AN EFFICIENT AVF ESTIMATION TECHNIQUE USING CIRCUIT PARTITIONING

By

Jugantor Chetia

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

May 2012

Nashville, Tennessee

Approved:

Professor Lloyd W. Massengill

Professor Bharat L. Bhuva

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Dr. Lloyd W. Massengill for giving me the opportunity to work on this project. I would like to thank Dr. Massengill for his patience and moral support during the course of this project. I would also like to thank Dr. Massengill for guiding me in every step of this project, without which it would have been difficult to successfully complete this project. Next, I would like to thank Dr. Bharat Bhuva for his guidance and assistance during this project. I would like to thank Andrew Sternberg, Brian Sierawski, Daniel Loveless and Corey Toomey for assisting me in this project with endless technical discussions. I would also like to thank my friends at Vanderbilt: Srikanth, Tania, Sandeepan, Nihaar, Adeola, Shubhajit, Gaurav, Siladitya, Indranil and Parikshit, for all their help throughout my stay at Vanderbilt. I would like to thank the Radiation Effects and Reliability group for providing me with immense guidance and motivation throughout this project.

I would also like to thank my friends Anuj, Anupam, Dipak, Manash, Deepak and Vaibhav. Finally I would like to thank my parents and my brother for their immense support and love throughout all these years.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS ii

LIST OF FIGURES v

LIST OF TABLES vii

Chapter

1. INTRODUCTION 1

2. RADIATION-INDUCED SOFT ERRORS OVERVIEW 3

2.1 Types of radiation particles 3
2.2 Physical origins of single event upsets 4
2.2.1 Charge deposition 4
2.2.2 Charge collection 5
2.3 Concept of single event upset 6
2.3.1 SEUs in DRAM 6
2.3.2 SEUs in latches/SRAM 7
2.3.3 Soft fault 8
2.3.4 Soft error 9
2.3.5 Observability 10
2.4 Architectural perspective 10
2.4.1 AVF Overview 11
2.5 Motivation behind modular approach 12

3. PROBABILITIC APPROACH 14

3.1 Definitions 15
3.2 Assumptions 18
3.3 Methodology 20

4. IMPLEMENTATION OF MODULAR TECHNIQUE 24

4.1 Input file description 25
4.1.1 Behavioral or algorithmic level 25
4.1.2 Dataflow level 25
4.2 Circuit partitioning 26
4.3 Logging input data 30

4.4 Fault generation phase 31
4.4.1 Verilog programming language interface 32
4.5 Fault propagation phase 37
4.6 Post simulation AVF calculation 39

5. APPLICATIONS 41

5.1 Circuit classification 41
5.1.1 Mathematical interpretation 42
5.1.2 Example partitions 42
5.2 Test circuits 45
5.2.1 Digital signal processing units 45
5.2.1.1 Double precision floating point units 45
5.2.1.2 Forward DCT Unit 49
5.2.1.3 Baseline JPEG encoder 53
5.2.2 Pipelined Processor Unit 56

6. CONCLUSIONS 59

REFERENCES 60

LIST OF FIGURES

Page

1. Thermal neutron reaction generating alpha and gamma particles 5
2. Charge collection mechanism and generated transient current........ 6
3. SEUs in DRAMs........ 7
4. SEU in SRAMs........ 8
5. Types of soft fault 9
6. Concept of soft error 9
7. Example circuit to demonstrate observability concept 10
8. Example design to demonstrate the motivation behind modular approach 12
9. Example design to develop probability model........ 14
10. Probability of fault generation 16
11. Probability of fault propagation 17
12. Flowchart of modular approach 24
13. Verilog code of a synchronous D flips flop........ 26
14. Top-down design methodology........ 27
15. Bottom-up design methodology........ 27
16. Ripple carry counter........ 28
17. T flip-flop........ 28
18. Verilog description of T flipflop 29
19. Verilog description of D flipflop........ 29

20. Verilog description of 4-bit ripple carry counter 29

21. Code snippet showing $fdisplay usage 30

22. Code snippet showing $dumpvars usage 31

23. Fault generation simulation approach 32

24. Flowchart for fault generation simulation 33

25. PLI function call from inside Verilog code 33

26. Fault injection PLI function example 34

27. AVF vs injection for 8-bit microprocessor 36

28. Example fault propagation simulation 37

29. Pseudo-code for fault propagation phase 37

30. Verilog code demonstrating $value$plusargs usage 38

31. Flowchart for fault propagation simulation 39

32. Example design for circuit classification analysis 41

33. Canonical clocked logic circuit 42

34. Double Precision Floating Point Unit 45

35. Forward DCT Unit 49

36. DCT unit 51

37. Baseline JPEG Encoder 54

38. DCT module for JPEG encoder 55

39. Five stage pipeline unit 57

LIST OF TABLES

Page

1. Time when fault is injected at input and observed at output for fpu_round 47

2. Time when fault is injected at input and observed at output for fpu_exceptions 47

3. Results for floating point unit 48

4. Results for DCT unit 52

5. Results for JPEG Encoder 56

6. Results for pipelined processor unit 58

CHAPTER I

INTRODUCTION

Because of the continued device size scaling in CMOS technology, reliability has become an increasing concern for VLSI designs today. Reduced feature size, increased chip density, reduced supply voltage, shrinking nodal capacitances have resulted in an increased susceptibility of these designs to single event effects due to ionizing radiations [1][2][3]. These effects can result in soft errors, in a design, that can further lead to either malfunctioning of a part of a design or a complete system failure. Therefore, it is very important to have efficient techniques in place for the research community that would allow designers to analyze, quantify and mitigate the effects of such events.

For two broad reasons, it is advantageous at times to look at soft errors from an architectural perspective [4]. First, many of the soft faults that are seen at the device/circuit level are masked at the architectural level. This allows the designers to implement fast and low cost solutions for soft errors. For example, Wang et. al has shown that 85% of the soft faults in a processor are masked at the architectural level [5]. The primary reason for such a high masking rate is because of relatively low resource utilization in a modern microprocessor, large number of bits that only affect performance but not correctness and high logical masking of soft faults [4]. Second, looking at a design from an architecture point of view helps in understanding the workload behavior of the design, which can further lead to potentially more efficient soft error mitigation techniques [6]. Thus in order to implement efficient radiation hardening techniques at an early stage of the design cycle, it is advantageous to have tools and techniques to analyze the design for its soft error vulnerability at its architectural level.

This thesis work presents a novel efficient technique to evaluate the soft error vulnerability of large ASIC designs by employing circuit partitioning and fault propagation techniques. The technique was tested on four test circuits of considerable size. For comparison purposes, traditional statistical fault injection was also performed on the test circuits to evaluate their soft error vulnerability. Results show that our technique estimates the vulnerability within a reasonable accuracy, but takes about a 4X less computation effort than the traditional statistical fault injection technique.

This thesis is organized is follows. Chapter II provides a brief overview of the various mechanisms that cause soft errors and some of the already available methodologies to determine soft error vulnerability of a design. Also we talk about the motivation for our approach in chapter II. Chapter III describes the probability model of our approach. Chapter IV describes the implementation details of our approach. Chapter V talks about circuit characterization and discussion of results. Finally we conclude this thesis with a summary in Chapter VI.

CHAPTER II

RADIATION-INDUCED SOFT ERRORS OVERVIEW

The universe we live in is filled with different types of radiation particles. These particles can originate from varying sources and exist in varying forms in the outer space. Until recently, these particles were a source of concern for space and military electronic applications [7][8]. These particles interact with the semiconductor device and can cause unwanted changes in the device operation and the application as a whole. As we are coming down to smaller technology node, the problem gets worse with these particles interacting with the device in more intricate ways to cause more damage to a device. Moreover, at such small technology node, terrestrial neutrons, which earlier were not a source of concern, start to affect commercial electronic applications as well [9].

2.1 Types of radiation particles

a) Heavy ions: The sources of these particles in the outer environment are the galactic cosmic rays. Examples of such particles are iron and titanium ions [10].

b) Protons: The sources of protons are solar event particles and Earth's radiation belts, Van Allen belts [10].

c) Alpha: These particles appear as contaminants in packaging materials. There are some on-chip sources of alpha particles as well. For example, on-chip metallization, bonding metallization etc [10][11].

d) Neutron: High energy neutrons are produced by the interaction of solar and galactic cosmic rays, principally protons and heavy ions, in the upper atmosphere [10].

2.2 Physical Origins of Single Event Upsets

2.2.1Charge Deposition

Charge deposition due to an ionization radiation particle can take place in two possible ways:

a) Direct ionization:

This takes place when the ionizing particle passes through the semiconductor material and loses energy by Rutherford scattering with the lattice nuclei. The energy gets transferred to the bound electrons which then get activated into the conduction band, thus imparting a dense track of electron-hole pairs. With decreasing transistor feature size, not only heavy ions but even lighter particles e.g. protons are able to cause damage to the semiconductor device via direct ionization [12][13][14].

b) Indirect ionization:

Light particles like protons and neutrons, which are incapable to produce any effect via direct ionization, can produce significant adverse effects via indirect mechanisms. When a high energy proton or neutron enters the semiconductor lattice, it may undergo an inelastic collision with the target nucleus causing a nuclear reaction. As a result of the reaction, several particles may get released which can, in turn, cause single event upsets via direct ionization. An example nuclear reaction of a neutron with a Boron atom is shown in figure 1.1. It generates gamma, alpha particle and a lithium nucleus [12][15][16].

Example thermal neutron reaction:

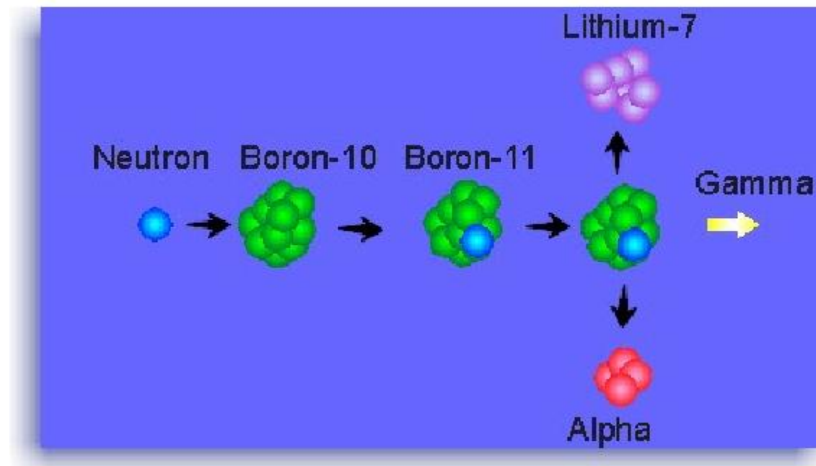

Figure 2.1 Thermal neutron reaction generating alpha and gamma particles

2.2.2 Charge Collection

The electron-hole pairs generated because of direct or indirect ionization by a charged particle may recombine and produce no observable adverse effect. But if the ion strike is in a reverse biased p-n junction of a semiconductor device, then the high electric field present in the reverse biased junction depletion region can collect the particle-induced charge (by causing separation of electron hole pairs) through drift processes causing a current flow at the junction contact. Even if the ion strike is not exactly in the p-n junction but near it, transient carriers can result as the carriers can diffuse into the vicinity of the diffusion region where they can again be efficiently collected [12]. Thus charge collection at the IC junctions leads to a circuit response to single events [17][18].

Figure 2.2 [1] shows the charge collection mechanism. Figure 2.2(a) shows an ion strike and the resulting ion track created, at the drain of a transistor. Figure 2.2(b) shows the drifting of the electron hole pairs and creation of a funnel shape extending the high field depletion region deeper into the substrate. Figure 2.2(c) shows the diffusion dominating the charge collection process. Figure 2.2(d) shows the result transient current generated as result of the charge collection.

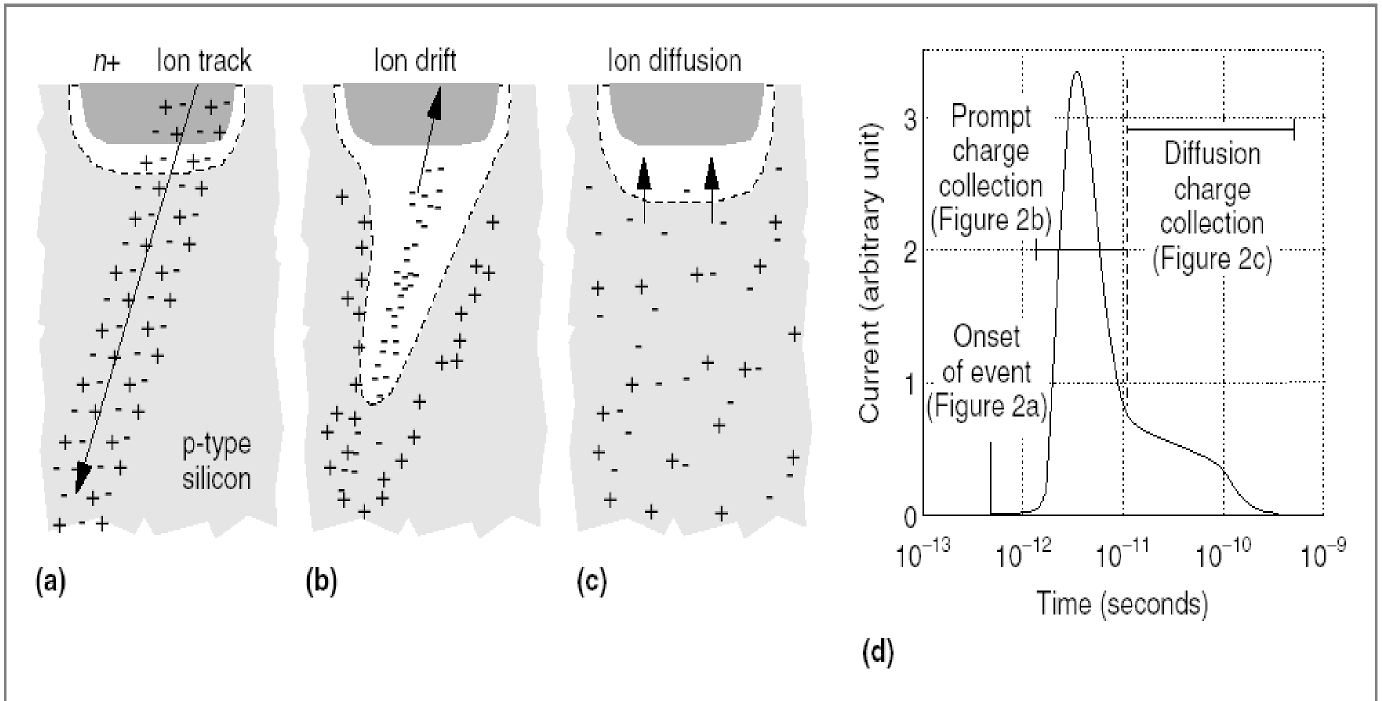

Figure 2.2 Charge collection mechanism and generated transient current

2.3 Concept of Single Event Upset

A single event upset (SEU) is defined as a bit flip or other corruption of stored information because of deposition or generation of charge due to a single event.

2.3.1 SEUs in DRAM

DRAMs store static charge passively on a capacitance. The charge is stored on a capacitance node and there is no active path to keep the charge regenerating. Hence, any disturbance to the stored information by a particle strike stays forever (unless corrected by an external circuitry) [19]. Thus any disturbance that can degrade the stored information to a level outside the noise margin associated with the bit signal is enough to manifest itself as an upset. Figure 2.3 shows this effect [10].

