eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Bilboko Ingeniaritza Eskola

Teknologia Elektronikoa Saila

DOCTORAL THESIS.

Contributions to the Fault Tolerance of Soft-Core Processors Implemented in SRAM-based FPGA Systems

Author: Julen Gomez-Cornejo Barrena

Supervisor: Dr. Aitzol Zuloaga

Bilbao, June 2018

Eskerrak - Thanks

Lan hau aurrera atera ahal izan dut inguruan izan dudan jendearen laguntza eta babesari esker. Lehenik, nire zuzendaria izan den Aitzol Zuloaga eskertu nahiko nuke. Era berean, lan honetan eragin zuzena izan duten Uli Kretzschmar eta Igor Villalta lankideei esker berezia luzatu nahi nieke. Bestalde, Iraide López eskertu nahi dut dokumentuaren portadaren diseinuarekin laguntzeagatik.

Ezin ahaztu APERT taldeko parte diren edo izan diren gainontzeko lankide eta lagunak, esker bereziak: Iñigo Kortabarria, Iñigo Martínez de Alegría, Jon Andreu, Carlos Cuadrado, José Luis Martín, Armando Astarloa, Jésus Lázaro, Jaime Jiménez, Unai Bidarte, Edorta Ibarra, Naiara Moreira, Víctor López, Ángel Pérez, Asier Matallana, Itxaso Aranzabal, Oier Oñederra, Estafania Planas, Markel Fernández, David Cabezuelo, Iker Aretxabaleta eta Endika Robles. Baita ere, Javier Del Ser, Roberto Fernández eta Leire Lopez-i eskeinitako laguntza bihotzez eskertu nahi nieke.

I also would like to thank all the kind people from the Computer Architecture and Embedded Systems research group for making me feel like home during the stay at TU Ilmenau, especially to Bernd Däne.

Ezin ahaztu, nire ondoan egon diren eta jasan nauten familia, lagunak eta Ufa. Eta bereziki eskerrak zuri ama, eredu zarelako.

Bukatzeko, UPV/EHU erakundea eta euskal gizartea eskertu nahi nituzke nire doktoretza aurrera eramateko baliabideak jartzeagatik.

Bilbon, 2018ko Ekainean.

Laburpena

Zirkuitu elektronikoak integrazio-maila apartetara iritsi dira diseinu eta ekoizpen tekniken garapenari esker. Hau dela eta, gaur egun posible da sistema osatu eta konplexuak gauzatzea gailu bakar batean. Kontzeptu honi SoC (System-on-Chip) deritzo. FPGA (Field Programmable Gate Array) teknologia da sistema horiek gauzatzeko euskarri interesgarrienetakoa, batez ere eskeintzen duen malgutasunari esker, bere inguruan ematen ari diren etengabeko ikerkuntza berrien garrantzia ahaztu gabe.

FPGAek duten konfigurazio ahalmenari esker, funtzio ezberdinetarako programatuak izateko aukera eskeintzen dute. SRAM eta flash teknologietan oinarritutako FPGA batzuek eskeitzen duten errekonfigurazio partzial dinamikoaren gaitasuna batuz gero, potentzial ikaragarrizko teknologia dela baiezta daiteke. Errekonfigurazio partzial dinamikoari esker, posiblea da FPGA baten matrizean konfiguratutako zirkuitoaren zati bakar bat aldatzea, bitartean gainontzeko zirkuitu-atalak funtzionatzen jarraitu dezaketelarik. Era horretan, hainbat diseinu partzial erabiltzeko aukera dago, FPGAaren baliabideen erabilera murriztuz.

Erradiazio kosmikotik datozen partikula energetikoekiko duten sentikortasuna da SRAM eta flash teknologietan oinarritutako gailuen erabilerak suposatzen duen arazo garrantzitsuenetakoa bat. Horien eraginez, hainbat arazo gerta daitezke, arazo arbuiagarrigarrietatik hasita, arazo benetan larrietara iritsi arte. Izan ere, gizakiak arriskuan jar daitezke alor konkretu batzuen sistemetan hutsegiteak izanez gero, esaterako: trenbide garraioetan, automozioan edo egiaztatze industrialean.

Hutsegite horiek sor ditzaketen arazoek hutsegite-tolerantzia kontzeptuaren garrantzia indartzen dute. Sistema digitaleei atxikitutako ezaugarria da hutsegite-tolerantzia. Horren bidez, funtzionamenduan kalitate maila jakin bat bermatzen da akatsen aurrean. Horrela, sistema hariek hutsegiteen efektuak jasan behar dituzte aldioro, funtzionamendu egokia mantenduz. Helburu hori lortzeko gogortze

teknikak atxikitzen zaizkie, hori da, erredundantzia software edota hardwarean oinarritutako teknikak, esate baterako.

Bestalde, eguneroko bizitzan erabiltzen diren produktu elektroniko gehienek sistema txertatuak izaten dituzte beraien diseinuetan. Sistema txertatu horien baitan prozesadore bat edo gehiago izatea da ohikoena. Sistemen konplexutasun-mailak gora egiten duen heinean, oso interresgarria suertatzen da aurrediseinatutako eta frogatutako sistema elektronikoen erabilera. IP core (Intellectual Property cores) bezala ezagutzen dira sistema hauek. Bereziki, azken urteotan zabaldu da IP core bidezko prozesadoreen (soft-core prozesadore bezala ezagunak) erabilerak garrantzia irabazi du.

Lan honetan, SRAM motako FPGA teknlogietan implementatutako soft-core prozesadoreen hutsegite-tolerantzia bermatzeko tekniken inguruan ikerketa burutu da. Era berean, prozesadore horien hutsegiteei aurre egiteko gogortze teknika berriak proposatzen dira. Hala nola, lockstep metodologia berriak, TMR implementazioetan moduluen sinkronizaziorako teknikak, kaltetutako interfazedun memorien datuen berreskurapena, etb. Bestalde, erabiltzaile-datuak bitstreamren bidez maneilatzea ahalbideratzen duten teknikak proposatu dira. Teknika horiek nahitaezkoak izan dira aurkeztutako zenbait hutsegite-tolerantzia handitzeko tekniken garapenerako.

Resumen

Gracias al desarrollo de las tecnologías de diseño y fabricación, los circuitos electrónicos han llegado a grandes niveles de integración. De esta forma, hoy en día es posible implementar completos y complejos sistemas dentro de un único dispositivo incorporando gran variedad de elementos como: procesadores, osciladores, lazos de seguimiento de fase (PLLs), interfaces, conversores ADC y DAC, módulos de memoria, etc. A este concepto de diseño se le denomina comúnmente SoC (System-on-Chip).

Una de las plataformas para implementar estos sistemas que más importancia está cobrando son las FPGAs (Field Programmable Gate Array). Históricamente la plataforma más utilizada para albergar los SoCs han sido las ASICs (Application-Specific Integrated Circuits), debido a su bajo consumo energético y su gran rendimiento. No obstante, su costoso proceso de desarrollo y fabricación hace que solo sean rentables en el caso de producciones masivas. Las FPGAs, por el contrario, al ser dispositivos configurables ofrecen, la posibilidad de implementar diseños personalizados a un coste mucho más reducido. Por otro lado, los continuos avances en la tecnología de las FPGAs están haciendo que éstas compitan con las ASICs a nivel de prestaciones (consumo, nivel de integración y eficiencia).

Ciertas tecnologías de FPGA, como las SRAM y Flash, poseen una característica que las hace especialmente interesantes en multitud de diseños: la capacidad de reconfiguración. Dicha característica, que incluso puede ser realizada de forma autónoma, permite cambiar completamente el diseño hardware implementado con solo cargar en la FPGA un archivo de configuración denominado *bitstream*. La reconfiguración puede incluso permitir modificar una parte del circuito configurado en la matriz de la FPGA, mientras el resto del circuito implementado continúa inalterado. Esto que se conoce como reconfiguración parcial dinámica, posibilita que un mismo chip albergue en su interior numerosos diseños hardware que pueden ser cargados a demanda. Gracias a la capacidad de reconfiguración, las FPGAs ofrecen numerosas ventajas como: posibilidad de personalización de

diseños, capacidad de re-adaptación durante el funcionamiento para responder a cambios o corregir errores, mitigación de obsolescencia, diferenciación, menores costes de diseño o reducido tiempo para el lanzamiento de productos al mercado.

Los SoC basados en FPGAs allanan el camino hacia un nuevo concepto de integración de hardware y software, permitiendo que los diseñadores de sistemas electrónicos sean capaces de integrar procesadores embebidos en los diseños para beneficiarse de su gran capacidad de computación. Gracias a esto, una parte importante de la electrónica hace uso de la tecnología FPGA abarcando un gran abanico de campos, como por ejemplo: la electrónica de consumo y el entretenimiento, la medicina o industrias como la espacial, la aviónica, la automovilística o la militar.

Las tecnologías de FPGA existentes ofrecen dos vías de utilización de procesadores embebidos: procesadores *hard-core* y procesadores *soft-core*. Los *hard-core* son procesadores discretos integrados en el mismo chip de la FPGA. Generalmente ofrecen altas frecuencias de trabajo y una mayor previsibilidad en términos de rendimiento y uso del área, pero su diseño hardware no puede alterarse para ser personalizado. Por otro lado, un procesador *soft-core*, es la descripción hardware en lenguaje HDL (normalmente VDHL o Verilog) de un procesador, sintetizable e implementable en una FPGA. Habitualmente, los procesadores *soft-core* suelen basarse en diseños hardware ya existentes, siendo compatibles con sus juegos de instrucciones, muchos de ellos en forma de *IP cores* (*Intellectual Property cores*). Los *IP cores* ofrecen procesadores *soft-core* prediseñados y testeados, que dependiendo del caso pueden ser de pago, gratuitos u otro tipo de licencias. Debido a su naturaleza, los procesadores *soft-core*, pueden ser personalizados para una adaptación óptima a diseños específicos. Así mismo, ofrecen la posibilidad de integrar en el diseño tantos procesadores como se desee (siempre que haya disponibles recursos lógicos suficientes). Otra ventaja importante es que, gracias a la reconfiguración parcial dinámica, es posible añadir el procesador al diseño únicamente en los casos necesarios, ahorrando de esta forma, recursos lógicos y consumo energético.

Uno de los mayores problemas que surgen al usar dispositivos basados en las tecnologías SRAM o la flash, como es el caso de las FPGAs, es que son especialmente sensibles a los efectos producidos por partículas energéticas provenientes de la radiación cósmica (como protones, neutrones, partículas alfa u otros iones pesados) denominados efectos de eventos simples o SEEs (*Single Event Effects*). Estos efectos pueden ocasionar diferentes tipos de fallos en los sistemas: desde fallos despreciables hasta fallos realmente graves que comprometan la funcionalidad del sistema. El correcto funcionamiento de los sistemas cobra especial relevancia cuando se trata de tecnologías de elevado costo o aquellas en las que

peligran vidas humanas, como por ejemplo, en campos tales como el transporte ferroviario, la automoción, la aviónica o la industria aeroespacial.

Dependiendo de distintos factores, los SEEs pueden causar fallos de operación transitorios, cambios de estados lógicos o daños permanentes en el dispositivo. Cuando se trata de un fallo físico permanente se denomina *hard-error*, mientras que cuando el fallo afecta el circuito momentáneamente se denomina *soft-error*. Los SEEs más frecuentes son los *soft-errors* y afectan tanto a aplicaciones comerciales a nivel terrestre, como a aplicaciones aeronáuticas y aeroespaciales (con mayor incidencia en estas últimas). La contribución exacta de este tipo de fallos a la tasa de errores depende del diseño específico de cada circuito, pero en general se asume que entorno al 90 % de la tasa de error se debe a fallos en elementos de memoria (latches, biestables o celdas de memoria). Los *soft-errors* pueden afectar tanto al circuito lógico como al *bitstream* cargado en la memoria de configuración de la FPGA. Debido a su gran tamaño, la memoria de configuración tiene más probabilidades de ser afectada por un SEE.

La existencia de problemas generados por estos efectos reafirma la importancia del concepto de tolerancia a fallos. La tolerancia a fallos es una propiedad relativa a los sistemas digitales, por la cual se asegura cierta calidad en el funcionamiento ante la presencia de fallos, debiendo los sistemas poder soportar los efectos de dichos fallos y funcionar correctamente en todo momento. Por tanto, para lograr un diseño robusto, es necesario garantizar la funcionalidad de los circuitos y asegurar la seguridad y confiabilidad en las aplicaciones críticas que puedan verse comprometidos por los SEE. A la hora de hacer frente a los SEE existe la posibilidad de explotar tecnologías específicas centradas en la tolerancia a fallos, como por ejemplo las FPGAs de tipo fusible, o por otro lado, utilizar la tecnología comercial combinada con técnicas de tolerancia a fallos. Esta última opción va cobrando importancia debido al menor precio y mayores prestaciones de las FPGAs comerciales.

Generalmente las técnicas de endurecimiento se aplican durante la fase de diseño. Existe un gran número de técnicas y se pueden llegar a combinar entre si. Las técnicas prevalentes se basan en emplear algún tipo de redundancia, ya sea hardware, software, temporal o de información. Cada tipo de técnica presenta diferentes ventajas e inconvenientes y se centra en atacar distintos tipos de SEE y sus efectos. Dentro de las técnicas de tipo redundancia, la más utilizada es la hardware, que se basa en replicar el módulo a endurecer. De esta forma, cada una de las réplicas es alimentada con la misma entrada y sus salidas son comparadas para detectar discrepancias. Esta redundancia puede implementarse a diferentes niveles. En términos generales, un mayor nivel de redundancia hardware implica una mayor robustez, pero también incrementa el uso de recursos. Este incremento

en el uso de recursos de una FPGA supone tener menos recursos disponibles para el diseño, mayor consumo energético, el tener más elementos susceptibles de ser afectados por un SEE y generalmente, una reducción de la máxima frecuencia alcanzable por el diseño. Por ello, los niveles de redundancia hardware más utilizados son la doble, conocida como DMR (Dual Modular Redundancy) y la triple o TMR (Triple Modular Redundancy).

La DMR minimiza el número de recursos redundantes, pero presenta el problema de no poder identificar el módulo fallido ya que solo es capaz de detectar que se ha producido un error. Ello hace necesario combinarlo con técnicas adicionales. Al caso de DMR aplicado a procesadores se le denomina *lockstep* y se suele combinar con las técnicas *checkpoint* y *rollback recovery*. El *checkpoint* consiste en guardar periódicamente el contexto (contenido de registros y memorias) de instantes identificados como correctos. Gracias a esto, una vez detectado y reparado un fallo es posible emplear el *rollback recovery* para cargar el último contexto correcto guardado. Las desventajas de estas estrategias son el tiempo requerido por ambas técnicas (*checkpoint* y *rollback recovery*) y la necesidad de elementos adicionales (como memorias auxiliares para guardar el contexto).

Por otro lado, el TMR ofrece la posibilidad de detectar el módulo fallido mediante la votación por mayoría. Es decir, si tras comparar las tres salidas una de ellas presenta un estado distinto, se asume que las otras dos son correctas. Esto permite que el sistema continúe funcionando correctamente (como sistema DMR) aún cuando uno de los módulos quede inutilizado. En todo caso, el TMR solo enmascara los errores, es decir, no los corrige. Una de las desventajas más destacables de ésta técnica es que incrementa el uso de recursos en más de un 300 %. También cabe la posibilidad de que la salida discrepante sea la realmente correcta (y que por tanto, las otras dos sean incorrectas), aunque este caso es bastante improbable. Uno de los problemas que no se ha analizado con profundidad en la bibliografía es el problema de la sincronización de procesadores *soft-core* en sistemas TMR (o de mayor nivel de redundancia). Dicho problema reside en que, si tras un fallo se inutiliza uno de los procesadores y el sistema continua funcionando con el resto de procesadores, una vez reparado el procesador fallido, éste necesita sincronizar su contexto al nuevo estado del sistema.

Una práctica bastante común en la implementación de sistemas redundantes es combinarlos con la técnica conocida como *scrubbing*. Esta técnica basada en la reconfiguración parcial dinámica, consiste en sobrescribir periódicamente el *bitstream* con una copia libre de errores apropiadamente guardada. Gracias a ella, es posible corregir los errores enmascarados por el uso de algunas técnicas de endurecimiento como la redundancia hardware. Esta copia libre de errores suele omitir los bits del *bitstream* correspondientes a la memoria de usuario, por lo que

solo actualiza los bits relacionados con la configuración de la FPGA. Por ello, a esta técnica también se la conoce como *configuration scrubbing*. En toda la literatura consultada se ha detectado un vacio en cuanto a técnicas que propongan estrategias de scrubbing para la memoria de usuario.

Con el objetivo de proponer alternativas innovadoras en el terreno de la tolerancia a fallos para procesadores *soft-core*, en este trabajo de investigación se han desarrollado varias técnicas y flujos de diseño para manejar los datos de usuario a través del *bitstream*, pudiendo leer, escribir o copiar la información de registros o de memorias implementadas en bloques RAMs de forma autónoma. Así mismo se ha desarrollado un abanico de propuestas tanto como para estrategias *lockstep* como para la sincronización de sistemas TMR, de las cuales varias hacen uso de las técnicas desarrolladas para manejar las memorias de usuario a través del *bitstream*. Estas últimas técnicas tienen en común la minimización de utilización de recursos respecto a las estrategias tradicionales. De forma similar, se proponen dos alternativas adicionales basadas en dichas técnicas: una propuesta de scrubbing para las memorias de usuario y una para la recuperación de información en memorias implementadas en bloques RAM cuyas interfaces hayan sido inutilizadas por SEEs.

Todas las propuestas han sido validadas en hardware utilizando una FPGA de Xilinx, la empresa líder en fabricación de dispositivos reconfigurables. De esta forma se proporcionan resultados sobre los impactos de las técnicas propuestas en términos de utilización de recursos, consumos energéticos y máximas frecuencias alcanzables.

Abstract

In the last years, the integration level of electronic circuits has been widely increased with the development of design and manufacturing techniques. As a result, complex system can be implemented inside of a single device. This high level of integration concept, in connection with design, is commonly known as SoC (System-on-Chip). One of the most interesting platforms to implement these systems are the FPGAs (Field Programmable Gate Array), due to their flexibility and the continued innovation in their technology.

The FPGAs can be programmed with different configurations to perform distinct functions. This capacity together with the partial dynamic reconfiguration of some SRAM-based or flash-based FPGAs, provide them with a valuable potential. The employment of partial dynamic reconfiguration permits modifying just one part in the array of the FPGA, while the rest of the circuit remains unchanged. Consequently, FPGAs may be programmed by different partial designs, increasing their functionality and decreasing the resource usage.

However, one of the most remarkable problems using SRAM and flash technologies is that certain FPGAs are especially sensitive to effects caused by energetic particles from the cosmic radiation. Owing to these effects, different system failures can be produced, from worthless ones to really serious ones. The worst failures might even hinder the correct performance of several important systems in relation to railway, automotion or industrial control. Hence, a wrong performance of these systems might cause permanent damages in highly expensive equipment and even endanger the human life itself.

The existence of these problems as a result of the aforesaid effects makes the importance of the fault tolerance concept unavoidable. Fault tolerance, is a property related to digital systems and ensures certain quality of operation in the presence of failures. Because of fault tolerance, the systems are supposed to work properly at any time even though failures take place. For this purpose,

hardening techniques are applied to these systems, such as, ones which are based on the redundancy of software and/or hardware.

On the other hand, most of the electronic devices which are used daily make use of embedded systems which incorporate one ore more processor cores for the execution of the different tasks. In addition, the increase in complexity of the designs makes the electronic systems which have been designed and tested beforehand more attractive. Those designs are called IP cores (Intellectual Property cores). Due to those reasons, the utilization of processor IP cores (known as soft-core processors) in FPGA designs is gaining momentum.

Following this idea this work performs a research in the field of fault tolerance for SRAM based FPGA designs especially focusing on soft-processors. With the aim of improving the well established hardening techniques, this work proposes several approaches that deal with distinct aspects related to fault tolerance. In this way, three lockstep (a dual redundancy based technique) approaches, five synchronization methods for repaired modules in modular redundancy schemes, a method to recover information from memories with damaged interfaces and a user-data scrubbing approach are proposed in this work. Several of these proposed methods are based on the utilization of two methodologies to manage user data of both, flip-flops and BRAMs, through the bitstream that have been also developed in this work.

The proposed methods provide the designers with valuable tools when developing fault tolerant designs implemented in SRAM based FPGA devices.

Contents

List of Figures

List of Tables

List of Acronyms

ALU	*Arithmetic Logic Unit*
AMBA	*Advanced Micro-controller Bus Architecture*
APU	*Aplication Processor Unit*
ASIC	*Application Specific Integrated Circuit*
ASMBL	*Advanced Silicon Modular Block*
ASTERICS	*Advanced System for the TEst under Radiation of Integrated Circuits and Systems*
BBBA	*Bitstream Based BRAM Approach*
BRAM	*BlockRAM*
BSD	*Berkeley Software Distribution*
CISC	*Complex Instruction Set Computer*
CLB	*Configurable Logic Block*
CLK	*Clock*
CMD	*Command Register*
CMOS	*Complementary Metal-Oxide Semiconductor*
CMT	*Clock Management Tiles*
COTS	*Commercial Off-The-Shelf*
CPLD	*Complex Programmable Logic Device*
CPU	*Controling Process Unit*
CRC	*Cyclic Redundancy Check*
CSoPC	*Configurable-System-on-Chip*
DD	*Displacement Damage*
DFI	*Direct Fault Injection*
DMIP	*Dhrystone Million Instructions Per Second*
DSP	*Digital Signal Processor*
DUT	*Device Under Test*

ECC	*Error Correction Code*
EDAC	*Error Detection and Correction*
EDC	*Error Detection Code*
EMIO	*Extended Multiplexed I/O*
EPLD	*Erasable Programmable Logic Device*
FAR	*Frame Address Register*
FDRI	*Frame Data Input Register*
FDRO	*Frame Data Output Register*
FI	*Fault Injection*
FIFA	*Fault-Injection Fault Analysis tool*
FITO	*FPGA-based Fault Injection Tool*
FPGA	*Field Programmable Gate Array*
FSBL	*First Stage Boot Loader*
FSM	*Finite State Machine*
FUSE	*Fault injection Using Semulation*
GNU	*General Public License*
GPIO	*General Purpose Input/Ouputs*
GSR	*Global Set Reset*
HAL	*Hardware Abstraction Layer*
HDL	*Hardware Description Languaje*
IDC	*Instruction Decode and Control*
IDE	*Integrated Device Electronic*
ILA	*Integrated Logic Analyzer*
IOB	*Input-Output-Block*
IP	*Intellectual Proprietary*
IPF	*In-place X-Filing*
ISR	*Interrupt Service Routine*
KCPSM	*Constant(K) Coded Programmable State Machine*
LET	*Linear Energy Transfer*
LGPL	*GNU Lesser General Public License*
LRR	*Least Recently Replaced*
LUT	*Look-Up Table*
MBU	*Multiple Bit Upset*
MCSoPC	*Multiprocessor-Configurable-System-on-Chip*
MCU	*Multiple Cell Upsets*
MIO	*Multiplexed I/O*
MIPS	*Million Instructions Per Second*

MIT	*Massachusetts Institute of Technology*
MMU	*Memory Management Unit*
MPU	*Memory Protection Unit*
NETFI	*NETlist Fault Injection*
PL	*Programmable Logic*
PLD	*Programmable Logic Device*
PowerPC	*Performance Optimization With Enhanced RISC - Performance Computing*
PS	*Processing System*
PWM	*Pulse-Width Modulation*
RAM	*Random Access Memory*
RCRC	*Reset Cyclic Redundancy Check*
RISC	*Reduced Instruction Set Computer*
ROM	*Read Only Memory*
RTC	*Real Time Clock*
RTL	*Register-Transfer-Level*
SCFIT	*Shadow Components-based Fault Injection Technique*
SCHJ	*Self-Checking Hardware Journal*
SEB	*Single Event Burnout*
SEE	*Single Event Effect*
SEFI	*Single Event Functional Interrupt*
SEGR	*Single Event Gate Rupture*
SEL	*Single Event Latch-up*
SEM	*Soft Error Mitigation*
SER	*Soft Error Rate*
SET	*Single Event TransientMBU*
SEU	*Single Event Upset*
SFR	*Special Function Registers*
SHE	*Single Hardware Errors*
SIHFT	*Software Implemented Hardware Fault Tolerance*
SoC	*System-On-Chip*
SOI	*Silicon on Insulator*
SoPC	*System-on-Programmable-Chip*
SPARC	*Scalable Processor Architecture*
SPLD	*Simple Programmable Logic Device*
SRAM	*Static Random Access Memory*
SWIFT	*Software Implemented Fault Tolerance*
TID	*Total Ionizing Dose*

TMR	*Triple Modular Redundancy*
UART	*Universal Asynchronous Receiver Transmitter*
VHDL	*VHSIC Hardware Description Languaje*
VLSI	*Very-Large-Scale Integration*

Chapter 1

Introduction

Nowadays, thanks to the continuous improvement of technology, complex systems that incorporate a wide range of components (processors, oscillators and phase-locked loops, external interfaces, ADCs and DACs, memory modules, etc.) can be integrated in a single device. This concept is commonly known as System-on-Chip (SoC). Several technologies are available to be used as an implementation platform for SoC, such as, Application-Specific Integrated Circuits (ASICs), Field Programmable Gates Arrays (FPGAs) or Application Specific Standard Products (ASSPs). ASICs are devices utilized to develop specific applications. Hence, they can be considered as custom made chips, mainly designed and used by a single company. The main benefits of this technology are high performance and low power consumption. Nevertheless, their development is very complex process (resource-intensive and time-consuming). Due to this, ASICs' cost effectiveness is directly related to the production volume. While large production volumes provide reduced per-unit costs, small ones generate prohibitive results. The design and development of ASSPs is similar to ASIC's, providing the same benefits and drawbacks. The most remarkable difference comparing with ASICs is that ASSPs are more general-purpose devices, making possible to be used by different customers.

FPGAs, especially SRAM-based ones, are lately gaining momentum as an alternative to implement SoCs due to the advantages they provide. FPGAs are integrated circuits that contain a huge number of programmable fabric that can be configured to implement desired functions. One of the most relevant features of certain FPGA types, like flash or SRAM based FPGAs, is their reconfiguration capability. It makes them suitable for achieving customizable designs, capable of

re-adapting in the field in response to the changes and correct possible issues, providing high flexibility, obsolescence mitigation, differentiation, low non-recurring engineering costs and short time to market with adequate levels of integration, power consumption and performance.

FPGA-based SoCs pave the way towards a new concept of hardware and software integration. As a consequence, designers can take advantage of the complex computation power of an embedded processor and the mentioned benefits of FPGAs. Furthermore, these SoC systems also are also suitable to create custom accelerators that enhance the performance of designs adapted to the requirements of particular applications. Hence, as Figure 1.1 illustrates, much of the electronic products used in daily life are starting to utilize SoCs implemented in FPGAs, such as, consumer electronics and entertainment industry [12, 13], automotive industry [14, 15], medicine [16, 17], military [18, 19], space and avionics [20, 21], etc.

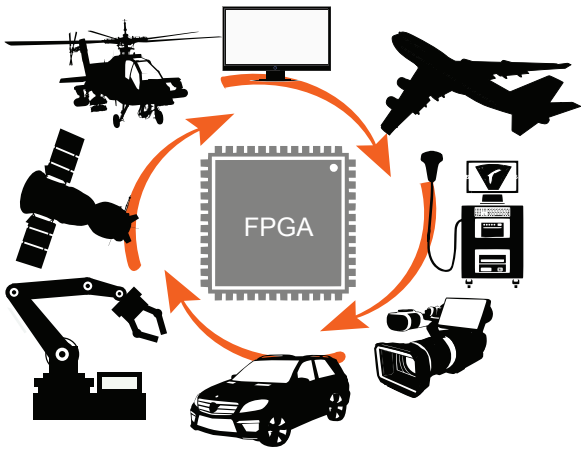


Figure 1.1: FPGA application fields examples.

