IMPLEMENTATION OF LOGIC FAULT TOLERANCE ON A DYNAMICALLY RECONFIGURABLE FPGA

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering

By

KIRAN JAYARAMA
B.E., Visvesvaraya Technological University, India, 2013

2016

Wright State University

WRIGHT STATE UNIVERSITY

GRADUATE SCHOOL

December 16th,2016

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Kiran Jayarama ENTITLED Implementation of Logic Fault Tolerance On a Dynamically Reconfigurable FPGA BE ACCEPT IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Electrical Engineering.

J. M. Emmert,Ph.D.
Thesis Director

Brian Rigling,Ph.D.
Chair, Electrical Engineering

Committee on Final Examination

J. M. Emmert, Ph.D.

Saiyu Ren, Ph.D.

Raymond E. Siferd, Ph.D.

Robert E.W.Fyffe,Ph.D.
Vice President for Research and
Dean of the Graduate School

Abstract

Jayarama, Kiran. M.S.E.E., Department of Electrical Engineering, Wright State University, 2016. Implementation Of Logic Fault Tolerance On A Dynamically Reconfigurable FPGA.

Relative to integrated circuit (IC) systems, on-chip fault detection entails determination of whether or not a fault exists. The cause of the fault could be some faulty logic resource or some faulty interconnect (wiring) resource, but typically, fault detection only determines if a fault exists, not what exactly is faulty. Beyond pure fault detection some work has been done relative to on-chip fault analysis to further determine not only if a fault exists, but exactly what is faulty. Even less work has been done to actually tolerate faulty resources once they have been found. For this work, we take advantage of previous work (ROVING STARS) that detects on-chip faults and analyzes those faults to determine exactly what is faulty. We developed, tested and demonstrated an on-chip technique that takes advantage of dynamic partially reconfigurable field programmable gate arrays (FPGAs) to automatically reconfigure the FPGA for tolerating logic faults.

TABLE OF CONTENTS

1 INTRODUCTION **1**

2 BACKGROUND **3**

2.1 Difference Between Redundancy and Fault Tolerance 3

2.2 Fault Methodologies 4

2.3 Field Programmable Gate Array (FPGA) 12

2.4 Fault Tolerance 15

2.5 Partial Runtime Reconfiguration (PRR) 17

3 FAULT TOLERANT APPROACH **24**

3.1 Overview of Methodology 24

3.2 Look Up Table (LUT) 25

3.3 FPGA Editor 27

3.4 Proposed Methodology 28

4 IMPLEMENTATION **36**

4.1 Demonstration 36

4.2 Limitations 39

5 CONCLUSION **41**

5.1 Summary 41

5.2 Future work 41

REFERENCES **43**

LIST OF FIGURES

Figure Page

2.1 Configuration of FPGA 6

2.2 A PLB and switch pair 7

2.3 Example of SRAM shifting 9

2.4 SRAM shifting architecture 10

2.5 Tiling method 11

2.6 Cluster-based Fault Tolerance 12

2.7 FPGA Architecture 13

2.8 Implementation k=4 input LUT 14

2.9 I/O block configuration 15

2.10 Programmable Interconnect 16

2.11 Module-based partial reconfiguration 19

2.12 Changing LUT equation 21

2.13 Changing block RAM contents 23

2.14 Modifying I/O 23

3.1 Examples of Look-Up table 26

3.2 Look Up Table configuration example 26

3.3 LUT implementation at the transistor level 27

3.4 Truth table for Boolean equation of Z 29

3.5 LUT showing with and without faults 30

3.6 FPGA Editor where D6LUT holds the Boolean equation 31

3.7 Example of .rbt file 31

3.8 Difference between the two files 33

3.9 FPGA architecture that shows Switch box and Connector box connection 34

3.10 Switch box 34

3.11 Example of a 3-input LUT 35

4.1 Boolean function of Z 37

4.2 Boolean function making use of the faulty cell 38

4.3 3-input LUT with and without a fault 40

ACKNOWLEDGEMENT

I would like to sincerely thank my advisor Dr. J. M. Emmert for his continuous guidance and excellent advice. I wish to submit to him my wholehearted gratitude for all the support and help I received from him. I would like to thank my thesis committee memebers Dr. Ray Siferd and Dr. Saiyu Ren. Finally, I would like to express my gratitude to my family for their love and encouragement in all my endeavors throughout my academic career.

Kiran Jayarama, MSEE 2016

Chapter 1

INTRODUCTION

The evolution of automation has revolutionized the world. We, as a society continue to increase our reliance on automated systems. These systems often hold our lives in their hands. For example, there are inventions like aerospace auto pilot systems and medical pacemakers that require high performance and reliability in almost any given condition. Remote, inaccessible space systems must have uninterrupted functionality in extreme conditions to successfully accomplish their missions. These mission critical systems placed in harsh environments require high availability and reliability to precisely perform their designed function. These types of systems are required to tolerate faults that occur during runtime in order to increase the life span of the sytem and to continuously ensure proper system function. Many types of adaptive computing systems use reconfigurable hardware in the form of field programmable gate arrays (FPGAs) to accomplish their tasks. Field programmable gate arrays offer partial runtime reconfiguration (PRR) capability, to continue uninterrupted functionality, while different portions of the FPGA are reconfigured to accommodate additional logic and functionality. Defects that occur due to aging or because of environmental factors are inevitable, and they may lead to system failure once the system is put into use to serve its pur-

pose. When human intervention for repair and maintenance is not possible, fault tolerant (FT) techniques must be used to achieve the desired functionality even in the presence of faults. In this thesis, we developed, tested, and demonstrated an on-chip technique that takes advantage of dynamic partially reconfigurable FPGAs for automatically reconfiguring FPGAs to tolerate logic faults. Our method takes advantage of previous work in on-chip fault detection and analysis (location). In this work we present and demonstrate a technique for generating the partial reconfiguration files required for PRR in order to tolerate faults during application runtime.

Chapter 2

BACKGROUND

2.1 Difference Between Redundancy and Fault Tolerance

The duplication of critical components or functions of a system with the intention of increasing reliability is called redundancy [1]. The property of a system to continue operating in the presence of a fault (or faults) due to the failure of some of its components is called fault tolerance [2]. Relative to integrated circuits, the idea behind both of the above mentioned terms is to keep the system up and running in an environment where components within the devices may become faulty. Even though the two terms are often used to refer to the same principle, they are actually very different from one another. The goal of redundancy is to improve the availability of a system by using duplicated components. There may be double, triple or even quadruple modular redundancy. In some systems the primary component is active and secondary or tertiary components are idle (but available). In some systems, all components are active, and voting circuitry is used to guarantee correct results. For fault tolerance, effort is focused on keeping the system running even in the presence of faulty components or bad signals due to

hardware failure. One example of a fault tolerant system is "RAID" – Redundant Array of Inexpensive Disks – but why does RAID have redundant in its name? The distinction comes down to the difference between the individual components and the entire system. The disks within a RAID array are redundant, but they are not fault tolerant. If a disk fails, then it is dead. In the RAID system, when a disk fails, another disk replaces it in the system. The failure is tolerated by replacing the faulty component, but the disk itself is not fault tolerant. Thus the overall system tolerates faults, but the individual disks are not fault tolerant.

2.2 Fault Methodologies

There are various fault tolerant approachs for FPGAs that have been proposed throughout the years. Approaches reach out from basic incorporation of additional segments that are not entirely important to working, in the event of failure in different segments to completely online versatile executions [3]. Fault tolerance methodologies could be classified into two types, keeping in view of the level of reflection at which deficiencies are endured.

The first manages shortcomings at the equipment level of the FPGA, subsequently the term device-level (DL). Here efforts are made to build a collection of fault-free hardware from a larger collection containing faulty resources. At this point when faults occur, routing changes are made within the FPGA, selecting the excess equipment asset to supplant the faulty ones. Nonetheless, the way that changes are made at the device level implies that endured shortcomings are straightforward to end-client instruments [3]. The second approach edures faults at a much higher-level, at the level of FPGA configuration. The setup level (CL) approach views

the FPGA as an arrangement of perfect graph structures, without considering the real physical structure of the device. In this method of fault tolerance, fault-free resources are selected from the set of available resources at the time of placement and routing of a circuit. The status of resources, either defective or fault free, are viewed as every time a circuit is placed and routed, so the CL technique can endure new faults in the locale. The CL strategies is not straightforward to the apparatuses, so resilience of new faults requires extra design time [3].

Below we describe a few of the fault tolerant approaches that lead up to the work done for this thesis effort.

1. Extra Rows: One of the first redundancy based fault tolerance methods, was introduced by Hatori et al. [1993] of Toshiba Corporation [3]. Along with the addition of extra wiring and extra rows of logic blocks, three other necessary additions had to be made to the design:

   (a) Selection logic was included between row and rows decoders.

   (b) Lengths of vertical wire portions were expanded by one row.

   (c) Extra arrangements of vertical wires were included.

   When a defective row is identified, the selection logic allows it to be bypassed during configuration. Figure 2.1 demonstrates this concept.

   The extra arrangements of vertical wires keep up the logical network as in the first circuit. However, there may be some degradation of the processing speed due to extra capacitance of the additional resources. When multiple defects at various levels of the same row are identified, more than one spare row is needed to tolerate the fault. This type of redundancy works

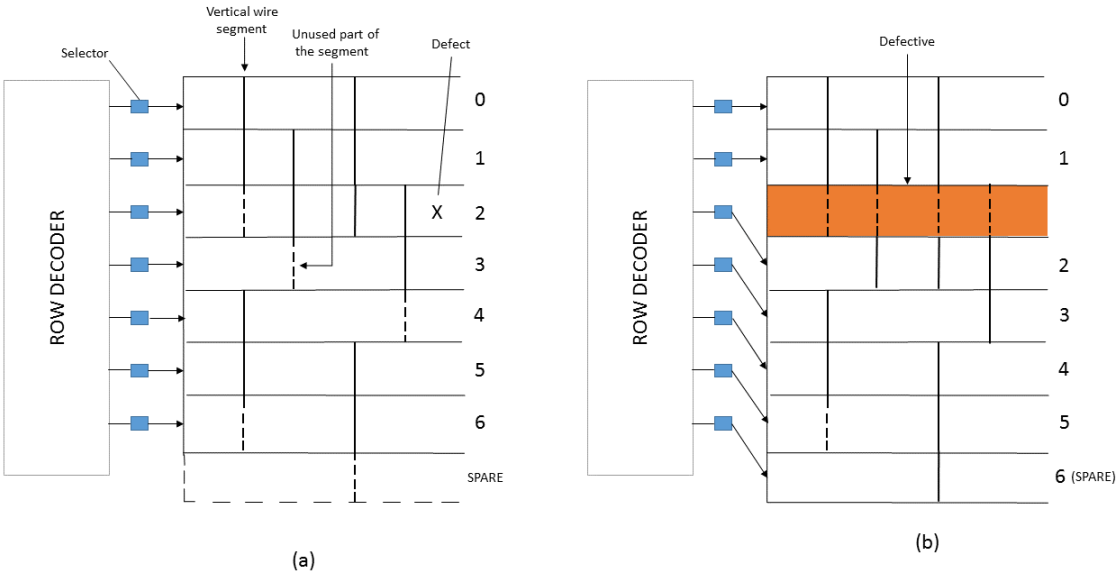


Figure 2.1: Arrangement of FPGA (a) before and (b) after fault is endured [3]

productively when the FPGA is separated in "blocks", where every piece gets extra row/rows and is administered by its own row decoder. Faults are analyzed amid assembling tests and are avoided utilizing spare row(s), as found in Figure 2.1. Thus, this strategy falls in the class of device–level (DL) adaptation to non-critical failure. There is no reconfiguration time discipline for this technique since reconfiguration is performed amid make. Expecting an arbitrary blame is disseminated, yield gauges demonstrated a yield of up to 80% for a typical FPGA with two extra rows contrasted with 20-30% without the extra rows [3].

