NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Evaluating the Impact of Hardware Faults in Modern Microprocessor Arithmetic Units

Charidimos - Porfyrios N. Papadakis

Supervisor: **Dimitris Gizopoulos,** Professor

ATHENS

SEPTEMBER 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αξιολόγηση της Επίπτωσης των Ελαττωμάτων Υλικού σε Αριθμητικές Μονάδες Σύγχρονων Μικροεπεξεργαστών

Χαρίδημος - Πορφύριος Ν. Παπαδάκης

Επιβλέπων: **Δημήτριος Γκιζόπουλος,** Καθηγητής

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2024

BSc THESIS

Evaluating the Impact of Hardware Faults in Modern Microprocessor Arithmetic Units

Charidimos - Porfyrios N. Papadakis
S.N.: 111520200161

SUPERVISOR: **Dimitris Gizopoulos,** Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αξιολόγηση της Επίπτωσης των Ελαττωμάτων Υλικού σε Αριθμητικές Μονάδες Σύγχρονων Μικροεπεξεργαστών

Χαρίδημος - Πορφύριος Ν. Παπαδάκης

Α.Μ.: 111520200161

ΕΠΙΒΛΕΠΩΝ: **Δημήτριος Γκιζόπουλος,** Καθηγητής

ABSTRACT

Nowadays, we are surrounded by an astonishing amount of electronic devices constantly transmitting data to each other or to the internet. Most of our everyday tasks, such as navigation, communications and payments are handled by a computer. It is obvious that our modern world relies vastly on calculations done on computers and, therefore, on computer (micro)processors.

Due to the ease, the high accuracy and speed that modern computing systems offer, we have come to accept that the result the computer's processor gives us is always the correct one. Indeed, the reliability of a electronic computer chip is much higher than every method we used before them (mechanical, magnetic). However, as has been pointed out for some years now, even computer chips can make silent errors. That is, errors which are caused by hardware (silicon) defects and not noticed (detected) by any hardware or software mechanism. Over the past few years, large tech companies, operating thousands of servers have pointed out the existence of such hardware defects [9], [11], [16]. It has also been made clear that the arithmetic units of modern processors are the most common culprits of output errors due to hardware defects.

Based on that context, the scope of this thesis is to study the patterns, the frequency, and the severity of the errors caused by such hardware defects in arithmetic units of processors. The employed methodology consists of HDL models synthesis of those modules and then, the intentional induction of certain models of hardware defects (faults) to them, in order to observe the results and patterns inside them. Two types of faults were introduced to the synthesized models; bridging faults (that is faults which are caused by bridging the output of two different gates) and stuck-at faults (faults which are caused by forcing the output of a gate to be either constantly high or constantly low).

By implementing such faults to a large number of different random gates inside the synthesized module and testing the modules with random and patterned inputs, the distribution of errors on the outputs can be observed and, therefore, produce an error model for the output of each arithmetic unit tested.

SUBJECT AREA: Silent Data Corruptions (SDCs), Chip reliability analysis

KEYWORDS: fault injection, bridging faults analysis, stuck at faults analysis, processor reliability study, processor fault model

ΠΕΡΙΛΗΨΗ

Καθημερινά περικλειόμαστε από έναν τεράστιο αριθμό ηλεκτρονικών συσκευών, οι οποίες μεταφέρουν διαρκώς δεδομένα μεταξύ τους και στο διαδίκτυο. Οι περισσότερες από τις καθημερινές μας δραστηριότητες, όπως η περιήγηση, η επικοινωνία και οι πληρωμές στηρίζονται σε κάποιας μορφής υπολογιστικό σύστημα. Είναι, επομένως, εμφανές ότι ο σύγχρονος κόσμος βασίζεται άρρηκτα σε υπολογισμούς που πραγματοποιούνται σε υπολογιστές και, συνεπώς, στα τσιπ επεξεργαστών.

Λόγω της ευκολίας, της πολύ υψηλής ακρίβειας και της ταχύτητας που τα σύγχρονα υπολογιστικά συστήματα προσφέρουν, πιστεύουμε ακράδαντα ότι το αποτέλεσμα που λαμβάνουμε από τον επεξεργαστή ενός συστήματος είναι πάντοτε σωστό. Πράγματι, η αξιοπιστία ενός ηλεκτρονικού υπολογιστικού συστήματος είναι αισθητά υψηλότερη από τους χειροκίνητους υπολογισμούς, τους οποίους αντικατέστησε ή άλλους μηχανικούς ή μαγνητικούς μηχανισμούς. Ωστόσο, όπως έχει γίνει εμφανές για μερικά χρόνια πλέον, ακόμα και οι επεξεργαστές των υπολογιστών μπορούν να κάνουν σιωπηλά σφάλματα. Αυτό σημαίνει ότι μπορούν να πραγματοποιηθούν σφάλματα στην έξοδο του επεξεργαστή τα οποία δεν μπορούν να εντοπιστούν άμεσα από διορθωτικούς μηχανισμούς ή άλλες τεχνολογίες ανίχνευσης σφαλμάτων. Μέσα στα τελευταία χρόνια, μεγάλες εταιρίες παροχής διαδικτυακών υπηρεσιών έχουν επισημάνει την ύπαρξη τέτοιου τύπου σφαλμάτων [9], [11], [16]. Έχει γίνει επίσης ξεκάθαρο από κορυφαίες εταιρίες του χώρου ότι οι αριθμητικές μονάδες των σύγχρονων επεξεργαστών αποτελούν τον κύριο υπαίτιο για αυτά τα σφάλματα.

Λαμβάνοντας υπόψη τα παραπάνω ευρήματα, σκοπός αυτής της πτυχιακής εργασίας είναι η μελέτη των μοτίβων, της συχνότητας, και της σφοδρότητας τέτοιου είδους σφαλμάτων στις αριθμητικές μονάδες των επεξεργαστών. Η μεθοδολογία που ακολουθήθηκε ήταν η σύνθεση των HDL μοντέλων των αριθμητικών μονάδων και η σκόπιμη εισαγωγή σφαλμάτων σε αυτά, με σκοπό τη παρακολούθηση των αποτελεσμάτων και των μοτίβων που ενδεχομένως εμφανίζονταν σε αυτά. Εισάγονται δύο τύποι σφαλμάτων στα synthesized μοντέλα: σφάλματα γεφύρωσης (bridging faults), δηλαδή σφάλματα που προκύπτουν από την ένωση της εξόδου δύο τυχαίων πυλών, και σφάλματα προσκόλλησης (stuck-at faults), δηλαδή σφάλματα που προκύπτουν από τον εξαναγκασμό της εξόδου μιας πύλης να είναι πάντα υψηλή/high ή πάντα χαμηλή/low).

Υλοποιώντας τέτοιου είδους σφάλματα σε έναν μεγάλο αριθμό διαφορετικών τυχαία επιλεγμένων πυλών στο synthesized μοντέλο της αριθμητικής μονάδας και δοκιμάζοντας το με έναν μεγάλο αριθμών τυχαίων αλλά και με συγκεκριμένο μοτίβο εισόδων, η κατανομή των σφαλμάτων στην έξοδο μπορεί να παρατηρηθεί και, επομένως, να παραχθεί ένα μοντέλο σφαλμάτων για την έξοδο της κάθε μιάς αριθμητικής μονάδας που δοκιμάστηκε.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Silent Data Corruptions (SDCs), ανάλυση αξιοπιστίας επεξεργαστών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: εισαγωγή σφαλμάτων σε επεξεργαστές, ανάλυση bridging faults, ανάλυση stuck at faults, ανάλυση αξιοπιστίας επεξεργαστών, μοντέλο σφαλμάτων επεξεργαστή

ACKNOWLEDGEMENTS

Θα ήθελα να ευχαριστήσω τον κ. Γκιζόπουλο που παρείχε μέσω των διαλέξεων και της επικοινωνίας που είχαμε τη δυνατότητα περαιτέρω εξερεύνησης του τομέα της αρχιτεκτονικής των υπολογιστών.

Επίσης, θα ήθελα να ευχαριστήσω την οικογένεια και τους φίλους μου για τη γενικότερη στήριξη και τη συντροφία τους, καθώς η εκπόνηση μιας ερευνητικής εργασίας έχει ως προαπαιτούμενο τη καλή ψυχολογία.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Importance of reliability analysis

As mentioned in the abstract, modern chips utilize several advanced technologies such as, but not limited to, dynamic frequency and voltage scaling, varying power limits, automatic over-clock mechanisms (see Intel's Turbo Boost which has been available since 2009 [8] or AMD's Turbo Core) and aggressive out-of-order execution mechanisms. While all the aforementioned technologies make modern chips even more complex, they greatly help in improving application performance (see Passmark's CPU Ratings for the past few years [3]).

Similarly, GPUs and other accelerators also rely heavily on complex technologies which, in return, yield to ever-increasing performance.

A side effect of having such complex chips, with extremely high transistor count, running several programs at once is that the probability of an error occurring and getting unnoticed is getting higher. That is not to imply that hardware faults causing output errors have not existed in the past [2].

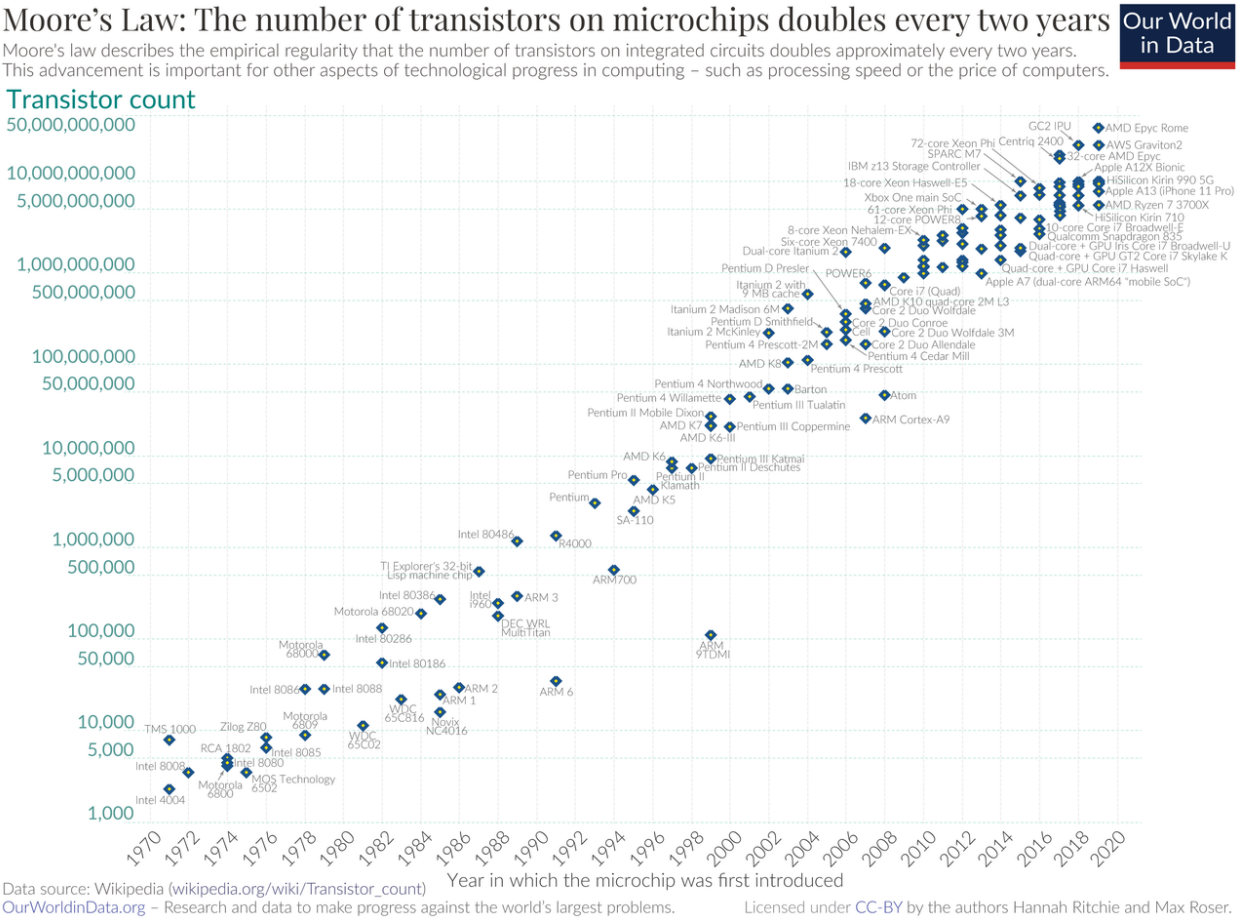


Figure 1.1: The number of transistors on popular computer chips

For the above reasons and due to the undeniable fact that our society relies increasingly on informatics systems, the need to analyze the reliability of the arithmetic units integrated into computer chips is becoming more and more apparent.

1.2 Reliability analysis methodology used

The research done in the context of that thesis consists of the following steps:

1. **Finding the appropriate arithmetic units for testing**

   Models of several arithmetic units of a modern processor have been tested. Those models were given in Hardware Description Language (Verilog).

2. **Synthesize those models**

   The HDL models were synthesized in a gate-level fashion.

3. **Inject faults on the tested modules**

   Certain fault types were injected into the synthesized models in order to test how greatly they affect the output. In this fashion, a misbehaving arithmetic unit with some gate-level fault is being simulated. Several different gate inputs / outputs, one at each time, were injected.

4. **Run tests on those injected modules**

   Several different inputs were given to each faulty model in order to observe how the injected gate-level fault affects the result. Comparison with the correct output was done.

5. **Observe the results and calculate the Bit Error Rate**

   By utilizing some scripts, the results of all tests done on an arithmetic unit are presented, and the bit error rate for that module, for each bit of the output, is calculated.

In the following chapters, detailed information about each step will be provided and the results found will be presented with great detail.

1.3 Arithmetic modules tested

In the scope of this thesis, the following arithmetic units were studied:

1. **FPU Adder [12]**
2. **FPU Multiplier [12]**
3. **Integer Adder Using 4 Bit adder blocks**
4. **Integer Adder Using 32Bit adder blocks**
5. **Integer Adder (Single 64Bit adder block)**
6. **Integer Array Multiplier**
7. **Integer DADDA Multiplier**
8. **Integer Wallace Multiplier**

Note: For The FPU Adder and Multiplier tests, only the corresponding parts of the project were synthesized so that the injected internal wires correspond to those of the adder or multiplier part accordingly.

The choice of arithmetic modules was made so as to test the most commonly used ones. Especially the tests done on the Integer Adder and Multiplier modules are very important because such modules are, essentially, the building blocks of GPUs and AI Accelerator units (such as neural network accelerators of all types).

2. FROM HARDWARE DEFECTS TO OUTPUT ERRORS

2.1 Defect versus Fault versus Error

The terms Defect, Fault and Error have different, but oftentimes mistakenly interchangeably used, meanings [6].

A Defect is a problem on the computer chip, caused by factors, such as silicone wear, hardware design problems and more, discussed in the next chapters of the unit.

A Fault is a model which expresses hardware defects and allows us to categorize and, ultimately, simulate them. By having an appropriate fault model, we attempt to express certain groups of hardware defects.

An Error is a wrong output produced by the computer chip, caused by a defect existing on that chip. In order to simulate such output errors, we simulate certain fault models.

2.2 Silent Data Corruptions or Silent Data Errors

The term Silent Data Corruption or Silent Data Error, refers to a wrongly calculated output from a CPU (e.g. 1+3=6) which is saved and used without causing the chip to crash or show any sign of failure.

While such undetected errors have existed for several years [2], the more advanced chip architectures of the last decade or so are more prone to causing such errors due to the extremely high transistor counts and advanced power management technologies such as very rapid voltage and frequency scaling.

For that reason, such errors have begun to affect production servers of large tech enterprises [9], [11]. As such, the research on the subject has skyrocketed over the past few years.

Oftentimes, hardware defects causing such errors are caught during the manufacturing phase of the chip and the defective chip is not delivered. However, as the possible erroneous outputs become more and more unnoticeable, and testing the whole possible set of functions of the chip is not viable, they have become more commonplace in production computers.

Thus, the study on how they manifest in a real arithmetic unit of a CPU is becoming more and more necessary in order to avoid errors in mission-critical operations.

2.3 Common reasons behind output errors

Output errors can occur due to several reasons, some of which are temporary, while others are permanent [9]. More information about the different types of faults will be presented on the next chapter. Output errors are often caused by:

- **Manufacturing or design problems:**

  In that case, output errors are caused due to defects during hardware manufacturing (such as some transistors not being etched correctly and, thus, in some power states

do not give a steady output) or due to timing errors caused by too high clocks for the given power level. In that case, the errors will be caused even when the chip is brand-new.

- **Early life failures**

  Here, output errors are caused due to weaknesses in one or more transistors which manifest after the chip has been shipped and occur before shortly after the chip is put into production.

- **Chip wear and degradation**

  Errors caused due to the chip aging. In that case, errors occur after the chip has been working correctly for a while and do not occur uniformly across chips of the same architecture. Finding and isolating those defective chips is pretty tricky because constant checking must be done across the fleet of computers. However, chips with that behavior are believed to be rarer than chips with early-life / manufacturing or end-of-life defects.

- **End-of-life wear**

  A chip is significantly more prone to cause errors after several years of constant operation because the whole silicon wears out. The probabilities of such errors occurring follow the bathtub curve [17].

The above reasons have been studied on chips operating constantly on data servers, which have very high uptime and optimal cooling solutions. For chips being used in home or corporate computers, the results may differ significantly and I believe those cases should be studied separately. However, due to the most important and mission-critical operations being run on large data server and cloud clusters nowadays, the significance of output errors being caused on chips of home computers is not, usually, that high.

2.4 Different types of faults in modern CPUs

The different models of hardware defects yielding to errors mentioned previously, result in errors which occur in a different fashion.

- **Transient faults**

  Those faults occur once and a repeat pattern cannot be observed. Such faults can be caused by external factors, such as cosmic rays [7]. However, such faults can also be caused by some sort of silicon defect, which yields to faults occurring at random intervals.

- **Intermittent faults**

  Those faults occur intermittently; for the same input, the chip may sometimes give the correct result, while others a wrong one. Due to modern chip's dynamic voltage and frequency scaling, the errors may occur on a specific voltage/frequency state and not on others. Research on such modern chip design has already been done [15].

- **Permanent errors**

  Those faults occur all the times; for a specific input, the output is always the same wrong one. The cause of such errors is the permanent corruption of one or more gates inside the chip. Such errors can be simulated with gate-level simulators, as will be shown on the following chapters.

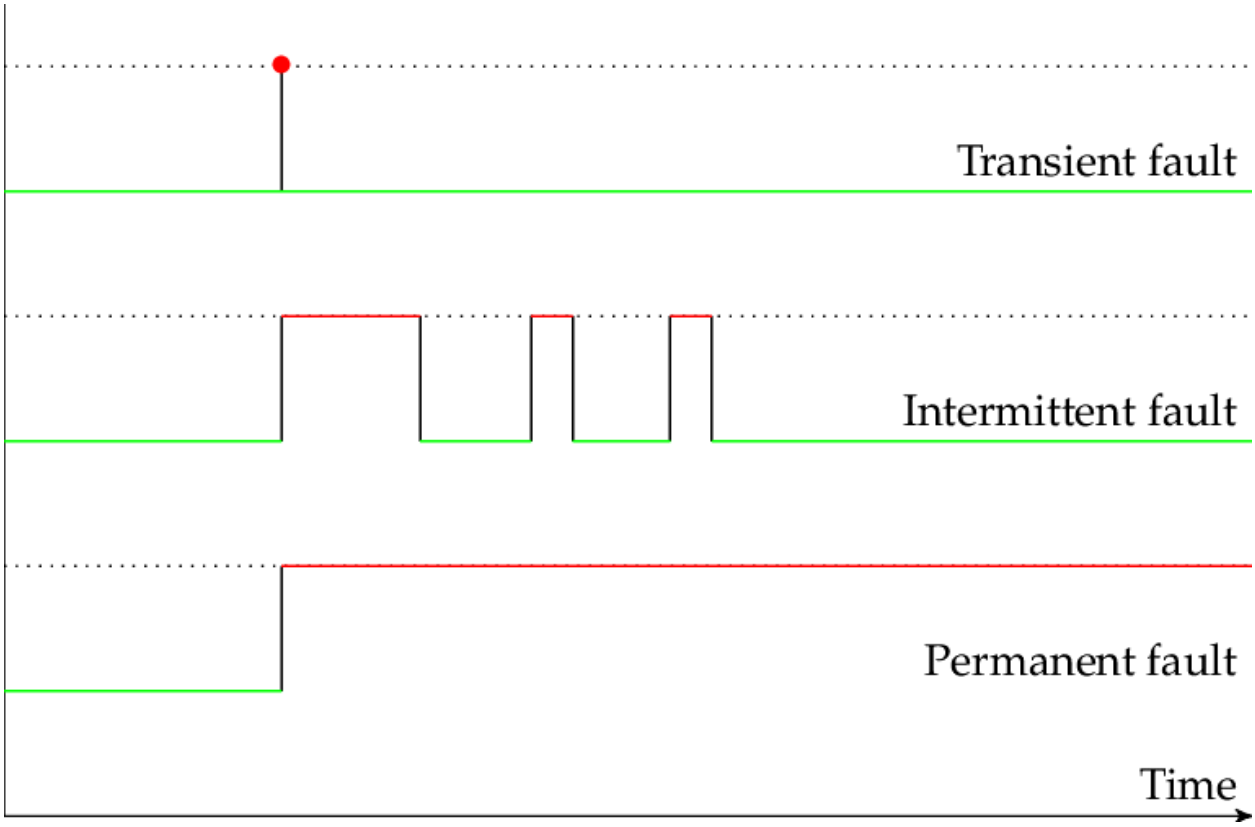


Figure 2.1: Graphical representation of different types of faults

2.5 Examples of such errors in production machines

Errors caused by silent data corruptions have already manifested on production machines. One example is from Meta / Facebook, where the file size of a real file has been saved as 0 bytes, interfering with the decompression procedure [9]. Another example is from Alibaba Cloud, where a certain server reported way more frequent than usual checksum mismatches between user data [11].

3. TESTING ENVIRONMENT

3.1 Brief introduction to typical testing methods

Over the past few years, large-scale internet services enterprises have attempted several methods to detect problematic CPUs on production machines. Those tests are run either parallel to the normal workload (that is, when the CPU is also running in production) or during scheduled offline/maintenance intervals [11], [16].

Regarding the methods those tests are being held, several methodologies, of varying complexity and interval, have been proposed and are actively being used. Some of those tests run at the application level (and thus are easier to monitor and schedule), while others run at kernel/OS level. The latter obviously requires for the system to be offline from production, which can be expensive, but allows way more detailed testing by possibly adjusting the frequency, temperature and voltage of the CPU [11]. When it comes to application level tests, the goal is to achieve the lowest possible overhead. Several intelligent application levels testing methods have already been developed, and that research area remains very lucrative [9], [11], [16], [14].

In the context of the current research, emphasis will mostly be placed on injecting faults to modern processors during gate-level simulation in order to observe the patterns of errors being produced. Having such an error model can greatly help improving testing methodology with low-performance overhead.

3.2 Tools used

The tools used for the current research were:

- Synopsys VCS functional verification solution [4] was utilized to simulate the faults injected in the arithmetic units models with their corresponding test benches, described in SystemVerilog
- Yosys Open SYnthesis Suite [5] was utilized to synthesize the arithmetic unit models in Verilog
- FreePDK45 [1] cells were used as the building cell blocks for the arithmetic units tested
- Python 3 programming language for developing the scripts required for automatic fault injection, test instrumentation, result parsing, graph plotting and error model generation.

More information about the testing procedure used will be provided in the following chapters.

3.3 Types of faults injected

In the context of this research, the following two models of hardware defects were injected into the arithmetic units [13]:

- **Bridging Faults**

  The terminology bridging fault refers to hardware defects caused inside a computer chip when two wires are “bridged” (connected) together when they shouldn’t. The behavior in those cases is unpredicted, because the state of the wire is defined by the momentarily stronger signal between the two bridged wires.

- **Stuck-at Faults**

  The terminology stuck-at fault refers to hardware defects caused inside a computer chip when the output of a gate, a wire, is always stuck to high or low.

Those faults can be permanent or caused under specific circumstances, as mentioned before. During this research, each fault is permanent across testing (see the proceeding chapter).

3.4 Detailed testing methodology

As briefly presented in the introduction, the research done consisted of the following steps:

1. **Finding the appropriate arithmetic units for testing**

   Tests on several arithmetic units of a modern processor have been tested (see chapter 1.3 for detailed information about those units). The selection has been made in order to take into consideration the building blocks of modern processors and accelerators.

2. **Synthesizing those models**

   The aforementioned models were synthesized using the YoSys synthesis suite. More specifically, the procedure followed for synthesizing the models using YoSys was the following:

   (a) Defining the top unit of each module

   ```
   $ hierarchy -top {module_name}
   ```

   (b) Converting the blocks to netlist, optimizing, converting the netlist to gate logic and optimizing again

   ```
   $  proc; opt; techmap; opt;
   ```

   (c) Mapping the flip-flops to the cell library

   ```
   $ dfflibmap -liberty {path to cell lib}
   ```

   (d) Mapping the logic to the cell library using abc

   ```
   $ abc -liberty {path to cell lib}
   ```

   (e) Cleaning up unnecessary wires

   ```
   $ clean
   ```

   (f) Writing verilog code

   ```
   $ write_verilog synth.v
   ```

3. **Inject faults on the tested modules**

   Stuck at and Bridging faults were being injected into the synthesized models. To achieve that, certain statements are being added inside the test bench of the modules, in order to force the outputs of some specific gates to be set manually, depending on the type of injection being held.

   It is obvious that running the gate-level simulation with those test benches will help us observe the behavior of a presumably misbehaving CPU.

   The methodology used to achieve the injection was the following:

   For stuck-at faults:

   (a) Finding a random wire from the synthesized file of the arithmetic module

   (b) Forcing that wire to be always high, by creating a copy of the test bench in which the line

   ```
   force {unit_name}._{selected_wire} = 1'b1
   ```

   is appended.