SoC designs commonly incorporate one (or more) processor core(s) [22] for the execution of the different tasks or to control the overall system. In many cases the processor can be considered as one of the most crucial elements of a SoC. Embedded processors are available either as built-in hard-cores or soft-cores (using the FPGA reconfigurable logic resources). Although hard-core processors provide better performance [23] [24], their usability is restricted. This is because, in case of having, the amount of embedded processors available in each SoC is very limited. In contrast, soft core processors are more flexible, since a large number of soft-core processors can be implemented (depending on the available resources of

the device) [25]. Besides, several soft-core processors with different architectures and complexities are available, enabling an accurate adaptation of the designs to specific requirements. An additional advantage is that due to the reconfiguration capability of FPGAs, soft-core processors can be implemented only on necessary cases, reducing the power consumption and maximizing the functionality of FPGA implementations. Furthermore, soft-cores can be designed independently of the platform [26], achieving greater immunity to obsolescence than the circuit or logic level descriptions.

1.1 Motivation

The continued innovation in the technology for developing and manufacturing FPGAs has allowed to increase the integration level. For instance, the latest Xilinx UltraScale+ are 16 nm FinFET+ based devices. This high integration enables features, such as, high capacity, low-power consumption and faster operation. However, the high integration level comes with different drawbacks. One of the most relevant one is the increase of the susceptibility to space radiation induced faults [27]. Hence, a susceptibility that initially was a problem in avionic or spacial applications, has become an actual concern even for ground-level applications. The group of different possible effects of these faults is called Single Event Effects (SEEs). When working with SRAM based FPGA, the most critical effect of this group are SEUs. SEUs are non-permanent soft errors [28] produced by contaminant alpha particles in electronics or when protons and cosmic rays from outer space interact with the atmosphere and generate subatomic particles that collide with silicon atoms. SEUs can affect FPGAs flipping bits in both, user memories or the configuration memory. SEUs located in the configuration memory are the most critical case because they can alter the implemented design by changing the configuration of crucial elements or their interconnections. The repercussion of the SEUs on the configurations relies on the application in which the FPGA device is implemented, including the number of configuration bits. FPGAs typically have one order of magnitude more configuration memory than BRAM [29–31]. However, faults in user memories can also be critical. Besides, some applications may contain a considerably large user memory. Hence, the reliability of user memories is also a concern in terms of fault tolerance.

As a result of an upset the entire system or a critical part of it can be unavailable or present malfunction. For this reason, in certain applications, especially in fields related to human safety or very valuable technologies, such as railway, avionic or spacial applications the result of a single error can be catastrophic. In those cases, high levels of reliability are required, which demands to apply fault

tolerance techniques to harden the designs.

Plenty of fault tolerance approaches have been proposed in the literature to increase the robustness of FPGA designs. Most of them are based on adopting some redundancy level distinct fields, like hardware, software, time or data. Other, methods exploit the dynamic partial reconfiguration capability of FPGAs. In any case, despite that all the alternatives can increase the reliability of FPGA implementations, each solution affects negatively to original designs in terms of resource overhead, performance penalty and/or availability.

Among the different options the most extended techniques to harden soft-core processor designs are based on both, double and triple, modular hardware redundancies. This is because they provide adequate reliability levels and acceptable performance penalties. However, the price to pay is the increase of hardware overhead.

The scarcity of low hardware-overhead demanding solutions has motivated to research for new alternatives that contribute to increase the fault tolerance level of soft-core processor designs for SRAM FPGAs, without drastically increasing the resource overhead. The increase of resource utilization commonly implies a variety of drawbacks, such as, higher power consumption, more resources are susceptible to induced faults, longer datapaths (which means on performance penalty), etc.

On the other hand, as it has been observed there is a lack of alternatives to manage user data through the bitstream, especially for the case of newer FPGA devices, like Xilinx 7 series. Bearing in mind the idea that a bitstream based user data management could provide new ways to harden FPGA designs, it has been also considered especially convenient to conduct a research in this field.

1.2 Objectives

The major objectives of this work can be summarized in the following ideas:

- To make an analysis of the state-of-the-art in fault-tolerance techniques to harden SRAM-based designs focusing on soft-core processors. This study includes different redundancy based schemes and reconfiguration based techniques.
- To propose new approaches to harden for soft-core processors implemented in SRAM-based FPGAs or improve existing ones in terms of reliability, hardware overhead, performance penalty and/or availability.

- To study the possibility of utilizing the bitstream to manage user data to provide new ways to harden FPGA designs and propose novel methods to exploit the potential of this approach.
- To design and implement different FPGA-based designs and tests in order to validate the proposed approaches, including several comparisons with well stablished schemes in order evaluate the results.

1.3 Structure of the Document

In addition to the actual introduction chapter, this document is divided into five main chapters:

- **Chapter 2.** The most remarkable aspects of soft-core processors, including the relevant IPs, and the most outstanding features and the architecture of SRAM-based FPGAs (especially focusing on the Zynq and the 7 series by Xilinx) are presented in Chapter 2. This chapter also discusses about the structure of the bitstream and the approaches available to manage user data utilizing it. This chapter closes introducing radiation effects on soft-core processors implemented in SRAM FPGAs related to the fault tolerance.
- **Chapter 3.** The state of the art of techniques to provide fault tolerance of SRAM-based FPGAs applicable to soft-core processors are introduced in Chapter 3. This chapter also introduces the main fault tolerance evaluation methods, including the physical fault injection methods and the bitstream based fault injection methods.
- **Chapter 4.** Chapter 4 presents the main contributions of this work, including a specifically designed soft-core processor IP and bitstream based approaches to manage user data from both, BRAM memories and registers. Based on those two main contributions different approaches related to fault tolerance application are also proposed in this chapter.
- **Chapter 5.** The validation of the different proposed approaches and the utilized evaluation platform are presented in Chapter 4. The results obtained in different validation and comparison tests are also provided.
- **Chapter 6.** The document closes with Chapter 6, which summarizes the entire work and provides an outlook to future researches based on the approaches proposed in it.

Chapter 2

Soft-Core Processors Implemented in SRAM Based FPGAs

Electronic systems and electronic products used in daily life usually incorporate one (or more) processor core(s) [22] for the execution of the different tasks, thanks to their wide compatibility with high-level applications. Due to the tendency of obtaining compact designs and the benefits that they provide, such as flexibility, hardware and software cost reduction, obsolescence mitigation and hardware acceleration [23, 32], embedded processors are gaining relevance over Commercial Off-The-Shelf (COTS) processors. Embedded processors can be implemented as hard-cores (also known as hard-macros) or soft-cores. While a soft-core is implemented in logic fabric, a hard-core processor is built similar to regular integrated circuits but maintaining a direct connection with the logic fabric. Although hard core processors may provide better performance and predictability [23] [24], their use may be limited taking into account that each implementation platform (in the case of having any) has a limited and, usually, small number of processors. In addition, hard-core designs are fixed designs that cannot be modified. In contrast, soft core processors are more flexible, since a large number of soft core processors can be implemented (depending on the available logic resources of the implementation platform). Besides, several soft-core processors with various architectures and complexities are available, enabling accurate adaptations of designs to specific requirements. Due to their platform independence [26], a wide range of soft-cores are available.

Two principal technologies are available to be used as an implementation platform for soft-core processors [33, 34]: Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs).

When implementing complex systems, ASIC circuits are more efficient in terms of resource usage compared to FPGA. It takes less logic gates to implement the same application in ASIC than in FPGA. In addition, FPGA designs lead to a higher power consumption and lower processing speed compared to similar designs implemented in ASICs [35]. Nevertheless, using ASICs the design process is extremely slow and the manufacturing requires high production volumes to be economically feasible.

FPGA systems offer several advantages like low cost design, powerful and easy software design environments, simulation and synthesis, short time to market and reconfigurability. Due to that, FPGAs provide high capability to implement complex designs with a reasonable level of idle power consumption. One of the greatest benefits provided by this technology is the dynamic reconfiguration capability of SRAM-based FPGAs. Thanks to this feature an entire hard implementation, or a part of it, can be completely modified even during operation.

These benefits make SRAM-based FPGAs one of the most remarkable candidates to implement soft-core processors. Although a soft-core processor implemented in SRAM-based FPGAs is commonly slower (it is limited by the speed of its technology) than its hard equivalent [23, 24], it provides interesting features [25] like flexibility and low design cost. This flexibility means that it is possible to customize the design for the applications that require particular characteristics as in [36]. Since the programming is done in Hardware Description Language (HDL) language with a high level of abstraction, the design task can be performed in a straightforward fashion. Another benefit of SRAM-based soft-core processors is that they can be implemented only when required. It is also possible to instantiate as many cores as desired (bearing in mind the limitations of the FPGA), maximizing the functionality and saving energy and logic resources. Considering that soft-cores can be designed independent to platform [26] and that FPGA implementation can be updated thanks to the reconfiguration, SRAM FPGA designs based soft-core processor provide a high level of immunity to obsolescence. As stated in [37], the obsolescence of Motorola 6800 hard processor present in some equipment of French nuclear plants became a problem. The study developed in [37] compares a soft-core version of the 6800 hard processor as a solution to cope with processor obsolescence.

An embedded soft-core processor uses the logic resources of the FPGA to build the different elements, such as internal memory, registers, processors and internal peripheral buses and external peripheral controllers. The more elements are

added to the processor, the larger power and usability are provided. Nevertheless, a big size comes with a performance reduction and higher FPGA area usage. Due to these reasons, the selection of a proper soft-core is a relevant decision to be adopted by designers. An interesting strategy may be to utilize different small soft-core processors in a large system, as in [38, 39], in a way that they can manage different small tasks (not critical in terms of time by themselves) to the main processor with more strict coupling with the hardware circuits.

This chapter opens with Section 2.1, discussing basic aspects of soft-core processors, including the common basic architectures and the most relevant IP cores available. Next, the technology of SRAM FPGAs is presented in Section 2.2, focusing on Xilinx 7 series devices. This section also introduces the structure of the bitstream of these devices and available methods to manage user data through its utilization. This chapter closes discussing radiation effects in SRAM FPGAs, especially targeting soft-core processor implementations.

2.1 General Aspects of Soft-Core Processors

Soft-core processors share the majority of features with their COTS counterparts. In fact, a large number of soft-core available processors are based on the design of classic microcontrollers or processing units. The more generalised soft-core is, the less closely it fits the low-level target architecture. Thus, the less efficiency is obtained in terms of area and performance. Thanks to the flexibility provided by a soft-core design, the basic architecture of original designs can be customized, enabling to develop soft-core implementations highly coupled with specific requirements. Nevertheless, the price to pay when making adaptations is a higher designing effort and the possibility of damaging the design. Hence, the designing process of a soft-core processor requires a trade-off between several aspects like, portability, efficiency, performance, power consumption, scalability, maintainability, extensibility, complexity, customisability, design-cost and development tool-chain support.

When customizing a design, aspects, such as, the available number of in/out ports, interfaces for memories or communications buses, timers, different peripherals, etc. have to be considered. Although they can increase the flexibility and functionality, they also come with a hardware overhead and a performance penalty. For that reason, while some of the developed soft-cores implement complete microcontroller architectures, other designs only implement the processor section.

Although each processor design has its own characteristics and elements, there

are some fundamental components shared by them. The core piece of a processor is the controller mechanism, which can be a single module or it can be divided in various blocks. It is usually implemented by utilizing Finite State Machine (FSM) structures and its purpose is to manage and synchronize the different elements.

Another crucial element is the Arithmetic Logic Unit (ALU), which performs arithmetic and bitwise logic operations. Due to the inherent need of storing various data, such as, instructions, user data or execution information, memory components are other elementary aspects in processor's architectures. Since each storage function requires different characteristics, distinct memory elements are usually implemented, like registers, memory stacks or large memory modules.

2.1.1 Basic Architectures of Soft-Core Processors

The most crucial aspect of a soft-core processor is its architecture because it determines the entire design. Harvard and Von Neumann are the two prevalent architectures.

The Harvard architecture (Figure 2.1), is a simple structure that includes an ALU, a control unit, two memories and input/output ports. As it can be seen from the figure, it separates physically the storage and signal pathways for the program memory and the data memory. Thanks to this feature, each memory can have its own specific characteristics (word width, address depth, timing, implementation technology, etc.). Besides, a Harvard architecture also provides simultaneous access to more than one memory system. Thus, it is possible to concurrently access to program and data memories. There are also various modified Harvard versions that add supplementary features to the original architecture, such as, separate program and data caches or the ones that provide a pathway between the program memory and the control unit. Its utilization is especially extended in microcontrollers and DSP units.

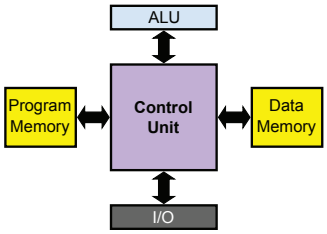


Figure 2.1: Block diagram of Harvard architecture.

As Figure 2.2 depicts, the Von Neumann architecture is simpler than the Harvard. It consists of a Central Processing Unit (CPU) that includes an ALU and a control unit, a shared memory for user data and program instructions and input/ouput devices. The shared memory and the usage of the same bus implies that it is not possible to read an instruction and to write/read to/from the memory at the same time. During program execution the CPU fetches a single instruction from the shared memory and executes it. This sequential instruction execution produces a relatively slow operation. This problem is usually referred as the *Von Neuman Bottleneck*. For this reason, usually the common application scope of this architecture usually are interface and control applications.

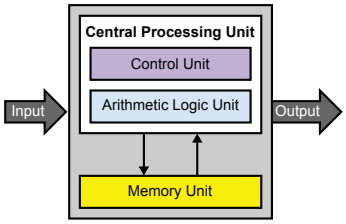


Figure 2.2: Block diagram of Von Neumann architecture.

Another decision that has to be adopted when designing a processor is to determine its instruction set. The two major strategies in this context are the Reduced Instruction Set Computing (RISC) and the Complex Instruction Set Computing (CISC):

- The CISC strategy is based on utilizing a single instruction to execute various low-level operations, enabling multi-step operations or different addressing modes within a single instruction. To do so, this strategy focuses on building the complex instructions in the hardware. Thanks to this a CISC architecture can complete tasks with few instructions which reduces the memory needs. In addition, a CISC scheme significantly reduces complier's work.

- The RISC strategy relies on the idea that a simplified instruction set can provide higher performance levels due to the more limited usage of clock cycles to execute each instruction. To adopt this strategy implies more instructions and, hence, larger program memories. In addition, the compiler has to perform more work to convert high-level code into assembler. Nevertheless, the utilization of reduced instruction strategies demands less hardware to implement the functionality of the instructions. Another significant advantage is that due to the uniform demand of clock cycles from

instructions, it is possible to utilize pipelining, which enables to increase performance.

The number of clock cycles required to execute each instruction is a relevant factor, which is known as instruction cycle. In many cases the demanded cycles may vary depending on the instruction type. For instance, some processors need more cycles to execute conditional branches than regular instructions. In any case, the less clock cycles needed for each instruction, the faster is executed the program. Execution with reduced clock cycles commonly comes with hardware complexity. Which on the other hand, can affect the design's overall performance and/or the resource overhead. In [39], three versions of the same microcontroller were developed, each of them with different structures of the instruction cycle: a single clock cycle with a dual clock, a double clock cycle, and a single clock cycle. The results obtained in terms of maximum achievable frequency were 36 MHz, 86.1 MHz and 55 MHz, respectively. In addition, the third version significantly increased the hardware overhead. Due to those aspects, a good design practice is to study the application or system requirements (system's clock frequency, available resources, etc.) in order to make a trade-off decision that fits better.

2.1.2 Soft-Core Processor IPs

Soft-core designs are usually available as Intellectual Property (IP) cores, which can be protected under intellectual property laws or patents. An IP core is a reusable unit of logic design that belongs to a designer or to a party. Hence, once a soft-core IP has been designed, it is available to be re-used across several designs. IPs often are available in HDL, such as, VHDL or Verilog, providing a synthesizable Register-Transfer Level (RTL) description. This allows developers to adapt designs at functional level by modifying the HDL code. However, considering that not all the IP vendors offer support or warranty for modified designs, this possibility can be an arduous task depending on the situation. In cases where vendors find protection against reverse engineering, the IP cores are provided as netlists. A netlist is a generic gate-level description of the connectivity of circuit elements.

Thanks to their adaptability, reusability, potential and the relative ease of designing HDL models, there is a remarkable quantity of soft-core IP processors available. Depending on the objective of developers, their utilization scope can be commercial or a decentralized development model that encourages open collaboration, known as open source. Commercial IP cores are commonly distributed under restrictive commercial licenses that have to be purchased by the client. Open source is commonly distributed under more flexible and free licenses. A

remarkable aspect of the open-source project is that, since they are based on community-work (which implies constant improvement), they can be classified in different stages of development, starting from the initial alpha and beta phases to the stable or mature categories.

One of the most remarkable open-source site is OpenCores [40]. OpenCores is a community for development of hardware IP cores as open source. Despite of the fact that OpenCores initially was a commercially owned organization, since 2015 is an independent Free and Open Source Silicon Foundation (FOSSi). Its site hosts source code of a great number of digital hardware projects including various sof-core processors. Projects from OpenCores have been used in a number of researches and even in private companies. Most of the projects in OpenCores use the GNU Lesser General Public License (LGPL) [41], a free software license that permits developers and vendors to utilize the IP without being required to release the source code of their own designs. However, any modification done in a component under LGPL has to be shared with the OpenCores community. Another widely used license is the Berkeley Software Distribution (BSD) [42], which is less restrictive and disclaims any warranty. Sourceforge is another relevant open source site that contains several soft-core processor projects. Nevertheless, the scope of this site is more general since it gathers a wide range of project types.

Several publications have compared most of the existing soft-core processors [23–25, 43–48]. For that reason, the following section only covers a brief summary of some of the most remarkable soft-core processors available from both, the major vendors and the open-source community.

Commercial Soft-Core Processor IPs

Most of FPGA vendors provide one or more soft-core processors to be used with their devices. These IPs are optimized for the vendor's technology which, usually are platform dependant. Other commercial vendors that do not manufacture hardware are exclusively focused on designing IPs. In these cases, the designed IPs have in general platform independence, which enables to implement them in different devices. In the following, a brief description of the major commercial soft-core processor IPs is presented.

- **MicroBlaze**. The MicroBlaze is a Xilinx's proprietary 32-bit soft-core processor. It has a RISC architecture that can be customized with different peripheral and memory configurations. It is highly optimized for Xilinx devices, in a way that occupies significantly less area than the majority of its counterparts, such as the OpenRISC 1200 or the LEON. Due to these

benefits, the MicroBlaze has been widely used in a number of projects [8, 21, 49–62].

- **PicoBlaze**. The PicoBlaze is a 8-bit proprietary soft-core processor provided by Xilinx. This small RISC microcontroller developed by Ken Chapman [38], is one of the most popular small soft-cores when using Xilinx devices [54, 58, 60, 63–76]. As Figure 2.3 shows, it is a simple design which has been a key factor for its success. The PicoBlaze was released under the KCPSM name. Despite that the official meaning of this acronym is *Constant(K) Coded Programmable State Machine*, the initial meaning was related with its creator ("Ken Chapman's PSM"). Xilinx provides distinct PicoBlaze versions for its device families (KCPSM, KCPSM2, KCPSM3 and KCPSM6). Third parties like Mediatronix and open-source projects provide various development tool-sets for the PicoBlaze.

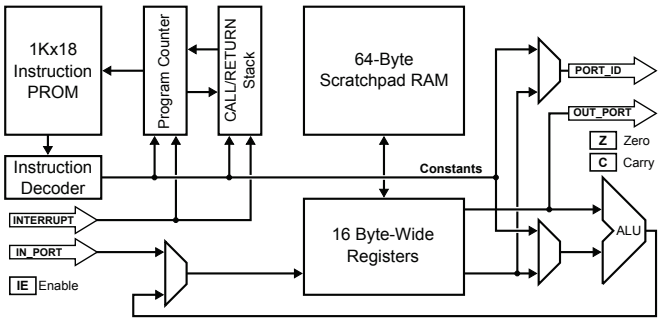


Figure 2.3: Block diagram of PicoBlaze.

- **Nios**. Nios was Altera's first 16-bit soft-core processor that was later replaced by the newer Nios II version. NiosII is a 32-bit soft-core designed specifically for the Intel (formerly Altera) family of FPGAs. It has a Hardware Abstraction Layer (HAL) architecture. Intel also allows to choose different Nios II versions (Nios II/f "Fast", Nios II/e "Economy" and Nios II/s "Standard"), each of them being optimized for specific applications. In addition, a software tool chain is available for all versions that permits to customize a set of peripherals, instruction, memories, and I/O circuitry. Bearing in mind that Nios II is limited to operating systems that utilize a simplified protection and virtual memory-model, an optional Memory Management Unit (MMU) can be implemented. This MMU allows to work with hardware-based paging and protection operating systems, like the Linux kernel. Finally, an also optional and more simpler Memory Protection Unit (MPU) is available to provide a similar memory protection. The main ben-

efit of using the MPU over the MMU is its better performance. Despite that the usage of this soft-core is less extended, various studies have used it. In [77], UT Nios, an adaptation of the Nios II, was introduced.

- **Cortex - M1**. This proprietary soft-core is a 32-bit RISC ARM processor licensed by ARM Holdings plc. Its design is based on the ARMv6-M architecture and is especially optimized for FPGAs. It has a three-stage pipeline, configurable instruction and data memories, and 1 to 32 interrupts. The utilization of this core in the literature is more limited.
- **Xtensa**. Xtensa is a set of 32-bit soft-core processors with RISC architecture featured by Tensilica. These are highly adaptable IPs that allow to customize several features like bus width, cache size, memory management and interrupt control. In addition, thanks to the supplied tools, users can extend Xtensa's instruction set by introducing new instructions.
- **TSK3000A**. TSK3000A is a 32-bit, no royalty-free, Wishbone-compatible and platform independent RISC soft-core processor from Altium Limited. It provides a fast register access and a zero-wait state block RAM amount that can be defined by the user, with a dual-port access. It has a five-stage pipeline and it can handle up to 32 interrupts that work in two modes (standard and vectored). Altium also provides the TSK51/52 IP, an 8-bit Intel 8051 instruction set compatible soft-core processor.
- **eSi-RISC**. As its name indicates, it is a RISC soft-core processor architecture developed by Ensilica. This IP has a five-stage pipeline, supports multiprocessing, can handle up to 32 interrupts and is available in five versions: eSi-1600, eSi-1650, eSi-3200, eSi-3250 and eSi-3260. While the first two IPs feature 16-bit data-path implementations, the rest feature 32-bit data-paths. Its instruction set can be freely intermixed. It supports both, floating-point and fixed-point arithmetic.
- **Other propietary IPs**. Further to the mentioned IPs, additional commercial soft-core processors are available, such as the ARC by Synopsys Inc., MCL51 and MCL86 by MicroCore Labsm, etc.

Open-Source Soft-Core Processor IPs

The majority of the open-source soft-core processors are based on existing architectures. While in many cases the reference design for these IPs is a classic hardware processor or a microcontroller, in other cases the IPs are open-source and platform independent version of a proprietary soft-core. Besides, some designers have developed new soft-core processor architectures. The majority of

these new architectures are simple 8-bit processors. Due to the huge amount of open-source soft-core processors available, the aim of the following list is to enumerate the most relevant ones, especially focusing on small processors.

- **LatticeMico32**. LatticeMico32 is an 32-bit soft-core processor optimized for FPGAs devices. Despite being developed by Lattice Semiconductor Corporation, this is an open source soft-core licensed under a free IP core license that can be implemented in FPGAs from any vendors. Both the processor IP and the development tool-set are provided in a source-code form. It has a RISC Harvard architecture and enables the possibility of combining its two buses by using its bus arbitration logic. It is a relatively small IP that has a six-stage pipeline. It can handle up to 32 interrupts and is available in three basic configurations (basis, standard and full).

- **LatticeMico8**. The LatticeMico8 is an 8-bit soft-core processor from Lattice Semiconductor Corporation. It combines an 18-bit instruction set with 32 general purpose registers to provide flexibility with a reduced resource consumption and it only takes two cycles per instruction. As the LatticeMico32, it is licensed under a free open IP license and can be implemented in devices from different vendors.

- **OpenRISC 1200**. The OpenRISC project, which gathers a series of designs based on the OpenRISC 1000 RISC architecture (the crown jewel project of the OpenCores community). Different cores are available, including 32 and 64-bit processors that support floating point and vector processing. The OpenRISC 1200 is a Verilog core, especially designed for FPGA implementations. The OpenRISC community provides a GNU toolchain for the OpenRISC that supports development in C and C++. Due to its characteristics, it is an accepted architecture, utilized in both, commercial and non-commercial scenarios.

- **SPARC based soft-cores**. The Scalable Processor Architecture (SPARC), depicted in Figure 2.4, has inspired a large number of implementations, including some popular soft-core designs. Its RISC architecture, initially designed by Sun Microsystems Inc., has various revisions. In 2006, Sun Microsystems released the source-code of the UltraSPARC T1 soft-core processor in open-source soft-core processor form. It is named OpenSPARC and is one of the few, if not the only, available 64-bit soft-core architectures. Two years later Sun Microsystems also released the upgraded OpenSPARC T2 version in open-source form.

  The UltraSPARC T1 is a complex multi-core design which utilizes too many resources to be implemented in many FPGAs. Due to that, Simply RISC

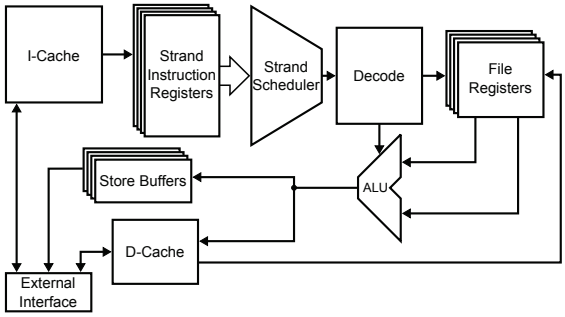


Figure 2.4: Block diagram of SPARC core.

developed the size reduced S1 soft-core based on its architecture but only containing the CPU core and a Wishbone controller.

One of the most popular version of the SPARC is the LEON. This IP is a very popular soft-core processor based on the instruction set of the SPARC-V8 processor and that can be used in FPGAs by different vendors. Despite that the LEON project was originally developed by the European Space Research and Technology Centre, time after it was distributed by Gaisler Research Inc. in 2008, Aeroflex Inc. acquired Gaisler Research, which in 2014 was also acquired by Cobham plc. Due to this, nowadays the LEON IPs are distributed by Cobham Gaisler AB. Two license modes are available for LEON. While the first alternative is a fee-free LGPL/GPL FLOSS license, the second option is a proprietary license. This configurable soft-core is provided in five versions: LEON2, LEON2-FT, LEON3, LEON3-FT and LEON4. LEON2 is the most basic version, which includes a five-stage pipeline and does not support symmetric multiprocessing. The upgraded LEON3 version provides superior features like symmetric multiprocessing support and a seven-stage pipeline. The last upgraded version is the LEON4 which includes new characteristics like a static branch prediction added to the pipeline and higher potential performance. Finally, LEON2-FT and LEON-FT are the fault tolerant versions of LEON2 and LEON3 IPs, respectively. FT versions are popular in space-related fault tolerance applications [78, 79]. Despite that they do not provide all the features of original versions, FT IPs offer protection against induced faults. This is achieved applying hardware triplication based hardening techniques to the flip-flops and utilizing error detection and correction methods with memories. The usage of LEON processors is widely extended in the literature [52, 80–96]. In [90], the SPARC core from a LEON3 soft-core is

substituted by a MIPS based core, adapting its functionality to different buses and peripherals.

- **MicroBlaze based soft-cores**. The extended use of the MicroBlaze from Xilinx has motivated the development of several open-source alternatives. The most representative ones are: aeMB (beta), MB-Lite (stable), OpenFire Processor Core (alpha) and myBlaze (mature) from OpenCores and secretBlaze developed by LIRMM, University of Montpellier. Although each of them provides distinct features, for instance not implementing floating-point support or utilizing different stage pipelines, all of them are MicroBlaze compatible cores. In addition, [97] proposes a multiprocessor system based on the MicroBlaze.

- **PicoBlaze based soft-cores** . This popular soft-core from Xilinx has also inspired different platform-independent open-source projects like the PacoBlaze (beta) and the copyBlaze (mature), available in Verilog and VHDL respectively. In [26], another platform-independent version was presented. As this work itself states, making adaptations in the highly optimized design of the PicoBlaze commonly comes with a resource overhead and performance penalty.