2. Reconfiguration Network: The addition of a reconfiguration network is another fault tolerant method, proposed by Kelly and Ivey [1994], where the faulty resource is bypassed by using a secondary interconnect network [3]. Their architecture is comprised of programmable logic blocks (PLBs), an interconnect matrix with switchboxes at cross-points, and long interconnect

lines. A PLB and the switch box are treated as a separate structure, and each structure has a corresponding set of long lines immediately to its right.

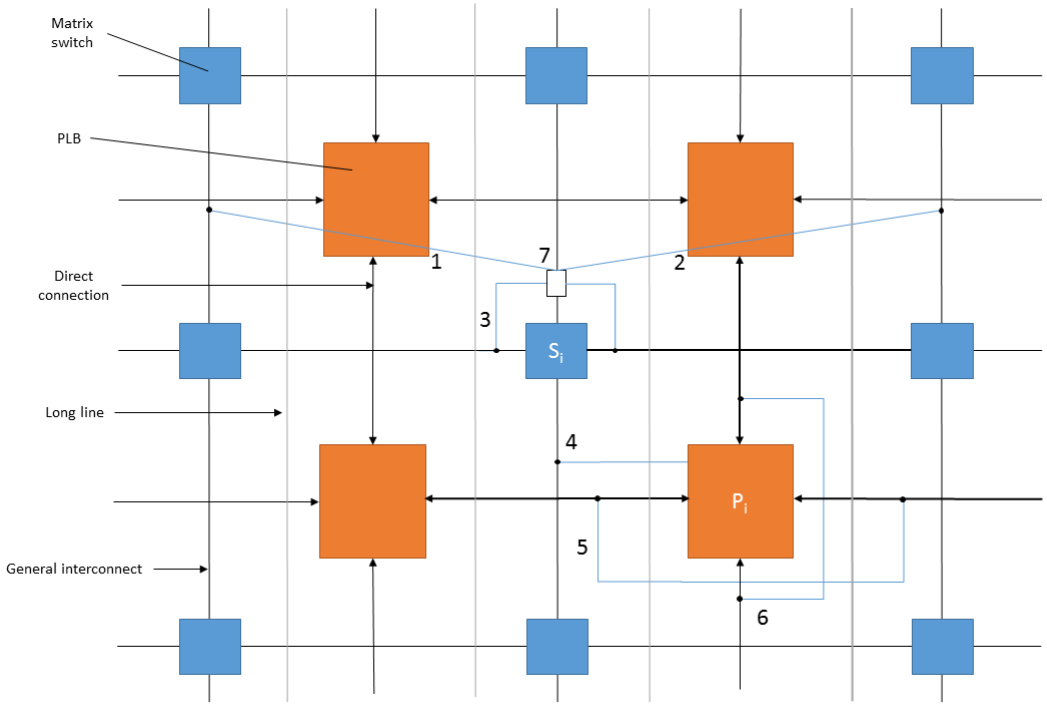


Figure 2.2: A PLB and switch pair is shown with labels. Alternate routing resources for pair i are shown by the blue lines. They are (1) connected left; (2) connected right; (3) switch bypass; (4) long line spur; (5) horizontal direct bypass; (6) vertical bypass. The small switch (7) controls the reconfiguration of switch Si – it can be used to bypass the switch specifically above Si to disconnect Si from the switch directly above it, or to bypass Si horizontally [3].

Four architectural changes are necessary to apply this method:

(a) Additional routing resources.

(b) Reconfiguration memory to store the area of defective assets.

(c) On-chip hardware to perform reconfiguration routing.

(d) Expansion of at least one column.

Initially, a custom testing process must be kept running on the FPGA to identify the broken assest. The outcomes turn into a fault guide for the device. The fault guided is put under the configuration memory of the FPGA

and is used for setting up a routing system that ceases from using the defective assets. The system might be developed by at first moving all PLBs in the line containing the fault towards the right (beginning from the broken PLB). For this to work properly, the association between the PLBs and Reconfiguration Network must be maintained. When faults are detected, shifted PLBs are connected diagonally. In the midst of this method, any fundamental affiliations are made over the flawed line. Last, goad associations are produced using the moved PLBs to keep up associations with the vertical worldwide routing. Once the routing reconfiguration is done, the present circuit course of action is stacked onto the chip. No adjustments to the FPGA outline/arrangement are required due to the on-board reconfiguration switch. The end client should just be worried with the vector loading the fault guide to the FPGA. As different faults are discovered, they might be added to the fault vector utilized by the on-board switch. Only faults occuring in vertical long lines are considered by this technique.

3. SRAM Shifting: Doumar and Ito [4] proposed a custom SRAM FPGA architecture and a technique that could be utilized with it for yield upgrade and additionally user adaptation to non-critical failure (FT) [3]. Their custom architecture comprises of a multiplexer and barrel shift registers.Each PLB and the routing resources above and to the left of it is considered as unit component. The configuration memory is moved vertically, horizontally, or serially with the help of the multiplexer that is attached to each unit. The unit component memories are intended to form vertical or horizontal barrel shift registers amid the vertical or horizontal move mode. The yield of the

component at the base of the section or toward the end of the line is encouraged back to the contribution of the principal component in the segment or column.

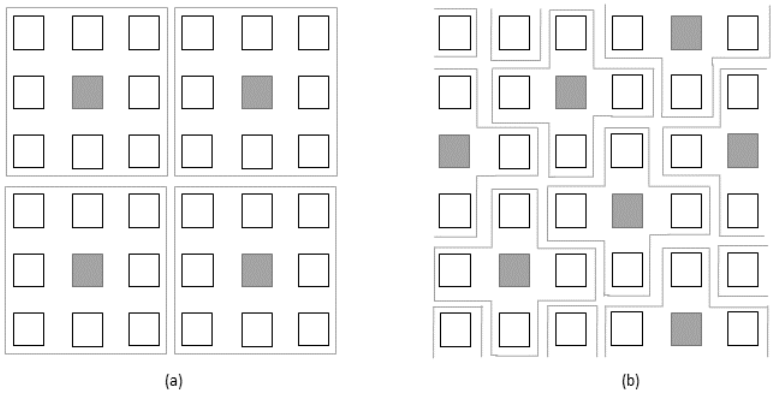


Figure 2.3: SRAM shifting (a) king and (b) horse spare allocation patterns [3]

Similarly, the plan bits related with the arrangement of programmable I/O cells on each of the four sides of the FPGA are organized as a solitary move enlist. The multiplexers are controlled by the shifter hardware.At first, the circuit is set and steered around an arrangement of extra unit components and stacked onto the FPGA. The defective components are recognized by testing and the data is bolstered into the shifter hardware of the FPGA. The shifter moves the entire circuit course of action so that the broken unit cells are up the held additional unit part [3].

Two diverse extra allotments were recommended: the king allocation and the horse allocation. Figure 2.3 illustrates the two methods. The dabbed edge in figure 2.3 around every extra demonstrates which PLBs can make utilization of that extra. An example of fault tolerance is shown in figure 2.4. In figure 2.4(a), the circuit is placed and routed in the king shifting pattern. The flawed cell is at the lower left-hand side of the framework.

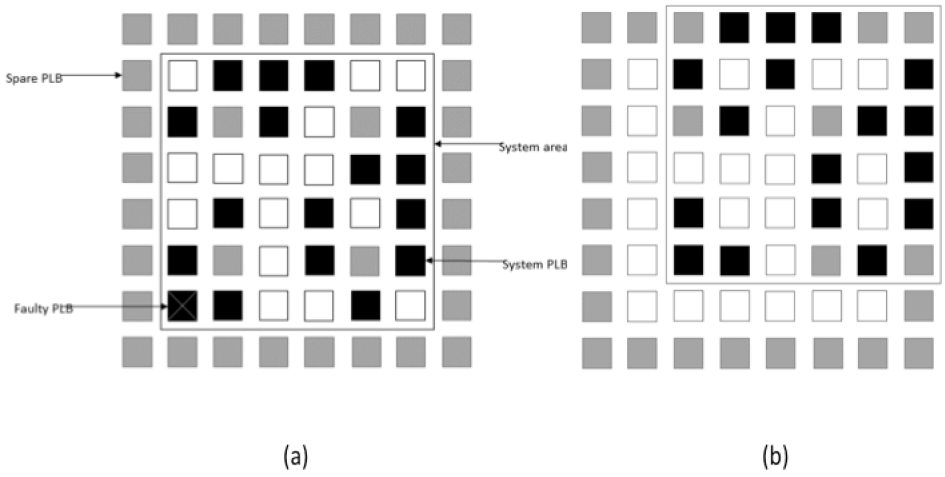


Figure 2.4: Fault tolerance with SRAM shifting architecture [3]

Figure 2.4(b) demonstrates the circuit after adaptation to internal failure, where the entire framework is moved up and to one side, so that the flawed cell is presently in an extra area.

4. Tiling: Lach et al. [1998, 1999] presented an offline logic and interconnect adaptation to internal failure that includes dividing the FPGA into tiles. In this approach, each tile comprises a bit of the system function and some extra logic and interconnect resources [3]. Here the OPLBs are assembled into bigger structures to minimize the impact of adaptation to non-critical failure on the general framework. Towards the starting, diverse arrangements of each tile are kept in the memory. Each of these set up places spare logic in a different location while maintaining the same tile boundary connections [3]. At the point when a faultis recognized in a tile, the tile is supplanted by a design that does not utilize the faulty resouces.Faults on the inter-tile interconnect may be tolerated when additional routing is specifically used to tolerate interconnect faults. An example of tiling fault tolerance is shown in Figure 2.5. The upper left tile in figure 2.5(a) consists of a faulty PLB.

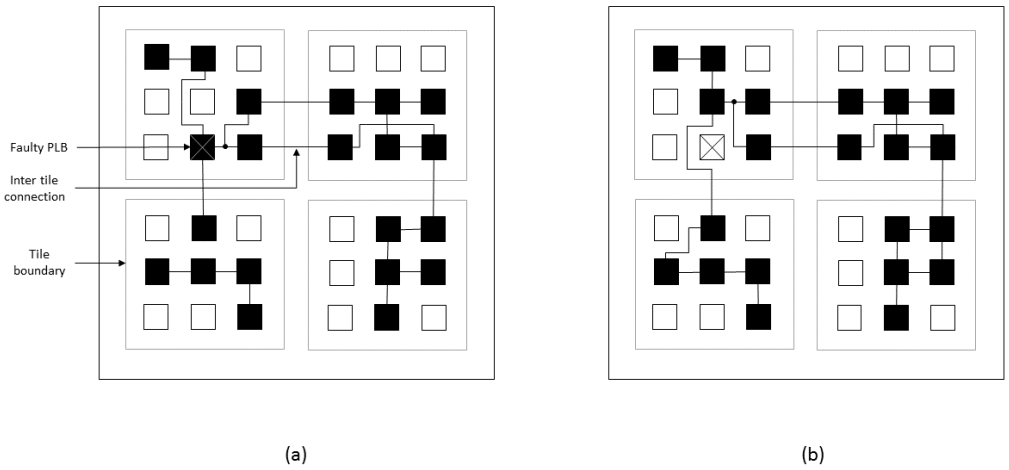


Figure 2.5: Fault tolerance with Tiling method [3]

In 2.5(b), the tile has been swapped for the cell which does not have a fault while maintaining the external connections.