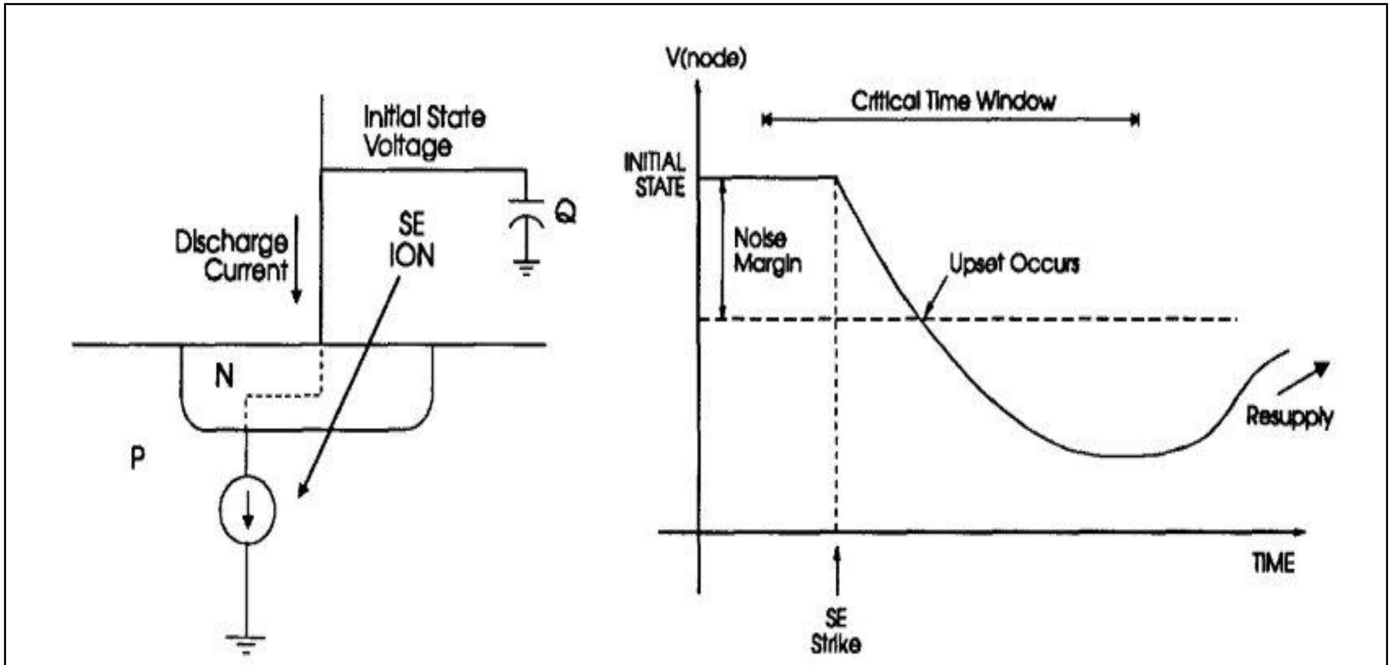

Figure 2.3 SEUs in DRAMs

2.3.2 SEUs in latches/SRAMs

SRAMs use an active high-gain positive feedback path to continuously regenerate the stored charge (active charge) [20][21]. Thus, if because of a particle strike at the information node, signal degrades, it gets regenerated by the active feedback. Hence, in order for an upset to occur, the charge deposited by the particle strike has to discharge the signal at the output node as well as overcome the self-compensation current of the circuit. Unlike in the case of DRAMs, here an upset occurs only if the stable state of the circuit switches completely (from 0->1 or 1->0). Figure 2.4 shows this effect [10].

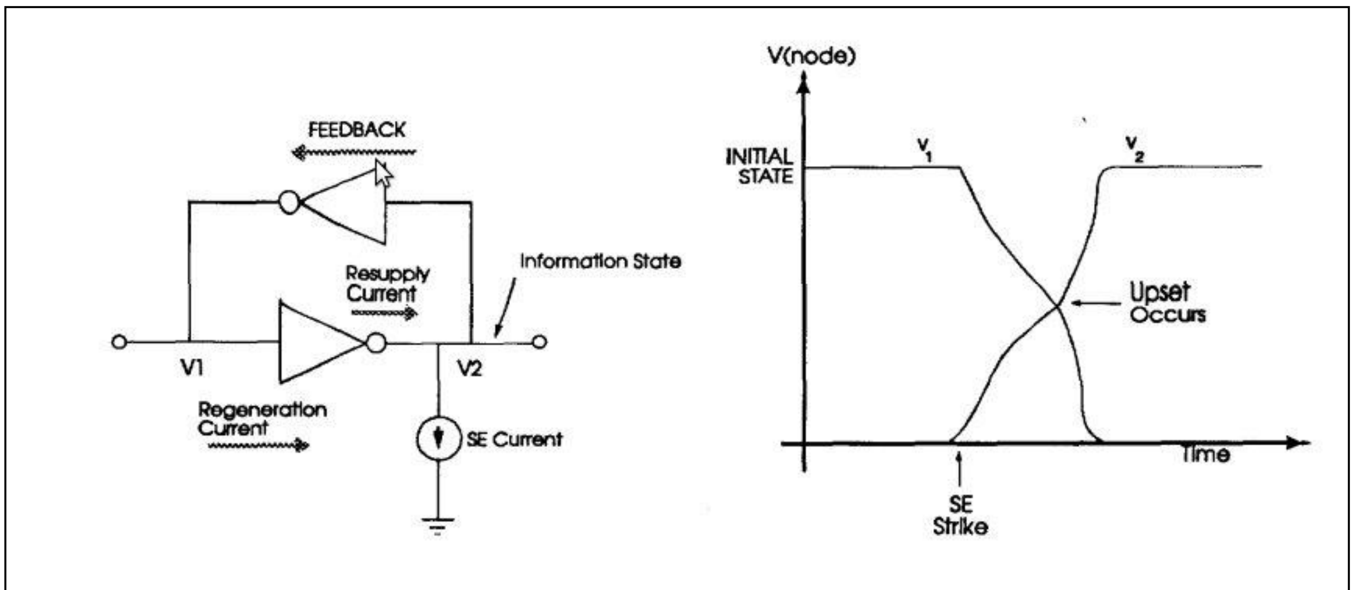

Figure 2.4 SEU in SRAMs

Before we delve into the system level aspect of single events, we define the following important concepts:

2.3.3 Soft fault:

A soft fault is defined as an incorrect state of the outputs of latches at the end of any clock cycle [22]. There are two ways in which a soft fault can take place:

- Direct latch hit: These are the soft faults that occur when the ionizing particle strikes the latch directly, thus flipping the output instantly.
- Combinational node hit: When the ionizing particle strikes a combinational node, it generates a transient noise pulse. Depending on logical, temporal or electrical masking, [23] if this pulse is travels through its propagation path and gets latched at the output of the latch, it results in a soft fault.

These two kinds of soft faults are shown in figure 2.5

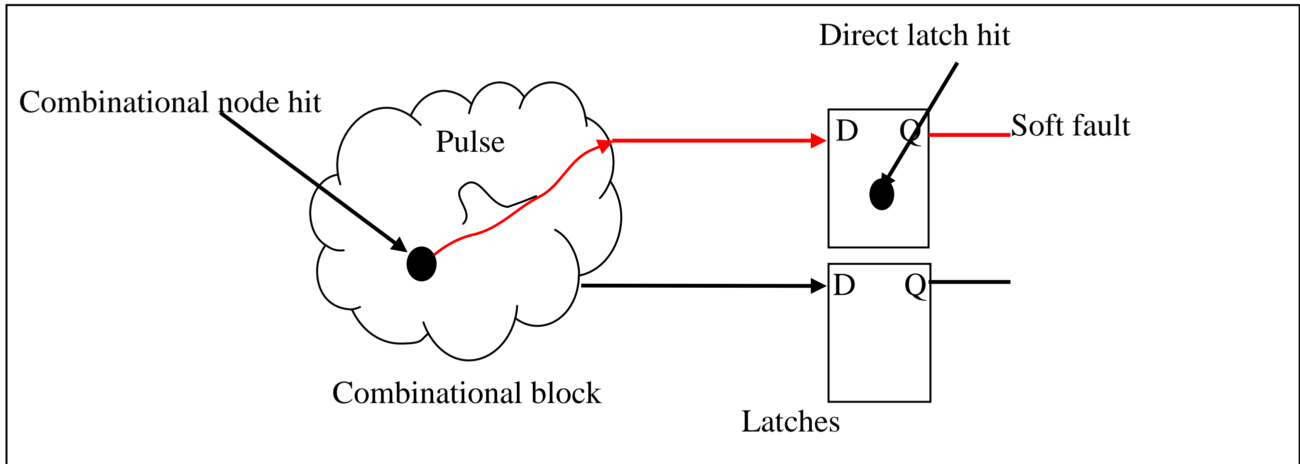

Figure 2.5 Types of soft fault: the red colored path shows the path through which the pulse travels and shows up at the output of the latch

2.3.4 Soft Error:

When a soft fault propagates through and becomes observable at the primary output of the system, we call it a soft error. Figure 2.6 shows a soft error.

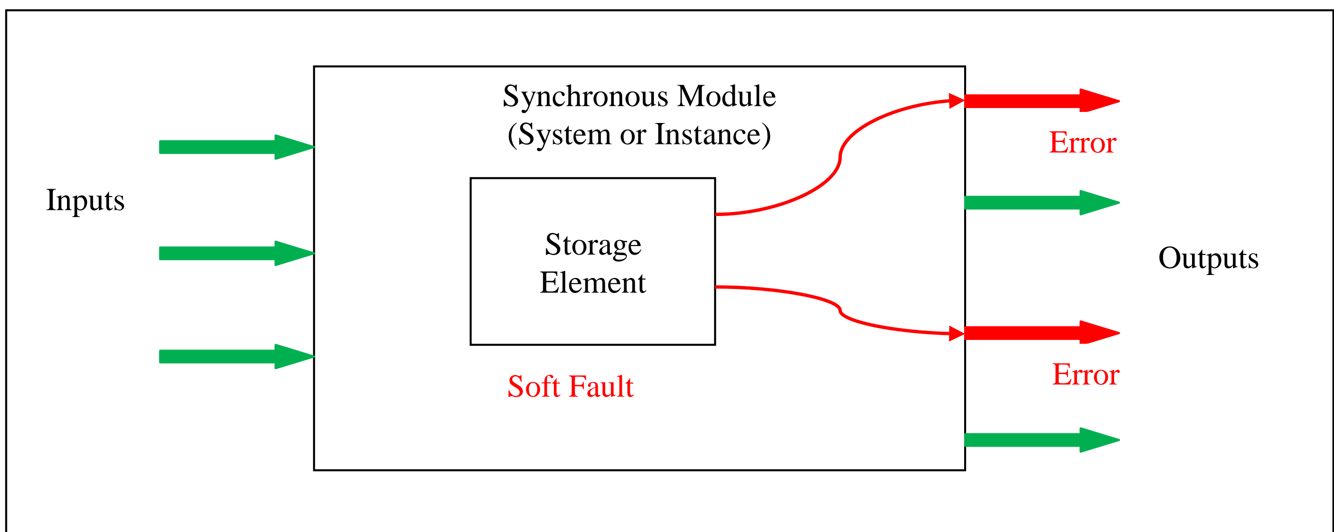

Figure 2.6: Concept of soft error

2.3.5 Observability:

It is defined as the ability to observe the logic value of an internal node at a primary output [24]. To test the soft error vulnerability of a node, all combinations of inputs going to that node must be applied and the output corresponding to each input must be observed. For example, in figure 2.7, to observe the output of gate 1, we need to set input C to 1 and D to 0. To observe the output of gate 2, we just need to set input D to 0. This implies that the output of gate 2 is more observable that that of 1.

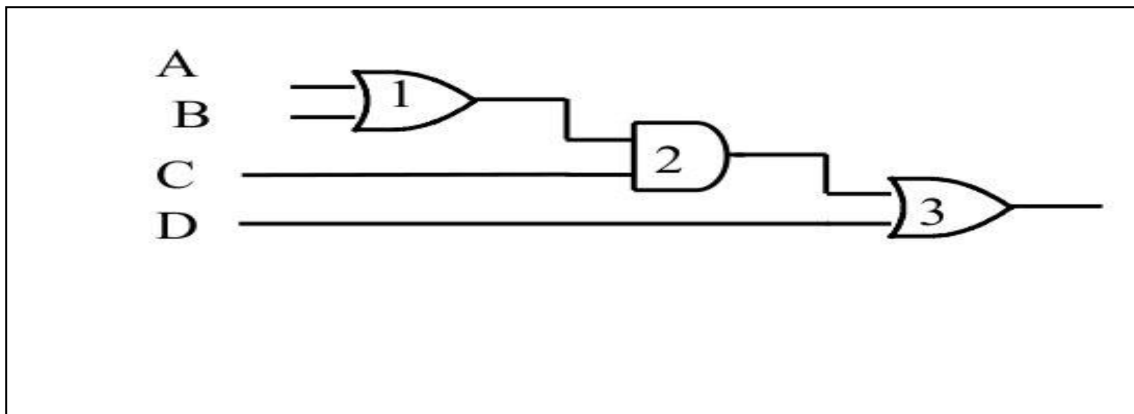

Figure 2.7 Example circuit to demonstrate observability concept

The number of test vectors increases as the number of testable nodes increases, making observability a difficult task.

2.4 Architectural Perspective

From an architecture point of view, module soft error reliability can be measured in terms of MTTF (mean time to failure) and FIT (failure in time). MTTF is defined as the time taken for a system to see an error for the first time. The reciprocal of MTTF is the rate of errors, generally reported as errors per billion hours of operation or FIT [25]. FIT can be observed as a measure of the three types of

masking (electrical, temporal, logical) that a transient pulse must overcome before it shows up at the concerned output. We use the Architectural vulnerability Factor (AVF) to quantify the amount of logical masking in a system [26].

2.4.1 AVF Overview

AVF is defined as the probability that a soft fault leads to an architecturally observable error. There are, in general, three ways to estimate the AVF of a system [26][27]: analytical models, performance models and statistical fault injection.

Analytical models are useful at an early stage of a design when neither a performance model nor an RTL (register transfer level) model is available. It computes the AVF using the average number of architecturally correct execution (ACE) bits in a structure.

Performance models estimate AVFs based on the fraction of time a bit is un-ACE in its life time. Although it is more accurate than statistical fault injection techniques, it can be very difficult to identify the un-ACE components. It often requires the person running the model to have a very good knowledge about the internal microarchitecture and operation of the design.

In statistical fault injection (SFI), an RTL model of the system is taken as input and bit flips, randomized in space and time, are introduced into its internal storage elements of the design. The model is then run forward and the architectural state of the output is compared to that of an error-free model of the system. A mismatch between the two is counted as an error. This process is repeated over until the desired number of errors is observed at the output. AVF is then computed as the ratio of the number of errors observed at the output to the number of bit flips introduced. SFI has the advantage of giving very accurate information about the AVF of a system without having to fully comprehend its internal details, unlike the analytical or performance models that need more human intervention than SFI. It makes since SFI is a statistical analysis, it is necessary to perform several such fault simulations with different injection points to obtain a desired confidence level in the AVF. This, in turn, necessitates

running thousands of fault simulation runs on the same system, consuming an enormous amount of simulation time and computational power. This makes SFI unsuitable in situations where we need a fast AVF approximation. The problem gets worse for a large circuit where a large number of simulation runs are required to observe the effects of a single fault injection.

In this thesis, we attempt to investigate a novel modular approach to achieve greater efficiency in our fault simulations and thus AVF estimation, than would normally be achieved by using only SFI , performance model or analytical models. Our modular technique is a hybrid of SFI and error probability propagation based analytical techniques. It exploits circuit partitioning and error propagation techniques to estimate the AVF of a design within a reasonable accuracy but taking a lot fewer simulation runs than a full SFI ran on the same design.

2.5 Motivation behind modular approach

Consider figure 2.8. TopMod is made up of two sub-modules, m1 and m2, each with 10 registers. m1 has a 4-bit input and a 4-bit output connected to m2 while m2 has a 2-bit output.

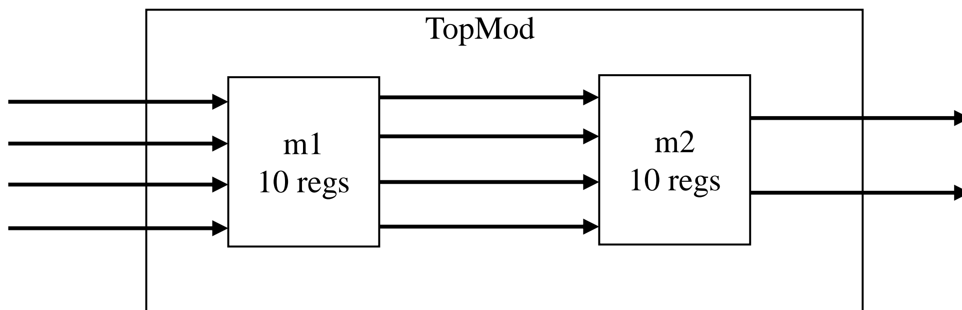

Figure 2.8 Example design to demonstrate the motivation behind modular approach

To observe the effect of all the possible faults in TopMod, the minimum number of fault injections required to cover the total fault population in TopMod would be: $2^{20} * 2^4 * 20 = 335$ million, taking into account the total number of possible states and input combinations. On the other hand, if we

perform SFI on m1 and m2 separately, the minimum number of fault injections required to cover the total fault population in TopMod would now be: $(2^{10} * 2^4 * 10) + (2^{10} * 2^4 * 10) + (2^{10} X 2^4) = 344064$. The last term accounts for the number of simulations required to propagate faults from m1 to the primary outputs of TopMod.

Thus, to estimate the AVF of TopMod within a certain confidence level, the number of simulation runs required is much less if we perform SFI on m1 and m2 separately than performing SFI on TopMod at one go. This is the motivation behind our approach presented. In the next chapter, we present a probabilistic analysis of our technique.

CHAPTER III

PROBABILISTIC APPROACH

In this chapter, we present a probabilistic model of our modular approach. Consider the example design in figure 3.1. TopMod has three sub-modules, Mod1, Mod2 and Mod3. The number of registers in Mod1, Mod2 and Mod3 is N1, N2 and N3 respectively. $\vec{w}$ is the primary input stream of TopMod and $\vec{z}$ is its primary output stream.