- **8080 based soft-cores**. Released in 1974 by Intel, the 8080 is considered one of the first usable microprocessors. Its architecture inspired different processors such as the Am9080 by AMD, Z80 by Zilog of the upgraded 8085 version by Intel. Several soft-core processors have been also developed inspired in its designs. Some of the most remarkable ones are available in OpenCores community: Lightweight 8080 compatible (stable), T80 cpu (stable), Wishbone High Performance Z80 (stable), z80control (alpha) and 80e - Z80/Z180 compatible processor extended by eZ80 instructions (stable).

- **6800 based soft-cores** The MC6800, known as 6800, is an 8-bit processor with a Von Neumann architecture released by Motolora in 1974. Its use was widely extended in control designs for computer peripherals and test equipment. OpenCores provides different soft-cores based on this processor: System68 (stable), System11 (alpha), 68HC05/68HC08 (stable) and HC11 compatible Gator Microprocessor (stable).

- **6502 based soft-cores**. The 6502 is another classic architecture that was developed in 1975 by two of the former designers of the 6800. Due to its reduced price, this microcontroller became very popular in its time. OpenCores provides several soft-core alternatives based on this design: T65 CPU (stable), ag 6502 soft core with phase-level accuracy (beta), CPU6502 TC -

CPU65C02 TC Processor Soft Core with accurate timing (stable), T6507LP (beta) and Lattice 6502 (beta), which is a 6502 based core optimized for FPGAs from Lattice.

- **PIC16 based soft-cores**. PIC is a series of microcontrollers developed by Microchip Technology, based on the PIC1650 by General Instrument's Microelectronics Division. Since its first release in 1976, the microcontrollers from the PIC series have gained a remarkable acceptance in both the industrial field and the hobbyists community. One of the most remarkable versions is the PIC16, which is an 8-bit minimalistic design with a Harvard RISC architecture. Aspects like its reduced price, wide availability, extensive user base, large documentation and the free development tools available have helped to increase its relevance. This popularity has motivated the development of a number of PIC16 based soft-cores, many of the available in OpenCores: RISC 16F84, PPX16 MCU (stable), ClaiRISC (stable), MINIRISC core (stable), RISC5x (stable), risc16f84 (stable). In addition, several soft-cores based on the PIC16 architecture have been proposed in the literature, such as the CQPIC [98], the UMASScore [45] and the microcontroller from [39]. In [99], a PIC16F84A based FPGA design was presented, which improves the power stability over the commercial microcontroller.

- **8051 based soft-cores**. In 1980 Intel released the 8051 microcontroller based on the previous 8048, 8051 and 8052 units. Its 8-bit Harvard architecture has also been a reference for several soft-core implementations. The most relevant IPs from OpenCores are: T51 MCU (stable), Lightweight 8051 compatible CPU (beta), Turbo 8051 (beta) and 8051 core (alpha). In [100], the LP805X, a low power, modular and adaptable implementation of the 8051 was presented.

- **AVR compatibles**. The AVR microcontroller developed by Atmel is another reference design for various soft-core implementations. Initially designed to execute C code, its simple RISC Harvard architecture and its ease of programming are the main reasons for its popularity. OpenCores provides several open-source soft-cores: AVR Core (stable), AVR HP (stable), Hyper Pipelined AVR Core (stable) and AVRtinyX61core (beta).

- **Other soft-core processors**. A large number of additional soft-core implementations are available based on other architectures or designed from scratch. The most remarkable IPs are: the DSPuva16 [34] by the University of Valladolid, the MicroSimplez based on the architecture presented in [101], the MSL16 proposed in [102] and the DSP block based iDEA soft-core processor developed in [103].

2.2 SRAM Based FPGAs

Field Programmable Gate Arrays (FPGAs) are one of the most interesting platforms for implementing soft-core processors, thanks to their flexibility, performance, low-cost design and short time to market. Furthermore, their reconfiguration capability makes them suitable for achieving flexible designs, capable of re-adapting in the field in response to changes and correct possible issues. This provides high reliability and obsolescence mitigation. For these reasons, the platform selected in this work is a SoC device that includes an SRAM-based FPGA. The following section summarizes the most relevant aspects of FPGAs, focusing on the SRAM-based FPGAs and the Zynq device used in this work.

2.2.1 Introducing Programmable Logic Devices

FPGAs are Programmable Logic Devices (PLDs) utilized in electronic designs to implement reconfigurable digital circuits. Unlike monolithic integrated circuits that integrate fixed designs, PLDs are manufactured with an unsettled function. This feature makes possible developing customized hardware designs that couple closely with requirements of the application.

Before the rise of PLD technologies, the only available programmable devices where memories, such as ROMs (Read Only Memories), PROMs (Programmable ROMs), EPROMs (Erasable PROMs) and EEPROMs (Electrically Erasable PROMs). Despite they were a valuable alternative to store data, they presented several drawbacks, such as a slow operation (compared with dedicated logic circuits), unreliable asynchronous function, high power consumption and high costs. Besides, due to the lack of input/output registers they were not a stand-alone alternative to implement sequential logic designs.

Due to the described limitations, there was a significant need for new approaches. This is the reason why the XC157 Multi-Gate Array (released by Motorola in 1968) depicted in Figure 2.5 (obtained from its data sheet) was a milestone in the history of programmable logic devices. This first programmable logic device was a simple gate array with 30 input/output pins and 12 gates. In the following years several devices arose, such as the TMS2000 from Texas Instruments Inc., the first erasable PLD developed by General Electric, the DM7575 (a mask-programmable logic array) from National Semiconductor or the MMI 5760 (programmable associative logic array) by General Electric.

Since these first release the programmable logic technology has experimented a huge evolution offering a wide range of devices. Despite the complexity and the

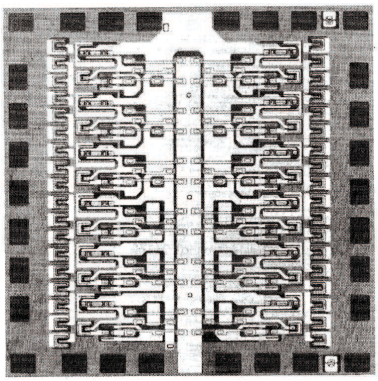


Figure 2.5: Picture of a silicon array of the XC157.

variety of available technologies, programmable logic devices can be divided in three major families: Simple Programmable Logic Devices (SPLDs), Complex Programmable Logic Devices (CPLDs) and Field Programmable Gate Arrays (FPGAs).

SPLDs are the simplest, smallest and cheapest type of PLD. SPLDs are mainly comprised of a small number of interconnected macrocells, which are composed by different elements, like flip-flops or combinatorial logic. Due to this small number of macrocells they provide a limited logic capacity. The concept of SPLD gathers several type of devices, such as Programmable logic arrays (PLAs), Programmable Array Logics (PALs), Generic Array Logics (GALs) or Field-programmable Logic Arrays (FPLAs). The technologies for the majority of SPLDs are either fuse or non-volatile memory cells (EPROM, EEPROM, Flash, etc.).

CLPDs can be considered as single-chip programmable devices containing a structure of several SPLD-like modules. They usually have a fast pin-to-pin delay that, once programmed, can lock their design. The structure of these devices consists of macrocells that contain the fabric logic with a sea-of-gates, a switch matrix and a functional block. CPLDs are considered coarse-grain devices. The configuration of all the resources is stored in its own (on-chip) memory. Although the number of gates is considerably larger than in SPLDs, it is a moderate quantity when comparing with other technologies. Due to this limited complexity they are generally used for glue logic applications. Bearing in mind the manufacturers and device families available, the CPLD's technology can be based on EPROM, EEPROM, Flash, or SRAM cells. The concept of CPLD brings together various devices like

Erasable Programmable Logic Devices (EPLDs), Simple Programmable Logic Devices (SPLDs), etc.

At the beginning of the 1980's there was a gap in the digital electronic business. On the one hand, they were configurable SPLD and CPLD technologies that provide a fast designing but couldn't implement large or complex applications. On the other hand, ASIC technology was able to support highly complex designs that required an extremely expensive and complicated design process. In order to fulfil this gap Xilinx Inc. released the XC2064, in 1985, which is considered the first commercial FPGA device. Figure 2.6 (obtained from its data sheet) shows the entire block diagram of the XC2064, which was based on the CMOS technology, that also used SRAM cells for configuration functions.

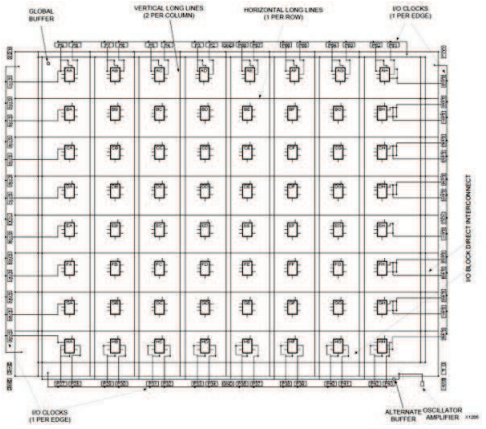


Figure 2.6: Block diagram of the Xilinx XC2064 logic cell array.

Despite that originally FPGAs started competing with CPLDs sharing similar application scopes, thanks to their evolution, FPGAs increased their capabilities and speed taking over new application fields. Nowadays, FPGAs are utilized to implement complete and autonomous SoC designs. Basically FPGAs are integrated circuits that can be configured by the final user after the manufacturing process. They are fine-grain devices, since they are composed by a large number of small logic blocks that contain distinct logic resources. Those logic blocks are distributed following an array structure and are interconnected with reconfigurable connections. Due to this FPGAs offer higher complexity and versatility levels. Distinct architectures are used to manufacture FPGAs. Some of the most relevant are: the CMOS based and one-time programmable Antifuse FPGAs widely used in fault tolerant space application, the CMOS and static memory

based reprogrammable SRAM FPGAs and the CMOS and Flash-erase EPROM technology based Flash FPGAs.

While some of the first technologies, such as Fuse, PROM and EPROM have become obsolete, new technologies are being investigated. Remarkable technologies are PCM based non-volatile SRAMs [104, 105]. Nowadays, thanks to its high capability and relatively reduced cost the prevalent technology are the SRAM based FPGAs.

FPGA market is facing a relevant globalization process. This process in which some enterprises acquire or merge with competition companies and others simply cease activity: As a consequence it has generated major changes in the FPGA market. For instance, Tabula shut down its activity on 2015 or Actel Corporation was acquired by Microsemi Corporation on 2010. Nevertheless, the most remarkable change came when, in 2015, Intel acquired Altera Corporation, one of the two historical leaders of the industry. Xilinx Inc., the other principal vendor continues growing and overtaking Intel, and hence leads the market. In addition to the two majors, different vendors provide FPGA devices, such as Achronix Semiconductor, Lattice Semiconductor Corporation or e2v. QuickLogic, which previously backed away from the FPGA market, has returned with the ArcticPro eFPGA IP.

Nowadays, Xilinx offers a wide spectrum of products to address different range of system requirements demanded by the market. In this way, Xilinx provides a multi-node product portfolio, which bearing in mind the different manufacturing technologies can be divided in four series:

- **Spartan-6** (45 nm): Highly optimized FPGAs in terms of cost and size.
- **7 series** (28 nm): They gather a wide range of devices, starting from low cost to high-end FPGAs.
- **Ultrascale** (20 nm): High integration level and routability improvements.
- **Ultrascale+** (16 nm): The latest architectures, one step further in integration.

Despite Ultrascale devices are the high-end product of Xilinx, 7 series devices offer the best cost-efficiency trade-off. Gathering some of the most remarkable products of Xilinx catalogue, 7 series provide solutions for a wide range of design requirements.

2.2.2 Architecture of 7 Series Devices and Zynq-7000 All Programmable SoC by Xilinx

Xilinx 7 series include four FPGA families (Spartan-7, Artix-7, Kintex-7 and Virtex-7) and the Zynq-7000 all programmable SoC whicht are based on a 28 nm high-k metal gate process technology. Each device of the different families targets a particular market segment. They offer distinct alternatives in terms of price range, form factor, high-end connectivity bandwidth, logic resources capacity, signal processing capability, etc. The technical evolution of 7 series devices has lead to considerably enhance system performance with 2.9 Tb/s of I/O bandwidth, obtaining capacities of up to 2 million logic cells and cutting power consumption by 50% compared with previous generations [106].

Despite that in many cases the synthesis tool decides and assigns the resources without needing any action from the designer, it is advisable to understand some concepts of the architecture and logic resources of FPGAs to take advantage of its full potential.

As Figure 2.7 shows, the architecture of 7 series devices, like most FPGAs, is based on a logic block array structure interconnected with the general routing network via switch matrices and surrounded by a periphery of I/O blocks. This reprogrammable network of signal paths is called routing. Due to the variety of resources and the interconnection options available, in most designs several routing alternatives that may produce different results in terms of efficiency or performance can be obtained. The complexity of the designs and the interconnection features require the usage of routing and placement tools. These tools provide optimized routings by using various analysis and algorithms. Although that in some cases it can help to understand and improve the design, generally is not necessary to acquire a specific knowledge about interconnection features. Usually the routing process is transparent to developers.

7 series FPGAs include several dedicated resources and primitives, some of the most relevant ones are introduced in the following lines.

Configurable Logic Block (CLB)

In Xilinx devices the fundamental building logic blocks used for implementing both, combinational and sequential circuits, are named Configurable Logic Block (CLB) [1]. As Figure 2.8 illustrates, each of the 7 series CLBs (which are identical to the CLBs of Virtex-6 FPGAs) contains two slices that are comprised by the following resources: four Look-Up Tables (LUTs), wide-function multiplexers, eight flip-flops and a dedicated high-speed carry logic. Depending on the

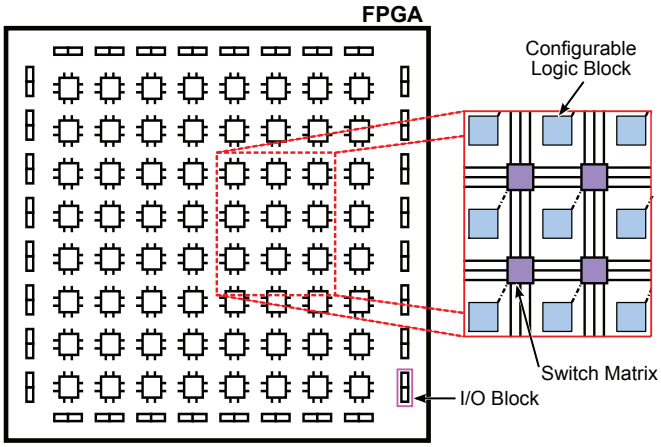


Figure 2.7: Arrangement of CLBs and switch matrices within the FPGA.

configuration, the slices utilize these resources to implement arithmetic, logic or ROM functions. Moreover, there are some special slices that can store data as distributed 64-bit RAM or as shift registers (a 32 bits register or a dual 16-bit register). While regular slices are named SLICEL, the special slices are called SLICEMs. A CLB can be comprised of two SLICEL or a SLICEL and a SLICEM. Around the 2/3 of the slices are SLICELs and the remaining 1/3 are SLICEMs.

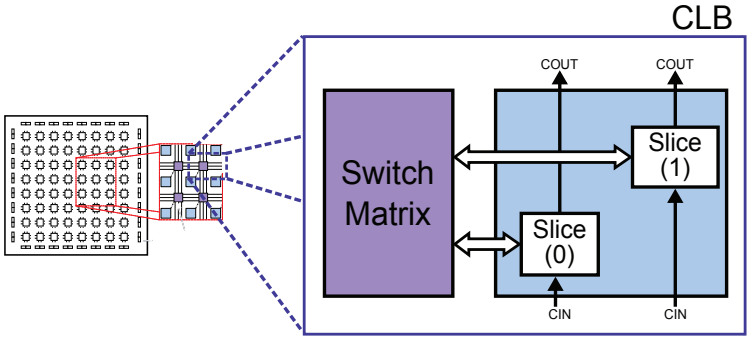
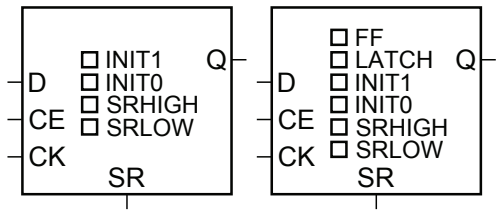


Figure 2.8: 7 series CLB example.

In 7 series FPGAs each slice contains four LUTs or logic-function generators. A remarkable feature of LUTs is that propagation delay through them is the same no matter the function implemented. In addition, each LUT can be set as

6-input and a single output LUTs or as 5-input and independent outputs dual LUTs (LUT5). The outputs of LUT5s can be optionally registered in flip-flops. Four of the eight flip-flops from each SLICE can be configured as latches, while the remaining flip-flops cannot be utilized. Apart from regular LUTs, each slice contains three additional multiplexers that can be utilized to combine up to four LUTs enabling functions of eight inputs.

As Figure 2.9 (obtained from [1]) shows each flip-flop of 7 series devices provides `SRHIGH/SRLOW` (SRVAL) and `INIT0/INIT1` (INIT) options. Determined by the HDL design, each option and its value are configured and stored in the bitstream. The example from Figure 2.10 shows a VHDL code for a flip-flop where both INIT and SRVAL values can be determined. While SRVAL values are controlled by the local `SR` control signal, INIT values are mainly utilized during the initialization after reconfiguration. However, INIT values can be updated performing a context capture with the `CAPTUREE2` primitive.



(a) Flip-flop block. (b) Flip-flop/LTACH block.

Figure 2.9: 7 series flip-flop and flip-flop/LATCH block symbols.

```
signal Q: std_logic:='1';   <-- INIT (INIT1 option)
.......
process (CLK)
begin
    if (CLK'event and CLK='1') then
        if (RST='1') then
            Q <= '0';       <-- SRVAL (SRLOW option)
        else
            Q <= INPUT_D;
        end if;
    endif;
end process;
```

Figure 2.10: VHDL code example of a flip-flop with INIT and SRVAL values.

Block RAM (BRAM)

When using FPGAs, apart from the external memories, memory elements can be implemented using dedicated Block RAM (BRAM) modules [2] or distributed general-purpose fabric logic. While distributed memories are more appropriate for storing small amounts of data and allow aster access, BRAMs are the *de facto* design selection if the application stores and manages lots of data. They are provided with customizable characteristics such as, data width and depth, single or dual input port, control ports and a protection based Error-Correcting Code (ECC). Hence, the designer should select the most adequate memory implementation depending on the specifications of the design. Otherwise, an improperly designed application with overestimated storage requirements would make the synthesizer reserve more memory space out of the FPGA logic slices than really needed.

BRAMs in Xilinx's FPGAs are very flexible. Each BRAM can be configured as a single or dual-port memory and can be set as a single 36 KB memory block or two independent 18 KB memory blocks that share nothing but the stored data. The 36 KB BRAMs are composed by a pair of 18KB BRAMs (top and bottom). Figure 2.11 (obtained from [2]) depicts the I/O ports of the RAMB36 primitive. Different 36 KB or 18 KB BRAMs can be chained together to form larger memories. Moreover, when using the dual port configuration, both ports can be implemented with different width. In this way, each port can be configured as 32Kx1, 16Kx2, 8Kx4, 4Kx9 (or x8), 2Kx18 (or x16), 1Kx36 (or 32) or 512x72 (or x64).

BRAMs are also equipped with an optional ECC protection based on a Hamming code. Each 64-bit-wide BRAM utilizes eight additional Hamming code bits to carry out single-bit error corrections and double-bit error detections during readings. The use of this feature can save logic resources from the FPGA, but also adds several drawbacks [107]. The ECC requires two clock cycles to complete the reading process, which in many cases may penalize the system with an extra clock cycle latency. Furthermore, when the data width is not a multiple of 64, a double-bit upset can point out errors in the unused bits.

CAPTUREE2 primitive

The `CAPTUREE2` primitive [3] (Figure 2.12 obtained from [3]), is a tool provided by Xilinx for 7 series devices. It offers the possibility to capture the state of user registers and store it in the bitstream by driving the `GCAPTURE` command. An alternative to the utilization of the `CAPTUREE2` primitive is to load the `GCAPTURE` by writing the 0x0000000C command in the CMD register through the bitstream.

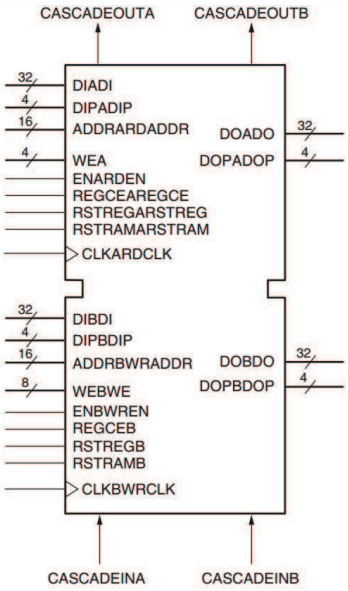


Figure 2.11: RAMB36 Block RAM primitive symbol.

By virtue of this functionality, current register (flip-flop and latch) values can be stored in the configuration memory by triggering the `CAP` input from this primitive. Once the `CAP` has been asserted, the content of registers is captured at next low-to-high clock transition. These register values are stored in the same configuration memory cell that programs the init state configuration of registers (INIT values). These register values can be read from the device along with all other configuration memory contents utilizing the readback function. Although the default option is to capture data after every trigger when transition on `CLK` while `CAP` is asserted, this primitive provides the `ONESHOT=TRUE` attribute to limit the readback operation to a single data capture. One significant limitation of this primitive is that it does not capture the content of SRL, LUT RAM and BRAM.

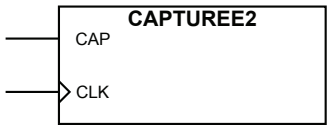


Figure 2.12: `CAPTUREE2` primitive symbol.

Another relevant limitation of this method is that it stores in the bitstream the content of all the registers from the device, even when only some particular registers are needed to be captured. Hence, the information of the registers that should remain unchanged would also be updated. In order to solve this problem, the resources of the device that must remain unchanged have to be protected by setting certain bits of the bitstream. This process requires a complex bitstream modification process, especially bearing in mind that Xilinx has not released enough information for 7 series devices bitstream.

To take advantage of the `CAPTUREE2` primitive, it is necessary to previously know the location of each data bit in the bitstream. The most straightforward way to obtain this information is to use Xilinx's software (Vivado, ISE, etc.) to generate the logic location text file (*.ll).

STARTUPE2 primitive

The `STARTUPE2` is a primitive that features an interface that relates the user logic resources with the status signal and the configuration logic control. This primitive can be utilized in an implementation to obtain user control over certain selected configuration signals during the operation. As Figure 2.13 (obtained from [4]) illustrates, it provides with different input and output ports, some of them being associated with the *startup sequence*. The *startup sequence*, which lasts a minimum of eight clock cycles, is an initialization process managed by an eight-phase sequence that is commonly performed after loading the configuration bitstream. The specific order of the initialization phases can be varied by the user through bitstream commands.

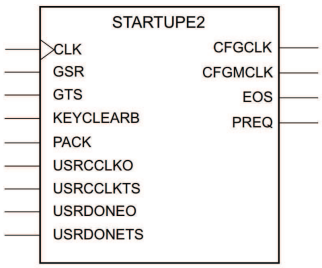


Figure 2.13: `STARTUPE2` **primitive symbol.**

One of the most relevant ports of this primitive is the `GSR` (Global Set/Reset). The `GSR` resets the device by driving the `GRESTORE` command. This active high input port leads an asynchronous set/reset signal that enables to initialize CLB/IO

flip-flops and DSP registers of the entire device in their initial state. The initialization values of flip-flops are determined by the `INIT0/INIT1` options (2.9) defined in the bitstream. Furthermore, INIT values can be updated with actual values of registers via `GCAPTURE` command. A remarkable aspect of the utilization of `GSR` signal is that it does not require general-purpose routing.

In 7 series FPGAS the `GRESTORE` command can be driven with the `GSR` signal of the `STARTUPE2` or loading the 0x0000000A command in the CMD register. `GRESTORE` command is commonly asserted during the initialisation process after configuring the FPGA to load the INIT values. Considering that the skew and release processes are done asynchronously, some flip-flops may be released in different clock cycles causing metastable events. For this reason, it is advisable to stop the clock before asserting the `GSR` signal and to wait until it spreads across the entire device. Another recommendable practice when using the GSR signal is to cluster tightly all the registers in order to minimize the path length.

Other Resources

Other remarkable resources available in 7 series FPGAs are the I/O banks, DSP slices, clocking resources and the ILA core.

- Devices from the 7 series feature high-performance and high-range I/O banks, each of them containing (with some exceptions) 50 SelectIO pins that can be configured for different standards. While high-performance I/O banks are designed for high-speed interfacing with low voltages (up to 1.8V), high-range I/O banks are tailored to support a variety I/O standards with higher voltages (up to 3.3V). 7 series FPGAs contain different combinations of these banks distributed along the periphery of the device. Moreover, additional I/O logic design primitives that provide specific features are available.

- An FPGA contains several clocking resources distributed (CLBs, BRAMs, DSPs, etc.) in different clock regions, which number varies depending on the device. Each clock region spans 50 CLBs from top to bottom up within an horizontal clock row. The management of regional and global I/O and clocking resources from 7 series devices enables to meet both simple and complex clocking requirements. This clock management is performed by utilizing the Clock Management Tiles (CMTs) that enable different functionalities, such as removing skews, synthesis of the clock frequency and jitter filtering. When utilizing clock management functions, it is advisable to avoid the use of non-clock resources, like the local routing. On the other hand, utilizing global clock trees permits clocking the synchronous elements

placed within the devices. 7 series FPGAs also support user clocks by using the dedicated *clock-capable* inputs available in each I/O bank.

- FPGAs usually include several dedicated Digital Signal Processors (DSPs), which are used to enhance the performance and save resources, like binary multipliers and accumulators, when implementing fully parallel algorithms. In 7 series FPGAs the DSP element is the DSP48E1 slice, which provides a small footprint, full adaptability, low-power consumption and high speed. In addition to signal processing applications, DSPs are useful to enhance several functions, like memory address generators, wide dynamic bus shifters, etc. Some of the most relevant features of DSP slices are: a 25x18 2's-complement multiplier, a 48-bit accumulator, a power saving pre-adder, a single-instruction-multiple-data arithmetic unit and a pattern detector.

- 7 series FPGAs integrate a customizable Integrated Logic Analyzer (ILA) IP core, Figure 2.14 shows its symbol (obtained from [5]). When utilizing the Vivado Desing Suite from Xilinx, this enables to monitor internal signals of an implementation. This logic analyser supports a number of features, such as boolean trigger equations or edge transition triggers. It also enables to utilize multiple probe ports that may be combined into a single condition or triggering. Bearing in mind that the ILA is a synchronous core, the clock constraints of the design are also included in the elements inside the ILA core. The ILA core utilizes resources from the FPGA to implement different features, such as BRAMs to store the processed data. This aspect has to be considered when analysing the utilization reports of implemented designs.

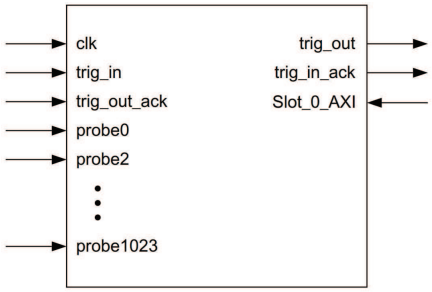


Figure 2.14: ILA core symbol.

2.2.3 Zynq-7000 All Programmable SoC

Zynq-7000 All Programmable SoC [6] is a remarkable example of the newest devices based on the SoC concept. It integrates an ARM-based processor system with the reconfigurable fabric logic of the Xilinx 7 series FPGAs. Two Zynq models are available: the single-core Zynq-7000S and the dual-core Zynq-7000. The main difference of the Zynq architecture over previous FPGAs with embedded hard processors (i.e. the PowerPC) is that, while previous devices were FPGA-centric models, Zynq is a processor-centric platform. Moreover, thanks to the huge number of traces (over 10.000) that connect the Processing System (PS) to the FPGA fabric, it provides the device with a wide bandwidth connection between the Processing System and the Programmable Logic (PL). Hence, it requires less infrastructure. Zynq also supports several features including two 12 bit analog-to-digital converters, three Phase-Locked Loops (PLLs), two JTAG debug ports, an integrated block for PCI express designs, low-power serial transceivers, etc. In this way, the Zynq SoC makes it possible to exploit the logic as an auxiliary resource that may be used to increase the performance of deployed applications. Zynq also offers different strategies to reduce the power consumption: shutting down the PL, dynamically reducing the clock speed in the PS or standby the ARM processor.

