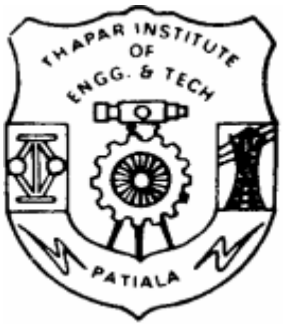
Satisfiability Application-Dependent Testing of FPGA using Quantum Computing

A thesis

Submitted in partial fulfillment of the requirement
for the award of degree
of

Master of Engineering
in
Software Engineering



Under the Supervision of
Mr. Amardeep Singh
Lecturer,
Computer Science and Engineering Department
Thapar Institute of Engineering and Technology, Patiala.

Submitted By
Surinder Pal Singh
(8023120)

Computer Science & Engineering Department
Thapar Institute Of Engineering & Technology
(Deemed University), Patiala-147004 (India)

May 2004

Declaration

I hereby certify that the work which is being presented in the thesis entitled, ***"Satisfiability Application-Dependent Testing of FPGA using Quantum Computing***", in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar Institute of Engineering and Technology (Deemed University), Patiala, is an authentic record of my own work carried out under the supervision of Mr. Amardeep Singh.

The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other University.

Surinder Pal Singh

This is to certify that the above statement made by the candidate is correct and true to best of my knowledge.

Mr. Amardeep Singh

Lecturer

Computer Sc. and Engg. Department

Thapar Institute Of Engg. And Technology, Patiala

Countersigned by:

(Seema Bawa)

Assistant Professor & Head,

Computer Sc. & Engg. Department,

Thapar Institute of Engg. & Technology,

Patiala.

(Dr. D.S. Bawa)

Dean (Academic Affairs)

Thapar Institute of Engg. & Technology,

Patiala.

Acknowledgement

First of all, I am greatly thankful to that Almighty, who blessed me with the determination and enthusiasm to complete this work.

I am very grateful to Mr. Amardeep Singh, Lecturer, and Computer Science & Engineering for their enthusiastic support, invaluable guidance, suggestions and motivation throughout the course of this thesis work, and I owe much to his wisdom, generosity and patience.

I am also thankful to Ms. Seema Bawa, Assistant Professor & Head, Computer Science & Engineering Department and Mr. Rajesh Bhatia, Assistant Professor & P.G. Coordinator, for their constant attention, support and inspiration that triggered me for the thesis work. I am also thankful to all the faculty and staff members of the Computer Science & Engineering Department for providing me all the facilities required for the completion of this work.

I would like to thank my friend Vinay Chopra, Rohatsh B. Pandhi, Umesh Gupta and Bupinder Singh, who was always with me at the time of need. And of course, my parents and family, who have always supported me and have been willing to make considerable sacrifices in order to give me all possible advantages in life.

(Surinder Pal Singh)

Abstract

The potential of Quantum computing has been used to solve many computationally hard problems. An algorithm for solving the Boolean satisfiability problem (SAT) or (K-SAT) on quantum computers, for testing the interconnects of an arbitrary design mapped into an FPGA. Satisfiability Application-dependent Testing of FPGA is a compute intensive problem. Existing conventional methods are unable to perform the required breakthrough in terms of complexity, time and cost. An evolutionary approach based on Quantum computing is presented, which exploits the computational power of Quantum Parallelism to solve the problem. A Boolean formula in conjunctive normal form is extracted from the FPGA under test and then the proposed algorithm based on Quantum computing is used to find the solution satisfying that formula. The test vector and configuration generation problem is systematically converted to a Quantum problem, which runs in a constant number of steps with any given number n of Boolean variables. Exploiting the massive parallelism and recombination properties of quantum, a test vectors is generated in polynomial time. Its effectiveness in terms of number of iterations is experimentally compared with some of existing approaches like simulated annealing and genetic algorithms. Using the proposed Quantum based approach it is also possible to find all the test vectors, which detect a particular fault, simultaneously.

List of Figures

List of Tables

Chapter 1

Introduction

A Field Programmable Gate Array (FPGA) is an integrated circuit that can be programmed in the field to have vastly wider potential application than, programmable read-only memory chips. FPGAs [1][2] are used by engineers in the design of specialized ICs that can later be produced hard-wired in large quantities for distribution to computer manufacturers and end users.

In the mid 1980s a new technology for implementing digital logic was introduced, the Field Programmable Gate Array (FPGA). These devices could either be viewed as small, slow gate arrays or large, expensive programmable logic devices (PLDs). FPGAs were capable of implementing significantly more logic than PLDs, especially because they could implement multi-level logic [2][3], while most PLDs were optimized for two-level logic. Field Programmable Gate Arrays (FPGAs) are also completely prefabricated, but instead of two-level logic they are optimized for multi-level circuits. This allows them to handle much more complex circuits on a single chip, but it sacrifices the predictable delays of PLDs. Note that FPGAs are often considered another form of PLD [2], often under the heading Complex Programmable Logic Device (CPLD).

Field programmable gate arrays are specific integrated circuits that can be user-programmed easily. The FPGA contains versatile functions, configurable interconnects and an input/output interface to adapt to the user specification. FPGAs allow rapid prototyping using custom logic structures, and are very popular for

limited production products. Modern FPGA are extremely dense, with a complexity of several millions of gates, which enable the emulation of very complex hardware such as parallel microprocessors, mixture of processor and signal processing, etc. The key advantage of FPGA is their ability to be reprogrammed [3], in order to create a completely different hardware by modifying the logic gate array.

By the early 1980's most of the logic circuits in typical systems where absorbed by a handful of standard large scale integrated circuits (LSI)[1] like Microprocessors, bus/IO controllers, system timers. Custom ICs was previously designed to replace the large amount of glue logic, which reduced system complexity and manufacturing cost, improved performance. However, Custom ICs are relatively very expensive to develop, and delay introduction of product to market (time to market) because of increased design time. Therefore the custom IC approach was only viable for products with very high volume (where NRE could be amortized), which were not time to market sensitive. FPGAs continue to compete with custom ICs for special processing functions (and glue logic) but now also compete with microprocessors [1][3] in dedicated and embedded applications.

Just as in PLDs, FPGAs are completely prefabricated, and contain special features for customization. These configuration points are normally either SRAM cells or antifuses. Antifuses are one-time programmable devices, which when "blown" create a connection, while when "unblown" no current can flow between their terminals. Because the configuration of an antifuse is permanent, antifuse-based FPGAs are one-time programmable, while SRAM-based FPGAs [14] are reprogrammable, even in the target system.. Note that FPGAs often have on-chip

control circuitry to automatically load this configuration data. SRAM cells are larger than antifuses, meaning that SRAM-based FPGAs [14] will have less configuration points than an antifuse based FPGA. Since they are reprogrammable, their configurations can be changed for bug fixes or upgrades.

Field programmable gate arrays (FPGAs) are programmable platforms for lots of applications such as networking, signal processing, and fault tolerant computing. In an SRAM-based FPGA [14], all logic elements and programmable switches can be reprogrammed by loading a configuration bitstream, giving an FPGA the flexibility to implement any digital circuit on the same piece of silicon. Generally, FPGAs can be configured in an incredibly large number of ways, in the order of billions. Since the reconfiguration time [5], time to load a configuration, varies between few seconds up to few minutes depending on the size of FPGA.

It is impossible to map all possible configurations during testing. Therefore, a set of test configurations based on a fault list must be developed in order to ensure the part is defect-free. Hence, interconnect testing is a very challenging problem [4][6] covering all faults in wires and programmable switches in a reasonably small number of test configurations.

These designs are not fully testable using conventional ASIC ATPG tools. On the other hand, some FPGA chips that do not pass manufacturing test may still be usable for some specific designs. In this case, the defects are located in some areas of the chip, which are not used by a particular design. By testing the resources of an FPGA with respect to a specific design to be implemented on it, some faulty chips can be sold to customers. These FPGAs, which are good only for particular designs and

do not have general programmability of typical FPGAs, are called application-specific FPGAs (ASFPGAs). ASFPGAs are profitable for relatively large volume designs, which have been completely finalized, i.e. the final placed and routed version is fixed. FPGA vendors can benefit from application-dependent testing of FPGAs [10], which tests only the FPGA resources used on a particular design, in order to increase the yield and make money from those chips that are previously marked as rejected parts. In this technique, every CLB used in the mapped design is reconfigured to implement transparent logic in the LUTs [13] followed by flip-flips in order to construct scan chains. Also, fanout branches of a net are tested in different test configurations, i.e. dependent logic cones are tested in different configuration, resulting in a few test configurations.

A new testing technique for application-dependent testing of bridging faults [10][13] in FPGA interconnects is presented. We consider both wired-OR and wire-AND bridging fault models. In the presented method, only the logic blocks of the FPGA used by the mapped design are reprogrammed to implement some special type of logic functions. Hence, no extra placement and routing are necessary for test configuration generation. The test configuration generation problem is systematically converted into a satisfiability (SAT) problem [23], and state-of-the-art SAT-solvers are exploited to solve this problem. The logic implemented in the logic blocks can be tested using well-known techniques for testing logic blocks.

A new technique for testing the interconnects of an arbitrary design mapped into an FPGA is presented. In this technique, only the configuration of logic blocks used in the design is changed. The test vector and configuration generation problem is

systematically converted to a satisfiability (SAT) problem [27], and state of the art SAT solvers are exploited for test configuration generation.

Quantum parallelism is considered as the magic key, which gives the quantum computers the ability to do some types of computation more powerfully than any classical computer Quantum parallelism is the ability to do processing simultaneously on many states because quantum systems can exist in a superposition of that states. Many quantum algorithms that have been presented recently use quantum parallelism. For example, Shor [20] presented a quantum algorithm for factorizing a composite integer into its prime factors in polynomial time. Grover [18] presented an algorithm for searching unstructured list of n items that runs in O ($\sqrt{n}$), an enhanced version of Grover's algorithm was also presented [19]. Many researchers are trying to find quantum algorithms for NP-complete problems [19] on quantum computers hoping that they can decrease the complexity class of those algorithms. The satisfiability problem is a typical NP-complete problem gaining a lot of attention. People are trying to handle it on quantum computers as a search problem; Grover [41] showed that his algorithm can be used to solve the propositional satisfiability problems with O ($\sqrt{2}^{n}$).

We will present a quantum algorithm for solving the general SAT problem, which runs in a constant number of steps; with any given number n of Boolean variables in the expression. We will see that in contrast to classical search algorithms, the probability that the algorithm may succeed to find a solution increases as the number of variables increases. We will present an algorithm for solving the Boolean satisfiability problem (SAT) or (K-SAT) on quantum computers, which runs in a constant number of steps; O(4), with any given number n of Boolean variables. We

will show that in contrast to classical algorithms the ability of the algorithm to solve the problem increases as the number of variables increases.

This thesis is organized as follows: Chapter 2 gives a brief introduction to FPGA architecture and its physical design of FPGA i.e. Partitioning, Placement and Routing.

Chapter 3 provides a brief overview of the various types of FPGA testing techniques. It also explains in detail the various approaches of ATPG. This gives a brief idea of various approaches applied to FPGA and Application-Dependent Testing of FPGA Interconnects is discussed in detail.

Chapter 4 gives a brief introduction to Quantum Fundamentals and tells about the Quantum gates, which is applied to solve the satisfiabilty problem.

Chapter 5 gives brief introduction to how Satisfiabilty solve the Application-Dependent Testing of FPGA Interconnects and, the problem at hand.

Chapter 6 shows how the Quantum satisfiabilty based algorithm solves the problem of Application-Dependent testing. This chapter discusses in detail the proposed algorithm, its explanation and experimental results and the last chapter 7 gives the conclusion of the thesis.

Chapter 2

FPGA Basics

The FPGA is an integrated circuit that contains many (64 to over 10,000) identical logic cells that can be viewed as standard components. Each logic cell can independently take on any one of a limited set of personalities.

