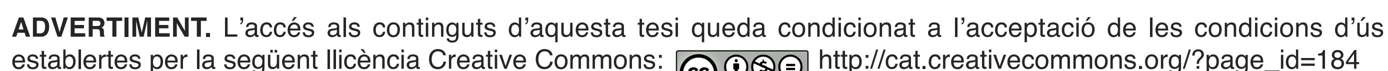
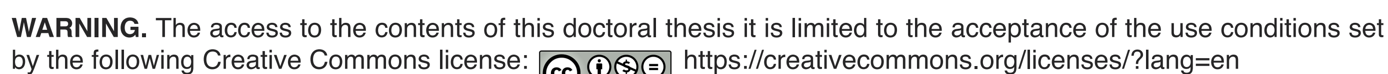
Universitat Autònoma de Barcelona



ADVERTIMENT. L'accés als continguts d'aquesta tesi queda condicionat a l'acceptació de les condicions d'ús establertes per la següent llicència Creative Commons: http://cat.creativecommons.org/?page_id=184

ADVERTENCIA. El acceso a los contenidos de esta tesis queda condicionado a la aceptación de las condiciones de uso establecidas por la siguiente licencia Creative Commons: http://es.creativecommons.org/blog/licencias/



WARNING. The access to the contents of this doctoral thesis it is limited to the acceptance of the use conditions set by the following Creative Commons license: https://creativecommons.org/licenses/?lang=en

Universitat Autònoma de Barcelona
Microelectronics and Electronic Systems Department

Dynamic Partial Reconfiguration in FPGAs for the Design and Evaluation of Critical Systems

Luis Andrés Cardona Cardona

Advisor: Prof. Carles Ferrer Ramis

A thesis submitted for the degree of

PhD in Electronic and Telecommunication Engineering

2016

Universitat Autònoma de Barcelona

Prof. Carles Ferrer Ramis, Professor of the Microelectronics and Electronic Systems Department at the Universitat Autònoma de Barcelona,

Certify

that the dissertation *Dynamic Partial Reconfiguration in FPGAs for the Design and Evaluation of Critical Systems* presented by Luis Andrés Cardona Cardona to obtain the degree of PhD in Electronic and Telecommunication Engineering has been done under his supervision at the Universitat Autònoma de Barcelona

Prof. Carles Ferrer Ramis ....

Barcelona, April 2016

a mis Padres,
David, Lida y Caro

Abstract

Field Programmable Gate Array (FPGA) devices persist as fundamental components in the design and evaluation of electronic systems. In the case of Xilinx SRAM-based FPGAs they support Dynamic Partial Reconfiguration (DPR) by means of the Internal Configuration Access Port (ICAP). This hardwired element allows the configuration memory to be accessed at run time and therefore change specific parts of the system while the rest continues to operate with no affection in its computations.

This thesis is focused on using DPR as a mechanism to: i) improve hardware flexibility, ii) implement precise fault emulation of ASIC designs mapped in FPGAs and iii) improve tolerance to accumulated or multiple faults in the configuration memory of Triple Modular Redundancy (TMR) circuits. This work addresses the three challenges considering as one of the most relevant figures of merit the speed at which the tasks can be performed. Therefore one of the main objectives we consider is the speed-up of DPR related tasks.

In the first place we developed a new high speed ICAP controller, named AC_ICAP, completely implemented in hardware. In addition to similar solutions to accelerate the management of partial bitstreams and frames, AC_ICAP also supports DPR of LUTs without requiring pre-computed partial bitstreams. This last characteristic was possible by performing reverse engineering on the bitstream. This allows DPR of single LUTs in Virtex-5 devices to be performed in less than 5 μs which implies a speed-up of more than 380x compared to the Xilinx XPS_HWICAP controller.

In the second place, the fine grain DPR obtained with the utilization of the AC_ICAP is used in the emulation of faults to test ASIC circuits implemented in FPGAs. It is achieved by designing a CAD flow that includes a custom technology mapping of the ASIC net-list to LUT-level FPGA net-list, the creation of fault dictionaries and the extraction of test patterns. A hardware platform takes the fault list and leverages the partial reconfiguration capabilities of the FPGA for fault injection followed by application of test patterns for fault analysis purposes.

Finally, we use DPR to improve the fault tolerance of TMR circuits implemented in SRAM-based FPGAs. In these devices the accumulation of faults in the configuration memory can cause the TMR replicas to fail. Therefore fast detection and correction of faults without stopping the system is a required constraint when these FPGAs in the implementation of critical systems. This is carried out by inserting flag error detector based on XNOR and carry-chain components, isolating and constraining the three domains to known areas and extracting partial bitstreams for each domain. The latter are used to correct faults when the flags are activated.

Resumen

Los dispositivos FPGA persisten como componentes fundamentales para el diseño y evaluación de sistemas electrónicos. En el caso de las FPGAs basadas en memoria SRAM de Xilinx, éstas soportan Reconfigurabilidad Parcial Dinámica (DPR) por medio del *Internal Configuration Access Port* (ICAP). Este componente físico permite acceder a la memoria de configuración mientras el sistema está operando y por lo tanto la DPR puede ser usada para modificar partes específicas del sistema mientras que el resto sigue funcionando sin ser afectado.

Esta tesis está enfocada en usar DPR como un mecanismo para: i) mejorar la flexibilidad del hardware, ii) emular fallos de forma precisa en diseños ASIC mapeados en FPGAs y iii) mejorar la tolerancia a fallos acumulados o múltiples en la memoria de configuración de circuitos con Triple Redundancia Modular (TMR). Este trabajo aborda los tres aspectos considerando como figura de mérito fundamental la velocidad de ejecución de las tareas. Por lo tanto, uno de los principales objetivos es acelerar las tareas relacionadas con DPR.

En primer lugar un controlador hardware para el ICAP fue diseñado. Éste es denominado como AC_ICAP y además de soportar lectura y escritura de frames, manejo de bitstreams parciales desde memoria flash y memoria interna de la FPGA, también permite DPR de alta velocidad a nivel de LUTs sin necesidad de *bitstreams* parciales previamente generados. Esta última característica es posible gracias a ingeniería inversa en el *bitstream* con la cual se puede ejecutar DPR de LUTs individuales en menos de 5 μs. Ésto representa una mejora en tiempo de reconfiguración de más de 380 veces comparado con el controlador XPS_HWICAP de Xilinx.

En segundo lugar, la DPR a nivel de LUTs obtenida gracias al AC_ICAP es utilizada en la emulación de fallos para evaluar circuitos ASIC mapeados en FPGAs. Esto se consigue gracias a una herramienta CAD que incluye un traductor de la descripción ASIC a una descripción basada en LUTs para ser implementada en FPGAs, generación de diccionarios de fallos y extracción de patones de prueba. Una plataforma hardware usa el listado de fallos y

aprovecha la DPR de la FPGA para la precisa inyección de fallos seguida de la aplicación de los patrones de test para analizar los efectos de los fallos en el circuito.

Finalmente la DPR es utilizada para mejorar la tolerancia a fallos de circuitos TMR implementados en FPGAs basados en memoria SRAM. En estos dispositivos la acumulación de fallos en la memoria de configuración puede generar fallos en las réplicas TMR. Por lo tanto la rápida detección y corrección de fallos sin detener el sistema es un requerimiento que se debe cumplir cuando se usan estas FPGAs en la implementación de sistemas críticos. Para ello se insertan detectores de errores de tipo XNOR que convergen en componentes carry-chain de la FPGA y además cada dominio es aislado en áreas diferentes del dispositivo para los cuales se extraen bitstreams parciales. Éstos son utilizados para corregir los fallos cuando los detectores son activados.

Acknowledgements

After this long trip through the PhD I want firstly to express my sincere appreciation to Carles Ferrer for giving me the opportunity to work in this thesis and for his long term support and encouragement to finish it. I am also grateful to all people I meet at the Microelectronics and Electronic Systems Department at UAB with whom I shared research activities, teaching, discussions and coffees.

I also want to declare my gratitude to Anees Ullah formerly at Politecnico di Torino and currently at City University of Science and Information Technology, Peshawar, Pakistan; Luca Sterpone and Ernesto Sanchez from Politecnico di Torino for their invaluable help and availability to cooperate and continue working on these research topics. Such cooperation is reflected in the chapters 4 and 5. The research visit to Torino was possible thanks to the Short Term Scientific Mission Reference: ECOST-STSMIC1204011014050179 as part of the COST Action: IC1204 (TRUDEVICE).

During the period of the PhD studies I had the opportunity to work on the EU-project: Trusted Computing for European Embedded Systems at the CNM-IMB. Thanks to all members of ICAS team for their help and kindness.

Finally I want to thank to my Family for their unconditional support and encouragement from the distance. Last but not least thanks to Carolina for being always there.

To all of them my most sincere acknowledgments.

Luis Andrés Cardona, Barcelona, April 2016

Contents

1 Introduction **1**

1.1 Thesis organization 4

2 DPR-FPGAs Critical Apps **5**

2.1 DPR in SRAM-based FPGAs 5

2.1.1 ICAP controllers 8

2.2 DPR for ASIC Fault Emulation on FPGAs 12

2.2.1 Circuit instrumentation based approaches 14

2.2.2 Reconfiguration based approaches 15

2.3 DPR for CDE improvement in TMR circuits 16

2.4 Contribution and Objectives of the Thesis 19

3 AC_ICAP Controller **21**

3.1 ICAP for Dynamic Partial Reconfiguration 21

3.2 Dynamic partial reconfiguration for LUTs 23
3.3 AC_ICAP Implementation 27
3.3.1 ReadFrames 29
3.3.2 WriteFrames 31
3.3.3 DPR of LUTs with LUT2Frames module 32
3.3.4 Load Partial bitstreams 35
3.4 AC_ICAP adapted to on-chip processor 36
3.4.1 PLB IP 37
3.4.2 FSL co-processor 38
3.5 Using the AC_ICAP in newer device families 38
3.6 Experimental Results 42
3.7 Summary 47

4 ASIC Fault emulator through DPR 49
4.1 ASIC fault emulation using an FPGA 49
4.2 Proposed Methodology 53
4.2.1 The proposed CAD Flow 54
4.2.2 Fault Injection and Emulation Platform 62
4.3 Experimental Results 70
4.4 Summary 75

5 DPR for TMR cross-domain errors 77
5.1 Dinamically reconfigurable X-TMR 77
5.2 CAD flow 78
5.2.1 Hierarchy formation 78

5.2.2 Re-mapping, error detection and flag convergence . . . 79
5.2.3 Partial bitstreams generation 82
5.2.4 Faulty bitstreams for emulation 84
5.3 Developed Testbed 85
5.4 Emulation of CDEs 88
5.5 Experimental Results 90
5.6 Summary 92

6 Conclusions **93**

List of Figures

2.1 FPGA architecture 6
2.2 XPS_HWICAP [1] 10

3.1 ICAP hardwired primitive 22
3.2 X, Y coordinates and Frame Addresses for XC5VLX110T . . 24
3.3 Frame bits for LUT configuration 26
3.4 AC_ICAP detail 27
3.5 BRAM memory map 29
3.6 Read frame FSM 30
3.7 LUT2Frames module 33
3.8 FSM for DPR of LUTs 34
3.9 Chipscope detail of LUT-DPR with AC_ICAP 34
3.10 Architecture with PLB_AC_ICAP IP 37
3.11 Architecture including FSL_AC_ICAP co-processor 39

3.12 Chipscope detail of LUT-DPR with AC_ICAP in Kintex7 . . 41

4.1 Slice architecture for Virtex-5 FPGA 50

4.2 Example of mapping a circuit to an FPGA LUT 52

4.3 Constrained Technology Mapping and Fault Dictionary Creation Flow 55

4.4 Mapping without duplication 56

4.5 Mapping with duplication 57

4.6 Partial Reconfiguration Compatible Fault Formats 60

4.7 Position of LUT frames in CLB frames 61

4.8 Binary format for storing of STIL file 63

4.9 Fault emulation scheme 64

4.10 Fault injection platform 66

5.1 TMR fracturable LUT 79

5.2 Proposed CAD flow 80

5.3 Partial bitstream alternative generation 84

5.4 Dynamically reconfigurable architecture 86

5.5 Hardware-based ICAP controller 87

List of Tables

3.1 Coding of tasks 28
3.2 Timing behavior of AC_ICAP 43
3.3 Resource utilization of AC_ICAP 45
3.4 Resource utilization of ICAP controllers 46
3.5 Resource utilization of full systems with different ICAP controllers 46

4.1 Fault Emulation Times 71
4.2 Fault Statistics 74
4.3 Area and Delay Comparison 75

5.1 CDEs for bubble sort 91
5.2 Timing bubble sort 92

List of Acronyms

DPR Dynamic partial reconfiguration 1

FPGA Field Programmable Gate Array 1

ICAP Internal Configuration Access Port 1

IP Intellectual Property 1

LUT Look-Up Table 1

MBU Multi-bit upset 1

SEU Single Event Upset 1

SRAM Static random access memory 1

Introduction | 1

Field Programmable Gate Array (Field Programmable Gate Array (FPGA)) devices persist as fundamental components in the design and evaluation of electronic systems. They are continuously reported as final implementation platforms rather than only prototype elements. FPGAs have moved according to VLSI scaling technology pace making it possible to develop these devices in state-of-the-art fabrication processes. The inherent reconfigurable characteristics that FPGAs offer are among one of the most important advantages in the actual hardware implementation and redesign of systems. In the case of Xilinx Static random access memory (SRAM)-based FPGAs, they support Dynamic Partial Reconfiguration (Dynamic partial reconfiguration (DPR)) by means of the Internal Configuration Access Port (Internal Configuration Access Port (ICAP)). This hardwired element allows the configuration memory to be accessed at run time. Dynamic Partial Reconfiguration can then be used to change specific parts of the system while the rest continues to operate with no affection in its computations [2]. Therefore, the architecture of the system can be modified at level of basic logic components, such as Look-Up-Tables (Look-Up Table (LUT)s), or bigger blocks, such as Intellectual Property (IP) cores, and in this way more flexible systems can be designed. It is a great advantage, especially in critical and aerospace applications, where the access to the system to re-design the hardware is not a trivial task. But on the other hand, the main problem FPGAs present when used for critical applications is their high sensitivity to external factors such as Single Event Upsets (Single Event Upset (SEU)) and Multi Bit Upsets (Multi-bit upset (MBU)) in the configuration memory. It is a limiting factor that must be considered to avoid misbehavior of

the implemented hardware.

Xilinx tools provide general controllers to drive the ICAP but they perform most of the processing as software routines in the processor. It implies flexibility but avoids reaching the maximum supported reconfiguration throughput. Diverse alternatives to these controllers have been reported [3], [4] to improve the reconfiguration speed. Most of them have been oriented to manage partial bitstreams generated at design time and also to manipulate frames, that are the minimum addressable configuration memory. But an efficient mechanism that allows LUTs to be modified at run-time is also required as LUTs are basic components to implement any logic function in FPGAs. Therefore, the ICAP controller should offer a way to perform DPR of LUTs (fine-grain DPR) at maximum supported speed but presenting a simple interface, doing the complexity of the architectural device transparent to the user. This limitation is addressed in this thesis as will be explained in chapter 3.

The aggressive technology scaling of CMOS circuits and the physical limitations of photo-lithography based chip fabrication process require faster and more affordable approaches for testing manufacturing defects. Software-based fault simulation approaches are traditionally used to evaluate the test patterns goodness against the targeted faults. However, this is a high timing consuming process that grows up with the size of the device under test. As an alternative, a hardware fault emulation approach for Application Specific Integrated Circuits (ASICs) on FPGAs can considerably reduce the time required for performing the fault simulation. In addition, this approach more closely mimics the hardware behavior of the device when the circuit is faulty, providing in this way, more realistic figures. With this in mind, the fine grain DPR can be applied in the emulation of faults to test circuits implemented in FPGAs [5].

Therefore, DPR can be used in different ways: as a mechanism to improve hardware flexibility [6], precise fault emulation [7] and fast fault correction [8]. For all this aspects one of the most relevant figure of merit is the speed at which the tasks can be performed. When DPR is used to improve the flexibility in hardware, a critical aspect is the speed at which the switching of the different hardware tasks is performed. The inherent speed up that the hardware-based tasks offer can be affected by the time required to reconfig-

ure the FPGA resources where such tasks will be implemented. This is why the time that the partial reconfiguration engine spent should be optimized to avoid considerable overheads in timing. Fast DPR switching is required to minimize timing overhead in the hardware-based functions.

For fault tolerant systems implemented in SRAM-based FPGAs [9] one critical aspects that affect the overall dependability of the system is related with how fast a fault can be detected and corrected [10]. Moreover, the better performance behavior between commercial and radiation hardened parts also increase the demands for efficient solutions to use the COTS devices in critical systems [11], [12].

Triple Modular Redundancy is a widely used fault tolerance methodology [13] to protect circuits against radiation-induced Single Event Upsets. But this technique, that is very efficient in masking errors, should be complemented when using SRAM-based FPGAs [14], [15]. In these devices the accumulation of SEUs or MBUs in the configuration memory can cause the TMR replicas to fail, requiring a periodic refresh of the configuration bitstream [16]. This imply a system downtime due to scrubbing and the probability of simultaneous failures of two TMR domains increase with growing device densities. Therefore fast detection [17] and correction of faults without stopping the system is a required constraint to use SRAM-based FPGAs in the implementation of real-time critical systems[18].

In this thesis we address the mentioned challenges by developing a new high speed ICAP controller, named AC_ICAP, completely implemented in hardware. In addition to similar solutions to accelerate the management of partial bitstreams and frames, AC_ICAP also supports DPR of LUTs without requiring pre-computed partial bitstreams.

Such fine grain DPR obtained with the utilization of the AC_ICAP is used in the emulation of faults to test ASIC circuits implemented in FPGAs.

Finally, we use DPR to improve the fault tolerance of TMR circuits implemented in SRAM-based FPGAs focused on CDE.

1.1 Thesis organization

The thesis is organized as follows. In chapter 2 we present the context, state-of-the-art and objectives of this thesis in the area of dynamic partial reconfiguration in FPGAs. Once the context has been set up, the chapter 3 describes the new ICAP controller as the core component of improved run-time reconfigurable systems. It is followed by chapter 4, which employs fine DPR as a mechanism to precisely emulate faults in ASIC design mapped into FPGAs. Chapter 5 presents an alternative partial bitstreams extraction for CDE improvement and fast recovery of TMR circuits. Finally, Chapter 6 summarizes the main contributions of this thesis and presents future research tasks based on these results.

Dynamic Partial Reconfiguration in FPGAs for Critical Applications 2

This chapter contextualizes the research by providing background information on Dynamic Partial Reconfiguration. The chapter is divided into three sections. In Section 2.1 the main characteristics of DPR and its associated controller are summarized. Section 2.2 analyzes the utilization of DPR to emulate faults on ASICs followed by Section 2.3 that addresses the correction of faults by means of DPR. Finally, Section 2.4 presents the objectives of this thesis.

2.1 DPR in SRAM-based FPGAs

Field Programmable Gate Array (FPGA) devices persist as fundamental components in the design and evaluation of electronic systems. They are continuously reported as final implementation platforms rather than only prototype elements [19]. FPGAs have moved according to VLSI scaling technology pace making it possible to develop these devices in state-of-the-art fabrication processes. For instance, 7-series family of Xilinx SRAM-based FPGAs are built on 28 nm, high-k metal gate process technology [20], Xilinx Virtex UltraScale+ uses 16 nm FinFET+ and Altera Stratix 10 devices are produced using Intel-14 nm Tri-Gate (FinFET) process technology [21].

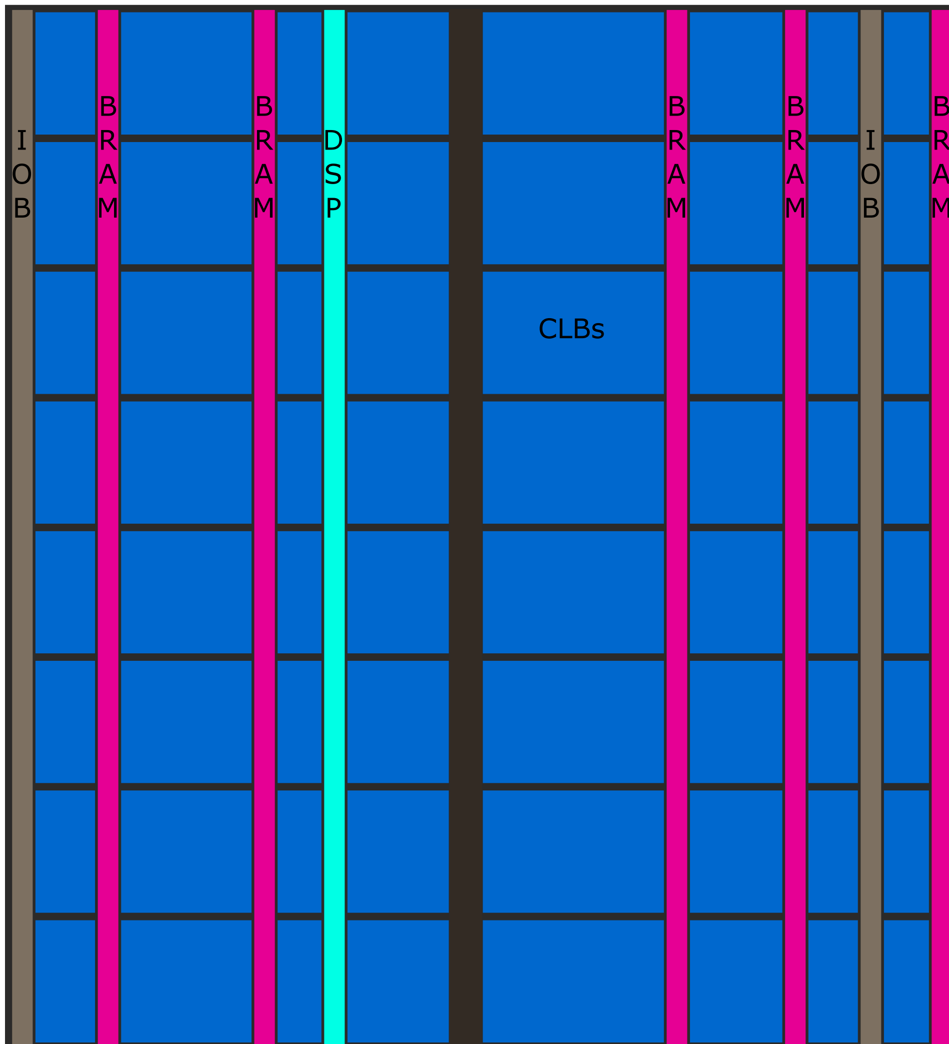


Figure 2.1 | FPGA architecture

This is one of the reasons that favor the increasing presence of such devices as programmable alternatives to ASICs.
In addition, technical improvements in the design and fabrication of FPGAs have produced more robust and flexible components embedding larger RAM

memory blocks (BRAMs), DSP blocks, processors and dedicated hardwired components as depicted in Fig. 2.1. The inherent reconfigurable characteristics that FPGAs offer are among one of the most important advantages in the actual hardware implementation and redesign of systems. In the case of Xilinx SRAM-based FPGAs, they support Dynamic Partial Reconfiguration (DPR) by means of the Internal Configuration Access Port (ICAP). This hardwired element, depicted in Fig. 3.1, allows the configuration memory to be accessed at run time. It is possible to modify specific parts of the system while the rest continue operating and is not affected by the specific run-time modification. Dynamic Partial Reconfiguration can be used at different granularity levels. Considering the architecture of the device, it can be employed to modify basic logic components, such as Look-Up-Tables (LUTs), or bigger blocks, such as IP cores. Therefore, DPR is employed in a wide range of applications involving the design of self-adaptive systems and the evaluation of critical systems that need to be exhaustively tested before final production.

Xilinx tools such as PlanAhead or command line "$bitgen\ \ -r$" take the difference between two implementations to produce partial bitstreams that allow for modifying the specific parts that have been defined to change at run-time. The partial bitstreams are then copied in external or internal memory of the FPGA and from there are sent to the ICAP when a new hardware task is required by the system. In addition to this type of run-time reconfiguration that is especially suitable for coarse grain modules, there are alternatives to dynamically modify basic elements, such as LUTs, using certain software functions executed in an On-chip processor.
With this in mind, the hardwired ICAP primitive and its associated controller become fundamental and inseparable modules in the design of dynamic run-time reconfigurable systems. The ICAP controller is responsible for performing all the commands to access and modify the configuration memory. Therefore it is desirable that such a controller meets at least two basic requirements: high reconfiguration throughput and flexibility.
Xilinx tools provide general controllers to drive the ICAP but they perform most of the processing as software routines in the processor. It implies flexibility but avoids to reaching the maximum supported reconfiguration throughput. Diverse alternatives to these controllers have been reported to improve the reconfiguration speed. Most of them have been oriented to

manage partial bitstreams generated at design time and also to manipulate frames that are the minimum addressable configuration memory.

Going deeper into the granularity of the device any dynamic modification on the LUTs of an implemented design should also be available to increase the flexibility of the system. Therefore, an efficient mechanism that allows LUTs to be modified at run-time is also required as LUTs are basic components to implement any logic function in Xilinx FPGAs. The ICAP controller should offer a way to perform DPR in LUTs at maximum supported speed but presenting a simple interface making the complexity of the architectural device transparent to the user.

2.1.1 ICAP controllers

In this section we outline some of the most relevant implementations of ICAP controllers used in FPGA Dynamic Partial Reconfiguration.

Partial reconfiguration has been widely used in diverse applications [22–24] that exploit the possibility of adapting hardware modules at run-time. A common requirement when using this technique is that the switching of hardware modules should be performed with minimal time overhead.

Taking as reference the Virtex-5 XC5VLX110T FPGA we analyze the configuration time when the configuration bitstream resides on a 16-bit FLASH memory 28F256P30.

