

Bucknell University Bucknell Digital Commons

Master's Theses

Student Theses

2010

FPGA Based Design for Accelerated Fault-testing of Integrated Circuits

Joe Dunbar
Bucknell University

Follow this and additional works at: https://digitalcommons.bucknell.edu/masters_theses

Recommended Citation

Dunbar, Joe, "FPGA Based Design for Accelerated Fault-testing of Integrated Circuits" (2010). *Master's Theses*. 28.
https://digitalcommons.bucknell.edu/masters_theses/28

This Masters Thesis is brought to you for free and open access by the Student Theses at Bucknell Digital Commons. It has been accepted for inclusion in Master's Theses by an authorized administrator of Bucknell Digital Commons. For more information, please contact dcadmin@bucknell.edu.

FPGA BASED DESIGN FOR ACCELERATED FAULT-TESTING OF INTEGRATED CIRCUITS

by

Carson Dunbar

A Thesis

Presented to the Faculty of
Bucknell University
In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Approved:



Advisor



Department Chairperson



Engineering Thesis Committee Member



Engineering Thesis Committee Member

04/2010
(Date: Month and Year)

I, Carson Dunbar, do grant permission for my thesis to be copied.

Dedication:

This work is dedicated to Carson and Carol Dunbar for their constant support in my studies and their constant encouragement with any of my endeavors.

I would also like to thank Professor Kundan Nepal for his advice and large contribution to this work with both experience and revisions.

Contents:

Dedication.....	ii
Table of Contents.....	iii
List of Tables.....	iv
List of Figures.....	v
Abstract.....	ix
Chapter 1: Introduction.....	1
Chapter 2: Background.....	6
Chapter 3: Xilinx Virtex II Architecture.....	18
Chapter 4: Error Insertion.....	30
Chapter 5: Algorithm For Testing.....	41
Chapter 6: Results.....	59
Chapter 7: Conclusions and Future Work.....	77
Bibliography.....	80
Appendices.....	84
Appendix 1: C++ Code for fault induction.....	84

Appendix 2: C++ code for creating all other verilog files.....	92
Appendix 3: Variable Instantiation Data.....	108
Appendix 4: Variable Input Test Vector Data.....	118

List of Tables:

Table 2-1. Typical defects in a standard PCB.....	7
Table 4-1: Example Directed Graph Table3.....	36
Table 4-2: C17 Directed Graph Output.....	38
Table 5-1: Example Output for Step 3-f-iii.....	54
Table 6-1: Results from behavioral simulation and hardware emulation for C432 benchmark circuit.....	60
Table 6-2: Area and timing results for C432 circuit simulated with 10000 vectors and variable instantiations running at clock frequency of 25MHz. 537 faults were detected with 36 input vectors.....	62
Table 6-3: Results from C432, static number of instantiations, variable vector tests.....	65
Table 6-4. Description of Benchmark circuits.....	72
Table 6-5: Best fault detection based on faults found, vectors used, and time (done with 32 iterations).....	74
Table 6-6: Comparison of ISCAS and MCNC circuit input test pattern counts using various tests.....	76

List of Figures:

Figure 2-1. Single Stuck-At Fault at an internal node of a circuit.....	8
Figure 2-2. Initialization Fault.....	9
Figure 2-3. Multiple Stuck-At-Faults.....	10
Figure 2-4. Bridging Fault.....	11
Figure 2-5. Stuck Open Fault.....	12
Figure 2-6. Stuck short Fault.....	12
Figure 3-1. FPGA General Architecture	18
Figure 3-2. Virtex II Pro slice configuration.....	19
Figure 3-3. Physical Layout of FPGA (XC2VP30)	20
Figure 3-4. Layout of C432 design on the XC2VP30.....	21
Figure 3-5. Block diagram of the Xilinx Development Board.....	22
Figure 3-6. Functional blocks of the PowerPC 405 processor.....	24
Figure 3-7. Interconnects in the Virtex II Pro.....	25
Figure 3-8. RS232 implementation.....	26
Figure 3-9. FIFO layout using BRAM.....	27

Figure 3-10. Clock Generator block diagram.....	29
Figure 4-1. Fault Equivalence in series of logic gates, wires, and fan outs.....	31
Figure 4-2. Collapsed fault set using dominance.....	32
Figure 4-3: a. Standard fault locations. B. A circuit with the checkpoints marked (X).....	34
Figure 4-4: Fault insertion multiplexer.....	35
Figure 4-5: Output Comparison.....	35
Figure 4-6. Example Directed Graph Circuit.....	36
Figure 5-1. A block diagram of the FPGA emulation system supported through software running on a host compute.....	41
Figure 5-2: Fault detection algorithm.....	43
Figure 5-3: A block diagram of the FPGA emulation system and Power PC supported through software running on a host computer.....	46
Figure 5-4: State diagram of the fault detection algorithm.....	49
Figure 5-5. Flowchart of emulation algorithm.....	50
Figure 5-6: A 4-bit LFSR with taps at bits 3 and 4.....	52
Figure 5-7: Fault changing process.....	55

Figure 5-8: Combining new test vector data with recorded data. All test vectors are displayed here in Hexadecimal notation.....	56
Figure 6-1. Total time taken for test pattern generation using random vectors for C432 as a function of the number of instantiations.....	63
Figure 6-2: Total FPGA hardware resources used for test pattern generation using random vectors for C432 as a function of the number of instantiations.....	64
Figure 6-3. Plot of number of vectors tested versus faults found for 32 instantiations of C432.....	66
Figure 6-4. Plot of total time taken versus number of vectors tested for 32 instantiations of C432.....	67
Figure 6-5. Graph of number of vectors tested versus number of vectors needed.....	69
Figure 6-6. Graph of vectors tested vs vectors needed for z9sym.....	71
Figure 6-7. Graph of Instantiations Vs Time for z9sym.....	72
Figure 6-8. Fault coverage percentage in the benchmark circuits.....	75
Figure 7-1. Bit counter in the current algorithm.....	78
Figure 7-2. Improved bit counter.....	79

Abstract:

In the past few decades, integrated circuits have become a major part of everyday life. Every circuit that is created needs to be tested for faults so faulty circuits are not sent to end-users. The creation of these tests is time consuming, costly and difficult to perform on larger circuits. This research presents a novel method for fault detection and test pattern reduction in integrated circuitry under test. By leveraging the FPGA's reconfigurability and parallel processing capabilities, a speed up in fault detection can be achieved over previous computer simulation techniques. This work presents the following contributions to the field of Stuck-At-Fault detection:

- We present a new method for inserting faults into a circuit net list. Given any circuit netlist, our tool can insert multiplexers into a circuit at correct internal nodes to aid in fault emulation on reconfigurable hardware.
- We present a parallel method of fault emulation. The benefit of the FPGA is not only its ability to implement any circuit, but its ability to process data in parallel. This research utilizes this to create a more efficient emulation method that implements numerous copies of the same circuit in the FPGA.
- A new method to organize the most efficient faults. Most methods for determining the minimum number of inputs to cover the most faults require sophisticated software programs that use heuristics. By utilizing hardware, this research is able to process data faster and use a simpler method for an efficient way of minimizing inputs.

Chapter 1: Introduction

Since the invention of the transistor circuit in 1925, computing has been changed drastically. Every year circuits become smaller as the technology to build them becomes more sophisticated. Researchers and circuit developers strive to accommodate Moore's Law and increase the number transistors on a circuit by two times every eighteen months. This trend has continued for almost four decades and thus current processing units have a transistor count on the order of 10^9 .

This large number of transistors and the increase in complexity of modern integrated circuits (ICs) make it almost impossible for engineers to manually check for errors in every part of their circuitry. Faults can manifest themselves as permanent defects in wires and transistors during the complex manufacturing steps. Transient errors can also arise during circuit runtime due to delay mismatch, excess leakage current and other parameter variations. Some of these faults and errors can be checked for during production and the IC can be discarded before it causes errors for the end-user. One such error, the stuck-at-fault deals with wires within the circuit being stuck at either a high voltage, a stuck-at-one fault (SA1), or stuck at a low voltage, a stuck-at-zero fault (SA0).

A general way to test for stuck-at faults after manufacturing is to run a set of input stimulus and compare the output of the integrated circuit with a set of expected outputs. Any deviation from the expected output set results in a functional error. Since the number of input stimulus grows exponentially with the number of inputs to the circuit, finding a representative subset of the circuit is of utmost importance. Automatic Test Pattern

Generation (ATPG) is a technique that aims at finding a compact and optimal test-set to detect all possible faults in the system. For test pattern generation, there have been two main methods of generating input vectors –circuit simulation and emulation.

Simulation uses a computer program to determine the best possible vectors. As with any process there are a number of strengths and weaknesses associated with simulation.

Simulation allows engineers to reproduce any detail of a circuit. Engineers may also decide to simulate higher level behavior before they decide to simulate a net list. This is done primarily to see that the higher levels of behavior fall within the acceptable performance boundaries. The major downfall to the idea of simulation is that to comprehensively simulate a complex circuit for all possible faults, a standard desktop computer will not be enough. This logically leads to the fact that it is not possible to test all possible inputs for circuits with larger input vectors. On the other hand emulation, a recent development due to improvements in the area of Field Programmable Gate Arrays (FPGA), allows for a hardware based solution to this issue.

With the advent of the FPGA and its proliferation in research laboratories, creating hardware implementation of simulation techniques has become a viable alternative. Until 1985, when the FPGA was first commercially available [1], it was not logical to use hardware to perform tasks where software was needed for its ability to modify routines with ease. Each change in software would correspond to a new hardware design and a new IC to being created in hardware. This would be costly, both in design and production. The true advantage of the FPGA over simulation techniques and producing

application specific integrated circuits (ASICs) to determine test patterns comes from its ability to be continually reprogrammed. FPGAs can perform any task that an ASIC can, but can be programmed with a new circuit after one has already been implemented. This also means that a circuit can be implemented in hardware without cost. FPGAs can also be tested thoroughly in hardware, not just simulated, a big advantage over its ASIC counterpart. FPGAs also have the advantage of being able to perform software-like processes more efficiently than a computer can. This means there is usually a significant speed up for a specific task that an FPGA performs over the same task performed on a central processing unit (CPU).

An FPGA uses a hardware description language (HDL), Verilog or VHDL, that allows for circuits to be programmed as either net lists, a group of predefined ICs such as AND and OR gates, or using a syntax similar to most current programming languages, such as C++ and JAVA. HDL facilitates the transfer of simulation techniques to an FPGA and allows for programmers to develop ICs easier using concepts that software developers use.

Main Contributions

This thesis presents a novel method for fault detection and test pattern reduction in integrated circuitry under test. By leveraging the FPGA's reconfigurability and parallel processing capabilities, a speed up in fault detection can be achieved over previous computer simulation techniques. This thesis presents the following contributions to the field of Stuck-At-Fault detection:

1. We present a new method for inserting faults into a circuit net list. Given any circuit netlist, our tool can insert multiplexers into a circuit at correct internal nodes to aid in fault emulation on reconfigurable hardware.
2. We present a parallel method of fault emulation. The benefit of the FPGA is not only its ability to implement any circuit, but its ability to process data in parallel. This research utilizes this to create a more efficient emulation method that implements numerous copies of the same circuit in the FPGA.
3. A new method to organize the most efficient faults. Most methods for determining the minimum number of inputs to cover the most faults require sophisticated software programs that use heuristics. By utilizing hardware, this research is able to process data faster and use a simpler method for an efficient way of minimizing inputs.

Organization of the Thesis

Chapter 2 of this thesis provides a brief background on fault-detection and ATPG in integrated circuits. It also provides a discussion on manufacturing tests; its importance in circuit production as well as an exploration of how different testing techniques are implemented. Lastly this chapter discusses existing work directly related to this thesis.

Chapter 3 provides an introduction to the architecture of the Xilinx Virtex II Pro FPGA - the main implementation platform for this research. The chapter will also go over the process of using the Xilinx for this research and the limitations that had to be dealt with.

Chapter 4 discusses stuck-at-fault models and presents our approach of accurately modeling them using multiplexers. Then the chapter also describes the physical process of mapping the faults to the FPGA and how each fault can be emulated with a simple command.

Chapter 5 introduces our algorithm for testing stuck-at-faults in an FPGA. It also discusses our technique of using test results generated in the FPGA to create a compact set of patterns that can be used to test all faults in the original circuit. Chapter 6 presents and analyzes the results obtained from our approaches described in chapters 4 and 5. Finally chapter 7 offers some conclusions and presents some relevant future research ideas.

Chapter 2: Background

Manufacturing Defects

During the manufacturing process of any circuit, any number of defects may occur and thus faults are generated. Wafer testing is implemented at the end of the manufacturing process for integrated circuits (ICs) to make sure that the IC is working correctly. During this process, a series of inputs drive the ICs and the outputs are observed and compared to expected results. Any ICs that do not pass the initial test are tested again to see if on chip resources can help to repair faults that are found. If this cannot be done then redundancy that is built into the chip will be tested and if this fails the chip is marked faulty. The only time that this process can be avoided is when the actual cost of producing the ICs is lower than that of testing them individually.

For every circuit that is produced there can be a large number of sources for permanent faults. Table 1 shows how these faults are distributed in standard printed circuit boards (PCBs). All possible faults are caused by four different types of defects: process defects, material defects, age defects, and package defects [2].

Process defects and *material defects* come from an imperfection in the materials used during the creating of a circuit. This is caused predominantly by dust on either the mask or wafer surface or in the chemicals during the photolithography process. The presence of dust and other foreign objects causes *unexposed photoresist* areas or *unresist pinholes*. This leads to excess unetched material in an improper area or excess etching in an incorrect area. These issues can lead to incorrectly placed or absent contact windows,

parasitic transistors, oxide breakdown, and material imperfections. Errors of this kind account for about 60% of the defects in manufactured circuits [3].

Age defects are problems that occur over time and with repeated use of a device. These defects include and are not limited to time-dependent dielectric breakdown, charge breakdown, hot carrier injection, negative bias temperature instability, and electromigration. While these defects were largely ignored before; the increased scaling of transistors and the increased variability in the manufacturing process has forced the IC industry to run efficient tests to measure reliability against aging. For testing, the wafer or device is subjected to variety of stress-measures (thermal, high current etc.) and a reliability curve is generated that extrapolates the measured data to predict the operating lifetime of a device [4].

Finally, *package defects* are issues that are related to the actual physical housing of any circuit. If there is an opening or leads are broken a different set of faults is created.

Usually these errors will be detected during testing as an open [5].

Defect Type	Frequency of Occurrence (%)
Shorts	51
Opens	1
Missing components	6
Wrong components	13
Reversed components	6
Bent leads	8
Wrong analog specifications	5
Defective digital logic	5
Performance defects	5

Table 2-1. Typical defects in a standard PCB [2]

Fault Models

Before we talk about integrated circuit testing, it is important to propose a model for how and why faults occur and how their existence impacts the rest of the circuitry. For efficient manufacturing test, the following fault models are generally used.

Stuck-At-Fault – This fault occurs when a signal line within a circuit is permanently set to either logic high, 1, or a logic low, 0. This fault model does not have a specific cause; rather, it is an abstract fault model with numerous causes, some of which are faults that are described below. Figure 2-1 shows a portion of a circuit where an internal circuit line has been permanently stuck-at-0. This incorrect value at this node propagates to the rest of the circuit and causes an incorrect outcome at the output. The incorrect output due to the fault is shown in parenthesis at each affected node. The stuck-at fault model is one of the most popular fault models and is the subject of research in this thesis.

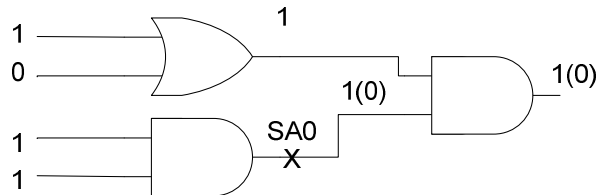


Figure 2-1. Single Stuck-At Fault at an internal node of a circuit.

As stated above, the category of stuck-at-faults contains a number of different classifications that current simulators use.

Potentially Detectable Fault – This fault is characterized as an unknown state (X) at a primary output (PO). There is a probability of detection associated with it mostly around 50%

Initialization Fault – An initialization fault is a error in a circuit that interferes with the initialization of any kind of memory device, i.e. flip flops. One such example of this is a flip-flop's clock which has a stuck-at-fault and is left in an inactive state. Figure 2-2 below shows how an initialization fault can affect a circuit. Due to the SA0 at the input, A, it becomes impossible to determine what the output of Q will be at any moment in time.

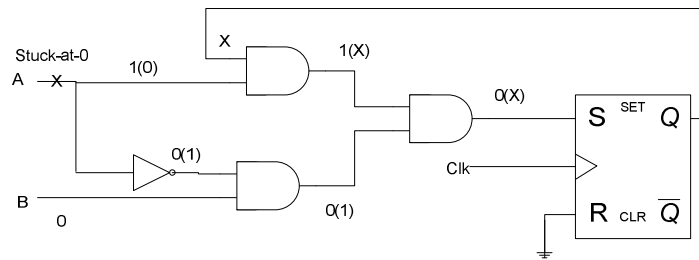


Figure 2-2. Initialization Fault

Hyperactive Fault – Hyperactive faults are defined as a signal that causes a large number of signals in a circuit to differ from their correct value.

Redundant Fault – This fault has no test because it does not affect the input-output function of a device in any detectable way. They are often removed simply for circuit optimization. This definition applies to combinational circuitry but it can also apply to sequential circuitry. In sequential circuits the definition is vague and so these faults more fall into the undetectable fault category.

Undetectable Fault – Faults that fall under this category simply are impossible for test generators to detect.

In addition to single stuck at faults, there are situations where numerous stuck-at-faults may be in a circuit. This is called a **multiple stuck-at-fault** model and it can cause single stuck-at-fault tests to fail. In most situations where there are multiple stuck at faults, the faults will not mask each other and not affect the effectiveness of a single stuck-at-fault test. If the faults do manage to mask each other the single stuck-at-fault test becomes ineffective although this is statistically unlikely to occur [2].

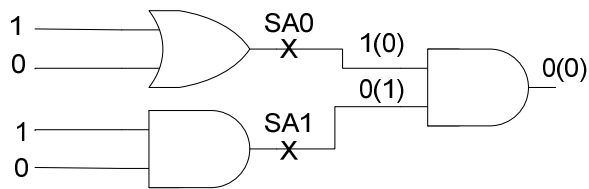


Figure 2-3. Multiple Stuck-At-Faults

Bridging Fault – This fault is defined as a short between groups of signals. This may cause an OR bridge or an AND bridge which are ones dominant or zeros dominant respectively. These faults can be found with tests used for finding stuck-at-faults.

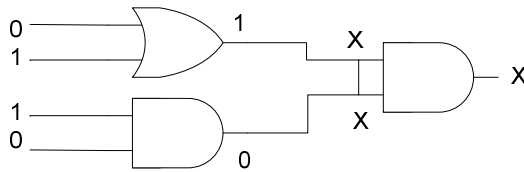


Figure 2-4. Bridging Fault

These faults can fall under the category of potentially detectable stuck-at-faults because it is not known whether a line will be pulled up or pulled down with a short at any point at time, this is why it is assigned an X value at the site of the short and from the output onwards. Because of this, the bridging fault will not always be detectable.

Delay Fault – This fault is active when the combinational logic delay exceeds the specified clock period. The variation in the manufacturing process can cause certain portions of a circuit to be slower than other parts of the circuits causing internal signals to arrive at different times and cause functional failure. The fault is usually modeled as either a gate delay fault model where a single gate is assumed responsible for producing the slow response or a path delay fault model where certain interconnects and paths are responsible for slow propagation of a signal. This fault is outside the scope of this research and will not be dealt with.

Stuck-Open and Stuck-Short Faults – This fault is defined as a single transistor that has either been stuck open, no current will ever pass through, or stuck short, lacking the ability to stop current. As with bridge faults, the stuck-open faults can be detected by running a sequence of stuck-at fault detection vectors.

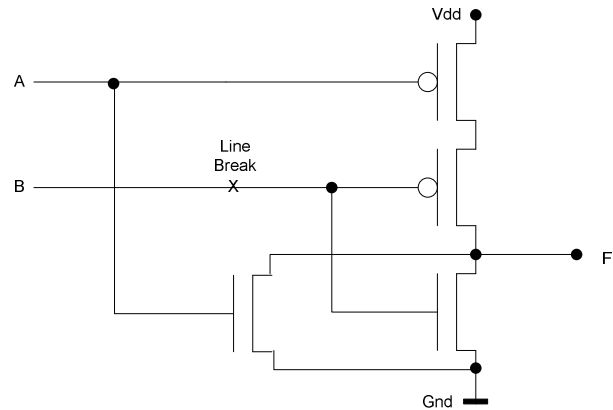


Figure 2-5. Stuck Open Fault

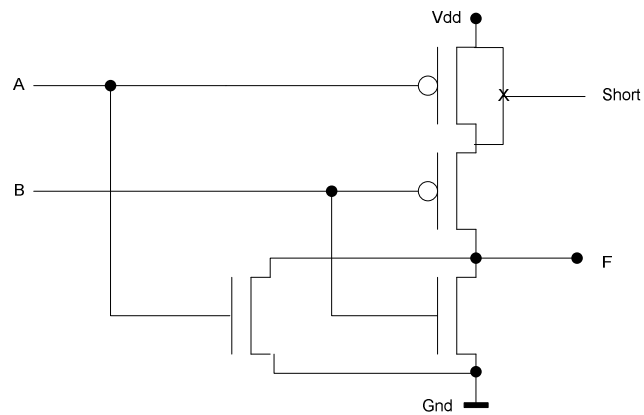


Figure 2-6. Stuck short Fault

These two faults are specific ways that stuck-at-faults can occur. From Figures 2-5 and 2-6, it can be seen that the outputs of specific transistors can take on a permanent low or high signal thus causing a SA1 or an SA0.

This thesis will primarily deal with the stuck-at fault model. Due to the nature of the stuck-open, stuck-close, and bridging faults, they will also be covered as stuck at faults in this thesis.

Related Work on fault simulation and detection

For the past few decades, fault checking has been done by a variety of different types of simulators. There are a variety of approaches that have been developed to simulate stuck-at faults in different ways [2]. **Serial fault simulation** enumerates all possible correct outcomes of a circuit, saves them to a file, and compares them with a circuit that has a certain input vector. This method takes, at most, N-times (where N is the number of faults) the computer time that a true-value simulator¹ does, although it can be shorter if numerous faults are found with one vector. This method is used primarily for combinational logic circuits as it cannot compensate for transitional signals without significant modification.

Building upon serial fault simulation, **parallel fault simulation** was developed [2]. This method can only be used with combinational logic circuitry. By using the word length of a machine it tests up to the word length minus one number of vectors i.e. a 32-bit word could test 31 vectors at a time. The remaining bit is used as the signal value of the fault free circuit as a comparison. This process cannot compensate for the fall and rise of signals, but it does allow for the ability to control which fault is being tested in a circuit.

Another simulation method is **deductive fault simulation**[6]. This method tests one vector at a time and creates lists of possible faults that can be detected on every wire. At every gate, logic is used to determine which of the faults in the lists will be propagated. For example, if an OR gate is present and an input is a true, no faults will propagate. If

¹ True-value simulator – A simulation and detection of one fault through comparison with a clean circuit.

the input is false, a fault will propagate. A fault that propagates to a primary output is considered a fault that is found by the vector. Given certain specifications, this method can be incredibly fast, but factors such as multiple signal states and variable delays can require major changes in the implementation of the method.

Concurrent fault simulation is a simulation technique that uses numerous copies of every gate in the circuit to determine if the circuit has faults [7]. Similar to deductive and serial fault simulation, this method iterates through each possible input vector. Then, by creating a number of gates that could possibly have faulty inputs it propagates all the possible outputs forward. If there are any incorrect values at the output, that vector is determined to be able to detect faults. This simulation technique is the most general of all of the previously listed techniques because it works with different circuit models, faults, signals states, and timing models.

Roth's TEST DETECT algorithm was developed in 1966 by J. P. Roth at IBM [8] to reduce the total computational complexity of parallel fault simulation by exploiting the relationship between faults. This algorithm starts by testing the true value of the circuit with a vector, then injects a fault at a certain wire and designates either a D (representing an error) or a D-not at the fault site. This technique is unique because it propagates the D throughout the circuit with a technique called *D-Calculus* and if the D reaches a primary output then the fault is designated as found with the tested vector.

The TEST DETECT technique was improved upon with **differential fault simulation** [9]. This technique eliminated the need for the D-calculus and the need to constantly

restore the true-value of the circuit every time a new stuck-at fault was tested. This makes the algorithm simple to implement and reduces the number of simulation runs by approximately half.

Recent research is moving toward improving fault detection speeds by hybridizing these simulation techniques and improving the technology upon which they run. In [10] the authors use Graphics Processing Units (GPUs) as opposed to general purpose processors (GPPs) to simulate faults. GPUs are designed for multithreaded processes which makes testing multiple faults more efficient. The findings of this research are that it is orders of magnitude faster to use a GPU rather than a GPP to process the gates in parallel. Results have also shown that systems with more GPUs, the results would be even faster.

Instead of just using simulation, researchers have also focused their efforts into fault emulation [11]. The use of FPGAs has shown to be more efficient at fault checking for sequential circuits which have large numbers of input vectors and long input sequences. Other improvements in the future are also discussed such as better hardware, an automated synthesis flow to the testing environment, and better partitioning methods for emulation of fault list partitioning in an academic environment where industrial grade hardware is not accessible.

The same authors use an FPGA to emulate faults for various circuits in a serial manner [12]. The authors created a process by which they create a circuit on an FPGA and use injected multiplexers to control test vectors and stuck-at faults. They also use a series of decoders to decide which vectors indicate faults in the circuit. The results showed that

the use of FPGA emulation was faster than simulation. The drawback to this was that there is a synthesis time required, where the circuit is mapped onto the FPGA, and this requires a much larger amount of time than simulation. Synthesis time in a larger circuit is longer than simulation, but it becomes less significant as the number of inputs and possible test vectors increases. The authors note, that if this technique is used in testing larger circuits as well as sequential circuits the technique can be a useful fault finding method.

In [13-14], the approach is to emulate circuit faults using the D-algorithm on FPGAs. The D-Algorithm has been proven to be able to be implemented in hardware. It allows for the speed of hardware without needing specific circuit-testing hardware. The authors were able to implement fine-grain parallel processing, such as forward/backward implications and conflict checking. Forward/backward implications are the process of checking for errors by first counting upwards in the inputs and then counting down. Error checking in both directions is necessary for generating the smallest number of test vectors. Conflict checking is a process that checks reactions within a circuit in response to a change in the inputs. These inputs are then propagated to the primary outputs in both correct and faulty circuits.

Other researchers have built upon the use of FPGAs and proposed their own techniques for speeding up the process. The researchers in [15] proposed two techniques to increase the speed of the fault simulation process by injecting numerous independent faults into the same circuit and injecting multiple dependent faults at the same time. Synthesis time

is the bottleneck of the FPGA's usage. By reducing the number of reconfigurations the overall time for emulation is dramatically reduced.

The authors of [16] introduce a technique that takes advantage of the internal look-up tables (LUTs) of FPGAs and is distinct from other FPGA-based fault emulation techniques. They use a technique of reducing stuck at faults called fault dominance, which states that certain faults “dominate” other faults in a circuit and allows for some faults to be ignored. This can help to both reduce the number of faults checked for and reduce the checking logic. If a set of gates with a dominant fault at the output can be grouped together and emulated by a single LUT the circuit size will reduce and the emulation time will drop significantly.

Emulation has also become a topic of research in the field of fault location. The researchers of [17] attempt to use fault emulation with FPGAs to figure out where a specific hardware fault is located within both sequential and combinational circuits. By using a ranking system they were able to remove the need for any software, including large software-based fault dictionaries. They found that the increase in performance was, on average, around 300 times faster than traditional simulation techniques.

Chapter 3: Xilinx Virtex II Architecture

Generic Xilinx FPGA Architecture

All Xilinx FPGAs follow a specific architecture scheme that starts with a number of input and output blocks (IOBs) that surround the core logic. The core logic is made up of configurable logic blocks (CLBs) which have a number of resources that help connect them to each other and the IOBs. These CLBs are what is responsible for producing the desired result of the programmer.

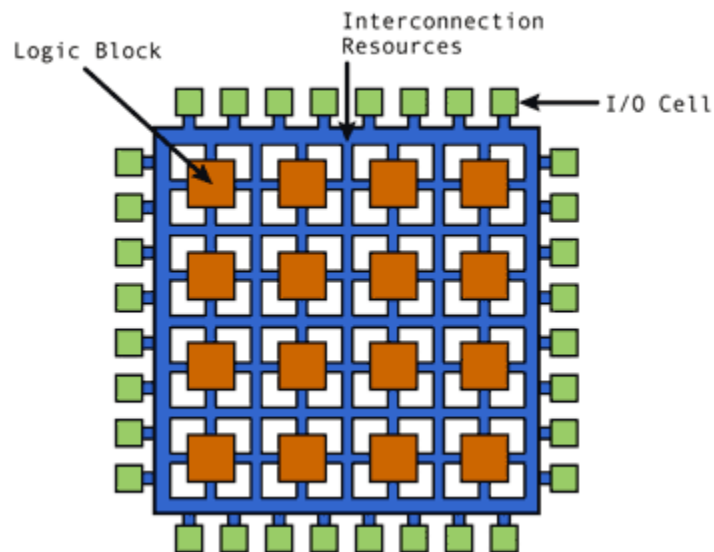


Figure 3-1. FPGA General Architecture [18]

To program the device, there is an internal static memory that determines how the CLBs will behave and how they will connect to one another. The data that goes here is loaded into the FPGA at power up or reconfigured by the user. Figure 3-2 shows what makes up a CLB. Each one contains sets of what are basically RAM devices with 16 addresses.

These act as LUTs and shift register loop up tables (SRLs). These are then connected to multiplexers and arithmetic logic which determine what data to pass into the register/latch for the final output.

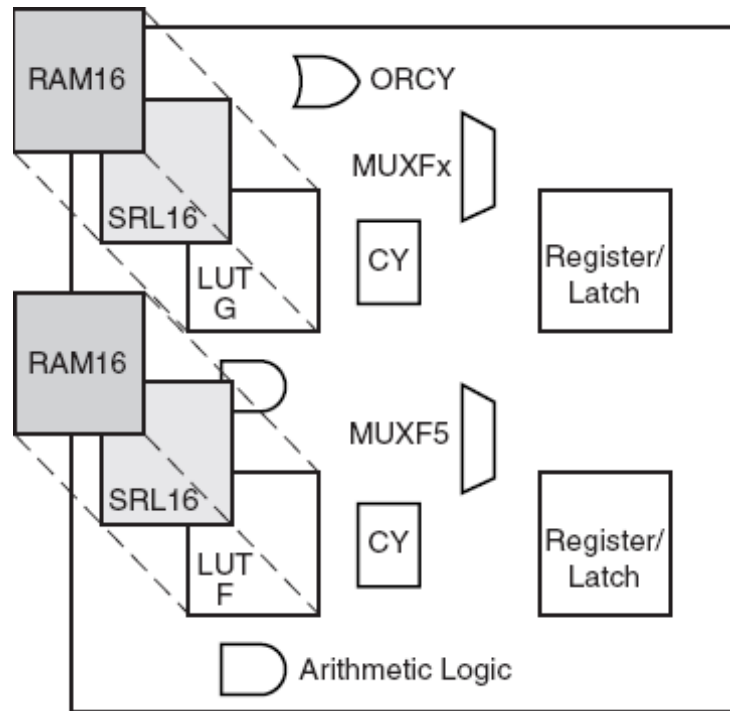


Figure 3-2. Virtex II Pro slice configuration [19]

The CLBs are where all of the data is stored and different logical processes are performed. The LUTs store data that is known before initialization while the register/latches are the places where processed data is stored.

Virtex II Pro FPGA Architecture

The primary processor in this research was the Virtex II Pro FPGA - the Xilinx Virtex-II Pro Development System. The board houses a Xilinx XC2VP30 FPGA with 30,816 Logic Cells, 136 18-bit multipliers, 2,448Kb of block RAM, and two PowerPC Processor

cores [19]. In Figure 3-3, the physical layout of an FPGA similar to the chip used in this design is shown. The only appreciable difference is the lack of two PowerPC CPUs in this, but because this design only uses one of these it is an acceptable example of the hardware being used.

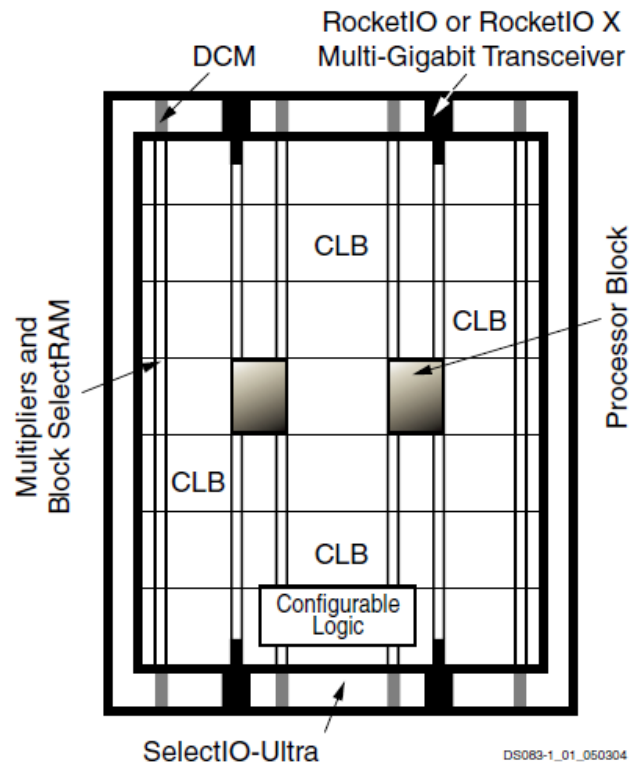


Figure 3-3. Physical Layout of FPGA (XC2VP30) [19]

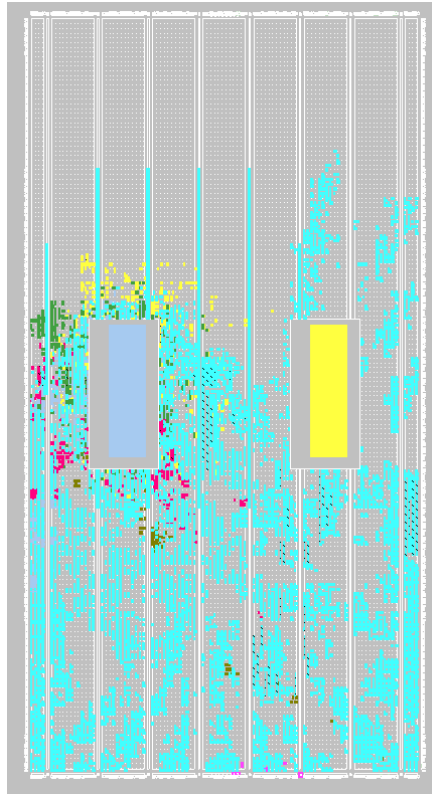


Figure 3-4. Layout of C432 design on the XC2VP30

It can be seen that there are several columns of CLBs, BRAM columns and the Digital Clock Managers throughout the device. In a direct comparison, Figure 3-4 shows the actual layout of the Virtex II Pro FPGA that we are using for this research. The major difference between the two is that there are two PowerPC CPUs in the FPGA used in this research. In addition to these difference, the actual implementation of the C432 fault detection algorithm (described in Chapters 5 and 6) is shown in the highlighted blocks of the layout. Figure 3-5 below is the block diagram of the Xilinx Development Board with the C432 fault detection core implemented. The components of both Figure 3-3 and Figure 3-5 will be explained thoroughly in this chapter.

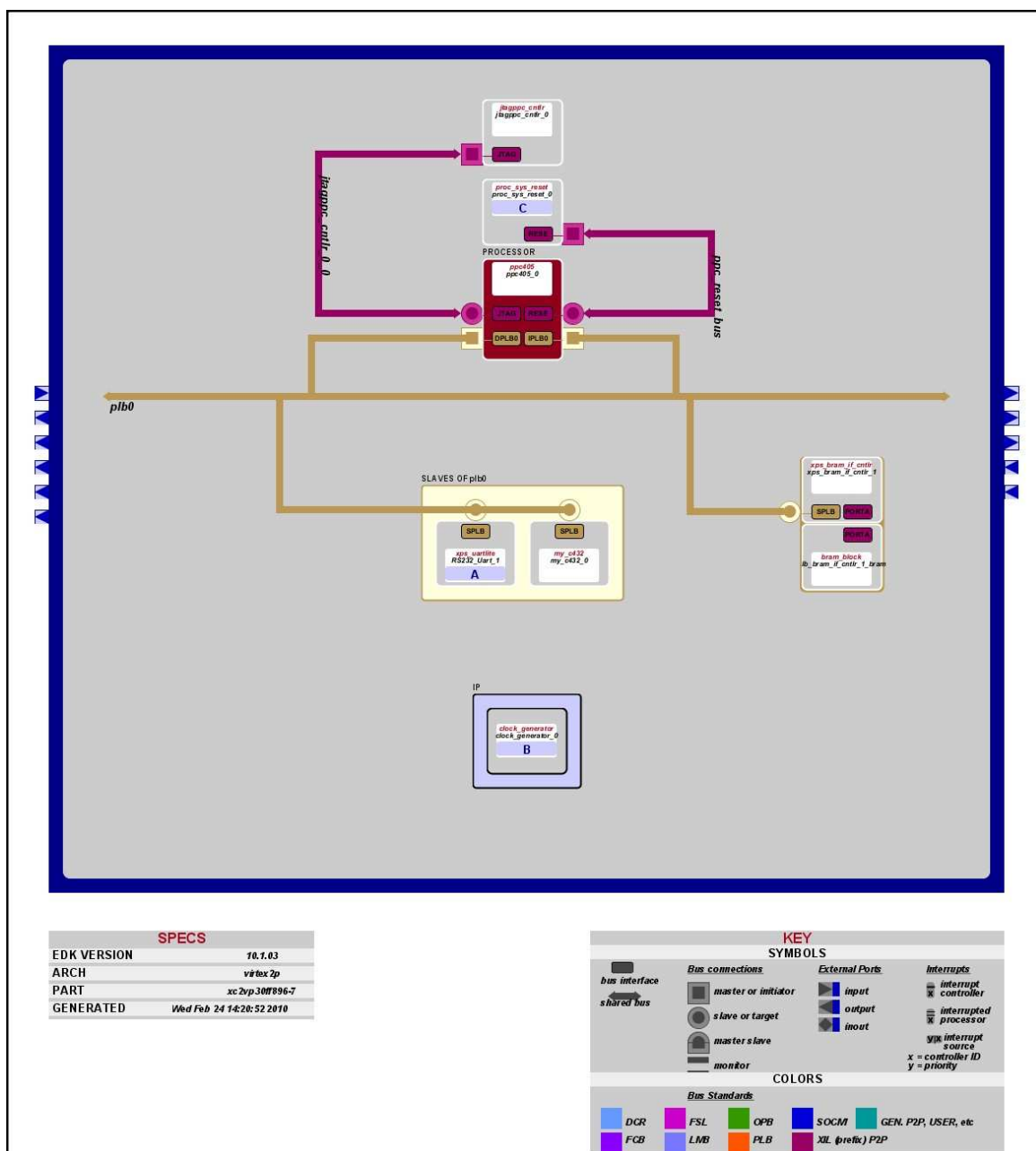


Figure 3-5. Block diagram of the Xilinx Development Board

Power PC

The PowerPC is built with a 64-bit architecture that can run in a 32-bit mode. It is meant to work as a processor for system-on-a-chip designs. This CPU was designed with 5 pipeline stages, 16 KB of instruction and data caches, and can run at clock rates of up to and above 400 MHz[20]. There are three different levels of architecture that the PowerPC provides and are described in [21]. These include the User Instruction-Set Architecture (UISA), Virtual Environment Architecture (VEA), and Operating Environment Architecture (OEA) .