5. Cluster-Based: Tessier and Lakamraju [5] suggests approaches to address intra-cluster faults and faulty segment interconnect situations [3]. Intra-cluster faults are addressed by exchange of LUT inputs, in cases of single faults in BLE multiplexers, LUT input wires or LUT SRAM bits; by the exchange of the entire BLE, in cases of LUT output failures, BLE flip flop failures or full usage of LUT inputs; and by the exchange of cluster inputs or outputs in cases of cluster I/O wire failures or multiplexer failures. Further, faulty segment interconnects and ineffective exchange methods are addressed by re-connecting nets affected by faults using incremental routing techniques.

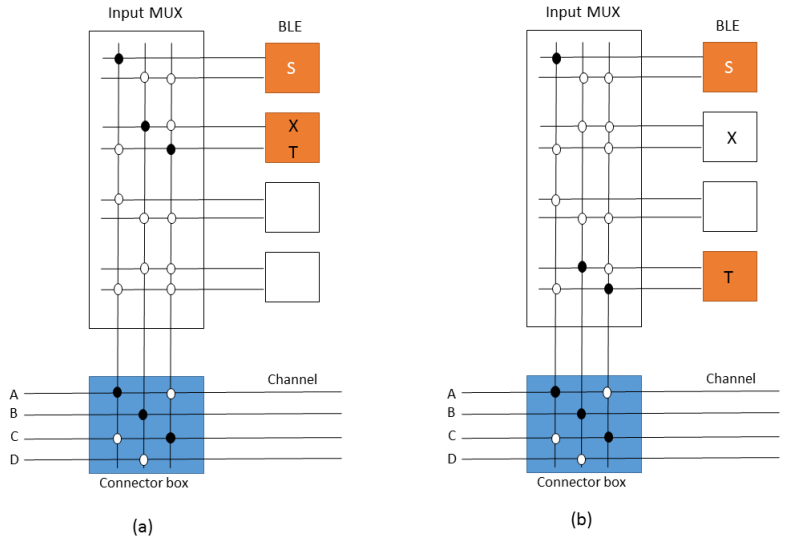


Figure 2.6: Example of Cluster-based Fault Tolerance [3]

2.3 Field Programmable Gate Array (FPGA)

A field programmable gate array (FPGA) can be described as a programmable integrated circuit for which the user (based on the required design function) can change the hardware configuration. There are many of these programmable integrated circuits available on the market. The main advantage that the FPGA offers over other integrated circuits is that they can be configured and reconfigured when desired. A standard IC is not programmable or reprogrammable which means that they have fixed interconnections between the transistors. They cannot be changed unless they have been physically damaged by unfortunate events or environmental conditions. An FPGA can be considered as an IC where there is additional circuitry added to allow reprogrammable functions and interconnections. Interconnections between the logic can be done according to the function the user defines.

So theoretically, almost any digital operation can be implemented by FPGAs, depending on the logic capacity [7]. In loose terms, almost any digital IC operation

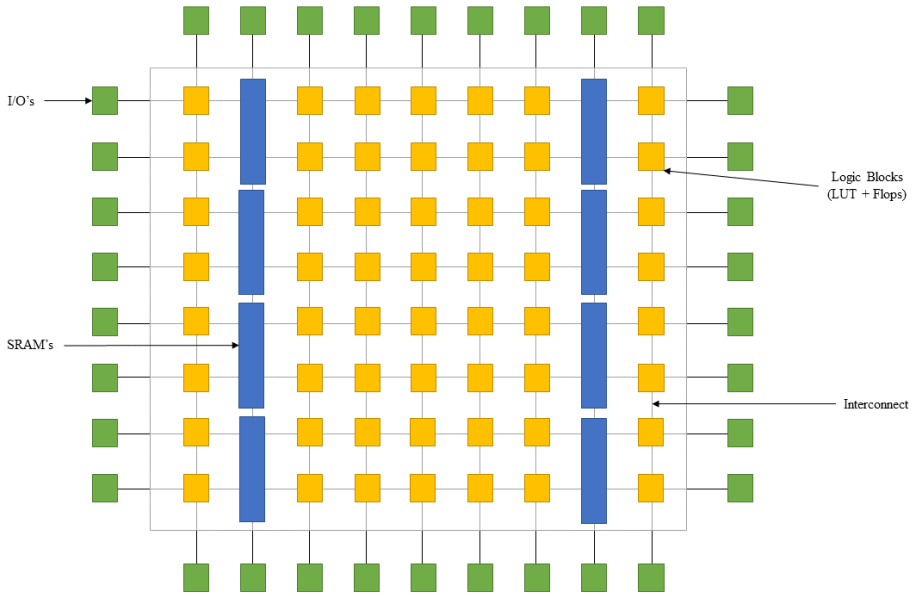


Figure 2.7: FPGA Architecture [12]

can be implemented (for example: operations of a microprocessor) on an FPGA. When it comes to the architecture of FPGAs, they usually consist of arrays of systematically arranged programmable logic blocks connected to each other by a programmable routing matrix. A FPGA setup describes the convenience of the FPGA, showing which logic blocks are used and which wire sections are used to partner them, furthermore what usefulness each block provides [3][6][9]. Various techniques for FPGA programming have been devised, of which the most prominent and current system is SRAM-based FPGAs, because of more number of reconfiguration cycles they permit and relative simplicity of programming. Figure 2.7 depicts a general structure of a FPGA. The primary parts of a FPGA are configurable rationale piece, configurable I/O and programmable interconnect. A clock circuit is available for driving clock signs to every logic block.

1. Configurable Logic Blocks (CLBs): These blocks contain the logic for the FPGA. The block contains RAM for creating arbitrary combinational logic

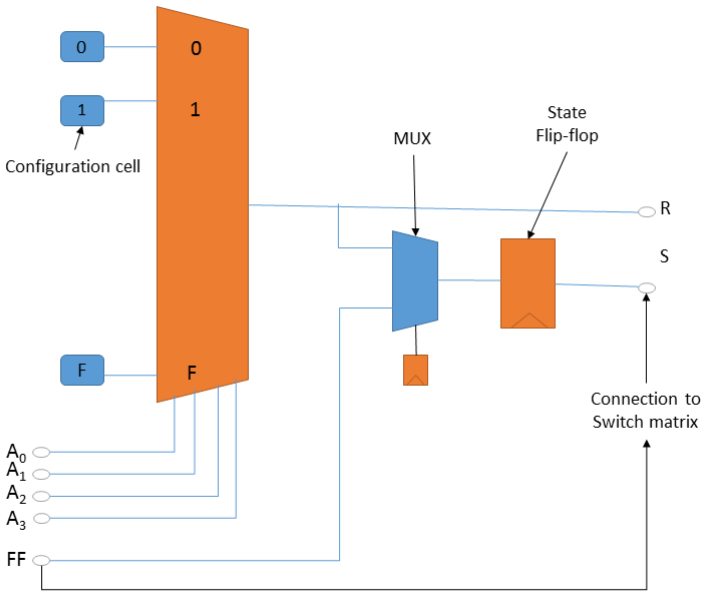


Figure 2.8: Implementation k=4 input LUT [11]

functions; also known as lookup tables (LUTs) [12]. These blocks also contain multiplexers in order to route the logic within the block and to and from external resources along with flip-flops for storing logic '1's and '0's. Look-up table is illustrated in figure 2.8.

2. Configurable I/O blocks: As the name suggests, the configurable block shown in figure 2.9 is used to send signals to and fro from the chip [12]. The configurable I/O block consists of an input buffer and an output buffer with three-state and open collector output controls. Typically the polarity of the output is programmed for active "high" or active "low" and slew rate of the output can be programmed for fast or slow rise and fall times. The flip-flops present at the outputs serve to reduce significant delay when clocked signals are directly applied to the output pins. Similarly, flip-flops on the inputs reduce delay on a signal before reaching a flip-flop, thus reducing the hold time requirement of the FPGA.

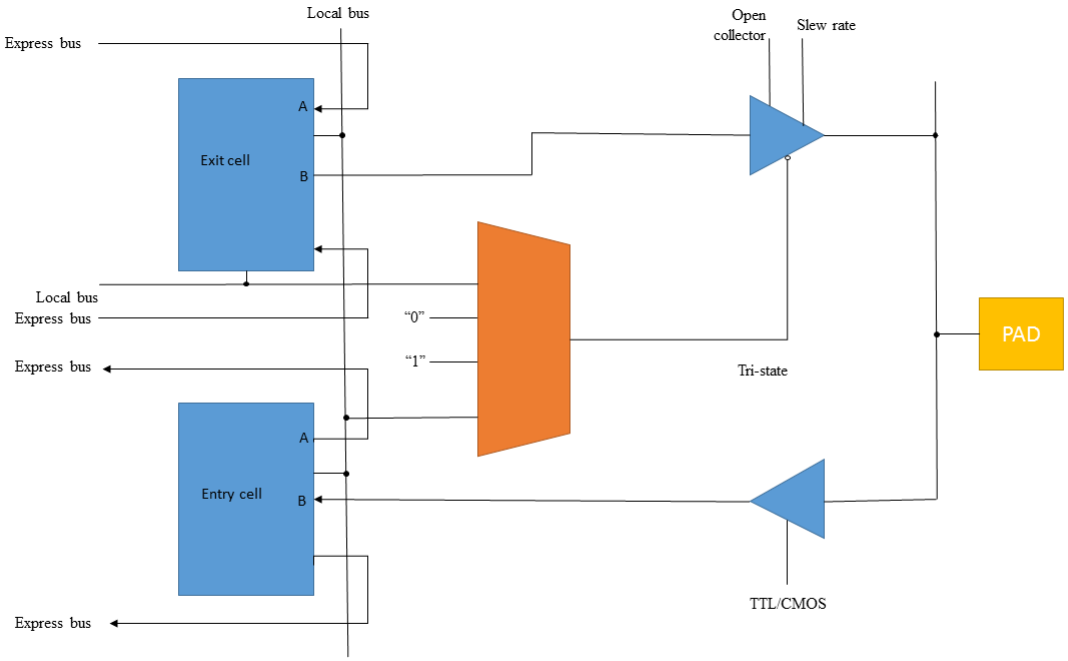


Figure 2.9: FPGA configuration I/O block [12]

3. Programmable Interconnect: The figure 2.10 shows the hierarchy of the interconnect system in the FPGA. The CLBs that are physically a long way from each other are associated utilizing long lines immediately. These long lines can likewise be utilized as buses inside the chip. There are additionally short lines that are utilized to interface CLBs that are found near each other. Transistors are utilized as switches to turn on or off associations between various lines. Programmable switches are utilized to interface these long and short lines together.