According to equation 2.1, for a given data bus width (16 bit in our case), the configuration time of an FPGA depends on the size of the configuration file and the configuration frequency.

$$Configuration\ time = \frac{bitstream\ size}{configuration\ frequency\ *\ data\ bus\ width} \tag{2.1}$$

$configuration\ frequency$ (also known as $ConfigRate$) depends on diverse

parameters as shown in equations 2.2 and 2.3[25]

$$\frac{1}{ConfigRate \ * \ (1 \ + \ FMCCKTOL_{MAX})} \geq TBPICCO{+}TACC{+}TBPIDCC \tag{2.2}$$

$$ConfigRate \leq \frac{1}{(TBPICCO + TACC + TBPIDCC) * (1 \ + \ FMCCKTOL_{MAX})} \tag{2.3}$$

Where the timing parameters, for Virtex 5 according to [26], represents:

$FMCCKTOL_{MAX}$: FPGA Master CCLK frequency tolerance = 50% (0.5)

$TBPICCO$: ADDR[25:0] outputs valid after CCLK rising edge = 10 ns

$TACC$: BPI flash address to output valid (access) time. For 28F256P30 it is t_{AVQV} = 95 ns.

$TBPIDCC$: FPGA data setup time = 3 ns

Applying the above values on Equation 2.3, we got $ConfigRate \leq 6.17MHz$. This value can be set in the ISE tools in the Generate Programming File properties. By default the $ConfigRate$ is set to 2 MHz, but for the mentioned case it can be increased to 6 MHz. In doing this the configuration time of the XC5VLX110T FPGA, when the bitstream resides on 28F256P30 Flash memory, using the Equation 2.1 is:

$Configuration\ time = \frac{31118848bits}{6MHz \ * \ 16bits} = \mathbf{324}\ ms$

This value shows the time required to modify the hardware by loading the full configuration bitstream. This time can be unaffordable for many applications, therefore a more efficient way to perform fast modifications on the hardware is by using dynamic partial reconfiguration where the size of the bitstream is smaller and consequently the reconfiguration time is lower.

The most frequent approach to implement systems with DPR capabilities is by using the ICAP controllers available in Xilinx tools. XPS_HWICAP [1], depicted in Fig. 2.2, AXI_HWICAP and OPB_HWICAP are IP cores designed to be connected to the PLB [27], AXI and low speed OPB buses re-

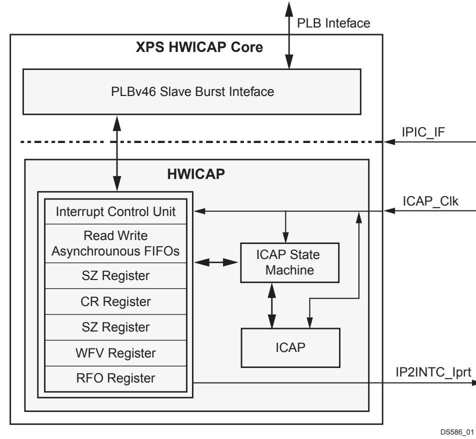


Figure 2.2 | XPS_HWICAP [1]

spectively. They are used as part of embedded processor systems (PicoBlaze or MicroBlaze) and the support for partial reconfiguration is given through a collection of software functions provided with the processor API. The functions allow to process partial bitstreams located in memory, access configuration frames (***XHwIcap_DeviceReadFrame***, ***XHwIcap_DeviceWriteFrame***), and modify LUTs (***XHwIcap_SetClbBits***, ***XHwIcap_GetClbBits***). An example of the utilization of the functions to modify specific LUTs is detailed in [28] and authors in [5] use the functions to modify frames to emulate faults on the configuration memory.

The Xilinx functions perform most of the operations as software routines in the processor. The commands to manage the ICAP and process the partial bitstream header executed in the processor, along with the bus latency affect the speed of the partial reconfiguration process. Therefore, diverse alternative controllers have been developed to overcome these limitations. Authors in [29] explore different ICAP controllers analyzing the reconfiguration speed and propose three variations to speed up the processing of partial bitstreams but all of them require the presence of a processor. It is also the case of [30] and [31]. In the latter case the controller is integrated in the processor data path using the FSL link to minimize the bus latency. In contrast, authors in [32] and [3] present controllers for Virtex-5 devices able to load partial bitstreams from BRAM and Flash memory totally im-

plemented in hardware and independent of the processor. In a similar way [33] and [24] report implementations of processor-independent ICAP controllers for Virtex-4 FPGAs. Authors in [34] exploit DPR for the design of fault tolerant systems. Such approaches show improvements in reconfiguration speed that can reach the maximum supported throughput when using BRAMs. Further, some works such as those presented in [24, 35], achieve throughput speed higher than the specified one in the technical documents by overclocking the ICAP.
All these works have been oriented to efficiently access the partial bitstreams and perform the tasks of hardware switching, but some further operations that a complete controller should support are not considered. A robust controller should be able to read back and write configuration frames and also give the possibility to modify LUTs besides only controlling partial bitstreams.

These last features are of paramount importance in the implementation of critical systems where the ICAP controller is a fundamental part of the design [36]. With this in mind, diverse approaches such as that reported in [37–39, 4] use improved ICAP controllers as fundamental parts of fault tolerant systems in SRAM-based FPGAs. In such systems, the ICAP is used for the detection and correction of faults in the configuration memory. To do that, it is not enough controlling pre-computed partial bitstreams, they implement reading and writing of frames as the fault detection is performed at this level. For instance, once a frame is read, its CRC can be obtained to check if errors are present in its constituent bits. In the case of erroneous values, the frame can be corrected and write back to the configuration memory with the right values. Therefore, these reported works include frame handling for both writing and reading of configuration frames.

To the best of our knowledge, the only work reported on performing runtime reconfiguration at LUT level implemented as part of the ICAP controller is presented in [40], but it is only valid for Xilinx Virtex-II devices where LUTs have four inputs and the architecture of the device is considerably different from newer Xilinx families. The frames cover the full height of the device and it is not detailed how the LUT configuration values are located on frames. In addition, this family is currently considered obsolete. With this in mind we identified that there is a lack of ICAP controllers that

perform fast DRP at different granularity levels and also support newer devices. In consequence this is one of the aspects addressed in this thesis.

2.2 DPR for ASIC Fault Emulation on FPGAs

Photo-lithography driven VLSI chip fabrication technology is reaching its physical limits due to aggressive technology scaling towards nano-metric dimensions. The dimensions of individual transistors and the wiring interconnection density considered proportional to the wavelength of etching beams in lithography are prone to introduce manufacturing defects during the last phases of the circuit production life. For the fast moving Application Specific Integrated Circuits (ASIC) industry, increasing the yield within stringent time-to-market requirements is essential; thus, fault simulation is a mandatory step to guarantee the quality of the final test set that leads to yield of any VLSI chip fabrication process. Roughly speaking, a fault simulation process consists of running a set of input patterns in the considered circuit, while injecting every one of the considered circuit faults (stored in a fault list) in order to determine whether the faults are detected or not. Clearly, for large designs the time required for fault simulation is extremely long. One reason of lengthy simulation times is the adoption of software based mechanisms. Although they provide flexibility, these techniques are inherently slow due to the high number of computations required to trace and analyze every signal related to every one of the circuit faults; this task is particularly critical on large circuits. An alternative solution is to use hardware based fault emulation on state-of-the-art reconfigurable Field Programmable Gate Arrays (FPGAs) which can greatly reduce the timing requirements. Furthermore, compared to the software based simulation approaches, FPGA-based approaches can more realistically emulate the actual design behavior in a faulty condition performing in this way a more accurate application behavioral analysis. Fault effects on the running applications on microprocessors is an important validation task that requires long times and careful consideration which are essential for safety-critical applications involved in automotive, space and avionic systems.

Since the birth of FPGA, designers are constantly exploiting their reconfigurable nature for prototyping digital VLSI circuits before final chip fabrica-

tion. This task usually intends to verify if the design goals are achievable. The usage of these devices in fault emulation and testing is relatively under explored due to lack of design methodologies and tools. For fault emulation of ASICs on FPGA-based systems, it is necessary to recall the differences on the technology nodes used by the two paradigms. On the one hand, ASIC net-lists are usually composed of single basic logic gates, as well as of standardized modules devoted to perform well defined operations, for example, integer addition, floating point division and multiplication. On the other hand, FPGA net-lists are comprised of Look Up Tables (LUTs). LUTs can implement any combinational Boolean function on “k” inputs, where “k” is the maximum number of inputs to the LUT.

Considering the synthesis processes for ASIC and FPGA, both of the processes try to optimize similar parameters, e.g., area, delay, power, performance; however, every one of them is basically oriented to optimize during the actual synthesis different basic elements: the ASIC synthesis is oriented to optimize logic gates, while FPGA synthesis optimizes LUTs. This difference on the synthesis processes makes it difficult to perform a one-to-one mapping of the ASIC net-list on an FPGA; in addition, it may lead to structural differences in the considered circuits, modifying the circuit structures that will be emulated. Thus, performing an accurate fault emulation of an ASIC device by using FPGA-based techniques requires initial handling with this particular issue. A possible solution may rely on partitioning the ASIC net-list to a set of circuit chunks suitable to be placed in single FPGA LUTs; however, this division process is hard to obtain by using commercial FPGA tools. Therefore, a customized technology mapping from the ASIC net-list to FPGA net-list is mandatory for maintaining the same circuit structure, that guarantees the actual emulation of all the ASIC faults.

This approach was first investigated by the authors in [41] exploiting a commercial fault emulation platform for serial fault emulation. This technique provides at the end a logic mapping for every gate of the original ASIC in the final FPGA design. As a result, the fault emulation for each ASIC fault is performed using a corresponding LUT configuration string. However, the technique was applied using a commercial fault emulation platform requiring a vendor specific tool-chain. A preliminary work presented in [42] developed a framework to apply constrained mapping to commercially available

SRAM-based FPGAs without resorting to expensive vendor-specific emulation platforms.

Next we outline some of the most relevant techniques used for emulating ASIC faults on reconfigurable hardware based on FPGAs. These methods can be mainly classified according to the adopted fault injection methodology:

2.2.1 Circuit instrumentation based approaches

Circuit Instrumentation approaches add extra hardware resources in the design for fault injection purposes. A dynamic fault injection approach is presented in [43], [44] which instruments the model of circuit with a global shift register controlling the activation of signals for fault injection. The activation signals selects between duplicated LUT representing faulty and fault-free functions. The faulty behavior is achieved by reconfiguration of LUT site designated to be faulty therefore no recompilation is needed. Independent faults are identified and instrumented in such a way to inject multiple faults at the same time reducing the reconfiguration time. To avoid reconfiguration and for fast injection of faults the authors in [45], [46] included all the desired faults in the model and design and synthesized them necessitating the activation of control signals in run-time using a scan-chain based fault injector circuit. Instrumentation at the gate-level for fault injection and scan-chain based activation mechanisms is used by the authors in [47] enabling them to avoid time consuming re-compilation steps for generating fault bit-streams. However, the size of the fault list is directly proportional to the hardware complexity of the injector circuitry, introducing a bottleneck for large VLSI circuits. To attain more speed up for the whole process of fault injection, test patterns applications and faults classification, a system is proposed by authors in [48], [49] that exploits the idea of minimum communication between host and emulation platform. To summarize, techniques based on circuit instrumentation are intrusive in nature and in some cases have a large area overhead.

2.2.2 Reconfiguration based approaches

Reconfiguration based fault injectors modify the configuration bit-stream of the design to emulate faults. The authors in [41] utilize a commercial fault emulation platform that constrains the technology mapping of logic cones to LUTs and generates a corresponding bit-stream for each fault in the logic cone in form of FPGA reconfiguration list. The authors show that for designs with more than 100,000 gates hardware emulation is two times faster compared to software fault simulation. JBits based tool-flow [50] is used for directly changing the configuration bits of a CLB in order to inject faults by authors in [51], [52], [53]. However, JBits is no longer supported for state-of-the-art commercial FPGAs. A direct bitstream manipulation based injection is used by the authors in [54], [55] to inject faults in LUTs without constraining technology mapping. They reduce the size of the fault list by considering only the active inputs of LUTs. However, this technique does not guarantee that every ASIC fault will be covered. The fault injection for a wide variety of fault models was presented in [56] where the authors develop faulty bit-stream by changing the HDL models. This requires time-consuming re-compilation for each injection of the fault model. Another interesting work exploiting partial reconfiguration for fault injection is presented in [57]. The authors present a methodology for the correlation of stuck-at fault model with that of Single Event Upset (SEU) fault model. Improving and generalizing upon the methodology presented by authors in [41] this work develops a general framework for fault emulation on commercial FPGA platforms without resorting to expensive platforms and vendor tools. It presents a general method that can be applied to any FPGA device without resorting to very costly commercial fault emulation platforms and vendor tools.

The chapter 4 of this thesis extends the approach presented in [42] by proposing a novel hardware emulation platform based on dynamic partial reconfiguration. The proposed methodology consists of a novel CAD flow able to map a synthesized ASIC circuit into a FPGA and accurately emulate all the circuit permanent faults; the fault simulation is performed by generating several versions of fault dictionaries that fully exploit the features of dynamic partial reconfiguration available through various interfaces and vendor software APIs. Thanks to the proposed flow flexibility, minor

changes are required to include the definition of new fault models as well as to define different ASIC logic gates.

2.3 DPR for CDE improvement in TMR circuits

As fabrication technology advances Integrated Circuits become more complex, compact and consequently more susceptible to external factors. These are sensitive to radiation and charged particles with effects that have been observed even at ground level [58].

SRAM-based FPGAs are very valuable for remote missions and long-time missions because of the possibility of being reprogrammed by the user as many times as necessary. It is a great advantage especially in critical and aerospace applications where the access to the system to re-design the hardware is not a trivial task. But on the other hand, the main problem SRAM-based FPGAs present when used for critical applications is their high sensitivity to Single Event Effect (SEE) such as Single Event Upset (SEU) in the configuration memory. It is a limiting factor that must be considered to avoid misbehavior of the implemented hardware.

Triple Modular Redundancy is a widely used fault tolerance methodology to protect circuits against radiation-induced Single Event Upsets. But this technique that is very efficient in masking errors should be complemented when using SRAM-based FPGAs [14]. In these devices the accumulation of SEUs in the configuration memory can cause the TMR replicas to fail, therefore some type of fast correction should be added. One common approach is to use continuous scrubbing. This is the periodic refreshing of the configuration memory with the golden bitstream located in memory, normally on external flash. This implies a system downtime due to scrubbing and the probability of simultaneous failures of two TMR domains increases with growing device densities. The main disadvantage of this approach is its blind characteristic which means that the system is rewriting the configuration memory even if no errors are present. In addition, this task must require the utilization of the ICAP port which implies that it cannot be employed for user hardware-based tasks switching: If the ICAP is performing the continuous scrubbing no user Dynamic Partial Reconfiguration can

be performed. On the contrary, if DPR is important for the application, while it is being performed no scrubbing can be done which implies longer times for possible error correction (MTTR increase). With this in mind, a more efficient approach is to use DPR for correction only when a fault is detected. Here the importance of efficiently and quickly detecting faults to avoid error propagation and perform corrections as soon as any anomaly is discovered.

Therefore fast detection [17] and correction of faults without stopping the system is a required constraint to use SRAM-based FPGAs in the implementation of real-time critical systems and such characteristics are investigated in this thesis.

A considerable amount of literature has been published on fault tolerance in SRAM-based FPGAs [59]. Studies such as the presented in [60], [61], [14], [62], [63] report approaches that use redundancy combined with certain type of fault detection mainly at coarse grain level. It means that the circuit to be protected is replicated and constrained to a defined reconfigurable area. Authors in [60], [14] and [61] utilize DPR as correction mechanism once a fault is detected. For detection they compare the outputs of the circuits using DWC or TMR applied at the complete circuit but any of them employ X-TMR. When the circuit to be protected is processed through the X-TMR tool a flat design is produced and it is not supported by the standard CAD tools to generate the partial bitstreams.

Other approaches focus on monitoring the configuration memory at frame level to detect and correct faults. For Virtex-5 devices [64] Xilinx offers a solution that read back frames, compute its CRC value along with the syndrome value that indicates the position of the faulty bit that is flipped and the frame written back. Such approach allows for detecting and correcting one SEU within one frame while MBUs or accumulated SEUs is limited to two and these can only be detected but the position of the faulty bits are not known. [65] is an improved version of this controller for newer families of Xilinx devices. It also uses the hardwired ICAP component to access the frames content and SEU correction is possible for two adjacent SEUs inside one frame. The work presented in [66] uses a coarse grain TMR design where every replica is configured with the same frames information and a scrubber can detect and correct MBUs on different frames. The voter and

scrubber should be protected to avoid single points of failure.

The approach presented in [67], [68] improves the recovery time and use flags and carry-chains but they do not employ the X-TMR. They improve the recovery time by avoiding full device scrubbing. Instead the starting point of the scrubbing is predicted by an on-chip algorithm.

[7] showed how 2-bits MBUs may corrupt TMR circuits 2.6 orders of magnitude more than SEUs. In [69] and [70]experimental results of radiation also analyzed the susceptibility of TMR protected circuits in Virtex devices to MBUs.

Therefore MBUs must be considered as an important design constraint when aerospace systems are developed in SRAM-based FPGAs. As described in [70] the routing network of CLBs is vulnerable to domain crossing errors. One error in two or more domains of the TMR circuit can cause the voter to select the wrong value. In the concrete case of the X-TMR generated circuits the domains are mixed and are not easily identified nor isolated. As a result MBUs or accumulated faults can affect the components of different domains and produce erroneous outputs.

We focus on the utilization of X-TMR as it has been reported to be highly reliable when using FPGAs based on SRAM. We take as reference design the X-TMR version of the circuit to be evaluated. This approach follows a similar flow as the presented in [71] but it is improved by using more efficiently the LUTs that implement the majority voters to manage flags activation.

Therefore our approach leverages the ease with which X-TMR tool can generate protected circuits but we improve it by resource-efficient domain fault detection and alternative partial bitstreams generation. This last feature allows run-time reconfiguration of individual domains to be used for correction. Our approach improves CDE resilience thanks to the individual flags per domain and the domain-based placement.

2.4 Contribution and Objectives of the Thesis

The aim of this thesis is to develop efficient DPR techniques at different granularity levels to improve the switching time of hardware tasks, precisely evaluate circuits implemented in SRAM-based FPGAs and improve cross-domain resilience of TMR circuits mapped in these devices. To achieve this goal the main tasks developed in this thesis are:

- Design a fast and flexible dynamic partial reconfiguration controller fully implemented in hardware with support for fine-grain DPR. This means that single LUTs could be modified at run time for which detailed analysis of the bitstream structure and FPGA architecture are required.
- Design a fault emulator platform and CAD flow to map ASIC circuits in FPGAs and leverage DPR of LUTs to precisely emulate faults.
- Proposes a TMR architecture that exploits the fracturable nature of Look Up Tables for simultaneously mapping of majority-voting and error detection at the granularity of TMR domains. An associated CAD flow is developed for partial reconfiguration of TMR domains incorporating changes to the technology mapping, placement and bitstream generation phases.

AC_ICAP: A Flexible High Speed ICAP Controller 3

In this chapter, we describe the AC_ICAP: A high speed ICAP controller, completely implemented in hardware. In addition to similar solutions to accelerate the management of partial bitstreams and frames, AC_ICAP also supports run-time reconfiguration of LUTs without requiring pre-computed partial bitstreams. This last characteristic was possible by performing reverse engineering on the bitstream. We give a general description of ICAP in Section 3.1. In Section 3.2 we present the main considerations regarding fine-grain partial reconfiguration. In Section 3.3 we detail the new AC_ICAP controller. In Section 3.4 the extension of the controller to be accessible from On-chip processors is presented. In Section 3.5 we describe the considerations to follow in porting the controller to a newer family of devices. In Section 5.5 we present the experimental results of the reconfiguration time and area followed by Section 3.7, that concludes the chapter.

3.1 ICAP for Dynamic Partial Reconfiguration

The Internal Configuration Access Port (ICAP) is the core component of any dynamic partial reconfigurable system implemented in Xilinx SRAM-based FPGAs. These devices support Dynamic Partial Reconfiguration (DPR) by means of the ICAP hardwired element, depicted in Fig. 3.1. Thanks to this physical component it is possible to access to the configuration memory

at run time to modify specific parts of the system while the rest continue working without being affected by the specific run-time modification. DPR can be used at fine grain level, such as Look-Up-Tables (LUTs), and to modify bigger blocks, such as IP cores.

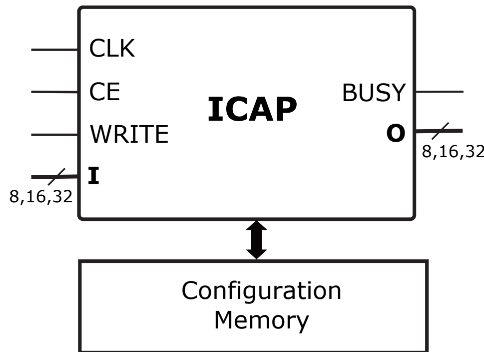


Figure 3.1 | ICAP hardwired primitive

Therefore, the hardwired ICAP primitive and its associated controller become fundamental and inseparable modules in the design of dynamic run-time reconfigurable systems. The ICAP controller is responsible for performing all the commands to access and modify the configuration memory. Therefore it is desirable for such controller to meet at least two requirements: high reconfiguration throughput and flexibility.

In addition to partial bitstream efficient management and frames read/write support, the ICAP controller should offer a way to perform DPR in LUTs at maximum supported speed but presenting a simple interface, doing the complexity of the architectural device transparent to the user.
In the following sections, we detail the novel run-time reconfiguration controller fully implemented in hardware and supporting partial reconfiguration of LUTs in Xilinx FPGAs.

3.2 Dynamic partial reconfiguration for LUTs

In this section we describe the general architecture of Xilinx FPGAs and the relevant concepts of partial reconfiguration taking as reference the Virtex-5 XC5VLX110T device. FPGAs are organized as an array of Configurable Logic Blocks (CLBs) connected to a switch matrix. Fig. 3.2 shows the disposition of the XC5VLX110T FPGA where it can be observed that it is horizontally divided into two halves. In the top (0) and bottom (1) halves, we find a fixed number of rows that depend on the size of the specific device. The Virtex-5 LX110T FPGA is divided into eight horizontal clock rows (HCLK): four in each half. Each HCLK includes a determined number of CLBs, BRAMs, DSPs, I/Os. CLBs are distributed in 160 rows by 54 columns covering the whole device. Each CLB consist of two Slices and every Slice contains 4 LUTs, 4 flip flops, multiplexers and carry logic. As a result, this FPGA has 17280 Slices, 69120 LUTs and 69120 registers.
One CLB column is defined as a group of 20 x 1 CLBs that spans the HCLK height. It means that in each CLB column inside the HCLK rows, there are 40 Slices and 160 LUTs.

The configuration memory is organized in frames. One frame is the smallest size of configuration memory able to be addressed. Therefore, any action on configuration memory should be carried out taking frames as reference. One frame consists of 41 words of 32 bits each one (1312 bits). Virtex-5 LX110T requires 23712 configuration frames to configure the whole chip. In consequence, the configuration file (bitstream) is composed of 972464 32-bit words (3.7 MB). It includes 272 words of control information in the header and the rest corresponds to configuration frames.
Every time we want to configure the whole device, the bitstream of 3.7 MB containing the description of the circuit to implement, should be loaded into configuration memory.

Dynamic partial reconfiguration allows to modify specific parts of the system, in consequence, the complete bitstream is not required but a smaller, partial bitstream, with the information of the specific region to modify is used. Partial bitstreams are generated at design time using the difference-based approach. PlanAhead [72] or bitgen command line [73] are used to generate them. The command: $bitgen -r\ config1.bit\ config2.ncd\ partial2.bit$

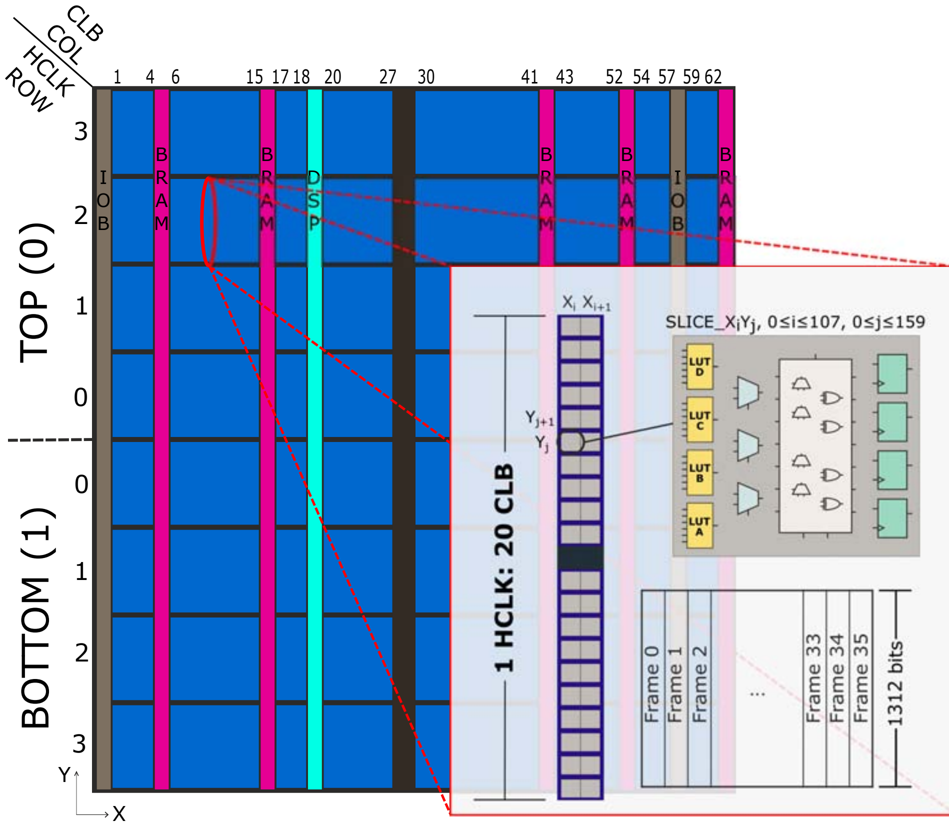


Figure 3.2 | X, Y coordinates and Frame Addresses for XC5VLX110T

takes as inputs the two different files for each configuration ($config1$ and $config2$) and the result is the partial bitstream $partial2.bit$, with the difference between them. The minimum size of partial bitstreams corresponds to one configuration frame increased with one extra dummy frame and control information.

To configure a column of CLBs, 36 frames are required. Inside the 36 frames, we have the information of every individual element present in the 20 CLBs. We focus on LUTs as these are the basic elements that implements all the combinatorial logic in FPGAs.