2.1. FPGA Architecture

Architecture mainly consists of two parts: the Logic Blocks, and the Routing networks. A logic block has a fixed numbers of inputs and one output. A wide range of functions can be implemented using a logic block., FPGA is first decomposed into a smaller sub-circuits such that each of the sub-circuit can be implemented using a single block. Logic Blocks consists of Look-Up Tables (LUTs), and Multiplexes.

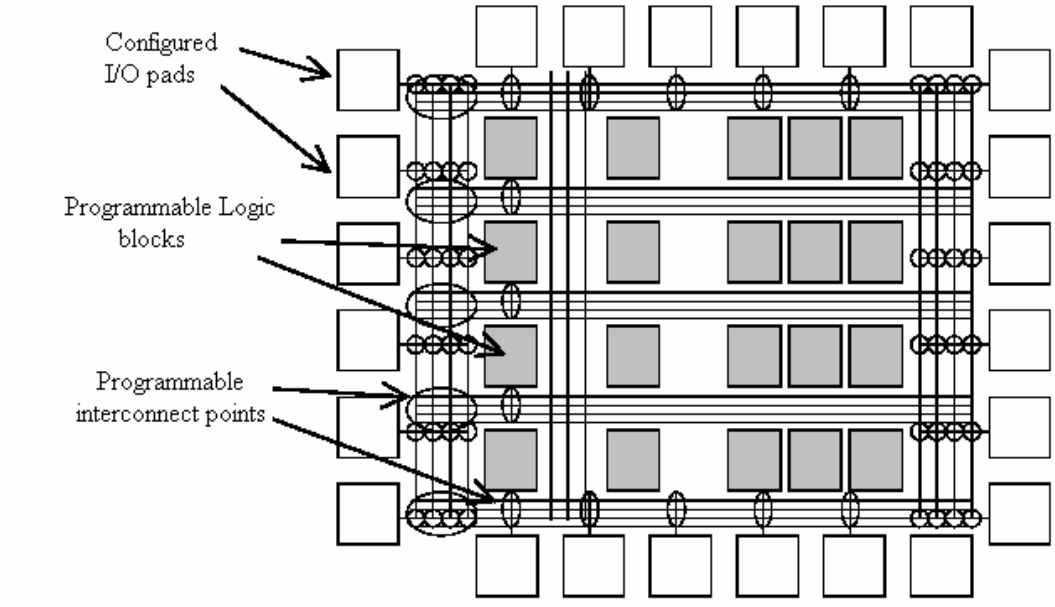


Fig 2.1 FPGA Architecture

The core of the device consists of simple logic modules used to implement the required logic gates and storage elements. These logic modules are interconnected with an abundance of segmented routing tracks. Unlike gate arrays, the segment lengths are predefined and can be connected with low-impedance switching elements to create the precise routing length required of the interconnect signal. This interface consists of I/O modules that translate and interconnect the logic signals from the core of the device to the FPGA output pads.

2.2. Physical design cycle for FPGAs

The Physical design for FPGAs consist of following steps:

2.2.1. Partitioning

The circuit to be mapped on FPGAs has to be partitioned into smaller sub circuits such that each sub circuit is to be mapped to a programmable logic block. Unlike the partitions in other design styles there are no constraints on the size of a partition. There are constraints on the inputs and outputs of a partition. FPGA systems are used for a great variety of applications such as dynamically re-configurable hardware applications, digital circuit emulation, and numerical computation.

The FPGA-based partitioning problem is quite different from the classical partitioning problem. This often results in FPGA partitions with very low logic utilization. In order to achieve timing improvement, conventional methods usually

impose higher weights on the critical paths during the partitioning process so that the delays on these critical paths can be improved by making sure components on these paths end up on the same chip.

Performance estimation involves accurate estimation of cost of the design and the partitioning involves the proper choice of design segments that satisfy user specified constraints. Pin constraints are also taken into account while fixing the partitions. If the design cannot be partitioned into the available number of chips, each with allowed number of I/O pins, the partitioner returns the best possible solution

Fig 2.2 Physical Design Cycle of FPGA

2.2.2. Placement

In this step of the design style the sub circuits which are formed in partitioning phase are allocated physical location on the FPGA i.e. logic block on the FPGAs is programmed to behave like the sub circuit that is mapped to it. This placement must to be carried in manner that the routers ca complete the interconnection, this is very critical as the routing resources are limited .the placement algorithm are general gate arrays normally used for the placement in FPGAs.

Placement is the operation of assigning each vertex of the Boolean network to a specific logic block in the FPGA device. The goal of placement for island-style FPGAs is to create a placed configuration of logic blocks that can be successfully interconnected in a subsequent routing step given the routing resources available. Swapping pairs of blocks in an effort to and intermediate placement configurations that have lower overall cost can optimize an initial logic block placement.

2.2.3. Routing

In this phase, blowing the fuse between the routing segments to achieve the interconnection interconnects all the sub circuits, which have been programmed on the FPGAs block. The programmable routing in an FPGA consists of two categories: (1) routing within each Logic Block/ Logic Cluster, and (2) routing between the Logic Blocks/Logic Clusters. An FPGA is created by replication of tile (a tile consists of one Logic Block and it's associated routing). The programmable routing within each Logic Block consists of the Interconnect Matrix. The programmable routing between the Logic Blocks consists of fixed metal tracks, Switch Blocks, Connection Blocks, and the programmable switches. The Connection Block defines all the possible connections

from a horizontal or vertical channel to a neighboring logic block. The connections in the switch blocks and connection blocks are made by

programmable switches. Part of the programmable routing also lies within each logic block, determining how different components are connected within the logic block.

Chapter 3

FPGA Testing

If integrated circuit test is difficult, Field Programmable Gate Arrays (FPGAs) represent a special and particularly difficult type of digital circuit. Because FPGA devices are highly configurable, it is often difficult to isolate and exercise all elements of the architecture. In addition, modern FPGA devices represent some of the largest, most complex integrated circuit devices available.

3.1 Testing Techniques of FPGA

FPGA tests so that faulty components can be quickly identified and recovered. Given the large range of applications and programmable configurations each FPGA device may support, an FPGA test can be substantially more complex than application-specified integrated circuit (ASIC) test, providing motivation for new efficient testing techniques.

Information regarding defect location is particularly important in today's test environment since new techniques have been developed that can reconfigure FPGAs to the fault be clearly identified.

During the testing process, a portion of the FPGA is configured as test generation and response circuitry for a cluster under test. As individual logic clusters and surrounding routing resources are verified, they subsequently may be used to perform testing on remaining, untested clusters. To demonstrate the approach, we present a technique to generate FPGA test configurations to detect and diagnose pair

wise bridging interconnect faults. By restricting the programming of the lookup tables in the FPGA, we formulate the testing and diagnosis conditions as a set of straightforward functions of the inputs of each tile. The testing and diagnosis conditions for each fault and fault pair are used to direct the configuration process. Our approach partitions the test configuration definition process to greatly improve the efficiency of the process. Since test configurations in our approach are replicated across the cluster array, the process of defining test configurations is independent of the size of the FPGA array avoid faults. The Different types of FPGA Testing techniques are :

- **Boundary Scan Test**
- **Fault modelling**
- **Built-in self Test**
- **Automatic Test-Pattern Generation (ATPG)**

3.1.1. Automatic Test-Pattern Generation (ATPG)

Automated Test Pattern Generation (ATPG) for digital circuits is a major research topic. As far as the single stuck-at fault model is considered, efficient algorithms have been devised for combinational networks. The economical importance of ATPG tools for digital circuits is continuously growing, and the demand for efficient algorithms and tools able to handle the current circuits is thus very strong. Correspondingly, there have been significant research efforts in this field, which produced tens of proposals in terms of ATPG algorithms and techniques.

In the last years, one of the main goals of these efforts was to develop effective algorithms for sequential circuits. Due to the great increase in the circuit size

and complexity, this task is now critical from the point of view of the required computational power, and a significant amount of research activities has been devoted to it in the past years1. The stuck at fault model has generally been adopted, and efficient algorithms have been proposed for combinational networks. On the opposite side, the problem of effectively dealing with very large sequential circuits is still open, even if some commercial tools are already available. In fact, no proposed method is able to successfully handle the complete set of real-world circuits test engineers have to face with: they either definitely get off or generate very poor results.

Very large sequential circuits still constitute a problem. None of the methods proposed can fully handle all the real-world circuits test engineers have to deal with, since they either definitely get off or generate sequences with very low fault coverage's. Three types of approaches have been suggested:

- **Topological Approach**
- **Symbolic Approach**
- **Simulation based Approach**

3.1.1.1. Topological Approach

The traditional topological approach [16], is based on extending to sequential circuits the branch and bound techniques developed for combinational circuits by adopting Huffman's Iterative Array Model, and the method's effectiveness heavily relies on the heuristics adopted to guide the search. The approach uses a complete algorithm, and therefore it can be exploited to identify untestable and redundant faults. Thresholds are normally introduced to avoid the exploration of the whole

search space, and redundancy identification often fails when applied to large circuits, where the search space is excessively large to be explored.

Random search and Exhaustive search techniques are used in Topological approach .The most widely used ATPGs usually adopt topological techniques, which suffice for circuits of moderate size and complexity. When highly sequential circuits are considered, the backtracking [16] activity inherent in the search space exploration reduces the quality of tests considerably because of the increased number of aborted. Most topological ATPG algorithms generate the test sequence for every fault by performing the following three phases:

- **Fault Excitation:** This phase is performed at time frame t0 to find a state and a set of values to be applied to the circuit Primary Inputs that excite the fault. The state is typically partially specified.
- **Fault Propagation:** Fault effects are propagated up to a Primary Output, either in time frame *t0* or in a succeeding time frame. Further assignments may be made to the state in time frame *t0*
- **Fault Justification:** A sequence of vectors that justify the state required in time frame t0 is obtained.

If conflicts are found during any of the three phases, the test generator backtracks to a previous decision point and makes an alternative decision. State justification is performed by reverse time processing [16][17]. Time frames are processed in reverse order until states containing all don't care values is reached. If a desired state is determined to be unreachable, then a backtrack is performed. If a state is

not determined to be unreachable, then the topological algorithms for fault propagation or state justification are used.

3.1.1.2 Symbolic Approach

The recently explored symbolic approach exploits techniques, which were initially developed for formal verification, based on the extraction and manipulation of the Boolean functions implemented by the circuit. This approach is based on a complete algorithm, too, and is very effective when small- and medium-sized circuits are considered. Being an exact method, it can identify all un testable faults [17] for circuits it is able to deal with. Unfortunately, it is completely inapplicable when dealing with circuits having more than some tens of Flip-Flops. This greatly limits its usefulness in practice.

Boolean Satisfiabilty is the technique used in the symbolic approach Symbolic Techniques can be used to reason about the behavior of a macro, regardless its Topology.. In this case, the circuit is modeled as a Finite State Machine, and is represented by the Boolean state transition function [17] computing the next state y from the current state s and the current input x, $y = \delta(s, x)$, and by the output function computing the output z starting from the same information, $z = \delta(s, x)$. Such functions are extracted from the circuit net list, and completely describe its input/output behavior.

The characteristic function of the state transition function $y = \delta(s, x)$ is a compact representation of all the triples (s, x, y)[17][18] satisfying function : the characteristic function $\chi\delta(s, x, y)$ takes the value 1 for all the triples (s', x', y') such that y'= δ(s',

x'). Using such a representation, it is extremely easy to extract the information items that are needed during ATPG:

- **Propagation:** given a set of input constraints and state constraints, compute the corresponding permitted output values. If the sets of allowed inputs and states are modeled by their characteristic functions, δx (x) and $\delta s(s)$, respectively, then the characteristic function of the output set of values is $\delta y(y) = \exists x, s\ [\delta x(x), \delta s(s), \chi\delta(s, x, y)]$.
- **Justification:** the same idea applies when trying to compute input constraint starting from output knowledge: $\delta x\ (x) = \exists y, s\ [\delta y\ (y)\ \delta s(s)\ \chi\delta(s, x, y)]$.
- **State traversal:** knowing which states are accessible to a circuit is essential to avoid trying to justify an invalid state. These equations can be efficiently computed when all the functions are represented by their respective BDD.

