creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

[UCI]I804:24011-200000263298

August 2012

Master's Degree Thesis

A Self-Repairing Bio-Inspired Fault-Tolerant FPGA Architecture

Graduate School of Chosun University

Department of Computer Engineering

Hasan Baig

자가복구 기능을 갖춘 생태모방형 결함허용 FPGA 구조

A Self-Repairing Bio-Inspired Fault-Tolerant FPGA Architecture

August 24, 2012

Graduate School of Chosun University

Department of Computer Engineering

Hasan Baig

A Self-Repairing Bio-Inspired Fault-Tolerant FPGA Architecture

Advisor: Prof. Jeong-A Lee

This Thesis is submitted to Graduate School of Chosun University in partial fulfillment of the requirements for a Master's degree

April 2012

Graduate School of Chosun University

Department of Computer Engineering

Hasan Baig

베이그 하산 석사학위논문을 인준함

위원장 조선대학교 교수 모상만 (인)

위 원 조선대학교 교수 최광석 (인)

위 원 조선대학교 교수 이정아 (인)

2012년 5월

조선대학교 대학원

Table of Contents

Table of Contents i
List of Figures iii
List of Tables iv
List of Acronyms v
Abstract (Korean) vi

I. Introduction 1
A. Research Motivation 1
B. Research Objectives 2
C. Contributions 2
D. Thesis Organization 3

II. Related Works 4

III. Developed Fault-Tolerant FPGA Architecture 8

A. Brief Overview of a Complete Architecture 8
B. A Fault-tolerant FPGA Computation Cell 9
1. A Generic K-input Fracturable LUT 13
2. An Original-function LUT 13
3. A Spare-function LUT 14
4. Free LUT 15
5. On-cell EDC Generator 16
6. On-cell Fault-checker 17
7. Function-router and Permanent Error Indicator 18
8. The Programmable CMOS Switch Box 20
C. A Computation Block 21
1. Intra-block routing of permanent error signals 23
D. A Computation Tile 24
E. Intra-tile Routing for the Generation of a Single Permanent Error Signal 27
F. The Fault-tolerant Core 29
1. Flow of Permanent Error Handler 31
2. Internal Structure of a Priority Controller 31
3. The Frame Generator 33
G. The Self-repairing Software 35

IV. Architectural Comparison with Previous Work 37

V. Experimental Testing 41

VI. Conclusions and Future Enhancements 49

Bibliography 51

Abstract (English) **53**

Acknowledgment **55**

List of Figures

Fig. 1.1: The hierarchy of tissue-to-gene in a human body. 1
Fig. 2.1: Island-style FPGA architecture. 5
Fig. 3.1: A self-repairing bio-inspired fault-tolerant FPGA architecture. 8
Fig. 3.2: I/Os of fault-tolerant FPGA computation cell. 10
Fig. 3.3: Internal components of a fault-tolerant computation cell. 12
Fig. 3.4: The generic K-input LUT that can be fractured into two (K-1)-LUTs. . 13
Fig. 3.5: The I/Os of an “original-function LUT”. 14
Fig. 3.6: The I/Os of a “spare-function LUT”. 15
Fig. 3.7: The I/Os of a “free LUT”. 15
Fig. 3.8: The I/Os and internal structure of an “on-cell EDC generator”. 16
Fig. 3.9: The I/Os and internal structure of an “on-cell fault-checker”. 17
Fig. 3.10: The I/Os & internal structure of a “Fn-router and PE indicator”. 18
Fig. 3.11. The internal structure of a programmable CMOS switch box. 20
Fig. 3.12: The computation block. 21
Fig. 3.13: The intra-block routing of a permanent error signal 23
Fig. 3.14: Two neighbouring computation tiles 25
Fig. 3.15: The intra-tile routing of a permanent error signal. 28
Fig. 3.16: The components of a fault-tolerant core. 29
Fig. 3.17: The tile indices when die is not divided for error handling. 30
Fig. 2.18: The logic flow of a “permanent error handler”. 32
Fig. 3.19: The internal structure of a priority controller. 33
Fig. 3.20: The frame generator. 34
Fig. 3.21: The self-repairing software. 35
Fig. 5.1: Functional diagram of a test application. 42
Fig. 5.2: Floor plan of a test application. 43
Fig. 5.3: Real implementation of a Tile 1 44
Fig. 5.4: The implemented computation cell 4 of computation Tile 1. 45
Fig. 5.5: The experimental setup. 48

List of Tables

Table 4-1: Architectural Comparison with Previous Work 40
Table 5-1: Manual Reconfiguration Options in Prototype Software 46

List of Acronyms

ALFSR	Autonomous Linear Feedback Shift Register
CB	Computation Block
CC	Computation Cell
CLB	Configurable Logic Block
CMOS	Complimentary Metal Oxide Semiconductor
CT	Computation Tile
CUT	Circuit Under Test
DMR	Double Modular Redundancy
DNA	Deoxy-Ribonucleic Acid
EDC	Error Detection Codes
EOF	End Of Frame
FPGA	Field Programmable Gate Array
FF	Flip Flop
GUI	Graphical User Interface
ICAP	Internal Configuration Access Port
ILA	Integrated Logic Analyzer
JTAG	Joint Test Action Group
LabVIEW	Laboratory Virtual Instrument Engineering Workbench
LED	Light Emitting Diodes
LUT	Look-up Table
SC	Stem Cell
SEE	Single Event Effect
SOF	Start Of Frame
SRAM	Static Random Access Memory
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver-Transmitter

초 록

자가복구 기능을 갖춘 생태모방형 결함허용 FPGA 구조

하산 베이그
지도교수: 이정아 교수, Ph.D.
컴퓨터공학과
조선대학교 대학원

새로운 FPGA 구조를 제안할 때 반드시 고려해야 하는 부분이, 제안하는 새로운 구조로 인하여 추가적인 오버헤드를 발생시킬 가능성이 있는, 연결망 문제들이다. 자이링크 또는 알테라 와 같은 FPGA 제품 회사들이 결함허용과 같은 새로운 기능을 추가하기 위하여, 그들의 기존 제품 FPGA 연결망 구조를 수정하는 문제는 매우 어렵다. 따라서, 기존 제품의 연결망 구조와 잘 통합될 수 있도록, 결함허용 기능이 새롭게 추가되는 FPGA 구조를 개발하는 것은 중요한 문제이다. 본 논문에서는 상동(homogeneous)구조로서 자가 복구 기능을 갖춘 생태모방형 결함허용 FPGA 구조를 제안한다. 본 연구에서 제안하는 구조는, 지금까지 제안된 결함허용 구조들과 달리, 기존 FPGA 장치에서 사용하는 아일랜드 형 연결망과 잘 통합되는 구조로, 기존 FPGA 장치에서의 구현이 용이할 뿐 아니라, 더 나아가 필요하다면, 아일랜드 형 연결망 방식에 따라 결함허용 내부 연결을 갖춘 새로운 FPGA 장치로 제조될 수 있다. 본 논문에서 제안된 결함허용 FPGA 구조는 LUT 단위에서의 일시적 오류 및 영구적인 오류 발생 모두를 인지할 수 있다는 특성이 있다. 결함허용기능을 갖춘 범용 연산 셀은 내부적으로 자가 복구 회로와 결함이 없는 회로 출력을 외부로 보내기 위해 사용하는 내부 연결망으로 구성되어 있다. FPGA 구조는 여러 개의 연산 타일로 구성되어 있고, 각 연산 타일은 N 개의 연산 셀로 구성되어 있다. 제안된 구조는, 연산 타일 내부에서 발생될 수 있는 일시적 오류 및 영구적인 결함을 동시에 자가 복구할 수 있는 구조이다. 여러 연산 타일에서 결함이 발생되었을 경우, 온 칩 복구 제어부에서 중앙 집중적인 방식으로 UART 인터페이스를 통하여 외부 PC 소프트웨어에 해당정보를 보내어 복구가 진행된다. 외부 PC 소프트웨어는 결함이 발생한 연산 타일 내의 스템 셀 (프로그래밍 되지 않은 FPGA 로직 공간)을 부분적으로 다시 프로그래밍 함으로써 결함을 복구한다. 결함이 복구되는 동안, 결함이 발생되지 않은 FPGA에 구현된 시스템 작동은 계속 유지된다. 본 논문에서 제안된 자가복구 기능을 갖춘 생태모방형 구조는 자일링크 Virtex-5 FPGA 장치에서 구현되었고, 이의 안정적인 작동을 확인하였다. 또, TMR 및 최근에 제안된 결함허용 구조와 비교하여 볼

때, 본 논문의 제안된 구조가 결함 허용을 위한 하드웨어의 추가적인 오버헤드가 훨씬 축소될 뿐 아니라 여러 측면에서 향상된 구조임을 보였다.

I. Introduction

A. Research Motivation

A self-repairing fault-tolerant system in a human body inspired us to implement the same logic at chip-level. The tissue, in a human body, is a group of cells which together carry out a specific function. The cell is the basic structural and functional unit of all known living organisms. It is often called the basic building block of life. Each cell is made up of several organelles like mitochondria, ribosome, nucleus etc. The cell's nucleus contains chromosomes made from long DNA molecules. DNA is further divided into a group of nucleotide sequences called genes, as shown in Fig. 1.1.

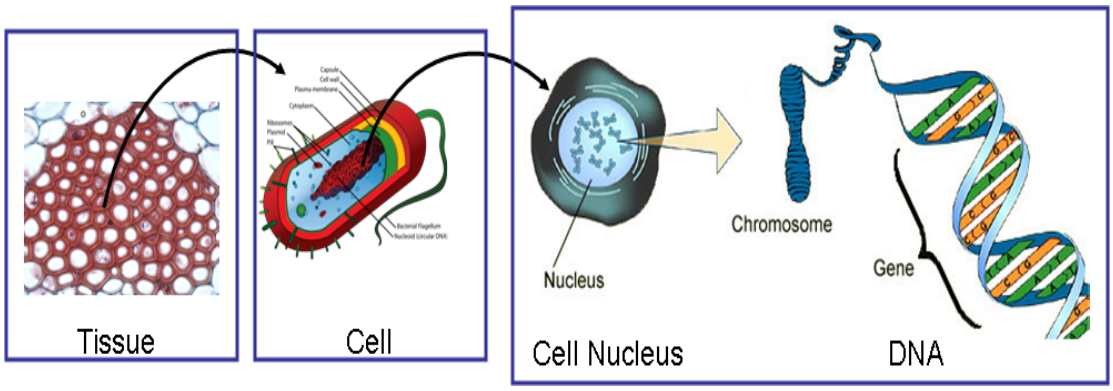


Fig. 1.1: The hierarchy of tissue-to-gene in a human body.

The genes that have a same DNA sequences are termed as Paralogous genes. These similar genes generally perform the same function. For instance, out of two Paralogous genes, A and B, if A goes faulty then B takes over the functionality immediately. However, if both of them are faulty then the cell dies. The cellular differentiation process of stem (or empty) cells then comes into an action to take over the functionality of that dead faulty cell. Such a fault-tolerant mechanism exists in living organism in which the sufficient DNA sequences are backed-up as redundant genes, and stem cells to heal the faulty cells.

The same approach has been implemented to develop a fault-tolerant FPGA architecture, in which the computation tile corresponds to a tissue and the pair of original and spare functions, within the cell, plays a role of paralogous genes. We also have employed the concept of stem cells differentiation (reprogramming) on FPGA through dynamic partial reconfiguration to recover permanent faults.

B. Research Objectives

The most complex thing in any FPGA architecture is its routing network, and to propose a new FPGA architecture means to keep all the routing issues in mind while developing a new FPGA architecture. It should also be keep in mind that 80-90% of the FPGA area is occupied by routing resources and the rest 10-20% is used by logic. Thus it is also difficult for the commercial FPGA vendors, like XILINX, Altera etc, to refine their existing routing architecture according to the newly proposed schemes to incorporate the fault tolerant capabilities. This arises the need to develop such a design that can easily be integrated with the existing routing architecture.

These facts led us to invent a generic fault-tolerant FPGA *computation cell (CC)* which can be arrayed over the whole die to buildup a symmetrical fault-tolerant FPGA architecture. This architecture can be integrated easily with the existing island style FPGA routing architecture.

C. Contributions

In this research, a generic fault-tolerant FPGA computation cell has been developed that can not only be able to serve as a basic building block in making a new fault-tolerant FPGA device, but can also be able to implement in an existing FPGA devices without taking care of routing issues. A novel fault-tolerant FPGA architecture, along with its controlling fault-tolerant core and self-healing software, is also developed in which the invented fault-tolerant computation cell can be efficiently utilized for the development of

fault-tolerant digital hardwares. The main contributions of this thesis are as follows

1. A fault-tolerant computation cell
2. A computation block and its components
3. A computation tile and its components
4. An intra-tile routing architecture for the generation of permanent error signal
5. A fault-tolerant core along with its components
6. An external PC software for
    a. Introducing errors in FPGA chip during run time (for testing purpose)
    b. Self-healing the stuck-at permanent faults occurring in FPGA device
7. A method of healing number of permanent faults, in a computation tile, at a time
8. A method of healing faulty tiles if an error occurs in more than one of them at a time

D. Thesis Organization

The rest of this thesis is organized as follows. The overview and previous work related to this topic has been presented in chapter II. The detailed description of proposed architecture, fault-tolerant computation cell, fault detection and self repairing mechanism, fault-tolerant core and the self-reconfiguring external PC software has been presented in chapter III. Chapter IV draws the performance comparison of a proposed architecture with the previous work. Chapter V comprises of the experimental testing. Chapter VI concludes the research work with future enhancements.

II. Related Work

Fault tolerance on semiconductor devices has been a meaningful matter since upsets were first experienced in space applications several years ago. Since then, the interest in studying fault-tolerant techniques in order to keep integrated circuits (ICs) operational in such hostile environment has increased, driven by all possible applications of radiation tolerant circuits, such as space missions, satellites, high-energy physics experiments and others (Nasa, 2003).

A Field Programmable Gate Array (FPGA), customizable by SRAM, device is a type of integrated circuit which consists of an array of programmable logic blocks interconnected by a programmable routing network and I/O blocks. SRAM based FPGA devices are getting popular in remote missions because of their high performance, reduced development cost and re-programmability.

Fig. 2.1 shows the Reconfigurable island-style FPGA Tile. The blocks shown in *purple* are I/O Blocks or I/O pads through which FPGA chip takes input or produce output. The blocks shown in *green* are Switch or S blocks and are responsible to interconnect horizontal and vertical channels (shown with *red* vertical and horizontal lines). They only switch the direction of the signals from horizontal to vertical path or vice versa. The *blue* blocks are Configurable Logic Blocks (CLBs) or sometimes called Basic Logic Elements (BLEs) or simply called Logic Blocks. *Grey* blocks, in between each two Switch blocks, are Connection or C Blocks. C blocks connect the signals between two BLEs via switch blocks. The red interconnects are the routing channels which occupies 80-90% area of FPGA device.

Radiations in the environment can seriously affect the functionality of a circuit. Single Event Effect (SEE) occurs when a charged particle, present in the environment, hits the silicon, transferring enough energy in order to provoke a fault in the system. SEE can have a permanent (destructive) or temporary (transient) effect depending on the amount of energy transferred by the

charged particles. The main consequences of transient effect are a bit flips in the memory elements whereas stuck-at faults can be an upshot of a permanent effect. In order to keep these programmable devices operational in such hostile environment where the human intervention for maintenance and repair is impossible, a promising fault-tolerant and a self-healing reconfigurable architecture is considered necessary.