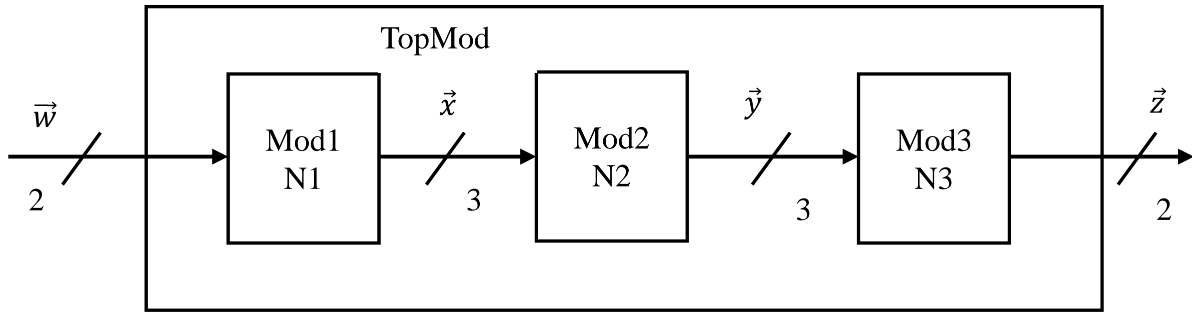

Figure 3.1 Example design to develop probability model

Also, in TopMod,

for Mod1: $\vec{w}$ is its 2-bit input stream and $\vec{x}$ is its 3-bit output stream that goes into Mod2

for Mod2: $\vec{x}$ is its 3-bit input stream and $\vec{y}$ is its 3-bit output stream that goes into Mod3

for Mod3: $\vec{y}$ is its 3-bit input stream and $\vec{z}$ is its 2-bit output.

We denote the scalar components of $\vec{w}$ as w[1] and w[2]. We use similar notations to denote the scalar components of $\vec{x}$, $\vec{y}$ and $\vec{z}$. Now, when put in a radiation environment, registers of TopMod can get soft faults at their outputs, depending either by a direct hit on the register bit or via latching of

propagating of transient pulses because of combinational node hit. This can happen at any point of time during the radiation exposure.

3.1 Definitions

We first create a golden copy, $\vec{s}_{golden}[t]$, of a data stream, either input or output of a sub-module, at time t. The golden copy represents its state in a non-radiation environment. Now, in a radiation environment, the data stream, say $\vec{s}$ at time t, may or may not have faults. We define the fault function Fr $(\vec{s},t)$ as:

$$\text{Fr}(\vec{s},t) = \begin{cases} \text{False if } \vec{s}[t] = \vec{s}_{golden}[t] \\ \text{True otherwise} \end{cases} \quad \dots (1)$$

Equation (1) implies that if at time t, the data stream has the same value as the golden copy, the fault function evaluates to false indicating that no fault has occurred on it. For all other cases, it evaluates to true. Similarly, we define the error function Er $(\vec{s}, t)$, if the data stream $\vec{s}$ is from the primary output of the design. Based on this definition, we define a set of fault probabilities that we are going to use later on to estimate the AVF of a system.

Let us denote the set of registers of a sub-module, say Mod2, as *Regs(Mod2)* and the event where a register $Reg_i \in Regs(Mod2)$ has been injected with a fault as $U_p(Reg_i)$. We define the following probabilities for Mod2 (and similarly for all the other sub-modules):

1) Probability of fault generation:

It is defined as the probability that a soft fault in a sub-module propagates through its internal propagation path and shows up at the output of the sub-module.

For example, in figure 3.2, a particle strike causes a bit flip at one of the register bits of Mod2.

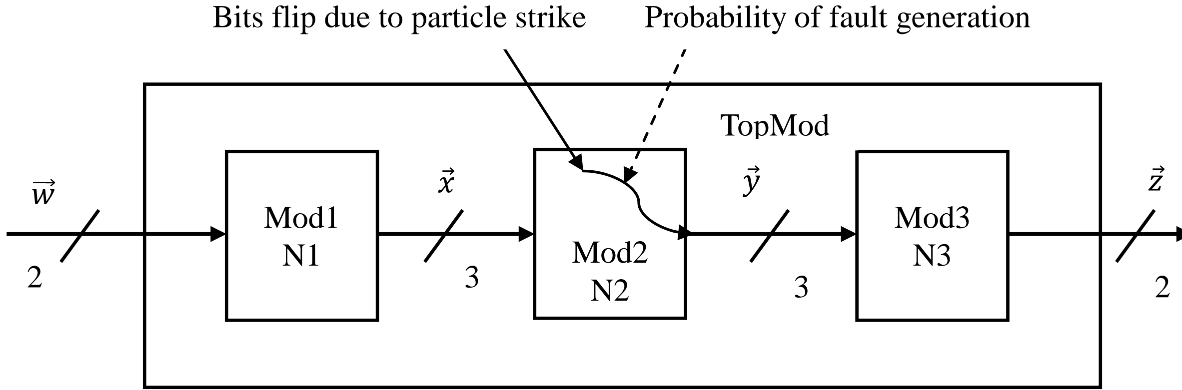

Figure 3.2 Probability of fault generation

Probability of fault generation defines the conditional probability that this bit flip causes the output $\vec{y}$ to change its original value.

Mathematically, we denote it as: $Pr \{Fr (\vec{y}, t) \mid U_p(Reg_i), Reg_i \in Regs(Mod2)\}$, where we define: $Fr (\vec{y}, t) \mid U_p(Reg_i), Reg_i \in Regs(Mod2)$ as the outcome of the stochastic process, where a fault in one of the registers, Reg_i of Mod2 produces a fault on the output stream, $\vec{y}$ at time *t*.

Similarly, the probability of fault generation for Mod1 is denoted by: $Pr \{Fr (\vec{x}, t) \mid U_p(Reg_i), Reg_i \in Regs(Mod1)\}$. For Mod3, since the outputs of Mod3 are also the primary outputs of TopMod, any fault that shows up at the output of Mod3 will manifest itself as an error in $\vec{z}$. Hence, we represent the probability of fault generation for Mod3 as: $Pr \{Er (\vec{z}, t) \mid U_p(Reg_i), Reg_i \in Regs(Mod1)\}$.

2) <u>Probability of fault propagation:</u>

It is defined as the probability that a fault at the input of a sub-module propagates through it and becomes observable at the output of the sub-module. The fault at the input of the

sub-module is assumed to have propagated from a previous module in its propagation path.

For example, in figure 3.3, faults generated in the sub-module Mod2 propagate from its output to the input of Mod3 via the interconnecting data stream $\vec{y}$. Now, these faults may or may not propagate to the output stream, $\vec{z}$, of Mod3, depending on the logical masking as well as register timing factors. Probability of fault propagation defines the probability of propagation of this fault from $\vec{y}$ to $\vec{z}$.

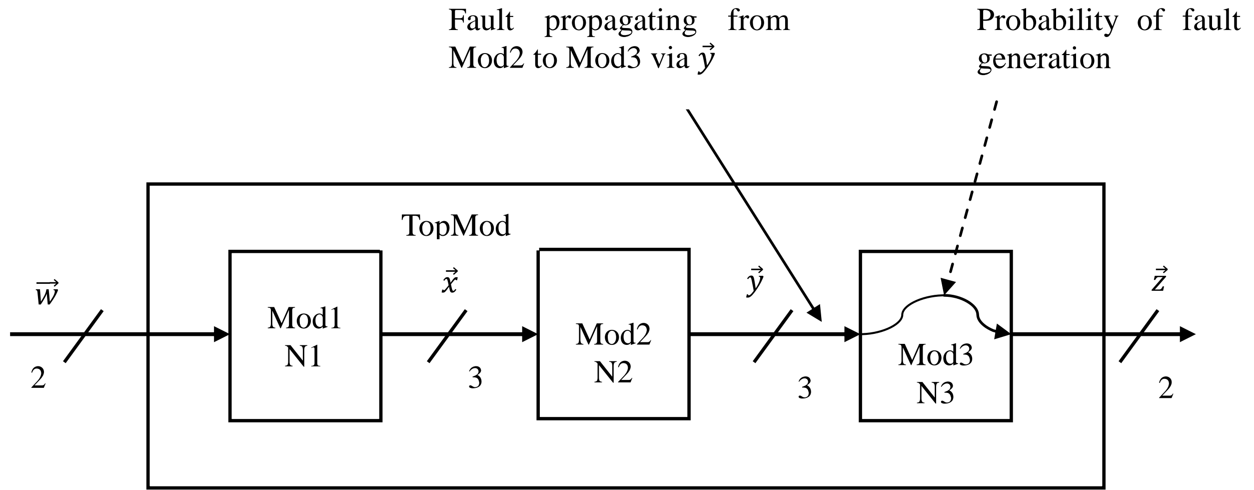

Figure 3.3 Probability of fault propagation

Mathematically, we denote it as: *Pr {Er* ($\vec{z}$, *t') | Fr* ($\vec{y}$, *t)}*, where *{Er* ($\vec{z}$, *t') | Fr* ($\vec{y}$, *t)}* is the event where a fault at $\vec{y}$ at time *t* propagates and shows up as an error at $\vec{z}$ at time *t'*. Similarly, for Mod2, the probability of fault propagation is denoted by: *Pr {Fr* ($\vec{y}$, *t') | Fr* ($\vec{x}$, *t)}*. We assume that there are no faults propagating through the primary inputs, $\vec{w}$, of TopMod, as we simulate a fault-free input bit stream.

In the next section, we lay out the assumptions that we must follow before applying our proposed modular approach for AVF estimation.

3.2 Assumptions

We make the following assumptions:

1) Consider TopMod in figure 3.1. Because in statistical fault injection, faults are injected randomly and uniformly in space, out of Mod1, Mod2 and Mod3, the sub-module with a higher number of register bits than another has a higher probability of faults getting injected into it than the one with less number of register bits. Mathematically, the probability that a random fault will get injected in Mod1 is given by: $\frac{N1}{N1+N2+N3}$. Similarly, the probability that a random fault will get injected in Mod2 and Mod3 is: $\frac{N2}{N1+N2+N3}$ and $\frac{N3}{N1+N2+N3}$.

2) Fault generation and fault propagation are strictly stationary processes. A strictly stationary process is defined as a stochastic process whose joint probability distribution doesn't change when shifted in time and space. Mathematically, if $\{X_t\}$ is a stationary process and if $F_X(x_{t1+\tau}, \ldots, x_{tk+\tau})$ represent the cumulative distribution function of the joint distribution of $\{X_t\}$ at times t1+ τ, …, tk+ τ. Then, $\{X_t\}$ is said to be stationary if, for all k, for all τ, and for all t1,…,tk,

$$F_X(x_{t1+\tau}, \ldots, x_{tk+\tau}) = F_X(x_{t1}, \ldots, x_{tk})$$

This gives rise to the following two cases:

a. In the case of the fault generation phase, the probability of a bit flip getting introduced in a register bit of a sub-module is time-independent. Hence, the probability of fault generation for that sub-module is time-independent as well. Mathematically, say for Mod2, we have:

$$Pr\{Fr(\vec{y}, t) \mid U_p(Reg_i), Reg_i \in Regs(Mod2)\} = Pr\{Fr(\vec{y}, t+\tau) \mid U_p(Reg_i), Reg_i \in Regs(Mod2)\}$$

where $\tau > 0$. … (2)

For simplicity, we get rid of the time variable and represent the probability of fault generation for Mod2 (and similarly for all other sub-modules) as:

$Pr\ \{Fr\ (\vec{y})\ |\ U_p(Reg_i),\ Reg_i \in Regs(Mod2)\}$ …(3)

b. In the case of the fault propagation phase, we assume that the probability of fault propagation for a sub-module is time-independent. Mathematically, say for Mod2, we have:

$Pr\ \{Fr\ (\vec{y},\ t')\ |\ Fr\ (\vec{x},\ t)\} = Pr\ \{Fr\ (\vec{y},\ t' + \tau)\ /\ Fr\ (\vec{x},\ t + \tau)\}$, where $\tau > 0$ …(4)

As with the previous case, we get rid of the time variables for simplicity and represent it for Mod2 (and similarly for Mod3) as:

$Pr\ \{Fr\ (\vec{y})\ /\ Fr\ (\vec{x})\}$ …(5)

3) The outputs of a sub-module are correlated to each other and so are the faults associated with them. This implies that for a sub-module, we should individually take into consideration all the possible ways in which faults can occur at its output. For example, output of Mod2 is of 3-bits. Hence, there are total 7 possible ways in which faults can occur at its outputs:

a) Faults at only one output at a time:

i) $Fr(y[1])\ /\ U_p(Reg_i),\ Reg_i \in Regs(Mod2)$

ii) $Fr(y[2])\ /\ U_p(Reg_i),\ Reg_i \in Regs(Mod2)$

iii) $Fr(y[3])\ /\ U_p(Reg_i),\ Reg_i \in Regs(Mod2)$

b) Faults at two outputs simultaneously:

i) $\{(Fr(y[1]),Fr(y[2]))\ /\ U_p(Reg_i),\ Reg_i \in Regs(Mod2)\}$

ii) $\{(Fr(y[2]),Fr(y[3]))\ /\ U_p(Reg_i),\ Reg_i \in Regs(Mod2)\}$

iii) $\{(Fr(y[1]),Fr(y[3]))\ /\ U_p(Reg_i),\ Reg_i \in Regs(Mod2)\}$

c) Faults at all the three outputs simultaneously:

$\{(Fr(y[1]),Fr(y[2]),Fr(y[3])) \mid U_p(Reg_i), Reg_i \in Regs(Mod2)\}$

3.3 Methodology

With the above assumptions in mind, we proceed to describe the steps to find the AVF of a design, in our case, TopMod. Hypothetically, we define the contribution of a sub-module Mod *i* to the AVF of TopMod, *AVF (TopMod | Mod i)*, as the AVF of TopMod when faults are injected only in sub-module Mod *i*. But because in reality faults occur randomly in space over TopMod, we assign weighting factors to each of the sub-modules. The weighting factors are derived from assumption 1. Thus mathematically, we represent the contribution of a Mod *i* to AVF as:

AVF (TopMod | Mod *i*) * (Probability of a fault getting injected in Mod *i*)

= AVF (TopMod | Mod *i*) * $\frac{Ni}{N1+N2+N3}$, where i = 1, 2, or 3 for Mod1, Mod2 and Mod3 respectively

… (6)

Summing over all the three sub-modules, the AVF of TopMod is then given by:

$$Pr\,\{Er\,(\vec{z}) \mid U_p(Reg_i),\ Reg_i \in Regs(TopMod)\} = \sum_{i=0}^{3} \frac{(AVF(TopMod \mid Mod\ i) * Ni)}{\sum_{i=0}^{3} Ni} \qquad \ldots (7)$$

In the next subsections, we describe the steps to compute the first factor, *AVF(TopMod | Mod i)* for each of the sub-modules.

a) <u>Computation of *AVF(TopMod | Mod3)*</u>:

Since the outputs of Mod3 are also the primary outputs of TopMod, we can compute *AVF (TopMod | Mod3)* using just the probability of fault generation. Thus, we have:

$$AVF\,(TopMod \mid Mod3) = Pr\,\{Er\,(\vec{z}) \mid U_p(Reg_i),\ Reg_i \in Regs(Mod3)\} \qquad \ldots (8)$$

Also, from assumption 3, we account for each type of errors on $\vec{z}$ and express equation (8) as:

$$AVF\ (TopMod\ /\ Mod3) = Pr\ \{Er\ (z[1])\ |\ U_p(Reg_i),\ Reg_i \in Regs(Mod3)\}$$

$$+\ Pr\ \{Er\ (z[2])\ |\ U_p(Reg_i),\ Reg_i \in Regs(Mod3)\}$$

$$+\ Pr\ \{(Er(z[1]),Er(z[2]))\ |\ U_p(Reg_i),\ Reg_i \in Regs(Mod3)\} \quad \ldots\ (9)$$

b) Computation of *AVF (TopMod / Mod2)*:

To observe the effect of a bit flip injected in Mod2 on $\vec{z}$, we have to propagate the effect of this bit flip from $\vec{y}$ to $\vec{z}$. Also, from assumption 2a and 2b, we conclude that both these events: fault generation in a sub-module and propagation of these faults through the next sub-module in the propagation path, are independent events. Hence, to observe the effect of a bit flip injected in Mod2 on $\vec{z}$, we apply the multiplication rule of probabilities and evaluate it as a product of the probability of fault generation (because of the flip) at $\vec{y}$ and probability of fault propagation from $\vec{y}$ to $\vec{z}$.