   (c) Forcing that wire to be always low, by creating a copy of the test bench in which the line

   ```
   force {unit_name}._{selected_wire} = 1'b0
   ```

   is appended.

   For bridging faults:

   (a) Finding two random wires from the synthesized file of the arithmetic module

   (b) Forcing the first wire to always follow the value of the second wire, by creating a copy of the test bench in which the line

   ```
   force {unit_name}._{first_wire} = {unit_name}._{second_wire)
   ```

   is appended.

4. **Run tests on those injected modules**

   With the modified test bench, created as mentioned in the previous step, the synthesized file of the arithmetic unit and the FreePDK45 cell library verilog file, the simulation stage can begin.

   In order to test several random injections, the following methodology has been followed:

   (a) Inject a random fault (of either type) on a random wire of the module being tested (with the methodology discussed in the previous step)

   (b) Add a line containing of a random number in the test bench, which will be used as seed for the Verilog simulation. Inside the testbench file, 1000 random numbers are generated and tested for each injection done to an arithmetic unit

   Note (1): For the FPU Adder, 2000 random numbers were tested; 1000 for the add operation and another 2000 for the subtract operation

   Note (2): For the DADDA Multiplier, apart from the 1000 random numbers, tests using specific sequences of 4Bit patterns were used. A total of 256 operations were run on those tests, the 1/4th of the 1000 random numbers. As will be seen later, those sequences yielded similar fault detection capabilities. Thus, they can be used for multiplier modules to reduce testing time. [10]

(c) Begin the simulation. Inside the test bench, a faultless (non-injected) copy of the module being tested is contained, so as to compare the results. The output of the simulation, in which both the result for each operation produced by the injected module and the result produced by the non-injected ("golden") module is saved to a text file.

(d) The above steps were repeated 1000 times for each module.

Note: For the stuck-at fault analysis, the same wire was stuck to both zero (Low) and one (High). So, those steps were run for 2000 times.

5. **Observe the results and calculate the Bit Error Rate**

By the end of the previous step, a spreadsheet which contains, for the tests done on each of the 1000 (or 2000) injections, the number of times each bit of the output was wrong (Erroneous Bit Positions) and the number of erroneous bits that the operations run caused (Erroneous Bits).

Simply put, for each specific injected fault (either a stuck at zero or one fault, or a wire bridge fault), in which a set number of operations was run (either 256 *(specific 4bit patterns)*, or 2000 *(only for the FPU Adder)* or 1000 *(all other cases)*) we get two rows:

(a) **Erroneous Bits**

For that set number of operations, the number of which did not have any wrong bit, the number of which had one wrong bit, the number of which had two wrong bits and so on.

(b) **Erroneous Bit position**

For all those operation done on that injection experiment, the locations of the erroneous bits in the output. That is, the number of times the bit 0 of the output was wrong, the number of times the bit 1 of the output was wrong and so on.

By knowing the average erroneous bits for each injection and the number of output bits of the given module, it is obvious that the bit error rate for that module can be easily calculated. In order to do so, we simply calculate average the average number of erroneous errors of each injections, and divide that with the number of output bits.

Obviously:

$$BER = \frac{\text{Total number of erroneous bits across all tests}}{\text{Total number of operations done * Total number of injections done * Number of output bits}} \text{ or}$$

$$BER = \frac{\sum \text{Average erroneous bits}}{\text{Total number of injections done * Number of output bits}}$$

As a result, we are able to say that, for a specific module which is known to suffer from stuck-at faults, or from bridging faults, or from both (depending on which data we used to calculate the bit error rate), when operated with random inputs, the probability of an output bit to be erroneous is known.

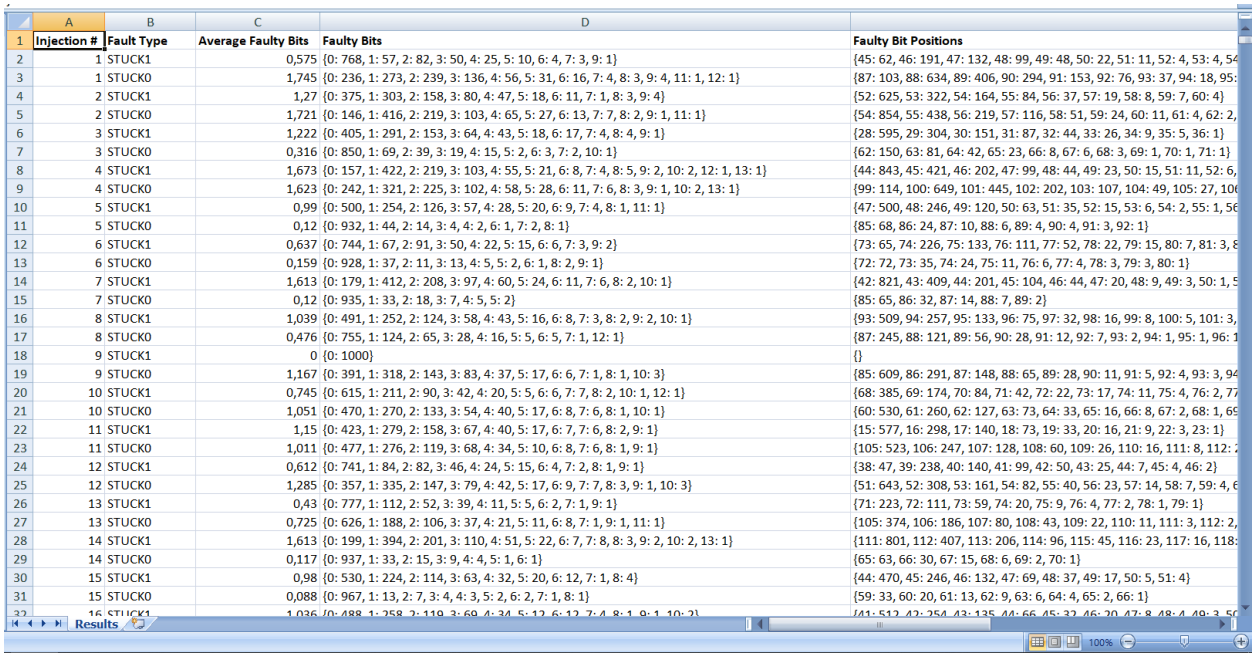
This is not the most accurate method possible for generating a fault model, and a slightly better approach will be discussed later.

At this point, I must point out that the numeric patterns used by some common programs are usually different than the random numbers tested. Thus, the error patterns of those programs might differ.

Thus, in order to calculate the error rate for a specific program or set of programs, the arithmetic patterns used by them should be observed and then, tests based on those patterns, and not random ones, should be used.

3.5 Result files example

As mentioned on the previous chapter, after each test done for a specific module, a spreadsheet is produced. An example of such a Spreadsheet is the following:



	A	B	C	D	
1	Injection #	Fault Type	Average Faulty Bits	Faulty Bits	Faulty Bit Positions
2	1	STUCK1	0,575	{0: 768, 1: 57, 2: 82, 3: 50, 4: 25, 5: 10, 6: 4, 7: 3, 9: 1}	{45: 62, 46: 191, 47: 132, 48: 99, 49: 48, 50: 22, 51: 11, 52: 4, 53: 4, 54
3	1	STUCK0	1,745	{0: 236, 1: 273, 2: 239, 3: 136, 4: 56, 5: 31, 6: 16, 7: 4, 8: 3, 9: 4, 11: 1, 12: 1}	{87: 103, 88: 634, 89: 406, 90: 294, 91: 153, 92: 76, 93: 37, 94: 18, 95:
4	2	STUCK1	1,27	{0: 375, 1: 303, 2: 158, 3: 80, 4: 47, 5: 18, 6: 11, 7: 1, 8: 3, 9: 4}	{52: 625, 53: 322, 54: 164, 55: 84, 56: 37, 57: 19, 58: 8, 59: 7, 60: 4}
5	2	STUCK0	1,721	{0: 146, 1: 416, 2: 219, 3: 103, 4: 65, 5: 27, 6: 13, 7: 7, 8: 2, 9: 1, 11: 1}	{54: 854, 55: 438, 56: 219, 57: 116, 58: 51, 59: 24, 60: 11, 61: 4, 62: 2,
6	3	STUCK1	1,222	{0: 405, 1: 291, 2: 153, 3: 64, 4: 43, 5: 18, 6: 17, 7: 4, 8: 4, 9: 1}	{28: 595, 29: 304, 30: 151, 31: 87, 32: 44, 33: 26, 34: 9, 35: 5, 36: 1}
7	3	STUCK0	0,316	{0: 850, 1: 69, 2: 39, 3: 19, 4: 15, 5: 2, 6: 3, 7: 2, 10: 1}	{62: 150, 63: 81, 64: 42, 65: 23, 66: 8, 67: 6, 68: 3, 69: 1, 70: 1, 71: 1}
8	4	STUCK1	1,673	{0: 157, 1: 422, 2: 219, 3: 103, 4: 55, 5: 21, 6: 8, 7: 4, 8: 5, 9: 2, 10: 2, 12: 1, 13: 1}	{44: 843, 45: 421, 46: 202, 47: 99, 48: 44, 49: 23, 50: 15, 51: 11, 52: 6,
9	4	STUCK0	1,623	{0: 242, 1: 321, 2: 225, 3: 102, 4: 58, 5: 28, 6: 11, 7: 6, 8: 3, 9: 1, 10: 2, 13: 1}	{99: 114, 100: 649, 101: 445, 102: 202, 103: 107, 104: 49, 105: 27, 10
10	5	STUCK1	0,99	{0: 500, 1: 254, 2: 126, 3: 57, 4: 28, 5: 20, 6: 9, 7: 4, 8: 1, 11: 1}	{47: 500, 48: 246, 49: 120, 50: 63, 51: 35, 52: 15, 53: 6, 54: 2, 55: 1, 56
11	5	STUCK0	0,12	{0: 932, 1: 44, 2: 14, 3: 4, 4: 2, 6: 1, 7: 2, 8: 1}	{85: 68, 86: 24, 87: 10, 88: 6, 89: 4, 90: 4, 91: 3, 92: 1}
12	6	STUCK1	0,637	{0: 744, 1: 67, 2: 91, 3: 50, 4: 22, 5: 15, 6: 6, 7: 3, 9: 2}	{73: 65, 74: 226, 75: 133, 76: 111, 77: 52, 78: 22, 79: 15, 80: 7, 81: 3, 8
13	6	STUCK0	0,159	{0: 928, 1: 37, 2: 11, 3: 13, 4: 5, 5: 2, 6: 1, 8: 2, 9: 1}	{72: 72, 73: 35, 74: 24, 75: 11, 76: 6, 77: 4, 78: 3, 79: 3, 80: 1}
14	7	STUCK1	1,613	{0: 179, 1: 412, 2: 208, 3: 97, 4: 60, 5: 24, 6: 11, 7: 6, 8: 2, 10: 1}	{42: 821, 43: 409, 44: 201, 45: 104, 46: 44, 47: 20, 48: 9, 49: 3, 50: 1, 5
15	7	STUCK0	0,12	{0: 935, 1: 33, 2: 18, 3: 7, 4: 5, 5: 2}	{85: 65, 86: 32, 87: 14, 88: 7, 89: 2}
16	8	STUCK1	1,039	{0: 491, 1: 252, 2: 124, 3: 58, 4: 43, 5: 16, 6: 8, 7: 3, 8: 2, 9: 2, 10: 1}	{93: 509, 94: 257, 95: 133, 96: 75, 97: 32, 98: 16, 99: 8, 100: 5, 101: 3,
17	8	STUCK0	0,476	{0: 755, 1: 124, 2: 65, 3: 28, 4: 16, 5: 5, 6: 5, 7: 1, 12: 1}	{87: 245, 88: 121, 89: 56, 90: 28, 91: 12, 92: 7, 93: 2, 94: 1, 95: 1, 96: 1
18	9	STUCK1	0	{0: 1000}	{}
19	9	STUCK0	1,167	{0: 391, 1: 318, 2: 143, 3: 83, 4: 37, 5: 17, 6: 6, 7: 1, 8: 1, 10: 3}	{85: 609, 86: 291, 87: 148, 88: 65, 89: 28, 90: 11, 91: 5, 92: 4, 93: 3, 94
20	10	STUCK1	0,745	{0: 615, 1: 211, 2: 90, 3: 42, 4: 20, 5: 5, 6: 6, 7: 7, 8: 2, 10: 1, 12: 1}	{68: 385, 69: 174, 70: 84, 71: 42, 72: 22, 73: 17, 74: 11, 75: 4, 76: 2, 77
21	10	STUCK0	1,051	{0: 470, 1: 270, 2: 133, 3: 54, 4: 40, 5: 17, 6: 8, 7: 6, 8: 1, 10: 1}	{60: 530, 61: 260, 62: 127, 63: 73, 64: 33, 65: 16, 66: 8, 67: 2, 68: 1, 69
22	11	STUCK1	1,15	{0: 423, 1: 279, 2: 158, 3: 67, 4: 40, 5: 17, 6: 7, 7: 6, 8: 2, 9: 1}	{15: 577, 16: 298, 17: 140, 18: 73, 19: 33, 20: 16, 21: 9, 22: 3, 23: 1}
23	11	STUCK0	1,011	{0: 477, 1: 276, 2: 119, 3: 68, 4: 34, 5: 10, 6: 8, 7: 6, 8: 1, 9: 1}	{105: 523, 106: 247, 107: 128, 108: 60, 109: 26, 110: 16, 111: 8, 112:
24	12	STUCK1	0,612	{0: 741, 1: 84, 2: 82, 3: 46, 4: 24, 5: 15, 6: 4, 7: 2, 8: 1, 9: 1}	{38: 47, 39: 238, 40: 140, 41: 99, 42: 50, 43: 25, 44: 7, 45: 4, 46: 2}
25	12	STUCK0	1,285	{0: 357, 1: 335, 2: 147, 3: 79, 4: 42, 5: 17, 6: 9, 7: 7, 8: 3, 9: 1, 10: 3}	{51: 643, 52: 308, 53: 161, 54: 82, 55: 40, 56: 23, 57: 14, 58: 7, 59: 4, 6
26	13	STUCK1	0,43	{0: 777, 1: 112, 2: 52, 3: 39, 4: 11, 5: 5, 6: 2, 7: 1, 9: 1}	{71: 223, 72: 111, 73: 59, 74: 20, 75: 9, 76: 4, 77: 2, 78: 1, 79: 1}
27	13	STUCK0	0,725	{0: 626, 1: 188, 2: 106, 3: 37, 4: 21, 5: 11, 6: 8, 7: 1, 9: 1, 11: 1}	{105: 374, 106: 186, 107: 80, 108: 43, 109: 22, 110: 11, 111: 3, 112: 2,
28	14	STUCK1	1,613	{0: 199, 1: 394, 2: 201, 3: 110, 4: 51, 5: 22, 6: 7, 7: 8, 8: 3, 9: 2, 10: 2, 13: 1}	{111: 801, 112: 407, 113: 206, 114: 96, 115: 45, 116: 23, 117: 16, 118:
29	14	STUCK0	0,117	{0: 937, 1: 33, 2: 15, 3: 9, 4: 4, 5: 1, 6: 1}	{65: 63, 66: 30, 67: 15, 68: 6, 69: 2, 70: 1}
30	15	STUCK1	0,98	{0: 530, 1: 224, 2: 114, 3: 63, 4: 32, 5: 20, 6: 12, 7: 1, 8: 4}	{44: 470, 45: 246, 46: 132, 47: 69, 48: 37, 49: 17, 50: 5, 51: 4}
31	15	STUCK0	0,088	{0: 967, 1: 13, 2: 7, 3: 4, 4: 3, 5: 2, 6: 2, 7: 1, 8: 1}	{59: 33, 60: 20, 61: 13, 62: 9, 63: 6, 64: 4, 65: 2, 66: 1}
32	16	STUCK1	1,036	{0: 488, 1: 258, 2: 119, 3: 69, 4: 34, 5: 12, 6: 12, 7: 4, 8: 1, 9: 1, 10: 2}	{41: 512, 42: 254, 43: 135, 44: 66, 45: 32, 46: 20, 47: 8, 48: 4, 49: 3, 50

Results

Figure 3.1: Example spreadsheet of DADDA Multiplier Stuck at Injections

As can be seen above, the column Average Erroneous Bits shows how many bits of the output, on average, were faulty, the column Erroneous Bits contains the number of tests having a specific number of errors, and the column Erroneous Bit Position contains the position of erroneous bits for each test.

By utilizing data from the spreadsheets generated for each arithmetic unit, graphs visualizing the effects of each type of faults injected can be produced.

Over the course of the next chapter, those graphs will be thoroughly presented.

4. RESULTS

4.1 Method of presentation

In the following chapters, the results produced by testing the aforementioned arithmetic units will be presented.

All the graphs are based on the spreadsheets produced by the testing methodology discussed extensively previously.

After presenting the distribution of faulty bits on the outputs of the modules, and the average number of errors per operation, the results will be commented.

For tests using Bridging Fault injections, 1000 different injections were done, with varying number of operations on each injection, depending on the test.

For tests using Stuck-at Fault injections, 2000 different injections were done (for half of which the random wire was stuck at High, and the other half the chosen wire was stuck at Low), again with varying number of operations done on each injection, depending on the test.

In all cases, the number of operations done for each injection will be noted.

Furthermore, the coverage (that is, the percentage of hardware faults *(caused by the injections done)* which, after being tested with a test-set *(random numbers or specific patterns)* yielded to at least one noticeable erroneous result) for each test method will be presented.

For modules where multiple test methods were used (i.e., the DADDA multiplier unit), the coverage of all test methods will be presented in order for comparison purposes.

Finally, the coverage for all tested modules and all testing methods for modules where multiple testing methods were utilized will be presented in order to observe which modules are more sensitive to output errors (for the given testing patterns).

4.2 DADDA Integer Multiplier

Input: Two 64 Bit Integer Numbers

Output: One 128 Bit Integer Number

4.2.1 Bridging Fault Injection

Testing 1000 random numbers for each injection Fault Detection coverage: 100%

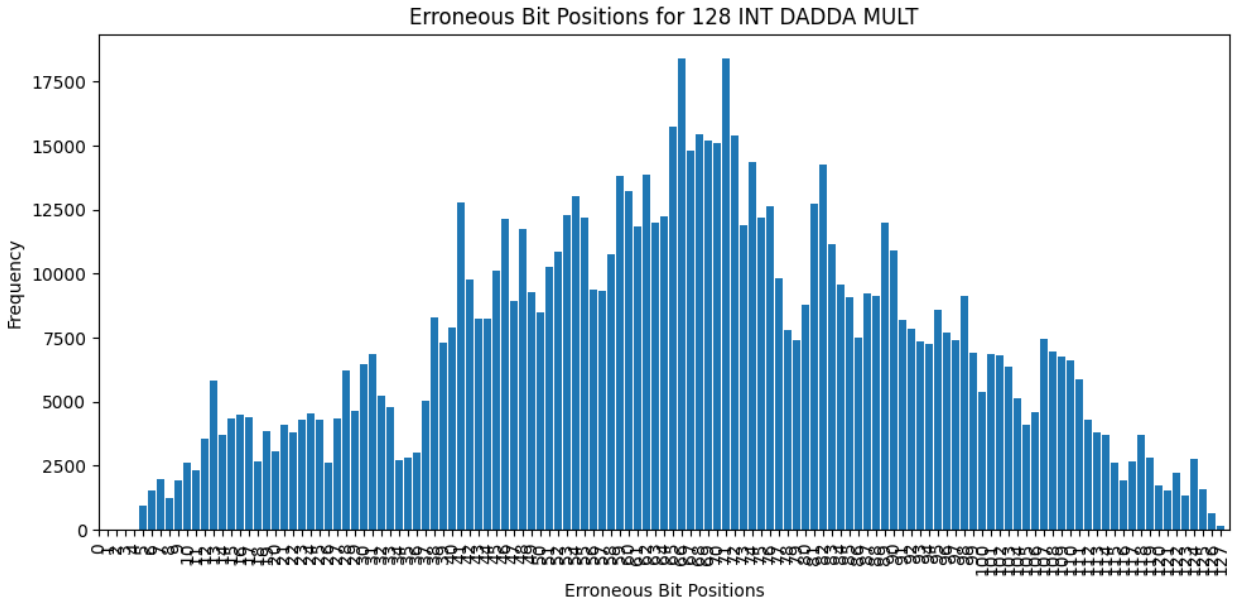


Figure 4.1: The distribution of erroneous bits on the output of DADDA Integer Multiplier when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

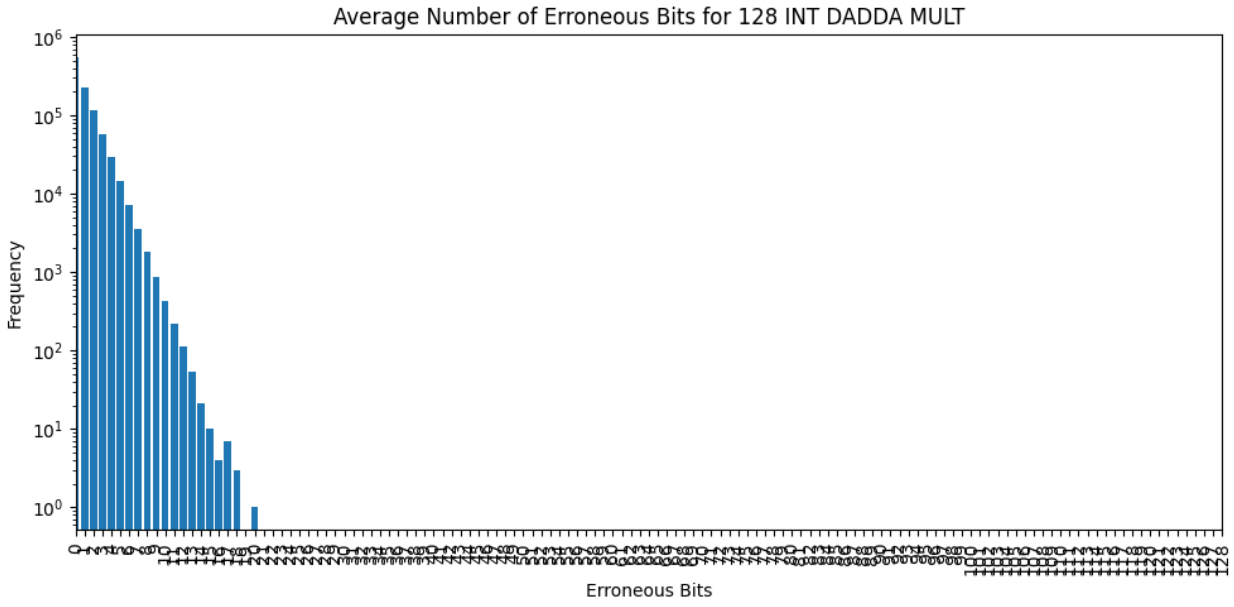


Figure 4.2: The average number of erroneous bits per operation for DADDA Integer Multiplier when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

Testing 256 numbers using specific 4Bit patterns [10] for each injection Fault Detection coverage: 99.6%

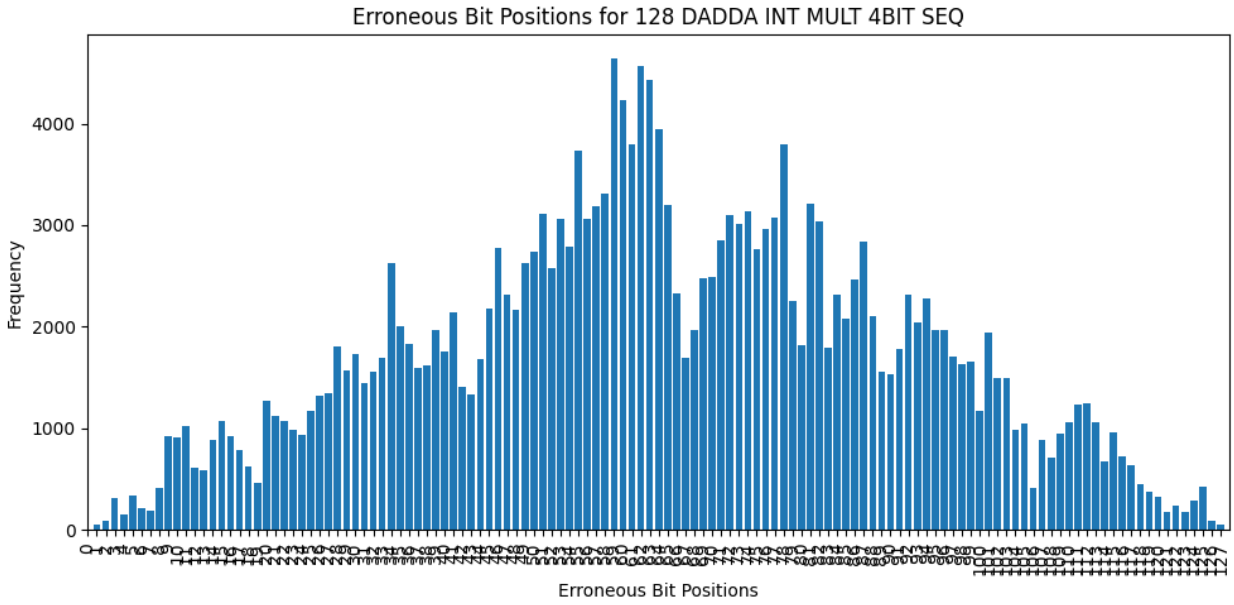


Figure 4.3: The distribution of erroneous bits on the output of DADDA Integer Multiplier when 1000 different Bridging Faults were injected and 256 operations using specific 4Bit patterns were done for each injection