2.4 Fault Tolerance

A fault can be defined as a defect or unsatisfactory condition of a system, which may lead to failure, or illogical function of the system [2]. Fault tolerance is the methodology of the hardware system that in case of occurrence of any logical error

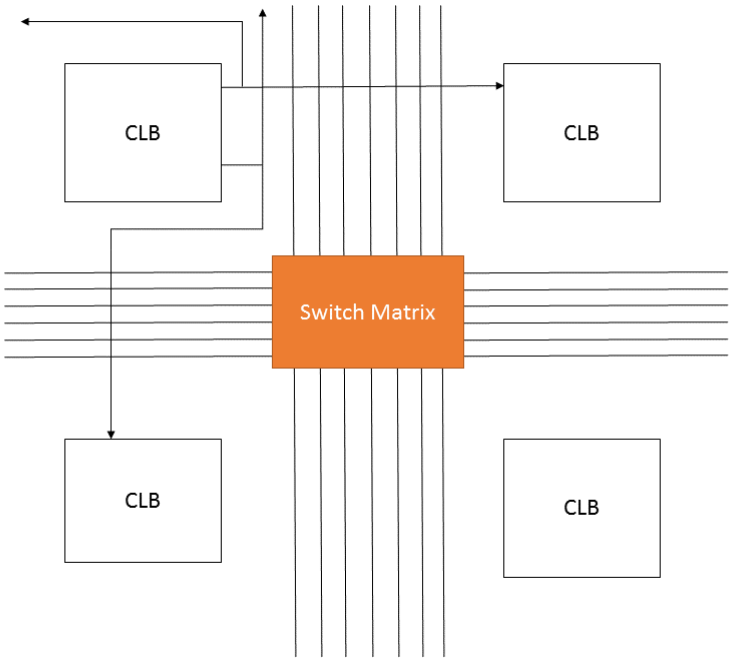


Figure 2.10: Programmable Interconnect [12]

or hardware faults, the correct logical working of the system can still be achieved by having an alternative backup procedure to immediately take up its place to maintain the precise, error-free functionality of the system [2]. One of the distinct advantages of using reconfigurable devices, such as FPGAs, is that the current functionality of the device can be altered or changed sometime in the future. This supplementary benefit allows the designers to completely reprogram the FPGA with a new logic. What if you want to change the logic within a part of an FPGA without disrupting the entire system? For example, you have a design that adds two numbers, now without disrupting the system and stopping the flow of data, you need to change the functionality from addition to subtraction, thereby you need a way to partially reconfigure the logic on the device.

2.5 Partial Runtime Reconfiguration (PRR)

Partial reconfiguration (PR) is a design process that allows design engineers to change the functionality of a portion of a FPGA system while the rest of the FPGA system remains intact. Runtime (R) reconfiguration is the process of re-programming the FPGA while it is actually still operating or actively performing its duties. Partial runtime reconfiguration (PRR) is the concept of partially re-programming the FPGA while the rest of the FPGA is operating or executing its application. These concepts are especially valuable when devices operate in a mission-critical environment that cannot be disrupted while some subsystems are being redefined [1].

With the help of PRR process, functionality of a single FPGA can be effectively increased. Even though a part of the design is being reconfigured, the rest of the system is operating with its usual functionality. There is no loss of performance or functionality with the unaffected (non-reprogrammed) portion of the design. It also allows for multiple applications on a single FPGA [1]. This increases system performance. The PRR design process gives us the opportunity to change or update the reprogrammable hardware any time, locally or remotely. Hardware sharing is achievable as partial reconfiguration allows you to run different applications on a single FPGA. Hardware sharing helps reduce power consumption, reduced device count, allows for smaller boards, and allows for overall lower costs. Configuration time is directly proportional to the size of the bitstream. Partial runtime reconfiguration allows you to make smalls changes without having to reconfigure the entire device.

Generating smaller bitstreams directly results in less time to reconfigure the device.

There are two approaches for partial runtime reconfiguration of FPGA devices: Module-based and Difference-based. Generally partial runtime reconfiguration is accomplished by loading the partial bit file of the updated design into FPGA configuration memory without stopping the execution of the FPGA.

1. Module-based Partial Runtime Reconfiguration

   The module-based partial runtime reconfiguration design process allows the FPGA to be divided into modules and by reconfiguring modules, reconfigure distinct parts of the FPGA [16]. Static modules consist of the parts of the design, which do not need to be reconfigured, and partial runtime reconfigurable modules are that part of the design that can be reconfigured according to the PRR design process. Specific buses must be designed to ensure communication between the static modules and the partial runtime reconfigurable modules. The issue with this plan stream is that the partial reconfiguration bit stream must be place in a settled district. Amid dynamic reconfiguration if the partial reconfiguration area is possessed by other paartial reconfiguration module, then another partial bit stream must be produced. This expands the cost of storing the partial bit stream [16]. As mentioned earlier, FPGA modules are categorized into two parts: static and partial reconfiguration modules. If these PRMs do not share dependent inputs then they need not be active at the same time. If they are not required to run in parallel then they can share common slots on the target device. Typically the bitstream is stored outside the FPGA, they can also be stored in the memory available inside the FPGA (BLOCK RAM). The

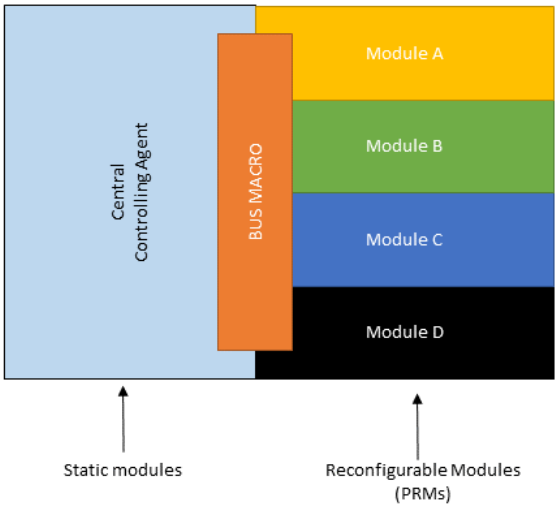


Figure 2.11: Module-based partial reconfiguration [16]

static part is that part of the system that is active during the entire runtime where the input design resides.

Notwithstanding its standard capacity it needs to give a framework to stack and empty other (element) parts of the outline, which requires framework planning, information administration, and interface administration [16]. The static part should incorporate logic blocks required for information and interface administration. All data sources and yields of the application are regulated by the static part that talks with the partial reconfiguration modules through a settled interface [16].

2. Difference-based partial runtime reconfiguration

Difference based partial runtime reconfiguration is useful to make small on the fly changes to the design parameters such as logic equations, filter parameters and I/O directions [18]. When making large changes in the

functionality or the structure of a design, this method is usually not preferred. In this design flow, a complete bitstream is generated and then the designer makes small changes using an FPGA editor and generates another sub-bitstream. The second bitstream is the difference between the two versions of the design. For example, when there are two partial reconfiguration modules, namely A and B. Rather than making the full design bitstream for each of them, we just make the full bitstream of form A. For module B, we take the distinction of its circuit portrayal memory frame-by-memory frame with that of module A and after that make a partial reconfiguration bitstream that lone alters the programming memory with the distinction [17]. An advantage is the second bitstream is smaller than the bitstream of the entire device bitstream. As mentioned previously, the main objective of the design is to make small design changes; for example, changing some LUT equations or changing some block RAM contents. Once the changes are made, a programming bit file can be produced. The new bitstream only programs the difference between the original design and the new one.

There are three methodologies to make small changes to the design:

(a) CHANGING LUT EQUATION

An FPGA editor can be opened using the programming file of the design from the design software once the design has been completely placed and routed. The smallest design element that can be selected using the software is the programming memory frame. After the programming memory frame that contains the LUT where a change is desired, the

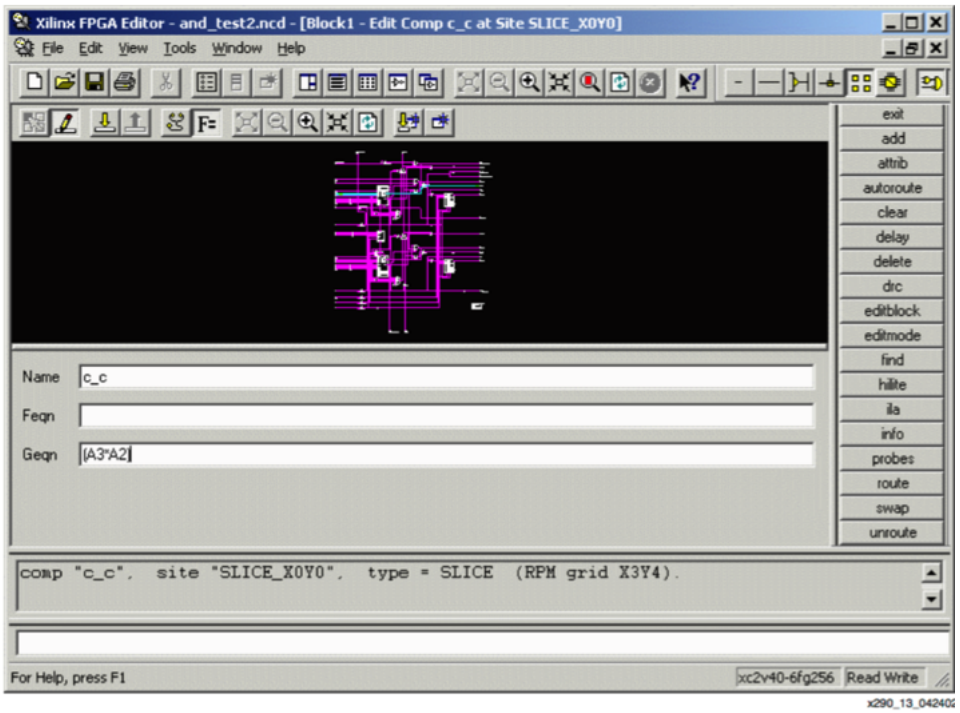


Figure 2.12: Example of changing LUT equation using Xilinx FPGA editor tool [18]

contents of the LUT are modified with new values. In figure 2.12 we see an example process using the XILINX ISE tool's editblock button. It is clicked to open a block editor toolbar where you can view a specific LUT equation as shown below.

In figure 2.12, a window opens with the LUT slice name and the corresponding equations. For the XILINX ISE tool example valid operators include:

- * $\Longrightarrow$ Logical AND
- + $\Longrightarrow$ Logical OR
- @ $\Longrightarrow$ Logical XOR
- ~ $\Longrightarrow$ Unary NOT

In this example, you can change the LUT equation from (A3 * A4) to (A3 * ~A4) to any other equation of the input variables. Once the

changes have been made, the changes are saved. Next a new programming memory file can be generated. By taking the difference of the original and the new programming file, you can generate a partial runtime reconfiguration bit file, which has only the difference of the original and new design.

(b) CHANGING BLOCK RAM CONTENTS

Block RAM contents are changed in a similar fashion as the LUT contents as shown in figure 2.13. A block RAM editor is used to change the values to be stored in the block RAM. Once changes have been made to the block RAM contents, the design is saved, and the same process is used to produce a partial reconfiguration bitfile.

(c) CHANGING I/O STANDARDS

Field programmable gate array I/O can be modified using an I/O editor. Figure 2.14 shows and example of using the Xilinx ISE I/O editor to change the programming of a Xilinx FPGA. Once the I/O have been modified, a partial configuration file can be generated to reconfigure the actual FPGA during PRR.