The LUTs or logic-function generators are six inputs elements that require 64 bits to define the function to perform. The logic behavior of the LUT depends on the values (INIT value) configured in these 64 bits. To handle any individual LUT it is necessary to define its location and its INIT value. The location uses three parameters: (X, Y, Bel). X and Y are the coordinates of the Slice and Bel is an index to select the individual LUT inside the Slice. The range of X and Y depends on the size of the FPGA (108 x 160 in the considered device). The Bel index, ranges from 0 to 3, to select one of the 4 LUTs (LUT-A, LUT-B, LUT-C, LUT-D) inside the Slice with coordinates (X,Y). Once the specific LUT has been identified, its INIT value can be modified through the 64 configuration bits. This LUT parameter can be modified at run-time thanks to certain software routines provided by Xilinx API. The function ***XHwIcap_GetClbBits*** is used to read back the INIT value of the LUT and store it in memory. ***XHwIcap_SetClbBits*** allows to write to the LUT the INIT value available in any location accessible from the processor. Both functions require the same type of parameters: X, Y, Bel coordinates and memory location where to read or write the INIT value.

Very limited information about these functions and the operations they perform is available. In addition, the time required to read and write the configuration value of a LUT using these functions is in the order of 2 ms while the time for reading and writing frames, using ***XHwIcap_DeviceReadFrame*** and ***XHwIcap_DeviceWriteFrame*** functions, is in the order of 30 μs. These numbers, experimentally obtained using a MicroBlaze-based system operating at 100 MHz, suggested us opportunities to improve the reconfiguration time for LUTs. Therefore, we performed experiments to deduce the relationship between LUT parameters and configuration frames. By combining the ***XHwIcap_SetClbBits*** function to write to a specific LUT with the ***XHwIcap_DeviceReadFrame*** to analyze the programmed values on frames, we found that four frames are used to reconfigure a single LUT.
As it is shown in Fig. 3.3, the 64 bits of the INIT value spans four consecutive frames with each frame containing 16 INIT bits. The 40 Slices inside every CLB column can be seen as 2 columns of 20 Slices. One Slice column contains the 20 Slices with even values on X coordinate while the other 20 Slices presents odd values. The frames 26 to 29 enclose the LUT configuration values for the 20 Slices with odd-X coordinates while the frames 32 to 35 have the corresponding information for the 20 Slices when X coordinate

is even. In a similar way, Slice-Y coordinate determines what specific word insde each frame to use. For any CLB column, Y takes 20 consecutive values. Depending on this value, a specific word in the frame corresponds to a single LUT. Two consecutive frame words have the partial information for the 4 LUTs of a Slice. 16-bits of INIT LUT-A and 16-bits of INIT LUT-B configuration values are in one 32-bit word. Similarly, LUT-C and LUT-D INIT values are located in the following word.

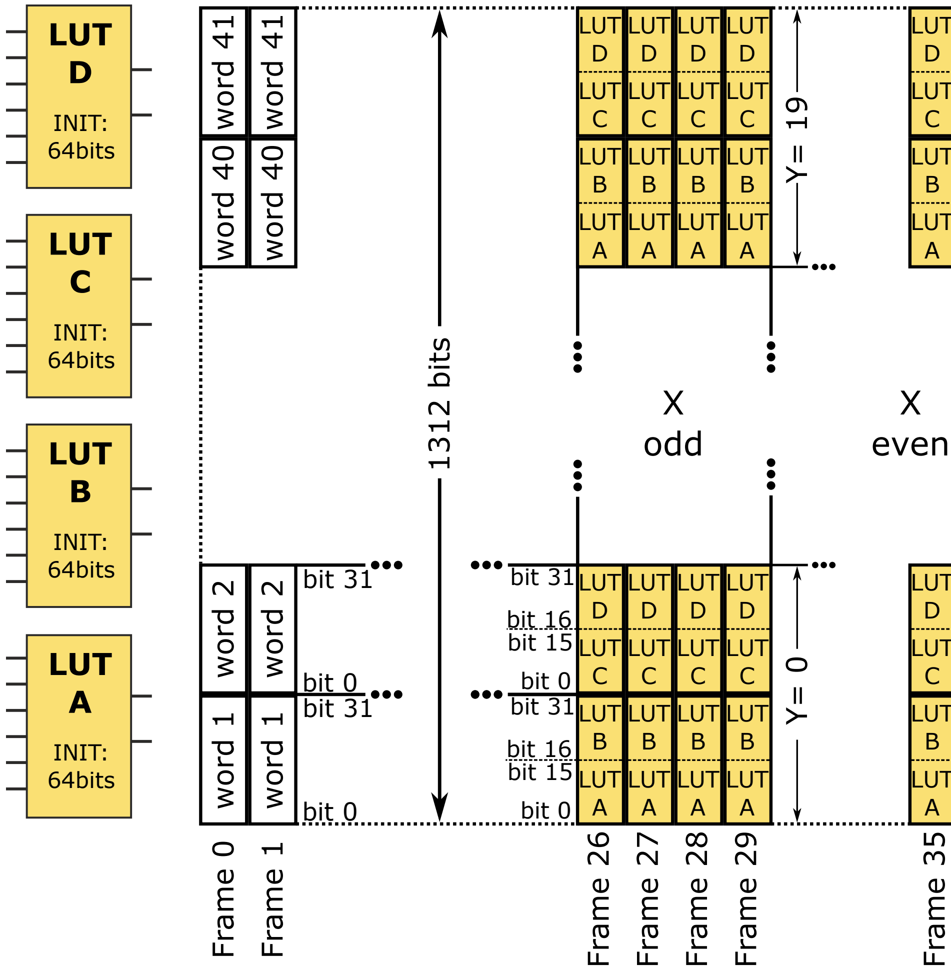


Figure 3.3 | Frame bits for LUT configuration

3.3 AC_ICAP Implementation

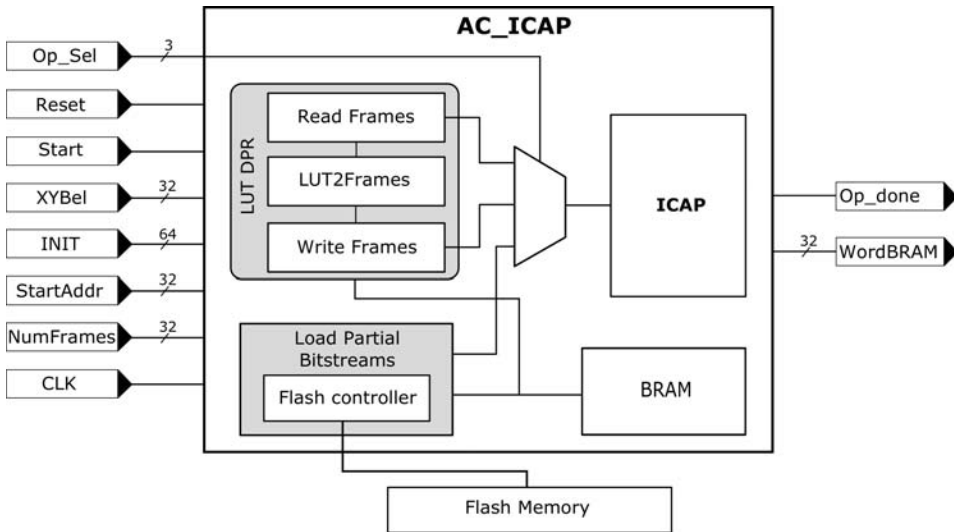


Figure 3.4 | AC_ICAP detail

The AC_ICAP controller, detailed in Fig. 3.4, offers similar functionality as the XPS_HWICAP available in Xilinx tools but AC_ICAP is fully implemented in hardware, instead of doing most of the tasks as software routines in the processor. It includes support for read frames, write frames, modify LUTs, and load partial bitstreams from Flash and BRAM memory. Compared to similar approaches, that also implement read and write of frames in hardware [4], our controller is improved by the run-time reconfiguration of LUTs without the need of pre-computed partial bitstreams. The controller and its internal modules use Finite State Machines (FSMs) to operate on diverse configuration levels according to the values of the input Op_sel specified in Table 3.1. The AC_ICAP was developed using a board equipped with the Virtex-5 LX110T FPGA and the implementation flow was performed in Xilinx tools, version 14.7.

As explained in section 3.2, DPR of LUTs require to modify specific parts of frames. Therefore, the two modules for read and write frames are indispensable in the implementation of LUT run-time reconfiguration. We designed

Table 3.1 | Coding of tasks

Operation	**Op_sel input**
Read BRAM	000
Read Frame	001
Write Frame	010
Modify LUT	011
Recover LUT	100
Load partial bitstream from Flash	101
Copy partial bitstream from Flash to BRAM	110
Load partial bitstream from BRAM	111

the AC_ICAP controller with 7-36Kbit BRAM elements (31.5 KB) configured as dual port memory. It represents less than 5% of the 148 BRAM memory blocks available in the device. This space of memory serves to store frames read and it is also used as the source of the frames to send to the ICAP. The initial 2800 Bytes are reserved to perform LUT modification and frame tasks. The remaining 28.7 KB can be used for both frame or partial bitstreams storage, as it is depicted in Fig. 3.5. When partial bitstreams fit into the available BRAM, its corresponding partial reconfiguration task can reach the maximum specified throughput because of the direct connection between the on-chip BRAM and the ICAP through a link of 32 bits. By using a clock of 100 MHz, one 32-bit word is available with every clock cycle, which corresponds to the maximum ICAP supported throughput (3.2 Gbps). We adhere to the constraints specified in the technical documents regarding the maximum operation frequency of the ICAP: 100 MHz [1]. But it should be taken into account that Hansen et al. [35] reported the correct operation of the ICAP when it is overclocked to achieve better reconfiguration throughput speed.
The constituent modules of the AC_ICAP controller are detailed next.

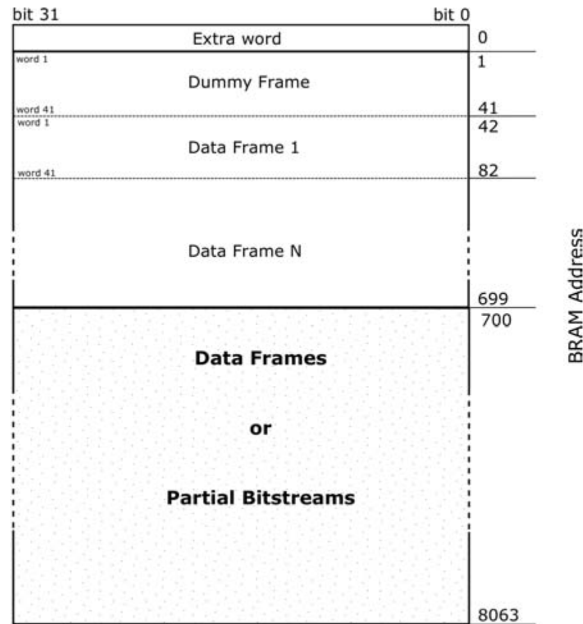


Figure 3.5 | BRAM memory map

3.3.1 ReadFrames

ReadFrames module uses two parameters to define the location ($FAddr$) and number of frames (Nf) to read. Nf takes the value 1 for a single frame read or any other value for multiple frame read. It is limited by the available BRAM memory on the controller. It should be noted that for LUT modification tasks, one BRAM block is enough but we included six extra blocks to store frames or small partial bitstreams. We store all the read frames on BRAM and there we can access to perform any operation on them. Alternatively, an external module able to process and store the read frames, could acquire more frames than the limited by the size of the BRAM.
In the case of multiple frames ($Nf > 1$), $FAddr$ is the address of the first frame where the reading process starts. From there, the routine will read Nf consecutive frames. The steps involved in ReadFrames routine are depicted in Fig. 3.6. When op_sel = “001" and the start signal is asserted,

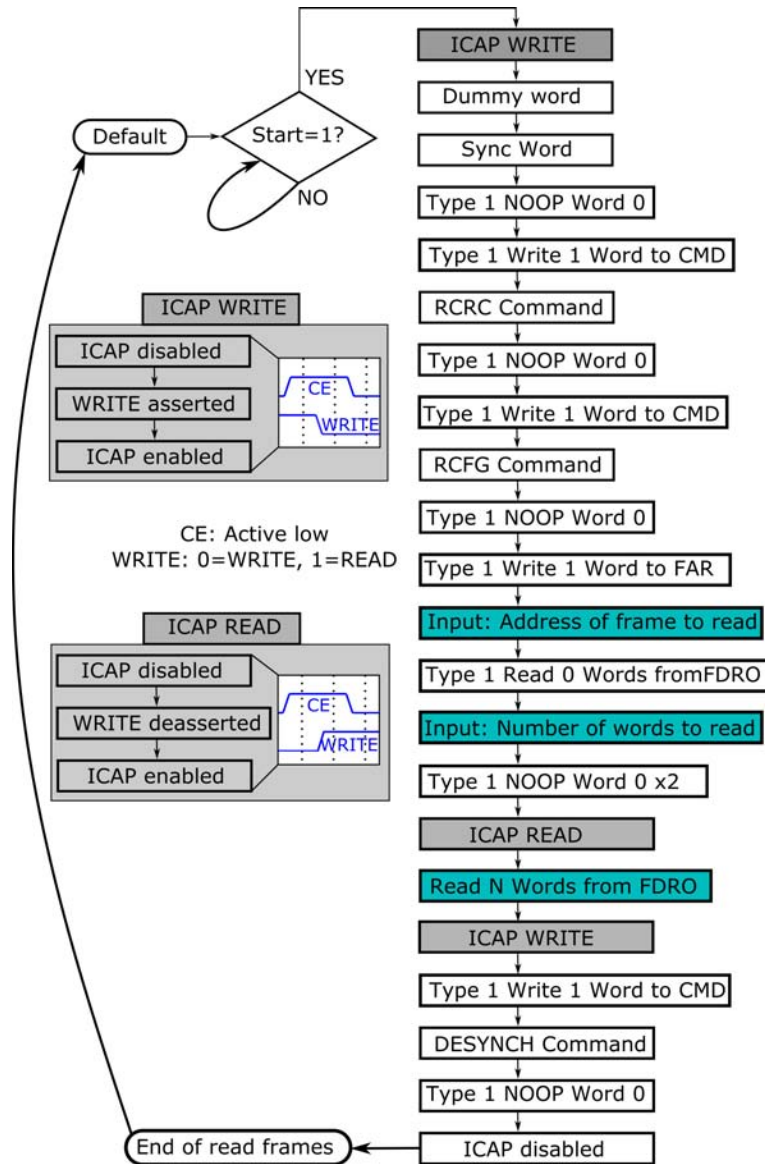


Figure 3.6 | Read frame FSM

the ICAP is configured to read the specified frames. This is done by writing

to certain registers of the ICAP as it is detailed in [74]. It is important to point out the correct assertion of the CE and WRITE inputs to define read or write operations on the ICAP. In both cases, WRITE should be modified before CE to avoid causing an abort sequence. It is detailed in the two boxes ICAP WRITE and ICAP READ in Fig. 3.6.
The inputs $FAddr$ and Nf are used in the two steps of the the flow identified with the word Input. These two values are adapted to the format of the corresponding registers. $FAddr$ should have the format of the Frame Address Register, it is, one 32-bit word with the fields: Block type, Top, HCLK row, column and frame inside the column. Nf is used to calculate the number of words to read (N) and generate a Type 2 word to send to the ICAP. $FAddr$ and Nf can be specified by the user through the inputs $StartAddr$ and $NumFrames$ respectively. Or they can be generated by the $LUT2Frames$ module, as it will be explained in subsection 3.3.3.

We must consider that any reading of frames includes one extra dummy frame generated at the beginning of the process and also one extra word. With this in mind, the number of words to read for the Virtex-5 device can be calculated as:

$$N = 41 * (Nf + 1) + 1 \tag{3.1}$$

Equation 3.1 is valid for any Virtex-5 FPGA as in these devices all the configuration frames have the same size. It is 41 32-bit words. The dummy frame is represented by the addition of 1 to Nf. The last addition represents the initial word.
The state **READ N Words from FDRO** performs the actual read of the N 32-bit words that compose the frames. With every word read from the FDRO register of the ICAP, the BRAM address is increased to store the frames on this memory. Fig. 3.5, shows the location of the frames and the extra word.

3.3.2 WriteFrames

This module was designed following the same approach as the presented in ReadFrames. The main differences are in the configuration commands required to prepare the ICAP to write to the configuration memory. WriteFrames module is activated when the Op_sel input, defined in Table 3.1, is

“010" and the Start signal is asserted. To reach the maximum throughput speed the preferred source for the frames to write is BRAM. If the frames are located in the BRAM of the AC_ICAP, one 32-bit word is available with every clock cycle.
As this module is normally used in combination with ReadFrames, the frames to be written have already been read and stored on BRAM. Then, WriteFrames uses the same memory space, detailed in Fig. 3.5, where ReadFrames placed the read back frames.
In the same way that ReadFrames needs to consider one dummy frame, in every write frames routine, the dummy frame should be sent to the ICAP at the last part of the process. Therefore, data frames starts at BRAM_address=42 and finishes at address $41 * (Nf + 1)$. Immediately after data frames are sent, the dummy frame should follow. To do that, the starting address changes to 1 and finishes when 41 words (1 frame) are sent. The extra word at address 0 is not used in writing processes.

We generate the Op_done output to signalize the end of a write process. It is necessary to guarantee that the ICAP tasks finish properly. After all words are sent, it is necessary to send the DESYNC command and disable the ICAP. Op_done goes high when the ICAP receives and process the DESYNC command. It is observed when the output port O changes from 0xDF to 0x9F. This process has a delay of 6 clock cycles independently of the value on the input CE.

3.3.3 DPR of LUTs with LUT2Frames module

The LUT2Frames module allows the dynamic partial reconfiguration of LUTs by doing the translation of the LUT parameters into Frames representation. As it was described in Section 3.2, the LUTs are characterized by the coordinates (X, Y, Bel) and the INIT value. The LUT2Frames module, depicted in Fig. 3.7, carries out two main tasks: (1). Translate the X, Y, Bel coordinates into FAR format and (2). Transform the INIT (64-bits) LUT function into 4 words of 16-bit each one.

The X, Y, Bel inputs, merged into one 32-bit word, and the INIT value are used by the LUT2Frames module when the Start input is set. Based on the coordinates values, one 32-bit word with the format of the Frame Address

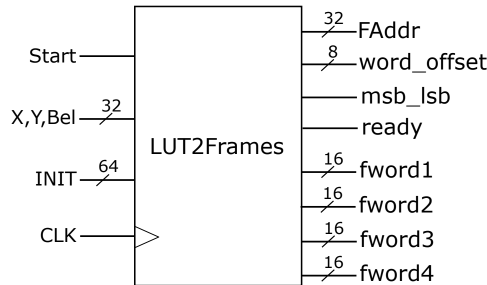


Figure 3.7 | LUT2Frames module

Register (*FAddr*) is generated to define the frame where the reading and writing start. In addition X, Y and Bel values determine the word_offset that is the concrete word of each frame (the first one of the 2-41 words) that need to be manipulated. From the 32-bit word only 16-bits correspond to a specific LUT. Therefore the signal msb_lsb indicates what part of the 32-bit word should be modified: 0 for the LSB part of the word (LUT-A or LUT-C) and 1 for the 16 MSB (LUT-B or LUT-D).
In parallel with the previous processing, LUT2Frames generates four 16-bit words (fword1... fword4) that corresponds to the INIT value transformed and adapted to the four frames.
All the complexity of the Frames location and addressing is transparent to the user. The LUT2Frames module implements all the translations and computes adequate addresses and memory management to allow to the user a simple operation when require to modify any LUT throughout the device.

When a LUT modification is required, the steps controlled by an FSM, as depicted in Fig. 3.8, are executed. The process is triggered by the Start signal, then the LUT2Frames module is activated. With the values generated by this module, 4 frames starting at *FAddr* are read and stored in BRAM (read frames). Word_offset and msb_lsb signalize the specific words that should be modified. These 4 words are backed up (backup words), modified with the four words that LUT2Frames produced and copied back to BRAM. At this point the BRAM contains the frames with the new words, and the write frames module performs the writing of the 4 frames corresponding to the LUT.

The Recover LUT routine uses the four backed up values obtained at backup

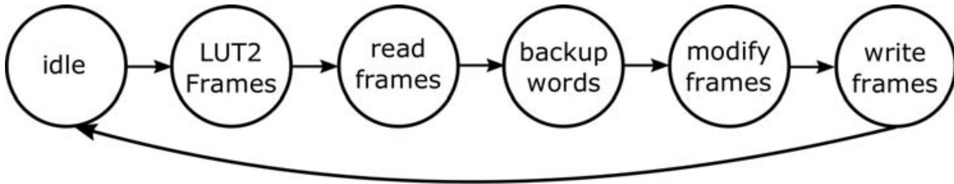


Figure 3.8 | FSM for DPR of LUTs

words stage to recover the LUT to its previous configuration value. Considering the Fig. 3.8, it only performs the last two steps of a LUT modification routine. It modifies the 4 frames on BRAM and then these are sent through Write frames module to recover the LUT to its previous INIT value. This routine is useful in applications that need to recover the previous function of a LUT before modifying another. By following this approach we avoid to read again four frames as these are already on BRAM.

The correct operation of the controller was verified using ChipScope Pro Debugger [75]. Fig. 3.9 shows the details for a LUT modification process. We specified the X, Y, Bel and INIT values of the LUT to modify. The steps shown in Fig. 3.8 can be identified in Fig. 3.9. The LUT2Frames module requires only two clock cycles and the information it generates is used to address the four frames to read and to modify the four specific words in these frames.

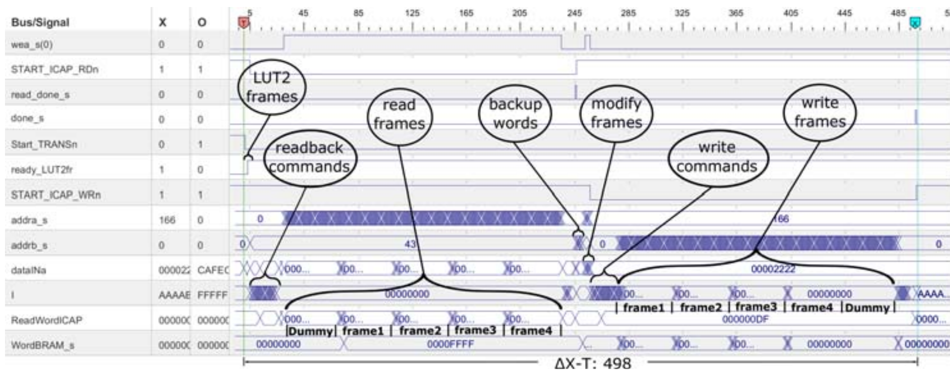


Figure 3.9 | Chipscope detail of LUT-DPR with AC_ICAP

3.3.4 Load Partial bitstreams

The Load partial bitstreams module performs three main tasks: (1). load partial bitstreams from Flash, (2). Copy partial bitstreams from flash to BRAM, and (3). Load partial bitstreams from BRAM. To do that, this module includes a memory access controller to read the partial bitstreams from flash memory. Therefore, the data read from flash can be directly send to the ICAP I port or it can be copied into internal BRAM. When partial bitstreams are on BRAM, the maximum configuration speed on the ICAP can be reached. If partial bitstreams are on external memory, the reconfiguration time depends on the latency of the access to the memory. In this case we use the Intel StrataFlash memory 28F256P30 which requires 26 clock cycles at 100 MHz to get a 32-bit word.

The size of the partial bitstreams that can be placed on BRAM is limited by the available BRAM memory on the controller. From the 7-36Kbit BRAM present in the AC_ICAP, we reserved 2800 Bytes to perform LUT modification and frame tasks. Therefore the maximum size of partial bitstreams that can be placed is of 28.7 KB. It can be increased as the FPGA includes more BRAMs (148 in the LX110T device) but it depends on the application constraints.

The partial bitstreams are generated following the standard Xilinx flow, it is using PlanAhead or bitgen tools. These configuration files include header information regarding the type of device, size of the configuration data, date and time of the generation of the bitstream, etc. We adapt the partial bitstreams to remove unnecessary information from the header and keep only the last header-field that corresponds to the size (in Bytes) of the partial bitstream not including the header. Therefore our controller firstly reads the word that contains the size of the partial bitstream and uses this information to calculate the number of words (16-bit word for flash and 32-bit word for BRAM) to read from memory. With this approach the only required parameter is the initial address where partial bitsrtreams are located. The controller automatically calculates the end address and performs the reading process. Depending on the operation selected by the input Op_sel, the data is sent to the ICAP or to BRAM. In a similar way, when Op_sel is set to "‘111"’, this module configures the ICAP control

signals and BRAM address to allow high throughput partial reconfiguration.

3.4 AC_ICAP adapted to on-chip processor

To make the controller able to be attached to processor-based designs, it was adapted to the Peripheral Local Bus and Fast Simplex Link interfaces used by MicroBlaze systems. To this end, the AC_ICAP was considered as a black box with the I/O ports depicted in Fig. 3.4 and these were adapted to the respective buses. This approach offers increased flexibility as the controller can be easily commanded from the processor. We created a collection of functions adapted to each interface to perform the tasks presented in Table 3.1. Such functions, depicted in Code 1, use specific routines from the Xilinx API to access to the PLB and FSL interfaces.
Code 1: Functions to drive the AC_ICAP IPs

```
ReadBRAM(StartAddr);
ReadFrame(StartAddr, NumFrames);
WriteFrame(StartAddr, NumFrames);
ModifyLUT(XYBel, INIT);
RecoverLUT(XYBel);
LoadPBitsFlash(StartAddr);
CopyFlash2BRAM(StartAddr);
LoadPBitsBRAM(StartAddr);
```

The *StartAddr* parameter refers to a unique input that should be adapted according to the *op_sel* value. In case of read and write frames, it corresponds to the address of the initial frame ($FAddr$). For the other functions it is the memory address where data are stored. $NumFrames$ is the number of frames to read or write and $X, Y, Bel, INIT$ are the parameters that control single LUTs. These are the only values required to command the AC_ICAP controller as this performs internally all the operations such as transforming the X, Y, Bel and $INIT$ into frame format, compute end address after reading the size of a partial bitstream, etc.