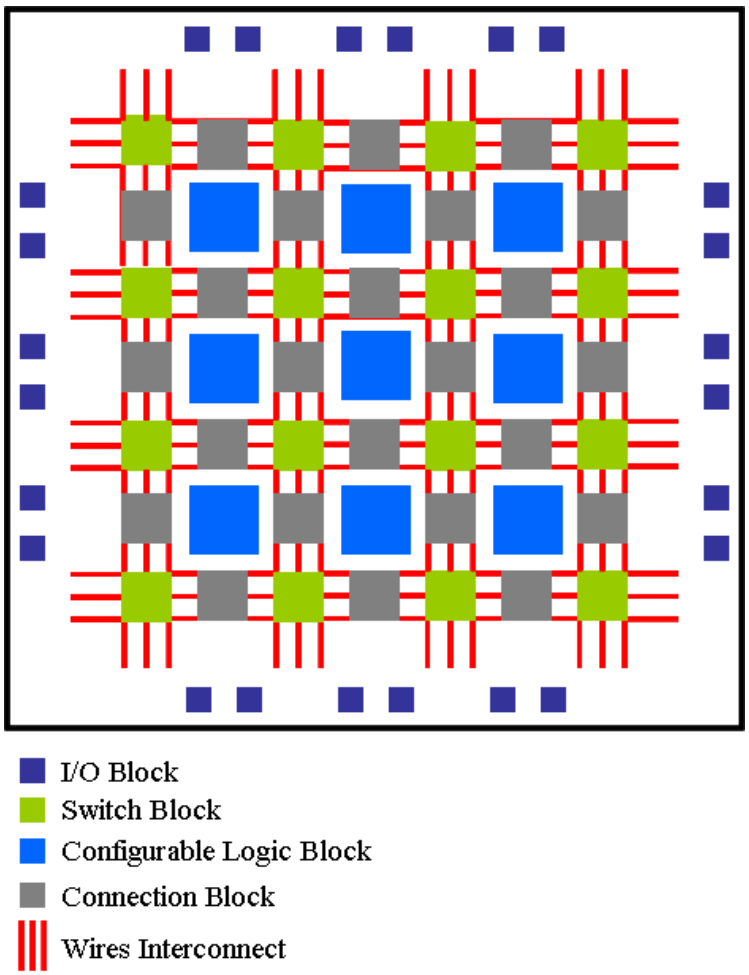


Fig. 2.1: Island-style FPGA architecture.

The typical approach of fault-tolerance is the use of redundancy where functions are replicated by n versions of protected hardware. Embryonics is taking this a stage further through the cellular organization and replication of hardware elements [13] [14]. The biological approach to fault tolerance is in the form of highly dependable distributed systems with a very high degree of redundancy. The human immune system protects the body from invaders,

preventing the onset of chemical and cellular imbalances that may affect the reliable operation of the body. Similarities between the human immune system and the requirements of fault tolerant system design were first highlighted by Avizienis [15] who noted the potential analogies between hardware fault tolerance and the immune system. Use of the immune system as an approach to fault tolerance within systems was first noted and demonstrated in [16] for the design and operation of reliable software systems.

Several fault-tolerant techniques have been studied in the past years to protect ASICs against transient faults, and because FPGAs are composed of combinational and sequential logic, and more recently embedded processors, previous work dealing with standard integrated circuits can be adapted to the programmable logic architecture by finding the best tradeoff among area overhead, performance penalties, single and multiple upset correction, process technology and implementation cost.

Tandem and Stratus used a fault-tolerant system called triple modular redundancy (TMR) in which three same hardware modules are implemented to carry out the same function and then a majority output is always used [1], [2]. In TMR, fault-detection is not necessary whereas only one fault is allowed. However, the high cost of its implementation limits its range of utilization. The concept of self-repairing embryonic systems was first introduced in the 1990s [3], [4]. Since then, self-repairing digital systems, as an advanced form of fault-tolerant systems, have received increasing attention, as modern digital systems are getting more complex and fast [5], [6]. In contrast to TMR, fault-detection modules are required in self-repairing systems [17]. In addition, cells without any specified function are initially implemented as logic blocks or routing resources in a fine-grained scale and they are programmed later to replace faulty cells in the case of failure. This is the primary difference compared to the fault-tolerant systems where hardware redundancies should be made in advance in a coarse-grained scale at a chip or a board level.

A number of researches have been carried out regarding self-repairing digital systems, including [7]–[9], in which a logical system realized by lookup tables

(LUTs) have been suggested. Some techniques have been suggested in the direction of precompiled configuration [7], [18]. In such techniques, alternative configuration was prepared in advance to replace permanent faulty parts and therefore reprogramming was not necessary. However, there was a severe overhead to prepare all configuration versions required to cover every possible faulty case. As a result, the storage overhead to keep various precompiled configuration versions gets increased enormously despite the current advancement of data compression techniques. A dynamic partial reconfiguration has also been studied for fast fault recovery [10], [11] in which the faulty partition of an FPGA is reconfigured without stopping the overall operation. The FPGA resources must be divided into fixed and dynamic parts where the fixed parts include essential elements for a constant operation and the dynamic parts include the dynamically reconfigurable elements. In addition, all the signal lines of the reconfigured elements should keep the same position. The hybrid scheme of wiring a redundant and a stem (empty) cell for fault-recovery is proposed by Kim, S. et al. in [12]. In this framework, if a fault occurs in a working cell, the corresponding redundancy replaces it instantly and the normal operation can be restored immediately. If there is no such corresponding redundancy for a working cell, a stem cell is immediately configured to the redundancy.

The in-depth comparison of our proposed architecture with that of previous architectures has been presented in chapter IV - "*Architectural Comparison with Previous Work*".

III. Developed Fault-Tolerant FPGA Architecture

A. Brief Overview of a Complete Architecture

The proposed architecture (depicted in Fig. 3.1) is an array of *computation tiles* (CT). Each computation tile is further divided into a number of *computation blocks* (CB) each of which contains *M* number of *computation cells* (see section II-C for general formula). Each computation block has its own *stem cell* which contains *N* stem functions for its corresponding computation cells. Since the permanent error condition is handled by downloading a fresh partial bitstream so a fault checking mechanism is not required for stem functions.

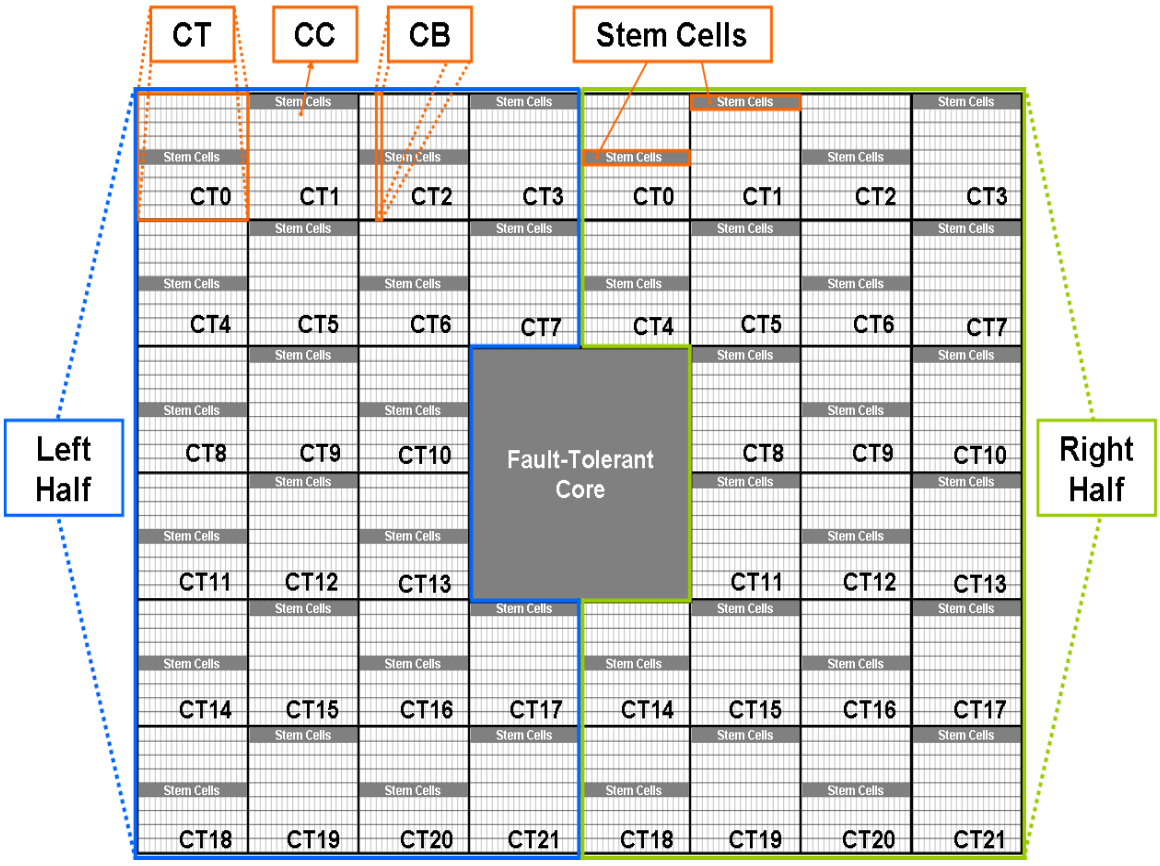


Fig. 3.1: A self-repairing bio-inspired fault-tolerant FPGA architecture.

There are two kinds of symmetry present in the neighboring tiles. Since the region of stem cells is partially reconfigurable, and due to the fact that the wiring nets can not pass through the partially reconfigurable region, this is why the different placements of stem cells is kept in neighboring tiles. If they would have arranged along a same row, the right and left half of each tile will not only be able to communicate with itself but also not with a neighboring tile. The die is divided into two halves. Also each tile is marked by its unique ID CT0, CT1 …, CT21, for both halves of the die to identify which tile has a permanent fault.

The permanent errors occurring in the computation tiles are handled by a *fault-tolerant core*. In order to use the existing architecture with the fault-tolerant capabilities, this core must be initialized and placed at the die-centre. Its purpose is to define the healing priority when the error occurs in more than one of the tiles at the same time. The *fault-tolerant core* sends the information to external PC software via UART interface which identifies the faulty tile and downloads its corresponding partial bitstream to reconfigure that particular tile. This process takes no more than 1 to 2 seconds. The stem cell consists of *N* different functions corresponding to *M* different computation cells. The stem cell is partially reconfigured during run time whenever both of the original and spare function goes faulty at the same time.

B. A Fault-tolerant FPGA Computation Cell

Fig. 2.2 shows the top-level input-output diagram of a fault-tolerant FPGA computation cell. The inputs to a function are depicted by 1 i.e., a K-input function can be implemented. The inputs 2 are the inputs to free LUTs that can either be used as an intra-tile router or can also be used to store pre-generated error detection codes. This is further described with the description of Fig. 2.3. The inputs 3 are the inputs to a free-router available inside the cell. The signal Fn_{stem} connects the internal function-router with its corresponding stem function (see section II-*C*). The outputs 4 are the cell outputs.

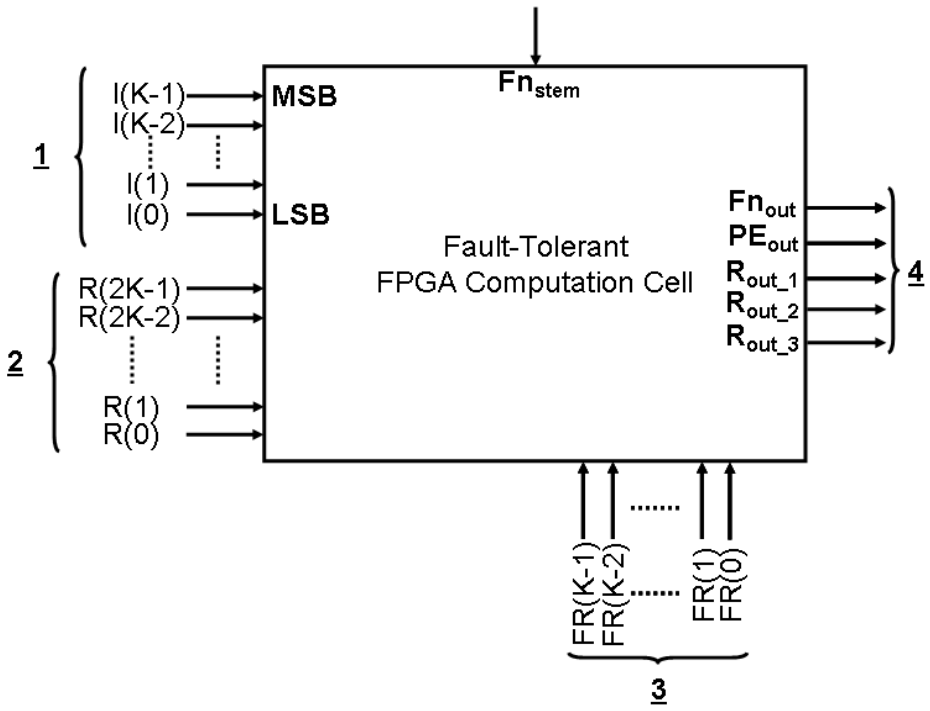


Fig. 3.2: I/Os of fault-tolerant FPGA computation cell.

The purpose and the signal naming convention of each output signal is mentioned below

- Fn_{out} (Function output) routes the cell function to another cell
- PE_{out} (Permanent Error output) generates an error indicating signal
- R_{out_1} (Route Out 1) and R_{out_2} (Route Out 2) are used for intra-tile routing
- R_{out_3} (Route Out 3) is an output from a free router

The invented fault-tolerant computation cell is capable of detecting all unidirectional errors in a function and able to replace it with its redundant spare function.

The cell consists of an original-function LUT, a spare-function LUT (which is used to replace the original function when a transient error occurs) and a pre-computed error detection codes (EDC), stored either along with the original/spare functions or in a separate two LUTs depending on the number of inputs of a

function. Beside these four reconfigurable LUTs in a developed computation cell, there is a self checking circuitry which not only identifies the error in an original-function but also in a spare-function, both at the same time. It is very least probable that both of the original and spare function gets faulty at the same time, but if it happens (which indicates the permanent error condition) then the proposed architecture is smart enough to replace its functionality with the *stem cell* through partial reconfiguration. A computation cell also consists of an internal router to route un-faulty function (an original function or a spare function or stem function) out of the cell.

The present invention assumes that the pre-computed Error Detection Codes (EDC), for each specific function, are loaded in the corresponding *computation cells* while configuring the FPGA with a configuration bitstream. These pre-stored EDCs are then verified with the on-cell generated EDCs for validating the output of original or spared function as correct or faulty.

Fig. 2.3 depicts the internal components of a fault-tolerant computation cell. Components 5, 6, 7 and 8 are reconfigurable K-input LUTs whereas the components 9, 10, 11 and 13 can either be hardened at CMOS level or can be implemented in the existing FPGA in K-input LUTs. If a function to be ported in a computation cell is K-input function then 5 and 6 LUTs can be utilized as whole to store original and spare functions respectively. However, when a function to be ported in a computation cell is of K-1 (or less) inputs, both 5 and 6 can be fractured into two halves – one half of each component store original and spare functions and the other half store the corresponding EDC, as shown in Fig. 2.3. The components 7 and 8 are used to store EDC for a former case or as a free router in a later one. The components 9 and 10 are collectively known as Totally Self Checker, depicted as 12 in Fig. 2.3.

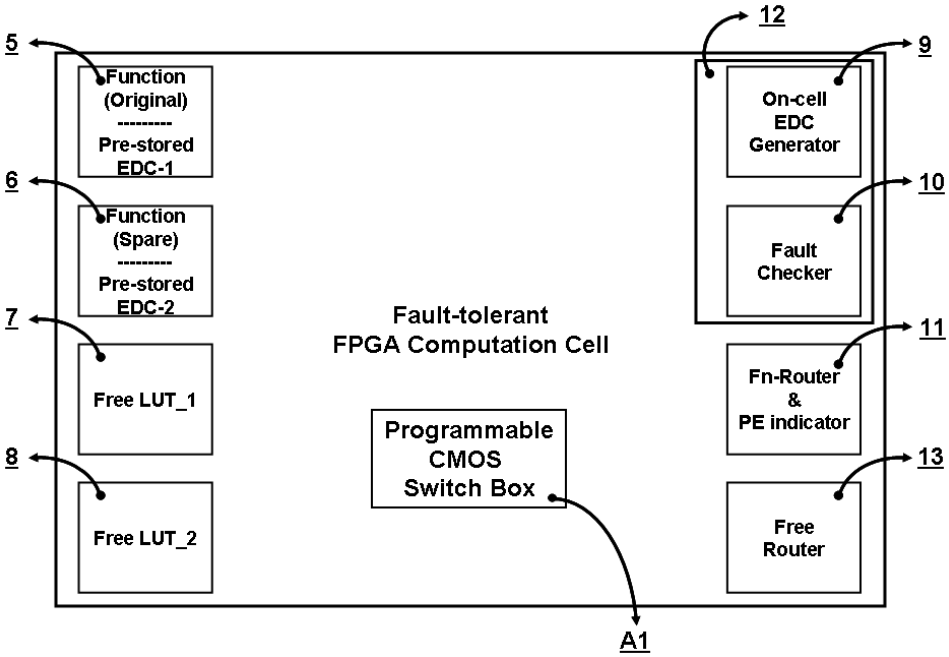


Fig. 3.3: Internal components of a fault-tolerant computation cell.

