartitioning. In this example, no directed path exists from v' to v''. v'' is selected to start the contraction first. It takes all nodes that directly connect to it until no direct edge available and no dependent node is left out. Another end node v' then starts the contraction. When the contraction is collecting dependent nodes of $v1$, a node $v4$ is found to be already taken. In this case, an outgoing signal of $v4$ is separated from the original signal and collected by the contraction process starting with v'.

4.4. Summary

This chapter introduces an application-dependent FPGA testing strategy by presenting the BIST architecture, testing method, partitioning algorithm and case considerations. In this strategy, a user-defined circuit is first evaluated and partitioned into an appropriate sub-circuits group. A BIST circuit is designed for the particular sub-circuit group. The sub-circuits and BIST circuit are implemented on an FPGA chip. Testing is initialized and proceeded until a configuration solution is found. Reconfiguration is required for the final connection of the user specified application.

No different methods and extra configuration are required for testing logic block and interconnection separately. Fault location is also accomplished without additional algorithms. The flexibility of FPGAs allows easy reconfiguration of sub-circuit in case that faults are detected and a certain area is required to be avoided. Using the method of partitioning a large and complex circuit to multiple smaller and simpler sub-circuits, this strategy attempts to achieve the goal of implementing each sub-circuit on a small area of FPGA board where high fault coverage can be obtained at the same time.

The core of the strategy is a recursive circuit partitioning algorithm. A graph model is used in the algorithm. A check point cut method is applied to simply improve fault coverage and a bipartitioning cut is applied to reduce the size of each sub-circuit and improve the fault coverage at the same time. Priority and random selection are

applied while applicable for cut point searching during a partitioning process. The experimental setup and implementation of the algorithm are described in Chapter 5.

5. Experimental Setup and Implementation

The testing strategy introduced in Chapter 4 is subject to being applied to an FPGA configuration for an arbitrary user-defined circuit. The objective of the recursive circuit partitioning algorithm, which is the core of the testing strategy, is to provide appropriate sub-circuit configuration solutions. The overall fault coverage of a solution should be improved and total number of cut points should ideally be minimized during the process. The experimental setup and implementation presented in this chapter are designed and used to conduct experiments using the circuit partitioning method.

We start with the experimental setup that is explained in section 5.1 to introduce the specification, design of the experiments and resources involved in the experiments. Software design and implementation for the partitioning algorithm is discussed in section 5.2. We also discuss the anticipated results in this section. Section 5.3 summarizes the chapter.

5.1. Experimental Setup

This section introduces the requirements, design and resources that are employed for the experiments.

5.1.1. Specification of the Experiments

The core of the proposed application-dependent FPGA testing strategy is a recursive circuit partitioning algorithm, which provides configuration solutions for the application-dependent FPGA testing strategy. The main purpose of the experiments

is to verify and evaluate the algorithm using the methods created for the experiments and the detailed results produced by the experiments. Two sets of the experiments are designed for the purpose. Fault coverage and the number of cut points are considered as the two primary evaluation factors for intermediate and final solution selection during a partitioning process. One set of the experiments is designed to observe the range of solutions and the possible evaluation methods that can be used for searching for an approximately optimal solution. Another set of the experiments considers solutions for various circuits by applying one of the evaluation methods and producing solutions for a set of circuits. The procedures for the two sets of experiments are introduced in the later sections.

The quality of a final solution is affected by several factors, namely the number of cut points, sub-circuits' and overall fault coverage, the number of sub-circuits, the size of each sub-circuit, and the total number of collapsed faults. These factors are also used for evaluating and selecting the final solution. The number of cut points is a very important factor in that it is relevant to the complexity of each sub-circuit and the configuration solution. Minimizing the number of cut points can reduce the complexity of each sub-circuit and the testing configuration. Fault coverage is another important aspect. Since the goal of the testing is to check all possible resources used by a user-defined circuit, high fault coverage is required to cover as many detectable faults as possible. These two factors are examined both during the partitioning process and in considering the final solution selection.

Other factors are also considered when looking at the quality of the final solution. A fairly small sub-circuit, which covers a small number of FPGA tiles in the testing configuration, is expected in final solutions. The number of sub-circuits in a final solution is related to the size of the original circuits and the required size range for each sub-circuit. In order to obtain a reasonable size for each sub-circuit, the bigger the original circuit is, the more sub-circuits are produced. The total number of collapsed faults of each solution is also used as a factor for solution evaluation.

As discussed in Chapter 2, Sequential circuits are considered in the experiments as the primary focus. The fault model selected for the experiments is single stuck-at faults, which covers the single line-open, line-close, logic stuck-at faults and other faults that can be modeled by this fault model in FPGAs. Fault simulation is applied to circuits and sub-circuits in order to get the value of the fault coverage. A set of sequential circuits, a test pattern file, a fault simulator and graph files are required during the experimental process. These resources are discussed in detail in the later sub-sections.

5.1.2. The Design of the Experiments

Figure 5.1 shows the experimental procedure for the recursive circuit partitioning algorithm. There are 6 main steps involved in the process:

1). Convert the circuit network into a graph;

2). Evaluate the graph according to the applicable restrictions defined for a final solution;

3). If the circuit graph or sub-circuit graphs do not reach the final state, go to the circuit partitioning process in 4); otherwise, skip steps 4) and 5);

4). Perform the circuit partitioning on required graph and produce sub-graphs;

5). For each sub-graph, go to step 2);

6). Output the final solutions in circuit network format.

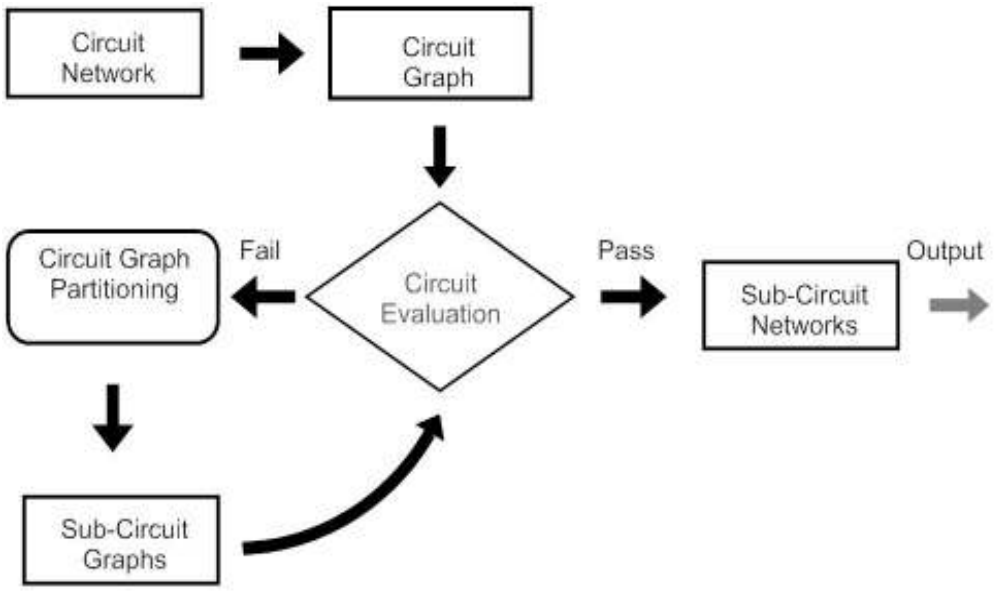


Figure 5.1 Circuit Partitioning Experiment Process

5.1.3. Resources Involved in the Experiments

A sequential circuit set, the ISCAS'89 benchmark set [42], is used in the experiments. The circuit network in the benchmark set is presented in a netlist format, BENCH file. An example of a circuit BENCH file is shown in Figure 5.2.

```
# 4 inputs
# 1 outputs
# 3 D-type flipflops
# 2 inverters
# 8 gates (1 ANDs + 1 NANDs + 2 ORs + 4 NORs)

INPUT(G0)
INPUT(G1)
INPUT(G2)
INPUT(G3)

OUTPUT(G17)

G5 = DFF(G10)
G6 = DFF(G11)
G7 = DFF(G13)

G14 = NOT(G0)
G17 = NOT(G11)

G8 = AND(G14, G6)

G15 = OR(G12, G8)
G16 = OR(G3, G8)

G9 = NAND(G16, G15)

G10 = NOR(G14, G11)
G11 = NOR(G5, G9)
G12 = NOR(G1, G7)
G13 = NOR(G2, G12)
```

Figure 5.2 An Example of Benchmark Netlist

As shown in Figure 5.2, the netlist gives some explicit information for the circuit. There is also some hidden information for each circuit that can be retrieved from calculation and testing. Table 5.1 provides a summary of the information for circuits in the benchmark set. In this set, a simple and small circuit S208.1 is selected as a typical sequence circuit in the illustration of the possible methods and solutions experiment. A

subset of the circuits in table 5.1 is used in the overall solution evaluation for the circuit partitioning algorithm.

Circuit	Level	FF's	PI's	PO's	Faults
s208	14	8	11	2	215
s208.1	11	8	10	1	217
s298	9	14	3	6	308
s344	20	15	9	11	342
s349	20	15	9	11	350
s382	9	21	3	6	399
s386	11	6	7	7	384
s400	9	21	3	6	424
s420	28	16	19	2	430
s444	11	21	3	6	474
s510	12	6	19	7	564
s526	9	21	3	6	555
s526n	9	21	3	6	553
s641	74	19	35	24	467
s713	74	19	35	23	581
s820	10	5	18	19	850
s832	10	5	18	19	870
s838	56	32	35	2	857
s953	16	29	16	23	1079
s1196	24	18	14	14	1242
s1238	22	18	14	14	1355
s1423	59	74	17	5	1515
s1488	17	6	8	19	1486
s1494	17	6	8	19	1506
s5378	25	179	35	49	4603
s9234	58	228	19	22	6927
s35932	29	1728	35	320	39094

Table 5.1 Circuit Information Summary of ISCAS89 Benchmark Set

In order to do fault simulation on the original circuits and sub-circuits produced by the partitioning algorithm, a fault simulator, HOPE, is employed in the experiments. HOPE is an efficient sequential circuit parallel fault simulator based on single fault propagation [28]. HOPE reads ISCAS89 circuit files as the fault simulation target, calculates collapsed faults of the input circuits, performs the fault simulation with selected test patterns, and outputs fault coverage and other relevant information. Throughout the experiments, the number of collapsed faults and the fault coverage are computed by HOPE and retrieved for result analysis.

There are two types of test pattern inputs for fault simulation using HOPE: using test pattern files or using randomly generated test patterns by HOPE. In the experiments, test pattern files are used in the recursive partitioning algorithm and randomly generated patterns are used to consider the possible fault coverage improvement in some evaluation cases.

The software is implemented in Java, an object oriented programming language, for the purpose of the algorithm evaluation. This takes an ISCAS89 circuit file and a test pattern file as inputs and generates circuit configuration solutions, which consist of an approximately optimal solution and other possible solutions. For each solution, all sub-circuits produced by the program are written to files in netlist format. A detailed information list is recorded for each selected solution. HOPE is employed by the program during the experiments with the input circuit file and the test pattern file.

5.2. Experimental Implementation

As mentioned in the previous section, two sets of experiments are performed for algorithm implementation and evaluation. In this section, the two sets of experiments are introduced in detail with their associated software work flow, experimental process explanations and algorithm implementations.

5.2.1. Work Flow

1). Possible Methods of Recursive Partitioning Algorithm

This experiment illustrates an approach that produces possible solutions by using two different main factors. Figure 5.3 shows the work flow of this experiment. Two files, a circuit file and a test pattern file, are taken as input files and the output is a set of solutions. One of the methods is selecting a sub-circuits set that is produced with the minimum number of cut points by each bipartitioning process, we call it the "min cut" method below; another one is choosing the sub-circuits set that obtained the maximum fault coverage, we call this the "max fault coverage" method.

A final state, or a termination state, of a sub-circuit is set by two conditions: the threshold of the number of inputs and the fault coverage. When a sub-circuit reaches the final state, it is added to a solution set. If a "good" solution with reasonable sub-circuits size and fault coverage can not be found, the circuit under the partitioning also reaches the final state.

Part of the circuit evaluation and solution selection in this experiment is performed manually. If we have two or more results that have same number of cut points in the min cut solution, the one with higher fault coverage is selected. Similarly, we choose the one with fewer cut points in the case where we have multiple results in the max fault coverage solution.