3.4.1 PLB IP

The PLB bus is used to connect peripherals to the MicroBlaze processor. The original AC_ICAP, designed in VHDL, is instantiated in a PLB wrapper to generate the custom PLB_AC_ICAP IP. The inputs and outputs of the controller are connected to signals of the PLB bus and then the processor can access to them using register addresses. In consequence, the PLB_AC_ICAP can be attached to any MicroBlaze-based system such as the depicted in Fig. 3.10. This architecture includes the Flash memory

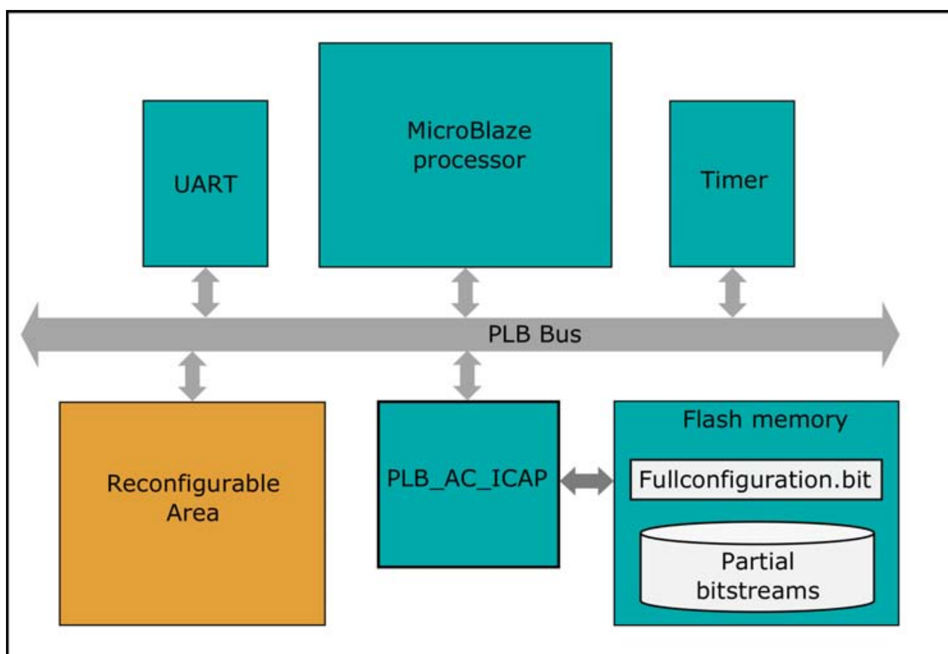


Figure 3.10 | Architecture with PLB_AC_ICAP IP

where the full and partial bitstreams that modify the reconfigurable areas are located. The direct connection to the flash memory is also performed in the IP design by defining the AC_ICAP connections to the Flash as external ports. Once included in the hardware design in EDK, the software running in the processor is able to control the PLB_AC_ICAP peripheral by using the functions listed in Code 1. In consequence, a partial reconfiguration

related task uses any of the functions specified in Code 1 and monitor the output op_done until it goes high as confirmation that the task has been completed.

3.4.2 FSL co-processor

Fast Simplex Link is an interface of the MicroBlaze processor that allows to include dedicated hardware routines with high execution priority and therefore, implies low latency in the communication with the processor. In this approach we adopted a solution similar to the presented in [31], in order to obtain minimal degradation in the performance of the controller due to the bus latency. Thus, the VHDL-based AC_ICAP was adapted to the FSL interface to be easily connected as a co-processor and consequently exploit all the flexibility of the processor but taking advantage of the hardware acceleration in the ICAP related tasks. Fig. 3.11 presents a system using the FSL_AC_ICAP co-processor.

The FSL_AC_ICAP co-processor is accessed in a similar way to the considered in the PLB_AC_ICAP IP. It is, by means of a collection of functions such as the presented in Code 1. The main differences are in the type of routines that these functions require to drive the FLS. In this case, we incorporate the blocking routines `putfsl` and `getfsl` available with Xilinx API as we consider that the reconfiguration tasks are of high priority.

3.5 Using the AC_ICAP in newer device families

To validate the controller in 7 series devices we use the KC705 board equipped with a Kintex7 XC7325T FPGA [76]. This FPGA contains 50,950 Slices, inside every Slice 4 6-input LUTs and 8 FF are present. The 445 BRAMs correspond to 2002 KB and the bitstream size is of 10.9 MB. To adapt the AC_ICAP, designed for Virtex-5, to 7 series devices, certain changes are required. The main differences are summarized below:

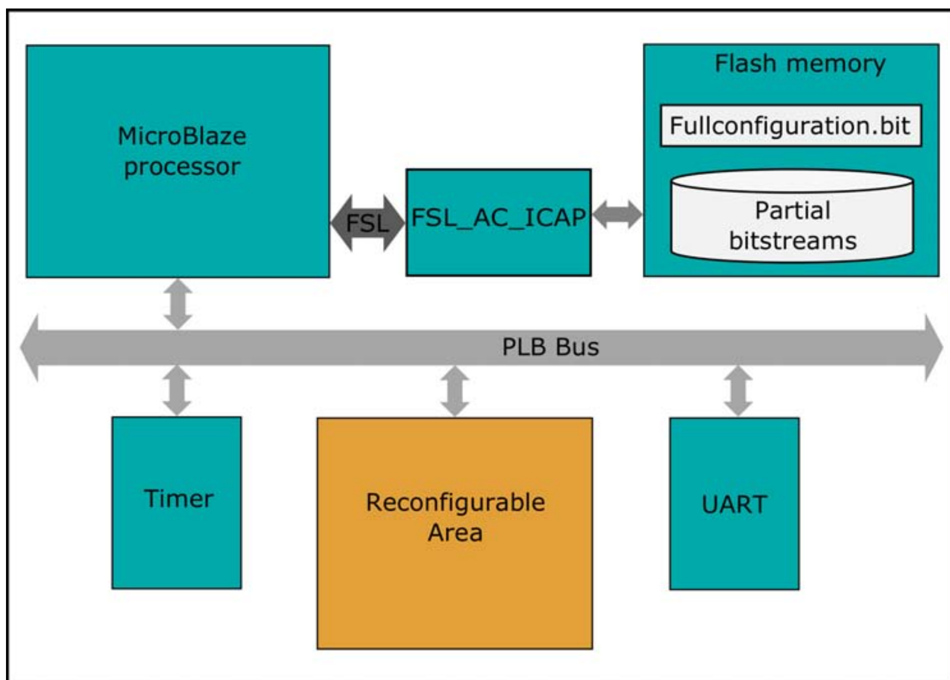


Figure 3.11 | Architecture including FSL_AC_ICAP co-processor

- The number of words per frame in 7 series family is 101 instead of 41 (Virtex-5). It is because the CLB columns in 7 series FPGAs are 50 high by 1 wide which implies that 100 Slices are present in the CLB columns. Similarly, the number of HCLK rows is different, for this specific device it is 7 (3 top and 4 bottom).

- The address of the frame where to start reading or writing is defined by the FAR register. For 7 series this register uses 26 bits of the 32 available while in Virtex5 FAR uses 24. It is due to the increased size of the FPGA.

- Differently from Virtex-5, for 7 series no extra word is required at the beginning of a read frames task. Therefore, the number of words ($Nwords7$) to read/write from these devices can be computed according to Equation 3.2 that is valid for any 7 series FPGA as in these devices all the configuration frames have the same size. The dummy

frame is represented by the addition of 1 to the number of frames (Nf).

$$N_{words7} = 101 * (Nf + 1) \tag{3.2}$$

- The word_offset that indicates what specific word on a frame should be modified in a LUT DPR process now have a range of 0 to 100. It varies between 0 and 40 for Virtex-5. In a similar way, the skip columns (columns that contain resources different from CLBs: BRAMs, DSPs I/O) and the the major column numbering require to be updated. The first column in Kintex7 has a major address of 2, while it is 1 for Virtex-5.

- In 7-family the primitive ICAPE2 does not have the BUSY output. Instead we should consider 3 clock cycles after CE assertion to get the valid data.

- The write frames module required also some changes. In Virtex-5 it is possible to bypass the CRC calculation by setting a configuration register (COR0-bit28) and loading the value 0xDEFC to the CRC register every time that the FAR is modified. In 7-family such register is not present, by default the new control register (COR1-bits15-16) is set to allow the system a continuous operation after CRC is computed and therefore such steps were removed.

- The FLASH memory available in this board is of the same type as the present in the Virtex-5 but as the size is different, the FLASH controller was modified to include two extra address lines.

The number of frames required to configure a CLB column remain the same (36), also as the specific frames that contain the information for LUTs. We used 22 BRAM blocks to occupy a similar percentage (5%) as in Virtex-5.

Once the presented changes were performed in the AC_ICAP it was implemented in the Kintex7 FPGA and tested with all the operations it supports. In Fig. 3.12 we present again the details for DPR of one LUT as it involves diverse tasks available in the controller.
This new AC_ICAP was adapted to the AXI interface as this is used for all

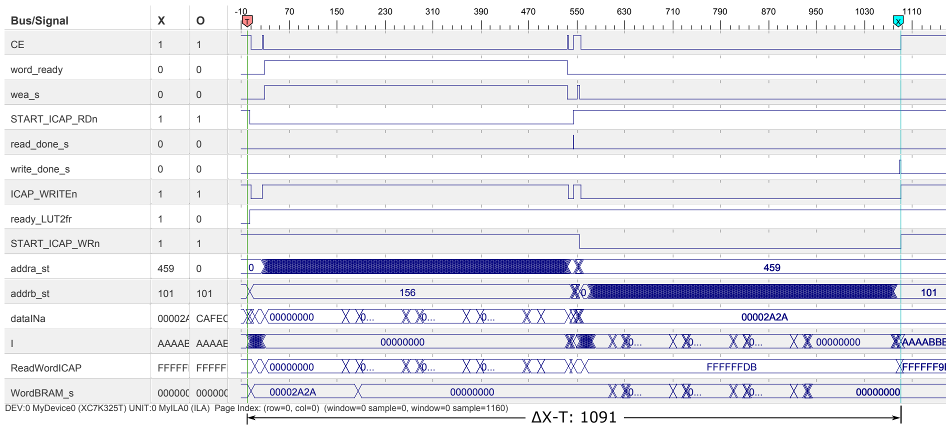


Figure 3.12 | Chipscope detail of LUT-DPR with AC_ICAP in Kintex7

new Xilinx families. This IP is identified as AXI_AC_ICAP and support the same functions presented in Code 1 that were adapted to the AXI API.

Based on the previous descriptions we have different variations of the controller to evaluate. AC_ICAP: the standalone hardware version, PLB_AC_ICAP and AXI_AC_ICAP: adapted to PLB and AXI buses respectively, and FSL_AC_ICAP: used as a co-processor. We used PlanAhead 14.7 and Vivado 2015.3 to define reconfigurable partitions of different sizes (from 1 to 10 CLB columns) and generate different partial bitstreams.
For the Xilinx-based controllers we implemented architectures such as the depicted in Fig. 3.10 but instead of using the PLB_AC_ICAP we added the XPS_HWICAP or the AXI_HWICAP with the parameters that allow the best performance in reconfiguration throughput (Write FIFO Depth=1024, Read FIFO Depth=256 and FIFO type enabled). For these two cases, the Xilinx Flash memory controller was also included to have access to the partial bitstreams located in this memory. In doing this we can get accurate comparisons as we used the same tools version and synthesis options.

3.6 Experimental Results

This section summarizes the main results regarding the reconfiguration speed and resources utilization of the diverse versions of the AC_ICAP controller. We consider as primary reference to compare the Xilinx XPS_HWICAP for Virtex-5 and the AXI_HWICAP for Kintex7 as these are, among the reported alternatives, the ones that support most of the DPR tasks. We take into account that for partial bitstreams that configure up to 4 CLB columns it is possible to copy them into BRAM as it is limited to 28.7 KB for Virtex-5 and to 99 KB for Kintex7. To record the time performance of the AC_ICAP (standalone version), the ChipScope Pro was used. For versions adapted to processor interfaces the timer included in the systems was used to register the number of clock cycles required for the specific tasks. These numbers are reported in Table 3.2. Here we want to mention some issues in regard to the values obtained for Kintex7 FPGA. The LUT functions that the AXI_HWICAP includes do not support the 7 family. With the most recent version of the tools at the moment of performing the experiments (Vivado 2015.3 and driver hwicap_v10_0), the support is only given for Virtex6 and previous devices and we cannot modify them as the source code is not available. The functions for read and write frames using the AXI_HWICAP required the modification of some header files as these present some erroneous values. The file xhwicap_i.h: uses the values for Virtex6 in the 7 family but these should not be the same. For instance, it is declared that the Number of Words in a frame for both families is 81. But for 7 series families the correct value is 101. Something similar happens with the FAR creation. The driver creates the FAR with some parameters that are valid for Virtex-6 but not for Kintex7 and these were modified to obtain correct operation.

Table 3.2 | Timing behavior of AC_ICAP

	Controller	LUT Reconf [μs]	Read Frame[μs]	Write Frame[μs]	Reconf BRAM [MB/s]	Reconf Flash [MB/s]
Kintex7*	AC_ICAP	10.91	2.39	2.33	380.47	14.66
	AXI_AC_ICAP	11.78	3.06	3.01	378.37	14.65
	AXI_HWICAP [77]	n/a	58.08	63.54	n/a	1.25
Virtex-5**	AC_ICAP	4.98	1.18	1.17	381.03	14.67
	PLB_AC_ICAP	5.88	1.90	1.90	378.73	14.66
	FSL_AC_ICAP	5.36	1.57	1.56	378.85	14.67
	XPS_HWICAP [1]	1912.17	29.21	32.16	n/a	1.32
	[3]	n/a	n/a	n/a	384.29§	6.57
	[32]	n/a	n/a	n/a	n/a	0.86
Virtex4†	[4]	n/a	n/a	n/a	371.42	n/a
	BRAM_HWICAP [29]	n/a	n/a	n/a	371.4	n/a
	ICAP-I [33]	n/a	n/a	n/a	180	29‡

* All 7 series FPGA are 6 input LUTs, Frames of 101 32-bit words.

** Virtex-5: 6 input LUTs, Frames of 41 32-bit words

† Virtex4: 4 input LUTs, Frames of 41 32-bit words

‡ Valid for SD memory AT49BV322A

§ Estimated value, not implemented

As it can be observed from Table 3.2 the reconfiguration time of LUTs using the AC_ICAP is, at the best of our knowledge, the fastest reported alternative. Compared to the XPS_HWICAP in Virtex-5, it implies speed up of more than 320 times for the PLB_AC_ICAP, the slowest version, and the standalone AC_ICAP offers improvement in reconfiguration time of LUTs of more than 380 times. In a similar way the speed up of read and write frame tasks, considering both Virtex-5 and Kintex7, experience improvements of more than 18 and 21 times respectively.

The reconfiguration throughput for load partial bitstreams from BRAM, for the AC_ICAP, is of 380.47 and 381.03 MB/s for Virtex-5 and Kintex7 respectively. It is close to the maximum supported throughput of 400 MB/s and to the reported values on [4] and [3]. For the work reported on [3] it should be noted that the value is estimated and not measured in a real implementation since that controller does not include BRAMs. The deviation of our controller from the 400 MB/s value is due to the extra clock cycles required to start reading the BRAM and processing the DESYNC command (0x0D) by the ICAP. For every ICAP related task, we consider it finishes when the DESYNC command is acknowledged. It is done by monitoring the O port of the ICAP which changes from 0xDF to 0x9F in Virtex-5 and from 0xFFFFFFDB to 0xFFFFFF9B in Kintex7, as a confirmation of the success in completing the tasks. This implies six extra clock cycles after the last data is sent to the ICAP.

For the PLB, AXI and FSL versions, there are some degradation in time due to the latency of the interfaces, but in all cases they offer improvements of more than 11times for load partial bitstream from Flash.
The time to copy the partial bitstrem from Flash to BRAM is on the same range as the required to load partial bitstream from Flash. Instead of send data to ICAP these are stored on BRAM. Therefore, it can be especially useful when the application can copy the partial bitstreams to BRAM before the execution starts, for instance at booting time.

In regard to resources utilization, Table 3.3 presents the details for every module of the AC_ICAP controller.

Table 3.3 | Resource utilization of AC_ICAP

Module	Virtex5			Kintex7		
	LUT	FF	BRAM	LUT	FF	BRAM
AC_ICAP	1667	1161	7	1286	1193	22
+Top FSM	691	471	0	452	585	0
++BRAM	36	3	7	27	3	22
++Load Partial bitstreams	109	133	0	70	130	0
+++Flash controller	129	112	0	73	68	0
++Lut2Frames	119	118	0	39	136	0
++Read Frames	230	138	0	232	113	0
++Write Frames	353	186	0	393	158	0

It should be noted that the AC_ICAP includes the flash memory controller, which is not the case for XPS_HWICAP and AXI_HWICAP. Table 5.1 summarizes the resources required by the diverse options of the controller. The extra resources of PLB, AXI and FSL versions of the AC_ICAP are due to the wrapper logic required to adapt the controller to these interfaces. It can be seen that the most resource demanding approach uses 5% of the Slices, which can be considered a reasonable size as all the operations are done in hardware.

Finally in Table 3.5 we compare the resources required by complete MicroBlaze-based architectures including different versions of the ICAP controller. We can see that the systems using the AC_ICAP adapted to the PLB and FSL require in average 3% more resources of the Virtex-5 FPGA than the XPS_HWICAP alternative. This is the area overhead to pay in order to speed-up all the reconfiguration tasks, such as the reconfiguration time of LUTs that is improved in 356X when the FSL_AC_ICAP is used. When we see the data for Kintex7 the area percentage are lower as the devices are bigger. Therefore the speed up of tasks takes increased relevance as the quantity of configuration data to manage has become bigger but the

Table 3.4 Resource utilization of ICAP controllers

	Controller	Slices	LUTs	Flip Flops	BRAM
Virtex-5	AC_ICAP	690 (3%)	1667 (2%)	1161 (1%)	7 (4%)
	PLB_AC_ICAP	952 (5%)	2375 (3%)	1609 (2%)	7 (4%)
	FSL_AC_ICAP	903 (5%)	2329 (3%)	1484 (2%)	7 (4%)
	XPS_HWICAP [1]	453 (2%)	714 (1%)	745 (1%)	3 (2%)
	[3]	*	96	87	0
	[32]	*	*	*	*
Kintex7	AC_ICAP	595 (1%)	1286 (1%)	1193(1%)	22 (5%)
	AXI_AC_ICAP	734 (1%)	1578 (1%)	1332 (1%)	22 (5%)
	AXI_HWICAP	248 (1%)	546 (1%)	741 (1%)	2 (1%)

* Not reported

speed and bus width supported by the ICAP primitive remains the same since the Virtex-4 generation (32-bits@100MHz). From the presented data, we can summarize that the best performance-area trade-off is given by the AC_ICAP which uses 3% of the FPGA resources but offers speed up of 380X in LUTs DPR.

Table 3.5 Resource utilization of full systems with different ICAP controllers

	System using:	Slices	LUTs	Flip Flops	BRAM
V5	PLB_AC_ICAP	2084 (12%)	4556 (6%)	3516 (5%)	23 (15%)
	FSL_AC_ICAP	2094 (12%)	4643 (6%)	3450 (4%)	23 (15%)
	XPS_HWICAP	1631 (9%)	3077 (4%)	2981 (4%)	19 (12%)
K7	AXI_AC_ICAP	2054 (4%)	4160 (2%)	3725 (1%)	37 (8%)
	AXI_HWICAP	1471 (2%)	3311 (1%)	2708 (1%)	17 (3%)

Dynamic partial reconfiguration of LUTs using this approach presents the

advantage that it does not require pre-computed partial bitstreams for each modification to be performed. It allows run-time LUT modification with any boolean value and it is not limited by the availability of partial bitstreams in memory. This fine partial run-time reconfiguration is of increasing relevance in applications such as fault injection platforms and in cryptographic implementations where the hardware can be modified at LUT level to avoid certain types of attacks.

3.7 Summary

In this chapter we have presented the AC_ICAP, a new ICAP controller verified in Virtex-5 and Kintex7 FPGAs. It is able to load partial bitstreams, read and write frames and also modify any LUT in the FPGA, in this last case without the need of pre-generated partial bitstreams.. The controller was adapted to be easily included in systems with embedded processors using the PLB, FSL and AXI links. Reconfiguration speed analysis of the processor independent version show improvement of more than 380 times in run-time reconfiguration of LUTs compared to XPS_HWICAP functions for Virtex-5 FPGAs. As our controller is fully implemented in hardware it obviously requires more resources, but in any case it occupies more than 5% of the available elements on the device. Therefore the AC_ICAP offers a complete high speed solution to perform diverse dynamic partial reconfiguration tasks with acceptable FPGA footprint.

The main contributions of this chapter are:

- Design and implementation of the AC_ICAP controller that supports DPR of LUTs.
- Transparent on-chip translation of LUT coordinates and LUT configuration values into frames locations.
- Speed up of the LUT DPR and similar reconfiguration speed (compared to existing solutions) for partial bitstreams located in BRAM or flash memory.

- FSM standalone operation and IP versions adapted to different Embedded Microprocessor interfaces (PLB, FSL).

ASIC Fault emulator through DPR | 4

Following the utilization of DPR in the evaluation of electronic systems we developed an emulator of permanent faults in ASIC designs mapped into FPGAs. In the case of the emulation of permanent faults in SRAM-based FPGAs it is mandatory to have precise control of the LUTs to modify its configuration value and consequently to change its logic behavior. The emulation approach is described in Section 4.1. In Section 4.2 we present the proposed methodology consisting of the CAD flow for constrained technology mapping and fault dictionary generation, the automatic extraction of test patterns and its representation and the architecture of hardware emulation platform. It is followed by Section 5.5 that supports the proposed methodology with experimental results; and finally, Section 5.6 concludes the chapter.

4.1 ASIC fault emulation using an FPGA

State-of-the-art SRAM based FPGAs consists of tiles of different primitive types mainly supporting routing infrastructure, logic and Input/Output functionality. There are tiles for Configurable Logic Blocks (CLBs), DSPs, BRAMs, IOBs and Interconnects. Each CLB can connect to the global horizontal and vertical interconnects lines using the Interconnect Tile located near to it. The CLBs are the workhorse of FPGAs for implementing combinatorial and sequential elements of a circuit. Each CLB consists of a number of function generators in form of LUTs. Within each CLB, flip-flops reside in close proximity to LUTs as can be seen in Fig. 4.1. For a LUT

with “k” inputs the maximum number of configurations is 2^{2^k}.

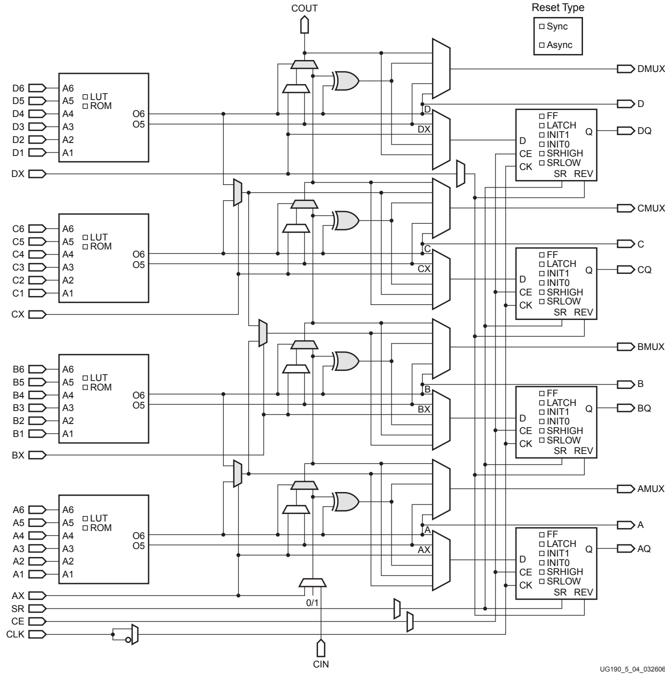


Figure 4.1 | Slice architecture for Virtex-5 FPGA

Internally, a LUT is implemented using multiplexers which select from the LUT configuration string or LUT INIT value stored in SRAM-cells at the input of multiplexers according to the current inputs at the address lines as shown in details in Fig. 4.2. This figure shows the circuit that needs to be mapped to a LUT labeled with wire names, a truth table showing the equivalent functionality and implementation of a LUT realized as a tree of multiplexers. The top row of the truth table is annotated with the LUT address line names and its assignment to the circuit’s primary inputs “a”, “b” and “c”. This assignment is a particularly important step that is determined by the routing phase of the FPGA CAD flow and affects the circuits timing to a great extent. The configuration of a LUT to the mapped circuit requires a correct LUT initialization value often termed as LUT INIT. Fig. 4.2 shows two methods to determine this LUT configuration

value for a circuit under consideration. In the truth table method, the combination of output bits into nibbles (half byte) and its arrangement position in the final LUT INIT value is vital for the correct configuration of a LUT. For example, in this case of a 3-LUT the final configuration LUT INIT value is “D5”. The truth table method become quite cumbersome when determining the LUT INIT value for a 6-LUT requiring a table of 64 entries to be constructed. The alternative equation method is more scalable and general and applies the basic Boolean logic operation for the formation of correct configuration value as shown in Fig. 4.2. The same methods can be used for finding the LUT configuration value if we are interested in permanent stuck-at fault emulation for one of the wire in the circuit under consideration. For example, for the wire “e” stuck-at zero the required LUT configuration value will be determined only by the value at wire “d" which is “C0”. This methodology can be applied to the primary inputs for the circuit which are bound to the LUT address lines or the internal wires, therefore, all the faults on the structure of the circuit can be emulated in this manner. Similarly, fault emulation for a flip-flop needs to be considered. Unlike, the ASIC gates which may undergo optimizations resulting in changes to circuit structure which must be retained for the guaranteed emulation of ASIC faults, the flip-flops in the ASIC netlist are always retained during the FPGA CAD flow process. There are several inputs lines for a flip-flop where fault emulation may be desired but the FPGA slice architecture can become a hindrance. All the flip-flops of a slice are connected to the same clock, set, reset and clock enable signals which means that it is not possible to separately emulated faults at the input of these lines. Also, most of the times all the slice flip-flops sites are not configured in this manner. This scenario can be considered equivalent to the case of a multiple fan-out nets when performing fault simulation with traditional software based approaches in which the faults at all the sinks are equivalent and are only performed once. Therefore, the fault emulation for the clock and control signals is performed once for all the slice flip-flops. For the flip-flop “D” input, assuming that the multiple fan-out faults are treated as equivalent, a LUT that connects to the primary input of the flip-flop, can be reconfigured for the fault

emulation purposes. This assumption eliminates the requirement to leverage the local interconnect tile to be configured as a local ground/vcc for stuck-at