These equations can efficiently be computed when all the functions are represented by their respective BDD. Symbolic approaches alone, however, do not represent a general solution to the ATPG problems; reasonable size can be computed for small and medium circuits, only. Symbolic techniques are applied to small macros. Thus, the symbolic techniques are used to provide an early backtrack indication only.

3.1.1.3. Simulation based Approach

The simulation-based approach consists of generating pseudo random sequences; fault simulating them, and then modifying their characteristics to increase the obtained fault coverage. Simulation-based approaches can be adopted thanks to the recent advancements of state-of-the-art fault simulation techniques.

We use DNA, Genetic Algorithm, Simulation Annealing and Quantum approaches in Simulation based approach. Simulation based approach consists of generating pseudo random sequences [18], fault simulating them, and then modifying their characteristics to increase the obtained fault coverage. Simulation-based approaches can be adopted thanks to the recent advancements of state-of-the-art fault simulation techniques.

The test generation process can be organized as two consecutive phases: fault excitation and fault propagation. To excite a Target Fault, TF, the ATPG has to drive an appropriate set of values on the circuit Primary Inputs (PIs)[17][18] and on the circuit Pseudo Primary Inputs (PPIs), i.e., on the flip-flops outputs. Therefore, to excite a target fault TF we need first to bring the circuit to a state where it is excited. We refer to this state as a fault excitation state. For a given fault TF, several fault excitation states may exist; hard-to-test faults typically have a small set of excitation states. On the other hand, in order to propagate a TF toward a Primary Output (PO), the ATPG has to compute a sequence of input patterns able to drive the circuit from the fault excitation state through several fault propagation states[16] which allow the fault to reach a PO.

Although the notion of fault excitation/propagation state is important to improve the ATPG capabilities, it is not enough to attain high fault coverage. Simulation-based approach is stopped when the four phases have been repeated for a given number of iterations. Given such assumptions, let us explain how the phases interact:

- **Phase (1)** has the objective of maximizing the fault-free gate activities, by exploiting information coming from logic simulation; it starts from a random population of sequences, and outputs a population which maximizes the circuit gates activity;
- **Phase (2)** computes sequences in which the fault free circuit reaches as many states as possible; it works on a population coming from phase (4), trying to add as many new states as possible to the set of already reached states;
- **Phase (3)** relies on fault simulation only, since it performs fault dropping and selects the faults which will be addressed by the following phase; its input is a population of sequences coming either from phase (1) or phase (2), while its outputs are the set of the least active gates within the circuit and the population of sequences coming from the previous phase;
- **Phase (4)** tries to propagate the previously selected faults toward the circuit outputs by means of both logic and fault simulation; its inputs are a population of vectors and a set of gates on which its attention is to be kept; its outputs are a set of test sequences and the initial population for the next phase (2).

3.2. Application-Dependent Testing of FPGA Interconnect

Field programmable gate arrays (FPGAs) are two-dimensional arrays of logic blocks and programmable switch matrices, surrounded by programmable input/output blocks on the periphery. FPGAs are widely used in many applications such as networking and adaptive computing, due to their reprogramability, reduced design cycle and time-to-market compared to conventional ASIC design flow. More than

80% of the transistors in an FPGA are used in the interconnect network. Hence, FPGAs are vulnerable to bridging faults in the interconnects [10]. Test generation for bridging faults is more complicated than traditional testing for stuck at faults. Testability issues are not considered in the design flow using FPGAs, as FPGA users typically rely on manufacturing test of FPGAs. There are no scan chains, BIST circuitry, or test points in typical FPGA-based designs.

Some FPGA chips that do not pass manufacturing test may still be usable for some specific designs. In this case, the defects are located in some areas of the chip which are not used by a particular design. By testing the resources of an FPGA with respect to a specific design to be implemented on it, some faulty chips can be sold to customers. These FPGAs, which are good only for particular designs and do not have general programmability of typical FPGAs, are called Application-Specific FPGAs (ASFPGAs). ASFPGAs are profitable for relatively large volume designs, which have been completely finalized, i.e. the final placed and routed version is fixed. FPGA vendors can benefit from application-dependent testing of FPGAs [13], which tests only the FPGA resources used on a particular design, in order to increase the yield and make money from those chips that are previously marked as rejected parts. This strategy has been adopted by Xilinx [10]. Application-dependent testing of FPGAs is described in [13].

In this technique, every CLB used in the mapped design is reconfigured as transparent logic followed by flip-flips in order to construct scan chains. Also, fanout branches of a net are tested in different test configurations, i.e. dependent logic cones are tested in different configuration, resulting in a few number of test configurations.

Due to complexity of configuration generation algorithm, it cannot be applied to large designs. In this method only the logic blocks of the FPGA used by the mapped design are reprogrammed to implement some special type of logic functions. The test configuration generation problem is systematically converted into a satisfiability (SAT) problem [26], and state-of-the-art SAT-solvers are exploited to solve this problem.

Some specific logic functions are introduced to be implemented in the logic blocks of mapped designs. Consider a logic function with m inputs. Each of the 2m input combinations corresponds to a term in the truth table of the function. A single-term function is a logic function which has only one minterm or only one maxterm. In other words, the truth table for a single-term function, F, has only one row with $F = 1$, or only one row with $F = 0$.

The input corresponding to this specific minterm or maxterm is called the activating input. This function has only one minterm, *A'BC'D'*. Note that *A/1* (*A* stuck-at-1 fault) is detectable under the applied input pattern, as the faulty output is 0, which is unequal to fault-free output [21]. Similarly*, B/0, C/1, and D/1* are also detectable. Consider the bridging fault between and *B* (denoted by *ABFB*). In the case of wired-and (WAND) model, both inputs become 0, and the effective input is *ABCD* = 0000, and faulty Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or redistribute to lists, require prior specific permission and/or a fee output is F(0000) =

0. Otherwise, in the case of wired-or (WOR), both inputs become 1 and faulty output is $F(1100) = 0$. In both cases, the bridging fault is detectable. Similarly, *BBFC* and *BBFD* are also detectable. A fault on a node is sensitized if the applied test vector sets that node a logic value opposite to the fault value. The following theorem generalizes the above example and explains the conditions for detectability of faults in single-term functions.

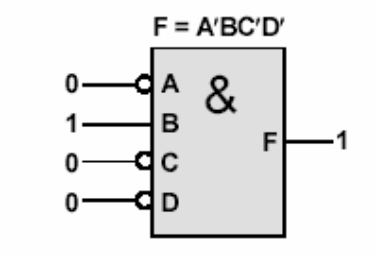


Fig 3.1 A Single-Term Function with Activating Input Pattern

Theorem 1. For a single-term function *F*, if the applied input vector is the activating input V, all sensitized stuck-at and bridging faults are detectable.

Proof. Consider a fault *f* (stuck-at or bridging fault)[13] such that it is sensitized, i.e. if *f* is *n/v*, *n* is set to *v '* by applying *V*, and if f is $a_{BF}b$, *a* and *b* have different values in *V*. In order to detect f, it is only sufficient to propagate the fault to the output. Since the fault is already sensitized, the fault term, the term corresponding to the faulty inputs, C_f, is different from the original term[13], C_V. Because the original term is the activating term, the value of the fault term must be different from the value of original term, $C_V \neq C_f$ as *F* is single-term. Hence, the fault-free output is different from the faulty output and the fault is detectable.

Theorem 2. Consider a network of single-term functions *N*, and the input pattern *V*, such that the inputs of every block is the activating input of that block. Then, all the

activated faults are detectable. In other word, for every net n with value V_n, n stuck-at V_n ′ is detectable, and for each pair of nets, n_i and n_j, with $V_{ni} \neq V_{nj}$, the bridging fault between ni and nj is detectable.

Proof. In this general case, the propagation path from the fault site to the primary output (PO) may consist more than one block. Consider a propagation path from the fault site to a PO. Based on Theorem 1, the fault effect appears on the output of the first block in the path, as a fault in the input of the second block in the path. Based on an induction on the length of the path, the fault reaches to the PO. Hence, the fault can be detected.

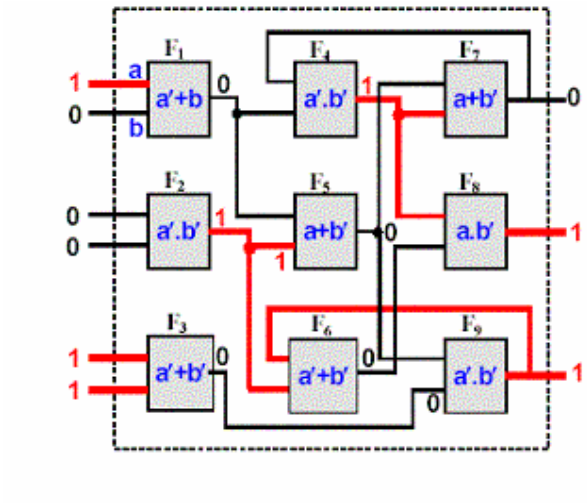


Fig 3.2 A logic Network for Single-Term Function

Note that if the network is sequential, i.e. some single-term functions are followed by flip-flops, the above theorem is still applicable. In this case, the preset value of the flip-flop must be equal the value of the net connected to its D input, as defined in the conditions of Theorem 2. For example, if F_5 is followed by a flip-flop in Fig.3.2, the preset value of that flip-flop must be 0, the value of the output of F_5. In this case, the required number of test clock cycles is equal to the maximum sequential

depth of the network, the maximum number of flip-flops on a path from a PI to a PO. The test vector must remain unchanged during all these test clock cycles.

Theorem 3. Consider an arbitrary network of (combinational) logic functions, N, and a given test vector *V*. If for every net n with value *Vn*, n stuck-at *Vn′* is detectable, then all the functions in the network must be single-term functions whose inputs are the activating inputs.

Proof. Consider an arbitrary function *F* in the network *N*. Let say the inputs of *F* under the primary input vector *V* is *v*. In the presence of any multiple stuck-at faults in the input pins of F[13], the actual input values seen by *F* change from v to a faulty value *v′*. Since all the possible stuck-at faults must be detectable, all combinations of input values other than *v* can occur as faulty inputs of *F*.

If F has m inputs, *v′* can take all *2m – 1* possible combinations other than v. The necessary condition to detect each fault in the inputs of *F*[13] is to propagate it to the output of *F*. Hence $F(v') \neq F(v)$ for all *2m – 1* possible values for *v′*. This implies that *F* is single-term and v is the activating input. Since we chose *F* as an arbitrary function in *N*, this property must hold for all the functions in *N*, so the proof is complete.

There are three types of FPGA interconnect testing :

- **Stuck –at Faults**
- **Bridging Faults**
- **Complete Test Set**

3.2.1. Stuck-at Fault

A short between two wires can be both Wired-AND and Wired-OR. Also it is to be noted that a stuck-closed CIP will create a short between two wires and a stuck-open CIP will create an open between the wires. The wires are divided into several bunches, where a bunch is a group of wires that may have a pair-wise shorts, but not every wire is necessarily adjacent with every other wire in the bunch and wires in the different bunches are not adjacent [7][12]. For example the entire horizontal wires located between two adjacent PLB rows can be treated as a bunch, even though some of the shorts are not physically feasible.

This approach makes interconnect testing easy to handle and makes the method layout independent. In order to detect the above mentioned faults, the applied tests must check if every wire and CIP is able to pass both 1 and 0 correctly and that every pair of wire segments that may be shorted can pass all four combinations of 1 and 0 namely (1; 0); (1; 1); (0; 1); (0; 0). This will make sure that both wired AND and wired OR are tested.