As shown in Figure 2.15 extracted from [6], Zynq is divided in two main regions (PS and PL) with separate power domains. Thanks to the power detachment of both regions, when required, the PL can be selectively power down. Since the PL uses the same FPGA technology from Xilinx 7 series devices, it includes all their previously presented logic resources (CLBs, BRAMs, DSP48E1, etc.). On the other hand, the PS allows to operate stand-alone programs and operating systems, such as Linux, and manages both boot and configuration processes. The PS can be divided into four main parts:

- **Application Processor Unit (APU)**. It consists of a single or dual-core ARM Cortex-A9 MPCore that boots immediately at power-up and can work with several operating systems without dependence of the programmable logic. It can operate in single processor and asymmetric or symmetric dual processor modes, and supports single and double precision floating point operations.

- **I/O Peripherals**. They support several industry-standard interfaces for external communication, including distinct General Purpose Input/Ouputs (GPIOs), two Gigabit Ethernet Controllers, two SD/SDIO Controllers, USB Controllers, two SPI Controllers, two CAN Controllers, two UART Controllers, two I2C Controllers and PS MIO I/Os. Up to 54 GPIO signals

are available for device pins routed through the Multiplexed I/O (MIO) and 192 GPIO signals communicate the PS and the PL via the Extended MIO (EMIO).

- **Memory interfaces**. It includes static and dynamic memory interface controllers. While the static memory controller enables to work with NAND and Quad-SPI flash, parallel NOR flash, and parallel data bus interfaces, the dynamic memory controller supports DDR2, LPDDR2, DDR3 and DDR3L memories.

- **Interconnection elements**. These elements provide communication between the PL, the APU, the memory interface and I/O peripherals via a non-blocking multilayered ARM AMBA AXI interconnection, which supports simultaneous master-slave operations.

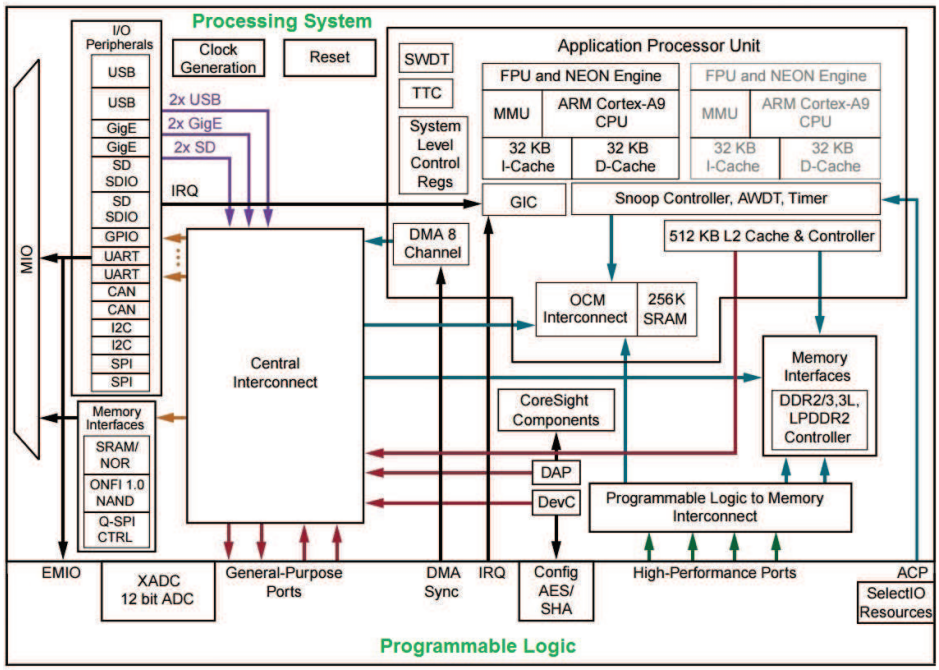


Figure 2.15: Diagram of the functional blocks that constitute the Zynq-7000.

Zynq Reset

The system reset process is a sequence that initializes the system and executes the First Stage Boot Loader (FSBL) from the selected boot memory. This process provides the user with the possibility to customize the PS and PL. The Zynq supports several reset types whitin the PS. For instance, a peripheral reset that resets a subsystem controlled by software or a power-on reset which resets the complete system. These reset sources conform the system reset, which drives reset signals to each module and system. Various alternatives are available to initiate a reset process:

- A hardware reset driven by the system reset signal (PS_SRST_B) and the power-on reset signal (PS_POR_B)
- A a software reset able to generate both system-level or a sub-module reset.
- A reset generated by the JTAG controller, which can reset both the entire system or a debug portion of the PS.
- A reset generated by the three watchdog timers available.

Xilinx also provides the `LogiCORE IP Processor System Reset Module` that enables to reset the complete PS, including the processor and peripherals. This core allows to customize several parameters by enabling or disabling different features in order to adapt it to user's specifications.

Zynq Boot

The boot of Zynq-7000 devices is a two-stage process managed by the PS. It enables to choose between a non-secure booting or a secure booting (JTAG disabled) that supports 256-bit AES, 256-bit SHA and 2048-bit public key decryption/authentication. During the booting process the ARM (or one of the ARMs) reads the boot program from the on-chip ROM, executes it and copies a FSBL code from of the flash memories (or downloaded through JTAG) to the on-chip memory. FSBL boot code can be entirely controlled by the user, enabling the customization of the boot code. After loading the FSBL, it is executed by the ARM, providing the possibility of loading the bitstream to configure the PL.

Reconfiguration on the Zynq

One of the most remarkable features of 7 series FPGAs, like most SRAM-based FPGAs, is that they can be reconfigured dynamically (while system running)

[108]. The configuration and reconfiguration processes are done by loading specific configuration bit stream files, named configuration bitstreams or .BIT files. The reconfiguration process can be performed by external systems (e.g. a microprocessor or a PC) or the FPGA can load bitstreams himself from an external non-volatile memory module.

Xilinx provides two data paths connected to special configuration pins to configure its 7 series FPGAs: the low pin demanding serial datapath and the parallel datapath (8-bit, 16-bit, or 32-bit) that provides higher performance and access to several standard interfaces. The pins from these datapaths serve as an interface for different configurations: master and slave serial configuration, master and slave SelectMAP parallel configuration, JTAG/boundary-scan configuration, master serial and byte peripheral interface flash configuration. Moreover, the Zynq can reconfigure itself by the programmable logic, or by any other processing system through the device configuration interface (DevC). This interface is supported by a dedicated DMA controller capable of transferring bitstreams from an external memory through the Processor Configuration Access Port (PCAP). The 7 series FPGAs also have an Internal Configuration Access Port (ICAPE2) [3], which provides the user logic with access to the 7 series FPGA configuration interface. In [109], both existing internal configuration interfaces, PCAP and ICAPE2, have been extensively evaluated in other to select the most convenient alternative. An additional architecture has been implemented in order to dynamically reconfigure the FPGA without the Processing System at the maximum bandwidth of 400 MB/s.

Besides reprogramming the entire device, most SRAM-based FPGAs feature partial reconfiguration, enabling to reconfigure a specific region while the rest of regions of the FPGA continue running. Due to this feature, it is possible to reduce both cost and area usage. Moreover, bearing in mind that a partial bitstream gathers only the information of the target reconfigurable zone, it contains much less information than a complete bitstream. Hence, the utilization of partial reconfiguration schemes allows to reduce the bitstream storage requirements and reconfiguration time.

Designing partial reconfigurable implementations is analogous to designing various complete reconfigurable designs that share certain resources. The first step is to designate a reconfigurable region, which has to be defined determining both a proper placement and an adequate size of the reconfigurable region. A proper floorplanning requires to consider several aspects, such as the resources demanded by the different designs to be loaded, the size and placement of the static zone or the other reconfigurable regions, etc. A wide number of 7 series FPGAs resources, like CLBs, BRAMs, DSPs or routing elements are available for reconfiguration

purposes. Nevertheless, several components like clocks and clock management blocks (e.g. BUFG, BUFR, MMCM and PLL), I/O and I/O related components, serial transceivers and some dedicated elements (e.g. BSCAN, STARTUP, ICAP and XADC) do not support reconfiguration. These non-reconfigurable elements must be placed in the static region of the device.

The regular way to configure an FPGA is to load at first a complete bitstream that contains one of the partial reconfiguration designs. After configuring the entire device, the partial bitstream can be loaded to partially reconfigurable regions, while the rest of the FPGA remains uninterrupted. After reconfiguration, it is advisable to initialize reconfigurable modules in order to ensure a predictable starting situation. Another recommendable practice is to disconnect the reconfigurable and the static regions during the reconfiguration process by utilizing decoupling logic.

The partial reconfiguration can also be used to reduce power consumption by disabling certain region(s). In [110], partial reconfiguration is utilized to replace circuits during idle periods with power saving circuits. In a similar way, [111] proposes to utilize empty bitstreams to blank partially reconfigurable regions.

Figure 2.16 illustrates an outstanding benefit of the partial reconfiguration, which is the feasibility of adapting the design in the field by loading distinct circuits within the reconfigurable region. This aspect enables to improve the fault tolerance, accelerates the configurable computing and provides real-time flexibility making it possible to develop new techniques in design security.

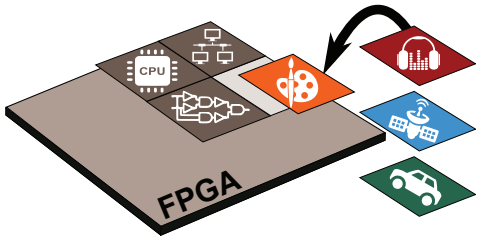


Figure 2.16: Replacing reconfigurable modules with the dynamic partial reconfiguration.

One of the most significant drawbacks of the reconfiguration process is its duration, which is limited by the maximum frequencies of the configuration interfaces. For instance, the PCAP's maximum frequency is 100 MHz. However, although its theoretical speed of reconfiguration is 400 MB/s, the real speed decreases because of the internal ARM interconnect architecture [112]. In addition, the duration of the reconfiguration process varies depending on the size of bitstream to be

configured. Hence, partial bitstreams are less time demanding than complete ones. The common duration of this process goes from microseconds to milliseconds [113]. A possible solution to reduce the reconfiguration time is to compress the bitstream, however it requires to previously process the bitstream and limits it usability. In [114], an alternative reconfiguration controller coined as ZyCAP was presented, which improves the reconfiguration throughput in Zynq when compared to standard methods. This controller allows overlapped executions, enhancing the system performance. ZyCAP can be used with soft-processors, but driver software modifications are required. In [112] an adaptive partial reconfigurable system to maximize the output performance is proposed reducing the reconfiguration time to 12% over a full configuration time.

2.2.4 Bitstream Structure of 7 series FPGAs

Bearing in mind that the hardware platform of this work is a Xilinx Z-7020 Zynq-7000 All Programmable SoC and that it makes use of its reconfiguration capability, some aspects of the configuration bitstream of the Xilinx 7 series family must be introduced.

The information around the bitstream is limited, especially since 1998, when the XC6200 series were discontinued [115]. Vendors have published scarce information about the bitstream composition of their devices, but the majority of them are still unknown. Thanks to various studies, the knowledge about the bitstream has continued growing over the last years [116, 117]. However, there is a lack of studies dealing with the new Xilinx 7 series devices.

In [118], a bitstream generator that does not use reverse-engineering was presented. This generator is able to imprint small changes onto existing bitstreams and to embed the bitstream generation inside the system that it targets. Although it does not support the full set of device resources, it is a good example of the potential of this field. In [119], a bitstream manipulation tool for Xilinx FPGAs based on C language and capable of creating partial bitstreams was presented. Although this work was implemented in a Xilinx Virtex II Pro FPGA, several aspects of the structure of the bitstreams exposed in this contribution are useful to understand the bitstream's composition of Xilinx 7 series FPGAs. Following this baseline, this section presents several interesting aspects of the 7 series bitstream based on previous literature, Xilinx's documentation [4, 120] and several off-line tests performed within for this research work.

The 7 series configuration memory is organized in frames that are tiled about the FPGA. A frame is the smallest addressable segment of the configuration memory and, for this reason, all operations have to act upon whole configuration frames.

The frame size is 101 words (32 bits), but may vary across families. Frames are organized with an interleaving mechanism, which imposes that adjacent bits do not belong to the same frame. This is a protection strategy against SEUs in consecutive bits, avoiding two errors in the same frame, but also making more difficult the inference of the location of the data content in the bitstream. There is a correspondence between the information in the bitstream and the tile map of the device. Therefore, the number of frames of the bitstream depends on the configuration array size and the additional options of the bitstream. In most cases of the FPGA's resources, the mapping of memory addresses to resources is unknown, because this relation is usually extremely complex. However, some specific resources, such as CLBs, DSP blocks and BRAM columns are represented explicitly in the bitstream addressing. The efforts in this work have been focused on understanding the distribution of data in BRAM columns.

The bitstream file starts with a header consisting of a concatenation of different command words. All commands are executed by reading or writing the different configuration registers of the FPGA. The commands can send two packet types: Type 1 and Type 2 (used to write long blocks). Both packets consist of a header followed by a specific data section. This data section contains the number of 32-bit words specified by the word count portion of the header. Table 2.1 shows the format of the header of each packet type. The opcode field defines the function mode, which can be `read, write` or `no operation`. The register address field in the Type 1 packet indicates the address of the target register of the operation at hand. The Type 2 does not use the register address field, because it must always follow a Type 1 packet and it uses its packet address. Between the available 20 registers, the most interesting ones for this research are the Cyclic Redundancy Check (CRC), the Frame Address Register (FAR), the Frame Data Input (FDRI), the Frame Data Output (FDRO) and Command (CMD). The data stored in the CRC register is used to carry out error detection over the bitstream data. The FAR is used to set the address of the FPGA logic over which the reading or writing process is performed.

Table 2.1: Packet header types (R: reserved bit; x: data bit).

(a) Type 1.

Header Type	Opcode	Register Address	Reserved	Word Count
[31:29]	[28:27]	[26:13]	[12:11]	[10:0]
001	xx	RRRRRRRRRxxxxx	RR	xxxxxxxxxxx

(b) Type 2.

Header Type	Opcode	Word Count
[31:29]	[28:27]	[26:0]
010	RR	xxxxxxxxxxxxxxxxxxxxxxxxxxx

As shown in Table 2.2, the FAR register is split into 5 parts: block type (CLB, I/O, CLK, BRAM content or CFG_CLB), top/bottom bit, row address, column address and minor address. The FDRI register is used to configure frame data at the address specified in the FAR register. The FDRO register provides readback data for configuration following the address stored in the FAR register. Finally, the CMD is used for the commands, such as, RCRC (Reset Cyclic Redundancy Check), START (FPGA initialization) and DESYNC(end of configuration to desynchronize the device).

Table 2.2: Frame Address Register (FAR) format.

Address Type	Bit Index
Block Type	[25:23]
Top/Bottom Bit	22
Row Address	[21:17]
Column Address	[16:7]
Minor Address	[6:0]

The first words of the bitstream header define the boot sequence and provide additional information for synchronization purposes. The next commands specify the width of the configuration bus. Then, a set of optional commands specify parameters related with transfer bitrate, encryption, authentication and CRC check. The last part of the header defines the address in the configuration memory and the amount of words that are going to be loaded. The header is followed by the body, which contains the information that is stored in the device. The structure of this is usually not specified by the FPGA manufacturers, and the only way to obtain information is reverse engineering. Finally, the trailer defines the finishing sequence, which contains CRC verification, the start command (optionally) and finally the DESYNQ command to end the communication. Along the entire bitstream many no operation (0x20000000) words are included. After performing several tests it has been observed that the data content of BRAMs cannot be written through the bitstream if the BRAM segment contains these 0x20000000 words. Due to this, is can be assumed that these 0x20000000 words act as mask words to protect/unprotect BRAM content against rewriting.

Bearing in mind that the FPGA configuration can be complete or partial, complete or partial bitstreams are accordingly produced. Both options share many common aspects, but several particular differences deserve special attention. To begin with, the complete bitstream uses the `START` command for starting the device, which it is not necessary for its partial counterpart. This occurs because the partial reconfiguration is usually performed dynamically, i.e. while the device is running. In the case of a complete bitstream, the FAR is filled with 0's and the number of words stored in the FDRI register corresponds to the total words of the entire bitstream. The FAR and number of words of the bitstream are strictly

related to the defined reconfigurable region. As a result, if the reconfigurable region is deployed over no consecutive zones following the address of the FAR, the bitstream is divided in different fragments. Each fragment is defined by its particular FAR and word amount. Therefore, the partial bitstream indicates the FAR and word amount of each resource column used in the design. For this reason, partial bitstreams have been used in this work to extract information about the data content of BRAMs.

The word amount stored in the FDRI register is a multiple of the frame size. Each frame is related to the configuration bits and initial values of a particular column of resources. Each resource may have a different number of frames, and these values may also vary among device families. In the case of the Z7020 device used in this study, the total number of frames is 10008. After several partial configuration experiments it has been observed that each column of slices includes 36 frames, each column of DSPs contains 28 frames and each column of BRAMs embeds 28 configuration frames and 128 data content frames.

The Z7020 device has a total number of 133 CLB columns, 11 DSP48 columns and 14 BRAM columns. Hence, the total number of configuration frames in this device is 5502 and the total number of BRAM data frames is 1972. Given the total number of frames of the complete bitstream, it can be assumed that the remaining frames are used for the configuration of other resources, such as IOBs or BUFGPs. However, since these resources cannot be used in partial bitstreams it is complex to determine the exact number of frames that they possess.

Xilinx enables to generate a logic location text file (*.ll) which contains the location of each user data bit in terms of relative offset to a frame of the entire design. In this way, knowing the location of a particular region, it can easily find the nets of the design that belong to the partial region. This file can be generated with the software provided by Xilinx, such as ISE or VIVADO.

2.2.5 Managing Data Content by Utilizing the Bitstream

Memories are essential elements in SoC designs as a standard way to store data on a temporary or permanent basis. This data may have different purposes. In SRAM FPGA based systems the most common functionality is to store information of specific applications, entire programs or sequences of instructions, program state information and/or configuration information of the device. As it has been introduced, when implementing memory elements in FPGAs the two alternatives that can be used, which are the BRAM resources and the distributed general-purpose logic fabric. In the case of soft-core processors, while program and data memories are commonly implemented as BRAM structures, the small

memory elements, such as the registers, are implemented by utilizing distributed fabric logic.

The standard way to access and manage data in memory modules is to use their input ports, such as `data output`, `data input`, `write-enable` and `address`. This access method requires a control mechanism to manage the inputs and read the outputs in a coordinated fashion. This is often accomplished by a memory controller, which can be implemented in different ways, such as using soft-core or hard processors, specific IP cores or custom FSM based modules. Frequently, additional elements are also required, like bus interfaces or auxiliary memories to store processed data. The implementation of all the above elements demands the usage of a variety of resources of the FPGA, further increasing the resource overhead. Besides, if it is required to read or write large amounts of data, resources committed to storing such data will be blocked and unavailable for other purposes. For example, it is not possible to read the data from the first two memory addresses while the last ten data addresses are being written.

Due to the limitations of standard data management, there are several scenarios where new data management alternatives could improve some applications or solve different existing issues. Bearing in mind that the user data is also stored in the configuration bitstream, the idea of using this bitstream to manage data content is an attractive alternative.

The impossibility of modifying data content of ROM memories, widely used as program memories in soft-core processors represents another relevant issue since this functionality may be needed in order to change the purpose of a processor or to recover after an SEU in the program memory. The regular way to modify the content of a ROM memory in an FPGA design is to re-synthesize and re-deploy the entire design with the new data onto the FPGA. In the best case scenario, if the program memory has been implemented as a reconfigurable partition, only this part should be re-synthesized and re-implemented. Following this approach, the program memory is implemented as a RAM in [39], and a dedicated IP core loads new memory content during the system operation by using a serial interface. However, this scheme increases requirements in terms of logic resources, and an upset in the input port of the memory may lead to changes in the memory content, increasing the probability of malfunction due to SEUs. Thus, modifying the program memory could be a good application example for a bitstream based data content management.

Another scenario where a bitstream based management can be an interesting alternative is the damage of the interfaces on memory resources. Thus, as a consequence of an SEU, an error can disable or affect to relevant ports of the memory, which it is likely to provoke permanent damages. Moreover, if ports

such as, address, reset, clock or data output are affected by the fault, it could make it unsuitable to recover the information stored in memories when resorting to conventional, off-the-self methods. This scenario can be a critical issue when the memory has not been hardened and the information stored has a special operational relevance. The extraction of such information by utilizing the data content stored in the bitstream could allow recovering the information in a straightforward manner.

Considering the direct relation between the integrity of the user data and the reliability, several fault tolerance related strategies could also take advantage of a bitstream based data management.

Bearing in mind the potential benefits that can be obtained with this strategy, there is a scarcity of investigations around this strategy. The lines below present the most significant works proposed in the literature, especially focusing in Xilinx devices.

When reading and writing back user data from one memory block to another, two main cases can be found: Homogeneous and heterogeneous architectures. While homogeneous memory modules share main characteristics, such as size, shape, and used resources (with only different fabric locations), heterogeneous modules support distinct features due to the usage of different resources and granularity levels. Data management utilizing the bitstream is more straightforward in the case of homogeneous implementations, since it mainly requires simple bitstream manipulations in order to relocate the data. Nevertheless, the utilization of heterogeneous architectures restraints the designing process.

In any case, extracting and writing user data content through the bitstream requires certain knowledge of its structure. Thus, they are device dependant strategies. While devices from the 7 series by Xilinx share the majority of the characteristics of the bitstream structure, many of them are different in previous FPGA series from the same vendor. Thus, despite in several cases the flows and the main ideas can be adapted to different devices, the developed approaches are commonly related with a particular FPGA vendor, a particular FPGA series or even to a particular device.

With the latest FPGA series (Virtex-4, Virtex-5, Virtex-6, Spartan-3A, Spartan-3AN, Spartan-3A DSP, Spartan-6 and 7 Series), Xilinx offers the possibility to use the Data2MEM [121] data translation software to initialize BRAMs. Among other features, Data2MEM can replace the contents of BRAMs in configuration bitstreams in a straightforward fashion, without requiring any implementation tool. However, this application undergoes several limitations. One of the most relevant is that this software must be executed by an operating system (Windows

or Linux). In addition, this program requires previously generated files to create the output files, such as BRAM Memory Map (BMM) files or Linkable Format (ELF) files. The configuration bitstream files to be updated by the tool must be created without compression and/or encryption. To sum up, this is a quite complex solution, not supported for partial bitstreams, feasible only for data writing in BRAMs and not for reading. It is therefore not proper data management approaches for autonomous standalone systems based on partial reconfiguration.

Jbits [122] was one of the first tools reported to be able to modify the bitstream, including the data content. It consisted on a set of Java classes that provided an application program interface into the configuration bitstream for XC4000 and Virtex families by Xilinx (by utilizing the SelectMAP interface). One of the biggest advantages of this approach was that it did not require additional hardware structures. Nevertheless, it came with a poor data efficiency. This was because it required to read the entire bitstream, while the user data only uses a relative small percentage of the bitstream. This tool became obsolete and it is not suitable for 7 series devices.

The work presented in [7] is a step forward towards bitstream based data content management. As Figure 2.17 extracted from that work describes, the approach proposed focuses on saving an relocating the context a of a soft-core processor, by extracting (and writing back once processed) the bitstream through the SelectMAP interface. In this process the entire configuration data is not read back, but only the frames that contain the target information. This context extraction is done while reading the configuration data and requires to stop the clock of the particular hardware task to prevent chances during the read process. In [123], a similar approach that proposes a partial displacement defragmentation algorithm for heterogeneous reconfigurable systems was presented. However, these approaches have several drawbacks. The most remarkable one is that, since they are only usable with the old one-dimensional partial reconfiguration FPGA families from Xilinx, they cannot be utilized in the newest devices. In addition, they require high processing effort (the use of tools, such as a configuration manager, *PARTBIT* software and the *REPLICA* filter) and to use a complex database to store the placement of each data bit.

Further approaches focused on dealing with newer Xilinx devices that support two-dimensional partial reconfiguration can be found. The approach presented in [61] is capable of capturing, storing and copying the content of flip-flops within a Virtex-V FPGA device on a particular reconfigurable region by utilizing the bitstream in combination with the `STARTUP_VIRTEX5` primitive, the `GCAPTURE` commands and the ICAP interface. In a later work [8], the previous approach was extended to enable to copy the content of flip-flops between heterogeneous

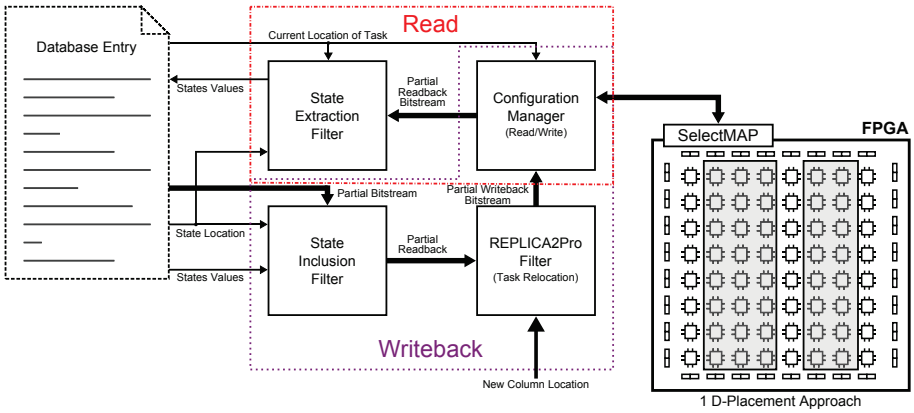


Figure 2.17: A relocation alternative.

reconfigurable regions by processing captured bitstreams. In [124], the same authors present an implementation of preemptive hardware multitasking for partially reconfigurable FPGAs that enables configuration prefetching and reuse. This approach based on their previous works reduces configuration overheads, improving the system performance.

As Figure 2.18 obtained from [8] describes, the strategy of the mentioned works follow different steps. The first is to capture the state of the flip-flops by using either the `GCAPTURE` command or the `STARTUP_VIRTEX5` block. It is advisable to stop the clocks before capturing the data in order to ensure that the register system is in a stable state. In a next step the captured data is stored in an external memory. After, if it is necessary (for instance due to data relocation purposes), this information can be processed creating a new partial bitstream. This bitstream processing can be a complicated task bearing in mind the complexity of the bitstream structure, especially when a large number of registers are utilized. This processing is mainly done by utilizing the information from the logic location text file (*.ll). The next step is to download the bitstream to the FPGA, which can be done in any moment because the changes don't take effect until the `GSR` signal from the the `STARTUP_VIRTEX5` primitive is trigged. An alternative to the `STARTUP_VIRTEX5` primitive is to toggle the `GSR` signal by using the `GRESTORE` command.

The main challenge of using this strategy is that by default the `GSR` signal copies all INT0/INT1 bits to all the flip-flops, changing the content of all the registers of the device. Since, in many application only some particular register are needed to be affected by the context restoring process an additional strategy has

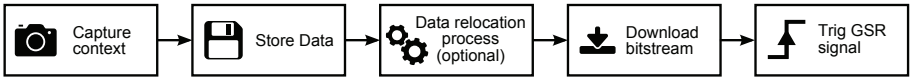


Figure 2.18: Flow chart of a context capture and restoration alternative.

to be followed. Every stack of configuration resource is related with a specific bit, commonly named mask bit. `GCAPTURE/GRESTORE` commands only affect the state of the flip-flops not marked with the mask bit. Each column is related to a single frame in the mask column. In this way, for every mask frame set, the entire related column in the clock region of DSP/CLB is protected/unprotected. Further information of this bit (for Virtex-V devices) can be found in either [30] or in [8]. This feature makes it possible to protect/unprotect specific areas determining which resources are to be modified. In this way, [8, 61] follow the strategy depicted in Figure 2.19 (extracted from [8]). Since the entire FPGA design is unprotected by default, in a first step the entire FPGA design is protected. When context is needed to be saved the target region has to be protected before capturing the data. After, capturing data the target region can optionally be protected, in any case before restoring the context the target region has to be unprotected. After finishing context saving/restoring processes the target region can be protected in order to avoid undesired changes in future startup sequences.