The UISA level is the base level for the PowerPC architecture. This level defines the architecture that the user's software should be compatible with. All of the instruction set, user-level registers, data types, floating-point memory conventions, memory model, programming model, and exception model as seen by the user are defined by the UISA.

After the UISA is the VEA which defines the features in the architecture that allow applications to create and change code, make memory storage discernable, and optimize the performance of the memory-access.[Xilinx ug018]. This environment also defines the cache and memory models, the timekeeping resources from a user perspective, user mode resources that are primarily used by system-library routines. The storage model of this architecture level, defines the storage-control instructions that are defined in the PowerPC VEA. Storage-control instructions are used to manage the interactions of instruction and data caches. The storage attributes, write-through, cacheability, memory coherence, guardian, and endian, are defined by the storage model as well. Finally the

storage model controls the operands, their requirements and how they affect the performance.

The OEA defines the parts of the architecture that allows the CPU to work as an operating system. This includes input and output interactions and memory control.

Privileged access and exception handling is also dealt with in this architecture.

Figure 3-6 below is a diagram of the PowerPC with blocks representing its capabilities as a CPU and the interconnections between the different processing elements. The PLB interface is described as being directly connected to the cache units. From here the data is sent to either the instruction or data processor.

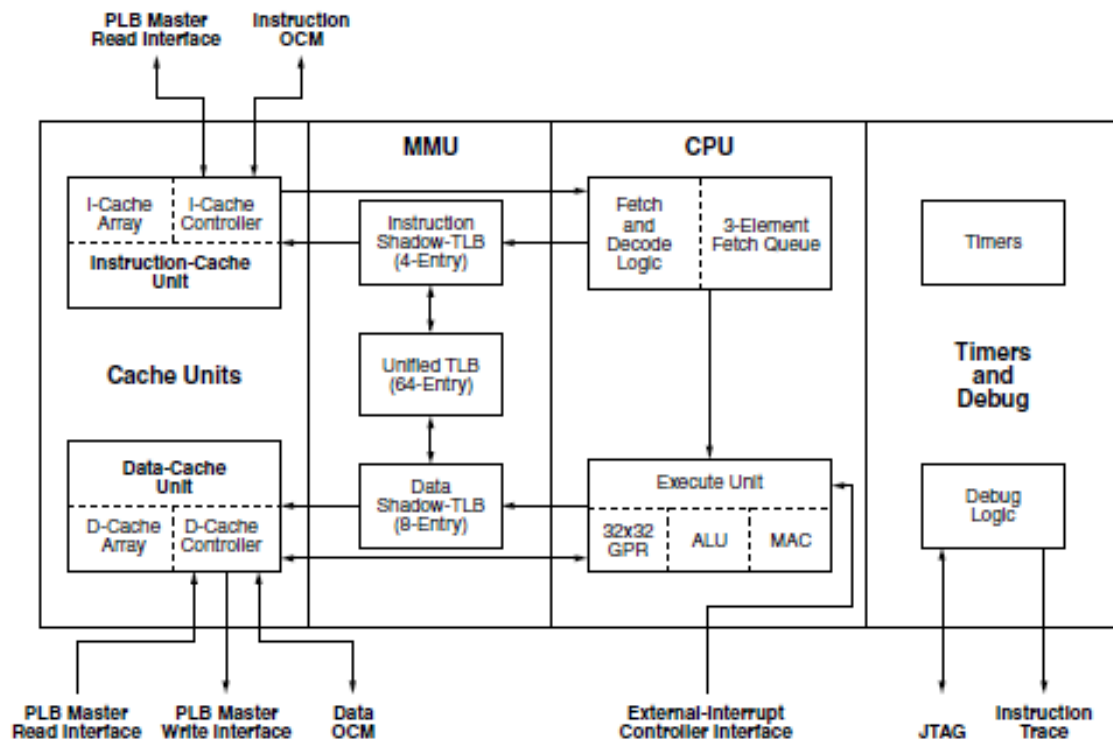


Figure 3-6. Functional blocks of the PowerPC 405 processor [21]

Processor Local Bus (PLB)

To connect the different parts of this design, the PLB is implemented. In Figure 3-4, this is represented by the beige lines that connect different parts of the design. Every device connected to this bus is regulated by the device. As described in [19] the PLB can run at either 32 or 64 bits, but for this design only the 32 bit operation is needed. Below in Figure 3-7, it can be seen how the PLB connects to the major blocks.

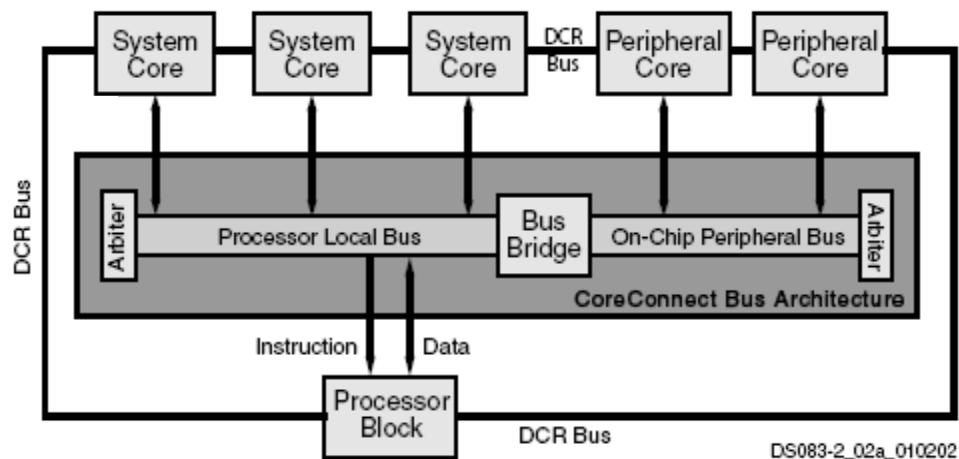


Figure 3-7. Interconnects in the Virtex II Pro [19]

RS232 UART

Because output speed was not a major concern, an RS232 serial communications port was used to connect the output of the Xilinx Development Board to a hyperterminal application on the user's computer. This is a standard 9-pin UART device that handshakes with a computer's COM port. Depending on the design, this device was used at speeds ranging from 9600-115200bits/sec. The implementation of this device is described in [22].

This RS232 serial port uses voltages ranging between +5 and +15V for a logical high and -5 and -15V for a logical low. These voltages ensure that the signal will be able to be read even at the maximum cable length of 50 feet. The RS232 port has five signals that are communicated between itself and the FPGA. These include the RS232_TX_DATA, RS232_DSR_OUT, RS232_CTS_OUT, RS232_RX_DATA, and RS232_RTS_IN. The RS232_TX_DATA, or transfer signal, carries the output signal and the RS232_RX_DATA, or receive signal, carries the input data (not used in this design). The other three signals, RS232_DSR_OUT, RS232_CTS_OUT, and RS232_RTS_IN, are the data set receive, clear to send, and request to send signals. These are the hardware control signals that allow both ends of the connection to know whether or not data is being sent, ready to be sent, or it needs permission to be sent respectively. Figure 3-8 below shows how the connections for each of these signals are connected to the RS232 transceiver on the Xilinx Board.

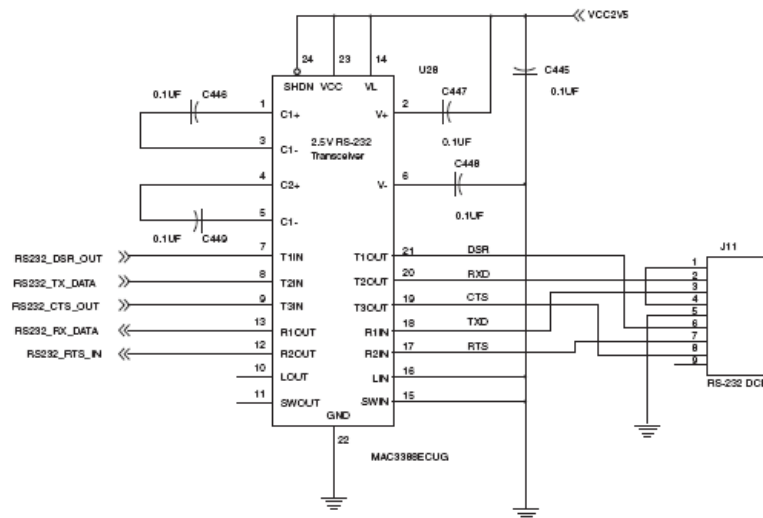


Figure 3-8. RS232 implementation [22]

First In First Out Memory/Block RAM

Between the Power PC and the FPGA core that was implemented was a first in, first out (FIFO) memory cell that exchanged input and output data between the two processors. These FIFOs act as data buffers that can be accessed at any time. There is both a write and receive FIFO with respect to the FPGA core. From the PowerPC a user may input data into the receive FIFO, and this data may be extracted from the FPGA. The write FIFO works in reverse, taking data in from the FPGA core and having it read in the PowerPC. In addition to the data inputs and outputs, there are other signals that can be tapped from these FIFO regarding how much data is stored in either, whether or not they are full, or if they are close to being full. In Figure 3-9, the block diagram of a FIFO interconnect is shown. It is seen where the signals described above are placed and how they are connected to the BRAM.

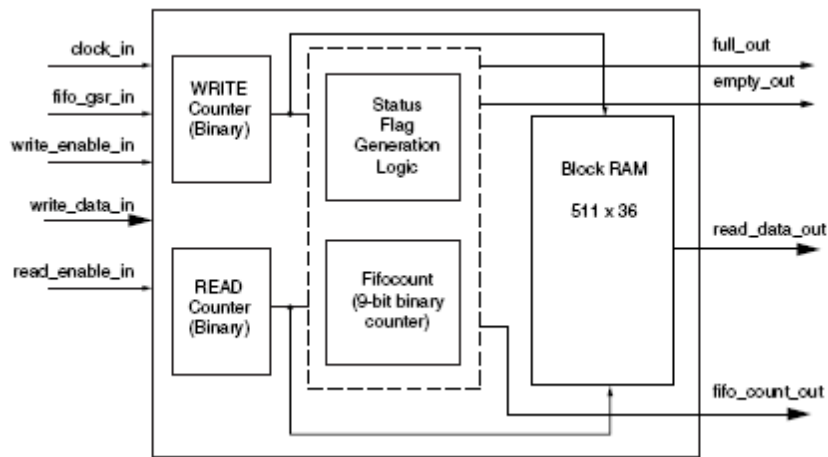


Figure 3-9. FIFO layout using BRAM [23]

Clock Generator

The block of this design that was used to control the clocks, was aptly name the clock generator. Described in [24] and made up of several Digital Clock Managers (DCMs), a Phase Locked Loop (PLL), and several buffers and inverters, this device takes in an external clock and allows the user to create several other clocks that run at different frequencies and with different phase shifts.

In Figure 3-10 below, the block diagram for the clock generator is shown. The first half of the device is where the user defined clocks are implemented. The input clock is put through a PLL so that the clocks may be synchronized and then through the DCM where different clocks can be created with specific frequency and phase shift requirements.

Using the DCM it is also possible to create faster clock speeds than original input signal. From here these clocks are buffered to improve signal strength and then sent out to the cores that use them.

The bottom half of the Figure 3-10 block diagram describes the clock signal sent throughout the FPGA. This clock signal is sent through a DCM for tuning and then sent out to a clock distribution network that is present throughout the entirety of the FPGA. This allows for clocks to be accessed from almost anywhere on the FPGA without losing significant signal strength or losing time due to delays. The clock is also sent back in so that it may be compared to the original input clock to make sure they are synchronized.

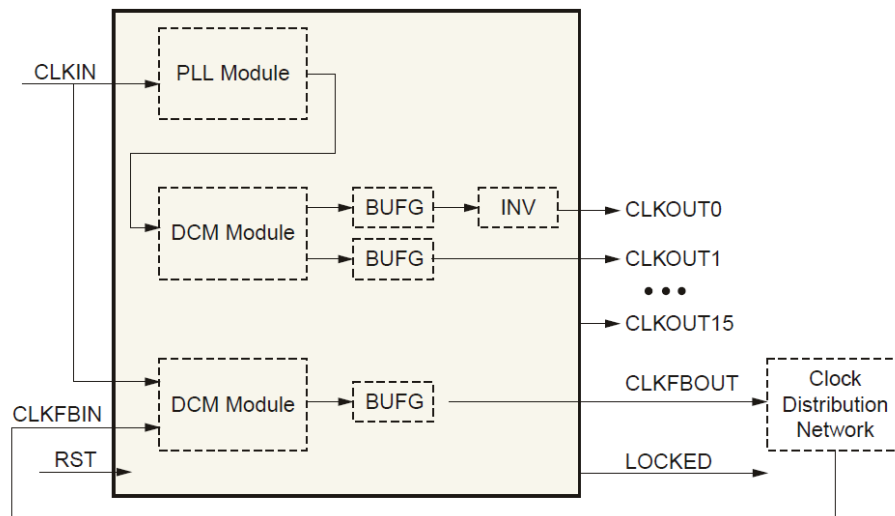


Figure 3-10. Clock Generator block diagram [24]

JTAG, System Reset, and Additional Buses

In addition to the devices mentioned above, there are several peripheral devices used for controlling interconnections within the development board shown in Figure 3-4. These include the JTAG PPC controller and the system reset and their respective busses which are described in [23-24]. Both of these devices are responsible for controlling the PowerPC CPUs in the Xilinx FPGA.

The JTAG PPC block is responsible for controlling the connection between two different PowerPC CPUs when they are implemented in the FPGA. Only one PowerPC is instantiated in this design so this device is negligible. The system reset block on the other hand is responsible for initializing both the PowerPC and the FPGA. All initializations determined in the Verilog code are implemented in this block as well as the PowerPC CPU.

Chapter 4: Error Insertion

With any method of fault testing and test vector generation, faults need to be simulated in a working version of a circuit. This can be a daunting task because of all of the possible fault sites. There are faults on every input, intermediate output, and fan out. To reduce this number, the concepts of fault dominance and fault equivalence are used.

Fault Equivalence

Fault equivalence is defined as two faults that are triggered by the same signals. If two faults x and y are equivalent; and if fault x is triggered by a specific input signal, then fault y will as well. The test for this is shown below in Equation 4-1.

$$f_1(V) \oplus f_2(V) = 0$$

Equation 4-1. The *Indistinguishability Condition*, determines if two faults are identical or equivalent [2]

By using Equation 4-1 a large number of faults can be eliminated from consideration immediately. In [2] it is stated that an n -line circuit this is an inefficient process and requires $2(n^2-n)$ pairs of faults to be compared. To improve this reduction algorithm, a comparison of faults that surround different Boolean gates can be used to find equivalent faults. These equivalences can be seen in Figure 4-1. For example, consider the two-input AND gate shown in Figure 4-1. It has two inputs and one output. Each input and output node can be either stuck at 1 or 0. This gives the total fault count for the AND as 3. However, consider the case input A stuck-at-0. To test this fault, we would excite A by exciting a 1 at the input and propagate the reaction of node A to the output by making

the second input B a logic 1. So input vector $\langle A, B \rangle = \langle 1, 1 \rangle$ propagates the fault A stuck-at-0 to the output, the expected value at the output is a 1 but we see a 0. Hence, $\langle 1, 1 \rangle$ detects A stuck-at-0. Similarly, $\langle 1, 1 \rangle$ also detects fault B-stuck-at-0 and C-stuck-at-0. Hence, faults A-stuck-at-0, B-stuck-at-0 and C-stuck-at-0 are considered equivalent. A vector that detects one will detect all the three faults. Similar analysis can be used for the Stuck-at-1 problem for the AND gate and can be expanded to find equivalence in the other logic gates.

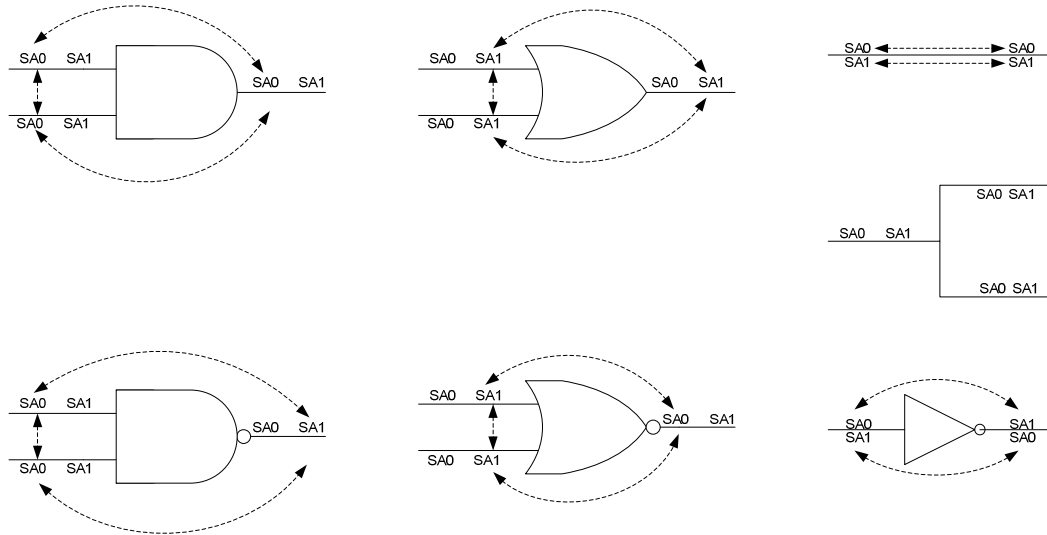


Figure 4-1. Fault Equivalence in series of logic gates, wires, and fan outs

Fan-out trees are special cases. Figure 4-1 suggests no equivalence between the faults in the fan out stem and the branches. The reason there appears to be no equivalent faults for the fan-out is because the two branches may have independent stuck-at-faults that differ from the stem. This means that fan outs cannot have their faults reduced through equivalence.

Fault Dominance

In addition to the concept of fault equivalence is the idea of fault dominance described in [2]. In fault equivalence two faults have the exact same tests and thus can be eliminated. Fault dominance states that if all the tests for fault $F1$ are a subset of the tests for fault $F2$, then $F2$ dominates $F1$. Figure 4-2 shows the idea of fault dominance in a regular combinational circuit. Consider two faults, input $F1$ output $F2$, both stuck-at-1 faults. For the $F1$ stuck-at-1, the only vector that would excite $F1$ and propagate the result to the output is $\langle 0,1,1 \rangle$. For the $F2$ stuck-at-1 we need to excite fault $F2$ (i.e. output $F2$ needs to be made a 0). This can be done by setting one or more of the inputs to be a 0. Figure 4-2 shows the 7 possible input vectors that could test $F2$ stuck-at-1. The test of $F1$ is a subset of the $F2$ test and hence can be eliminated. Using such dominance relationship, it can be found that for a three input AND gate, the only faults that need to be tested are stuck-at-0 on all of the inputs and a stuck-at-1 on only any one input.

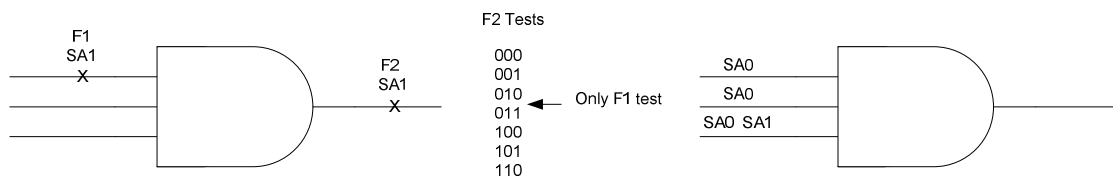


Figure 4-2. Collapsed fault set using dominance

Fault Dominance can be stated with three basic rules laid out in [2]:

1. Any Boolean gate with n inputs will require $n+1$ stuck-at-faults to be considered.
2. To collapse faults on any gate the first step is to eliminate the faults on the output as long as one type of fault (SA1 for AND and NAND; SA1 for OR and NOR) is

- modeled on all of the inputs. The other type of fault (SA0 for AND and NAND; SA0 for OR and NOR) can be reduced to only one input.
3. For NOT gates and non-inverting buffers, the output faults may remain as long as both fault types are modeled on the inputs. Fan outs have no collapsing methods.

Checkpoint Theorem

With the use of the concepts of fault equivalence and fault dominance, a more efficient method of fault modeling can be hybridized. The method used for determining fault locations in this research is the checkpoint theorem. In [2] a checkpoint is defined as all of the primary inputs to a circuit as well as all fan out branches that occur within the circuit. The checkpoint theorem then states that by testing all checkpoints for stuck-at-faults, all possible stuck-at-faults are tested for as well. Looking at Figure 4-3, the difference between standard fault locations and checkpoint fault locations can be seen. The out puts of the first level of AND gates do not need to be checked due to fault dominance. This also applies to the outputs of the second level of AND gates assuming they are not primary outputs. In a circuit that has more levels, the number of faults drops drastically. This would mean more fan outs and lines and the same number of primary inputs.

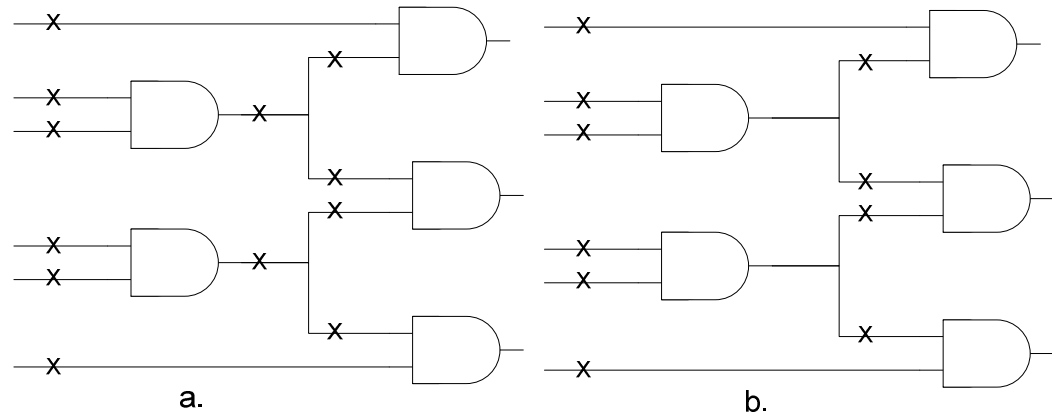


Figure 4-3: a. Standard fault locations. B. A circuit with the checkpoints marked (X).

Although the checkpoint theorem does not reduce as far as using a complete combination of fault equivalence and dominance, it is easy to implement in code and does provide a significant reduction in fault sites. For example in the benchmark C499 there are 499 lines which would imply 998 potential fault sites. With the checkpoint theorem, this is immediately reduced to 594 fault sites, a reduction of approximately 59.5%.

Physical Description of Error Insertion

For this research the method of simulating errors was the use of a two to one multiplexer. At every check point a multiplexer was inserted with connections as specified in Figure 4-4. The first input is the original line from the circuit that would carry the correct data. When the select bit is on it specifies that this specific fault is being activated and depending on the input to S1, either a SA1 or SA0 will be inserted into the circuit.

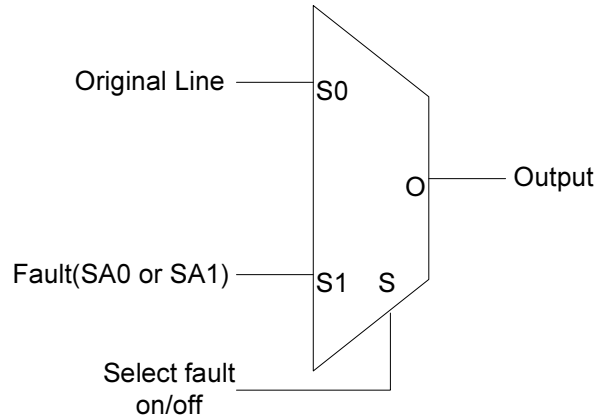


Figure 4-4: Fault insertion multiplexer

To simplify the outputs for the user, the output of the circuit with the fault is compared to that of a fault-free circuit with the use of a series of XOR gates as shown in Figure 4-5. Each output from the faulty circuit is compared to its fault-free equivalent and if there is a high output, the fault has been detected. All of the outputs are then put through an OR gate and this gives a simple high or low answer as to whether or not any of the outputs detected the fault.

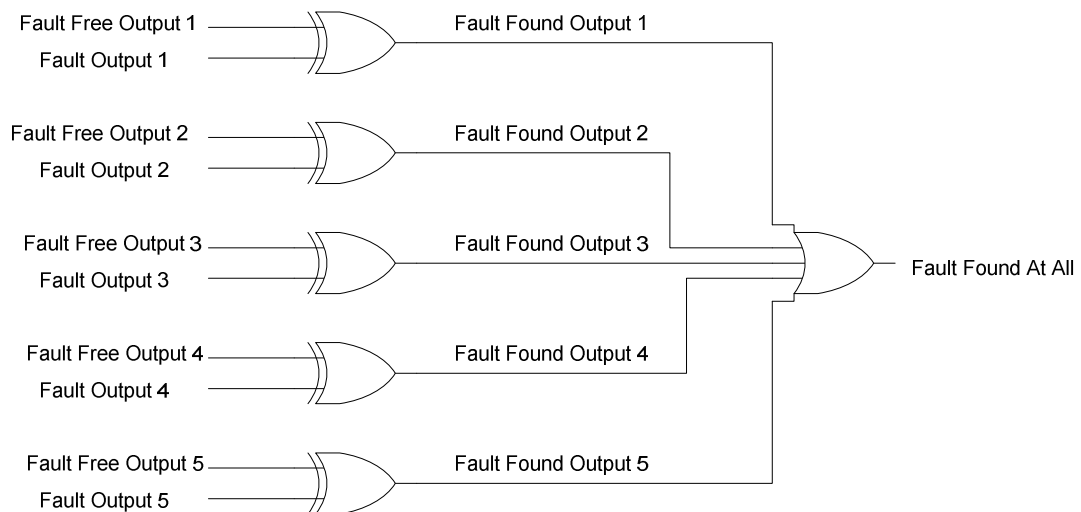


Figure 4-5: Output Comparison

Multiplexer Installation Algorithm

For this research it is assumed that circuits that will be produced will be in a structural netlist format. That is, all circuitry will be described in gate format and all lines between gates will be specified directly. Under this assumption a program in C++ was developed to determine where and how to implement a multiplexer. It uses a directed graph similar to Table 4-1 to log all gates and their inputs and outputs. Figure 4-6 represents the circuit that is described in Table 4-1. This helps to determine the number of fan outs and inputs as needed by checkpoint faults.

Name	Type	Parents
N1	Primary Input	Null
N2	Primary Input	Null
N3	NAND	N1, N2
N4	NOR	N1,N2
N5	AND	N3,N4
N6	OR	N2,N3
N7	AND	N5,N6
N8	XOR	N7,N4

Table 4-1: Example Directed Graph Table

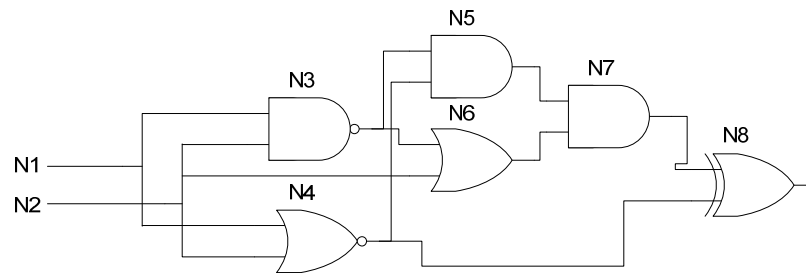


Figure 4-6. Example Directed Graph Circuit

Because this research uses checkpoint faults to determine all stuck-at-faults, the first locations to check are the inputs. Standard Verilog netlist files, such as Example 1 the

C17 benchmark circuit, begin with an instantiation where a “module” is declared with a listing of all of the input and output ports. The next line is usually an input declaration. This line is detected by the code and all of the input wires are logged. They are flagged as primary inputs of the circuit and will be used later. After this, the program goes through all of the gates and logs their inputs and outputs.

```
module c17 (N1,N2,N3,N6,N7,N22,N23);

input N1,N2,N3,N6,N7;

output N22,N23;

wire N10,N11,N16,N19;

nand NAND2_1 (N10, N1, N3);
nand NAND2_2 (N11, N3, N6);
nand NAND2_3 (N16, N2, N11);
nand NAND2_4 (N19, N11, N7);
nand NAND2_5 (N22, N10, N16);
nand NAND2_6 (N23, N16, N19);

endmodule
```

Example 4-1: Code for benchmark circuit c17

After all of the gates have been successfully logged in a format similar to Table 4-1, the output wires from other gates that lead into them are all stored into a vector. This vector is cycled through numerous times, and if the wire appears more than once it is stored in another vector along with the number of times it has been detected. For example in Table 4-1 N1 is used two times, N2 is used three times; N3 is used two times and so on. By doing this, all possible fan outs are detected. With all fan outs and primary inputs detected, it is now possible to install multiplexers into the circuit accurately.

Name	Type	Parents
N1	PI	Null
N2	PI	Null
N3	PI	Null
N6	PI	Null
N7	PI	Null
N10	NAND	N1, N3
N11	NAND	N3, N6
N16	NAND	N2, N11
N19	NAND	N11, N7
N22	NAND	N10, N16
N23	NAND	N16, N19
Parents used more than once: Null,N3,N11,N16		

Table 4-2: C17 Directed Graph Output

In the next step of this process, a new file is opened for writing. The module, input, and output lines are copied directly from the original netlist into the new file. The declaration of wires is also copied over but it has an addendum of all the extra wires needed by the multiplexers. After this is finished, the input multiplexers are installed in a manner similar to Figure 4-2. With primary inputs, the original wire goes into the multiplexer and a wire named the same with a “_0” is the designated output wire. This is done with the fan outs except that the number isn’t always zero. They are enumerated based on the number of fan out wires there are i.e.(“_0”, “_1”, “_2”, ...) At the end of the file, a module is instantiated for the multiplexer. The completed product can be seen below in Example 4-2. The multiplexers are number in the order they are created and all additional inputs can also be seen.

```

module c17test(N1,N2,N3,N6,N7,N22,N23,SS0,SS1,SS2,SS3,SS4,SS5,SS6,SS7,SS8,SS9,SS10, errsig);

input N1,N2,N3,N6,N7,SS0,SS1,SS2,SS3,SS4,SS5,SS6,SS7,SS8,SS9,SS10, errsig;

output N22,N23;

wire N10,N11,N16,N19, N1_0, N2_0, N3_0, N6_0, N7_0, N3_1, N3_2, N11_1, N11_2, N16_1, N16_2;

mux Mux0(N1_0, SS0, errsig, N1);
mux Mux1(N2_0, SS1, errsig, N2);
mux Mux2(N3_0, SS2, errsig, N3);
mux Mux3(N6_0, SS3, errsig, N6);
mux Mux4(N7_0, SS4, errsig, N7);
mux Mux5(N3_2, SS5, errsig, N3_0);
nand NAND2_1 (N10,N1_0, N3_2);
mux Mux6(N3_1, SS6, errsig, N3_0);
nand NAND2_2 (N11,N3_1,N6_0);
mux Mux7(N11_2, SS7, errsig, N11);
nand NAND2_3 (N16,N2_0, N11_2);
mux Mux8(N11_1, SS8, errsig, N11);
nand NAND2_4 (N19,N11_1,N7_0);
mux Mux9(N16_2, SS9, errsig, N16);
nand NAND2_5 (N22, N10, N16_2);
mux Mux10(N16_1, SS10, errsig, N16);
nand NAND2_6 (N23,N16_1,N19);

endmodule

module mux(y,sel,b,a);
input a,b,sel;
output y;

wire sel,a_sel,b_sel;

not U_inv (inv_sel,sel);
and U_anda (a_sel,a,inv_sel),
U_andb (b_sel,b,sel);
or U_or (y,a_sel,b_sel);

endmodule

```

Example 4-2: C17 Benchmark Circuit with Multiplexers added into the design

Instantiation of the Multiplexers

To implement any one of these errors at a time, an excitation signal must be sent from the main algorithm to an instantiated copy of a circuit, i.e. the circuit in Example 4-2. This is done through a series of bits with the width equaling the number of potential fault sites.

Example 4-3 shows how this is done. When a specific fault needs to be excited, the main algorithm will store a value of all 0s and one 1 into a vector, that shares its width with the number of fault sites. This vector is passed from the main algorithm to an instantiation of a circuit and the values are each sent to the select bit of the multiplexer it corresponds to.

	Mux10	Mux9	Mux8	Mux7	Mux6	Mux5	Mux4	Mux3	Mux2	Mux1	Mux0
C17-1	0	0	0	0	0	0	0	0	0	0	1
C17-2	0	0	0	0	0	0	0	0	0	1	0
C17-3	0	0	0	0	0	0	0	0	1	0	0

Example 4-3: Excitation table for three instantiations of C17.

In this way, all faults can be simulated individually or if need be in different combinations. As stated in previous chapters, it is not efficient to test numerous faults at once because of masking. If there are not enough instantiations to detect all faults at once, this process is cycled until all faults are detected.

Chapter 5: Algorithm For Testing

While performing this research there were a number of phases of implementation, assessment, and redesign. Essentially, there were three main phases where the algorithm concept was sound but the programming and implementation needed to be changed due to a bottleneck that plagued the speed of the algorithm. While the FPGA core itself was able to perform the fault detection algorithm quickly, we noticed a severe bottleneck in the transmission of the results for further processing. In this Chapter, we detail the different approaches taken, and describe the algorithms used for fault detection and test-set compression.

Approach 1: Hardware emulation with support software on host computer

For the first part of this research, the test pattern generation algorithm was in its most basic form. The FPGA was connected to a host computer that supplied the input stimulus to the circuit under test, and acted as a support machine to receive output data from the FPGA and do post-processing on the data. The basic configuration is shown below in Figure 5-1.

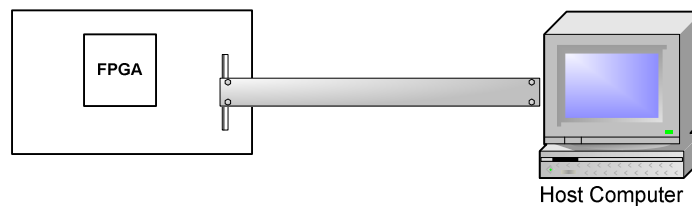


Figure 5-2. A block diagram of the FPGA emulation system supported through software running on a host computer.

The key steps of this approach are outlined below:

1. Implement as many instantiations of a benchmark circuit on the FPGA as the FPGA fabric will permit.
2. Insert a unique fault in each instantiation.
3. Generate random input vector on the host computer and, through a RS232 serial cable, send them to the FPGA board.
4. Apply the same random input vectors to the original copy and the fault inserted copies of the benchmark circuit
5. Send the outputs out from the FPGA to a hyperterminal connection on the host computer through the RS232 serial port.
6. Repeat steps 2-5 until all possible faults are detected with given number of random input vectors.
7. Save data to a text file
8. Using the computer, analyze the data for the best possible input vectors based on output data

After all of the benchmark circuit input vectors and fault detection output data was displayed on the hyperterminal display, a way of determining the most efficient input vectors was needed. To do this a program was written in Matlab that would determine which input could detect the most faults, and then eliminate those faults from further consideration.

This **vector compression algorithm** proceeds in the following steps:

1. Detect the vectors with the most faults
2. Log the vector
3. Eliminate faults found by vector from future consideration
4. Repeat steps 1-3 until all detected faults have been accounted for

11011000100101001	00000000000000001
11000111110000101	00000001000000001 ←Most faults detected
11111110111101100 ←Most faults detected	00000000000000000
11010100000011010	00000000000010010
(a)	(b)
00000000000000000	00000000000000000
00000000000000000	00000000000000000
00000000000000000	00000000000000000
00000000000010010 ←Most faults detected	00000000000000000
(c)	(d)

Figure 5-2: Fault detection algorithm (Rows are vectors, Columns are detected faults)

- (a)Original fault detection with first vector found (b) Second vector chose to cover most faults left (c)Third vector chosen to cover the rest of the faults (d) All faults have been detected

Consider the example shown in Figure 5-2(a). The example shows a circuit's fault and vector pair. The circuit had a total of 17 faults and was simulated with four random vectors. Figure 5-2(a) is the original list received directly through the RS232 port from the FPGA. Our goal is to find the minimum set of vectors that will detect all 17 faults. Vector #3 detects 13 out of the 17 faults - the largest number of faults in this example. Our algorithm compares the four vectors, detects the ability of Vector #3 to detect 13 faults and retains this vector in the **compressed test-set**. We then create a binary mask equal to the width of the faults. This mask is initialized with 0s in the location of the faults detected already by the vectors saved in the compressed test-set and 1s in other locations. After retaining Vector #3, the binary mask (000000001000010011) is put

through an AND gate with the fault-vector pair creating the pair shown in Figure 5-2(b). In Figure 5-2(b), Vectors #2 and #4 both detect the largest number of faults, and the first vector of the two is chosen automatically. A new mask is created and the process repeats until all detected faults are accounted for as shown in Figures 5-2(c) and (d). In this very trivial example, we started with four random vectors that were used to simulate the faults and our compressed test-set retained only three of those four vectors.

The main benefit of this approach that involved fault simulation in the FPGA and vector compression on the computer was mostly in the ease of use. Xilinx ISE was a familiar program and provided the infrastructure to simulate and implement the designs with ease. It also allowed for designs to be implemented in schematic form, an alternative to Verilog. This allowed for easy visualization of the modules and made the RS232 interface between the FPGA and computer easier to understand and implement. Matlab provided another simpler interface yet a powerful mathematical tool to perform the compression algorithm described earlier.