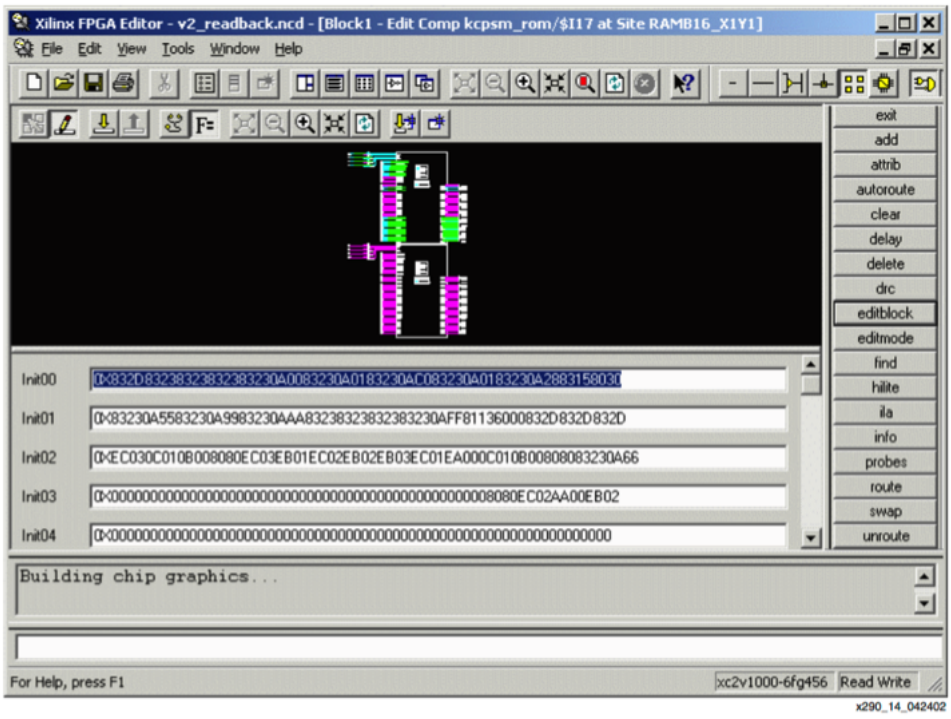


Figure 2.13: Example of changing block RAM contents using the XILINX ISE Block RAM Editor [18]

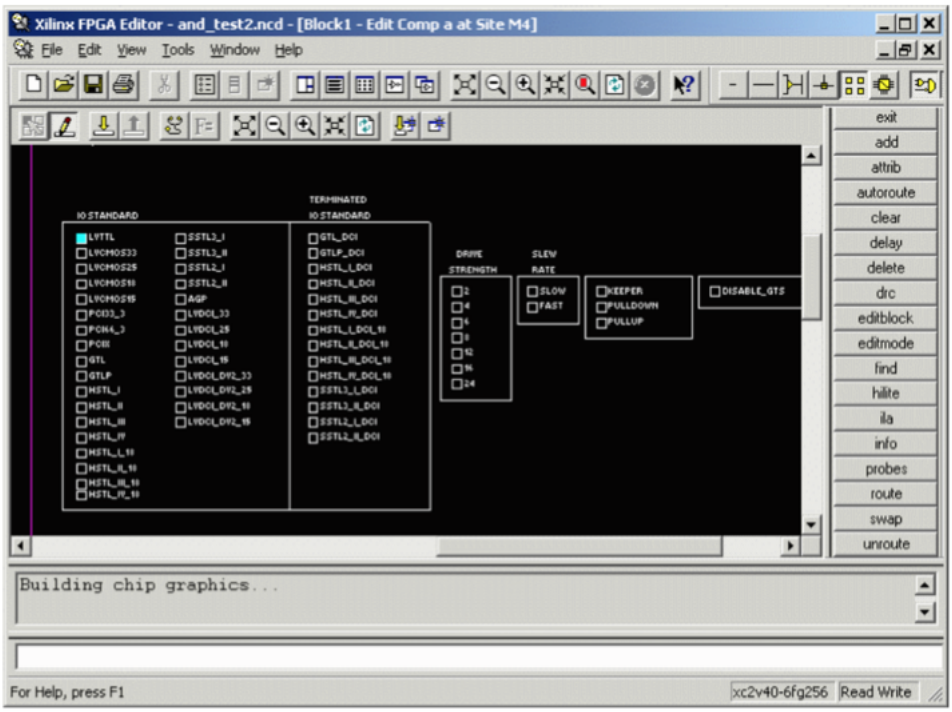


Figure 2.14: Example showing how I/O can be modified using the XILINX ISE Editor tool [18]

Chapter 3

FAULT TOLERANT APPROACH

In this section, we will discuss the design approach that we have taken to address partial runtime reconfiguration (PRR) for fault tolerant FPGA applications.

3.1 Overview of Methodology

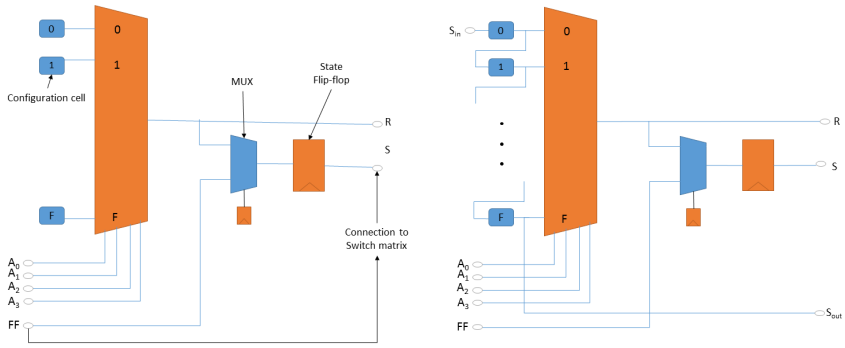
Fault tolerance is a method that makes the assumption that the system has unavoidable faults and aims to make provisions for the system to operate correctly even in the presence of faults [12]. Here we describe our procedure to tolerate faults. This particular work builds on previous work for logic fault tolerance. Our goal is to demonstrate PRR capabilities for interconnect using the existing ROVING STARS fault tolerant technique for FPGAs [20]. From knowledge of the roving process, the logic of the system is not fixed to a particular PLB in the FPGA. The same PLB can be time shared among different logic functions. We can improve the efficiency of the system by reusing faulty PLBs. We make use of faulty resources in a non-faulty mode [16]. For example, when we encounter stuck-at-0 or stuck-at-1 faults, we can change the functionality of the LUT to implement a function that requires a 0 or 1 in that faulty cell. In other words we can reprogram the FPGA so that a function that needs the faulty value uses

that LUT. If the LUT needs to be reconfigured to make use of the faulty cell, we may need to rearrange the inputs of the LUT so that the correct function in implemented in the LUT. To do this, we make as few changes as possible. In other words we create a partial reconfiguration file that only changes LUT values so the faulty cell is used properly (likely just one frame of programming memory), and if any input routings need to be changed, a partial file that moves as few routes as possible in order to make use of the faulty LUT.

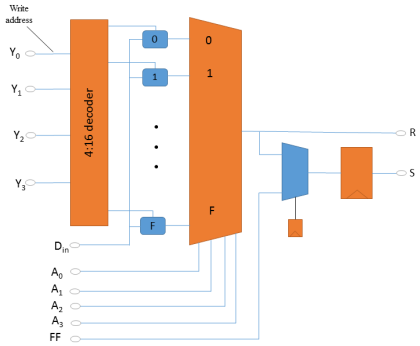
3.2 Look Up Table (LUT)

A loop up table (LUT) in an FPGA basically is like a truth table implemented as a small RAM memory. In other words, in terms of combinational logic, it describes the behavior of the truth table. A k-input look up table can implement any Boolean function of k input variables. The inputs are used as addresses that can retrieve values stored in the 2k memory cells that store the values in the truth table of the Boolean function that is stored in SRAM cells.Figure 3.2 gives the example for the configuration of different logic functions. Here we have to note that all inputs need not be used and the same Boolean function can be implemented by swapping the inputs. The LUT output is connected to a state flip-flop of a synchronous circuit. Most recent SRAM based FPGAs allow the user to configure most of the look up tables as shift registers as shown in figure 3.1(b) and figure 3.1(c) shows a LUT in optional memory mode.

When we go from logic level implementation to transistor level implementation the LUT is actually implemented using pass transistors as shown in figure 3.3. For a k-input look-up table, the multiplexer tree requires $k*2^k$ transistors plus



(a) k=4 input LUT (b) In shift register mode



(c) In memory mode

Figure 3.1: Examples of Look-Up table

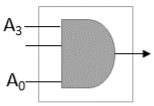
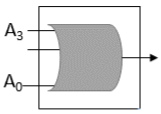
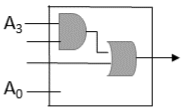
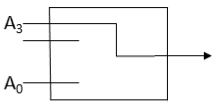
	A_3	A_2	A_1	A_0				
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	1	0
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	0	0
5	0	1	0	1	0	1	0	0
6	0	1	1	0	0	1	1	0
7	0	1	1	1	0	1	1	0
8	1	0	0	0	0	1	0	1
9	1	0	0	1	0	1	0	1
A	1	0	1	0	0	1	1	1
B	1	0	1	1	0	1	1	1
C	1	1	0	0	0	1	1	1
D	1	1	0	1	0	1	1	1
E	1	1	1	0	0	1	1	1
F	1	1	1	1	1	1	1	1

Figure 3.2: Look Up Table configuration example

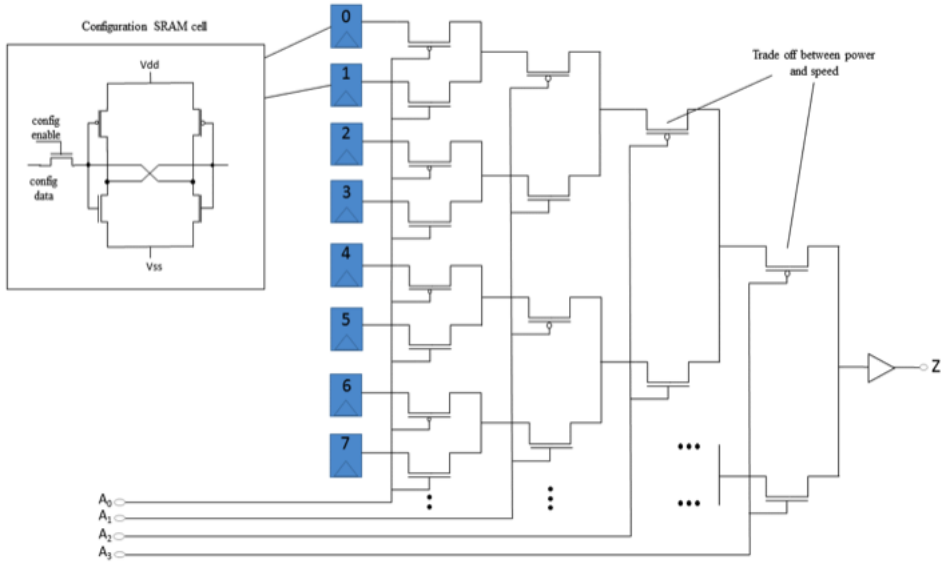


Figure 3.3: 4-input LUT implementation at the transistor level [11]

a level restorer to compensate the voltage drop of the pass transistors [11]. To implement an SRAM cell, a total of 5 transistors per cell or at least $5*2^k$ transistors is required. Therefore, a 4-input LUT requires at least $(5+k)*2^k = 9(16) = 144$ transistors [11].

3.3 FPGA Editor

The FPGA Editor is a graphical application for displaying and configuring Field Programmable Gate Arrays (FPGAs). Native Circuit Description (.ncd) file is required by the Xilinx FPGA editor to make the necessary operation as this file contains the logic of your design mapped to components such as CLBs and IOBs [18]. The following is a list of functions that is offered by most FPGA Editors:

• Place and route critical components before running the automatic place and route tools.

• Finish placement and routing if the routing program does not completely route your design.