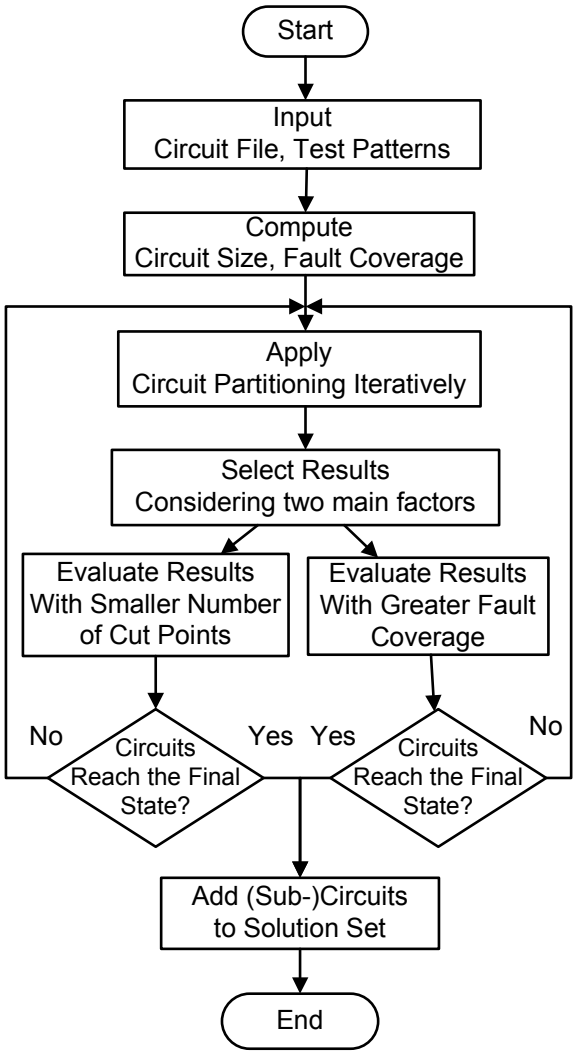


Figure 5.3 Work Flow of Possible Methods Experiment

2). Minimum Cut Solution of Recursive Partitioning Algorithm

This experiment applies the min cut method to produce a set of solutions for each circuit. An approximately optimal solution is chosen from the solution set. The work flow that is used to produce a partitioning solution for a circuit is illustrated in Figure 5.4.

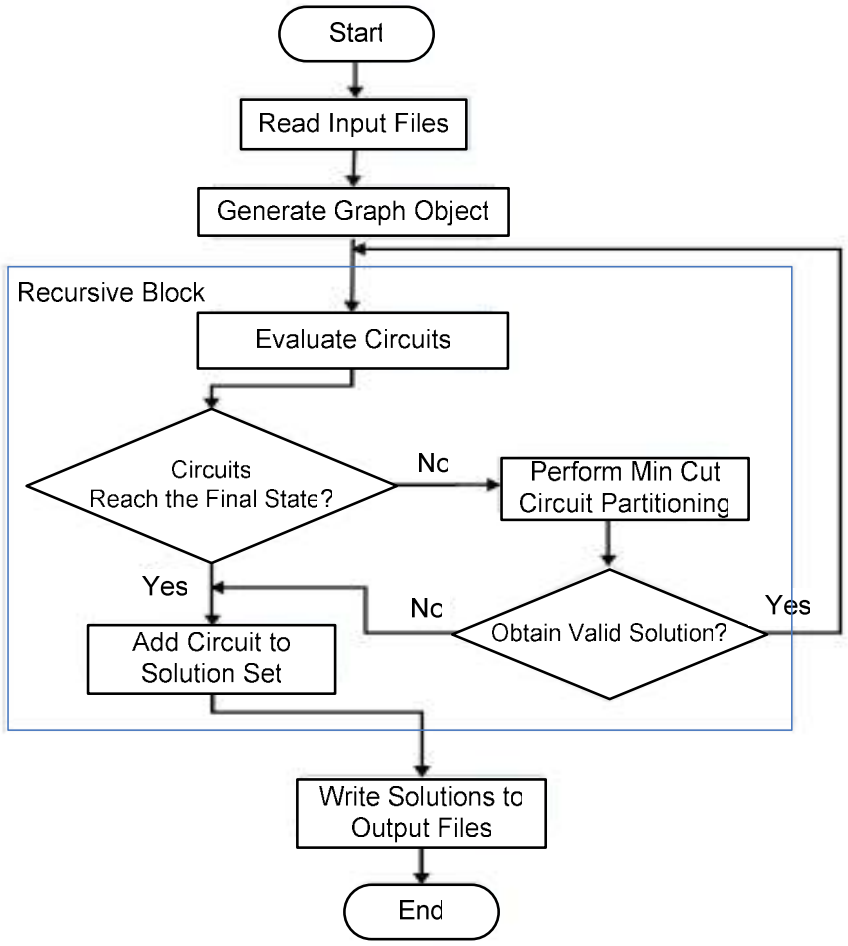


Figure 5.4 Work Flow of the Recursive Circuit Partitioning

In each experiment, a circuit file and a property file are taken as input files. The evaluation conditions, such as the number of inputs for a sub-circuit, size of sub-circuits and fault coverage threshold, are stated in the property file. During the partitioning process, the input circuit is represented by a graph object that contains

attributes of a circuit graph including data provided by the circuit file, values calculated from the file and information generated from the fault simulation.

The partitioning process is executed in a recursive block. In this block, the circuit is partitioned into a set of sub-circuits. The process is completed when all branches of the block are terminated with a base condition. A solution is produced and sent to an output file.

A random selection is applied to find cut points during the partitioning process. Therefore, solutions that are produced by executing the partitioning process multiple times for a circuit file are not identical. In order to find an approximately optimal solution, the experiment is performed a number of times to produce a solution set and a final solution is selected from the solution set.

5.2.2. Algorithm Implementations

Recursive partitioning is the key function of the experimental implementation. Recursive partitioning is applied in an attempt to partition a circuit into smaller units so that high fault coverage can be obtained by applying a particular test vector set. In order to reduce the circuit complexity, the result with the minimum number of cut points is selected during the partitioning.

```
CircuitGraphPartitioning Algorithm

    Setting: PropertyFile P, a file contains project properties that can be modified by users
    Input: a Graph File G, a .Bench circuit file
    Output: GraphList graphList, a set of sub-circuit graphs
        nodeList cutPointList, a set of nodes

    Global Attributes: nodeList cutPointList
        int InputConstrain ← P.InputNum;
        double FaultCoverageConstrain ←P.FaultCoverage;

    RecursivePartitionFunction(GraphFile g)
        Set: resultGraph ← Empty
            int intputNodeNum ← g.inputNum
            int faultCoverage ← g.faultCoverage
            List nodeList ← g.nodes.AscendingSort

        Base Case: inputNodeNum <=InputConstrain and
                faultCoverage >= FaultCoverageConstrain
            graphList.Add(g);
            Return;

        Case 1: inputNodeNum <= InputConstrain and faultCoverage < FaultCoverageConstrain
            CicuirtGraph[2] resultGraph ← TestPointCut(g, nodeList);
            If resultGraph isEmpty
                graphList.Add(g);
                Return;
            EndIf
            RecursivePartitionFunction(resultGraph[0]);
            Return;

        Case 2: inputNodeNum > InputConstrain
            CicuirtGraph[2] resultGraph ← BiPartitioning(g, nodeList);
            If resultGraph isEmpty //no "good" solution found
                graphList.Add(g);
                Return;
            Else
                RecursivePartitionFunction(resultGraph[0]);
                RecursivePartitionFunction(resultGraph[1]);
                Return;
            EndIfElse
        End Cases
    End RecursivePartitionFunction
```

Figure 5.5 Recursive Circuit Graph Partitioning Algorithm

The partitioning algorithm is a recursive procedure of performing the partitioning process on the input circuit until the final state is reached. The main functions of the procedure include circuit evaluation, circuit partitioning and solution collection. Figure 5.5 describes the algorithm in pseudo-code.

A property file is used in this implementation to set the restrictions for the experiment such as the expected input number, called "size constraint," desired fault coverage of resulting sub-circuits, called "fault coverage constraint," and size ratio of sub-circuits produced by bipartitioning, called "ratio constraint." For example, if we have 8, 80% and 1/3 as size, fault coverage and ratio constraints, then we are looking for sub-circuits whose number of inputs are fewer than 9 and the fault coverage is greater than 80% in the final solution, and each sub-circuit has at least 1/3 nodes of the circuit being partitioned during the bipartitioning. A circuit graph list is declared to collect sub-circuit graphs in a solution. And a node list is used to accommodate cut points of the solution.

Three cases are considered during the partitioning algorithm. For the base case, circuit graphs are added to the solution and the program returns; for circuit graphs within the range of desired size, but not the fault coverage, the CheckPointCut algorithm is performed to improve the fault coverage; for circuit graphs that do not meet the expected state of both size and fault coverage constraints, the BiPartitioning algorithm is applied to divide the circuit into two sub-circuit graphs. In the latter two cases, the partitioning process continues recursively on the resulting circuit graphs

until a final state is reached. The algorithms of these two cases are described in Figure 5.6 and Figure 5.7 respectively.

BiPartitioning Algorithm

BiPartitioning(GraphFile g, nodeList n, nodeList cutPoints,GraphFile[] gf)

Case 1: candidate node list n is empty //Bipartitiong process is completed
cutPointList.add(cutPoints.nodes);
return sub-graph files gf;

Case 2: candidate node list is not empty //Proceed Bipartitioning process

1) Find first cut point Node: v_f ← FindCutPoint (null, n);
2) Add v_f to cut points list : cutPoints.add(v);
3) Find cut points V (v) for all directed path form v_f --> v_f ;
4) Add all cut points v in V to cut point list: cutPoints.add(V);
5) Initialize two sub-graphs G and G' with node v_f and the copy of v_f, v_f';
6) Perform contraction on node v_f and v_f';
7) Add cut points derived from contraction to cut points list cutPoints;
8) Disconnect two sub-graphs G and G';
9) Return sub-graph file gf ← G, G'.

End BiPartitioning

Figure 5.6 BiPartitioning Algorithm

CheckPointCut(GraphFile g, nodeList N)

If N is empty
Return null;
Else

1) Find the cut point v ← FindCutPoint(null, N);
2) Divide v into v and v';
3) Evaluate the fault coverage;
4) Return the new graph if the fault coverage increase based on the previous value of this graph; otherwise, keep the previous graph;
5) Return the new graph or null if no new graph can be produced

End CheckPointCut

Figure 5.7 TestPointCut Algorithm

Both TestPointCut and BiPartitioning are iterated a number of times by selecting different cut point sets. The final results are selected from a set of results. In the TestPointCut algorithm, since the purpose of the process is to improve fault coverage, the result with highest fault coverage is selected. In the BiPartitiong algorithm, the main factor considered is the number of cut points. Therefore, the result requiring the minimum number of cut points is selected.

Searching for an appropriate cut point is a basic step to finding a good solution. This process is implemented by a function, FindCutPoint, which is applied in both the CheckPointCut and the BiPartitioning algorithms. In this implementation, cut points are determined by two main methods: the priority candidate selection and the random selection. Figure 5.8 gives the pseudo-code of FindingCutPoint.

In the FindCutPoint algorithm, cut points are chosen from a list of candidate nodes that are passed in as an argument. Depending on the stage of the partitioning process, Potential candidate nodes can be obtained from different sets of nodes. During the stage of searching for the first cut point of a circuit or a sub-circuit, the candidates consist of all nodes of the circuit. The priority of each node is decided by the number of fanouts. During the stage of finding a cut point on a directed path between two nodes, the candidates are nodes on the directed paths that have common nodes. The selection priority at this stage is decided by whether a node is a common node and the location of the node. If a path does not share nodes with others, the candidates

are the nodes on the single path. If we have multiple candidates with same priority, a node is selected at random from those candidates.

FindCutPoint Algrithm

Input: A cut points set $C = \{v_a, v_b, ..., v_m\}$, a list of nodes;

A set of candidate nodes $V = \{v_i, v_j, ..., v_l\}$;

Output: Node v, the cut points.

```
FindCutPoint(nodeList V, nodeList C)

  //No legitimate node is available
  Case 1: If V = φ then
            Return null;
          EndIf
  //Find the first cut point
  Case2:  If V ≠ φ and C = φ then
            1). Sort nodes in V in ascendant order;
            2). Get the first class of nodes V_temp with most fan-outs;
            3). If V_temp.Count >1 then Random select a node v from V'
            4). Return v;
          EndIf
  //Find an appropriate cut point on directed paths
  Case 3: If V ≠ φ and C ≠ φ then
            1). Sort nodes in V in position order on paths
            2). For each node v in V
                  If v.next.count > 1 and
                     v.next[i] ∈ V and v.next[j] ∈ V, where i ≠ j;
                     Return v;
                  EndIf
            3). If V.Count >1 then Random select a node v from V'
            4). Else Get v in V;
            5) Return v;
          EndIf
End FindCutPoint
```

Figure 5.8 FindCutPoint Algorithm

In the FindCutPoint function, the most time consuming case is the second since all given nodes are examined in the process of finding the proper node. The runtime for this case in FindCutPoint is $T(FindCutPoint) = T1+T2+T3$, where T1 is the time used to sort nodes, T2 and T3 are constant time used to select the cut point and return it respectively. Therefore, the complexity FindCutPoint is $O(n \log n)$ if we use the Heapsort or Quicksort algorithm.

The runtime for function CheckPointCut is $T(CheckPointCut) = m(T1+T2)$, where "T1" is the time used to find a cut point, "T2" is a constant time used to cut a node and evaluate the fault coverage, and "m" is the number of this function being iterated. The upper bound of this function is $O(n^2 \log n)$ when we assume the worst case of m is n.