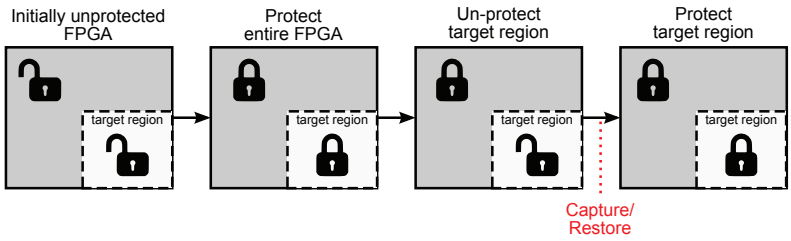


Figure 2.19: Flow chart of an FPGA protection and unprotection alternative.

In general terms, these published approaches present some common drawbacks. First, they require a controller module and a memory block to store the read data. In addition, considering that the reconfiguration is a time demanding process, the price to pay when using these strategies is a low availability. Another limitation of these approaches is that they require to disable the CRC feature of the generated bitstream. This is because due to the modification of the bitstream content the original CRC value would not be valid, making unsuitable to use it for the reconfiguration. Hence, the utilization of the CRC feature would require to generate the new CRC value which requires a high processing effort. Finally,

no approaches have been proposed to perform such applications in newer devices, such as, 7 series vy Xilinx.

In summary, despite of the presented disadvantages, bitstream based user data content management techniques provide an alternative access to data content without increasing the area overhead and without interfering the logic of the FPGA logic which, in many cases, can be an interesting way to overcome certain limitations of the classic memory management.

2.3 Radiation Effects on Soft-Core Processors implemented in SRAM FPGAs

Thanks to the continued innovation in the technology for developing and manufacturing electronics, the integration level has been increasing, enabling features, such as, high capacity, low-power consumption and fast devices. However, the high integration level comes with a drawback since it increases the susceptibility to induced faults [27]. These faults are mainly generated by contaminant alpha particles [28] or ionizing radiation from protons and cosmic rays from outer space which interact with the atmosphere and generate subatomic particles that collide with solid-state devices and integrated circuits [125]. Alpha particles are caused by impurities (mostly Thorium and Uranium) in integrated circuit encapsulate compounds and by atmospheric neutrons that slam into the silicon.

The effects provoked by space radiation can be mainly divided into two categories [126]: Single Event Effect (SEE) and accumulative effects like Total Ionizing Dose (TID) or Displacement Damage (DD). SEE are the result from the prompt effects of a strike by a single ionizing particle in a sensitive region of the device. TID is caused by the gradual accumulation of ionizing dose deposition over a long time. Finally, DD is the consequence of an accumulation of crystal lattice defects lead by high energy radiation.

In the case of FPGA devices, the different technologies (SRAM, flash, and anti-fuse) present distinct susceptibility levels to radiation effects [127], mainly because of the specific characteristics of their switch technology. Nevertheless, due to the utilization of CMOS based logic modules they share some level of susceptibility to SEE and TID. The switch of the non-volatile anti-fuse and flash devices is completely immune to SEE, for this reason they are less sensitive to them than their SRAM counterparts. Regarding TID, while the switch of anti-fuse devices are immune to these effects, it is known that they affect SRAM and flash based FPGAs [128–130].

Due to its benefits as a platform for soft-core implementation described in 2.2, the scope of this work is the reliability of SRAM FPGA technology. Although both accumulative effects and SEE are a main concern for SRAM designers, SEE are the prevalent case of study in the literature [131–136]. A reason for this is that, although there is a risk of performance degradation and an eventual malfunction, the effects of TID and DD do not commonly lead to system-level failures (as long as the degradation does not surpasses system tolerances). In addition, as [137] states the effect given by TID neglected in the typical task cycle and environment. The mitigation of TID effects is typically done by using radiation hardened by process techniques and/or shielding. On the other hand the hardening by design in SRAM FPGA is quite expensive in terms of resource usage since the majority of switches should to be hardened.

Since the first detection in 1975 [138], SEEs have become a main issue in space applications [62, 83, 139–142] because when the devices work in this environment they are directly exposed to space radiation. Nevertheless, despite the fact that at ground level and avionics systems are less exposed to radiation effects due to the planet's atmospheric and magnetic radiation shields, they are also sensitive to SEE [143]. In fact, the first reference to deal with SEE, published in the early 60's [144], was related with microelectronics at terrestrial level. The detection of SEE in terrestrial applications [145] came few years after the mentioned first detection at space level. Since then, the concern on SEE has continuously been increasing due to several factors. One of them is that, thanks to their obvious benefits, the application fields of microelectronics also continues growing daily. In fact, trying to imagine application fields with no presence of electronics is a tricky task. In addition, several of these application fields, such as, medical, military, avionics, space or automotive. require high fault tolerance levels. Another relevant factor is the continuous technology shrinking, for instance, Xilinx and Altera have presented 16nm and 14nm devices [146, 147], respectively. This density increase demands lower polarization voltages, which increases the susceptibility to SEE, since the necessary energy to generate upsets for charged particles is also decreased [148] and a single particle strike can affect more than a single transistor at a time. The increase of working frequencies is another relevant aspect factor. Due to this speed boost, the delay levels of logic gates are becoming comparable to the duration of SEE, leading to a higher error propagation probability. On the other hand, studies like [149] by the NASA in 2009 or the newer ones [150], reveal that the intensity of cosmic beans is increasing.

SEE gather a number of effects caused high energy particles that have been classified in two main groups in the literature [126, 151, 152]: soft errors and hard errors. Figure 2.20 summarizes the different SEE types.

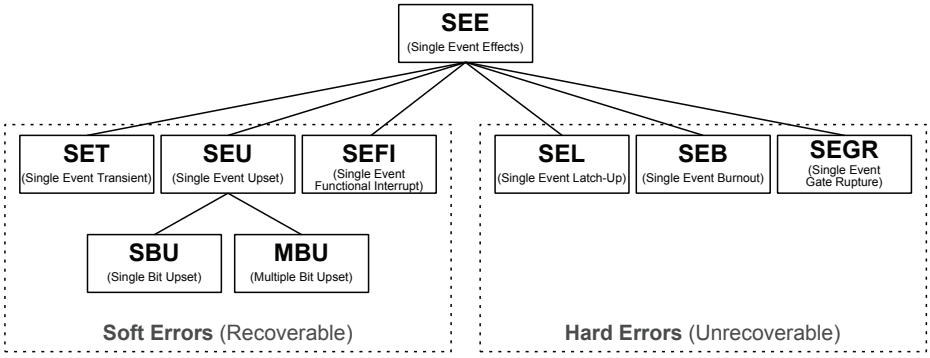


Figure 2.20: Classification of Single Event Effects

Soft errors are non-destructive upsets that can be self-corrected in time or that can be corrected by rewriting a memory element. It has to be mentioned that, although these effects do not cause a permanent damage in the device in which the effect occurred they can lead to permanent faults in the system in which that device resides. These effects can be divided in three subclasses:

1. **Single Event Upsets (SEUs).** They appear when high-energy particles provoke bit-flips in memory elements (flip-flop, latch, SRAM, or flash). Depending on the quantity of flipped bits they can be categorized as Multi Bit Upsets (MUBs) or Single Bit Upsets (SBU). Despite the fact that SBUs are known to be the most prevalent type of SEE, MUBs are also studied in the literature [153].

2. **Single Event Transients (SETs).** The effect caused by a transient is determined by its duration and magnitude, and by if the transient is latched turning it into an upset at system level. They occur when a high energy particle impacts FPGA's internal interconnection system inducing a current or voltage spike. Depending on when the impact occurs and the pulse-width the SET could be registered, and as a consequence, it may propagate thought the circuit and, eventually, provoke errors. When this latching of occurs the SET becomes an SEU. In the rest of scenarios the repercussion of these effects is negligible. The increase of clock frequencies increments the likelihood of an SET registration, and as a consequence, increases the probability of faults.

3. **Single Event Function Interrupts (SEFIs).** SEFIs cause a malfunction in the device, such as, reset or lock up. They usually are related with upsets in a bit or a register of device's control [154].

Hard errors are permanent or unrecoverable upsets. These types of SEE are also divided in three subclasses:

1. **Single Event Latch-up (SEL).** This is an induced circuit latch-up. An SEL can be either permanent or fixable by power cycling.
2. **Single Event Burnout (SEB).** An SEB occurs when a high-energy particle strikes a transistor source causing a short-circuit due to forward biasing. An SEB can lead to overheating related issues.
3. **Single Event Gate Rupture (SEGR).** An SEGR is the damaging burnout of a gate insulator in a power MOSFET.

Xilinx states in [151] that their FPGA devices are not susceptible to latch-up and gate rupture caused by neutron radiation. In addition, Xilinx space-grade parts are said to be immune to latch-up from heavy ions as well.

Among all the SEE types, SEUs are the prevalent source of errors in SRAM FPGAs [73, 155], especially those produced by radiation [151]. SEUs can affect not only user memory bits but also configuration bits in FPGA based design. Bearing in mind that the configuration memory determines the entire functionality of devices (interconnection and configuration of resources, data content, etc.), SEUs in configuration memory can provoke a wide variety of issues in designs. They can even affect the power consumption of the device [156]. Hence, they are a special concern. Figure 2.21 shows an example of an SEU in the configuration memory affecting the functionality of the hardware implementation. SEUs in user memory are less likely to happen, because generally this data content is an order of magnitude smaller than configuration memory's content. However, the relative upset ratio is very much alike in both, user and configuration memories. Following this idea, in FPGA designs with big data module implementations or complex registers systems, the risk becomes higher. Despite that an SEU in user's data does not affect the functionality of designs, depending of the application both types of SEUs can lead to a critical malfunction.

Although most frequently SEUs affect to configuration memory, many bits of the bitstream are unused. In fact, it is considered that in a regular design less than 20% of the configuration bitstream bits are critical to designs' functionality [157]. The quantity of critical bit vary depending on different implementation factors of a design, such as, resource quantity, routing, type of resources utilized, designed application, etc. Nevertheless, the critical bit rate is usually a small number [158, 159]. In the majority of cases a bit-flip in configuration memory does not affect the functionality.

The Soft Error Rate (SER) in a relevant concept when evaluating fault tolerance

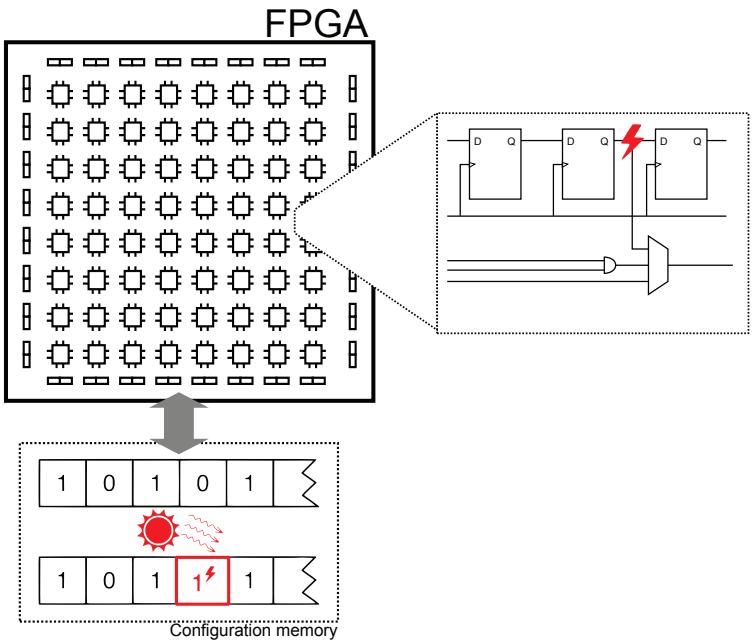


Figure 2.21: Configuration memory SEU example in an FPGA design.

of a design [160]. The SER is the rate at which an FPGA (or other electronic devices/systems) is predicted to undergo soft errors. The exact contribution of SEE to SER vary depending on the design. For instance, asynchronous designs are less susceptible to SEE than synchronous ones [161]. However, asynchronous versions of digital designs are normally 10 times slower and they need 10 times more resources. The SER is determined by two factors: Failure in Time (FIT) and Meant Time to Failure (MTTF), also known as Mean Time Between Failures (MTBF). As Equation 2.1 shows, FIT is the measure of failure rate in 10^9 device hours. On the other hand, MTTF defined in Equation 2.2 is the mean of the life allocation for the number of devices under operation or expected lifetime duration of an individual.

$$FIT = Fail\_quantity \cdot 10^9 \cdot (Device\_quantity \cdot Working\_hours) \tag{2.1}$$

$$MTBF(h) = \frac{Fail\_quantity \cdot 10^9}{Device\_quantity \cdot FIT} \tag{2.2}$$

In SRAM FPGAs the SER can be estimated performing fault injection campaigns, simulations or analytic approaches [162–165]. The Rosetta Experiment is the most remarkable exponent of SER estimation performed by Xilinx. This experiment was conducted in 2005 [166] and it linked two well-known techniques of estimating atmospheric neutron SEUs and undocumented real effects of atmospheric particles on electronic devices. Thanks to that, it correlated atmospheric upsets to simulated results, and to beam tests. Due to the publication of this article several previous works, such as, the JEDEC89A standard [167] were corrected. It also showed that the sensibility to atmospheric radiation is at least eight times lower than Boeing predicted [143]. The Rosetta Experiment continues performing a wide range of studies at ten different altitudes and analysing new trends of Xilinx devices [168].

Another aspect to be considered is the resource overhead since it is directly related with the fault tolerance. First, the more resources utilized in an FPGA implementation, the more resources are susceptible to induced faults. Secondly, the number of bits from the configuration memory sensitive to SEE increases with the resource overhead. In addition, more utilized resources means more resources to be hardened when hardening a design. In conclusion, the reduction of resource overhead is a highly recommendable practice in terms of fault tolerance.

Once the most significant aspects of the SRAM based FPGA technology have been introduced, the next chapter gathers the most significant alternatives to harden designs implemented in such devices, specially focusing on soft-core implementations.

Chapter 3

Hardening Soft-Core Processors Implemented in SRAM FPGAs

An ideal design should always work without malfunction. Nevertheless, in a real scenario, especially when dealing with SRAM based FPGAs, electronic designs are exposed to certain risks. A dependable design should try both, to avoid the impact off errors and to repair the produced ones. Related with those ideas, two concepts have to be remarked: availability and reliability. While availability is determined by the mean time to repair after a failure, reliability is determined by the design itself, the platform utilized and the operation-environment. It is important to obtain a proper balance of both concepts. For instance, a design with poor reliability level but with a fast repairing method most probably will meet its goals in presence of faults. On the other hand, a very reliable design that needs complex and time demanding repairing techniques perhaps will not achieve its goals after an error. Reliability is a valuable characteristic, but availability is what the final user is going to experience. In order to provide satisfactory reliability and availability levels for FPGA designs, an adequate fault tolerance level must be guaranteed. Due to this, increasing fault tolerance level of SRAM designs is one of the main objectives of this research work. Different strategies can be adopted to increase the fault tolerance level of FPGA designs in order to avoid possible negative consequences of induced faults.

The most obvious way to mitigate faults is to improve production aspects to

manufacture less susceptible technology. In fact, several manufacturers have developed hardened FPGA products [169–173]. Regularly these devices are regularly based on older technologies (i.e. anti-fuse FPGAs), offering significantly less resources and lower working frequencies at higher prices than the newest commercial FPGAs. An alternative strategy that is being followed is to use packagings with lower alpha particle content. Manufacturers like Xilinx have progressed from low alpha to ultra-low alpha materials. Nevertheless, these are not secure enough to ignore the package alpha contribution to the atmospheric alpha upset rate. Another fabrication aspect to improve fault tolerance is to provide higher susceptibility to SRAM cells utilized to store the configuration memory and the BRAM content. However, despite of all the efforts to improve the fabrication processes, the fast grown of configuration memories makes it a hard race.

Another alternative that can be utilized to harden electronic designs against both SEEs and TIDs is to protect them with shielding. Although that provides proper reliability levels, it is not practical for most applications since the amount of material required to build the shield is too expensive, e.g., as much as 30 meters of water for neutrons with high energy [174]. It also increases the weight and size of designs, which in many cases could be a significant limitation.

A further relevant way to harden FPGA based designs is trying to avoid using of unsafe design practices. Several examples can be found in [175], such as, avoiding use of asynchronous reset signals, not incorporate more circuitry than necessary, distribute the clock signals inside the FPGA in a safe and stable way, perform a worst/best case timing analysis, an adequate floorplanning, etc. Despite these aspect could improve the reliability of a design, commonly they are not enough to guarantee a certain fault tolerance level.

Nevertheles, due to the limitations and drawbacks of presented strategies, there is a main tendency of utilizing commercial SRAM FPGAs and providing them with fault tolerance by using design-based hardening techniques [49, 89]. Hence, an adequately hardened design implemented in a commercial FPGA represents an attractive alternative, even for space applications [176], due to their flexibility, good working frequencies and their competitive price.

When implemented in SRAM FPGA, soft-core processors are exposed to the same risks than any FPGA design. Nevertheless, their architecture may require to exploit certain specific aspects. This fact has motivated a number of researches, like in [79] where the reliability analysis of a LEON 3FT soft-core processor (designed to be fault tolerant in a low ion flux operational environment) has been done. The research presented in [155] is a remarkable reliability study of a soft-core processor. In this case, a methodology to evaluate the critical elements when exposing the device to SET effects is proposed using an in-house developed

PIC18 soft-core processor. Bearing in mind the relevance of the fault tolerance issue, this research work seeks to contribute with new hardening alternatives to harden soft-core processors implemented in SRAM FPGAs, improving existing techniques and providing new alternatives. Hence, a study of existing hardening techniques to harden soft-core processors implemented in SRAM FPGAs is a especially relevant step.

The present chapter gathers the most relevant fault mitigation techniques to increase the reliability and availability levels of SRAM based FPGA designs, especially focusing on soft-core processors. In this way, Section 3.1 presents the scrubbing, which a widely utilized technique for mainly repair fault in the configuration memory. Next, Section 3.2 introduces hardware redundancy based techniques, giving special importance to Dual Modular Redundancy (DMR) and Triple Modular Redundancy (TMR), the most utilized redundancy levels. Section 3.3.1 continues presenting techniques based on other types of redundancy (time, software and data). The utilization of Dynamic Partial Reconfiguration (DPR) based approaches to fix permanent faults is studied in Section 3.4. Section 3.5 introduces other, less utilized but interesting approaches to harden designs. All presented hardening strategies are summarized in Section 3.7. Finally, this chapter closes with Section 3.6, which discusses about the most remarkable fault tolerance evaluation alternatives utilized to determine the reliability level of designs.

3.1 Scrubbing

The most straightforward method to account for and overcome potential faults is the so-called bitstream scrubbing [57, 95, 177] which takes advantage of the partial reconfiguration capability of FPGAs. It is performed by rewriting a known bitstream (named *golden* bitstream) with a correct configuration. The scrubbing cleans upsets from the configuration memory, prevents the accumulation of configuration upsets and significantly reduces the probability of two SEUs being present at the same time.

Two strategies can be adopted to decide when performing the scrubbing. The first consists in periodically reconfiguring the device by utilizing the *golden* bitstream. This strategy is known as *blind scrubbing* [178]. The second strategy is based on carrying out the scrubbing after a fault detection [179]. Based on this strategy, [180] presented the *lazy scrubbing* approach. This method was applied to a hardware redundancy based scheme in which, the configuration bitstream was read from all replicated modules analyzing data and offsets to repair the

faulty module. As this work stated the *lazy scrubbing* demands less power and resource overhead, and produces less single point of failures than the traditional scrubbing. There is also the possibility of combining both strategies. In this way, a periodical scrubbing can be scheduled to be performed in convenient stages maintaining the possibility of triggering and emergency scrubbing after a fault detection.

The scrubbing requires some type of control mechanism to communicate with the reconfiguration interface in order to load the *golden* bitstream. Due to the complexity of this task the prevalent solution is to utilize a processor as scrubbing controller. Another fundamental requirement is a memory module to store the *golden* bitstream. Following these ideas three predominant scenarios [73] can be introduced:

- **On-chip scrubbing.** As Figure 3.1 shows this is the most compact solution, since no external element is required. A soft-core based scrubbing controller implemented in the same FPGA [181] carries out the bitstream rewriting process by reading the *golden* bitstream from a BRAM based memory block and downloading it to itself through the internal configuration port (i.e., ICAP, PCAP). The main benefits of this auto-configuration process are simplicity and self-sufficiency. Nevertheless, it presents a critical drawback: the control logic and the gold bitstream storage are implemented in the FPGA fabric they are susceptible to SEE, providing a low reliability level. In [182], a fault tolerant ICAP scrubber was presented to overcome this limitation. It consist in triplicating the internal ICAP circuit. However, it does not avoid the presence of single point of failures.

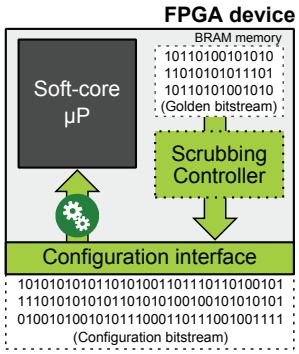


Figure 3.1: On-chip scrubbing.

- **External scrubbing.** Another alternative is to utilize an external device to implement the scrubbing controller and the bitstream storage memory

[178, 183]. In this case the scrubbing controller performs the scrubbing through the external configuration port (i.e., JTAG, SelectMap). Figure 3.2 presents this scheme. The main advantage of this approach is the higher fault tolerance level, especially when the external device presents high reliability. As happens in [83] where the scrubbing controller is implemented in an external anti-fuse FPGA and the *golden* bitstream is stored in a hardened memory. The main drawback of this alternative is the complexity, since such a design requires higher power consumption (two devices to be feed), more physical space and additional hardware (communication buses, conditioners, etc.). Hence, this alternative demands a bigger design effort and increases the design's size, which is likely to increase the costs.

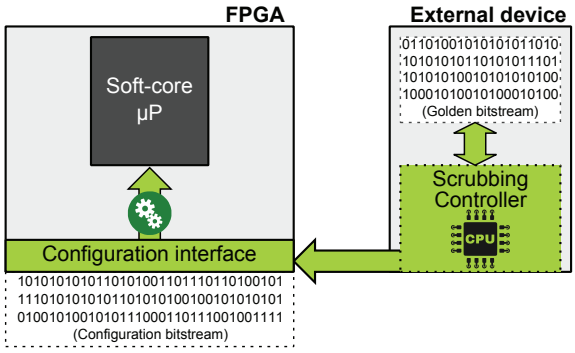


Figure 3.2: Scrubbing with an external device.

- **SoC scrubbing**. Finally, as Figure 3.3 shows, the third alternative is based on the utilization an SoC device, such as, the Zynq by Xilinx. In this case a hard processor performs the scrubbing process utilizing the internal configuration port, while the *golden* bitstream is stored in an SoC's memory block, like a DDR module. This is a medium-cost compact solution, after all an SoC device is likely to be more expensive than a simple FPGA but cheaper than implementing a two devices based system. It also presents an adequate reliability due to the high fault tolerance level of hard processors. The main drawback of this approach is that makes use of a valuable resource of the SoC, due to the fact that an SoC commonly has only one or two hard processors. Nevertheless, the utilization of this approach only demands the hard processor while the scrubbing process. This means that during the time between scrubbings the hard processor is suitable for other tasks. Lower scrubbing frequencies mean higher availability of the hard processor.

Bearing in mind that the bitstream controls both functionality (configuration and

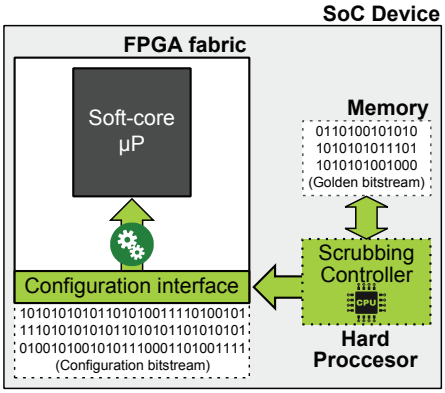


Figure 3.3: Scrubbing with a hard processor in a SoC device.

interconnection of logic resources) and user data content (BRAM, registers, etc.), two types of scrubbing can be defined: configuration and memory (data content) scrubbing. While the configuration content is a static portion, the data content is generally constantly changing during the operation. By virtue of this fact, while the configuration is valid for almost all cases, the data content scrubbing is only suitable for certain cases, such as soft-core processors' program memories. The most elementary approach is to perform a complete scrubbing by rewriting a bitstream with both, configuration and data content information. This scenario requires a power-cycling, which needs to stop the application. Hence, it is not a valid solution for applications that need to be continuously active.

In addition, as a result of the data content rewriting the system is brought back to a previous state of operation, which is a handicap. For this reason, the common practice is to perform a configuration scrubbing repairing the static portion of the configuration memory that controls the logic of the FPGA. This can be performed in an straightforward fashion thanks to the possibility of masking the bits related to data content. On the contrary, the logic configuration bits of the bitstream cannot be masked and for this reason few data content scrubbing solution are available.

In [184], a user memory scrubbing approach to clean errors in memories was proposed. This user memory scrubbing hinges on the addition of an FSM based module to the design. Similar approach were presented in [182, 185]. These methods increase resource overhead and need to use a second memory port. Since Xilinx FPGA BRAMs can only be implemented as single or dual-port memories, if a BRAM memory block is already being used as a dual-port memory this

approach not be feasible. In [177], the memory coherence problem is described, which occurs when configuration data contain user information that is updated by system-level user operations between the configuration readback and writeback operations. It proposes a *dirty-bit* technique which deals with this problem. Although it shows to outperform previous approaches, it also comes along with a performance penalty.

Xilinx also provides with the possibility of taking advantage of the ECC protection of the bitstream by utilizing the so-called *Soft Error Mitigation (SEM) Core* [186] or the *SEU controller macro* [74]. They check the ECC code for SEUs in the configuration memory repairing it when it is necessary. In [187], a similar strategy was presented.

The main weakness of the scrubbing resides in its slowness. The reconfiguration is a time demanding process due to the limitations of configuration interfaces. In addition the more data is to be rewritten (bigger partial bitstreams) the more time is required. Different solutions can be adopted to alleviate this handicap. The most straightforward, as presented in [95, 188], is to use the placing constraints to locate the utilized resources in near placements. Due to this, the size of the partial bitstream is likely to be smaller. Nevertheless, placement constraints limit the freedom of the implementation software that can lead to worst results in terms of resource usage an performance. Other researches propose compressing the bitstream [189] to shrink it and reduce the duration of the partial reconfiguration process. Nevertheless, these methods are relatively complex since they require a compression mechanism, and they are also time demanding.

Considering that scrubbing is not able to repair certain elements of the FPGA (digital clocks managers, power on resets, selectMAP interfaces, etc.) it is not a definitive solution. In addition, considering the time demand on the reconfiguration operation, even in scenarios where a periodical scrubbing is a suitable alternative, the frequency of the scrubbing process is usually lower than application's, and hence the scrubbing by itself is not generally sufficient. Due to these factors, the majority of approaches utilize the scrubbing in combination with other hardening techniques, like hardware redundancy based approaches [52, 53, 83, 189–195].

3.2 Hardware Redundancy

Hardware-based techniques result to be the most frequently addressed by both industry and academia in diverse technological architectures [196]. In the case of FPGA designs, these techniques are a widely used solution to harden them. As

Figure 3.4 depicts, they mainly consist in replicating the hardware modules and using a comparison mechanism for detecting the presence of faults. The replicated module can consist of a mere flip-flop or a complete design. When applying redundancy the replicated modules can be exactly identical or functionally equivalent. Despite that [62] states that replicating the functionality provides better results, because of its design effort demand, the most utilized alternative is to replicate the exact hardware module. Redundancy can also be implemented in different platform alternatives depending on the needs. The most simple alternative is to replicate specific modules in the same FPGA device. Another alternative is to implement the same module in different FPGAs but in the same board. While the last option is to utilize multiple boards with the same FPGA designs. The replication of modules in a single FPGA is obviously the most straightforward and the cheaper option. In any case, these techniques are useful to make designs more robust against both user and configuration memory upsets.

