

FPGA-based Switch-level Fault Emulation using Modular-based Dynamic Partial Reconfiguration

QA
76-9
F38
L44
2006

By
Ming-Han Peter Lee
Bachelor of Applied Science
University of Toronto
Toronto, Ontario, Canada, 2004

A project report
presented to Ryerson University
in partial fulfillment of the
requirements for the degree of
Master of Engineering
in the program of
Electrical and Computer Engineering

Toronto, Ontario, Canada, 2006/05/09

©Ming-Han Peter Lee 2006

PROPERTY OF
RYERSON UNIVERSITY LIBRARY

UMI Number: EC53517

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.




UMI Microform EC53517
Copyright 2009 by ProQuest LLC
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Author's Declaration


I hereby declare that I am the sole author of this project report. I authorize Ryerson University to disclose this thesis to other institutions or individuals for the purpose of scholarly research.

Signature



I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for purpose of scholarly research.

Signature



ABSTRACT

FPGA-based Switch-level Fault Emulation using Modular-based Dynamic Partial Reconfiguration

Ming-Han Peter Lee, Master of Engineering, 2006

Project Directed By: Dr. Reza Sedaghat,
Department of Electrical and Computer Engineering, Ryerson University

Fault simulation is a process of purposely injecting faults into a target circuit and observing a circuit's behavior in the presence of faulty logic. This observation helps designers to implement certain fault tolerance schemes thereby combating hardware failures. Fault simulation in most implementations has until now been software-based. Several fault emulation approaches have been proposed to accelerate fault simulation process using FPGA. There are generally two types of hardware fault injection: injector-based and reconfiguration-based. Injector-based methods require inserting fault injector circuitry into the circuit under test thus adding hardware overhead. On the other hand, reconfigurable-based methods require much less hardware overhead. However, these methods may be very slow because reconfiguring an entire FPGA device can take several seconds. This long configuration time is usually the bottleneck of the emulation system.

This project proposes a novel switch-level fault emulation system utilizing FPGA modular-based dynamic partial reconfiguration (DPR). In the proposed approach, faults are modeled at switch-level for an accurate fault list and mapped to gate-level for efficient synthesis. In addition, circuit-under-test is partitioned using an unbalanced tree structure to facilitate modular-based DPR. Modular-based DPR partitions a design into modules, and each module can be reconfigured independently without shutting down the FPGA. This capability is applied to fault injection where each circuit partition can be reconfigured individually without erasing the rest of FPGA. First a partial configuration bitstream representing the faulty partition is created. Fault injection can then be performed by downloading only this partial bitstream to FPGA, thereby eliminating the need for full-device reconfiguration and therefore reducing fault emulation runtime.

This report presents both a theoretical explanation and the implementation details regarding this approach. Experimental results are also be provided.

Acknowledgment

I would like to thank Dr. Sedaghat and OPR-AL lab for their contribution and constant support.

I would like to express my gratitude to my mother and my two aunts who provide me with valuable guidance in life. I would also like to thank my brothers and sisters from C.E.A.C.T. fellowship for their encouragement and prayers. Last but not the least, a sincere thank-you from the bottom of my heart to my true love, Rachel. Without her kind care and patience, I would never have gotten to where I am today.

Table of Contents

Chapter 1. Introduction	1
1.1 Background	2
1.2 Summary of Contribution	4
1.3 Report Outline	5
 Chapter 2. Switch-level Fault Modeling	 6
2.1 Switch-level to Gate-level Fault Mapping	6
 Chapter 3. Unbalanced Circuit Partitioning	 11
 Chapter 4. Dynamic Partial Reconfiguration	 14
4.1 Difference-based DPR Flow	14
4.2 Modular-based DPR Flow	14
4.2.1 Module Synthesis Phase	15
4.2.2 Initial Budgeting Phase	17
4.2.3 Active Module Implementation Phase	19
4.2.4 Final Assembly Phase	20
4.3 Bus Macro	21
4.3.1 Bus Macro Placement	21
4.3.2 Bus Macro Usage	23
 Chapter 5. Fault Emulation System	 26
5.1 Fault Emulation Controller	26
5.1.1 Hardware Overview	26
5.1.2 Software Overview	27
5.1.3 Microblaze Processor System	28
5.1.4 Hardware Setup	28
5.2 Test Pattern Generator	31
5.3 Interface Logic Core	32
5.4 Internal Configuration Access Port	33
5.5 Serial Communication Controller	34
5.6 Final System Integration	35
5.7 Inject Faults in Other Partitions	37
5.8 Programming the FPGA with Partial Bitstreams	37

Chapter 6. Experimental Results	38
6.1 Switch-level Fault Injection Result	38
6.2 Linear Growth of Fault Injection Runtime	40
6.3 Switch-level Fault Coverage	42
6.4 Runtime Prediction for Large Circuits.....	42
 Chapter 7. Conclusion.....	 47
7.1 Project Contribution.....	47
7.2 Future Work.....	48
 References	 49

List of Tables

Table 2.1	Inverter Short Faults
Table 2.2	Inverter Open Faults
Table 2.3	Inverter I/O Stuck-At Faults
Table 6.1	ISCAS-89 Benchmark Circuits (C: Combinational, S: Sequential)
Table 6.2	Fault Injection (FI) Time
Table 6.3	Fault Emulation Result

List of Figures

Figure 1.1	Fault Injection inside NAND Gate Issue after Optimization
Figure 2.1	CMOS Inverter
Figure 2.2	Stuck-At Fault: Input is stuck-at-0
Figure 2.3	Short Fault: Input is shorted to Output
Figure 2.4	Open Fault: S1 is open
Figure 2.5	Inject a fault into s298 (Inverter G131 pMOS Source is Open)
Figure 3.1	Full Binary Tree Circuit Partitioning (L: Left, R: Right)
Figure 3.2	Unbalanced Partitioning Graph (DP: LLR)
Figure 4.1	Unbalanced Partitioning FPGA Implementation
Figure 4.2	XST Script for Synthesis (xst.scr)
Figure 4.3	Pin Assignment and Clock Constraints
Figure 4.4	Module Area and Bus Macro Location Constraints
Figure 4.5	Bus Macro Schematic [8]
Figure 4.6	Bus Macro Used for Intermodule Signals
Figure 4.7	Bus Macro Placement
Figure 4.8	Examples of Bus Macro Placement
Figure 4.9	Bus Macro Declaration in VHDL
Figure 4.10	Bus Macro Port Mapping Example
Figure 4.11	Bus Macro Schematic with I/O Signals
Figure 5.1	Emulation Controller Module Overview
Figure 5.2	Fault Emulation Program Flow
Figure 5.3	Base System Builder Wizard
Figure 5.4	Add/Edit Hardware Platform Specifications
Figure 5.5	Address Assignment for Various Components
Figure 5.6	Peripheral Block Diagram (PBD) view generated by Xilinx XPS
Figure 5.7	Important Interface Core Port List
Figure 5.8	Software Code to perform Dynamic Partial Reconfiguration
Figure 5.9	Microblaze OS Parameter Setup
Figure 5.10	Block Diagram for the Final System
Figure 5.11	(a) Initial Floorplan (b) Final Assembled System for s5378
Figure 6.1	Fault Injection Time vs. Circuit Size
Figure 6.2	Dynamic partition's size grows linearly with circuit size
Figure 6.3	Fault Emulation Time vs. Number of Test Patterns for s13207
Figure 6.4	Emulation time vs. Circuit size for large number of Test Patterns
Figure 6.5	Emulation Time vs. Circuit Size

Nomenclature

B	Number of bits in the partial bitstream
C	system clock speed
D ₁	Inverter pMOS drain
D ₂	Inverter pMOS drain
F	Fault location
i	i th level in the tree
I	Input
I ₁	First input of a two-input logic gate
I ₂	Second input of a two-input logic gate
M1	Inverter pMOS transistor
M2	Inverter nMOS transistor
n	total number of faults to inject
n _g	Number of gates
N _{PO}	Number of primary outputs
O	Output
p _i	Number of test patterns needed to detect the ith fault
p _{MAX}	Number of test patterns allowed to run
S ₁	Inverter pMOS source
S ₂	Inverter nMOS source
T _{EC}	Time for processing each emulation result chunk
T _{FE1}	Time to perform fault emulation for one test pattern
T _{FR}	Time to perform full-device reconfiguration
T _{GE1}	Time to perform good emulation for one test pattern
T _{INIT}	Initialization time for the system and components
T _{OH}	System overhead
T _P	Time to process each configuration data chunk
T _{PR}	Time to perform partial reconfiguration
W _{Bus}	Width of the system bus, size of data chunk
W _C	Width of configuration medium
x	NAND gate output
y	Inverter output
z	AND gate output
Z	High impedance

Chapter 1

Introduction

The rapid advancement of CMOS process technology enables designing larger and more complex digital systems. To ensure such complex designs work flawlessly is one of the main challenges for both researchers and manufacturers. Fault simulation is a process of purposely injecting faults into a fault-free circuit to observe its faulty behavior. The fault simulation results allow circuit designers to implement possible fault tolerance and repairing schemes. Most of the traditional fault simulators are software-based. The drawback of software simulators is that output evaluation is usually computed serially. For a large and complex circuit, such serial computation may result in exponentially increasing runtime [19].

Fault emulation is fault simulation implemented in hardware [18]. Because circuit elements in hardware are evaluated in parallel, the exponential growth in runtime can be avoided. Recently, FPGA has been utilized as an efficient, low-cost platform for fault emulation [17]. In these approaches, faults are injected by the means of fault injector hardware activation [21,22,24] or by FPGA reconfiguration [1,10]. Inserting extra fault injector hardware increases both the cost and design complexity. On the other hand, reconfiguring an entire FPGA may take a long time (up to several seconds) and becomes the bottleneck of the emulation process.

Moreover, most of the existing fault injection approaches are performed at gate-level or at lookup-table level. Faults modeled at these levels do not represent realistic and complete faults. In a real hardware environment, faults can occur inside a logic gate at the transistor-level. These faults are impossible to include in a gate-level fault list. For lookup-table level fault injection [1,10] the fault list may also be incomplete due to the loss of fault locations. Original circuit structure is usually modified by the optimization performed by the tool during technology mapping prior to fault injection. This causes inaccessibility of certain fault locations in a circuit. This issue will be discussed in the next section.

In this report, a novel method is developed to emulate switch-level faults using FPGA and to perform fast fault injection utilizing modular-based dynamic partial reconfiguration. Faults are modeled at switch-level to represent a more realistic and accurate fault list in comparison to traditional gate-level fault model [25]. However,

this switch-level fault list needs to be mapped to gate-level description so that it can be efficiently synthesized and implemented on FPGA. Generally, FPGA-based fault injection runtime mainly depends on the following factors:

1. The speed of reconfiguring medium (number of bits transferred per second)
2. Speed of fault injection controller (PC, on-chip or off-chip hardware, etc.)
3. Configuration bitstream size (number of bits)

Since the first two factors are technology dependent, having better equipment will result in faster fault injection. Therefore, this project is focused on reducing the total configuration bitstream size. To accomplish this, a circuit-under-test is first partitioned into an unbalanced structure such that faults are only injected into a small sub-circuit. This results in significant resynthesis time reduction because only a small faulty partition needs to be synthesized for each fault. Next, partial bitstreams of these individually synthesized partitions are generated using modular-based dynamic partial reconfiguration flow. Each of these partial bitstreams represents the same circuit partition with different faults injected. As a result, fault injection is accomplished by downloading only the partial bitstream into FPGA. This fault injection method requires much less time because for each fault injection only a small portion of the FPGA is reconfigured.

The following section discusses the requirements for switch-level fault emulation and the issues of the existing switch-level fault injection schemes.

1.1 Background

Traditional FPGA-based fault emulation approaches emulate faults at the gate level. These approaches involve gate-level fault injection by means of hardware description language (HDL) modification. However, faults modeled at gate-level are not as realistic as switch-level faults because in real hardware, circuit outputs are evaluated by the switching of transistors inside logic gates. Moreover, a switch-level fault list is usually more accurate and thorough in comparison to a gate-level fault list because each CMOS logic gate contains more than one switch, and each switch (or transistor) has several locations where faults could occur. As a result, switch-level fault list is usually larger than gate-level. In this project, switch-level fault model is considered.

FPGA-based switch-level fault emulation has already been proposed in [25] to improve fault emulation speed. This approach involves inserting extra hardware into the circuit-under-test as fault injectors. This not only increases hardware overhead in

the system, but also the design complexity of the configuration controller. The author in [25] foresees the possibility for improvement if partial reconfiguration is utilized.

Dynamic Partial Reconfiguration has already been exploited as an efficient FPGA fault injection technique in [1,10,12,13,14]. These approaches involve injecting faults directly into the fault-free bitstream by using Xilinx JBit [2] software tool. Bitstream content is directly modified without going back to the top of design flow in which synthesis and routing are performed. Therefore, this type of fault injection scheme injects faults at the bitstream level, which can also be called the lookup-table (LUT) level. In SRAM-based FPGA, logic functions are implemented by the use of lookup tables (LUT), whose entries are set by configuration bitstream. By directly changing the content of lookup tables, faults can be injected with very fast speed since this process does not involve resynthesis and rerouting. However, logic functions implemented in LUTs may not reflect the original circuit structure due to optimization done by the tools at synthesis and technology mapping stage [3]. A synthesis tool usually tries to optimize the original circuit into a different structure with equivalent logic function in order to reduce resource usage. This is a problem for switch-level fault injection because transistor-level faults are modeled according to the original circuit description. For example, consider the circuit in Figure 1.1.