3.2.2. Bridging Faults

It is not appropriate to consider bridging faults between all possible pairs of nets in the design. First, the size of the fault list becomes intractable. Second, bridging faults between some pairs of nets are improbable due to physical neighborhood obtained from layout information. Inductive fault analysis (IFA)[11][15] techniques are proposed to extract the fault list from the physical layout information [Ferguson 88]. These techniques are very time consuming for a medium to large size designs. The presented technique supports both internally generated fault list for bridging fault,

or any user-specified fault lists. All the nets in the design are partition into some groups. All the nets in the same group are tested for all possible pair-wise bridging faults. A group of m nets has *m(m-1)/2* bridging faults.

The bridging fault between two nets in different groups is not included in the fault list. In the internally generated fault list, all the inputs of a look-up table (LUT) are considered in the same group. Hence, the number of the groups is equal to the number of LUTs [12] used for the design. As a result, the size of the fault list is a constant factor of the number of logic blocks used in the design. Note that this fault list exploits physical neighborhood, since the nets in different blocks have less probability of bridging faults than those in the same block.

3.2.3. Complete Test Set

In this section, a fault list is considered which is more comprehensive that stuck-at fault list. The fault list consists of stuck at and all combinations of bridging faults for all the nets of the mapped design. Note that the number of all combinations of bridging faults for a design with n nets is equal to *n (n-1)/2* faults. In conventional bus testing at board-level, only *[log2 (M+2)]* test vectors are sufficient to test for all possible stuck-at, open, and bridging faults for M bus lines [7][11][13].

The test vectors correspond to the Walsh code, which are the columns of binary representation of the numbers from 1 to M using *[log2 (M+2)]* bits. Fig 3.3 shows the test vectors for 6 wires. The number of required test vectors is *[log2(6+2)] = 3*. Note that all-0 and all-1 patterns are not used, as test vectors; otherwise one stuck-at-1 and one stuck-at-0 fault cannot be detected. These test vectors are used in conjunction with single-term functions to generate test configurations for the

presented fault list. As a result, adding this technique to single-term functions, all the bridging faults can be detected.

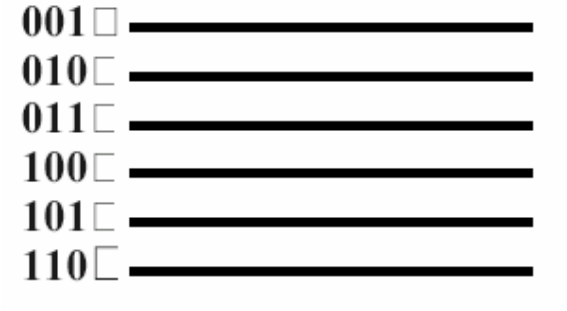


Fig.3.3 Complete Test Set for 6 Wires

So, for a mapped design with N nets, we transform each of *[log2 (N+2)]* test vectors for all the nets in the mapped design into the activating inputs for LUTs. Based on the activating inputs for each LUT, the single-term logic function of the LUT is determined. Note in this approach,[13] that no ATPG algorithm is needed for test configuration generation. As a result, test configuration generation is very fast.

Chapter 4

Quantum Computing

4.1. Introduction

Why build a quantum computer? Because it's not there, for one thing, and the theory of quantum computation, which far outstrips the degree of implementation as of today, suggests that a quantum computer would be a incredible machine to have around. It could factor numbers exponentially faster than any known algorithm, and it could extract information from unordered databases in square-root the number of instructions required of a classical computer. As computation is a nebulous field, the power of quantum computing to solve problems of traditional complexity theory is essentially unknown. Quantum computers [23][24] would take exponentially .By their very nature, quantum computers would take exponentially less space and time than classical computers to simulate real quantum systems, and proposals have been made for efficiently simulating many-body systems using a quantum computer.

Quantum Computation and Quantum Information is the study of the information processing tasks that can be accomplished using quantum mechanical systems. Quantum computation [31] taught us to think physically about computation, and this approach yields many new and exciting capabilities for information processing and communication. Quantum computing believes that what is computable and the laws of physics limit what is not computable.

Traditional computer science is based on Boolean logic and algorithms. Its basic variable is a bit with two possible values, 0 or 1. These values are represented in the computer as stable saturated states off or on. Quantum mechanics offer a new set of rules that go beyond this classical paradigm. The basic variable is now a qubit, represented as a vector in a two dimensional complex Hilbert space. $|0\rangle$ and $|1\rangle$ form a basis in this space. The logic that can be implemented with such qubits is quite distinct from Boolean logic, and this is what has made quantum computing exciting by opening new possibilities.

Quantum computation and Quantum information provide a useful series of challenges at varied levels of difficulty for people devising methods to better manipulate single quantum systems[31], and simulate the development of new experimental techniques. The ability to control single quantum systems is essential if we are to harness the power of quantum mechanics for applications to quantum computation and quantum information.

Quantum effects will begin to interfere in the functioning of electronic devices as they are made smaller and smaller. Now the question is will Moore's law fail? The answer is no because one possible solution to the problem posed by the eventual failure of Moore's law is to move to a different computing paradigm. One such paradigm is provided by the theory of quantum computation, which is based on the idea of using quantum mechanics to perform computations, instead of classical physics.

4.1.1. Quantum Algorithms

The spectacular promise of quantum computers is to enable new algorithms (called quantum algorithms) that give solutions to problems requiring exorbitant resources for their solution on a classical computer. Two broad classes of quantum algorithms are known which fulfill this promise. The first class of algorithms is based upon Shor's quantum Fourier transform [32] and includes remarkable algorithms for solving the factoring and discrete logarithm problems. These algorithms provide a striking exponential speed up over the best-known classical algorithms. The second class of algorithm is based upon Grover's algorithm for performing quantum searching. These provide a remarkable quadratic speed up over the best possible classical algorithms.

A quantum memory register can exist in a superposition of states, each component of this superposition may be thought of as a single argument to a function. A function performed on the register in a superposition [32] of states is thus performed on each of the components of the superposition, but this function is only applied one time. Since the number of possible states is 2^n where n is the number of qubits in the quantum register, we can perform in one operation on a quantum computer what would take an exponential number of operations on a classical computer. This is fantastic, but as the number of superposed states increases in the quantum register, the probability of measuring any particular one will decrease.

In a quantum search process, the inspection of search space need not be carried out picking one item at a with a classical computer e.g. superposition [31]. A major breakthrough happened with Lov Grover's very fast algorithm that is proven to be the fastest possible for searching through unstructured databases. The algorithm is so efficient that it requires roughly $\sqrt{N}$

(where *N* is the total number of elements in the search space) searches to find the desired item, as opposed to search in classical computing, which on average requires *N/2* searches [23], [32]. The quantum search algorithm offers only a quadratic speedup, as opposed to the more impressive exponential speedup offered by algorithms based on the quantum Fourier transform. However, the quantum search algorithm is still of great interest, since searching heuristics have a wider range of application than the problems solved using the quantum Fourier transform, and adaptations of the quantum search algorithm may have utility for a wide range of problems.

Grover's algorithm[31] has another very useful application, in the field of cracking encrypted data. Quantum computers can crack DES (Data Encryption Standard) - widely used system to protect data. An exhaustive search by conventional means would take a long time (even more than a year). But Grover's algorithm could find the DES enciphering key in just after few searches as compared to classical algorithm. Grover states that "A search engine could examine every nook and cranny of the Internet in half an hour, a "brute-force" decoder could unscramble the DES transmission in five minutes". These numbers are staggering compared to even the fastest classical algorithm's number of today.

4.1.2. Quantum bits

Quantum computation and quantum information are built upon an analogous concept, the quantum bit, o

the freedom to construct a general theory of quantum computation and quantum information, which does not depend upon a specific system for its realization.

Just as a classical bit has a state – either 0 or 1 – a qubit also has a state. Two possible states for a qubit are the states $|0\rangle$ and $|1\rangle$, which correspond to the states 0 and 1 for a classical bit. Notation like '$|\ \rangle$' is called the **Dirac Notation,** and it is a standard notation for states in quantum mechanics. The difference between bits and qubits is that a qubit can be in a state other than $|0\rangle$ or $|1\rangle$. It is also possible to form linear combination of states, often called superposition:

$$|\Psi\rangle=\alpha|0\rangle+\beta|1\rangle \qquad (1)$$

The numbers α and β are complex numbers, although for many purposes not much is lost by thinking of them as real numbers. Put another way, the state of a qubit is a vector in a two-dimensional complex vector space. The special states $|0\rangle$ and $|1\rangle$ are known as computational basis states, and form an orthonormal [36] basis for this vector space.

We can examine a bit to determine whether it is in the state 0 or 1. For example, computers do this all the time when they retrieve the contents of their memory. Rather remarkably, we cannot examine a qubit to determine its quantum state, that is, the values of α and β. Instead, quantum mechanics tells us that we can only acquire much more restricted information about the quantum state. When we measure a qubit we get either the result 0, with probability $|\alpha|^2$, or the result 1, with probability $|\beta|^2$. Naturally, $|\alpha|^2+|\beta|^2=1$, since the probabilities must sum to one. Geometrically, we can interpret this as the condition that the qubit's state be normalized to length 1. Thus, in general a qubit's state is a unit vector in a two-dimensional complex vector space.

The ability of a qubit to be in a superposition state runs counter to our 'common sense' understanding of the physical world around us. A classical bit is like a coin: either heads or tails up. For imperfect coins [35], there may be intermediate states like having it balanced on an edge, but those can be disregarded in the ideal case. By contrast, a qubit can exist in a continuum of states between $|0\rangle$ and $|1\rangle$ until it is observed. Let us emphasize again that when a qubit is measured, it only ever gives '0' or '1' as the measurement result probabilistically. For example, a qubit can be in the state

$$1/\sqrt{2}|0\rangle+1/\sqrt{2}|1\rangle, \tag{2}$$

which, when measured, gives the result 0 fifty percent ($|1/\sqrt{2}|^2$) of the time, and the result 1 fifty percent of the time. This state is sometimes denoted $|+\rangle$. Many of the operations on single qubit can be neatly described within the Bloch Sphere picture

4.1.3. Multiple qubits

we have two qubits. If these were two classical bits, then there would be four possible states, 00, 01, 10 and 11. Correspondingly, a two qubit system[35] has four computational basis states denoted $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$. A pair of qubits can also exist in superpositions of these four states, so the quantum state of two qubits involves associating a complex coefficient – sometimes called amplitude – with each computational basis state, such that the state vector describing the two qubits is

$$|\Psi\rangle=\alpha_{00}|00\rangle+\alpha_{01}|01\rangle+\alpha_{10}|10\rangle+\alpha_{11}|11\rangle. \tag{3}$$

Similar to the case for a single qubit, the measurement result x (=00,01,10 or 11) occurs with the probability $|\alpha_x|^2$, with the state of the qubits after the measurement being $|x\rangle$. The condition that $\Sigma_{x\varepsilon\{0,1\}^2} |\alpha_x|^2=1$, where the notation '$\{0,1\}^{2}$' means 'the set of strings of length [35][36] two with each letter being either zero or one'. For a two qubit system, we could measure just a subset of the qubits, say the first qubit, and we can probably guess how this works: measuring the first qubit alone gives 0 with probability $|\alpha_{00}|^2+|\alpha_{01}|^2$, leaving the post measurement state

$$|\Psi'\rangle=\alpha_{00}|00\rangle+\alpha_{01}|01\rangle/\sqrt{|\alpha_{00}|^2+|\alpha_{01}|^2}. \quad (4)$$

Note how the post measurement state is re-normalized by the factor $\sqrt{|\alpha_{00}|^2+|\alpha_{01}|^2}$ so that it still satisfies the normalization condition, just as we expect for a legitimate quantum state.

4.2. Quantum Gates

Changes occurring to a quantum state can be described using the language of quantum computation. Analogous to the way a classical computer is built from an electrical circuit containing wires and logic gates, a quantum computer is built from a quantum circuit containing wires and elementary quantum gates to carry around and manipulate the quantum information.

4.2.1. Single Qubit gates

Classical circuits consist of wires and logic gates. The wires are used to carry information around the circuit, while the logic gates perform manipulations of the information, converting it from one form to another, Consider, for example, classical single bit logic [29] gate. The only non-trivial gate of this family is the *NOT* gate,

whose operation is defined by its truth table, in which $0 \rightarrow 1$ and $1 \rightarrow 0$, that is, the states are interchanged.