The concept of a *totally self checking (TSC)* mechanism is defined as

Definition 1: A cell is self-testing if, for every fault in a function, the generated EDC for a function doesn't match with at least one pre-stored EDC.

Definition 2: A cell is fault-secure if, for every fault in a function, the cell never produces an incorrect code output for any valid input.

Definition 3: A cell is totally self-checking if it is both self-testing and fault secure.

The component 11 is used to route non-faulty signal out of the cell. It also generates a flag of permanent error which indicates that both the original and spare functions are faulty at the same time (a permanent error condition). The component 13 is used as a free router to route different permanent error signals either from it or from the neighboring cells. It is simply an OR gate which can take up to K-inputs and route a single output. The component A1 is optional and must be used only when 9, 10, 11 and 13 needs to be hardened at CMOS level. It is

further described in section II-B.*8*. The internal components of a *computation cell* are described in the following sub-sections.

1. A Generic K-input Fracturable LUT

Fig. 2.4 shows the generic K-input LUT that can be fractured into two K-1 shared input LUTs as shown. All the latest FPGAs nowadays have at least 6-input fracturable LUTs. If the invented architecture needs to be ported in an existing FPGAs then the target device should have at least 6-input LUTs in a Configurable Logic Block (CLB) to implement components 9, 10, 11 and 13 of Fig. 2.3. There is no such restriction on reconfigurable components 5, 6, 7 and 8 of Fig. 2.3.

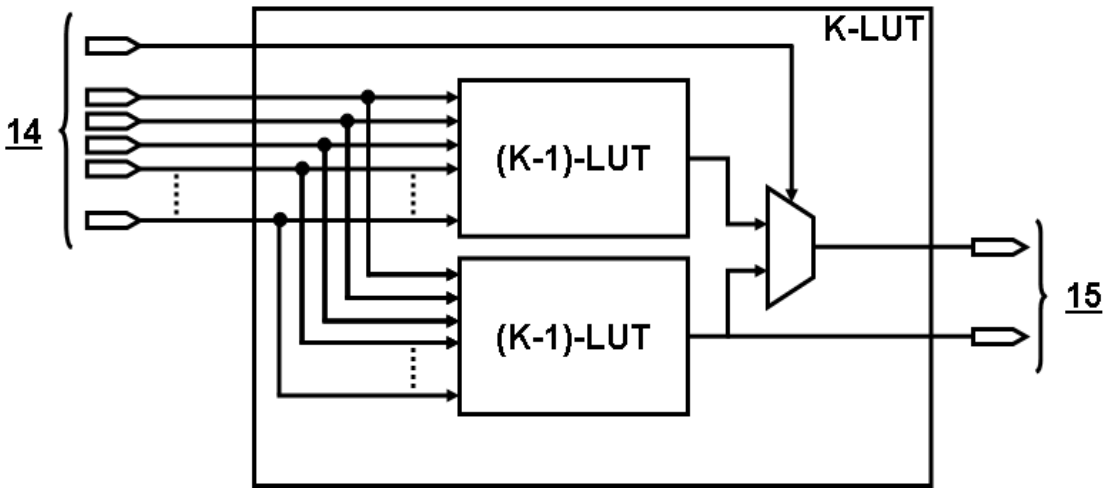


Fig. 3.4: The generic K-input LUT that can be fractured into two (K-1)-LUTs.

2. An Original-function LUT

Fig. 2.5 shows a 6-input (16) LUT (taken as an example) to hold an original function. This LUT can be fractured into two 5-input (shared) LUTs. If a function to be ported has 6 inputs then the LUT is utilized as a whole otherwise the

upper half, 19, stores the original function and the lower half, 20, stores its corresponding error detection codes.

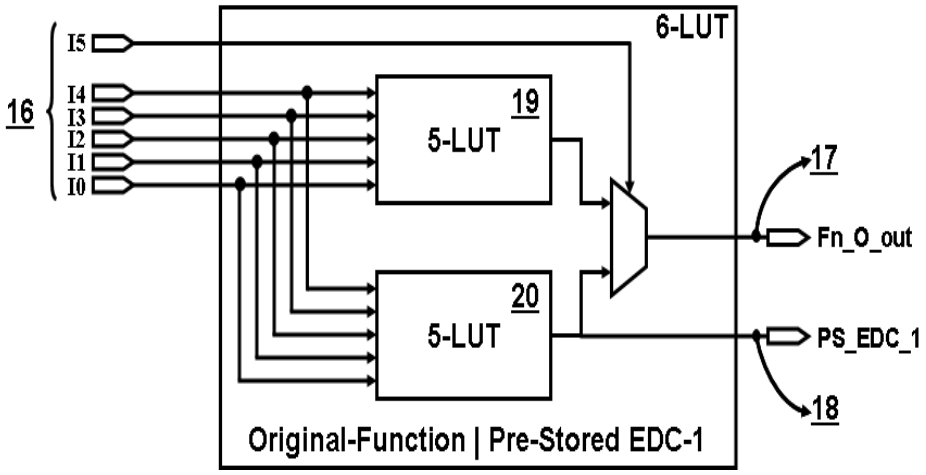


Fig. 3.5: The I/Os of an "original-function LUT".

The output 17 and 18 are the original function and EDC outputs respectively. The signal naming conventions are mentioned below

- *Fn_O_out* (Original Function output)
- *PS_EDC_1* (Pre-Stored EDC-1 to check original function)

3. A Spare-function LUT

Fig. 2.6 shows a 6-input (21) LUT (taken as an example) to hold a spare function. If a function to be ported has 6 inputs then the LUT is utilized as a whole otherwise the upper half, 24, stores the spare function and the lower half, 25, stores its corresponding error detection codes. The outputs 22 and 23 are the spare function and EDC outputs respectively. The signal naming conventions are mentioned below

- *Fn_Sp_out* (Spare Function output)
- *PS_EDC_2* (Pre-Stored EDC-2 to check spare function)

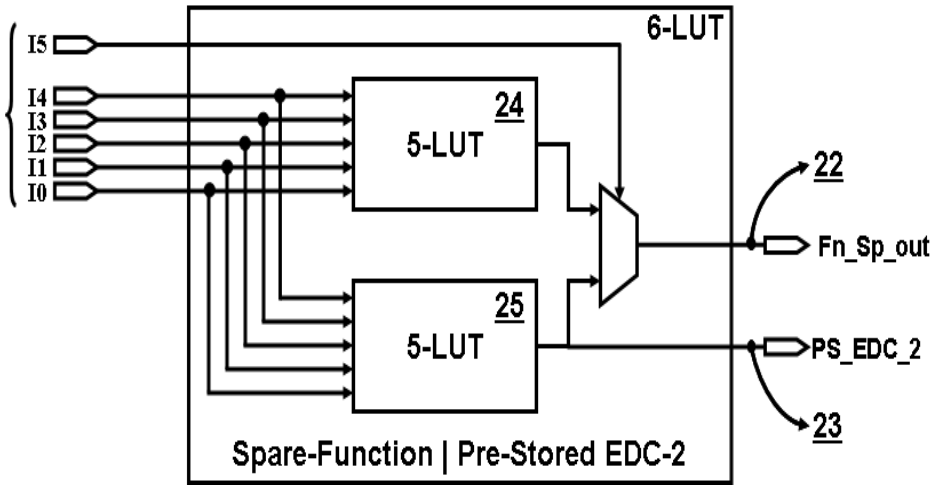


Fig. 3.6: The I/Os of a “spare-function LUT”.

4. Free LUT

Fig. 8 shows a 6-input (26) LUT (taken as an example) to be used either as an intra-tile router or to hold 6-input EDCs. If a function ported in Fig. 6 is a 6-input function then 28 will be utilized to hold a corresponding 6-input EDC. If it is less than 6-inputs function, then 28 will be employed as an intra-tile router to route *Permanent Error* signal (discussed in section II-C and II-*E*).

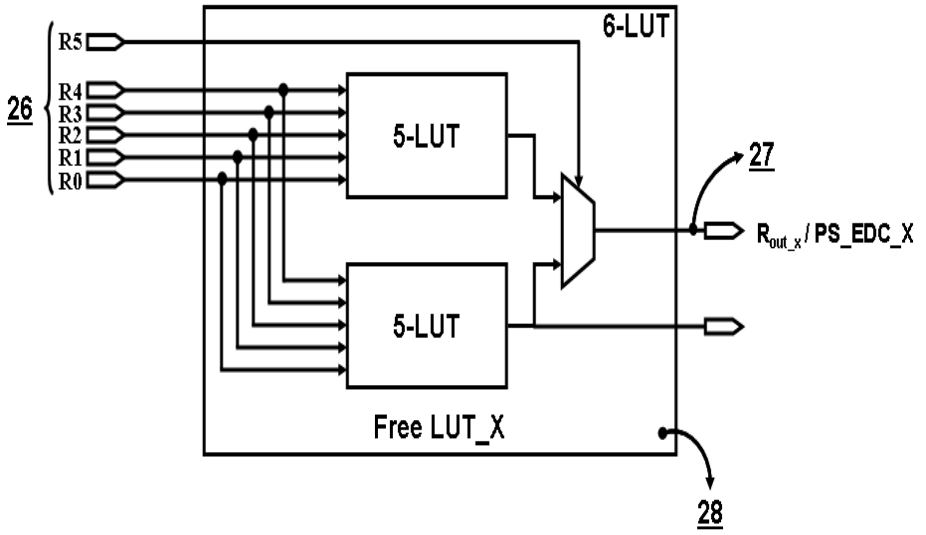


Fig. 3.7: The I/Os of a “free LUT”.

The signal naming convention of output 27 in both the cases are mentioned below

- R_{out_x} (Route Out 1 or Route Out 2 of components 7 and 8 respectively of Fig. 4)
- *PS_EDC_X* (Pre-Stored EDC-1 or Pre-Store EDC-2 for components 7 and 8 respectively of Fig. 2.3)

5. On-cell EDC Generator

Fig. 2.8 portrays the internal structure and input/outputs of an "on-cell EDC generator". The output 17 and 22 of Fig. 6 and Fig. 7, in order, connects here at the inputs 49 and 50 respectively. The signal naming convention of outputs 51 and 52 are mentioned below

- *EDC_G_O* (Generated EDC for Original function)
- *EDC_G_Sp* (Generated EDC for Spare function)

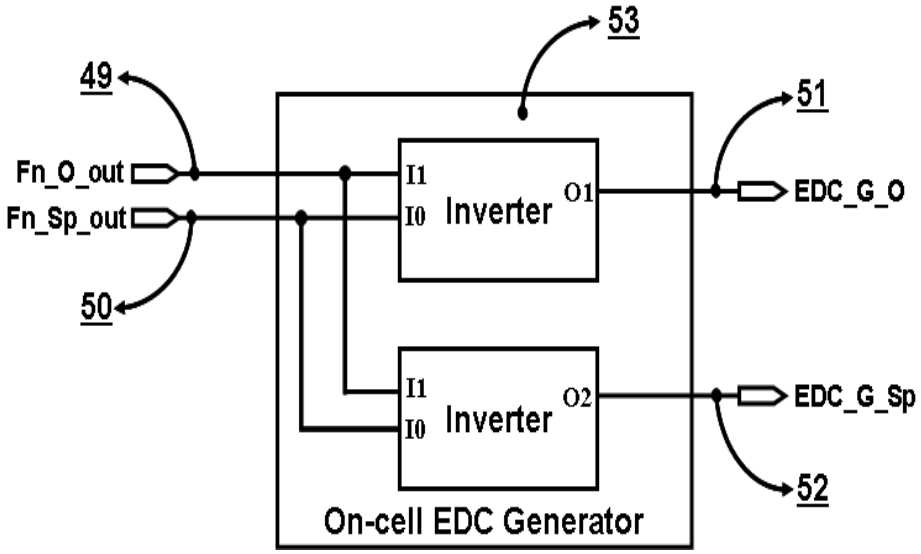


Fig. 3.8: The I/Os and internal structure of an "on-cell EDC generator".

The existing architecture of a fracturable LUT of some vendors is able to implement more than 1 function in a same LUT, as long as the inputs are common to all of them (like XILINX devices). However, other vendors provide the flexibility of fracturable LUT that can even implement more than 1 function

having unshared inputs (like Altera devices). So in order to make the present invention compatible with all the existing commercial devices we have designed all the sub-modules of a fault-tolerant cell which implements different functions with shared inputs. The output equations of 51 and 52 are mentioned below

$$\mathbf{O}1 = \overline{\mathbf{I}1} \tag{2.1}$$

$$\mathbf{O}2 = \overline{\mathbf{I}0} \tag{2.2}$$

6. On-cell Fault-checker

Fig. 2.9 depicts the internal structure and input/outputs of a "fault checker". The outputs 51 and 18 of Fig. 9 and Fig. 6**,** and 52 and 23 of Fig. 9 and Fig. 7, in order, are routed here at the inputs 56, 57, 58 and 59 respectively. The signal naming convention of outputs 60 and 61 are mentioned below

- *Faulty_O* (indicates that an Original function is Faulty)
- *Faulty_Sp* (indicates that a Spare function is Faulty)

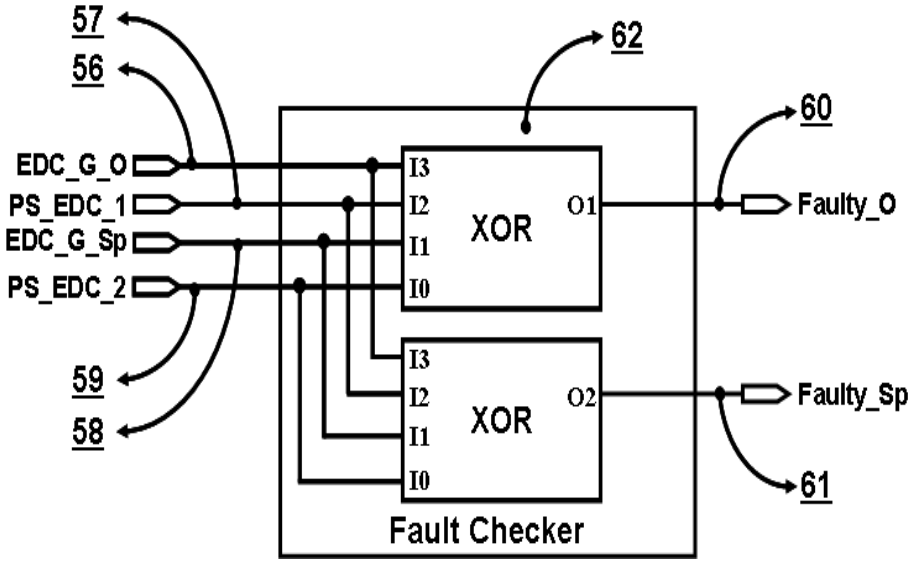


Fig. 3.9: The I/Os and internal structure of an "on-cell fault-checker".

Depending on the number of inputs to an original function (as discussed above) 62 takes either the outputs 18 and 23 of Fig. 6 and Fig.7, in order, or it take the outputs from components 7 and 8 of Fig. 4 at its inputs 57 and 59 respectively. It indicates whether an original or a spare function is faulty via outputs 60 and 61 respectively. In this module, the real-time generated EDCs are compared with the pre-stored EDCs to check whether the generated and pre-stored EDCs are same. The non-matching value indicates a fault either in original or in a spare function. The output equations of 60 and 61 are mentioned below

$$\mathbf{O}1 = \mathbf{I}3 \oplus \mathbf{I}2 \tag{2.3}$$

$$\mathbf{O}2 = \mathbf{I}1 \oplus \mathbf{I}0 \tag{2.4}$$

7. Function-router and Permanent Error Indicator

Fig. 2.10 shows the internal structure of a "Function-Router and PE Indicator". It take the outputs 60 and 61 of Fig. 2.9, 17 of Fig 2.5, 22 of Fig. 2.6, and input Fn_{stem} of Fig. 2.2, in order, at its inputs 67, 68, 69, 70 and 71 respectively. It routes the un-faulty function and generates permanent error signal via outputs 72 and 73 respectively.