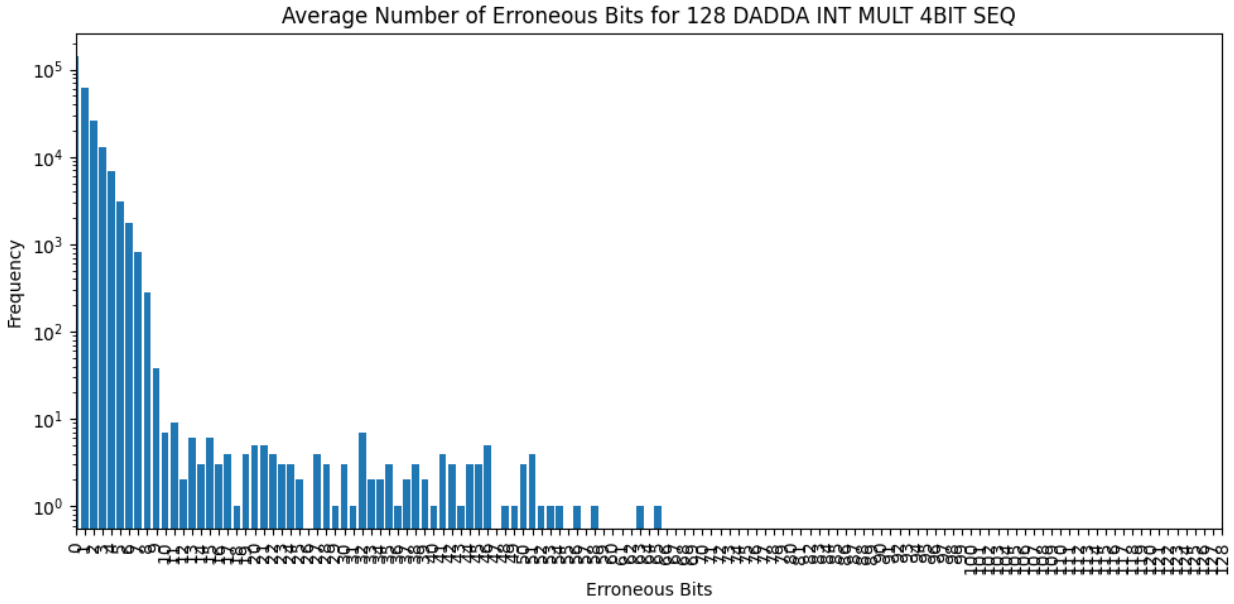


Figure 4.4: The average number of erroneous bits per operation for DADDA Integer Multiplier when 1000 different Bridging Faults were injected and 256 operations using specific 4Bit patterns were done for each injection

Testing 256 random numbers for each injection Fault Detection coverage: 100%

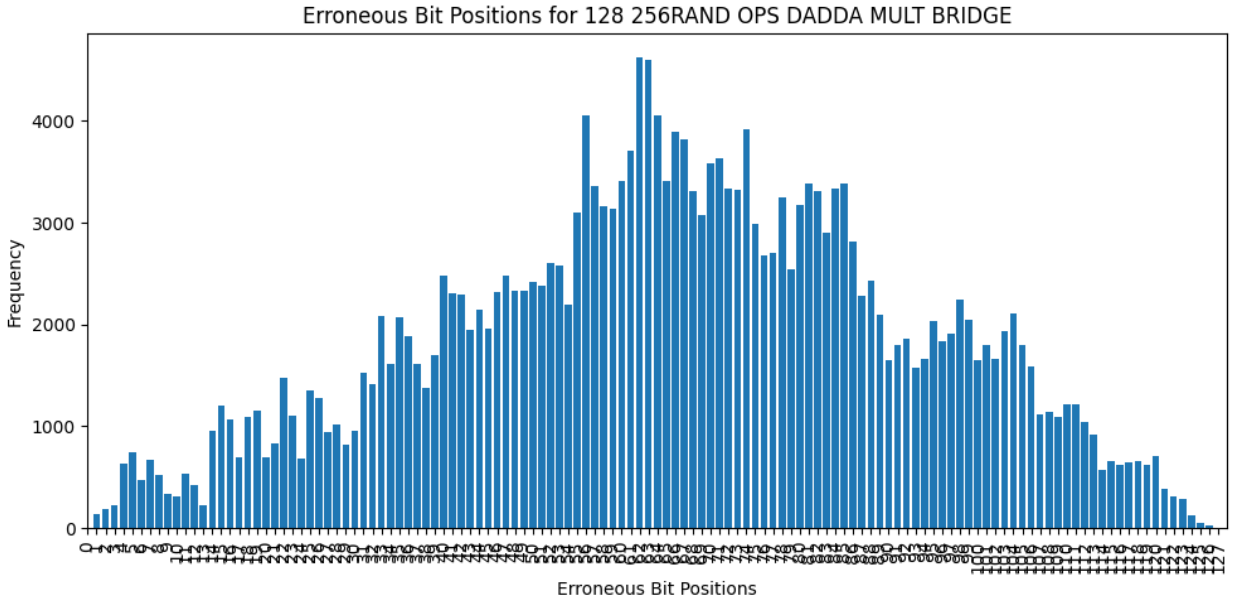


Figure 4.5: The distribution of erroneous bits on the output of DADDA Integer Multiplier when 1000 different Bridging Faults were injected and 256 random operations were done for each injection

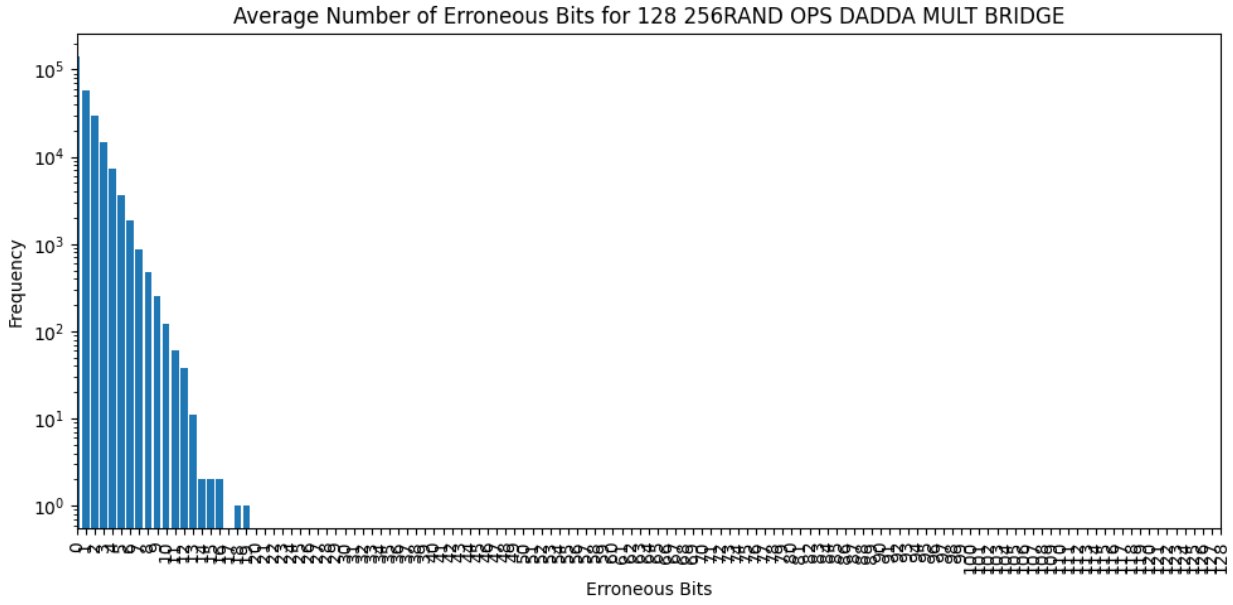


Figure 4.6: The average number of erroneous bits per operation for DADDA Integer Multiplier when 1000 different Bridging Faults were injected and 256 random operations were done for each injection

4.2.2 Stuck-At Fault Injection

Testing 1000 random numbers for each injection Fault Detection coverage: 99.9%

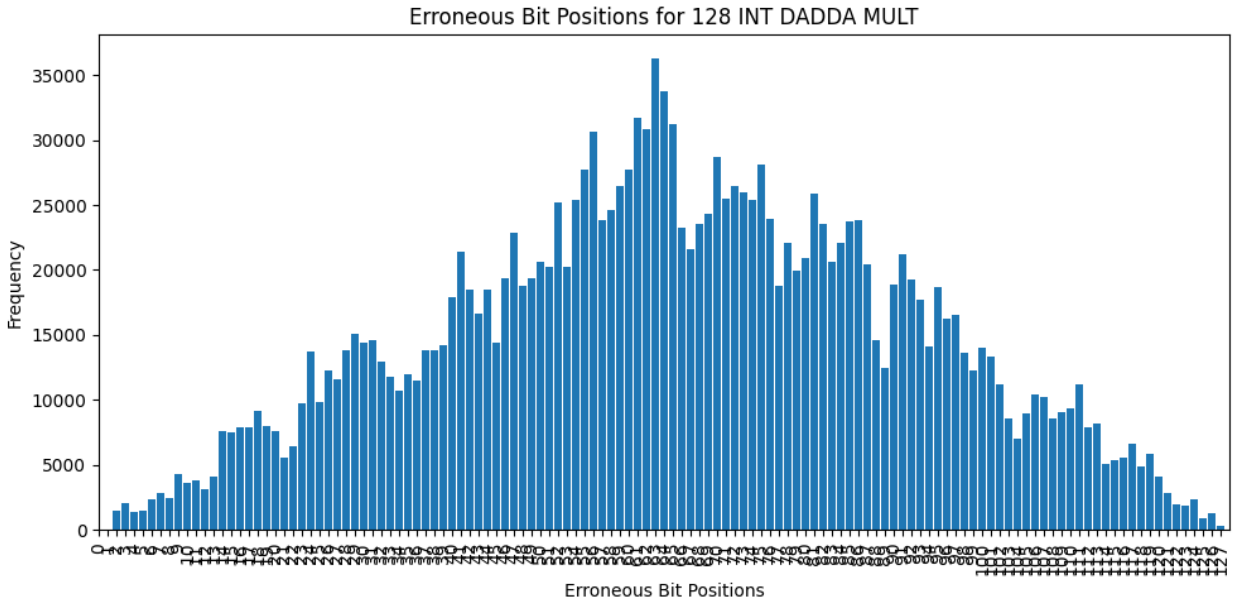


Figure 4.7: The distribution of erroneous bits on the output of DADDA Integer Multiplier when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

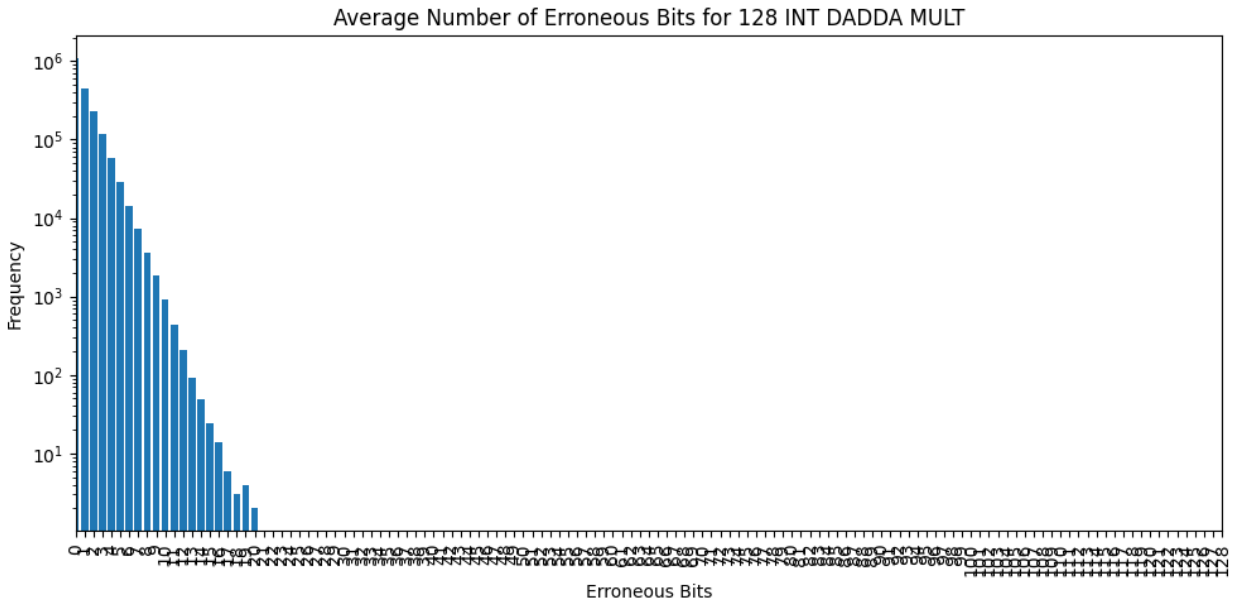


Figure 4.8: The average number of erroneous bits per operation for DADDA Integer Multiplier when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

Testing 256 numbers using specific 4Bit patterns [10] for each injection

Fault Detection coverage: 100%

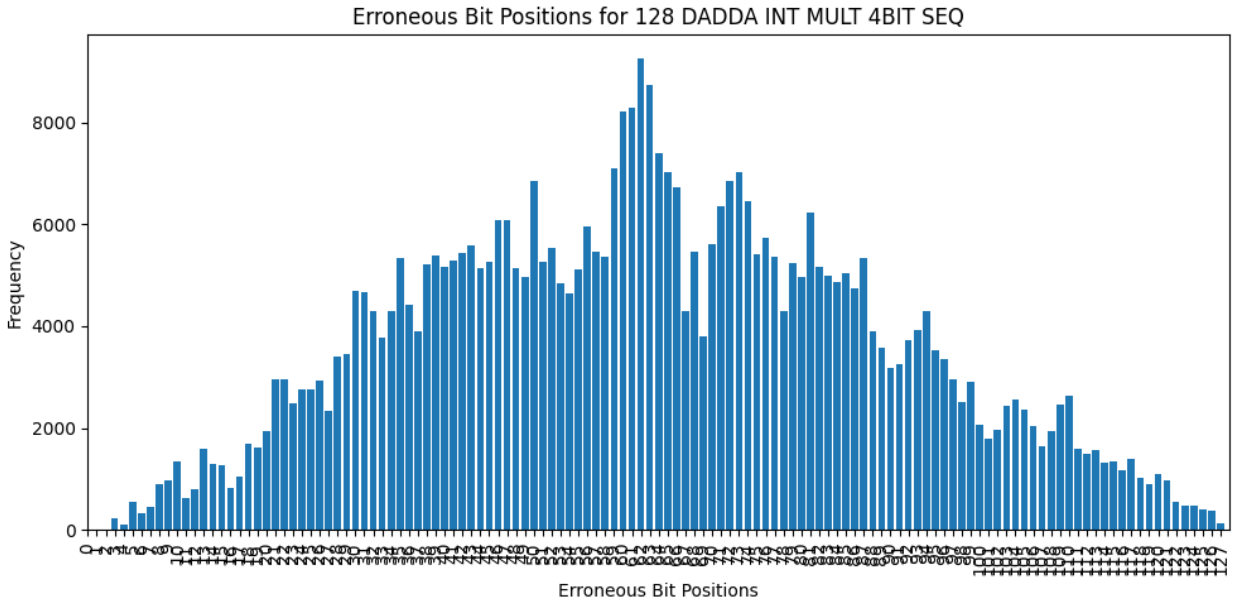


Figure 4.9: The distribution of erroneous bits on the output of DADDA Integer Multiplier when 2000 different Stuck-At Faults were injected and 256 operations using specific 4Bit patterns were done for each injection

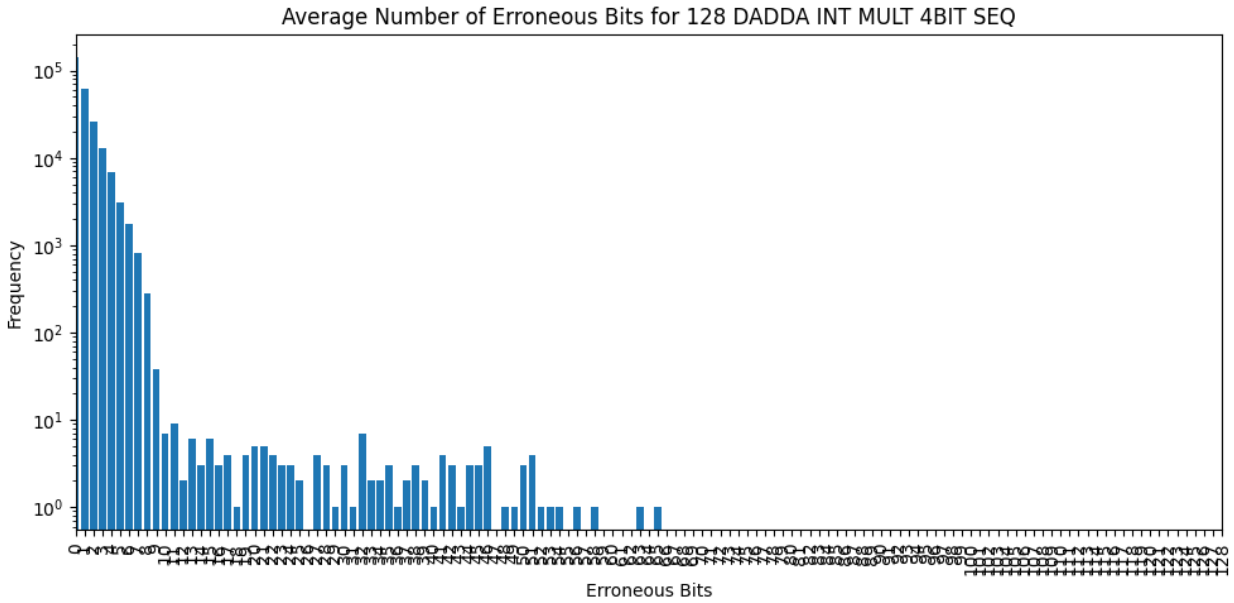


Figure 4.10: The average number of erroneous bits per operation for DADDA Integer Multiplier when 2000 different Stuck-At Faults were injected and 256 operations using specific 4Bit patterns were done for each injection

Testing 256 random numbers for each injection

Fault Detection coverage: 99.45%

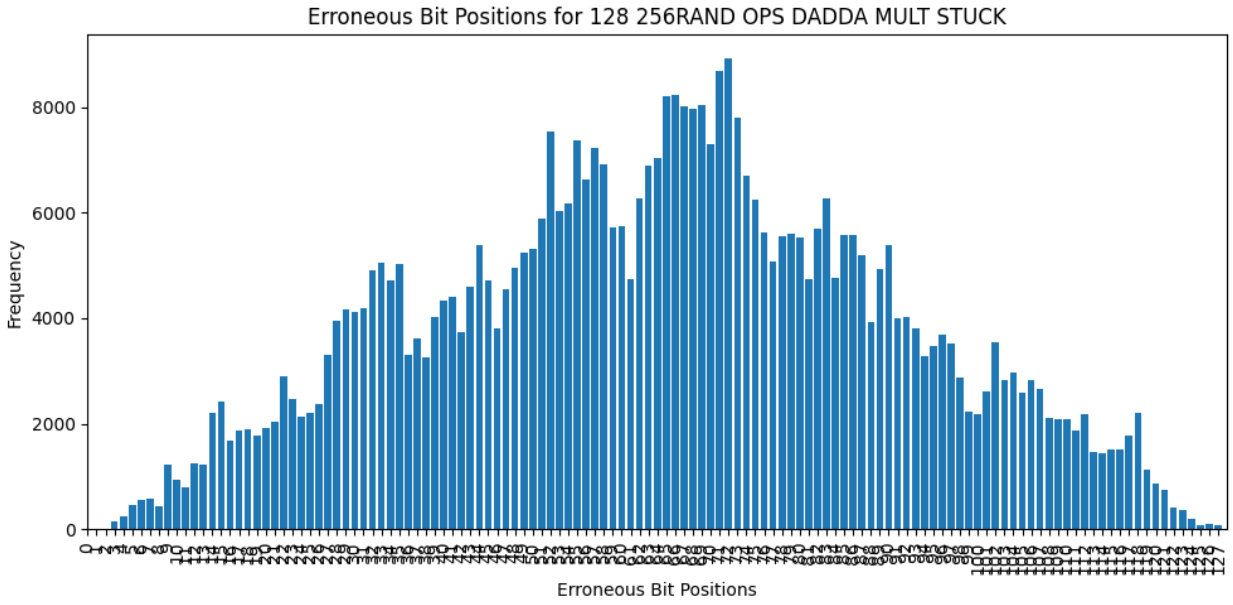


Figure 4.11: The distribution of erroneous bits on the output of DADDA Integer Multiplier when 2000 different Stuck-At Faults were injected and 256 random operations were done for each injection

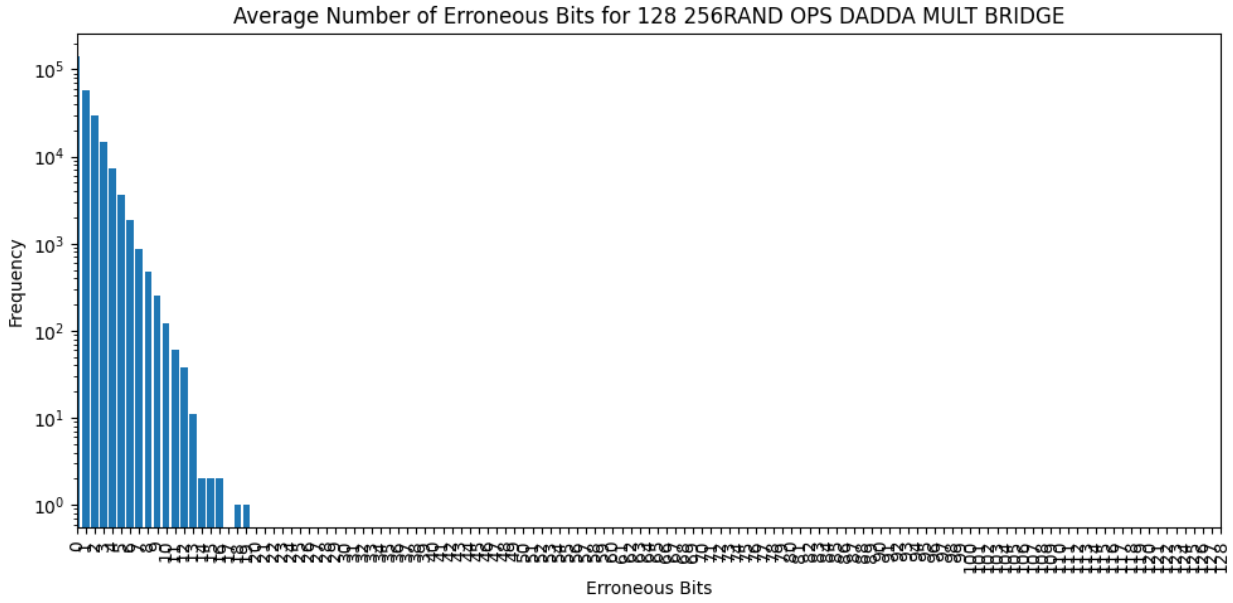


Figure 4.12: The average number of erroneous bits per operation for DADDA Integer Multiplier when 2000 different Stuck-At Faults were injected and 256 random operations were done for each injection

Note: In order to compare fairly the two testing methods (that is, to test with the same number of operations per injection), 1000 injections with 256 random operations were also done

4.2.3 Comments

As can be seen, in all test scenarios, the middle bits of the output were the ones mostly affected. As can be seen by the coverage charts, for the DADDA Multiplier, and most likely for the other multiplier units as well [10], using the 4Bit patterns is the most effective way of detecting possible hardware errors, because we get almost equal, if not better, coverage and a higher number of average faulty bits. This means that the error is easier to manifest and, thus, the detection of a possibly faulty chip is facilitated.