There were several issues with this approach to test pattern generation and compression. The first issue was that of the serial interface between the FPGA and the computer. This first implementation was designed and implemented using Xilinx ISE and thus did not have native RS232 serial port compatibility. To work around this, an existing sample Xilinx design was found that implemented an RS232 serial port. This design was then stripped down to just its serial port utility. Once this was determined to be working, data had to be formatted in groups of eight bits and sent to a relatively small buffer provided

by the original design. In addition to this, the timing of the entire circuit had to be monitored and controlled diligently so that data could be sent out correctly.

The second major issue with this design was the need to take data from the hyperterminal and in turn put it in a text file and then run the Matlab code with the text file of all the collected data. The constant saving and opening of large text files took extra time.

Although the compression algorithm was easy to write in MATLAB, the runtime was relatively slow because Matlab is an interpreted language and runs usually slower than compiled languages.

The next design would address these two issues by eliminating the need for a basic serial port design and the need to run external code to determine the best test vectors.

Approach 2: Hardware emulation with support software on PowerPC core within the FPGA

The second approach to an improved ATPG emulation on an FPGA was to use the embedded PowerPC CPU on the FPGA circuit. The PowerPC CPU is able to run code written in C and input and output data to the FPGA cores through the use of a dedicated data bus (described in Chapter 3). The data transmission over serial lines between the FPGA core and the host computer was no longer needed removing the serial communication bottleneck. At the same time, the compression algorithm could be written directly in C and run in the PowerPC so constant saving and opening of large text files in MATLAB was not needed eliminating the slow runtime experienced in Approach 1.

The use of the PowerPC core required the use of the Xilinx Platform Studio, a significantly more powerful utility for running different circuits on the Virtex II Pro board. This design suite came with preconfigured RS232 serial port which was significantly more versatile and user friendly.

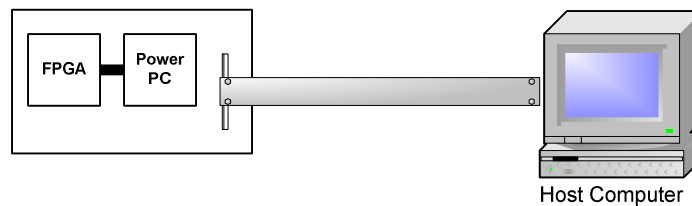


Figure 5-3: A block diagram of the FPGA emulation system and Power PC supported through software running on a host computer.

This new setup is shown in Figure 5-3. The host computer is primarily used to compile and download the code to the PowerPC and the FPGA core. The new approach can be summarized using the following steps:

1. Implement as many instantiations of a benchmark circuit on the FPGA as the FPGA fabric will permit
2. Insert a unique fault in each instantiation.
3. Randomly generate an input in the PowerPC and store in on-board RAM.
4. Send the input vector to the FPGA through the internal bus.
5. Run the original circuit and all instantiated faulty circuits using this random vector.
6. Send the outputs from the FPGA core back to the PowerPC to be stored in RAM
7. Repeat steps 1-6 for the number of vectors that are to be tested

8. Perform test-vector compression algorithm described in Approach 1, on the PowerPC and store the compressed result in RAM.
9. Send only the compressed test-set to the host computer using RS232 interface of the PowerPC and display the result using a hyperterminal.

With this new setup, the C code on the PowerPC sent randomly generated input vectors to the circuits created in the FPGA. The FPGA core with the fault detection algorithm then processed the input vectors and sent back the detected faults. All of the inputs and faults were stored in a DDR RAM module present in the FPGA board. The DDR RAM module was utilized because local BRAM was too small to hold all the data that the Power PC needed to store. Finally, the PowerPC ran the algorithm for deciding the most efficient inputs and then using the RS232 serial port, it outputted in an easier to read format which inputs would cover the most faults.

There were significant issues with this design as well. Despite the speed of the FPGA fault detection algorithm, there was a significant bottleneck with the Power PC's speed when determining the best input vectors. With any large circuit, the algorithm for determining the best inputs took minutes to run. Because this circuit needed to access external RAM and was burdened with overhead processing, it could not process the sheer amount of data needed to in a timely manner. Another revision was made to eliminate these issues.

Approach 3: Hardware emulation with built-in random vector generator

The culmination of previous experiences led to the final incarnation of this emulation algorithm and implementation technique. In the final rendition of the algorithm, most functionality is transferred to the implemented FPGA core so that hardware speeds can be effectively utilized. This required a re-thinking of how the core would work and how data would be processed. The inputs and outputs were also drastically changed and the need for DDR RAM was eliminated. The core itself runs as a finite state machine which is shown in Figure 5-4. As seen in flowchart in Figure 5-5, the algorithm goes through a number of decisions and processes that are repeated several times. Each of these will be described in detail in the following paragraphs.

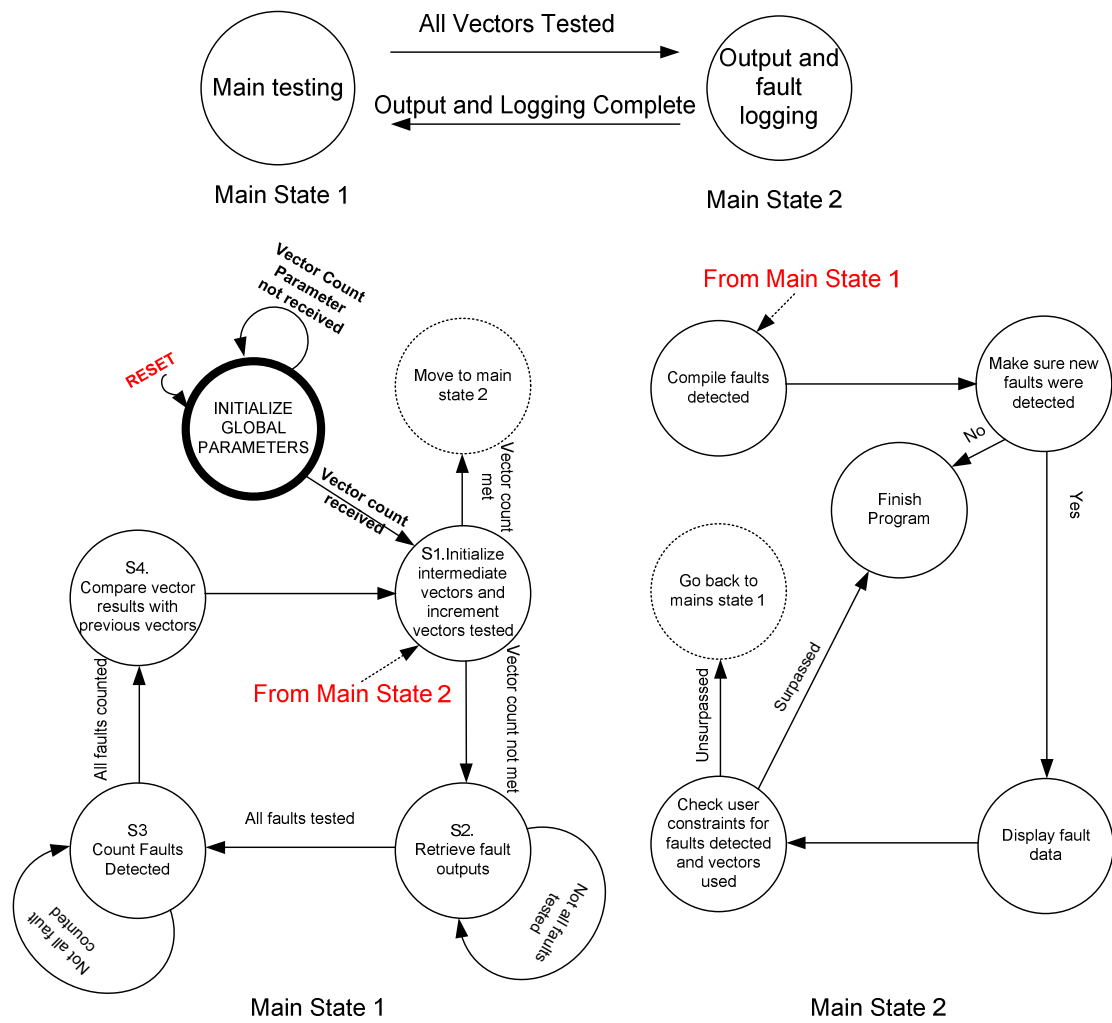


Figure 5-4: State diagram of the fault detection algorithm.

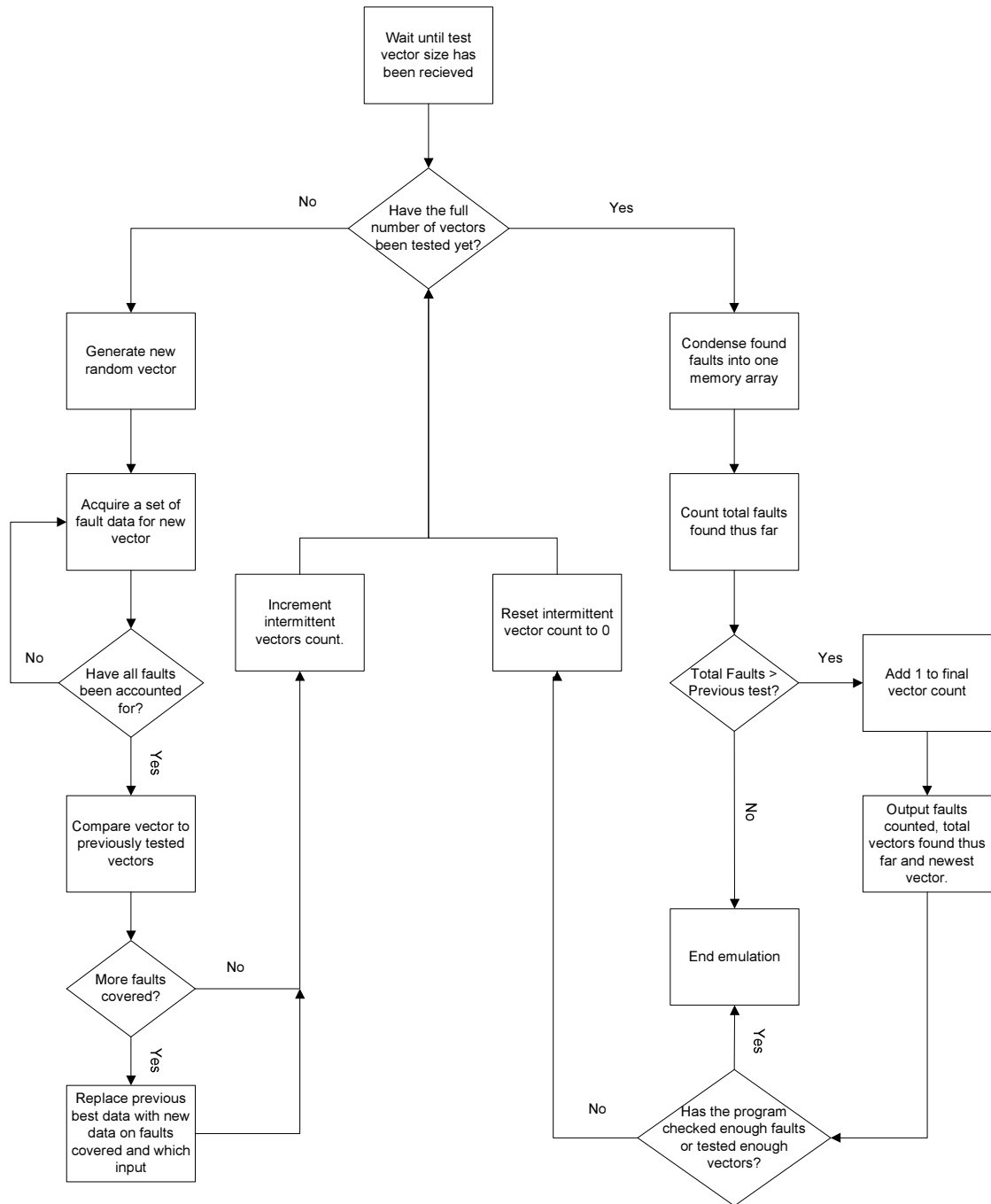


Figure 5-5. Flowchart of emulation algorithm

The core of this program is run on the FPGA. The FPGA core contains the good version of the circuit and several faulty instantiations. After the circuits have been instantiated in the core, the core waits for the user to specify the number of randomly generated test vectors to examine. This user generated number is received from the PowerPC core. Until this number is received, the FPGA core containing the fault emulation hardware simply remains in the idle state.

Once this user generated number is received, the program starts with its initializations in **Main State 1** of Figure 5-4. This is shown as the first decision box in the flowchart of Figure 5-5. If the user specified number has not been reached, a new test vector is generated with the use of a linear feedback shift register (LFSR). A number of intermediate variables (e.g. total faults count) are reset to zero, and the threshold counter is incremented by one. Once this is complete the state is changed from S1 to S2.

An LFSR is a circuit that is used for the generation of pseudo-random vectors. The width of the LFSR is the width of the number of inputs to the circuit. Figure 5-6 shows an example of a 4-bit LFSR. The four-bit shift register with feedbacks from Q1 and Q2 as shown below produces a four-bit sequence (seemingly in random fashion) as long as the register is initialized in a sequence other than 0000 (An XNOR gate can be used instead of the XOR gate, but the initialization has to be something other than 1111). Since this is a four-bit register, the sequences will repeat after the 15th clock cycle. The initialization sequence is called the seed, and as can be seen, the newest state of the register is a linear function of its previous state. Depending on the seed, the values will follow a seemingly

random pattern until all possible values are extinguished and the original value is the current state again.

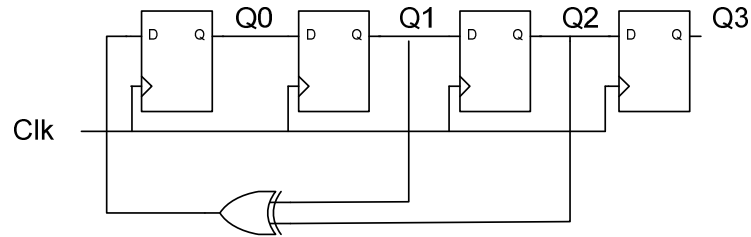


Figure 5-6: A 4-bit LFSR with taps at bits 3 and 4

The algorithm is summarized in the following steps:

1. Define constants for running the algorithm
2. Instantiate as many possible copies of fault injected benchmark circuits and one clean circuit into the FPGA
3. Run the algorithm which has several steps and sub-steps:
 - a. Initialize all intermediary values and generate new input test vectors with the use of an LFSR.
 - b. WHILE all fault locations have NOT been tested:
 - i. Insert SA0 faults into the circuit.
 - ii. Compare outputs of fault injected circuits and clean circuit. Record any discrepancy into memory location as *fault data*.
 - iii. Change fault signal from an SA0 to an SA1 by tying the ERROR line from Logic 0 to Logic 1.
 - iv. Repeat Step ii for SA1 fault.

- v. Change fault locations by increasing specific registers and go back to step i.
 - c. Count all NEW faults detected with the new input vector.
 - d. If the faults detected thus far number larger than previous data:
 - i. Overwrite stored previous best input vector with current input vector.
 - ii. Overwrite previous best fault data with current fault data.
 - iii. Overwrite previous best detected faults total with current detected faults total.
 - e. Repeat steps a-d until the user defined number of vectors has been tested.
 - f. When this is done, the program moves into main state 2 of Figure 3 with the following sub-steps
 - i. Add the newly detected faults to currently saved faults detected list.
 - ii. Detect if any new faults were actually generated.
 - iii. If new faults were found, output new data as seen in Table 1.
 - iv. Re-initialize data and move back to main state 1 of Figure 3 (i.e. Step 3 (a)).
- 4. Repeat step 3 until all possible faults found are detected or another user-defined trigger is flagged.

Input Vector	Total Faults Detected	Total vectors used	Time(1/100 sec)
F12BCD	123	1	20
A3DAF8	205	2	40
B98321C	267	3	61
C18EB24	301	4	81

Table 5-1: Example Output for Step 3-f-iii. The Input vector is in Hexadecimal notation.

Table 5-1 gives a typical display of how the output from the FPGA after being processed and sent to the computer would look like. Each line represents step 3f being run one time. First you have the newest useful input vector in a hexadecimal format. Next is the cumulative total of faults that have been detected so far. After that is the cumulative number of input vectors that detect the fault recorded thus far. Finally the time is cumulative so in Figure 5-4 each step takes about 20ms.

Part b of step 3 is where all of the fault detection takes place. This state runs through four different sub-states. The first sub-state is where the fault data from the instantiated circuits is recorded in a predefined memory array (step 3-b-i). Because this array may change in size dependant on a user, the number of times data must be recorded is variable.

The second state is where the faults that are being simulated are switched from an SA0 to an SA1 (step 3-b-ii). This transition can be seen in the change from Figure 5-7a to 5-7b or Figure 5-7c to 5-7d. Here simulated fault bit is being changed from an logic low to a logic high. Technically this fault signal is sent to every fault site, but only one fault site is active at a time.

After this is completed the data is recorded again (step 3-b-ii). The positions of the faults that are being tested are then changed along with a re-initialization of the fault type back to SA0 (step 3-b-iv). The difference between an active fault and an inactive fault is the select signal. To change the fault site, the select signal to fault 1 changed from logic high to logic low, and the fault 2 site has its select signal changed from logic low to logic high. Also the simulated fault line goes back to a logic low from logic high. This can also be seen in the transition from Figure 5-7b to 5-7c. This process is repeated until all possible fault locations have been accounted for.

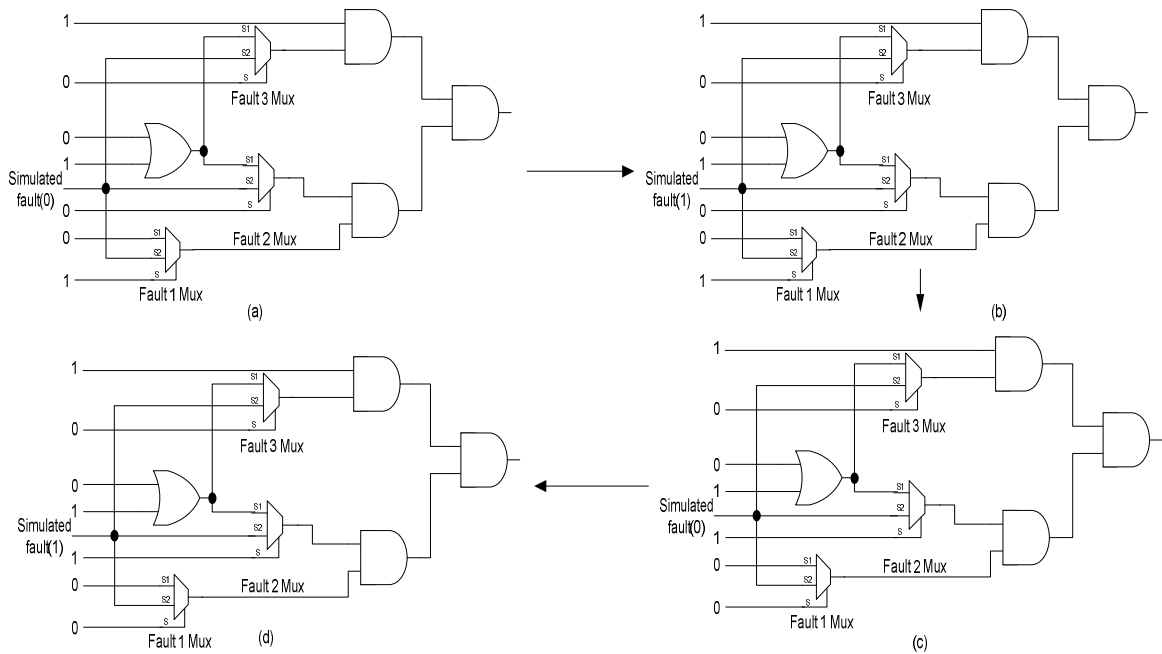


Figure 5-7: Fault changing process

- (a) fault one activated with SA0 (b) fault one activated with SA1
(c) fault two activated with SA0 (d) fault two activated with SA1

The third state of main state 1, or part c of step 3, is this process involved with counting all of the faults that were detected with a test vector. This is done in a series of loops. Each loop starts with putting the first memory cell of the new data and the first memory cell of the stored data through a bit-wise OR gate and storing it in a temporary vector. This temporary vector holds the potential faults that could be detected for certain fault locations with the currently tested input vector and can be seen in Figure 5-8.

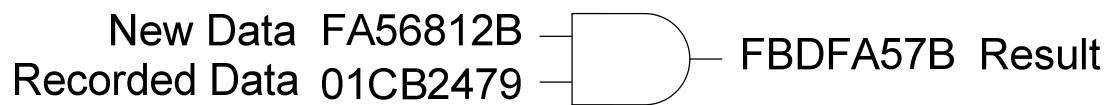


Figure 5-8: Combining new test vector data with recorded data. All test vectors are displayed here in Hexadecimal notation.

In Figure 5-8, the new data is the series of bits that was taken from comparison of the clean benchmark circuit and the fault inserted circuit. The recorded data is the series of bits that accounts for all faults that have been previously detected. As stated above the result represents potential data to be recorded in a later state. Every memory cell goes through this until all data is accounted for. To reduce the number of clock cycles needed to run this, the fault testing data is reset at the end of this state as well.

For the fourth state, or part d of step 3, a comparison of previously recorded data is used to determine which the best vector is. When the algorithm runs the first time, the first vector is recorded and then subsequent vectors are compared and replace previously stored vectors if they can cover more faults. When the new vector is stored, the faults it can cover are stored as well for a comparison against subsequent test vectors.

Main state 2 is the second main state of this algorithm, represented as part f of step 3.

When the user-defined number of vectors has been tested, main state 2 is initialized.

Within this state are a number of sub-states. The first of these sub-states combines the fault coverage of the best vector recorded thus far in a manner similar to Figure 7, but this time it is permanent. After this is complete the next state determines how many faults have been covered by the recorded vectors until this point using data calculated in the third main state. If this number is not greater than previous iterations, then the algorithm finishes and outputs a finished signal. If the number is greater, the data regarding this vector is outputted to the hyperterminal. This data includes the vector, how many faults have been detected so far, and how many vectors have been recorded. The final sub-state checks the number of faults detected and vectors tested. If either of these numbers is greater than the user-defined limits the program will terminate at this point, otherwise the entire process begins again to find another vector.

With this new incarnation of the fault detection algorithm there are several improvements over the previous two approaches. First there is a major increase in speed because all processing is done in hardware and not on the PowerPC. There is also an increase in simplicity for the interaction between the FPGA and PowerPC. The algorithm that controls the FIFO data transfers only needs to worry about one vector of data being sent to the FPGA. Every other data transfer is from the FPGA to the PowerPC. Because of this, after the FIFO control sends the vector of data from the PowerPC to the FPGA, it can be put in an infinite loop of waiting for data from the FPGA to be sent to the

PowerPC. Because of the primary use of the FPGA, no RAM is needed and simplifies the design on the PowerPC end by a significant amount.

Although there are several benefits to this design, there were several drawbacks that were unavoidable. Because this design is more centered on the FPGA, synthesis time had to be dealt with when testing. There was also an added level of complexity that is associated with transferring code designed for a CPU and HDL code designed for an FPGA. Due to the size of the design, simulation was nearly impossible because it took longer to process data than actually synthesis.

Chapter 6: Results

In order to verify the validity of our test-set generation algorithm, we used some circuits from the ISCAS and the MCNC benchmark suites. All experiments were run on the Xilinx Virtex-II Pro Development System. The board houses a Xilinx XC2VP30 FPGA with 30,816 Logic Cells, 136 18-bit multipliers, 2,448Kb of block RAM, and two PowerPC Processor cores. The design was synthesized using the Xilinx ISE Design Suite 10.1.

The verilog code generated from our C++ script, was tested in a behavioral simulator to verify its correctness and to get a general idea of how long the test would take when implemented in hardware. Once this was completed, the algorithm was implemented in an actual FPGA. This chapter will discuss the result of both the simulation and implementation.

Simulation Results:

For this research, simulation was used as a step between design and implementation. Simulation was used to verify the output of the FPGA . While simulation of the Verilog code was able to display all internal signals, allowing for better debugging, the time taken to complete was orders of magnitude larger than implementation. This bottleneck limited the circuit size being simulated to smaller benchmarks circuits with a few hundred gates.

C432 Emulation				C432 Simulation			
Input Vector	Faults Detected	Input Vector Number	Time(ns) – 100 MHz	Input Vector	Faults Detected	Input Vector Number	Time(ns) – 1 GHz
9784DC851	84	1	7673140	9784DC851	84	1	116150
DC62AA93B	147	2	12381300	DC62AA93B	147	2	232190
A08F52AEE	187	3	17250780	A08F52AEE	187	3	348230
97227207C	220	4	22120740	97227207C	220	4	464270
7041256E3	253	5	26987740	7041256E3	253	5	580310
2BEF86068	286	6	31855220	2BEF86068	286	6	696350
53326F5A2	314	7	36722700	53326F5A2	314	7	812390
146C583D1	341	8	41590180	146C583D1	341	8	928430
7207C84B2	367	9	46455180	7207C84B2	367	9	1044470
8B2E44E40	389	10	51330020	8B2E44E40	389	10	1160510
585C10495	406	11	56204860	585C10495	406	11	1276550
BCF94BE5E	421	12	61079700	BCF94BE5E	421	12	1392590
62AA93BC9	435	13	65954540	62AA93BC9	435	13	1508630
55047A957	448	14	70829380	55047A957	448	14	1624670
7CBCED4CC	459	15	75704220	7CBCED4CC	459	15	1740710
F55CF7708	468	16	80579060	F55CF7708	468	16	1856750
ED9784DC8	475	17	85451420	ED9784DC8	475	17	1972790
5916D060C	481	18	90323780	5916D060C	481	18	2088830
4ED9784DC	487	19	95196140	4ED9784DC	487	19	2204870
9BCF94BE5	493	20	100068500	9BCF94BE5	493	20	2320910
2F979DA99	498	21	104950700	2F979DA99	498	21	2436950
BBB93C46C	503	22	109832900	BBB93C46C	503	22	2552990
94C145EAB	507	23	114715100	94C145EAB	507	23	2669030
92F16F482	510	24	119597300	92F16F482	510	24	2785070
CB913903E	513	25	124479500	CB913903E	513	25	2901110
3E52F979D	516	26	129361700	3E52F979D	516	26	3017150
AC978B7A4	518	27	134243900	AC978B7A4	518	27	3133190
90964988B	520	28	139125620	90964988B	520	28	3249230
5C10495B8	522	29	144007820	5C10495B8	522	29	3365270
7EC8B5AA9	524	30	148890020	7EC8B5AA9	524	30	3481310
97CBCED4C	526	31	153772220	97CBCED4C	526	31	3597350
4580917C7	528	32	158654420	4580917C7	528	32	3713390
592F16F48	529	33	163536620	592F16F48	529	33	3829430
E44E40F90	530	34	168416340	E44E40F90	530	34	3945470
F212C9311	531	35	173297860	F212C9311	531	35	4061510
45916D060	532	36	178179380	45916D060	532	36	4177550
C8B683067	533	37	183060900	C8B683067	533	37	4293590

95D617041	534	38	187942420	95D617041	534	38	4409630
54D282BCC	535	39	192823940	54D282BCC	535	39	4525670
D282BCCC0	536	40	197705460	D282BCCC0	536	40	4641710

Table 6-1: Results from behavioral simulation and hardware emulation for C432 benchmark circuit.

As an example, the results of the hardware emulation and functional simulation of the circuit C432 are presented in Table 6-1. The circuit is a 27-channel interrupt controller consisting of 160 logic gates, 36 inputs and 7 outputs. Both simulation and emulation are done with the exact same Verilog netlist generated from the C++ code. The C++ netlist generation code took 0.734 seconds to run. The generation time varies based on the size of the circuit netlist and is presented in Table 6-5 for every benchmark circuit tested. Table 6-1 shows that the best input vectors (shown in the table in hexadecimal format), number of faults detected and the number of useful input vectors are exactly the same in both the emulation and simulation. This agreement verifies the functional correctness of the hardware implementation of the fault detection algorithm. There is however a difference in the times reported for the simulation and hardware emulation. During simulation, the Verilog simulation software interacts with a test bench file differently than how the PowerPC and the data bus interact during hardware emulation. The simulation hence underestimates the time taken for hardware emulation by an order of magnitude.

Implementation Results:

Although simulation can help determine whether or not a set of Verilog coded algorithms could work, they do not necessarily translate to hardware correctly. The final goal of this research was to implement a series of benchmark circuits in a Xilinx Virtex II Pro FPGA and record the final outputs given a number of different stimuli.

In this section, we present the results for circuit C432 in detail and summarize the results we obtained for the other benchmark circuits. Details of each circuit and extended results can be found in Appendices 3-5.

Results for Benchmark Circuit C432

Instantiations	Time(sec)	Slices %	Slice FF %	Slice LUTs %
1	23.56	27	9	23
2	11.81	24	8	23
4	5.93	23	8	22
8	3	23	8	22
16	1.53	24	8	23
32	0.84	29	8	28
64	0.49	38	9	36

Table 6-2: Area and timing results for C432 circuit simulated with 10000 vectors and variable instantiations running at clock frequency of 25MHz. 537 faults were detected with 36 input vectors

For the first set of data, Table 6-2 shows how increasing the number of instantiated faulty circuits can affect the time it takes for the algorithm to run. For every 2^n number of instantiations, where n is increasing, the number of vectors needed and faults found remained static. This was somewhat expected because, increasing the instantiation allows different faults to be detected in parallel without really changing which input vectors are

needed for the detection of the faults. From column 2 of Table 6-2, we see that as the number of C432 instances are increased, the time required to find all detectable faults decreased inversely to the number of faulty C432 instantiations ($1/x$). This logarithmic relationship is clear in Figure 6-1.

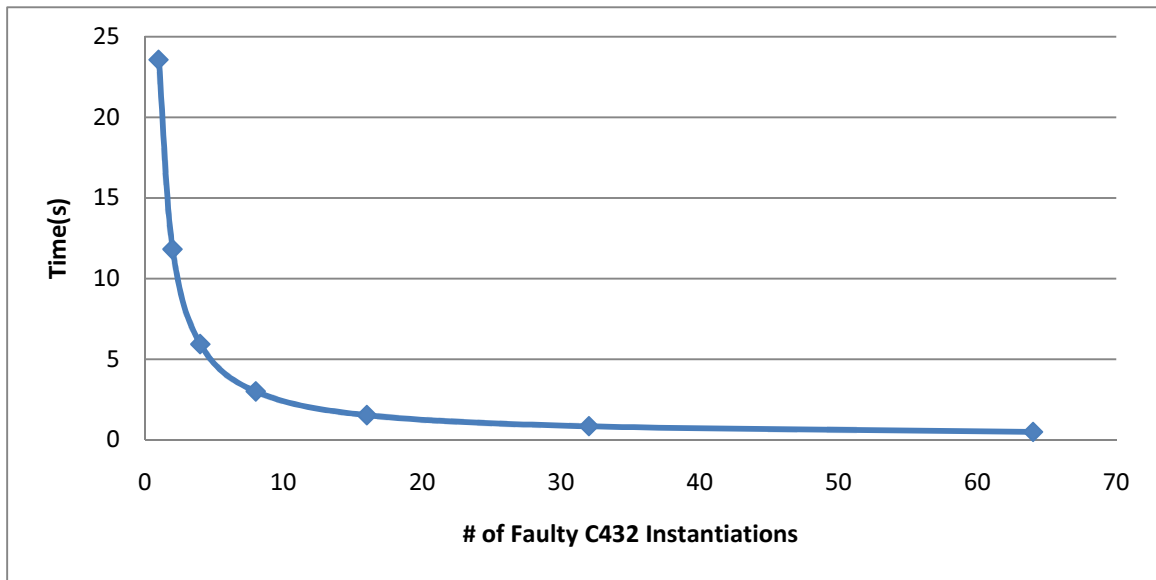


Figure 6-1. Total time taken for test pattern generation using random vectors for C432 as a function of the number of instantiations.

By observing the data in Table 6-1 and Figure 6-1, it can be seen that time decreases logarithmically with the number of instantiations. For every increase in the number of instantiations by two times the previous test, the time halves as well. This shows how the scalability of this can be very beneficial to larger circuits where the fault testing with one fault at a time would have an extremely long runtime.

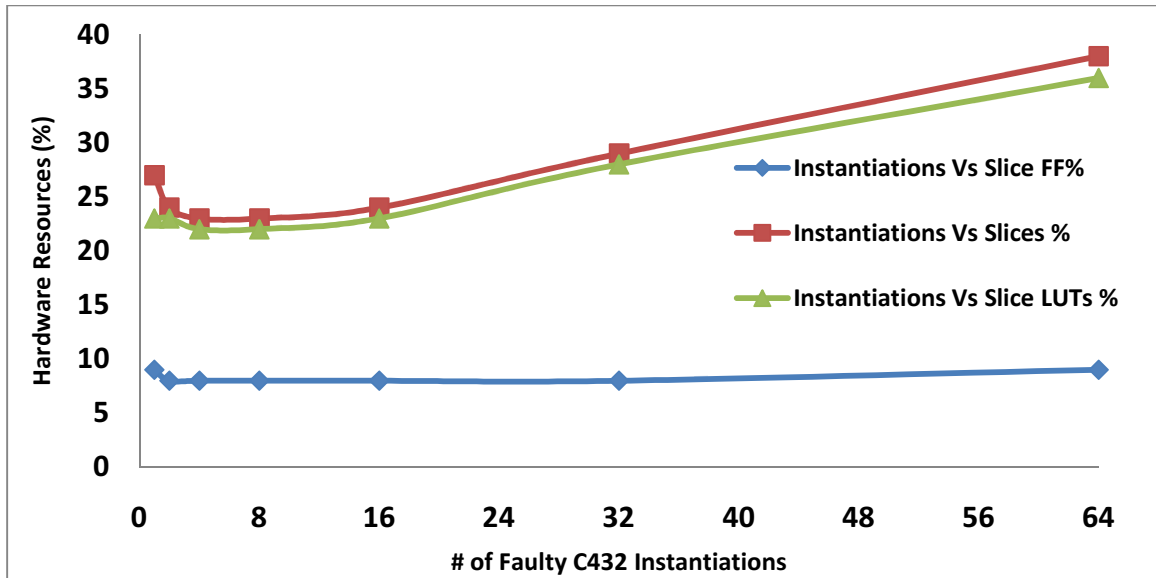


Figure 6-2: Total FPGA hardware resources used for test pattern generation using random vectors for C432 as a function of the number of instantiations.

In Figure 6-2, we also compare the hardware resource usage on the FPGA as a function of the number of instantiations. The primary hardware resources on the FPGA fabric are the Look-up-tables (LUTs), the slices and the slice registers. It can be seen that the Look-up-table hardware and the slices go up as more copies of C432 are instantiated in the fabric. We see that the slice registers are almost constant at around 8-9%. However, when only one module was instantiated, we found that the number of slices and the LUTs are higher than that when four modules are instantiated. This is probably because of the synthesis tool's resource sharing optimization routines.

Using 32 instantiations of the C432 circuit and a varying number of random input vectors, we next ran experiments to determine how many faults were found, the number of vectors needed to find these faults, and the time it took to process them. C432 has

68719476736 (2^{36}) possible input vectors. Given the rate at which the time increases for different amounts of tested vectors, seen below in Column 4 of Table 6-3, to test every input vector of C432 would take approximately 60.90 days. This is an unacceptable amount of time. Thus the goal of this experiment was to find exactly how many random input vectors needed to be tested to obtain a reliable result.

Vectors Tested	Faults Found	Number of Vectors Needed	Time (s)
10	258	10	0.04
20	374	19	0.09
40	405	23	0.11
80	460	33	0.16
160	496	40	0.19
320	526	46	0.22
640	536	47	0.23
1280	536	43	0.21
2560	537	39	0.23
5120	537	36	0.43
10240	537	36	0.86
20480	537	35	1.66
40960	537	35	3.33
81920	537	33	6.27
163840	537	32	12.16
327680	537	33	25.09

Table 6-3: Results from C432, static number of instantiations, variable vector tests

The number of vectors to be tested for each iteration of the test was entered manually.

The initial number was chosen so that a significant range of vectors could be tested given the maximum number of possible input vectors. From here the next iteration the number of vectors was doubled and the experiment was run again. From Table 6-3, it is clear that for very small input test set (from 10-160 vectors), a significant number of faults are left undetected. However, once 2560 vectors are tested, all 537 detectable faults in the circuit

are captured. There is no increase in detection even when 327680 vectors are tested as can be seen in Figure 6-3. Figure 3 clearly shows the stability point around 2560 input vectors. This stability point is based around the number of faults found and the number of vectors that are needed to detect them. This confirms that only a subset of input vectors must be tested to represent all possible input vectors.

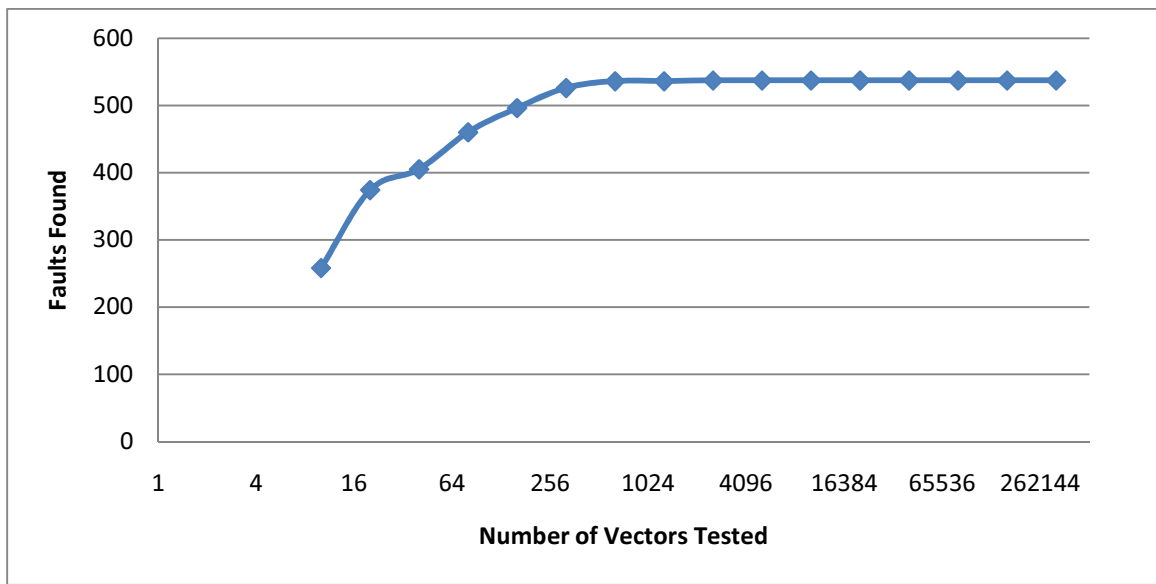


Figure 6-3. Plot of number of vectors tested vs. faults found for 32 instantiations of C432.