The runtime for the BiPartitioning algorithm is $T(BP) = k(T1 + T2 + T3 + T4 + T5)$, where "T1" is the time for searching for paths, "T2" is the time for finding cut points, "T3" is the time of cutting and adding cut points, "T4" is the time for the contraction process, "T5" is a constant time for constant time operations such as set, initialization and return operations. Assume "k" is a constant number of the function being iterated and the worst case of the number of cut points is n, which is the total number of nodes. Depth-First Search (DFS), the runtime of which is $O(n + E)$ where E is the number of edges in the graph, is used as the method in finding paths. The contraction process travels through each node, which is n as well. The complexity of BiPartitioning is

$$O(T(BP)) = k(O(E+n) + O(n^2 \log n) + O(n) + O(n) + C) = \begin{cases} O(E+n), \text{ if } E >= n \log n \\ O(n^2 \log n), \text{ if } E < n \log n \end{cases}$$

.

Suppose we set a constant c as the ratio constraint in the BiPartitioning. The recursive partitioning algorithm requires $O(T(n)) = O(BP) + O(T(cn) + T((1-c)n))$ time because the upper bound of runtime for CheckPointCut is $O(T(BP))$. Suppose $E < nlong$, the best case of the time required is $O(n^2 \log n)$ where $c = 1/2$. The required time increases, but not significantly, when c decreases because although the upper bound of runtime for each recursive step is the same, the worst case depth of the recursive process increases when c is smaller. For example, worst case depth of the recursive process is $\log_2 n$ when $c = 1/2$. However, if $c = 1/3$, the depth of the recursive process increases to $\log_{3/2} n$, which is larger. Figure 5.9 illustrates the example. Similarly, this situation also applies to $E < n \log n$.

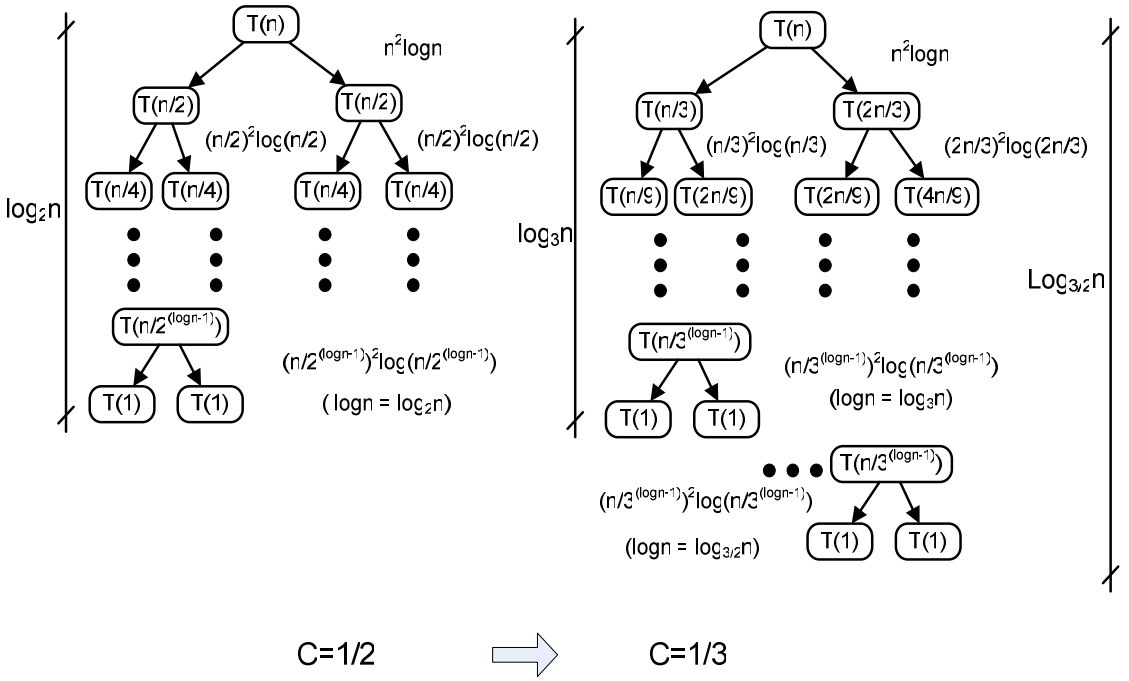


Figure 5.9 An Example on Depths of the Recursive Process with Different Ratio Constraint

5.2.3. Anticipated Result

The purpose of the recursive circuit partitioning algorithm is to divide a large circuit into small units such that those small sub-circuits can be applied during application dependent FPGA testing configuration. An optimal solution is defined as a partitioning result with the minimum number of cut points and maximum fault coverage. An approximately optimal solution is expected to be produced by the partitioning algorithm.

In this algorithm implementation, a fault coverage threshold is defined in order to set a fault coverage restriction for the resulting sub-circuits. Partitioning processes are performed multiple times during each solution search and a final solution is obtained

by comparing the number of cut points and overall fault coverage. According to the partitioning theory we introduced in Chapter 4 and the implementation of the algorithm, the fault coverage of the resulting solution is expected to be greater than the fault coverage threshold. An approximately optimal solution with a smaller number of cut points and higher fault coverage is expected to be obtained by executing the partitioning process a number of times.

The size of sub-circuits is also restricted in order to obtain small sub-circuits so that each sub-circuit can be configured in a small area on an FPGA chip. There are possibilities that a sub-circuit that does not meet the size constraint is collected by a final solution. This happens when a "good" cut cannot be found. However, because cut points are selected multiple times in an attempt to eliminate bad results, a large percentage of sub-circuits that satisfy the requirements is expected. Therefore, larger circuits should have more sub-circuits than do smaller ones.

5.3. Summary

The experiments are designed and implemented to provide results for the recursive circuit partitioning technique we introduced in Chapter 4. Two sets of the experiments are designed in order to observe the possible solutions and evaluate the recursive circuit partitioning algorithm respectively.

Main factors that are considered during a partitioning process are the number of cut points and the fault coverage. A set of ISCAS89 benchmark circuit files, the Hope

fault simulator, a test pattern file and the software implemented in Java are employed in the experiments. The recursive block in the software includes the base case, where a partitioning process terminates, the CheckPointCut, where a check point is partitioned in order to improve fault coverage, and the BiPartitioning, where a circuit is divided into two sub-circuits.

In the first set of experiments, we try different combinations of the min cut and the max fault coverage methods to provide a set of possible solutions. In the second set of experiments, we apply the recursive circuit partitioning process to a set of circuits in an attempt to provide information for algorithm evaluations. The results and analysis of the experiments are presented in Chapter 6.

6. Experimental Results and Analysis

Two sets of experiments, possible methods of partitioning and recursive circuit partitioning experiments, are performed and the results are presented and analyzed in this chapter. The experimental results and analysis provides more solid information about the feasibility of the testing strategy, whether the results meet the expectation, and the behaviour during the partitioning.

Results of the experiments that show the possible methods and solutions are presented in section 6.1 and the analysis of the results is discussed in the same section. The feasibility of applying a different method to select possible solutions is mainly concerned in this set of experiments. Results and analysis of the experiments that provide solutions to a set of circuits by applying recursive circuit partitioning are presented in section 6.2. A final solution is selected for each circuit file being experimented to produce a set of data on this set of experiments. Fault coverage, the number of cut points and the size of each sub-circuit are main interests during the analysis of the results. Section 6.3 summarizes this chapter.

6.1. Results of the Possible Methods of Partitioning

In this set of experiments, we apply the partitioning process step by step to a circuit using two approaches, the min cut and the max fault coverage methods. The result of the experiment is a set of possible solutions for a particular circuit.

Two sequential circuits are involved in the experiments and the result of one of the circuits, S208.1, is presented as the example in this section. The circuit file is attached in appendix1 and other information for this circuit can be found in Table 5.1. The test pattern file used in this set of experiments is generated by a 6-bit address generator with initial state 0, that is, 64 test patterns are applied in the fault simulation. For the circuit that has more than 6 primary inputs, test patterns are applied in parallel to satisfy the input vector's size.

The constraints of the partitioning are set as follows: size constraint is set to 8 inputs; the fault coverage constraint is set to 80% and the ratio constraint is set to 1/5. The terminate states of a sub-circuit are 1) input number is less than 8 and the fault coverage is greater than 80%, 2) it reaches a point that no "good" cut point can be found that satisfy the constraints.

The intermediate steps and final results are shown as a tree structure in Figure 6.1. In the result tree, the root is the original circuit and each node is either a sub-circuit, called a circuit node, or a partitioning result, called a solution node. The structure is explained in Figure 6.1(a). The complete solution tree is shown in Figure 6.1(b) (c).

All leaves represent sub-circuits in solution sets. Information for each node indicates the attributes of the result in the solution selection. A solution node is either a min cut solution node, which represents the solution by applying the min cut method, or a max fault coverage node, which represents the solution obtained by applying max

fault coverage method. The level of a node is the number of solution nodes on the path from the root of the tree to the node. The level of a solution tree is the number of solution nodes on the longest path of the tree.

A sub-circuit node can be presented as $n = (size\ of\ the\ sub-circuit,\ fault\ coverage)$, where size is the number of gates. A partitioning solution node can be presented as $n = < fault\ coverage,\ the\ number\ of\ cut\ points >$. Two methods are considered in this experiment, and for each result, one or two sub-circuits are obtained by the partitioning process. Each non-leaf generally has either one or two nodes. Two nodes are obtained if 1) two different solutions are found for min cut and max fault coverage method; 2) two sub-circuits are obtained from the bipartitioning process. There are two situations that one child is present. One case is that the min cut and max fault coverage solutions are the same solution; another case is that a check point cut is performed on the sub-circuit.

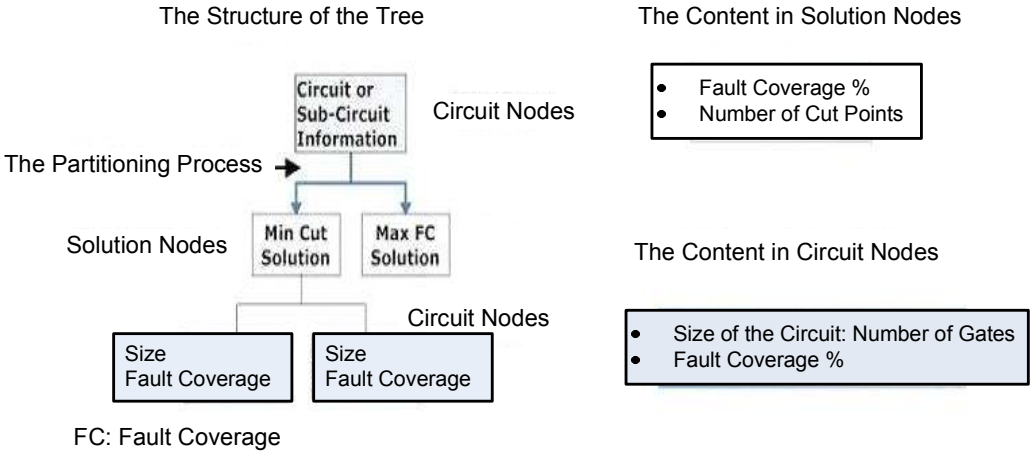


Figure 6.1 (a) The Tree Structure for the Possible Solutions

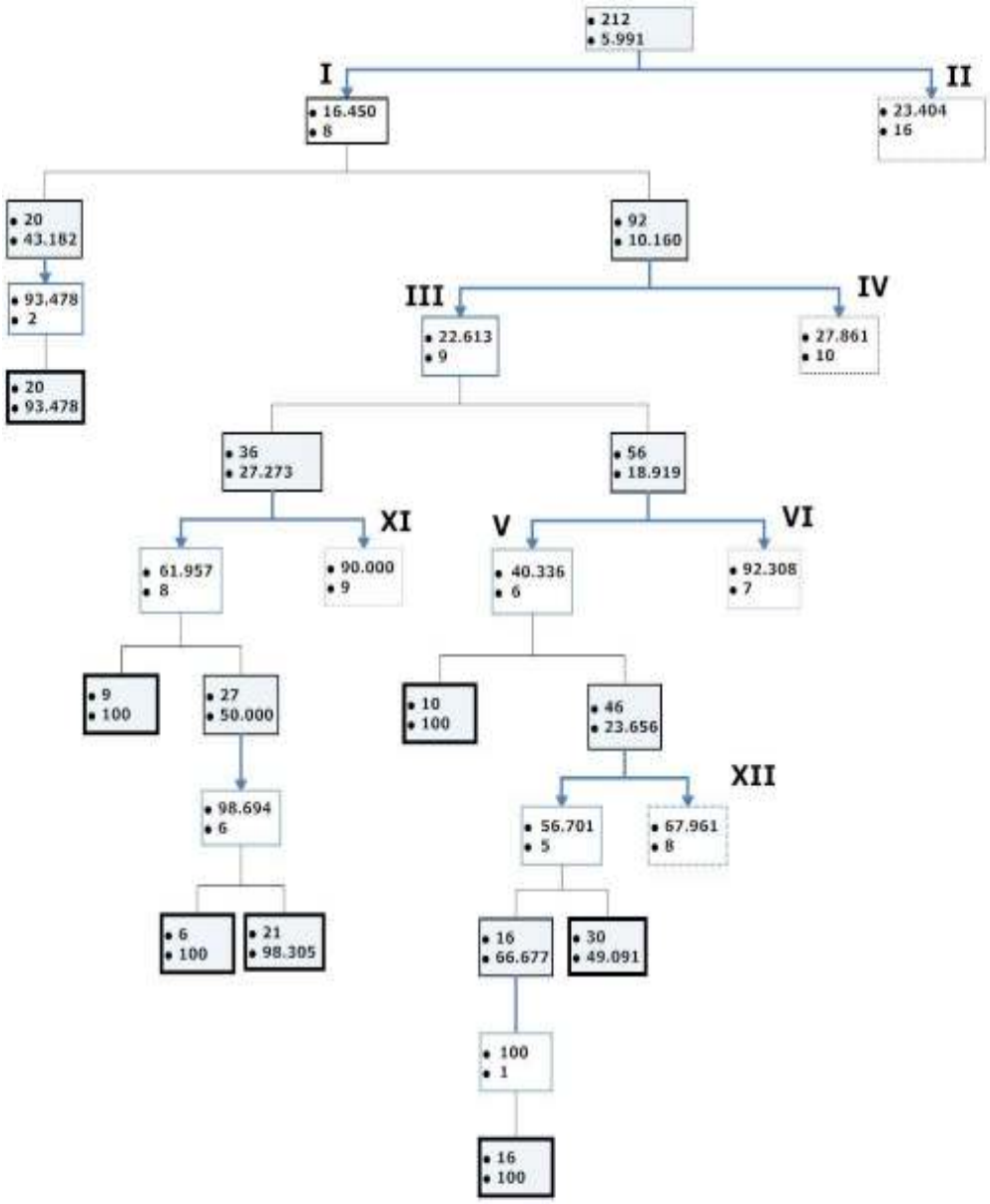


Figure 6.1(b) Solution Tree for Circuit S208.1 (Part 1)