Can an analogous quantum *NOT* gate for qubits be defined? Imagine that we had some process, which took the state $|0\rangle$ to the state $|1\rangle$, and vice versa. Such a process would obviously be a good candidate for a quantum analogous to the *NOT* gate. However, specifying the action of the gate on the states $|0\rangle$ and $|1\rangle$ does not tell us what happens to superposition [23][29] of the states $|0\rangle$ and $|1\rangle$, without further knowledge about the properties of quantum gates. In fact, the quantum NOT gate acts linearly, that is, it takes the state

$$\alpha|0\rangle+\beta|1\rangle \tag{5}$$

to the corresponding state in which the roles of $|0\rangle$ and $|1\rangle$ have been interchanged,

$$\alpha|1\rangle+\beta|0\rangle. \tag{6}$$

Why the quantum NOT gate acts linearly and not in some nonlinear fashion is a very interesting question. It turns out that this linear behavior is a general property of quantum mechanics.

There is a convenient way of representing the quantum *NOT* gate in matrix form, which follows directly from the linearity of quantum gates. Suppose we define a

$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Matrix *X* to represent the quantum *NOT* gate as follows:

(The notation *X* is used for quantum *NOT* gate.) If the quantum state $\alpha|0\rangle+\beta|1\rangle$ is Written in a vector notation as

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

With the top entry corresponding to the amplitude for $|0\rangle$ and the bottom entry the amplitude for $|1\rangle$, then the corresponding output from the quantum *NOT* gate is

$$X\begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

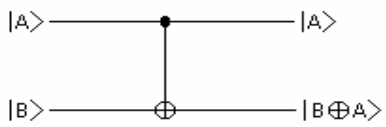
Notice that the action of the *NOT* gate is to take the state $|0\rangle$ and replace it by the state corresponding to the first column of the matrix *X*. Similarly, the state $|1\rangle$ is replaced by the state corresponding to the second column of the matrix *X*.

So quantum gates [26] on a single qubit can be described by two by two matrices. Are there any constraints on what matrices may be used as quantum gates? It turns out that there are. Normalization condition requires $|\alpha|^2+|\beta|^2=1$ for a quantum state $\alpha|0>+\beta|1>$. This must also be true of the quantum state $|\psi'> = \alpha'|0> + \beta'|1>$ after the gate has acted.

4.2.2. Multiple qubit gates

Now let us generalize from one to multiple qubits. Five notable multiple bit classical gates are *AND, OR, XOR* (exclusive-*OR*), *NAND* and *NOR* gates. An important theoretical result is that any function on bits can be computed from the composition of *NAND* gates[29][37][39] alone, which is thus known as a universal gate. By contrast, the *XOR* alone or even together with *NOT* is not universal. One way of seeing this is to note that applying an *XOR* gate does not change the total parity of the bits. As a result, any circuit involving only *NOT* and *XOR* gates will, if two inputs x and y have the same parity, give outputs with the same parity, resulting the class of functions which may be computed, and thus precluding universality.

The prototypical multi-qubit quantum logic gate is ***Controlled-NOT or CNOT*** gate. This gate has two input qubits, known as the control qubit and the target qubit, respectively. The circuit representation for the *CNOT* is shown in the figure. The top line represents the control qubit, while the bottom line represents the target qubit. The action of the gate may be described as follows:



controlled-NOT

$$U_{CN} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Fig 4.1 Controlled-NOT Gate and its Matrix Representation

If the control qubit is set to 0, then the target qubit is left alone. If the control qubit is set to 1, then the target qubit is flipped. In equations:

$$|00\rangle \rightarrow |00\rangle;\ |01\rangle \rightarrow |01\rangle;$$

$$|10\rangle \rightarrow |11\rangle;\ |11\rangle \rightarrow |10\rangle.$$

Another way of describing the *CNOT* is as a generalization of the classical *XOR* gate[29], since the action of the gate may be summarized as $|A, B\rangle \rightarrow |A, B \oplus A\rangle$, where $\oplus$ is addition modulo two, which is exactly what the XOR gate does.

Yet another way of describing the action of the *CNOT* is to give a matrix representation, as shown in the figure 2.4. We can easily verify that the first column of U_{CN} describes the transformation that occurs to $|00\rangle$, and similarly for the other computational basis states, $|01\rangle$, $|10\rangle$ and $|11\rangle$. As for the single qubit case, the requirement that probability be conserved is expressed in the fact that U_{CN} is a unitary matrix.

4.3. Power of Quantum Computing

4.3.1. Quantum Parallelism

Quantum parallelism is fundamental feature of many quantum algorithms. Quantum parallelism [25] allows quantum computers to evaluate a function *f (x)* for

many different values of x simultaneously. Here I will explain how quantum parallelism works, and some of its limitations.

Suppose $f(x)$: $\{0,1\}\rightarrow\{0,1\}$ is a function with a one bit domain and range. A convenient way of computing this function on a quantum computer is to consider a two qubit quantum computer which starts in the state $|x, y\rangle$. With an appropriate sequence of logic gates it is possible to transform this state into $|x, y \oplus f(x)\rangle$, where $\oplus$ indicates addition modulo *2*;[25][38] the first register is called the 'data' register, and the second register the 'target' register.

Consider the function shown in figure, which applies U_f to an input not in the computational basis. Instead, the data register is prepared in the superposition $(|0\rangle+|1\rangle)/\sqrt{2}$, which can be created with a Hadamard gate acting on $|0\rangle$. Then apply U_f resulting in the state:

$$(|0, f(0)\rangle+ |1, f(1)\rangle)/\sqrt{2} \qquad (7)$$

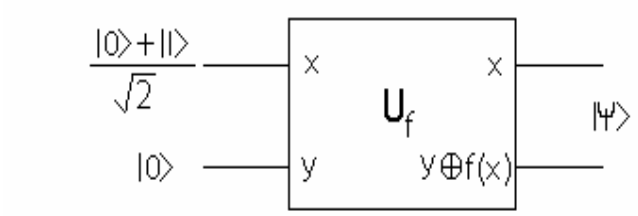


Fig 4.2 Quantum Circuit for evaluating f(0) and f(1) simultaneously

This is a remarkable state! The different terms contain information about both *f (0) and f (1)*; it is almost as if we have evaluated *f (x)* for two values of *x* simultaneously, a feature known as 'quantum parallelism'. Unlike classical parallelism, where multiple circuits each built to compute *f (x)* are executed

simultaneously, here a single *f(x)* circuit[29] is employed to evaluate the function for multiple values of *x* simultaneously, by exploiting the ability of a quantum computer to be in superpositions of different states.

Quantum parallel evaluation of a function with n bit input *x* and 1 bit output, *f(x),* can thus be performed in the following manner. Prepare the *n+1* qubit states $|0\rangle^{\oplus n}|0\rangle$, then apply the Hadamard transform to the first n qubits, followed by the quantum circuit implementing U_f. This produces the state

$$(1/\sqrt{2^n})\Sigma\ |x\rangle|f(x)\rangle, \qquad (8)$$

In some sense, quantum parallelism enables all possible values of the function *f* to be evaluated simultaneously, even though we apparently only evaluated f once. However, this parallelism is not immediately [29] useful. In our single qubit example, measurement of the states gives only either $|0, f(0)\rangle$ or $|1, f(1)\rangle$! Similarly, in the general case, measurement of the state $\Sigma_x|x, f(x)\rangle$ would give only *f(x)* for a single value of x. Of course a classical computer can do this easily! Quantum computations requires something more than just quantum parallelism to be useful; it requires the ability to extract information about more than one value of *f (x)* from superposition states like $\Sigma_x|x, f\rangle$.

4.3.2. Quantum Search

On a classical computer, if there are *N* possible routes, it obviously takes *O (N)* operations to determine the shortest route using this method. Remarkably, there is a quantum search algorithm, sometimes known as Grover's algorithm [27], which

enables this search method to be speed up substantially, requiring only $O(\sqrt{N})$ operations. Moreover, the quantum search algorithm is general in the sense that it can be applied far beyond the route-finding example just described to speed up many (though not all) classical algorithms that use search heuristics. Thus quantum search algorithms solves the following problem:

Given a search space of size N, and no prior knowledge about the structure of information in it, we want to find an element of that search space satisfying a known property. This problem requires approximately N operations, but the quantum search algorithm allows it to be solved using approximately $\sqrt{N}$ operations.

4.3.2.1. The Oracle

Suppose we wish to search through a search space of N elements. Rather than search the elements directly, we concentrate on the index to those elements, which is just a number in the range 0 to $N-1$. For convenience, assume $N = 2^n$, so the index can be stored in n bits, and that the search problem has exactly M solutions, with $1<=M<=N$. A particular instance of the search problem can conveniently be represented by a function f, which takes as input an integer x, in the range 0 *to* $N-1$. By definition, $f(x)=1$ if x is a solution to the search problem [33], and $f(x)=0$ if x is not a solution to the search problem. Suppose we are supplied with a quantum oracle – a black box whose internal workings are not important at this stage – with the ability to recognize solutions to the search problem. Making use of an oracle qubit signals this recognition. More precisely, the oracle is a unitary operator, O, defined by its action on the computational basis:

$$|x\rangle|q\rangle \rightarrow |x\rangle|q \oplus f(x)\rangle \qquad (9)$$

where $|x\rangle$ is an index register, $\oplus$ denotes addition modulo 2, and the oracle qubit $|q\rangle$ is a single qubit which is flipped if $f(x)=1$, and is unchanged otherwise. We can check whether x is a solution to our search problem by preparing $|x\rangle|0\rangle$, applying the oracle, and checking to see if the oracle qubit has been flipped to $|1\rangle$.

In the quantum search algorithm it is useful to apply the oracle with the oracle qubit initially in the state $(|0\rangle-|1\rangle)/\sqrt{2}$. If x is a solution to the search problem, applying the oracle to the state $|x\rangle (|0\rangle-|1\rangle)/\sqrt{2}$ do not change the state. On the other hand, if x is a solution to the search problem [27][33], then $|0\rangle$ and $|1\rangle$ are interchanged by the action of the oracle, giving a final state $-|x\rangle (|0\rangle-|1\rangle)/\sqrt{2}$. The action of the oracle is thus:

$$|x\rangle ((|0\rangle-|1\rangle)/\sqrt{2}) \rightarrow (-1)^{f(x)} |x\rangle (|0\rangle-|1\rangle)/\sqrt{2} \qquad (10)$$

Notice that the state of the oracle qubit is not changed. It turns out that this remains $(|0\rangle-|1\rangle)/\sqrt{2}$ throughout the quantum search algorithm, and can therefore be omitted. With this convention, the action of the oracle may be written:

$$|x\rangle \rightarrow |x\rangle(-1)^{f(x)} |x\rangle \qquad (11)$$

Oracle marks the solutions to the search problem, by shifting the phase of the solution. For an N item search problem with M solutions, it turns out that we need only apply the search oracle $O(\sqrt{N/M})$ times in order to obtain a solution, on a quantum computer.

Applying the quantum search algorithm with this oracle yields the smaller of the two prime factors with high probability. But to make the algorithm work, we need to construct an efficient circuit implementing the oracle. Let the function $f(x) = 1$ if x divides m, and $f(x) = 0$ otherwise. $f(x)$ tells us whether the trail division is successful or not. The circuit takes (x, q) – representing an input register initially set to q – to (x, $q \oplus f(x)$). Using this oracle and the quantum search algorithm we can search the range 2 to $m^{1/2}$ using $O(m^{1/4})$ oracle consultations. That is, we need only perform the trail division roughly $m^{1/4}$ times, instead of $m^{1/2}$ times, as with the classical algorithm. This illustrates the general way in which the quantum search algorithm can be applied.