Mathematically,

$$AVF\ (TopMod\ /\ Mod2) = Pr\ \{Er\ (\vec{z})\ |\ U_p(Reg_i),\ Reg_i \in Regs(Mod2)\}$$

$$= (Pr\ \{Er\ (\vec{z})\ /\ Fr\ (\vec{y})\}) * (Pr\ \{Fr\ (\vec{y})\ |\ U_p(Reg_i),\ Reg_i \in Regs(Mod2)\}) \quad \ldots(10)$$

$\vec{y}$ and $\vec{z}$ are multi bit vectors. Hence, there are more than one way in which the injected bit flip can generate faults in $\vec{y}$ as well as more than one ways in which the fault can show up in $\vec{z}$. To evaluate equation (10), we use probability matrices to represent both the factors, since it is more convenient to deal with multi bit vectors with matrices.

Probability matrix for fault generation:

We represent it as K X 1 matrix where K is the total number of possible ways faults can be observed at the output of Mod2. In our example, the value of K is 7. Hence, we have the

following 7X1 probability matrix:

$$(Pr\ \{Fr\ (\vec{y}) \mid U_p(Reg_i),\ Reg_i \in Regs(Mod2)\})$$

$$= \begin{bmatrix} Pr\ \{Fr\ (y[1]) \mid U_p(Reg_i), Reg_i\ \in\ Regs(Mod2)\} \\ Pr\ \{Fr\ (y[2]) \mid U_p(Reg_i), Reg_i\ \in\ Regs(Mod2)\} \\ Pr\ \{Fr\ (y[3]) \mid U_p(Reg_i), Reg_i\ \in\ Regs(Mod2)\} \\ Pr\ \{(Fr\ (y[1]), Fr(y[2])) \mid U_p(Reg_i), Reg_i\ \in\ Regs(Mod2)\} \\ Pr\ \{(Fr\ (y[1]), Fr(y[3])) \mid U_p(Reg_i), Reg_i\ \in\ Regs(Mod2)\} \\ Pr\ \{(Fr\ (y[2]), Fr(y[3])) \mid U_p(Reg_i), Reg_i\ \in\ Regs(Mod2)\} \\ Pr\ \{(Fr\ (y[1]), Fr(y[2]), Fr(y[3])) \mid U_p(Reg_i), Reg_i\ \in\ Regs(Mod2)\} \end{bmatrix} \quad \ldots (11)$$

<u>Probability matrix for fault propagation</u>

We represent it as a J X K matrix where J is the total number of ways errors can occur in $\vec{z}$ and K is the total number of ways faults can propagate through $\vec{y}$. In this example, the value of J is 3 and K is 7. Hence we have the following 3X7 probability matrix:

$Pr\ \{Er\ (\vec{z})\ /\ Fr\ (\vec{y})\} =$

$$\begin{bmatrix} Pr\ \{Er(z[1])|Fr(y[1])\} & Pr\ \{Er(z[1])|Fr(y[2])\} & \ldots & Pr\ \{Er(z[1])|(Fr(y[1]), Fr(y[2]))\} & \ldots \\ Pr\ \{Er(z[2])|Fr(y[1])\} & Pr\ \{Er(z[2])|Fr(y[2])\} & \ldots & Pr\ \{Er(z[2])|(Fr(y[1]), Fr(y[2]))\} & \ldots \\ Pr\ \{(Er(z[1]), Er(z[2]))|Fr(y[1])\} & Pr\ \{(Er(z[1]), Er(z[2]))|Fr(y[2])\} & \ldots & Pr\ \{(Er(z[1]), Er(z[2]))|(Fr(y[1]), Fr(y[2]))\} & \ldots \end{bmatrix}$$

…(12)

Thus, by solving equation (10) as a product of equation (11) and equation (12), we get *AVF (TopMod / Mod2)* as a 3X1 matrix. Another way to express *AVF (TopMod / Mod2)* by adding the elements 3X1 matrix and using assumption 3 is:

$$Pr\ \{Er\ (z[1]) \mid U_p(Reg_i),\ Reg_i \in Regs(Mod3)\} + Pr\ \{Er\ (z[2]) \mid U_p(Reg_i),\ Reg_i \in Regs(Mod3)\}$$

$$+\ Pr\ \{(Er(z[1]), Er(z[2])) \mid U_p(Reg_i),\ Reg_i \in Regs(Mod3)\} = AVF\ (TopMod\ /\ Mod2) \quad \ldots (13)$$

c) Computation of *AVF(TopMod | Mod1)*

AVF(TopMod | Mod1) can be computed in a very similar manner as equation (10). The effect of a bit flip injected in Mod1 has to be propagated through Mod2 as well as Mod3. Mathematically,

$AVF(TopMod \mid Mod1) = Pr\{Er(\vec{z}) \mid U_p(Reg_i), Reg_i \in Regs(Mod1)\}$

= *Probability of fault propagation through Mod3 * Probability of fault propagation through Mod2 * Probability of fault generation in Mod1*

$$= (Pr\{Er(\vec{z}) \mid Fr(\vec{y})\}) * (Pr\{Fr(\vec{y}) \mid Fr(\vec{x})\}) * (Pr\{Fr(\vec{y}) \mid U_p(Reg_i), Reg_i \in Regs(Mod2)\}) \quad \ldots(14)$$

We use probability matrix multiplication again to evaluate equation (14). The first factor $(Pr\{Er(\vec{z}) \mid Fr(\vec{y})\})$ can be expressed as a 3X7 matrix, the second factor $(Pr\{Fr(\vec{y}) \mid Fr(\vec{x})\})$, can be expressed as a 7X7 matrix, since both $\vec{y}$ and $\vec{x}$ are 3-bit. The third factor $(Pr\{Fr(\vec{y}) \mid U_p(Reg_i), Reg_i \in Regs(Mod2)\})$ can be expressed as a 7X1 matrix, thus giving us a 3X1 final matrix for *AVF(TopMod | Mod1)*.

CHAPTER IV

IMPLEMENTATION OF MODULAR TECHNIQUE

In this chapter, we describe how to go about implementing our modular technique. Figure 4.1 shows a high level flowchart view of our technique:

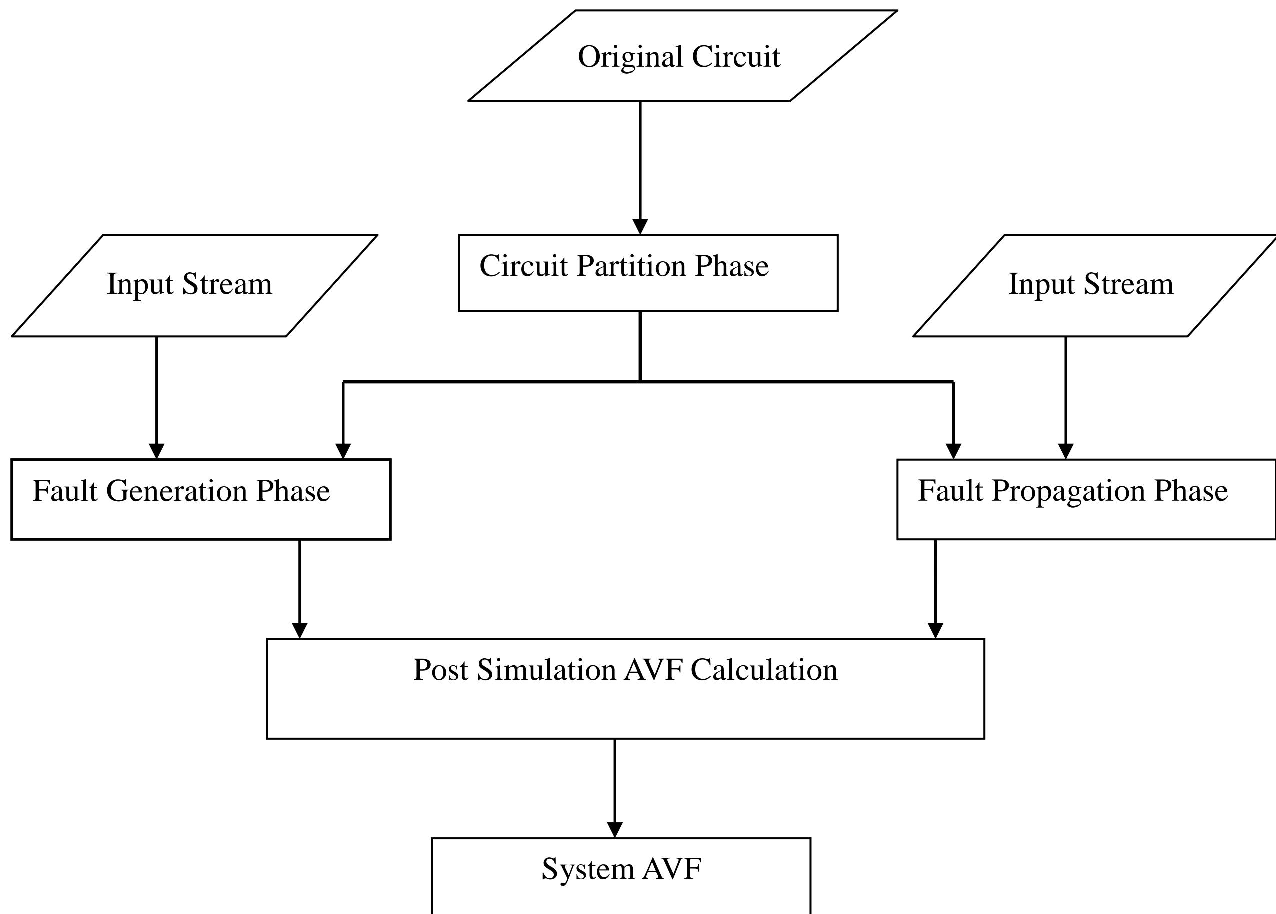

Figure 4.1 Flowchart of modular approach

We take in a Verilog HDL RTL description of a circuit as input and apply a circuit partitioning algorithm to make partitions of the circuit. We run a predefined testbench on the original circuit and log the inputs of each partition. Next, we take up each partition and operate fault generation and propagation simulations on them to find their probabilities of the two. We need not perform fault propagation through the primary inputs of the circuit as we are using an external fault-free testbench. Once we perform these two simulations and generate probability matrices, we perform post simulation calculations using the information about the interconnection between the different partitions/sub-modules, as described chapter II. In the next sub-sections, we describe RTL file format followed by techniques to partition circuits, to log input data and to perform fault generation and propagation simulations.

4.1 Input file description

Typically, the input file can be either a Verilog or a VHDL description. In our case, we take a Verilog RTL description. RTL or register transfer language is a term used for a Verilog description that uses a combination of behavioral and dataflow constructs (described below) and is acceptable to logic synthesis tools. It describes how data is transformed as it flows from register to register. The transforming of the data is performed by the combinational logic.

4.1.1 Behavioral or algorithmic level

This is the highest level of abstraction provided by Verilog HDL. In this level, a design module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming [28].

4.1.2 Dataflow level

At this level, the module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how the data is processed in the design.

Describing a circuit at the register transfer level makes it very convenient and flexible to create high level representations of a circuit. Also, it gives us the freedom to design technology independent circuits. In terms of simulation, it takes smaller simulation time and computational effort to simulate a design described in RTL than a design that is described at a lower level (e.g. gate level or switch level). Figure 4.2 shows a sample RTL code of a synchronous D flip flop written in Verilog.

```
module dff_sync_reset (
data   , // Data Input
clk    , // Clock Input
reset  , // Reset input
q        // Q output
);
              input data, clk, reset ;
              output q;

              reg q;

              always @ ( posedge clk)
              if (~reset) begin
              q <= 1'b0;
              end  else begin
              q <= data;
              end

endmodule //End Of Module dff_sync_reset
```

Figure 4.2 Verilog code of a synchronous D flips flop

4.2 Circuit Partitioning

Once we have the input circuit file in RTL format, we create partitions of the circuit. Circuit partitioning can be accomplished in several ways, based on a given set of criteria (described in Chapter IV). Typically, it can be done based on the hierarchy of the design. Most RTL Verilog codes are written in accordance with two most common hierarchical modeling concepts: top-down design methodology

and bottom-up design methodology [28].

In a top-down design methodology, we define the top-level block and identify the sub-blocks necessary to build the top-level block. We further subdivide the sub-blocks until we come to leaf cells, which are the cells that cannot further be divided. Figure 4.3 shows this methodology.

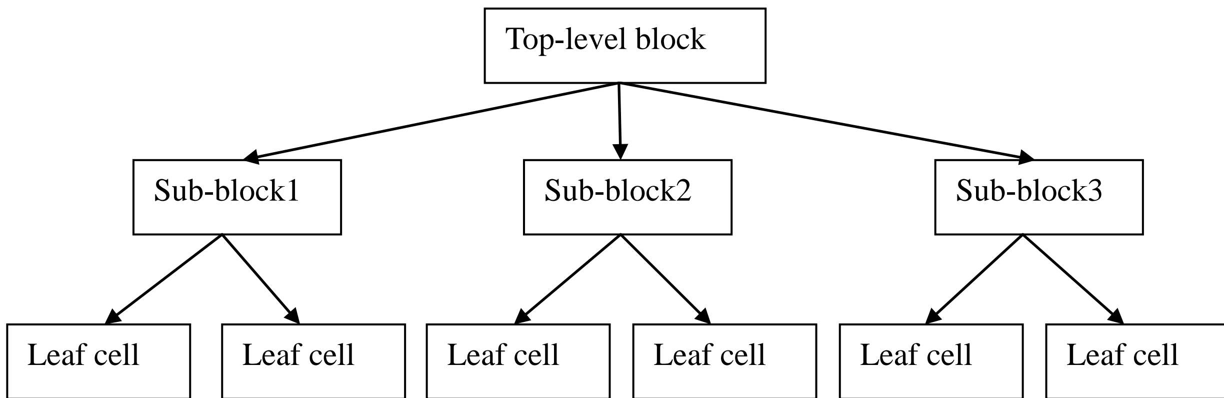

Figure 4.3 Top-down design methodology

In a bottom-up design methodology, the building blocks that are available to the designed are first identified. Using these blocks, bigger cells are built, which are then used to build higher-level blocks until the top-level block of the design is built. Figure 4.4 shows this methodology.

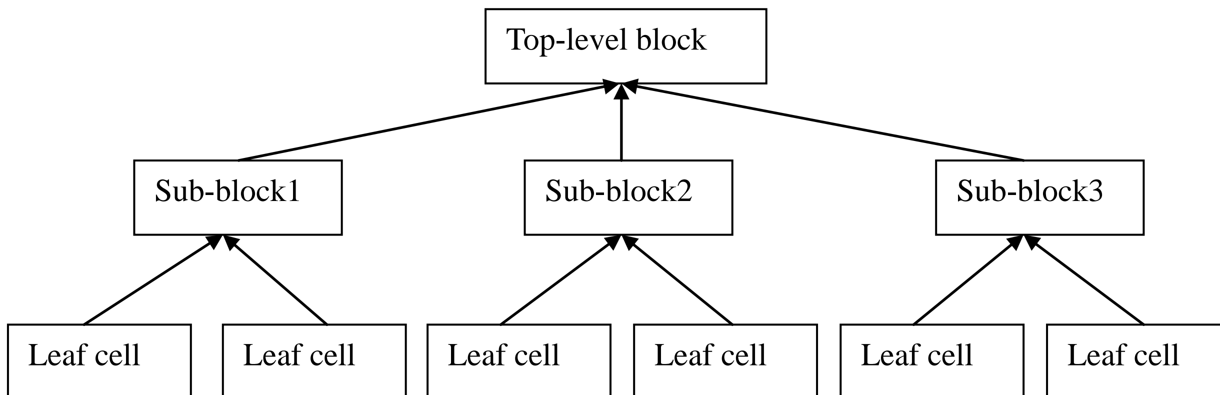

Figure 4.4 Bottom-up design methodology

In a typical system design, a combination of top-down and bottom-up methodology is used. To illustrate circuit partitioning using hierarchical modeling, we consider a 4-bit ripple carry counter as shown in figure 4.5.