The circuit C432 has 277 checkpoint locations and this means there are 544 possible faults. Table 6-3 shows that 537 total faults were found; that leaves 7 faults unaccounted for. This is a 98.7% fault detection rate which is generally considered an acceptable detection rate. Upon further investigation, we found that these 7 missing faults were actually *redundant faults*. A redundant fault is defined as an untestable fault in a combinational circuit that does not cause any change in the input/output logic function of the circuit [2]. It is common for circuit designer to add redundancy in the design to counteract glitches due to logic hazards. These redundant logic

structures do not change the overall function of the circuit – they merely allow the circuit output to remain stable and glitch free when different circuit paths switch with different delay.

Table 6-3 also shows that these 537 faults are found in 0.23 seconds which leads to another observation. Consider Figure 6-4, which shows a plot of the total time taken as a function of the Number of Vectors tested. There is very little change in the time due to the number of vectors, until approximately 4096 vectors are tested. The PowerPC directive that outputs data takes significant amount of time to run and masks the time of the fault detection algorithm. It is not until a significant number of vectors are tested, that the algorithm's speed begins to slow down into a detectable range. Once the number of vectors being tested fall into this range, the increase in time follows the number of vectors being tested in a linear fashion.

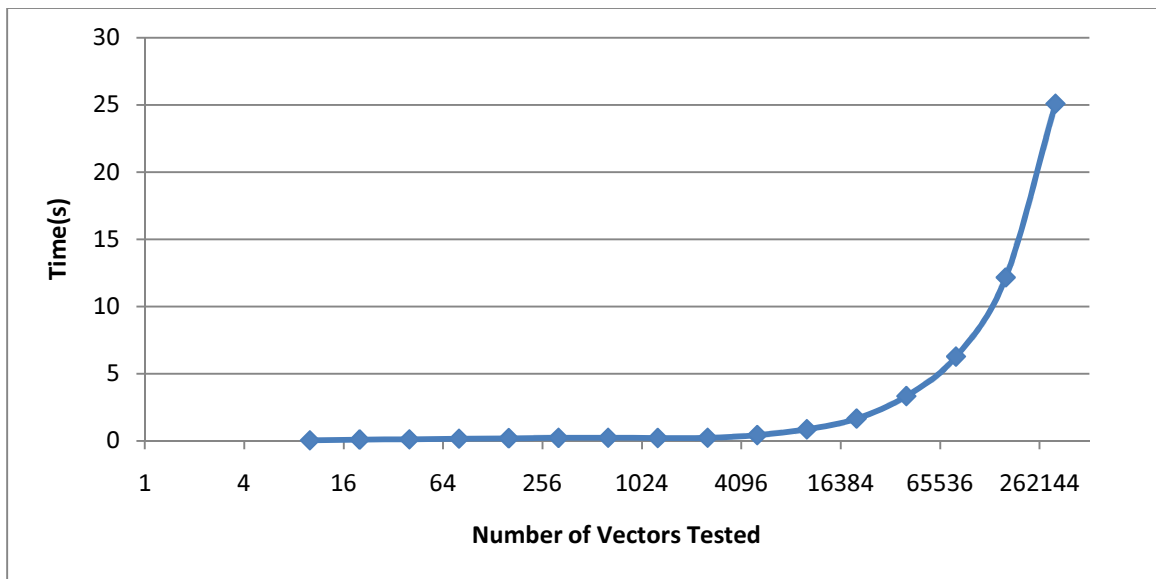


Figure 6-4. Plot of total time taken vs. number of vectors tested for 32 instantiations of C432.

The other fault detection stability point is the number of vectors that are needed to test a circuit. Figure 6-5 shows how when the number of vectors tested increases, so does the

final test set size until a certain point. This point is where the maximum number of faults was found in Figure 6-3 and Table 6-3. After this point no new faults are to be detected, but by testing more vectors, it is possible to decrease the number of input vectors needed obtain the maximum fault coverage. Although this test can be run for all possible inputs, this is not an efficient testing method and should be avoided. From Figure 6-5 it can be seen that at around 4096 tested vectors the number of useful vectors begins to stabilize in range of 32-34.

In this graph there is an interesting change in the number of useful vectors at 163840 tested vectors. At this point there is a local minimum of 32 useful vectors and it is surrounded by 33 useful vectors on both sides. This can be explained by maximum fault-minimum vector algorithm that is described in chapter 5. Without the use of advanced heuristic techniques, which would slow down this algorithm, it is possible that one useful vector may mask another vector that would help reduce the final number of useful vectors. This is one trade-off this algorithm takes for the advantage of increased processing speed.

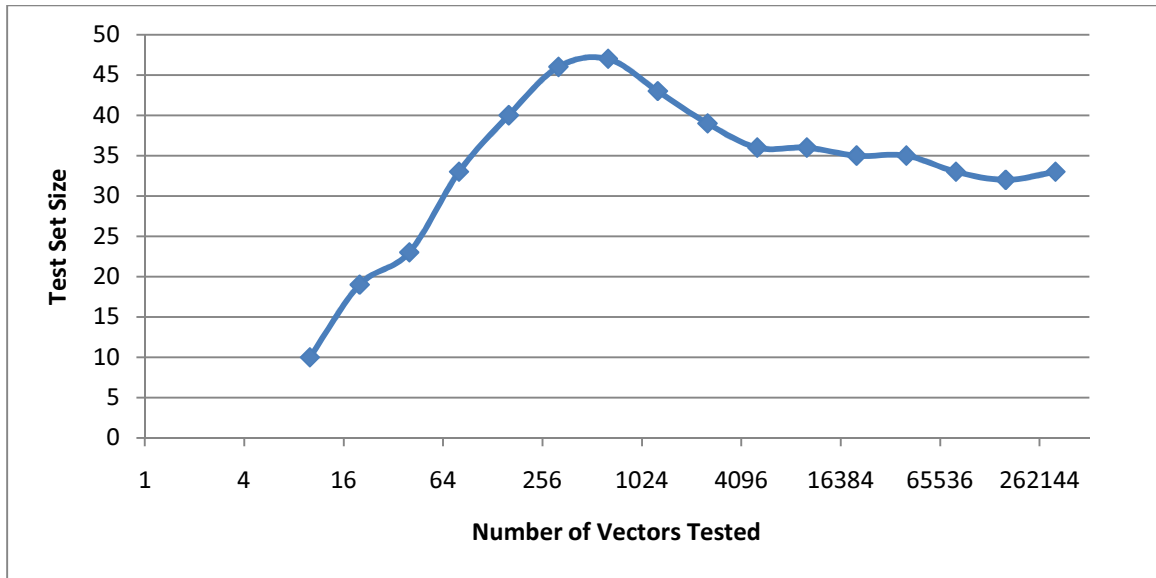


Figure 6-5. Graph of number of vectors tested vs. test set size

Outlying Data

In a number of other circuits we found some variation from the trend we saw for C432.

We found variations usually in circuits that were either significantly smaller (eg. Z9sym from MCNC benchmark) or significantly larger (eg. C1355 and C880 from ISACS) than C432. In both C1355 and C808, due to their large size, when 64 instantiations were implemented, the standard 25MHz clock speed could not be met. To fix this, the speed was dropped to 12.5MHz, and both circuits were implemented without error. We found that certain area constraints are due to timing requirements. The FPGA uses more LUTs to overcome certain timing issues and as a result the design becomes too large to fit on the device. By dropping the speed the number of LUTs implemented drops.

The issue with dropping the speed is that the time improvement due to the number of instantiations was negated by the drop in clock speed. In a larger FPGA such as the

Virtex V, this would not be an issue because there are more resources, including the timing dependent LUTs. For simplicity the speed was dropped by one half, which does not need to be done for professional use.

Another issue that varies from circuit to circuit is the stability point. Certain circuits such as C1355 have a stability point that is much more apparent, without fluctuations (similar to the plot for C432 shown in Figure 6-3). On the other hand there are circuits that seem to never reach a stability point. This is due to the small size of the circuit and the smaller number of inputs. Circuits like Z5xp1 and z9sym have 7 and 9 inputs respectively restricting the total number of input vectors that can be tested to 128 and 512. For both these circuits, we found some faults that were difficult to observe at the output and required the full input set to be tested. This can be seen in Figure 6-6 which shows the plot of the vectors tested versus the vectors need for circuit z9sym. As can be seen in Figure 6-6, we were able to find faults up until all possible vectors are tested.

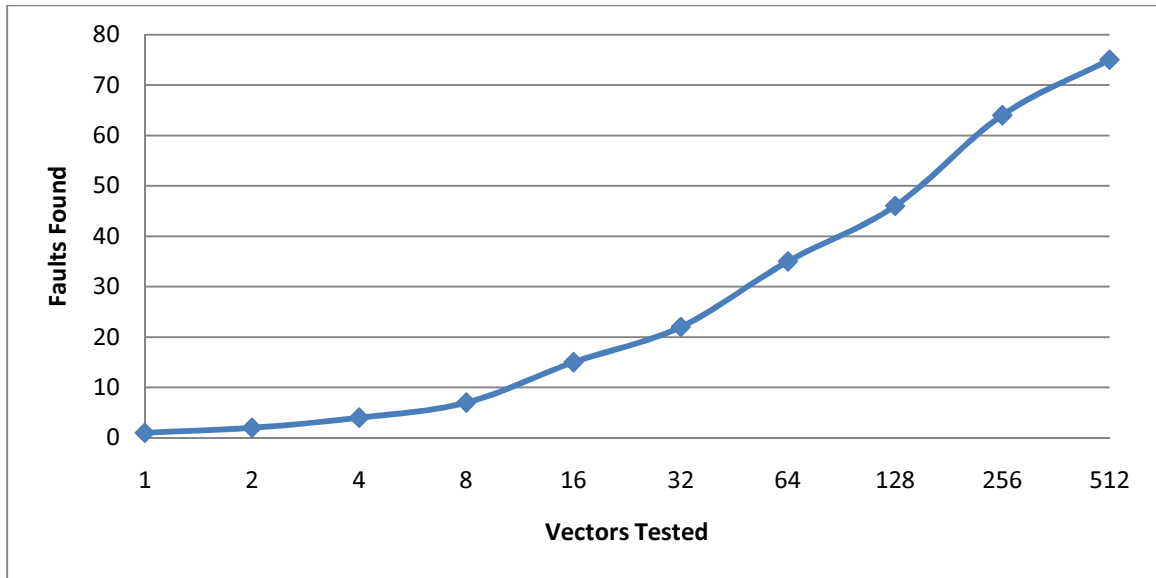


Figure 6-6. Graph of vectors tested vs. vectors needed for z9sym

The last piece of interesting data to be found in the appendices is that some times do not decrease logarithmically with the number of vectors or instantiations tested. As was stated in the explanation of the data for C432, when an algorithm's time is below a certain threshold the output directive in the PowerPC takes longer and masks the algorithm's timing. For the smaller circuits like C17, z9sym, and Z5xp1 this is apparent through all or most of their instantiation or vector tests. In the graph Figure 6-7 below, the line at the bottom is where the speed of the PowerPC output directive started to mask the speed of the FPGA fault detection algorithm. Despite increases in the number of instantiations, the time remained at 0.39 seconds because the number of useful input vectors remained the same.

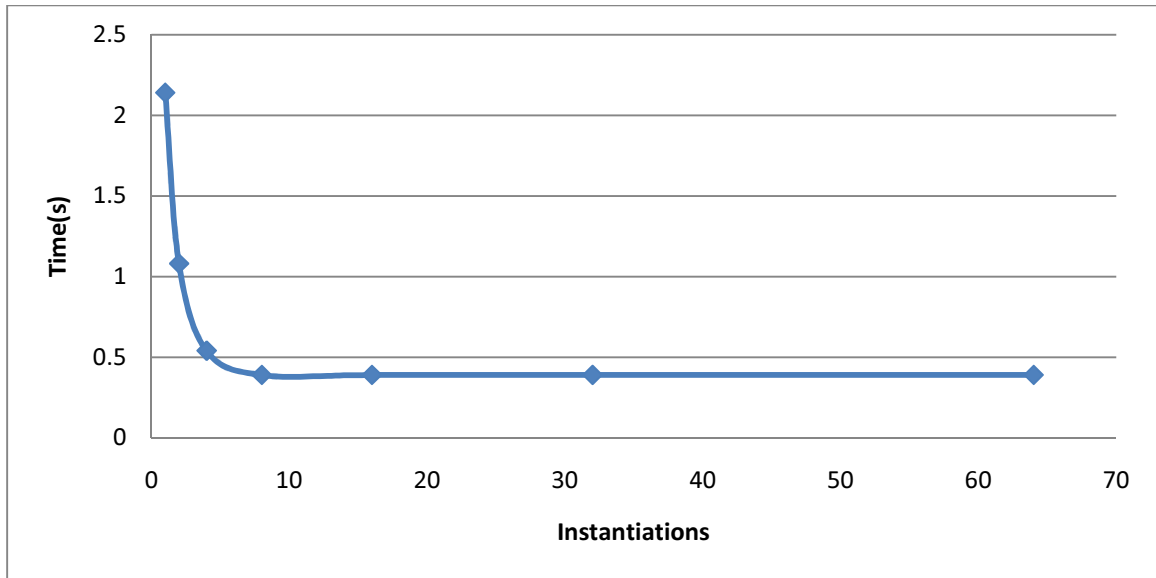


Figure 6-7. Graph of Instantiations vs. Time for z9sym

Summary of Results for the ISCAS and MCNC benchmark circuits

In this section, we summarize the result of our algorithm applied to circuits from the ISCAS and MCNC benchmark suites. The benchmark circuits have the following characteristics:

Circuit	Inputs	Outputs	# of Gates	Circuit Depth	Function	Benchmark Suite
c1355	41	32	546	23	32-bit Error Correcting circuit	ISCAS
c17	5	2	6	3	Simple NAND chain	ISCAS
c432	36	7	160	29	27-channel interrupt controller	ISCAS
c499	41	32	202	23	32-bit Error Correcting circuit	ISCAS
c880	60	26	383	23	8-bit Arithmetic Logic Unit	ISCAS
Clip	9	5	144	11	Unknown	MCNC
rd73	7	3	148	13	Unknown	MCNC
t481	16	1	74	14	Unknown	MCNC
Z5xp1	7	10	166	10	Unknown	MCNC
z9sym	9	1	147	12	Unknown	MCNC

Table 6-4. Description of Benchmark circuits

As a summary of the results that were found, Table 6-5 describes the best possible outcomes dependent on three factors.

1. Largest number of faults detected
2. Smallest number of vectors to detect 1.
3. Shortest time to detect previous two factors

The first criterion for deciding the best data was the fault detection percentage. This is the most important factor because missing potential stuck-at-faults can cost a manufacturer money, especially if an error systematically occurs in an undetected region.

The next criterion was the size of the test set. Smaller test sets mean faster chip verification and saves money for manufacturers. Although the time for running this test may be longer, as is the case for C880, reducing the test set will be beneficial during manufacturing where the same circuit verifications will be run numerous times. Once these criteria are met, the best data was organized based on the time it took to perform the fault detection.

Circuit	Vectors Tested	Vectors Needed	Maximum number of faults	Faults Found	Fault Coverage	Time Taken (s)	Verilog Code Generation Time	Maximum Number of vectors
c1355	5120	84	1618	1610	0.995	2.75	5.031	2.19902E+12
c17	8	6	22	22	1	0.0329	0.14	32
c432	163840	32	544	537	0.987	12.16	0.734	68719476736
c499	1280	52	594	586	0.986	0.34	0.938	2.19902E+12
c880	327680	25	994	994	1	32.77	2.563	1.15292E+18
Clip	512	48	428	415	0.969	0.24	0.453	512
rd73	128	72	404	404	1	0.367	0.438	128
t481	2048	33	194	194	1	0.17	0.188	65536
Z5xpl	128	46	476	470	0.987	0.23	0.578	128
z9sym	512	75	462	410	0.887	0.39	0.531	512

Table 6-5: Best fault detection based on faults found, vectors used, and time (done with 32 instantiations)

Table 6-5 shows the results for the total number of random vectors generated, total number of vectors in the compressed set, the maximum number of faults in the circuit and the number detected. It also reports the total time taken for vector generation and the total time taken by our C++ code to recognize the checkpoints, insert faults and generate a new Verilog netlist. Some benchmark circuits (e.g. C432 and C880) were run longer even though all possible detectable faults were found with less vectors. Having more vectors enabled us to do the final compression more efficiently. As seen in Figure 6-8, the fault coverage of our approach is excellent with most circuits showing a coverage well over 97%. In all circuits, we are able to detect all non-redundant faults. The lowest coverage was for *Z9sym* at a rate of 88.7%. No additional faults can be detected for *Z9sym* because all possible vectors were tested. We found that the coverage of the collapsed fault-set for this circuit using Mentor's FastScan also provides only a 87.5% coverage. The less than

100% coverage can be attributed to redundant faults and masking of some faults due to the use of dominance relationships as stated in Chapter 4.

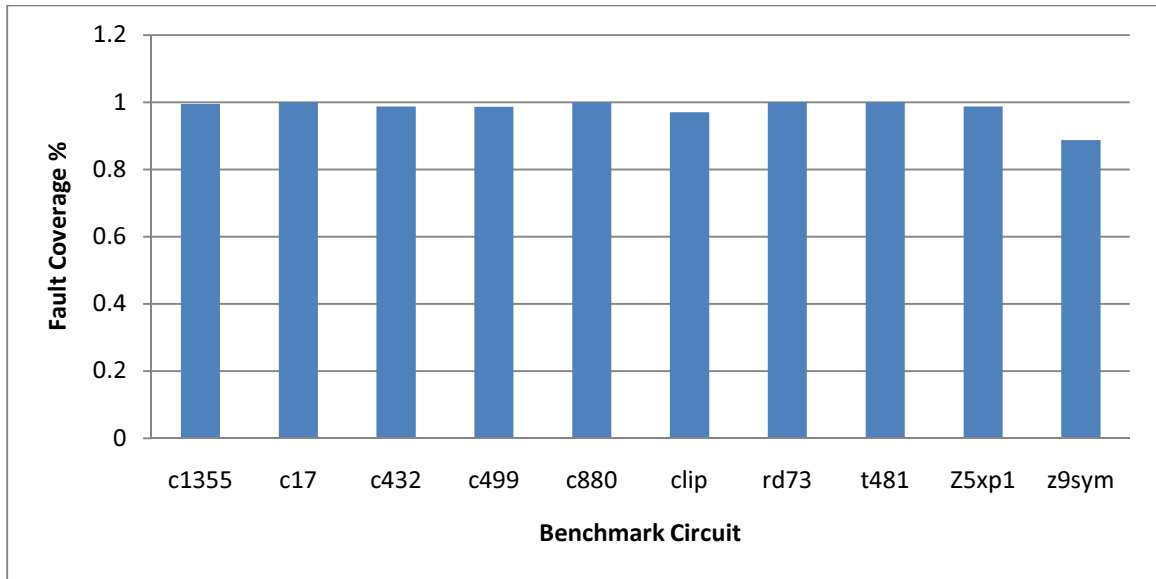


Figure 6-8. Fault coverage percentage in the benchmark circuits

To measure the effectiveness of our approach, we compare our results to existing results from [27], [28], [29] and to Mentor Graphic's FastScan ATPG tool. All circuits were run with 32 instantiations on the FPGA with a system clock frequency of 25MHz. Table 6-6 compares and summarizes the results for the different benchmark circuits. Results for the MCNC benchmark circuits were not available in previously published work of [27], [28] and [29] so the comparison is made only with FastScan for these circuits.

Circuit	[27]	[28]	[29]	FastScan	Our
C432	47	32	57	50	32
C499	59	52	54	54	52
C880	30	17	62	43	25
C1355	86	84	86	87	84
Clip	N/A	N/A	N/A	59	48
Rd73	N/A	N/A	N/A	71	72
T481	N/A	N/A	N/A	35	33
Z5xp1	N/A	N/A	N/A	47	46
Z9sym	N/A	N/A	N/A	77	75

Table 6-6: Comparison of ISCAS and MCNC circuit input test pattern counts using various tests

Table 6-6 suggests that in several cases, our results produce the smallest test set and match the pattern count of the best test set sizes shown in [2]. For MCNC circuits where previously published data was not available, our approach produces a smaller test set than Mentor Graphic's FastScan for all circuits except *rd73*.

Chapter 7: Conclusions and Future Work

Conclusions:

As circuitry becomes larger and more complicated, computers and other computational devices will be utilized more for design checking and fault modeling. Without this, it would take years to do what is now done in seconds. Also, with FPGA's becoming more widespread, it is logical that they be utilized in ways that would replace software analogs. Hardware will almost always be faster than software and with improved synthesis techniques, using an FPGA can be a good replacement for a large number of software algorithms.

As stated in previous chapters, the top priority of this fault detection algorithm was to determine how increasing the number of parallel instantiations of a fault injected circuit could improve the run time. It has been proven that the speed increases inversely, $(1/x)$, to the number of faulty circuit instantiations implemented, especially in the larger circuits. The use in smaller circuit will not be advantageous, but at the same time, circuits that cannot utilize this concept efficiently are small enough to be simulated in software quickly anyway. It is only in the larger circuits that have millions of possible inputs that would need hardware speed increases over traditional simulation.

In addition to this, the algorithm has been shown to have an excellent fault coverage rate. Some run times may be seen as much longer than others, but this is because there was a trade-off in terms of the number of useful vectors dropping or the number of faults

detected increasing. In the appendices it can be seen that this can algorithm can run significantly quicker if certain trade-offs are acceptable.

Future Work

The work presented in this research can be improved upon in a number of ways:

Due to time constraints certain aspects of the algorithm were not able to be completed or optimized. The first issue was that of timing. There are a number of places where combinational logic delay forced the FPGA core to run at slower speeds. This included the section of code where the faults were counted. This was done with a section of code that counted bits individually in a sequential order. To improve this, adding groups of bits would have been more efficient and would have improved the speed.

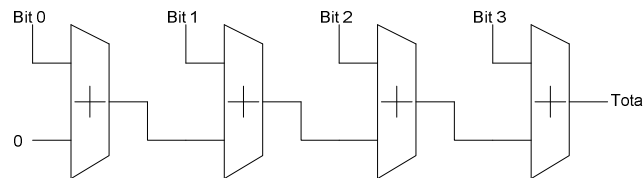


Figure 7-1. Bit counter in the current algorithm

Figure 7-1 is an example of how the adding was done in the current fault detection algorithm.

The program simply lines up a large number of adders in a row. One input is a bit from a vector whose logic high bits are being counted and the other input is the total number of logic high bits counted thus far.

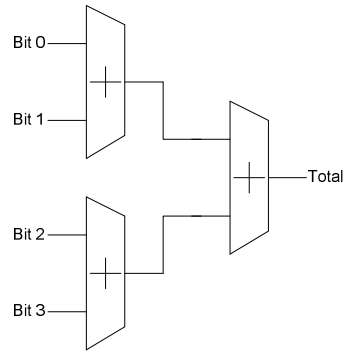


Figure 7-2. Improved bit counter

Figure 7-2 represents the improved bit counter. In this design there are only \log_2 (rounded up to the next integer) levels of logic, significantly reducing the logic delay associated with the Figure 7-1. In this design, every group of two bits is added together, these totals are then added together until all logic high bits are counted.

In addition to this, more efficient fault detection algorithms could be used with this research to further decrease runtime. Because the objective of this research was to determine if the parallel nature of FPGAs could be utilized to increase the speed of fault detection, the actual fault detection algorithm was not made as efficiently as possible. Research such as [12] and [13] have shown significant strides in serial emulation, and combined with this research could potentially show even faster speeds.

Bibliography:

- [1] P. Clarke. (2010, March 29) Xilinx, ASIC Vendors Talk Licensing. [Online].
Available: <http://www.eetimes.com/story/OEG20010622S0091>
- [2] M.L. Bushnell, V.D. Agrawal, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. Boston: Springer, pp.57-120, 2005.
- [3] D. Moore and H. Walker, *Yield Simulation for Integrated Circuits*.
Boston/Dordrecht/Lancaster: Kluwer Academic Publishers, pp.22-55, 1987.
- [4] S. Foley, J. Molyneaux, A. Mathewson, An evaluation of fast wafer level test methods for interconnect reliability control, *Microelectronics Reliability*, Volume 39, Issue 11, November 1999, Pages 1707-1714
- [5] C. Cohn and C. A. Harper, *Failure-free Integrated Circuit Packages: Systematic Elimination of Failures*. New York: McGraw Hill, p. 151, 2005.
- [6] H. Y Chang and J. G. Chappel, "Deductive techniques for simulating logic circuits", *Computers*, pp. 52-59, March 1975
- [7] E. G. Ulrich and T. Baker, "The concurrent simulation of nearly identical digital networks," In *ACM IEEE Design Automation Conference*, pp 145-150.
- [8] J. P. Roth, "Diagnosis of automata failures: A calculus and a method", *IBM Journal of Research & Development*, vol. 10, pp. 278-291, July 1966

- [9] W. T. Cheng and M. L. Yu, "Differential fault simulation for sequential circuits", *Journal of Electronic Testing*, vol. 1, pp. 7-13, July 1990
- [10] K. Gulati, and S. Khatri, "Towards acceleration of fault simulation using graphics processing units," In *Proceedings of the 45th Annual Conference on Design Automation*, pp. 822-827, 2008.
- [11] P. Ellervee, J. Raik, and V. Tihhomirov, "Fault emulation on FPGA: a feasibility study," In *Proceedings of the 21st NORCHIP Conference*, Riga, Latvia, pp.92-95, 2003.
- [12] P. Ellervee, J. Raik, and V. Tihhomirov, "Environment for fault simulation acceleration on FPGA," In *Proceedings of the 9th Biennial Baltic Electronics Convergence*, pp.217-220, 2004.
- [13] D. Saab, F. Kocan, and J. Abraham, "Massively parallel/reconfigurable emulation model for the d-algorithm," In *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pp. 1172-1176, 2002.
- [14] F. Kocan, and D. Saab, "Concurrent d-algorithm on reconfigurable hardware," *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pp.152-156, 1999.
- [15] K. Cheng, S. Huang, and W. Dai, "Fault emulation: a new methodology for fault grading," In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol 18, Issue 10, pp.1487-1495, 1999.

- [16] J. S. Augusto, C. B. Almeida, and H. C. Neto., “A modular reconfigurable architecture for efficient fault simulation in digital circuits,” In *Lecture Notes in Computer Science: Field-Programmable Logic and Applications*, pp.818-827, 2003.
- [17] F. Kocan, and D. Saab, “Dynamic fault diagnosis of combinational and sequential circuits on reconfigurable hardware,” *Journal of Electronic Testing: Theory and Applications*, Vol. 23, Issue 5 pp.405-420, 2007.
- [18] B. Zeidman. (2010, March 29) The Death of the Structured ASIC. [Online]. Available: <http://chipdesignmag.com/display.php?>
- [19] Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, [Online] Available at: http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf.
- [20] PowerPC 405-S Embedded Processor Core User’s Manual Version 1.1, [Online] Available at: <http://www-03.ibm.com/technology/power/licensing/cores/ppc405.html>
- [21] PowerPC 405 Processor Block Reference Guide, [Online] Available at: http://www.xilinx.com/support/documentation/user_guides/ug018.pdf
- [22] Xilinx University Program Virtex-II Pro Development System, [Online] Available at: http://www.xilinx.com/univ/XUPV2P/Documentation/XUPV2P_User_Guide.pdf
- [23] FIFOs Using Virtex-II Block RAM, [Online] Available at: http://www.xilinx.com/support/documentation/application_notes/xapp258.pdf

[24] Clock Generator (v3.02a), [Online] Available at:

http://www.xilinx.com/support/documentation/ip_documentation/clock_generator.pdf

[25] JTAGPPC Controller, [Online] Available at:

http://www.xilinx.com/support/documentation/ip_documentation/jtagppc_cntlr.pdf

[26] Processor System Reset Module, [Online] Available at:

http://www.xilinx.com/support/documentation/ip_documentation/proc_sys_reset.pdf

[27] I. Pomeranz, L. N. Reddy, and S. M. Reddy, “COMPACTEST: a method to generate compact test sets for combinational circuits,” in *International Test Conference*, 1991, pp. 194–203.

[28] M. C. Hansen and J. P. Hayes, “High-level test generation using physically-induced faults,” in *IEEE VLSI Test Symposium*, 1995, p. 20.

[29] E. Bareisa, V. Jusas, K. Motiejunas, and R. Seinauskas, “Test generation at the algorithm-level for gate-level fault coverage,” *Microelectronics Reliability*, vol. 48, no. 7, pp. 1093 – 1101, 2008.

Appendix 1: C++ Code for fault induction

```
// ErrorAnalysisAndMuxInsertion.cpp : Defines the entry point for the console
// application.
//

#include "stdafx.h"
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
#include <stdio.h>
#include <conio.h>
#include <ctime>
using namespace std;

class cirGraph{
public:
    string Name;                //Name of node
    string Type;                //Node Type
    int Level;                  //Level
    vector<string> Parents;      //Nodes above
    vector<string> Children;     //Nodes below
    void clrGraph();            //Clears all data in class
};

void cirGraph::clrGraph(){
    this->Name = "";
    this->Type = "";
    this->Level= 0;
    this->Parents.clear();
    this->Children.clear();
}

string deleteSpaces(string);
void GraphEntries(string, vector<cirGraph> &nodeFill);
void ParentCheck(vector<cirGraph> &checkParents, vector<string> &usedMore, vector<string>
&inputs, vector<int> &usedMoreNum);
void textInsert(string,string,vector<string> &usedMultiple, vector<string> &inputs,
vector<int> &usedMoreNum);
int main(void)
{
    std::clock_t start = std::clock();
    int i = 0;
    vector<cirGraph> nodes;
    vector<string> usedNumerous, inputs;
    string benchmarkname = "t481";
    vector<int> usedNumerousNum;
    string fileName = "E:\\Users\\Joe
Dunbar\\Documents\\Verilog_files\\Verilog_files\\" + benchmarkname + "\\" + benchmarkname
+ ".v";          //input file
    string outFile = "E:\\Users\\Joe
Dunbar\\Documents\\Verilog_files\\Verilog_files\\" + benchmarkname + "\\" + benchmarkname
+ "test.v";      //output file
    cout << outFile;
    GraphEntries(fileName, nodes);
    int size = nodes.size();
    cout << "Name, Type, Parents" << endl;
    for (i=0 ; i< size; i++){
        //Outputs the
        current data from the clrgraph vector
        cout << nodes.at(i).Name << ", " << nodes.at(i).Type;
        int childrenSize = nodes.at(i).Children.size();
        int parentSize = nodes.at(i).Parents.size();
        for(int k = 0; k < parentSize; k++)
            cout << ", " << nodes.at(i).Parents.at(k);
        cout << endl;
    }
}
```

```

ParentCheck(nodes,usedNumerous,inputs, usedNumerousNum);
size = usedNumerous.size();
cout << "Parents used more than once: " << usedNumerous.at(0);
for(i=1;i<size;i++)//Outputs to terminal the nodes that are used numerous times
    cout << "," << usedNumerous.at(i);    //in the file
cout << endl;
textInsert(fileName, outFile, usedNumerous, inputs, usedNumerousNum);
std::cout<< ( ( std::clock() - start ) / (double)CLOCKS_PER_SEC ) <<'\n';
getch();
return 0;
}
/* Searches the input file given, and sets up the cirGraph class with data from
it. */
void GraphEntries(string file, vector<cirGraph> &nodeFill)
{
    size_t commapos, semicolpos;
    ifstream inFile;
    inFile.open(file.c_str());
    string templine, line;
    cirGraph x;
    int pos=0;
    int input = 0;

    if(!inFile){    //Makes sure that the file is usable
        cerr << "Unable to open file datafile.txt";
        //exit(1);
    }
    //Search File For Graph Entries
    while(!inFile.eof()){
        getline(inFile,line);
        //Find Primary Inputs
        string input = "input";
        if (line.rfind(input) != string::npos){
            semicolpos = line.find_first_of(";");
            line.replace(line.rfind(input),input.length(), " ");
            commapos = line.find_first_of(",");
            while(semicolpos != string::npos){
                commapos = line.find_first_of(",");
                if(commapos != string::npos){
                    templine = line.substr(0, line.find_first_of(","));
                    templine = deleteSpaces(templine);
                    line =
line.substr(line.find_first_of(",")+1,line.size()-line.find_first_of(",")-1);
                }
                else if(commapos == string::npos){
                    templine = line.substr(0, line.find_first_of(";"));
                    templine = deleteSpaces(templine);
                    line =
line.substr(line.find_first_of(";")+1,line.size()-line.find_first_of(";")-1);
                }
                x.Name = templine;
                x.Level = 1;
                x.Parents.push_back("Null");
                x.Type = "PI";
                nodeFill.push_back(x);
                x.clrGraph();
                input = 1;
                semicolpos = line.find_first_of(";");
                commapos = line.find_first_of(",");
            }
            if((semicolpos == string::npos) && (commapos != string::npos)){
                templine = line.substr(0, line.find_first_of(","));
                templine = deleteSpaces(templine);
                line = line.substr(line.find_first_of(",")+1,line.size()-
line.find_first_of(",")-1);
                x.Name = templine;
                x.Level = 1;
            }
        }
    }
}

```



```

        x.Parents.push_back("Null");
        x.Type = "PI";
        nodeFill.push_back(x);
        x.clrGraph();
    }
}
else if((line.find_first_of("(") != string::npos) &&
(line.rfind("module")==string::npos)){
    if(line.find_first_of(" ") < 1)
        line.replace(0,line.find_first_not_of(" "), "");
    templine = line.substr(0, line.find_first_of(" "));
    x.Type = templine;
    line = line.substr(line.find_first_of("(")+1,line.size()-
line.find_first_of("(")-1);
    commapos = line.find_first_of(",");
    templine = line.substr(0,line.find_first_of(","));
    templine = deleteSpaces(templine);
    x.Name = templine;
    line = line.substr(line.find_first_of(",")+1, line.size()-
line.find_first_of(",")-1);
    commapos = line.find_first_of(",");
    while(commapos != string::npos){
        templine = line.substr(0,line.find_first_of(","));
        templine = deleteSpaces(templine);
        x.Parents.push_back(templine);
        line = line.substr(line.find_first_of(",")+1, line.size()-
line.find_first_of(",")-1);
        commapos = line.find_first_of(",");
    }
    templine = line.substr(0,line.find_first_of(")"));
    templine = deleteSpaces(templine);
    x.Parents.push_back(templine);
    line.clear();
    templine.clear();
    nodeFill.push_back(x);
    x.clrGraph();
}
}
inFile.close();
}
/*Given a specific string, this function deletes all white spaces in front of and behind
the string*/
string deleteSpaces(string phrase)
{
    size_t space = phrase.find_first_of(" ");
    while(space != string::npos){
        phrase.replace(space,1,"");
        space = phrase.find_first_of(" ");
    }
    return phrase;
}
//Checks all the nodes found in cirGraph vector and creates two vectors.
//One contains strings with all the node names used more than once.
//The other contains the number of times each of these nodes are called.
void ParentCheck(vector<cirGraph> &checkParents, vector<string> &usedMore, vector<string>
&inputs, vector<int> &usedMoreNum){
    vector<string> all;
    int i, j, pSize, count, allSize,aSize;
    int written =0;
    int checkParentsSize = checkParents.size();
    string check;
    for(i=0;i<checkParentsSize;i++){

        //Cycles through all parents and assigns them to a
        pSize = checkParents.at(i).Parents.size();

        //vector of strings called "all"

```

```

        for(j=0;j<pSize;j++){
            all.push_back(checkParents.at(i).Parents.at(j));
            if(strcmp(checkParents.at(i).Parents.at(0).c_str(), "Null") == 0){
                inputs.push_back(checkParents.at(i).Name);
            }
        }
        allSize = all.size();
        for(i=0;i<allSize;i++){

            //Cycles through vector "all"
            check = all.at(i);
            count = 0;
            for(j = 0; j<allSize;j++){

                //Counts the number of times a string shows up in "all"
                if(strcmp(check.c_str(),all.at(j).c_str()) == 0)
                    count++;
            }
            aSize = usedMore.size();

            for(j = 0;j<aSize;j++){

                //If a parent is used more than once it is placed in the
                if(strcmp(usedMore.at(j).c_str(),check.c_str()) == 0)
                    //string
                vector usedMore and the number of times it is used
                    written = 1;

                //is placed in int vector usedMoreNum. If a Parent is already
                }

                //accounted for it is skipped here.
                if((count > 1) && (written == 0)){
                    usedMore.push_back(check);
                    usedMoreNum.push_back(count);
                }
                written = 0;
            }
        }
        //Inserts all the text into the output file
        //Creates a file with updated module, inputs, outputs, wires, and gates
        //Also inserts Muxs where needed
        //At the end it inserts a mux function to every file
        void textInsert(string file,string ofile, vector<string> &usedMultiple, vector<string>
        &inputs, vector<int> &usedMoreNum){
            //declarations
            ifstream inFile(file.c_str());
            ofstream outFile(ofile.c_str());
            if(!inFile){

                //Make sure the file is accessible
                cerr << "Unable to open file datafile.txt";
                exit(1);
            }
            if(!outFile){
                cerr << "Unable to open file datafile.txt";
                exit(1);
            }
            bool inputMux = false;

            bool inputUsed = false;
            int i, j;
            int count = 0;
            int size = usedMultiple.size();
            int done = 0;
            int inputNum = inputs.size();
            int muxNum = 0;