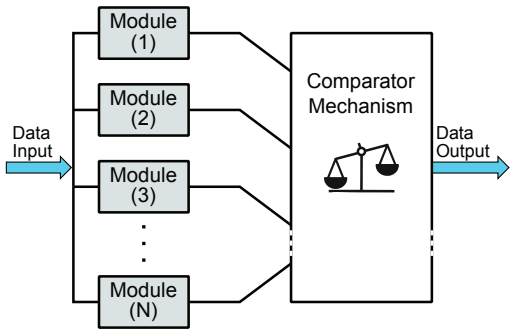


Figure 3.4: Example of hardware redundancy with N modules.

Different levels of hardware redundancy can be adopted, going from the Dual Modular Redundancy (DMR) to the Quadruple (QMR) [72] or higher. Higher redundancy levels commonly provide higher fault tolerance but also bring higher usage of resources. Therefore, it is very convenient to find a right balance between reliability and overhead depending on the requirements of each application. One of the most common redundancy levels when utilizing processors is DMR, which introduces the lowest resource overhead. The other most established redundancy level definitely is the Triple Modular Redundancy TMR [197], because of its good reliability/overhead relation.

An advisable strategy is to floor plan separately within the FPGA substratum, when the design requirements allow it. This procedure helps to reduce the probability of one faulty module affecting other modules. This concept is known as *Spatial Diversity* [191].

Taking into account that it provides the output of the entire system, the most relevant element of a hardware redundancy scheme is the comparison mechanism. Depending on the adopted hardware redundancy approach the implementation of this element requires different characteristics. However, it mostly is a relatively simple implementation. Hence, a small number of configuration bits are related to it. For this reason, an SEU in the comparator block is an unlikely situation. Nevertheless, if such scenario occurs, the response of the entire system will be affected. These fatal consequences commonly demand to apply an additional redundancy based hardening to the comparator block. In any case, regardless of the redundancy level or the hardening of the comparator block, there is always a risk of an SEU in the output of the comparator. This situation represents a single point of failure, since if the output fails, the reliability of the entire system is compromised. Therefore, it has to be remarked that, despite of their valuable results as hardening solutions, hardware redundancy techniques cannot provide an absolute fault tolerance.

Another relevant aspect related to hardware redundancy is the granularity level. Three general granularity levels are defined in the literature [198]: logic level (fine grained), block level (medium grained) and a complete module level (coarse grained). The robustness level rises when redundancy is performed on fine granular modules [199–201].

Considering that more voters have to be implemented, fine grained implementations come with a price in terms of FPGA resources usage and design complexity. Nevertheless, fine granularity offers the possibility of prioritizing which elements are tripled depending on the available space, as [202] proposes. In this way, the most sensitive components can be mitigated first and, if the available area allows, others may also be hardened. [203] proposes the utilization of intermediate voters, like depicted in Figure 3.5, to connect different replicated partitions in order to eliminate single point of failures in the interconnections. In [204], three different algorithms for determining where to insert voters were proposed. Besides, a fine granularity may reduce the performance since the voters are added to the critical path. In addition, as [205] states depending on the design, a bigger number of voters does not assure better performance for all scenarios. This is because the physical placement of resources can affect to their effectiveness.

On the other hand, the design of coarse-grained redundancies demands the lowest design effort and it utilizes less area than fine grained ones. Coarse grained implementations are capable to mask all single points of failure within the replicated modules, but they are unsafe against the accumulation of errors or some multiple error types. For these reasons, depending on the system requirements, designers should select a trade-off among robustness level, overhead and performance,

applying different granularity levels.

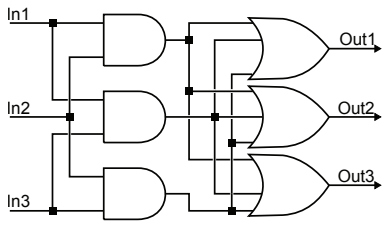


Figure 3.5: Intermediate voter for the interconnection of tripled modules.

3.2.1 Dual Modular Redundancy

Dual Modular Redundancy (DMR) or Duplication With Comparison (DWC) is one of the prevailing hardware redundancy based strategies. The main advantage over higher redundancy levels is that the DMR requires less hardware overhead. This is an interesting characteristic when designing complex architectures, such as soft-core processors [86, 89, 189, 206, 207]. When the duplication is applied to processors, the structure is named lockstep. This approach is based on utilizing two identical processors executing the same application in parallel, feed by the same inputs and comparing the outputs. Most commonly, the output checking process is done at every clock cycle. Both processors must be synchronized to start at the same time initialized with the same context.

As is shown in Figure 3.6, when an input discrepancy is detected an error flag is triggered. A DMR scheme permits to detect errors but neither mask nor correct them. This is because it is not possible to identify the faulty module, requiring a further solution that will stop the operation. Figure 3.7(a) depicts the most basic approach, which is to restart the process or program after detecting a fault. A further straightforward solution is to perform a configuration scrubbing like in [189]. However, since the scrubbing does not affect to the state of the registers and the memory content, this strategy is only valid for errors in logic and not for registered data. In [66], the RESO method was introduced to detect the faulty core and to switch the correct core to output. The idea behind this approach is to modify the operands before performing the re-computation so that a fault affects different parts of the circuits. Due to this, the computations are carried out twice, once on the basic input and once on the shifted input.

Due to obvious reasons, both presented approaches are not commonly sufficient to meet requirements of most applications. In this way, the prevalent approach when

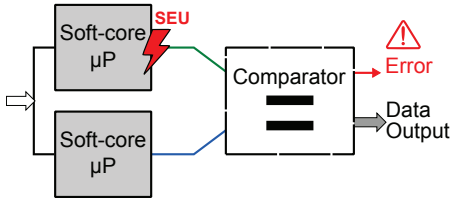
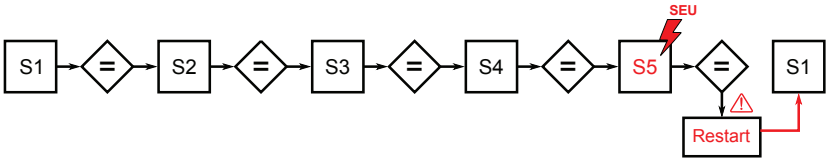


Figure 3.6: Error detection in a DMR scheme.

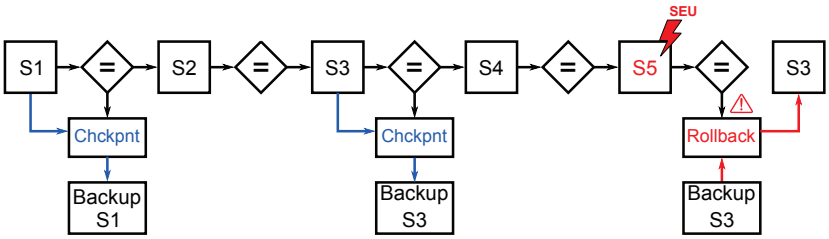
implementing a DMR based hardening is to combine it with checkpointing and rollback techniques [66, 80, 208–211]. Both techniques can be also combined with configuration scrubbing [52, 212]. Checkpointing consists in periodically saving an error free state. Hence, after a correct result in the comparison the checkpointing may be executed (depending on the checkpointing frequency). On the other hand, rollback consists in going back to the previous saved error free state by loading the context saved in the checkpointing. An alternative method based on a Self-Checking Hardware Journal (SCHJ) was presented in [81]. The SCHJ method relies on the utilization of a specialized self-checking hardware journal to design a hardened a soft-core processor. Thanks to the use of this journalization the soft-core processor designed in this work achieves a fast rollback.

The checkpointing technique can be performed by software, by adding instructions for data saving to the program; or by hardware, by implementing data saving mechanisms. In both alternatives, a backup memory space to store the saved checkpoint is always required. The checkpointing frequency determines the fault tolerance level [213]: higher checkpointing frequency means higher fault tolerance but also lower both operation speed and availability, because checkpointing requires stopping the system so as to read memory a save data from modules and registers. As it can be observed in Figure 3.7(c) checking and saving the context of both processors at every execution cycle provides the shortest error recovery times, but usually comes with a performance reduction. Otherwise, as Figure 3.7(b) describes, low checkpointing frequencies come with worst recovery results and more danger of error propagation. Hence, in all the cases checkpointing requires software or hardware overhead and generates some level of performance penalty. For these reasons a trade-off decision has to be adopted during the designing process.

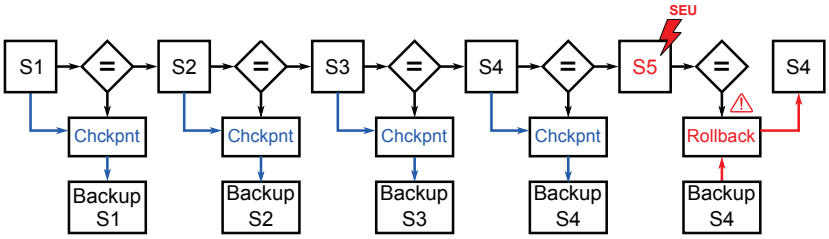
In DMR schemes the comparator only needs to identify a discrepancy. Hence, as shown in Figure 3.8, the comparator mechanism is more basic than TMR (or higher). In [214], two voters were proposed in order to implement them by using abundant and underused carry-chains in Xilinx FPGAs. The comparator



(a) Basic lockstep based (whithout checkpointing).



(b) Lockstep combined with checkpointing and rollback (low checkpoint-freq.).



(c) Lockstep combined with checkpointing and rollback (high checkpoint-freq.).

Figure 3.7: Flow charts of lockstep approaches.

of DMR approaches is a sensitive element, since an error in the comparison may lead to flag as erroneous a correct output, and vice versa. For this reason, the voter can be also harden applying to it hardware redundancy strategies.

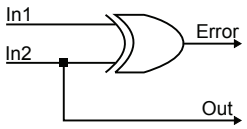


Figure 3.8: Basic error detection implementation for a DMR setup.

The granularity concept can be also applied in DMR schemes as was proposed in [215], where *DMR+* a technique to protect flip-flops in Xilinx FPGAs was presented. Comparing with TMR it obtains a significant hardware overhead reduction, the same circuit delay and only a slightly worse reliability.

A relevant aspect to be considered is the susceptibility of context-saving backup memories. Since an error in them may lead to a faulty rollback they should be hardened with, for instance, a TMR protection or an error correction code strategy.

In [163], an remarkable lockstep approach based on the utilization of the bitstream was presented. Thanks to it, a hardware resource overhead is avoided. The error detection is done comparing the configuration bitstream with a previously obtained *golden* bitstream. Hence, the error detection does no check register's or memories' content and the error detection speed is limited to the bitstream reading speed. This approach also performs the checkpointing by reading the content of flip-flops through the bitstream. Thanks to this, the operation is not affected during the checkpointing, but the checkpoint-frequency is limited. This approach presents the limitation of not saving the content of user memories. On the other hand, due to the use of the Select-Map interface, the implementation of this approach requires external hardware. This hardware consists of an auxiliary board to manage the bitstream operation the hardware requirements are higher. In [216], the author introduces a similar approach based on the bitstream utilization. In this case, the error detection is done by using the extended hamming code in each configuration frame and the checkpointing of flip-flops content is performed utilizing the ICAP interface and the GCAPTURE primitive. Bearing in mind that the ICAP utilizes FPGA's logic resources, it is exposed to induced faults that could affect its internal logic. This an unlikely but possible situation that would override the reliability of the approach. In addition, like in the previous approach, data content of user memories is not targeted. It is important to remark that these approaches are not suitable for newer 7 series devices.

3.2.2 Triple Modular Redundancy

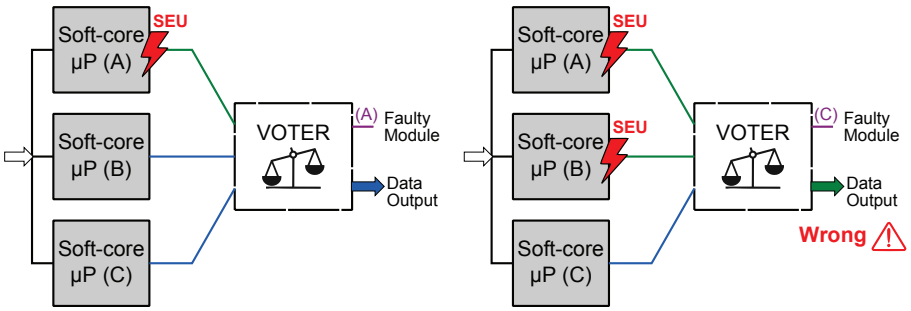
Unlike what happens with DMR, TMR and higher redundancy levels enable the possibility of identifying the faulty module and provides higher fault tolerance levels. However, increasing the redundancy level also implies a resource overhead directly related with the redundancy level. Among the different hardware redundancy alternatives the Triple Modular Redundancy (TMR) is considered to be the most extended one to harden FPGA designs because it offers a remarkable trade-off between reliability and resource overhead. Since first introduced in [217] this technique has become in a standard hardening approach inspiring a vast number of researches in the literature [54, 59, 66, 137, 218–223]. Nevertheless, its biggest handicap is the resource overhead, which is usually about %200. The work [224] presents a TMR alternative with a resource overhead reduction. It proposes to implement a TMR scheme composed by a full precision module and two reduced precision modules. These two reduced-precision blocks will generate an upper and lower bound on the correct function output. The research itself states that despite the acceptable results with complex functions, the result with simple operations are not efficient in terms off fault tolerance.

Essentially in a TMR scheme, three exactly alike or functionally equivalent units of the element to be hardened are used. Once the element is tripled, the final output is determined by a majority voting. Thanks to that, as is shown in 3.9(a), even when a fault is present in one of the three replicas, a voter would still be fed with two correct values. Hence, the wrong third result is masked providing a valid output without stopping the operation.

However, as depicted in Figure 3.9(b), when multiple errors are induced in different modules at the same time provoking an equal faulty output, they can be set as correct. This is an unlikely but possible situation. Another remarkable drawback of using a voter is that in the best scenario errors are masked but not corrected. Hence, when utilizing hardware redundancy there is an inherent risk of accumulation of SEUs. Due to this, as it has been mentioned in section 3.1 the common practice is to combine it with the scrubbing.

Considering the crucial relevance of the voter in these type of schemes, its design is a main concern [191, 225, 226]. Figure 3.10(a) depicts the most basic voting and as it can be observed it consists of three 2-input AND gates connected to a 3-input OR gate. This scheme is capable of providing a correct output in presence of a faulty module. However, this affirmation presumes that the voter cannot be affected by induced faults, which does not match reality.

When a fault affects the voter it can provoke wrong behaviour, from partial failures to total disruptions. The criticalness of such scenario have motivated a



(a) Correct voting under an SEU in a TMR scheme.

(b) Wrong voting due to MBUs in a TMR scheme.

Figure 3.9: Examples of different TMR scenarios.

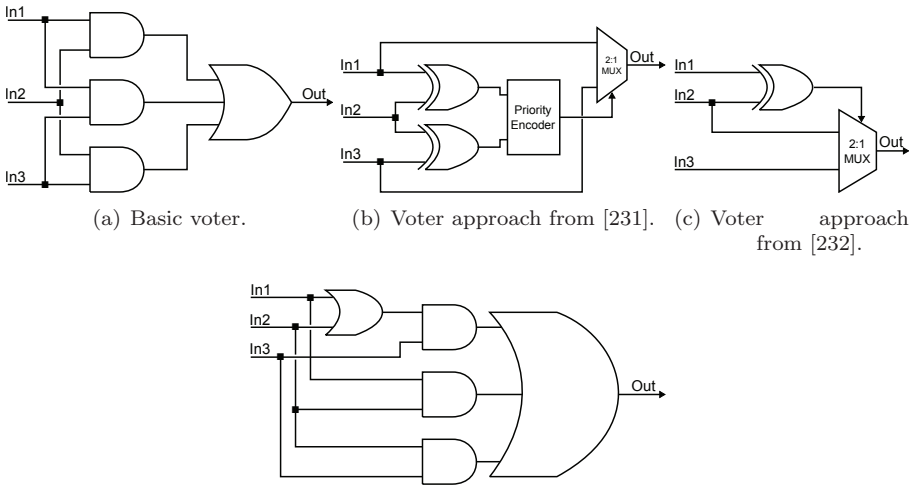
number of researches [227–229]. An alternative to harden the voter can be to triplicate it like in [230], where a suitable partition of the replicated module/voter structure is implemented in different devices. However, its hardware overhead and the fact that the three outputs have to be voted at same point, make this alternative not advisable for many situations. In the article [231], a voter reliability study and a novel voter (depicted in Figure 3.10(b)) implemented using XOR gates, priority encoders and multiplexers was presented. This approach increases the reliability of the voter by six times but also the resource overhead. Figure 3.10(c) portrays the approach presented in [232]. In this case, the proposed voter obtains higher robustness than the basic voter but less than [231]. The main benefit of this voter is a resource overhead reduction over even the basic voter. In [233], a deep analysis of the reliability of the previously presented majority voter approaches was introduced, proposing a new voter hardened for TMR designs. Figure 3.10(d) shows this voter which is composed by several OR and AND gates. This scheme is similiar to the voter's proposed in [232]. Nevertheless, it improves the resilience to potential internal and/or external faults. [87] presents a complex voter composed by different logic gates and managed by an external controller, which detect and recover from permanent faults. [234] proposed a scan-chain based approach applied to the inputs and outputs of each one of the flip-flops in the circuit to detect any functional fault affecting the majority voter, enabling to determine which module has to be fixed. Furthermore, extending the scan chain to inside the module and wrapping on it the different combinational blocks and registers enables the precise location of the fault.

Despite all these approaches improve the reliability over a basic TMR voter, they are not capable to eliminate the presence of a single point of failure in the output

of the final voter.

A totally different approach was presented in [226], where an ICAP-based voting is proposed in order to overcome the consequences of faults in the voter. In this case the voting is performed by an external processor after reading the content of the ouputs registers of each module from the bitstream through the ICAP reconfigurable port and utilizing the `GCAPTURE` primitive. Nevertheless this approach does not solve the problem of multiple errors in different modules provoking an equal faulty output. In a similar way, it does not prevent the accumulation of errors in registers and data memories. The approach presented in [235] also utilizes the ICAP to read the bitstream and detect errors. However, it requires an external device to implement a watchdog timer. Another similar strategy was presented in [236], where the error detection is performed through a direct readback and comparison of the current configuration bitstream. This approach increases the hardware usage since the fault detection and recovery is performed outside the FPGA by a dedicated on-board CPU via the SelectMAP port. In general bitstream based voting methods present low availability due to their time requirements.

As a conclusion, despite they increase the reliability level, all the published approaches present hardware overhead or a single point of failure, and in the majority of them both.



(a) Basic voter. (b) Voter approach from [231]. (c) Voter approach from [232].

(d) Voter approach from [233].

Figure 3.10: Voter alternatives for TMR approaches.

In contrast to coarse grained, the implementation of fine grained TMR schemes can be a tedious task. To help the designers to deal with providing TMR hardening to design, several software tools are available. The *TMRTool* by Xilinx [237] is a remarkable tool when working with Xilinx's Virtex-4QV and Virtex-5QV FPGAs. Nevertheless, since it has been discontinued, new tools has to be considered especially when working with newer devices, such as the Zynq. Another relevant tool is the *BLTmr* developed at Brigham Young University (in collaboration with Los Alamos National Laboratory) [202]. *BLTmr* is a CAD tool to implement partial TMR designs by applying selective triplication in order to target the most sensitive components of designs, hence, reducing the hardware overhead. Another alternative is *Synplify Premier* by the Synopsys, Inc. [238]. This software offers multiple options for implementing error detection and mitigation circuitry, such as, memory protection by inferring error correcting code (ECC) memory primitives and by inserting triple modular redundancy (TMR) on BRAMs to mitigate single-bit errors, safe FSM implementation and fault-tolerant FSMs with Hamming-3 encoding. In [236], a set of tools that allow to manipulate partial bitstreams to perform fine and coarse grained redundancy was presented.

3.3 Other Types of Redundancy

As an alternative to hardware redundancy other three redundancy types can be applied: data redundancy, time redundancy and software redundancy. Each type exploits distinct features providing specific benefits and introducing particular drawbacks. The present section introduces the features of each type, discussing the most remarkable approaches proposed in the literature.

3.3.1 Data Redundancy

Data redundancy is another remarkable fault tolerance alternative to harden SRAM based FPGA designs. This concept is based on adding additional data in order to be able of verifying and even correcting the original information. Data redundancy is mainly used to reduce error rates in memories, which is especially interesting in SRAM based ones. The prevalent alternative is the use of built-in Cyclic Redundancy Check (CRC) and Error Correction Code (ECC) or Error Detection and Correction (EDAC) strategies [59, 60, 73, 239]. With these techniques a bit-flip can be detected, automatically corrected and even recorded, with the possibility of saving a time stamp for further actions. ECC techniques can be implemented in hardware or by software [240]. With software

ECC approaches, transient faults in the combinational logic are not stored in storage cell, and bit-flips in the storage cells can be avoided or instantly corrected.

However, single-bit errors may cause failures in software ECC if an error occurs when reading data from a memory and it is coincident with the time between the last scrubbing and the time of reading. In contrast, hardware ECC checks all the data read from the memory correcting single-bit errors. Hence, bearing in mind that hardware ECC provides better reliability, it is a most advisable strategy.

Thanks to the reliability and the limited hardware overheard introduced by these techniques, they are a widely used alternative to TMR schemes when hardening memories in FPGA designs [194, 241]. While TMR boost the reliability by increasing significantly the area of memory cells (especially with fine granularity), ECC codes produces a small hardware overhead but it needs large logic blocks (with multiple levels) to implement coders and decoders, which increases the length of the critical paths. Due to this, the convenience of each approach depends on the design needs.

Several ECCs are available in the literature [242, 243], such as, Hamming, One-Step Majority-Logic Decoder (serial and parallel), Majority Gate, Bose Chaudhuri Hocquenghem, Berger, m-Of-n, m-Out-n, Residue codes, Reed, Solomon, Hsiao, Checksum, etc. One of the most utilized is the Hamming code [66, 75, 192, 194, 241, 244, 245]. As depited in Figure 3.11, it adopts the parity concept, but uses more than a single parity bit. To detect k-single bit error, the minimum number of bit positions at which the corresponding symbols are distinct or minimum hamming distance (D) is $D= k+1$. A code word of n bits with m data bits and p check bits, where $n = m+p$, can correct $(D-1)/2$ errors and can detect $D-1$ errors. Moreover, with the addition of an extra parity bit, it can be determined whether it is 1-bit or 2-bit error. Nevertheless, even being a good candidate to be used in memories or register files, the Hamming code can be a non advisable alternative in some scenarios. Specially when the huge number of bits leads to long path of serial XOR gates in the decoding and coding modules. In this scenario a suitable alternative is to divide the data in smaller data words. As [241] states, in the case of data words up to 16 bits, the difference in area and delay between hamming code and TMR is nearly negligible. Due to this a TMR scheme is commonly an interesting alternative for blocks made-up of registers and pipelines, while Hamming ECC is more appropriate to harden register files and embedded memories.

Xilinx has a built-in ECC module to protect the BRAM structures [192]. This module is developed for BRAM primitives of data widths greater than 64 bits [246] by using Hamming code. Its goal is to detect and correct errors with a good performance and small resource utilization. Using this module one single bit error

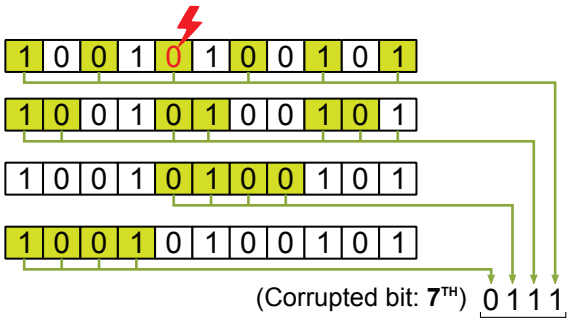


Figure 3.11: Example of error detection using Hamming code.

can be automatically corrected and double errors can be detected. However, its use presents some drawbacks. For instance, the need of 2 clock cycles for data reading, which implies an extra clock cycle latency. To solve this problem, [54] it presents a hardware interface which solves the synchronization problem by "looking ahead" the next instruction. But the drawback of this approach is the resource overhead. Another drawback when using Xilinx ECC module for BRAM can be found when the data width chosen by user is not a multiple of 64, because a double-bit error signal may point out errors that have occurred in the unused bits. In addition, when using the Built-In ECC there are some limitations, such as, the non-availability of Byte-Write enable, RST[AB] Pin and the "output reset value" options and the impossibility of initialization. This last limitation is an important disadvantage for the process of writing the instructions in the program memory. Another problem when utilizing the built-in ECC is the synchronization of the BRAM memory block and the user application. Since, while processors mainly expects memories with a latency of a single clock-cycle, ECC BRAM implementations need two clock-cycles to read data. To circumvent this issue, in [54] a hardware interface called EPA (ECC Processor Adaptor), which looks ahead for the next instruction, was presented. [247] presents a similar approach. The implementation of these interfaces, comes which a resource overhead.

In addition to the hardware based ECC approaches, vendors offer hardening alternatives for the bitstream. Xilinx provides the Frame ECC logic (FRAME_ECC_VIRTEX6 primitive) for Virtex-6 and 7 Series FPGAs [248]. It enables de possibility of detecting single or double bit errors in configuration frame data thanks to the use of a 13-bit Hamming code parity value that is calculated based on the frame data generated by BitGen. During the readback porcess (by utilizing SelectMAP, JTAG, or ICAP interfaces), the Frame ECC logic generates a syndrome value utilizing all the bits in the frame (including the

ECC bits). If no bit change happens from the original programmed values, the [12:00] bits from the *SYNDROME* word are all zeros. On the contrary, if a fault provokes a bit-flip (including the ECC bits), the [11-0] bits from the *SYNDROME* word indicate its location. If the flipped bits are two or more, the [12:00] bits from *SYNDROME* are indeterminate. In this scenario, the error output of the block is asserted. In addition, after reading each frame the *syndrome_valid* signal is asserted. Repairing flipped bits demands a user design since the frame ECC logic does not repair them. Xilinx provides an SEU controller IP [186] capable of repairing, an even injecting, configuration-frame faults. In any case, the design has to be able to save at least one data-frame or be able to fetch original data-frames for reloading them. In [248], it is addressed that the simplest operation is to read the frame through the configuration interface, store it in a BRAM, analyse it and if it required repair it before writing it back. However, if the BRAM which stores the frame is affected by an induced fault, the entire design can be compromised.

Different alternatives have been proposed in the literature to deal with the frame ECC issue. In [249] an embedded IP core, which can be implanted into the FPGAs to detect and correct soft errors automatically was proposed. [250] also presents an SEU-Monitor System which is able of injecting, detecting and correcting single-bit errors and injecting and detecting double-bit errors in the FPGA configuration memory. On the other hand, in [251] a low-cost ECC was presented to detect and correct MBUs in configuration frames. This code is based on the conception of Erasure codes and utilized vertical and horizontal parity-bits to avoid redundant data. It also proposed the utilization of *Mutation* codes. As it states by using *Erasure* or *Mutation* codes the delay can be reduced. This approach does not demand alterations in the FPGA design. The parity bit operation is performed for all configuration frames increasing the computation time.

3.3.2 Software Redundancy

Software redundancy is a well known alternative to the hardware redundancy when protecting processors because it does not require any modification at hardware level. Hence, the impact in terms of hardware overhead is very limited. Software redundancy is based on replicating the program code and adding extra instructions to perform comparison functions. Due to this fact, software based approaches require to increase program memories. Another remarkable drawback is the performance penalty generated by the execution of a larger number of instructions.

Another software based approach commonly utilized in combination with software

redundancy is the utilization of consistency checks. This technique consists in periodically executing a specific set of instructions with the purpose of verifying the faultlessness of memories and their functionality.

In [252, 253], similar software redundancy approaches were presented. Depending on if the faults provoke errors in the program execution flow or in the stored data, different strategies were proposed. In order to deal with errors in the program execution, a modification of the application code by introducing new instructions is done, either in low assembler code or high-level code. Due to this, a checking is performed allowing to know if faults have affected the expected instruction flow. When working with errors affecting stored data, proposed methods mainly consist in providing information redundancy in order to store several copies of the information and in adding consistency checks. Thanks to the utilization of a proper combination of these approaches, permanent and transient errors can be detected and repaired. In addition, their use provides the capability of addressing fault detections without the need of additional hardware.

In [254], the so-called *N-version* programming technique was presented. This classical approach hardens program with an high-level code, which mainly consists in tripling the instruction and utilizing a majority voting. Hence, the performance penalty is significant. In addition, since this approach demands a triplication of instructions and data, it demands bigger memory modules.