4.3 Array Integer Multiplier

Input: Two 64 Bit Integer Numbers

Output: One 128 Bit Integer Number

4.3.1 Bridging Fault Injection

Testing 1000 random numbers for each injection Fault Detection coverage: 100%

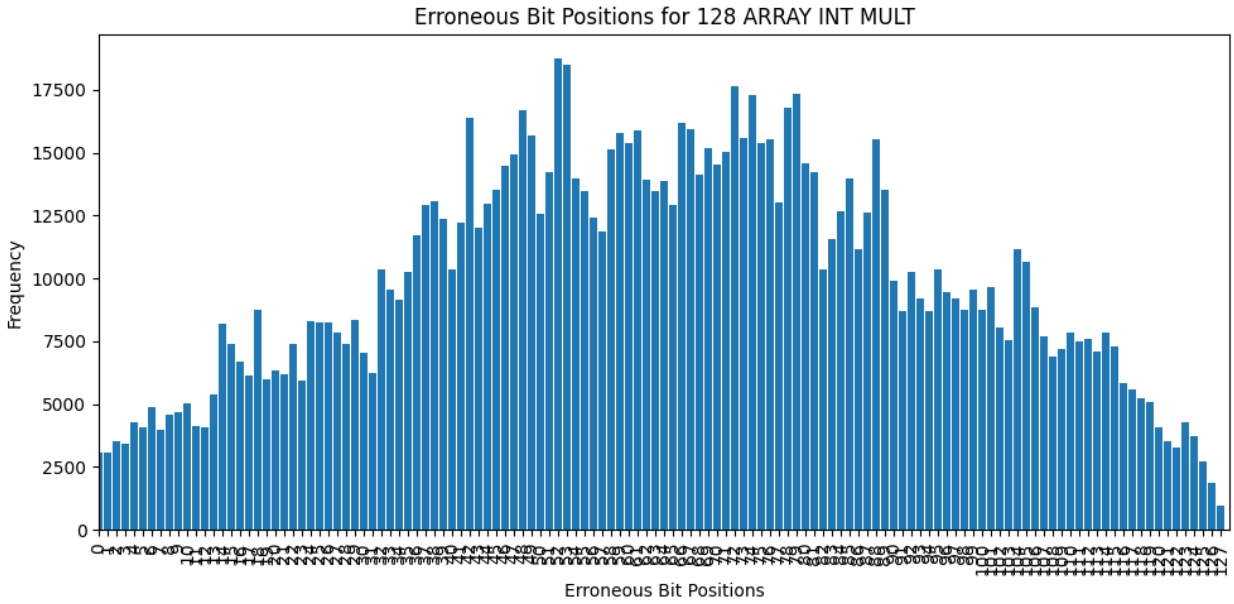


Figure 4.13: The distribution of erroneous bits on the output of Array Integer Multiplier when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

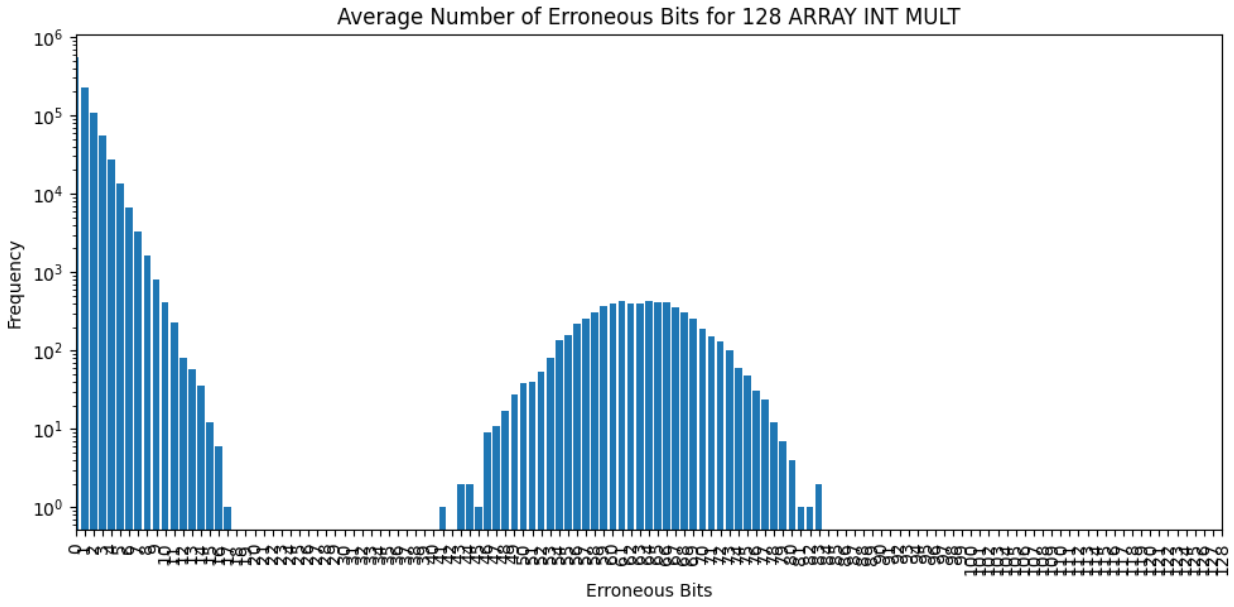


Figure 4.14: The average number of erroneous bits per operation for Array Integer Multiplier when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

4.3.2 Stuck-At Fault Injections

Testing 1000 random numbers for each injection Fault Detection coverage: 99.95%

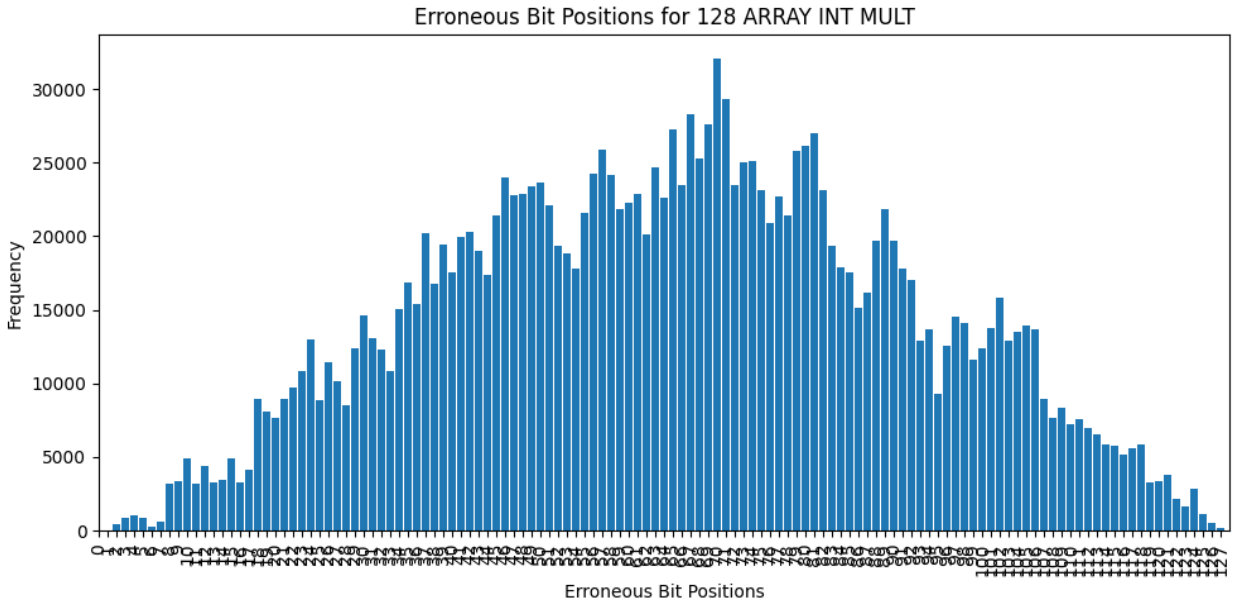


Figure 4.15: The distribution of erroneous bits on the output of DADDA Integer Multiplier when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

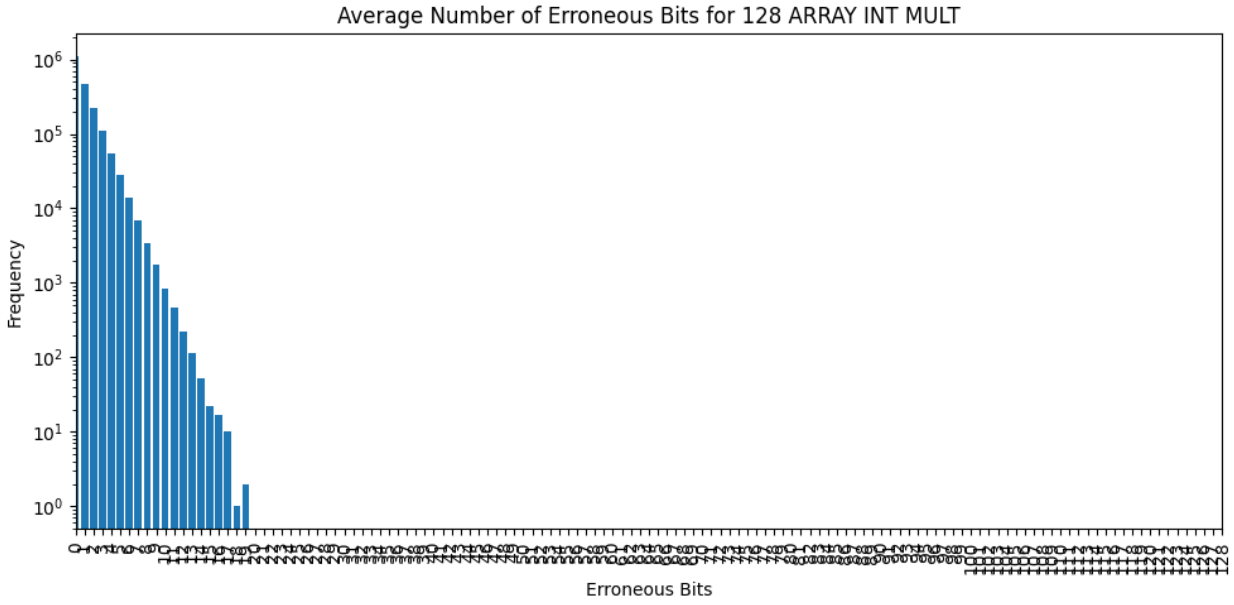


Figure 4.16: The average number of erroneous bits per operation for Array Integer Multiplier when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

4.3.3 Comments

Again, most of the faulty bits are located on the middle of the output, with the distribution approaching the normal one. It is worth noticing that in the case of Bridging Fault Injection

for the Array Integer Multiplier, it was way more frequent to have a high number of faulty bits on the output.

In both cases, the detection rates using 1000 random numbers per test were pretty high.

4.4 Wallace Integer Multiplier

Input: Two 64 Bit Integer Numbers

Output: One 128 Bit Integer Number

4.4.1 Bridging Fault Injection

Testing 1000 random numbers for each injection Fault Detection coverage: 100%

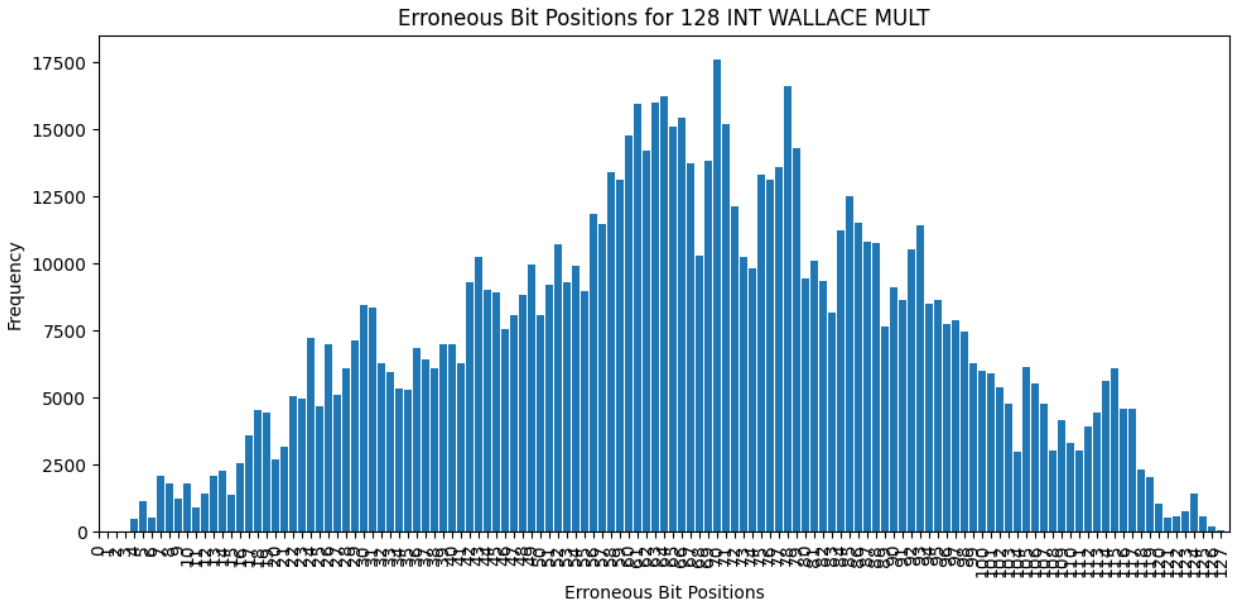


Figure 4.17: The distribution of erroneous bits on the output of Wallace Integer Multiplier when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

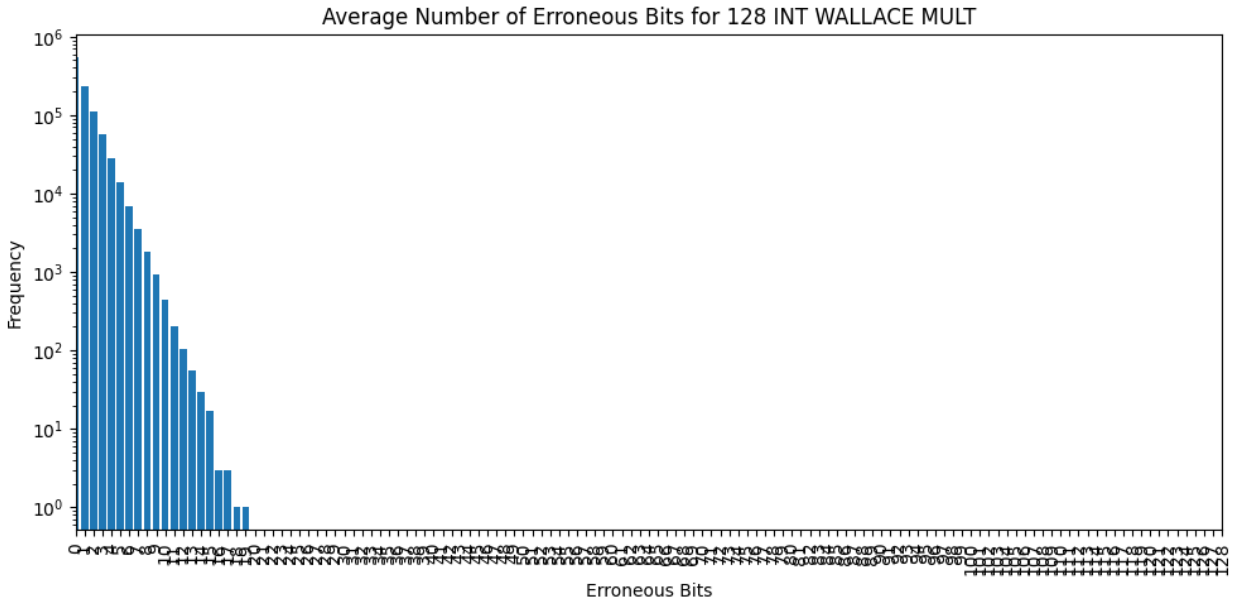


Figure 4.18: The average number of erroneous bits per operation for Wallace Integer Multiplier when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

4.4.2 Stuck-At Fault Injections

Testing 1000 random numbers for each injection Fault Detection coverage: 100%

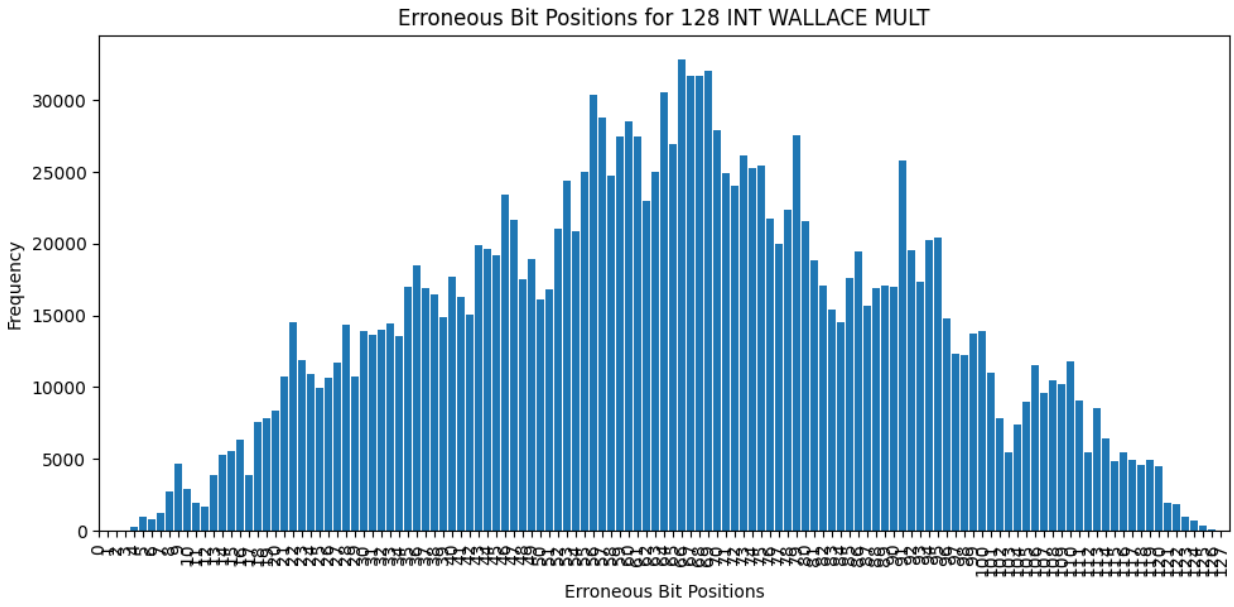


Figure 4.19: The distribution of erroneous bits on the output of Wallace Integer Multiplier when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

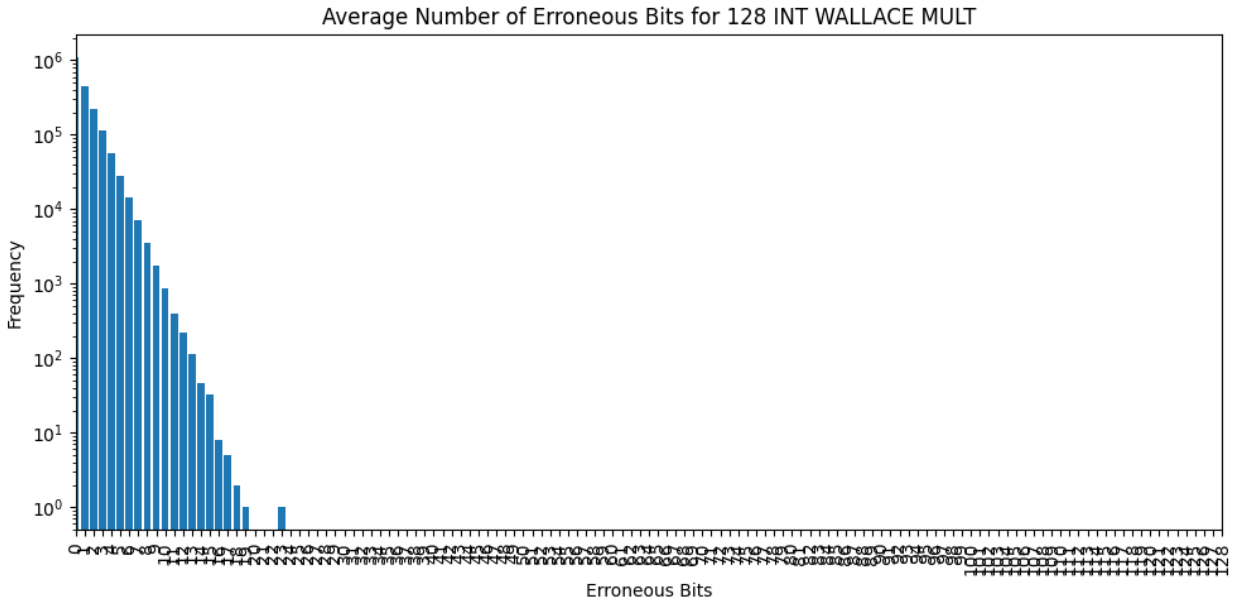


Figure 4.20: The average number of erroneous bits per operation for Wallace Integer Multiplier when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

4.4.3 Comments

The response of the Wallace Integer Multiplier was pretty similar for both bridging fault injections and stuck-at fault injections. The distribution of faulty bits is similar to the other multiplier units. The coverage is excellent as all faults were detected.

4.5 Floating Point Unit: Adder

Input: Two 64 Bit Float IEEE 754 Double Numbers

Output: One 64 Bit Float IEEE 754 Double Number

Note: For the FPU Adder module, when the adder was tested, logic from the FPU Multiplier and other sub-modules has been disabled so the injections affect the logic of the FPU Adder only.

4.5.1 Bridging Fault Injection

Testing 2000 random numbers for each injection (1000 additions + 1000 subtractions)

Fault Detection coverage: 67.3%

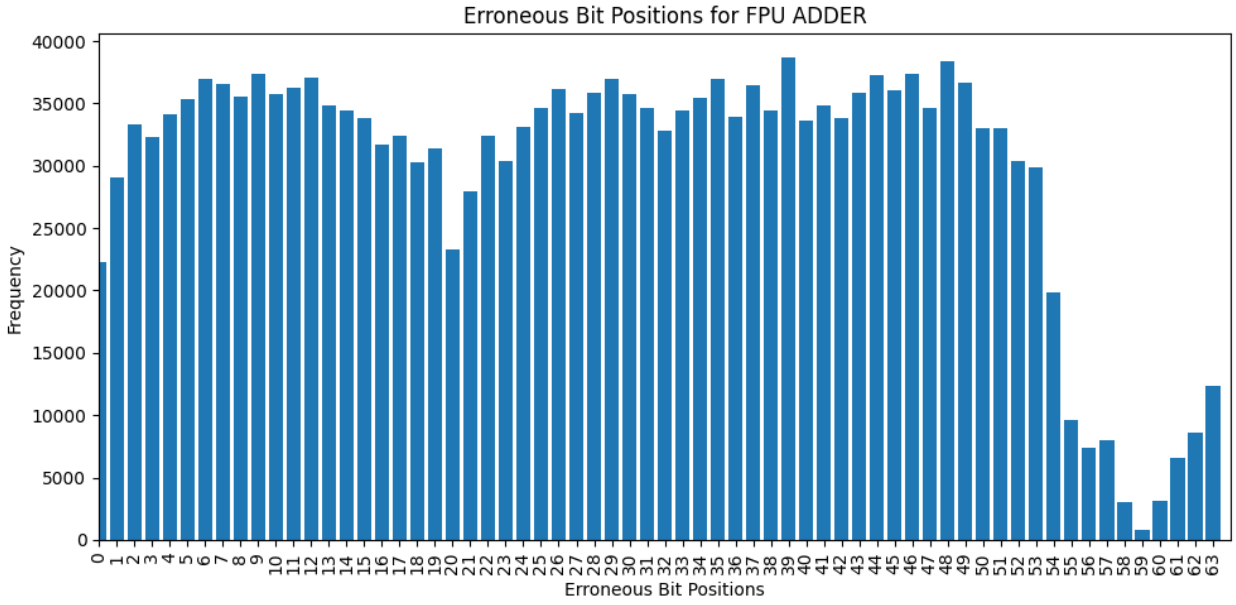


Figure 4.21: The distribution of erroneous bits on the output of the FPU Adder when 1000 different Bridging Faults were injected and 2000 random operations (1000 additions and 1000 subtractions) were done for each injection

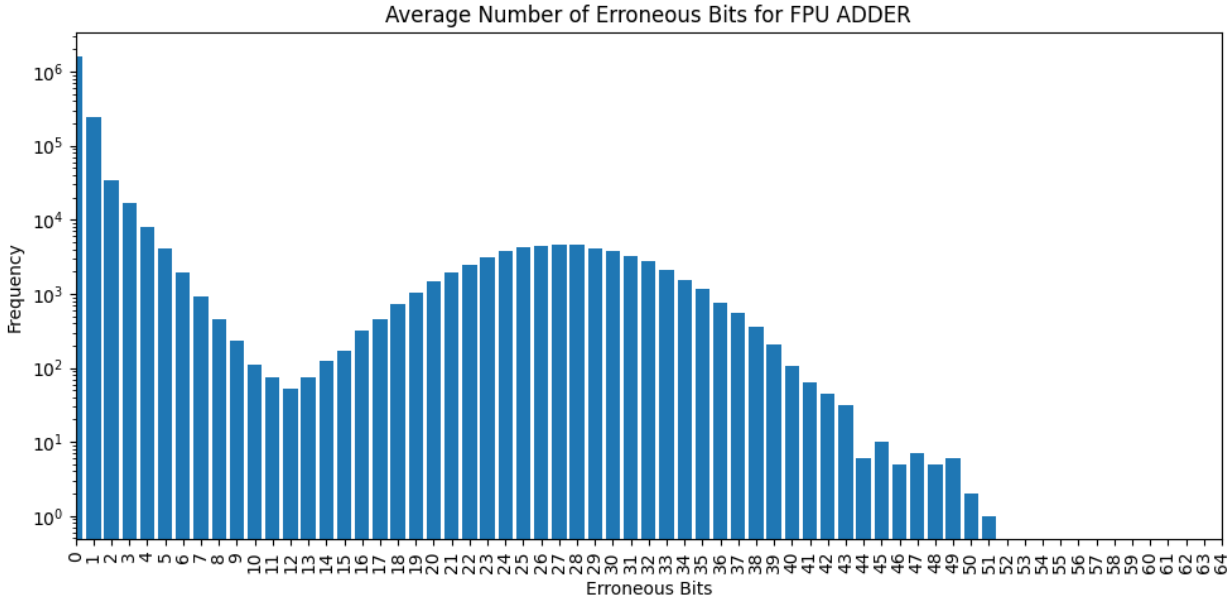


Figure 4.22: The average number of erroneous bits per operation for the FPU Adder when 1000 different Bridging Faults were injected and 2000 random operations (1000 additions and 1000 subtractions) were done for each injection

4.5.2 Stuck-At Fault Injections

Testing 2000 random numbers for each injection (1000 additions + 1000 subtractions)

Fault Detection coverage: 60.25%

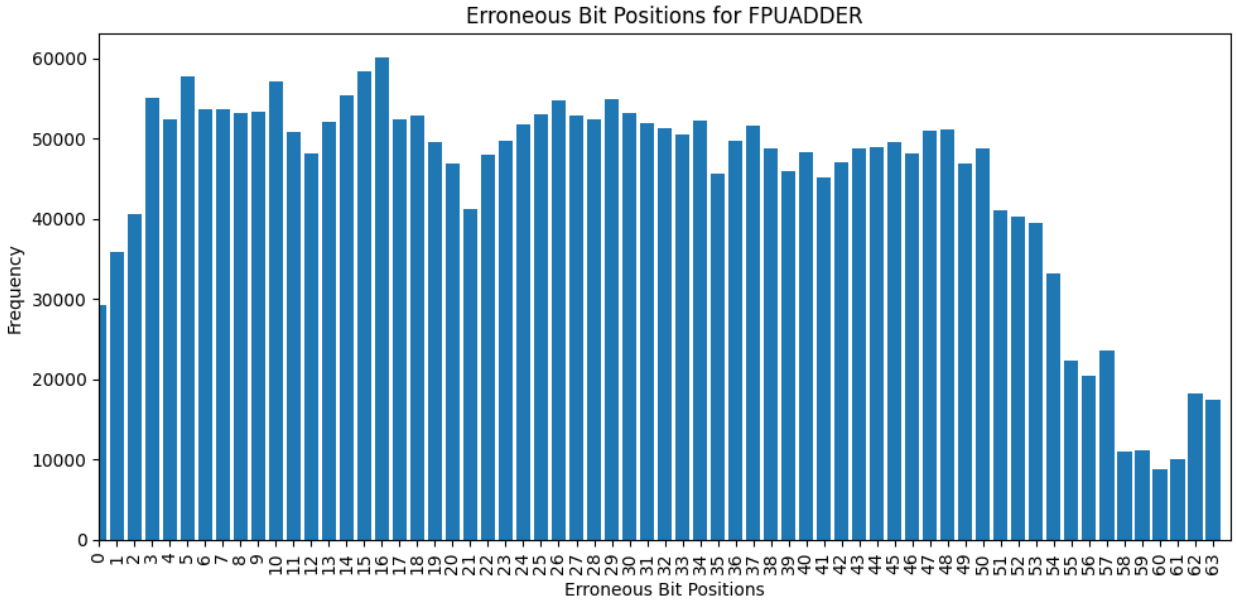


Figure 4.23: The distribution of erroneous bits on the output of the FPU Adder when 2000 different Stuck-At Faults were injected and 2000 random operations (1000 additions and 1000 subtractions) were done for each injection