```

```

int erased = 0;
int used = 0;
int condition = 1;
char buffer[100] ;
vector<int> num;
vector<int>::iterator it;
string line, templine, templine2, fullLine;
string module = "module ";
size_t commaspos;
for(i=0;i<size;i++){
    count += usedMoreNum.at(i);
while(!inFile.eof()){
    erased = 0;
    used = 0;
    getline(inFile,line);
    fullLine = line;
    if((line.rfind("module") != string::npos) && (line.rfind("end") ==
string::npos)){ //Adds all the inputs that
are needed to the module line
        templine = line.substr(line.find_first_of("e")+1,
line.find_first_of("(")-line.find_first_of("e")-1);
        templine = deleteSpaces(templine);
        templine = templine.append("test");
        templine = module.append(templine);
        line = line.substr(line.find_first_of("("), line.size()-
line.find_first_of("("));
        line = templine.append(line);
        templine = line.substr(0,line.find_first_of("("));
        for(i = 0;i<count;i++){
            itoa(i,buffer,10);
            templine.append(",SS");
            templine.append(buffer);
        }
        templine.append(", errsigs");
        fullLine = templine;
    }
    templine.clear();
    if((line.rfind("input") != string::npos) && (line.rfind("/") ==
string::npos)){ //Adds all the inputs that
are needed to the input line
        templine = line.substr(0,line.find_first_of(";"));
        for(i = 0;i<count;i++){
            itoa(i,buffer,10);
            templine.append(",SS");
            templine.append(buffer);
        }
        templine.append(", errsigs");
        fullLine = templine;
    }
    templine.clear();
    if((line.rfind("wire") != string::npos) && (line.rfind("/") ==
string::npos)){ //Puts all wires that
are needed at nodes on the wire line
        templine = line.substr(0, line.find_first_of(";"));
        for(i = 0; i<inputNum;i++){
            templine.append(", ");
            templine.append(inputs.at(i));
            templine.append("_0");
        }
        for(i = 0; i<size; i++){
            for(j = 0; j<usedMoreNum.at(i); j++){
                if(usedMultiple.at(i).rfind("Null") ==
string::npos){
                    itoa(j+1,buffer,10);
                    templine.append(", ");
                    templine.append(usedMultiple.at(i));
                    templine.append("_");
                }
            }
        }
    }
}

```

```

        templine.append(buffer);
    }
}
templine.append(";");
fullLine = templine;
}
if((line.find_first_of("(") != string::npos) &&
(line.rfind("module")==string::npos)){ //Finds first
line with a "(" that doesn't have a "module"
    if(done == 0){

        //All new inputs muxes are put here if done != 1
        for(i = 0;i<inputNum;i++){
            outFile << "mux Mux" << muxNum << "(" <<
inputs.at(i) << "_0, SS" << muxNum << ", errsig, " << inputs.at(i) << ");" << endl;
            muxNum++;
        }
        done = 1;
    }
    templine2 = line.substr(0, line.find_first_of(",")+1);
//On the first
line it checks all of the inputs and outputs
    line = line.substr(line.find_first_of(",")+1, line.size()-
line.find_first_of(",")-1);
    commapos = line.find_first_of(",");

    while(commapos != string::npos){

        //If there is a comma the while loop starts checking inputs
        inputMux = false;
        templine = line.substr(0,line.find_first_of(","));
        templine = deleteSpaces(templine);
        for(j = 0;j<inputNum;j++){
            if(templine.compare(inputs.at(j)) == 0){
                inputMux = true;
            }
        }
        for(i = 0;i<size;i++){
            if((templine.compare(usedMultiple.at(i)) == 0) &&
(templine.compare("Null") != 0)){ //If an input matches a usedMultiple
entry a mux is added to the file
                if(erased == 0){

                    //and the inputs to the original line are changed to match the input
                    fullLine = templine2;
                    erased = 1;
                }
                for(j = 0; j < inputNum; j++){
                    if(templine.compare(inputs.at(j)) ==
0){
                        outFile << "mux Mux" <<
muxNum << "(" << usedMultiple.at(i) << "_" << usedMoreNum.at(i) << ", SS" << muxNum << ",
errsig, " << usedMultiple.at(i) << "_0);" << endl;
                        inputUsed = true;
                    }
                }
                if(!inputUsed)
                    outFile << "mux Mux" << muxNum << "("
<< usedMultiple.at(i) << "_" << usedMoreNum.at(i) << ", SS" << muxNum << ", errsig, " <<
usedMultiple.at(i) << ");" << endl;
                    inputUsed = false;
                    fullLine.append(usedMultiple.at(i));
                    fullLine.append("_");
                    itoa(usedMoreNum.at(i), buffer, 10);
                    fullLine.append(buffer);

```

```

        fullLine.append(",");
        muxNum++;
        usedMoreNum.at(i) = usedMoreNum.at(i) - 1;
        used = 1;
    }
    for(j = 0; j < inputNum; j++){

        //If a line contains one of the original
inputs
        if((inputMux) && (templine.compare("Null") != 0) &&
        (used == 0) && (templine.compare(inputs.at(j)) == 0)){
            //The input must be changed
            to what the input mux has
            if(erase == 0){
                fullLine = templine2;
                erase = 1;
            }
            fullLine.append(inputs.at(j));
            fullLine.append("_0");
            fullLine.append(",");
            used = 1;
        }
        if((used == 0) && (erase == 1)){
            fullLine.append(line.substr(0, line.find_first_of(",") + 1));
        }
        templine2.append(line.substr(0, line.find_first_of(",") + 1));
        line = line.substr(line.find_first_of(",") + 1, line.size() -
line.find_first_of(",") - 1);
        commapos = line.find_first_of(",");
        used = 0;
    }
    templine = line.substr(0, line.find_first_of(","));
    templine = deleteSpaces(templine);
    inputMux = false;
    for(j = 0; j < inputNum; j++){
        if(templine.compare(inputs.at(j)) == 0)
            inputMux = true;
    }
    for(i = 0; i < size; i++){
        if(templine.compare(usedMultiple.at(i)) == 0){
            if(erase == 0){
                fullLine = templine2;
                erase = 1;
            }
            for(j = 0; j < inputNum; j++){
                if(templine.compare(inputs.at(j)) == 0){
                    outFile << "mux Mux" << muxNum << "(" <<
usedMultiple.at(i) << "_" << usedMoreNum.at(i) << ", SS" << muxNum << ", errsig, " <<
usedMultiple.at(i) << "_0;" << endl;
                    inputUsed = true;
                }
            }
            if(!inputUsed)
                outFile << "mux Mux" << muxNum << "(" <<
usedMultiple.at(i) << "_" << usedMoreNum.at(i) << ", SS" << muxNum << ", errsig, " <<
usedMultiple.at(i) << ";" << endl;

            inputUsed = false;
            fullLine.append(" ");
            fullLine.append(usedMultiple.at(i));
            fullLine.append("_");
            itoa(usedMoreNum.at(i), buffer, 10);
            fullLine.append(buffer);
            fullLine.append(");");
            muxNum++;

```

```

        used = 1;
        usedMoreNum.at(i) = usedMoreNum.at(i) - 1;
    }
}
for(j = 0; j < inputNum; j++){
    if((templine.compare(inputs.at(j)) == 0) &&
(templine.compare("Null") != 0) && (fullLine.find(inputs.at(j)) == string::npos)){
        if(erased == 0){
            fullLine = templine2;
            erased = 1;
        }
        fullLine.append(inputs.at(j));
        fullLine.append("_0");
        fullLine.append(";");
        used = 1;
    }
}
for(j = 0; j < inputNum; j++){

    //If a line contains one of the original inputs
    if((inputMux) && (templine.compare("Null") != 0) && (used
== 0) && (templine.compare(inputs.at(j)) == 0)){ //The input must be changed
to what the input mux has
        if(erased == 0){
            fullLine = templine2;
            erased = 1;
        }
        fullLine.append(inputs.at(j));
        fullLine.append("_0");
        fullLine.append(";");
        used = 1;
    }
}
if((used == 0) && (erased == 0)){
    templine2.append(line.substr(0, line.find_first_of(";")));
    templine2.append(";");
    fullLine = templine2;
}
else if(used == 0){
    fullLine.append(templine);
    fullLine.append(";");
}
line.clear();
templine.clear();

}
outFile << fullLine << endl;

//Add line to file
}

//Next two lines are adding the mux information
outFile << "module mux(y,sel,b,a); \ninput a,b,sel; \noutput y; \n \nwire
sel,a_sel,b_sel; \n \nnot U_inv (inv_sel,sel); \nnand U_anda (a_sel,a,inv_sel), \nU_andb
(b_sel,b,sel); \nnor U_or (y,a_sel,b_sel); \n \nendmodule \n \n";
inFile.close();
outFile.close();
}

```

Appendix 2: C++ code for creating all other verilog files

```
// InstantiationCreator.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
#include <stdio.h>
#include <conio.h>
#include <math.h>
using namespace std;

string readFile(string);
int numIn(string);
int ssIn(string);
int numOut(string);
void writeSimulate(string, string, string, int, int, int, int);
void writeBlock(string, string, string, int, int, int, int);
void write_user_logic(string, string, string, int, int, int, int);
void writeTest(string, string, int, string);
void writePPCcode(string, string);
string deleteSpaces(string);

int main(void)
{
    int instantiations = 32;
    string benchmarkname = "t481";
    string fileName = "E:\\Users\\Joe
Dunbar\\Documents\\Verilog_files\\Verilog_files\\" + benchmarkname + "\\" + benchmarkname
+ ".v"; //input file
    string testFileName = "E:\\Users\\Joe
Dunbar\\Documents\\Verilog_files\\Verilog_files\\" + benchmarkname + "\\" + benchmarkname
+ "test.v"; //input file
    string outSim = "E:\\Users\\Joe Dunbar\\Documents\\Verilog_files\\Verilog_files\\"
+ benchmarkname + "\\" + benchmarkname + "simulate.v"; //output file
    string outBlock = "E:\\Users\\Joe
Dunbar\\Documents\\Verilog_files\\Verilog_files\\" + benchmarkname + "\\" + benchmarkname
+ "block.v"; //output
    string user_logic = "E:\\Users\\Joe
Dunbar\\Documents\\Verilog_files\\Verilog_files\\" + benchmarkname + "\\user_logic_" +
benchmarkname + ".vhd"; //output
    string simtest = "E:\\Users\\Joe
Dunbar\\Documents\\Verilog_files\\Verilog_files\\" + benchmarkname + "\\" + benchmarkname
+ "simtest.v";
    string cCode = "E:\\Users\\Joe Dunbar\\Documents\\Verilog_files\\Verilog_files\\"
+ benchmarkname + "\\TestApp_Peripheral.c";
    string testName = readFile(fileName);
    int testInputs = numIn(testFileName);
    int SSIn = ssIn(testFileName);
    int testOuts = numOut(testFileName);
    string Name = readFile(fileName);
    cout << testName << " " << testInputs << " " << SSIn << " " << testOuts << endl;
    cout << Name << " " << testInputs << endl;
    writeSimulate(testName, Name, outSim, testInputs, SSIn, instantiations, testOuts);
    writeBlock(testName, Name, outBlock, testInputs, SSIn, instantiations, testOuts);
    write_user_logic(testName, Name, user_logic, testInputs, SSIn, instantiations,
testOuts);
    writeTest(testName, Name, testInputs, simtest);
    writePPCcode(cCode, Name);
    getch();
    return 0;
}
string readFile(string fileName)
```

```

{
    ifstream inFile;
    inFile.open(fileName.c_str());
    if(!inFile){
//Makes
sure that the file is usable
        cerr << "Unable to open file 2.txt";
        exit(1);
    }
    bool stored = false;
    string line;
    while(!inFile.eof() && !stored){
        getline(inFile,line);
        if((line.rfind("module") != string::npos)){
            line =
line.substr(line.find_first_of("e")+1,line.find_first_of("(")-line.find_first_of("e")-1);
            line = deleteSpaces(line);
            stored = true;
        }
    }
    return line;
}
int numIn(string fileName)
{
    ifstream inFile;
    inFile.open(fileName.c_str());
    if(!inFile){
//Makes
sure that the file is usable
        cerr << "Unable to open file 3.txt";
        exit(1);
    }
    string line;
    string lineChar;
    string comma = ",";
    int lineSize;
    int count = 0;
    bool stored = false;
    while(!inFile.eof() && !stored){
        getline(inFile,line);
        if((line.rfind("input") != string::npos) && (line.rfind(";") !=
string::npos)){
            if(line.rfind("SS") != string::npos){
                line = line.substr(line.find_first_of("t")+1,
line.find_first_of("SS")-line.find_first_of("t")-1);
                lineSize = line.size();
                for(int i = 0; i<lineSize; i++){
                    lineChar = line[i];
                    if(comma.compare(lineChar) == 0)
                        count++;
                }
                stored = true;
                return count;
            }
            else{
                line = line.substr(line.find_first_of("t")+1,
line.find_first_of(";")-line.find_first_of("t")-1);
                lineSize = line.size();
                for(int i = 0; i<lineSize; i++){
                    lineChar = line[i];
                    if(comma.compare(lineChar) == 0)
                        count++;
                }
                count++;
                return count;
            }
        }
    }
}

```



```

    }
    }
    return 0;
}
int numOut(string fileName)
{
    ifstream inFile;
    inFile.open(fileName.c_str());
    if(!inFile){
        //Makes
        sure that the file is usable
        cerr << "Unable to open file 4.txt";
        exit(1);
    }
    string line;
    string lineChar;
    string comma = ",";
    int lineSize;
    int count = 0;
    bool stored = false;
    while(!inFile.eof() && !stored){
        getline(inFile,line);
        if((line.rfind("output") != string::npos) && (line.rfind(";") !=
string::npos)){
            line = line.substr(line.find_first_of("t")+1,
line.find_first_of(";")-line.find_first_of("t")-1);
            lineSize = line.size();
            for(int i = 0; i<lineSize; i++){
                lineChar = line[i];
                if(comma.compare(lineChar) == 0)
                    count++;
            }
            count++;
            return count;
        }
    }
    return 0;
}
int ssIn(string fileName)
{
    ifstream inFile;
    inFile.open(fileName.c_str());
    if(!inFile){
        //Makes
        sure that the file is usable
        cerr << "Unable to open file 5.txt";
        exit(1);
    }
    string line;
    string lineChar;
    string comma = ",";
    int lineSize;
    int count = 0;
    bool stored = false;
    while(!inFile.eof() && !stored){
        getline(inFile,line);
        if((line.rfind("input") != string::npos) && (line.rfind("SS") !=
string::npos)){
            line = line.substr(line.find_first_of("SS"), line.rfind(",
errsig")-1-line.find_first_of("SS"));
            lineSize = line.size();
            for(int i = 0; i<lineSize; i++){
                lineChar = line[i];
                if(comma.compare(lineChar) == 0)
                    count++;
            }
            count++;
        }
    }
    return 0;
}

```

```

        stored = true;
        return count;
    }
}
inFile.close();
return 0;
}

void writeSimulate(string testName, string Name, string ofile, int testInputs, int SSIn,
int instantiations, int testOuts)
{
    ofstream outFile(ofile.c_str());
    if(!outFile){
        cerr << "Unable to open file 6.txt";
        exit(1);
    }
    char buffer [10];
    char buf2 [10];
    int i, j;
    outFile << "module " << Name << "simulate (Ins, errsig, SS, Fouts);\n\n";
    outFile << "input errsig; \n\ninput [" << itoa(testInputs-1,buffer,10) << "]:0]
Ins;\n\ninput [" << itoa(SSIn-1,buf2,10) << "]:0] SS;\n\n" ;
    outFile << "output Fouts;\n\n";
    outFile << "wire [" << itoa(testOuts-1,buffer,10) << "]:0] Outs, Outsfault,
Iouts;\n\n";
    outFile << Name << " " << Name <<"original(";
    for(i = 0; i < testInputs; i++)
        outFile << "Ins["<< itoa(i,buffer,10) << "], ";
    for(i = 0; i < testOuts; i++){
        outFile << "Outs[" << itoa(i,buf2,10) << " ] "; //Ouputs
        if(i < testOuts-1)
            outFile << ",";
    }
    outFile<< ");\n";
    outFile << testName << "test " << testName << "testfault(";
    for(j = 0; j< testInputs; j++)
        outFile << "Ins["<< itoa(j,buffer,10) << "], ";
    for(j = 0; j < testOuts; j++)
        outFile << "Outsfault[" << itoa(j,buf2,10) << "], "; //Ouputs
    for(j = 0; j < SSIn; j++){
        outFile << "SS[" << itoa(j,buf2,10) << "], ";
    }
    outFile << "errsig);\n";
    for(j = 0; j<testOuts; j++)
        outFile << "xor (Iouts[" << itoa(j,buf2,10) << "], Outs[" <<
itoa(j,buf2,10) << "], Outsfault[" << itoa(j,buf2,10) << "]);\n";
    outFile << "or " << "(Fouts";
    for(j = 0; j<testOuts; j++)
        outFile << ", Iouts[" << itoa(j,buf2,10) << "];
    outFile <<");\n\n";

    outFile << "endmodule" << endl;
    outFile.close();
}

void writeBlock(string testName, string Name, string ofile, int testInputs, int SSIn, int
instantiations, int testOuts)
{
    int i,j;
    double loopnum;
    double num = (double)SSIn/instantiations;
    int outloop = ceil((double)testInputs/instantiations);
    double bitstemp = (double)(SSIn*2);
    bitstemp = log(bitstemp)/ log((double)2);
    double bits = ceil(bitstemp);
    loopnum = ceil(num) * 2;
    ofstream outFile(ofile.c_str());
    if(!outFile){
        cout << "Unable to open file 7.txt";
    }
}

```

```

        exit(1);
    }
    char buffer [10];
    char buf2 [10];
    char buf3 [10];
    outFile << "module " << testName << "block(WrAck, clk, RdAck, outinput, WrBurst,
threshnum, trigout);\n\n";
    outFile << "    //Parameters\n";
    outFile << "    parameter faultsfound = "<< itoa(SSIn*2, buffer, 10) << ";\n";
    outFile << "    parameter vecsused = 100;\n";
    outFile << "    parameter loopnum = "<< itoa(loopnum, buffer, 10) << ";\n";
    outFile << "    parameter bitstransfer = "<< itoa(instantiations, buffer, 10) <<
";\n";
    outFile << "    parameter interthresh = 1000;\n";
    outFile << "    // Inputs\n";
    outFile << "    input WrAck, clk, RdAck;\n";
    outFile << "    input [31:0] threshnum;\n\n";
    outFile << "    //Outputs\n";
    outFile << "    output [31:0] outinput;\n";
    outFile << "    output WrBurst, trigout;\n\n";
    outFile << "    //Registers\n";
    outFile << "    reg [" << itoa(SSIn-1, buffer, 10) << ":0] SS[bitstransfer-
1:0];\n";
    outFile << "    reg [" << itoa(testInputs-1, buffer, 10) << ":0] Input = " <<
itoa(testInputs,buf2, 10) << "'d2895612794, usefulinp=0;\n";
    outFile << "    reg [31:0] count = 0, burstcount = 3, burstcount3 = 0,
burstcount2=0, temp3=0, threshold = 0, outinput = 0;\n";
    outFile << "    reg [bitstransfer-1:0] mem[loopnum-1:0], mem2[loopnum-1:0],
mem3[loopnum-1:0], temp2 = 0;\n";
    outFile << "    reg [" << itoa(bits-1, buffer, 10) << ":0] total=0, oldtotalfaults
=0,totalfaults = 0, totalvecs = 0;\n";
    outFile << "    reg errsig = 0, WrBurst = 0, done = 0, trigout = 0, trig = 0,
tempbit;\n\n";
    outFile << "    //Wires\n";
    outFile << "    wire [bitstransfer-1:0] Fouts;\n\n";
    outFile << "    //Integers\n";
    outFile << "    integer i;\n";
    outFile << "    genvar j;\n\n";
    outFile << "    //Circuit Declarations\n\n";
    outFile << "    generate\n";
    outFile << "        for(j = 0; j < bitstransfer; j = j +
1)begin:INSTANTIATIONS\n";
    outFile << "            " << testName << "simulate " << testName <<
"simulateX(Input, errsig, SS[j], Fouts[j]);\n";
    outFile << "            end\n";
    outFile << "        endgenerate\n";
    outFile << "\n//Fault Detection Program\n";
    outFile << "always @(posedge clk) begin\n";
    outFile << "    if(WrAck == 1)begin\n";
    outFile << "        for(i = 0; i < loopnum; i = i + 1)begin\n";
    outFile << "            mem[i] = 0;\n";
    outFile << "            mem2[i] = 0;\n";
    outFile << "            mem3[i] = 0;\n";
    outFile << "        end\n";
    outFile << "        trig = 1;\n";
    outFile << "    end\n";
    outFile << "    if(WrAck == 0 && trig == 1)begin\n";
    outFile << "        count = count + 1;\n";
    outFile << "        WrBurst = 0;\n\n";
    outFile << "    end\n";
    outFile << "////////////////////////////////////////\n\n";
    outFile << "////////////////////////////////////////\n\n";
    outFile << "    //Error Counting\n";
    outFile << "        if(burstcount == 3)begin\n";
    outFile << "            if(threshold < threshnum) begin\n";
    outFile << "                tempbit = Input[" << itoa(testInputs-1,
buffer, 10) << "]^Input[change];\n";

```

```

        outFile << "
        Input[" << itoa(testInputs-1, buffer, 10) <<
":1] = Input[" << itoa(testInputs-2, buf2, 10) <<"0";\n";
        outFile << "
        Input[0] = tempbit;\n";
        outFile << "
        temp3 = count+1;\n";
        outFile << "
        total = 0;\n";
        outFile << "
        burstcount = 0;\n";
        outFile << "
        threshold = threshold + 1;\n";
        outFile << "
        SS[0] = 1;\n";
        outFile << "
        end\n";
        else if(threshold == threshnum)begin\n";
        if(burstcount2 == 0)begin\n";
        if(burstcount3 == 0)begin\n";
        for(i = 0; i < loopnum;
i=i+1)begin\n";
        outFile << "
        mem3[i] = mem2[i] |
        mem3[i];\n";
        outFile << "
        end\n";
        outFile << "
        burstcount3 = 0;\n";
        outFile << "
        burstcount2 = burstcount2 + 1;\n";
        outFile << "
        //outFile << "
        totalfaults = 0;\n";
        outFile << "
        //outinput = mem[burstcount3];\n";
        outFile << "
        //WrBurst = 1;\n";
        outFile << "
        end\n";
        else if(RdAck == 1 && burstcount3 <
loopnum)begin\n";
        outFile << "
        outinput = mem[burstcount3];\n";
        outFile << "
        WrBurst = 1;\n";
        outFile << "
        burstcount3 = burstcount3 +1;\n";
        outFile << "
        end\n";
        else if(burstcount3 == 1 /*&& RdAck ==
1)begin\n";
        outFile << "
        totalfaults = 0; \n";
        outFile << "
        burstcount3 = 0;\n";
        outFile << "
        burstcount2 = burstcount2 +1;\n";
        outFile << "
        end*/\n";
        outFile << "
        end\n\n";

        else if(burstcount2 == 1)begin\n";
        for(i = 0; i < loopnum; i =
i+1)begin\n";
        outFile << "
        mem[i] = 0;\n";
        outFile << "
        mem2[i] = 0;\n";
        outFile << "
        end\n";
        outFile << "
        burstcount2 = burstcount2 + 1;\n";
        outFile << "
        //end\n";
        outFile << "
        end\n\n";

        else if(burstcount2 == 2)begin\n";
        if(totalfaults <= oldtotalfaults)begin\n";
        trigout = 1;\n";
        end\n";
        else begin\n";
        burstcount2 = burstcount2 +1;\n";
        end\n";
        end
        \n\n";

        else if(burstcount2 > 2)begin\n";
        if(burstcount2 == 3)begin\n";
        outinput = usefulinput[" <<
        itoa(testInputs-1,buffer, 10) << ":" << itoa((outloop-1)*32,buf2,10) << "];\n";
        outFile << "
        totalvecs = totalvecs+1;\n";
        outFile << "
        WrBurst = 1;\n";
        outFile << "
        oldtotalfaults = totalfaults;\n";
        outFile << "
        burstcount2 = burstcount2+1;\n";
        outFile << "
        end\n";
        for(i = outloop-1; i > 0; i--){

```



```

        outFile << "                end\n\n";
        outFile <<
"////////////////////////////////////\n";
////////////////////////////////////\n";
        outFile << "/// Count total errors in individual \n\n";
        outFile << "                else if(burstcount == 1)begin \n";
        outFile << "                        if(burstcount2 < loopnum)begin\n";
        outFile << "                                temp2 = mem3[burstcount2] |
mem[burstcount2];\n";
        outFile << "                                for(i = 0; i < bitstransfer; i=
i+1)begin\n";
        outFile << "                                        total = total+temp2[i];\n";
        outFile << "                                        end\n";
        outFile << "                                        burstcount2 = burstcount2 + 1;\n";
        outFile << "                                end\n";
        outFile << "                else if(burstcount2 == loopnum) begin\n";
        outFile << "                        burstcount2 = 0;\n";
        outFile << "                        burstcount = burstcount + 1;\n";
        outFile << "                end\n";
        outFile << "        end\n\n";
        outFile << "        else if(burstcount == 2)begin\n";
        outFile << "                if(total > totalfaults) begin\n";
        outFile << "                        for(i = 0; i < loopnum; i= i+1)begin\n";
        outFile << "                                mem2[i] = mem[i];\n";
        outFile << "                        end\n";
        outFile << "                        totalfaults = total;\n";
        outFile << "                        usefulinput = Input;\n";
        outFile << "                        burstcount2 = 0;\n";
        outFile << "                        burstcount = burstcount +1;\n";
        outFile << "                end \n";
        outFile << "        else begin\n";
        outFile << "                burstcount2 = 0;\n";
        outFile << "                burstcount = burstcount+1;\n";
        outFile << "        end\n";
        outFile << "        end\n";
        outFile << "        end\n";
        outFile << "        end\n\n";
        outFile << "always@(SS[0])begin\n";
        outFile << "        for(i = 1; i < bitstransfer; i = i+1)begin\n";
        outFile << "                SS[i] = SS[0] << i;\n";
        outFile << "        end\n";
        outFile << "end\n\n";
        outFile << "endmodule\n\n";
}

void writeTest(string testName, string name, int testInputs, string simtest)
{
    int i;
    int outloop = ceil((double)testInputs/32);
    ofstream outFile(simtest.c_str());
    if(!outFile){
        cout << "Unable to open file 7.txt";
        exit(1);
    }
    char buffer [10];
    outFile << "    `timescale 1ns / 1ns\n";
    outFile <<
"////////////////////////////////////\n";
    outFile << "\n";
    outFile << "module test;\n";
    outFile << "\n";
    outFile << "    // Inputs\n";
    outFile << "    reg WrAck;\n";
    outFile << "    reg clk;\n";
    outFile << "    reg RdAck, trig = 0;\n";
    outFile << "    reg [31:0] threshnum, Ins[" << itoa(outloop-1, buffer, 10) << ":0],
totalfaults, totalvecs;\n";
    outFile << "    reg [1:0] count = 0;\n";

```

```

outFile << " // Outputs\n";
outFile << " wire [31:0] outinput, threshhold;\n";
outFile << " wire WrBurst;\n";
outFile << " wire trigout;\n";
outFile << " \n";
outFile << " integer file;\n\n\n";
outFile << " // Instantiate the Unit Under Test (UUT)\n";
outFile << " " << testName << "block uut (\n";
outFile << "     .WrAck(WrAck), \n";
outFile << "     .clk(clk), \n";
outFile << "     .RdAck(RdAck), \n";
outFile << "     .outinput(outinput), \n";
outFile << "     .WrBurst(WrBurst), \n";
outFile << "     .threshnum(threshnum), \n";
outFile << "     .trigout(trigout),\n";
outFile << "     .threshhold(threshhold)\n";
outFile << " );\n\n";
outFile << " initial begin\n";
outFile << "     // Initialize Inputs\n";
outFile << "     WrAck = 0;\n";
outFile << "     clk = 0;\n";
outFile << "     RdAck = 0;\n";
outFile << "     threshnum = 0;\n";
outFile << "     file = $fopen(\"output.txt\");\n";
outFile << "     // Wait 100 ns for global reset to finish\n";
outFile << "     #100;\n";
outFile << "     threshnum = 1000;\n";
outFile << "     WrAck = 1;\n";
outFile << "     #20;\n";
outFile << "     WrAck = 0;\n";
outFile << "     // Add stimulus here\n";
outFile << " end\n\n";
outFile << " initial begin\n";
outFile << "     while(1) begin\n";
outFile << "         clk = ~clk;\n";
outFile << "         #1;\n";
outFile << "     end\n";
outFile << " end\n\n";
outFile << " always@(posedge WrBurst)begin\n";
outFile << "     if(count == 0)begin\n";
outFile << "         Ins[0] = outinput;\n";
outFile << "         count = count + 1;\n";
outFile << "     end\n";
for(i = 1; i < outloop; i++)
{
    outFile << "         else if(count == " << itoa(i, buffer, 10) <<
")begin\n";
    outFile << "             Ins[" << itoa(i, buffer, 10) << "] =
outinput;\n";
    outFile << "             count = count + 1;\n";
    outFile << "         end\n";
}
outFile << "         else if(count == " << itoa(outloop, buffer, 10) <<
")begin\n";
    outFile << "             totalfaults = outinput;\n";
    outFile << "             count = count + 1;\n";
    outFile << "         end\n";
    outFile << "         else if(count == " << itoa(outloop+1, buffer, 10) <<
")begin\n";
    outFile << "             totalvecs = outinput;\n";
    outFile << "             $fdisplay(file, \"");
for(i = 0; i < outloop; i++)
    outFile << " %8h";
outFile << " %d %d %d\", ";
for(i = 0; i < outloop; i++)
    outFile << "Ins[" << itoa(i, buffer, 10) << "],";
outFile << " totalfaults, totalvecs, $time);\n";

```

```

        outFile << "                count = 0;\n";
        outFile << "                end\n";
        outFile << "            end\n\n";
        outFile << "        always@(negedge WrBurst)begin\n";
        outFile << "            #6;\n";
        outFile << "            RdAck = 1;\n";
        outFile << "            #1;\n";
        outFile << "            RdAck = 0;\n";
        outFile << "        end\n\n";
        outFile << "        always@(trigout)begin\n";
        outFile << "            $stop;\n";
        outFile << "        end\n\n";
        outFile << "endmodule\n";
    }
}

void write_user_logic(string testName, string Name, string ofile, int testInputs, int
SSIn, int instantiations, int testOuts)
{
    int testiter;
    testiter = testInputs/32;
    ofstream outFile(ofile.c_str());
    if(!outFile){
        cout << "Unable to open file 8.txt";
        exit(1);
    }
    char buf2 [10];
    outFile << "-----\n";
    outFile << "-- user_logic.vhd - entity/architecture pair\n";
    outFile << "-----\n";
    outFile << "--\n";
    outFile << "--\n";
    outFile << "*****\n";
    outFile << "-- ** Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.\n";
    outFile << "-- **\n";
    outFile << "-- ** Xilinx, Inc.\n";
    outFile << "-- ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION \"AS IS\"\n";
    outFile << "-- ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND\n";
    outFile << "-- ** SOLUTIONS FOR XILINX DEVICES. BY PROVIDING THIS DESIGN, CODE,\n";
    outFile << "-- ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,\n";
    outFile << "-- ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION\n";
    outFile << "-- ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF\n";
    outFile << "-- ** INFRINGEMENT, AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE\n";
    outFile << "-- ** FOR YOUR IMPLEMENTATION. XILINX EXPRESSLY DISCLAIMS ANY\n";
    outFile << "-- ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE\n";
    outFile << "-- ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR\n";
    outFile << "-- ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF\n";
    outFile << "-- ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS\n";
    outFile << "-- ** FOR A PARTICULAR PURPOSE.\n";
    outFile << "-- **\n";
    outFile << "-- **\n";
}

```



```

outFile << "--
*****\n";
outFile << "--\n";
outFile << "-----
\n";
outFile << "-- Filename:          user_logic.vhd\n";
outFile << "-- Version:            1.00.a\n";
outFile << "-- Description:         User logic.\n";
outFile << "-- Date:              Sat Jul 25 20:34:42 2009 (by Create and Import
Peripheral Wizard)\n";
outFile << "-- VHDL Standard:      VHDL'93\n";
outFile << "-----
\n";
outFile << "-- Naming Conventions:\n";
outFile << "--   active low signals:      \*_n\\"\n";
outFile << "--   clock signals:           \\"clk\\"", \\"clk_div#\\"",
\n\"clk_#x\\"\n";
outFile << "--   reset signals:          \\"rst\\"", \\"rst_n\\"\n";
outFile << "--   generics:                 \\"C_*\\"\n";
outFile << "--   user defined types:         \\"*_TYPE\\"\n";
outFile << "--   state machine next state:   \\"*_ns\\"\n";
outFile << "--   state machine current state: \\"*_cs\\"\n";
outFile << "--   combinatorial signals:     \\"*_com\\"\n";
outFile << "--   pipelined or register delay signals: \\"*_d#\\"\n";
outFile << "--   counter signals:           \\"*_cnt*\\"\n";
outFile << "--   clock enable signals:       \\"*_ce\\"\n";
outFile << "--   internal version of output port: \\"*_i\\"\n";
outFile << "--   device pins:               \\"*_pin\\"\n";
outFile << "--   ports:                     \\"- Names begin with
Uppercase\\"\n";
outFile << "--   processes:              \\"*_PROCESS\\"\n";
outFile << "--   component instantiations:
\n<ENTITY>_I_<#|FUNC>\n";
outFile << "-----
\n\n";
outFile << "-- DO NOT EDIT BELOW THIS LINE -----\n";
outFile << "library ieee;\n";
outFile << "use ieee.std_logic_1164.all;\n";
outFile << "use ieee.std_logic_arith.all;\n";
outFile << "use ieee.std_logic_unsigned.all;\n\n";
outFile << "library proc_common_v2_00_a;\n";
outFile << "use proc_common_v2_00_a.proc_common_pkg.all;\n\n";
outFile << "-- DO NOT EDIT ABOVE THIS LINE -----\n\n";
outFile << "--USER libraries added here\n\n";
outFile << "-----
\n";
outFile << "-- Entity section\n";
outFile << "-----
\n";
outFile << "-- Definition of Generics:\n";
outFile << "--   C_SLV_DWIDTH              -- Slave interface data bus
width\n";
outFile << "--   C_NUM_REG                 -- Number of software accessible
registers\n";
outFile << "--   C_RDFIFO_DEPTH           -- Read FIFO depth\n";
outFile << "--   C_WRFIFO_DEPTH           -- Write FIFO depth\n";
outFile << "--\n";
outFile << "-- Definition of Ports:\n";
outFile << "--   Bus2IP_Clk               -- Bus to IP clock\n";
outFile << "--   Bus2IP_Reset             -- Bus to IP reset\n";
outFile << "--   Bus2IP_Data              -- Bus to IP data bus\n";
outFile << "--   Bus2IP_BE                -- Bus to IP byte enables\n";
outFile << "--   Bus2IP_RdCE              -- Bus to IP read chip enable\n";
outFile << "--   Bus2IP_WrCE              -- Bus to IP write chip enable\n";
outFile << "--   IP2Bus_Data              -- IP to Bus data bus\n";
outFile << "--   IP2Bus_RdAck             -- IP to Bus read transfer
acknowledgement\n";

```

```

        outFile << "--    IP2Bus_WrAck                -- IP to Bus write transfer
acknowledgement\n";
        outFile << "--    IP2Bus_Error              -- IP to Bus error response\n";
        outFile << "--    IP2RFIFO_WrReq              -- IP to RFIFO : IP write
request\n";
        outFile << "--    IP2RFIFO_Data              -- IP to RFIFO : IP write data
bus\n";
        outFile << "--    IP2RFIFO_WrMark            -- IP to RFIFO : mark beginning of
packet being written\n";
        outFile << "--    IP2RFIFO_WrRelease          -- IP to RFIFO : return RFIFO to
normal FIFO operation\n";
        outFile << "--    IP2RFIFO_WrRestore          -- IP to RFIFO : restore the RFIFO
to the last packet mark\n";
        outFile << "--    RFIFO2IP_WrAck              -- RFIFO to IP : RFIFO write
acknowledgement\n";
        outFile << "--    RFIFO2IP_AlmostFull          -- RFIFO to IP : RFIFO almost
full\n";
        outFile << "--    RFIFO2IP_Full              -- RFIFO to IP : RFIFO full\n";
        outFile << "--    RFIFO2IP_Vacancy            -- RFIFO to IP : RFIFO vacancy\n";
        outFile << "--    IP2WFIFO_RdReq              -- IP to WFIFO : IP read request\n";
        outFile << "--    IP2WFIFO_RdMark            -- IP to WFIFO : mark beginning of
packet being read\n";
        outFile << "--    IP2WFIFO_RdRelease          -- IP to WFIFO : return WFIFO to
normal FIFO operation\n";
        outFile << "--    IP2WFIFO_RdRestore          -- IP to WFIFO : restore the WFIFO
to the last packet mark\n";
        outFile << "--    WFIFO2IP_Data              -- WFIFO to IP : WFIFO read data\n";
        outFile << "--    WFIFO2IP_RdAck              -- WFIFO to IP : WFIFO read
acknowledgement\n";
        outFile << "--    WFIFO2IP_AlmostEmpty        -- WFIFO to IP : WFIFO almost
empty\n";
        outFile << "--    WFIFO2IP_Empty              -- WFIFO to IP : WFIFO empty\n";
        outFile << "--    WFIFO2IP_Occupancy            -- WFIFO to IP : WFIFO occupancy\n";
        outFile << "-----\n\n";
        outFile << "entity user_logic is\n";
        outFile << "  generic\n";
        outFile << "  (\n";
        outFile << "    -- ADD USER GENERICS BELOW THIS LINE -----\n";
        outFile << "    --USER generics added here\n";
        outFile << "    -- ADD USER GENERICS ABOVE THIS LINE -----\n\n";
        outFile << "    -- DO NOT EDIT BELOW THIS LINE -----\n";
        outFile << "    -- Bus protocol parameters, do not add to or delete\n";
        outFile << "    C_SLV_DWIDTH              : integer              := 32;\n";
        outFile << "    C_NUM_REG                 : integer              := 1;\n";
        outFile << "    C_RDFIFO_DEPTH            : integer              := 8192;\n";
        outFile << "    C_WRFIFO_DEPTH            : integer              := 512;\n";
        outFile << "    -- DO NOT EDIT ABOVE THIS LINE -----\n";
        outFile << "  );\n";
        outFile << "  port\n";
        outFile << "  (\n";
        outFile << "    -- ADD USER PORTS BELOW THIS LINE -----\n";
        outFile << "    --USER ports added here\n";
        outFile << "    -- ADD USER PORTS ABOVE THIS LINE -----\n\n";
        outFile << "    -- DO NOT EDIT BELOW THIS LINE -----\n";
        outFile << "    -- Bus protocol ports, do not add to or delete\n";
        outFile << "    Bus2IP_Clk                : in  std_logic;\n";
        outFile << "    Bus2IP_Reset              : in  std_logic;\n";
        outFile << "    Bus2IP_Data               : in  std_logic_vector(0 to
C_SLV_DWIDTH-1);\n";
        outFile << "    Bus2IP_BE                 : in  std_logic_vector(0 to
C_SLV_DWIDTH/8-1);\n";
        outFile << "    Bus2IP_RdCE               : in  std_logic_vector(0 to
C_NUM_REG-1);\n";
        outFile << "    Bus2IP_WrCE               : in  std_logic_vector(0 to
C_NUM_REG-1);\n";