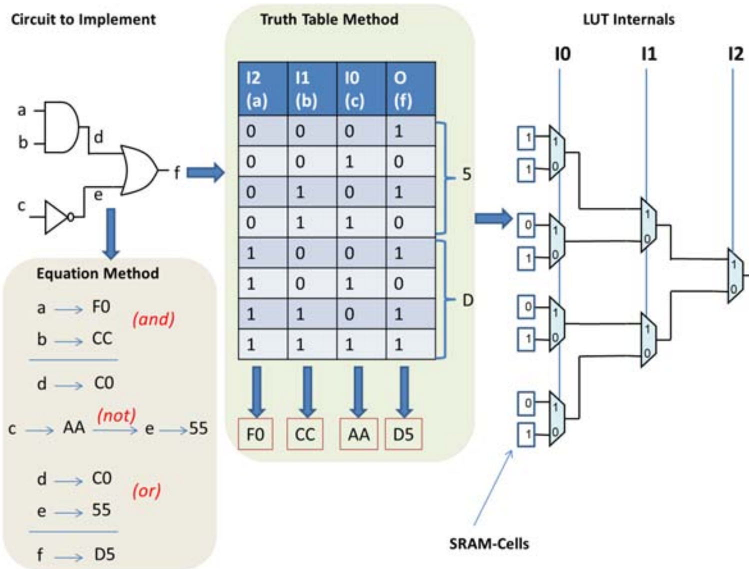


Figure 4.2 | Example of mapping a circuit to an FPGA LUT

fault emulation at the input of a flip-flop which was utilized in a preliminary work by the authors in [42]. These relaxations achieved with the assumptions of fault equivalence relieves us from the dependence on routing-level changes required for emulation of faults at the input of flip-flops, therefore, post-place and route level modification of the physical description of the circuits in form of Xilinx Design Language (XDL) format are totally avoided. This not only helps with the development of a cleaner and seamlessly integratable CAD flow but also optimizes the reconfiguration times with alternative fault injection approaches based on available APIs from the commercial vendor tools as will be explained in details in the hardware platform (section 4.2.2). Exploiting the LUTs and flip-flops for fault emulation assumes that the ASIC net-list is maintained during the CAD flow, however, the flow significantly differs from ASIC's one because the underlying technology both utilize are very different. Therefore, a standard FPGA tool-flow may result in different circuit structure since these tools usually make several logic optimizations modifying the ASIC net-list. The mapping of ASIC gates to LUTs, is usually referred to as K-LUT technol-

ogy mapping. This process may duplicate part of the original ASIC gates for reducing the number of LUTs or reducing delay or both [78]. Another optimization uses functional decomposition of ASIC gates to smaller gates for reducing the number of LUTs [79]. However, it is not possible to know how the FPGA tools partitioned the ASIC net-list for mapping to LUTs. In any case, the different approaches for K-LUT technology mapping combine multiple ASIC gates into one FPGA LUT, while maintaining the circuit flip-flops. Thus, it is possible to exploit the similarity between the two net-lists for fault emulation purposes. However, for guaranteed fault emulation (i.e., every ASIC fault has a corresponding LUT configuration) the behavior of both the ASIC circuit and the FPGA model must be the same for all possible inputs. Using this criteria for finding the equivalent LUT configuration for each ASIC fault using commercial software CAD tools for FPGA is hugely time consuming. Consider the case when multiple LUTs reside in a combinational logic between two flip-flops. For finding the equivalent LUT configuration for an ASIC fault the size of search space is shown in equation 4.1.

$$N_{PIs} * (N_{luts} * 2^{2^k}) \tag{4.1}$$

Where "Nluts" represents the number of LUTs that resides in the combination logic between two flip-flops, "NPIs" is the number of primary inputs and 2^{2^k} is the number of possible configuration for each LUT. Obviously, for guaranteeing fault emulation this is prohibitively time consuming task, and diminishes the returns of hardware emulation when all faults are considered. The search space for this problem can be limited by using the active number of inputs to the LUT as presented in [54] and [55]. However, for guaranteed fault emulation it is still excessively large. Therefore, it is necessary to use a custom technology mapping that avoids changes to net-list and results in a known mapping of ASIC gates to FPGA LUTs.

4.2 Proposed Methodology

In this section, we discuss in detail the proposed methodology for ASIC fault emulation on state of the art Xilinx FPGAs. The overall scheme consists of a novel CAD flow and a hardware fault injection and emulation platform. The

CAD flow enables the translation of ASIC designs into equivalent FPGA descriptions. This is achieved thanks to the custom mapper that translates the ASIC circuit net-list to FPGA LUT-level net-list and allows the equivalence between ASIC logic and FPGA LUT configuration values to be preserved. A commercial tool produces the ASIC fault list and the test patterns. Both of them are transformed into fault list and test patterns adapted to the FPGA-based implementation. The outcomes of the mapper are used in the design of the fault emulator. The FPGA LUT-level net-list hardware is included in the fault emulation platform which also takes this the fault list and leverages the partial reconfiguration capabilities of state-of-the-art reconfigurable FPGAs for fault injection. Based on the fault list a single LUT modification is performed to emulate a fault. It is followed by the application of test patterns for fault emulation purposesto analyze the effects of the emulated fault on the circuit. The complete methodology is described in detail in the following sections.

4.2.1 The proposed CAD Flow

The proposed CAD flow is responsible for generating equivalent fault dictionaries for the ASIC fault emulation. The flow consists of three main phases: a custom technology mapping of the ASIC net-list to LUT-level FPGA net-list, the creation of a fault dictionary and the extraction of test patterns. The developed tools integrate in-house ad-hoc tools with commercial FPGAs tool-chain and uses Boost C++ libraries while extending the framework presented in [80]. The flow starts by parsing a gate-level synthesized ASIC netlist to build a Directed Acyclic Graph (DAG). A duplication-free mapping algorithm then processes the in-memory graph representation of the ASIC netlist to generate a new DAG with LUT-based nodes combining multiple ASIC gates as will be discussed in details in section 4.2.1. This mapped LUT-level netlist is then passed through commercial tools to implement the design on an FPGA and produce place and route level information that needs to be fed to the Fault Dictionary Generator tool. The Fault dictionary generator algorithm produces several versions for the dictionary that have different space and time trade-offs and are also influenced by the fault location in the FPGA netlist. This will be described in detail in section 4.2.1. The test patterns are generated with industrial strength

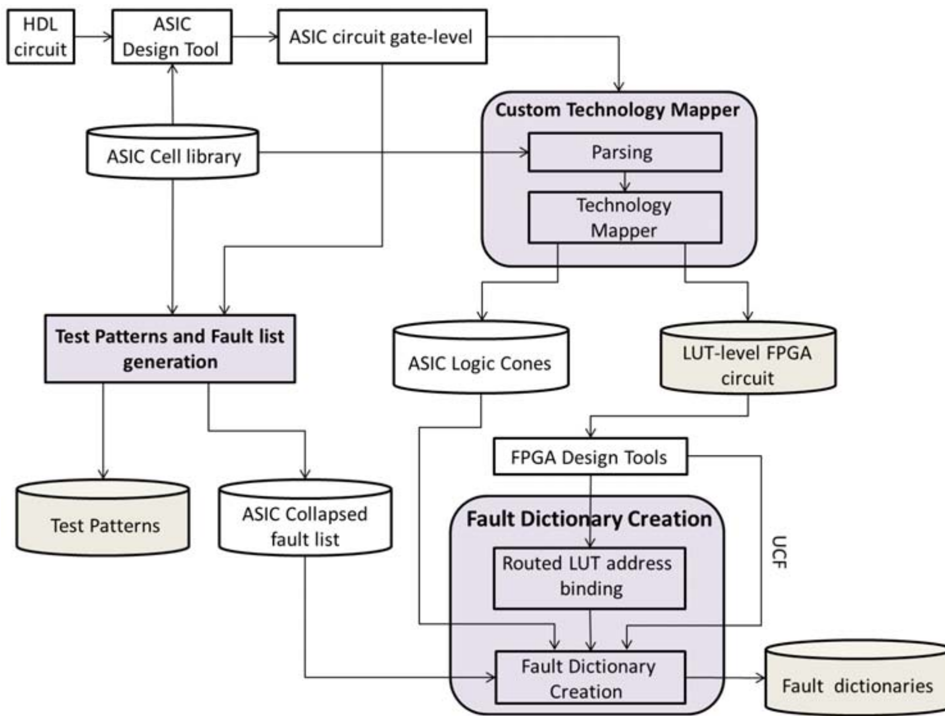


Figure 4.3 | Constrained Technology Mapping and Fault Dictionary Creation Flow

ATPG tools and stored in Standard Test Interface Language (STIL) format. An in-house STIL parser is developed to extract the test patterns to be used later with the fault emulation on FPGAs as will be described in section 4.2.1. Fig. 4.3 pictorially represents the whole flow and the following sub-section describes it in more detail.

Mapping ASIC design to FPGA

The goal of this step is to translate an ASIC gate-level net-list to a LUT-level FPGA net-list suitable for fault emulation. This problem of converting the ASIC gates to LUTs is usually called K-LUT technology mapping. The existence of multiple fan-out nodes makes the problem of optimal technology

mapping very challenging [81].

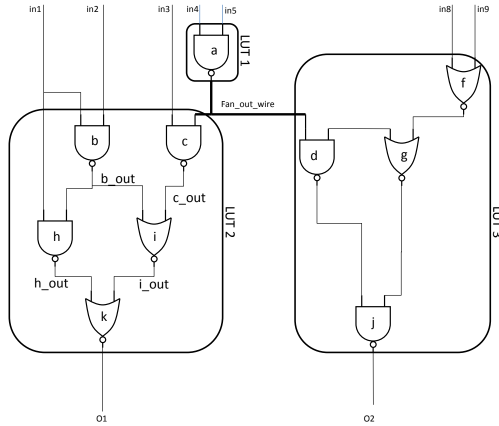


Figure 4.4 | Mapping without duplication

Fig. 4.4 shows an ASIC net-list with a couple of multiple fan-out gates. The circuit is then mapped to three FPGA LUTs, represented in Fig. 4.4 by the boxes labeled as LUT1, LUT2, and LUT3. This technology mapping uses the concept of Maximum Fan-out Free Cones ($MFFCs$) [82]. The MFFC of a node "v" represents the maximum number of nodes in the transitive fan-in of a gate in such a way that the fan-out of every node in the MFFC except the node "v" is inside the MFFC. For example, the MFFC of node "k" represented by $MFFC_k$ is composed of all the gates in the transitive fan-in "h, i, b, c" except the node "a" because it fan-outs to $MFFC_j$. It is interesting to note that gate "b" also has multiple fan-out but they re-converge on gate "k" and therefore are a part of $MFFC_k$. If the MFFC is k-feasible (i.e., the number of inputs are less than or equal to k, the maximum number of inputs to a LUT) can be collapsed into a LUT. All the $MFFCs$ in Fig. 4.5 are k-feasible and therefore are a candidate for a LUT. The advantage of MFFC based mapping is that emulating a fault on a multiple fan-out node requires reconfiguring only a single LUT and thus can exploit a collapsed ASIC fault list to reduce the fault emulation time. However, the required number of LUTs can be reduced from three to two if duplication is allowed as shown in Fig. 4.5. It can be noted that node "a" has been duplicated. For fault emulation purposes every ASIC fault on the

structure of node “a” should be emulated in both LUTs. With state of the art ASIC synthesis tools it is often the case that a gate fan-outs to more than two gates. As a result, allowing duplication during technology mapping would mean that a single ASIC fault is converted to multiple FPGA faults.

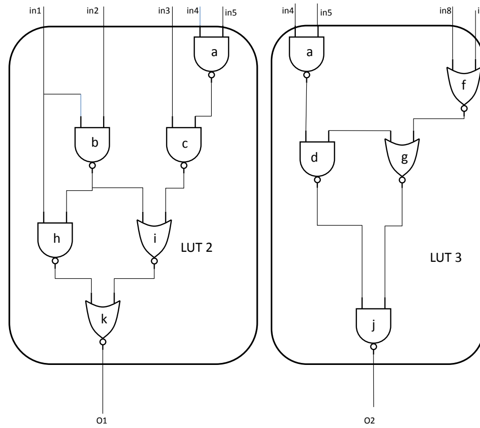


Figure 4.5 | Mapping with duplication

In the proposed approach the presented mapping process uses MFFC duplication-free mapping of the ASIC net-list avoiding nodes duplication in the resulting LUTs. In order to create the actual LUT-level net-list the initial values every LUT is going to assume need to be identified. Using the equation method with arbitrary binding of cones input wires to LUT address lines, a LUT initialization value is calculated and the resultant description is translated to the required format to be mapped into the FPGA. The standard FPGA CAD steps are performed for the implementation of the golden version, followed by the bitstream generation phase.

Fault Dictionary Creation

The Fault Dictionary Creation (FDC) phase is responsible for generating fault dictionaries compatible with the requirements for partial reconfiguration based fault injection. The FDC algorithm requires inputs from the mapper and commercial FPGA implementation phases. The information from the mapper includes the databases for where the individual ASIC gates

end up in the FPGA LUTs and the local gate-level net-list database for each of these LUTs. While the information from the commercial implementation phase consists of the BEL-level User Constraints File (UCF) and the post place and route level Verilog simulation model. Also, the collapsed fault list for the ASIC net-list derived using a commercial ASIC fault simulator is also required for the FDC phase. The FDC algorithm is presented in detail in Algorithm 4.1. In the initialization phase, a number of databases are created from the input files from the mapper and the physical information generated through commercial tools. After population of these data structures, the next phase of the algorithm reads as ASIC collapsed fault list picking up one fault and generating a corresponding FPGA fault. In detail, for each ASIC fault in the fault list file the corresponding gate, port and fault type (SA0 or SA1) are extracted. A quick lookup of the gate in the gates to cones database generated during the mapping process yields the corresponding LUT in which the gate resides. As the resident cones in each LUT are stored as an EDIF net-list, they are imported back to generate an in-memory Directed Acyclic Graph (DAG) representation. To get the corresponding fault site in terms of the net in the DAG, the gates and port names are used. At this point we know the net on which fault has to be emulated and the LUT DAG. The other important information is LUT input address lines binding for the DAG inputs wires which are determined by the routing phase of the FPGA flow.

Listing 4.1 | Fault Dictionary Creation (FDC) algorithm

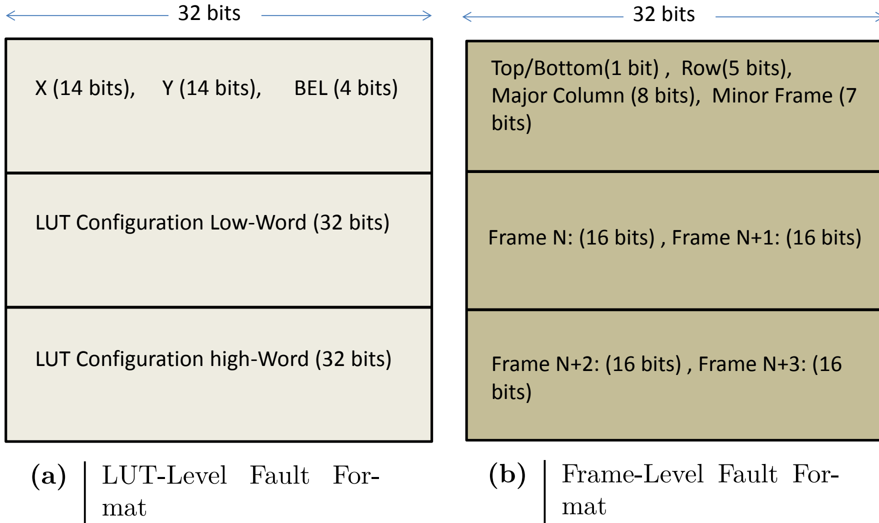
```
Input:     ASIC Logic Cones filepath, ASIC Gates2Lut filepath,
           ASIC Collapsed Fault List,
           UCF, Verilog Model, XDL description
Output:    LUT-level Fault Dictionary,
      Frame-level Fault Dictionary,
           Partial Bit-stream level Fault Dictionary
Definitions: map<Net,bitset> WireValueMap; //LUT-level FD
   map<Net,string> WireValueMap; //Partial Bitstream FD

//******* Initialization Phase ****************
1:   LutConesDB    = buildLutConesDB(ASIC Logic Cones filepath);
2:   Gates2LutDB   = buildGates2LutDB(ASIC Gates2Lut filepath);
3:   LutPlacementInfoDB = buildREsourceLocationMap(UCF);
```

```
4:  LutRouterInfoDB  = LutParser(Verilog Model);
//******* Fault Dictionary Creation Phase ****************
5:  While (ASIC fault list file is not read) do
6:   [gateName, portName, fault] = split_read_line(current_line);
7:      LogicConeName = Gates2LutDB[gateName];
8: LogicConePath = LutConesDB[LogicConeName];
9: LogicConeDAG = buildLogicConeDAG(LogicConePath);
10: faultyNet   = getFaultSite(LogicConeDAG,gateName,portName);
11: WireValueMap = LutRouterInfoDB[LogicConeDAG];
12: InstanceStack = TopologicalSort(LogicConeDAG);
13: while(InstanceStack is not empty) do
14:  Instance = InstanceStack.top();
15:  OutputNet = gateValueCalculation(WireValueMap,Instance);
16:  if(OutputNet==faultyNet) do
        //StuckAtValue: 0xFFFFFFFFFFFFFFFF or 0x0000000000000000 or
      //an equivalent equation
17:    WireValueMap[OutputNet]=StuckAtValue;
18:  endif
19:  InstanceStack.pop();
20: endwhile
21: [Slice , BEL]  = LutPlacementInfoDB[LogicConeName];
22: LUT_faulty_Value = WireValueMap[Output];
23: formatted_fault =
    faultformatconverter(Slice,BEL,LUT_faulty_Value);
24: WriteToBinaryFile(formatted_fault);
25:  endwhile
```

The annotated LUT DAG is topologically sorted and the sorted instances are stored on a stack. The calculation of faulty LUT INIT value follows by popping instances from the stack and applying the Boolean operation according to LUT DAG structure following the equation method presented in section 4.1. Whenever the faulty net is encountered a precedence set on the faulty net overwrites the current values calculated through the traversal of LUT DAG with predetermined values for SA0 and SA1 faults. In this manner, the faulty LUT INIT values are generated. After that, the physical placement information in form of slice and bel location is extracted from the user constraint file for the LUT DAG. This information uniquely identifies each LUT in the placed and routed design and is vital for the fault dictionary formats that the algorithm generates. Specifically, the algorithm

can generate fault dictionaries in three formats that are compatible with partial reconfiguration capabilities at different granularities exploiting the available APIs.



(a) | LUT-Level Fault Format

(b) | Frame-Level Fault Format

Figure 4.6 | Partial Reconfiguration Compatible Fault Formats

Fig. 4.6 shows the fault representation formats utilized by CLB-level and frame-level fault dictionaries while the third representation is based on the standard format of partial bitstreams generated by vendor tools. The CLB-level fault dictionary along with the pointer to reconfiguration controller and physical information necessary to identify the LUT requires the 64 bit LUT INIT value. The binary format defined for a fault in this dictionary consists of three fields each one of 32 bits as shown in Fig. 4.6a. The first field of 32 bits contains the LUT identification information as (x , y) coordinates each one represented by 14 bits locating the FPGA's slice where the LUT resides while the remaining 4 bits are related to the LUT BEL identification within each FPGA's slice. The remaining two 32 bit fields contain the low and high words for faulty LUT INIT value. The CLB-level fault dictionaries representation is based on an architectural representation and slice coordinate system making it necessary to convert this representation to the corresponding frames. As this processing is to be performed by the software running on the processor, this reduces the efficiency of fault emu-

lation. An alternative fault representation based directly on the frame-level details is shown in Fig. 4.6b. However, directly identifying the LUT frames is not an easy task because the vendor documentation barely provides the required level of details leaving only the option of extensive experimentation and hit and trial methods. Each CLB column is reconfigured by 36 frames each one 1 bit wide and 20 CLBs high. Four frames are required for a LUT reconfiguration as shown in Fig. 4.7. It is worth mentioning that the position of LUT frames in the CLB frames and the division of 64 bits across these frames is vital for correct fault injection as illustrated by the diagram in Fig. 4.7. It can be noted that the 64 bits spans four frames with each frame containing 16 bits for a LUT. Moreover, the function in this dictionary requires the minor frames offset of LUT to reconfigure, the major CLB column and Top/bottom division of the chip as shown Fig. 4.6b. The three fields in the binary format each one of 32 bits represents the physical frame position and the corresponding half-words that should be written to four consecutive frames for reconfiguration.

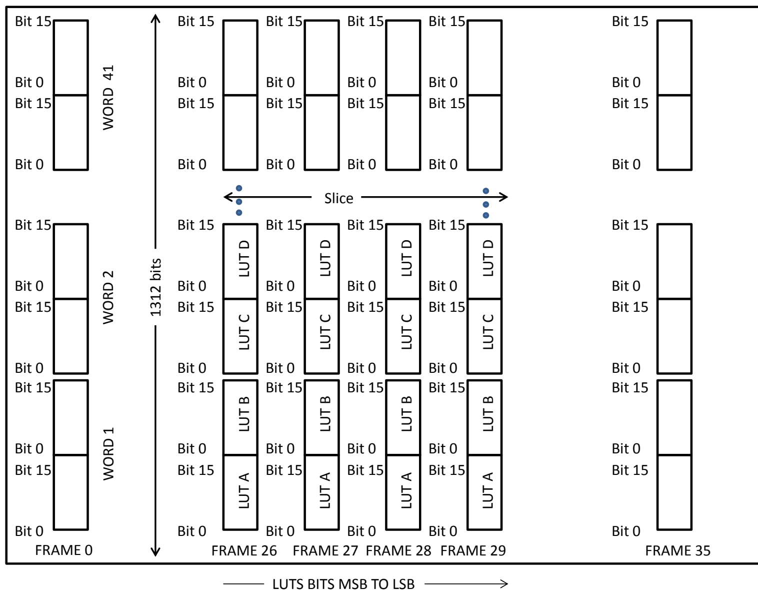


Figure 4.7 | Position of LUT frames in CLB frames

Test Patterns Extraction and Representation

Standard Test Language Interface (STIL) is an IEEE standard for semiconductor testing across various platforms and systems. The language is specifically designed for test vectors applications for combinational and sequential logic and is also extended to support mixed signal designs. After the fault dictionary generation phase, the proposed methodology utilizes an industrial strength Automatic Test Pattern Generation (ATPG) tool to generate test vectors that can cover the faults in the circuit. The test vectors are exported as a STIL format file. An ad-hoc parser tool is developed to automatically extract the signal names, signal widths and the patterns from the file and store them in binary format as shown in Fig. 4.8. It can be noted that the extracted information from the STIL file consists of several important pieces of information in order for successful application of the test patterns. Firstly, in Pattern block each pattern consists of several other patterns in form of primary inputs “pi”. The number of “pi” in each pattern are counted and stored in a binary file. Secondly, the individual “pi” are extracted and stored in binary format. It is important to mention here that the “pi” are further divided into signals whose width is determined from the signal block at the beginning of the STIL file. Thirdly, before the application of these “pi” the specification can either have a reset, clock pulse or no clock pulse. This information is stored for each “pi” and represented using a 2 bit number where “0” means no clock/reset, “1” means reset and “2” means clock. The resultant binary files successfully enables the translation of the test program in STIL format to our custom binary format that is used for test pattern application to our custom IP cores as will be explained in details in section 4.2.2.

4.2.2 Fault Injection and Emulation Platform

The fault emulator scheme is depicted in Fig 4.9. This was developed using a board equipped with the Virtex-5 LX110T FPGA. The fault injector is completely implemented in the FPGA board and a host computer, connected to the board through JTAG and serial links, is used to initialize the system and visualize the results of the experiments. The hardware/software co-design of the platform was made with Xilinx tools. This takes the LUT-

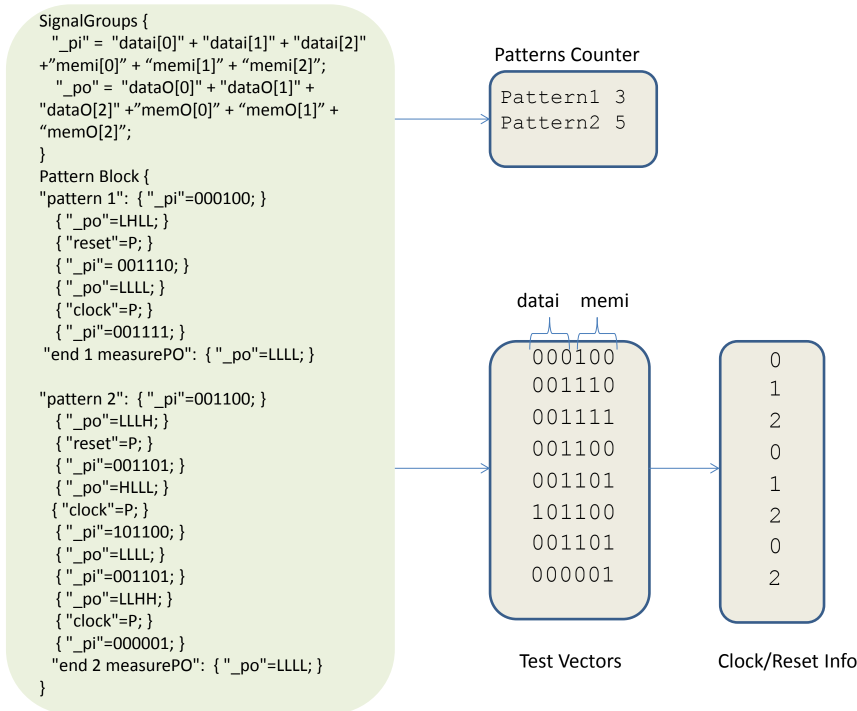


Figure 4.8 | Binary format for storing of STIL file

level FPGA netlist produced by the mapper to generate two instances of the circuit to be tested: one to compute the golden results and the other (DUT) to perform the fault injection. The DUT area placement step confines the DUT circuit into a region of the FPGA as required by the partial reconfiguration tasks. The post-route information of the DUT (location and INIT function of the LUTs) is used by the Fault Dictionary Creation process to generate the Fault dictionaries. The fault injector is controlled by the software running on the MicroBlaze processor, which employs the functions to perform dynamic partial reconfiguration and allow the faults to be emulated by modifying the configuration memory of the FPGA. After programming the FPGA, the Fault dictionaries and Test patterns are copied to the DRAM memory of the platform and then the fault injection loop can be started. The details of the hardware architecture and flow of the fault injector are presented next.