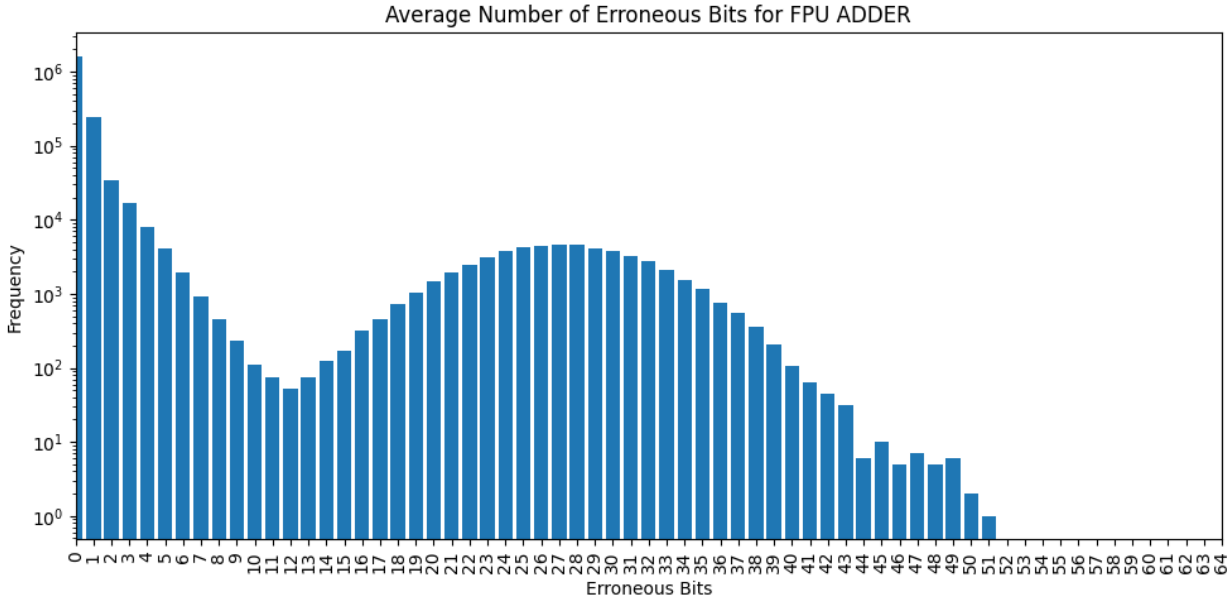


Figure 4.24: The average number of erroneous bits per operation for the FPU Adder when 2000 different Stuck-At Faults were injected and 2000 random operations (1000 additions and 1000 subtractions) were done for each injection

4.5.3 Comments

Both the distribution and the average number of errors caused by fault injecting the Floating Point Unit Adder differs significantly from the ones of the Integer Units. The most affected bits of the output are the ones of the exponent and the high digits of the mantissa. Also, the coverage rates are way lower than the integer modules. That can be linked to the FPU modules being significantly more complex and, thus, having way more possible separate data paths, contrary to the integer multiplier, which usually has a tree-like structure with fewer possible data paths. As such, the probability of an operation not going through the problematic data path is increased.

4.6 Floating Point Unit: Multiplier

Input: Two 64 Bit Float IEEE 754 Double Numbers

Output: One 64 Bit Float IEEE 754 Double Number

Note: For the FPU Multiplier module, when the adder was tested, logic from the FPU Adder and other sub-modules has been disabled so the injections affect the logic of the FPU Multiplier only.

4.6.1 Bridging Fault Injection

Testing 1000 random numbers for each injection

Fault Detection coverage: 60.6%

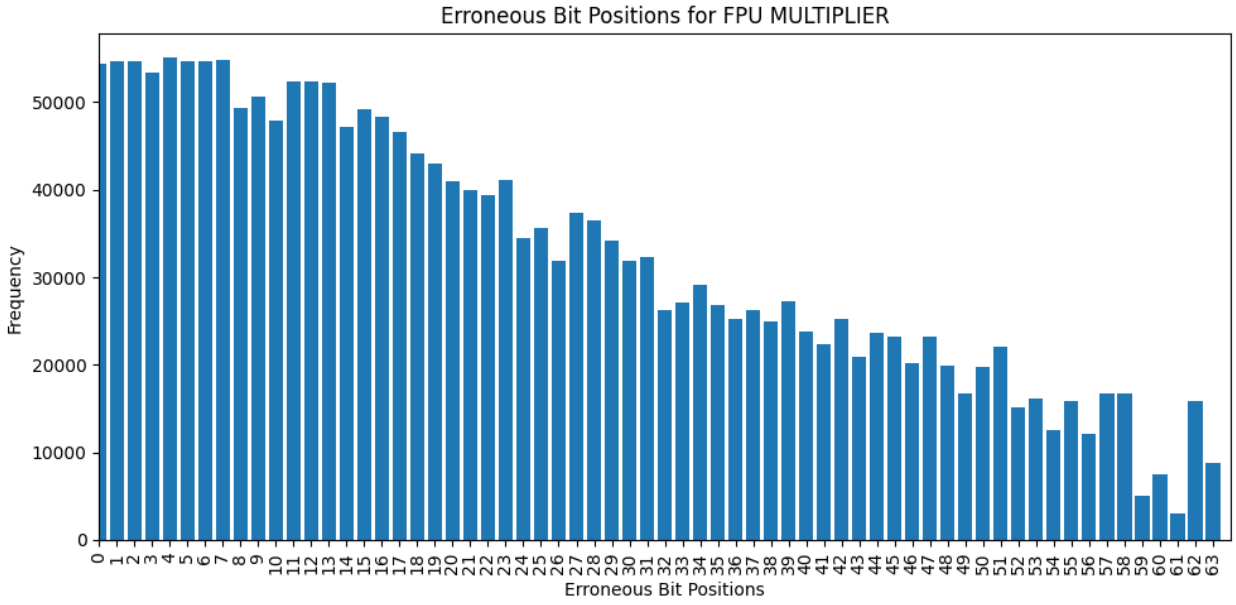


Figure 4.25: The distribution of erroneous bits on the output of the FPU Multiplier when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

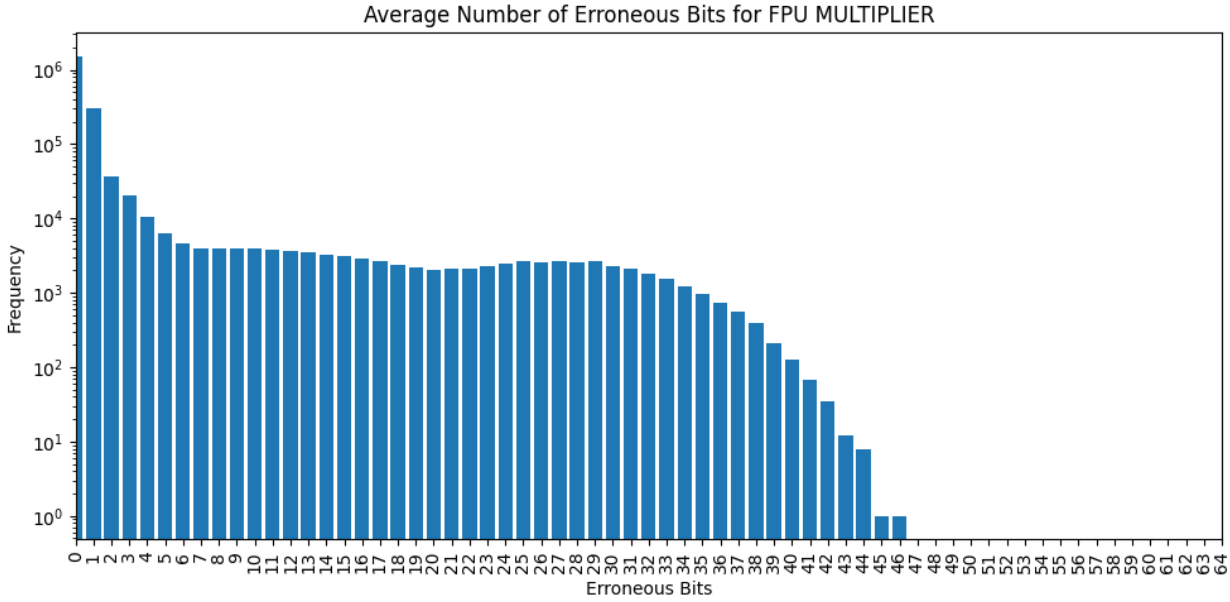


Figure 4.26: The average number of erroneous bits per operation for the FPU Multiplier when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

4.6.2 Stuck-At Fault Injections

Testing 2000 random numbers for each injection (1000 additions + 1000 subtractions)

Fault Detection coverage: 56.45%

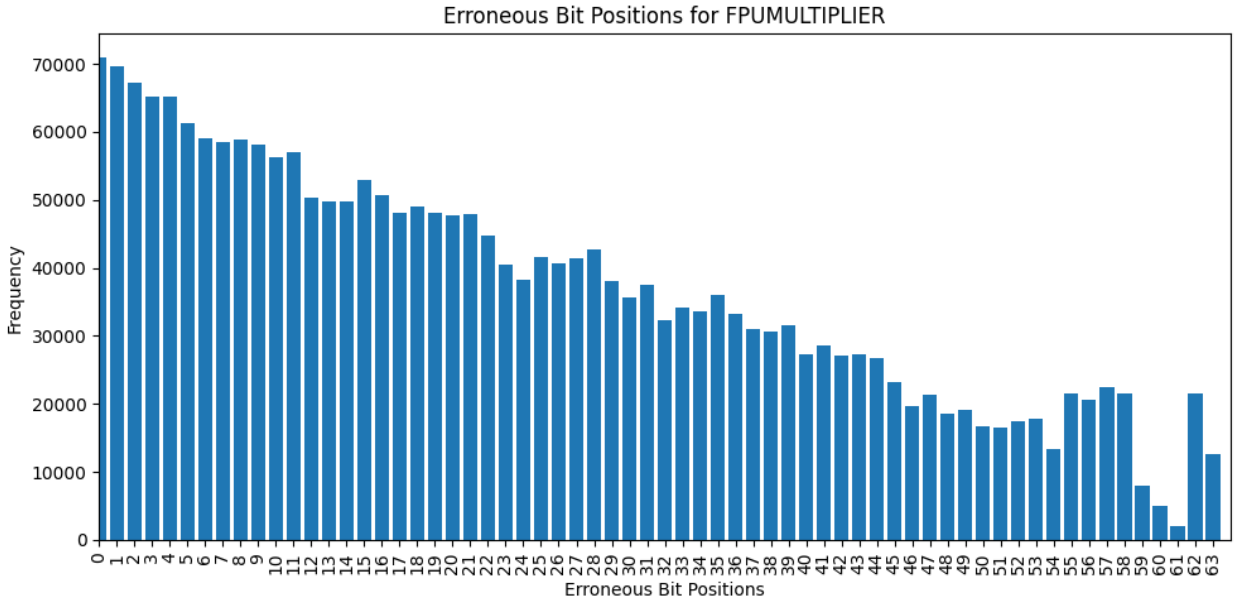


Figure 4.27: The distribution of erroneous bits on the output of the FPU Multiplier when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

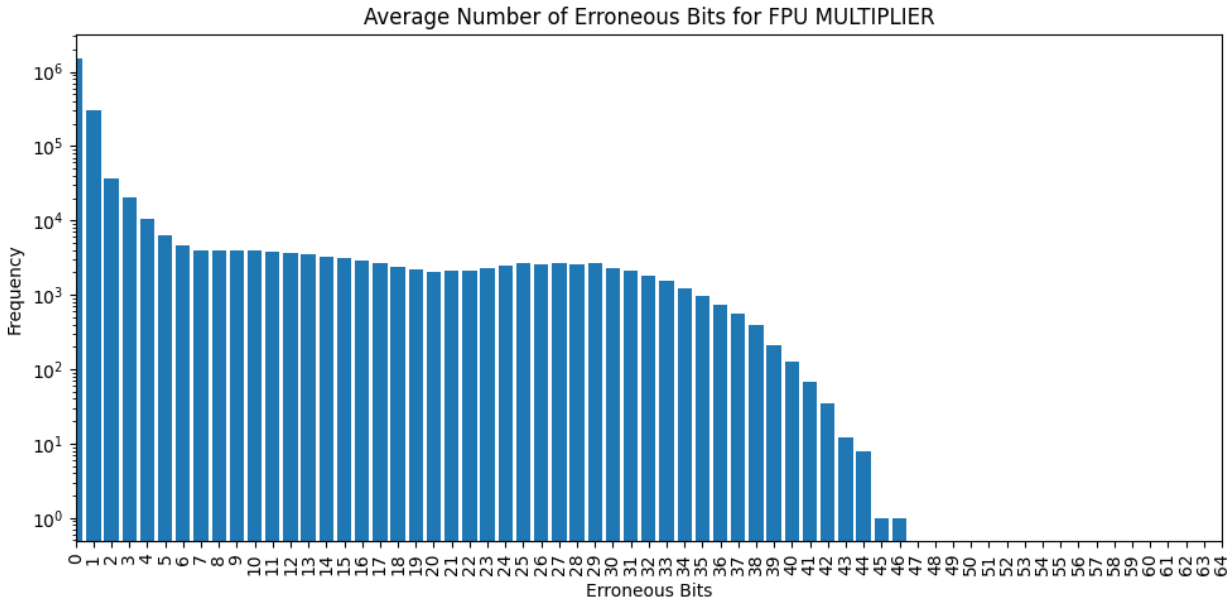


Figure 4.28: The average number of erroneous bits per operation for the FPU Multiplier when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

4.6.3 Comments

Similarly to the FPU Adder, the error distribution and the average number of erroneous bits differs significantly from the ones of the Integer Units. As for the fault detection coverage, the same conclusions with the FPU Adder also apply.

4.7 Integer Adder Using 4 Bit adder blocks

Input: Two 64 Bit Integer Numbers

Output: One 65 Bit Number (highest bit being the carry)

4.7.1 Bridging Fault Injection

Testing 1000 random numbers for each injection Fault Detection coverage: 99.9%

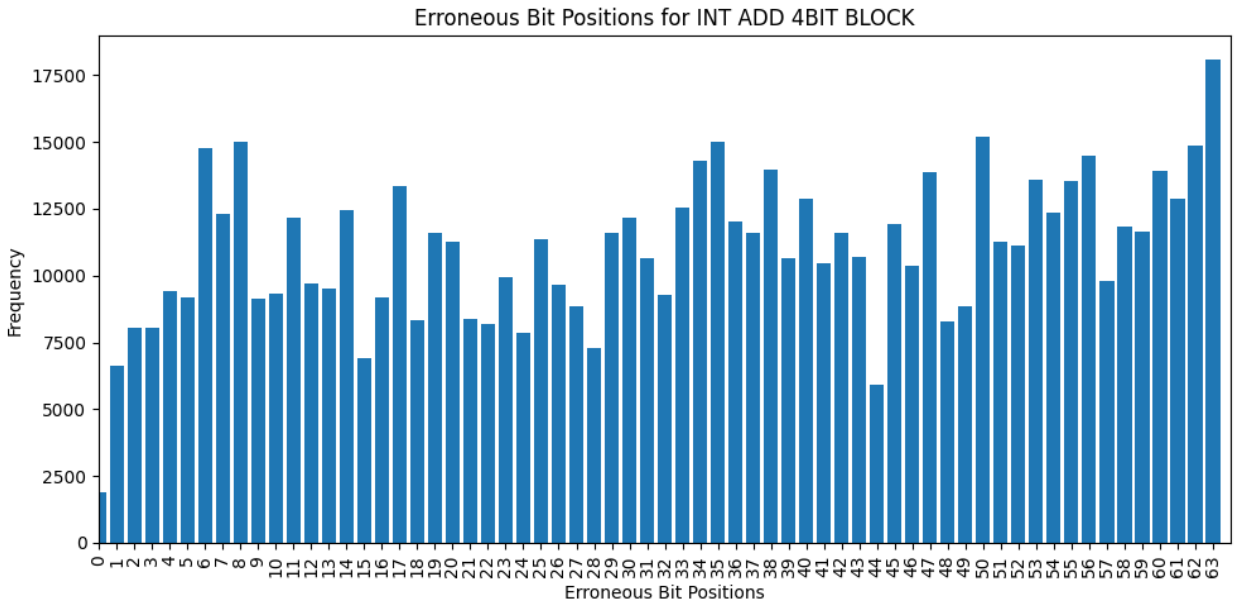


Figure 4.29: The distribution of erroneous bits on the output of Integer Adder Using 4 Bit adder blocks when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

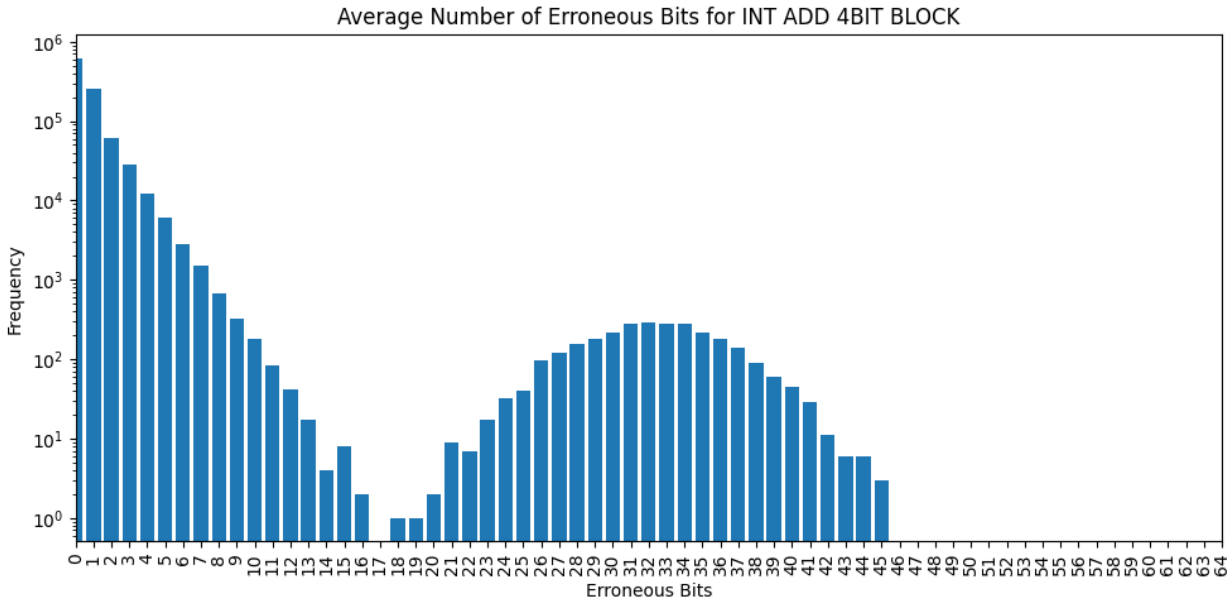


Figure 4.30: The average number of erroneous bits per operation for Integer Adder Using 4 Bit adder blocks when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

4.7.2 Stuck-At Fault Injections

Testing 1000 random numbers for each injection Fault Detection coverage: 100%

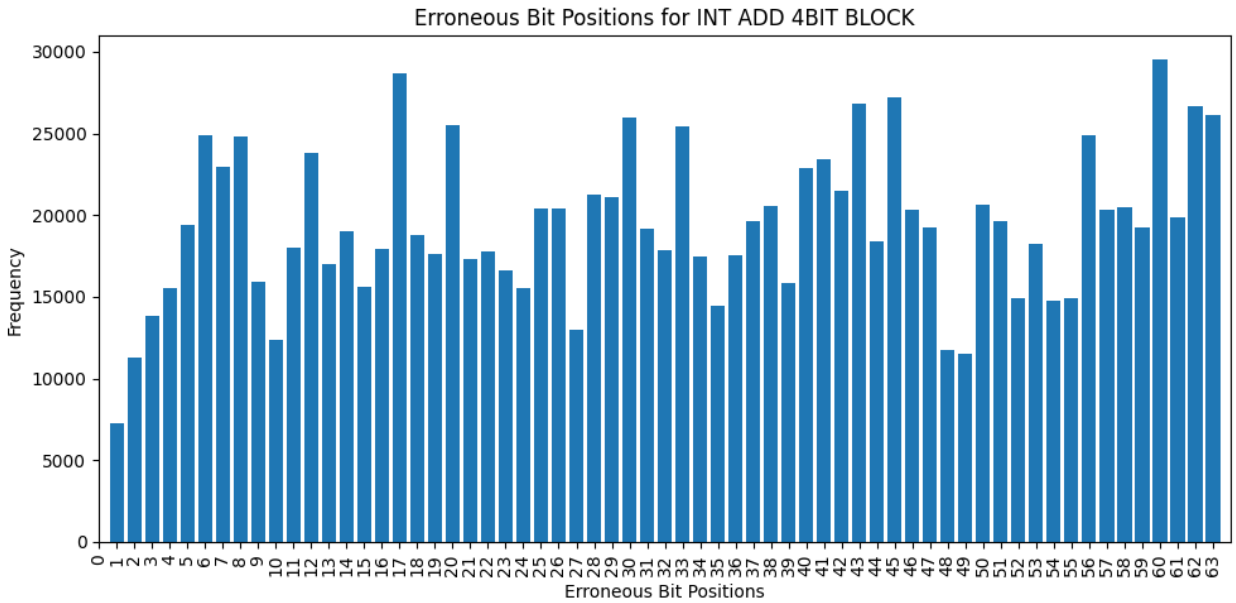


Figure 4.31: The distribution of erroneous bits on the output of Adder Using 4 Bit adder blocks when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

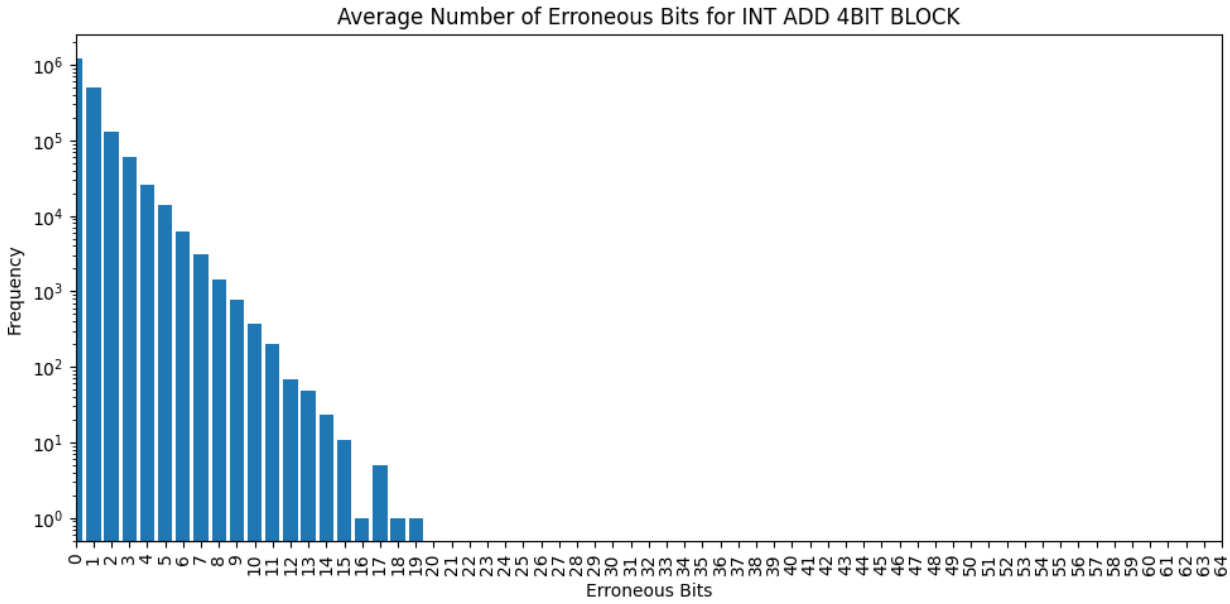


Figure 4.32: The average number of erroneous bits per operation for Adder Using 4 Bit adder blocks when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

4.7.3 Comments

The Fault Rate is pretty similar to the other integer modules when stuck-at injections were done. For bridging faults, a higher fault rate is observed. Also, the faulty bits on the output are more evenly distributed. The detection rates are relatively high.