```

```

        outFile << "        IP2Bus_Data                                : out std_logic_vector(0 to
C_SLV_DWIDTH-1);\n";
        outFile << "        IP2Bus_RdAck                                : out std_logic;\n";
        outFile << "        IP2Bus_WrAck                                : out std_logic;\n";
        outFile << "        IP2Bus_Error                                : out std_logic;\n";
        outFile << "        IP2RFIFO_WrReq                                : out std_logic;\n";
        outFile << "        IP2RFIFO_Data                                : out std_logic_vector(0 to
C_SLV_DWIDTH-1);\n";
        outFile << "        IP2RFIFO_WrMark                                : out std_logic;\n";
        outFile << "        IP2RFIFO_WrRelease                            : out std_logic;\n";
        outFile << "        IP2RFIFO_WrRestore                            : out std_logic;\n";
        outFile << "        RFIFO2IP_WrAck                                : in std_logic;\n";
        outFile << "        RFIFO2IP_AlmostFull                            : in std_logic;\n";
        outFile << "        RFIFO2IP_Full                                : in std_logic;\n";
        outFile << "        RFIFO2IP_Vacancy                            : in std_logic_vector(0 to
log2(C_RDFIFO_DEPTH)));\n";
        outFile << "        IP2WFIFO_RdReq                                : out std_logic;\n";
        outFile << "        IP2WFIFO_RdMark                                : out std_logic;\n";
        outFile << "        IP2WFIFO_RdRelease                            : out std_logic;\n";
        outFile << "        IP2WFIFO_RdRestore                            : out std_logic;\n";
        outFile << "        WFIFO2IP_Data                                : in std_logic_vector(0 to
C_SLV_DWIDTH-1);\n";
        outFile << "        WFIFO2IP_RdAck                                : in std_logic;\n";
        outFile << "        WFIFO2IP_AlmostEmpty                            : in std_logic;\n";
        outFile << "        WFIFO2IP_Empty                                : in std_logic;\n";
        outFile << "        WFIFO2IP_Occupancy                            : in std_logic_vector(0 to
log2(C_WRFIFO_DEPTH)));\n";
        outFile << "        -- DO NOT EDIT ABOVE THIS LINE ----- \n";
        outFile << "    );\n\n";
        outFile << "    attribute SIGIS : string;\n";
        outFile << "    attribute SIGIS of Bus2IP_Clk      : signal is \"CLK\";\n";
        outFile << "    attribute SIGIS of Bus2IP_Reset   : signal is \"RST\";\n\n";
        outFile << "end entity user_logic;\n\n";
        outFile << "-----\n";
        outFile << "-- Architecture section\n";
        outFile << "-----\n\n";
        outFile << "architecture IMP of user_logic is\n\n";
        outFile << "    --USER signal declarations added here, as needed for user
logic\n\n";
        outFile << "    component " << Name << "block\n";
        outFile << "    port (\n";
        outFile << "        clk: in std_logic;\n";
        outFile << "        threshnum: in std_logic_VECTOR(31 downto 0);\n";
        outFile << "        outinput: out std_logic_VECTOR(31 downto 0);\n";
        outFile << "        WrBurst: out std_logic;\n";
        outFile << "        RdAck: in std_logic;\n";
        outFile << "        trigout: out std_logic;\n";
        outFile << "        WrAck: in std_logic);\n";
        outFile << "    end component;\n\n\n";
        outFile << "    ----- \n";
        outFile << "    -- Signals for read/write fifo loopback example\n";
        outFile << "    ----- \n";
        outFile << "    type FIFO_CNTL_SM_TYPE is (IDLE, RD_REQ, WR_REQ, RD_REQ_0);\n";
        outFile << "    signal fifo_cntl_ns                                : FIFO_CNTL_SM_TYPE;\n";
        outFile << "    signal fifo_cntl_cs                                : FIFO_CNTL_SM_TYPE;\n";
        outFile << "    signal fifo_rdreq_cmb                                : std_logic;\n";
        outFile << "    signal fifo_wrreq_cmb                                : std_logic;\n";
        outFile << "    signal user_write
: std_logic;\n";
        outFile << "    signal user_write_long                                :
std_logic;\n";
        outFile << "    signal writelock
: std_logic;\n\n";
        outFile << "begin\n\n";
        outFile << "    --USER logic implementation added here\n";

```

```

outFile << " \n";
outFile << " " << Name << "block_0 : " << Name << "block\n";
outFile << " port map (\n";
outFile << "     clk => Bus2IP_Clk,\n";
outFile << "     threshnum => WFIFO2IP_Data,\n";
outFile << "     outinput => IP2RFIFO_Data,\n";
outFile << "     WrBurst => user_write,\n";
outFile << "     RdAck => RFIFO2IP_WrAck,\n";
outFile << "     trigout => user_write_long,\n";
outFile << "     WrAck => WFIFO2IP_RdAck);\n";
outFile << " ----- \n";
outFile << " -- Example code to transfer data between read/write fifo\n";
outFile << " -- \n";
outFile << " -- Note:\n";
outFile << " -- The example code presented here is to show you one way of
operating on\n";
outFile << " -- the read/write FIFOs provided for you. There's a set of IPIC
ports\n";
outFile << " -- dedicated to the FIFO operations, beginning with RFIFO2IP_* or
IP2RFIFO_*\n";
outFile << " -- or WFIFO2IP_* or IP2WFIFO_*. Some FIFO ports are only available
when\n";
outFile << " -- certain FIFO services are present, s.t. vacancy calculation,
etc.\n";
outFile << " -- Typically you will need to have a state machine to read data from
the\n";
outFile << " -- write FIFO or write data to the read FIFO. This code snippet
simply\n";
outFile << " -- transfer the data from the write FIFO to the read FIFO.\n";
outFile << " ----- \n";
outFile << " IP2RFIFO_WrMark      <= '0';\n";
outFile << " IP2RFIFO_WrRelease    <= '0';\n";
outFile << " IP2RFIFO_WrRestore    <= '0';\n";
outFile << " IP2WFIFO_RdMark       <= '0';\n";
outFile << " IP2WFIFO_RdRelease    <= '0';\n";
outFile << " IP2WFIFO_RdRestore    <= '0';\n";
outFile << " FIFO_CNTL_SM_COMB : process( WFIFO2IP_empty, WFIFO2IP_RdAck,
RFIFO2IP_full, RFIFO2IP_WrAck, fifo_cntl_cs, user_write_long, Bus2IP_Clk ) is\n";
outFile << " begin\n";
outFile << "     -- set defaults\n";
outFile << "     fifo_rdreq_cmb <= '0';\n";
outFile << "     fifo_wrreq_cmb <= '0';\n";
outFile << "     fifo_cntl_ns    <= fifo_cntl_cs;\n";
outFile << "     case fifo_cntl_cs is\n";
outFile << "         when IDLE =>\n";
outFile << "             -- data is available in the write fifo and there's space in
the read fifo,\n";
outFile << "             -- so we can start transferring the data from write fifo to
read fifo\n";
outFile << "             if ( WFIFO2IP_empty = '0' and RFIFO2IP_full = '0' ) then\n";
outFile << "                 fifo_rdreq_cmb <= '1';\n";
outFile << "                 fifo_cntl_ns    <= RD_REQ;\n";
outFile << "             end if;\n";
outFile << "         when RD_REQ =>\n";
outFile << "             -- data has been read from the write fifo,\n";
outFile << "             -- so we can write it to the read fifo\n";
outFile << "                 if (user_write = '1') then\n";
outFile << "                     if(user_write_long = '1') then\n";
outFile << "                         fifo_wrreq_cmb <= '1';\n";
outFile << "                         fifo_cntl_ns <= WR_REQ;\n";
outFile << "                     else\n";
outFile << "                         fifo_wrreq_cmb <= '1';\n";
outFile << "                         fifo_cntl_ns <= RD_REQ;\n";
outFile << "                     end if;\n";
outFile << "                 end if;\n";
outFile << "             when WR_REQ =>\n";
outFile << "                 -- data has been written to the read fifo,\n";

```

```

outFile << "          -- so data transfer is done\n";
outFile << "          if ( RFIFO2IP_WrAck = '1' ) then\n";
outFile << "              fifo_cntl_ns <= IDLE;\n";
outFile << "          end if;\n";
outFile << "          when others =>\n";
outFile << "              fifo_cntl_ns <= IDLE;\n";
outFile << "          end case;\n\n\n";
outFile << "      end process FIFO_CNTL_SM_COMB;\n\n";
outFile << "      FIFO_CNTL_SM_SEQ : process( Bus2IP_Clk ) is\n";
outFile << "      begin\n\n";
outFile << "          if ( Bus2IP_Clk'event and Bus2IP_Clk = '1' ) then\n";
outFile << "              if (Bus2IP_Reset = '1') then\n";
outFile << "                  IP2WFIFO_RdReq <= '0';\n";
outFile << "                  IP2RFIFO_WrReq <= '0';\n";
outFile << "                  fifo_cntl_cs  <= IDLE;\n";
outFile << "              else\n";
outFile << "                  IP2WFIFO_RdReq <= fifo_rdreq_cmb;\n";
outFile << "                  IP2RFIFO_WrReq <= fifo_wrreq_cmb;\n";
outFile << "                  fifo_cntl_cs  <= fifo_cntl_ns;\n";
outFile << "              end if;\n";
outFile << "          end if;\n";
outFile << "\n";
outFile << "      end process FIFO_CNTL_SM_SEQ;\n";
outFile << "\n";
outFile << "      --IP2RFIFO_Data <= WFIFO2IP_Data;\n";
outFile << "\n";
outFile << "      -----\n";
outFile << "      -- Example code to drive IP to Bus signals\n";
outFile << "      -----\n";
outFile << "      IP2Bus_Data  <= (others => '0');\n\n";
outFile << "      IP2Bus_WrAck <= '0';\n";
outFile << "      IP2Bus_RdAck <= '0';\n";
outFile << "      IP2Bus_Error <= '0';\n\n";
outFile << "end IMP;\n";
}
void writePPCcode(string cCode, string Name)
{
    ofstream outFile(cCode.c_str());
    int len = strlen(Name.c_str());
    for (int i = 0; i < len; i++)
    {
        Name[i] = toupper( (unsigned char) Name[i] );
    }

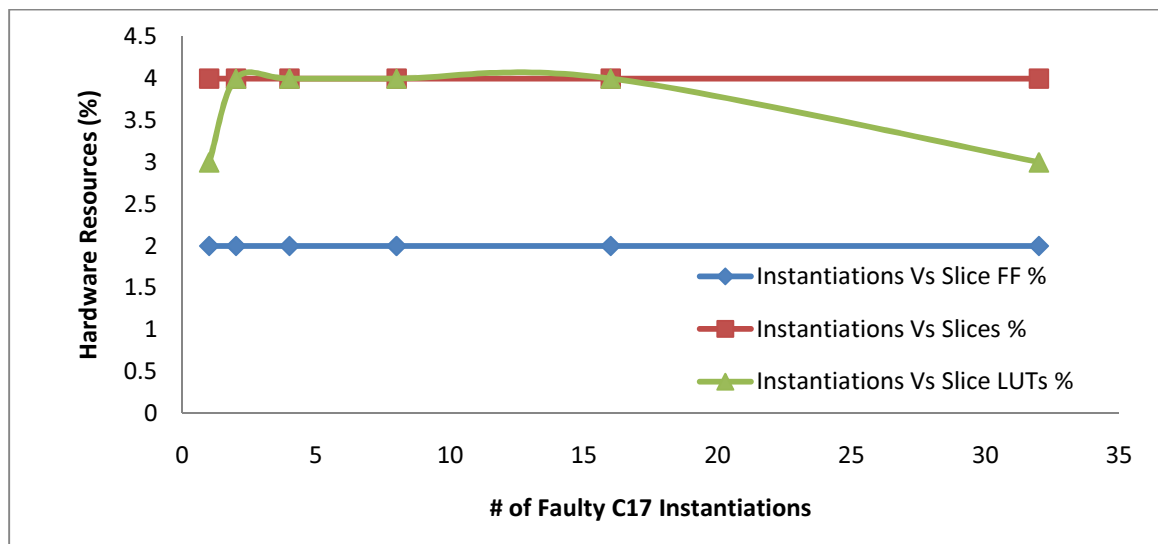
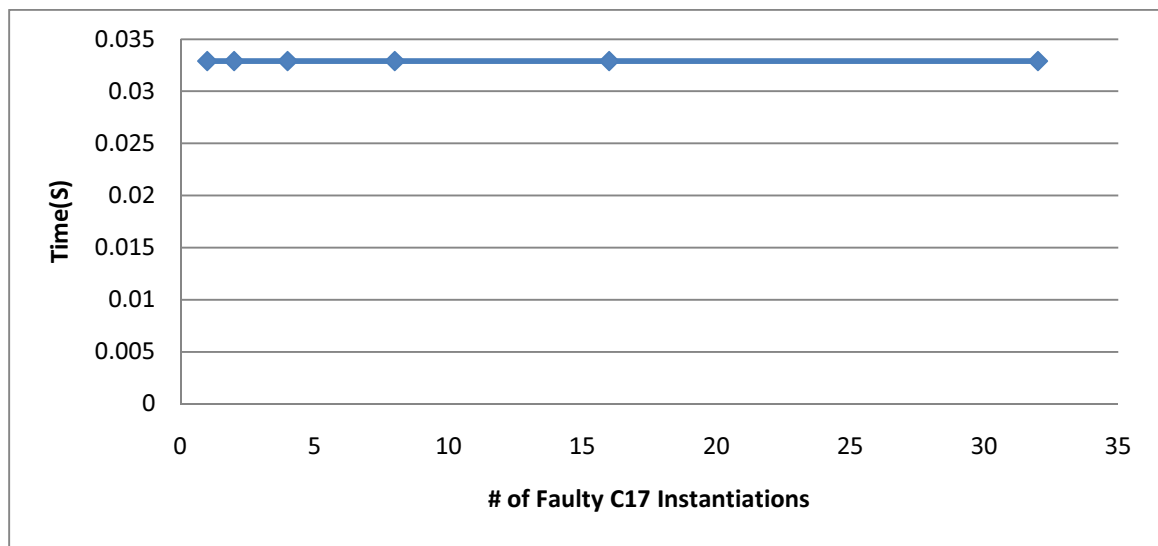
    if(!outFile){
        cout << "Unable to open file 8.txt";
        exit(1);
    }
    cout << Name << endl;
    outFile << "#include \"xparameters.h\"\n";
    outFile << "#include \"xbasic_types.h\"\n";
    outFile << "#include \"xstatus.h\"\n";
    outFile << "#include \"MY_\" << Name << ".h\"\n";
    outFile << "#include \"stdlib.h\"\n";
    outFile << "#include \"stdio.h\"\n";
    outFile << "#include \"xtime_l.h\"\n";
    outFile << "#include \"xpseudo_asm.h\"\n\n";
    outFile << "Xuint32 *baseaddr_p = (Xuint32 *)XPAR_MY_\" << Name <<
    \"_0_BASEADDR;\n";
    outFile << "Xuint8 dec2bin(Xuint32, Xuint32);\n";
    outFile << "int maxrow(Xuint8 *Binary, Xuint32 *rowsums);\n\n";
    outFile << "int main (void) {\n";
    outFile << "    Xuint64 starttime, endtime, elapsedtime;\n";
    outFile << "    XTime_SetTime(0);\n";
    outFile << "    endtime.Upper = mfspr(XREG_SPR_TBU_READ);\n";
    outFile << "    endtime.Lower = mfspr(XREG_SPR_TBL_READ);\n";
    outFile << "    xil_printf(\"Time: %8x\", XUIN64_MSW(endtime));\n";

```


Appendix 3: Variable Instantiation Data

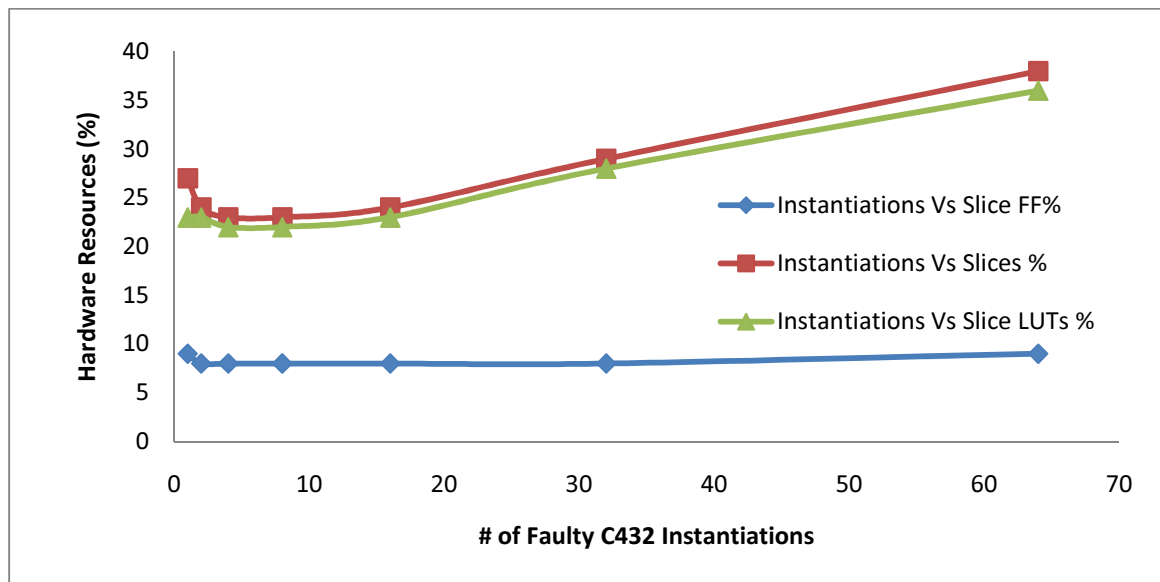
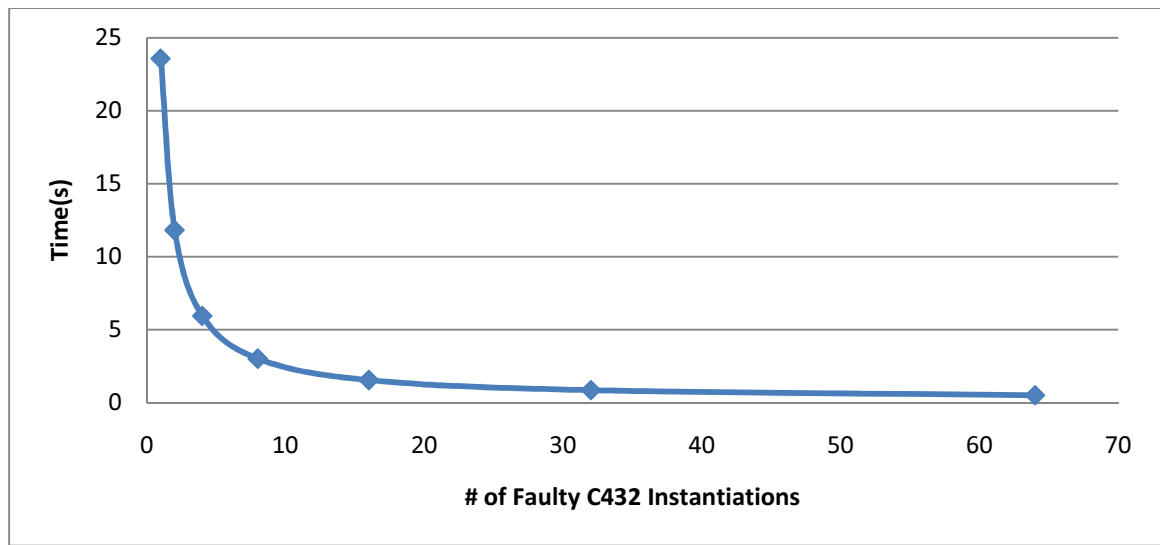
C17

Instantiations	Time(sec)	Slices %	Slice FF %	Slice LUTs %	Speed MHz
1	0.0329	4	2	3	25
2	0.0329	4	2	4	25
4	0.0329	4	2	4	25
8	0.0329	4	2	4	25
16	0.0329	4	2	4	25
32	0.0329	4	2	3	25



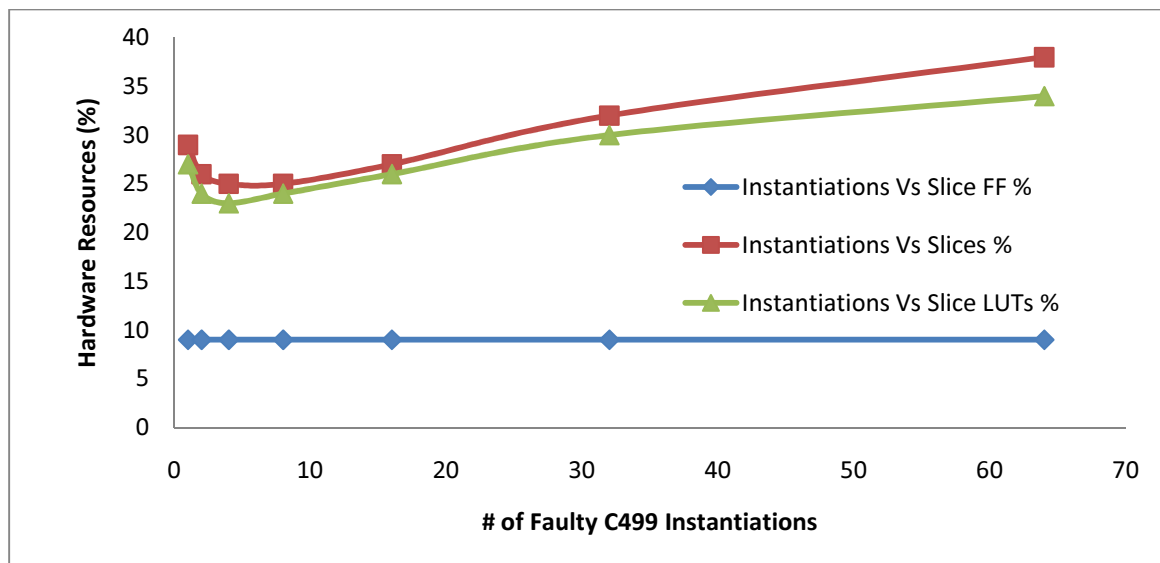
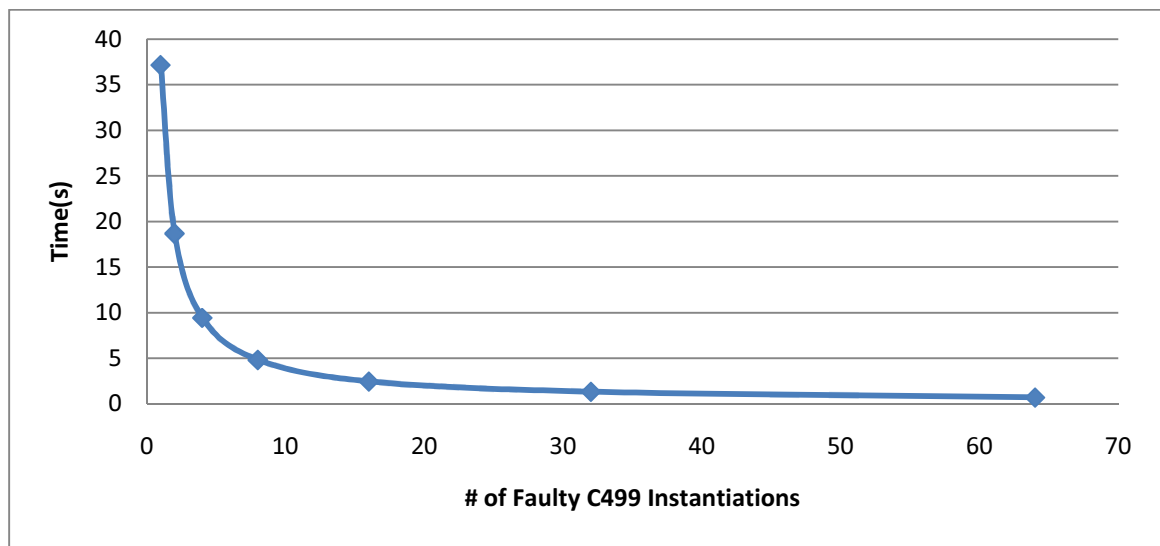
C432

Instantiations	Time(sec)	Slices %	Slice FF %	Slice LUTs %	Speed MHz
1	23.56	27	9	23	25
2	11.81	24	8	23	25
4	5.93	23	8	22	25
8	3	23	8	22	25
16	1.53	24	8	23	25
32	0.84	29	8	28	25
64	0.49	38	9	36	25



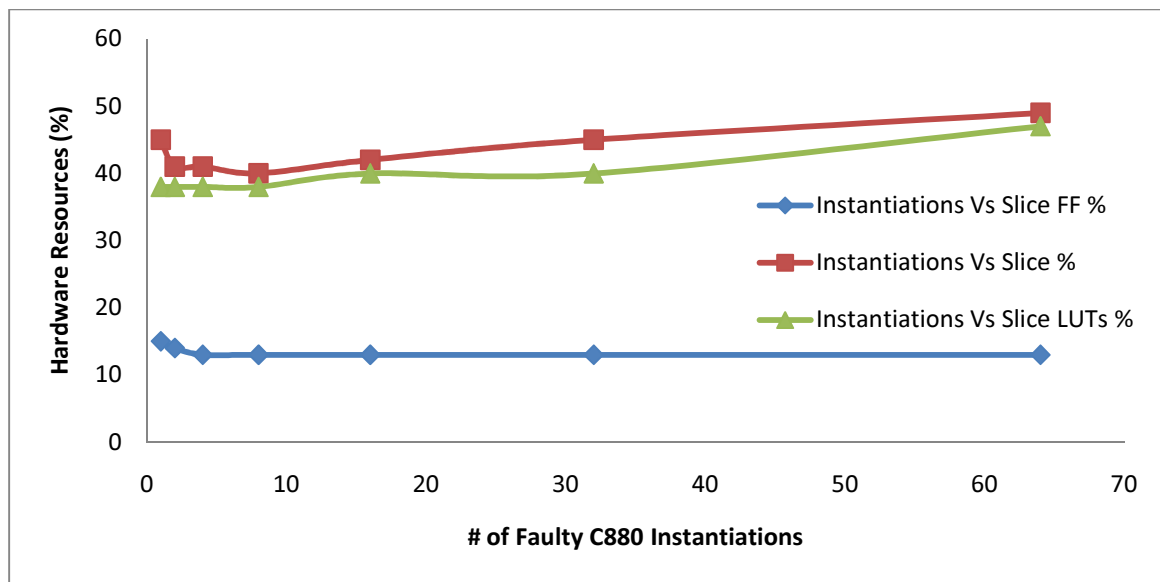
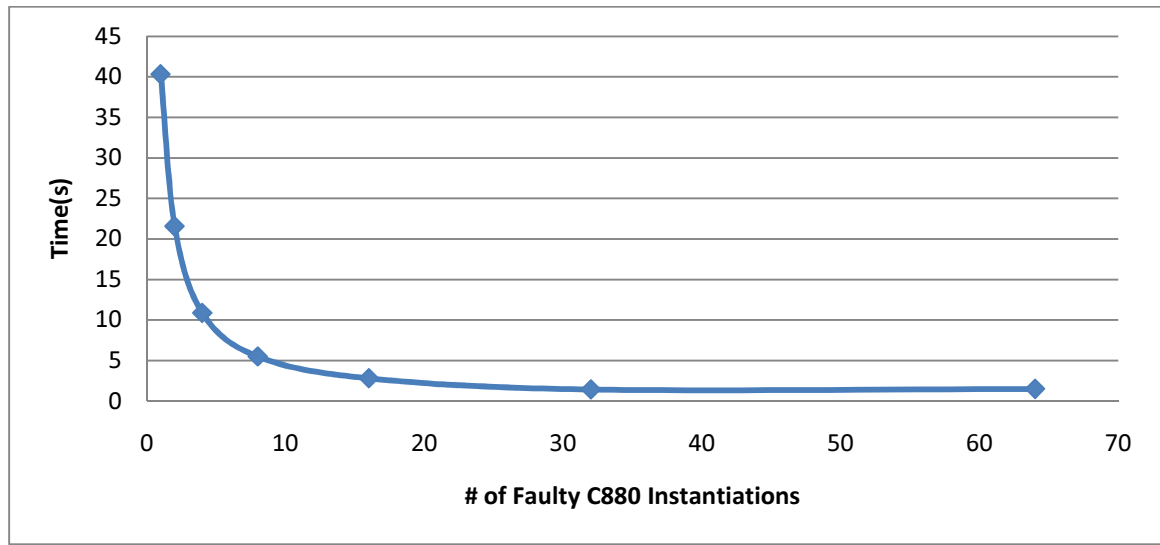
C499

Instantiations	Time(sec)	Slices %	Slice FF %	Slice LUTs %	Speed MHz
1	37.15	29	9	27	25
2	18.68	26	9	24	25
4	9.44	25	9	23	25
8	4.83	25	9	24	25
16	2.46	27	9	26	25
32	1.33	32	9	30	25
64	0.71	38	9	34	25



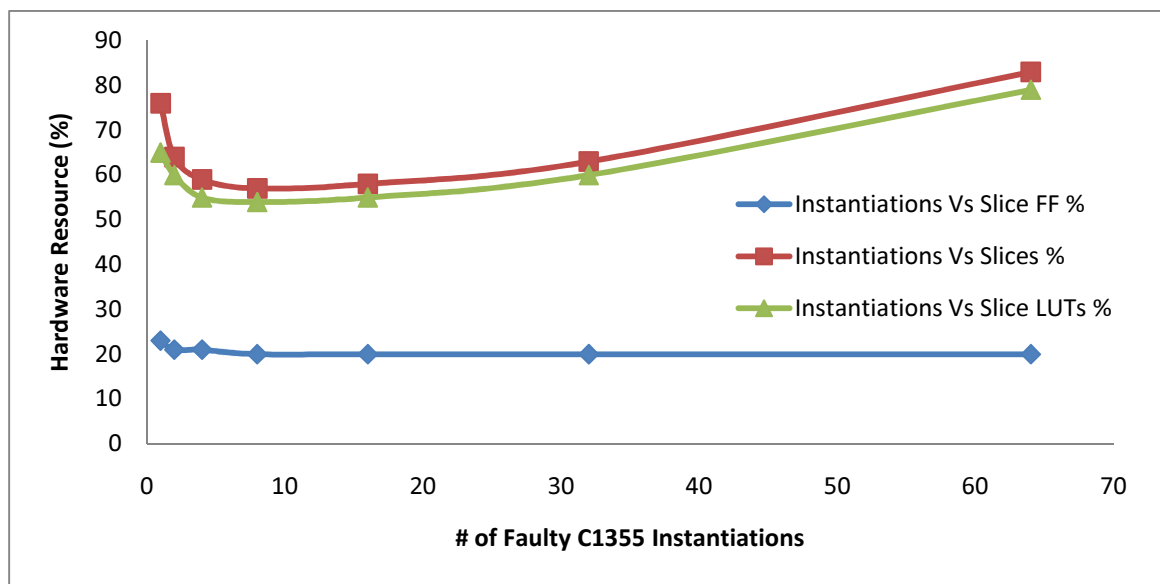
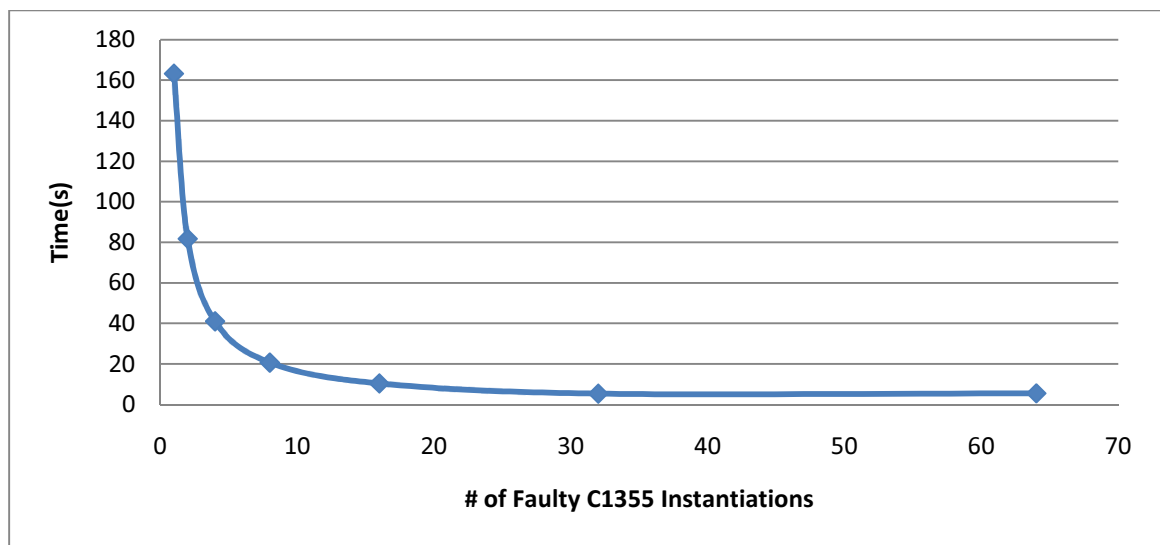
C880

Instantiations	Time(sec)	Slices %	Slice FF %	Slice LUTs %	Speed MHz
1	40.31	45	15	38	25
2	21.57	41	14	38	25
4	10.86	41	13	38	25
8	5.5	40	13	38	25
16	2.82	42	13	40	25
32	1.44	45	13	40	25
64	1.5	49	13	47	12.5



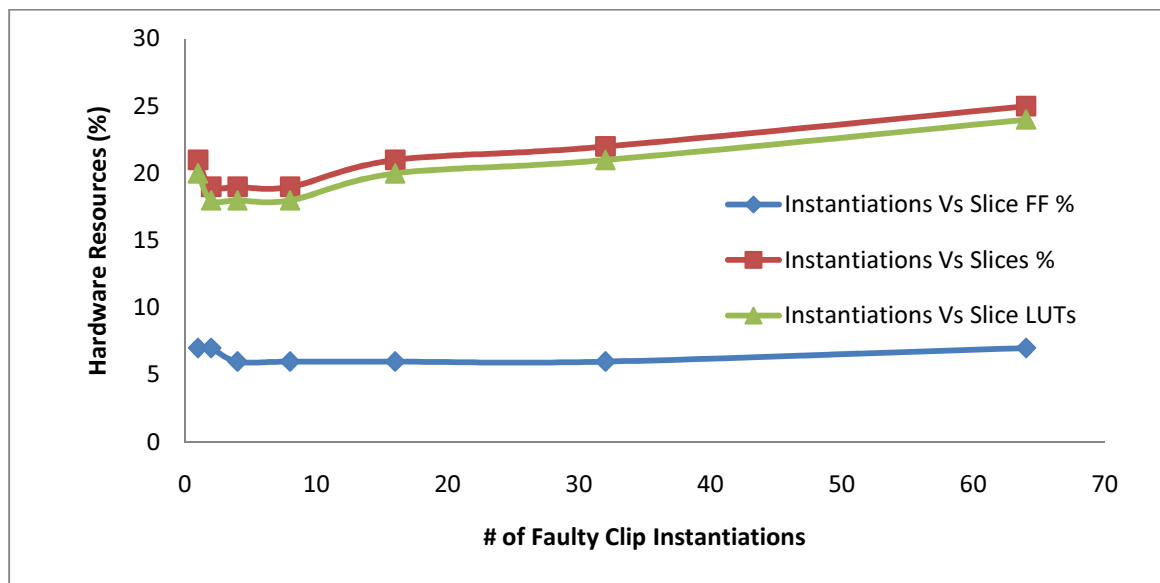
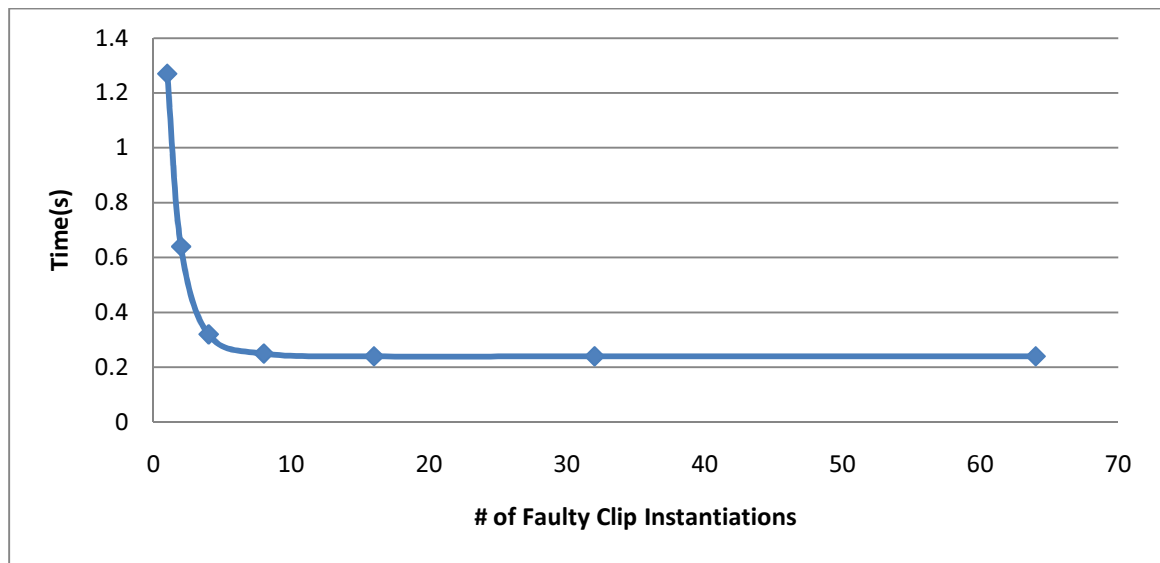
C1355

Instantiations	Time(sec)	Slices %	Slice FF %	Slice LUTs %	Speed MHz
1	163.23	76	23	65	25
2	81.79	64	21	60	25
4	41.06	59	21	55	25
8	20.7	57	20	54	25
16	10.42	58	20	55	25
32	5.38	63	20	60	25
64	5.51	83	20	79	12.5