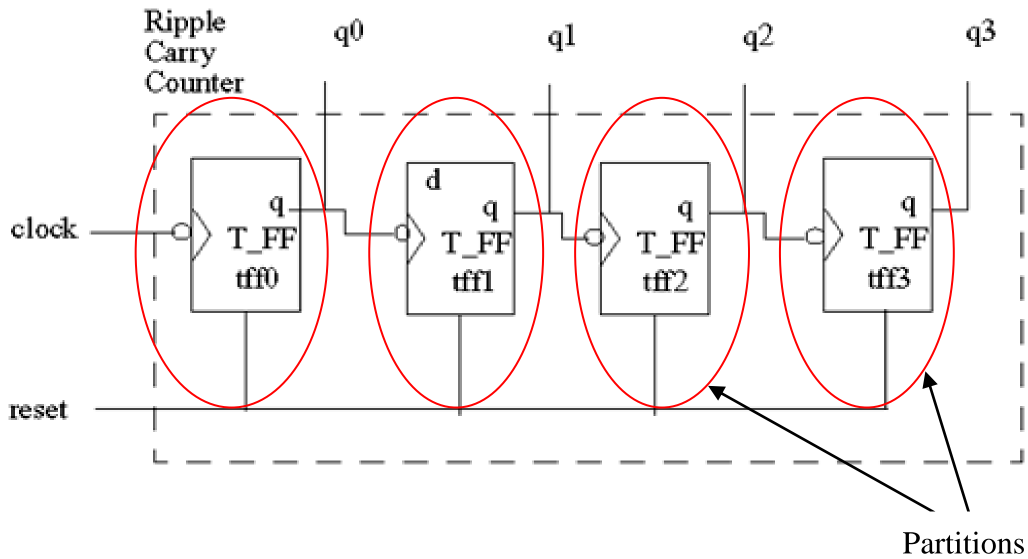

Figure 4.5 Ripple carry counter

The ripple carry counter is made up of 4 T flipflops. Each T flipflop is made up of a D flip flop and an inverter as shown in figure 4.6

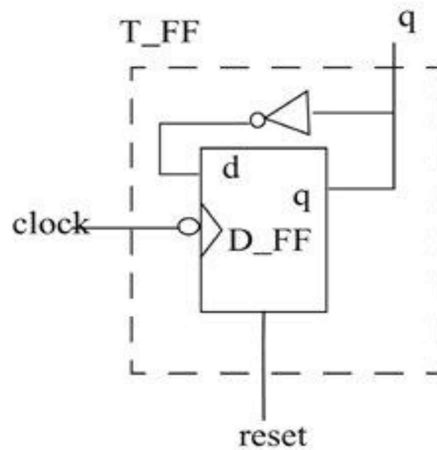

Figure 4.6 T-flipflop

Thus the ripple carry counter is built in a hierarchical fashion by using building blocks. Now, we can safely divide this counter, without losing any functionality, into four partitions, each with one flipflop as shown by the red ellipses. From RTL point of view, figure 4.7 shows the Verilog description of the T flipflop using a D flipflop (figure 4.8) and an inverter. Figure 4.9 shows the Verilog description of the 4-bit ripple carry counter.

```
module T_FF(q, clk, reset);
            output q;
            input clk, reset;
            wire d;
            D_FF dff0(q, d, clk, reset); //instantiate D flip flop
            not n1(d, q); //inverter
endmodule
```

Figure 4.7 Verilog description of T flipflop

```
module D_FF(q, d, clk, reset);
            output q;
            input d, clk, reset;
            reg q;
            always @(posedge reset or negedge clk)
                  if (reset)
                        q = 1'b0;
                  else
                        q = d;
endmodule
```

Figure 4.8 Verilog description of D flipflop

```
module ripple_carry_counter(q, clk, reset);
            output [3:0] q;
            input clk, reset;
            //instantiate T flipflops
            T_FF tff0(q[0], clk, reset);
            T_FF tff1(q[1], q[0], reset);
            T_FF tff2(q[2], q[1], reset);
            T_FF tff3(q[3], q[2], reset);
endmodule
```

Figure 4.9 Verilog description of 4-bit ripple carry counter.

Thus, given the Verilog description of figure 4.9, we can take up each of the instances of T flip flop, t_ff0, t_ff1, t_ff2 and t_ff3, along with the information about their internal interconnections and utilize them individually for further analyses.

4.3 Logging Input data

To log the input of each partition, we use two Verilog system tasks depending on the duration of simulation: $fdisplay and $dumpvars. We incorporate either one of these tasks in the predefined testbench for the original circuit and invoke them appropriately and Verilog writes their output to user-defined files.

$fdisplay

This system task can be used when the design runs for a relatively small number of simulation cycles. The code snippet below shows how to log the value of a signal and write it to a file.

```
integer fP;
initial
begin
       fP = $fopen( "input_log.txt" ," w" ); //open the file where
                                              //you want to write the logged input
end

//at every positive clock edge, log the values of input1 and input2.
//input1 and input2 are the example inputs of a sub-module that we are
//trying to record
always@(posedge clk)
begin
       $fdisplay(fP," %b%b" ,<input1 with hierarchy>,<input2 with hierarchy>);
end
```

Figure 4.10 Code snippet showing $fdisplay usage

$dumpvars

This system task is used to select module instances (or sub-modules) and module instance

signals and dump their values to a VCD file. A VCD (value change dump) file is an ASCII file that contains information about simulation time, scope, signal definitions and signal value changes in the simulation run. VCD files can be very handy to store signal values when the design runs for long simulation cycles or when the signal values change a lot during the simulation. An added advantage of VCD files is that post processing tools can take the VCD file as input and visually display hierarchical information, signal values and signal waveforms. Figure 4.11 shows how to use $dumpvars and other VCD associated system tasks.

```
//specify name of VCD file
initial
       $dumpfile(myfile.dmp);//simulation info dumped to myfile.dmp

initial
begin
       //dump input signals of the required module instances here
       $dumpvars(0, <input names along with their hierarchy>);
end

initial
begin
       $dumpon; //start the dump process
       #t $dumpoff; //where t is the simulation time after which we stop
                      //the dump process. In our case,t would be
                      //total simulation time
end
```

Figure 4.11 Code snippet showing $dumpvars usage

4.4 Fault Generation Phase

Fault generation in the sub-modules as well as the entire original circuit was accomplished using statistical fault injection (SFI) technique. Figure 4.12 shows the simulation approach for our SFI technique [29]. Figure 4.13 shows the flowchart of our approach.

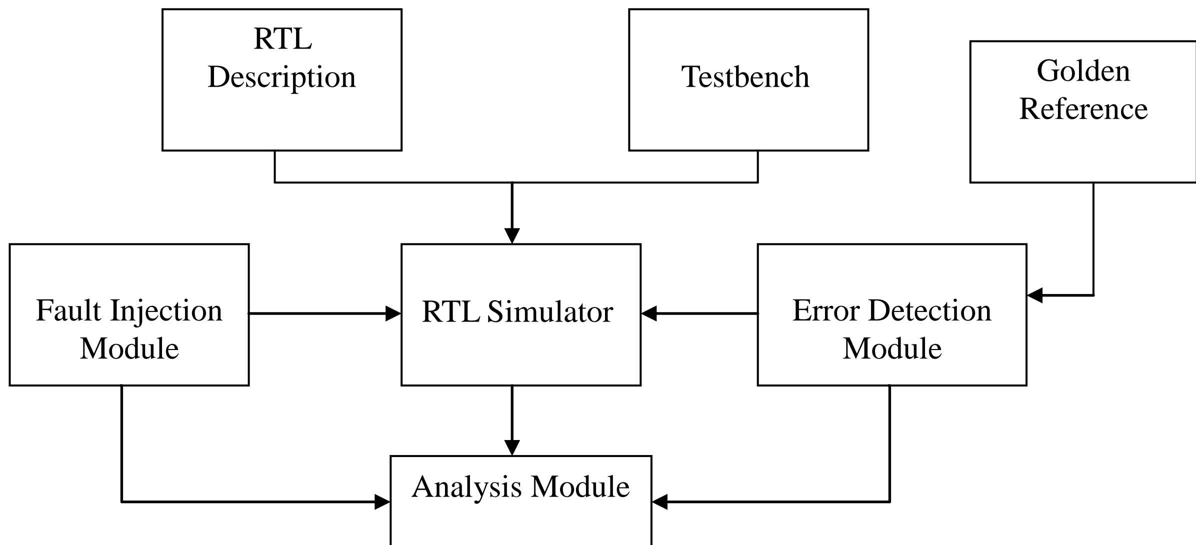

Figure 4.12 Fault generation simulation approach

We provide the RTL description of our partitions (or sub-modules) as the input source code. Using the input data stream that we had logged in the previous step, we create testbenches for each of the partitions. These testbenches propagate the logged input data stream into their respective partitions and record the output data stream as per the requirements, thereby creating a golden copy of a fault-free output data stream of the partition.

The next module in figure 4.12 is the fault injection module. Fault injection in our implementation has been accomplished using the Verilog Programming Language Interface (PLI) feature [30].

4.4.1 Verilog Programming Language Interface

PLI provides a set of interface routines that allows the designer to read internal data representation, write to internal data representation, and extract information about the simulation environment. Thus, PLI, with their predefined set of interface routines, allows the designer to customize the capability of the Verilog language by defining their own system tasks and functions. Designers use high level languages like C or C++ to write their own functions/tasks and then use these

functions/tasks (after compilation and linking) like a normal system function/tasks in their Verilog testbenches. Figure 4.14 gives a pictorial description.

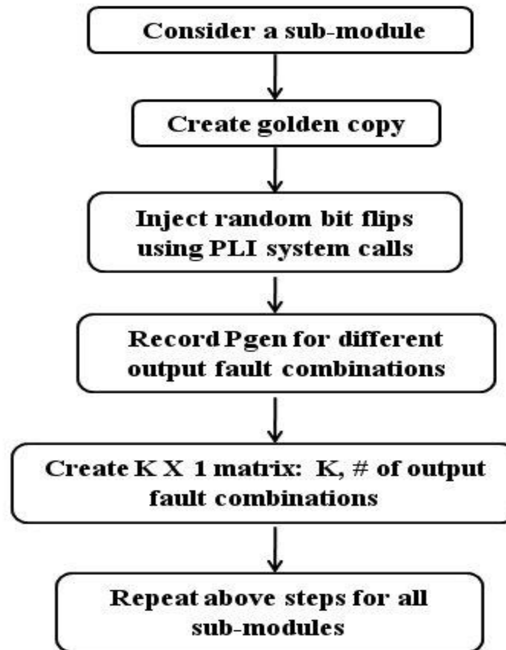

Figure 4.13 Flowchart for fault generation simulation

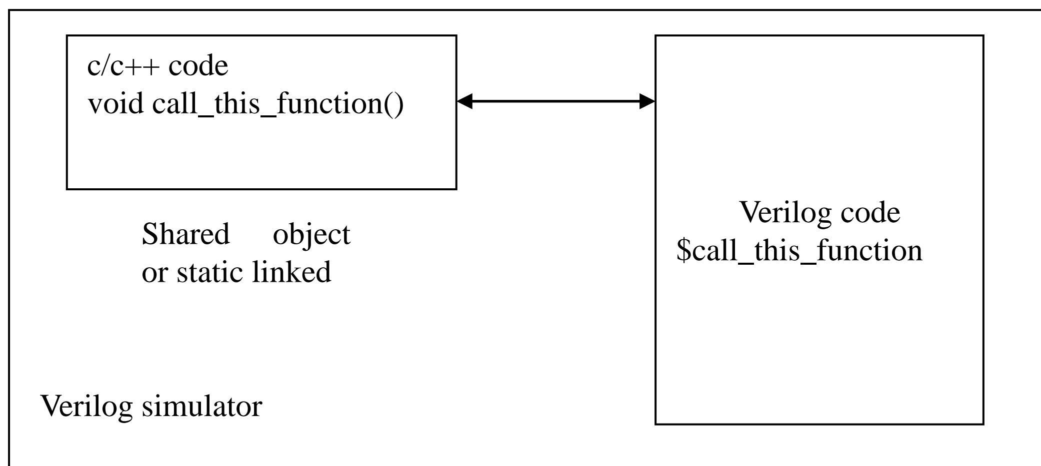

Figure 4.14 PLI function call from inside Verilog code

In this work, the PLI function for fault injection was written in C++ (and compiled using gcc) by Brian Sierawski of the Institute for Space and Defense Electronics. The PLI function selects a register bit randomly and uniformly across the sub-circuit under test and flips the stored value of the bit. If the value of the bit is undefined, it stores a random value into it. Also, the PLI function gets invoked at a random and uniform time during the simulation length of the testbench. For example, let's assume our testbench runs for 2000 time units. We invoke the fault injection PLI function at a random time between the start of simulation and 2000 time units as shown in the code snippet in figure 4.15.

```
initial
begin
        $singleEventInit();
        #($pseudoRandom(2000));
        $singleEventUpset(testbench.sub_circuit);
end
```

Figure 4.15 Fault injection PLI function example

The `$singleEventInit()` function sets up the initialization for the fault injection module as well as generates the random seed which is then used to randomly select time and location of fault injection. The `$pseudoRandom(<value>)` function is used to generate a random number between 0 and value. Using this function after '#' in the above code snippet forces the simulator to delay the execution of the next step by time units equal to the generated random number. The `$singleEventUpset(DUT)` function is used to randomly (depending on the seed) select a register bit and simulate a fault by flipping its value. The DUT argument is the hierarchical path of the sub-module we are trying to inject fault into. The output of this function, which includes bit flip location, bit flip time with respect to simulation time and the nature of the flip (e.g. 0 -> 1 or 1-> 0) is logged onto standard output.

The RTL simulator that we use to perform our simulations is Synopsys VCS. However, this

methodology is independent of RTL simulator and is supposed to work with any standard commercial RTL simulator.

The primary purpose of the error detection module is to operate at run time, log the output of the testbench and compare the output of two different testbenches ran on the same design. This module has been implemented in C and linked to Verilog using the PLI feature. In our implementation, it performs a twofold task. First, it is used to create the golden fault free copy by running it along with the testbench, but without the fault injection module. Second, it is then run on the same design along with the same testbench but this time with the fault injection module. The error detection module gets invoked every clock cycle, either positive or negative edge, and records the output of the two testbenches. Alongside, it also compares the output of the two testbenches every time it gets invoked. As soon as it finds a mismatch between the two, it writes out a message indicating about the occurrence of a fault/error to the standard output along with the real time when it occurred. It also writes out the name of the output bit(s) where the fault (or error) has occurred.

The analysis module is a post-simulation module that takes in the information generated using the fault simulation and error detection and processes it according to our requirements. In our case, it logs out all the output bit locations where faults (or errors) have occurred along with the number of times the faults (or errors) have occurred at a particular output bit. Also, it logs the various combinations of faults seen at the output. E.g. if a sub-module has a 3-bit output and bit 0 and bit 2 of the output has faults while bit 1 doesn't have a fault, then it gets recorded as ENE (for error-no error-error). The analysis module then counts the frequency of such combinations. This information is then used to create the probability of fault generation matrix (described in Chapter III) for that sub-module. The analysis module has been implemented using Shell and Python scripts. The Python script also provides information about the accuracy in our fault generation simulation results along with the precision of the results. The Python scripts then create a matrix of precisions where the elements of this

matrix are the precisions of the corresponding elements of the probability of fault generation matrix.

To determine the number of simulations to perform for the fault generation phase, we adopt a similar approach as explained in [29]. Sufficient number of simulations is performed until the AVF for the design starts to saturate as shown in figure 4.16 for an example microprocessor [29].

Once we determine the number of simulations at which the AVF starts to saturate, we perform multiple experiments with the same number of simulation runs with same experimental conditions. To find the true AVF, we take the average of the AVFs computed during each experiment. To find the precision in our computed AVF, we first find the deviations of the computed AVFs (from each experiment) from the average AVF.

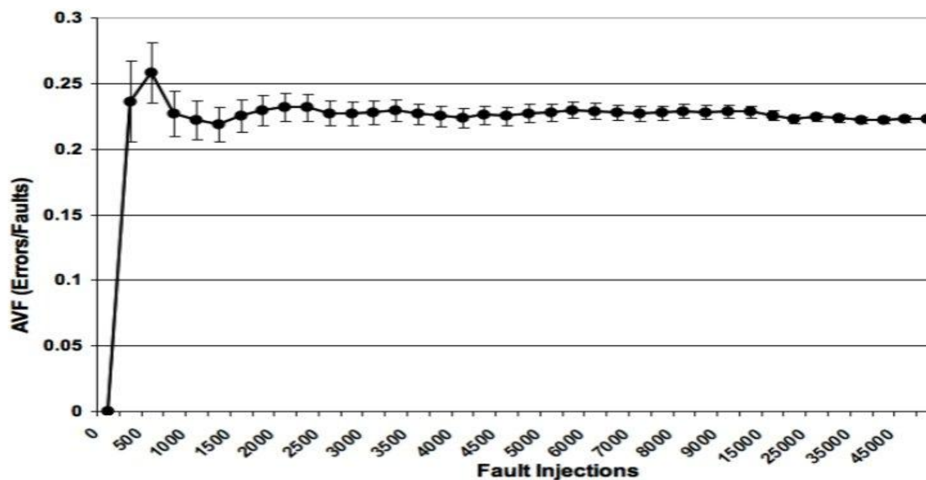

Figure 4.16 AVF vs injection for 8-bit microprocesssor

Absolute deviation = Measured AVF – Mean AVF

We then compute the average of all absolute deviations. Precision is then given by the following equation: Precision = (Average deviation / Mean AVF) X 100%

4.5 Fault Propagation Phase

We perform a second set of fault simulations where we simulate a fault in the input data stream of the RTL model of the sub-module, run the model forward and compare the output to that of a golden copy. The golden copy is a fault free simulation of the sub-module.