4.8 Integer Adder Using 32 Bit adder blocks

Input: Two 64 Bit Integer Numbers

Output: One 65 Bit Number (highest bit being the carry)

4.8.1 Bridging Fault Injection

Testing 1000 random numbers for each injection Fault Detection coverage: 82.7%

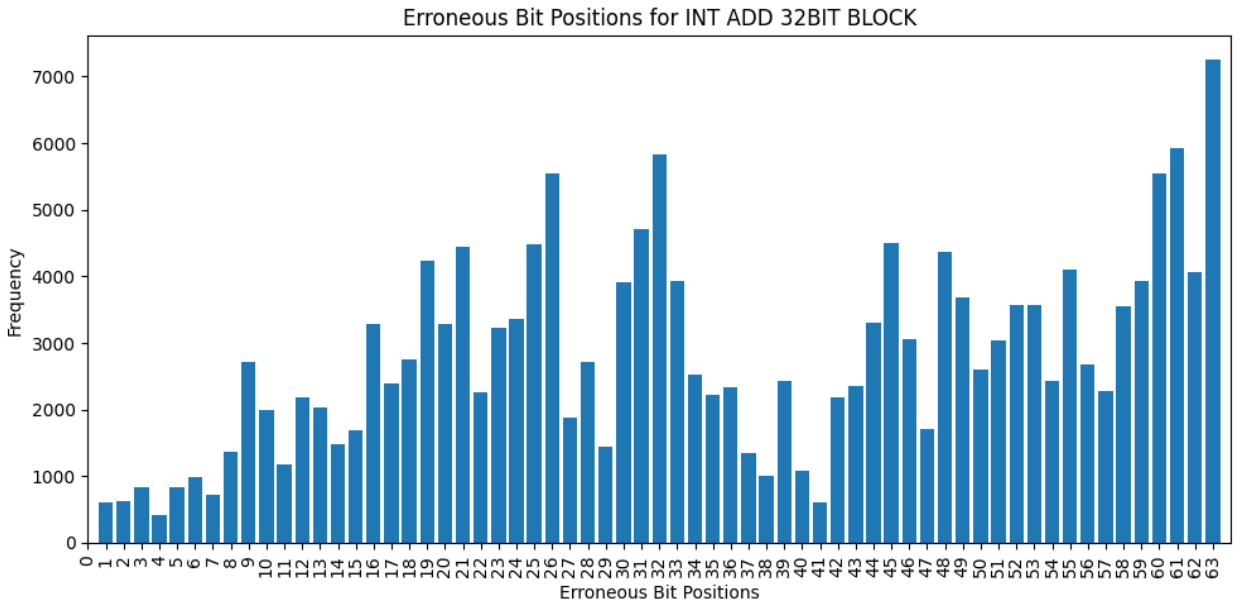


Figure 4.33: The distribution of erroneous bits on the output of Integer Adder Using 32Bit adder blocks when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

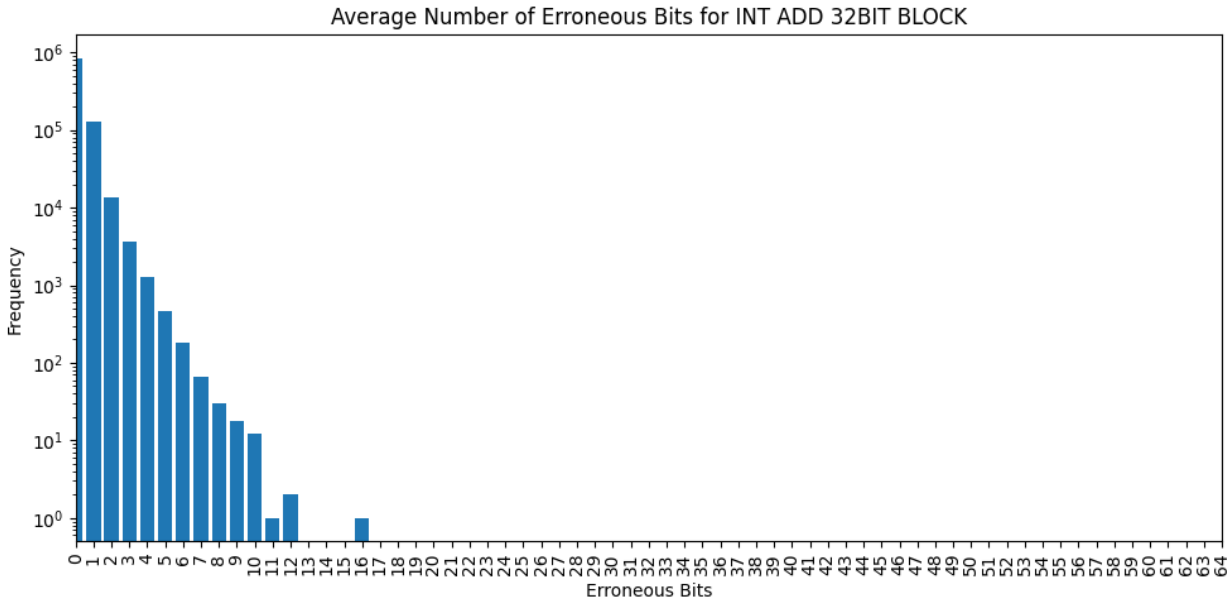


Figure 4.34: The average number of erroneous bits per operation for Integer Adder Using 32Bit adder blocks when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

4.8.2 Stuck-At Fault Injections

Testing 1000 random numbers for each injection Fault Detection coverage: 64.55%

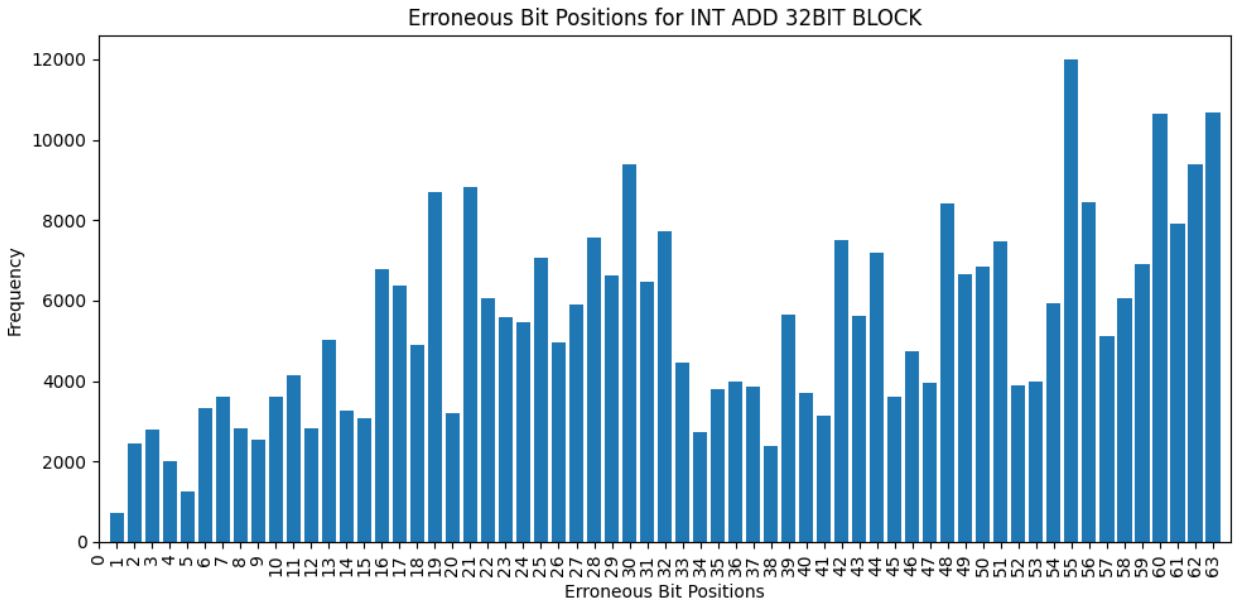


Figure 4.35: The distribution of erroneous bits on the output of Adder Using 32 Bit adder blocks when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

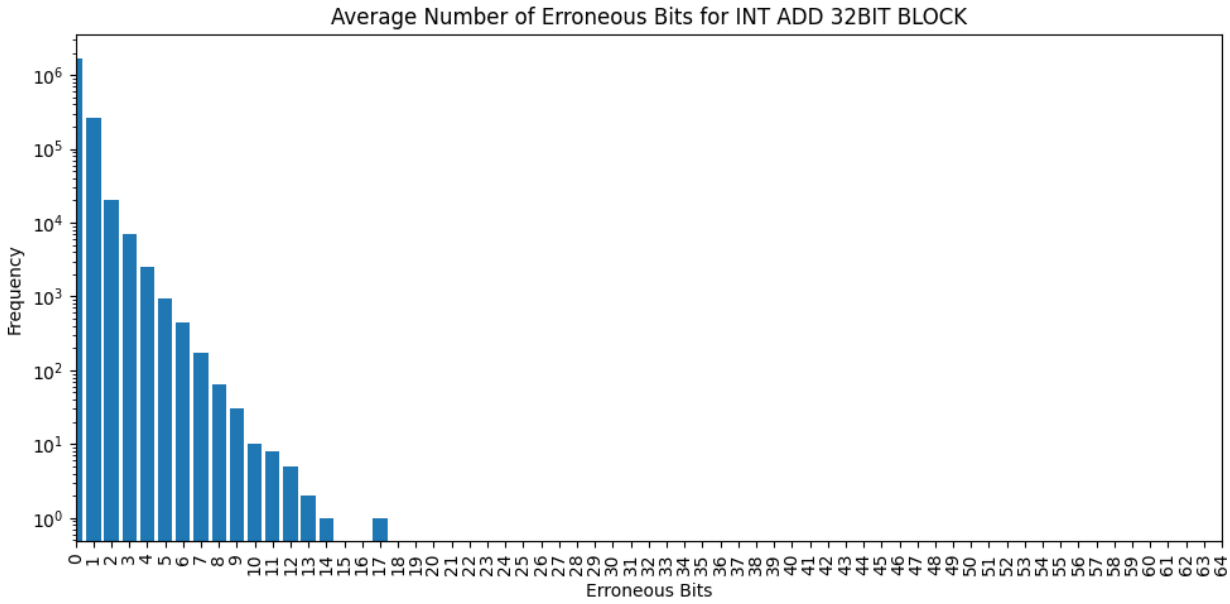


Figure 4.36: The average number of erroneous bits per operation for Adder Using 32 Bit adder blocks when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

4.8.3 Comments

The fault rate is reduced compared to the one of the adder which consisted of 4 Bit modules. From the distribution of the erroneous bits, it can be said that the high bits of the output of each adder block were the ones less affected. It is indeed, obvious, that the coverage rates are reduced. Again, that is linked to the more complex structure and the more possible data paths of the 32Bit-block-based adder compared to the 4Bit-block-based one.

4.9 Integer Adder (Single 64Bit adder block)

Input: Two 64 Bit Integer Numbers

Output: One 65 Bit Number (highest bit being the carry)

4.9.1 Bridging Fault Injection

Testing 1000 random numbers for each injection Fault Detection coverage: 82.7%

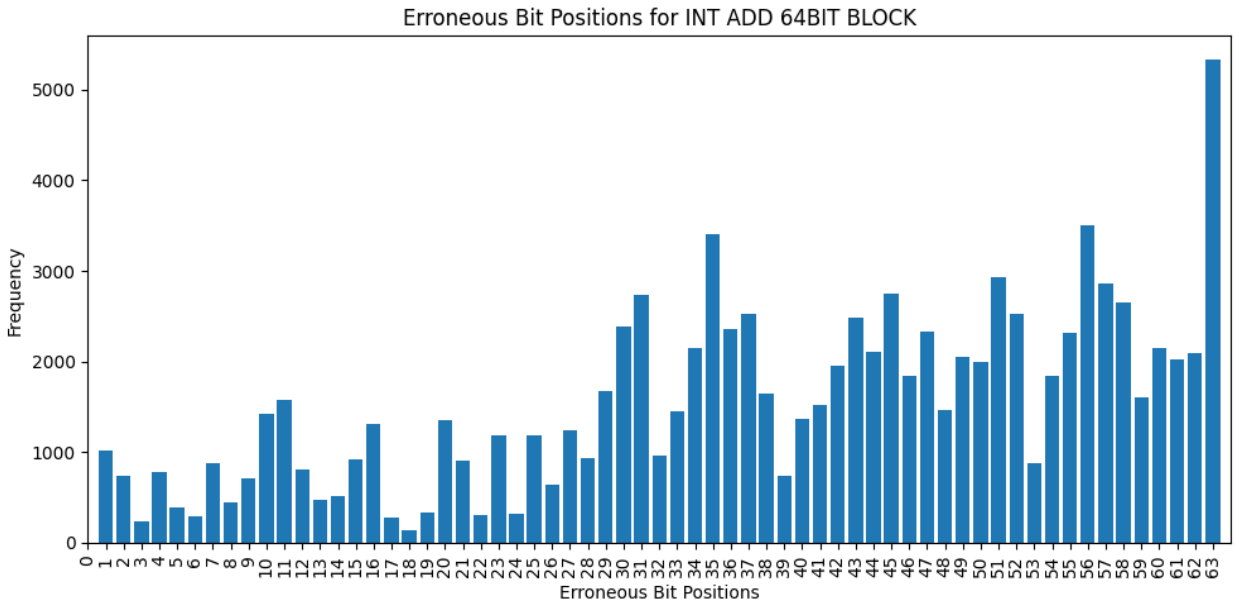


Figure 4.37: The distribution of erroneous bits on the output of Integer Adder Using a single 64Bit adder block when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

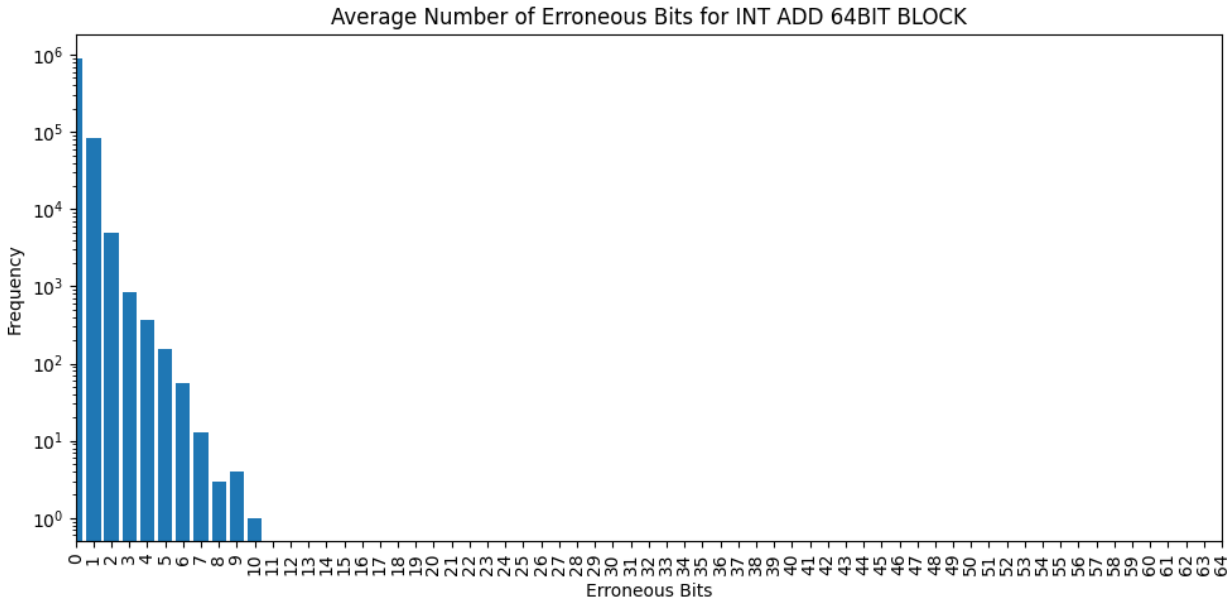


Figure 4.38: The average number of erroneous bits per operation for Integer Adder Using a single 64Bit adder block when 1000 different Bridging Faults were injected and 1000 random operations were done for each injection

4.9.2 Stuck-At Fault Injections

Testing 1000 random numbers for each injection Fault Detection coverage: 64.55%

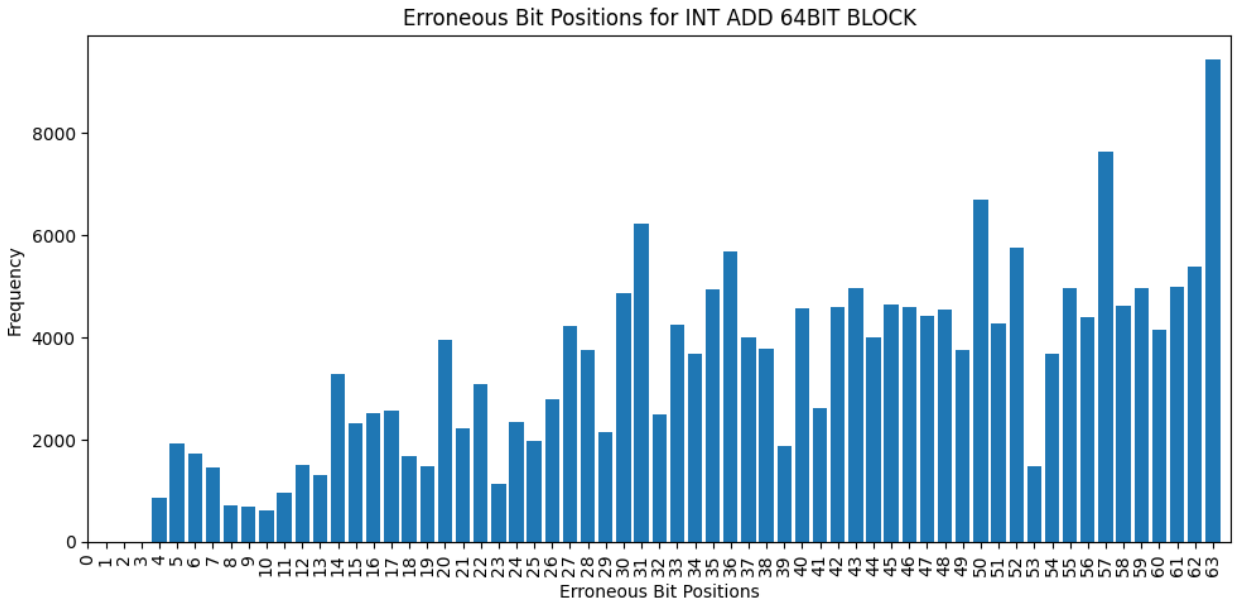


Figure 4.39: The distribution of erroneous bits on the output of Integer Adder Using a single 64Bit adder block when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

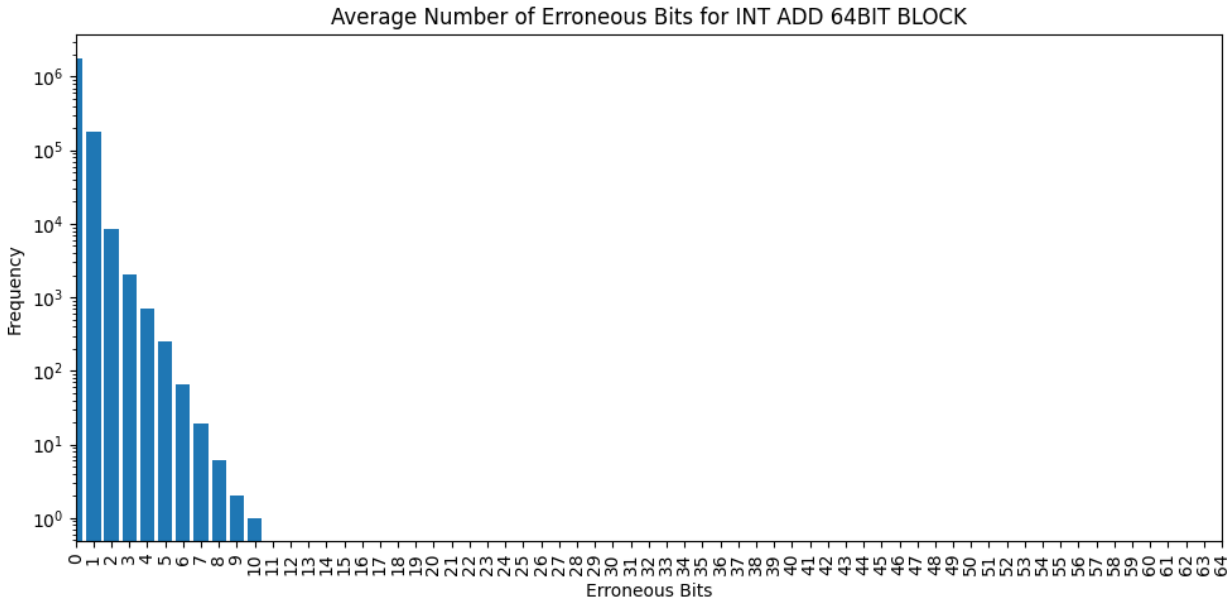


Figure 4.40: The average number of erroneous bits per operation for Integer Adder Using a single 64Bit adder block when 2000 different Stuck-At Faults were injected and 1000 random operations were done for each injection

4.9.3 Comments

The fault rate is, across the board, the lowest of all the integer adder modules tested. The lowest bits of the output are the ones mostly affected, which is to be expected due to not all of the numbers tested being large enough to use the logic of the high bits. As for the coverage, which is the lowest of all the other adder modules, the same conclusions mentioned on the 32Bit-block-based adder also apply here.

4.10 Bit Error Rates and Coverage Rates

On the following graphs, the Bit Error Rate and fault detection coverage for each module tested will be presented.

The Bit Error Rate, indicates the probability that any given output bit is erroneous.

The Fault Detection Coverage Rate shows the percentage of faulty hardware modules which were detected (i.e., generated at least one erroneous bit in at least one operation) by the test methodology used.

4.10.1 Bridging Fault Injection

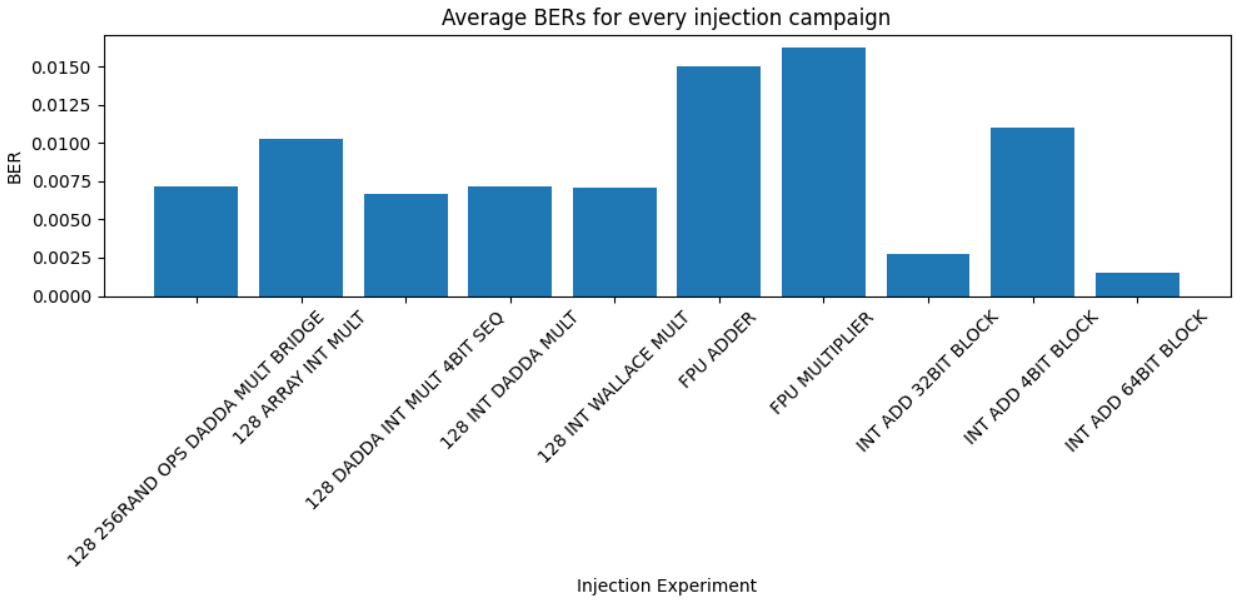


Figure 4.41: The Bit Error Rate for each module tested for the 1000 different Bridging Faults injected on each module

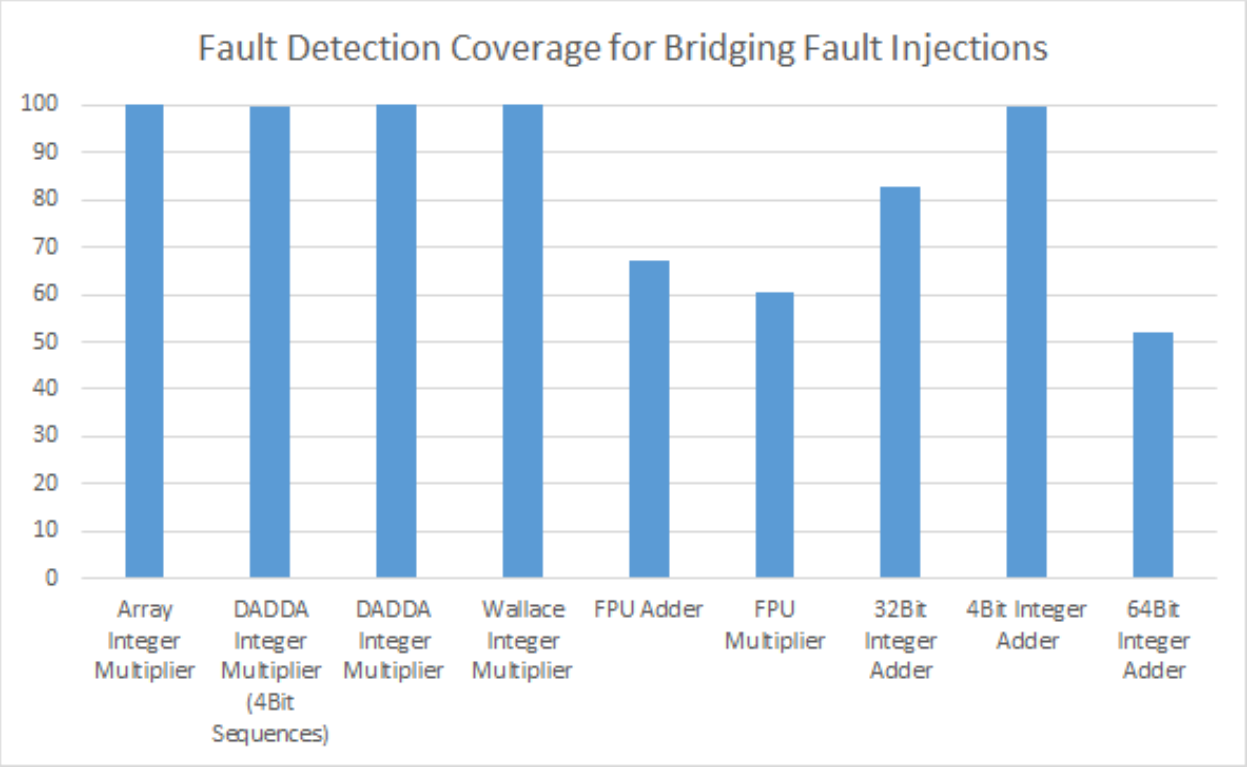


Figure 4.42: The coverage rate for each module tested for the 1000 different Bridging Faults injected on each module