4.3.2.2. The procedure

The algorithm proper makes use of a single n qubit register. The internal workings of the oracle, including the possibility of it needing extra work qubits, are not important to the description of the quantum search algorithm search proper. The goal of the algorithm is to find a solution to the search problem [34], using the smallest possible number of applications of the oracle.

The algorithm begins with the computer in the state $|0\rangle^{\otimes n}$. The Hadamard transform is used to put the computer in the equal superposition state,

$$|\psi\rangle=(1/N^{1/2})\Sigma\,|x\rangle \qquad (x=0 \text{ to } N-1) \qquad (12)$$

The quantum search algorithm then consists of repeated application of a quantum subroutine, known as the **Grover iteration or Grover operator** [27][33],

which is denoted by *G*. The Grover iteration, whose quantum circuit is illustrated in figure, may be broken up into four steps:

(1) Apply the oracle.

(2) Apply the Hadamard transform $H^{\otimes n}$.

(3) Perform a conditional phase shift on the computer.

(4) Apply the Hadamard transform $H^{\otimes n}$.

Each of the operation in the Grover iteration can be efficiently implemented on a quantum computer. The cost of the oracle call depends upon the specific application. We merely need note that the Grover iteration requires only a single oracle call. It is useful to note that the combined effect of steps 2,3 and 4 is

$$H^{\otimes n}(2|0\rangle\langle 0|-I)H^{\otimes n} = 2|\Psi\rangle\langle\Psi|-I \qquad (13)$$

where $|\Psi\rangle$ is the equally weighted superposition of states. Thus the Grover iteration, *G*, may be written as $G = (2|\Psi\rangle\langle\Psi| - I)\,O$. The operation $(2|\Psi\rangle\langle\Psi| - I)$ applied to a general state $\sum_k \alpha_k|k\rangle$ produces $\sum_k[-\alpha_k+2\langle\alpha\rangle]|k\rangle$,[33][34] where $\langle\alpha\rangle \equiv \sum_k \alpha_k/N$ is the mean value of α_k. $(2|\Psi\rangle\langle\Psi| - I)$ is referred to as the inversion about mean operation [1]. Grover iteration can be regarded as a rotation in the two-dimensional space spanned by the starting vector $|\Psi\rangle$ and the state consisting of a uniform superposition of solutions to the search space.

Chapter 5

Classical Computation on a Quantum Computer

5.1 Introduction

Can we simulate a classical logic circuit using a quantum circuit? Not surprisingly, the answer to this question turns out to be yes. It would be very surprising if this were not the case, as physicists believe that all aspects of the world around us, including classical logic circuits can ultimately be explained using quantum mechanics. The reason quantum circuits can not be directly used to simulate classical circuits is because unitary quantum logic gates are inherently reversible, whereas many classical logic gates such as the *NAND* gate [29] are inherently irreversible.

Any classical circuit can be replaced by an equivalent circuit containing only reversible elements, by making use of a reversible gate known as the **Toffoli gate.** The Toffoli gate has three input bits and three output bits.

Inputs			Outputs		
a	b	c	a′	b′	c′
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

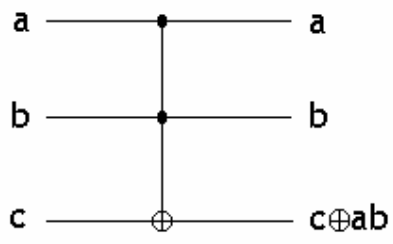


Table 5.1 Toffoli gate and its Circuit Representation

Two of the bits are control bits that are unaffected that is flipped if both control bits are set to 1, and otherwise is left alone. by the action of the Toffoli gate. The third bit is a target bit Note that applying the Toffoli gate twice to a set of bits has the effect (a, b, c)→ (a, b, c ⊕ a b)→(a, b, c), and thus the Toffoli gate is a reversible gate, since it has an inverse – itself.

The Toffoli gate can be used to simulate *NAND* gates [25][29], as shown. It is possible to simulate all other elements in a classical circuit, and thus an equivalent reversible circuit can simulate an arbitrary classical circuit.

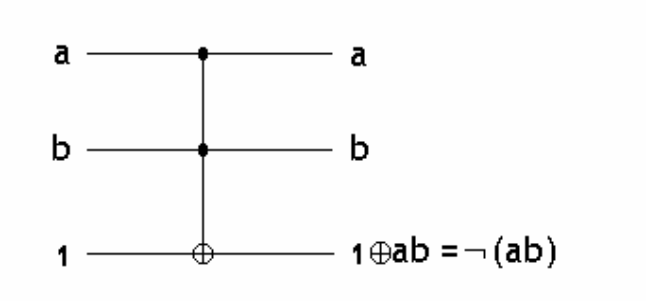


Fig 5.2 Circuit implementing a NAND gate using Toffoli gate

The Toffoli gate has been described as a classical gate, but it can also be implemented as a quantum logic gate. By definition, the quantum logic implementation of the Toffoli gate[25]simply permutes computational basis states in the same way as the classical Toffoli gate. For example, the quantum Toffoli gate acting on the state $|110\rangle$ flips the third qubit because the first two are set, resulting in the state $|111\rangle$. It is tedious but not difficult [43] to write this transformation out as an 8 by 8 matrix, *U*, and verify explicitly that *U* is a unitary matrix, and thus the Toffoli gate is a legitimate gate. The quantum Toffoli gate can be used to simulate irreversible classical logic gates, just as the classical Toffoli gate was, and ensures that quantum computers are capable of performing any computation, which a classical (deterministic) computer may do. What if the classical computer [37] is non-deterministic, that is, has the ability to generate random bits to be used in the computations? Not surprisingly, it is easy for a quantum computer to simulate this. To perform such a simulation it turns out to be sufficient to produce random fair coin tosses. Of course, if the ability to simulate classical computers were the only feature of quantum computers there would be little point in going to all the trouble of

exploiting quantum effects! The advantage of quantum computing is that much more powerful functions may be computed using qubits and quantum gates.

5.2. Boolean algebra for quantum Computing

In building quantum circuits [19] for Boolean functions, auxiliary qubit will be used which we initially set to zero, to hold the result of the Boolean function, together with CNOT [16] based transformations. CNOT is a gate where the target qubit *t* is controlled by a set of qubits *C* such that $t \notin C$. *T*he state of the qubit *t* will be flipped from $|0\rangle$ to $|1\rangle$ or from $|1\rangle$ to $|0\rangle$ if and only if the conditions stated by the CNOT gate is evaluated to true. The condition that a certain qubit evaluates to true depends on whether the state of the qubit is $|0\rangle$ or $|1\rangle$ according to the condition being set. Consider the Boolean function $F(X_1, X_2, X_3) = X_1 + X_2X_3$, for quantum computing purposes, the representation will be as shown in Table.

X_1	X_2	X_3	F_{ini}	X_1	X_2	X_3	F_{fin}
0	0	0	0	0	0	0	1
0	0	1	0	0	0	1	1
0	1	0	0	0	1	0	1

0	1	1	0	0	1	1	1
1	0	0	0	1	0	0	0
1	0	1	0	1	0	1	0
1	1	0	0	1	1	0	0
1	1	1	0	1	1	1	1

Table 5.2 Quantum computing version of the truth table for $F(X_1, X_2, X_3) = X_1 + X_2X_3$

From the above representation, we can see that the F_{ini} will be flipped only if the result of the function F is 1, $F(X_1, X_2, X_3) = 1$.

5.2.1. Converting Boolean function in Quantum Circuits

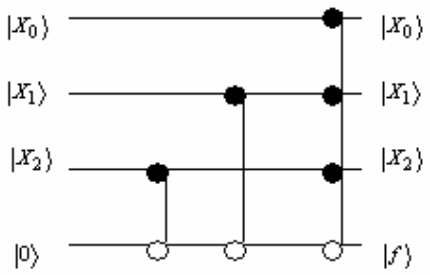
The Boolean logic in quantum circuits is represented using Reed-Muller logic [26]. The operation used in Reed-Muller logic *RM* are {*AND, XOR, NOT*}. RM logic is used to represent any arbitrary Boolean expression (*SAT or K-SAT*). Consider the following Boolean expression:

$$f(X_0, X_1, X_2) = \overline{X_0} + X_1 X_2 \qquad (14)$$

Any Boolean function represented in RM can be implemented as a reversible Boolean quantum circuit, so RM expression [19][26] for the above Boolean expression takes following form:

$$F = X_0 X_1 X_2 \oplus X_0 \oplus 1 \qquad (15)$$

The Boolean Quantum Circuit that represents this expression for Positive Polarity RM is as follows



$$f = \overline{X_0} + X_1 X_2 = X_0 X_1 X_2 \oplus X_0 \oplus 1$$

Fig 5.3 Boolean Quantum Circuit for Boolean Expression

To implement the oracle we will use this technique in the proposed algorithm to evaluate the Boolean expression. Multi valued Boolean function *f* [42] can be calculated through oracle calls. If *f* is one to one and the size of its domain and range is the same, then problem can be formulated as follows:

Given an oracle $f(a,x):\{0,1\}^n \times \{0.1\}^n \rightarrow \{0,1\}$ and a fixed value a_o we wish to find the value of a_O by querying the oracle $f(a_o, x)$. The oracle that is used in this approach is EQ oracle defined as $f(a, x) = 1$ if $x=a$. the query complexity for the EQ oracle is $\Theta\sqrt{N}$ $(N=2^n)$.

5.3. Configuration Generation Problem into a Satisfiability problem

For the FPGA with n-input LUTs, log2n test configurations with single-term functions are sufficient to cover all bridging faults with respect to the fault list. The activating inputs are columns of binary numbers using log2n bits for n-input LUTs (so called Walsh codes). Consider the binary representation of mth input of the LUT using log2n bits. Each bit position corresponds to one of the log2n [26][30][42] test

configurations, as follows. If the i^{th} bit position is 1 for the m^{th} input it means that the mth bit of the activating input is 1 in the ith configuration, otherwise 0. The configuration of the LUT is determined by its activating input and the value of the output for that input The corresponding bit of the activating input is 1 in the first configuration and 0 in the second configuration.

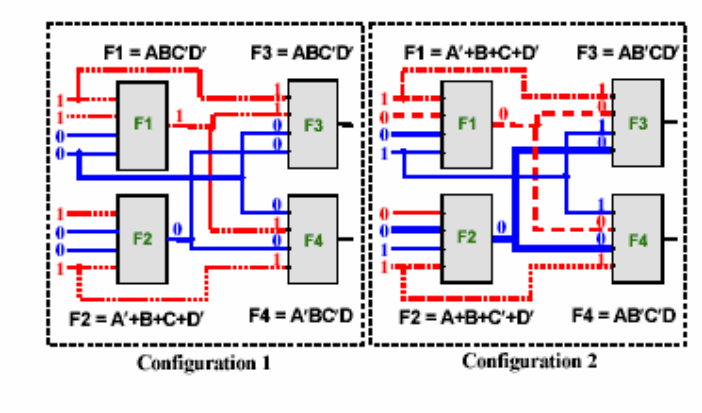


Fig 5.4 Two Configurations for a 4-input LUT

We present a technique to convert the configuration generation problem into a satisfiability problem by constructing a Boolean function *F*. A solution to *F=1* can be converted to the desired test vectors and configurations.

Consider the symmetric logic function $S_k^n(x1, x2, x3....., xn)$, where *k = 1* if and only if exactly *k* inputs of S_k^n are 1.

For example, $S_1^3 = \chi 1\chi 2' x3' + \chi 1' \chi 2\chi 3' + \chi 1' \chi 2' \chi 3$.