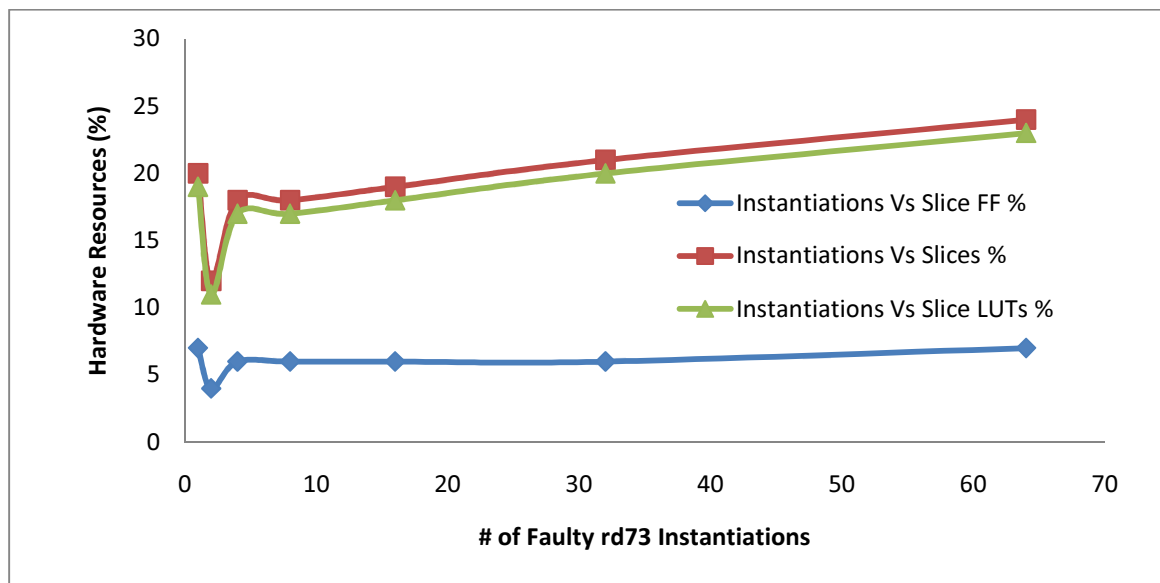
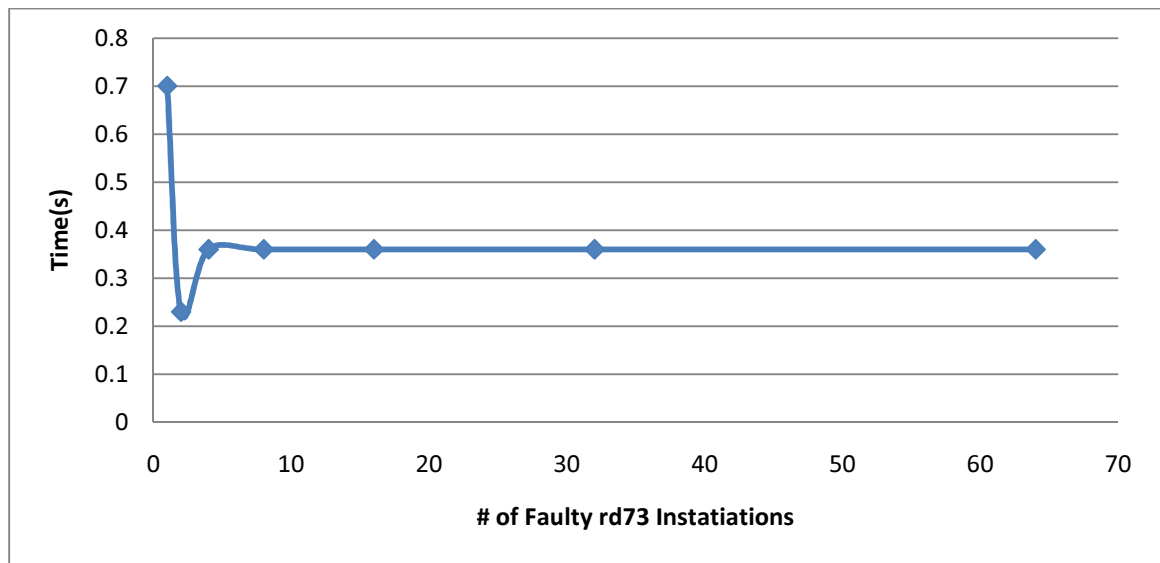
Clip

Instantiations	Time(sec)	Slices %	Slice FF %	Slice LUTs %	Speed MHz
1	1.27	21	7	20	25
2	0.64	19	7	18	25
4	0.32	19	6	18	25
8	0.25	19	6	18	25
16	0.24	21	6	20	25
32	0.24	22	6	21	25
64	0.24	25	7	24	25



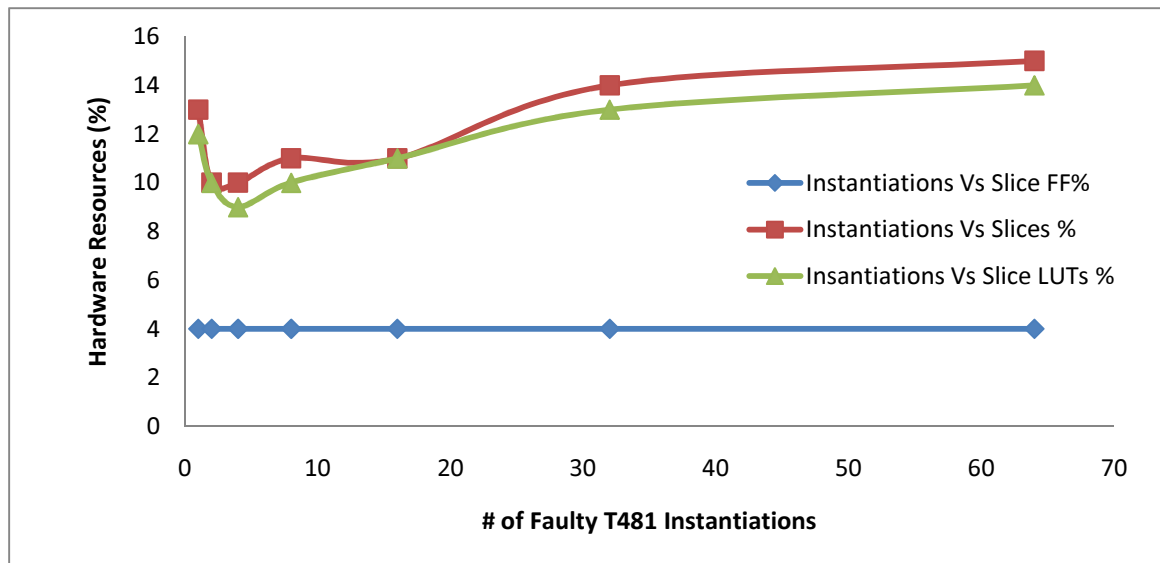
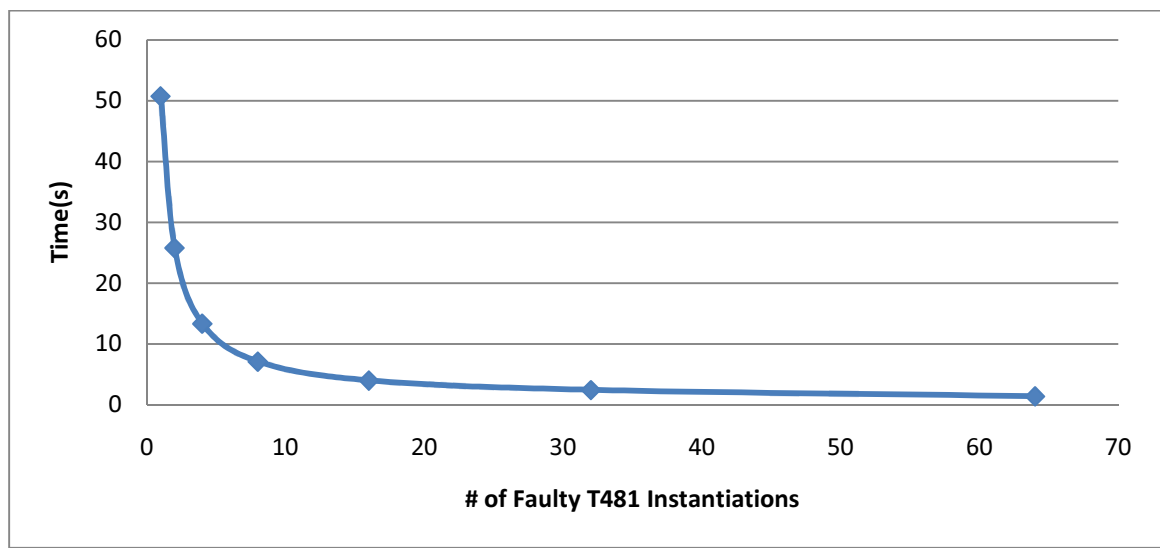
rd73

Instantiations	Time(sec)	Slices %	Slice FF %	Slice LUTs %	Speed MHz
1	0.7	20	7	19	25
2	0.23	12	4	11	25
4	0.36	18	6	17	25
8	0.36	18	6	17	25
16	0.36	19	6	18	25
32	0.36	21	6	20	25
64	0.36	24	7	23	25



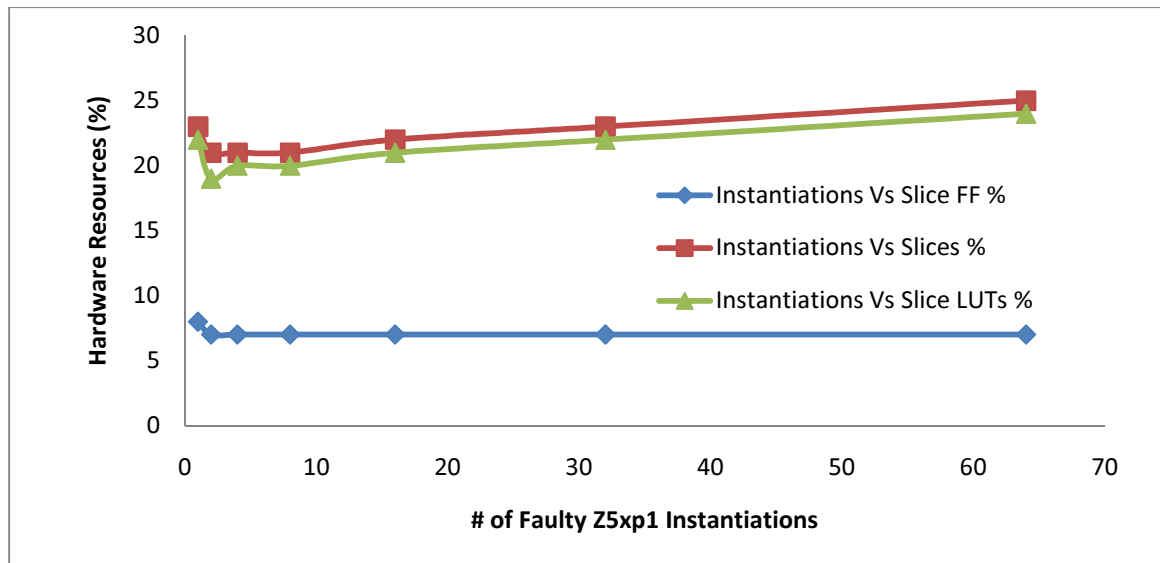
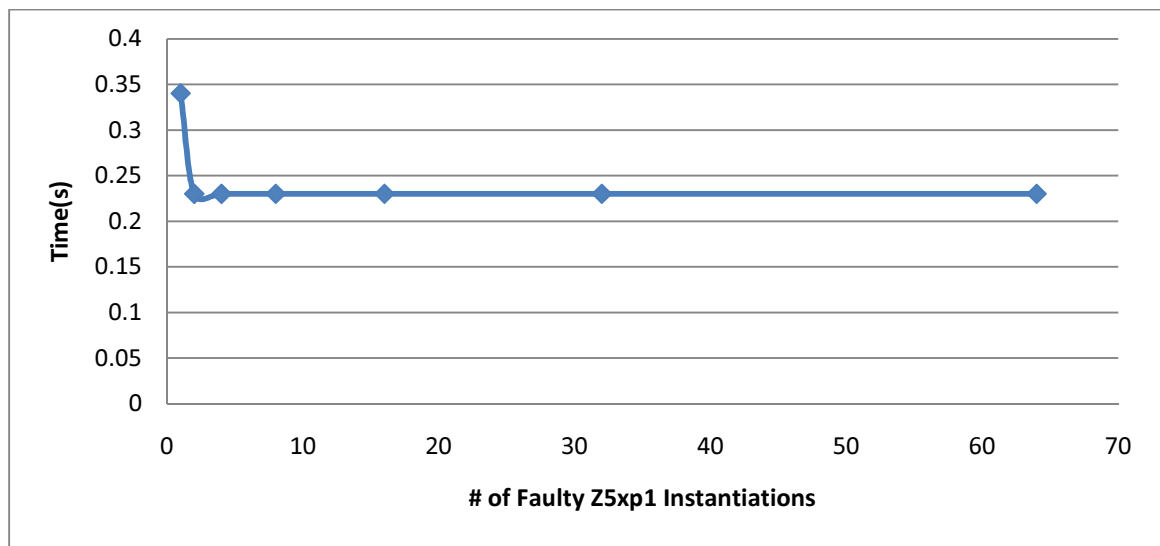
T481

Instantiations	Time(sec)	Slices %	Slice FF %	Slice LUTs %	Speed MHz
1	50.69	13	4	12	25
2	25.78	10	4	10	25
4	13.32	10	4	9	25
8	7.09	11	4	10	25
16	3.98	11	4	11	25
32	2.42	14	4	13	25
64	1.38	15	4	14	25



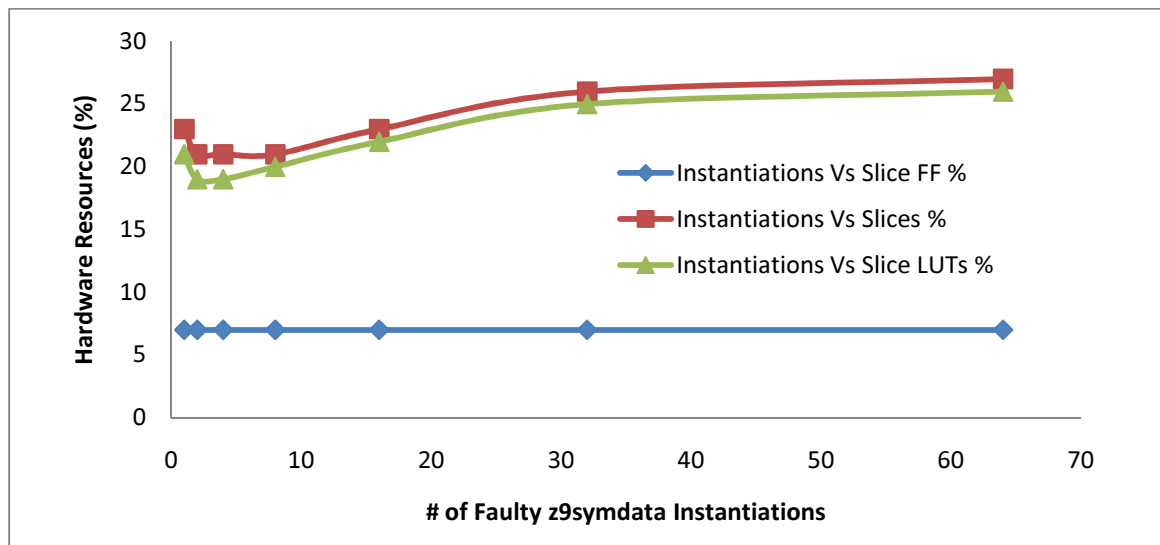
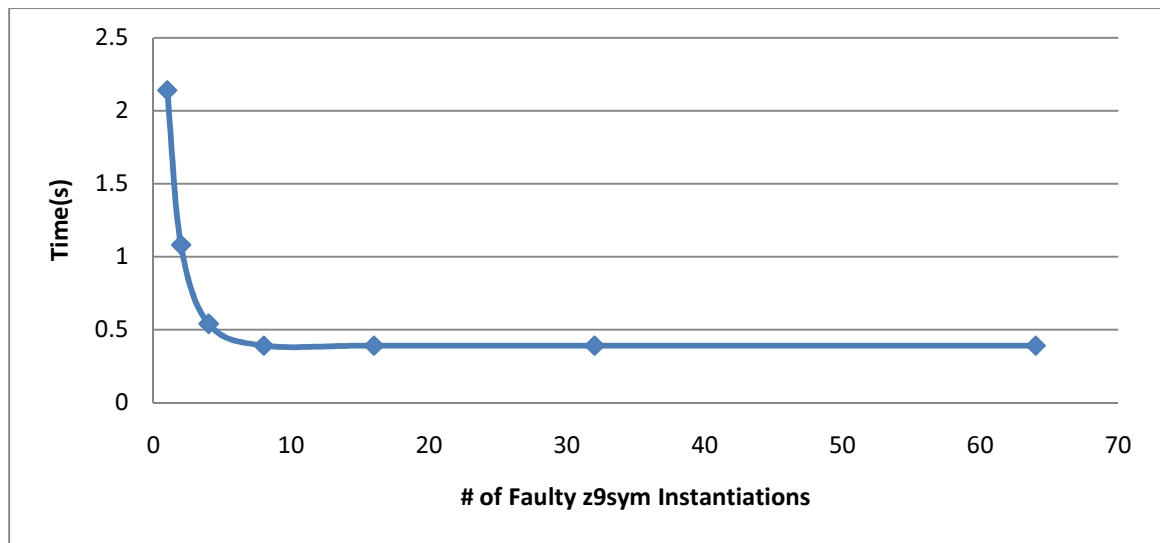
Z5xp1

Instantiations	Time(sec)	Slices %	Slice FF %	Slice LUTs %	Speed MHz
1	0.34	23	8	22	25
2	0.23	21	7	19	25
4	0.23	21	7	20	25
8	0.23	21	7	20	25
16	0.23	22	7	21	25
32	0.23	23	7	22	25
64	0.23	25	7	24	25



Z9sym

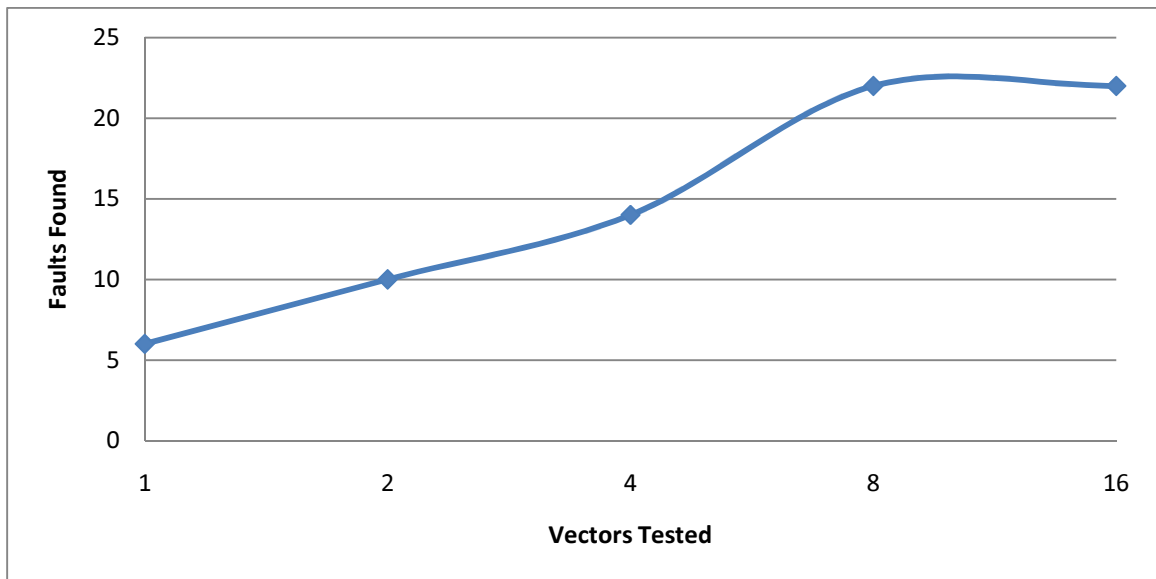
Instantiations	Time(sec)	Slices %	Slice FF %	Slice LUTs %	Speed MHz
1	2.14	23	7	21	25
2	1.08	21	7	19	25
4	0.54	21	7	19	25
8	0.39	21	7	20	25
16	0.39	23	7	22	25
32	0.39	26	7	25	25
64	0.39	27	7	26	25

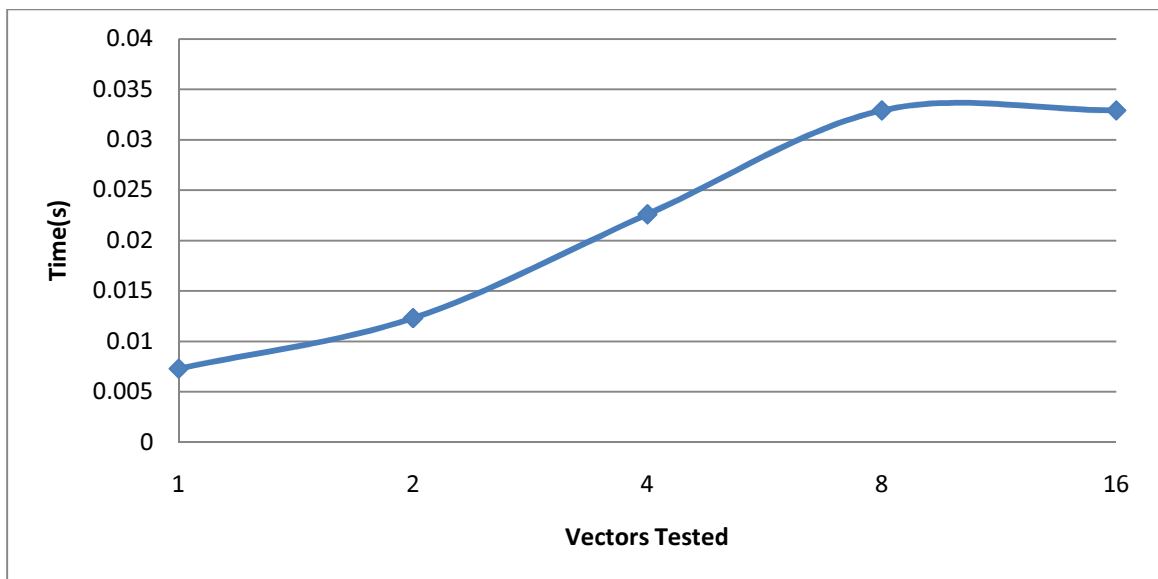
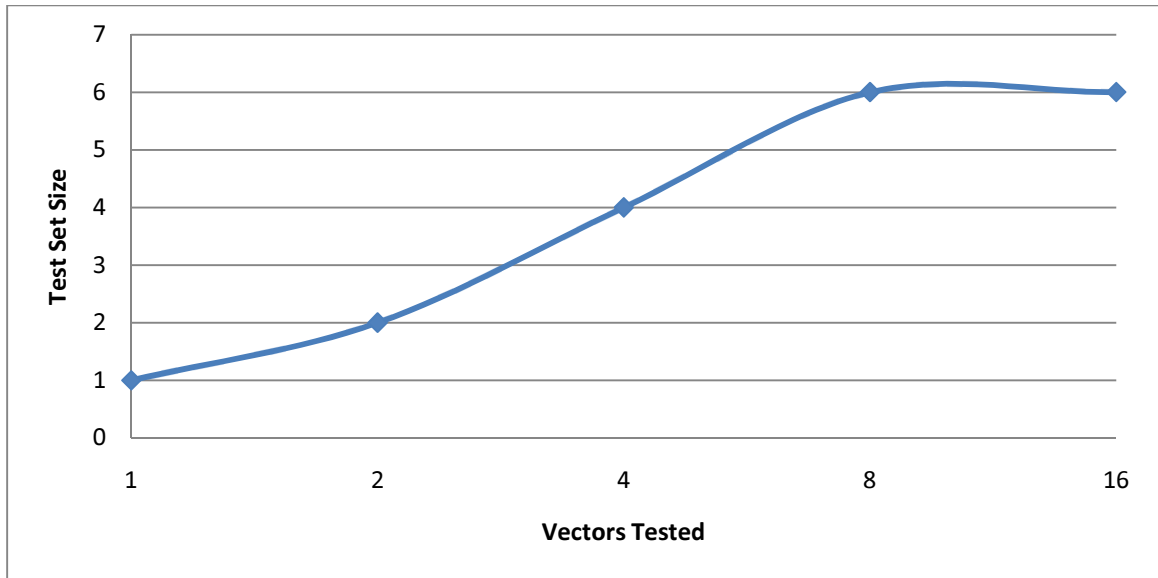


Appendix 4: Variable Input Test Vector Data

C17

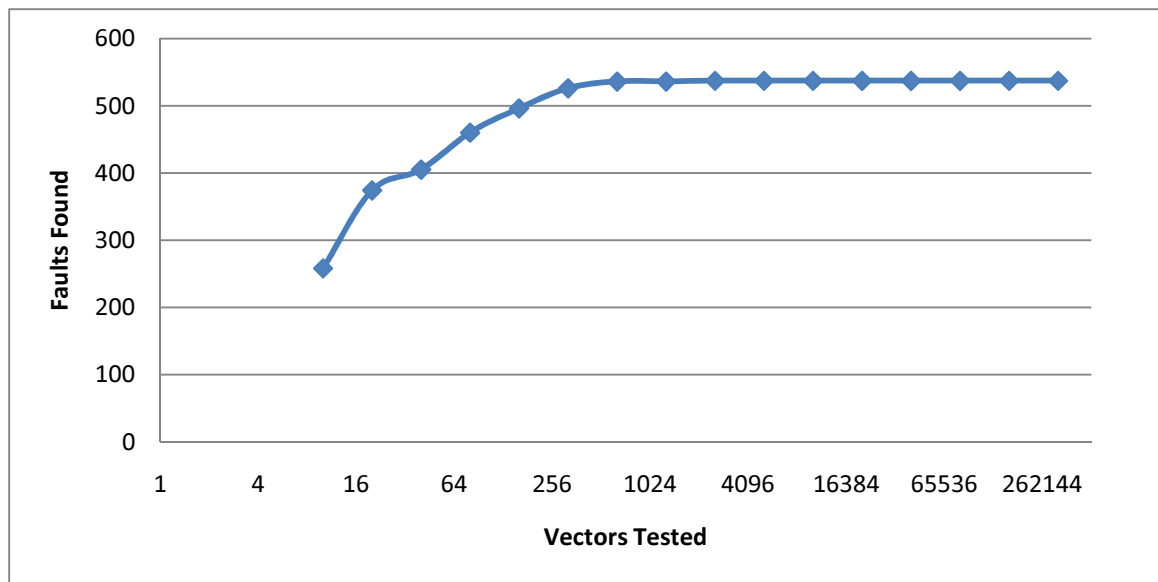
Vectors Tested	Faults Found	Number of Vectors Needed	Time
1	6	1	0.00728
2	10	2	0.0123
4	14	4	0.0226
8	22	6	0.0329
16	22	6	0.0329
32	22	6	0.0329

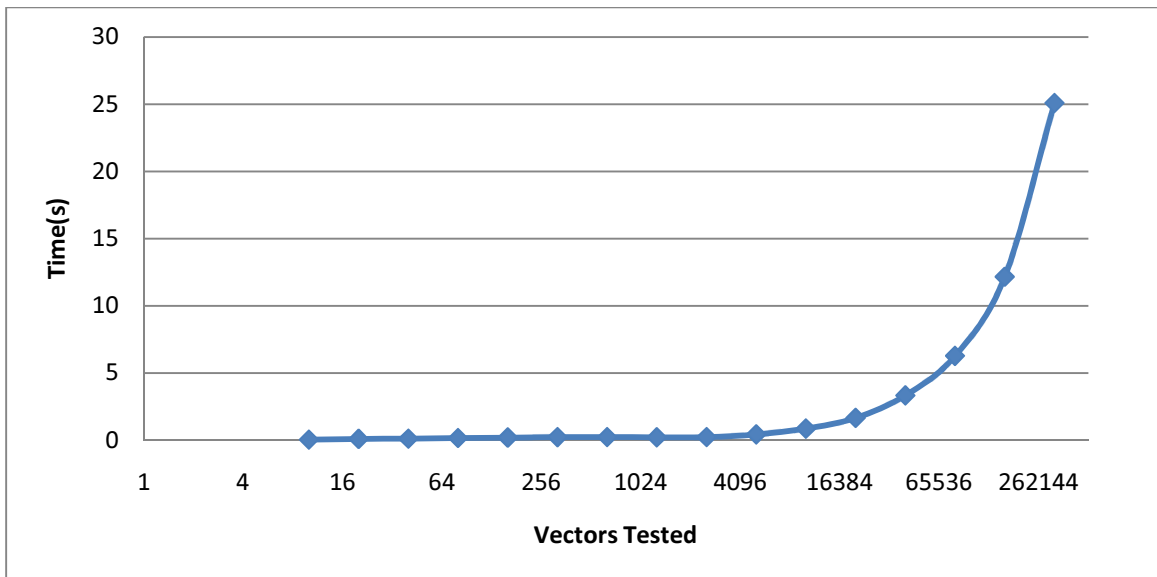
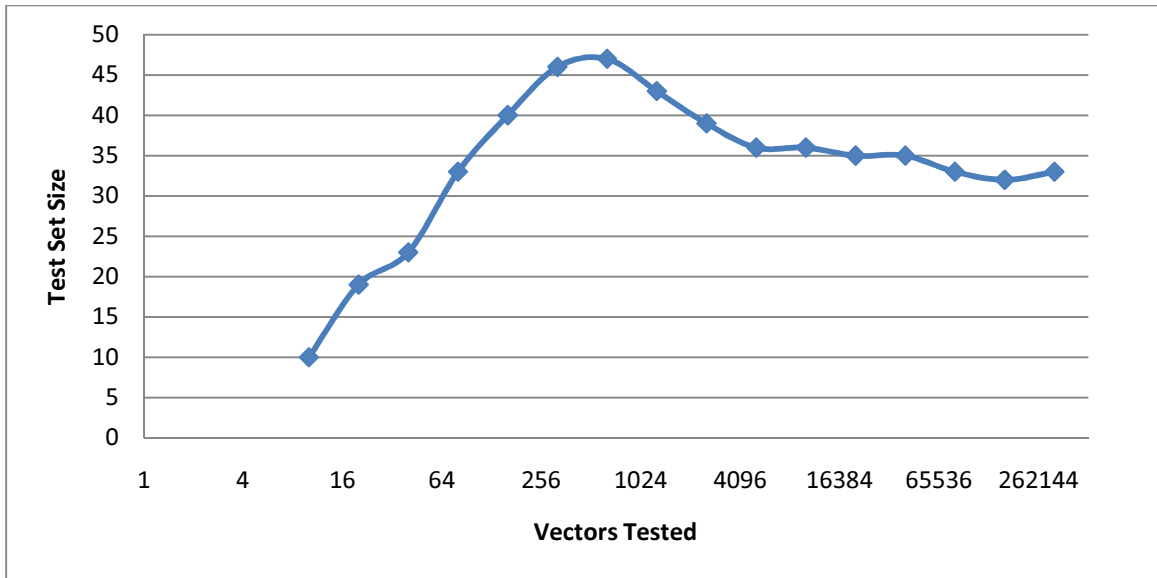




C432

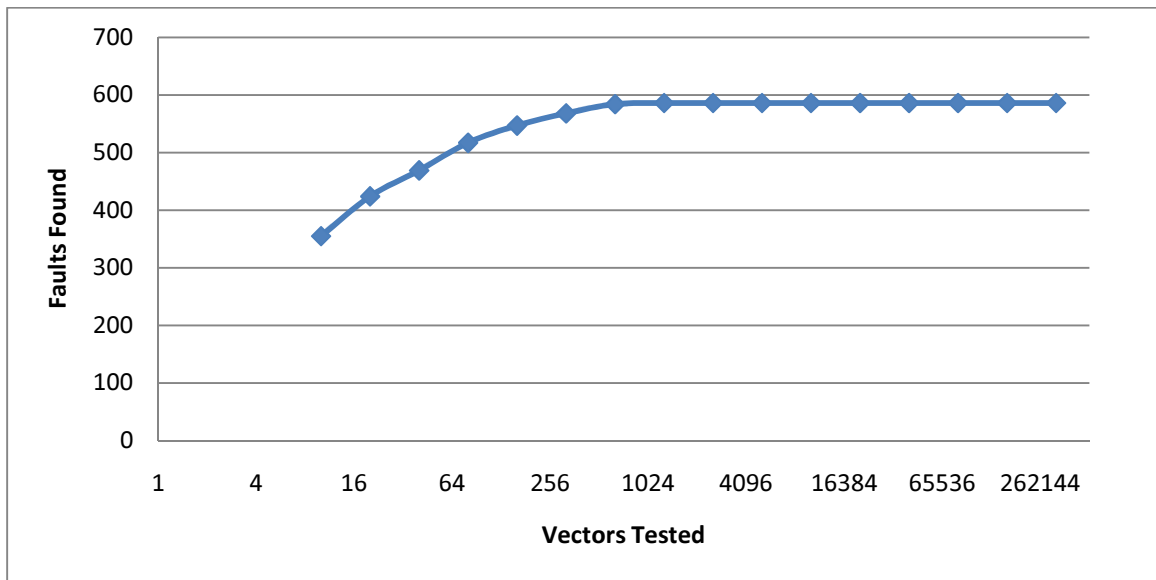
Vectors Tested	Faults Found	Number of Vectors Needed	Time
10	258	10	0.04
20	374	19	0.09
40	405	23	0.11
80	460	33	0.16
160	496	40	0.19
320	526	46	0.22
640	536	47	0.23
1280	536	43	0.21
2560	537	39	0.23
5120	537	36	0.43
10240	537	36	0.86
20480	537	35	1.66
40960	537	35	3.33
81920	537	33	6.27
163840	537	32	12.16
327680	537	33	25.09

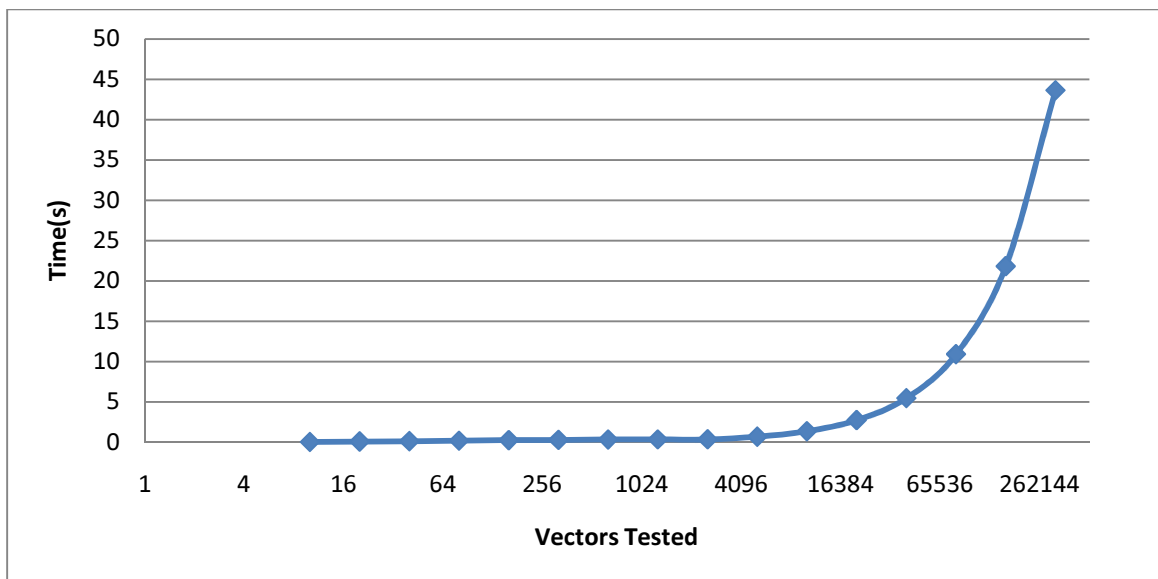
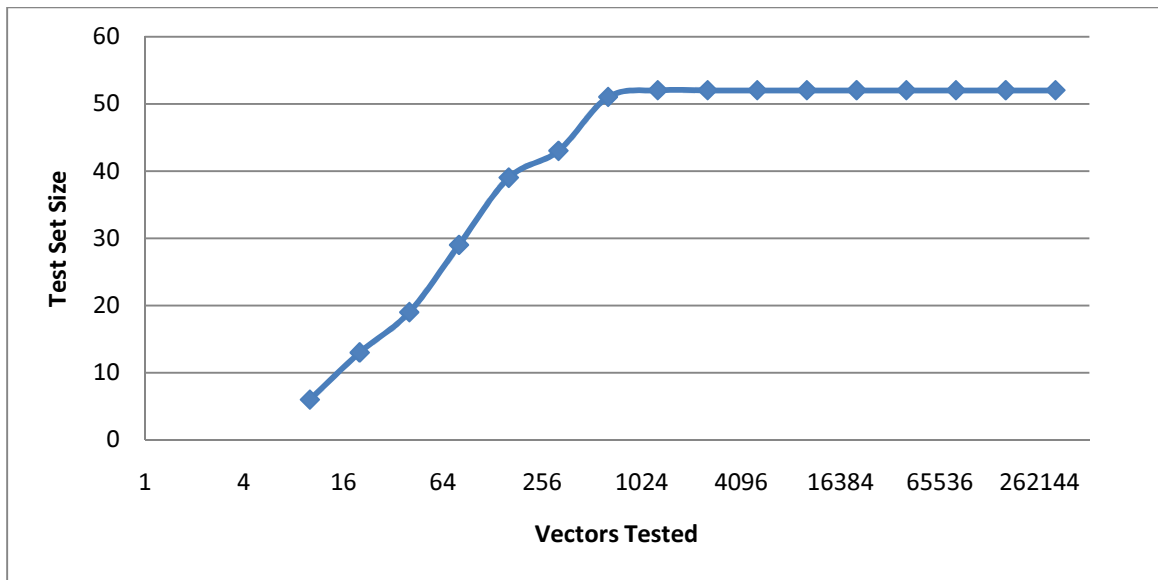