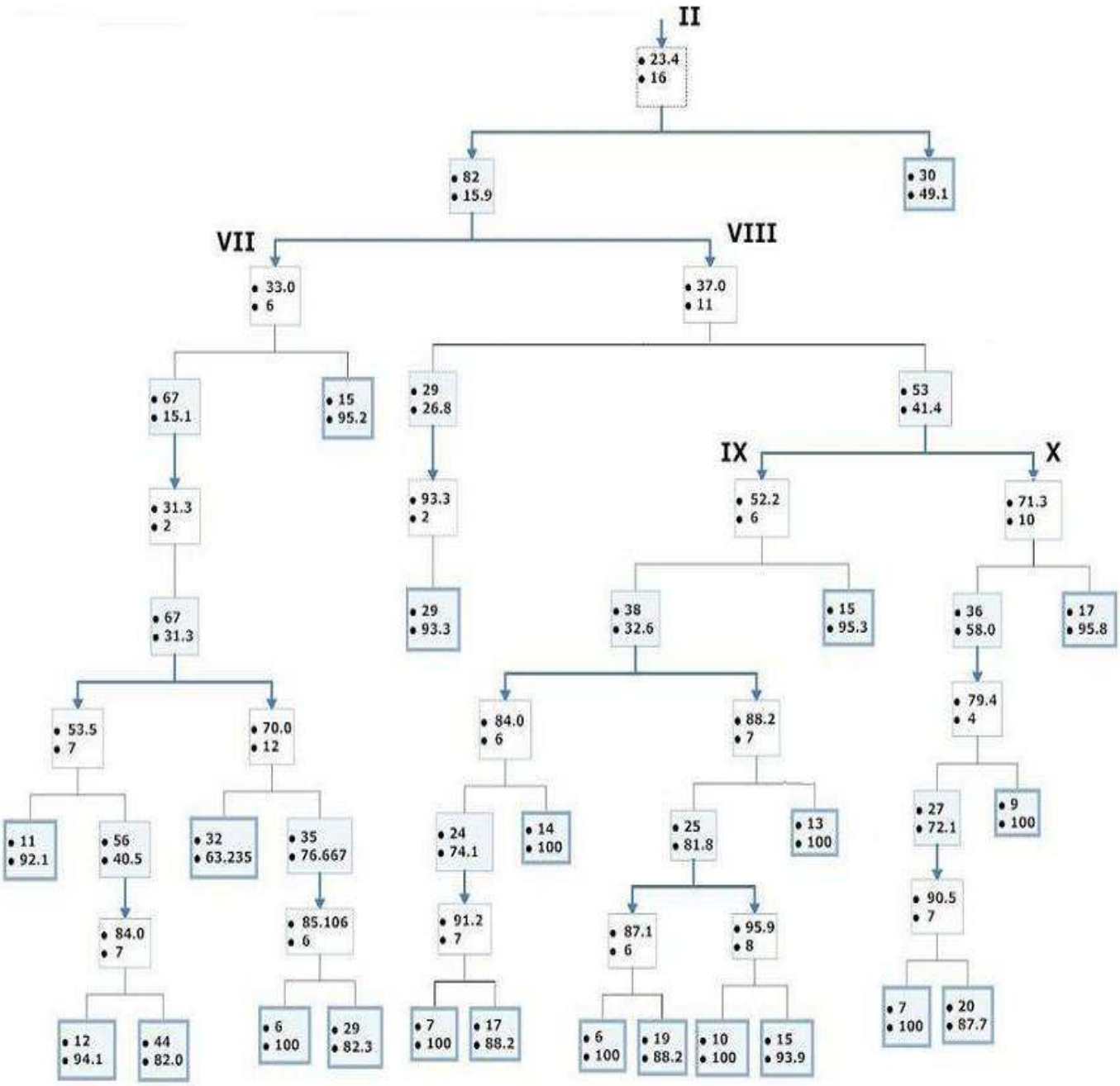


Figure 6.1 (c) Solution Tree for Circuit 208.1(Part 2)

Figure 6.1 Solution Tree for Circuit S208.1

A solution can be obtained by following paths of selected methods. For example, a solution with mixed methods can be selected by following steps described below:

- Select the min cut result for the first step, therefore follow the solution node I;
- Two children of I are selected as two sub-circuits from the first step;
- A leaf node is found by following the left child of node I, and the sub-circuit presented by the node (20, 93.5) is added to the solution set;
- The partitioning continues for the right child of I and min cut result is selected, the path follows solution node III;
- For both children of III, max fault coverage result is selected, so the path follows solution nodes XI and VI;
- The left child of XI is a leave node $n = (8,100)$, therefore n is added to the solution set; two children nodes of VI, $n = (11,100)$ and $n = (45,30.1)$ are leaves and added to the solution set as well;
- The path goes through the right child of XI with selected methods, assuming min cut method, and the sub-circuits presented by the leaves, $n = (5,100)$ and $n = (23,84.1)$, can be added to the solution.

The solution obtained by the above steps consists of 6 sub-circuits with 41 cut points.

A solution set consists of 12 solutions (Table 5.2) shows a subset of all solutions that are generated from the result tree for circuit S208.1. The first solution (solution 1) is obtained by applying only min cut method and the last solution (solution 12) is obtained by applying the max fault coverage method during the partitioning process. Other solutions are obtained by employing different methods at different steps. A

sub-circuit set is represented by the node notation we introduced earlier in this sub-section.

ID	Method(s)	Num.of Cut Points	Num. of Sub-Circuits	Sub-Circuit Set
1	Min Cut	44	7	{(20,93.5), (9,100), (6,100), (21, 98.3), (10,100), (16,100), (30,49.1)}
2	Min Cut & Max FC	53	8	{(20,93.5), (9,100), (6,100), (21, 98.3), (10,100), (12,100), (11,100), (23,71.4)}
3	Min Cut & Max FC	55	8	{(20,93.5), (9,100), (6,100), (21, 98.3), (10,100), (12,100), (17,89.7), (17,88.4)}
4	Min Cut & Max FC	57	8	{(20,93.5), (8,100), (6,100), (22,90.5), (10,100), (12,100), (17,89.7), (17,88.4)}
5	Min Cut & Max FC	18	3	{(20,93.5), (35,31.4), (57,25.2)}
6	Min Cut & Max FC	40	6	{(20,93.5), (9,100), (6,100), (21, 98.3), (11,100), (45,30.1)}
7	Min Cut & Max FC	38	5	{(30, 49.1), (11,92.1), (12,94.1), (44,52.0), (15,95.2)}
8	Min Cut & Max FC	42	5	{(30, 49.1), (32,63.2), (6,100), (29,82.3), (15,95.2)}
9	Min Cut & Max FC	48	6	{(30, 49.1), (29,93.3), (15,95.3), (14,100), (7,100), (17,88.2)}
10	Min Cut & Max FC	48	6	{(30, 49.1), (29,93.3), (15,95.3), (13,100), (6,100), (19,88.2)}
11	Min Cut & Max FC	50	6	{(30, 49.1), (29,93.3), (15,95.3), (13,100), (10,100), (15,93.9)}
12	Max FC	50	6	{(30, 49.1), (29,93.3), (17,95.8), (9,100), (7,100), (20,87.7)}

Test Pattern: 64 Test patterns (0 - 63)
Sub-Circuit Set: {(Size of the Circuit, Fault Coverage)}

Table 6.1 Results of Possible Methods for Circuit S208.1

6.2. Experimental Analysis of Possible Methods of Partitioning

As the tree structure shows, a large number of solutions can be generated by following different paths of a solution tree. For example, a perfect binary solution tree with level n can produce solution set $Ns = 2 \times 2^2 \times 2^3 \times \cdots \times 2^n = 2^{1+2+3+\cdots+n} = 2^{n(n+1)/2}$. For a large n, the number of solutions in a solution set is very large. It is very difficult to obtain an optimal

solution using two major factors and considering solutions produced by applying mixed methods. Generate solutions considering only one major factor and another one as reference is more feasible than generating a very large solution set and selecting a solution from it.

In the results for circuit s208.1, sub-circuits that are generated by applying the max fault coverage method tend to have a larger number of cut points. Also, a final solution obtained from the max fault coverage method is more likely to have sub-circuits that do not have the expected properties. Therefore the result has fewer sub-circuits and lower fault coverage for each sub-circuit. In the other hand, selecting solutions that produce a minimum number of cut points may obtain sub-circuits with lower fault coverage first. But it is more likely that the partitioning is able to be performed continually and sub-circuit graphs tend to have smaller size and higher fault coverage.

As shown in Table 6.1, the first result is produced by using min cut method. In this result, 7 sub-circuits are produced with 44 cut points. Comparing to the result 12, which is produced by using max fault coverage, the first result has more sub-circuits, fewer cut points and equal or higher fault coverage for each sub-circuit. Therefore, for circuit S208.1, using the min cut method produces a better solution than using the max fault coverage method.

The recursive circuit partitioning algorithm applies min cut method as the main approach for the bipartitioning process. This example shows that for some circuits, the approach is effective and can produce a reasonable partitioning result.

6.3. Results and Analysis of the Recursive Circuit Partitioning Experiments

The recursive circuit partitioning process is applied to a set of ISCAS89 circuits in this experiment and a set of solutions is produced for each circuit. A final solution for each circuit is selected from a set of candidate solutions by comparing the number of cut points and fault coverage. As we introduced in Chapter 5, a candidate solution set is obtained by applying the partitioning process a number of times.

Ten circuits in total are involved in this set of experiments. The same test pattern file, referred as “Pattern set A” below, which we used in the experiments of possible methods, is applied in this set of experiments as well. A random test pattern set, called “Pattern set B” below, which is generated by HOPE and consists of 10000 random test vectors, is used for comparison in some experiments. Since recursive partitioning applies min cut method in the bipartitioning process, other restrictions and conditions for min cut method introduced before are also applied in this set of experiments. The number of times that the partitioning process is applied for each circuit is the level of the circuit. The size constraint of the experiments is set to 6; the fault coverage threshold is 80% and the ratio constraint is 1/4.

In section 6.3.1, two examples are given to show the behavior of solution sets for different circuits. In section 6.3.2, ten final solutions are obtained for ten circuits for the algorithm evaluation of the recursive circuit partitioning.

6.3.1. Results and Analysis of the Recursive Circuit Partitioning Examples

In this section, the candidate solutions of circuit S208 and S382 are presented and analyzed in order to study the behavior of solution sets for different circuits.

Table 6.2 is an experimental result that consists of a partitioning solution set for circuit S208. A set of solutions that are produced for circuit S382 is shown in Table 6.3.

Information of Circuit S208

Circuit File	Graph size	Circuit Size	Circuit Level	Num.of Input	Num.of Output	Collapsed Faults	Detected Faults(A)	Fault coverage(A)(%)	Detected Faults (B)	Fault coverage(B)(%)
s208.bench	115	104	14	11	2	215	107	49.767	94	43.721

Partitioning Solution of S208

Solutions	Num. of Sub-Circuits	Num. of Cut Points	Total Collapsed Faults	Total Detected Faults(A)	Fault Coverage(A)(%)	Total Detected Faults(B)	Fault Coverage(B) (%)
1	7	39	269	230	85.502	264	98.141
2	8	43	269	236	87.732	267	99.257
3	8	44	271	245	90.406	269	99.262
4	9	48	279	267	95.699	279	100
5	9	49	269	251	93.309	269	100
6	9	50	267	249	93.258	267	100
7	9	50	281	267	95.018	281	100
8	9	50	271	264	97.417	271	100
9	9	51	265	241	90.943	265	100
10	9	51	263	243	92.395	263	100
11	9	51	267	250	93.633	267	100
12	9	51	257	242	94.163	257	100
13	9	52	267	247	92.509	267	100
14	9	52	265	255	96.226	265	100

Note: Results with (A) and (B) are obtained by applying test pattern set A and B respectively.