The approach proposed in [255] hardens programs against transient faults without the drawback that comes with tripled instructions. It also reduces the performance penalty over the instruction triplication. This fault tolerance method is able to overcome SEUs in processor-based system memory. The basis of this approach is to duplicate the program variables and the operations and performing a checksum, which is updated every time the content of a variable is written. The fault detection is done by checking the coherence of both copies. An error recovery procedure, which corrects the error by relying on the duplicated variables and the stored checksum, is launched after detecting an error.

The Software Implemented Fault Tolerance (SWIFT) approach presented in [256] is a software redundancy technique designed to detect upsets in processor's memories and registers. The SWIFT approach can be divided in two parts. The first part's goal is to protect the processor against upsets in data by instruction duplication. While the second part's aim is to protect the design against control errors but utilizing a software control-flow monitoring. The study [94] extends the intended use of SWIFT to protect the processor logic and routing. It also performs consistency checks as an interrupt service routine to detect upsets in the logic and routing leading to the units.

Software based approach are dependant of the specific program. Hence, the results in terms of overhead, reliability and performance penalty obtained with their use may vary depending on the software programmed in the device. Different works, such as presented in [257, 258], focus on providing strategies to obtain reliable software in order to harden unreliable hardware designs. These kind of techniques are out of the scope of this work.

3.3.3 Time Redundancy

Time redundancy based techniques attempt to reduce the hardware overhead of hardware redundancy approaches at the expense of using extra time [259–263]. As depicted in Figure 3.12, these techniques are based on performing the same or functionally equivalent operation several times and comparing each result. After detecting an error it is possible to perform an additional operation in order to check if the error persists. They mainly add low hardware overhead to designs but performance penalty instead. For this reason, they are mainly advisable to be used in designs where the area is a more critical aspect than the temporal efficiency.

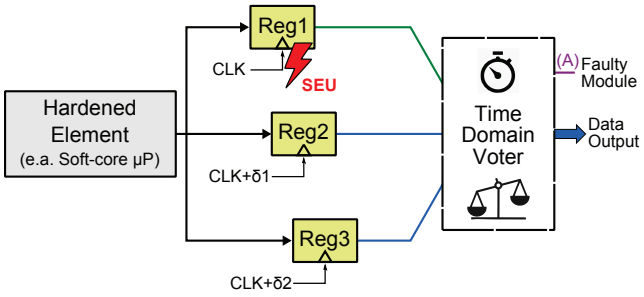


Figure 3.12: Example of a basic time redundancy scheme.

There are two main strategies when hardening a design with time redundancy:

- To repeat the calculation in different instants and compare results. While the negative aspect of this strategy is low execution performance, the advantage is that it requires low hardware overhead. It also provides the capability of detecting transient and even permanent errors.
- To store data in different instants and detect discrepancy. This strategy avoids to repeat the calculations and they can be utilized to detect and mask transient errors.

Although redundancy based techniques are especially suitable to detect or/and mask transient fault due to their temporal character, they can be utilized in combination with coding and decoding techniques to improve the detection of permanent faults [264].

In [193], an approach based on a temporal data sampling mechanism was presented for error detection and correction. It is based on sampling data at three different time intervals, requiring three clocks for proper operation. This work states that thanks to the utilization of this approach the performance can be increased by 55.93 percent over conventional TMR schemes, while providing near 100% fault coverage.

The approach proposed in [265] is an hybrid technique that combines hardware and time redundancies with some special consideration to provide a dissimilar condition for hard copies over the time. The main advantage of the proposed method is that it reduces SEU propagation among the replicated modules. The results of this study conclude that reliability of the harden design increases up to 70 times with respect to a standard TMR.

The logic-level circuit transformation technique based on time redundancy for the automatic insertion of fault-tolerance properties was proposed in [266]. This approach permits to adapt the time redundancy dynamically. In this way, the fault-tolerance characteristics can be traded-off for throughput during the running. This permits to activate the hardening only in critical situations.

In [267], automated behavioural design flow based on a parametrizable architecture to enable time redundancy of loosely coupled hardware accelerators mapped as slaves onto a memory mapped shared bus was presented. The proposed method concurrently re-computes the output twice or thrice from each hardware accelerator while sending and receiving data from and to the master to detect soft-errors. This scheme makes it possible to determine if a transient fault has been produced. Thanks to re-computing three times, masking the errors is also suitable. Despite that the proposed approach reduces the hardware overhead, it cannot guarantee full fault tolerance if the performance of the design to be harden has to be preserved.

Although time redundancy presents lower resource overhead than hardware redundancy, it requires some kind of logic to store the results of the different redundant operations, perform the comparisons, etc. Bearing in mind that this storing and control logic is susceptible to induced faults, the utilization of this type of techniques requires additional hardening strategies to avoid single points of failure. Due to this, they can be considered as complementary tools for certain type of design able to tolerate their inherent performance penalty. These type

of hardening techniques are mainly program/application dependant, and for this reason they are out of the scope of this work.

3.4 Dynamic Partial Reconfiguration to Fix Permanent Faults

While transient faults can be fixed by reprogramming strategies, such as scrubbing, permanent faults produced by Single Hardware Errors (SHE), SEGRs, SELs, SEBs, the TID or the ageing of the die, demand the utilization of spare elements to avoid the use of damaged resources [268]. Due to this, the approaches proposed to overcome from permanent faults are based on utilizing a combination of different techniques, such as, hardware redundancy, permanent fault detection, partial reconfiguration and repaired module synchronization.

3.4.1 Detection of Permanent Faults

The first step before adopting any repairing action is to determine whether the detected fault is transient or permanent. A straightforward solution is described in Figure 3.13. First, the error has to be detected during the application running, by utilizing a technique with error detection capability, like DMR or TMR. After detecting the error a repairing strategy, like scrubbing, must be applied to fix configuration errors. After performing the repairing process a second error detection has to be performed in order to detect the error persistence. In this step, the MTBF has to be analysed, since it is assumed that a permanent fault has produced if two faults are reported at the same element in a time interval smaller than the MTBF. Following the fault identification, the fault fixing mechanism based on partial reconfiguration has to be launched. Once again, after the repairing process a fault persistence check has to be conducted in order to ensure the correctness of the design. If the fault persist the error can be classified as unrecoverable. In [268, 269], different algorithms to determine the presence of permanent errors based on the same idea were presented.

In addition to this, [270] proposed a LUT and CLB level approach to detect permanent faults in a TMR scheme. Despite that this of approach provides an easy identification in both, the replicated modules and the voter, it is technology dependant and its implementation implies significant complexity. A special voter mechanism to detect permanent faults was proposed in [87]. This voter utilizes three input signals that are derived by a special controller which employs two

internal registers to identify permanent errors. In [271], an algorithm for the discrimination of faults in FPGAs based on their recovery possibility was proposed. This algorithm is executed each time an error signal from the independently recoverable areas detects a fault. The identification of a permanent fault is based on the analysis of the fault's frequency thanks to recording of faults in subsequent observations.

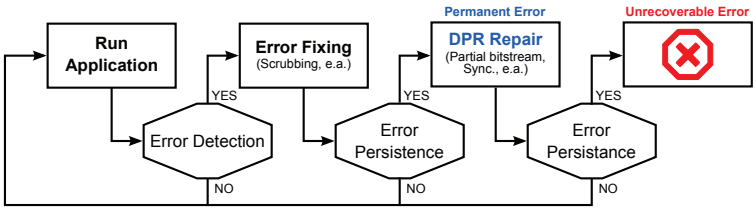


Figure 3.13: Flow chart for detection and correction of permanent faults.

3.4.2 Repairing Permanent Faults

The replacement of faulty resources by utilizing dynamic partial reconfiguration permits reprogramming the FPGA using alternative configurations with the same logic design, but implemented with a different set of fault-free resources and omitting the faulty ones. This strategy commonly requires to generate previously the different back-up partial bitstreams, and to store them in a fault tolerant memory (i.e. a hardened memory or a memory implemented in a reliable technology). In addition a controlling mechanism is required to perform operations, such as, system management, permanent faults identification, reconfiguration, synchronization tasks, etc. Due to its importance, this module has also to be implemented in a fault tolerant manner. Bearing in mind that this alternative makes use of partial reconfiguration is closely dependent on the FPGA technology selected. In addition, the utilization of a partial configuration, with the technology available nowadays, implies a reduction of the MTBF.

In [180], the *jiggling* approach which avoids the use of previously generated bitstreams was introduced. The *jiggling* repairs permanent errors generating alternative configuration bitstreams for a faulty module by making use of the healthy modules. In this way, this technique utilizes the remaining healthy functional modules as a template for an algorithm which can generate different bitstream versions of the demanded behaviour that avoid the utilization of faulty resources. The algorithm creates mutations using single bit-flips in the FPGA configuration bitstream of the faulty module. This apporach also reserves spare resources to

relocate generated modules. In this way, permanent faults can be repaired until all the spare elements are consumed.

This idea can be carried out in different granularities. The most utilized one is the fine granularity known as the *tiling* technique [21, 60, 191, 191, 269, 272], which is described in Figure 3.14 extracted from [9]. It consists in pre-generating several partial bitstreams of the same design for the same reconfigurable block in order to avoid the utilization of the faulty tile. Each of the pre-generated partial bitstreams has to contain a forbidden zone. Thanks to that, a permanent error can be masked by downloading a convenient partial bitstream, in which the forbidden zone overlaps the faulty tile. A straightforward way to define the forbidden zone when utilizing devices by Xilinx, is to utilize placement constraints, like `PROHIBIT` [273]. The smaller is the forbidden zone (finer granularity) the more partial bitstreams have to be generated, increasing the developing time and the external memory space need. A solution to decrease data size of partial bitstreams is the usage of compression techniques, as proposed in [189]. In addition to this, as is stated in [9], storage needs can be noticeably reduced by generating only the differential bitstream between two adjacent columns. This is because whether a forbidden zone corresponds to a column the number of needed partial bitstreams equals the number of columns utilized by the target block. In [274, 275], *CLB column overlapping* strategies were proposed. The utilization of adjacent columns requires to use strict constraints in order to generate the almost equal (except for the forbidden zone) partial bitstreams. In addition the occurrence of persistent errors inside the configuration frame that contains a permanent fault within a column can generate relevant issues. This happens because, since a tile may contain data content shared by several modules. The reconfiguration process can affect not only damaged modules, but also adjacent correct modules corrupting the stored data. To avoid this problem additional strategies, such as, the utilization of a checkpointing and rollback mechanism, are utilized.

In [276], a self-healing strategy with fault diagnosis was proposed. This process is carried out by executing evolutionary runs which generate evolved circuits that can avoid the faults being accumulated in the configuration matrix.

A coarse-grained approach can be adopted when the tiling is not able to repair a permanent fault. In this case, the strategy consists in reserving spare reconfigurable partitions for the different modules [21, 206]. Figures from 3.15(a) explains this idea applied to a TMR scheme. As Figure 3.15(a) shows, during normal operation the three modules work together while three spare reconfigurable partitions remain reserved as backup spaces. After the detection of a permanent fault, the faulty block is discarded while the remaining modules continue working. In the

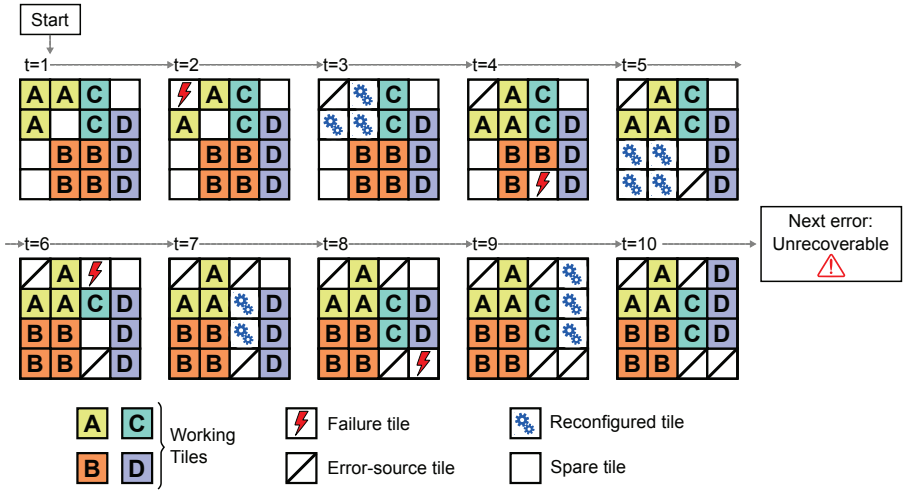
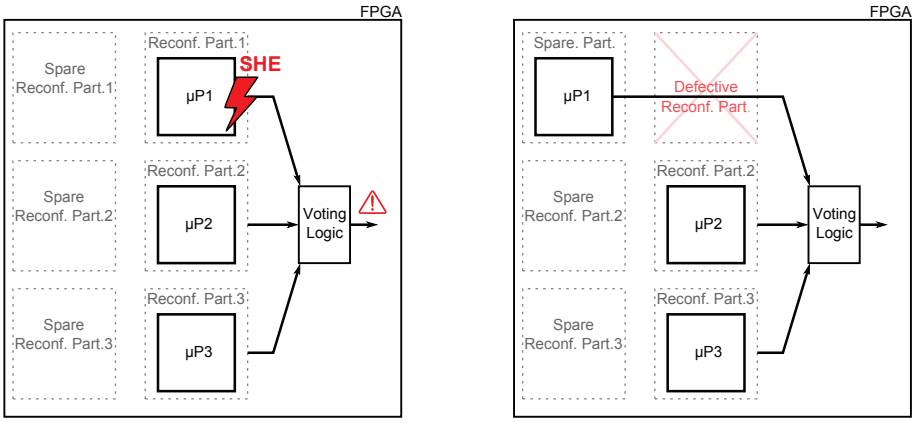


Figure 3.14: Permanent error repair with tiling strategy.

repairing process a new partial bitstream containing a version of the module is written into the FPGA, as is shown in 3.15(b). Thanks to this approach, the utilization of a defective hardware is avoided in a straightforward fashion. This method permits to implement repaired modules in distant reconfigurable regions along the FPGA. This idea can be especially interesting when a remarkable area has been damaged which permanent faults. Nevertheless, the main drawback of this method is that a substantial area remains unavailable to be reserved as spare reconfigurable partitions. The number of spare partition is a design decision, the more partitions reserved the more possibility of obtaining alternative error-free spaces. Hence, it increases the unavailable device-area and requires to generate and store more partial bitstreams. As is stated in [189], a medium-grained spare allocation strategy obtains better results in both storage and performance frequency overhead. However, a coarse-grained approach presents better reliability level than a fine-grained approach (especially when the faults take place in attached slices).

Some references suggest the possibility of transforming the TMR scheme into a DMR system when a faulty module cannot be recovered [277, 278]. Nevertheless, this implies to lost the benefits of the TMR strategy and also demands to implement the additional mechanism necessary for DMR schemes.



(a) Permanent error detection.

(b) Faulty module repair.

Figure 3.15: Permanent error repair with spare reconfigurable partitions.

3.4.3 Synchronization of Repaired Modules

In the majority of cases a reconfiguration by itself does not suffice for recovering faulty systems hardened by the combination of hardware redundancy and partial reconfiguration techniques. If the repaired instance features some kind of internal state, it needs to be synchronized with the rest of the healthy instances after the reconfiguration.

The straightforward way to synchronize a redundancy scheme is to reset the entire system and to start the execution in all the replicas at the same point. Nevertheless, this kind of approach provides poor availability results. Another alternative to obtain higher availability levels is the utilization of the combination of checkpointing and rollback techniques, which has been widely discussed in 3.2.1.

A common synchronization technique for redundancy based schemes is the so called roll-forward [60, 68, 68, 89, 210, 213]. This technique consists in copying the correct state from the fault-free replicas and downloading it to the repaired unit. This approach avoids the utilization of any re-computation process and, as is stated in [263], it presents lower performance overhead and increases the reliability over re-starting and re-executing the entire process. The main drawback of this approach is that if error-free modules continue operating, their states continue changing which makes this approach unsuitable for this scenario. A solution

for this issue is to pause the system during the synchronization, which comes with a significant performance reduction. The approach presented in [279] proposes a modified roll-forward approach that also includes checkpointing. After detecting an error, the three replicas stop operation while the inputs are buffered. After repairing the faulty module, the buffered data feed the three instances. This approach does not add any re-computation delay. Nevertheless, it increases resource overhead because of the required buffers and the synchronization reduces the system's availability. In addition, it is less capable of recovering multiple faults than the voting scheme.

In [280], the *ScTMR* technique was introduced. This technique proposes a roll-forward approach which re-utilizes the scan chain implemented in the processors for testability purposes to recover the system's fault-free state. This avoids any re-computation and meets the specifications for real-time systems adding a low resource overhead. Although this approach significantly decreases susceptibility, it is unable to recover the system from simultaneous errors in two modules and from masked errors in single faulty modules. In [87], an updated version of this approach that addresses the shortcomings of ScTMR was presented. In this case, the proposed technique named *scan chain-based multiple error recovery TMR* (SMERTMR) is a roll-forward technique for TMR-based designs which offers the capability of fault recovery in the presence of multiple masked error and also two faulty modules. This technique is only applicable to systems where the replicas are always synchronized and it introduces a performance penalty.

The work presented in [10] proposes a *present-input and healthy-state based synchronization method for TMR schemes* called *PIHS3TMR*, which is depicted in Figure 3.16 (obtained from [10]). This real-time approach avoids the utilization of checkpointing and provides availability during the synchronization process without stopping the operation. The *PIHS3TMR* is based on modifying the FSM by introducing a healthy present state and a synchronization control signal. This approach is an interesting synchronization alternative. However, its implementation for more complex designs like soft-core processors can be impractical.

The so-called *known-blocking* method was presented in [86]. This approach increases de reliability of soft-core processors implemented in SRAM FPGAs by utilizing TMR in combination with DPR. Its key feature is to avoid system from blocking situations. To perform this technique the both processors that are still running properly are lead to a known *safe-loop* (all outputs reset to low logic level). Due to this, the system continues running in a safe mode. Although this approach enables the possibility of non stopping the system, the processors are not available for the target application. Hence, it cannot continue working during the *safe-loop*.

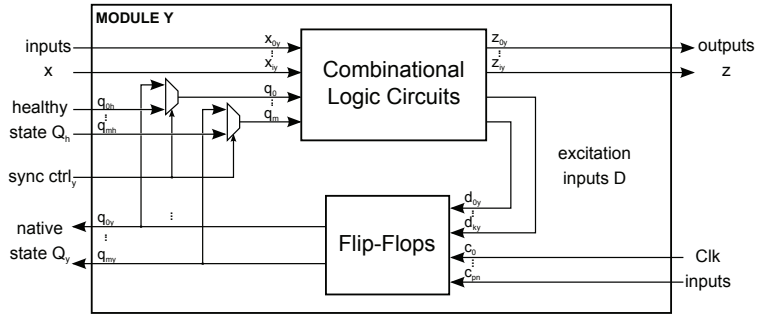


Figure 3.16: Block diagram of the PIHS3TMR.

Another synchronization method valid for small FSM was proposed in [281], introducing the notion of *state prediction*. This suggests that each FSM has (at least) one state to which the machine always returns after a finite amount of time. Therefore, by setting the FSM of a reconfigured module to this state it is possible to wait for the other two instances to reach this point during their normal operation, and thereafter continue seamlessly operating with all three instances. The work presented in[282] proposes a similar approach which deals with the synchronization of a recovered module in a TMR scheme. It suggests to perform the synchronization while the other two modules kept running, waiting for a predictive a future state in which converge. These approaches are useful for simple designs, such as, FSMs. Nevertheless, they are not advisable alternatives for complex architectures where future states cannot be predicted.

Synchronization has been also considered in [59], where a fault tolerant MicroBlaze architecture using TMR and DPR was presented. In that work, three MicroBlaze processors sharing peripherals and memory are implemented in partially reconfigurable regions. The peripherals and the shared memory are protected by TMR and ECC, respectively. Sharing one memory between the three processors reduces synchronization to a process of reading and writing to the memory. Whenever the processors write data to the memory a voting process is started. It masks wrong data from the newly reconfigured instance, by storing the correct values sent from the two remaining functional instances. In a subsequent read cycle the three processors can read the synchronized value back to their memories. While this synchronization approach is suitable for MicroBlaze processors, it is not applicable to all processor architectures. For the MicroBlaze processor it is possible to access all registers of relevance for synchronization, such as e.g. the stack pointer, the status register and the program counter. In this manner, a synchronization by reading and writing to the shared memory becomes feasible.

On the contrary, many other popular processor architectures (e.g. PicoBlaze or PIC) do not provide reading access to all registers representing the state of the processor.

Another related contribution which uses rollback is the work [60], where the synchronization between two MicroBlaze processors operating in (DMR) is addressed. After one of those processors is partially reconfigured, a similar technique to *state prediction* is used. Once the faulty processor has been identified, the roll-forward is executed to set the processor to a state the other MicroBlaze will reach in the future. Since the state is assumed to only consist of the program counter, a synchronization similar to the one in [59] is required after the roll-forward to update the register contents. Hence, this method presents to the same drawbacks as those previously identified for the shared memory approach in [59].

In [50], an approach based on the use of the TOPPERS/JSP open-source RTOS kernel was introduced. It utilizes three MicroBlaze processors in a TMR scheme. After detecting a fault and reconfiguring the faulty module, an interruption in the RTOS triggers the synchronization process. As the work states, this approach requires a large area usage and it decreases the maximum operating frequency of the design.

Another synchronization approach that consists of a down-counter to determine when the newly reconfigured module has re-established its state was proposed in [283]. To carry out this approach, a countdown value is set to the latency in clock cycles of the longest path through the component. In this way, the outputs of the reconfigured block are ignored until the resynchronization. The application scope of this approach is very limited, since applications need to present a cyclic behaviour to return a previous state, where all the three modules can be synchronized.

3.5 Other Fault Tolerance Approaches

Apart from the mainly established fault tolerance strategies presented in previous sections, different approaches that explore alternative solutions have also been proposed in the literature. Although they can be suitable in some scenarios, most of them have a limited application scope. For this reason, their adequacy will depend on the particular application to be hardened.

A fault tolerance approach based on the concept of self-stabilization, utilized in distributed computing, was presented in [85, 91]. The research proposed a self-recovering algorithm for processors (hard-core and soft-core). It follows the idea

of that the self-stabilization permits to distributed systems to reach a correct state regardless of in which state has been initialized with in a finite number of execution steps. The proposed algorithm utilizes a specific interruption code combined with asynchronous interruption signals at random clock cycles to modify the content of memory cells randomly. In the presence of corrupt data cause by failures, the algorithm leads the system to an arbitrary configuration that corrects the behaviour in a finite amount of time. The approach focuses on specific resources, such as, registers (special and general purpose), internal SRAM, memoy caches, etc. Due to this approach, the system can bear transient failures, especially those failures that do not affect the code executed by any node. However, since this approach is program dependant and especially focused on preserving the convergence of transmitted data, its application scope is limited. In addition, the approach is not effective if an error affects one of the registers that contain the variables utilized by the processor while executing the self-converging program.

The approach presented in [239] is another algorithm-based hardening alternative. The proposed Algorithm-Based Fault Tolerance (ABFT) technique reduces the susceptibility by 99% with a limited resource overhead (about 25%) by hardening both the datapath and the configuration memory. The application scope for the ABFT is linear-algebra operations. However, as the author states, for applications that are not comprised of linear algebra operations the ABFT cannot provide sufficient protection.

A different strategy was proposed in [284] where the configuration bitstream is modified by the hardware inside the chip. A controller implemented with logic resources of the FPGA manages the bitstream adaptation following two algorithms (*Modify Placement* and *Modify Routing*). This approach affects the design negatively in terms of area and performance. Besides, bearing in mind that the controller is implemented with logic resources of the FPGA it is susceptible to induced faults.

Other fault tolerance approaches hinge on design aspects in order to harden FPGA based designs. In [65], a methodology for the hardware/software co-design of embedded systems has been proposed. This method takes the advantage of different software and hardware strategies (hardware or software redundancy, etc.). It presents the concept of Software Implemented Hardware Fault Tolerance (SIHFT), which allows to achieve a trade-off solution that meets the specific requirements of designs. In this way, after defining the specific requirements, different SIHFT approaches are used in an incremental way in order to obtain a concrete number of candidate implementations of the software. After that, all of these generated candidates are compared to estimate the code and execution time overheads. After discarding the implementations that not meet

requirements, the selected candidates are tested by an SEU emulation tool. The ones that present better fault tolerance results are again tested in deeper fault injection campaigns identifying the critical elements in order to protect them with hardware redundancy. In this way, a range of trade-offs hardware/software implementations is obtained. Although the positive results achieved hardening a PicoBlaze based design this approach is a software and hardware dependant solution and it demands high design efforts.

Another alternative to harden FPGA systems in the design process is to harden designs during the synthesis like in [285], where the In-place X-Filing (IPF) technique was introduced. The IPF is a synthesis-based algorithm that masks SEUs at a logic level in both LUTs and interconnects. This approach requires to analyse the configuration bits in other to identify the bits with no failure rate. This process demands some kind of analysis mechanism such as a logic simulation. After identifying the bits with no failure rate, they are filled in order to mask errors. The more of these bits are reconfigured to mask errors, the smaller failure rate is obtained. The utilization of this solution also requires a high design effort and its results are dependant of the number or configuration bits with no failure rate. In addition, the results in terms of resource overhead of performance obtained when utilizing this kind of approaches may be worst than the ones achieved when utilizing vendor's tools, which are optimized for their devices.

In [11], a hybrid error-detection technique based on assertions and a non-intrusive enhanced watchdog module to recognize induced faults in processors, named HETA, was introduced. This technique depicted in Figure 3.17 (obtained from [11]), utilizes the combination of software-based techniques in tandem with a non-intrusive hardware module. By virtue of this idea, the approach analyses and adds static instructions to the original program-code. During operation it also constantly updates the content of a signature register, which is connected to different program nodes. These program nodes are the basis for the error detecting control flow. On the other hand, the purpose of the hardware module is to detect incorrect jumps to unused memory positions and control flow loops. This approach utilizes 66% less resources than a TMR implementation. However, the reliability level is lower and the performance is considerably reduced.

An approach inspired by the immune system that can be found in higher organisms was presented in [286]. The proposed *hardware immune system* is based on the *negative-selection* algorithm, which utilizes binary matching rules to discriminate invalid states. These matching rules are implemented utilizing the logic resources of the FPGA in order to obtain a higher operation speed with more logical gates cost. Bearing in mind that almost hardware systems can be represented as an individual or a set of interconnected FSM, the discrimination methods are

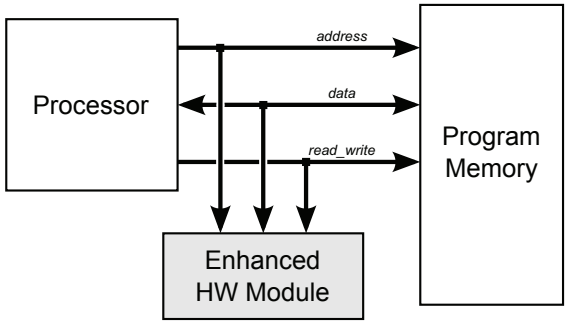


Figure 3.17: Block diagram of the HETA approach.

implemented using an FSM in order to represent the system to be immunized. The detection of faults is performed following the idea that an error creates an invalid state. In this way a controller module brings together the inputs, combining them with previous states to generate different strings for the FSM and the detection. Monitoring the internal states of the FPGA makes it possible to detect an error before it propagates to the output. The results in terms of capacity and performance are non optimal and the utilization of an internal controller compromises the reliability of the entire design.

3.6 Evaluation of Hardening Techniques

After designing hardening techniques it is necessary to evaluate the fault tolerance level that they provide. Nevertheless, apart from the hardening technique to be evaluated, several factors affect the tolerance level of a hardened design, such as, the platform device, the environment, the design itself or even the running application. Due to this, the evaluation process is frequently performed in different levels. The first evaluation could be to check the concept to be designed itself by analysing its architecture (the selection and configuration of the elements, etc.). Being the root of the whole designing process, taking care of those aspects could significantly improve the result for future steps. Another aspect to be previously analysed is the reliability of the device to be utilized as an implementation platform. This task is usually performed by manufacturers. The exponent of this idea is the previously presented Rosetta Experiment [166] in which Xilinx is continuously evaluation the reliability level of their devices under several conditions, such as, different altitudes and geographic locations.