C499

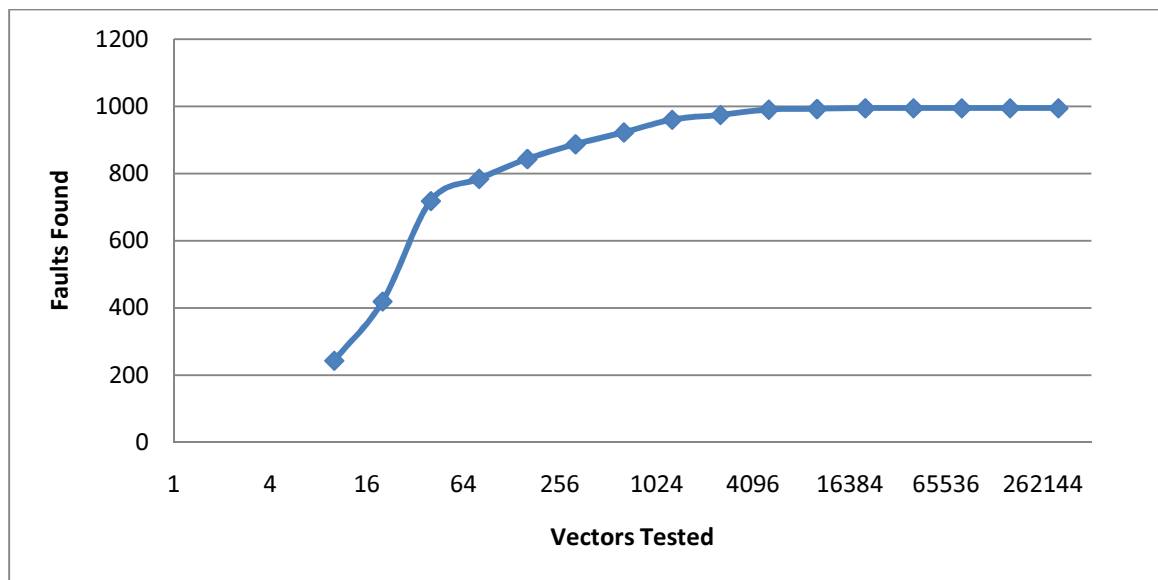
Vectors Tested	Faults Found	Number of Vectors Needed	Time
10	355	6	0.03
20	424	13	0.08
40	469	19	0.12
80	517	29	0.18
160	547	39	0.25
320	568	43	0.28
640	584	51	0.33
1280	586	52	0.34
2560	586	52	0.35
5120	586	52	0.68
10240	586	52	1.36
20480	586	52	2.73
40960	586	52	5.45
81920	586	52	10.91
163840	586	52	21.81
327680	586	52	43.61

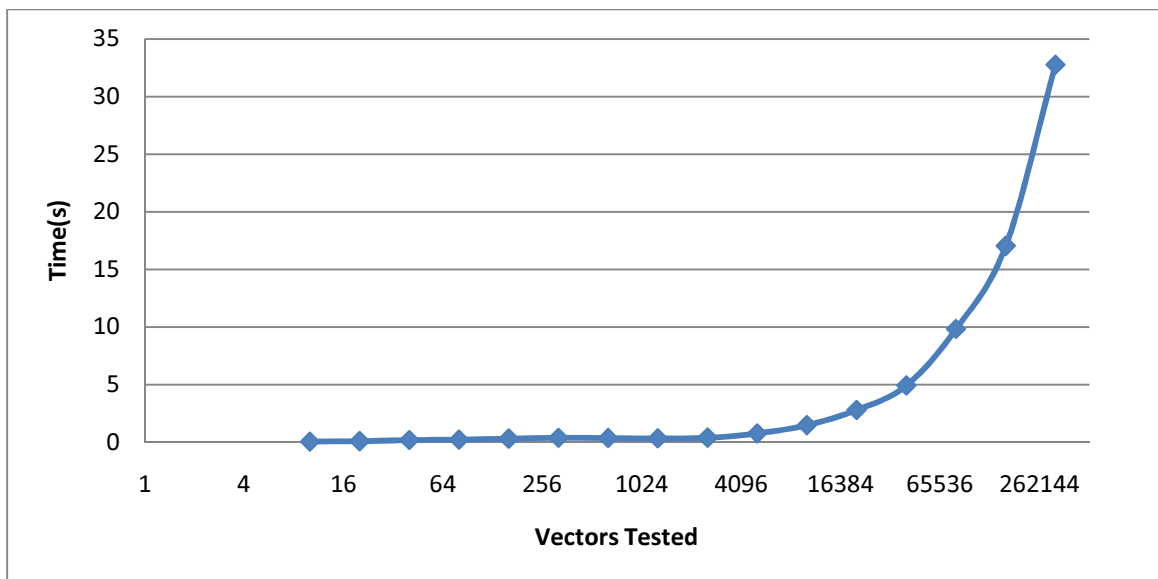
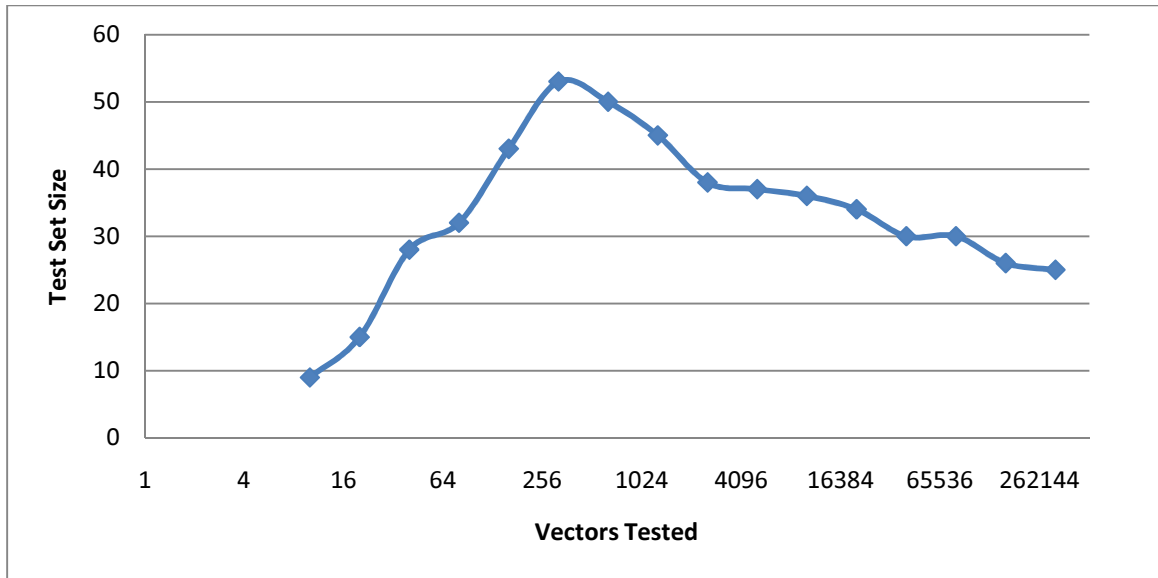




C880

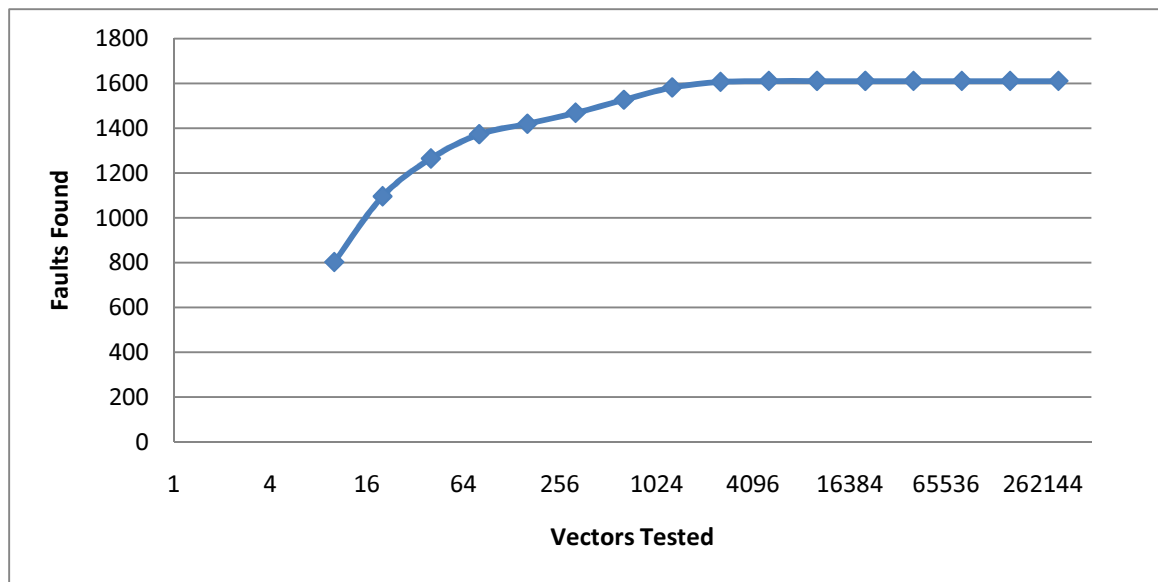
Vectors Tested	Faults Found	Number of Vectors Needed	Time
10	243	9	0.05
20	419	15	0.09
40	718	28	0.19
80	784	32	0.22
160	843	43	0.31
320	887	53	0.38
640	922	50	0.36
1280	960	45	0.33
2560	974	38	0.39
5120	990	37	0.76
10240	992	36	1.48
20480	994	34	2.79
40960	994	30	4.92
81920	994	30	9.83
163840	994	26	17.04
327680	994	25	32.77

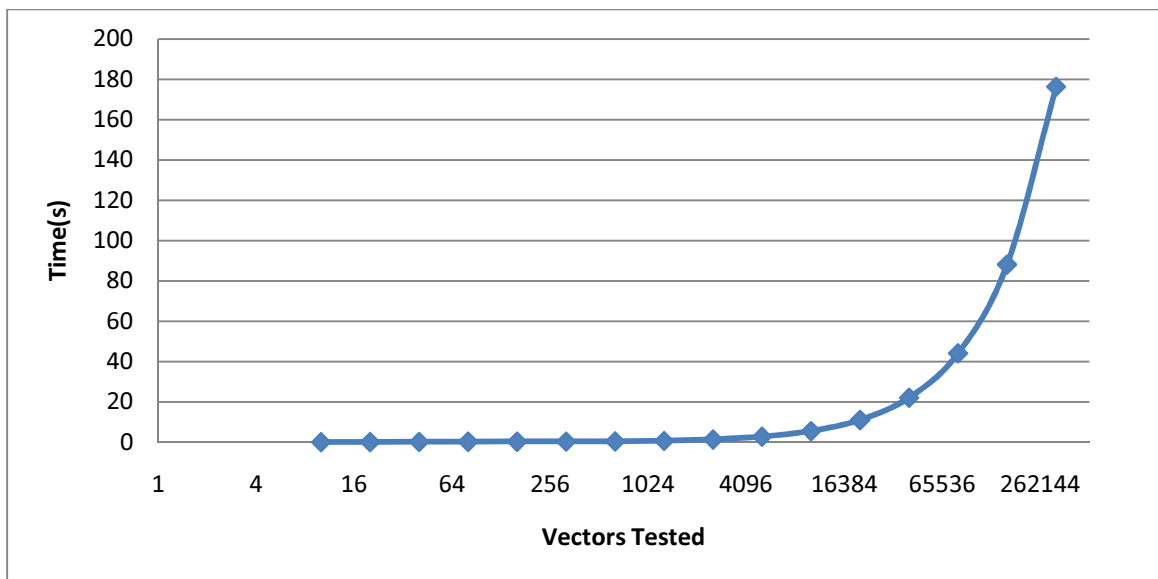




C1355

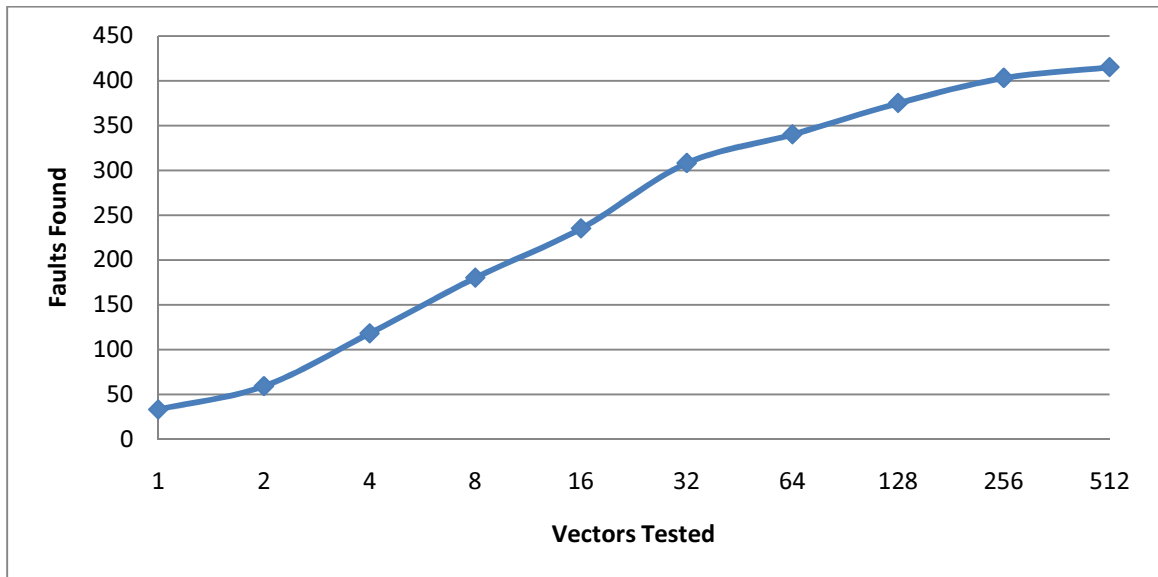
Vectors Tested	Faults Found	Number of Vectors Needed	Time
10	802	6	0.03
20	1095	15	0.09
40	1264	22	0.14
80	1372	33	0.22
160	1419	42	0.28
320	1468	49	0.33
640	1526	61	0.41
1280	1581	76	0.62
2560	1606	83	1.36
5120	1610	84	2.75
10240	1610	84	5.51
20480	1610	84	11.01
40960	1610	84	22.02
81920	1610	84	44.06
163840	1610	84	88.09
327680	1610	84	176.17

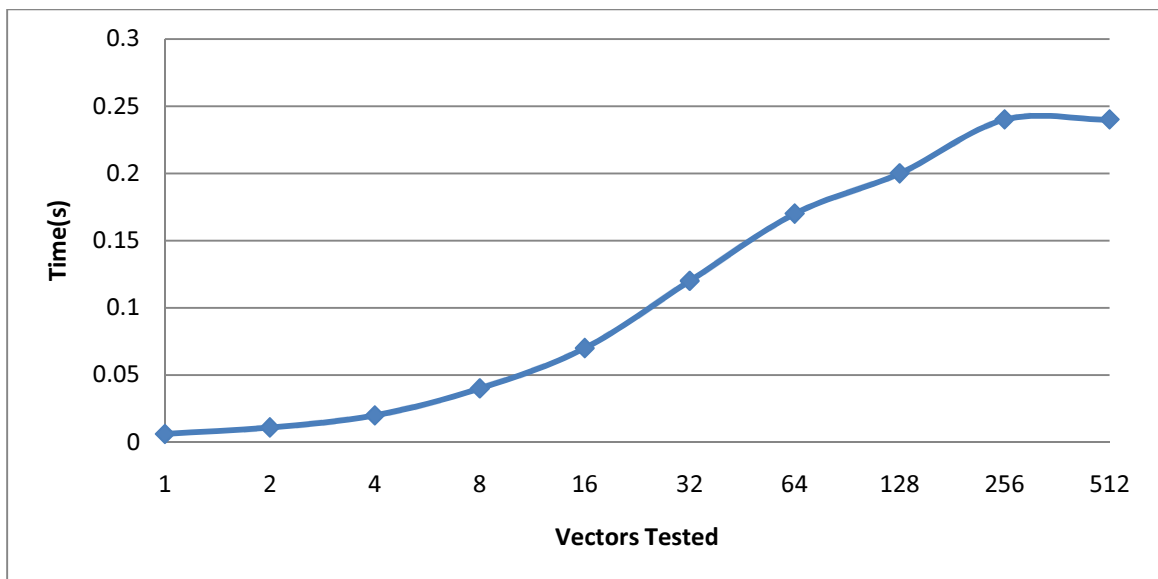
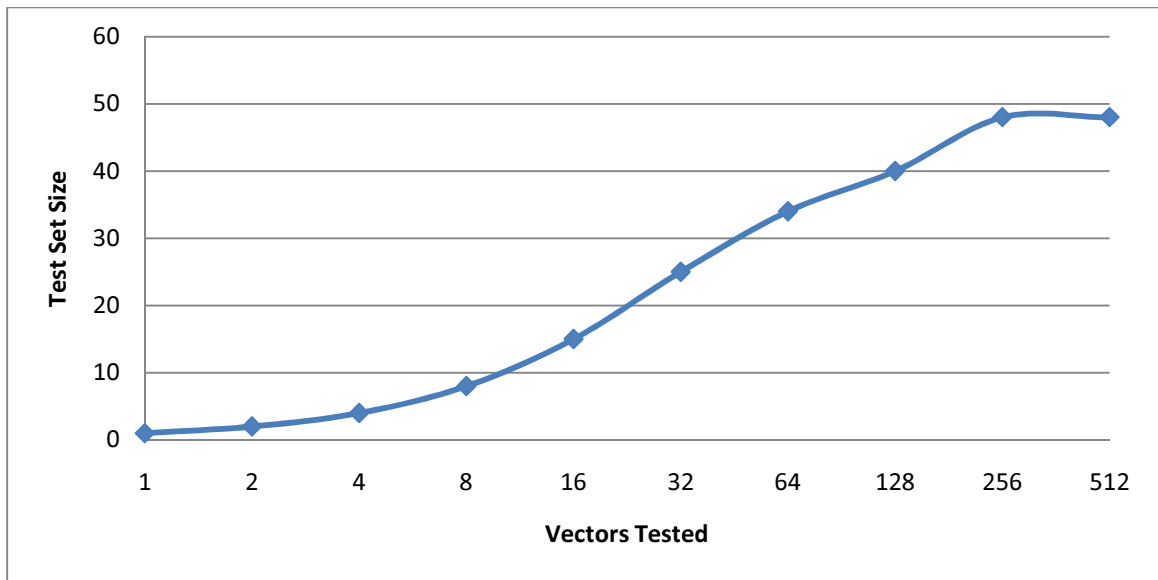




Clip

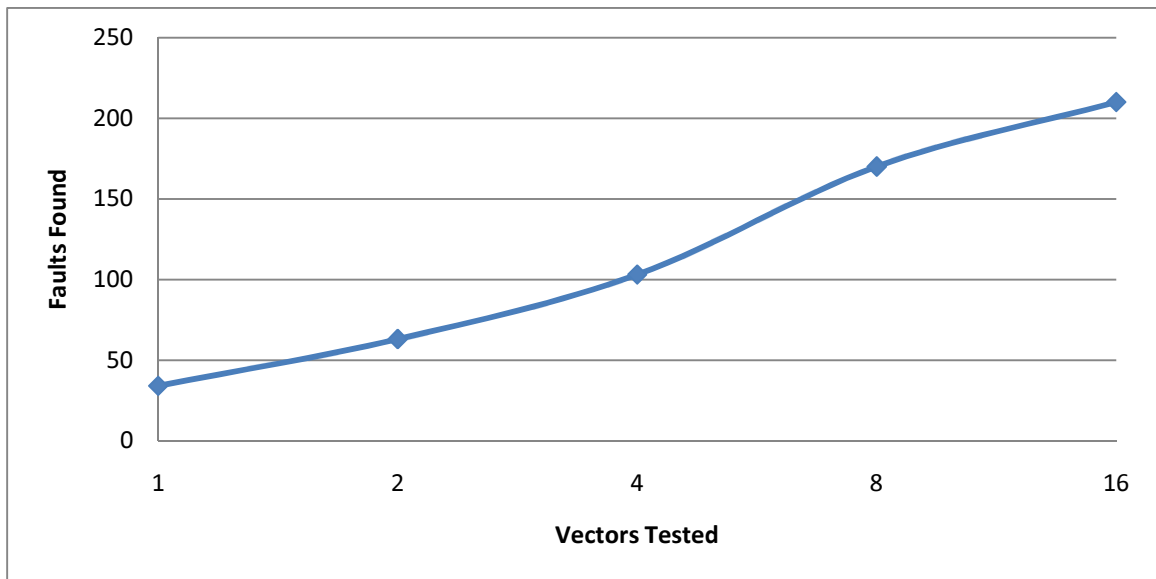
Vectors Tested	Faults Found	Number of Vectors Needed	Time
1	33	1	0.0062
2	59	2	0.011
4	118	4	0.02
8	180	8	0.04
16	235	15	0.07
32	308	25	0.12
64	340	34	0.17
128	375	40	0.2
256	403	48	0.24
512	415	48	0.24

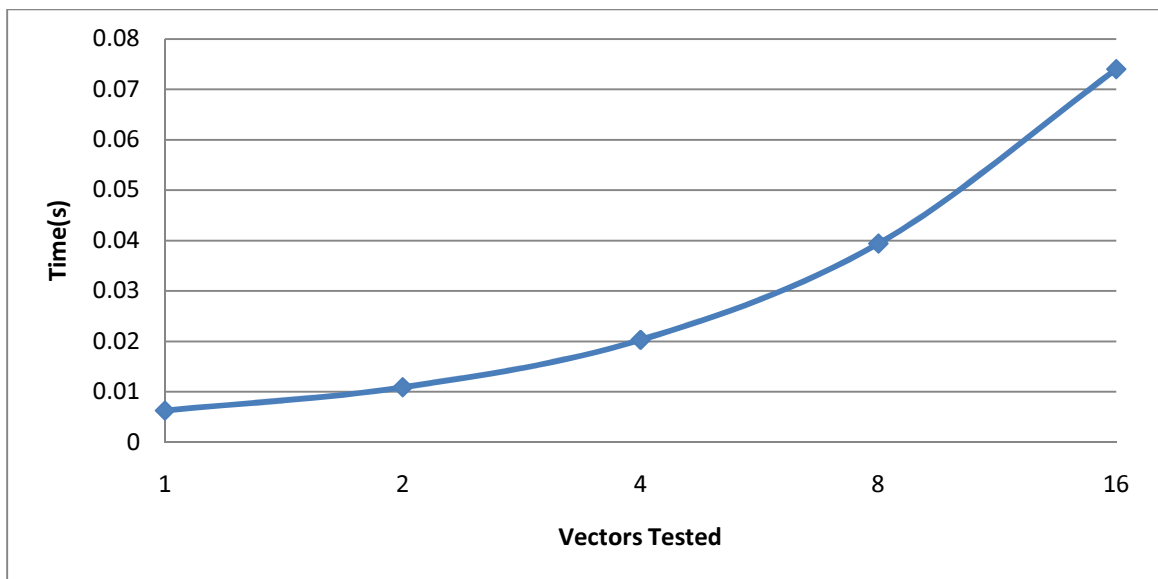
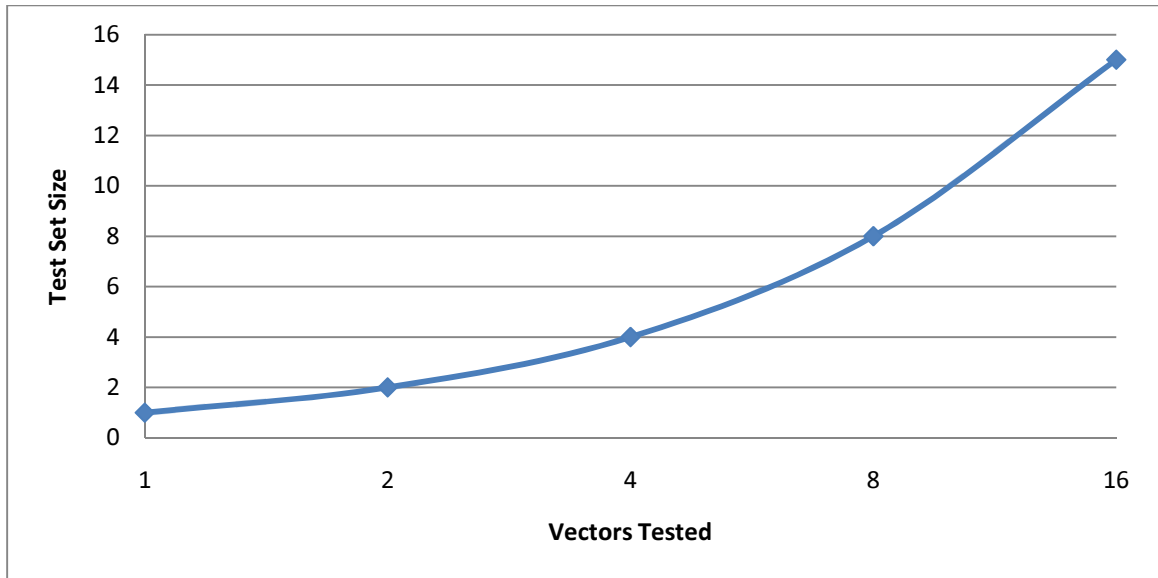




Rd73

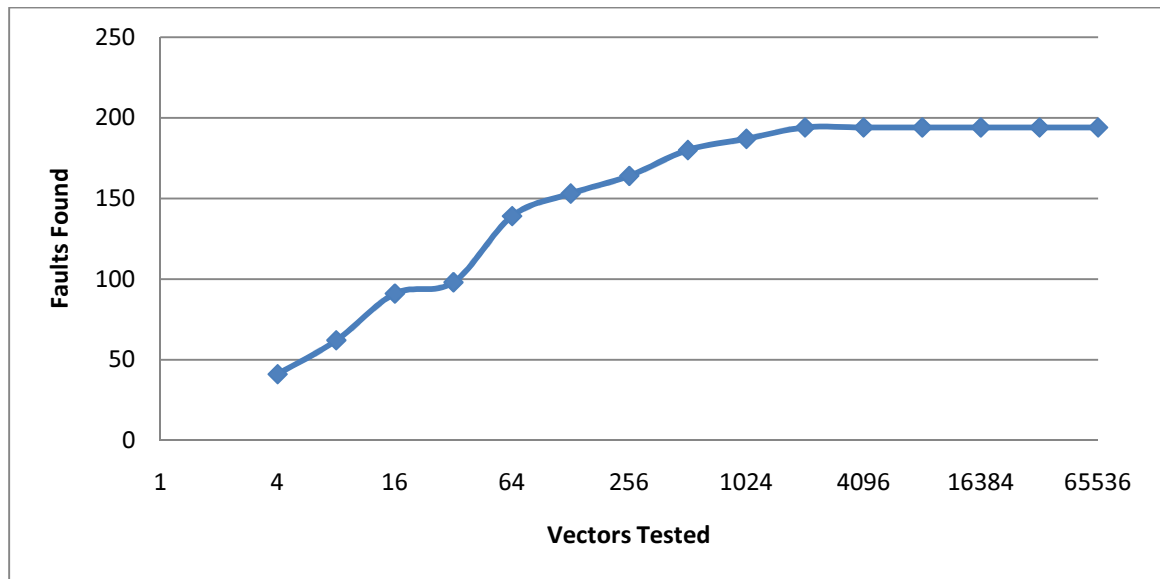
Vectors Tested	Faults Found	Number of Vectors Needed	Time
1	34	1	0.00628
2	63	2	0.0109
4	103	4	0.0203
8	170	8	0.0394
16	210	15	0.074
32	279	29	0.146
64	343	50	0.254
128	404	72	0.367

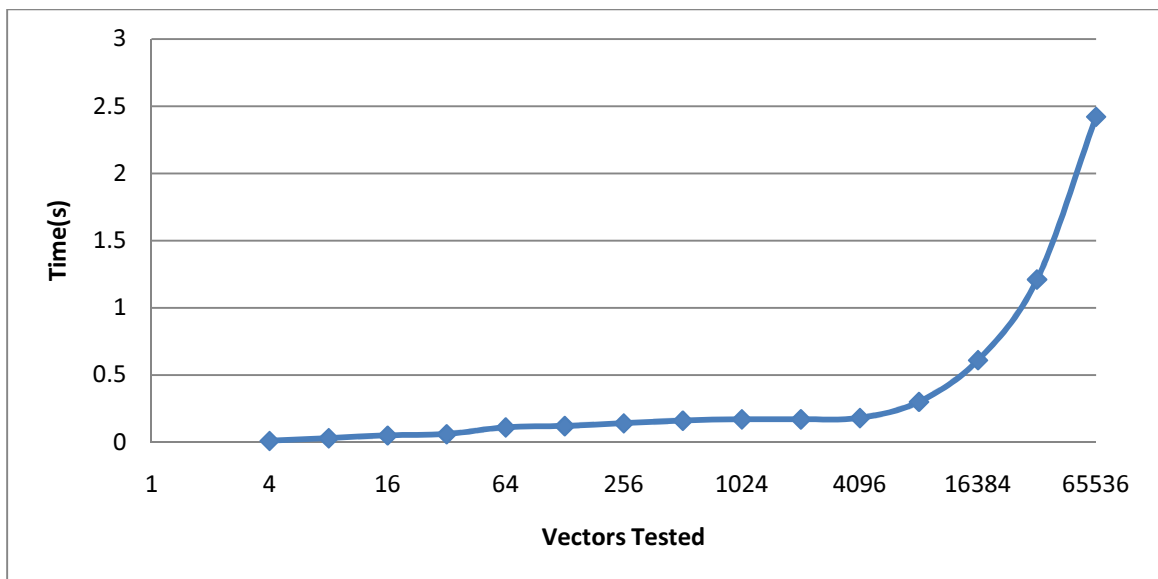
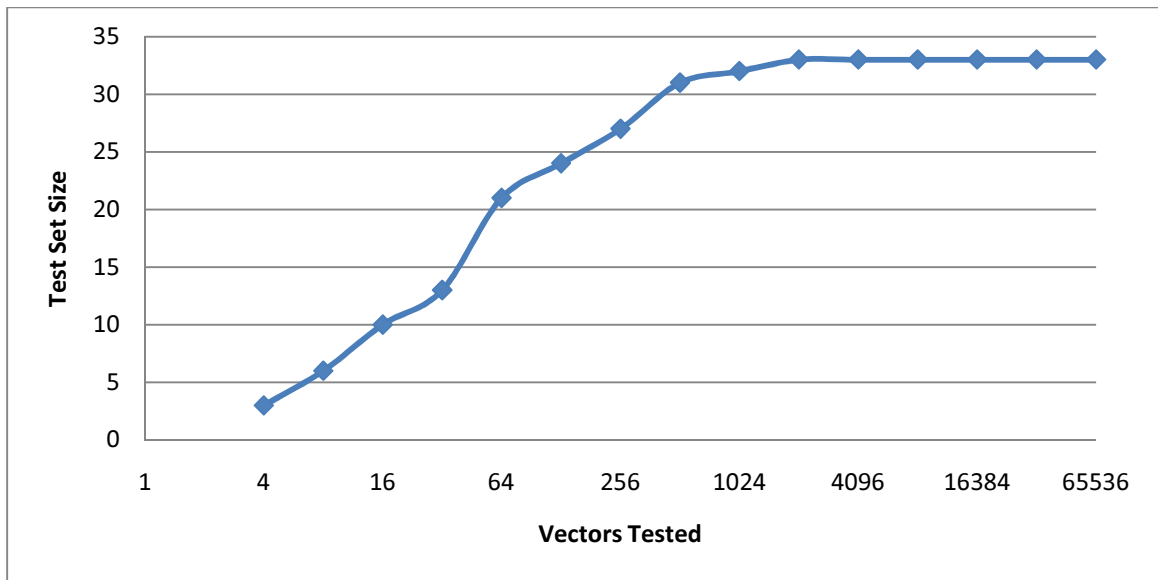




T481

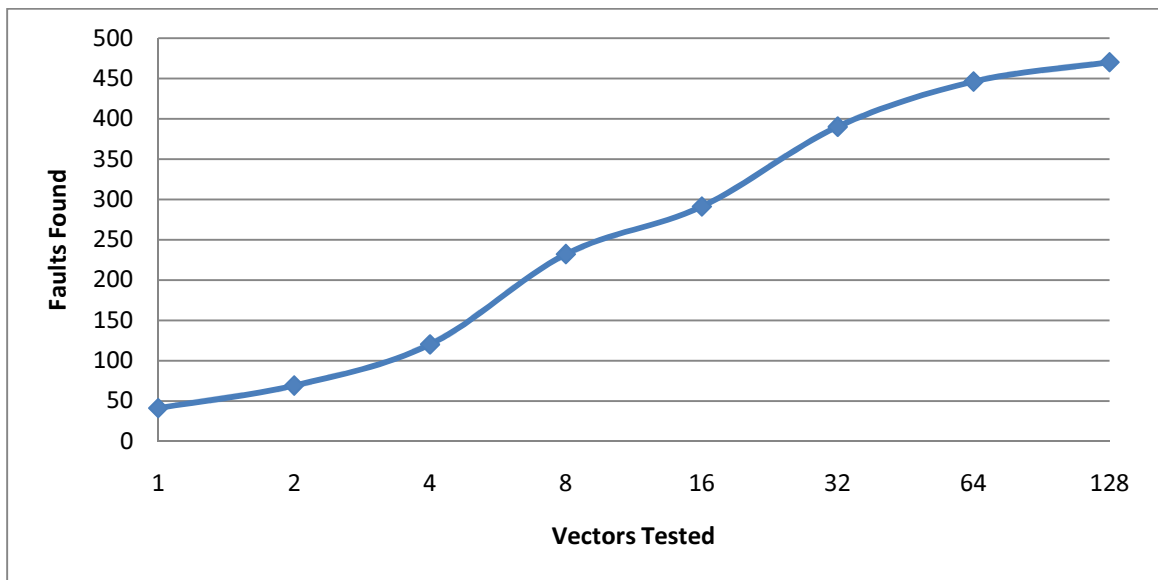
Vectors Tested	Faults Found	Number of Vectors Needed	Time
4	41	3	0.01
8	62	6	0.03
16	91	10	0.05
32	98	13	0.06
64	139	21	0.11
128	153	24	0.12
256	164	27	0.14
512	180	31	0.16
1024	187	32	0.17
2048	194	33	0.17
4096	194	33	0.18
8192	194	33	0.3
16384	194	33	0.61
32768	194	33	1.21
65536	194	33	2.42

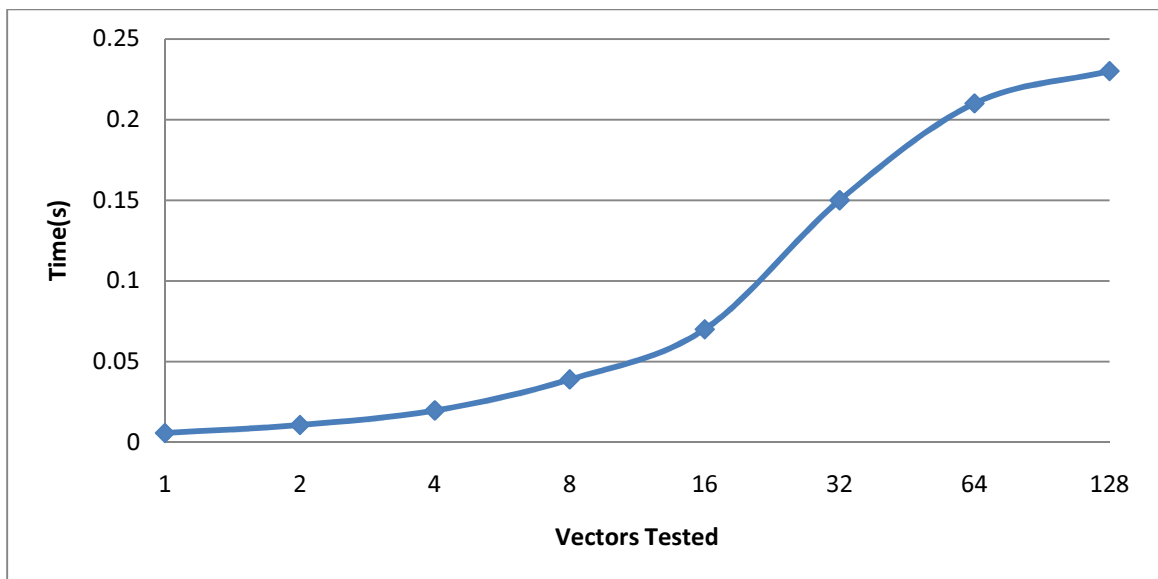
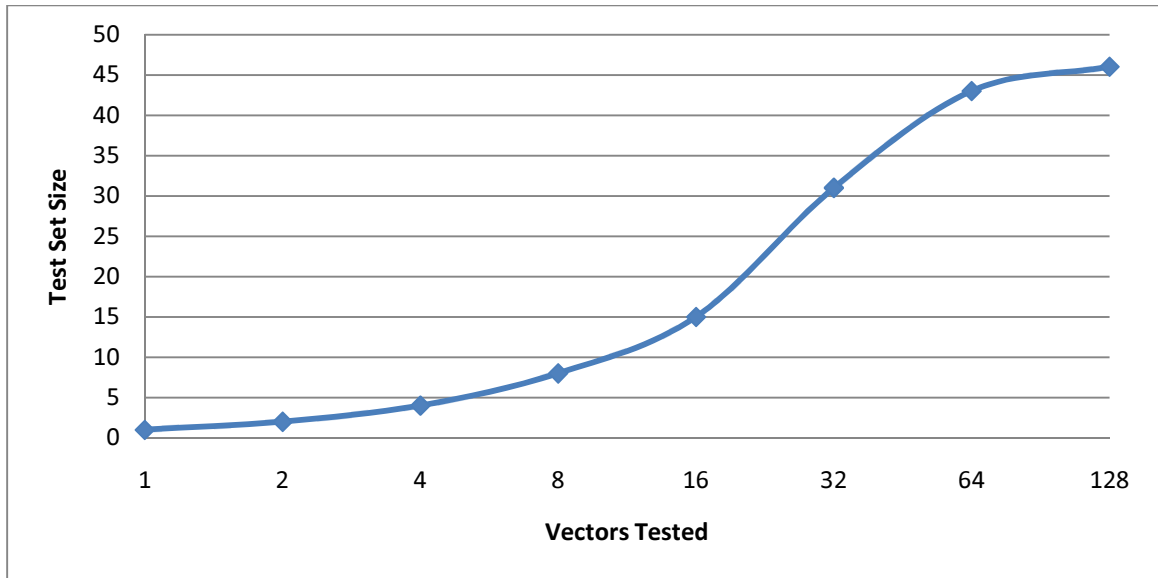




Z5xp1

Vectors Tested	Faults Found	Number of Vectors Needed	Time
1	41	1	0.00576
2	69	2	0.0107
4	120	4	0.0197
8	232	8	0.0389
16	291	15	0.07
32	390	31	0.15
64	446	43	0.21
128	470	46	0.23





Z9sym

Vectors Tested	Faults Found	Number of Vectors Needed	Time
1	9	1	0.00626
2	26	2	0.0111
4	41	4	0.0208
8	77	7	0.035
16	117	15	0.0749
32	180	22	0.111
64	276	35	0.18
128	330	46	0.237
256	381	64	0.332
512	410	75	0.39