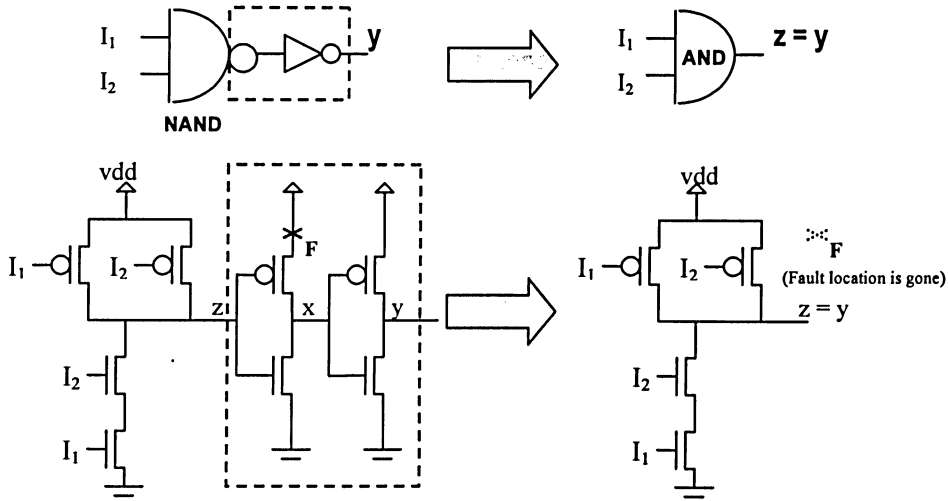


Figure 1.1 Fault Injection inside NAND Gate Issue after Optimization

The original circuit contains a NAND gate followed by an Inverter (NOT gate). A switch-level fault is to be injected at location F. If a fault is injected after synthesis and technology mapping, the circuit might be optimized into a lookup-table that implements the function of an AND gate since both circuits are logically equivalent. As a result, the target fault location F is lost when part of the circuit is removed (the area enclosed by the dash line). Therefore, some of the fault locations may become

inaccessible when faults are injected into lookup-tables after technology mapping optimization. Furthermore, sometimes more than one LUT needs to be modified because the same component may be replicated during technology mapping to reduce routing cost [9]. As a result, the fault injection process can become very complex.

In order to inject faults into the original circuit structure before being altered by all sorts of optimization processes, faults should be injected at the hardware description (HDL) level. However, this requires a complete circuit description to be synthesized, routed and programmed into the FPGA. These steps significantly increase the total emulation time. For instance, reconfiguring an entire Virtex-II device requires about 0.5 seconds, whereas the time to emulate one fault is in the neighborhood of a few microseconds. It is obvious that fault injection time is the major speed bottleneck for approaches where full-device reconfiguration is required for each fault injection. A solution was proposed to inject multiple independent faults together and thus reduce the total number of reconfigurations required [5]. However, this approach requires extra fault injection circuitry and control pins to active faults in sequence. In addition, even though the number of reconfigurations is reduced, for each reconfiguration there are still unnecessary configuration bits being downloaded to the FPGA.

In the proposed approach, modular-based dynamic partial reconfiguration is exploited to provide significant speedup without the need for extra fault injector hardware overhead and control pins. In summary, the key difference between the proposed approach and the existing ones is that faults are injected and emulated at switch-level instead of lookup-table level. In addition, fault injection time and hardware overhead is significantly reduced by reconfiguring only the necessary circuit partition. This report will provide both theoretical explanation and the implementation details for this approach.

1.2 Summary of Contribution

This project contributes to the following areas:

- Conducting research on dynamic partial reconfiguration and its implementation methodology
- Implementing Modular-based Dynamic Partial Reconfiguration on FPGA for all the selected benchmark circuits listed in Chapter 6
- Writing scripts and Java programs to automate Modular-based Dynamic Partial Reconfiguration flow
- Creating template files for automatic bus macro generation and port mapping

- Designing the configuration controller using a Microblaze processor
- Designing pseudo-random input generator and necessary interface logic
- Collecting fault injection time and fault coverage data for analysis

1.3 Report Outline

Switch-level fault modeling is discussed in Chapter 2. Chapter 3 presents the Unbalanced Partitioning scheme, its advantage, and how it applies to the proposed fault injection campaign. Dynamic Partial Reconfiguration flow is explained in Chapter 4. In Chapter 5, the proposed fault emulation system is presented. All system components are also individually introduced. Then, fault injection time and emulation results are provided in Chapter 6 along with analysis. Chapter 7 concludes this paper.

Chapter 2

Switch-level Fault Modeling

Switch-level faults are faults that occur in transistors inside a logic gate. Every transistor in a gate is treated as a switch with on and off characteristics. Since circuit outputs are evaluated by the switching of transistors in real hardware environment, faults modeled at switch-level are more realistic compared to those modeled at other levels such as gate-level or behavior-level. Moreover, since each logic gate is constructed by several transistors in CMOS technology, a switch-level fault list is much larger in comparison with gate-level fault list. These many faults offer a more accurate and thorough fault model.

Switch-level faults need to be mapped to gate-level description so that circuit description can be efficiently synthesized by common design tools [27]. For design entry, Hardware Description Language (HDL) is one of the most popular methods. However, it is not possible to describe a circuit in terms of its switching characteristics in HDL. Thus, a library is necessary to map the switching characteristics into logic behavior.

2.1 Switch-level to Gate-level Fault Mapping

In order to have greater confidence in a system's fault tolerance ability in the presence of faults, a complete and accurate fault list is needed in the fault simulation process. Such a fault list can be attained if all switch-level (or transistor-level) faults are considered thoroughly. However, these faults need to be mapped to gate-level for efficient synthesis and test pattern generation. This section presents the methodology used to map switch-level faults to gate-level description.

Failures in CMOS circuits can be classified into shorts, opens, and circuit degradation. In reality, short and open faults occur most of the time. In this project, for any two-input CMOS gate the following switch-level faults are considered:

1. Short Faults:

- Short between gate and source in both p-channels
- Short between gate and drain in both p-channels
- Short between source and drain in both p-channels

- Short between gate and source in both n-channels
- Short between gate and drain in both n-channels
- Short between source and drain in both n-channels

2. Open (Floating) Faults:

- Open gate in both p-channels and n-channels
- Open source in both p-channels and n-channels
- Open drain in both p-channels and n-channels

In addition, I/O line stuck-at faults need to be considered. For a two-input logic gate, these include:

- Input #1 stuck-at-0 or stuck-at-1 fault;
- Input #2 stuck-at-0 or stuck-at-1 fault;
- Output stuck-at-0 or stuck-at-1 fault.

To better understand this switch-level fault model, consider the following fault model for a CMOS inverter. The inverter is the most essential logic gate in CMOS circuit design. It is used to build many other logic gates such as AND and OR gate. It consists of only two transistors (switches), one n-type and one p-type transistor as shown in Figure 2.1.

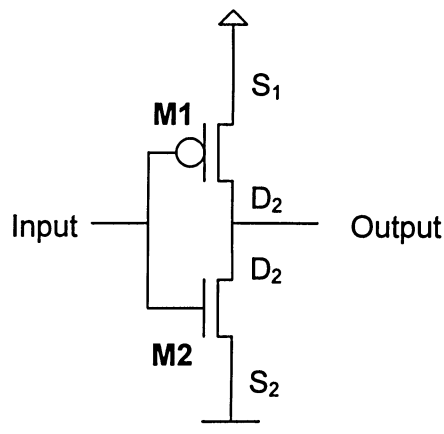


Figure 2.1 CMOS Inverter

The output pattern in the presence of transistor-level fault is presented in the following tables. Table 2.1 contains the possible short faults, Table 2.2 contains the open faults, and Table 2.3 lists all the possible I/O stuck-at faults. Consider variable I as Input, O as output, I1 and I2 as the input to transistors M1 and M2 respectively.

Short Faults	Faulty Output
O shorted with I	I

Table 2.1 Inverter Short Faults

Open Faults	Input Condition	Faulty Output
O is open	For any I	Z
I1 is open	1	0
	0	Z
I2 is open	0	1
	1	Z
I is open	For any I	Z
S1 is open	1	0
	0	Z
S2 is open	0	1
	1	Z
D1 is open	1	0
	0	Z
D2 is open	0	1
	1	Z

Table 2.2 Inverter Open Faults

I/O Stuck-At Faults	Faulty Output
I stuck-at 0	1
I stuck-at 1	0
O stuck-at 0	0
O stuck-at 1	1

Table 2.3 Inverter I/O Stuck-At Faults

After all possible faults have been modeled, a custom VHDL library of these faults was created to make these faults synthesizable. This custom VHDL library also helps simplify the fault injection process, which involves replacing gate descriptions with faulty versions. Let us consider the above inverter fault model. The mapping of switch-level (or transistor-level) faults to gate-level VHDL description is shown in the following figure:

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY INV_F2 is

    port(
        I : IN  STD_LOGIC;
        O  :    OUT  STD_LOGIC);
    END INV_F2;

    ARCHITECTURE faulty of INV_F2 IS

    BEGIN
        -- input is stuck-at-0
        O <= '1';
    END faulty;

```

Figure 2.2 Stuck-At Fault: Input is stuck-at-0

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY INV_F5 is
    port(
        I  :    IN  STD_LOGIC;
        O  :    OUT  STD_LOGIC);
    END INV_F5;

    ARCHITECTURE faulty of INV_F5 IS

    BEGIN
        -- I is shorted to Output
        O <= I;
    END faulty;

```

Figure 2.3 Short Fault: Input is shorted to Output

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY INV_F7 is
    port(
        I    :    IN    STD_LOGIC;
        O    :    OUT   STD_LOGIC);
END INV_F7;

ARCHITECTURE faulty of INV_F7 IS
BEGIN
    -- S1 is open
    O <= '0' WHEN I = '1' ELSE 'Z';
END faulty;

```

Figure 2.4 Open Fault: S1 is open

In the above example, faults for the inverter are mapped to gate-level described in VHDL. This mapping is straightforward as it is according to the above tables that list all the possible faulty output pattern. These various versions of the inverter (INV_F2, INV_F5, INV_F7) are used for fault injection in a circuit by replacing the inverter at the desired location with the faulty version. This is shown in the following code:

```

G47:NOR2    PORT MAP (G50_port, G40_port, G47_port);
I210:INV    PORT MAP (G18_port, I210_port);
G24:OR4     PORT MAP (G38_port, G46_port, G45_port_TEMP_ASSIGN, G40_port, G24_port);
G131:INV_F7 PORT MAP (I232_port, G131_port);
G113:NOR2   PORT MAP (G115_port, G116_port, G113_port);
G112:NOR2   PORT MAP (G62_port, G63_port, G112_port);
G92:NOR3    PORT MAP (G94_port, G95_port, G97_port, G92_port);
G48:AND4    PORT MAP (G45_port_TEMP_ASSIGN, G46_port, G10_port, G47_port, G48_port);
G132:INV    PORT MAP (I235_port, G132_PO);

```

Figure 2.5 Inject a fault into s298 (Inverter G131 pMOS Source is Open)

This fault injection process is semi-automated by a Java program and Perl scripts developed for this project. For the fault models of other basic logic gates, refer to [27].

Chapter 3

Unbalanced Circuit Partitioning

The basic idea of the proposed fault injection campaign is to partition the circuit so that faults are injected into a smaller sub-circuit, resulting in shorter synthesis and reconfiguration time. Two types of partitioning schemes are considered: balanced and unbalanced partitioning [17]. To illustrate these partitioning schemes and their difference, consider the following example.

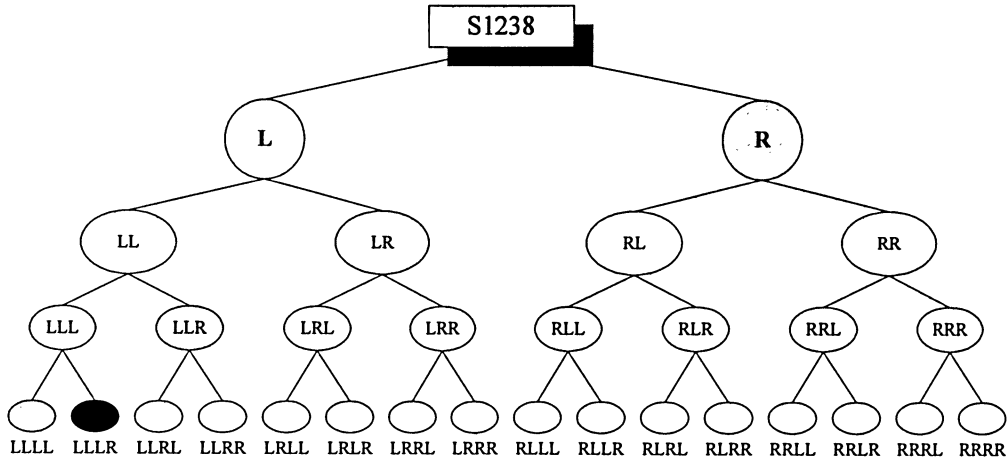


Figure 3.1 Full Binary Tree Circuit Partitioning (L: Left, R: Right)

A circuit is partitioned into a full binary tree structure as shown in Figure 3.1. Each node in the tree represents a certain circuit partition. The size of a partition is roughly half of its parent (ex. $L = LL + LR$). A circuit can be represented by all the leaf nodes with the same (or very similar) size. This partitioning scheme is called balanced partitioning. It has been shown in [17] that in order to reduce the size of the partitions, the circuit must be partitioned with more levels. This results in 2^i leaf nodes, where i is the tree depth level. For example, depth-of-4 partitioning results in $2^4 = 16$ partitions. This increasing number of partitions is not suitable for the implementation of modular-based dynamic partial reconfiguration [8].

Modular-based partial reconfiguration design is considered to be quite complex mainly because it requires the use of pre-synthesized, hard bus macros (explained in Section 4.3) to facilitate the communication between two neighboring modules [8]. If

there are many modules in the design (i.e. many partitions), additional bus macros are needed, which not only consume more hardware resources but also increase the design time and complexity. As suggested in [8], the most efficient design for modular-based dynamic partial reconfiguration includes only two partitions with only one set of bus macros needed between them.

In order to reduce the number of partitions and partition size, an unbalanced partitioning scheme is proposed [17]. This circuit partitioning method uses some of the parent nodes to describe the circuit instead of using all the leaf nodes. This technique greatly reduces the number of partitions. The use of a wrapper file can efficiently group many partitions together to form just one partition. To better explain the application of this partitioning scheme to the proposed fault injection campaign, consider the following example.

After a circuit is partitioned in a full binary tree structure shown in Figure 3.1, a partition (node LLLR) is chosen to be the Dynamic Partition (DP), and the rest of the partitions are wrapped together to be the Static Partition (SP). Instead of having to wrap all 15 partitions (leaf nodes) together, parent nodes can be wrapped together to represent the same sub-circuit. To illustrate this, Figure 3.1 shows the balanced tree partitioning of circuit s1238, and Figure 3.2 shows the unbalanced partitioning of the same circuit.

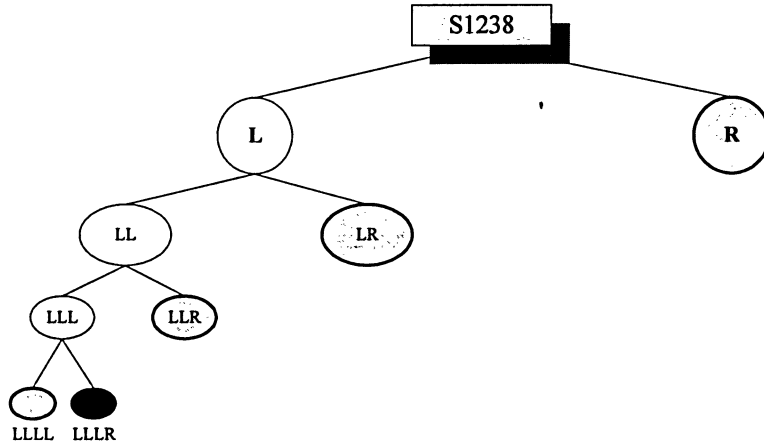


Figure 3.2 Unbalanced Partitioning Graph (DP: LLR)

In this example, partition LLLR (the dark-shaded circle) is chosen to be the Dynamic Partition (DP). The rest of the partitions, LLL, LR and R (the light-shaded portions) are wrapped together to become the Static Partition (SP). Note that only four

partitions are wrapped together instead of 15 in the balanced partitioning case. This greatly simplifies the design complexity.

To apply this partitioning structure in the proposed fault injection campaign, faults are injected in the Dynamic Partition (DP). DP is “dynamic” in the sense that its content changes each time a different fault is injected. In another words, to inject one particular fault, only the partial bitstream that represents that faulty version of DR needs to be downloaded into the device. The rest of the device where SP resides is left untouched.

Chapter 4

Dynamic Partial Reconfiguration

Dynamic Partial Reconfiguration (DPR) is a capability to reconfigure only a portion of an FPGA during runtime without having to shut down the device. This is different from traditional offline reconfiguration, which requires the full device to be reconfigured every time. By utilizing dynamic partial reconfiguration, the fault injection process can be accelerated by only reconfiguring the circuit partition where faults are to be injected.

There are generally two types of DPR design flow: Difference-based and Modular-based flow [8].

4.1 Difference-based DPR Flow

In this design flow, a partial bitstream is generated to represent the difference between the reference design and the new design. A new design can be obtained by changing part of the existing design using this partial bitstream. Therefore, for two designs that are very similar, a very small partial bitstream is generated to reflect its minor differences. This is an ideal DPR flow when designs are very similar and when only small changes need to be made (ex. changing the content of the memory or lookup table configuration).

4.2 Modular-based DPR Flow

Modular-based Dynamic Partial Reconfiguration flow is very similar to regular modular design flow [36]. This flow involves partitioning the design into modules with distinct boundaries. Each module occupies a certain portion of the FPGA and does not overlap with the others. Each module can be individually reconfigured. Two neighboring modules can only communicate with each other using the pre-routed bus macros (explained in Section 4.3). Bus macros are needed to guarantee the same routing configuration is preserved for the new module to properly communicate with

others. Finally, each module is converted to a partial bitstream that can be downloaded at runtime to update the FPGA.

Since circuit partitions are individual entities with distinct boundaries, circuit partitions can be treated as modules. Modular-based partial reconfiguration flow naturally fits the proposed approach. In the proposed scheme, a partition named the Dynamic Partition (DP) will be treated as one module, and the rest of the partitions are wrapped together to form a partition named the Static Partition (SP) module. In this project, faults are injected into Dynamic Partition (DP). This structure is based on the unbalanced partitioning structure explained in Chapter 3. The fault emulation controller (EC) is also considered a module. One thing to note is that the EC module is not needed if the emulation controller is not implemented on FPGA. Microprocessor or PC software may also be used as the emulation controller to eliminate the use of this EC module. The following figure illustrates this implementation structure.

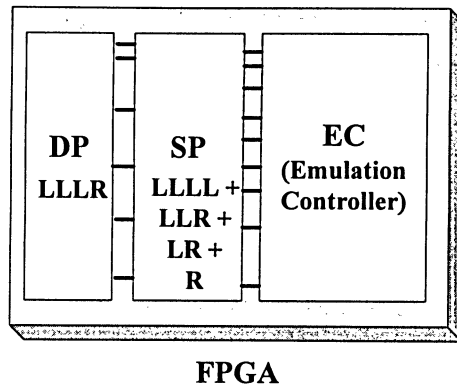


Figure 4.1 Unbalanced Partitioning FPGA Implementation

As shown in Figure 4.1, there are three modules in the modular-based dynamic partial reconfiguration design, namely Dynamic Partition module (DP), Static Partition module (SP) and the optional Emulation Controller module (EC). Modules can only communicate through bus macros [8] represented by the small horizontal lines between the modules. To inject a fault, only DP needs to be reconfigured, and SP, EC are left untouched. As a result, the fault injection time is greatly reduced, and so is the total emulation time.

There are several design phases in modular-based DPR flow. The following section presents the essential details of each phase.

4.2.1 Module Synthesis Phase

Before each module can be placed and routed, it needs to be synthesized. In this

project, modules are synthesized using Xilinx Synthesis Technology (XST) tool [29] through Xilinx Integrated Software Environment (ISE) interface [33]. The most important requirement for modular synthesis is that each module has to be synthesized individually with the “Add I/O Buffer” option disabled. The generated netlist file (.ngc) for all three modules will be used later in the module implementation stage. In addition, the top-level wrapper file that connects the three modules together with bus macro description also needs to be individually synthesized. However, the synthesis for the top-level module should be done with the “Add I/O Buffer” option enabled and with each functional module defined as a block-box level of hierarchy. Therefore, the HDL files for all the modules should not be included in the top-level module synthesis.

The bus macro file (bm_4b_v2.nmc) [8], which is provided by Xilinx, needs to be placed in the same directory with the top-level wrapper module so that bus macro blocks will be expanded during synthesis. That is, a bus macro should not be treated as a black-box because it is already pre-routed and its location needs to be explicitly specified in the constraint file. Other than bus macros, no other modules should be expanded. These are the requirements made by Xilinx in terms of the synthesis process [8]. The following figure contains the synthesis script used in this project.

```
run
-ifn s1238_PARTITION_LLLL.prj
-ifmt mixed
-ofn s1238_PARTITION_LLLL.ngc
-ofmt NGC
-p xc2v1000-fg456-4
-top s1238_PARTITION_LLLL
-iobuf No
```

Explanation

Line 2: Project File Name
Line 3: Project format. Xilinx suggests using mixed format whether it is a real mixed language project or not
Line 4: Output file name
Line 5: Output file format
Line 6: Target technology
Line 7: Top level block name
Line 8: Add I/O buffers option

Figure 4.2 XST Script for Synthesis (xst.scr)

More detail can be found in the XST user guide [29]. After placing the XST script file and the HDL file, the synthesis procedure can be invoked by running the following

command:

```
XST -ifn xst.scr -ofn synthesis_report.log
```

After synthesis, a netlist file is created in NGC format, which is placed and routed later in the implementation stage. The reason that XST is chosen is that XST progresses in each release, improving clock frequencies and decreasing area usage, as well as reducing run time and memory utilization. Moreover, XST is made by Xilinx to provide simpler integration with other tools such as Xilinx Platform Studio [34], which is the processor system design tool used in this project.

4.2.2 Initial Budgeting Phase

Initial Budgeting Phase involves setting up area constraints for the modules and bus macro locations. These constraints are essential to DPR floorplan design. Floorplanning is a process of allocating resource space for each module and placing pre-routed bus macros in between the modules if necessary. The top-level and all three modules must follow the same spatial and temporal constraints, namely location, boundary and timing constraints. An example of such constraints is provided below:

```
#-----  
# Global Constraints  
#-----  
## System level constraints  
Net "sys_clk_pin" LOC=A11;  
# A11 => 24MHz    B6=> 100MHz;  
Net "sys_clk_pin" PERIOD = 41666 ps;  
  
Net sys_rst_pin LOC=B6;  
Net sys_rst_pin TIG;  
  
NET "fpga_0_RS232_TX_pin" LOC = "A7"  ;  
NET "fpga_0_RS232_RX_pin" LOC = "B7"  ;
```

Figure 4.3 Pin Assignment and Clock Constraints

```

#-----
# Area & Location Constraints
#-----

INST "dynamic" AREA_GROUP = "AG_d" ;
AREA_GROUP "AG_d" RANGE = SLICE_X0Y0:SLICE_X7Y79;
AREA_GROUP "AG_d" RANGE = TBUF_X0Y0:TBUF_X6Y79;
AREA_GROUP "AG_d" MODE=RECONFIG;

INST "static" AREA_GROUP = "AG_s" ;
AREA_GROUP "AG_s" RANGE = SLICE_X8Y0:SLICE_X23Y79 ;
AREA_GROUP "AG_s" RANGE = TBUF_X8Y0:TBUF_X22Y79;
AREA_GROUP "AG_s" MODE=RECONFIG;

INST "sys" AREA_GROUP = "AG_sys" ;
AREA_GROUP "AG_sys" RANGE = SLICE_X24Y0:SLICE_X63Y79 ;
AREA_GROUP "AG_sys" RANGE = TBUF_X24Y0:TBUF_X62Y79;
AREA_GROUP "AG_sys" RANGE = RAMB16_X1Y0:RAMB16_X3Y9;
AREA_GROUP "AG_sys" RANGE = MULT18X18_X3Y0:MULT18X18_X3Y9 ;
AREA_GROUP "AG_sys" MODE=RECONFIG;

# Bus Macro Location Constraints -----
INST "bm_D_S_1" LOC = "TBUF_X4Y30";
INST "bm_D_S_2" LOC = "TBUF_X4Y34";
INST "bm_S_H_1" LOC = "TBUF_X32Y30";
INST "bm_S_H_2" LOC = "TBUF_X32Y34";

```

Figure 4.4 Module Area and Bus Macro Location Constraints

In the area constraints, the DP module is assigned 8 slices from the left of the device and the SP module is assigned 16 slices from slice position X8 to X23. Finally, EC module spans 40 slices wide from slice position X24 to the end of the device X63. Note that for all modules, the area spans the entire column of the FPGA. This is one of the requirements of the current DPR technology. Moreover, all resources that reside within the module boundary must be explicitly constrained. These resources include tri-state buffers (TBUF), block RAMs (RAMB16) and Multipliers (MULT). In addition, the key word `AREA_GROUP "<group name>" MODE = RECONFIG;` must be included so that the tool knows that the DPR system is being designed. Lastly, the bus macro location is explicitly defined as shown in Figure 4.4. In the example, only four bus macros are used (two for each boundary intersection). The bus macro position exactly straddles the dividing boundary line between two neighboring modules. Bus macro placement and usage will be discussed in Section 4.2.

After the constraints are set, the following command is run to translate and merge the various source files of the module into a single database file in NGD format [36].

```
ngdbuild -p xc2v1000-4fg456 -modular initial top.ngc
```

```

ngdbuild -p xc2v1000-4fg456 -uc top_sys.ucf -modular
    module -active system ..\top_level_initial\top.ngc
ngdbuild -p xc2v1000-4fg456 -modular module -active
    s27 ..\top_level_initial\top.ngc
ngdbuild -p xc2v1000-4fg456 -modular module -active
    s27_lfsr ..\top_level_initial\top.ngc

```

This concludes the Initial Budgeting Phase. Partial bitstream for reconfigurable modules is generated in the next phase.

4.2.3 Active Module Implementation Phase

In this phase, each netlist is placed and routed individually within its assigned boundary, and the corresponding partial bitstreams are generated. Note that signals that need to cross the boundary are only routed up to the I/O of the bus macros. Final routing is done in the Assembly Phase.

As described in the previous section, the User Constraint File (UCF) [35] contains information about the size of each module (i.e. the boundaries) and bus macro positions on the device. With this file, the tool knows where to place the module and the bus macros. After each module is placed and routed, a viewable configuration file (in NCD format) of each module is generated. Partial bitstream of each module is generated at this stage. In this project, partial bitstreams for each faulty DP version are generated.

A full device design must have been loaded into the device prior to the downloading of partial bitstreams. In this case, this initial design consists of the fault-free DP, its corresponding SP and the Microblaze system. The final Assembly stage will assemble these three modules together to generate a full bitstream as the base design. The following commands are used in this phase.