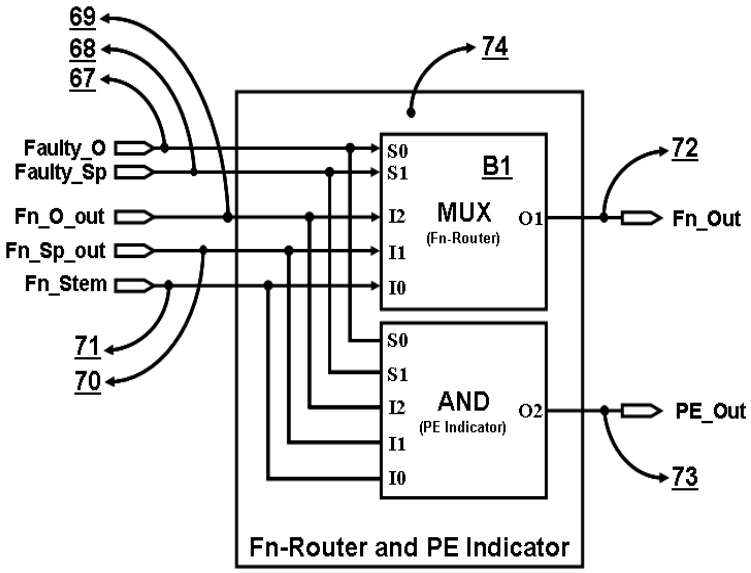


Fig. 3.10: The I/Os & internal structure of a "Fn-router and PE indicator".

The signal naming convention of outputs 72 and 73 are mentioned below

- *Fn_Out* (route the un-faulty Function Out from the cell)
- *PE_Out* (route a Permanent Error flag signal Out from the cell)

The component B1 (Fig. 2.10) is a MUX whose select signals are routed with the fault indication signals generated from a “Fault Checker”. The output behavior of 72 and 73 is mentioned empirically in equation (2.5) and (2.6) respectively.

$$\mathbf{O}1 = \overline{\mathbf{S0}} \cdot \overline{\mathbf{S}1} \cdot \mathbf{I}2 + \overline{\mathbf{S0}} \cdot \mathbf{S1} \cdot \mathbf{I}2 + \mathbf{S0} \cdot \overline{\mathbf{S}1} \cdot \mathbf{I}1 + \mathbf{S0} \cdot \mathbf{S1} \cdot \mathbf{I}0 \quad (2.5)$$

$$\mathbf{O}2 = \mathbf{S0} \cdot \mathbf{S}1 \quad (2.6)$$

When,

1. none of the original or spare function is faulty (S0 = S1 = 0)
    - Original-function is routed out of the cell

2. only a spare function is faulty (S0 = 0, S1 = 1)
    - Original-function is routed out of the cell

3. original function is faulty and spare function is un-faulty (S0 = 1, S1 = 0)
    - Spare-function is routed out of the cell

4. original and spare both functions are faulty (S0 = 1, S1 = 1)
    - Stem function is routed out of the cell
    - Permanent Error signal is generated

8. The Programmable CMOS Switch Box

Fig. 2.11 shows the internal structure of a CMOS switch box that must be fabricated at CMOS level if and only if the components 9, 10, 11 and 13 of Fig. 4 also needs to be fabricated at CMOS level.

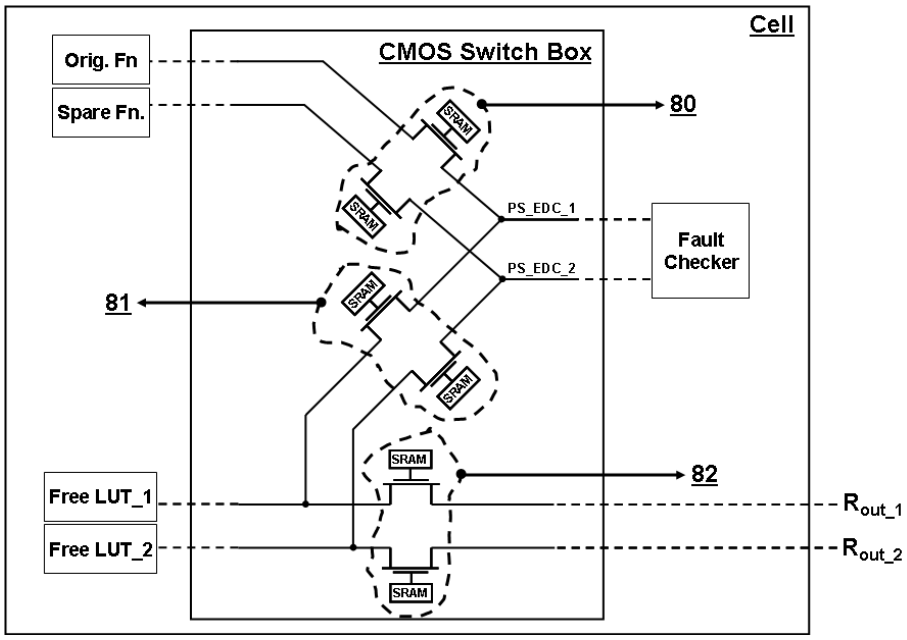


Fig. 3.11. The internal structure of a programmable CMOS switch box.

If the proposed architecture is implemented entirely as a new device, using the existing routing architecture, then the above mentioned components must be hardened at CMOS level and there should be an internal switch to route the appropriate signals with component 10 of Fig. 2.3.

As mentioned before, when a K-input function is configured in fault-tolerant cell, the pre-generated EDC-1 and EDC-2 will be stored in the LUTs 7 and 8 of Fig. 2.3 respectively. In this case, the SRAM bits of 80 and 82 are configured 0, and that of 81 are configured 1. Similarly, when a function, of less than K inputs, is configured then pre-generated EDC-1 and EDC-2 will be stored in the LUTs 5 and 6 of Fig. 2.3 respectively. In this case, the SRAM bits of 80 and 82 are configured 1, and that of 81 are configured 0. The CMOS switch box is software

programmable (just like a routing network) and configured during implementation process.

C. A Computation Block

Fig. 2.12 shows the *computation block* with two different placements and routings of stem cell, 83, with its corresponding computation cells. The stem cell contains 8 different functions (in this example) of its corresponding 8 *computation cells*. Thus a single stem cell in a computation block is able to heal all of its corresponding computation cells. In Fig. 2.12(a), 84 A and 84 B shows two *computation cells* with their corresponding stem functions St-A and St-B respectively. Whenever a *computation cell* gets a permanent fault, these stem functions are partially reconfigured to take over the functionality of their corresponding computation cells.

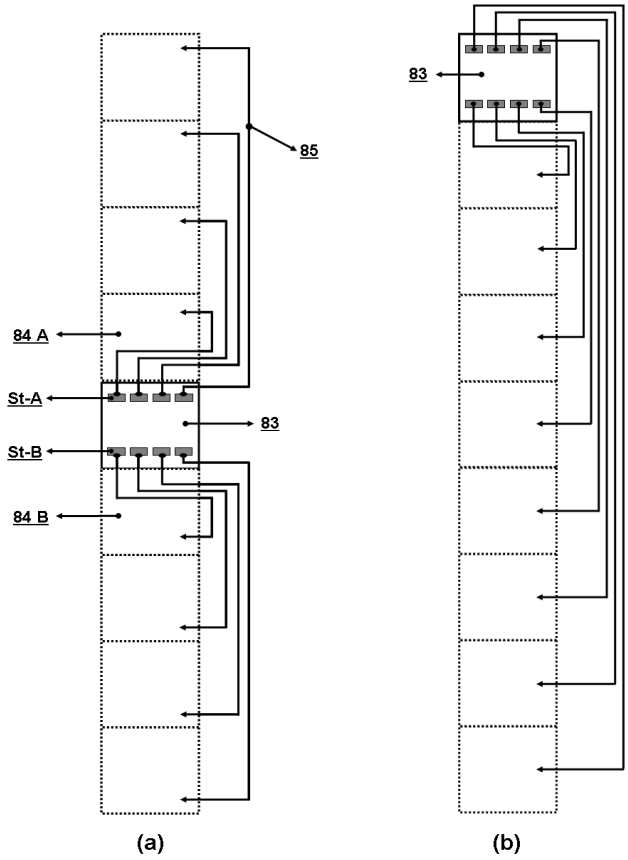


Fig. 3.12: The computation block – showing stem function connections with their corresponding computation cells when the stem cell is placed at (a) center (b) corner.

The signal 85 is actually the signal Fn_{stem} (in Fig. 2.2) which comes from the stem cell, shown as 83 in Fig. 2.12.

The width of a *computation block* depends on the number of LUTs a computation cell can have. In order to port this fault-tolerant architecture in an existing FPGA then the width of a computation block depends on the number of LUTs a Configurable Logic Block (CLB) can have. It should have at least 8 LUTs in order to port all the components of a computation cell in a single CLB.

Here we have assumed that each computation cell contains 8 LUTs so 8 stem functions for each computation cell can easily be accommodated. Thus a computation block size here is 8 + 1 (1 for stem cell) = 9 cells. It can be noticed that for 8 computation cells, only 1 stem cell is needed. If the present architecture is fabricated at CMOS level then block size totally depends on the size of a stem cell, in terms of LUTs. To port a proposed architecture in existing FPGA, 1 computation cell corresponds to 1 CLB. The general formula of computation block size (in terms of CLB) to port a present fault-tolerant architecture in an existing FPGA is

$$B = N + W \text{ } (CLBs) \tag{2.7}$$

where,

B is the computation block size;

N is the number of LUTs in a CLB, where $N \geq 8$;

W is the configuration frame width. The width of a stem cell should exactly be equal to or greater than the width of a configuration frame in order to partially reconfigure it properly.

The designer should take care of the width of a stem cell while choosing the type of partial reconfiguration. In *modular-based* partial reconfiguration, the software automatically inserts proxy logic. This proxy logic is a single LUT1 element for each partition pin as a fixed interface between static and reconfigurable region. In

such a case, the width of a stem cell should be large enough to accommodate all partition pins of a stem cell. However, this is not the case with the *differential-based* partial reconfiguration where there is no such requirement of extra proxy logic insertion.

1. Intra-block routing of permanent error signals

Fig. 2.13(a) and (b) shows the routing of a permanent error signal within a computation block, when a stem cell is placed at its center and at the corner, respectively. The components 86 show the permanent error indicators, i.e., a component 11 of Fig. 2.3, of two different computation cells.

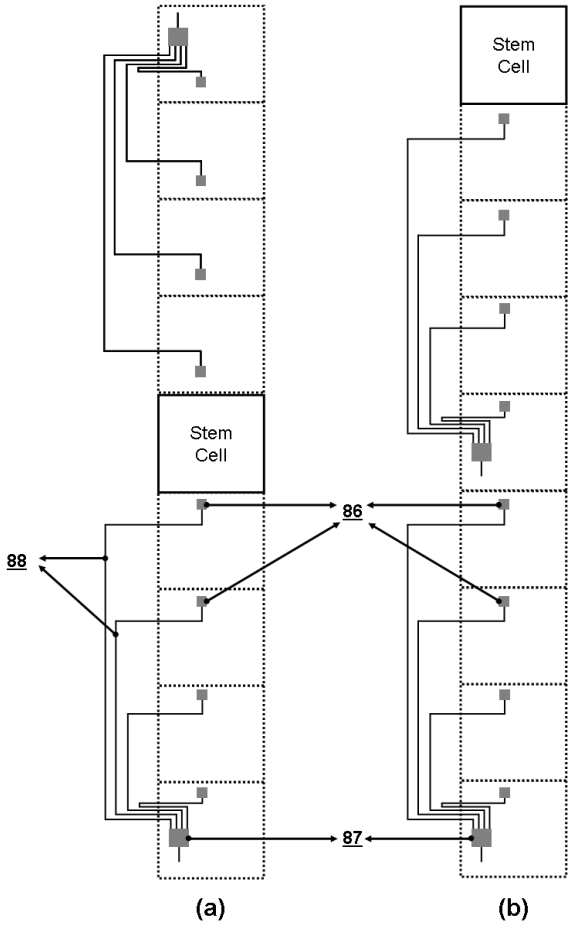


Fig. 3.13: The intra-block routing of a permanent error signal when a stem cell is placed at (a) centre (b) corner.

The signals 88 are the routes taken out from the permanent error indicator from each computation cell and routed into the inputs of 87, which could either be a component 7/8 or a component 13 of Fig. 2.3.

The components 7/8 must be in a free router mode. The same routing strategy is followed on the other half of a computation block, as depicted in Fig. 2.13(a) and (b). This shows that, at least two computation cells out of a single computation block must have less than K input function, in order to use component 7 and 8 of Fig. 2.3 as a free router. So, for 8 computation cells in a computation block, 8 **–** 2 = 6 K-input functions can be implemented in it. For 10 computation cells in a computation block, 10 **–** 2 = 8 K-input functions can be implemented in it. Generally the number of implementable K-input functions in a block is,

$$\text{Number of K-input functions} = M - 2 \tag{2.8}$$

where,

M is the number of computation cells in a computation block.

D. A Computation Tile

Fig. 2.14(a) and (b) shows two neighboring computation tiles with two different placements of stem cells corresponding to Fig. 2.12(a) and (b) (rotated left by 90^0 with the extension of computation blocks) respectively. 89 and 90 are the width and length of a computation tile respectively. 91 indicates the width of stem cells as discussed in equation (7) which is greater or equal to the width of configuration frame. CC indicates a computation cell in the second computation block, B2**,** of a computation tile. The reason of having two different placements of stem cells in a computation tile is already discussed in section II-A.

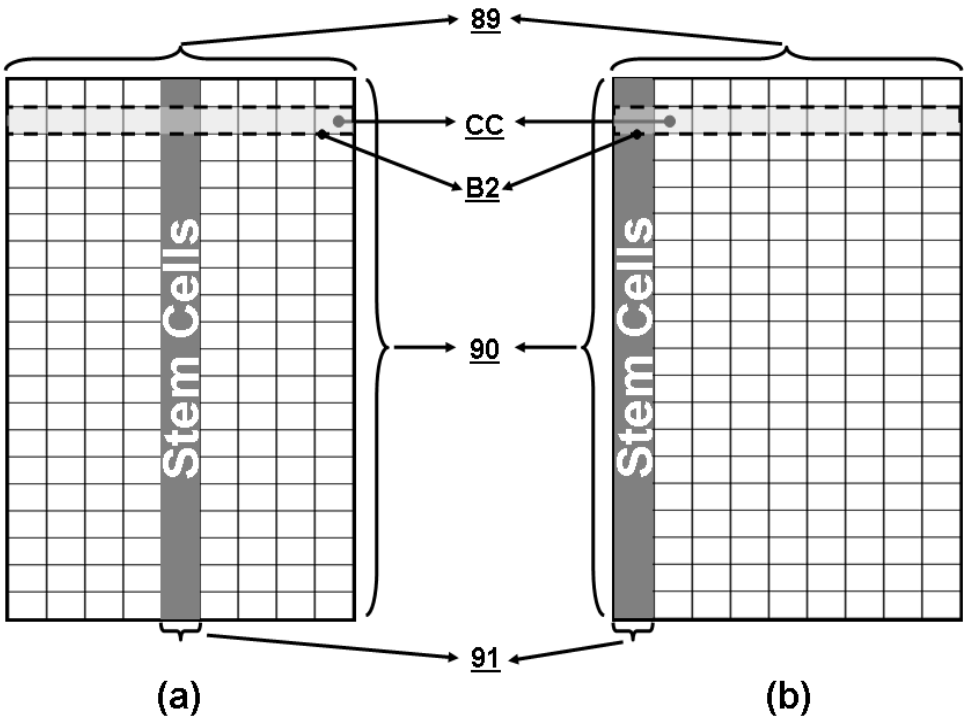


Fig. 3.14: Two neighbouring computation tiles with stem cells placed at (a) centre (b) corner.

The length of a computation tile is proportional to the configuration frame length. The configuration frame defines the granularity of a partial reconfiguration, that is, how many CLBs at least, are reconfigured during partial reconfiguration. The general formulation of a tile size, to accommodate the proposed computation tile in existing FPGA, is

$$\text{Tile Size} = L \times (N + W) \textit{ (CLBs)} \tag{2.9}$$

where,

N is the number of LUTs in a CLB, where $N \geq 8$.

Suppose, for a configuration frame size of 20×1 (20 CLB long by 1 CLB wide), and a CLB size of 8 LUTs, the size of a computation tile will be

$$\text{Tile Size} = 20 \times (8 + 1) = 180 \text{ (CLBs)}$$

This means that out of 180 CLBs, 20 vertical CLBs will be used as stem cells, and the rest 160 will be served as computation cells. Here the configuration frame size is 20×1 CLBs which indicates that while reconfiguring a portion of a device, 20×1 CLBs are reconfigured at once. Here we have 160 computation cells and thus permanent errors occurring in all of them at the same time can be cured instantly. Similarly for a configuration frame size of 40×2 (40 CLB long by 2 CLB wide), and a CLB size of 8 LUTs, the size of a computation tile will be

$$\text{Tile Size} = 40 \times (8 + 2) = 400 \ \textit{(CLBs)}$$

Out of 400 CLBs, 40×2 vertical CLBs will be used for stem cells, and the rest 320 will be served as computation cells. Since from equation (2.7), N+W = B, therefore equation 2.9 can be reduced into equation 2.10

$$\text{Tile Size} = L \times B \ \textit{(CLBs)} \tag{2.10}$$

where,

L is the length of a configuration frame;

B is the size of a computation block.