Pattern set A: 64 Exhausitive Test Patterns (0 - 63)
Pattern set B: 10000 Random Test Patterns

Table 6.2 Circuit Partitioning Result for Circuit S208

Information of Circuit S382

Circuit File	Graph size	Circuit Size	Circuit Level	Num.of Input	Num.of Output	Collapsed Faults	Detected Faults(A)	Fault coverage(A)(%)	Detected Faults(B)	Fault coverage(B)(%)
s382.bench	182	179	9	3	6	399	49	12.281	53	13.283

Partitioning Solution of S382

Solutions	Num.of Sub-Circuits	Num. of Cut Points	Total Collapsed Faults	Total Detected Faults(A)	Fault Coverage(A)(%)	Total Detected Faults(B)	Fault Coverage(B)(%)
1	7	53	459	389	84.749	451	98.257
2	9	54	461	386	83.731	442	95.879
3	10	63	470	394	83.83	453	96.383
4	10	64	469	406	86.567	448	95.522
5	11	65	469	387	82.516	459	97.868
6	12	72	475	418	88	470	98.947
7	12	77	484	427	88.223	468	96.694
8	12	83	487	424	87.064	486	99.795
9	11	86	487	437	89.733	487	100

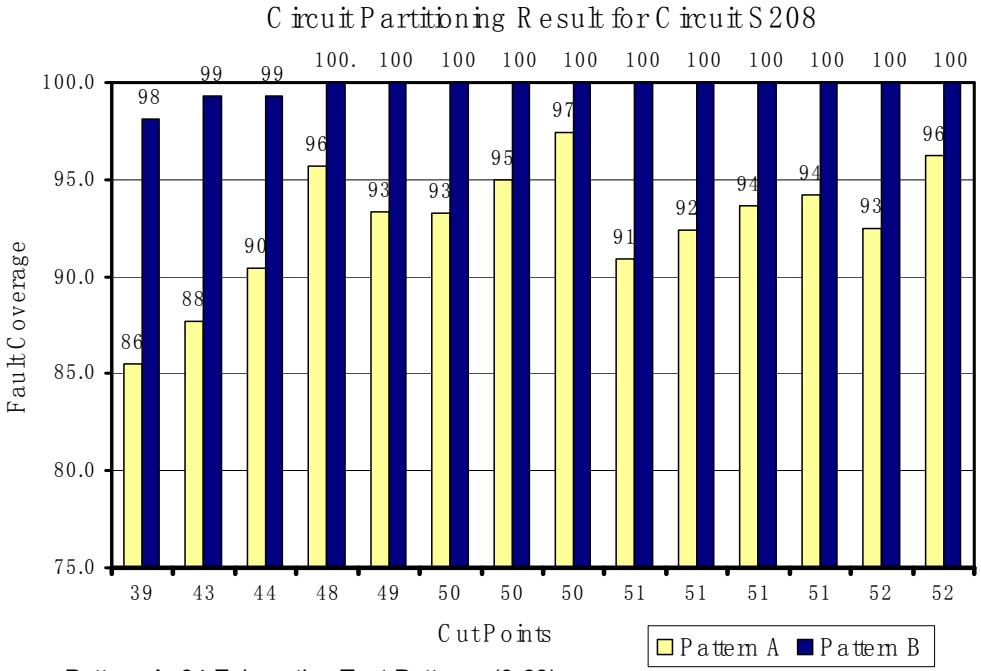
Note: Results with (A) and (B) are obtained by applying test pattern set A and B respectively.

Pattern set A: 64 Exhausitive Test Patterns (0 - 63)
Pattern set B: 10000 Random Test Patterns

Table 6.3 Circuit Partitioning Result for Circuit S382

As the information of circuits shown in Table 6.2 and 6.3, S208 has 11 inputs and 2 outputs with circuit level 14; and S382 has only 3 inputs and 6 outputs with circuit level 9. The fault coverage of the original circuit S208 with 64 test patterns is 49.8%. With an increased number of test patterns, 10000 random test vectors here, there is no improvement in fault coverage. For circuit S382, of which the fault coverage with 64 test patterns is 12.3%, the fault coverage increased to 13.3% with 10000 random patterns.

The partitioning solution sets show a discrepancy for these two circuits. Figure 5.11 and Figure 6.2 show the fault coverage difference of solutions with respect to the different number of cut points for the two examples. For circuit S208, fault coverage improves with the increase in the number of cut points at the starting range. After a certain point, 48 in this case, fault coverage becomes relatively stable and the solution produced a greater number of cut points with no better fault coverage.



Pattern A: 64 Exhaustive Test Patterns (0-63)
Pattern B: 1000 Random Test Patterns

Figure 6.2 Solution Comparison for Circuit S208

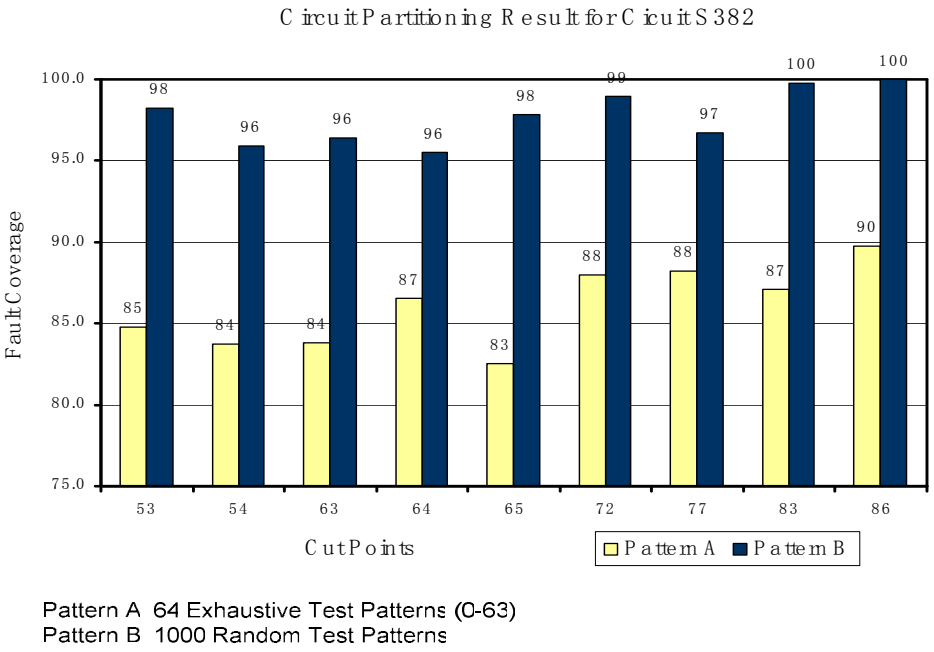


Figure 6.3 Solution Comparison for Circuit S382

As we can see in Figure 6.3, the number of cut points does not strongly affect the fault coverage in the solution set for S382. Solutions with a greater number of cut points do not necessarily have better fault coverage. The fault coverage with 10000 random test patterns has the same trend as the fault coverage with 64 test patterns for both circuits. We can say that the partitioning process has different effects on circuits with different properties during the solution searching.

6.3.2. Overall Results and Analysis of the Circuit Partitioning Experiments

As we introduced before, 10 circuits are involved in this set of experiments. For each circuit in this set, a solution with a minimum number of cut points is selected from a set of solutions, two examples of which are presented in the previous section. This

solution should also satisfy the fault coverage restriction. An overall result of the experiments is presented in table 6.4, where the fault coverage restriction is 80%.

Experimental Results of Recursive Circuit Partitioning

Original Circuit						Solution			
Bench File	Size	Circuit Level	Num.of Flip-Flops	collapsed Faults	FC	Num.of Sub-Circuit(s)	Num.of Cut Points	Total Collapsed Faults	Overall FC
s386	165	11	6	384	21.6	6	74	512	79.5
s208	104	14	8	215	49.8	7	39	269	85.5
s208.1	112	11	8	217	6	7	47	269	82.9
s298	133	9	14	308	12.7	7	61	388	87.6
s344	175	20	15	342	19.6	3	18	368	88.6
s349	176	20	15	350	19.1	4	25	370	85.7
s420	212	28	16	430	5.3	14	85	530	85.3
s382	179	9	21	399	12.3	7	53	459	84.7
s400	183	9	21	242	12	10	59	506	86
s526n	215	9	21	553	7	17	145	739	80.4

FC: Fault Coverage with Pattern set A

Table 6.4 Circuit Partitioning Solutions for a Set of Circuits

1. Analysis of Collapsed Faults and Fault Coverage

In Table 6.4, the original properties of each circuit are listed for comparison. Circuit size is defined by the number of gates of the circuits. The properties of a solution consist of the number of sub-circuits, the number of cut points, the number of total collapsed faults of all sub-circuits and the overall fault coverage $FC = total\ \det ected\ \ faults \div total\ \ collapsed\ \ faults$. The result fault coverage is produced by fault simulation on HOPE with the Pattern set A. The final solution for each circuit is the result that produces the minimum number of cut points and the fault coverage not smaller than the fault coverage threshold. For a solution set that does not contain a solution that satisfies the fault coverage constraint, the solution with the best fault coverage is selected.

As shown in Table 6.4, the number of collapsed faults, which is calculated during fault simulation by HOPE, of each set of sub-circuits produced by the partitioning algorithm is greater than the original circuit due to the improvement of the controllability and observability. Figure 6.4 shows the increase of the number of collapsed faults for each circuit. There is no obvious relation between the size of the circuit and the percentage of the collapsed fault improvement. According to the algorithm introduced in Chapter 4, the number and location of cut points are associated with the structure of a circuit graph. Consequently, the increase of the number of collapsed faults is also relevant to the structure of the circuit graph.

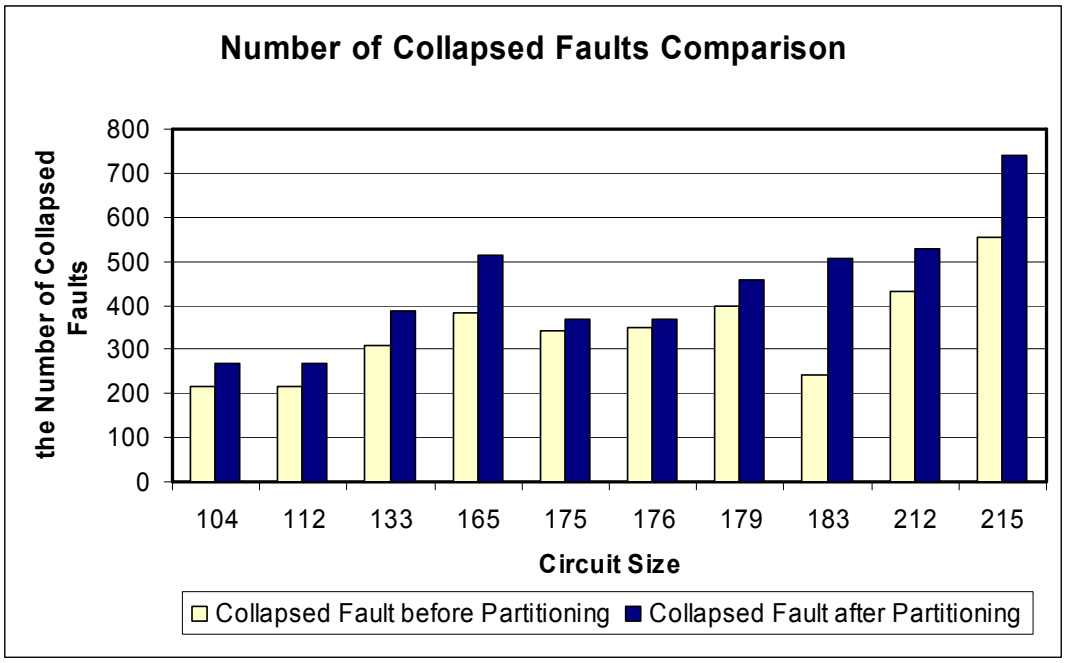


Figure 6.4 The Number of Collapsed Fault Comparison

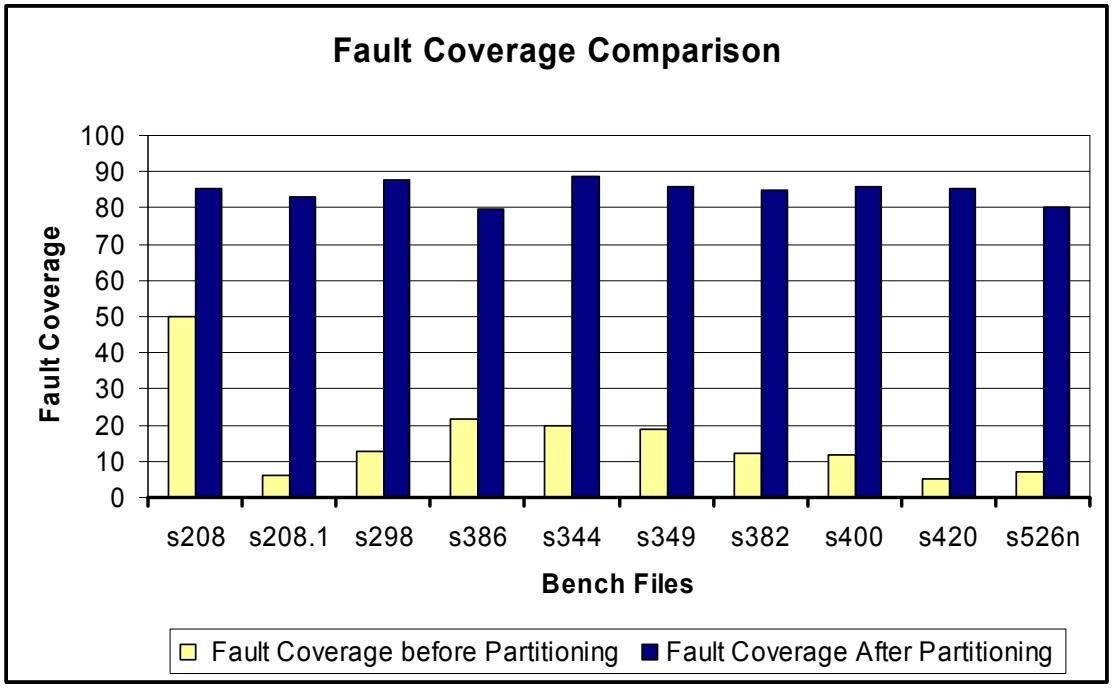


Figure 6.5 Fault Coverage Comparison

The fault coverage of most solutions is within the range, which is significantly greater than the fault coverage of the original circuit configuration. Figure 6.5 shows the fault coverage increase with respect to size of the circuits. The percentage of fault coverage improvement also has no obvious relation to the size of the circuits because of the same reason as the analysis on the increase of the number of collapsed faults.