```

map -u -pr b -p xc2v1000-4fg456 top.ngd -o top.ncd top.pcf
par -w top.ncd top1.ncd
bitgen -d -b -f bitgen_v2_jtag.ut -g binary:yes -g
    ActiveReconfig:Yes top1.ncd top_partial.bit
pimcreate -ncd top1.ncd ..\Pim

```

The MAP command [36] invokes the technology mapping of the module on the floorplan design (top.ngd) obtained from the Initial Budgeting Phase. The PAR command [36] performs the routing of the module within its boundary. Note that even though the routed design is named top1.ncd, it is actually the routed module, not the overall top-level design. BITGEN [36] is the command that generates bitstreams. The option `ActiveReconfig`: must be set to `Yes` to enable dynamic partial bitstream generation. This option makes sure the device is not shut down during the reconfiguration of this partial bitstream. Partial bitstreams for the fault-free and all faulty version of the DP module are generated. In addition, even though SP and EC are not reconfigurable modules (i.e. static modules), they still need to be individually routed. However, the partial bitstream generation process can be skipped. Finally, PIMCREATE command [36] publishes the routed design (NCD file) of each module to a directory where all the routed modules are collected and ready to be assembled.

In the next phase, modules are assembled to form a working base design.

4.2.4 Final Assembly Phase

Before a partial bitstream can be loaded into the device, a full design must be loaded into the device. The initial design consists of the fault-free DP, its corresponding SP and the Emulation Controller module. The following commands are required to perform the final assembly:

```
ngdbuild -uc top_final.ucf -bm top.bmm -p xc2v1000-4fg456
    -modular assemble -pimpath ..\Pim top.ngc
map -detail -pr b -p xc2v1000-4fg456 top.ngd
par -w top.ncd top_routed.ncd
bitgen -d -b -f bitgen_v2_jtag.ut top_routed.ncd
    top_routed.bit
```

The NGDBUILD option `-modular assemble` invokes the assembly action. As described in the previous phase, routed modules are published to a directory to be collected. The directory location is specified using the `-pimpath` option [36].

Final technology mapping and routing are performed to finalize the design. Finally, a full bitstream is created. Note that this phase is only needed to produce the bitstream for the initial design. Partial bitstreams for the faulty dynamic partitions have already been created in the Active Module Implementation phase described in Section

4.2.3. In the next section, more explanation is provided regarding the use of Bus Macros.

4.3 Bus Macro

A bus macro is the essential component in the modular-based partial reconfiguration flow [8]. It facilitates communication between two neighboring modules and keeps the routing resources across module boundaries static and fixed. It is a pre-synthesized, pre-routed circuitry, which consists of a pair of four tri-state buffers at each end of a short 4-bit bus. This structure is illustrated in the following figure.

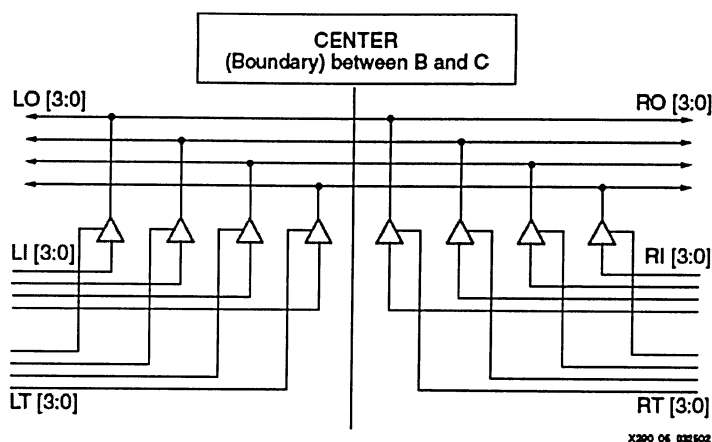


Figure 4.5 Bus Macro Schematic [8]

As shown, a set of two tri-state buffers with one at each end of a one-bit lane controls the direction of data flow. Each lane can be configured to carry traffic from left to right or the opposite. As a result, a bus macro acts as a four-lane bridge between two neighboring modules. Bus macros are needed to ensure the same routing paths are preserved between two neighboring modules after the content of one module is changed by partial reconfiguration. By preserving the same routing path between modules, it allows a modified module to communicate with its neighboring modules.

4.3.1 Bus Macro Placement

Bus macros must be placed in the middle between two neighboring modules that wish to communicate with each other (see Figure 4.6).

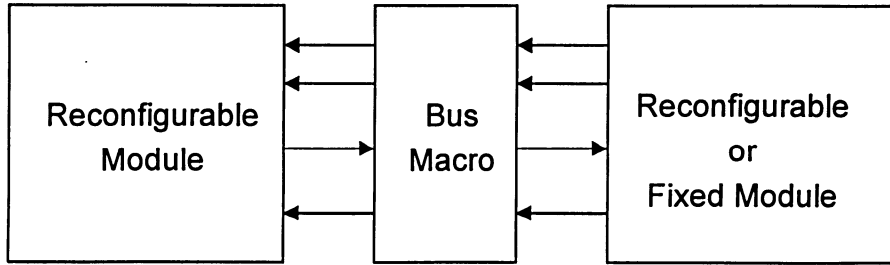


Figure 4.6 Bus Macro Used for Intermodule Signals

In this project, two sets of bus macros are needed for Dynamic Partition module (DP) and Static Partition module (SP) to communicate with each other, and for SP module and Emulation Controller (EC) module to communicate with each other. Figure 4.7 illustrates such placement of these bus macros. The area boundaries of the two neighboring modules must each enclose half of the bus macros. In other words, bus macros must be situated exactly at the centre between two neighboring modules as shown in Figure 4.7. Bus macro placement should be consistent for all compilations of the same design.

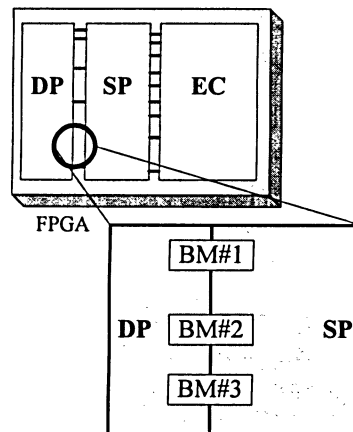


Figure 4.7 Bus Macro Placement

As suggested in [8], the first slice (left-most) of the 8-slice wide bus macros should be placed on a position that is divisible by 4 (ex. X24 or X28) as illustrated in Figure 4.8.

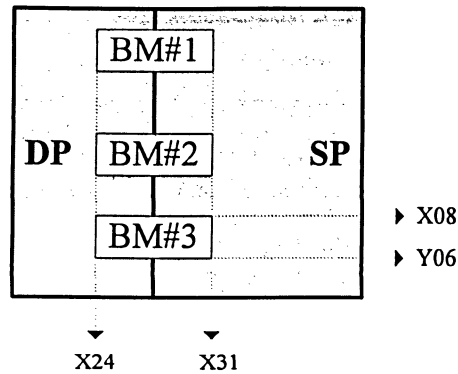


Figure 4.8 Examples of Bus Macro Placement

As shown in Figure 4.8, each bus macro occupies two slices in height. The FPGA used in this project, Virtex-II VC2V1000-4fg456, has only 80 slices in height. Therefore, 40 bus macros is the maximum number of bus macros that can be placed between two neighboring modules. In another words, two neighboring modules can have up to 160 signals across their boundaries. Therefore, the FPGA used in this project prevents the system from testing a circuit with more than 160 primary I/Os because these I/O signals need to be sent from and to the emulation controller (EC). To overcome this, the Test Pattern Generator is implemented inside SP to eliminate the need for bus macros to carry primary inputs from EC to SP. This will be discussed in more detail in Section 5.2.

4.3.2 Bus Macro Usage

A macro file (.nmc) provided by Xilinx needs to be placed in the same directory with the top-level wrapper file. After the initial floorplanning stage, all bus macros will be placed and routed. To use a bus macro, the following declaration (see Figure 4.9) must be included in the top-level wrapper file.

```
-- Bus Macro Declaration
COMPONENT bm_4b_v2 IS
  PORT (  LI, RI  : in std_logic_vector(3 downto 0);
         LT, RT  : in std_logic_vector(3 downto 0);
         O       : out std_logic_vector(3 downto 0)
  );
END COMPONENT;
```

Figure 4.9 Bus Macro Declaration in VHDL

For each bus macro instance, proper port mapping is very important as this process can become quite confusing. The following VHDL code shows such port mapping:

```

bm_example : bm_4b_v2
PORT MAP(
  LI(0) => a_in,    -- signal 'a' from left to right
  LI(1) => b_in,    -- signal 'b' from left to right
  LI(2) => GND_D,   -- to ground
  LI(3) => GND_D,   -- to ground

  LT(0) => GND_D,   -- logic-0
  LT(1) => GND_D,   -- logic-0
  LT(2) => VCC_D,   -- logic-1
  LT(3) => VCC_D,   -- logic-1

  RI(0) => GND_S,   -- to ground
  RI(1) => GND_S,   -- to ground
  RI(2) => c_in,    -- signal 'c' from right to left
  RI(3) => d_in,    -- signal 'd' from right to left

  RT(0) => VCC_S,   -- logic-1
  RT(1) => VCC_S,   -- logic-1
  RT(2) => GND_S,   -- logic-0
  RT(3) => GND_S,   -- logic-0