Also the number of stem cells, SC, in a computation tile is given by

$$SC = L \times W \ \textit{(CLBs)} \tag{2.11}$$

where,

L is the length of a configuration frame;

W is the width of a configuration frame.

If an error occurs in one cell of a computation tile, all of the stem cells in that tile are reconfigured due to the configuration granularity of a device. Of course, the reconfiguration of those stem cells whose corresponding computation cells are not faulty doesn't affect their functionality because they keep on running while the reconfiguration is performed.

This strategy leads to the fact that whether a permanent error occurs in a single computation cell or multiple cells, single flag signal should be taken out of a computation tile indicating that a permanent error has occurred in it. No matters, how many permanent errors are there in that computation tile – all of them will be cured at once. Therefore, the total number of cells in a computation tile, *C*, which can be healed from permanent errors concurrently at the same time, is

$$C = \text{Tile Size} - SC \tag{2.12}$$

where,

SC is the number of stem cells which have enough number of LUTs to recover all the errors in a computation tile.

E. Intra-tile Routing for the Generation of a Single Permanent Error Signal

Fig. 2.15 shows the intra-tile routing architecture which generates a single permanent error flag out of a computation tile. Since the permanent error signals from each cell of a computation block are routed into the free router of a cell (see section II-*C.1*) therefore a single permanent error signal is generated from each half of the computation block, as depicted in Fig. 2.13 (a) and (b). This scenario of Fig 2.13(a) is pictured again in Fig. 2.15. If there are 20 computation blocks, there will be 2 such signals, one from each half of every computation block, and thus 40 signals altogether from a tile. These signals are further divided into a group of 5 signals. The focused box, Blk-A, shows these 5 signals are input to a second available free router 93, which is the component 8 in Fig. 2.3. The number of these signals coming out from the second router will be 4 (for this particular case) from each half of the computation tile.

These signals are again routed into a component 95, as shown in a focused box, Blk-B. Now a single signal is generated, from each half of a computation tile,

each of which is input to 94 to generate a single permanent error flag, 96, out of a computation tile.

Following this strategy of intra-tile permanent error routing, the permanent error signal generating from each cell will have a same routing delay. It should be noticed that there are still free routers available that can easily be used for different tile sizes.

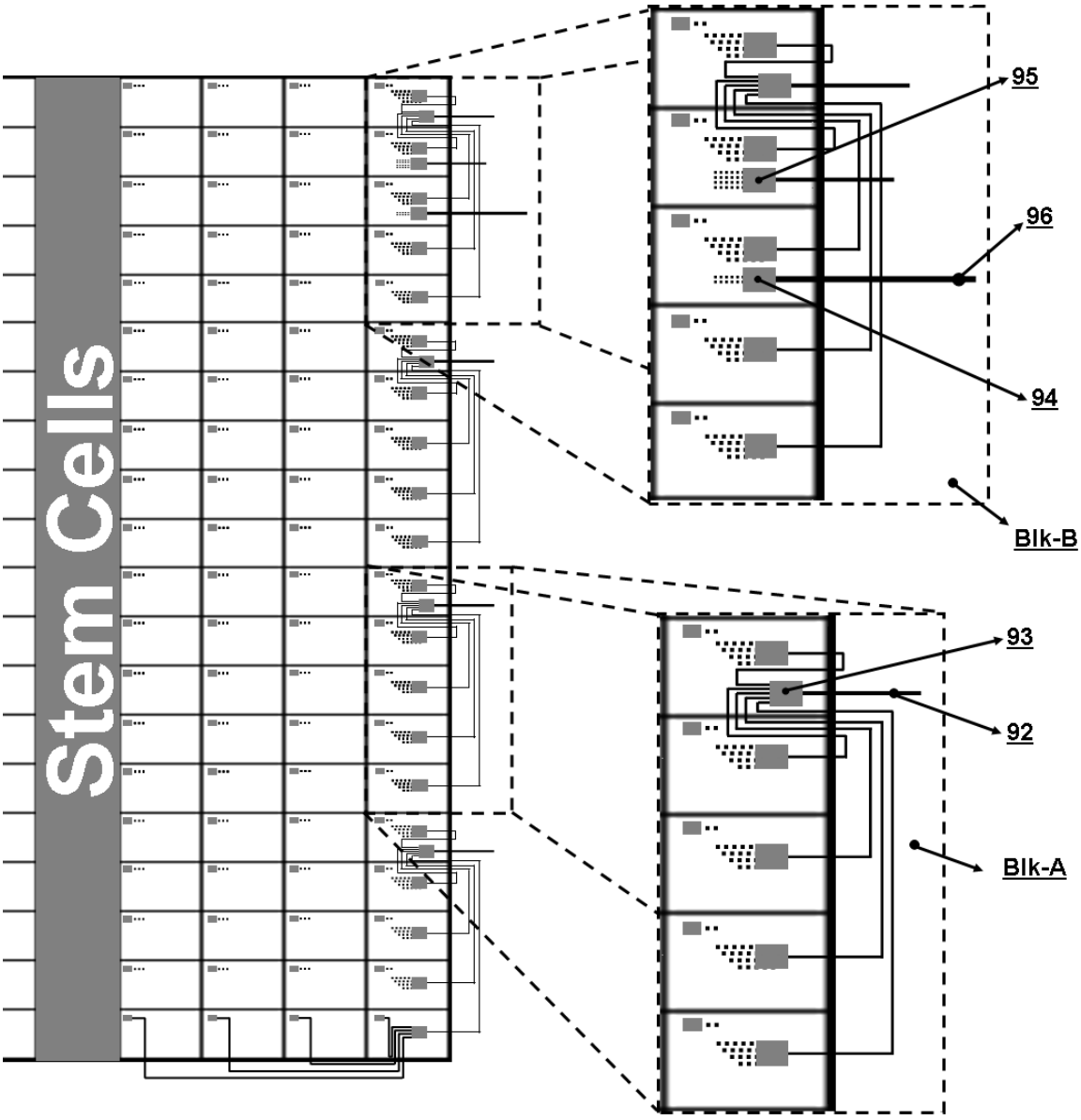


Fig. 3.15: The intra-tile routing of a permanent error signal.

F. The Fault-tolerant Core

Fig. 2.16 shows the internal components of a fault-tolerant core. It consists of two Permanent Error handlers, for both right and left halves of the FPGA chip. These permanent error handlers are nothing but 32-to-5 priority encoders with some additional circuitry. The priority encoder on the left decides which tile, out of the left side of a die, should be healed first when a permanent error occurs in more than one of them at a time. Similarly, permanent error handler (on right) makes a same decision for the right half. Priority Controller takes a decision which half of a die should be given highest priority.

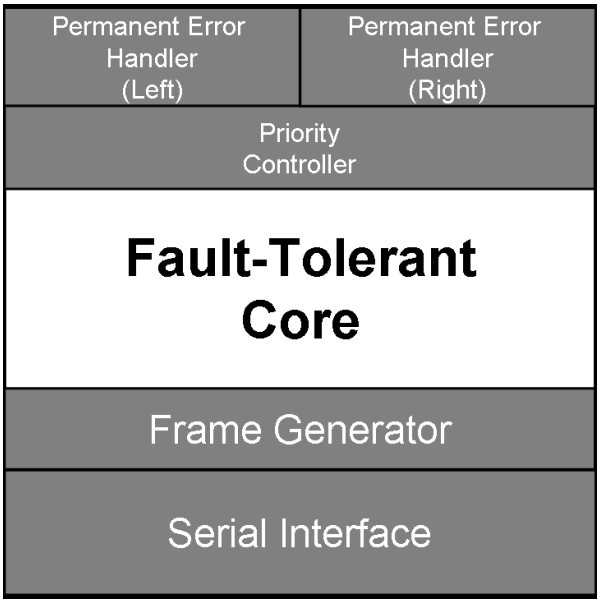


Fig. 3.16: The components of a fault-tolerant core.

The reason why we have divided the die into two halves can be explained by considering first the case when die is not divided for permanent error handling. In this case the architecture will be similar to Fig. 2.17 having tile indices from CT0 – CT43. Any tile, either CT0 or CT43 can be given highest preference for healing the permanent error. Suppose the highest preference is given to CT0 and that the lowest is given to CT43. In this scenario, if the permanent error occurs in, CT[0:7] for example, at the same time then the error present in CT4 has to wait until the permanent errors present in CT[0:3] gets healed.

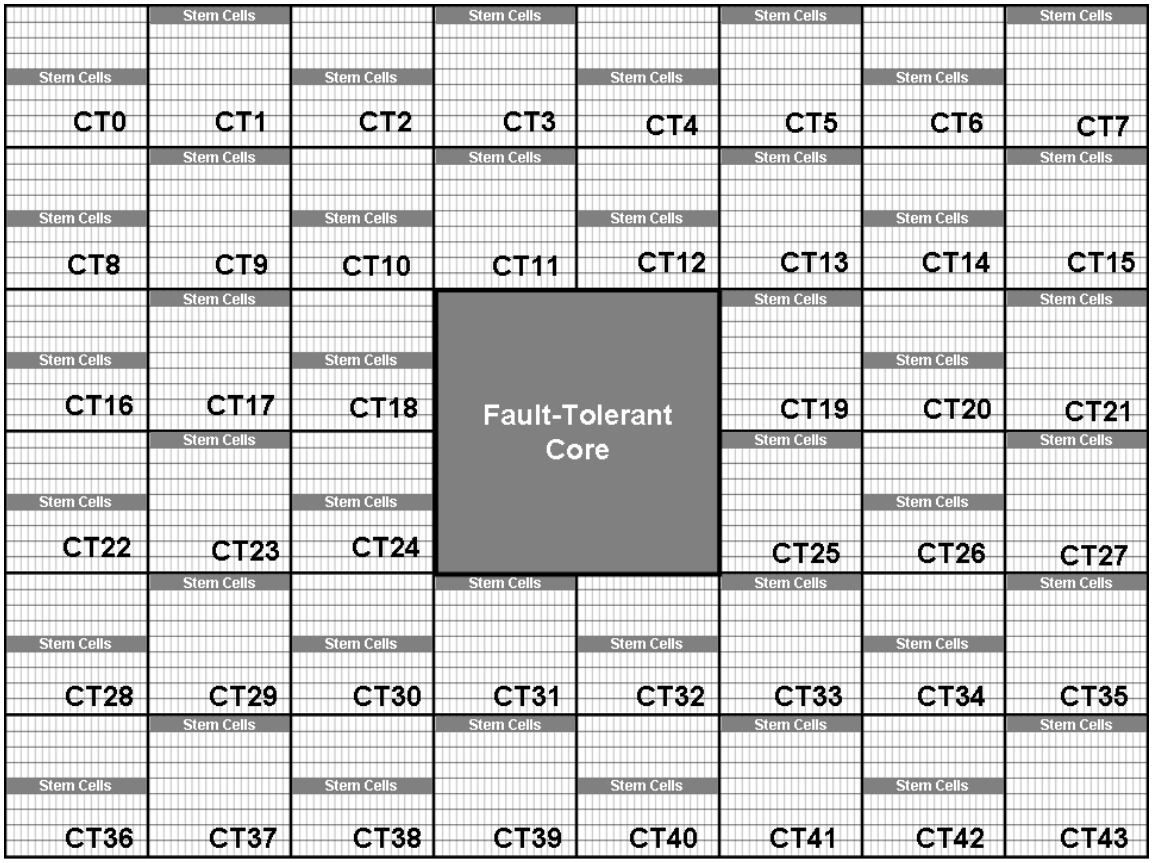


Fig. 3.17: The tile indices when die is not divided for error handling.

In order to avoid such situation, we divided the die into two halves, as depicted in Fig. 2 where each tile is marked by its unique ID CT0, CT1 …, CT21, for both halves of the die to identify which tile has a permanent fault. If the left half is given highest priority then according to the erroneous scenario discussed above, CT0 of left would be healed first and then CT0 of right half (corresponding to CT4 of Fig. 18). Then it will heal the CT1 of left and that of right half sequentially, and so on. Similarly, die can be divided into 4 halves in order to further improve the permanent error handling.

Frame generator, generates a frame containing sufficient information used by online external PC software to heal permanent errors through dynamic partial reconfiguration. The fault-tolerant on-chip core then sends this frame to external PC software through a serial UART interface.

1. Flow of Permanent Error Handler

Fig. 2.18 shows a sample logic flow of a “permanent error handler” for handling 8 computation tiles (having indices CT [7:0]) on either half. Since we have taken an example of 8 tiles here, therefore *Data_In* input register is 8-bits wide. The received data is first stored in the temporary register. In this flow, the highest indexed tile (i.e. CT7) is given the highest priority. After being acknowledged from the external PC software that the highest prior tile is healed, the “permanent error handler” checks whether the other tiles are also faulty or not, as depicted in the flow. In this manner all the permanent faults on either half of the die are healed sequentially due to the fact that only one bitstream can be downloaded at once.

2. Internal Structure of a Priority Controller

Fig. 2.19 shows the internal structure of a Priority Controller. Permanent error handlers generate computation tile IDs of those tiles which contain a permanent error in a single or multiple cells. When an error occurs in both halves of the die at a same time, the left half is given precedence (according to 103). There is no such reason of giving priority to left half of the die – any half can be given highest priority. The tile IDs, 108, permanent error indication bits, 106, and control signal bit, 107, are then passed to frame generator module. The signal 106 indicates whether a fault is present in both halves or in either half of the die. The control module, 105, is simply an AND gate whose one input 104 is by default 1. The signal 107 indicates the tile IDs (108) belongs to which half of the die. Suppose an error is present on both the halves of a die (value of 106 will be 11) then the output of 105 will be 1 and online software will heal the errors present in left half first. After healing the first error on the left half, the external PC software de-asserts the signal 104 to select right half of the die.

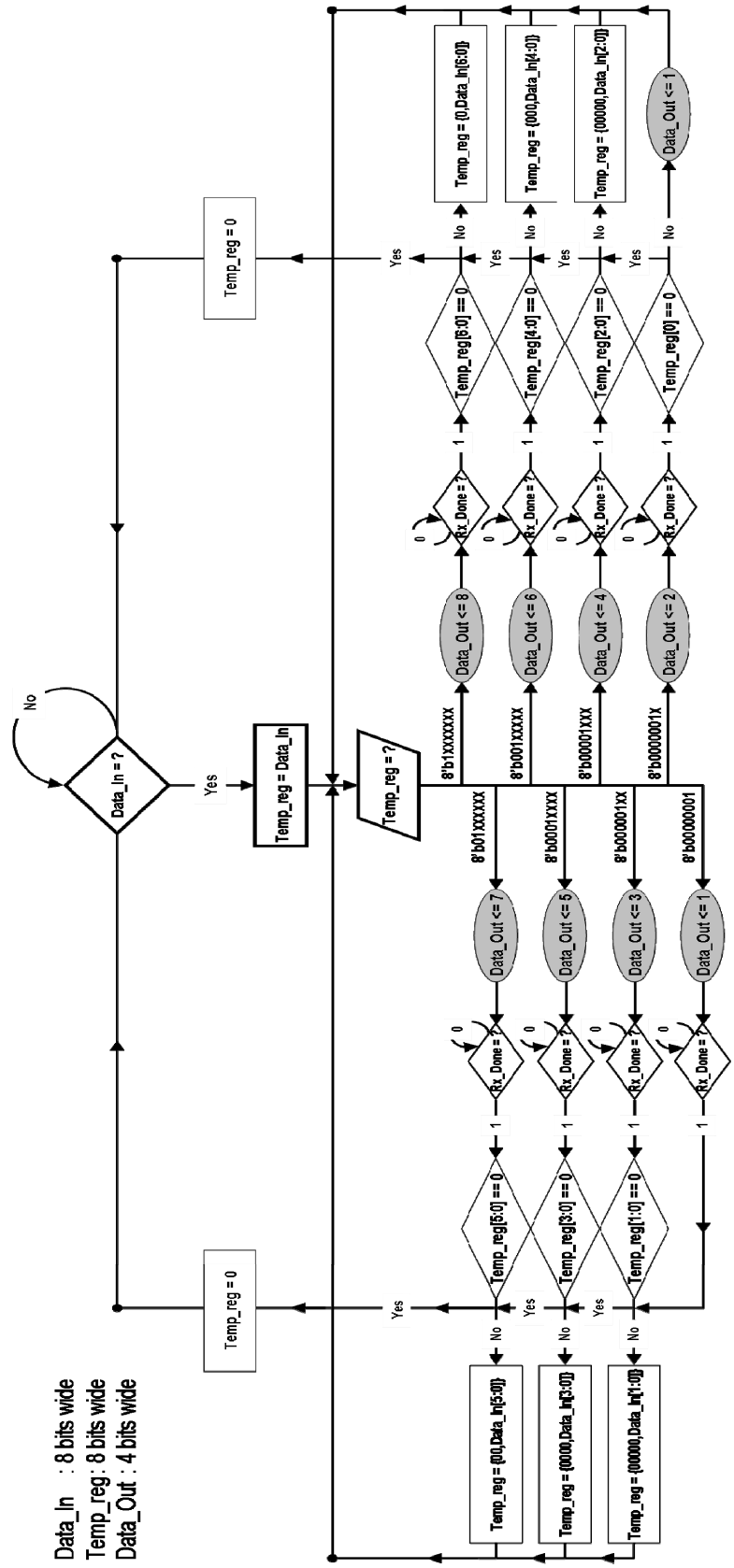


Fig. 3.18: The logic flow of a "permanent error handler".