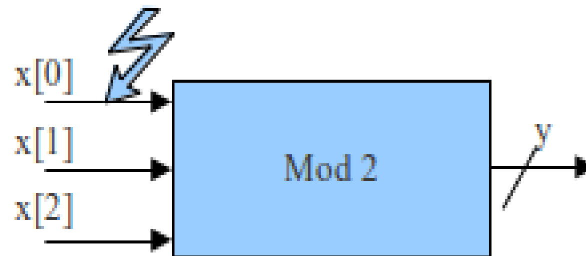

Figure 4.17 Simulation of a fault at x[0]

Figure 4.17 shows an example where the input stream at x[0] is flipped, allowing the fault to propagate and then checking the state of output stream $\vec{y}$.

The simulation approach is similar to fault generation phase except the fault injection module, which is replaced by a fault propagation module. We use the same Synopsys VCS simulator. The pseudo code below shows an example where, in a sub-circuit, it is performing 1000 fault injections at random input bits at a random simulation time.

```
do 1000:
        at rand (i/p bits, time);
             inject fault;
        compare the o/p with golden copy;
        log the o/p combinations with errors;
```

Figure 4.18 Pseudo-code for fault propagation phase

The fault propagation module uses the system task $value$plusargs to select the following three parameters:

1) Random select a time to simulate the fault at the input.
2) Randomly select the number of input bits to flip.
3) Randomly select the bit location or bit indices.

$value$plusargs provides the capability of conditional execution of the Verilog code by allowing the user to control variables in the code from outside with the help of flags which can be set on a run-time basis. Figure 4.19 shows some example code snippets:

```
//Conditional execution with $value$plusargs
module test;
          reg [8*128-1:0] test_string;
          integer clk_period;
          ...
          ...
          initial
          begin
                 if($value$plusargs("testname=%s", test_string))
                        $readmemh(test_string, vectors); //Read test vectors
                 else
                        //otherwise display error message
                        $display("Test name option not specified");

                 if($value$plusargs("clk_t=%d", clk_period))
                        forever #(clk_period/2) clk = ~clk; //Set up clock
                 else
                        //otherwise display error message
                        $display("Clock period option name not specified");
                 end

          //For example, to invoke the above options invoke simulator with
          //+testname=test1.vec +clk_t=10
          //Test name = "test1.vec" and clk_period = 10
endmodule
```

Figure 4.19 Verilog code demonstrating $value$plusargs usage

Once the fault propagation simulations are done, the analysis module (consisting of Python and Shell scripts) looks for all the mismatches detected by the error detection module and prints out the output bit locations that have errors along with the corresponding input bit locations with errors. Using the frequency of such output-input combinations, we then create the probability of fault propagation matrix (chapter III). Also, we build the corresponding matrix of precisions associated with the probability numbers. Figure 4.20 shows the flowchart for fault propagation simulation.

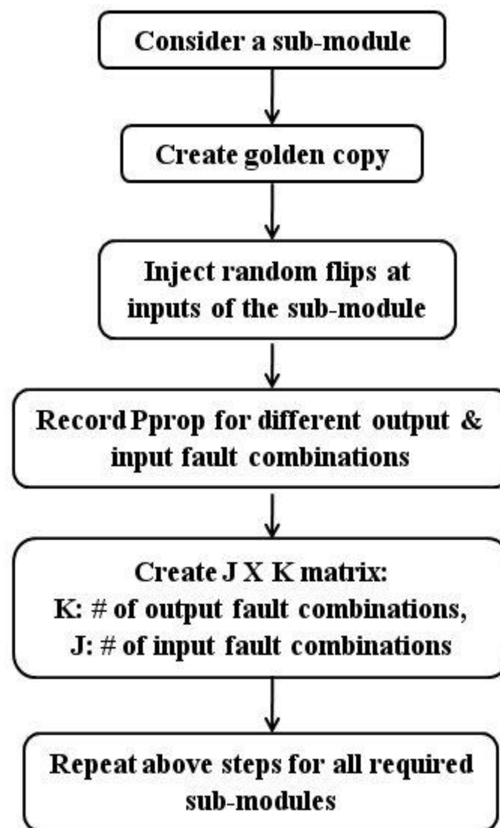

Figure 4.20 Flowchart for fault propagation simulation

4.6 Post Simulation AVF Calculation

Python scripts are used to take the two probabilities created in the previous step: probability of fault propagation and probability of fault generation and perform matrix multiplication as described in

Chapter III. The Python scripts use the open source *numpy* package to automate the multiplication process [31].

The multiplication of the standard deviation matrices is done using similar Python scripts, but the multiplication rules, apart from normal matrix multiplication, also follow the rules for propagation of standard deviations as described in [32]:

If A, B are real variables with standard deviations of σ_A and σ_B, then:

1) For F = A + B,

   Standard deviation of F is given by $\sigma^2_F = \sigma^2_A + \sigma^2_B$

2) For F = AB

   Standard deviation of F is given by $\left(\frac{\sigma_F}{F}\right)^2 = \left(\frac{\sigma_A}{A}\right)^2 + \left(\frac{\sigma_B}{B}\right)^2$

The correlation coefficient terms are set to zero (described in detail in chapter V).

CHAPTER V

APPLICATIONS

In this chapter, we present the basic criteria required to identify the class of circuits on which modular technique could be employed to estimate their AVFs. We then discuss some applications on which we had tested our technique and present their results.

5.1 Circuit Classification

Consider our example design again (figure 5.1).

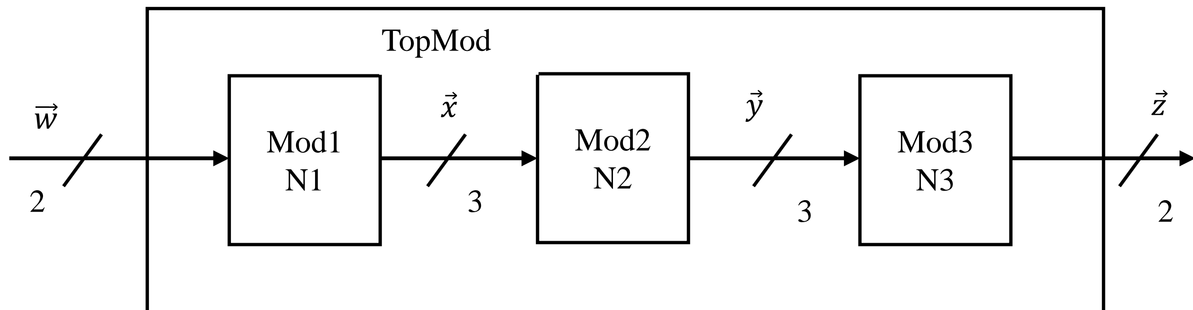

Figure 5.1 Example design for circuit classification analysis

During fault generation phase of Mod1, the probabilities of faults observed for the seven different output combinations of Mod1 are recorded. Now ideally, if Mod1 and Mod2 are assumed to be uncorrelated sub-modules, then these faults from Mod1 can be propagated through Mod2 by introducing bit flips - randomized over time - at the corresponding inputs of Mod2. Similarly, if Mod2 and Mod3 are uncorrelated, faults can be propagated through Mod3 independent of time. Under such a condition, the systematic error introduced in the AVF estimation using modular approach is going to be

ideally zero and no loss in accuracy of AVF would be introduced. Practically, modular approach can be applied on circuits where it is possible to create weakly correlated partitions.

5.1.1 Mathematical Interpretation

Let's say it takes τ time units to propagate a signal through Mod2. Hence, for a simulation period from t = 1 to t = n time units, we have:

$$Pr\{Fr(\vec{y}, i+\tau) \mid Fr(\vec{x}, i)\} = Pr\{Fr(\vec{y}, j+\tau) \mid Fr(\vec{x}, j)\}, \text{ where } 1 < i, j < n \quad \ldots (1)$$

Similarly for Mod3,

$$Pr\{Fr(\vec{y}, i+\tau') / Fr(\vec{x}, i)\} = Pr\{Fr(\vec{y}, j+\tau') / Fr(\vec{x}, j)\}, \text{ where } 1 < i, j < n \quad \ldots (2)$$

where τ' is the time required to propagate a signal through Mod3.

5.1.2 Example partitions

Example partitions that satisfy (1) or (2) include Canonical Clocked Logic Circuits (CCLC) and their variations.

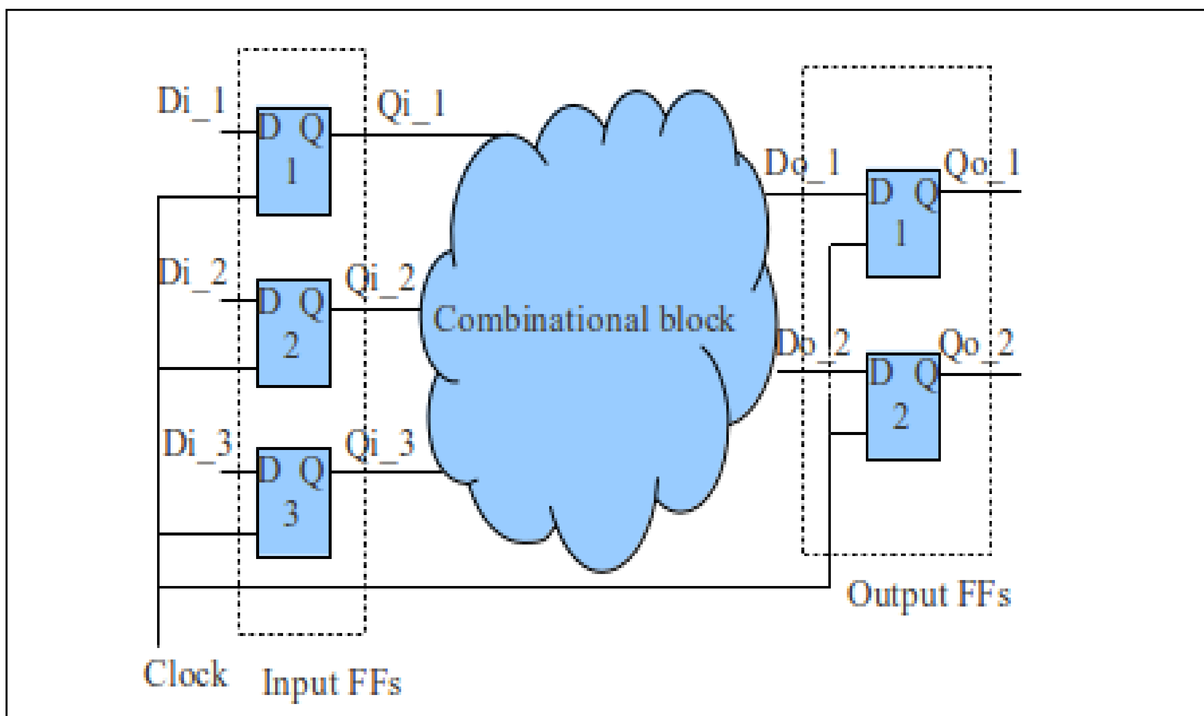

Figure 5.2 Canonical clocked logic circuit

Consider the CCLC shown in figure 5.2, composed of a combinational block with latches at its primary inputs and outputs. A chip can be considered as a network of CCLCs [33]**.** A soft error is said to have occurred in a CCLC if a DFF captures the single event transient (SET) generated by a particle hit.

The upper bound on the soft error rate, SER (number of soft errors per unit time), of a chip is given by

$$SER_{chip} \leq \sum_{k=1}^{Nc} SER_{CCLC,k} \qquad \ldots(3)$$

where Nc is the number of CCLCs on the chip. The above equation becomes tight bound when all the CCLCs are independent from each other.

Suppose that a fault is observed at the input DFFs: Di_1 and Di_3, but not in Di_2 (outputs of a previous module(s) in the propagation path) at time t = i < n. Depending on the logical state of the combinational block, the error may or may not show up at its outputs Do_1 and Do_2. Let's assume that it shows up only at Do_1 at t = i + 1(neglecting the propagation delay of the combinational block). It would then show up at Qo_1 at t = i + 2 (after one clock cycle). The operation of the combinational logic is independent of time. So, irrespective of the time it receives its input data, the combinational block is always going to produce the same output throughout the entire simulation period. Hence, the behavior of CCLC is constant for any value of *i* and hence time-independent.

Mathematically, for all t = i, 1 < i < n, in this circuit for the mentioned scenario,

$$Pr\{Er(Qo_1, i+2) \mid Er(Di_1, Di_3, i)\} = 1 \text{ and}$$

$$Pr\{Er(Qo_2, i+2) \mid Er(Di_1, Di_3, i)\} = 0 \qquad \ldots(4)$$

This is going to hold true for any variation of CCLC as well. Hence, circuits where it is possible to create such time independent CCLC or variations of CCLC partitions are going to yield accurate results with modular approach.

A second method to determine if modular approach is applicable on a circuit is through fault

propagation simulations. Let's consider that in figure 5.1, Fb represents the set of combinations in which faults occur at $\vec{x}$ (can be determined from fault generation phase of Mod1) during a simulation period from t = 1 to t = n. E.g. Fb = {Fx[1], Fx[2], Fx[3], Fx[1][2]} showing that faults occur at only x[1], only x[2], only x[3] and, x[1] and x[2] simultaneously. We pick an element of the set, say Fx[1] and propagate a fault through Mod2 by flipping only x[1] at a random time i (i < n) and then monitoring $\vec{y}$ (Mod2's output) for the rest of the simulation period. This process is repeated for different values of i: e.g. i1, i2, i3 etc. Now, for these different i's, we record the following as well:

When the fault occurs at t = i1,

$$\text{record: } \{Fr(\vec{y}, i1 + 1) \mid Fr(x[1], i1)\}, \{Fr(\vec{y}, i1 + 2) \mid Fr(x[1], i1)\} \ldots \{Fr(\vec{y}, i1 + n) \mid Fr(x[1], i1)\} \quad \ldots (5)$$

When the fault occurs at t = i2,

$$\text{record: } \{Fr(\vec{y}, i2 + 1) \mid Fr(x[1], i2)\}, \{Fr(\vec{y}, i2 + 2) \mid Fr(x[1], i2)\} \ldots \{Fr(\vec{y}, i2 + n) \mid Fr(x[1], i2)\} \quad \ldots (6)$$

Similarly, we repeat this process for other values of i.

If (5), (6) etc. follow similar pattern, it implies that fault propagation through Mod2 follow a similar pattern for faults propagating through x[1], irrespective of when it occurs during the simulation. This process should be repeated for all other combination of faults present in the set Fb. Once we ensure that all of those fault combinations propagate through Mod2 irrespective of the time they are simulated, we can be sure that the partition Mod2 agrees with our conditions (1) and (2). It is recommended to neglect i's that are either too early or too late in the simulation period.

Also, this has to be repeated for every partition through which errors can possibly propagate to make sure all of them agree with the conditions (1) and (2).

5.2 Test Circuits

In this section, we discuss two applications we tested our approach on. All the circuits are implemented in RTL Verilog and are available at [35]. All the simulations were performed using Synopsys VCS on a dual core 8GB RAM AMD Opteron processor running Redhat Linux.

5.2.1 Digital signal processing units

We tested our modular approach on three DSP units: double precision floating point unit, forward discrete cosine transform unit and a jpeg encoder unit. In this section, we will take up each of these one by one further discussion.

5.2.1.1 Double Precision Floating Point Unit (DPFPU)

Figure 5.3 shows the hierarchy of the DPFPU.