During the fault coverage analysis, we notice that the numbers of flip-flops affects the fault coverage of a circuit. As shown in Figure 6.6, circuits with more flip-flops have lower fault coverage in most cases. One possible cause of the phenomenon is that the difficulties of controlling and observing states of the circuit rise when the number of flip-flops increases. This situation changes after the partitioning process.

Most of the fault coverage values are within the defined range and the number of flip-flops is no longer a factor that affects the fault coverage of a circuit.

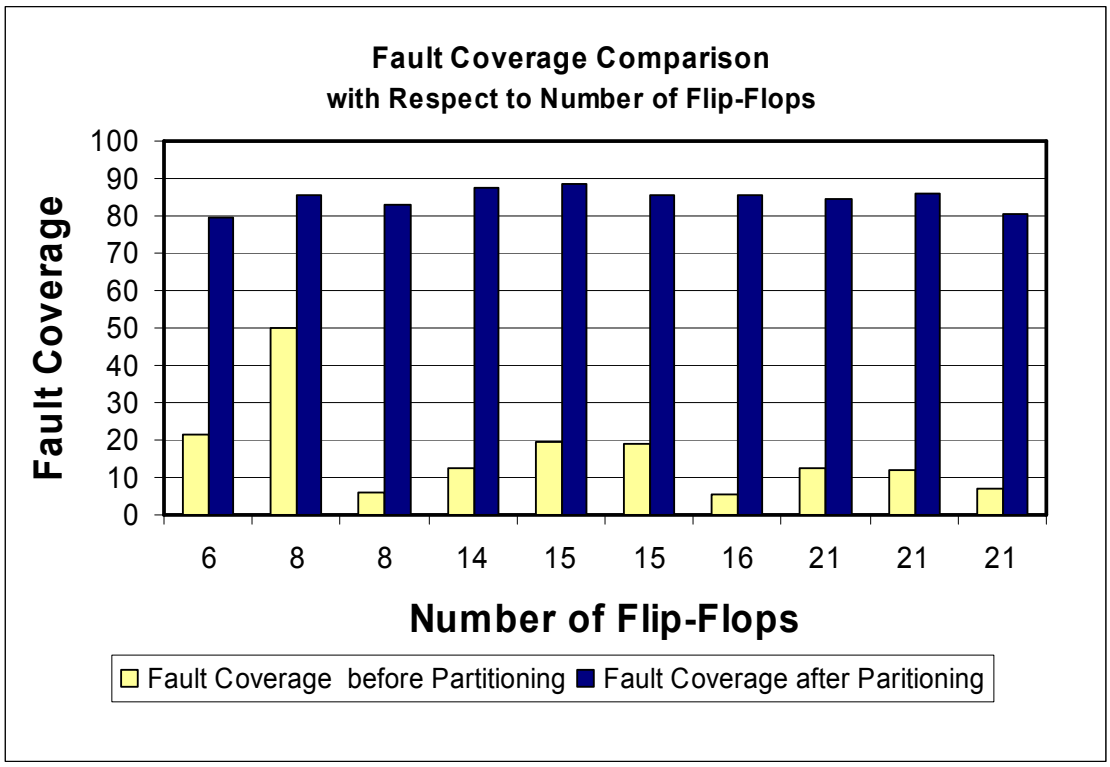


Figure 6.6 Fault Coverage Comparison with Respect to Number of Flip-Flops

Since the objective of partitioning process is to obtain smaller sub-circuits for a large circuit, the size of sub-circuits in a solution is expected to be in a smaller range and approximately balanced.

2. Analysis of the size and number of sub-circuits

If a circuit is going to be implemented on an FPGA chip, the size of the area required for a circuit configuration is related to several factors including the number of gates, the number of flip-flops and the number of inputs and outputs. In this set of experiments, we assume that the factor of number of flip-flops is not critical

considering that the ratio between the number of gates and the number of flip-flops are fairly low. Suppose we use 4-input and 1-output logic block based FPGAs for circuit configuration and we use the number of logic blocks as the size unit of the sub-circuit in this section.

We take two sample solutions, sub-circuit sets for S208 and S420, to show the size of the sub-circuits after the partitioning. S208 is a smaller circuit with 104 gates and S420 is a larger circuit with 212 gates. Table 6.5 and 6.6 show the solution sets and the properties of each sub-circuit of S208 and S420 respectively.

As shown in Table 6.5, there are 7 sub-circuits in the final solution of S208. The sub-circuit with the maximum number of gates and maximum number of inputs is sub-circuit 6, which has 25 gates, 16 inputs and 2 outputs. Calculated by the number of inputs only, we need 4 logic blocks to configure this circuit; calculated by the number of outputs only, we need 2 logic blocks. Obviously, we need 4 logic blocks for the configuration of the sub-circuit. The sub-circuit with the maximum number of outputs is sub-circuit 3, which requires 6 logic blocks of an FPGA since the number of outputs is 6. The sub-circuit that requires the minimum number of logic blocks is sub-circuit 2, which requires 3 logic blocks of an FPGA. Figure 6.7 shows the size of each sub-circuit of S208.

Sub-Circuit Set of S208

Sub-Circuit	Ciruit Size	Num. of Input	Num. of Output	Detected Faults(A)	Collapsed Faults	Fault coverage(A)(%)	Detected Faults(B)	Fault coverage(B)(%)
1	8	4	4	22	22	100	22	100
2	6	4	3	20	20	100	20	100
3	12	7	6	37	39	94.9	39	100
4	18	11	5	41	46	89.1	46	100
5	13	6	4	30	30	100	30	100
6	25	16	2	22	47	46.8	47	100
7	22	5	4	58	65	89.2	60	92.3

Total Sub-Circuit number	Total cut points	Total gate number	Total Detected Faults(A)	Total Collapsed Faults	Total Fault Coverage(A)(%)	Total Detected Faults(B)	Total Fault Coverage(B)(%)
7	39	104	230	269	85.5	264	98.1

Note: Results with (A) and (B) are obtained by applying test pattern set A and B respectively.

Pattern set A: 64 Exhausitive Test Patterns (0 - 63)
Pattern set B: 10000 Random Test Patterns

Table 6.5 Final Solution Set for S208

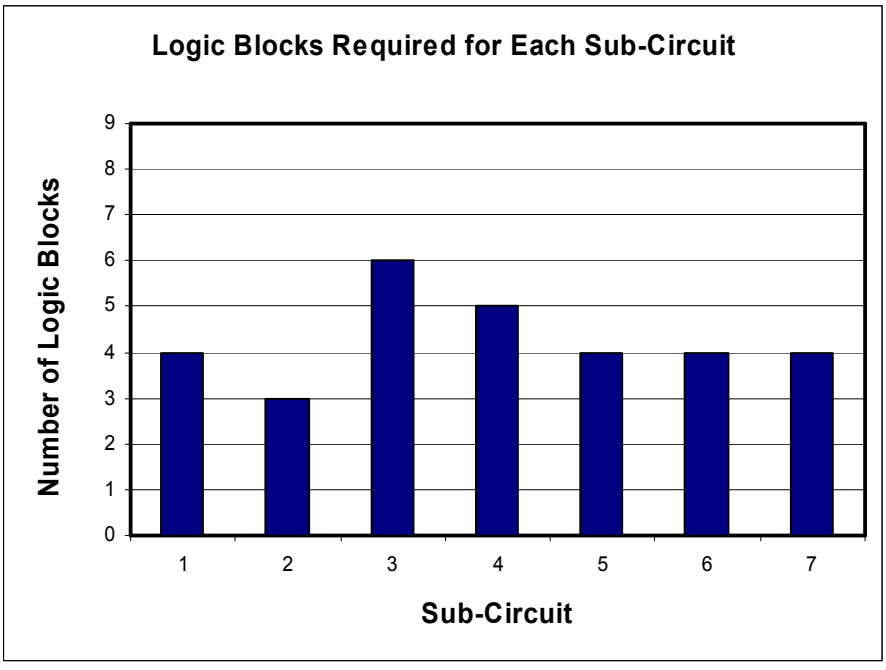


Figure 6.7 FPGA Area Required for Each Sub-circuit of S208

Sub-Circuit Set of S420

Sub-Circuit	Ciruit Size	Num. of Input	Num. of Output	Detected Faults(A)	Collapsed Faults	Fault coverage (A)(%)	Detected Faults(B)	Fault coverage (B)(%)
1	12	4	4	24	24	100	24	100
2	8	5	3	22	22	100	22	100
3	13	8	4	41	43	95.3	43	100
4	14	6	5	32	33	97	33	100
5	5	6	2	15	15	100	15	100
6	7	5	2	16	16	100	16	100
7	31	12	6	72	79	91.1	79	100
8	8	4	3	16	16	100	16	100
9	12	6	3	22	22	100	22	100
10	9	4	4	26	26	100	26	100
11	15	7	5	47	47	100	47	100
12	14	10	5	38	40	95	40	100
13	42	29	3	23	82	28	79	96.3
14	22	5	4	58	65	89.2	60	92.3

Total Sub-Circuit number	Total cut points	Total gate number	Total Detected Faults(A)	Total Collapsed Faults	Total Fault Coverage (A)(%)	Total Detected Faults(B)	Total Fault Coverage (B)(%)
14	85	212	452	530	85.3	522	98.5

Note: Results with (A) and (B) are obtained by applying test pattern set A and B respectively.

Pattern set A: 64 Exhausitive Test Patterns (0 - 63)
Pattern set B: 10000 Random Test Patterns

Table 6.6 Final Solution Set for S420

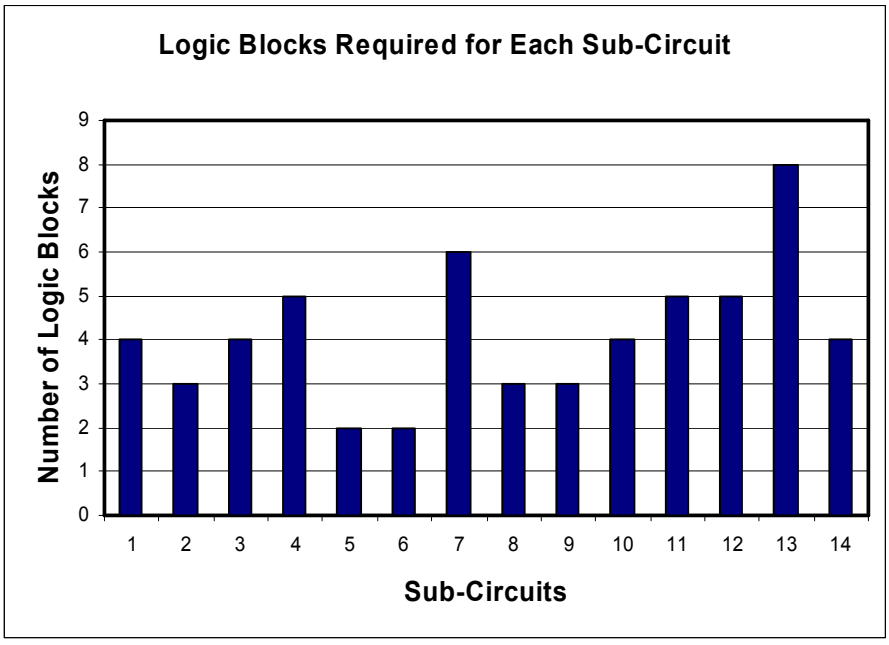


Figure 6.8 FPGA Area Required for Each Sub-Circuit of S420

We do the same computing on the sub-circuits of S420 and Figure 6.8, which is derived from Table 6.6, shows the result. The maximum number of logic blocks required for this set of sub-circuits is 8.