• Add probes to your design to examine the signal states of the targeted device. Probes are used to route the value of internal nets to an IOB for observation during the debugging of a device.

• Cross-probe your design with a Timing Analyzer.

• Run the BitGen program and download the resulting BIT file to the targeted device.

• View and change the nets connected to the capture units of a ChipScope ILA core in your design.

• Use the ILA command to write a ChipScope Import document (.cdc) file.

• Create an entire design by hand (advanced users).

Again, these are example functions offered by the tool provided by Xilinx ISE tools suite for the procedure that we use to tolerate faults on the FPGA.

3.4 Proposed Methodology

To demonstrate the technique described in the ROVING STARS methodology we implement a simple boolean equation. For example, to accesses a 4-input LUT we need to implement the Boolean equation which looks like Z= BC'D'+AB'C'+A'B'D +A'CD+CD'. The truth able for the function is shown in figure 3.4. As an example we can say an error has occurred when you get a one in a cell where you expect a zero to be present. We make use of the ROVING STAR technique to tolerate such faults. To demonstrate the technique we propose three methods to tolerate the faults:

	A_3	A_2	A_1	A_0	Z
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	0
A	1	0	1	0	1
B	1	0	1	1	0
C	1	1	0	0	1
D	1	1	0	1	0
E	1	1	1	0	1
F	1	1	1	1	1

Figure 3.4: Truth table for Boolean equation of Z

• Changing the LUT Equation

One of the methods implemented to tolerate faults is to change the logic function of a particular cell where an error is detected. Figure 3.4 depicts the behavior of a LUT and Z gives the LUT equation. Let us say that we encounter a stuck–at-0 error in cell 10. Figure 3.5(a) represents the occurrence of a stuck-at-0 fault at cell number 10. To tolerate this error we can change the logic function of Z to a function that requires the use a logic zero at cell number 10. Figure 3.5(a) represents the logic of Z without an error and figure 3.5(b) a boolean function that make use of the error that occurs at cell number 10. With the help of an FPGA Editor we can change the LUT equation to make use of the stuck-at-0 fault.

Figure 3.6 shows the tool (FPGA Editor), D6LUT holds the Boolean equation where you can reconfigure the LUT equation. D6LUT shown in figure 3.6 holds the reconfigured Boolean equation (A=A1; B=A2; C=A3; D=A4). Once the LUT

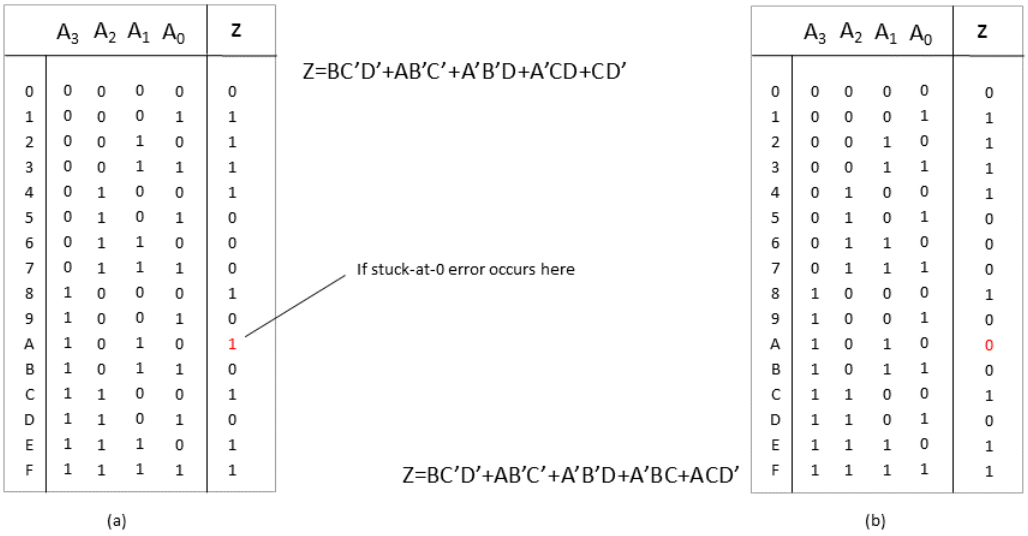


Figure 3.5: (a) Implementation of 4-input LUT without an error; (b) Implementation of 4-input LUT with an error

is reconfigured, a secondary bit file can be generated right from the tool itself. The difference between the first bit file (bit file of the original Boolean equation) and secondary bit file (bit file with the reconfigured LUT equation) will result in partial file that contains only the data needed to tolerate the fault. This small, resultant is loaded onto the FPGA so the faulty LUT is being correctly used.

• Changing The LUT Contents

Every FPGA vendor has their own CAD tools that map to their family of FPGA parts. Typically, they have an editor that allows modification of the LUT contents. For this discussion, we are using the Xilinx FPGA tools to demonstrate the process. A similar process can be used for almost any PRR capable FPGA.

Xilinx uses an *.RBT and *.BIT file to store the configuration of the FPGA. Figure 3.7 shows the file format. The *.RBT file is a raw version (ASCII) of the bit file. It contains the configuration information in binary format such that it can be read or modified with a text editor. However, to modify it, you need to know the map of the FPGA memory frames. Which bit goes in which frame and

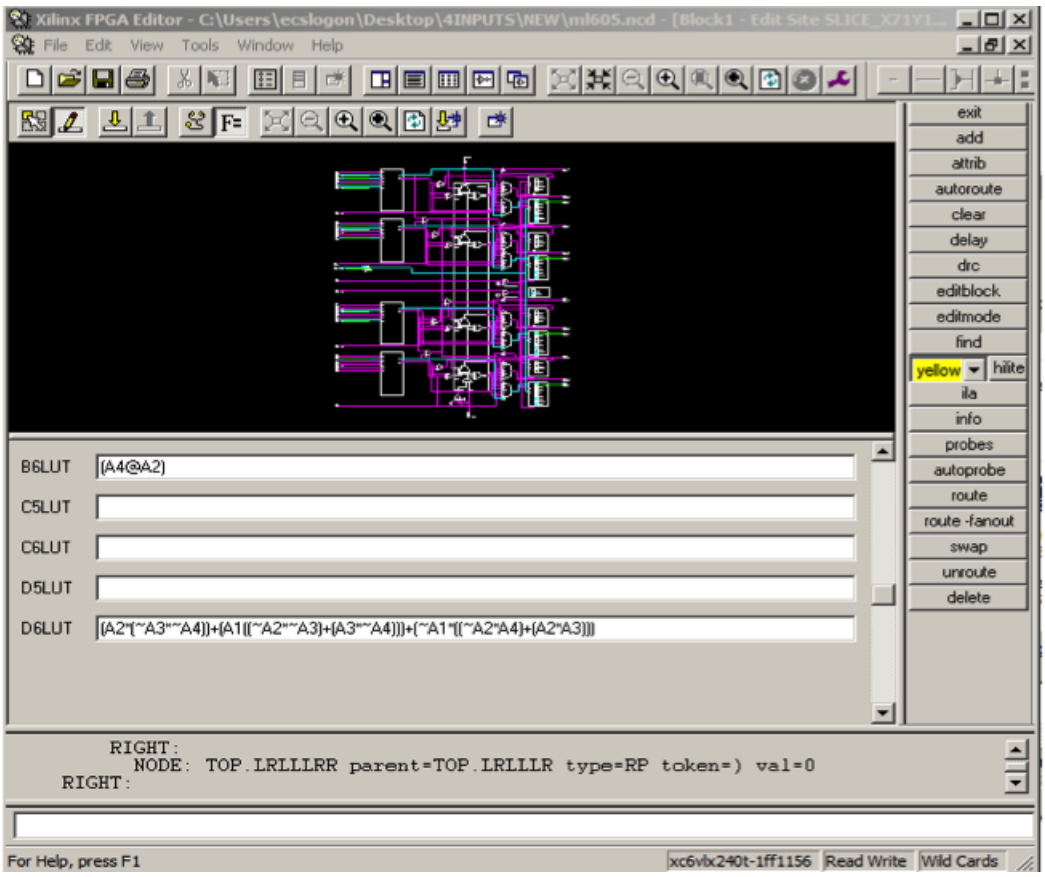


Figure 3.6: FPGA Editor where D6LUT holds the Boolean equation

```
Xilinx ASCII Bitstream
Created by Bitstream 0.87xd
Design name:     ml605.ncd;UserID=0xFFFFFFFF
Architecture:    virtex6
Part:            6vlx240tff1156
Date:            Thu Jan 28 14:06:50 2016
Bits:            73859552
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
00000000000000000000000010111011
00010001001000100000000001000100
11111111111111111111111111111111
11111111111111111111111111111111
10101010100110010101010101100110
00100000000000000000000000000000
00110000000000100000000000000001
00000000000000000000000000000000
00110000000000001000000000000001
00000000000000000000000000000000
00100000000000000000000000000000
00110000000000001000000000000001
00000000000000000000000000000111
00100000000000000000000000000000
00100000000000000000000000000000
00110000000000100010000000000001
00000000000000000000000000000000
00110000000000100110000000000001
00000000000000000000000000000000
```

Figure 3.7: Example of .rbt file

which frame LUT, RAM, Switch Information etc. is stored.

Figure 3.7 shows the beginning part of a configuration file in the raw (*.RBT) format. It is much easier to modify than the (*.BIT) file format because the *.BIT file is in binary format. This raw bit format contains all the configuration information. It contains the values of the LUTs etc. We know that the LUT depicts the behavior of a truth table. We can say that the Raw Bit file contains the values of the output of the truth tables. When a stuck-at-0 or stuck-at-1 occurs at the any cell we can fix the value of that particular cell in the raw bit file and make sure that any FPGA application using that LUT has the stuck value in that memory location. When we need to tolerate the faulty value, we try to make as few modifications to the file as possible in order to minimize the size of the partial configuration file that needs to be downloaded to tolerate the fault.

To locate the contents of a LUT in the *.RBT file, we can use the following procedure. A simple program file is written where the first and last cell of the LUT is set high. For example, Z = (a' AND b' AND c') AND (a AND b AND c) this gives a value of "1" for the addresses "000" and "111". A complete bit file is generated. Now in the FPGA Editor the equation is changed to Z = (a AND b AND c) and another complete bit file is generated. If we find the difference between the first and second bit files, we will be able to locate where the contents of the LUT is located in the *.RBT file.

• Changing Switch Box connection

Sometimes, we need to not only change the values in LUT cells to tolerate faults, but we have to actually re-arrange the inputs to accommodate the stuck memory cell. To do this, we need to re-route some of the signal nets coming into the LUT

```
> Xilinx ASCII Bitstream
6c6
< Date:         Thu Jan 28 14:06:50 2016
---
> Date:         Thu Jan 28 16:23:54 2016
126791d126790
< 00000000000000000001000000000010
126814d126812
< 00000000000001010000000000000000
126815a126814
> 00000101000001010000000000000000
126872d126870
< 00000000000000000001000000000010
126895c126893,126895
< 00000000000001010000000000000000
---
> 00000000000000000000000000000000
> 00000000000000000000000000000000
> 00000101000001010000000000000000
2307596c2307596
< 01000111011001101011110101001111
---
> 10010010101010001111010010110010_
```

Figure 3.8: Difference between the two files

inputs. Again, we want to change as little as possible to minimize the partial reconfiguration files used to tolerate faults. To determine where the inputs we need to rearrange are located in the *.RBT file, we apply a similar process as that to determine LUT locations. We then take the difference of the files. This allows us to create a minimum sized reconfiguration file. In generic terms we can describe this process below. Switch-Box (SB): A switch box is collections of transistor-based switches located between CLB blocks that enable the connection of two interconnect lines. Figure 3.9 shows a general diagram of an FPGA and Switch Block (SB) is box, which connects CLBs and CBs.