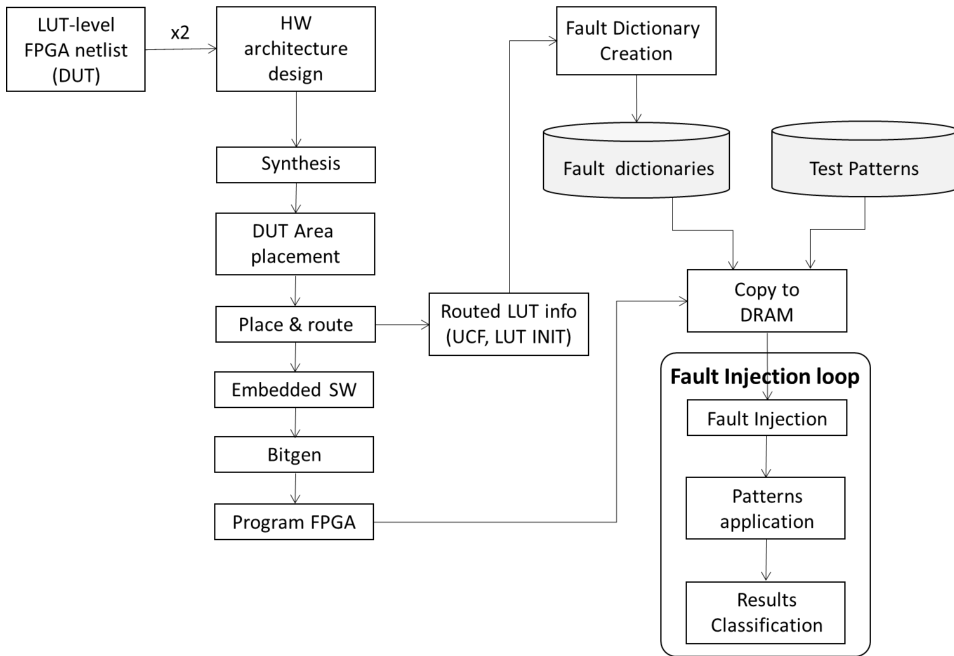


Figure 4.9 | Fault emulation scheme

Hardware Fault Emulation Platform

The hardware architecture of the fault emulator is depicted in Fig. 4.10. It is designed around a MicroBlaze-based system using the Peripheral Local Bus (PLB) to connect the components required by the platform. The circuits to be tested are designed as IP cores adapted to the PLB interface. The main elements of the platform are described next.

- DUT and GOLD IPs: The Design Under Test (DUT) IP contains the circuit to be tested. The netlist description of the circuit generated by the mapper (described in subsection 4.2.1) is instantiated in a PLB wrapper to obtain a custom IP. The inputs and outputs of the circuits are connected to the PLB bus and then the processor can access them using register addresses. In addition the outputs of the circuit are routed to user defined ports for later connection to the Error Flag

comparison logic. If the DUT is a sequential circuit, the reset and clock inputs are also routed to external ports as these should be connected to the Clock Control IP. The GOLD IP is a replica of the DUT and is used to compute online the correct outcomes of the circuit. In this way it is not necessary to pre-calculate the correct values for the pattern inputs nor store them in memory for later comparison. By doing this, we avoid delays in memory access and comparison in the processor software. Instead of that hardware comparison is performed in the Error Flag component which allows faster error detection using minimal resources. Once these two IPs are connected to the system location constraints are applied to confine the circuit to a specific region of the FPGA.

- ICAP: The Internal Configuration Access Port (ICAP) is the hard-wired element of the FPGA that gives access to its configuration memory. Xilinx tools offer the XPS_HWICAP controller to be used with the MicroBlaze processor. The combination of drivers and software routines provided with the XPS_HWICAP makes it possible to perform partial reconfiguration at different levels. It can be used to modify the hardware with partial bitstreams located in the system memory and to modify basic elements of the system such as LUTs. The CLB functions (XHwIcap_SetClbBits, XHwIcap_GetClbBits) and FRAME functions (XHwIcap_DeviceReadFrame, XHwIcap_DeviceWriteFrame) access the configuration memory thanks to this component.
  As presented in chapter 3 the developed AC_ICAP is an alternative controller to the XPS_HWICAP available from Xilinx tools with improved time performance. We use the PLB_AC_ICAP which is the version adapted to the PLB bus, and the XPS_HWICAP in the design of the platform in order to analyze the fault emulation speed. Therefore two versions of the platform are implemented. The first one uses the XPS_HWICAP controller and the second one uses the PLB_AC_ICAP.

- Clock Control: This IP generates the clock and reset signals for the GOLD and DUT IPs when these implement sequential circuits. This is required as the clock and reset signals should be precisely applied to propagate the test patterns. As explained in section 4.2.1, the STIL

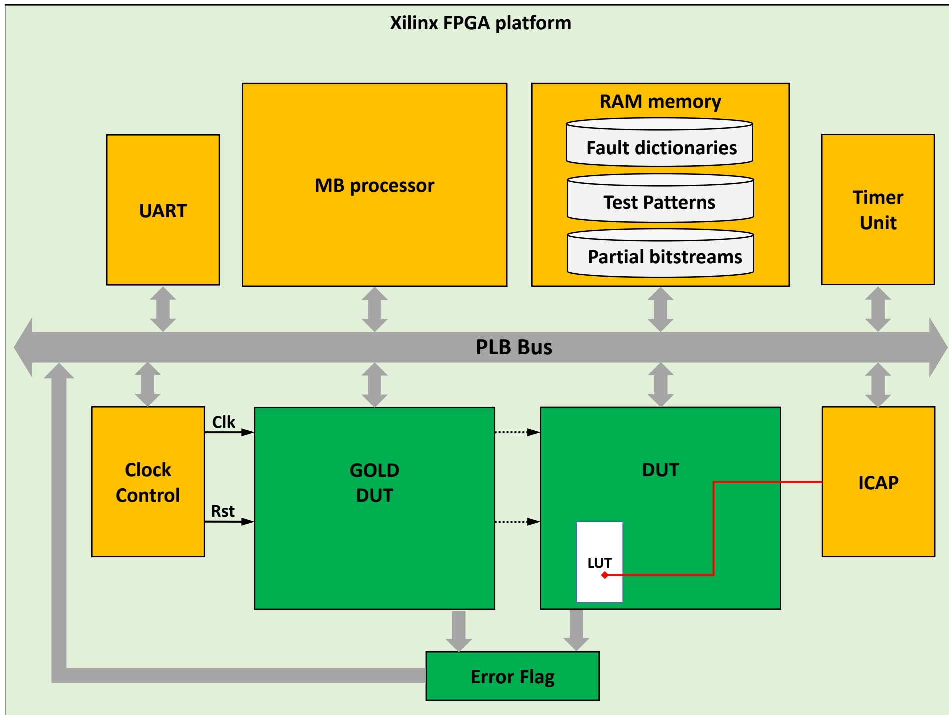


Figure 4.10 | Fault injection platform

format demands a reset, a clock pulse or any of these, before applying each test vector. The clock frequency is selected at the IP design phase taking into account the maximum throughput supported by the DUT circuit. Such information is obtained from the reports of the implementation tools. The processor commands this IP to generate the clock/reset signals that the DUTs require.

- MicroBlaze processor: This is a soft-core implemented in the logic of the FPGA and runs the algorithm that controls the fault injection process. Its software libraries include the functions to access the configuration memory using the ICAP. The UART serves as a user interface to control certain initial features of the experiments and also to visualize the results in the host computer. The reports include timing information of the experiments, collected using the Timer Unit along

with an Interrupt Controller, to precisely record the number of clock cycles elapsed during the execution of any task of the fault injection process.

Fault Injection Flow

The hardware system, designed according to the scheme depicted in the previous subsection, produces the FPGA configuration file. This bitstream is merged with the software application that runs in the MicroBlaze to obtain a single configuration file including the hardware and software components of the fault emulator system. The fault injection flow, depicted in Algorithm 5.1, is divided in three main parts. The initial steps are controlled from the host computer and the remaining steps are completely performed in the MicroBlaze-based system. First, the host computer downloads the bitstream to the FPGA. Once the device is configured the Load_memory() step is performed to copy the fault dictionaries and the test patterns into the RAM memory of the board. These two tasks use the JTAG link that connects the host computer to the board. From this point the system runs completely on the FPGA board but the Select_mode()function expects from the user one command to continue. By means of the serial communication link, Select_mode() gives the user the possibility to choose between different options to perform the fault injection process. As two versions of the platform are available, these options depend on the ICAP controller used.

For the XPS_HWICAP the options are:
(1) Using CLB functions
(2) Using Frame functions
(3) Using partial bitfiles

For the PLB_AC_ICAP there is only one option:
(1) Using ModifyLUT functions
After selecting the injection mode, the MicroBlaze application follows the next steps to run the fault injection:

1) The reset_DUT()function applies a Reset to the GOLD and DUT IPs.
2) For each fault N, fromDRAM(N) function reads the LOC coordinates

and LUT faulty value (FV) from the fault dictionary located in the RAM memory. Depending on the mode selected for the injection, LOC and FV change their format: If CLB functions are selected, LOC has the form of (x, y), bel coordinates and the FV is the LUT INIT that emulates the fault. If frame functions are used, LOC and FV have the form depicted Fig. 4.6b (Top, Rows, Major, Minor and 4 16-bit words for LUT value).

Listing 4.2 | Fault Injection Algorithm

```
//******* Initialization Phase ****************
1:  Load_memory();
2:  Select_mode();
//******* Run Injection Phase ******************
3: for each fault N
4: {
5:    reset_DUT();
6:    {LOC, FV} = fromDRAM(N);
7:    correct_lut = read_LUT(LOC);
8:    write_LUT(LOC, FV); //inject fault
9:    {PI} = fromDRAM(NP);
10:   RES = Gold_DUT_comp(PI);
11:   write_LUT(LOC, correct_lut); //correct fault
12: }
//******** Results Communication ***************
13: send_results(RES)
```

3) Read back, with the read_LUT(LOC) function, the configured value of the LUT to be modified. This non-faulty value is saved in correct_lut memory for later fault recovery. Again, the internal details of read_LUT(LOC)vary according to the selected mode and therefore uses XHwIcap_GetClbBits for CLB functions and XHwIcap_DeviceReadFrame for Frame functions. 4) Inject the fault by writing the FV to the LUT with LOC coordinates: write_LUT(LOC, FV). Following the same rule as the previous step, this function uses XHwIcap_SetClbBits or XHwIcap_DeviceWriteFrame. 5) Read from RAM memory (fromDRAM(NP)) the patterns to apply on the Golden and DUT circuits. PI refers to the patterns input to test the DUT. For combinational circuits, these test vectors are sent directly to the GOLD

and DUT inputs. For sequential circuits, these includes the clock/reset information and in these cases, the processor commands the Clock Control Unit to manage the reset and clock signals. 6) When a pattern block (PI) has been applied, the results of the GOLD IP are the correct values to be compared to the outcomes of the DUT at the Error Flag element (Gold_DUT_comp(PI)). The MicroBlaze reads the Error Flag output to classify the effect of the injected fault on the circuit behavior. The result of the classification is saved in the RES memory. 7) After the fault injection and the pattern applications, the LUT must be corrected. This is achieved by writing the correct_lut value read at step (3): write_LUT(LOC, correct_lut). At this point, continuing with the next fault until all the available faults in the dictionary are covered.

When all the faults are applied, the consolidated results are communicated to the host computer: send_results(RES).

For the version of the platform that uses the PLB_AC_ICAP, instead of using the XHwIcap_GetClbBits and XHwIcap_SetClbBits functions, the ModifyLUT (XYBel, INIT) and RecoverLUT (XYBel) functions mimic the same behavior but adapted to the PLB_AC_ICAP controller.

In addition to the previous descriptions, we implemented another fault injection scheme using DPR with pre-computed partial bitstreams. This option requires the generation of partial bitstreams for each fault to be emulated. It uses the difference between the circuit with the correct LUT and the circuit with the faulty one. According to the fault dictionary information we identify the LUT where a fault should be emulated, modify the LUT INIT value in the XDL description file, and generate the ndc file (xdl2ncd). The tool bitgen (with –r option) uses as inputs the original and modified circuit files to produce the partial bitstream based on the difference between them. This process should be repeated for each fault and obtain two partial bitstreams per fault. One with the faulty LUT and other with the original non-faulty LUT. The latter is necessary to correct the circuit after applying the patterns. As a result the number of partial bitstreams is twice the number of faults to be emulated. The partial bitstreams are then copied to the system RAM memory and the MicroBlaze software application performs the run-time reconfiguration using certain functions available in the Xilinx software libraries. We adapted the function XHwIcap_FLASH2Icap,

designed to read partial bitstreams from flash memory, to do the same when partial bitstreams reside in DRAM memory. Therefore, the modified function, ram2icap, requires the DRAM memory start address where the partial bitstream is located. The partial bitstream contains all the information regarding the size of the data and the specific configuration memory to be modified. When using this approach some modifications in the fault injection flow should be done. The LOC and FV are included in the partial bitstreams and therefore it is not necessary to read them from memory (fromDRAM(N)). In a similar way, the correct LUT, necessary for fault correction, is contained in the non-faulty partial bitstream and the correct_lut = read_LUT(LOC)step is not required. Instead of that, the ram2icap is used for both injection and correction.

4.3 Experimental Results

The experimental results were collected on a set of representative circuits relating to combination and sequential logic including some circuits from the ITC benchmarks. The selection of ITC benchmarks was to test the proposed methodology with circuits specifically designed for testing and fault simulation including hard to test circuits. The collected results are related to timing efficiency comparison, fault coverage, fault statistics and the area/delay overhead. Table 4.1 represents the comparison of fault simulation and fault emulation times considering the ASIC netlist and the FPGA netlist utilizing the proposed approach. The ASIC design was synthesized with Design Vision for pdt2002 library while the FPGA version utilizes Xilinx tools and the developed CAD tools in the proposed methodology.

Table 4.1 | Fault Emulation Times

Circuits	ASIC	XPS_HWICAP			PLB_AC_ICAP
	$T_{modelsim}$ (s)	$T_{lutlevel}$ (s)	$T_{partialbitstream}$ (s)	$T_{framelevel}$ (s)	$T_{ModifyLUT}$ (s)
Adder32	4.13	3.94	0.928	0.225	0.000271
Adder64	7.14	7.97	1.93	0.453	0.000543
Multiplier32	158.87	183.14	47.41	10.28	0.012397
Multiplier64	618.71	762.34	176.32	43.07	0.051942
B03	3.95	4.22	1.08	0.241	0.000291
B04	10.18	12.59	2.94	0.707	0.000852
B05	11.84	13.15	3.02	0.746	0.000899
B07	6.77	8.93	2.19	0.513	0.000618
B08	3.51	4.14	1.05	0.231	0.000278
B14	278.45	390.78	89.86	22.08	0.026628

The FPGA's results comprise four versions of the fault dictionaries that were previously discussed. It can be noted that the CLB-level fault dictionary version has relatively poorer performance than the traditional ASIC fault simulation times with software-based fault simulation. This is due to the large overhead of the software APIs to convert from abstract-level LUT representation to frame-level representation which requires processing times. Obviously, in this case the vendor APIs for reconfiguration are very slow and the advantages offered by at-hardware processing speed of FPGA are amortized. The partial bitfiles version of the fault dictionary has improved performance compared to the ASIC and the CLB-level fault dictionary because the processor is not required to perform intensive work for frame identification. The partial bitfiles version has header information and synchronization data making them slow compared to the frame-level fault dictionary version. In order to quantitatively analyze the differences between the fault emulation speeds of the three versions of fault dictionaries which mainly depend on the fault injection times we defined TPR_InjectCorrect as the time required to inject and correct a fault in a LUT. As detailed in section 4.2.2, when using CLB functions and Frame functions, for every single fault to be emulated we need to read the LUT programmed value (TreadLUT), perform the fault injection (TwriteLUT) and once all the patterns are applied the faulty LUT should be corrected (TwriteLUT). Therefore the time, considering all the LUT read and writes for a single fault, can be expressed according to equation 4.2.

$$TPR_{InjectCorrect} = T_{readLUT} + 2 * T_{writeLUT} \tag{4.2}$$

For LUT level representation, TreadLUT = 1.91ms and TwriteLUT = 2.88 ms, producing TPR_InjectCorrect = 7.67 ms. When Frame level representation is used, TreadLUT = 134 μs (4* reading one frame + backup its content), TwriteLUT = 150 μs (copy 4 words in frames + 4* writing one frame), and consequently, TPR_InjectCorrect = 434 μs. In the case of partial bitstreams, it is not necessary to read the LUT content before injecting a fault. The fault and correction are applied by loading the partial bitstreams. TloadPBits is the time required to perform the partial reconfiguration. It includes memory access time to read the partial bitstream, processing of the bitstream header, and data sending to the ICAP. This value depends on the size of the partial bitstream. For the modification of

a single LUT the size of the produced files is of 1588 bytes, which includes the header and data corresponding to four frames different between original and modified files (the same for faulty and non-faulty bitstreams). For such size TloadPBits= 873 μs. Then, TPR_InjectCorrect = 2*TloadPBits= 1.75 ms. This lower performance compared to frame functions is due to the processing of the partial bitstream data, which imply reading the bitstream header, determining bitstream length and adapting the format of data to send to ICAP. This is processed in the software of the MicroBlaze and affects the overall processing time. Specifically, frame level fault dictionary is 17 times faster than CLB-level fault dictionary and 4 times faster than partial bitsfiles based fault dictionary.

When using the PLB_AC_ICAP controller the time required to modify a single LUT is 5.88 μs and the recovery time of a LUT is 3.37 μs. This means that TPR_InjectCorrect = 9.25 μ. The advantage of this approach is that the controller is designed to keep the original configuration value of the last modified LUT in BRAM memory to be used by the RecoverLUT function. Therefore the RecoverLUT only requires the XYBel identification that should match the value of the previously used ModifyLUT function. In this way it is not necessary to read four frames again. Every time that ModifyLUT is performed the value before modification is kept in BRAM.

With these numbers the speed up obtained by using the PLB_AC_ICAP is of 829 times compared to frame functions of the XPS_HWICAP. Such speed improvement is possible thanks to the hardware-based processing performed by the AC_ICAP controller.

After getting the complete timing information for fault emulation of the presented circuits, we came up with a formula to calculate the time required to perform any other experiment. It is presented in equation 4.3

$$T_{FPGAemul} = Ncf*[T_{readDict}+T_{PR_InjectCorrect}+Np*(T_{Patterns}+T_{Classification})] \tag{4.3}$$

In this case we consider all the steps involved in performing full injection campaigns. $T_{readDict}$ represents the time required to read the LUT coordinates and configuration values from RAM memory and extract the informa-

tion (e.g. get x, y, bel from one 32-bit word). This time is not considered for partial bitstreams scheme. $T_{PR_{I}njectCorrect}$, as explained previously, depends on the injection scheme. $T_{Patterns}$ includes memory access time; bus latency for sending the data to the DUTs and circuit delay or clock period in case of sequential circuits (T_{pd}). In addition, for sequential circuits, the delay produced by commanding the Clock Control Unit is considered. Finally, $T_{Classification}$ measures the time for error flag reading and results classification.

Table 4.2 | Fault Statistics

Circuits	Total Faults	Ncf	Np	Coverage %
Adder32	1226	712	25	100
Adder64	2436	1410	34	100
Multiplier32	42304	23431	137	99.81
Multiplier64	170612	94125	288	99.87
B03	1018	527	14	73.48
B04	2768	1403	58	88.35
B05	3398	1703	2	9.81
B07	2128	1133	3	48.32
B08	1074	519	5	73.26
B14	49002	21457	396	87.21

Table 4.2 illustrates the results related to fault statistics including the total faults, the collapsed faults, the average number of patters to detect faults and the coverage achieved with these patterns. The deterministic mapping of ASIC gates to FPGA LUTs enables the achievement of the same coverage. Some of the benchmark circuits (for example B05) have a very low coverage and a small number of patterns because the ATPG tool from the Tetramax tool was taking a prohibitively long time because of the hard to test structure of the circuit. It would be interesting to investigate such behavior when the platform is extended to be used for automatic pattern

generation.

Table 4.3 | Area and Delay Comparison

Circuits			Xilinx Mapper		Custom Mapper	
	Gates	FFs	LUTs	Delay[ns]	LUTs	Delay[ns]
Adder32	160	0	49	17.031	64	23.167
Adder64	320	0	96	29.81	111	34.86
Multiplier32	7025	0	1624	43.4	2679	71.19
Multiplier64	28359	0	7110	87.68	11114	131.32
B03	122	30	39	2.392	76	5.212
B04	362	66	114	2.914	241	7.49
B05	498	34	142	3.895	384	13.161
B07	282	44	99	3.001	211	7.054
B08	145	21	25	2.401	97	5.076
B14	7706	215	1238	15.836	4534	50.125

Table 4.3 shows the characteristics of the ASIC and FPGA designs in terms of area and delay. It can be observed that the FPGA designs mapped with Xilinx mapper has better performance and area utilization than the custom mapper utilized by the proposed approach. It is due to the fact that the mapping phase is not optimized for minimizing area and delay which could result in increasing the size of the fault list by converting single ASIC faults to multiple FPGA faults. It would be interesting to utilize multi-objective optimization to simultaneously improve the area utilization while not drastically increasing the size of fault list.

4.4 Summary

This chapter presented a methodology to emulate ASIC permanent faults using FPGAs. A novel hardware fault emulation platform utilizing a semi-custom CAD flow was presented. It performs the injection of faults in

run-time using dynamic partial reconfiguration. The flow utilizes a custom technology mapping for directly converting the post layout gate-level net-list into a LUT level net-list. This known mapping of gates to LUTs enables us to develop equivalent fault dictionaries from the ASIC fault list to FPGA faults representation. The approach is flexible enough to generate fault dictionaries in different formats compatible with partial reconfiguration requirements which results in significant variation in achievable fault injection speedups. The methodology avoids drastic changes to the net-list, therefore, does not need lengthy re-compilation times during the fault emulation process. Furthermore, our experimental results demonstrate a significant speed up for fault emulation compared to software based fault simulation.

Zero-overhead partially reconfigurable and cross-domain errors resilient TMR scheme for SRAM-based FPGAs 5

This chapter focuses on Triple Modular Redundancy as it is a widely used fault-tolerance methodology for highly-reliable electronic systems mapped on SRAM-based FPGAs. However, the state-of-the-art TMR techniques are unable to effectively deal with cross-domain errors and increased scrubbing time due to growing size of configuration memory. Section 5.1 describes the generation of modified X-TMR circuit for fast fault detection and the post-mapping manipulation to re-use the LUTs that implement majority voters. It is followed by Section 5.2 that presents the proposed CAD flow based on the EDIF modification and alternative partial bitstream generation. Section 5.3 details the developed testbed to validate the approach which includes fault injection campaigns as described in Section 5.4. The results of the experiments are presented in Section 5.5.

5.1 Dinamically reconfigurable X-TMR

Xilinx TMR (X-TMR) is particularly designed for SRAM-based FPGAs, however, the commercial CAD tools are unaware of reliability-oriented physical design rules which are vital for avoiding Multiple Cell Upsets (MCUs)

causing Cross-Domain Errors (CDE). Avoiding CDEs with non-overlapping domain placement has been proposed by the authors in [71]. This work improves upon it by exploiting the fact that majority voters are residing in 6-LUT while only three inputs are utilized. An equivalence function can also be implemented in the same LUT with majority voter generating a logic-0 when the three inputs are not the same, otherwise, a logic-1 values is produced. This value is then steered towards the carry-chain multiplexers for realization of AND-tree required for generating a unique flag per TMR domain. The whole scheme is illustrated in figure 5.1. Instrumentation of the TMR circuit with the mentioned error detection logic is achieved with modification in a post-synthesis netlist. In particular, the usage of fracturable LUT for realization of majority voting and equivalence function is achieved through a custom re-mapper. However, this circuit-instrumentation at TMR domain granularity and non-overlapping domain placement is happening to a flat netlist which does not conform to the partial reconfiguration flow requirements supported by vendor tool-chains. Therefore, an alternative method for synthesizing the partial bitstreams by accumulation of frames corresponding to reconfigurable slots to which domains are mapped are extracted and stitched together to build valid partial bitstreams. The details of this developed methodology are given in the following sections.

5.2 CAD flow

This section presents the proposed CAD as depicted in Fig. 5.2. It details the steps required to implement and evaluate the modified X-TMR-based circuits.

5.2.1 Hierarchy formation

The proposed CAD flow in Fig. 5.2 starts with the conventional steps required for applying Xilinx TMR (XTMR) approach. The flat netlist represented in Electronic Design Interchange Format (EDIF) is passed through a hierarchy formation block. A Directed Acyclic Graph (DAG) is developed by parsing the EDIF representation. The DAG is processed using conventional

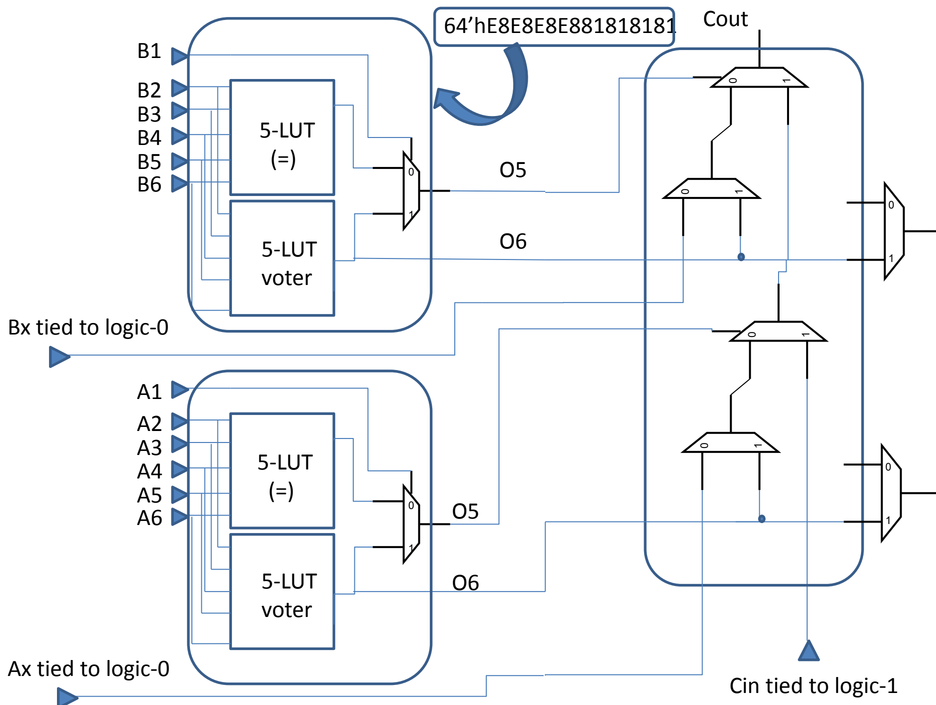


Figure 5.1 | TMR fracturable LUT

graph search algorithms to identify logic resources in each TMR domain. In particular, Voter elements are identified for the re-mapping phase in the flow. Each logic element is renamed in a hierarchical manner. In this way, the whole flat netlist is converted to hierarchical netlist to which user constrains can be applied seamlessly conforming to conventional requirements which the implementation tools supports.