For an FPGA with thousands of logic blocks, 6 or 8 is a fairly small size for a configuration block, therefore the partitioning technique is a reasonable solution in terms of configuration area for each sub-circuit.

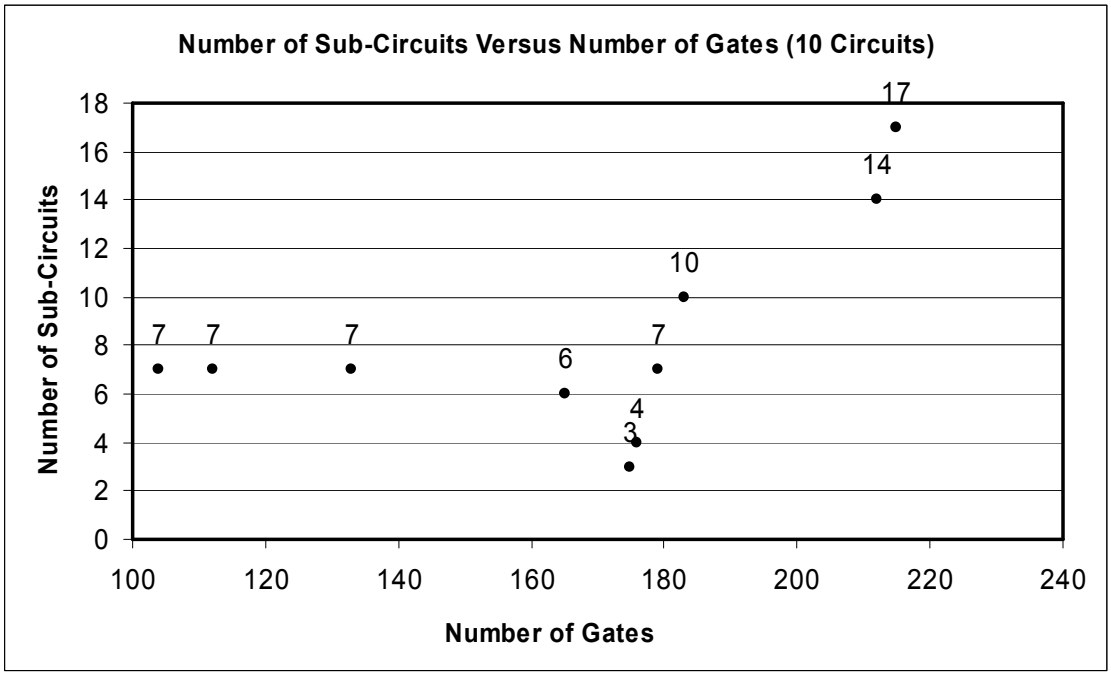


Figure 6.9 Number of Sub-Circuits Comparison

It is also expected that the number of sub-circuits of the solution for a larger circuit is greater than the one for a smaller circuit since we are looking for sub-circuits with size within a defined range. Figure 6.9 shows the trend of the number of sub-circuits with respect to the circuit size. As we can see, with the increase of the circuit size,

the number of sub-circuits increases in most cases with a few exceptions within a small range of circuit size. As illustrated in Figure 6.9, the solution for the circuit with size 175 consists of fewer sub-circuits than the ones for the circuit with size 165 and 133. This may happen when the size of the sub-circuits in the solution for a circuit are larger than the size of the solution for another circuit or some sub-circuits in a solution with smaller number of sub-circuits are larger than the expected range.

3. Analysis of the Number of Cut Points

The number of cut points in a solution mainly depends on the structure of a circuit graph instead of the circuit size as we discussed previously. Figure 6.10 shows that in this experiment set, the trend of the number of cut points is similar to the trend of the number of sub-circuits. This may be caused by the similarity in the circuit graph structure of the circuit set.

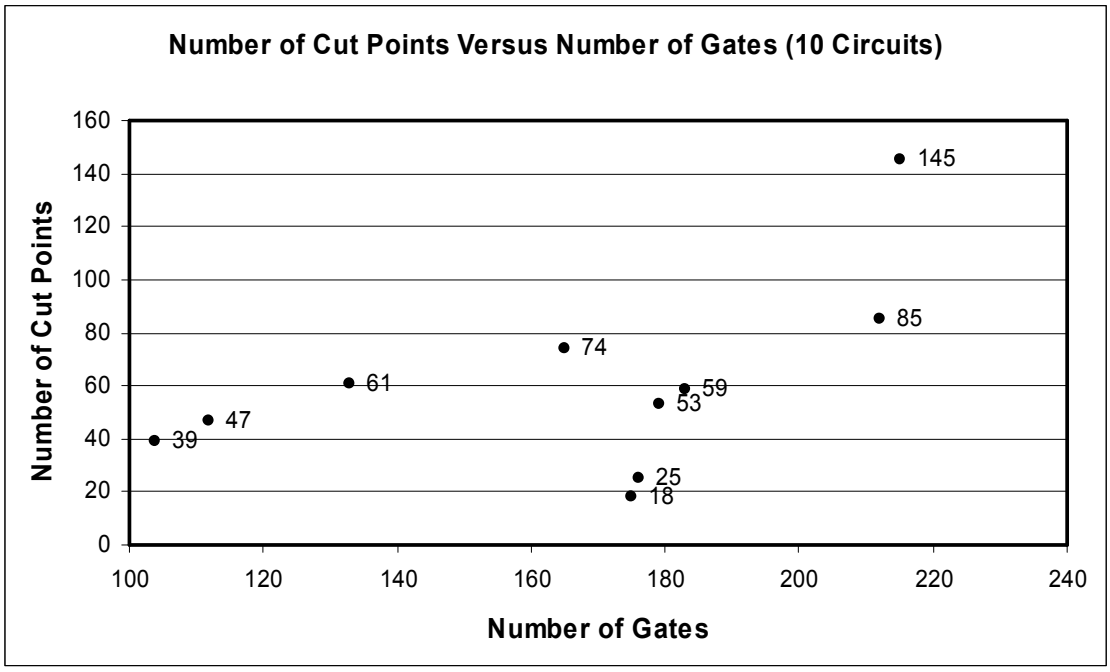


Figure 6.10 Number of Cut Points Versus Number of Gates

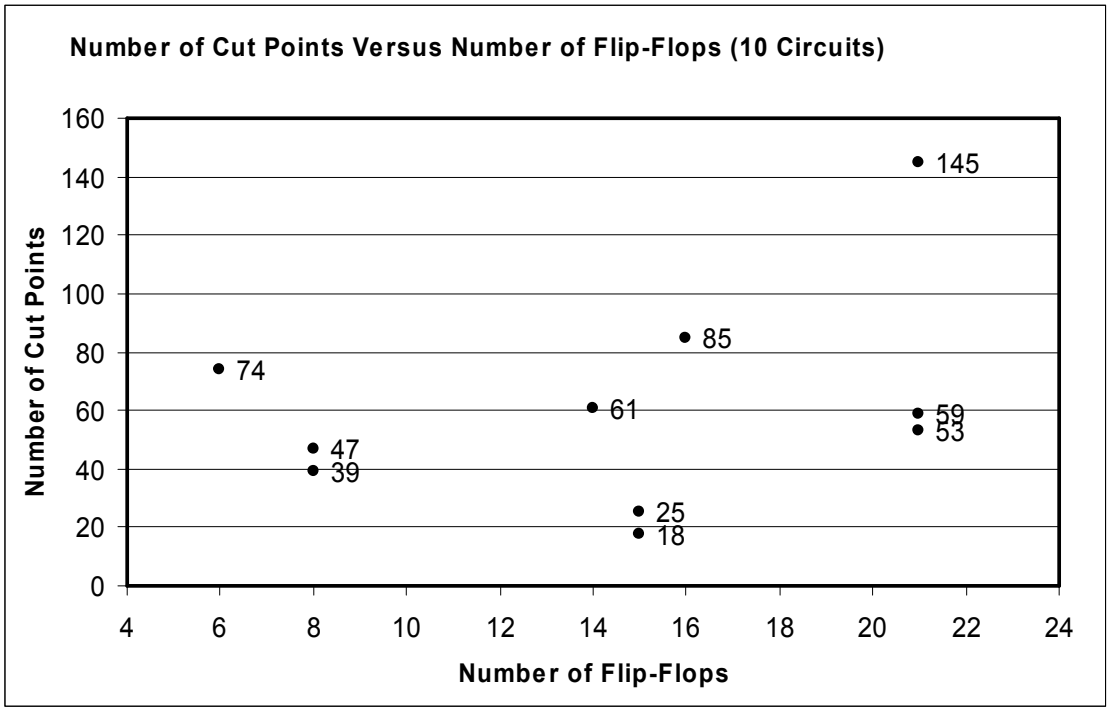


Figure 6.11 Number of Cut Points Versus Number of Flip-Flops

Figure 5.20 is created to show if there is any relationship between the number of flip-flops and the number of cut points. As we can see in the chart, the number of flip-flops is not obviously relevant to the number of cut points.

6.4. Summary

The purpose of providing the experimental results and analysis is to evaluate the recursive circuit partitioning algorithm by analyzing the results produced from the experiments.

Two sets of experiments are performed, namely the possible methods of the partitioning algorithm and the experiments for recursive circuit partitioning. A tree

structure result is presented for the first experiment. A set of solutions can be obtained by tree traversal. A set of tables consists of results produced for individual examples and a set of circuits are presented for the second experiments.

The possible solution experiment illustrates that it is very difficult to consider multiple methods during a solution search. The recursive circuit partitioning experiment selects the min cut method and produces a set of solutions for each circuit network. the final solution is selected by comparing the number of cut points and then the fault coverage.

The experimental results of the recursive circuit partitioning experiments show that we can obtain an expected FPGA configuration solution by performing the recursive circuit partitioning process. Although candidate solution sets may show different behavior for particular circuits during the solution searching, the final results are within the expected range. Fault coverage is improved significantly for each circuit under test and the size of most of the sub-circuits in the solution set is in a reasonable range. The obvious factor that affects the number of sub-circuits and the number of cut points is the size of a circuit, which is one of the factors in our expectation.

With smaller sized and higher fault coverage sub-circuits in a solution set, no extra fault location technique is required for this testing technique. Comparing to other techniques of application-dependent FPGA testing, which apply logic function

decomposition or/and separate test configurations for logic block and interconnections, this testing strategy is less complicated and time consuming.

At this stage, there are still some details of this technique that need to be polished and some methods can be improved. Future work for this strategy is discussed in the conclusion chapter.

7. Conclusions

A new strategy for application-dependent FPGA testing is investigated in this thesis. The purpose of the testing approach is to provide a more straightforward and effective way to identify fault locations on an FPGA chip for a user-defined circuit. A BIST structure, the method of test configuration and a recursive circuit partitioning process are introduced to accomplish the goal. In this Chapter, we summarize the main contributions of this research and suggest some possible directions for the future work.

7.1. Contributions

Application-dependent FPGA testing has been studied in order to provide a more efficient way to locate faulty areas on an FPGA. Some previous approaches include: 1) applying function decomposition methods to decompose a large function to smaller sub-functions so these sub-functions can be implemented on individual logic blocks; 2) testing logic blocks and interconnections separately by using different test configurations; 3) applying multiple test sessions to perform the testing on different parts of a circuit. The high fault coverage and small test complexity achieved by these methods are accompanied by the trade-off of using more complex techniques and providing less accurate test configuration solutions, or larger time complexity of the testing procedures. The technique introduced in this thesis applies a more straightforward approach and provides reasonable test configuration solutions that abide by the original function and reduce the time complexity as well as the test complexity.

A BIST architecture is designed to satisfy the different requirements for individual user-defined circuits. It is flexible to choose TPGs and ORAs in order to achieve the best result for a particular application.

A test configuration technique is provided for application-dependent testing procedures. In this technique, sub-circuits implemented on fault-free areas remain while sub-circuits implemented on faulty areas are re-configured and re-tested. Sub-circuits of the original design are connected by the completion of the test. No further re-configuration is required.

A recursive circuit partitioning algorithm is developed and implemented as the main focus of this research. The goal of the algorithm is to generate a test configuration solution for a particular user-defined circuit. The output of the algorithm is a set of sub-circuits, which are implemented on an FPGA for application-dependent testing configuration. The focus of the algorithm is to obtain sub-circuits with high fault coverage and minimum number of cut points. According to the experimental results and analysis, this approach reaches the anticipated results. It can be used to partition a large circuit and provide a feasible solution with high fault coverage and a reasonable number of cut points. The fault coverage is significantly improved for all circuits involved in the experiments and the number of cut points is approximately minimized by selecting the optimal solution from a set of candidates.

7.2. Future Work

There are many aspects of this research that can be further studied. The following topics are listed as some interesting ones.

- Develop a random algorithm for the recursive circuit partitioning process and study the probabilities of obtaining the minimum cut of a circuit graph. This may provide evidence for the trade-off between minimizing the number of cut points and maximizing the improvement in fault coverage.

- Improve the proposed circuit partitioning algorithm and obtain more balanced sub-circuits in the final solution. This resolution could provide a better configuration solution where the differences of configuration area for each sub-circuit can be minimized and the fault location can be identified more accurately.

- Verify the testing strategy on FPGA chips for a set of large circuits. This may lead to the practical usage verification on various applications.