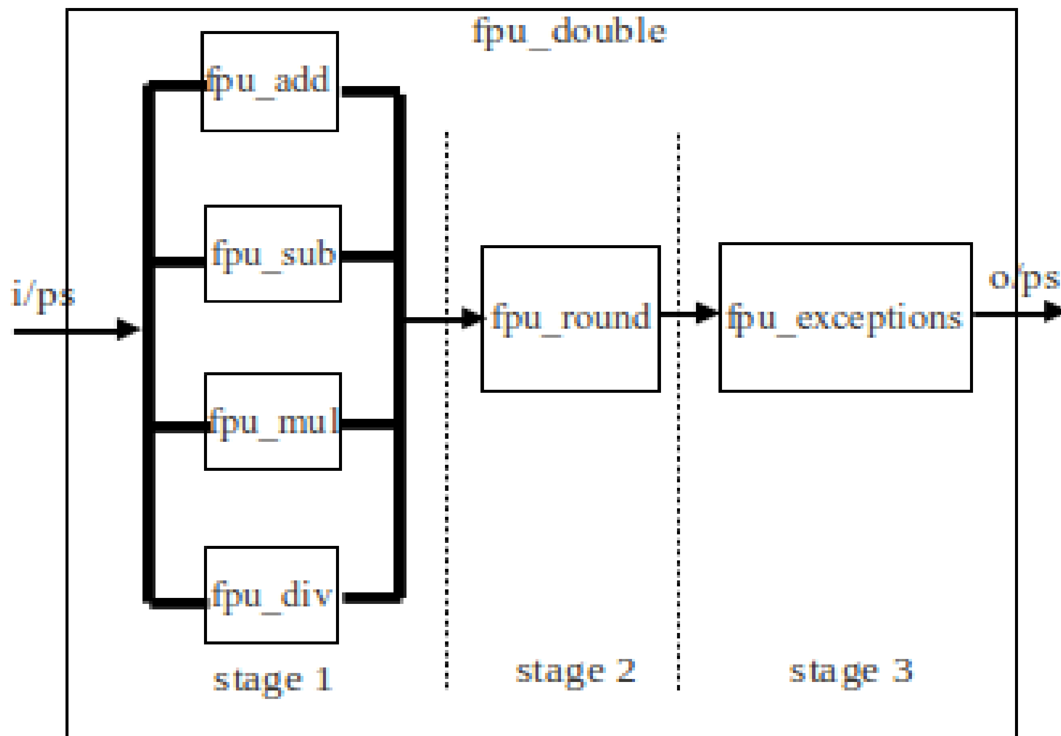

Figure 5.3. Double Precision Floating Point Unit

The DPFPU has 6 major sub-units. Hence, we partition into the corresponding 6 sub-modules. The unit can be broadly divided into three stages: arithmetic unit (consisting of fpu_add, fpu_sub, fpu_mul & fpu_div), rounding unit and the exceptions module. The inputs signals consist of two 64-bit operands, 3-bit opcode, rounding mode along with clock, reset and enable signals. The output signals primarily consist of the 64-bit output from the operation along with a ready signal that goes high when the output is available. Depending on the opcode, the appropriate arithmetic operation is performed on the operands in stage 1. The top level, fpu_double, runs a counter to count the number of clock cycles required for the specific operation that is being performed. The output signals of this stage are the three components that make up a floating point number: sign, mantissa and exponent. These are then passed to the rounding stage where rounding of the mantissa is performed, based on the four possible rounding modes (round to nearest, round to zero, round to positive infinity and round to negative infinity). The rounding unit generates the final mantissa. The 64-bit output from the rounding unit is then passed to the exceptions unit. In the exception unit, all of the special cases are checked for, and if they are found, the appropriate output is created, and the individual output signals of underflow, overflow, inexact, exception, and invalid are accordingly asserted.

Because of the nature of operation, the three stages of DPFPU operate independently. For a particular set of inputs, only one stage is active at a time, thus making the stages weakly correlated with each other. We perform SFIs on all the 6 sub-modules to find the probability of fault generation. We perform fault propagation simulations on the sub-modules fpu_round and fpu_exceptions. It is not required to perform fault propagation simulations on the stage 1 sub-modules as we use a fixed testbench assuming no errors to be propagating from the primary inputs.

To show that faults propagate through fpu_round and fpu_exceptions regardless of when they occur and hence agree with our assumption 2b, we evaluate equation (5) and (6) for these two sub-modules. This is shown in table 5.1 and 5.2. The first column shows the different times at which faults

have been injected into the input stream. The second column shows the times at which the first fault shows up in the output stream as a result of the fault propagating from the input. It can be seen that the first fault always takes the same time (w.r.t. the time when the fault is injected in the input stream) to show up in the output stream.

Time of fault injection in the input stream (in ns)	Time at which first fault was observed in the output stream (in ns)
1000	2500
3000	4500
8000	9500
12000	13500
15000	16500

Table 5.1 Table showing the time when fault is injected in input and observed at output for module fpu_round

Time of fault injection in the input stream (in ns)	Time at which first fault was observed in the output stream (in ns)
1000	2500
3000	4500
8000	9500
12000	13500
15000	16500

Table 5.2 Table showing the time when fault is injected in input and observed at output for module fpu_exceptions

Circuit	Statistical Parameters	Full Circuit SFI	Modular Approach					
Double Precision Floating point Unit	Sub-circuits		1	2	3	4	5	6
	Registers	5254	627	696	1741	1004	406	780
	Faults injected	4830	300	332	810	720	194	354
	Faults propagated		-	-	-	-	100	100
	CPU Time (in mins)	163.4	10.1	10.3	17.3	16.1	4.3	10.5
			68.6					
	AVF (%)	22.1 ±1	23.3±0.4	24.3±0.4	33.6±0.4	28.2±0.4	19.9±0.4	22.4±0.4
			21.5±1					

Table 5.3 Results table comparing SFI and our technique for DPFPU

Table 5.3 shows the results of modular approach tested on the DPFPU. Also, we ran SFI on the entire circuit for comparison purposes. The table shows the number of registers in an entire circuit as well as in every of the sub-circuits. It shows the faults injected in the entire circuit as well as in the sub-circuits. It also shows the faults propagated through different sub-circuits. The next row shows the CPU time taken to perform the fault generation and propagation simulations. The last row shows the percentage AVF of the entire circuit as well as that of the sub-circuits along with their percentage standard deviations.

It can be seen from the results table 5.3 that with our proposed approach, for the same standard deviation, we estimate the AVF within approximately 3% of that estimated using a full circuit SFI. But, in case of modular approach, the number of simulations that we need to run is 300+332+810+720+194+354+100+100 = 2910 while we have to run 4830 simulations in case of a full circuit SFI. Thus, we achieve a reduction factor of roughly 2X in terms of number of total number of

simulation runs required. Also, the total time required for our modular approach is 68.6 mins while it takes 163.4 mins in case of full circuit SFI, giving us a computational speed up of about 2.4X.

In a second set of analysis, the number of simulation runs was kept same (2910) for traditional SFI and modular approach and comparisons of speed and accuracy were performed. It was found that the AVF computed using traditional SFI is (12.2 ± 1) %. The decrease in AVF shows that because of the decreased observability of the design, less number of injected faults makes it to the primary output of the design. Thus, the number of simulations is not sufficient to make an accurate decision about the AVF. This results in a decrease in accuracy of the AVF (to 4%) estimated using modular approach. Because the number of simulations ran is the same, the speed in AVF estimation remains approximately the same for both cases.

A final analysis was done where for a given AVF (21.5% in this case), the number of simulations and the speed of experiments for the two approaches were compared. It was found that we need to perform about 1.5X times more simulations for traditional SFI than modular approach. As a result, the CPU time is about 2.1X times more in case of traditional SFI than modular approach.

5.2.1.2 Forward Discrete Cosine Transform Unit

Figure 5.6 shows the hierarchical diagram of the forward discrete cosine transform unit we tested our modular approach on.

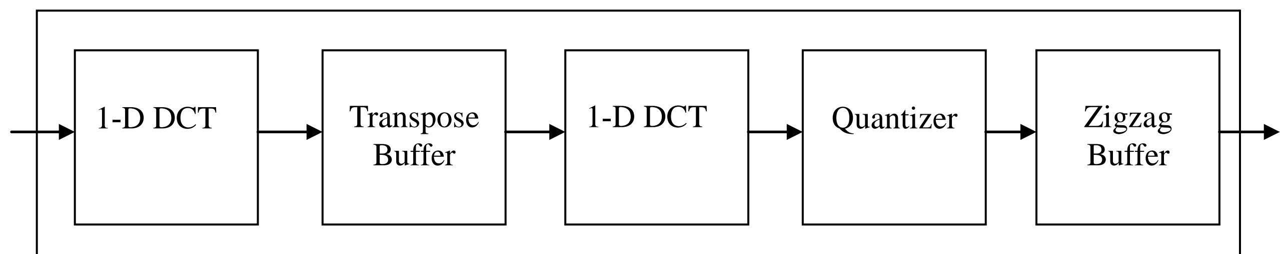

Figure 5.6 Forward DCT Unit

The DCT blocks have 8 input/output points and an 8-bit input data is inserted into the system through the first DCT in a sequential manner. As a result, it takes 8 clock cycles for each input and output process. In total, each of the 8-point 1-DCT blocks takes 22 clock cycles for computation. The first DCT block performs 1-D discrete cosine transform on row-wise input samples. Results of this computation stage are stored into the Transpose Memory. This processing stage comprises a set of multiply accumulate units as well as cosine lookup table for respective DCT computation. The 2nd DCT block performs 1-D Discrete Cosine Transform on column-wise data stored in Transpose Memory by 1st stage. More information on the DCT computation process can be found at [34].

The DCT blocks are extensions of CCLCs as shown in figure 5.7, hence they tend to agree with our conditions (1) and (2).

The transpose buffer is a static RAM, designed with two set of data and address bus, acts as a temporal barrier between the first and the second DCT. Input address is generated in normal sequence but output address is generated in transposed sequence.

The quantizer divides each and every 2DDCT coefficient by quantizing values from a quantization table. Varying levels of image compression and quality are obtainable through selection of specific quantization matrices. The quantizer module consists of ROM and divider. The quantizing values are first stored in ROM and the divider carries out division in a pipelined manner.

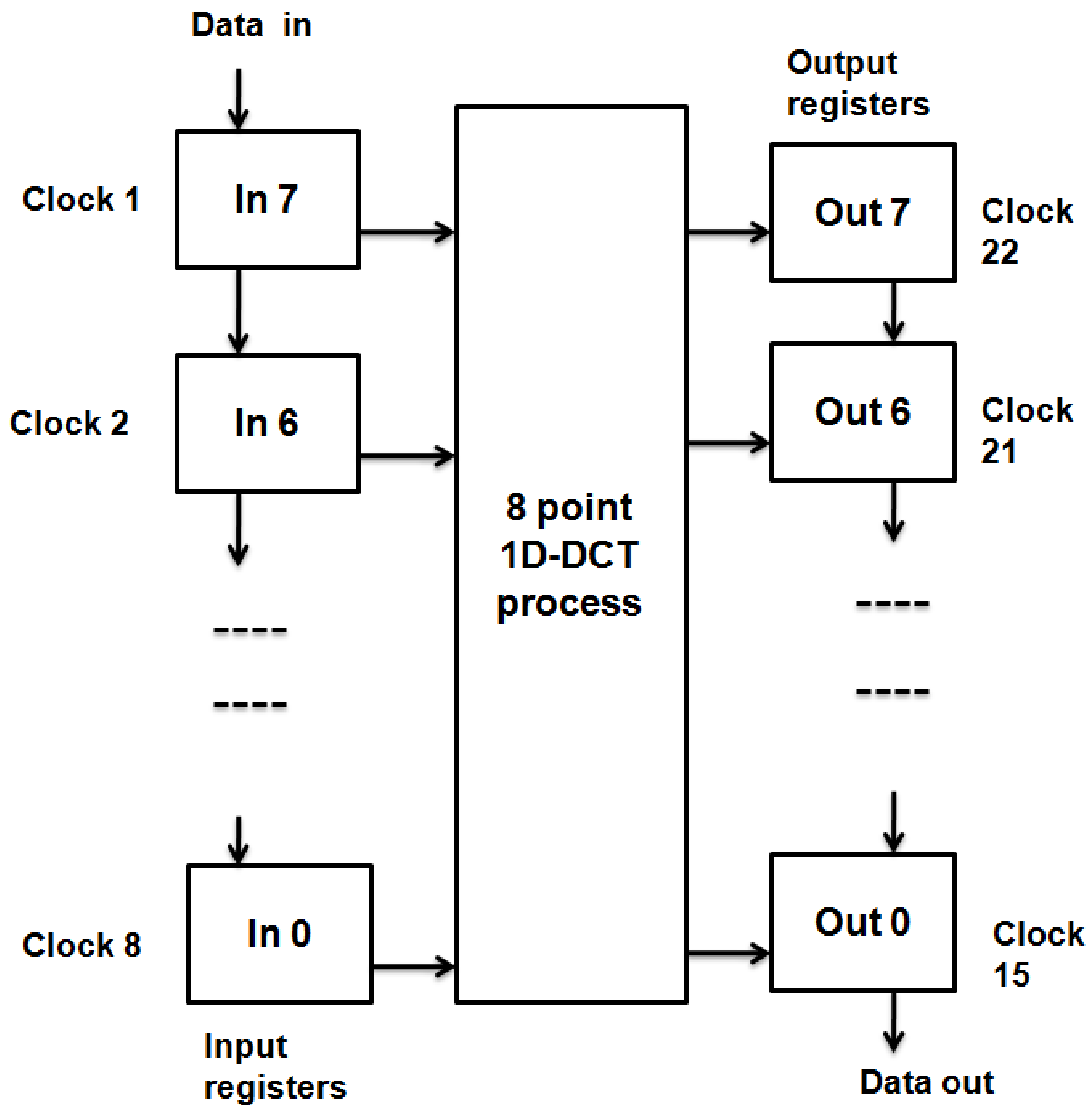

Figure 5.7 DCT unit

Zigzag buffer is made from static RAM. Its construction is similar to that of the transpose buffer. It has two sets of data-address bus. Input address bus is accessed by normal sequence, but output address is given some zigzag sequence. Zigzag address is generated by a zigzag RAM. The sequence is stored in the RAM. When the RAM address bus is accessed by normal address sequence, RAM data bus will emit zigzag value. Table 5.4 shows the results.

Circuit	Statistical Parameters	Full Circuit SFI	Modular Approach				
Forward Discrete Cosine Transform Unit	Sub-circuits		1	2	3	4	5
	Registers	3945	771	771	740	526	1106
	Faults injected	2720	210	210	190	160	350
	Faults propagated		-	50	120	100	100
	CPU Time (in mins)	116.7	6.3	6.3	6.2	6.5	8.7
			34				
	AVF (%)	17.3±2	24.3±1	23.7±1	22.9±1	10.8±1	19.2±1
			16.6±2				

Table 5.4 Table showing the results for the forward discrete cosine transform unit

It can be seen that with our proposed approach, for the same standard deviation, we estimate the AVF within approximately 4% of that estimated using a full circuit SFI. But, in case of modular approach, we achieve a reduction factor of 1.8X in terms of number of total number of simulation runs required. Also, we gain a computational speedup of 3.4X using our approach.

With the number of simulation runs kept same (1490) for traditional SFI and modular approach, it was found that the AVF computed using traditional SFI is (10.8 ± 2) %. The decrease in AVF shows that because of the decreased observability of the design, less number of injected faults makes it to the primary output of the design. Thus, the number of simulations is not sufficient to make an accurate decision about the AVF. This results in a decrease in accuracy of the AVF (to 5.4%) estimated using modular approach. Because the number of simulations ran is the same, the speed in AVF estimation remains approximately the same for both cases.

For the third analysis, for an AVF of 16.6% the number of simulations and the speed of

experiments for the two approaches were compared. It was found that we need to perform about 1.3X times more simulations for traditional SFI than modular approach. As a result, the CPU time is about 2X times more in case of traditional SFI than modular approach.

5.2.1.3 Baseline JPEG encoder

The architecture of the JPEG encoder that we tested on is shown in figure 5.3. The entire architecture is organized as a linear multistage pipeline in order to achieve high throughput. This figure reflects the sequence of computation in the JPEG Baseline process. The image to be compressed is provided as input to the system at one pixel per clock cycle rate, which is then processed by the various internal modules in a linear fashion. The compressed data is then output by the system at a variable rate depending on the amount of compression achieved.

The first module in the JPEG encoder is the DCT (discrete cosine transform) module. The 2-D DCT (8 x 8 DCT) is implemented by the row-column decomposition technique. It first computes the 1-D DCT (8 x 1 DCT) of each column of the input data matrix. After appropriate rounding or truncation, the transpose of the resulting matrix is stored in an intermediate memory. It then computes another 1-D DCT (8 x 1 DCT) of each row of the resulting matrix to yield the desired 2-D DCT. A block diagram of the design is shown in Figure 5.4.