5.2.2 Re-mapping, error detection and flag convergence

In the re-mapping phase, the LUT elements which implements the majority-voter functions identified in the previous steps are modified to realize a fracturable LUT as shown in Fig. 5.1. The majority-voters are instances of an EDIF cell TRV LUT which is a based upon a three input LUT configured

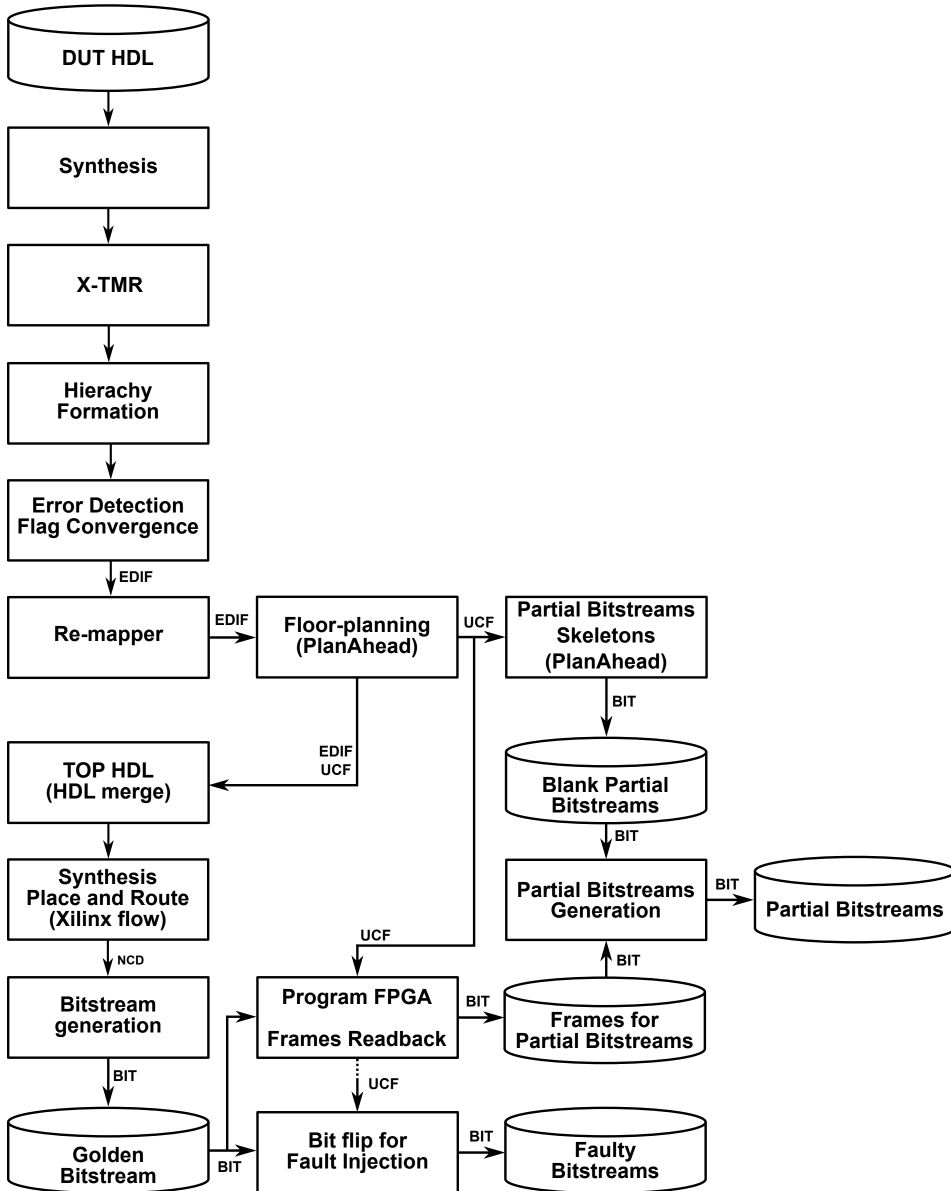


Figure 5.2 | Proposed CAD flow

with the string "E8". As the proposed methodology is based on utilization of dual output LUT, which is only possible with a six input LUT. Therefore, a new EDIF cell is designed based upon a 6-LUT and configured with the string "E8E8E8E881818181" to simultaneously implement majority voting and equivalence functions with the corresponding outputs on "O5" and "O6" output lines. The majority-voted "O6" output has to pass through an extra slice mux "AOUTMUX" for connections to other slice logic, consequently, introduction a propagation delay compared to the standard X-TMR circuits.

However, the propagation delay associated with this extra logic element has negligible effect on the circuit's critical path and hence its performance. The equivalence function's "O5" output has to control the select line of the carry-chain multiplexer according to the scheme outlines in Fig 5.1. However, the multiplexers belonging to slice's carry-chain have to be instantiated at cell-level and connected for realization of error detection and flag convergence. It is worth mentioning that the "Cin" line has to be connected to a logic-1 value and the auxiliary slice lines "Ax/Bx" lines have to be connected to logic-0 values. The "O5" output of the LUT will select which logic value propagates along the chain. These logic values are vital for a working AND-tree required for error detection and flag convergence. These logic values are readily available as TIE-OFF elements near every slice in modern FPGAs. The mapping tools will automatically infer the need of TIE-OFF elements in the place and route design. Each TMR domain is instrumented in this way and the corresponding flag signals are added as top-level EDIF ports in the netlist. Then, the standard tool-flow for mapping designs to FPGA is followed. After these custom modification, it is possible to add user constraints in ucf file to floor-plan each TMR domain to a separate non-overlapping region. However, as the requirements for partial reconfiguration flow dictates the partially re-configurable regions should be mapped into from separate modules and not a flat netlist, therefore, even after these custom modification for error detection and placement control, the partial bitstream generation is non-trivial and challenging due to no support through standard bitstream generation tools.

5.2.3 Partial bitstreams generation

The standard flow available in Xilinx does not support partial bitstream generation for each domain taking the modified X-TMR flat design as input. Even though the modified EDIF can be constrained in area to improve resilience to CDE, Xilinx tools require independent netlists for each partitioned area. As the X-TMR is a flat design independent netlists for each domain are not available. It would imply complex split of the flat edif design and therefore we use an alternative approach using the post-implemented information.

Initially we include the modified DUT-XTMR edif design and perform the floor-planing step to isolate each domain to a defined area. This is done in PlanAhead and it allows us to generate the User Constraint File (UCF) with the area in terms of (X,Y) Slice coordinates for each domain. As a result three regions are available to be used at different states of the design flow. These regions correspond to partitions for which partial bitstreams are required.

Our approach uses partial bitstreams skeletons for which we created a design with empty partitions occupying the same regions previously defined at the floor-planing step. These three partitions, also known as black boxes, permit blank partial bitstreams to be generated: one per reconfigurable partition. Each generated partial bitstreams contains key information such as the bitstream header, the initial control commands followed by the empty frames (all zero) and the final control commands. These Partial bitstreams are seen as skeletons to latter replace the empty frames with the actual configuration frames of the DUT. The standard flow for partial bitstream generation would require at this point to specify the different variations that each reconfigurable partition would perform (individual ngc files). As it is not available for the X-TMR flat design we are not able to use such approach. As alternative we instantiate the DUT-XTMR edif design, include the hardware based ICAP controller (FC_ICAP) and follow the standard flow to produce the full bitstream and program the FPGA. Once this is programmed the FSM commands the FC_ICAP to read back the frames corresponding to the three domains of the DUT-XTMR. Based on the information of the area where each domain is placed (UCF) we compute two

values: The initial frame address where the read back should start (StartAddr input of the FC_ICAP) and the number of frames that needs to be read (NumFrames input of the FC_ICAP).

The starting address has the format of the Frame Address Register (FAR). For its generation we consider the equation 5.1.

$$StartAddr = Init_{Slice}/2 + Family_{offset} + Col_{offset} \tag{5.1}$$

where $Init_{Slice}$ is the most left X coordinate of the region and $Family_{offset}$ is a device-family-dependent value that identifies the first CLB column (increases from left to right). It corresponds to the major address field of the FAR for that column. It is 1 for Virtex-5 and 2 for 7-series devices. Col_{offset} corresponds to the information of the intra columns of other components different from CLBs such as BRAMs, DSPs, IOBs.

The second value depends on the region and components of the FPGA required to implement the circuits. To this end we only use CLBs for which the corresponding number of frames is obtained according to equation 5.2;

$$N_{frames} = N_{CLB} * 36 \tag{5.2}$$

where N_{CLB} is the number of CLB columns occupied by a given area. It means that for every CLB column 36 frames are required. Depending on the family of the device the size of the CLB columns varies. For Virtex-5 one column of CLBs corresponds to 20x1 CLBs that spans one HCLK height. This information is used to command the FC_ICAP to read back the configuration memory of specific areas of the design. With these two parameters the FC_ICAP computes the number of words to read from the configuration memory. For Virtex-5 it is: $N_{words} = 41*(N_{frames}+1)$. These values for a specific domain are obtained using the ChipScope debugger and are used to replace the blank frames previously generated by planAhead as shown in Fig. 5.3. It should be taken into account that the CRC value of the new partial bitstreams does not correspond to its frame contents. Therefore its calculation is bypassed by setting the value 0xDEFC in the field corresponding to the CRC register at the end of the partial bitstream.

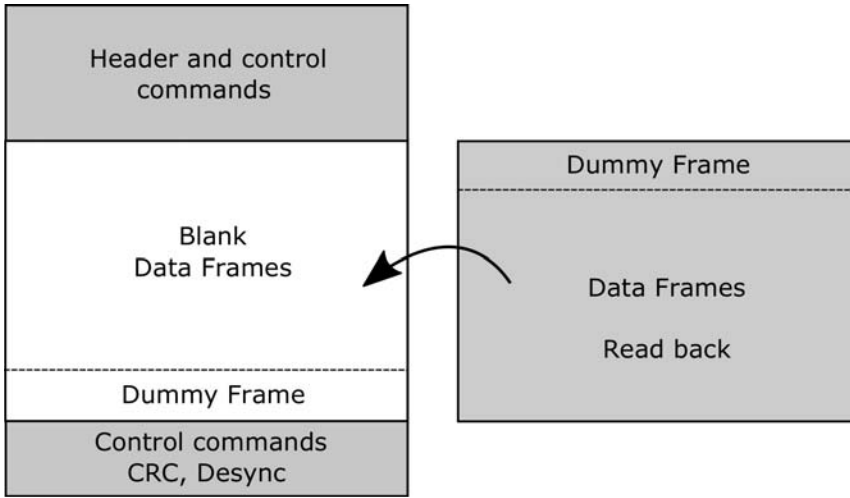


Figure 5.3 | Partial bitstream alternative generation

5.2.4 Faulty bitstreams for emulation

The generation of bitstreams to emulate SEUs in the configuration memory of the X-TMR domains is done by modifying the bits on the bitstream that configure the corresponding replicas of the circuit to evaluate. To this end we designed a SW to analyze such information and deduce what frames and bit inside them correspond to the DUT domains. This is done thanks to the knowledge of the architecture and how it is related to the frames in the bitstream. In other words, we can find the part of the bitstream that corresponds to any CLB region of the device. As we know the location of the DUT replicas we can deduce what exact frames of the full bitstream correspond to any of the domains and consequently we can manipulate the bits for each domain. Based on the number of CLB columns that the circuit to be evaluated requires, the SW computes the number of bits that such region requires to be configured. Similarly as the FAR and Nframes was calculated in the prior description, the number of bits is obtained using the equation 5.3:

$$Nbits = N_{CLB} * 36 * 1312 \tag{5.3}$$

For a circuit as the one that will be detailed in section 5.5, each domain occupies 4 CLB columns requiring 188928 bits. Considering the three domains, the total number of bits that can be considered for evaluation are 566784.

With this information the golden bitstream is used to generate the faulty bitstreams by modifying the bits that configure the DUT domains. To avoid a huge amount of generated bitstreams (one per fault for every bit flipped) we generate the faulty bitstream and perform the fault injection for that specific fault. The details of this approach will be explained in Section 5.4.

5.3 Developed Testbed

This section describes the constituent components of the architecture shown in Fig 5.4. These are required to implement the designs in a board equipped with a Virtex5 XC5VLX110T FPGA. VHDL is used to instantiate and control the elements as explained in the following.

- **FSM manager:** This component is responsible for arbitrating all the tasks related to the operation of the system. Its main functions involve controlling the FC_ICAP, sending the pattern inputs to the DUT and GOLD circuits, monitoring the FLAGS coming from the three DUT domains to account for faults and launch partial reconfiguration in case that any domain reports an error through its associated FLAG. It also controls the timers used to record diverse statistics such as the time required to detect a fault and correct a domain. The results for each applied fault are stored and sent through the UART module.
- **FC_ICAP:** The fast Fault Correction ICAP controller is the component that makes it possible to perform run-time reconfiguration. Thanks to this it is possible to access the configuration memory at run-time. The FC_ICAP is a reduced version of the AC_ICAP, designed to support frames readback and manipulation of partial bitstreams located in both flash and BRAM memories. As the recovery time is directly related to the time required to perform the partial reconfiguration tasks, the run-time reconfiguration controller should

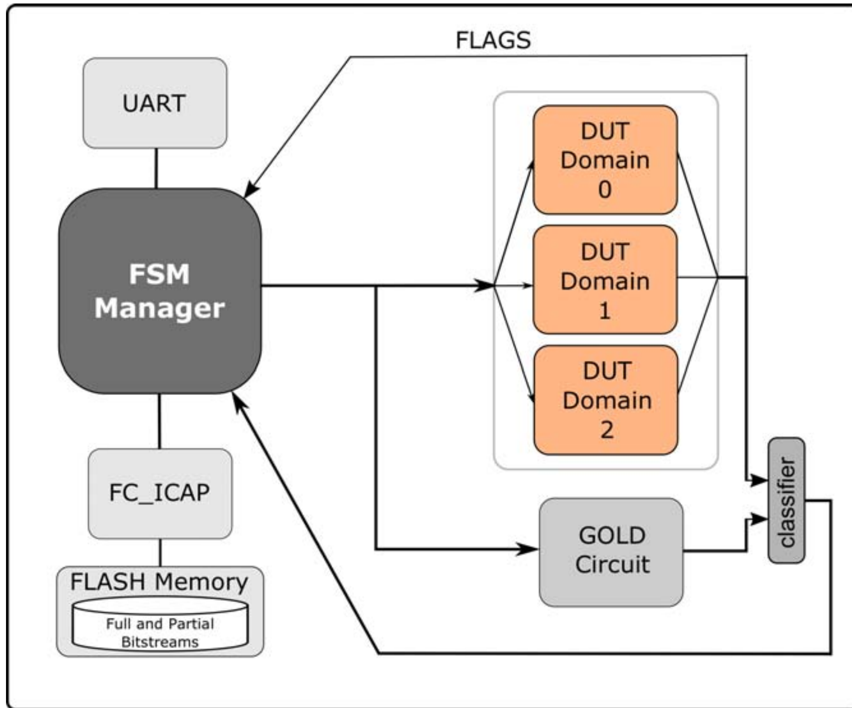


Figure 5.4 | Dynamically reconfigurable architecture

performs these tasks at maximum supported speed. In this regard, the available controller provided by Xilinx (XPS_HWICAP for Virtex5) does not meet these requirements as this is designed to be used along with an embedded processor, such as the MicroBlaze, and many of the run-time reconfiguration tasks are performed as software routines in the processor. This limits reaching the maximum supported configuration throughput and demands the presence of a processor to control it. Therefore an alternative ICAP controller was designed to speed-up the process and avoid the presence of the processor. The hardware-based run-time controller, depicted in Fig. 5.5, performs two main tasks: read frames and load partial bitstreams from BRAM and FLASH memories. These tasks are required to extract (read frames) the information that allows the partial bitstreams to be generated as explained in Section 5.2.3 and to correct faults in specific do-

mains by dynamically reconfiguring the faulty domains. Internally the FC_ICAP computes the number of words to read from the configuration memory.

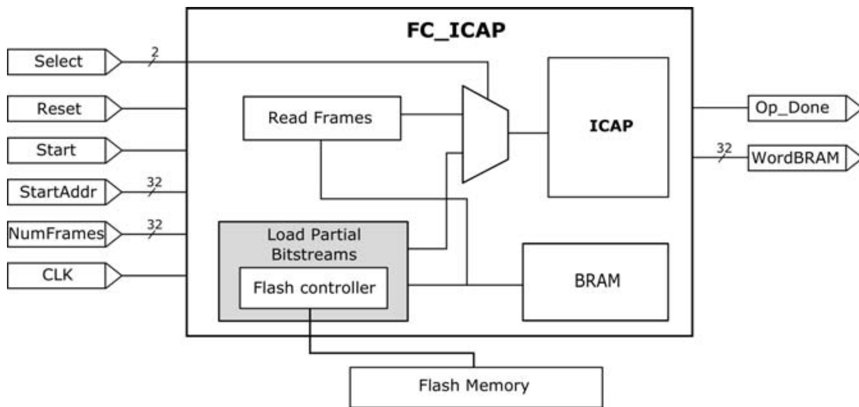


Figure 5.5 | Hardware-based ICAP controller

- **DUT and GOLD circuits:** The GOLD or original circuit to be tested is processed by the X-TMR tool that generates the triplicated version of the design under test (DUT). This X-TMR_DUT description is in EDIF format and it is processed by the SW to perform hierarchy formation, insert the XNOR-based error detectors reusing the LUTs where the majority voters reside and add the flag convergence using carry chains. The architecture to validate the presented CAD flow uses the GOLD circuit to perform online comparison of the outcomes of the DUT. The GOLD circuit produces the correct results that are compared to the produced by the DUT in the classifier component. To avoid synchronization issues both the golden and DUT circuits operates at a frequency lower than the maximum supported for the slower of them. In the case of the RISC5x[ref to opencores], that is the circuit used for this test, its frequency is set to 40 MHz. The RISC-based processor is considered for the tests as this offers the possibility to include different algorithms described in C and consequently to have a flexible testbed that allows for fault analysis at different levels. In this way we can analyze what effects the faults injected in the RISC hardware produce on the SW-based algorithm

running on it.

5.4 Emulation of CDEs

The evaluation of the presented approach requires a way to emulate CDEs. To this end we perform fault injection on the configuration bits specific to the circuits to evaluate. This can be a flat design confined to a known area, such as the X-TMR version or our modified version where each domain can be managed independently. We developed a SW to emulate MBUs (also SEUs) in the configuration memory of the DUT circuits according to the flow detailed in algorithm 5.1

Listing 5.1 | Fault Injection Algorithm

```
---------------------------------------------------
input: golden.bit
input: Region(s) to be tested (RANGE=SLICE_X_Y_)
input: N Number of faults to emulate
input: M = Number of MBUs (2, 3, 4,...)
output: UART report
output: Injector report
**** Fault Injection loop *****
1: for N {
2:  for M {
3:   {MBU_LOCs} = BaseAdd+rand()
4:  }
5: fin=fopen("golden.bit","rb")
6: fout=fopen("faulty.bit","wb")
7: for all bits{
8:  at MBU_LOCs addresses:
9:  faulty_bit = not(golden_bit) //flip bits
10: }
11: programFPGA(faulty.bit)
12: run_algorithm(1s)
13: {REP} = error_flag(DUT vs GOLD)
14: {REP} = FLAGS
15: {REP} = timer_values
16: if(FLAGS)
17:  DPR(faultyDomain)
```

```
18:  run_algorithm
16: UART_report(REP)
17: }
---------------------------------------------------
```

Once the DUT is instantiated and instrumented with the components described in section 5.3 the full bitsrtream is used to perform the fault injection experiments. This is the golden.bit input of the algorithm. The read back values obtained to generate the partial bitstreams, described in section 5.2.3, are also used to find what specific parts of the full bitstream contains the DUT configuration bits. This information is used by the fault injection software to define the location of the bits to be flipped. Therefore the physical location of a given area is translated to an address range of the bitstream. For instance, if one domain of the DUT that only requires CLBs is confined to the area Slice_X48Y120:Slice_X55Y139, the configuration bits will be located between the addresses 0xE4A12 and 0xEA651 of the full bitstream.

N is the number of MBU sets to emulate (number of multiple faults) and M the number of MBUs. For instance, N=1000 and M=2 will produce 2000 bits flipped, 2 for each one of the 1000 configurations to perform.

The position of the bitstream where the fault is injected is selected by a pseudo random number generation function. As we know the area where the domains of the DUT are implemented also as the sizes of such regions we can deduce the exact number of bits that correspond to these areas. So the PRNG function is limited by this maximum number of bits (Maxbits), and it generates a number between one and Maxbits that is used to compute the address of the bitstream where the fault(s) will be injected: MBU_LOCs.

The SW then reads the golden bitstream until the addresses match MBU_LOCs and once there the bits are flipped and stored in the faulty bitstream. The golden is never modified. As our focus is to analyze resilience against CDEs it is important to have a way to emulate and detect such type of effects. Thanks to the isolation of the domains made possible by the hierarchy formation phase we are able to place any domain in different regions of the FPGA. So it is possible to have the three domains one after the other in such a way similar to the X-TMR flat design or use different placements schemes to analyze its influence in the CDEs. The flags available for each

domain are used to easily detect errors in any of the circuit domains. These outcomes are considered when comparing the outcomes of the DUT and the GOLD circuit to signalize if CDEs have been detected. If the output of the circuit is erroneous as it does not match with the GOLD values and flags of different domains are activated it represents detection of CDEs.

5.5 Experimental Results

To evaluate the presented approach we chose the RISC5x processor [83]. This is a RISC-based processor compatible wit 12-bit PIC architecture. It allows applications described in C code to be compiled and adapted to be executed in the FPGA. A set of typical software applications [84] are selected to run on the RISC5x processor. Every algorithm is coded and compiled using the MPLAB IDE [85] with the CC5X C compiler which generates HEX files. These are transformed into VHDL to be stored in a ROM memory block that contains the program code.

As shown in table 5.2 the test bed platform can operate to frequencies higher than 60 MHz but for the experiments the operating frequency of the RISC5x processor was fixed to 40 MHz.

For the experiments a total of 3000 runs were done. This value represents the number of iterations the experiments perform but in every iteration different number of bits are flipped. For instance in the case of two MBUs two bits are flipped in every iteration. Table 5.1 summarizes the experiments for the three versions of the circuit and for 2, 3 and 4 MBUs.

As the X-TMR version is a flat design where the elements of the different domains are mixed, it is more susceptible to CDE. For such a design there is not detectors implemented inside the circuit and that is why the CDE detected columns are zero.

The second row of table 5.1 corresponds to the X-TMR design modified according to the description presented above where every domain is identified and the error flags are included for every domain. In this case the three domains remain one after the other occupying the same region as the X-TMR base design. Such domains isolation improve CDE resilience as now

Table 5.1 CDEs for bubble sort

Circuit	CDE 2MBU (%)	CDE detected (%)	CDE 3MBU (%)	CDE detected (%)	CDE 4MBU (%)	CDE detected (%)
X-TMR	0.633	0	2.133	0	2.567	0
Modif X-TMR single area	0.633	0	0.724	0	0.833	4
Modif X-TMR three areas	0.567	100	0.667	100	1.633	100

the probability MBU affect different domains is lower. But the flags that signalize error on the domains are rarely activated.

The third row represents exactly the same circuit of second row but now the three domains are placed in different areas of the FPGA. For this specific case the domains are constrained to different HCLK regions. Only with this placement modification the flags for domains error are always activated when CDE are present. In addition the CDE resilience is improved in this way.

According to table 5.2 T_{Error_prop} is the error propagation time that corresponds to the average time until injected faults manifest as erroneous outputs of the circuit. It can be noticed that such time is improved by isolating the domains and constraint them to regions of the FPGA with certain distances. In the test case the three domains corresponding to the third row of table 5.2 were placed in different HCLK regions.

In a similar way T_{Flags_detect} is the average time until flags of different domains are activated. For the bubble sort algorithm these flags are mainly activated when we use the placement with the three domains at different regions. In such a case the carry-chain-based flags are activated within 5 ns after applying the faults.

Table 5.2 Timing bubble sort

Circuit	Maximum Frequency (MHz)	T_{Error_prop} (ms) MBUs:			T_{Flags_detect} (ns)
		2	3	4	
X-TMR	62.231	5.769	4.327	3.876	–
Modif X-TMR single area	66.300	15.081	12.549	10.430	–
Modif X-TMR three areas	60.727	38.781	37.449	23.543	5

As described in Section 5.2.3 the presented flow allows partial bitstreams to be generated for each domain. For the RISC5x circuit very domain occupies 4 CLB columns giving as a result partial bitstreams of 36 KB. The recovery time represent the time required to perform the DPR with the gold configuration values. For this test circuit the partial bitstreams can be located in BRAM memory and in consequence the FC_ICAP performs the DPR of each domain in 92.266 μs.

5.6 Summary

In this chapter we present an efficient way to improve resilience of TMR circuits against CDE by using an alternative CAD flow that includes flag insertion on individual domains also a post-implemented partial bitstream generation of each domain. Compared to X-TMR, our approach allows for detection of CDE and makes it possible to correct the domains thanks to the partial bitstreams available for them.