- Consider applications of fault tolerant systems and redundant designs as the focus for the circuit partitioning process of application-dependent FPGA testing.

- Develop an automatic testing configuration tool for the testing strategy. This may reduce the time complexity of the entire testing process. Solution

evaluation can be also included in order to ensure the quality of the configuration.

- Consider other fault models for the circuit partitioning algorithm especially time related fault models such as delay faults. The configuration solution may be different when the resulting implementation is required to satisfy the time constraints of the original circuit.

Bibliography

[1] E. Ahmed and J. Rose, "The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density," in *FPGA 2000, ACM Symp. FPGAs,* February 2000, pp. 3-12.

[2] M. B. Tahoori, E. J. McCluskey, M. Renovell and P. Faure, "A multi-configuration strategy for an application dependent testing of FPGAs," in *VLSI Test Symposium,* 2004, pp. 154-159.

[3] M. B. Tahoori, "Application-dependent testing of FPGA interconnects," in *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on,* 2003, pp. 409-416.

[4] M. B. Tahoori, "Using satisfiability in application-dependent testing of FPGA," in *Design Automation Conference, 2003. Proceedings,* 2003, pp. 678-681.

[5] Altera Corporation. Stratix II GX density & logic efficiency, 2006.

[6] A. Bio. FPGA architecture for the challenge. [online]. Available: http://www.eecg.toronto.edu/~vaughn/challenge/fpga_arch.html.

[7] F. Toth. (2004, 1/15/04). The easy path to cost reductionss. [online]. Available: http://www.xilinx.com/publications/xcellonline/xcell_48/xc_easypath48.htm.

[8] S. Srinivasan, P. Mangalagiri, Y. Xie, N. Viiaykrishnan and K. Sarpatwari, "FLAW: FPGA lifetime AWareness," in *Design Automation Conference, 43rd ACM/IEEE,* 2006, pp. 630-635.

[9] M. Abramovici, M. A. Breuer and A. D. Friedman, *Digital Systems Testing and Testable Design.* USA: Computer Science Press, 1990,

[10] M. Renovell, J. Figueras and Y. Zorian, "Test of RAM-based FPGA: Methodology and application to the interconnect," in *VLSI Test Symposium, 15th IEEE,* 1997, pp. 230-237.

[11] M. Serra, "Digital IC testing: An introduction," in *EE Handbook* Anonymous CRC Press, 2005,

[12] P. R. Menon, W. Xu and R. Tessier, "Design-Specific Path Delay T esting in Lookup Table-based FPGAs," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, pp. 867-877, May. 2006.

[13] E. Atoofian and Z. Navabi, "A BIST architecture for FPGA look-up table testing reduces reconfigurations," in *Proceedings of the 12th Asian Test Symposium,* 2003, pp. 84-89.

[14] M. S. Yarandi, A. Alaghi and Z. Navabi, "An optimized BIST architecture for FPGA look-up table testing," in *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures ,* 2006, pp. 420-421.

[15] I. G. Harris and R. Tessier, "Testing and diagnosis of interconnect faults in cluster-based FPGA architectures," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* vol. 21, pp. 1337-1343, Nov. 2002.

[16] A. Krasniewski and M. Nowicka, "Application-dependent testability of FPGA-based circuits designed using functional decomposition," in *Proc. IFIP Workshop on Logic and Architecture Synthesis,* 1996, pp. 167-175.

[17] A. Krasniewski, "Design for application-dependent testability of FPGAs," in *Proc. Int'l Workshop on Logic and Architecture Synthesis,* 1997, pp. 245-254.

[18] A. Krasniewski, "Application-dependent testing of FPGA delay faults," in *EUROMICRO Conference, Proceedings. 25th,* 1999, pp. 260-267.

[19] C. Stroud, P. Chen and S. Konala, "Evaluation of FPGA resources for built-in self-test of programmable logic blocks," in *Proceedings of the Fourth International ACM Symposium on Field-Programmable Gate Arrays,* 96, pp. 107-113.

[20] Y. Choi, Y. S. Jeong and C. S. Rim, "A topology-based multi-way circuit partition for ASIC prototyping," in *Circuits and Systems, IEEE 39th Midwest Symposium on,* 1996, pp. 357-360.

[21] H. Liu and D. F. Wong, "Circuit partitioning for dynamically reconfigurable FPGAs," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays,* 1999, pp. 187-194.

[22] C. Kim, H. Shin and Y. Yu, "Performance-driven circuit partitioning for prototyping by using multiple FPGA chips," in *Proceedings of the 1995 Conference on Asia Pacific Design Automation,* 1995, pp. 113-118.

[23] D. R. Brasen and G. Saucier, "Using Cone Structures for Circuit Partitioning into FPGA Packages," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* vol. 17, pp. 592-600, Jul. 1998.

[24] S. Lu and C. Chen, "Fault detection and fault diagnosis techniques for lookup table FPGAs," in *Proceedings of the 11th AsianT Est Symposium,* 2002, pp. 236-241.

[25] M. B. Tahoori, E. J. McCluskey, M. Renovell and P. Faure, "A multi-configuration strategy for an application dependent testing of FPGAs," in *Proceedings of the 22nd IEEE VLSI Test Symposium,* 2004, pp. 154-159.

[26] M. B. Tahoori, "Application-Dependent Testing of FPGAsfor Bridging Faults," *Journal of Electronic Testing: Theory and Applications,* vol. 20, pp. 279-289, June. 2003.

[27] A. Kusiak and C. Huang, "Design of Modular Digital Circuits for Testability," *IEEE Trans. on Componets, Packaging, and Manufacturing Technology,* vol. 20, pp. 48-57, January. 1997.

[28] H. K. Lee and D. S. Ha, "
HOPE: An efficient parallel fault simulator for synchronous sequential circuits," in *Proceedings of the 29th ACM/IEEE Conference on Design Automation ,* 1992, pp. 336-340.

[29] M. Stoer and F. Wagner, "A simple min-cut algorithm," in *Proceedings of the 2nd Annual European Symposium on Algorithms,* 1994, pp. 141-147.

[30] H. H. Yang and D. F. Wong, "Efficient Network Flow Based Min-Cut Balanced Partitioning," *IEEE Computer-Aided Design of Integrated Circuits and System,* vol. 15, DEC. 1996.

[31] D. R. Karger and C. Stein, "A New Approach to the Minimum Cut Problem," *Journal of the ACM,* vol. 43, pp. 601-640, July. 1996.

[32] Y. Choi, Y. S. Jeong and C. S. Rim, "A topology-based multi-way circuit partition for ASIC prototyping," in *Circuits and Systems, IEEE 39th Midwest Symposium on,* 1996, pp. 357-360.

[33] M. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Fransisco, CA: W H Freeman & Co, 1979,

[34] H. Liu and D. F. Wong, "Circuit partitioning for dynamically reconfigurable FPGAs," in *Proceedings of the ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays,* 1999, pp. 187-194.

[35] C. M. Fiduccia and R. M. Mattheyses, "A linear time heuristic for improving network partitions," in *Proc. ACM/IEEE Design Automation Conf.* 1982, pp. 175-181.

[36] Z. Bar-Yossef. (2006, Feb. 5). Design and analysis of algorithms. [online]. Available: http://www.ee.technion.ac.il/courses/046002.

[37] E. Chmelar, "FPGA interconnect delay fault testing," in *Proc. Int. Test Conf. (ITC),* 2003., pp. 1239-1247.

[38] M. B. Tahoori, S. Mitra, S. Toutounchi and E. J. McCluskey, "Fault grading FPGA interconnect test configurations," in *Proceedings of the 2002 IEEE International Test Conference ,* 2002, pp. 608-617.

[39] E. J. McCluskey, "Built-in self-test techniques," in *IEEE Design and Test of Computers,* 1985, pp. 21-28.

[40] W. E. Huang and F. Lombardi, "An approach for testing Programmable/Configurable field pirogrammable gate arrays," in *IEEE 14'h VLSl Test Symposium,* 1996, pp. 450-455.

[41] C. Stroud, S. Konala, P. Chen and M. Abramovici, "Built-in self-test of logic blocks in FPGAs," in *IEEE 14'h VLSI Test Symposium ,* 1996, pp. 387-392.

[42] Ralf Krueger and Brent Przybus" Virtex Variable-Input LUTArchitecture" White Paper: Virtex and Virtex-II Series FPGAs. Xilinx, Inc.WP209 (v1.0) January 12, 2004.

Appendix 1: S208.1 Circuit File

```
# s208.1
# 10 inputs
# 1 outputs
# 8 D-type flipflops
# 38 inverters
# 66 gates (21 ANDs + 15 NANDs + 14 ORs + 16 NORs)

INPUT(P.0)
INPUT(C.8)
INPUT(C.7)
INPUT(C.6)
INPUT(C.5)
INPUT(C.4)
INPUT(C.3)
INPUT(C.2)
INPUT(C.1)
INPUT(C.0)

OUTPUT(Z)

X.4 = DFF(I12)
X.3 = DFF(I13)
X.2 = DFF(I14)
X.1 = DFF(I15)
X.8 = DFF(I110)
X.7 = DFF(I111)
X.6 = DFF(I112)
X.5 = DFF(I113)

I73.1 = NOT(I69)
I73.2 = NOT(X.3)
I7.1 = NOT(I66)
I7.2 = NOT(X.2)
I88.1 = NOT(X.1)
I88.2 = NOT(P.0)
I48 = NOT(P.0)
I49 = NOT(X.4)
I50 = NOT(X.3)
I68 = NOT(I69)
I105.1 = NOT(I163)
I105.2 = NOT(X.6)
I182.1 = NOT(X.5)
I182.2 = NOT(I1.2)
I148 = NOT(X.7)
I149 = NOT(X.6)
I161 = NOT(I162)
I164 = NOT(I163)
I212 = NOT(P.0)
I213 = NOT(X.1)
I214 = NOT(X.2)
I215 = NOT(X.3)
I216 = NOT(X.4)
I225 = NOT(I224)
I240 = NOT(P.0)
I241 = NOT(X.5)
I242 = NOT(X.6)
I243 = NOT(X.7)
I244 = NOT(X.8)
I252 = NOT(I251)
I282 = NOT(P.2)
I283 = NOT(P.3)
I286 = NOT(C.2)
I287 = NOT(C.3)
I306 = NOT(P.6)
I307 = NOT(P.7)
I310 = NOT(C.6)
I311 = NOT(C.7)

I73.3 = AND(I69, I73.2)
I73.4 = AND(X.3, I73.1)
I7.3 = AND(I66, I7.2)
I7.4 = AND(X.2, I7.1)
I88.3 = AND(X.1, I88.2)
I88.4 = AND(P.0, I88.1)
I105.3 = AND(I163, I105.2)
I105.4 = AND(X.6, I105.1)
I182.3 = AND(X.5, I182.2)
I182.4 = AND(I1.2, I182.1)
I191.1 = AND(I164, X.6)
I1.2 = AND(I2.1, P.0)
P.5 = AND(I209.1, I205.2)
P.6 = AND(I209.1, I206.2)
P.7 = AND(I209.1, I207.2)
P.8 = AND(I209.1, I208.2)
I295.1 = AND(P.1, C.1)
```

I295.2 = AND(P.0, C.0)
I319.1 = AND(P.5, C.5)
I319.2 = AND(P.4, C.4)
I270.3 = AND(P.8, C.8)

I70.1 = OR(I68, X.4, I50)
I13 = OR(I73.3, I73.4)
I15 = OR(I88.3, I88.4)
I95.1 = OR(I64, I50, I48)
I167.1 = OR(I165, X.8, I148)
I170.1 = OR(I165, X.7)
I113 = OR(I182.3, I182.4)
I188.1 = OR(I163, I149, I148)
I291.1 = OR(I283, I287)
I291.2 = OR(I282, I286)
I315.1 = OR(I307, I311)
I315.2 = OR(I306, I310)
I270.2 = OR(I269.1, I269.2)
Z = OR(I270.2, I270.3)

I12 = NAND(I70.1, I62)
I62 = NAND(I95.1, X.4)
I64 = NAND(X.1, X.2)
I66 = NAND(X.1, P.0)
I110 = NAND(I167.1, I159)
I111 = NAND(I170.1, I161)
I159 = NAND(I188.1, X.8)
I163 = NAND(X.5, I1.2)
I165 = NAND(I164, X.6)
I222 = NAND(I225, I214)
I224 = NAND(I213, P.0)
I249 = NAND(I252, I242)
I251 = NAND(I241, P.0)
I269.1 = NAND(I291.1, I291.2, I290)
I269.2 = NAND(I315.1, I315.2, I314)

I14 = NOR(I7.3, I7.4)
I2.1 = NOR(I64, I49, I50)
I69 = NOR(I64, I48)
I112 = NOR(I105.3, I105.4)
I162 = NOR(I148, I191.1)
P.1 = NOR(I212, I213)
P.2 = NOR(I214, I224)
P.3 = NOR(I215, I222)
P.4 = NOR(X.3, I222, I216)
I209.1 = NOR(X.4, X.2, X.3, X.1)
I205.2 = NOR(I240, I241)
I206.2 = NOR(I242, I251)
I207.2 = NOR(I243, I249)
I208.2 = NOR(X.7, I249, I244)
I290 = NOR(I295.1, I295.2)
I314 = NOR(I319.1, I319.2)