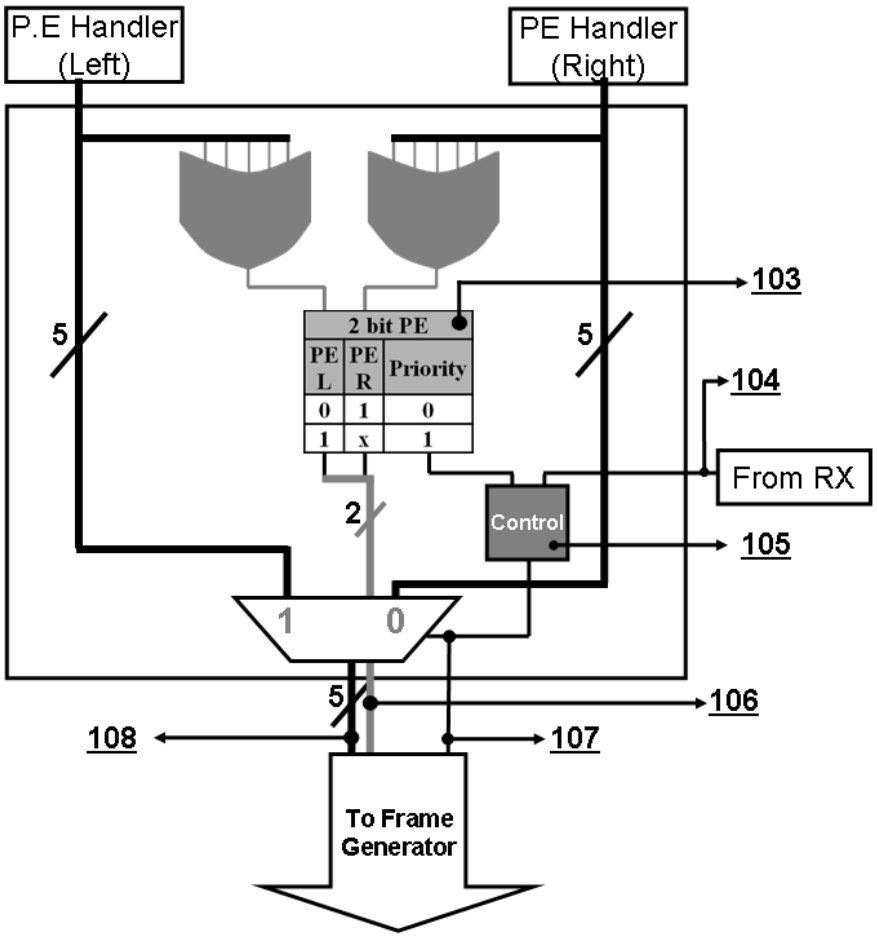


Fig. 3.19: The internal structure of a priority controller.

When an error in computation tile of the right half is healed, software again asserts signal 104 to select left half, and this process continues until all the permanent faults get healed.

3. The Frame Generator

Fig. 2.20 shows the standard frame generation for the present architecture. Fault-tolerant core encapsulates the information in this frame and sends it to online software through serial interface.

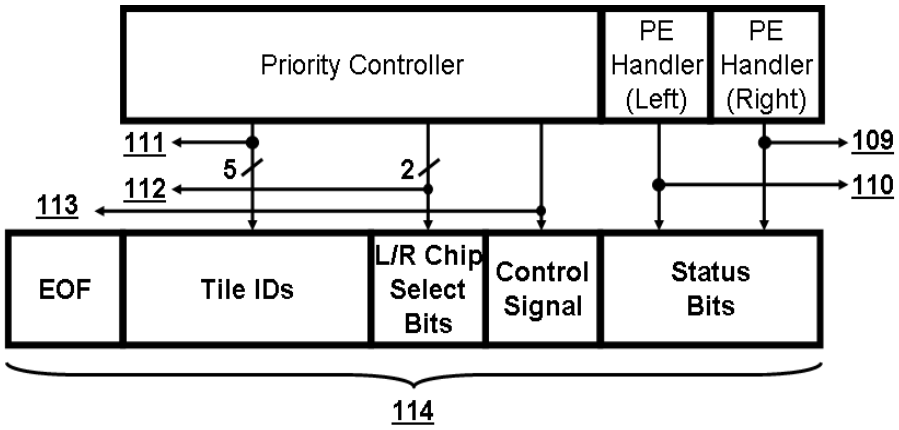


Fig. 3.20: The frame generator.

Other than EOF (End Of Frame) field, the frame 114, consists of 4 fields –

- Tile IDs hold the tile IDs of faulty tiles that inputs from priority controller through interface 111
- L/R Chip Select Bits stores the information of which half of the die has an error and feeds from a priority controller through interface 112
  - 10→Left half has an error
  - 01→Right half has an error
  - 11→Both halves have an error
- Control signal indicates that the data present in Tile IDs belongs to which half of the die and flows from a priority controller through interface 113
  - 0→Right half of a die
  - 1→Left half of a die
- Status bits are the VALID (V) signals taken from 32-to-5 priority encoders (Permanent Error Handlers), from each half of die, through interface 109 and 110.
  - 00→No permanent error
  - 01→Permanent error present on the right half
  - 10→Permanent error present on the left half
  - 11→Permanent error present on both the halves

The status bits serve the purpose of a SOF (Start of Frame) field. The external PC software processes the frame only when a non-zero value is present in this field, otherwise the frame is discarded.

G. The Self-repairing Software

Fig. 2.21 shows the screen shot of a self-repairing external PC software that is designed on LabVIEW platform. This software receives the information frame (shown in Fig. 2.20) from on-chip fault-tolerant core via UART interface at 38400 bits per second. This prototype software is designed to test the demo application (see section IV) on FPGA device. The designed software not only repairs the fault in FPGA tile but also introduces transient or permanent errors in order to verify the functionality of a proposed architecture. The error introducing technique is further described in chapter IV.

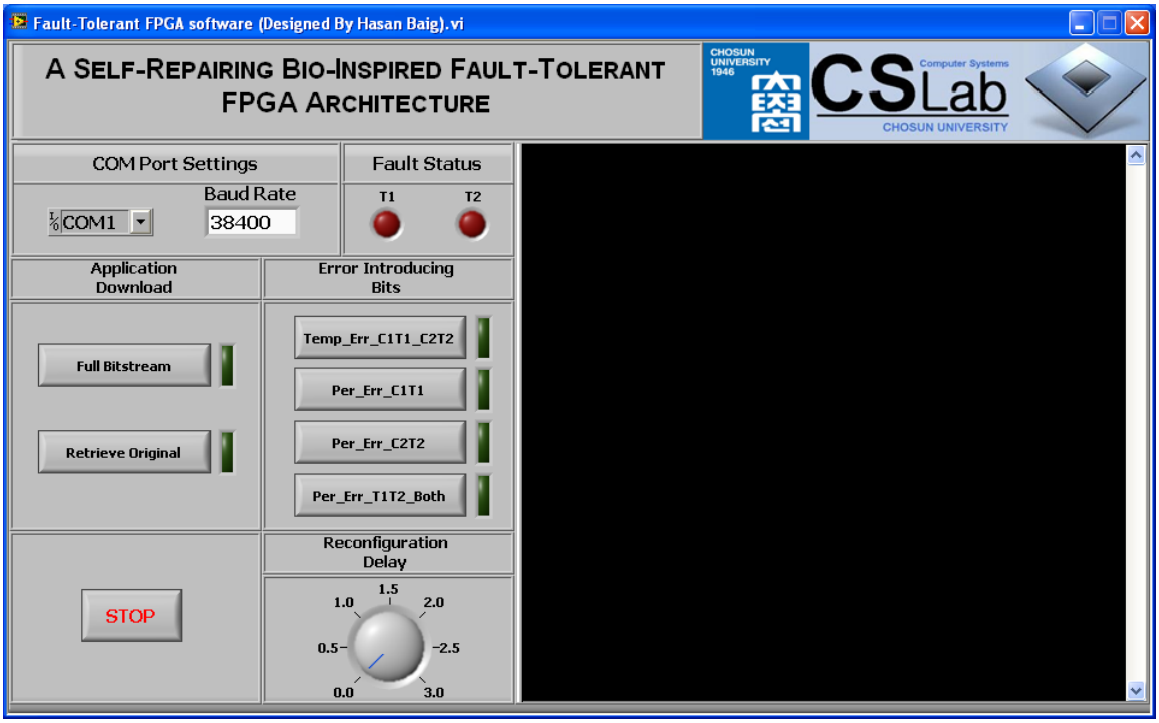


Fig. 3.21: The self-repairing software.

The black screen (Fig. 2.21) is the output of a command prompt. The software runs command files, corresponding to each button available in GUI, containing the commands of downloading a bitstream onto the target FPGA device. *Full Bitstream* button shown in Fig. 2.21 is used to download the application bitstream right away from the software. *Retrieve Original* button downloads the partial bitstream to retrieve the original functionality from the induced errors. There are four other options available for introducing errors in different computation cells of different computation tiles. *Fault Status* LEDs indicate the status of faults in two different tiles, T1 and T2. *Reconfiguration Delay* dial is an optional switch, given to introduce a time delay before auto-repairing the faulty tile of an FPGA. Since the software repairs the fault within the blink of an eye, so this option is quite helpful to let the viewers observe that the error has been introduced and it is auto-recovered by the software.

IV. Architectural Comparison with Previous Work

We have compared our proposed architecture with previously developed fault-tolerant architectures [12] and [8]. P. K. Lala et al [8] and Kim, S. et al [12] developed such an architecture which consists of separate original cell, spare cell and a cell router to route un-faulty cell (either original or a spare) with other cells. Both the original and spare cells have their own fault checking mechanisms due to which the area overhead is increased. Also, the spare function activates and goes through a self-checking mechanism only when an original function goes faulty. The point to be noted here is that the delay is increased to route the un-faulty function with another cell because the original function is checked first and if it is faulty then the spare function is activated and checked before routing its output to the input of another cell. Therefore, these schemes not only have the high area overhead but also experience the high inter-cell routing delay.

We have overcome these issues in our architecture. The proposed architecture consists of computation cells rather than a pair of original and spare cells. This computation cell consists of original function along with its redundant spare function. The function router is also a part of this computation cell. Due to this fact, not only the original and spare both functions check at the same time but also a single LUT (for fault-checking circuitry) is required for both of them (see Fig. 10). This reduces the area and routing delay overheads both by 50% as compared to [8] and [12].

In [8], function cell contains an original function and its corresponding complementary function. The fault is detected on the basis of the comparison of these two functions.

Similarly in [12], double modular redundancy is used for fault detection. It also has an additional functionality of self-testing whether the error occurred is temporary or a permanent. According to the architecture proposed in [12], each

working module has 16 working cells and a single self-tester is available for each working module. Therefore, it is evident that for more than one faulty working cells in a working module, self-test can be carried out on only one of them at a time. The self-test module generates pseudo-random bit patterns and passes it to the circuit under test (CUT). Then it examines the fault of logic by comparing the resulting values from CUT. It categorizes a fault to be transient if two output trains from CUT are exactly same, and permanent if both are different from each other. This self-testing facility is not provided in [8].

However, in our proposed architecture, error detection codes of original and spare functions are downloaded into a device during device configuration. These pre-generated codes are compared with the run-time generated error detection codes (see section III-*B.5*). Since, functional state of both the original and spare functions is verified at the same time, and it is very least probable that the same state of both functions gets faulty at a time, therefore we categorize a permanent error if both of them gets faulty together at the same time, else we call it a temporary error if only one of them is faulty. One of the positive points of our approach is that, unlike [8] and [12], in which an additional register is required to remember the erroneous state and faulty cell ID, the spare function instantly takes over the erroneous state of an original function without memorizing its current state in any extra register (see section III-*B.7*). The stem function reconfiguration heals the permanent error condition in [12] as well as in our proposed architecture.

According to the strategy of reconfiguring a stem cell of [12], only one cell of a working module can be healed from a permanent error at a time. It is also claimed in this research that the partial reconfiguration techniques has been employed to speed-up the permanent fault recovery process, which is contradictory to their video demonstration. The permanent fault recovery demonstration video, referred in [12], shows that the complete design flow reruns again from the process of synthesis till bitstream generation and downloading,

which takes 2 minutes and 20 seconds approx. including bitstream downloading time. The device resets for 9-10 seconds approx. during reconfiguration which clearly shows that the full device is reconfigured rather than the partial area of that faulty cell.

However in comparison to [12], our architecture heals permanent error of all of the computation cells, in the same computation tile, at once (see section III-*D*). Also, unlike [12], our self-repairing software reconfigures a faulty tile with its corresponding pre-generated partial bitstream. It takes no more than a second to reconfigure the faulty tile without interfering the other on-chip running process. Table 3-1 summarizes the architectural comparison between [8], [12] and our proposed architecture.

Table 4-1: Architectural Comparison with Previous Work

		P. K. Lala et al [8]	Kim, S. et al [12]	Our proposed architecture
Area overhead		Separate original and spare cells each of which having their own fault checking mechanism.	Same as P. K. Lala approach	Spare and original functions both are the part of same computation cell which shares same fault checking mechanism. (50% area overhead is reduced)
Non-faulty function routing delay overhead		Original function is checked first, if faulty, then spare function is checked followed by routing the output to another cell.	Same as P. K. Lala approach	Both the original and spare functions check at the same time and correct output is routed to another cell. (50% routing delay overhead is reduced)
Routing Architecture		Ambiguous	Ambiguous for inter working-modules routing	Don't care because it follows same existing routing strategies.
Temporary fault coverage	**Worst Case[1]**	50%	100 %	100 %
	Best Case[2]	100% [a]	100 % [b]	100 % [b]
Permanent fault coverage	**Worst Case[1]**	No permanent error healing	25 %	100 %
	Best Case[2]	No permanent error healing	100 %	100 %
Error detection in original and spare functions at a time		No	No	Yes
The number of times a permanent error, occurring in a same cell, can be healed		No permanent error healing	Five times.	As many time as it occurs
The recoverable cells from a permanent faults, at a time, in a module/tile		No permanent error healing	One	All
Resources required by Self-testing circuitry		No self-testing mechanism	More than 16 FF approx. (because of the presence of 16-bit ALFSR and counter)	4 LUTs
Requirement of extra register		Yes, to hold the address of a cell	Yes, to hold the address of a working cell, working modules and also to hold the erroneous state of a faulty cell.	No extra register is required to hold the address or an erroneous state of a cell.
Mechanism of handling permanent errors in more than one module/tile at a time		No permanent error healing	No mechanism is defined	Proper mechanism is developed
Mechanism to reconfigure the stem cell of a particular module/tile		No permanent error healing via stem cells	No mechanism is defined to reconfigure a particular "Stem cell" of a specific Working Module	Proper mechanism is developed to reconfigure the stem cells of a particular Computation Tile
External interface and master control logic		Not defined	It seems to be mixed with computation logic.	It has a separate portion at the centre of a die.
Mechanism of stem cell reconfiguration		No permanent error healing via stem cells	Though claimed via partial reconfiguration, but contradictory to their video demonstration	Through partial reconfiguration with pre-generated partial bits of each computation tile.
Time of stem cell healing		–	2 minutes and 20 seconds (approx.)	0 to 1 second (approx.)