  O(0) => a_out,    -- crossed the boundary to the right
  O(1) => b_out,    -- crossed the boundary to the right
  O(2) => c_out,    -- crossed the boundary to the right
  O(3) => d_out     -- crossed the boundary to the right
);

```

Figure 4.10 Bus Macro Port Mapping Example

To better illustrate the above port mapping structure, consider the following schematic:

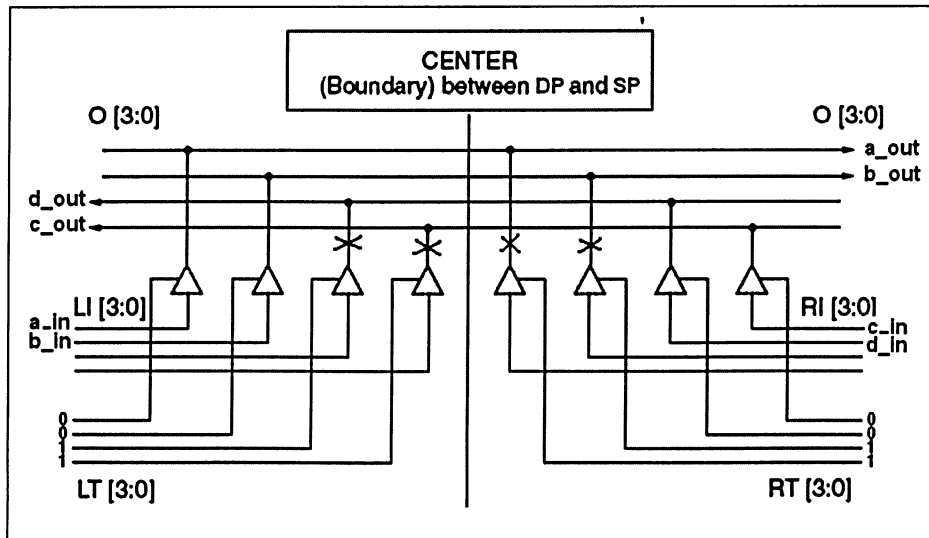


Figure 4.11 Bus Macro Schematic with I/O Signals

In Figure 4.11, signal a_in and signal b_in need to be routed across the DP-SP boundary to signal a_out and signal b_out. On the other hand, signal c_in and signal d_in need to travel from SP to DP. The control signal for a tri-state buffer (LT, RT) controls the direction of the data flow. When LT or RT is pulled low (logic-0), the corresponding tri-state buffer's output is driven by its input data. On the other hand, when LT or RT is pulled high (logic-1), the tri-state buffer's output is at high impedance state, and is driven by the tri-state buffer at the other end. This facilitates bi-directional communication. However, to keep the design simple, the data flow direction should be set during the design stage and should not be changed afterwards.

For signal a_in and signal b_in, signal flows from DP to SP (left to right) because the tri-state buffers on the left side are on, and the ones on the right are at high impedance. The situation is opposite for signal c_in and signal d_in. One important thing to note is that for signals traveling across one or several modules, bus macros are still needed to help route across all boundaries along the way. Therefore, bus macros are needed when a signal needs to travel across any boundary, except for global clock signals.

Chapter 5

Fault Emulation System

The fault emulation system consists of the partial reconfiguration implementation for the partitioned circuit-under-test and the fault emulator module implemented on a soft-core Microblaze processor. Detail information and implementation steps of the system are provided in this section.

5.1 Fault Emulation Controller

5.1.1 Hardware Overview

The Emulation Controller (EC) module is responsible for controlling the entire emulation flow. The system block diagram is presented in Figure 5.1.

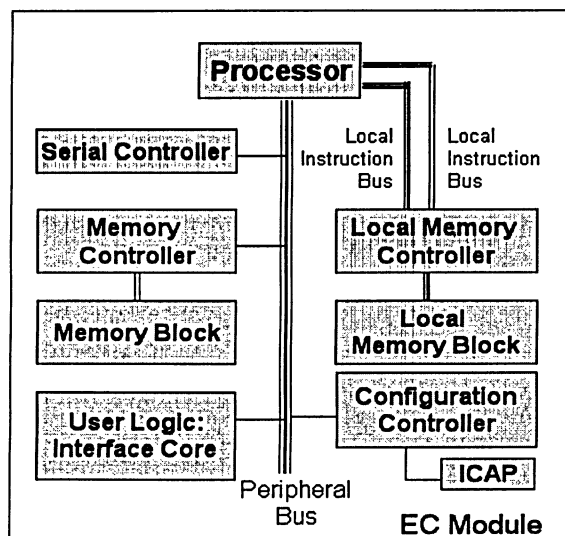


Figure 5.1 Emulation Controller Module Overview

An interface core is designed to provide the interface between the circuit-under-test and the processor. It also interfaces between the test pattern generator and the processor.

This prototype system contains 16KB local instruction/data memory and 32KB Block RAM to store the good outputs and partial bitstreams. Serial communication is needed for debug purposes.

The FPGA chosen has a build-in Internal Configuration Access Port (ICAP), which enables the FPGA to reconfigure itself. By using Microblaze processor, such utilization is very simple because the interface IP core (opb_hwicap core [31]) is already provided for this purpose.

5.1.2 Software Overview

The software flow of the fault emulation and configuration controller is summarized in Figure 5.2. First, the system performs good emulation to obtain the outputs when no faults are present in the circuit. After that the system performs fault emulation by first injecting a fault into the circuit (i.e. partially reconfigure the Dynamic Partition), then the same input patterns are fed into the circuit to obtain outputs with fault present in the circuit. Then the emulation controller determines if a fault is detected by comparing these outputs against the outputs collected during good emulation. The fault coverage result is recorded as the program progresses.

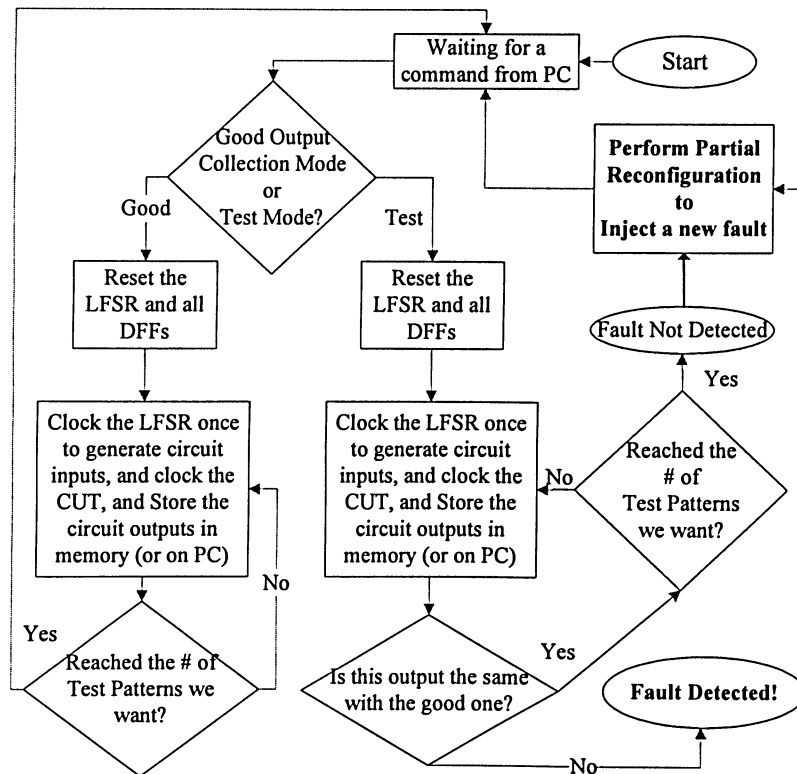


Figure 5.2 Fault Emulation Program Flow

5.1.3 Microblaze Processor System

The presented hardware and software generalization is realized using Xilinx soft-core Microblaze processor. As described earlier, the reason for a processor system is simplicity. Fault emulation process can be easily setup and modified through simple programming. Therefore, this should not count as hardware overhead since one can choose external hardware or PC as the fault emulation controller.

Xilinx Microblaze is a soft-core 32-bit RISC processor. It is soft-core in the sense that its configuration and peripheral can be customized by the designer. The customization is converted to HDL code and gets synthesized. The synthesized processor code is then able to be implemented on the FPGA alone with other circuitry. Since the FPGA board used in this project does not come with a physical processor, Microblaze processor is a suitable choice for this experiment. The following section presents the hardware and software setup for Microblaze system.

5.1.4 Hardware Setup

Xilinx Platform Studio (XPS) [34] simplifies the design flow for Microblaze processor system. The Base System Builder wizard creates a base system with all the fundamental features based on designer's choice (see Figure 5.3). After the wizard generates a base system, custom user core can be added into the system (see Figure 23). As shown in Figure 5.4, LFSR is the user-core added to the base system. The purpose for this user-core is explained in Section 5.3. In addition, address mapping is also done through this interface (see Figure 5.5). Finally, a schematic is generated to provide a graphical view of the system (see Figure 5.6). This system module is then synthesized individually and ready to be implemented along with the circuit-under-test. The integration process is explained in Section 5.6.

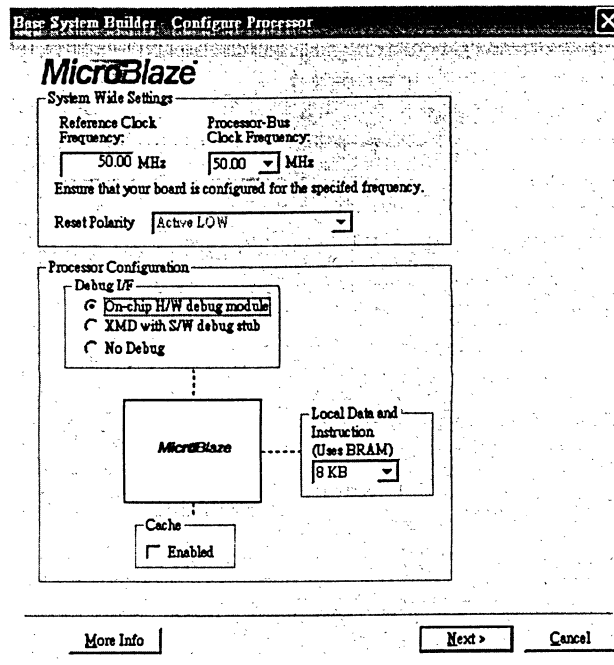


Figure 5.3 Base System Builder Wizard

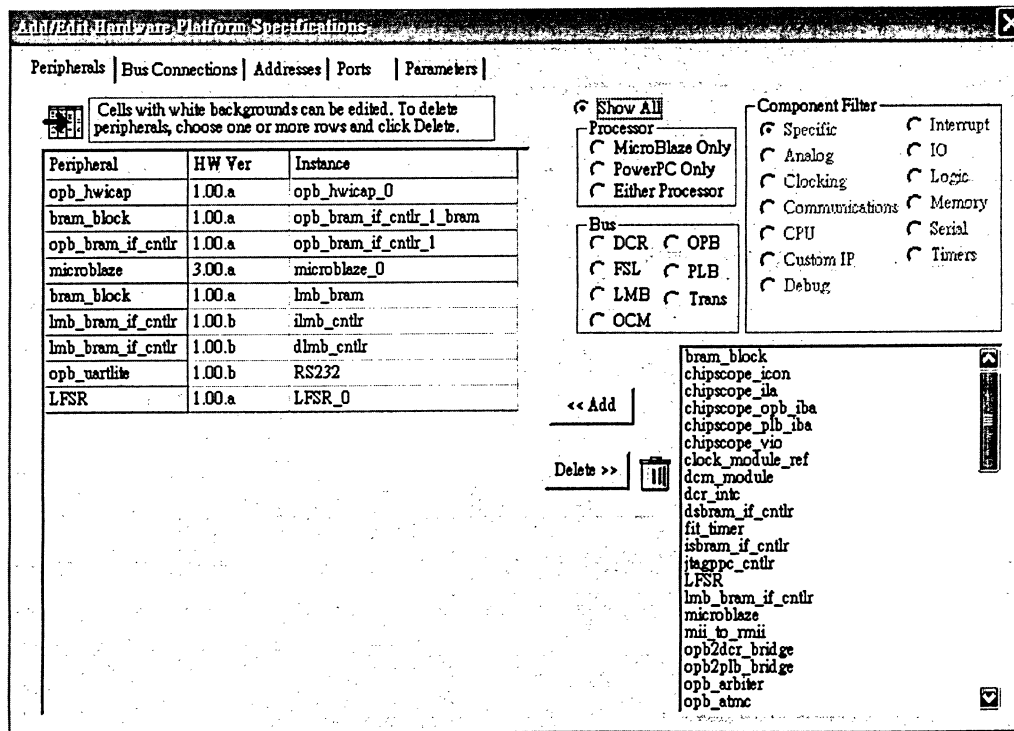


Figure 5.4 Add/Edit Hardware Platform Specifications

Add/Edit Hardware Platform Specifications

Peripherals | Bus Connections | Addresses | Ports | Parameters

Assign Address ranges for all peripherals and bus arbiters

Peripheral	Instance	Lock	Base Address	High Address	Min Size	Size (KB)	I Cache	D Cache
opb_hwicap	opb_hwicap_0	<input type="checkbox"/>	0xFFFF8000	0xffff8fff	0x1000			
opb_bram_if_cntlr	opb_bram_if_cntlr...	<input type="checkbox"/>	0x20000000	0x20007fff	0x800	32	<input type="checkbox"/>	<input type="checkbox"/>
opb_uartlite	RS232	<input type="checkbox"/>	0x80001000	0x800010ff	0x100			
LFSR	LFSR_0	<input type="checkbox"/>	0xffff5000	0xffff50ff	0x100			
lmb_bram_if_cntlr	ilmb_cntlr	<input type="checkbox"/>	0x00000000	0x00003fff	0x800	16		
lmb_bram_if_cntlr	dlmb_cntlr	<input type="checkbox"/>	0x00000000	0x00003fff	0x800	16		

Generate Addresses
Generate addresses for peripherals that do not have locked addresses

Figure 5.5 Address Assignment for Various Components

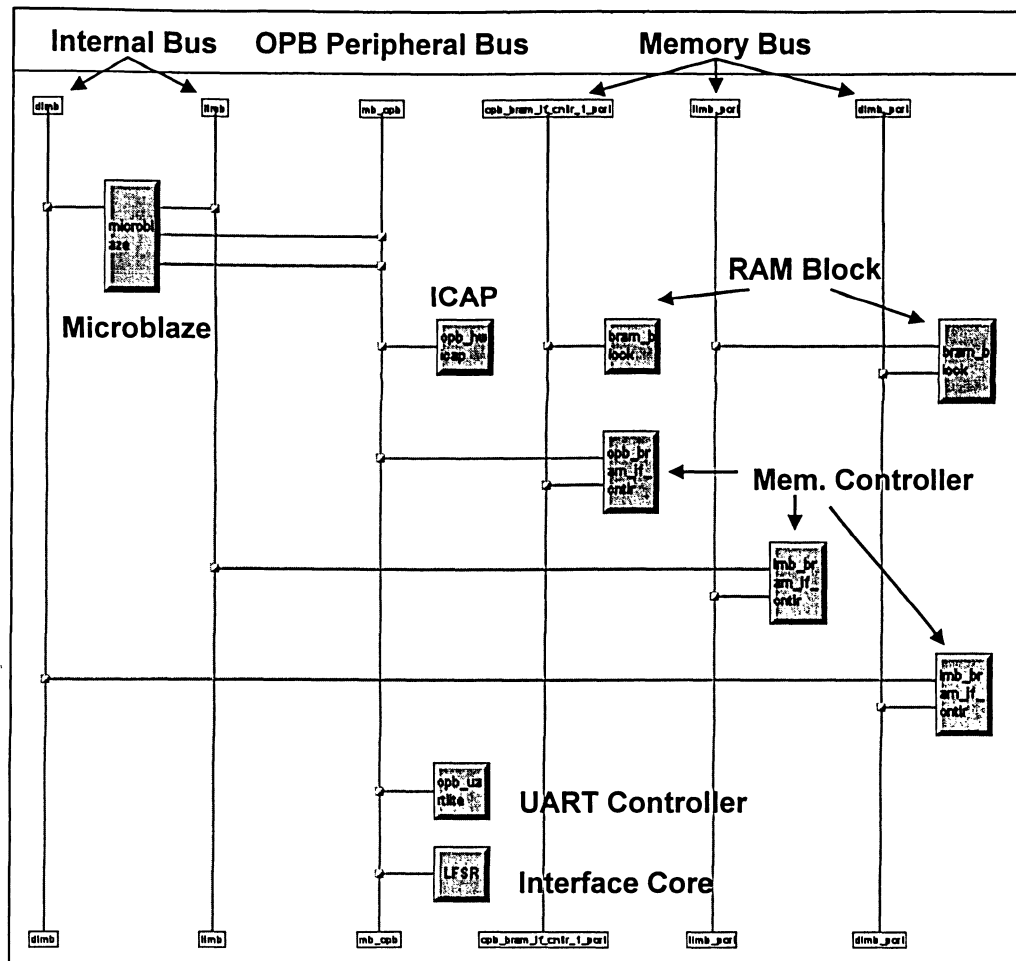


Figure 5.6 Peripheral Block Diagram (PBD) view generated by Xilinx XPS

5.2 Test Pattern Generator

Pseudo-random inputs are used for emulation because the objective of the proposed approach at this stage is not to achieve high fault coverage. Pseudo-random input patterns are generated by the longest sequence XOR Linear feedback shift register (LFSR) [15].

One of the difficulties encountered in this project is the limitations of bus macros. As described in Section 4.3.1, only 160 signals are allowed to travel between two neighboring modules due to hardware limitation. LFSR was originally implemented inside the Emulation Controller (EC) module boundary. Thus, bus macros are needed to carry the generated input patterns from EC module to the circuit-under-test (SP and DP module). This puts a constraint on the circuits that this system can work with. In

order to test a circuit with more than 160 I/Os, the LFSR is implemented in the static partition. Since LFSR is moved from EC module to SP, primary inputs are generated and fed into the static partition of the circuit-under-test circuit within the SP boundary. As a result, the need for bus macros to carry primary inputs from EC to SP is eliminated. This allows the proposed system to work with more circuits with large number of I/Os because the generated test patterns no longer need to travel from EC to SP. However, since the result analyzer (output comparator) is done by the Microblaze processor, it is within the EC boundary. As a result, outputs of the circuit-under-test still have to travel from the DP to SP (if any), and then from SP to EC using bus macros.

5.3 Interface Logic Core

As described in the previous section, input generator is implemented in the static partition module (SP), and the output analyzer is situated in the emulation controller module (EC). Interface core is necessary to interface the LFSR with the Microblaze processor, receive circuit outputs, and send them to Microblaze for comparison. A user-logic core is designed to accomplish these tasks. A user-logic core is an IP core customized by the designer. In this project, this user-logic core is written in VHDL, and is able to communicate with Microblaze processor using the On-Chip Peripheral Bus (OPB) [7]. It is named Interface Core (iCore). In order to communicate through Microblaze processor's bus, certain bus protocol ports are required. Some of the important ports are listed here:

```

entity user_logic is
  generic
  (
    -- Bus protocol parameters
    C_AWIDTH   : integer := 32; -- bus address width
    C_DWIDTH   : integer := 32; -- bus data width
    C_NUM_CE   : integer := 1  -- number of processor chip
  );
  port
  (
    -- Bus protocol ports -----
    Bus2IP_Clk : in   std_logic;      -- Bus to IP clock
    -- Bus to IP address bus
    Bus2IP_Addr : in   std_logic_vector(0 to C_AWIDTH-1);
    -- Bus to IP data bus for user logic
    Bus2IP_Data : in   std_logic_vector(0 to C_DWIDTH-1);
    -- Bus to IP read chip enable for user logic
    Bus2IP_RdCE : in   std_logic_vector(0 to C_NUM_CE-1);
    -- Bus to IP write chip enable for user logic
    Bus2IP_WrCE : in   std_logic_vector(0 to C_NUM_CE-1);
    -- IP to Bus data bus for user logic
    IP2Bus_Data : out  std_logic_vector(0 to C_DWIDTH-1);
    -- IP to Bus acknowledgement
    IP2Bus_Ack  : out  std_logic;
  );
end entity user_logic;

```

Figure 5.7 Important Interface Core Port List

Bus2IP_Data port carries the signal sent from the processor to iCore. Such signals include the clock signal for the LFSR, reset signal for the LFSR, clock signal for the circuit-under-test (CUT), and the reset signal for the CUT. In addition, the IP2Bus_Data port is used by iCore to send the received CUT outputs to the processor for comparison. Bus2IP_Clk is the clock signal for the bus. For more information on user core design, please refer to [32].

5.4 Internal Configuration Access Port

Internal Configuration Access Port (ICAP) is the configuration mode used in the project. Before choosing this configuration mode, standard Boundary Scan (JTAG) configuration mode is used. Both configuration modes support dynamic partial reconfiguration. However, ICAP has the ability to allow FPGA to self-reconfigure itself with the control of an ICAP controller. Microblaze has a pre-designed ICAP controller IP Core called OPB HWIPAP [31]. HWICAP module enables an embedded microprocessor (either Microblaze or PowerPC) to read and write the FPGA configuration memory through ICAP during runtime. FPGA configuration can be changed at runtime through software modification. Such modification is done using the read-modify-write mechanism. Several configuration frames are read into the on-chip memory one at a time. A frame is the smallest granularity where the FPGA

allows configuration data to be read and written. It is a collection of bits, which is one-bit wide and spans the entire column of the FPGA device. Once a frame is read into the memory, modification is done by the processor through software. After modification, the frame is written back to where it is read on the device.

In addition, ICAP core has the ability to load a partial bitstream into the device, therefore the ability to self-reconfigure the FPGA. This feature is used in the project to perform dynamic partial reconfiguration. The following piece of software code is used to accomplish this:

```
#include "xhwicap.h"
:
:
void main () {

    Xuint32 *data = 0x20000000,size = 6309;
    XHwIcap myICAP;
    :
    :
    XHwIcap_Initialize(&myICAP, XPAR_OPB_HWICAP_0_DEVICE_ID ,XHI_XC2V1000);
    XHwIcap_SetConfiguration(&myICAP, data, size); // size in 32bit word
    :
    :
}
```

Figure 5.8 Software Code to perform Dynamic Partial Reconfiguration

The above code loads a partial bitstream stored in the on-chip memory at location 0x20000000. The partial bitstream size is 6309 x 32-bit =201,888 bits.

5.5 Serial Communication Controller

The standard RS232 serial communication port is used in this project mainly for debugging purposes. It is also used by a Java program running on PC to send commands to Microblaze processor running on FPGA. Note that even though Microblaze is responsible for controlling the fault emulation process, a Java program is designed to test the system. Therefore, this is an optional part of this system prototype to help debug and simplify the proposed fault emulation approach.

The serial communication IP core used by Microblaze is called the OPB UART Lite [30]. UART stands for Universal Asynchronous Receiver-Transmitter. It is a full-duplex (one transmit and one receive channel) transmission module with a configurable transmission rate. The transmission speed (baud rate) is set to 115200 bits per second with no parity. UART Lite core can be easily added into the system by setting it as Microblaze's standard I/O as shown in the following figure.

Library and OS Parameters				
Instance	Current Value	Default Value	Type	Description
microblaze_0: standalone				
stdin	RS232	none	periph...	stdin peripheral
stdout	RS232	none	periph...	stdout peripheral

Figure 5.9 Microblaze OS Parameter Setup

5.6 Final System Integration

Before a partial bitstream can be loaded into the device, an initial design must already exist on the device. This initial design consists of the fault-free DP, its corresponding SP and the emulation controller module presented in Section 5.1. The initial design is created at the final Assembly phase where the three modules are assembled to form a full bitstream. Figure 5.10 shows the assembled design, which is an elaborated version of the system block in Figure 4.1.

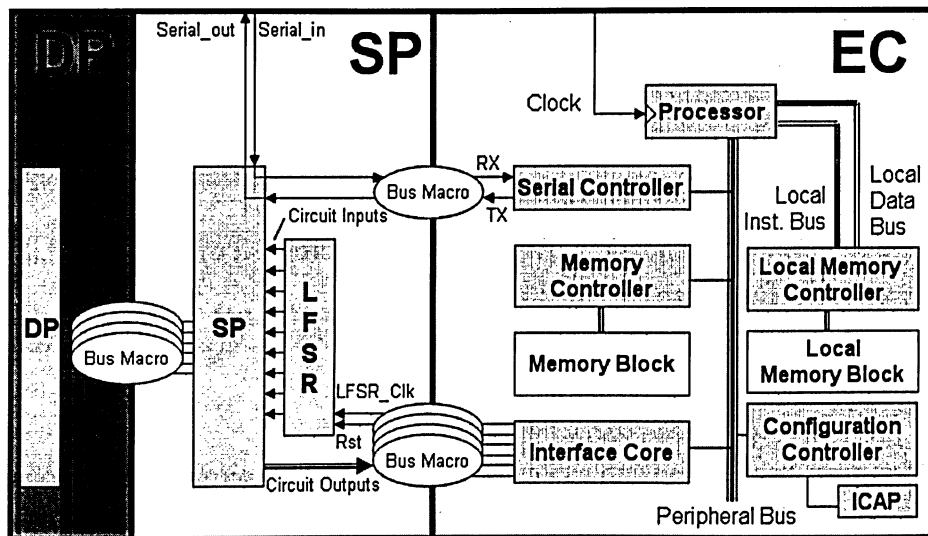


Figure 5.10 Block Diagram for the Final System

After the final place and route of the assembled system, the actual FPGA resource view can be obtained (see Figure 5.11).

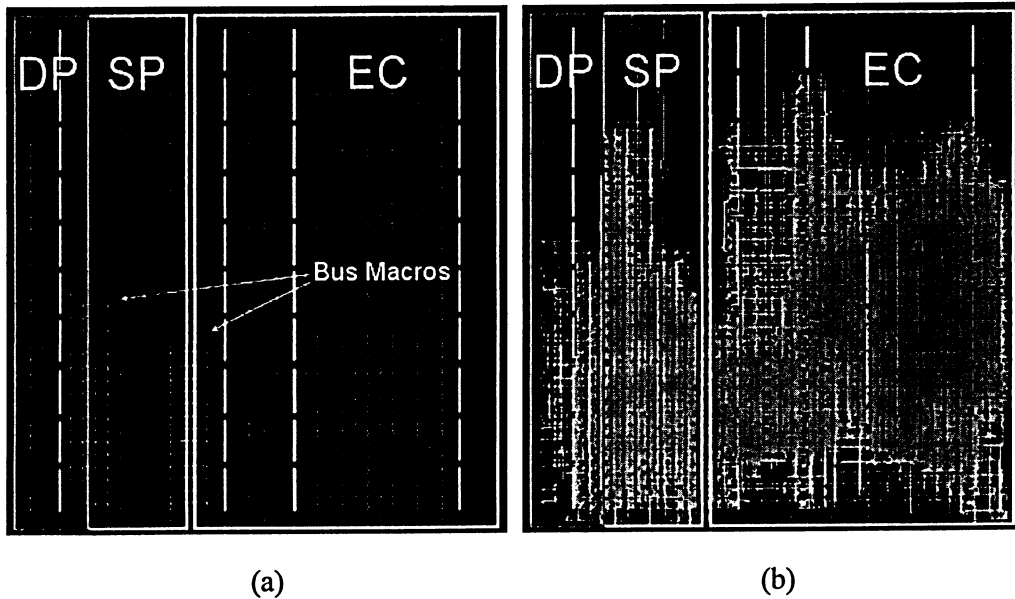


Figure 5.11 (a) Initial Floorplan (b) Final Assembled System for s5378

In Figure 5.11, clear boundaries between the modules are shown. Signals can only cross module boundary through bus macros. Moreover, in Figure 5.11, 14 bus macros are used between SP and EC because there are 56 signals traveling across this boundary. These signals are:

- 49 primary output signals for s5378
- 2 signals for transmission (TX) and receiver port (RX) of the UART Lite core
- 2 signals for circuit reset and LFSR reset
- 2 signals for circuit clock and LFSR clock
- 1 signal for global reset to go into EC

Therefore, since each bus macro can carry four 1-bit signals across module boundary, a total of $56/4 = 14$ bus macros are needed. Bus macros for UART Lite's TX and RX signals are needed for them to route between SP and EC. This is because the physical location of these two I/O pins resides within the SP boundary. It is the same case for the global reset signal, which needs to be routed from the pin location residing within the SP boundary to Microblaze processor residing within the EC boundary. This final design is the realization of the proposed system illustrated in Figure 5.10, which implements the unbalanced partitioning structure illustrated in Figure 3.2 and Figure 4.1.

5.7 Inject Faults in Other Partitions

To inject faults into other partitions, one first needs to determine the partition where the faults are to be injected. Then rerun the modular-based partial reconfiguration flow with this partition being the DP, and the rest of the circuit being the SP. The remaining steps are the same. All the above steps are automated through the developed scripts and Java programs.

5.8 Programming the FPGA with Partial Bitstreams

After the full bitstream is generated, it can be loaded into the FPGA. Internal Configuration Access Port (ICAP) is the method used to reconfigure the device. ICAP is a subset of the SelectMap configuration architecture [31]. The difference is that ICAP can be accessed internally in the FPGA. Originally, JTAG was used to perform partial reconfiguration. However, JTAG requires a configuration controller program running on a PC. In addition, by utilizing the provided ICAP IP Core, Microblaze is able to self-reconfigure the device with partial bitstreams stored in the on-chip memory. This approach reduces the reconfiguration time and greatly simplifies the system.

Chapter 6

Experimental Results

Two main results are presented in this section: fault injection time and fault coverage. Fault coverage results demonstrate the ability of the proposed system to perform switch-level fault emulation. Fault injection time indicates the efficiency of this approach.

The presented fault emulation system is evaluated using ISCAS-89 benchmark suite [13]. Ten circuits are selected for this experiment. The following table summarizes these benchmark circuits.

Circuit	No. of Switches	No. of Inputs	No. of Outputs
C17	24	5	2
S27	66	4	1
S298	694	3	6
S1238	2,718	14	14
C2670	6,212	233	140
S5378	15,456	35	49
C7552	18,802	207	108
S13207	30,984	62	152
S35932	80,730	35	320
S38584	93,194	38	304

Table 6.1 ISCAS-89 Benchmark Circuits (C: Combinational, S: Sequential)

A selected number of faults are injected into the above benchmark circuits, and fault injection and coverage results are collected. The following section provides more detail.

6.1 Switch-level Fault Injection Result

This experiment demonstrates that the proposed fault injection method is able to significantly shorten the fault injection time. As described earlier, the speed of fault injection depends on:

1. Transmission speed of the configuration medium,
2. Processing speed of the configuration controller,
3. Size of the dynamic partition (DP).

Mathematically, fault injection time (T_{PR}) can be described in the following equation (values in the parentheses are values used in this experiment):

$$T_{PR} = \frac{B}{W_C \times C} + \frac{B}{W_{Bus}} \times T_P \quad (1)$$

where,

B = the number of bits in the partial bitstream

W_C = width of configuration medium (8-bit)

C = system clock speed (24MHz)

W_{Bus} = width of the system bus, size of data chunk (32-bit)

T_P = time to process each configuration data chunk ($\sim 3\mu s$)

The first term $\frac{B}{W_C \times C}$ represents the configuration data transmission time. Since a byte-serial configuration medium, Internal Configuration Access Port (ICAP) [31], is used in this experiment, each 8-bit data block can be transmitted every clock cycle without handshaking as long as the system clock speed is under 50MHz. Moreover, the configuration controller needs to fetch the configuration data from memory and send it to the configuration controller. This processing time is represented as the second term in equation (1). The following table presents the fault injection results.

Circuit	Switch No.	Partial Bitstream Size	FI ¹ Time (ms)	Calculated FI ¹ Time (ms)
C17	24	195,104	19.35	19.31
S27	66	201,888	20.03	19.98
S298	694	255,840	25.38	25.32
S1238	2,718	258,496	25.64	25.58
C2670	6,212	262,144	26.00	25.94
S5378	15,456	251,744	24.97	24.91
C7552	18,802	258,208	25.61	25.55
S13207	30,984	258,208	25.62	25.55
S35932	80,730	258,208	25.62 ²	25.55
S38584	93,194	258,208	25.622	25.55

¹FI: Fault Injection ² Predicted using calculated value

Table 6.2 Fault Injection Time

As shown in Table 6.2, the calculated result using equation (1) is very close to the experimental results. Thus, it is safe to assume that the calculated result for circuits, which are too large to be implemented on the FPGA used in this experiment (i.e. S35932, S38584), will be very close to the real value. Clearly, by utilizing dynamic partial reconfiguration, fault injection time can be significantly reduced compared to traditional full-device reconfiguration, which takes about 405ms for each fault injection.

Several steps are involved when performing reconfiguration. These detail configuration steps are explained in [16]. In this experiment, the average transmission time is around 1.1ms. Therefore, the processing speed of the configuration controller contributes most of the runtime. Since both transmission and processing speed are technology dependent, fault injection time can be significantly reduced by using faster FPGA and configuration mode. Furthermore, if faster fault injection time is desired, the size of dynamic partition can be reduced by increasing the depth of the partitioning tree described in Chapter 3.

6.2 Linear Growth of Fault Injection Runtime

As indicated in Table 6.2, fault injection time is almost constant with respect to the circuit size. This is also shown in the following graph:

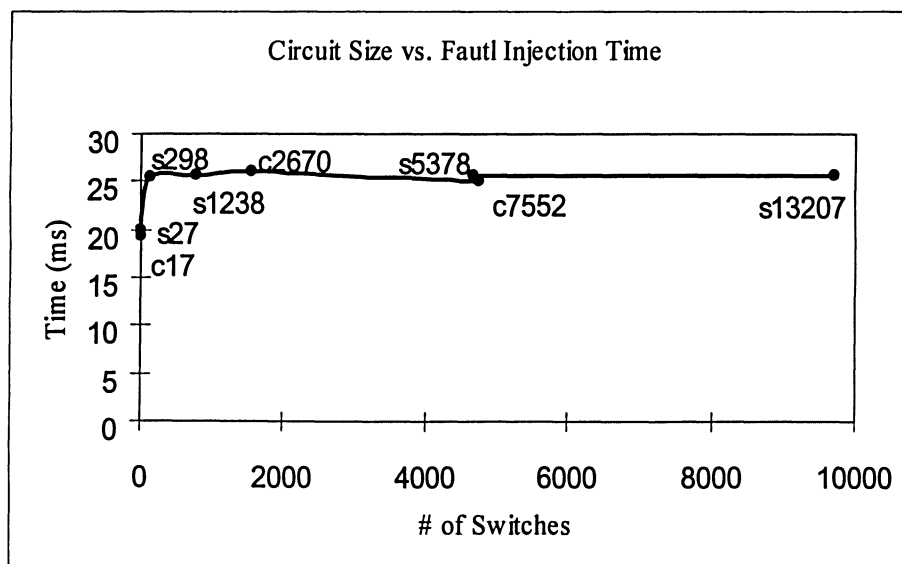


Figure 6.1 Fault Injection Time vs. Circuit Size

This is expected because for each benchmark circuit, dynamic partition (DP) size is chosen to be the same. Therefore, the same number of configuration frames must be downloaded into the device. A frame is the smallest unit of configuration data, which is 1-bit wide and extends from the top of the device to the bottom. It is the smallest portion of the configuration memory that can be written to [16]. The slight time variation is due to the optimization done by the bitstream generation tool, which tries to reduce the bitstream size. In general, the fault injection time is considered constant for the same circuit size because DP size stays the same.

Fault injection time of the presented system grows linearly with circuit size due to the binary-tree partitioning scheme described in Chapter 3. This is because each level down the partitioning tree the partition's size is reduced to approximately half of its parent. Consequently, increasing the size of the parent partition results in the increase of its children's partition size. For example, a circuit-under-test (CUT) with size of n_g gates is partitioned using unbalanced partitioning with depth of three levels as shown in Figure 6.2. As a result, the size of the dynamic partition (DP) becomes $n_g/8$. If circuit size doubles (n_g changes to $n_g/2$), it is clear that the size of DP also doubles ($n_g/8$ changes to $n_g/4$). Since DP size increases linearly with circuit size, and fault injection is directly proportional to the size of DP module, fault injection time is not affected by circuit complexity. This is one of the major advantages of the proposed fault injection campaign. When CUT gets large and complex, the resulting fault injection time still increases linearly as opposed to exponential growth in software approach [19].

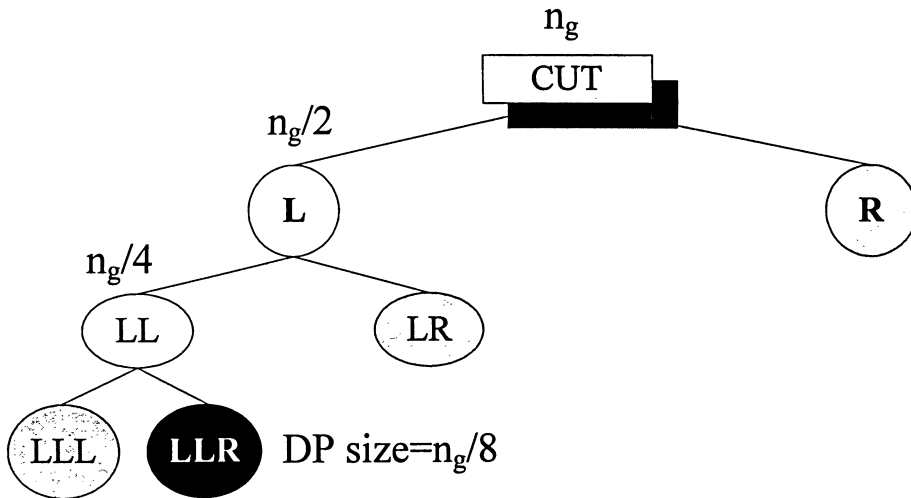


Figure 6.2 Dynamic partition's size grows linearly with circuit size

6.3 Switch-level Fault Coverage

Table 6.3 presents the switch-level fault coverage (FC) for the selected benchmark circuits.

Circuit	No. of switches	No. of faults injected	FC (%)
C17	24	60	100
S27	66	120	99.69
S298	694	130	68.46
S1238	2,718	526	68.82
C2670	6,212	1,550	67.80
S5378	15,456	4,754	58.94
C7552	18,802	4,670	59.12
S13207	30,984	9,700	57.03

Table 6.3 Switch-level Fault Emulation Result

In most cases, switch-level fault coverage is usually lower than gate-level fault coverage. This is mainly because test pattern for detecting switch-level faults is more difficult to generate [37]. One thing to note is that the proposed approach does not focus on achieving high fault coverage as that will require high quality test pattern. The fault coverage obtained from this experiment is similar to that of software switch-level simulation approaches such as [28]. By modeling faults at switch-level, the resulting fault list is more accurate than approaches in [1,3], in which faults are modeled and injected at gate-level and lookup-table level.

6.4 Runtime Prediction for Large Circuits

The worst case total emulation runtime occurs when all the faults injected are undetected after applying maximum allowable number of test patterns. This can be further explained in the following equation:

$$\begin{aligned}
 T_{\text{RUNTIME}} = & T_{\text{FR}} + T_{\text{INIT}} + (T_{\text{GE1}} \times p_{\text{MAX}}) + (T_{\text{PR}} \times n) \\
 & + \sum_{i=1}^n (T_{\text{FE1}} \times \max\{p_{\text{MAX}}, p_i\})
 \end{aligned} \tag{2}$$

where,

p_i = the number of test patterns needed to detect the i th fault

p_{MAX} = the number of test patterns allowed to run

n = total number of faults to inject

T_{FR} = time to perform full-device reconfiguration

T_{PR} = time to perform partial reconfiguration

T_{INIT} = initialization time for the system and components

T_{GEI} = time to perform good emulation for 1 test pattern

T_{FEI} = time to perform fault emulation for 1 test pattern

In equation (1), T_{PR} is considered a constant term if the size of the dynamic partition does not change with circuit size. If circuit size increases, DP size will increase proportionally (as explained in Section 6.2). As a result, T_{PR} is linear with respect circuit size. Since fault-free emulation is only performed once for each circuit, T_{FR} can T_{INIT} can also be considered as constant. In hardware, normally as circuit size increases, the emulation time for one test pattern (T_{GEI}, T_{FEI}) will also increase proportionally. Furthermore, total fault emulation time also increases linearly with respect to the number of test patterns applied. This is illustrated in the following graph:

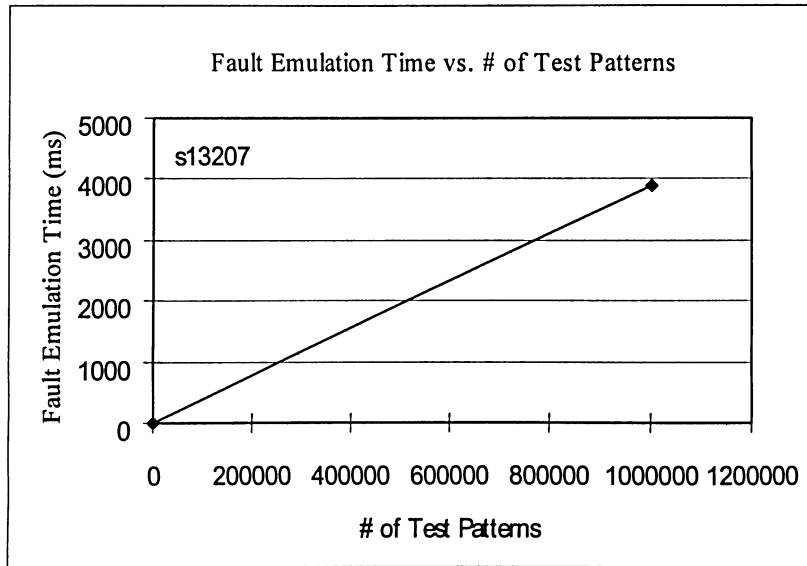


Figure 6.3 Fault Emulation Time vs. Number of Test Patterns for s13207

In addition, when a large number of test patterns are applied to emulation system, the emulation time and circuit size (number of switches) still have a linear relationship as shown in the following graph:

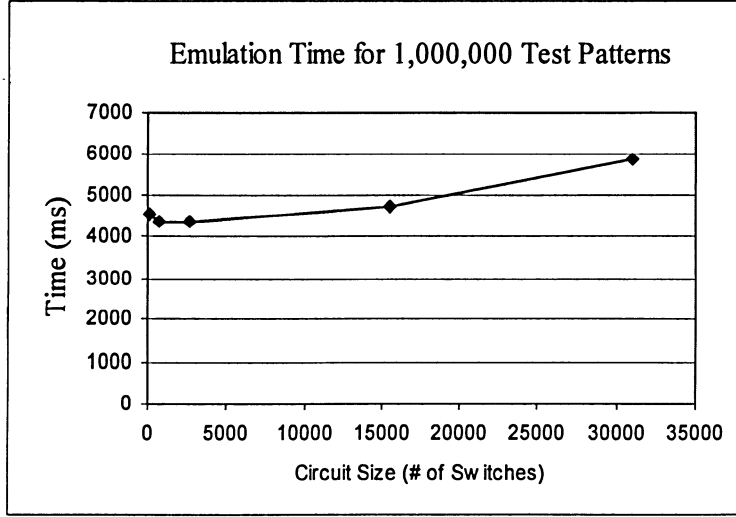


Figure 6.4 Emulation time vs. Circuit size for large number of Test Patterns

To better understand the relationship between the fault emulation time and circuit size, a mathematical formula is developed for variable T_{FEI} . T_{FEI} is the fault emulation time for the application of one test pattern, it can be described as follows:

$$T_{FEI} = T_{EC} \times \left\lceil \frac{N_{PO}}{W_{Bus}} \right\rceil + T_{OH} \quad (3)$$

where,

T_{EC} = Time for processing each emulation result block. This includes storing circuit outputs into memory, fetching fault-free emulation result from memory and comparing results to determine if a fault is detected (0.375 μ s in this experiment)

N_{PO} = The number of primary outputs (for output comparison)

(ex. Circuit s35932 has 320 primary outputs)

T_{OH} = System overhead (2.25 μ s in this experiment)

W_{BUS} = The width of the bus used in the system (32-bit in this experiment)

However, equation (3) is only true in this experiment because it is based on the processor-based emulation controller. For example, Microblaze uses 32-bit wide bus to receive the circuit outputs from the interface core. As a result, only 32 output signals can be processed at one time. Therefore, in equation (3), total primary outputs of the circuit-under-test divided by the width of the bus gives the number of data blocks that the processor needs to process. Ceiling operator is necessary because the processor can only process block of data. Therefore, even if there is only one output signal needs to be processed, a data block is required to include this signal. Figure 6.5

shows the relationship between the emulation time (not including fault injection time) and circuit size:

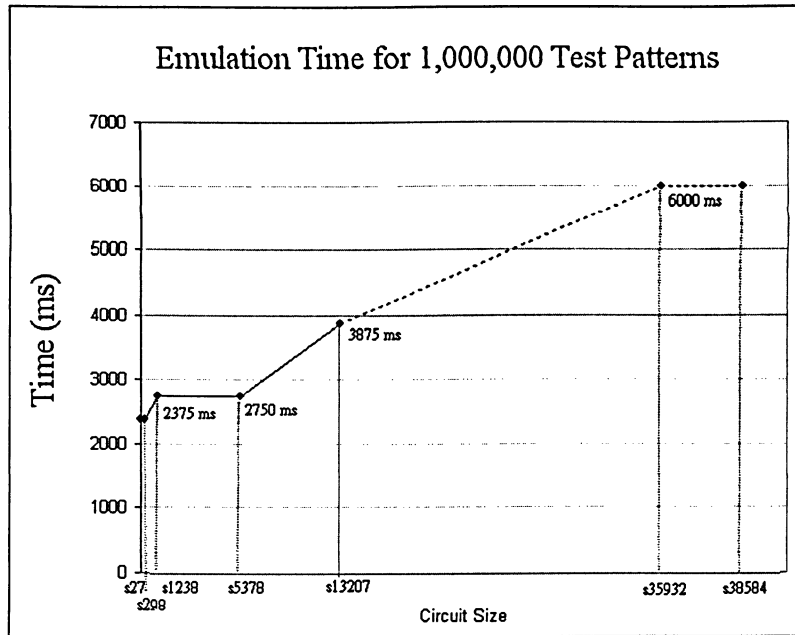


Figure 6.5 Emulation Time vs. Circuit Size

As shown in Figure 6.5, the emulation time increases linearly with circuit size. Emulation time for circuit s35932 and s38584, which are too large to fit on the FPGA, are calculated using equation (3). This linear growth of emulation time is expected because circuit logic is evaluated using lookup-table-based FPGA. For this type of hardware emulator, circuit complexity does not cause exponential growth in runtime because circuit outputs are evaluated in parallel and by table lookups, whereas for software emulators circuit outputs are computed serially. Since all terms in equation (2) are either fixed or linear with respect to circuit size, it is safe to state that the total runtime of the presented system still grows linearly when a larger and more complex circuit is emulated.

In addition, the linear growth in runtime of the presented system is expected because of the following characteristics:

1. The fault-free emulation only needs to be performed once for each circuit. Therefore, the time it takes can be considered constant.
2. LFSR generates a new input pattern every clock cycle regardless of circuit complexity.
3. Faulty circuit (when a fault is present) outputs are compared with good emulation outputs as soon as they are ready. Emulation process stops as soon as a fault is

detected.

4. Partial reconfiguration time depends on the size of the partial bitstream (dynamic partition size), which is constant or grows linearly with circuit size as described in Section 6.2.

One of the advantages of the proposed system is that it does not require extra fault injector circuitry as required in [25]. The only hardware overhead is the bus macros, which occupy very small areas as shown in Figure 5.11. One thing to note is that the EC module is optional and not hardware overhead. The emulation controller implemented using Microblaze soft-core processor is only for simplifying and accelerating the implementation steps. A custom circuitry, PC software, or a hard-core microprocessor can serve as the emulation controller to achieve minimum hardware overhead.

Chapter 7

Conclusion

In this report, a new FPGA-based switch-level fault emulation approach is introduced. In this approach, a novel fault injection campaign that utilizes modular-based dynamic partial reconfiguration is proposed to shorten fault injection time. Since switch-level fault model offers a more accurate and thorough fault list in comparison to gate-level fault model, switch-level fault emulation was considered in this project. However, the existing lookup-table fault injection approaches are unsuitable for switch-level fault emulation because of possible circuit information loss. Therefore, switch-level fault injection requires faults to be injected at circuit description level where original circuit structure is still preserved. However, many full-device synthesis, routing and reconfigurations are required in this approach. These design steps take up too much time for this approach to be practical. By using modular-based dynamic partial reconfiguration, faults are injected into a smaller sub-circuit's HDL file, resulting in synthesis time reduction. In addition, faulty circuit partition is converted into partial bitstream, which can be downloaded into FPGA at runtime instead of having to perform full-device reconfiguration for each fault injection.

The resulting system implemented in this project is able to emulate faults at switch-level without extra fault injector circuitry, thus keeping hardware overhead minimal with only small bus macros. The experimental results show that the proposed fault emulation approach is able to achieve linear runtime with respect to circuit size. Thus, the presented approach is an efficient method when emulating large, complex circuits.

7.1 Project Contribution

In this project, a Virtex-II FPGA is used to emulate switch-level faults. Switch-level faults are mapped to gate-level and synthesized in VHDL. The unbalanced partitioning structure of the circuit-under-test is implemented on FPGA to facilitate modular-based dynamic partial reconfiguration design. Modular-based DPR allows a circuit partition to be reconfigured instead of requiring a full-device

reconfiguration. This capability is used in the proposed fault injection campaign. In the proposed fault injection method, faults are only injected into a small dynamic partition, which. This greatly reduces the reconfiguration-based fault injection time. In addition, the proposed approach also achieves linear growth in total runtime with respect to circuit size.

7.2 Future Work

Areas for future work are now discussed. Recently, region-based dynamic partial reconfiguration (DPR) is developed by Xilinx in their Virtex-4 FPGA series. Region-based DPR allows partial reconfiguration to a region, which could consist of only a few CLBs (Configurable Logic Block) rather than full-column resources on the device. Full-column reconfiguration required in the proposed approach is not efficient because resources that have not been modified still need to be reconfigured as long as they are within the boundary of a module being reconfigured. This new capability will further accelerate the proposed fault injection campaign and result in shorter fault emulation time. In addition to design flow improvement, a complete suite of scripts and programs can be created to provide simpler and robust fault injection process. Furthermore, the proposed fault injection campaign can be further improved by using faster FPGA with faster configuration mode. This would allow faster partial reconfiguration, thus improving the total fault emulation runtime.

References

- [1] L. Antoni, R. Leveugle, and B. Feher, Using Run-Time Reconfiguration for Fault Injection Applications, IEEE Transaction on Instrumentation And Measurement, October 2003, Volumn 52, Issue 5, pp. 1468-1473.
- [2] Xilinx Documentation: JBits 2.8, Xilinx, San Jose, CA, September 2001
- [3] A. Parreira, J.P. Teixeira, and M. Santos, A Novel Approach to FPGA-Based Hardware Fault Modeling and Simulation, the Proceedings of the Design and Diagnostics of Electronic Circuits and Synt. Workshop, April 2003, pp. 17-24.
- [4] R. Leveugle, Towards modeling for dependability of complex integrated circuits, IEEE Internatinal On-Line Testing Workshop, July 1999, pp. 194-198.
- [5] K.T. Cheng, S.Y.Hunag, and W.J. Dai, Fault Emulation: A New Methodology for Fault Grading, IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, October 1999, Volumn 18, Issue 10, pp. 1487-1495.
- [6] L. Burgun, F. Reblewski, G. Fenelon, J. Barbier, O. Lepape, Serial Fault Emulation, the Proceedings of the 33rd IEEE Design Automation Conference, June 1996, pp. 801-806.
- [7] Xilinx Documentation: LogiCore On-Chip Peripheral Bus V2.0, July 2004, DS401.
- [8] Xilinx Application Note: Two Flows for Partial Reconfiguration: Module based or Difference Based (v1.2), September 2004, XAPP290.
- [9] S.A. Hwang, J.H. Hong, C.W. Wu, Sequential Circuit Fault Simulation Using Logic Emulation, IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, August 1998, Volumn 17, Issue 8, pp. 724-736.
- [10] L. Antoni, R. Leveugle, and B. Feher, Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes, the Proceedings of the 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, November 2002, pp. 245-253.
- [11] B. Blodget, S. McMillan, A lightweight approach for embedded reconfiguration of FPGAs, the Proceedings of the IEEE Design Automation and Test in Eurpore Conference and Exhibition, 2003, pp. 399-400.
- [12] A. Parreira, J.P. Teixeira, A. Pantelimon, M.B. Santos, J.T. Sousa, Fault Simulation using Partially Reconfigurable Hardware, the Proceedings of the 13th International Conference on Field Programmable Logic and Applications, September 2003, (page number not found). URL: http://fidelio.inesc-id.pt/~jts/fpl2003_final.PDF
- [13] ISCAS-85 Benchmark Circuits: <http://www.fm.vslib.cz/~kes/asic/iscas/>

- [14] R. Leveugle, L. Antoni, B. Feher, Dependability Analysis: a New Application for Run-Time Reconfiguration, Internatinal Parallel and Distributed Processing Symposium (IPDPS'03), April 2003, pp. 7.
- [15] Xilinx Application Note: Linear Feedback Shift Registers in Virtex Devices (v1.2), Januray 2001, XAPP210.
- [16] Xilinx Application Note: Virtex Series Configuration Architecture User Guide (v1.7), October 2004, XAPP151.
- [17] R. Abedi, R. Sedaghat, Classical and Non-classical Transistor Level Fault Injection into FPGAs, WSEAS Transaction on Circuits and Systems, February 2006, Volume 5, Issue 2, pp. 234-240.
- [18] R. Sedaghat, M. Kunchwar, R. Abedi, R. Javaheri, Transistor-level to gate-level comprehensive fault synthesis for n -input primitive gates, Elsevier Microelectronics Reliability, December 2005. (Article in Press)
- [19] P. Civera, L. Macchiarulo, M. Rebaudengo, M.S. Reorda, M. Violante, Exploiting FPGA-based Techniques for Fault Injection Campaigns on VLSI Circuits, the Proceedings of the 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, October 2001, pp. 250-258.
- [20] S.A. Hwang, J.H. Hong, Sequential Circuit Fault Simulation Using Logic Emulation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, August 1998, Volume 17, Issue 8, pp. 724-736.
- [21] P. Folkesson, S. Svensson, J. Karlsson, A comparison of simulation based and scan chain implemented fault injection, Proceedings of 28th International Symposium on Fault-Tolerant Computing, 1998, pp. 284-293.
- [22] H. Madeira, M. Rela, J.G. Silva, A general purpose pin-level fault injector, the Proceedings of First European Dependable Computing Conference, 1994, pp. 199-216.
- [23] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, Exploiting circuit emulation for fast hardness evaluation, IEEE Transactions Nuclear Science, December 2001, Volume 48, Issue 6, Part 1, pp. 2210-2216.
- [24] T.J. Chakraborty, C.H. Chiang, A novel fault injection method for system verification based on FPGA boundary scan architecture, the Proceedings of International Test conference, October 2002, pp. 923-929.
- [25] A. Ejlali, S.G. Miremadi, FPGA-based fault injection into switch-level models, Elsevier Microprocessors and Microsystems, 2004, Volume 28, pp. 317-327.
- [26] G.S. Choi, R.K. Iyer, FOCUS: an experimental environment for fault sensitivity analysis, IEEE Transaction Computers, December 1992, Volume 41, Issue 12, pp.1515-1526.

- [27] Raha Abedi, Synthesis of Classical and Non-classical CMOS Transistor Fault Models Mapped to Gate-level for Reconfigurable Hardware-based Fault Injection, Thesis Report, Ryerson University, 2005.
- [28] R. Javaheri, R. Sedaghat, J. Zalev, Verification and fault synthesis algorithm at switch-level, Elsevier Microprocessors and Microsystems, 2006, pp. 1-10.
- [29] Xilinx Documentation: XST User Guide:
http://toolbox.xilinx.com/docsan/xilinx6/books/data/docs/xst/xst0001_1.html
- [30] Xilinx Documentation: LogiCore OPB UART Lite, August 2004, DS422.
- [31] Xilinx Documentation: LogiCore OPB HWICAP (v1.00a), August 2004, DS280.
- [32] Xilinx Documentation: User Core Templates Reference Guide, January 2003.
- [33] Xilinx Documentation: ISE Quick Start Tutorial:
http://www.xilinx.com/support/sw_manuals/xilinx6/download/
- [34] Xilinx Documentation: Platform Studio Tools User Guide for EDK 6.3i
http://www.xilinx.com/ise/embedded/edk_docs.htm
- [35] Xilinx Documentation: Constraints Guide for ISE6.3i:
http://www.xilinx.com/support/sw_manuals/xilinx6/download/
- [36] Xilinx Documentation: Modular Design
http://toolbox.xilinx.com/docsan/xilinx7/books/data/docs/dev/dev0025_7.html
- [37] J. Alt, U. Mahlstedt, Simulation of Non-classical Faults on the Gate Level – Fault Modeling, the Proceedings of the 11th VLSI Test Symposium, April 1993, pp. 351-354.

Author's Publications:

1. M.H. Lee, R. Sedaghat, FPGA-based Switch-level Fault Emulation Using Modular-based Dynamic Partial Reconfiguration, will submit to Microelectronics Reliability Journal, ELSEVIER in May 2006.
2. M.H. Lee, R. Sedaghat, Using Modular-based Dynamic Partial Reconfiguration for Fast Switch-level Fault Injection, will submit to Microprocessors and Microsystems, ELSEVIER in May 2006.
3. M.H. Lee, R. Sedaghat, Fast Switch-level Fault Injection Campaign Using FPGA with Modular-based Dynamic Partial Reconfiguration Capability, will submit to ASP-DAC Conference in July 2006.