S_k^n can be recursively defined as fallows

$$S_k^n(x_1,\ldots,x_n)=\begin{cases}\prod_{i=1}^{n} x_i \ , k=n \\ \prod_{i=1}^{n} \overline{x_i} \ , k=0 \\ x_1 \cdot S_{k-1}^{n-1}(x_2,\ldots,x_n)+\overline{x_1}\cdot S_k^{n-1}(x_2,\ldots,x_n) \ , 0<k<n\end{cases}$$

(Π is the logical AND function):

Consider the set of the nets in the design that must be dichotomized. Let M be the number of LUT inputs that participate in this grouping algorithm. M = $n/2^{i-1}$ at ith configuration for N-input LUTs. We assign [42][43] one variable per each net. All the fanout stem and branches of the same net share the same variable. For each LUT L, let $x_1^l, x_2^l \ldots, x_M^l$ be the M inputs of L, which have to be dichotomized. Let

$$F_l = S_{M/2}^M(x_1^l, x_{2\ldots\ldots}^l x_M^l) \qquad (16)$$

For the solution for $F_l = 1$ is an equal-size dichotomization [42] of those M inputs of L, a sub-solution to the problem for only one LUT.

The function that is the SAT formulization for the entire FPGA is:

$$F = \prod_{For each LUT l} F_l$$

In the variable assignment that satisfies F, every variable whose with logic value 0 is put in the minus group, otherwise is assigned to the plus group. All the nets are assigned with unique variables. For the first configuration, S_2^4 [42] must be used for all LUTs, since M = 4. The SAT function for the first configuration is as follows:

$$F = S_2^4(x_1, x_2, x_3, x_4).S_2^4(x_5, x_6, x_7, x_8)S_2^4(x_7, x_9, x_4, x_{10})S_2^4(x_4, x_9, x_{10}, x_8)$$

where:

$$S_2^4(x_1, x_2, x_3, x_4) = \begin{bmatrix} (x'_1 + x'_2 + x'_3)(x'_1 + x'_2 + x'_4)(x'_1 + x'_3 + x'_4)(x'_2 + x'_3 + x'_4) \\ (x_1 + x_2 + x_3)(x_1 + x_2 + x_4)(x_1 + x_3 + x_4)(x_2 + x_3 + x_{4)} \end{bmatrix}$$

An example design implemented with four LUTs One variable assignment that satisfies F is: <x1, x2, x3, x4, x5, x6, x7, x8, x9, x10>=<0,0,1,1, 0,1,1,0,0,1>. It implies that

plus group = {x3,x4,x6,x7,x10} and minus group = {x1,x2,x5,x8,x9}. For the second configuration, each of the above groups must be partitioned [43]. Hence, there are two SAT functions on two disjoint sets of different variables. These functions are as follows:

$$F_1 = S_1^2(x_1, x_2).S_1^2(x_5, x_8).S_1^2(x_1, x_9)S_1^2(x_8, x_9)$$
$$F_2 = S_1^2(x_3, x_4).S_1^2(x_6, x_7).S_1^2(x_4, x_{10})S_1^2(x_7, x_{10})$$

Where $S_1^2(x_1, x_2) = (x'_1 + x'_2)(x_1 + x_2)$

One variable assignment that satisfies F1 is: <x1, x2, x5, x8, x9> = <1,0,0,1,0>. Therefore the groups are {x2, x5, x9} and {x1, x8}. Similarly, [42] a variable assignment that satisfies F2 is: <x3, x4, x6, x7, x10> = <0,1,0,1,0> and the groups are {x3, x6, x10} and {x4, x7}. The results interpreted in terms of the activating inputs and LUT configurations for two test configurations.

5.3.1. Problem with the Satisfiability

Satisfiability problem (SAT) is one of the famous NP-complete problems of special interest in theory of computation, artificial intelligence and the study of mathematical logic. It can be understood as follows: We have an n variable Boolean expression and we need to know if there are any possible variable assignments within the 2n possible variable assignments that will make the expression evaluate to TRUE. An important variation of the above problem is the *K-SAT* problem [26] where the Boolean expression consists of the *AND*-ing of a number of clauses, each clause consists of the *OR*-ing of *K* Boolean variables, and each variable may be negated. There are two well-developed paradigms of Boolean logic. The first uses the operations of *AND, OR* and *NOT* and called canonical Boolean logic. The second uses the operations *AND, XOR* and *NOT* and is called Reed-Muller logic (RM) [19]. This means that any Boolean expression that may be given in a *SAT* or *K-SAT* problem can be represented *RM* form. In [20], we showed that any Boolean function represented in *RM* could be implemented as a reversible.

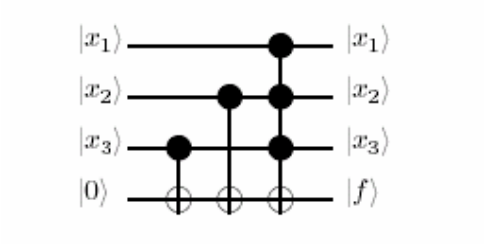


Fig 5.4 Boolean quantum circuit for the Boolean expression

Boolean quantum circuit for the Boolean expression shown in Eqn.(18) and Eqn.(19). Boolean quantum circuit, we will use this technique to implement the oracle

used in the proposed algorithm to evaluate the Boolean expression[19][30] under inspection. For example, consider the following 3-SAT Boolean expression:

$$(x_1+x_2+x_3)(x'_1+x'_2+x'_3)(x_1+x'_2+x'_3) \quad (17)$$

Where + indicates the inclusive-*OR*, product of two Boolean variables indicates *AND* operation and xi in the negation of xi, its positive polarity *RM* expression is as follows:

$$(\chi_1 \cdot \chi_2 \cdot \chi_3 \otimes \chi_2 \otimes \chi_3) \quad (18)$$

Where + indicates the exclusive-*OR* operation, according to the Boolean quantum circuit that represents this expression. For the above expression [42] shown in Eqn.(18) or Eqn.(19), we can show that there are 5 out of 8 possible variable assignments will make this expression evaluate to TRUE as follows:

$$\{x1, x2, x3\} = \{001, 010, 101, 110, 111\} \quad (19)$$

Chapter 6

The Proposed Method

6.1. Introduction

In previous chapter, configuration of logic blocks in the design is changed and test vector and configuration generation problem is converted to a satisfiability (SAT) problem, and state of the art SAT solvers are exploited for test configuration generation.

In the proposed method, we minimize the problem involves searching over a vector search space. I have viewed this problem of generating test configuration and test vector. We generate the test vector that satisfies the property that energy function for that particular vector should be equal to 1. I am applying the quantum computing algorithm to find the test vector and test configuration, quantum employ parallelism and superposition and by using these quantum concepts, we can find the test vector at a speed much faster than is possible on a classical computer.

A solution varies according to the number of possible desired variable assignments exists for the given Boolean expression. The algorithm has been tested for different number of variables in the Boolean expression and for all possible Boolean expressions (cases) for a given number of variables.

The Satisfiability problem is a typical NP-complete problem gaining a lot of attention. we will present a quantum algorithm for solving the general SAT problem which runs in a constant number of steps; O(4), with any given number n of Boolean variables in the expression.

6.2. Algorithm

1.**Register Preparation**: For an n variable Boolean expression, prepare a quantum register of n+1 qubits all in state |0 >i, where the extra qubit will be used as a workspace or evaluating the Boolean function.

$$W_0 |0....0\rangle = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} |i\rangle \tag{20}$$

2. Register initialization: Apply Hadamard gates on the first n qubits so they contain the 2n possible variable assignments of the Boolean expression, where i is the integer representation of the input Boolean configuration to the Boolean function f . Prepare the system to equal superposition of all possible states i.e. giving amplitude of $1/\sqrt{2^n}$ to each test vector of search space.

$$W_1 |\underset{n}{0..0.0}>= \left(\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle \right) \otimes |0\rangle; N = 2^n \tag{21}$$

3. Boolean Function Evaluation: Apply the oracle U_f, for the evaluation of the Boolean function *f* simultaneously for all possible variable assignments and put the result on the extra workspace qubit. .

$$|W_2\rangle = U_f |W_1\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} \left(|i\rangle \otimes |0 \otimes f(i)\rangle \right) = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} \left(|i\rangle \otimes |f(i)\rangle \right) \tag{22}$$

4. Completing Superposition and Changing Sign: Apply Hadamard gate for completing superposition. Invert the amplitudes of the vector if |q> flips in step 3, such that from a_j to $-a_j$ for all such that P (x_i)=1.

$$\left| W_3 \right\rangle = \left(I^{\otimes n} \otimes H \right) \left| W_2 \right\rangle = \frac{1}{\sqrt{P}} \sum_{i=0}^{N-1} \left(\left| i \right\rangle \otimes \left(\left| 0 \right\rangle + (-1)^{f(i)} \left| 1 \right\rangle \right) \right); P = 2N = 2^{n+1} \quad (23)$$

5. Inversion About the Mean: Diffusion Operator D is applied on the n + 1 qubits to increase the amplitudes of x_i with $P(x_i)=1$ negative signs and decrease the probabilities of the amplitudes with positive signs.

$$D = H^{\otimes n+1} \left(2 \left| 0 \right\rangle \left\langle 0 \right| - I \right) H^{\otimes n+1} = 2 \left| \Psi \right\rangle \left\langle \Psi \right| - I \quad (24)$$

6. Repeat steps 2 through 4 $\frac{\pi}{4}\sqrt{2^n}$ times.

7. Measurement: Measure the last qubit of the quantum state; representing *P (x)* because of the amplitude change there is a high probability that the result will be 1. If this is the case, the measurement has projected the state onto the subspace $1/\sqrt{2^k}\,\Sigma \left| x_i ,1 \right|$ where *k* is the number of solutions and the result will be a test vector.

6.3. Explanation

The first *n* qubits of the algorithm corresponds to the *n* binary variables that are used to represent clauses for finding the test configuration and test vector in Application-dependent testing of FPGA Interconnects. There is a way to unitarily generate a superposition of all possible values of an *n* qubit register. That is, if the initial state of the register is all zero, then transforms it in a superposition of all 2*n*

values of the first *n* qubits. For an n variable Boolean expression, prepare a quantum register of n+1 qubits all in state |0 >i, where the extra qubit will be used as a workspace or evaluating the Boolean function.

Apply the oracle U_f constructed according to [19, 20], which will evaluate the Boolean function f simultaneously for all possible variable assignments and put the result on the extra workspace qubit. we are taking all the *n+1* qubits in |0> state thereafter in step 2 qubits are prepared in the superposition state [8] of the values 0 and 1.This is achieved by applying the *H* gate to each qubit. As discussed earlier, the result of applying Hadamard gate on state |0> leads to the state $\left(|0\rangle + |1\rangle\right)/\sqrt{2}$, which is a state midway between |0> and |0>. After step two, all qubits will have same probability amplitude of $1/\sqrt{2^n}$ shown in Equation (4). In step 3 we will evaluate the Boolean function *f* simultaneously for all possible variable assignments using oracle U_f Equation (5). This is so because we have prepared the *n* qubit in superposition and we are applying the oracle with these superposed values.

Apply Hadamard gate for completing superposition. Invert the amplitudes of the vector if |q> flips in step 3, such that from a_j to $-a_j$ for all such that $P(x_i)=1$. The remaining one qubit [7] is the oracle qubit that will flipped if the value of the Boolean function comes out to be zero for a particular oracle call. In the algorithm, we are considering a single n qubit register, a single one-qubit register. The n qubit register will represent the values of the following eq.

$$|\Psi\rangle = 1/\sqrt{N}\left(\Sigma|x\rangle\right) \text{ , x=0 to N-1} \qquad (25)$$

Here we are taking the advantage of quantum parallelism, which is not possible on classical computer. In step 4 Hadamard transformation is applied having

Equations (6,7,8). The loop represented by step 6 is the central node of the algorithm. Each iteration of the loop will increase the probability amplitude of desired state. As a result after $\sqrt{N/M}$ where ($N= 2^{n}$) repetition of the loop, the probability amplitude of the desired state will be higher than the probability amplitudes of the other states.

Diffusion Operator D is applied on the $n + 1$ qubits to increase the amplitudes of x_i with $P(x_i)=1$ negative signs and decrease the probabilities of the amplitudes with positive signs. If the value for Boolean function comes to be zero, then the application of oracle will flip the second qubit. In that case we will invert the probability amplitude of the state using inversion about mean. If the value of the Boolean function is not zero then second qubit will not be flipped and we will apply the same transformation to that state but without inverting the probability amplitude of that state .The result is that it will increase the probability amplitude of the desired routing solution.