4.10.2 Stuck-At Fault Injection

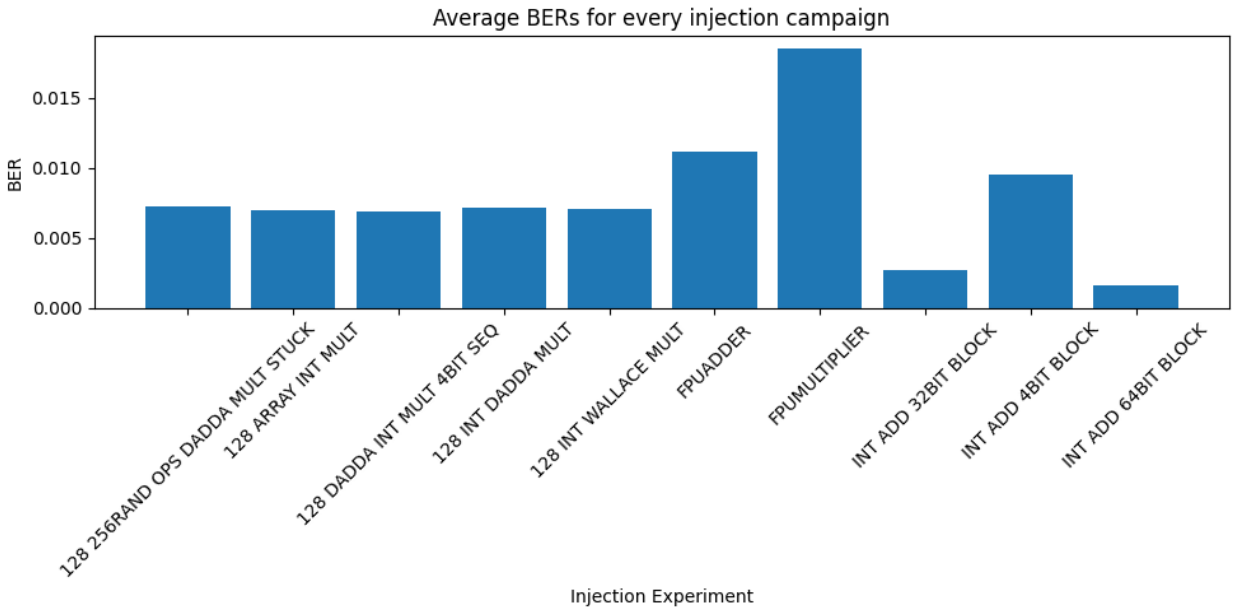


Figure 4.43: The Bit Error Rate for each module tested for the 2000 different Stuck-At Faults injected on each module

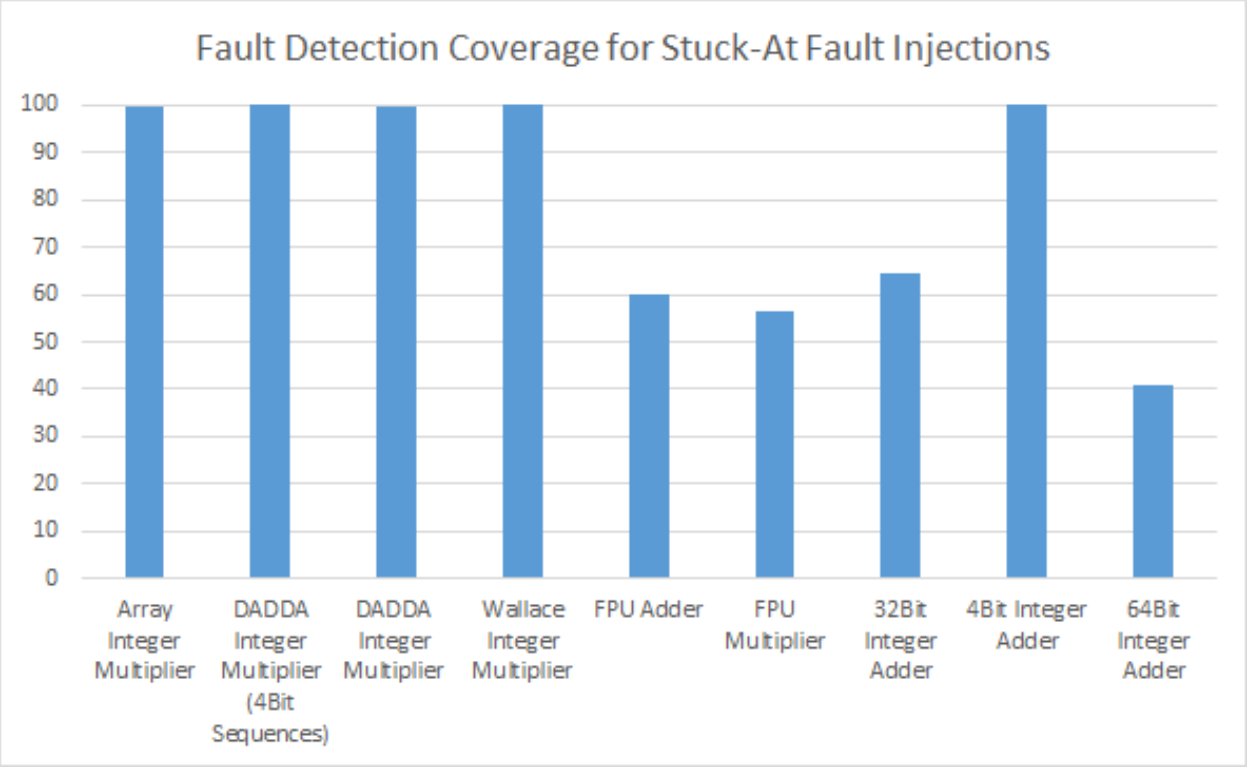


Figure 4.44: The average Bit Error Rate for each module tested for the 2000 different Stuck-At Faults injected on each module

4.10.3 Comments

For both Stuck-At and Bridging Fault injection tests, the most resilient to hardware faults modules were the Integer Adders using high-bit blocks (two 32Bit blocks or a single 64Bit block). The FPU and the Multiplier modules show across the board higher bit error rate. That is a bit expected, due to the tree-like structures these module use to calculate the results, causing an erroneous bit to affect more parts of the output. Especially FPU Modules either give a wrong result in which several bits are erroneous or give the correct result. As mentioned previously, this has to do with the several possible data paths that those modules contain.

When it comes to the fault detection coverage percentages, the multiplier modules and the adder module using 4Bit adders were the ones in which almost all possible faults were detected with the used testing methods. Those modules rely on fewer possible data paths (as someone can see by generating their netlist). Thus, a possible hardware fault is more likely to be in the way of the used data path for a tested input.

4.10.4 Fault Model Generation

By knowing the average number of erroneous bits per output (BER), a simple fault model can be generated. That fault model describes the probability of each individual output bit being erroneous, provided that the distribution of erroneous output bits is uniform.

Let Y be the output and $P(Y_i)$ be the probability of the bit i of the output being erroneous.

$P(Y_i) = BER$

However, as was noticeable in the graphs presented on this chapter, for most of the tested

units, the distribution of erroneous bits on the output was not uniform.

For this reason, a more accurate fault model can be generated by calculating the probability of each different bit of the output to be erroneous.

In order to achieve that, the total number of errors for each output bit of the module is divided by the number of tests done plus the number of operations done inside each test.

$P(Y_i) = \frac{totalerrors_i}{N_{tests}+N_{operations/test}}$

In this way, a more precise fault model is generated, as the behavior of each output bit is taken into consideration. That fault model is a vector, with each element of the vector indicating the probability of the bit of the index being erroneous.

Obviously, the graph of the fault model of each module has the same figure as the graph of the position of erroneous bits of that module, with the values having been divided by $N_{tests} + N_{(operations/test)}$ in order to express the probability.

The fault model of each module tested has been calculated and can be seen at the first annex.

5. CONCLUSION

5.1 Previous Work

As mentioned in the introduction, the topic of reliability analysis of computer chips is already a “hot” research area. Several methods for detecting such faults, both during production and during maintenance have been proposed [9], [11], [14], [17]. Also, more advanced error models, which take into consideration features of modern CPUs such as the dynamic voltage scaling, and test both integer and Floating Point Units have been proposed [15]. Similarly to the findings of this research, it was observed that the error model of each unit differs significantly. So, testing multiple modules in order to find personalized error models for each one results in significantly better accuracy.

5.2 Future work

As of now, the number of available error models for different hardware modules is still limited. This means that research on different hardware modules and testing scenarios will yield to more concrete and generalized observations. Also, due to the nature of certain lossy algorithms, knowing the error rate of a processor at specific power states allows us to run the processor at higher efficiency level, while tolerating some errors. The opposite also applies; for mission-critical applications, reducing the efficiency in order to achieve the highest possible reliability is required. Thus, testing more complex programs with the generated error models is required in order to see the conceivable impact of hardware errors in different programs. Research has already began in importing the error models of this research into gem-5 in order to observe the behavior in more complex programs.

5.2.1 Conclusion

As pointed out before, the topic of reliability analysis is growing in significance and newer fault detection and prevention methods are being proposed.

In the context of this thesis, the error rate of several commonly used arithmetic units was studied by injecting specific faults to those units and a fault model for each of those units was generated.

Consequently, with that fault model in hand, the impact of a possible hardware defect in the output of some programs can be predicted. In this fashion, that program can be simulated with that fault model into consideration in order to observe the real-life impact of possible hardware defects.

Furthermore, by taking into account the coverage rates of the testing methods presented, the decision of whether or not a specific testing method is effective in detecting possible hardware detects can be made.

ABBREVIATIONS - ACRONYMS

BER	Bit Error Rate
SDE	Silent Data Error
SDC	Silent Data Corruption
ALU	Arithmetic Logic Unit
FPU	Floating Point Unit

APPENDIX A. ERROR MODELS OF TESTED MODULES

The Error Models are presented in the form of a dictionary.

For example:

```
{4:0.2, 5: 0.1, 6: 0.4}
```

That means that for the bit 4 of the output, the probability of it being erroneous is 0,2, for the bit 5 of the output the probability of it being erroneous is 0.1 and so on. If that probability is zero, then the bit is omitted from the dictionary.

Error Models when modules were injected with Bridging Faults

- **FPU Adder**

  10: 0.017879, 11: 0.018118, 12: 0.01852, 13: 0.0174155, 14: 0.0172285, 15: 0.0169225, 16: 0.0158585, 17: 0.016185, 18: 0.015139, 19: 0.0157135, 20: 0.0116295, 21: 0.0139935, 33: 0.0172285, 8: 0.017759, 9: 0.018673, 6: 0.0184795, 5: 0.0176585, 50: 0.016527, 49: 0.0183235, 0: 0.0111285, 1: 0.014541, 2: 0.016655, 3: 0.0161465, 4: 0.0170705, 7: 0.018293, 22: 0.0162245, 23: 0.015165, 24: 0.0165825, 25: 0.017336, 26: 0.0180975, 27: 0.017117, 28: 0.017914, 29: 0.01847, 30: 0.0178855, 31: 0.017318, 32: 0.016389, 34: 0.017702, 35: 0.01848, 36: 0.016984, 37: 0.018209, 38: 0.017197, 39: 0.0193215, 40: 0.0168165, 41: 0.0174025, 42: 0.0169095, 43: 0.0179325, 44: 0.0186575, 45: 0.018008, 46: 0.0186815, 47: 0.0173205, 48: 0.019198, 51: 0.016495, 52: 0.0152055, 53: 0.014931, 54: 0.0099035, 55: 0.004805, 63: 0.0061895, 56: 0.0036955, 57: 0.0039975, 62: 0.0042825, 59: 0.000407, 61: 0.003314, 60: 0.0015815, 58: 0.0014965

- **FPU Multiplier**

  34: 0.0145555, 44: 0.011859, 0: 0.027161, 1: 0.027346, 2: 0.027307, 3: 0.0266865, 4: 0.027523, 5: 0.027344, 6: 0.0273285, 7: 0.027421, 8: 0.0246905, 42: 0.0126375, 53: 0.008092, 60: 0.003725, 62: 0.0079275, 26: 0.0159545, 27: 0.018693, 28: 0.0182375, 29: 0.0170675, 30: 0.0159035, 31: 0.0161405, 32: 0.013135, 33: 0.0135675, 35: 0.013406, 55: 0.007909, 45: 0.011634, 51: 0.0110525, 9: 0.025303, 10: 0.023934, 11: 0.0261765, 12: 0.026152, 13: 0.0261035, 14: 0.023609, 15: 0.0245645, 16: 0.0241805, 17: 0.0232995, 18: 0.022104, 19: 0.0215115, 20: 0.0204545, 21: 0.019954, 22: 0.019715, 23: 0.020562, 24: 0.0172095, 25: 0.017841, 36: 0.012641, 37: 0.013097, 38: 0.0125015, 39: 0.0135965, 40: 0.0119115, 41: 0.0112025, 43: 0.010489, 46: 0.010108, 47: 0.0115825, 48: 0.0099265, 49: 0.008387, 50: 0.0099125, 52: 0.007538, 59: 0.002548, 56: 0.006059, 54: 0.006308, 57: 0.008334, 58: 0.0083425, 63: 0.0043665, 61: 0.0015335

- **Integer Adder Using 4 Bit adder blocks**

  35: 0.014993, 36: 0.012035, 44: 0.005931, 11: 0.012172, 12: 0.009687, 13: 0.009536, 14: 0.012471, 15: 0.006916, 16: 0.009208, 17: 0.013368, 19: 0.011619, 18: 0.008341, 20: 0.011245, 21: 0.008359, 22: 0.008193, 23: 0.009966, 24: 0.007857, 25: 0.011385, 26: 0.009662, 27: 0.008864, 28: 0.00731, 29: 0.011609, 30: 0.012157, 31: 0.010652, 32: 0.009263, 7: 0.012288, 8: 0.015008, 9: 0.009139, 10: 0.009313, 47: 0.013856, 54: 0.012346, 61: 0.012858, 40: 0.012898, 41: 0.010485, 42: 0.011613, 43: 0.010711, 45: 0.011954, 46: 0.010392, 48: 0.008302, 49: 0.008869, 50: 0.015204, 51: 0.011276,

56: 0.014473, 57: 0.009779, 58: 0.011842, 59: 0.011628, 60: 0.013911, 62: 0.014845, 63: 0.018068, 33: 0.012546, 34: 0.014292, 53: 0.013596, 55: 0.013562, 37: 0.011601, 38: 0.013985, 39: 0.01066, 52: 0.01112, 6: 0.014766, 2: 0.008059, 3: 0.008066, 4: 0.009438, 5: 0.009163, 1: 0.006608, 0: 0.001887

- **Integer Adder Using 32 Bit adder blocks**

  55: 0.004095, 51: 0.003044, 52: 0.003576, 57: 0.00228, 58: 0.003553, 12: 0.002178, 18: 0.002759, 19: 0.004233, 20: 0.00329, 61: 0.005933, 39: 0.00244, 40: 0.001083, 41: 0.000599, 42: 0.002177, 43: 0.002357, 59: 0.003939, 60: 0.005548, 62: 0.004058, 63: 0.007248, 26: 0.005545, 28: 0.002725, 29: 0.001448, 9: 0.002711, 25: 0.004481, 10: 0.001988, 47: 0.001708, 49: 0.003685, 50: 0.002606, 27: 0.001871, 21: 0.004439, 22: 0.002263, 23: 0.003226, 56: 0.002682, 44: 0.00331, 45: 0.004504, 53: 0.003571, 32: 0.005838, 33: 0.003934, 34: 0.002521, 35: 0.002222, 36: 0.002332, 37: 0.00135, 38: 0.001011, 48: 0.004366, 16: 0.003288, 30: 0.003915, 31: 0.004706, 54: 0.00243, 24: 0.003368, 46: 0.003056, 13: 0.002036, 7: 0.000717, 17: 0.002398, 11: 0.001173, 15: 0.001687, 14: 0.001487, 3: 0.00083, 8: 0.001363, 5: 0.000838, 2: 0.000633, 1: 0.000611, 4: 0.000426, 6: 0.000993

- **Integer Adder Using a single 64 Bit adder block**

  55: 0.004095, 51: 0.003044, 52: 0.003576, 57: 0.00228, 58: 0.003553, 12: 0.002178, 18: 0.002759, 19: 0.004233, 20: 0.00329, 61: 0.005933, 39: 0.00244, 40: 0.001083, 41: 0.000599, 42: 0.002177, 43: 0.002357, 59: 0.003939, 60: 0.005548, 62: 0.004058, 63: 0.007248, 26: 0.005545, 28: 0.002725, 29: 0.001448, 9: 0.002711, 25: 0.004481, 10: 0.001988, 47: 0.001708, 49: 0.003685, 50: 0.002606, 27: 0.001871, 21: 0.004439, 22: 0.002263, 23: 0.003226, 56: 0.002682, 44: 0.00331, 45: 0.004504, 53: 0.003571, 32: 0.005838, 33: 0.003934, 34: 0.002521, 35: 0.002222, 36: 0.002332, 37: 0.00135, 38: 0.001011, 48: 0.004366, 16: 0.003288, 30: 0.003915, 31: 0.004706, 54: 0.00243, 24: 0.003368, 46: 0.003056, 13: 0.002036, 7: 0.000717, 17: 0.002398, 11: 0.001173, 15: 0.001687, 14: 0.001487, 3: 0.00083, 8: 0.001363, 5: 0.000838, 2: 0.000633, 1: 0.000611, 4: 0.000426, 6: 0.000993

- **DADDA Integer Multiplier (tests using 4Bit sequences discussed before)**

  60: 0.016546875, 61: 0.0148359375, 62: 0.0178515625, 63: 0.0173203125, 64: 0.01541015625, 65: 0.012484375, 66: 0.00910546875, 67: 0.00662109375, 68: 0.0076875, 69: 0.00969921875, 70: 0.00974609375, 71: 0.011125, 75: 0.01081640625, 76: 0.0115625, 77: 0.01201171875, 78: 0.0148125, 115: 0.00373828125, 48: 0.00848046875, 49: 0.01023828125, 50: 0.01070703125, 51: 0.0121640625, 52: 0.0100703125, 53: 0.01196484375, 72: 0.01208203125, 105: 0.004109375, 106: 0.0016171875, 107: 0.00344140625, 108: 0.0027890625, 109: 0.0036875, 110: 0.0041484375, 92: 0.0090390625, 93: 0.00799609375, 94: 0.0088984375, 95: 0.007703125, 96: 0.00770703125, 97: 0.00666015625, 86: 0.009640625, 87: 0.01107421875, 88: 0.00821875, 89: 0.00607421875, 90: 0.00598046875, 91: 0.0069609375, 54: 0.010890625, 55: 0.01456640625, 79: 0.0088125, 80: 0.00711328125, 81: 0.01255859375, 82: 0.0118515625, 83: 0.00701953125, 122: 0.00093359375, 123: 0.0006640625, 124: 0.00110546875, 125: 0.00165625, 126: 0.00032421875, 127: 0.00019921875, 84: 0.0090546875, 85: 0.00814453125, 103: 0.00584375, 104: 0.003859375, 10: 0.00355859375, 11: 0.0039765625, 12: 0.00236328125, 13: 0.0022890625, 14: 0.0034765625, 15: 0.00418359375, 16: 0.00359765625, 17: 0.00304296875, 18: 0.00245703125, 19: 0.00178125, 20: 0.00498046875, 21: 0.004390625, 22: 0.00418359375, 23: 0.003828125, 24: 0.00365625, 25:

0.0045625, 26: 0.00514453125, 27: 0.00527734375, 28: 0.0070625, 29: 0.0061484375, 30: 0.00675, 31: 0.0056328125, 32: 0.0060625, 33: 0.006609375, 34: 0.0102578125, 35: 0.007828125, 36: 0.00713671875, 37: 0.0062265625, 38: 0.0063359375, 39: 0.00767578125, 40: 0.00683984375, 41: 0.00836328125, 42: 0.00551171875, 43: 0.00520703125, 44: 0.0065859375, 45: 0.0085234375, 46: 0.01085546875, 47: 0.00906640625, 56: 0.01196484375, 57: 0.01244140625, 58: 0.01291796875, 59: 0.0181328125, 101: 0.00757421875, 102: 0.005828125, 111: 0.004828125, 112: 0.00487890625, 113: 0.00415234375, 114: 0.002640625, 116: 0.0028046875, 117: 0.002484375, 98: 0.00636328125, 99: 0.00646875, 100: 0.00456640625, 73: 0.0117421875, 74: 0.01225390625, 119: 0.00146484375, 120: 0.00126171875, 121: 0.000671875, 118: 0.0017734375, 9: 0.00362109375, 8: 0.001625, 5: 0.001328125, 6: 0.00080859375, 7: 0.0007265625, 3: 0.0012109375, 4: 0.0005703125, 1: 0.000203125, 2: 0.00034375

- **DADDA Integer Multiplier**

  64: 0.012214, 65: 0.015732, 66: 0.018405, 67: 0.014794, 68: 0.015448, 69: 0.01522, 70: 0.015075, 71: 0.018386, 72: 0.01539, 10: 0.002624, 11: 0.002334, 12: 0.003543, 13: 0.005822, 14: 0.003714, 15: 0.004341, 16: 0.004479, 17: 0.004399, 18: 0.002668, 19: 0.003849, 20: 0.003043, 21: 0.004102, 117: 0.002653, 118: 0.003693, 119: 0.002838, 120: 0.001713, 121: 0.001522, 122: 0.002234, 123: 0.001317, 124: 0.002752, 125: 0.00158, 126: 0.000624, 41: 0.012777, 42: 0.00979, 43: 0.00822, 44: 0.008253, 45: 0.010132, 46: 0.012147, 47: 0.008939, 48: 0.011761, 49: 0.009255, 81: 0.012746, 82: 0.014242, 83: 0.011147, 84: 0.009583, 85: 0.009078, 86: 0.007483, 87: 0.009215, 88: 0.009133, 89: 0.011998, 90: 0.010896, 91: 0.008217, 92: 0.007856, 93: 0.007379, 94: 0.007256, 74: 0.01435, 75: 0.012182, 76: 0.012634, 77: 0.009843, 78: 0.007806, 79: 0.007409, 80: 0.008809, 22: 0.003796, 55: 0.012211, 56: 0.009388, 57: 0.009333, 58: 0.010759, 59: 0.013811, 60: 0.013209, 61: 0.011838, 62: 0.013862, 63: 0.012006, 50: 0.008489, 51: 0.01026, 52: 0.010834, 53: 0.012298, 54: 0.013004, 109: 0.006759, 110: 0.006638, 111: 0.005869, 112: 0.004315, 113: 0.003789, 114: 0.003683, 115: 0.002638, 116: 0.001916, 107: 0.007465, 108: 0.006957, 95: 0.008576, 96: 0.007692, 97: 0.007401, 98: 0.009146, 99: 0.0069, 100: 0.005405, 73: 0.011895, 23: 0.004305, 24: 0.004523, 25: 0.004317, 28: 0.006215, 29: 0.004653, 30: 0.006478, 31: 0.006878, 32: 0.005243, 33: 0.0048, 34: 0.002737, 35: 0.002802, 36: 0.003002, 37: 0.005042, 38: 0.008302, 39: 0.00733, 40: 0.007876, 101: 0.00685, 102: 0.006794, 103: 0.006379, 104: 0.005117, 105: 0.004077, 106: 0.004617, 26: 0.002643, 27: 0.004324, 9: 0.001932, 127: 0.000174, 7: 0.002, 8: 0.001252, 6: 0.001519, 5: 0.000952

- **Integer Array Multiplier**

  49: 0.015698, 50: 0.012588, 51: 0.014216, 52: 0.018755, 53: 0.018486, 54: 0.014001, 55: 0.013478, 56: 0.012411, 57: 0.011872, 58: 0.015116, 114: 0.007824, 115: 0.007287, 116: 0.005812, 117: 0.005607, 118: 0.005214, 119: 0.005095, 120: 0.004087, 121: 0.00353, 122: 0.003249, 69: 0.0152, 70: 0.014554, 71: 0.015041, 72: 0.017656, 73: 0.015601, 74: 0.017281, 75: 0.015371, 76: 0.015546, 77: 0.013026, 78: 0.016805, 79: 0.01734, 80: 0.014569, 81: 0.014254, 82: 0.010345, 83: 0.01157, 84: 0.012674, 18: 0.008772, 19: 0.005969, 20: 0.006345, 21: 0.006199, 22: 0.007384, 23: 0.005949, 24: 0.008287, 25: 0.008225, 26: 0.008256, 27: 0.007831, 111: 0.007519, 112: 0.007582, 113: 0.007081, 85: 0.013964, 86: 0.011171, 87: 0.012631, 88: 0.015535, 89: 0.013505, 67: 0.015916, 68: 0.014146, 44: 0.01295, 45: 0.013533, 46: 0.014482, 47: 0.014931, 48: 0.016678, 90: 0.009889, 91: 0.008726, 92: 0.010257, 110: 0.007854, 0: 0.003056, 1: 0.003082, 2: 0.003532,