Figure 3.10 depicts a switch box with interconnections. The inputs are coming in from the left, the blue lines represent the connections of inputs to all the other connection lines. Figure 3.11 (a) is a representation of a switch box with simple connection to the inputs of a LUT. This type of input arrangement results in the truth table example shown in figure 3.11 (a). We also know that a LUT depicts the behavior of a truth table. Again when an error is encountered at a particular cell,

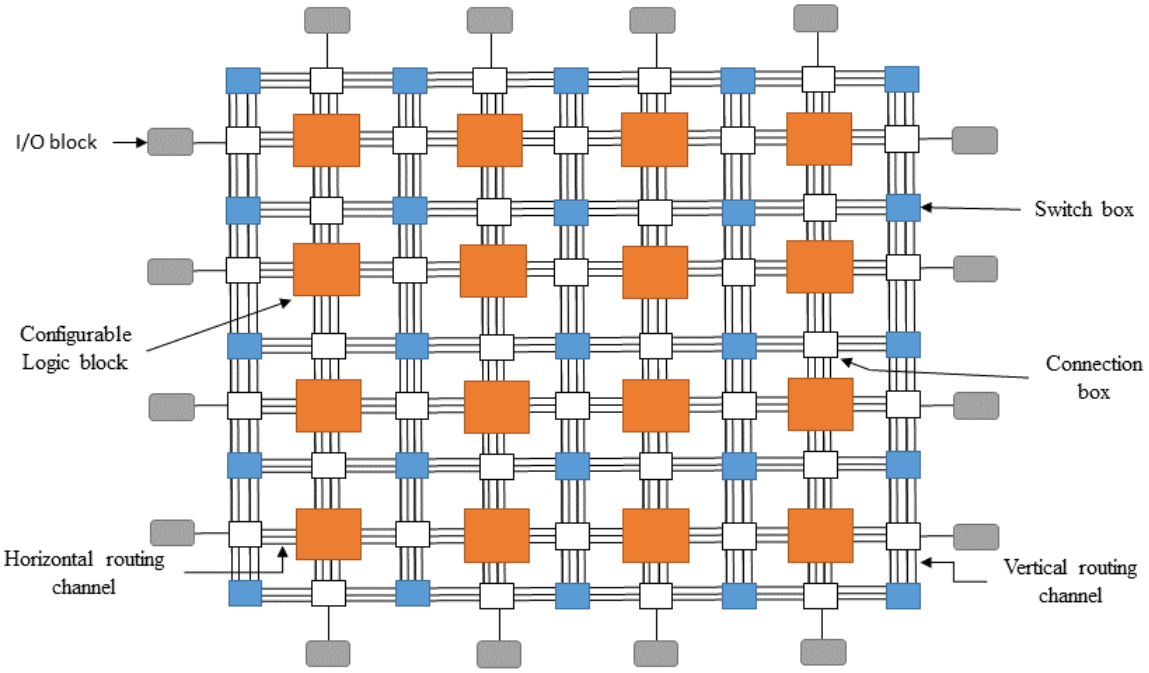


Figure 3.9: FPGA architecture that shows Switch box and Connector box connection[11]

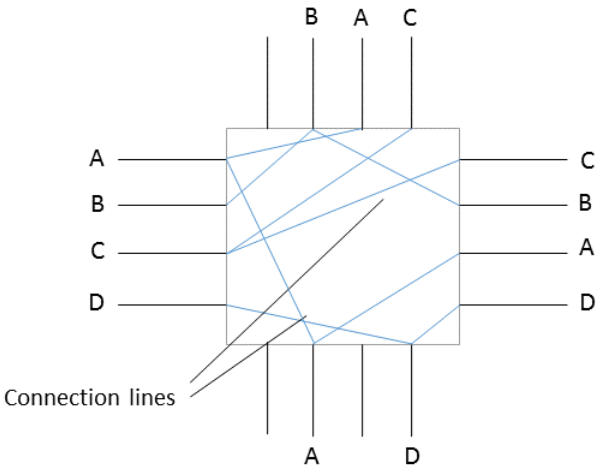


Figure 3.10: Switch box

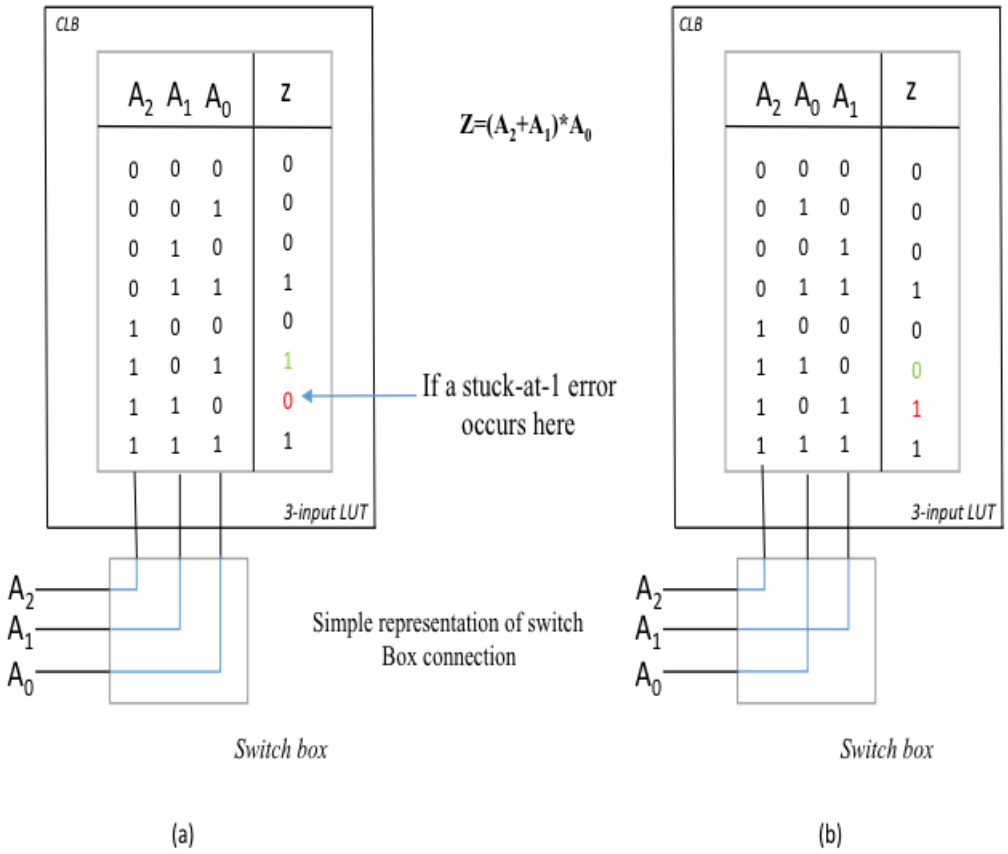


Figure 3.11: (a) 3-LUT; (b) 3-LUT with interchanged switch box connections

it must be tolerated. In order to do so, we can change the connections in switch box and change the way the inputs are fed to the LUT. We need to reroute the connection in the switch box in such a way that the input fed to LUT is changed, this is shown in figure 3.11 (b). When a stuck-at-1 fault occurs for the memory cell for the value "110" as shown in figure 3.11 (a), the switch box connections are rerouted to make use of a faulty resource as shown in figure 3.11 (b). Again, a bit file is generated before and after rerouting the switch box connection to tolerate the fault. The difference between the two is taken, and the resulting partial file contains the information of the changed routing of the switch. This partial file is then loaded into the FPGA.

Chapter 4

IMPLEMENTATION

4.1 Demonstration

Fault tolerance was achieved by implementing the concept of the ROVING STARS technique [20]. Partial Runtime Reconfiguration (PRR) was adapted to achieve fault tolerance for the procedure discussed in the thesis. For demonstration purposes, we used Xilinx virtex-6 FPGA. A simple 4-input program is written in VHDL, the main idea is to display the contents of the LUT on the given LCD screen and observe the implementation. To make it simple and to readily figure the data present in the .rbt file, AND operation is performed on all the four inputs so that only the last memory cell of LUT holds a '1'. The new partial bit file for this Boolean function is then generated and loaded on to the board.

Figure 4.1 shows the values that are present in a 4-input LUT. According to the idea discussed in this thesis, if a stuck-at-1fault occurs at the 14th memory cell we need to reconfigure the memory to make use of that fault. We need the faulty value used correctly in some function. When it comes to changing to the LUT equation, as mentioned in section 3.4, the FPGA Editor tools come in handy, and we can easily change the Boolean function to reuse the faulty cell of a LUT as

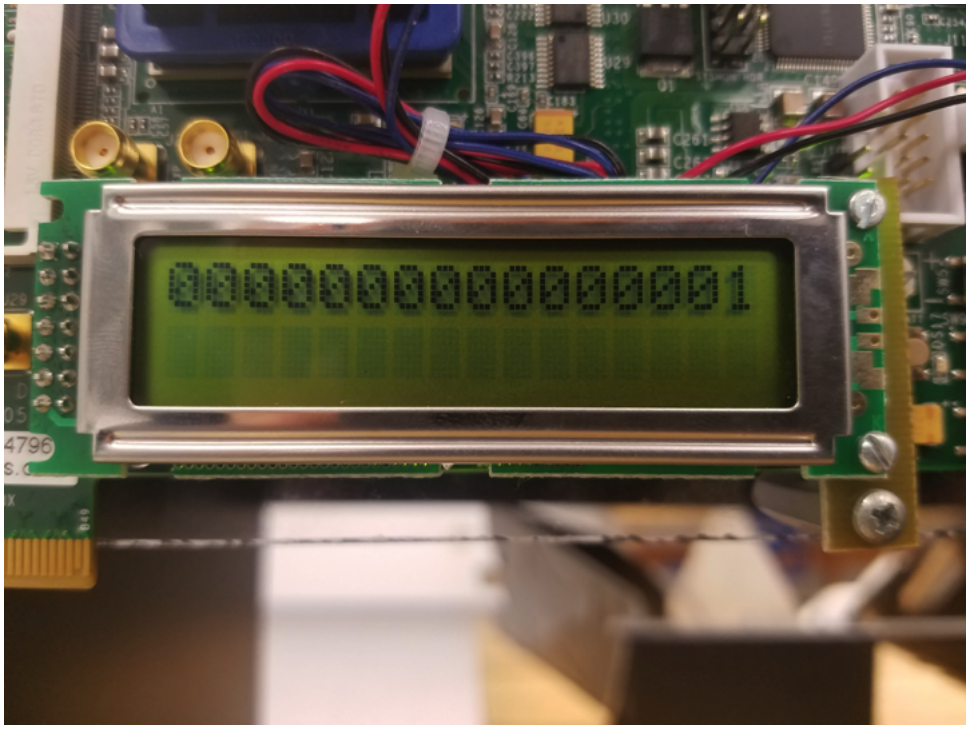


Figure 4.1: Boolean function of Z = (a AND b AND c AND d)

shown in figure 4.2.