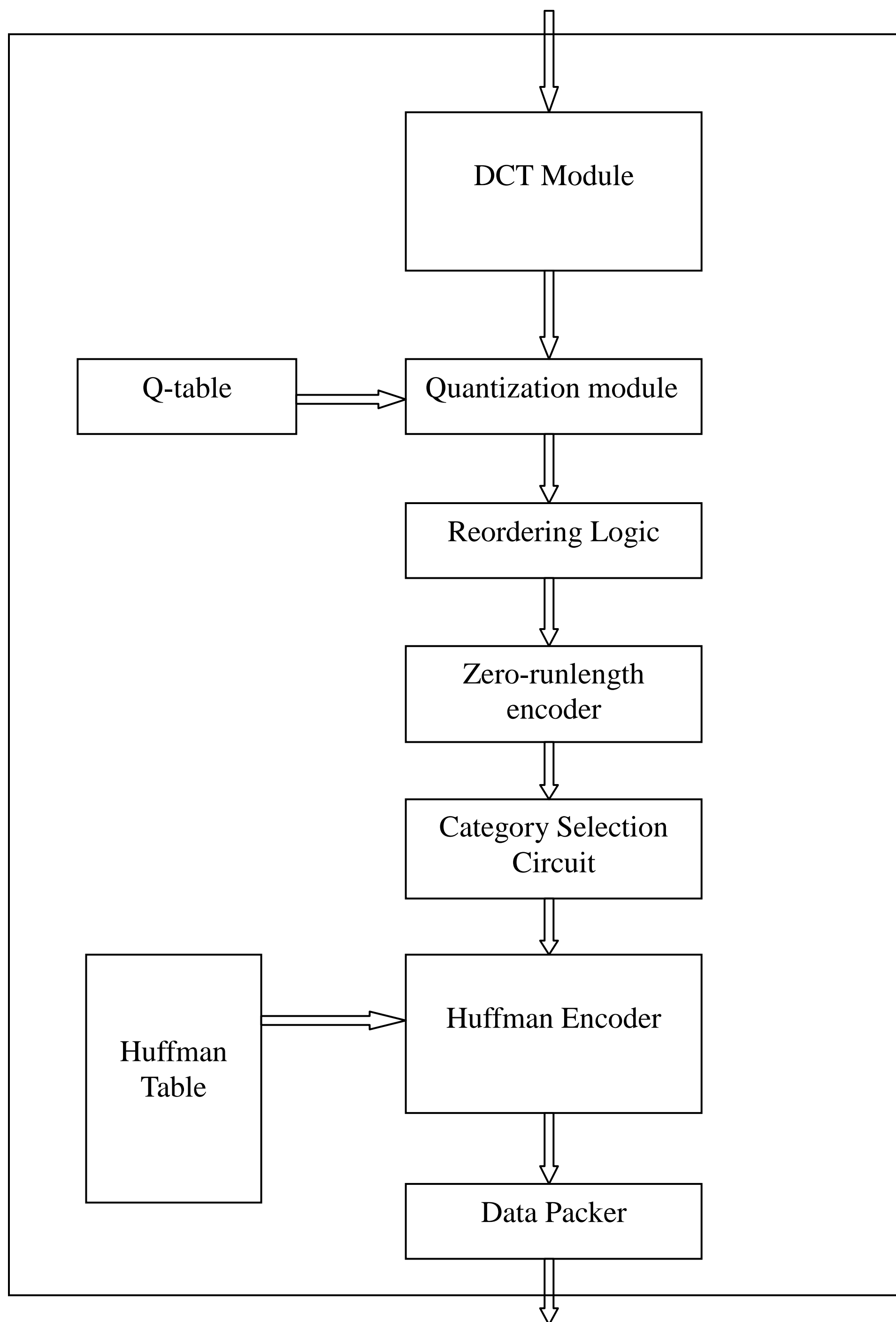

Figure 5.3 Baseline JPEG Encoder

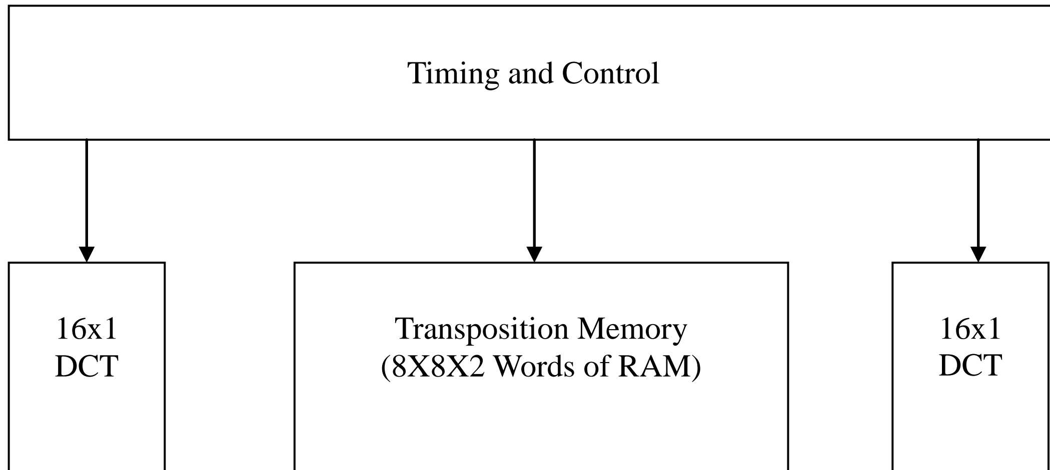

Figure 5.4 DCT module for the JPEG Encoder

The quantization module consists of a ROM to store the quantization table and an 11 x 8 bit multiplier. The quantization step in the JPEG algorithm involves multiplying the output of the DCT stage with a set of predefined values from a quantization table. The 8-bit multiplier value is retrieved from the quantization table each clock cycle, and the coefficient values from the DCT stage are input as the 12-bit multiplicands.

The reordering logic consists of a zigzag buffer that reorders each block of data that is output by the quantization module zigzag fashion before being forwarded to the entropy encoder. This reordering is achieved using an 8 x 8 array of register pairs organized in a fashion similar to the transpose buffer.
The zero-runlength coder module performs the functions as described in the earlier part of this section. The module consists of three stages and thus a latency of 3 cycles. The first stage consists of logic for computing ΔDC (the difference between the current DC coefficient and the DC coefficient of the previous block) while the second stage derives the runlength count and the third stage is used for decrementing negative coefficients.

The category selection unit associates each DC and AC coefficient with a corresponding

category depending on the magnitude of the coefficient.

The Huffman encoder consists of Huffman code tables stored in random access memory modules and logic for replacing the category, runlength count pairs with the corresponding Huffman codes.

Circuit	Statistical Parameters	Full Circuit SFI	Modular Approach						
Baseline JPEG Encoder	Sub-circuits		1	2	3	4	5	6	7
	Registers	52995	3893	3840	8394	9009	9191	4201	14467
	Faults injected	21000	240	240	770	860	860	320	1570
	Faults propagated		80	80	120	100	100	120	-
	CPU Time (in mins)	580	12.8	12.8	17.1	18.3	18.3	14.0	28.2
			121.5						
	AVF (%)	11.7±2.4	13.4±1	14.2±1	16.3±1	15.1±1	15.7±1	14±1	17.2±1
			12.4±2.4						

Table 5.5 Table showing the results for the JPEG encoder

In this case, with our proposed approach, for the same standard deviation, we estimate the AVF within approximately 6% of that estimated using a full circuit SFI. But, in case of modular approach, we achieve a reduction factor of 4X in terms of number of total number of simulation runs required. Also, we gain a computational speedup of roughly 5X using our approach. The results are shown in table 5.5.

5.2.2 Pipelined Processor Unit

We tested our modular approach on the data path of a 5-stage MIPS pipelined processor unit as well. In this unit, each stage is processing a different instruction at any given point of time, making

them operate independent of each other. Furthermore, pipelining the data path requires that values passed from one pipeline stage to the next must be placed in registers called pipelined registers**.** Any instruction is active in exactly one stage of the pipeline at a time; therefore, any actions taken on behalf of an instruction occur between a pair of pipeline registers. Hence, we can look at the activities of the pipeline by examining what has to happen on any pipeline stage. This allows us to slice the pipelined unit along the peripherals of the pipelined registers, thus partitioning them into variations of CCLCs.

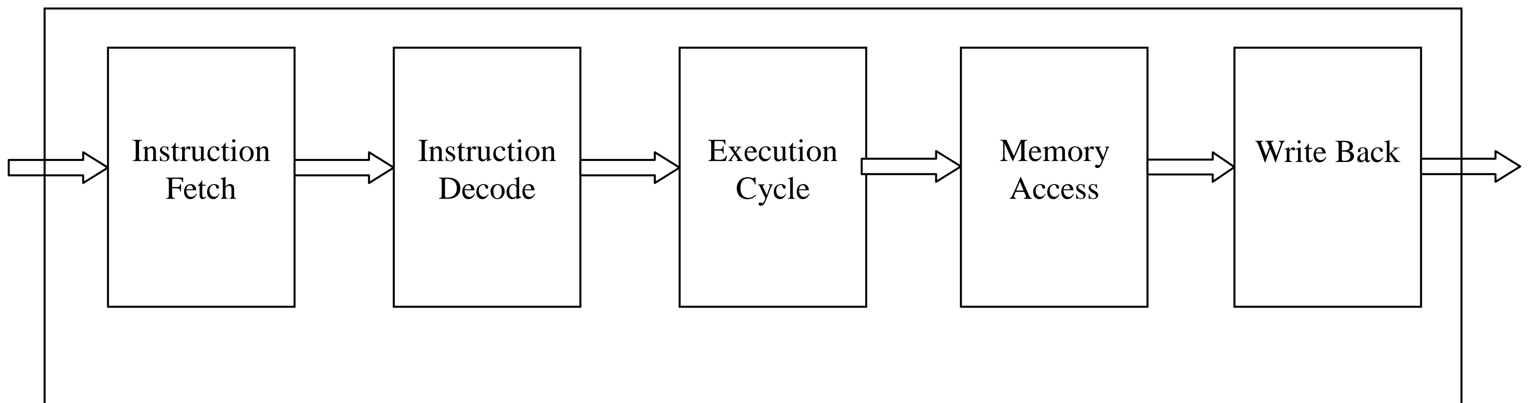

Figure 5.6 Five stage pipeline unit

Consider an example, where a fault injected into an internal register in the instruction decode stage at time t shows up as an error at the output of the decode stage at time t+1. To observe if it propagates through the next stage in the pipeline (execution stage), we need not simulate this fault at the input of the execution stage at exactly t+1, as it's a whole new instruction for it. Hence, this makes our partitions to be in conformity with equations (5) and (6).

From the results table 5.6, it can be seen that with our proposed approach, for the same standard deviation, we estimate the AVF within approximately 6% of that estimated using a full circuit SFI. But, in case of modular approach, we achieve a reduction factor of 2.8X in terms of number of total number of simulation runs required. Also, we gain a computational speed of 3.7X using our approach.

Circuit	Statistical Parameters	Full Circuit SFI	Modular Approach				
Pipelined processor Unit	Sub-circuits		1	2	3	4	5
	Registers	4420	1131	936	1255	721	377
	Faults injected	6970	510	420	625	370	190
	Faults propagated			100	90	90	90
	CPU Time (in mins)	230.2	14.6	13.4	16.1	10.5	8.5
			30.9				
	AVF (%)	24.3±3	26.2±2	25.2±2	28.1±2	22.4±2	25.1±2
			22.8±3				

Table 5.6 Table showing the results for the pipeline unit

With the number of simulation runs kept same (2485) for traditional SFI and modular approach, it was found that the AVF computed using traditional SFI is (12.8 ± 3) %. The decrease in AVF shows that because of the decreased observability of the design, less number of injected faults makes it to the primary output of the design. Thus, the number of simulations is not sufficient to make an accurate decision about the AVF. This results in a decrease in accuracy of the AVF (to 7.8%) estimated using modular approach. Because the number of simulations ran is the same, the speed in AVF estimation remains approximately the same for both cases.

For the third analysis, for an AVF of 22.8% the number of simulations and the speed of experiments for the two approaches were compared. It was found that we need to perform about 2.1X times more simulations for traditional SFI than modular approach. As a result, the CPU time is about 3.2X times more in case of traditional SFI than modular approach.

CHAPTER V

CONCLUSION

This thesis presented a novel technique for efficient fault simulations that can further be used to assess the soft error vulnerability of a design. Fault simulation efficiency is accomplished by enhancing node observability using circuit partitioning. Detailed analysis of creating optimum size partitions was also done. This work also identified the class of circuits on which this methodology can be used efficiently for AVF estimation. Experiments performed on four test circuits show that our technique works reasonably well on them and we achieve benefits in terms of simulation speed and number of simulations runs. This methodology can be used not only for commercial applications but for space applications as well.

To further validate the approach, hardware radiation experiments could be performed and the results could be compared with that of our simulation results. This work could also be extended to include analogue effects like SET propagation etc. This would require synthesizing the test circuit, targeted to a technology library, and generating its gate-level description.

REFERENCES

[1] R. Baumann, “Soft errors in advanced computer systems,” IEEE D&T, 22 (3), May 2005.

[2] P. Shivakumar, et al., “Modeling the effect of technology trends on the soft error rate of combinational logic,” DSN 2002.

[3] T. Karnik, et al., “Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18u,” VLSI Symposium, 2001.

[4] Xiaodong Li , et. al, “SoftArch: An Architecture-Level Tool for Modeling and Analyzing Soft Errors,” Proc. International Conference on Dependable Systems and Networks, 2005. DSN 2005.

[5] N. Wang et al. Characterizing the Effects of Transient Faults on a Modern High-Performance Processor Pipeline. In Proc. Intl. Conf. on Dependable Systems and Networks, 2004.

[6] C. Weaver et al. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In Proc. 31st Intl. Symp. on Computer Architecture, 2004.

[7] J. R. Srour, J. M. McGarrity, "Radiation effects on microelectronics in space", Proceedings of the IEEE, vol. 76, pp. 1443-1469, 1988.

[8] M. Santarini, "Cosmic radiation comes to ASIC and SOC design", EDN, 2005.

[9] E. Normand, “Single event upset at ground level,” IEEE Trans. Nucl. Sci., vol. 43, pp. 2742–2750, Dec. 1996.

[10] IEEE NSREC Short Course 1993

[11] T. C. May and M. H. Woods, “Alpha-particle-induced soft errors in dynamic memories”, IEEE Trans. Electron. Devices, vol. 26, pp. 2-9, Feb. 1979.

[12] P. E. Dodd, L. W. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics", IEEE Transactions on Nuclear Science, vol.50, no.3, pp. 583-602, June 2003.

[13] J. Barak, J. Levinson, M. Victoria, and W. Hajdas, “Direct processes in the energy deposition of protons in silicon,” IEEE Trans. Nucl. Sci., vol. 43, pp. 2820–2826, Dec. 1996.

[14] S. Duzellier, R. Ecoffet, D. Falguère, T. Nuns, L. Guibert, W. Hajdas, and M. C. Calvet, “Low energy proton induced SEE in memories,” IEEE Trans. Nucl. Sci., vol. 44, pp. 2306–2310, Dec. 1997.

[15] E. Petersen, “Soft errors due to protons in the radiation belt,” IEEE Trans. Nucl. Sci., vol. 28, pp. 3981–3986, Dec. 1981.

[16] F. Wrobel, J.-M. Palau, M. C. Calvet, O. Bersillon, and H. Duarte, “Incidence of multi-particle events on soft error rates caused by n-Si nuclear reactions,” IEEE Trans. Nucl. Sci., vol. 47, pp. 2580–2585, Dec. 2000.

[17] C. M. Hsieh, P. C. Murley, and R. R. O’Brien, “A field-funneling effect on the collection of alpha-particle-generated carriers in silicon devices,” IEEE Electron. Device Lett., vol. 2, pp. 103–105, Dec. 1981.

[18] C. M. Hsieh, P. C. Murley, and R. R. O’Brien, “Collection of charge from alpha-particle tracks in silicon devices,” IEEE Trans. Electron. Devices, vol. 30, pp. 686–693, Dec. 1983

[19] L. W. Massengill, “Cosmic and terrestrial single-event radiation effects in dynamic random access memories,” IEEE Trans. Nucl. Sci., vol. 43, pp. 576–593, Apr. 1996.

[20] P. E. Dodd, F. W. Sexton, G. L. Hash, M. R. Shaneyfelt, B. L. Draper, A. J. Farino, and R. S. Flores, “Impact of technology trends on SEU in CMOS SRAMs,” IEEE Trans. Nucl. Sci., vol. 43, pp. 2797–2804, Dec. 1996.

[21] H. T. Weaver, “Soft error stability of p-well versus n-well CMOS latches derived from 2D, transient simulations,” in IEDM Tech. Dig., 1988, pp. 512–515.

[22] Jie Meng MS Thesis, Vanderbilt University, May 2000

[23] L. W. Massengill et. al., “Analysis of Single-Event Effects in Combinational Logic – Simulation of the AM2901 Bitslice Processor”, IEEE Trans. On Nucl. Science, vol. 47, no. 6, Dec 2000

[24] Mark Zwolinksi, Digital Systems Design using VHDL

[25] J. L. Hannessy and D. A. Patterson, Computer Architecture: A quantitative approach

[26] S.S. Mukherjee, et al., “The soft error problem: an architectural perspective,” Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11 2005), pp. 243-247, Feb. 2005.

[27] S.S. Mukherjee, et al., “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor,” 36th Annual International Symposium on Microarchitecture (MICRO), December 2003.

[28] Samir Palnitkar, Verilog HDL: A guide to digital design and synthesis

[29] Corey Toomey, MS Thesis, Vanderbilt University, 2010

[30] Stuart Sutherland, The Verilog PLI Handbook

[31] Python scientific numpy package: http://numpy.scipy.org/

[32] Propagation of standard deviation: http://en.wikipedia.org/wiki/Propagation_of_uncertainty

[33] Ming Zhang and Naresh R. Shanhhag , " A soft error rate analysis (SERA) methodology", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

[34] Ming-tung-sun et. al., "VLSI implementation of a 16X16 discrete cosine transform", IEEE transactions on circuits and systems, vol. 36, no. 4, April 1989

[35] All test circuits source codes available at: www.opencores.org