1. Worst Case → When all of the cells have an error.
2. Best Case → When more than one cell have an error.
    a. For errors in up to two cells.
    b. For errors in any number of cells.

V. Experimental Testing

We have tested our proposed architecture on XC5VLX110T Virtex-5 FPGA device mounted on XUPV5-LX110T general purpose development board. To make a testing simpler and observable, we divided the application into eight functions which works together to perform LED shifting operation. We placed four functions in one tile and the remaining four in another. This application uses 8 general purpose on-board user LEDs to demonstrate the results of error induction and its healing. We placed one tile in the left half and another in the right half of the die in order to test permanent error handling operation of a priority controller (see section III-*F.2*). We also placed four functions of each tile in different computation blocks. So in total, each computation tile consists of

- 4 active computation cells
- 4 active computation blocks
- 4 stem functions (in 4 different stem cells)
- 2-input 4 original and 4 spare functions

Fig. 2.22 shows the functional diagram of a test application. This figure also shows the output connections of cells with external LEDs. Each tile contains four cells namely C1, C2, C3, and C4. As mentioned earlier, tile 1 is placed on the left side of a die and that of tile 2 is on the right side. The function table of original function in each cell is also pictured in Fig. 2.22. The input generator continuously generates the output sequence of 00→01→10→11→00, with the delay of 0.25 seconds, which inputs to all of the cells in both the tiles. Thus for the viewers, it becomes LED shifting application which performs a continuous left-shift operation on tile 1 LEDs and that of right-shift operation on tile 2 LEDs.

Fig. 2.23 shows the screen-shot view of a floor plan of testing application which is a cropped device-view taken from XILINX PlanAhead software tool. Fault-tolerant core is placed at the device center.

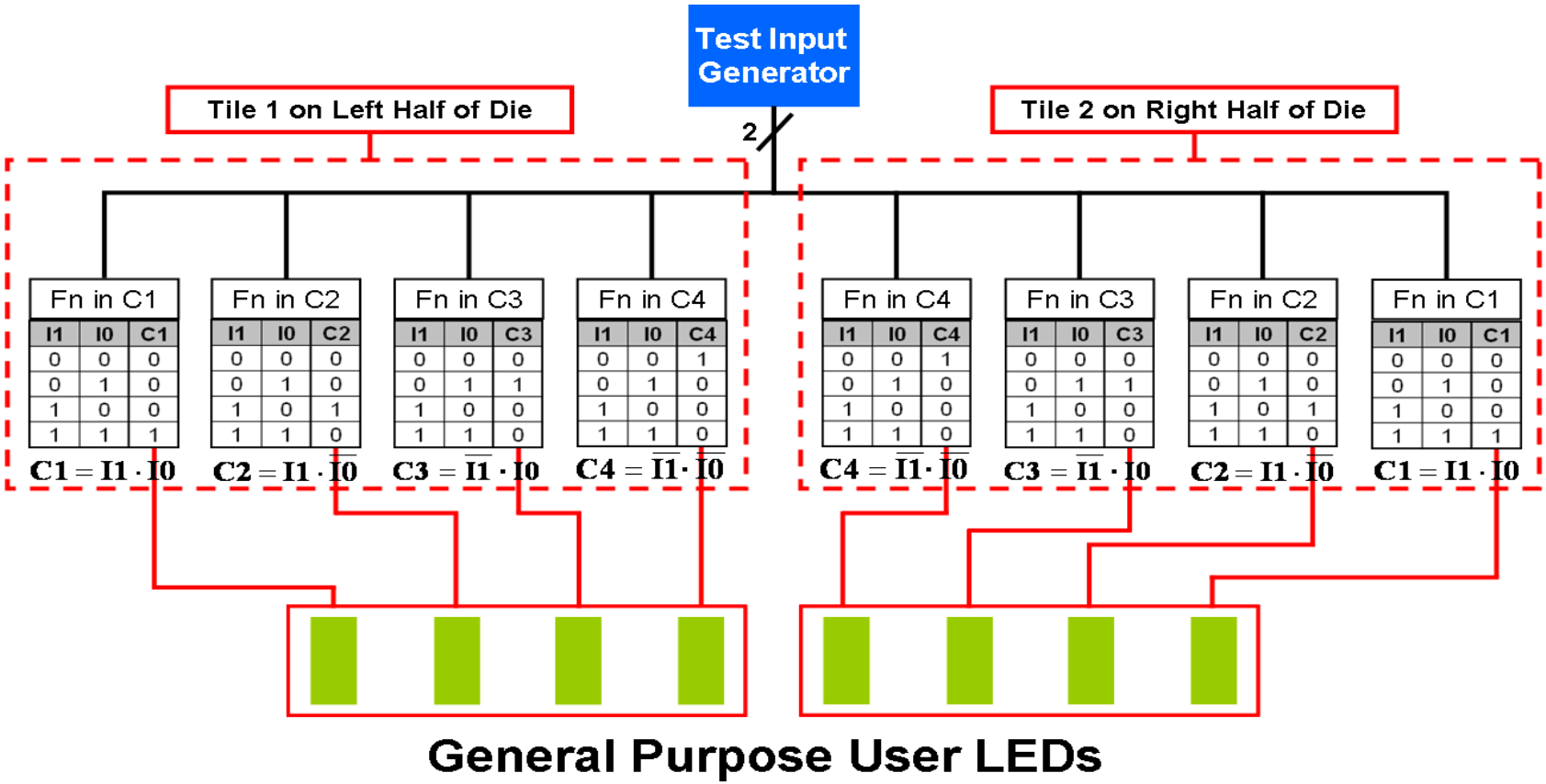


Fig. 5.1: Functional diagram of a test application.

Tile 1 and Tile 2, depicted in Fig. 2.23 (after rotating right by 90^0) corresponds to CT7 (of left half) and CT4 (of right half) of Fig. 2, respectively. Clearly, Tile 1 is on the left and Tile 2 is on the right half of the die.

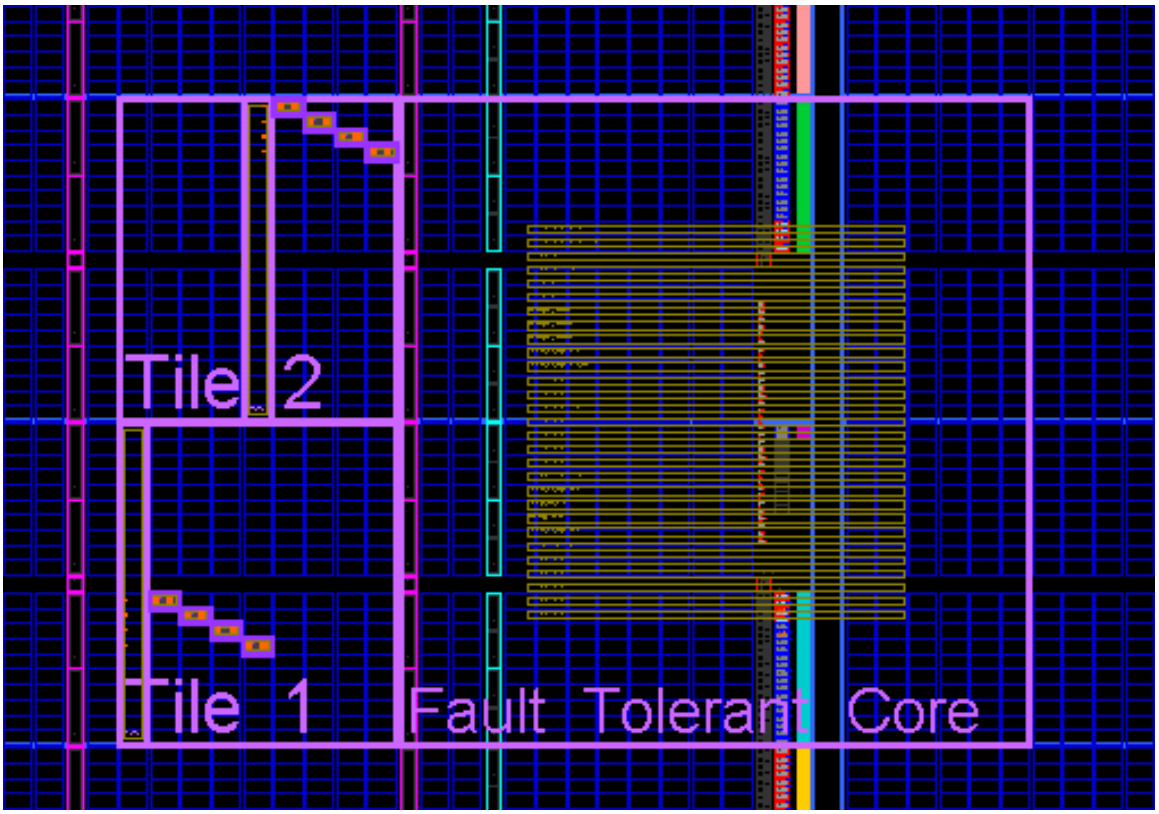
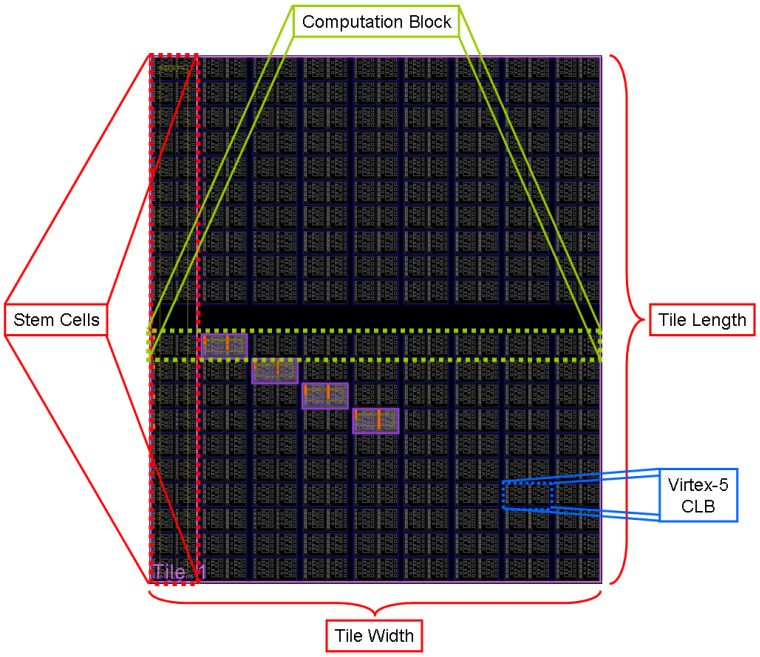


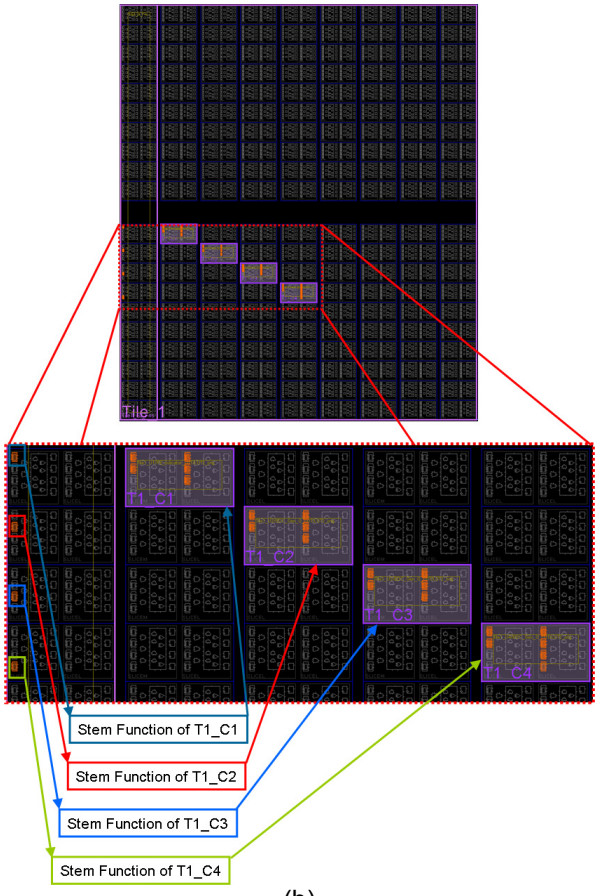
Fig. 5.2: Floor plan of a test application.

It is also evident that the cells' placement in each tile is different. Here we placed the cells diagonally in different computation blocks. There is no specific reason for this placement – any random placement can be followed.

Fig. 25 (a) shows the enlarged screen-shot view of a Tile 1. Virtex-5 CLB is also shown highlighted in Fig. 25 (a). Since the number of LUTs, in a Virtex-5 XC5VLX110T device, per CLB is 8 and the size of its configuration frame is 20x1 CLB (shown as stem cells in Fig. 25(a)), therefore the size of a computation block and computation tile, for our test application, is calculated according to the equations (2.7) and (2.10) respectively.



(a)



(b)

Fig. 5.3: Real implementation of a Tile 1 (a) its dimensions (b) zoomed view of its functional components.

This block of stem cells is partially reconfigurable and the whole block is reconfigured when its corresponding partial bit is downloaded. This is how any number of faulty cells in a computation tile can be repaired all at once.

Fig. 25 (b) shows the zoomed screen-shot view of the placement of computation cells and their corresponding stem cells connection in Tile 1. T1_C1 means cell 1 of tile 1, T1_C2 corresponds to cell 2 of tile 1 and so on. It also shows the stem functions for each computation cell. Since, in this application each computation block has one cell only therefore its corresponding stem cell contains only one stem function for that computation cell. Rest of the LUTs are left unused.

Fig. 26 shows the enlarged screen-shot version of a computation cell 4 of tile 1. The real placement of the internal components of implemented cell is clearly shown in this figure. The components 7 and 8 of Fig. 4 are left unused here (Fig. 26) because we have implemented 2-input functions in 6-input LUTs (number of inputs to LUT in a virtex-5 device is 6), so the components 5 and 6 of Fig. 4 are fractured here into two LUTs to store original and spare functions and their corresponding EDCs.

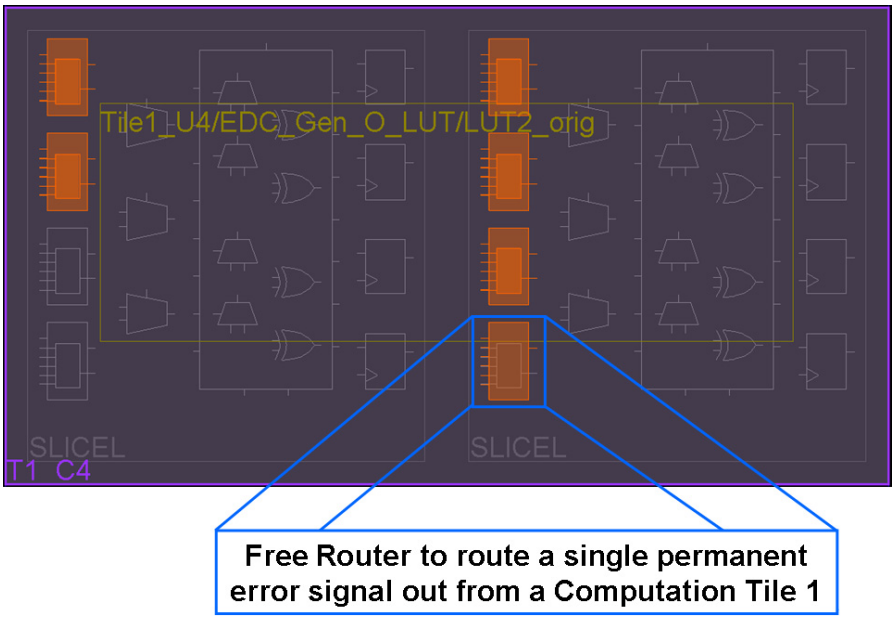


Fig. 5.4: The implemented computation cell 4 of computation Tile 1(shown in Fig. 2.23).

The component 13 of Fig. 2.3, i.e. a Free Router, is used here to take the permanent error signal from all 4 computation cells and route at its input. This component is simply an OR gate whose output indicates a permanent error in Tile 1. There is no specific reason of using 13 of Fig. 2.3 – either of 7 or 8 could also be used. Since there are only 4 cells in each tile, therefore all the four permanent error signals can easily be routed using one router only.