Measure the first n qubits, we will get the desired solution with probability given below. Firstly Probability P_s to find a solution out of the M possible solutions. The function is such that $f(x)=1$ if x is the solution to the Boolean function and $f(x)=0$ if x is not the solution of to the vector function. The algorithm makes use of a single qubit register where n is the number of the Boolean variables assigned to each clauses of the Application-dependent testing in FPGA Interconnect problem. The goal of the algorithm is to find a solution to the problem, using the smallest possible number of application of the oracle. The algorithm begins with the computer in the state $\left|0\right\rangle^{\otimes n}$. This is the state of equal superposition. We can use hadamard transformation put the computer in equal superposition state.

$$|\Psi\rangle = {}^{1}\!/_{\sqrt{N}}\left(\Sigma|x\rangle\right) \quad , x=0 \text{ to } N\text{-}1. \qquad (26)$$

After putting the computer in equal superposition we will repeatedly apply proposed algorithm to the system $\sqrt{N/M}$ times where $N= 2^n$ and M is the total number of clauses The overall result of these iterations is to move the state of the system to the required state .In each iteration, we are consulting oracle, which moves the state from $|X,Y\rangle$ to $|x, y \oplus f(x)\rangle$. This way we measure the system and hence the result of the measurement is the generation of the test vector. This is the very important step that takes the advantages of quantum parallelism .If we prepare y with 0 then this second qubit will flip if $f(x)=1$, otherwise it will remain 0. As we have prepared the x in superposition by applying the Hadamard transform in first step so the oracle consultation will occur for multiple values of x in single iteration and this is one reason behind less number of iteration by quantum algorithms as compared to classical algorithms.

6.4. Experimental Results

Taking the different sets of circuits for different values of n for the proposed Quantum Satisfiability has performed the experiment. A theoretical comparison has been made between Exhaustive Search and Quantum Computing. in Table 6.1. In exhaustive search, we have to go through each possible combination to find the test vector. We specify a search space of size 2^n and total number of iterations taken to find the test vector. Quantum search algorithm on the other hand has always taken less

iteration. Table 6.1 indicating that Exhaustive search taking *N/2* iterations compared to quantum search having $\sqrt{N}$ iteration*s*.

n	*k*	ES	QC
8	12	242	27
10	19	985	53
12	22	3748	137
14	24	14672	257
16	29	57631	393
18	33	217291	617
20	36	901501	913
22	40	2995719	1295
24	43	12914026	1797
26	45	55717931	3663
28	50	103749193	6951

n –input variables (2^n search space size), k – no. of clauses, ES- Exhaustive Search Simulation Annealing, Genetic Algorithm and Quantum computing

Table 6.1: Comparison and Experimental Results of Exhaustive search and Quantum Computing

The exhaustive search for the same requires searching for a test vector in complete *n*-Dimensional *(0, 1)* state space, which comes out to be $O(2^n)$ and it is clear from the algorithm that the total complexity of our new algorithm comes out to be $O(\sqrt{N/m})$ for some Boolean formula with *n* variables and *k* clauses, which is too much

improved over classical methods used for test case generation. From experimental results, it can be concluded that quantum search algorithm is more effective, especially for large values of n as it always take less than $\sqrt{N}$ iterations to find the required test vector.

Conclusion

The Quantum Computing based algorithm used is very flexible and efficient, especially for the problems, which are NP- complete. In this thesis, NP-complete Boolean satisfiability problem could be solved with a constant complexity algorithm on quantum computers, where we use quantum parallelism to load all possible variable assignments and then evaluate the function simultaneously over all possible variable assignments. The above results show that Quantum algorithm is able to find the test vector in only $O(\sqrt{N/m})$ accesses to the vector search space, having a Boolean formula with n variables and k clauses. We use quantum parallelism to load all possible variable assignments and then evaluate the function simultaneously over all possible variable assignments. We apply two more operations to increase the probabilities of finding a solution. In this paper, we will present a quantum algorithm for solving the general SAT problem which runs in a constant number of steps of Boolean variables in the expression.. This improved efficiency is achieved due to the ability of Quantum computing to evaluate all possible combinations simultaneously. One other advantage of proposed method is that it is able to find more than one test patterns simultaneously, and the Quantum computing representing all the test patterns that can detect a particular fault. In the future, we will refine the implementation of the algorithm on Quantum, and perform various evaluations on implemented FPGA.

References

[1] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic. Field-Programmable Gate Arrays. Kluwer Academic Publishers,1992.

[2] Xilinx, Inc., “The Programmable Gate Array Data Book”, 1992.

[3] Concurrent Logic, Inc., “CFA6006 Field Programmable Gate Array”, March 1991.

[4] M. Renovell, J. M. Portal, J. Figueras, and Y. Zorian. Testing the interconnect of RAM-based FPGAs. IEEE Design & Test of Computers, 15(1):45–50, January-March 1998.

[5] C. Stroud, S. Wijesuriya, C. Hamilton, and M. Abramovici. Built-in self-test of FPGA interconnect. In International Test Conference, pages 404–411, October 1998.

[6] S.-J. Wang, and C.-N. Huang. Testing and diagnosis of interconnect structures in FPGAs. Proceedings of the Seventh Asian Test Symposium, 1998. ATS '98. Page(s): 283–287.

[7] X. Sun, J. Xu, B. Chan, and P. Trouborst. Novel technique for built-in self-test of FPGA interconnects. Proceedings of the International Test Conference, 2000. Page(s): 795 –803 ,1998.

[8] H. Michinishi, T. Yokohira, T. Okamoto, T. Inoue, and H. Fujiwara. A test methodology for interconnect structures of LUT-based FPGAs. Proceedings of the Fifth Asian Test Symposium, 1996. Page(s): 68 –74

[9] I.G. Harris, and R. Tessier. Interconnect testing in cluster-based FPGA architectures. Proceedings Design automation Conference, 2000. Page(s): 49 –54

[10] M. B. Tahoori, Application-Dependent Testing of FPGA Interconnects, CRC Technical Report 02-4, Stanford University,2002.

[11] Mehdi Baradaran Tahoori,“Interconnect Testing of FPGA”, March 2002.

[12] Huang, W.K., X.T. Chen, and F. Lombardi, "On the Diagnosis of Programmable Interconnect Systems: Theory and Application," IEEE VLSI Test Symp., pp. 204-209, 1996.

[13] A. Krasniewski, "Application-dependent testing of FPGA delay faults," in EUROMICRO'99, 1999, pp. 260–267.

[14] M. Renovell, J. M. Portal, J. Figueras, and Y. Zorian, "SRAM-based FPGAs: Testing the LUT/RAM modules," in International Test Conference, October 1998, pp. 1102–1111.

[15] D. Das, N. A. Touba, A Low Cost Approach for Detecting, Locating, and Avoiding Interconnect Faults in FPGA-Based Reconfigurable Systems, Proc. Int'l Conf. On VLSI Design, 1999.

[16] T. Kirkland, M. R. Mercer. A Topological Search Algorithm for ATPG. Proc. ACM/IEEE 24th Design Automation Conf.,p. 502-508. June, 1987

[17] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. ACM Computing Surveys, Vol. 24, Nr. 3, 1992, p. 293-318.

[18] R.E. Bryant. A methodology for hardware verification based on logic simulation. Technical Report CMU-CS-87-128, Computer Science Department, Carnegie Mellon University, Pittsburgh, June 1987.
University, October, 2001

[19] L. Zhang, C. Madigan, M. Moskewicz, S. Malik, Efficient Conflict Driven Learning in a Boolean Satisfiability Solver, Proc. ICCAD, 2001.

[20] F.J. Ferguson, J.P. Shen, A CMOS fault extractor for inductive fault analysis, IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, Vol. 7 Issue: 11 , pp. 1181-1194, Nov. 1988.

[21] P. Dirac (1947), The Principles of Quantum Mechanics. Clarendon Press, Oxford, United Kingdom.

[22] M.R. Garey and D. S. Johnson (1979), Computers and Intractability: A Guide to the Theory of NP-completeness. W.H. Freeman, San Francisco.

[23] R.P. Feynman (1986), Quantum Mechanical Computers. Foundations of Physics, 16, pp. 507-531.

[24] E. Bernstein and U. Vazirani (1993), Quantum Complexity Theory. In Proceedings of the 25th Annual ACM Symposium on Theory of Computing,pp. 11-20.

[25] D. R. Simon (1994), On the Power of Quantum Computation. In Proceedings of the 35th Annual symposium on Foundations of Computer Science, pp. 116-123.

[26] A. Younes, and J. Miller(2003), Representation of Boolean Quantum Circuits as Reed-Muller Expansions. Los Alamos Physics Preprint Archive.

[27] Review Goong Chen, Stephen A. Fulling, and Marlan O. Scully, "Grover's Algorithm for Multiobject search in Quantum Computing".

[28] V.D. Agrawal, K.-T. Cheng, P. Agrawal,"CONTEST: A Concurrent Test Generator for Sequential Circuits," Proc. 25th Design Automation Conference,1988,pp. 84-89.

[29] A. Barenco, C. Bennett, R. Cleve, D. P. Divincenzo, N. Margolus, P.Shor, T. Sleator, J. Smolin and H. Weinfurter (1995), Elementary Gates for Quantum Computation. Physical Review A, 52(5), pp. 3457-3467.

[30] T. Larrabee, 1992 "Test pattern generation using Boolean Satisfiability," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp.4-15,

[31] Michael A. Nielsen & Isaac L. Chuang, "Quantum Computation and Quantum Information", Cambridge University Press.

[32] L. K. Grover, "A fast Quantum Mechanical Algorithm for estimating the median", lanl e-print quant-ph/9607024.

[33] A. Patel, "Quantum Database Search can do without Sorting", quant-ph/0012149 v3 8 Jun 2001.

[34] Lov K. Grover, "A Fast Quantum Mechanical Algorithm for Database Search". 3C-404A Bell Labs, 600 Mountain Avenue, Murray Hill NJ 07974.

[35] Dong Pyo Chi and Jinsoo Kim, "Quantum Computation", Department of Mathematics, Seoul National University.

[36] R. Cleve, A. Ekert, C.Macchiavello and M. Mosca, “Quantum Algorithms Revisited”, Clarendon Laboratory, Department of Physics, University of Oxford, Parks Road, Oxford, OX1 3PU, U.K.

[37] T. Hogg (1999), Solving Highly Constrained Search problems with Quantum Computers. Journal of Artificial Intelligence Research, 10, pp. 39-66.

[38] Ajit Narayanan, “Quantum Algorithms”, Department of Computer Science, Old Library, University of Exeter, Exeter EX4 4PT, UK.

[39] M. Boyer, G. Brassard, P. Hoyer and A. Tapp (1996), Tight Bounds on Quantum Searching. In Proceedings of the 4th Workshop on Physics and Computation, pp. 36-43.

[40] Dong Pyo Chi and Jinsoo Kim, “Quantum Database search by a single query”,Department of Mathematics, Seoul National University.

[41] O. Angelsmark, V. Dahllof, and P. Jansson (2002), Finite Domain Constraint Satisfaction Using Quantum Computation. In Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science, pp. 93-103.

[42] A. Younes, J. Miller (2003), Automated Method for Building CNOT Based Quantum Circuits for Boolean Functions. Los Alamos Physics Preprint Archive, quant-ph/0304099.

[43] K. Iwama, Y. Kambayashi and S. Yamashita (2002), Transformation Rules for Designing CNOT–based Quantum Circuits. In Proceedings of the 39th Conference on Design Automation, ACM Press, pp. 419-424.

[44] N. J. Cerf, L.K. Grover, and C. P.Williams (1998), Nested Quantum Search and NP-complete Problems. Los Alamos Physics Preprint Archive, quant.

[45] A. Singh, S. Singh, "Applying Quantum Search to Automated Test Pattern Generation for VLSI Circuits", International Journal of Quantum Information, 2003