The FPGA command line input offers a number of useful commands, one such command line instruction is BitGen. A few BitGen command line options are listed:

- -b (create rawbits)
- -bd (update block rams)
- -j (now BIT file)
- -r (create a Partial Bit File)
- -w (overwrite existing output file)

One of the most useful BitGen output file formats is .rbt file format. This is the same as the bit file, but in ASCII format. It is produced when bitgen –b is specified. The BitGen command line syntax is:

bitgen [options] infile[.ncd] [outfile]

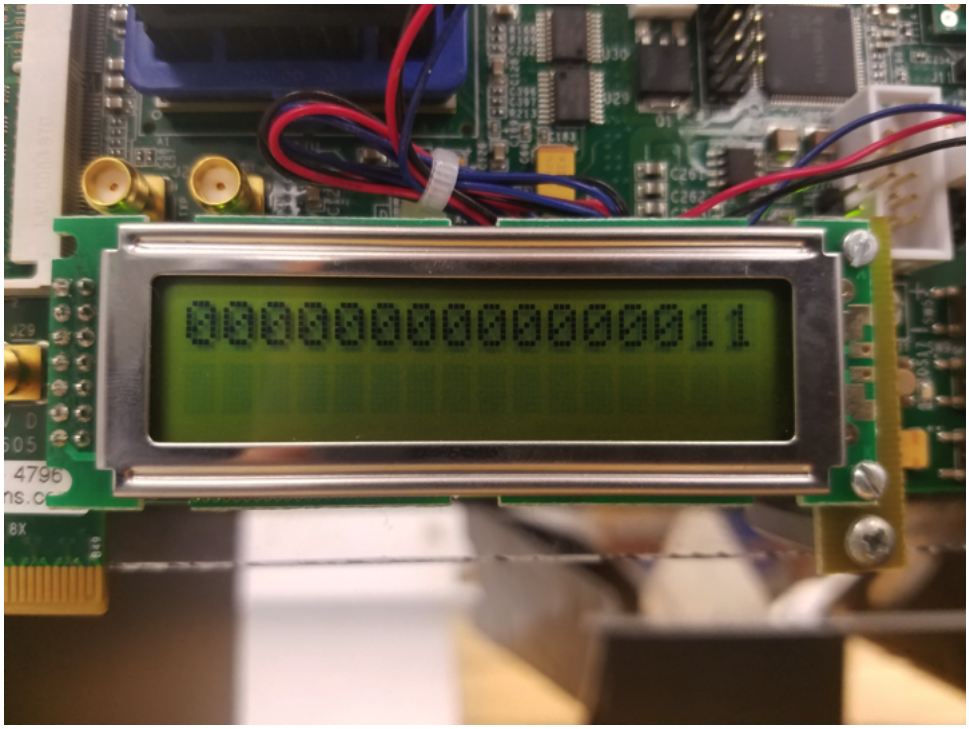


Figure 4.2: Boolean function making use of the faulty cell

$\Longrightarrow$ Options are one or more options listed in the BitGEn command line options, some of them are mentioned above.

$\Longrightarrow$ Infile is the name of the Native Circuit Description (NCD) format for which you want to create the bitstream.

$\Longrightarrow$ Outfile is the name of the output file. If you don't specify an output name, BitGen creates a Bitstream (BIT) file in your input directory. For our proposed procedure, we need a Raw (RBT) file format.

With the help of the FPGA editor tools and their command line options, a partial bit file is generated to tolerate a fault.

Another procedure to achieve this is by changing the contents of the LUT in the .rbt file and generating a partial bit file. To find contents of the LUT in the .rbt file, we generate an .rbt file of the Boolean equation where the first and last memory cells of the LUT is set to '1'. Then we generate two files where the files

generates values where only the first cell is set to '1', where as the second file contains values where only the last memory cell set to '1'. The difference of these two files with the files the where the first and last memory cell is set to '1' will give the information of the LUT contents. We can directly change the value from '1' to '0' or vise-versa and generate a partial bit using an FPGA Editor. Again, loading the partial bit file, with the necessary changes that is required to tolerate the fault, will result in the LUT value which is shown in figure 4.2. Similarly, the same result can be achieved by swapping the inputs to the LUT, as described in Section 3.4. FPGA editor offers a swap command to do so.

4.2 Limitations

While two fault tolerant methods implemented by changing LUT equation and LUT contents where successfully implemented and demonstrated, the third proposed method rerouting the switch box connection was unsuccessful due to software limitation. The swap command offered by the FPGA editor swaps the LUT pins along with the signal connections (we prefer not to rip up the entire route). For the case of rerouting the switch box connections, the following two conditions should be considered:

1. When interchanging the switch box connections, as shown in figure 4.3, we must notice the first and the last memory cell. No matter which way the connection are exchanged between the LUT inputs, if a fault occurs in this position it would be difficult to overcome.

2. In figure 4.3 (a), observe the LUT output for the fifth memory cell. When a stuck-at-1 fault occurs at the sixth position of the memory cell, it can be

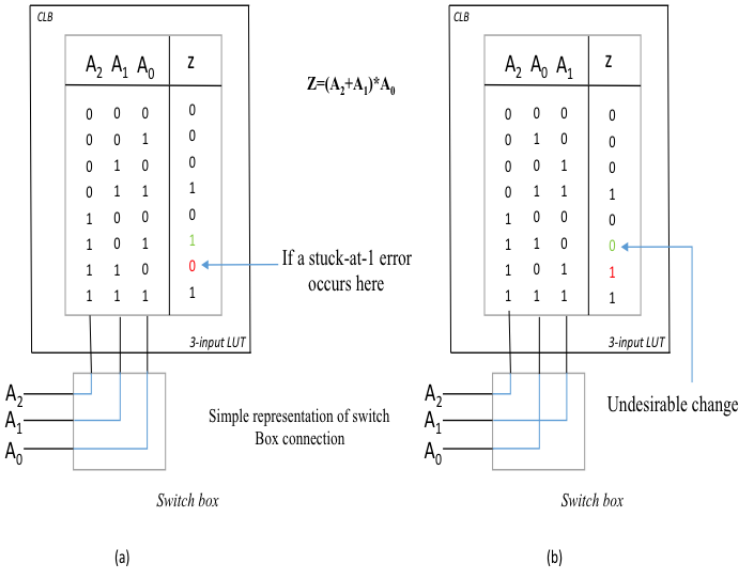


Figure 4.3: (a) 3-input LUT with a fault; (b) 3-input LUT with an undesirable change

tolerated by interchanging the way the inputs are fed to the LUT. When we do this, some of the other memory cell values may also change. As we can see in figure 4.3 (b) the output value of the fifth memory cell is also changed. In some cases, this can become an undesirable change that we must keep in mind when we reroute the switch box connections.

Chapter 5

CONCLUSION

5.1 Summary

In this thesis, partial reconfiguration files for tolerating faults using the ROVING STARS technique was implemented. The was demonstrated using partial runtime reconfiguration technology on a Xilinx Virtex-6 FPGA board. Three different methodologies were proposed in this thesis which are as follows:

1. Changing LUT equation
2. Changing LUT contents
3. Changing switch box connections

Changing LUT equations and LUT contents were successfully implemented and demonstrated, whereas changing Switch box connections was left for future research work.

5.2 Future work

Below, we briefly list some of the important future works.

1. Automating the configuration of partial bit file on the detection of a fault.

2. Finding an alternative way to reroute the switch box connection.

3. Developing an efficient algorithm to overcome the limitation that occurs when changing the switch box connections.

REFERENCES

[1] https://en.wikipedia.org/wiki/Redundancy

[2] https://en.wikipedia.org/wiki/Fault_tolerance

[3] Cheatham, Jason A., John M. Emmert, and Stan Baumgart. "A survey of fault tolerant methodologies for FPGAs." ACM Transactions on Design Automation of Electronic Systems (TODAES) 11.2 (2006): 501-533.

[4] Doumar, Abderrahim. "Defect and fault tolerance SRAM-based FPGAs by shifting the configuration data." IEICE TRANSACTIONS on Information and Systems 83.5 (2000): 1104-1115.

[5] Lakamraju, Vijay, and Russell Tessier. "Tolerating operational faults in cluster-based FPGAs." Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays. ACM, 2000.

[6] Emmert, John M., Charles E. Stroud, and Miron Abramovici. "Online fault tolerance for FPGA logic blocks." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 15.2 (2007): 216-226.

[7] Stroud, Charles, et al. "BIST-based diagnosis of FPGA interconnect." Test Conference, 2002. Proceedings. International. IEEE, 2002.

[8] Doumar, Abderrahim. "Defect and fault tolerance SRAM-based FPGAs by shifting the configuration data." IEICE TRANSACTIONS on Information and Systems 83.5 (2000): 1104-1115.

[9] http://fpgacenter.com/fpga/index.php

[10] https://www.altera.com/en_US/pdfs/literature/wp/wp01003.pdf

[11] Koch, Dirk. Partial Reconfiguration on FPGAs: Architectures, Tools and Applications. Vol. 153. Springer Science & Business Media, 2012.

[12] http://www.eetimes.com/document.asp?doc_id1274496

[13] https://en.wikipedia.org/wiki/Fault_tolerance

[14] http://searchdisasterrecovery.techtarget.com/definition/faulttolerant

[15] https://en.wikipedia.org/wiki/Fault-tolerant_computer_system

[16] Sasamal, Trailokya Nath, and Rajendra Prasad. "Module based and difference based implementation of partial reconfiguration on FPGA: A review." International Journal of Engineering Research and Applications (IJERA) 1.4 (2011): 18981903.

[17] Sedcole, Pete, et al. "Modular dynamic reconfiguration in Virtex FPGAs." IEE ProceedingsComputers and Digital Techniques 153.3 (2006): 157164.

[18] Eto, Emi. "Differencebased partial reconfiguration." XILINX Application Note, XAPP 290 (2003).

[19] Kao, Cindy. "Benefits of partial reconfiguration." Xcell journal 55 (2005): 6567.

[20] Emmert, John M., Charles E. Stroud, and Miron Abramovici. "Online fault tolerance for FPGA logic blocks." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 15.2 (2007): 216-226.

[21] Emmert, John, et al. "Dynamic fault tolerance in FPGAs via partial reconfiguration." FieldProgrammable Custom Computing Machines, 2000 IEEE Symposium on. IEEE, 2000.

[22] Farooq, Umer, Zied Marrakchi, and Habib Mehrez. "FPGA architectures: An overview." Tree-based Heterogeneous FPGA Architectures. Springer New York, 2012. 748.

[23] Betz, Vaughn, Jonathan Rose, and Alexander Marquardt. Architecture and CAD for deep-submicron FPGAs. Vol. 497. Springer Science & Business Media, 2012.

[24] Masud, Muhammad Imran. FPGA routing structures: A novel switch block and depopulated interconnect matrix architectures. Diss. University of British Columbia, 2000.

[25] Guide, Early Access Partial Reconfiguration User. "Xilinx Inc." San Jose, CA, USA (2008).

[26] Guide, FPGA Editor User Guide. "Xilinx Inc." San Jose, CA, USA (2008).

[27] Guide, Command Line Tools User Guide. "Xilinx Inc." San Jose, CA, USA (2008).

[28] Lim, Davin, and Mike Peattie. "Two flows for partial reconfiguration: Module based or small bit manipulations." Application Note XAPP 290 (2002).

[29] Kota, Solomon Raju, et al. "Module based implementation of Partial Reconfiguration using VHDL on Xilinx FPGA." International Journal of Recent Trends in Engineering 2.7 (2009): 1921.

[30] Mitra, Subhasish, Philip P. Shirvani, and Edward J. McCluskey. "Fault location in FPGA-based reconfigurable systems." IEEE Intl. High Level Design Validation and Test Workshop. 1998.