As discussed earlier in section II-*G*, the developed prototype software not only automatically repairs the permanent faults in FPGA, but also has a facility to induce temporary and permanent errors. Unlike [12] in which an extra circuitry is implemented for error induction, we employed *differential-based* partial reconfiguration technique to induce an error in any particular cell. With the help of differential-based partial reconfiguration, one can dynamically change the LUT contents without interrupting the on-going process. We used this technique to dynamically change the contents of original and spare functions to insert temporary and permanent errors. We generated separate bitstreams for each error case summarize in Table 4-2.

Table 5-1: Manual Reconfiguration Options in Prototype Software

Options	Description
Full Bitstream	Downloads main bitstream
Retrieve Original	Removes all temporary and permanent errors
Temp_Err_C1T1_C2T2	Introduce temporary error in Cell 1 of Tile 1 and Cell 2 of Tile 2
Per_Err_C1T1	Introduce permanent error in Cell 1 of Tile 1 (stuck-at-1 fault)
Per_Err_C2T2	Introduce permanent error in Cell 2 of Tile 2 (stuck-at-0 fault)
Per_Err_T1T2_Both	Introduce permanent error in Tile 1 & Tile 2 at a time – (stuck-at-0 fault) in Cell 4 of Tile 1 – (stuck-at-1 fault) in Cell 4 of Tile 2 – (stuck-at-0 fault) in Cell 3 of Tile 2

All the bitstreams should be generated before performing a test. These bitstreams are stored in the predefined folder accessible to prototype software. We first stored the bitstream downloading commands in separate command files for each bitstream. When the action (mentioned in Table II) is initiated the prototype software runs the command file associated with that option, to download that particular bitstream. Hence, we do not need to run XILINX iMPACT software for bitstream downloading.

After running the software, full application bitstream is downloaded first with the help of “Full Bitstream” button shown in Fig. 2.21. Then different error conditions can be tested according to Table II. For temporary error, we change the contents to original function in cell 1 in tile 1 and the contents of spare function of cell 2 in tile 2. The function keeps on running without hindrance whenever a temporary error occurs because it follows the rule 2 and 3 described in section II-*B.7*. When a permanent error occurs, fault-tolerant core sends the information frame to external PC software through UART interface. Software then analyzes the frame to find out which computation tile is faulty and download its corresponding partial bitstream. This whole process, including receiving information frame, its analysis and bitstream downloading, takes few seconds (1-2 sec. approx.) to complete. Out of this time, the partial bit downloading time is no more than a second. This is how the software auto-repairs the faults in our proposed fault-tolerant FPGA architecture.

The software also indicates which tile has an error via T1 and T2 LEDs shown in Fig. 2.21. We have also tested the scenario when the error occurs in more than one of the tile at a time. After exciting “Per_Err_T1T2_Both” option, the error induces in both the tiles at the same time. Software then heals tile 1 first, as it is on the left half of die, and then tile 2. The user can also use “Reconfiguration Delay” option (Fig. 2.21) to observe the fault induction and its sequential recovery by the self-repairing software. Fig. 2.26 shows the snap-shot view of

experimental setup. It was taken when a permanent error in tile 1 was introduced which can also be noticed on monitor screen by the glow of T1 LED indicator.

Hence the self-repairing fault-tolerant operation of a proposed architecture is well demonstrated through this experiment.

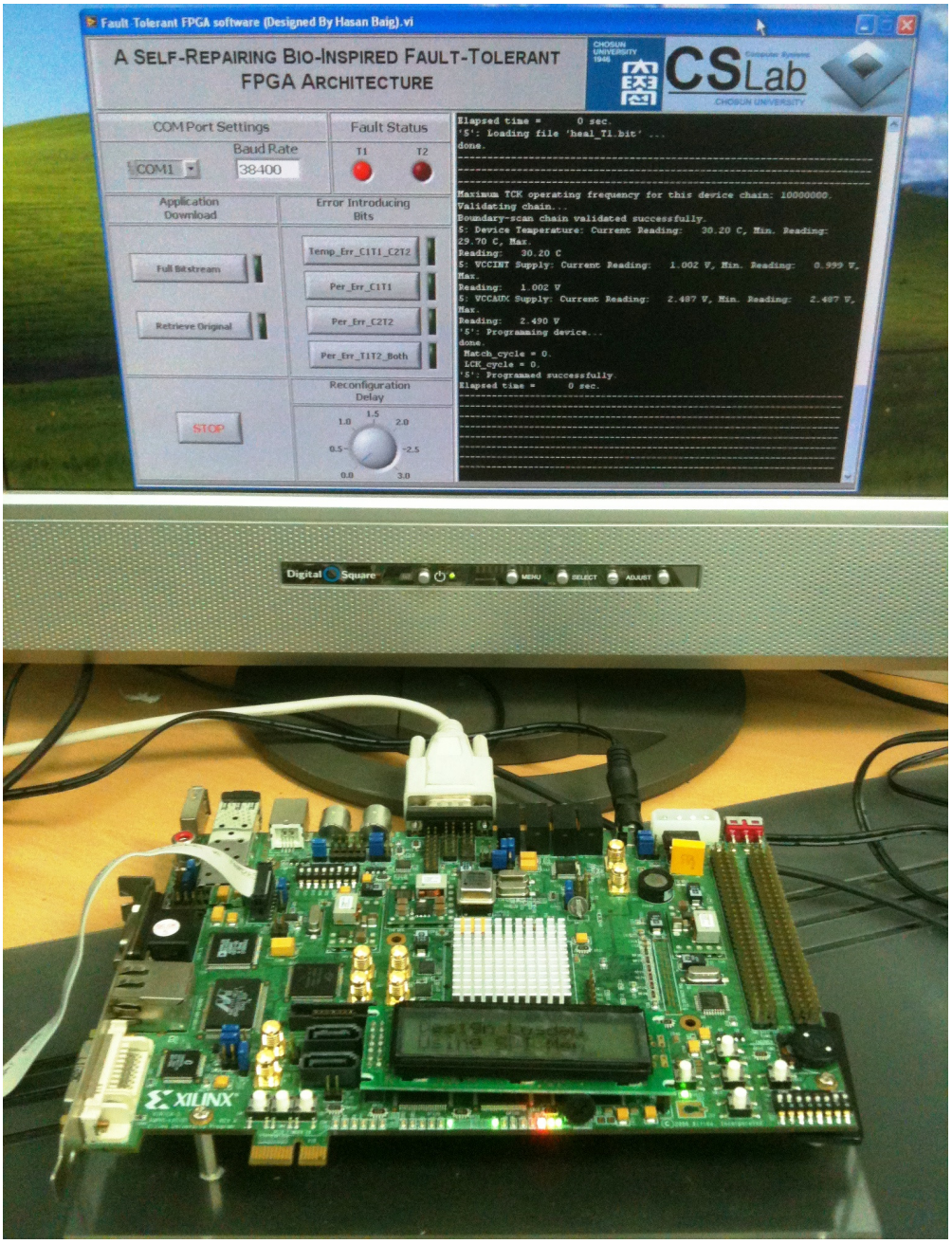


Fig. 5.5: The experimental setup.

VI. Conclusions and Future Enhancements

A novel fault-tolerant FPGA architecture has been presented which not only heals the temporary faults instantly but is also capable to repair the permanent stuck-at faults. We have developed a generic fault-tolerant FPGA computation cell that can either be implemented in existing FPGA devices or can be fabricated as a basic building block of an entire new FPGA device with fault-tolerant capabilities. The best thing is that the designers do not need to worry about the routing issues to implement the proposed architecture on existing FPGA device.

This architecture consists of computation tiles each of which contains computation blocks and cells. We have presented a generic formula for calculating the computation tile and block sizes. We have also claimed and devised that our proposed architecture heals the number of permanent errors in a computation tile all at once.

The permanent fault-tolerance is managed by a fault-tolerant core. We came up with a permanent error handling algorithm which manages the number of permanent errors occurring in different areas of a die at the same time. This fault-tolerant core communicates with external PC software that identifies which part of a die is faulty and recovers it without human intervention. The software heals the permanent errors by downloading pre-generated partial bits (separate for each tile) via JTAG interface to achieve fast fault-recovery.

Unlike previously proposed architectures, an unlimited number of temporary and permanent faults, in a computation tile, can be healed through our proposed architecture. We have compared our proposed work with previously published work and proved that our scheme is far much better than others in many ways.
However, the proposed architecture is dependent on external PC software to heal permanent errors. This shouldn't be considered as a limitation rather a

prototype developing environment. Just like some of the existing FPGA vendors provide facility to monitor on-chip physical parameters including on-chip power supply voltages and die temperatures, or internal signals etc. For instance a SYSMON and CHIPSCOPE ILA (Integrated Logic Analyzer) cores, provided by XILINX, monitors the on-chip physical parameters and signals, respectively on external PC software through JTAG interface. Similarly, the developed fault-tolerant core can also be initialized to monitor and manage the fault-tolerant operation of a device, together with the collaboration of off-chip self-healing PC software.

This project can be made completely standalone as a further enhancement by making it independent of off-chip PC software. For this purpose, a soft-core microprocessor can be employed, as a part of fault-tolerant core, to heal permanent errors by downloading partial bits through ICAP (Internal Configuration Access Port) interface. Embedded software algorithm, equivalent to the present external PC software, needs to be developed for the identification of faulty computation tiles.

Bibliography

[1]. R. E. Lyons and W.Vanderkulk, “The use of triple-modular redundancy to improve computer reliability,” *IBM J.*, vol. 6, no. 2, pp. 200–209, Apr. 1962.

[2]. C. LaFrieda, E. Ipek, J. F. Martínez, and R. Manohar, “Utilizing dynamically coupled cores to form a resilient chip multiprocessor,” *Proc. 37th Annu. Int. Conf. Depend. Syst. Netw.*, 2007, pp. 317–326.

[3]. D. Mange, E. Sanchez, A. Stauffer, G. Tempesti, P. Marchal, and C. Piguet, “Embryonics: A new methodology for designing field-programmable gate arrays with self-repair and self-replicating properties,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 6, no. 3, pp. 387–399, Sep. 1998.

[4]. D. Mange, M. Sipper, A. Stauffer, and G. Tempesti, “Towards robust integrated circuits: The embryonics approach,” *Proc. IEEE*, vol. 88, no. 4, pp. 516–541, Apr. 2000.

[5]. L. J. K. Durbeck and N. J.Macias, “Defect-tolerant, fine-grained parallel testing of a Cell Matrix,” *Proc. SPIE ITCom 2002 Series 4867*, J. Schewel, P. James-Roxby, H. Schmit, and J. McHenry, Eds., 2002, pp. 71–85.

[6]. J. M. Emmert, C. E. Stroud, and M. Abramovici, “Online fault tolerance for FPGA logic blocks,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 2, pp. 216–226, Feb. 2007.

[7]. S.Mitra,W.-J. Huang, N. R. Saxena, S.-Y. Yu, and E. J. McCluskey, “Reconfigurable architecture for autonomous self-repair,” *IEEE Design Test Comput.*, vol. 21, no. 3, pp. 228–240, May 2004.

[8]. P. K. Lala, B. K. Kumar, and J. P. Parkerson, “On self-healing digital system design,” *ELSEVIER Microelectron. J.*, vol. 37, no. 4, pp. 353–362, Apr. 2006.

[9]. W. Barker, D. M. Halliday, Y. Thoma, E. Sanchez, G. Tempesti, and A. M. Tyrrell, “Fault tolerance using dynamic reconfiguration on the poetic tissue,” *IEEE Trans. Evol. Comput.*, vol. 11, no. 5, pp. 666–684, Oct. 2007.

[10]. J. D. Hadley and B. L. Hutchings, “Designing a partially reconfigured system in FPGAs for fast board development and reconfigurable computing,” *Proc. SPIE 2607*, 1995, pp. 210–220.

[11]. E. J. McDonald, “Runtime FPGA partial reconfiguration,” *IEEE A&E Syst. Mag.*, vol. 23, no. 7, pp. 10–15, Jul. 2008.

[12]. Kim, S. et al. “A Hierarchical Self-Repairing Architecture for Fast Fault Recovery of Digital Systems Inspired From Paralogous Gene Regulatory Circuits”, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* vol. PP, Issue. 99, pp. 1–14, Dec 2011.
Digital Object Identifier: 10.1109/TVLSI.2011.2176544

[13]. S. Durand and C. Piguet, "FPGA with Self-Repair Capabilities", *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, pp 1-10, Berkeley, California, February 1994.

[14]. C. Ortega-Sanchez and A.M. Tyrrell, "Design of a Basic Cell to Construct Embryonic Arrays" *IEE Proceedings on Computers and Digital Techniques*, volume 143, pp 242-248, May 1998.

[15]. A. Avizienis. "Towards Systematic Design of Fault-Tolerant Systems", *IEEE Computer*, vol. 30, Issue 4, pp. 51-58, April 1997.

[16]. S. Xanthakis, S. Karapoulios, R. Pajot, and A. Rozz, "Immune System and Fault Tolerant Computing", *J.M. Alliot, editor, Artificial Evolution, volume 1063 of Lecture Notes in Computer Science*, pp 181-197. Springer-Verlag, 1996.

[17]. C. Öztürkeri and M. S. Capcarrere, “Self-repair ability of a toroidal and non-toroidal cellular developmental model,” *Lecture Notes in Computer Science*. Berlin: Heidelberg: Springer-Verlag, 2005, vol. 3630, pp. 138–148.

[18]. W.-J. Huang and E. J. McCluskey, “Column-based precompiled configuration techniques for FPGA fault tolerance,” *Proc. IEEE Field-Program. Custom Comput. Mach. (FCCM)*, 2001, pp. 137–146.

ABSTRACT

A Self-Repairing Bio-Inspired Fault-Tolerant FPGA Architecture

Hasan Baig
Advisor: Prof. Jeong-A Lee, Ph. D.
Department of Computer Engineering
Graduate School of Chosun University

The most complex thing in any FPGA architecture is its routing network, and to propose a new FPGA architecture means to keep all the routing issues in mind. Also it is difficult for the commercial vendors of FPGA devices, like XILINX, Altera etc, to refine their existing routing architecture according to the newly proposed schemes to incorporate the fault tolerant capabilities. This arises the need to develop such a design that can easily be integrated with the existing routing architecture. In this research, we have developed a complete homogenous fault-tolerant FPGA architecture with self-repairing capabilities. Unlike previously proposed architectures, the present one can not only be implemented on the existing island-style FPGA architecture but can also be able to fabricate entirely as a new device utilizing the existing routing network strategies. The developed architecture is unique in a way that it is able to identify transient and permanent errors (at LUT level) both at the same time. A generic fault-tolerant Computation Cell is developed which, along with its self-checking circuitry, also consists of an internal router to route un-faulty function out of the cell. The proposed fault-tolerant FPGA architecture is comprised of Computation Tiles, each of which consists of *N* computation cells which are able to heal transient or permanent errors all at once. This architecture is centrally controlled by an on-chip fault-tolerant core whose main responsibility is to communicate with the external PC

software, via UART interface, if an error occurs in any of the computation tile. The external PC software identifies and partially reconfigures the stem cells of faulty computation tile without intervening the functionality of rest of the device. The robust operation of a proposed architecture is implemented and verified on XILINX Virtex-5 FPGA device. The ratio of the hardware overhead to fault coverage in our approach is much lesser than that of TMR and recently developed fault-tolerant architectures. We have proved that our architecture is much better than others in many ways.

ACKNOWLEDGMENT

I would first of all thank Allah Almighty Who enabled me to carry out this project with full devotion and consistency. It is only because of His blessings that I could find my way up to the completion of this task.

Next, I would like to express my immense regards and honest gratitude to my supervisor, Prof. Jeong-A Lee. Her valuable support, guidance, appreciation and supervision, throughout the course of this novel research, motivationally steered me to accomplish this task successfully. I would also like to thank her for supporting me financially to attend a professional training of “Partial Reconfiguration” by XILINX in Taipei, Taiwan. The concepts I gained during this training helped me a lot in carrying out this project in a right direction.

I also appreciate the efforts of reviewing committee members, Prof. Sangman Moh and Prof. Choi Goang Seog for their critical analysis and precious advices during the preparation of this thesis.

I would also like to thank Prof. Parimal Patel (XILINX) for his precious help over and over regarding partial reconfiguration issues. I am also grateful to my father, Dr. Jawed Altaf Baig and my brother, Dr. Muhammad Baig for giving me a biological inspiration of fault-tolerant system exists in human body.

Finally, I also pay my esteem regards to MKE (Ministry of Knowledge Economy), Korea under the Global IT Talents Program supervised by NIPA (National IT Industry Promotion Agency), for its financial support during the period of my Master studies.

I dedicate this thesis to my family, my loving wife Sumaiya, and to my two cutie niece and nephew - Maria and Saif.