3: 0.00341, 4: 0.004263, 5: 0.004068, 6: 0.004881, 7: 0.00397, 8: 0.004562, 9: 0.004682, 10: 0.005042, 11: 0.004104, 12: 0.004087, 13: 0.005392, 14: 0.008202, 15: 0.007392, 16: 0.006695, 17: 0.00616, 28: 0.007401, 29: 0.00836, 30: 0.007032, 31: 0.006213, 32: 0.010335, 33: 0.009572, 34: 0.009133, 35: 0.010257, 36: 0.011709, 37: 0.012934, 38: 0.013056, 39: 0.012354, 40: 0.01036, 41: 0.012195, 42: 0.016395, 43: 0.012014, 59: 0.015804, 60: 0.015378, 61: 0.015894, 62: 0.013919, 63: 0.013481, 64: 0.013884, 65: 0.012905, 66: 0.016193, 93: 0.009189, 94: 0.008726, 95: 0.010339, 96: 0.009473, 97: 0.009214, 98: 0.008773, 99: 0.009571, 100: 0.008737, 101: 0.009644, 102: 0.008052, 103: 0.007569, 104: 0.011181, 105: 0.010667, 106: 0.008829, 107: 0.00768, 108: 0.006882, 109: 0.007213, 123: 0.004292, 124: 0.003721, 125: 0.002727, 126: 0.00186, 127: 0.000947

- **Integer Wallace Multiplier**

38: 0.006093, 39: 0.007012, 40: 0.006999, 41: 0.006274, 42: 0.009297, 43: 0.010258, 44: 0.009024, 45: 0.008926, 76: 0.013121, 77: 0.013618, 78: 0.016629, 79: 0.014306, 80: 0.009431, 81: 0.010126, 82: 0.009361, 83: 0.008158, 84: 0.011238, 85: 0.012508, 86: 0.011508, 18: 0.004559, 19: 0.004442, 20: 0.00271, 21: 0.003175, 22: 0.005058, 23: 0.00497, 24: 0.007251, 25: 0.00466, 26: 0.006974, 27: 0.005084, 28: 0.006118, 29: 0.007122, 91: 0.008665, 92: 0.010517, 93: 0.011453, 94: 0.008522, 95: 0.00866, 96: 0.007734, 97: 0.00791, 98: 0.007442, 99: 0.006283, 73: 0.010255, 74: 0.009805, 75: 0.013341, 71: 0.015204, 72: 0.01213, 55: 0.008996, 56: 0.011862, 57: 0.011459, 58: 0.013429, 59: 0.013137, 60: 0.014774, 61: 0.015941, 62: 0.014216, 63: 0.01602, 64: 0.016242, 65: 0.015093, 66: 0.015441, 87: 0.010811, 88: 0.010763, 89: 0.007665, 90: 0.009134, 46: 0.007552, 47: 0.008065, 48: 0.008821, 49: 0.009976, 50: 0.008067, 51: 0.009203, 52: 0.010732, 53: 0.009323, 100: 0.005987, 101: 0.005924, 102: 0.005376, 103: 0.004786, 67: 0.013737, 68: 0.010279, 69: 0.013855, 70: 0.017617, 30: 0.00847, 31: 0.008376, 32: 0.006293, 33: 0.005977, 34: 0.005344, 35: 0.00527, 36: 0.00687, 54: 0.009903, 104: 0.002974, 105: 0.006156, 106: 0.005544, 107: 0.004766, 108: 0.003019, 109: 0.004156, 110: 0.003331, 37: 0.006446, 16: 0.002563, 17: 0.003581, 111: 0.003006, 112: 0.003915, 113: 0.004422, 114: 0.005634, 115: 0.006096, 116: 0.004602, 117: 0.004562, 118: 0.002301, 119: 0.00203, 120: 0.001059, 121: 0.000524, 122: 0.000575, 123: 0.000778, 124: 0.001417, 125: 0.000579, 7: 0.002101, 8: 0.001822, 9: 0.001243, 10: 0.001807, 11: 0.000919, 12: 0.001398, 13: 0.002086, 14: 0.002272, 15: 0.001361, 126: 0.000215, 127: 6.1e-05, 5: 0.00116, 6: 0.000527, 4: 0.000454

Error Models when modules were injected with Stuck-At Faults

- **FPU Adder**

21: 0.0103005, 22: 0.01198175, 23: 0.0124355, 24: 0.01293675, 25: 0.013257, 26: 0.0136965, 27: 0.0131955, 28: 0.0130985, 29: 0.01371975, 30: 0.01327925, 31: 0.012995, 32: 0.0128205, 33: 0.012616, 34: 0.013057, 35: 0.01139525, 36: 0.01241675, 37: 0.012915, 16: 0.01500525, 17: 0.0131035, 18: 0.013231, 19: 0.012403, 20: 0.01171475, 6: 0.01342525, 8: 0.01330575, 9: 0.0133175, 10: 0.01426525, 11: 0.0127175, 12: 0.0120315, 13: 0.0129965, 14: 0.013842, 15: 0.0145845, 53: 0.009885, 38: 0.012206, 58: 0.00274225, 0: 0.00730525, 1: 0.008947, 2: 0.01016225, 3: 0.01376875, 4: 0.013077, 7: 0.013406, 39: 0.01147975, 40: 0.012067, 41: 0.01127725, 42: 0.011739, 44: 0.0122345, 45: 0.01238725, 46: 0.0120275, 47: 0.01274375, 48: 0.0127665, 49: 0.011719, 50: 0.0121865, 51:

0.0102555, 52: 0.0100475, 5: 0.014447, 43: 0.01218125, 54: 0.00827875, 55: 0.0056005, 63: 0.00436775, 56: 0.005112, 61: 0.0025, 60: 0.00219325, 62: 0.00457925, 57: 0.005916, 59: 0.0027985

- **FPU Multiplier**

  57: 0.005628, 6: 0.01477975, 16: 0.0126565, 0: 0.01771725, 1: 0.01740975, 2: 0.0168135, 3: 0.01630875, 4: 0.016301, 5: 0.0153035, 7: 0.01461125, 8: 0.014739, 9: 0.0145265, 10: 0.01405525, 11: 0.01426575, 12: 0.01258275, 13: 0.012466, 14: 0.01245725, 15: 0.0132325, 17: 0.012004, 18: 0.012234, 19: 0.01204225, 20: 0.011909, 21: 0.0119635, 22: 0.01116725, 23: 0.010141, 24: 0.0095645, 25: 0.01039025, 26: 0.01019, 27: 0.010358, 53: 0.00445525, 54: 0.00334625, 55: 0.005374, 52: 0.0043415, 35: 0.009024, 28: 0.01066725, 29: 0.0095155, 30: 0.0089345, 31: 0.0093755, 32: 0.00808275, 58: 0.00537725, 39: 0.00791525, 33: 0.00852625, 34: 0.00842475, 36: 0.00829, 37: 0.007746, 38: 0.00764125, 40: 0.006826, 41: 0.00716225, 42: 0.006773, 43: 0.00682475, 44: 0.0066665, 45: 0.005793, 46: 0.004941, 47: 0.00534475, 48: 0.004621, 49: 0.00477975, 50: 0.00418325, 51: 0.00412175, 63: 0.00315325, 56: 0.005134, 59: 0.0020025, 62: 0.0053705, 60: 0.00125, 61: 0.0005

- **Integer Adder Using 4 Bit adder blocks**

  51: 0.009832, 15: 0.007788, 16: 0.0089545, 17: 0.014352, 18: 0.009395, 19: 0.008795, 20: 0.0127615, 21: 0.0086525, 22: 0.008876, 23: 0.0083265, 24: 0.0077495, 8: 0.012397, 9: 0.0079705, 10: 0.0061825, 11: 0.009003, 12: 0.011889, 13: 0.0085, 14: 0.009496, 46: 0.010173, 45: 0.013614, 47: 0.009605, 25: 0.0102115, 26: 0.010198, 27: 0.0064925, 28: 0.0106345, 29: 0.0105465, 30: 0.012972, 31: 0.0095945, 32: 0.008911, 33: 0.012717, 34: 0.008738, 55: 0.007461, 56: 0.012455, 57: 0.0101605, 58: 0.0102325, 59: 0.00961, 60: 0.0147455, 61: 0.00994, 62: 0.013337, 63: 0.013045, 37: 0.009831, 38: 0.010267, 39: 0.007915, 40: 0.0114545, 41: 0.0117115, 42: 0.010728, 43: 0.013397, 44: 0.009193, 48: 0.005892, 49: 0.005745, 50: 0.010313, 52: 0.007449, 53: 0.0091355, 54: 0.0073725, 2: 0.0056305, 3: 0.0069135, 4: 0.0077505, 5: 0.009701, 6: 0.0124295, 7: 0.011468, 35: 0.0072265, 36: 0.008753, 1: 0.0036215

- **Integer Adder Using 32 Bit adder blocks**

  52: 0.001941, 25: 0.0035355, 27: 0.002956, 8: 0.001417, 9: 0.001279, 10: 0.001804, 11: 0.0020755, 12: 0.00142, 18: 0.002446, 51: 0.0037335, 29: 0.0033135, 42: 0.0037455, 43: 0.002803, 14: 0.00164, 15: 0.0015445, 16: 0.0033865, 17: 0.0031875, 19: 0.0043575, 20: 0.001607, 21: 0.0044095, 22: 0.003026, 58: 0.0030385, 63: 0.005339, 23: 0.002802, 24: 0.002733, 31: 0.003238, 35: 0.0018995, 36: 0.001991, 37: 0.001931, 38: 0.0011955, 39: 0.002823, 40: 0.001853, 41: 0.0015685, 53: 0.0020015, 47: 0.001984, 62: 0.004689, 55: 0.0059925, 26: 0.002475, 61: 0.003956, 59: 0.0034625, 60: 0.0053265, 49: 0.0033285, 50: 0.003419, 44: 0.003595, 45: 0.0018, 30: 0.004703, 46: 0.002375, 48: 0.00421, 54: 0.002966, 57: 0.002554, 28: 0.0037825, 6: 0.001659, 7: 0.0018, 56: 0.004224, 3: 0.001398, 13: 0.0025095, 32: 0.003861, 33: 0.0022285, 4: 0.0010085, 5: 0.0006255, 1: 0.000365, 34: 0.0013695, 2: 0.00123

- **Integer Adder Using a single 64 Bit adder block**

  63: 0.0047175, 23: 0.000566, 30: 0.0024385, 34: 0.0018465, 40: 0.0022825, 28: 0.0018795, 29: 0.00107, 31: 0.0031195, 32: 0.001248, 35: 0.0024745, 36:

0.002844, 26: 0.0014005, 10: 0.0003125, 11: 0.0004775, 12: 0.0007565, 62: 0.002694, 5: 0.0009695, 48: 0.0022715, 49: 0.001874, 52: 0.002877, 53: 0.0007475, 44: 0.0020055, 19: 0.0007355, 20: 0.0019835, 27: 0.00212, 57: 0.003818, 37: 0.0020025, 38: 0.0018915, 43: 0.002486, 55: 0.0024865, 51: 0.0021375, 54: 0.001843, 56: 0.0022, 58: 0.0023145, 59: 0.0024825, 41: 0.001316, 16: 0.001262, 47: 0.002209, 50: 0.003347, 33: 0.002123, 22: 0.0015475, 25: 0.0009875, 42: 0.0022965, 46: 0.002303, 17: 0.00128, 13: 0.0006535, 18: 0.0008425, 61: 0.0024935, 39: 0.0009435, 15: 0.001164, 60: 0.0020715, 14: 0.00164, 21: 0.0011085, 45: 0.002325, 4: 0.000435, 7: 0.0007255, 8: 0.000362, 9: 0.0003405, 24: 0.0011695, 6: 0.000864

- **DADDA Integer Multiplier (tests using 4Bit sequences, discussed before)**

  49: 0.009712890625, 50: 0.0133984375, 51: 0.010267578125, 52: 0.0108203125, 53: 0.00948046875, 54: 0.009056640625, 55: 0.009970703125, 97: 0.00576953125, 98: 0.004896484375, 99: 0.005697265625, 100: 0.004005859375, 56: 0.011623046875, 57: 0.01065625, 58: 0.01046875, 108: 0.00380078125, 109: 0.004794921875, 110: 0.0051328125, 111: 0.0031171875, 112: 0.00290625, 65: 0.013732421875, 66: 0.013130859375, 67: 0.008375, 68: 0.010681640625, 69: 0.007408203125, 70: 0.010970703125, 71: 0.012412109375, 72: 0.013369140625, 73: 0.01373046875, 60: 0.016048828125, 61: 0.01619140625, 62: 0.01809375, 63: 0.017076171875, 64: 0.01447265625, 46: 0.0118671875, 47: 0.01189453125, 48: 0.0100390625, 30: 0.00915234375, 31: 0.009126953125, 32: 0.008412109375, 33: 0.00739453125, 23: 0.004833984375, 24: 0.005376953125, 25: 0.00538671875, 26: 0.005724609375, 27: 0.0045859375, 28: 0.006654296875, 29: 0.006759765625, 113: 0.003064453125, 114: 0.0025859375, 115: 0.002625, 116: 0.0022734375, 59: 0.0138828125, 74: 0.01258984375, 75: 0.010560546875, 17: 0.002048828125, 18: 0.00332421875, 19: 0.00317578125, 20: 0.003763671875, 21: 0.005763671875, 22: 0.005767578125, 38: 0.010201171875, 39: 0.010517578125, 40: 0.010103515625, 41: 0.0103125, 42: 0.01062109375, 43: 0.010927734375, 44: 0.010025390625, 95: 0.006904296875, 96: 0.006541015625, 76: 0.0111875, 77: 0.010501953125, 78: 0.00841796875, 93: 0.007654296875, 94: 0.008408203125, 80: 0.009705078125, 81: 0.012185546875, 82: 0.010107421875, 83: 0.00976171875, 84: 0.009521484375, 85: 0.009841796875, 86: 0.00928515625, 87: 0.01043359375, 88: 0.007626953125, 89: 0.00700390625, 92: 0.007271484375, 36: 0.008626953125, 37: 0.007625, 34: 0.008376953125, 35: 0.01042578125, 104: 0.004986328125, 105: 0.004615234375, 106: 0.003962890625, 107: 0.0031875, 79: 0.010236328125, 119: 0.00175, 120: 0.00215234375, 121: 0.001912109375, 122: 0.00105859375, 123: 0.000921875, 124: 0.0009140625, 125: 0.00078125, 126: 0.000732421875, 45: 0.0103046875, 8: 0.00174609375, 9: 0.00189453125, 10: 0.002626953125, 11: 0.001205078125, 12: 0.00155078125, 13: 0.003126953125, 14: 0.00253515625, 15: 0.002501953125, 16: 0.00158984375, 101: 0.00347265625, 102: 0.003814453125, 103: 0.004748046875, 90: 0.00621875, 91: 0.006353515625, 117: 0.00272265625, 7: 0.00087109375, 5: 0.001080078125, 6: 0.0006171875, 118: 0.00197265625, 127: 0.0002265625, 3: 0.000453125, 4: 0.0002109375

- **DADDA Integer Multiplier**

  45: 0.0071935, 46: 0.0096825, 47: 0.0114545, 48: 0.009396, 49: 0.009682, 50: 0.0103095, 51: 0.0100975, 52: 0.0125775, 53: 0.010141, 54: 0.0126985, 55: 0.01388, 87: 0.010216, 88: 0.007279, 89: 0.006251, 90: 0.0094225, 91: 0.010616, 92: 0.009643, 93: 0.0088365, 94: 0.007038, 95: 0.009349, 96: 0.0081395, 97:

0.0082855, 98: 0.0067945, 99: 0.006137, 100: 0.0070085, 56: 0.0153135, 57: 0.0119315, 58: 0.012296, 59: 0.0132525, 60: 0.0138695, 61: 0.0158335, 62: 0.015411, 63: 0.0181425, 64: 0.0168825, 28: 0.0069245, 29: 0.0075445, 30: 0.0071895, 31: 0.00729, 32: 0.0064675, 33: 0.0058995, 34: 0.005368, 35: 0.0059735, 36: 0.005725, 65: 0.0156275, 66: 0.0116465, 67: 0.0108075, 68: 0.011754, 69: 0.0121505, 70: 0.0143705, 71: 0.01273, 44: 0.0092385, 101: 0.006684, 102: 0.005591, 103: 0.004269, 104: 0.003491, 105: 0.0045, 106: 0.005228, 107: 0.0051255, 108: 0.0043, 109: 0.004524, 110: 0.004662, 111: 0.0055935, 85: 0.011858, 86: 0.0119335, 73: 0.013001, 74: 0.012705, 75: 0.014066, 76: 0.0119835, 77: 0.0093655, 78: 0.0110225, 79: 0.0099495, 80: 0.010454, 81: 0.0129425, 82: 0.011765, 83: 0.0103195, 72: 0.0132275, 42: 0.009222, 43: 0.008317, 15: 0.0037555, 16: 0.0039405, 17: 0.00394, 18: 0.004588, 19: 0.003993, 20: 0.0037885, 21: 0.002758, 22: 0.003233, 23: 0.0048875, 112: 0.003926, 113: 0.00407, 38: 0.00693, 39: 0.007083, 40: 0.0089715, 41: 0.0106825, 114: 0.002537, 115: 0.0026895, 116: 0.0027625, 117: 0.0032915, 118: 0.00246, 119: 0.0029365, 120: 0.002058, 121: 0.0013935, 122: 0.0009885, 123: 0.000914, 27: 0.005808, 37: 0.0069085, 124: 0.0011505, 125: 0.000447, 126: 0.000627, 127: 0.000164, 84: 0.011046, 26: 0.0061395, 24: 0.0068585, 25: 0.00493, 13: 0.0020325, 14: 0.003786, 11: 0.001878, 12: 0.001584, 6: 0.001155, 7: 0.0014005, 8: 0.001213, 9: 0.002143, 10: 0.0017865, 3: 0.001036, 4: 0.0006925, 5: 0.000744, 2: 0.0007415

- **Integer Array Multiplier**

  48: 0.0114295, 49: 0.011684, 50: 0.0118135, 51: 0.011058, 52: 0.009652, 53: 0.009407, 54: 0.0088985, 55: 0.0107805, 64: 0.01129, 65: 0.013639, 66: 0.01174, 67: 0.0141205, 68: 0.0126465, 69: 0.0138175, 70: 0.016032, 71: 0.014641, 72: 0.0117415, 73: 0.012527, 74: 0.0125715, 75: 0.0115685, 76: 0.0104315, 77: 0.0113555, 59: 0.0109345, 60: 0.011122, 61: 0.0114485, 62: 0.0100415, 63: 0.012318, 41: 0.0099705, 42: 0.010165, 43: 0.0095035, 44: 0.008699, 45: 0.010718, 46: 0.0119825, 47: 0.0113755, 30: 0.0073085, 31: 0.0065485, 32: 0.0061295, 33: 0.005421, 34: 0.007523, 35: 0.008423, 36: 0.0077115, 37: 0.010089, 38: 0.008402, 39: 0.0097135, 101: 0.0068875, 102: 0.007909, 103: 0.0064345, 104: 0.006744, 105: 0.006957, 106: 0.0068405, 107: 0.004486, 108: 0.0038105, 109: 0.0041525, 110: 0.0036155, 111: 0.003782, 78: 0.010715, 79: 0.012913, 80: 0.013081, 81: 0.013495, 82: 0.011557, 83: 0.0096565, 84: 0.0089355, 85: 0.008776, 86: 0.007584, 87: 0.008099, 88: 0.0098625, 89: 0.0109145, 90: 0.0098465, 91: 0.008892, 92: 0.008499, 93: 0.0064445, 94: 0.0068485, 95: 0.004663, 96: 0.0062955, 57: 0.012931, 58: 0.012072, 40: 0.00876, 97: 0.007276, 98: 0.007063, 99: 0.0058075, 112: 0.003465, 113: 0.003273, 114: 0.0029205, 115: 0.002869, 116: 0.0025695, 117: 0.002785, 118: 0.0029125, 100: 0.006179, 119: 0.001658, 56: 0.012133, 28: 0.0042435, 29: 0.0062045, 10: 0.0024745, 11: 0.0016035, 12: 0.002203, 13: 0.0016165, 14: 0.001713, 15: 0.002447, 16: 0.0016255, 17: 0.0020695, 18: 0.0044765, 120: 0.0016775, 121: 0.001893, 8: 0.001595, 9: 0.001668, 19: 0.004057, 20: 0.003834, 122: 0.001064, 123: 0.0008385, 21: 0.004461, 22: 0.0048465, 23: 0.005425, 24: 0.0064765, 25: 0.0044085, 26: 0.005707, 27: 0.0050585, 124: 0.0014015, 125: 0.0005495, 126: 0.0002755, 127: 8.05e-05, 4: 0.0005145, 5: 0.000442, 6: 0.000152, 7: 0.0003085, 2: 0.000227, 3: 0.00045

- **Integer Wallace Multiplier**

  77: 0.0100055, 78: 0.011183, 79: 0.013796, 80: 0.010809, 81: 0.0094265, 82: 0.0085485, 83: 0.0076955, 32: 0.0070195, 33: 0.007235, 34: 0.006799, 35: 0.008488,

36: 0.0092675, 37: 0.008447, 38: 0.0082195, 39: 0.007448, 40: 0.0088375, 119: 0.0024805, 120: 0.002246, 121: 0.000982, 122: 0.000917, 123: 0.0004755, 124: 0.000354, 125: 0.000168, 126: 6.95e-05, 50: 0.0080785, 51: 0.008403, 52: 0.010503, 53: 0.0121945, 54: 0.010445, 55: 0.012497, 56: 0.015209, 57: 0.0143935, 58: 0.0123595, 59: 0.0137225, 99: 0.0068915, 100: 0.006965, 101: 0.0054955, 102: 0.0039425, 103: 0.0027295, 104: 0.003706, 105: 0.0045115, 106: 0.0057485, 107: 0.0048015, 85: 0.0088285, 86: 0.009732, 87: 0.007844, 88: 0.008454, 89: 0.008549, 90: 0.008518, 91: 0.0128805, 92: 0.009777, 93: 0.0086825, 94: 0.010138, 95: 0.0102165, 96: 0.007394, 97: 0.006157, 98: 0.006111, 108: 0.005243, 109: 0.005099, 41: 0.008162, 42: 0.007531, 43: 0.0099525, 44: 0.0097985, 45: 0.0095925, 46: 0.0117155, 47: 0.010836, 48: 0.008759, 49: 0.0094555, 31: 0.0068225, 110: 0.0059, 111: 0.004555, 112: 0.0027495, 113: 0.0042805, 114: 0.0032085, 115: 0.0024095, 116: 0.0027225, 60: 0.014281, 61: 0.013728, 62: 0.011504, 63: 0.0125065, 64: 0.0152855, 65: 0.013477, 73: 0.0130835, 74: 0.012617, 75: 0.012722, 76: 0.010876, 66: 0.016421, 67: 0.015844, 68: 0.015861, 69: 0.0160305, 70: 0.013975, 71: 0.012459, 72: 0.0120285, 117: 0.002476, 118: 0.0022935, 22: 0.0072785, 23: 0.0059555, 24: 0.00545, 25: 0.004958, 26: 0.005316, 27: 0.005872, 28: 0.007191, 84: 0.00725, 29: 0.005364, 30: 0.0069555, 14: 0.002628, 15: 0.0027675, 16: 0.003157, 17: 0.001933, 18: 0.0038, 19: 0.0039415, 20: 0.004175, 21: 0.0053745, 127: 1.45e-05, 10: 0.0014425, 11: 0.0009765, 12: 0.00083, 13: 0.001959, 9: 0.0023225, 8: 0.0013605, 7: 0.0006095, 4: 0.0001185, 5: 0.000468, 6: 0.0004175

BIBLIOGRAPHY

[1] Freepdk 45 cell library. URL: https://eda.ncsu.edu/freepdk/freepdk45/.

[2] Google's introduction to silent data errors. URL: https://support.google.com/cloud/answer/10759085?hl=en.

[3] Passmark cpu benchmark results. URL: https://www.cpubenchmark.net/.

[4] Synopsis vcs functional verification solution. URL: https://www.synopsys.com/verification/simulation/vcs.html.

[5] Yosys open synthesis suite. URL: https://github.com/YosysHQ/yosys.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. doi:10.1109/TDSC.2004.2.

[7] IEEE Spectrum Blog. Impact of cosmic rays in hardware. URL: https://spectrum.ieee.org/cosmic-ray-failures-of-power-semiconductor-devices.

[8] James Charles, Preet Jassi, Narayan S Ananth, Abbas Sadat, and Alexandra Fedorova. Evaluation of the intel® core™ i7 turbo boost feature. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 188–197, 2009. doi:10.1109/IISWC.2009.5306782.

[9] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. *CoRR*, abs/2102.11245, 2021. URL: https://arxiv.org/abs/2102.11245, arXiv:2102.11245.

[10] D. Gizopoulos, A. Paschalis, and Y. Zorian. An effective built-in self-test scheme for parallel multipliers. *IEEE Transactions on Computers*, 48(9):936–950, 1999. doi:10.1109/12.795222.

[11] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 9–16, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3458336.3465297.

[12] David Lundgren. Double fpu verilog model. URL: https://opencores.org/projects/double_fpu.

[13] Arthur D. Friedman Miron Abramovici, Melvin A. Breuer. Digital systems testing and testable design. URL: https://ieeexplore.ieee.org/book/5266057.

[14] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. Silifuzz: Fuzzing cpus by proxy. *CoRR*, abs/2110.11519, 2021. URL: https://arxiv.org/abs/2110.11519, arXiv:2110.11519.

[15] G. Tziantzioulis, A. M. Gok, S M Faisal, N. Hardavellas, S. Ogrenci-Memik, and S. Parthasarathy. b-hive: A bit-level history-based error model with value correlation for voltage-scaled integer and floating point units. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015. doi:10.1145/2744769.2744805.

[16] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. Understanding silent data corruptions in a large production cpu population. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 216–230, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3600006.3613149.

[17] Yun-Ting Wang, Kai-Chiang Wu, Chung Han Chou, and Shih-Chieh Chang. Aging-aware chip health prediction adopting an innovative monitoring strategy. pages 179–184, 01 2019. doi:10.1145/3287624.3287687.