Conclusions | 6

In this dissertation we focused on using Dynamic Partial Reconfiguration in the design and evaluation of critical systems. To this end we designed the AC_ICAP, a new ICAP controller verified in Virtex-5 and Kintex7 FPGAs. It is able to load partial bitstreams, read and write frames, and also modify any LUT in the FPGA, in this last case without the need for pre-generated partial bitstreams. This fine grain DPR was possible thanks to reverse engineering on the bitstream and analysis of the architecture of the FPGA. The controller was adapted to be easily included in systems with embedded processors using the PLB, FSL, and AXI links.

Reconfiguration speed analysis of the processor-independent version presents improvement of more than 380 times in run-time reconfiguration of LUTs compared to XPS_HWICAP functions for Virtex-5 FPGAs. As the controller is fully implemented in hardware, it obviously requires more resources, but in any case it occupies more than 5% of the available elements on the XC5VLX110T device. Therefore, the AC_ICAP offers a complete high speed solution to perform diverse Dynamic Partial Reconfiguration tasks with acceptable FPGA footprint.

This DPR knowledge and its low level relationship with the device architecture is used in the chapter 4 where we presented a methodology to emulate ASIC permanent faults using FPGAs. A novel hardware fault emulation platform utilizing a semi-custom CAD flow is presented that can inject fault in run-time using dynamic partial reconfiguration. The flow utilizes a custom technology mapping for directly converting the post layout gate-level

net-list into a LUT level net-list. This known mapping of gates to LUTs enables us to develop equivalent fault dictionaries from the ASIC fault list to FPGA faults representation. The approach is flexible enough to generate fault dictionaries in different formats compatible with partial reconfiguration requirements which results in significant variation in achievable fault injection speedups. The methodology avoids drastic changes to the net-lists therefore, does not need lengthy re-compilation times during the fault emulation process. Furthermore, our experimental results demonstrate a significant speed up for fault emulation compared to software based fault simulation. In the case of using the PLB_AC_ICAP the speed up is of 829 times compared to fastest XPS_HWICAP alternative.

The approach presented in chapter 5 proposes a TMR architecture that exploits the fracturable nature of Look Up Tables for simultaneously mapping of majority-voting and error detection at the granularity of TMR domains. An associated CAD flow was developed for partial reconfiguration of TMR domains incorporating changes to the technology mapping, placement and bitstream generation phases. In doing this we achieved better resilience to cross domain errors with zero hardware overhead and the three domains can be independently reconfigured to correct faults improving MTTR compared to full circuit configuration. As the alternative CAD flow includes flag insertion on individual domains also a post-implemented partial bitstream generation for each domain, our approach allows for detection of CDE and makes it possible to correct the domains thanks to the partial bitstreams available for them. For the test circuit analyzed the recovery time of each domain corresponds to the DPR of the domains with the partial bitstreams located in BRAM memory. This is 92.266 μs.

As future work, we plan to extend the AC_ICAP with a DDR controller to speed up the reconfiguration tasks when these are based on precomputed partial bitstreams not able to be copied into BRAM due to their sizes. In addition we will use the presented mechanisms for error detection and flag convergence described in chapter 5 to improve the AC_ICAP controller as this is a critical component of the system. In the worst case scenario a full reconfiguration could be launched when an error FLAG is activated on the logic that is controlling the ICAP port.

We also plan to extend the methodology presented in chapter 4 to automatic test pattern generation and fault behavior analysis for a wide set of permanent and dynamic fault models.

Publications

- Luis Andres Cardona and Carles Ferrer, "AC_ICAP: A Flexible High Speed ICAP Controller," International Journal of Reconfigurable Computing, vol. 2015, Article ID 314358, 15 pages, 2015. doi:10.1155/2015/314358

- L.A. Cardona, B. Lorente, C. Ferrer, "Partial crypto-reconfiguration of nodes based on FPGA for WSN", Proceedings - International Carnahan Conference on Security Technology, vol. 2014-October, no. October, 2014.

- L.A. Cardona, Y. Guo, C. Ferrer, "Dependability Improvement by Partial Reconfiguration in SRAM-Based FPGAs for Critical Applications". Poster, Work-in progress session at 50th Design Automation Conference, Austin,USA, June 2013.

- L.A. Cardona, Y. Guo, C. Ferrer, "Fault tolerant architectures by partial reconfiguration", Proceedings of SPIE - The International Society for Optical Engineering, vol. 8764, 2013.

- L.A. Cardona, S. de la Fe, B. Lorente, S. Villar, C. Ferrer "Secure key management in low power wireless sensor networks", Proceedings - International Carnahan Conference on Security Technology, 2013.

- L.A. Cardona, J. Agrawal, Y. Guo, J. Oliver, C. Ferrer, "Performance-area improvement by partial reconfiguration for an aerospace remote sensing application", Proceedings - 2011 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2011, pp. 497-500, 2011.

- Andres Cardona, Guo Yi, Carles Ferrer, "Partial Reconfiguration of a peripheral in an FPGA-based SoC to analyse performance-area behaviour", Proceedings of SPIE - The International Society for Optical Engineering, vol. 8067, 2011.

References

[1] Xilinx. *LogiCORE IP XPS HWICAP (v5.01a) DS586*, 2011.

[2] D. Koch. *Partial Reconfiguration on FPGAs - Architectures, Tools and Applications*. Springer, 2012. ISBN 978-1-4614-1224-3.

[3] J. Tarrillo, F. A. Escobar, F. L. Kastensmidt and C. Valderrama. *Dynamic partial reconfiguration manager*. In *Circuits and Systems (LASCAS), 2014 IEEE 5th Latin American Symposium on*, pages 1–4. feb 2014. doi:10.1109/LASCAS.2014.6820293.

[4] A. Ebrahim, K. Benkrid, X. Iturbe and C. Hong. *A novel high-performance fault-tolerant ICAP controller*. In *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, pages 259–263. 2012. doi:10.1109/AHS.2012.6268660.

[5] L. Sterpone and M. Violante. *A New Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs*. Nuclear Science, IEEE Transactions on, 54(4):965–970, 2007. ISSN 0018-9499. doi:10.1109/TNS.2007.904080.

[6] M. S. Abdelfattah, L. Bauer, C. Braun, M. E. Imhof, M. A. Kochte, H. Zhang, J. Henkel and H. Wunderlich. *Transparent Structural Online Test for Reconfigurable Systems*. In *On-Line Testing Symposium*

(IOLTS), 2012 IEEE 18th International, pages 37–42. 2012. doi: 10.1109/IOLTS.2012.6313838.

[7] L. Sterpone and M. Violante. *A New Algorithm for the Analysis of the MCUs Sensitiveness of TMR Architectures in SRAM-Based FPGAs.* IEEE Transactions on Nuclear Science, 55(4):2019–2027, Aug 2008. ISSN 0018-9499. doi:10.1109/TNS.2008.2001858.

[8] V. Dumitriu, L. Kirischian and V. Kirischian. *Run-Time Recovery Mechanism for Transient and Permanent Hardware Faults Based on Distributed, Self-organized Dynamic Partially Reconfigurable Systems.* Computers, IEEE Transactions on, PP(99):1, 2015. ISSN 0018-9340. doi:10.1109/TC.2015.2506558.

[9] L. Sterpone and M. Violante. *Analysis of the robustness of the TMR architecture in SRAM-based FPGAs.* Nuclear Science, IEEE Transactions on, 52(5):1545–1549, 2005. ISSN 0018-9499. doi:10.1109/TNS.2005.856543.

[10] L. Sterpone and A. Ullah. *On the optimal reconfiguration times for TMR circuits on SRAM based FPGAs.* In *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, pages 9–14. 2013. doi:10.1109/AHS.2013.6604220.

[11] A. Sari, M. Psarakis and D. Gizopoulos. *Combining checkpointing and scrubbing in FPGA-based real-time systems.* In *VLSI Test Symposium (VTS), 2013 IEEE 31st*, pages 1–6. 2013. ISSN 1093-0167. doi:10.1109/VTS.2013.6548910.

[12] H. Baig, J.-A. Lee and Z. A. Siddiqui. *A Low-Overhead Multiple-SEU Mitigation Approach for SRAM-based FPGAs with Increased Reliability.* Nuclear Science, IEEE Transactions on, PP(99):1, 2014. ISSN 0018-9499. doi:10.1109/TNS.2014.2315432.

[13] F. Kastensmidt, L. Carro and R. Reis. *Fault-tolerance techniques for SRAM-based FPGAs*, volume 32. 2006. ISBN 0387310681. doi:10.1007/978-0-387-31069-5.

[14] C. Bolchini, A. Miele and C. Sandionigi. *A Novel Design Methodology for Implementing Reliability-Aware Systems on SRAM-Based FPGAs.*

Computers, IEEE Transactions on, 60(12):1744–1758, 2011. ISSN 0018-9340. doi:10.1109/TC.2010.281.

[15] M. Psarakis, A. Vavousis, C. Bolchini and A. Miele. *Design and implementation of a self-healing processor on SRAM-based FPGAs.* In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014 IEEE International Symposium on*, pages 165–170. oct 2014. doi:10.1109/DFT.2014.6962076.

[16] F. Ghaffari, F. Sahraoui, M. El Amine Benkhelifa, B. Granado, M. A. Kacou and O. Romain. *Fast SRAM-FPGA fault injection platform based on dynamic partial reconfiguration.* In *Microelectronics (ICM), 2014 26th International Conference on*, pages 144–147. 2014. doi:10.1109/ICM.2014.7071827.

[17] G. L. Nazar, P. Rech, C. Frost and L. Carro. *Radiation and Fault Injection Testing of a Fine-Grained Error Detection Technique for FPGAs.* Nuclear Science, IEEE Transactions on, 60(4):2742–2749, 2013. ISSN 0018-9499. doi:10.1109/TNS.2013.2261319.

[18] S. Di Carlo, G. Gambardella, P. Prinetto, F. Reichenbach, T. Lokstad and G. Rafiq. *On enhancing fault injection's capabilities and performances for safety critical systems.* In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 583–590. Aug 2014. doi:10.1109/DSD.2014.12.

[19] Keysight Technologies. *M9451A-DPD PXIe Measurement Accelerator*, jun 2015.

[20] Xilinx. *7 Series FPGAs Overview DS180 (v1.16.1)*, 2014.

[21] Altera. *A New FPGA Architecture and Leading-Edge FinFET Process Technology Promise to Meet NextGeneration System Requirements WP-01220-1.1*, 2015.

[22] L. A. Cardona, J. Agrawal, Y. Guo, J. Oliver and C. Ferrer. *Performance-Area Improvement by Partial Reconfiguration for an Aerospace Remote Sensing Application.* In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 497–500. nov 2011. doi:10.1109/ReConFig.2011.69.

[23] C. Claus, R. Ahmed, F. Altenried and W. Stechele. *Towards Rapid Dynamic Partial Reconfiguration in Video-Based Driver Assistance Systems*. In P. Sirisuk, F. Morgan, T. El-Ghazawi and H. Amano, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 5992 of *Lecture Notes in Computer Science*, pages 55–67. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-12132-6. doi:10.1007/978-3-642-12133-3{_}8.

[24] S. Bhandari, S. Subbaraman, S. Pujari, F. Cancare, F. Bruschi, M. D. Santambrogio and P. R. Grassi. *High Speed Dynamic Partial Reconfiguration for Real Time Multimedia Signal Processing*. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 319–326. 2012. doi:10.1109/DSD.2012.74.

[25] Xilinx. *Indirect Programming of BPI PROMs with Virtex-5 FPGAs XAPP973 (v1.4)*, mar 2010.

[26] Xilinx. *Virtex-5 FPGA Data Sheet: DC and Switching Characteristics DS202 (v5.3)*, may 2010.

[27] IBM. *128-Bit Processor Local Bus Architecture Specifications*, may 2007.

[28] K. Glette and P. Kaufmann. *Lookup table partial reconfiguration for an evolvable hardware classifier system*. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 1706–1713. IEEE, jul 2014. ISBN 978-1-4799-1488-3. doi:10.1109/CEC.2014.6900503.

[29] M. Liu, W. Kuehn, Z. Lu and A. Jantsch. *Run-time Partial Reconfiguration speed investigation and architectural design space exploration*. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 498–502. 2009. ISSN 1946-1488. doi:10.1109/FPL.2009.5272463.

[30] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hubner and J. Becker. *A multi-platform controller allowing for maximum Dynamic Partial Reconfiguration throughput*. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 535–538. 2008. doi:10.1109/FPL.2008.4630002.

[31] M. Hubner, D. Gohringer, J. Noguera and J. Becker. *Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs*. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. 2010. doi:10.1109/IPDPSW.2010.5470736.

[32] S. Lamonnier, M. Thoris and M. Ambielle. *Accelerate Partial Reconfiguration with a 100% Hardware Solution*. Xcell journal, (79):44–49, 2012.

[33] V. Lai and O. Diessel. *ICAP-I: A reusable interface for the internal reconfiguration of Xilinx FPGAs*. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 357–360. 2009. doi:10.1109/FPT.2009.5377616.

[34] M. Straka, J. Kastil and Z. Kotasek. *Generic partial dynamic reconfiguration controller for fault tolerant designs based on FPGA*. In *NORCHIP, 2010*, pages 1–4. nov 2010. doi:10.1109/NORCHIP.2010.5669477.

[35] S. G. Hansen, D. Koch and J. Torresen. *High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro*. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 174–180. 2011. ISSN 1530-2075. doi:10.1109/IPDPS.2011.139.

[36] A. Ebrahim, T. Arslan and X. Iturbe. *On enhancing the reliability of internal configuration controllers in FPGAs*. In *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, pages 83–88. 2014. doi:10.1109/AHS.2014.6880162.

[37] J. Heiner, N. Collins and M. Wirthlin. *Fault Tolerant ICAP Controller for High-Reliable Internal Scrubbing*. In *Aerospace Conference, 2008 IEEE*, pages 1–10. 2008. ISSN 1095-323X. doi:10.1109/AERO.2008.4526471.

[38] A. Ebrahim, K. Benkrid, X. Iturbe and C. Hong. *Multiple-clone configuration of relocatable partial bitstreams in Xilinx Virtex FPGAs*. In *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, pages 178–183. 2013. doi:10.1109/AHS.2013.6604243.

[39] U. Legat, A. Biasizzo and F. Novak. *SEU Recovery Mechanism for SRAM-Based FPGAs.* Nuclear Science, IEEE Transactions on, 59(5):2562–2571, 2012. ISSN 0018-9499. doi:10.1109/TNS.2012.2211617.

[40] C. Schuck, B. Haetzer and J. Becker. *An Interface for a Decentralized 2D Reconfiguration on Xilinx Virtex-FPGAs for Organic Computing.* International Journal of Reconfigurable Computing, 2009. doi:10.1155/2009/273791. URL `http://dx.doi.org/10.1155/2009/273791`.

[41] L. Burgun, F. Reblewski, G. Fenelon, J. Barbier and O. Lepape. *Serial fault emulation.* In *Design Automation Conference Proceedings 1996, 33rd*, pages 801–806. jun 1996. ISSN 0738-100X. doi:10.1109/DAC.1996.545681.

[42] E. Sanchez, L. Sterpone and A. Ullah. *Effective emulation of permanent faults in ASICs through dynamically reconfigurable FPGAs.* In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6. sep 2014. doi:10.1109/FPL.2014.6927478.

[43] K.-T. Cheng, S.-Y. Huang and W.-J. Dai. *Fault emulation: a new approach to fault grading.* In *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, pages 681–686. nov 1995. ISSN 1092-3152. doi:10.1109/ICCAD.1995.480203.

[44] K.-T. Cheng, S.-Y. Huang and W.-J. Dai. *Fault emulation: A new methodology for fault grading.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 18(10):1487–1495, oct 1999. ISSN 0278-0070. doi:10.1109/43.790625.

[45] J.-H. Hong, S.-A. Hwang and C.-W. Wu. *An FPGA-based hardware emulator for fast fault emulation.* In *Circuits and Systems, 1996., IEEE 39th Midwest symposium on*, volume 1, pages 345–348 vol.1. 1996. doi:10.1109/MWSCAS.1996.594168.

[46] S.-A. Hwang, J.-H. Hong and C.-W. Wu. *Sequential circuit fault simulation using logic emulation.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 17(8):724–736, 1998. ISSN 0278-0070. doi:10.1109/43.712103.

[47] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Reorda and M. Violante. *An FPGA-Based Approach for Speeding-Up Fault Injection Campaigns on Safety-Critical Circuits*. Journal of Electronic Testing, 18(3):261–271, 2002. ISSN 0923-8174. doi:10.1023/A:1015079004512. URL `http://dx.doi.org/10.1023/A{%}3A1015079004512`.

[48] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcia and L. Entrena-Arrontes. *An autonomous FPGA-based emulation system for fast fault tolerant evaluation*. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 397–402. 2005. doi:10.1109/FPL.2005.1515754.

[49] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcia and L. Entrena. *Autonomous Fault Emulation: A New FPGA-Based Acceleration System for Hardness Evaluation*. Nuclear Science, IEEE Transactions on, 54(1):252–261, feb 2007. ISSN 0018-9499. doi:10.1109/TNS.2006.889115.

[50] Xilinx. *JBits 2.8 SDK for Virtex.*, 2009.

[51] L. Antoni, R. Leveugle and B. Feher. *Using run-time reconfiguration for fault injection applications*. In *Instrumentation and Measurement Technology Conference, 2001. IMTC 2001. Proceedings of the 18th IEEE*, volume 3, pages 1773–1777 vol.3. 2001. ISSN 1091-5281. doi:10.1109/IMTC.2001.929505.

[52] L. Antoni, R. Leveugle and B. Feher. *Using run-time reconfiguration for fault injection in hardware prototypes*. In *Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings. 17th IEEE International Symposium on*, pages 245–253. 2002. ISSN 1550-5774. doi:10.1109/DFTVS.2002.1173521.

[53] L. Antoni, R. Leveugle and B. Feher. *Using run-time reconfiguration for fault injection applications*. Instrumentation and Measurement, IEEE Transactions on, 52(5):1468–1473, oct 2003. ISSN 0018-9456. doi:10.1109/TIM.2003.817144.

[54] A. Parreira, J. P. Teixeira, A. Pantelimon, M. B. Santos and J. T. de Sousa. *Fault Simulation Using Partially Reconfigurable Hard-*

ware. In P. Y. K. Cheung and G. Constantinides, editors, *Field Programmable Logic and Application*, volume 2778 of *Lecture Notes in Computer Science*, pages 839–848. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40822-2. doi:10.1007/978-3-540-45234-8{_}81. URL http://dx.doi.org/10.1007/978-3-540-45234-8{_}81.

[55] A. Parreira, J. P. Teixeira and M. Santos. *A Novel Approach to FPGA-Based Hardware Fault Modeling and Simulation.* In *Proc. of the Design and Diagnostics of Electronic Circuits and Syst. Workshop*, pages 17–24. 2003.

[56] D. de Andres, J. C. Ruiz, D. Gil and P. Gil. *Fast Emulation of Permanent Faults in VLSI Systems.* In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6. 2006. doi:10.1109/FPL.2006.311221.

[57] M. Grosso, H. Guzman-Miranda and M. A. Aguirre. *Exploiting Fault Model Correlations to Accelerate SEU Sensitivity Assessment.* Industrial Informatics, IEEE Transactions on, 9(1):142–148, feb 2013. ISSN 1551-3203. doi:10.1109/TII.2012.2226096.

[58] E. Normand. *Single event upset at ground level.* Nuclear Science, IEEE Transactions on, 43(6):2742–2750, 1996. ISSN 0018-9499. doi:10.1109/23.556861.

[59] F. Siegle, T. Vladimirova, J. Ilstad and O. Emam. *Mitigation of Radiation Effects in SRAM-Based FPGAs for Space Applications.* ACM Comput. Surv., 47(2):37:1–37:34, January 2015. ISSN 0360-0300. doi:10.1145/2671181. URL http://doi.acm.org/10.1145/2671181.

[60] M. Straka, J. Kastil, Z. Kotasek and L. Miculka. *Fault tolerant system design and SEU injection based testing.* Microprocessors and Microsystems, 37(2):155–173, 2013. ISSN 0141-9331. doi:http://dx.doi.org/10.1016/j.micpro.2012.09.006. URL http://www.sciencedirect.com/science/article/pii/S0141933112001688.

[61] C. Bolchini, A. Miele and C. Sandionigi. *Autonomous Fault-Tolerant Systems onto SRAM-based FPGA Platforms.* Journal of

Electronic Testing, 29(6):779–793, 2013. ISSN 0923-8174. doi:10.1007/s10836-013-5418-4. URL `http://dx.doi.org/10.1007/s10836-013-5418-4`.

[62] C. Pilotto, J. R. Azambuja and F. L. Kastensmidt. *Synchronizing triple modular redundant designs in dynamic partial reconfiguration applications.* In *Proceedings of the 21st annual symposium on Integrated circuits and system design*, SBCCI '08, pages 199–204. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-231-3. doi:10.1145/1404371.1404426. URL `http://doi.acm.org/10.1145/1404371.1404426`.

[63] F. L. Kastensmidt, L. Sterpone, L. Carro and M. S. Reorda. *On the optimal design of triple modular redundancy logic for SRAM-based FPGAs.* In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 1290–1295 Vol. 2. 2005. ISSN 1530-1591. doi:10.1109/DATE.2005.229.

[64] k. Chapman and L. Jones. *Xilinx XAPP864: SEU strategies for virtex-5 devices (v1.0.1)*, mar 2009.

[65] Xilinx. *Soft Error Mitigation Controller v4.1 Product Guide*, sep 2015.

[66] J. Tonfat, F. L. Kastensmidt, P. Rech, R. Reis and H. M. Quinn. *Analyzing the Effectiveness of a Frame-Level Redundancy Scrubbing Technique for SRAM-based FPGAs.* IEEE Transactions on Nuclear Science, 62(6):3080–3087, Dec 2015. ISSN 0018-9499. doi:10.1109/TNS.2015.2489601.

[67] G. L. Nazar. *Improving FPGA repair under real-time constraints.* Microelectronics Reliability, 55(7):1109–1119, jun 2015. ISSN 00262714. doi:10.1016/j.microrel.2015.04.003. URL `http://www.sciencedirect.com/science/article/pii/S0026271415000918`.

[68] G. Nazar, L. Pereira Santos and L. Carro. *Fine-grained fast field-programmable gate array scrubbing.* Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 23(5):893–904, May 2015. ISSN 1063-8210. doi:10.1109/TVLSI.2014.2330742.

[69] H. Quinn, P. Graham, J. Krone, M. Caffrey and S. Rezgui. *Radiation-induced multi-bit upsets in sram-based fpgas.* IEEE Transactions on

Nuclear Science, 52(6):2455–2461, Dec 2005. ISSN 0018-9499. doi:10.1109/TNS.2005.860742.

[70] H. Quinn, K. Morgan, P. Graham, J. Krone, M. Caffrey and K. Lundgreen. *Domain Crossing Errors: Limitations on Single Device Triple-Modular Redundancy Circuits in Xilinx FPGAs.* Nuclear Science, IEEE Transactions on, 54(6):2037–2043, 2007. ISSN 0018-9499. doi:10.1109/TNS.2007.910870.

[71] A. Ullah and L. Sterpone. *Recovery Time and Fault Tolerance Improvement for Circuits mapped on SRAM-based FPGAs.* Journal of Electronic Testing, 30(4):425–442, 2014. ISSN 0923-8174. doi:10.1007/s10836-014-5463-7. URL `http://dx.doi.org/10.1007/s10836-014-5463-7`.

[72] Xilinx. *Partial Reconfiguration User Guide UG702 (v14.7)*, oct 2013.

[73] Xilinx. *Command Line Tools User Guide UG628 (v 14.7)*, oct 2013.

[74] Xilinx. *Virtex-5 FPGA Configuration Guide UG191 (v3.11)*, oct 2012.

[75] Xilinx. *ChipScope Pro Software and Cores*, 2012.

[76] Xilinx. *Xilinx Kintex-7 FPGA KC705 Evaluation Kit*, 2015.

[77] Xilinx. *AXI HWICAP v3.0*, 2015.

[78] J. Cong, C. Wu and Y. Ding. *Cut Ranking and Pruning: Enabling a General and Efficient FPGA Mapping Solution.* In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, FPGA '99, pages 29–35. ACM, New York, NY, USA, 1999. ISBN 1-58113-088-0. doi:10.1145/296399.296425. URL `http://doi.acm.org/10.1145/296399.296425`.

[79] R. Francis, J. Rose and Z. Vranesic. *Chortle-crf: Fast Technology Mapping for Lookup Table-based FPGAs.* In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, DAC '91, pages 227–233. ACM, New York, NY, USA, 1991. ISBN 0-89791-395-7. doi:10.1145/127601.127670. URL `http://doi.acm.org/10.1145/127601.127670`.

[80] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas and M. French. *Torc: Towards an Open-source Tool Flow.* In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 41–44. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0554-9. doi:10.1145/1950413.1950425. URL `http://doi.acm.org/10.1145/1950413.1950425`.

[81] A. H. Farrahi and M. Sarrafzadeh. *Complexity of the lookup-table minimization problem for FPGA technology mapping.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 13(11):1319–1332, nov 1994. ISSN 0278-0070. doi:10.1109/43.329262.

[82] J. Cong and Y. Ding. *On area/depth trade-off in LUT-based FPGA technology mapping.* Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2(2):137–148, 1994. ISSN 1063-8210. doi:10.1109/92.285741.

[83] Opencores. *RISC5x processor*, nov 2015. URL `http://opencores.org`.

[84] H. Quinn, W. H. Robinson, P. Rech, M. Aguirre, A. Barnard, M. Desogus, L. Entrena, M. Garcia-Valderas, S. M. Guertin, D. Kaeli, F. L. Kastensmidt, B. T. Kiddie, A. Sanchez-Clemente, M. S. Reorda, L. Sterpone and M. Wirthlin. *Using Benchmarks for Radiation Testing of Microprocessors and FPGAs.* Nuclear Science, IEEE Transactions on, 62(6):2547–2554, 2015. ISSN 0018-9499. doi:10.1109/TNS.2015.2498313.

[85] Microchip. *Microchip MPLAB IDE*, nov 2015. URL `http://www.microchip.com/MPLAB`.