After obtaining a satisfactory design, the hardening strategy has to be selected. A number of features have to be considered before adopting a design decision, such as, the available resources, the acceptable performance penalty, susceptibility and availability levels, etc. An aspect that helps to optimize the hardening approach to fit with these requirements is to detect the most susceptible elements and evaluate their reliability. This analysis permits to identify in which elements has to be focused when hardening the entire system. Finally, after developing and applying the different hardening strategies the entire design has to be validated in order to confirm its robustness. The idea of fault tolerance evaluation is also utilize to hardening techniques as a way to validate the effectiveness of developed approaches.

Fault tolerance evaluation of a particular design also considers the determination of the critical or essential bits from the configuration memory. A bit can be classified as critical if its faulty state can affect the functionality. Not all the bit from the bitstream are critical. In fact in regular designs less than 20% of the configuration memory bits are usually classified as essential or critical to the functionality of a design [157, 289]. Nevertheless, their reliability is a crucial factor, since they jeopardize the integrity of the entire design.

Considering the typical error rates, analysing the reliability during normal operating conditions would drastically delay the designing process. Hence, several strategies have been proposed to estimate systems' behaviour under the presence of errors.

The easiest fashion to evaluate the fault tolerance level of a design is to simulate a fault by utilizing an HDL simulator [200, 245]. Several HDL simulators are available. There are commercial releases like *Modelsim* by Mentor Graphics or *Vivado Simulator* by Xilinx, or open-source software, such as, *Verilator* by Veripool and *OSS CVC* by Tachyon Design Automation. These alternatives provide a straightforward way to analyse the behaviour of a design by utilizing its HDL file due to the observability and controllability provided. This is especially helpful to improve hardening techniques, since it enables to adjust different characteristics and check them in a relative easy way. HEARTLESS [290], MEFISTO-C [291] and VERIFY [292] remarkable simulation based tools and methods. Comparing with other existing strategies simulation is a very time demanding alternative. In addition, considering that these simulations in a better scenario are post-synthesis simulations, the accuracy obtained is limited because they are many physical aspects that can't be checked.

Another alternative is to use circuit instrumentation to substitute cells of the target device by equivalent instrumentation hardware cells, obtaining a prototyped design. In [293], an approach called *Autonomous Emulation* which follows this

idea was presented. The AMUSE (Autonomous MUltilevel emulation system for Soft Error evaluation) approach presented in [294] extends this idea to emulate SETs. SETs are more difficult to evaluate than SEUs because they are many more possible SETs than SEUs. For this reason, this approach implements all demanded functionalities in the FPGA along with the circuit model, without requiring any FPGA-host iteration which improves the injection speed. However, bearing in mind that the circuit tested is not exactly the target design itself, the accuracy of the obtained results is not optimal.

The most effective and extended way to evaluate the fault tolerance of a design is to inject faults in it. In these cases the device to be tested is usually referred as Device Under Test (DUT). The two main strategies to inject faults in SRAM based FPGA designs are to induce faults physically by utilizing radiation emitting devices and to inject fault in the bitstream [11, 265]. Each of them offer different benefits that also come with their subsequent drawbacks. Another alternative is to combine both strategies, like in [295], in order to obtain the benefits of both worlds. These two strategies have inspired a large number of researches, some of the most relevant ones and the main characteristics of these strategies are discussed in the following lines.

3.6.1 Physical Fault Injection Techniques

Physical Injection Techniques are the most realistic and fast way to inject faults in SRAM-Based FPGA systems since they can simulate the real environment in a direct fashion. In addition, they don not require to have access to RTL and netlists of designs. They consist in exposing the DUT to a certain radiation level by utilizing specific instrumentation. The most common physical fault injection techniques are laser testing and particle beam testing. Although the utilization of these techniques provides the most realistic results, they require the utilization of very expensive machinery (in the range of $100K per day) and they provide a small sample of faults. Despite the combination of multiple techniques is not the most extended practice, the work [20] proposes a complete and deep fault tolerance evaluation which performs multiple tests. It includes simulation modeling, fault emulation and laser fault injection. Nevertheless, the most remarkable fact is that the DUT has been also tested in a flight experiment, as part of the Space Test Program-Houston 4-ISS SpaceCube Experiment 2.0 (STP-H4-ISE 2.0). Exposing devices to space radiation in real-time is one the most accurate approaches that can be adopted. However, this is not an available alternative for regular designers, since the economical cost and the time. In addition, as is mentioned in [296], in some cases it is necessary to reset and reprogram the device increasing the time requirements.

The characterization of the resistance of a device to induced errors is done by the cross section (σ), given by expression 3.1 [297]. Fluence (ϕ) is the particle density (particle number per 1 cm^2 area), utilized to describe the output of the laser beam. n in the quantity of configuration bits utilized in the DUT. Finally, e is the number of detected errors.

$$\sigma = \frac{e}{\phi \times n} \tag{3.1}$$

There are two validation schemes to evaluate the fault tolerance when utilizing these fault injection methods [298]:

- **Static test** This test type is useful to determine the device static cross area, which is defined as the ratio between the fluence of hitting particles and the SEU quantity. Their main advantage is that they provide a way to quantify the device's susceptibility to a particular radiation type. Static test are based on initializing the DUT with a *golden copy* of the bitstream, which is used as a reference in periodical comparisons between it and the bitstreams read after the different radiation exposures.
- **Dynamic test** These test are utilized to evaluate the device dynamic cross section, which is defined as the ratio between the fluence of hitting particles and the quantity of SEUs that cause incorrect outputs. These test are a valuable tool to determine the sensitivity of a particular FPGA implementation to a particular radiation type. Dynamic tests consists in checking outputs during the running of a specific application implemented in a device. To obtain valuable information, the inputs have to be predefined and the outputs have to be pre-calculated. An alternative, solution is a parallel-execution of an equal or equivalent application implemented in a stable and secure platform and utilize it as a reference. There is also the possibility of reading back the bitstream of the DUT in case of mismatch in order to determine the faulty resource.

After performing one of these or both tests the results have to be collected and analysed in order to identify the modifications in the resources of the FPGA induced by SEUs [70, 298]. In [298], the CILANTO (CIrcuit-Level ANalysis TOol) is utilized for this task. This tool takes advantage of a database with a relation of configuration memory bits and the related FPGA resources. In this way, it can be used to carry out a bit-level comparison between the *gold copy* of the bitstream and the recollected data.

Depending on the technology and physical principles used, physical fault injection techniques can be divided in two groups: Particle-beam testing and laser testing.

Particle-Beam Testing

Particle beam testing is the most extended physical fault injection method [128, 178, 297–300]. This testing consists in exposing the DUT to radiation beams that produce a particular quantity of particles per second per area. The SEU is generated when an ionized particle interacts with a sensitive area portion of the FPGA, producing bit-flips. Three different particle types can be produced for these test: Heavy-ion, Proton and Neutron. Each of them is more relevant in specific environments at certain time intervals. Based on these concepts different tests can be performed, like in [301] where neutrons test, high energy protons test, thermal neutrons test and alpha foil test were conducted.

The concept of Linear Energy Transfer (LET) is closely related to particle-beam testing. LET is the amount of ionizing energy transferred to the material in the from of ionizations and excitations. Hence, LET can help to determine the potential radiation damage.

The main objective of heavy-ion tests is to determine the lowest LET rate that produces detectable SEUs. Despite different LET ranges can be found in the space, the more prevalent LET for heavy-ion particles are those with high LET. Before performing heavy-ion tests the adequate dose has to be determined [299].

Protons commonly present lower LET rates compared with heavy-ions. Protons can produce ionization directly or indirectly. While in direct ionization the proton itself creates the charge that provokes the SEU, in indirect ionization the charge is produced by a collinsion a proton-nucleus. Despite both ways can generate SEEs in electronics, only indirect ionization has been demonstrated to affect FPGA's logic. This is because the direct ionization does not generate sufficient charge. The configuration memory of SRAM FPGAs have shown to be highly susceptible to protons [302]. Figure 3.18 shows the Isochronous-Cyclotron located at the Nuclear Physics Institute of the Academy of Sciences of the Czech Republic, which can provide proton beams tests. Scheduling of particle beam facilities is quite difficult and rigid.

Neutrons are generated by the spallation process [298], which consists in bombarding a heavy-metal target (tungsten) with pulses of highly energetic protons. Due to this process neutrons are produced from the nuclei of the target atoms. The energy of the produced neutrons is normally reduced through a moderator module. Due the high cost of the infrastructure, only a limited number of dedicated facilities able to perform test based on neutron beams matching the terrestrial flux are available, such as, ISIS, LANSCE, TRIUMF, and RCNP.

In general terms, fault tolerance evaluation by particle beam technologies presents

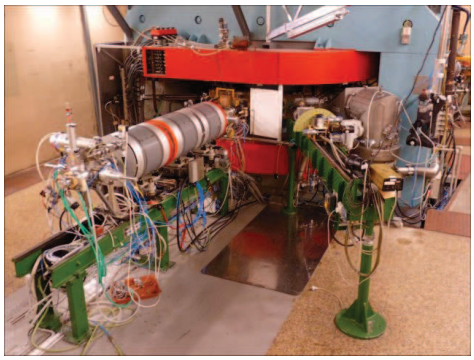


Figure 3.18: The Isochronous Cyclotron U-120M.

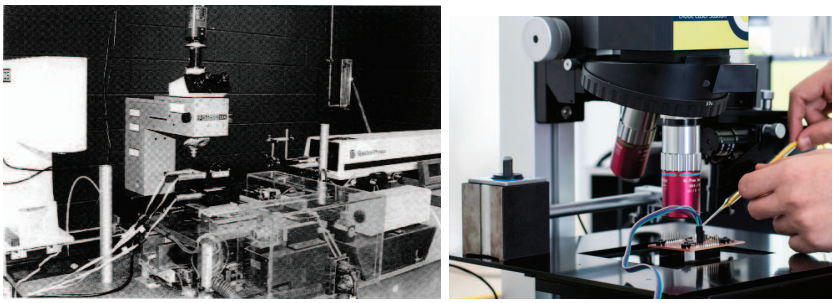
Figure 3.19: ISIS pulsed neutron source at the Science and Technology Facilities

the challenge of synchronizing, locating and controlling the bean precisely. Hence, the implementation of an accurate SEU injection at a precise area in the FPGA is a complicated task. Moreover, the risks that come with the utilization of this technology demand special precautions for managing the radiation source, and running with the vacuum interface.

Laser Testing

The principle of this type of test is that when the laser energy is accumulated in a semiconductor it produces free charge carriers that are able to generate upsets. In fact, the impact of laser induced faults in SRAM-based FPGAs are analogous to SEUs produced by energetic particles [303]. Unlike particle beam, laser testing offers a finer control in means of space and time. Due to this, laser tests permit

to have a precise control of the injection target to the micron level. Different laser-testing setups are shown in Figure 3.20(a) and Figure 3.20(b).



(a) Laser testing setup from [303]. (b) Laser testing setup by Riscure Inc.

Figure 3.20: Different laser testing setups.

The laser pulse can ionize a specific region of the DUT, inducing bit-flips in both, combinational or sequential logic elements. Unlike collision tests, due to the differences in the physical fault mechanisms of both methods, laser testing is not able to characterize the rates of raw faults. Nevertheless, thanks to the precision in the transient fault generation it is possible to characterize the fault-to-error probability. Another interesting characteristic is that, unlike bitstream based fault injections, the laser testing permits to create realistic multi-bit upsets and SETs.

Like in particle bean testing, one of the main drawbacks of this method is that the technology necessary perform the test is highly costly. The preparation and development efforts demanded for a system-level test are high also. Another problem when utilizing laser testing at application-levels is to inject single faults per application execution [304]. The work [295] circumvents this problem by driving a trigger signal to a control block that opens or closes a specific shutter in the laser path, while the energy magnitude is measured by a photo-detection based module. Maintaining the laser correctly aligned during the entire test is another challenging task.

3.6.2 Bitstream Based Fault Injection Techniques

Bitstream based fault injection techniques, also known as fault emulation techniques, are an inexpensive and a widely extended tool [73, 85, 87, 198, 234, 239,

305–307] to characterize specific design implementations. They enable to collect huge amount of data due to continuous injection over a period of hours or months and they are a valuable tool for studying the behaviour of DUT under the presence of bit errors. Despite being less realistic than radiation test, the results obtained with these techniques are highly accurate. For this reason, fault emulation is commonly utilized as a complementary tool to physical injection.

They mainly consist in reconfiguring the DUT with a corrupted configuration bitstream and checking its behaviour in order to evaluate the reliability. Figure 3.21 shows the basic flow of a bitstream based fault injection.

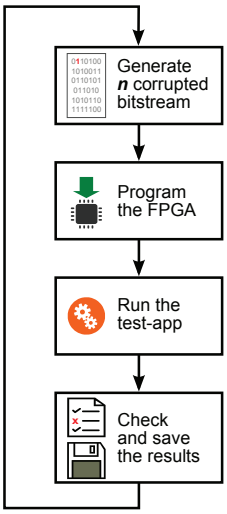


Figure 3.21: Basic flow of bitstream based fault injection.

First, the corrupted bitstream has to be created. The usual way is to flip the desired bits of the configuration bitstream. When determining the critical bits of a particular implementation, an alternative is to generate as much of corrupted bitstream as configuration bits in order to check the effects produced of each configuration bit. Another way is to generate random positions of the corrupted bits like in [307]. Considering the large amount of bits of a configuration bitstream, these alternatives require lots of time and large memory storages. Since the reconfiguration is a time demanding process, the size of the bitstream to be loaded will directly affect the duration of the fault injection process. Due to this, instead of loading the entire bitstream, an alternative is to generate partial bitstreams and utilize the dynamic partial reconfiguration capability of SRAM-based FPGA [308, 309]. This practice reduces both, the injection time and the

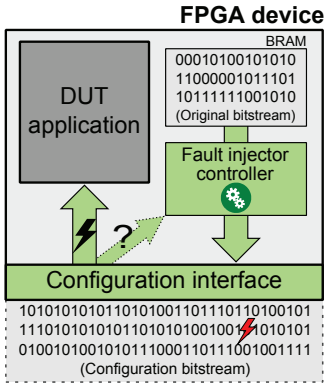
memory demand to store the different corrupted bitstreams.

If the hardening approach is focused on the memory elements, it is relevant to put special efforts in injecting errors in the stored data. This strategy requires to study the bitstream structure. In the case of Xilinx devices the *.ll* location file it is a very helpful resource. However, it requires certain level of processing effort.

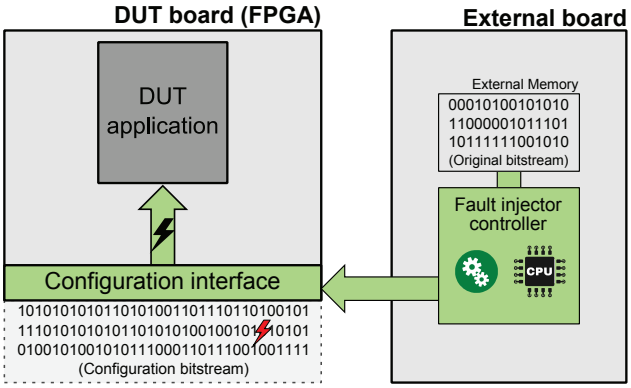
The second step in the basic fault injection flow is to reprogram the FPGA with the corrupted file. Since this can be done utilizing different configuration interfaces, this aspect directly involves the experimental setup. Figure 3.22(a) shows the most inexpensive strategy, which is to utilize a single FPGA device, as both, DUT and fault injector. Despite this approach can be practical to test particular design portions, it requires to perform a deep study of the implementation and its bitstream and it does not permit to perform a complete fault injection test. This is because it implies to implement the fault injection logic in the DUT itself. Hence, a faulty bitstream could also affect the fault injection, and even the test logic. A widely extended practice, as Figure 3.22(b) depicts, is to utilize and additional external board to the manage the reconfiguration of the DUT [310]. This scheme avoids the possibility of corrupting the fault injection logic itself. Nevertheless, it increases the costs due to the need of additional devices. An in-between solution is to utilize SoC devices like in [165], where the Zynq device has been utilized. As Figure 3.22(c) shows, in this case the Processing System manages the creation of the corrupted bitstream and downloads it through the PCAP interface to configure the programmable logic. This approach avoids both, the need of additional hardware and corrupting the fault injection controller.

After reconfiguring the FPGA with the corrupted bitstream the functionality has to be tested by running the test application and comparing the results obtained with reference values. There are several alternatives to obtaining these reference results. The most simple one is to pre-calculate and store the responses of the application to be tested under specific inputs and conditions. In this way after running the application the results can be compared with the pre-calculated responses. Another alternative is to utilize an additional implementation of the target application and run both, the DUT and the replica in parallel, checking the outputs in runtime.

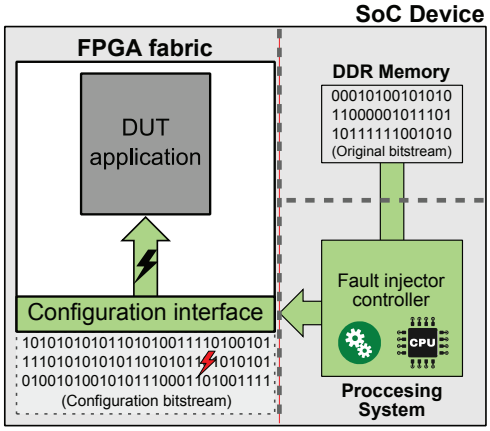
An extended practice to increase the efficiency of functionality tests is to utilize test vectors. This implies that each injected fault is tested with a vector of different inputs. Bearing in mind that different input can affect to distinct bits this strategy opens the range of possible generated errors. After performing the actual test the results obtained can be saved in a file permitting to run a next test. In this way all the obtained results can be analysed in future fault tolerance evaluations.



(a) Autonomous internal fault injection scheme.



(b) External fault injection scheme.



(c) SoC fault injection scheme.

Figure 3.22: Different bitstream based fault injection setups.

Several fault injection tools have been presented in the literature. Some of the most representative ones are listed in the lines below:

- Xilinx provides a so called Soft Error Mitigation Controller **(SEM)** [186] that can be used to inject, detect and correct errors in the configuration memory of 7 series devices. Nevertheless, it is only suitable for the configuration memory and not for BRAMs or distributed memories. Regarding BRAMs, Xilinx offers a fault injection mechanism for BRAMs with its CORE Generator. However, due to its limitations this mechanism is not practical for many applications.

- Fault Injection **(FI)** tool presented in [311] is a bitstream based tool for Virtex FPGAs that addresses only configuration memory cells and user registers. This tool is especially designed to fit with the requirements of those applications in which the FPGA undergoes in frequent reconfigurations. It modifies the configuration bitstream while this is loaded into the device without utilizing standard synthesis tools making it independent to the system utilized. Nevertheless, since this tool cannot access to the built-in FSM state transitions it requires the utilization additional fault injection techniques to test them.

- **XRTC Virtex-5 Fault Injector** is another fault injection tool for Virtex devices presented in [312]. This FPGA fault injection system for testing digital FPGA circuits has been designed in conjunction with the Xilinx Radiation Test Consortium. The main goal is that it achieves a high customization and full bitstream coverage at a high fault injection rates.

- FPGA-based Fault Injection TOol **(FITO)** proposed in [313] permits to inject faults emulating SEUs and SETs at RTL level of an FPGA design by adding extra ports and connections to the flip-flops.

- **FLIPPER** tool presented in [314], as its name suggests, is able to provoke bit-flips within the configuration memory utilizing the partial reconfiguration capability of SRAM-FPGAs.

- Fault injection Using SEmulation **FUSE** [315] is a tool that includes the concept of semulation (a combination of simulation and emulation). It permits to take advantage of the benefits of both techniques: the higher fault injection speed of the emulation and the flexibility and the observability of the HDL simulation.

- Shadow Components-based Fault Injection Technique **(SCFIT)** [93] tool is also based on the semulation. It utilizes TCL scripts to access to the FPGA by using the JTAG configuration interface.

- Flexible on-chip fault Injector for run-time Dependability validation with target specific COmmand language, **FIDYCO** [316] is a tool that combines both hardware and software schemes. While, the hardware includes the target FPGA implementation, the software is located in an external computer. Due to the flexibility provided by this tool, the designer is able to test a variety of components. It mainly consists in moving the fault injector towards the target and after it, translating the target to the FPGA.
- Direct Fault Injection **(DFI)** [96] is focused on injecting errors in soft-core processors implementations. This approach is a combination of multiple fault-injection methods such as FITO, FUSE, etc.
- NETlist Fault Injection **(NETFI)** [317] is a tool that permits to inject faults in designs written in any HDL language (Verilog, VHDL, etc.). The main idea of this approach resides on modifyin the built-in FPGA resources that are to be used by the netlist after the synthesis process.
- Fault-Injection Fault Analysis tool **(FIFA)** [318] allows to inject faults at RTL level. It implements two versions of the DUT in the FPGA device.
- The platform presented in [319] called **FT-UNSHADES2** is an approach focused on carrying out the fault injection utilizing a hardware assistant that can accelerate the analysis process. It uses a mother board connected to two daughter boards. It has different operating modes to deal with ASICs and FPGAs. The *FPGA mode* injects the fault utilizing the bitstream. However, it also has an additional *Beam Testing mode* to be used with the system inserted in an ion beam. This mode allows to place one daughter board exposed to the ion beam and maintain the other safe in other to compare them acting as a coincidence detector.
- Advanced System for the TEst under Radiation of Integrated Circuits and Systems **(ASTERICS)** is a platform used in [91]. This upgraded version of the THESIC+ platform, is built utilizing two FPGAs. While the first one manages the communication between a computer and the ASTERIC board utilizing a hard processor, the second contains the DUT with the user design to be tested, the injection module and the memory controller. A watchdog implemented in the first board checks the possible errors generating a loss in the sequence.
- In [310], a low hardware overhead injection approach for FPGA designs, avoiding the need of special injection boards was presented. This works offers two fault injection approaches: An external SEU flow which is a fault injection approach managed by an external device, and a single bit error test flow, that utilizes the internal reconfiguration obtaining a high injection

performance.

- The work [320] proposes a high-speed fault injection system along with a methodology able to test the sensitivity of a design's bitstream to SEUs. This system is especially designed for soft-core processors and it also can be utilized for radiation testing purposes.

One relevant drawback of these techniques is that they are not able to inject faults across all sequential elements of the design, especially when the design utilizes proprietary IP cores, where physical mapping is unspecified. Another drawback is that these techniques tend to overemphasize cache and register errors [295]. Due to the inherent nature of bitstreams these techniques are highly technology dependant. Although the flows and concepts can be utilized in different devices, the implementation of each approach has to be adapted to the particular specifications of the DUT, which may require to investigate its design.

Despite the utilization of this tool represents a very helpful aspect, to investigate new fault injection approaches is out of the scope of this work.

3.7 Conclusions

Taking into account the diverse application fields and the different soft-core processor implementations available, providing them with fault tolerance requires trade-off decisions to develop a tailored design. The aspect to evaluate mainly are the hardware overhead, the reliability level, the performance penalty and the availability. The work presented in [287] proposes several models to evaluate the reliability and availability of the most relevant fault tolerance techniques for SRAM-based FPGAs. This kind of researches provide valuable information to obtain appropriate mitigation schemes.

The most relevant item to be hardened in any FPGA design is the configuration memory, since it manages the settings, interconnections and data content of almost the components within the device. Many manufacturers provide ECC based protection for the configuration memory. However, due to its limitations (i.e. it is not able to repair multiple errors), in some cases this hardening technique is not sufficient. In those scenarios, a periodical configuration scrubbing is a remarkable complementary alternative. This technique is able to repair most of configuration errors, even those masked in hardware redundancy schemes. Nevertheless, this technique comes with some drawbacks, such as, lack of protection for data content, time demand and requirement of a controller mechanism. Hence, both the ECC and scrubbing techniques are mainly combined (or substituted) with

other hardening methods, like hardware, data, software or time redundancies.

Data redundancy techniques are based on encoding data when writing it in memory and decode it when reading, for checking and fixing errors. The advantages of ECC techniques are good performance and small resource utilization, especially with large data width. This is because the added encoding data decreases proportionally with data width. Software redundancy based techniques increase fault tolerance level with a low resource overhead. However, they are time demanding and relatively reliable strategies. The benefits or drawbacks of these techniques are closely related to the characteristics of each application program. Time redundancy based approaches offer similar features: the reduced hardware overhead impact comes with a performance penalty and an improvable reliability. Hence, the most utilized approaches are based on hardware redundancy. Despite being resource demanding techniques, they offer high availability and dependability.

When hardening a soft-core processor with a hardware redundancy scheme, the two main alternatives are TMR and lockstep combined with checkpointing and rollback. While TMR provides higher reliability and availability the area impact of the lockstep is smaller. After selecting the redundancy level, an appropriate granularity has to be adopted, choosing between a range of possibilities that start from the lower hardware overhead and simply design of coarse grained implementation and that end in the higher reliability level of a fine grained one. Is has to be remarked that due to the use of a voter, hardware redundancy schemes always present a single point of failure.

In any case, considering that a soft-core processor consists of several elements, it is interesting to analyse the different alternatives to harden each of them. While elements, such as, the ALU, the control logic, etc. based on logic operations, have to be hardened with hardware redundancy, user memory elements may accept distinct techniques like ECC strategies.

There are four main user data storage elements in a soft-core processor. The first one is the program memory, which is a read only memory. The second is the data memory block that can be read and written. The third are the registers, that can be user registers (read/write) or special function registers (only read or read/write). Last but no least, there is a stack memory block, that can't be accessed. Bearing in mind the unique characteristics of each memory element, the available methods and their adequacy for each module may differ. Recent spacial experiments have proved that BRAMs are especially sensitive to radiation-induced upsets, indicating that distributed RAM based memory structures present higher reliability leveld [288]. Hence the study of BRAM protecting techniques gains special relevance.

When hardening data and program memories the most suitable alternatives are an ECC encoding and TMR implementation. DMR scheme is not a valid option for these modules, since the errors can only be detected and cannot be fixed, requiring an additional backup memory block to save the checkpoints. In fact, the use of this backup memory together with the duplicated memory blocks implies a TMR implementation. When selecting an ECC hardening, the data memory requires both encoding and decoding modules, while the program memory only needs a decoding module, because the coding has to be done in the programming process. There are not big implementation differences between data and program memory blocks when picking out a TMR hardening. Is has to be remembered that both, ECC and TMR, do not repair fault but mask them. This problem could be less crucial in the case of data memories, since in many application cases the data is constantly be re-written during program execution, cleaning masked errors. For program memories, a memory data scrubbing would be an interesting alternative to wipe the masked faults.

The stack memory can be hardened with TMR or ECC approaches because, DMR scheme does not offer a proper reliability. Depending on the depth of this module, ECC can be too costly in terms of area. In this way, a TMR implementation is the adequate alternative to harden this element in most cases. Bearing in mind that the stack memory is mostly frequently being written during the program execution, the masked errors are likely to been wiped during the program execution. Nevertheless, considering that an error in the stack memory can affect the entire execution flow of the program, it is highly advisable to harden it properly.

On the other hand, when hardening registers the most advisable method is the TMR. Considering the small data width of registers and that each register demands its own coding and encoding logic the resource overhead added by the ECC makes it unreasonable to use it. An alternative to use an ECC strategy reducing the resource overhead can be to utilize a single ECC coding/decoding combined with a multiplexer. However, the integrity of the multiplexer could compromise the reliability of the entire system. The reason to discard a DMR hardening is the same as in data and program memory blocks.

Regarding the evaluation of the fault tolerance level of a hardened soft-core processors the most suitable alternatives are bitstream based fault injection or/and physical fault injection strategies. While bitstream based fault injection techniques provide an inexpensive way to estimate the failure rate, physical fault injection strategies offer more accurate information at higher costs. Although both strategies provide valuable information, they also require complex and time consuming experimental setups. Due to this, in some scenarios more basic alter-

natives like modifying the hardware design to provoke faults, could offer faster and adequate enough ways to evaluate the correctness of a hardening strategy.

Considering all these aspects it can be stated that there is no hardening solution that is completely efficient for all types of designs, so the implementation of fault tolerant architectures and its validation requires a special study of system's characteristics and the application requirements.

Chapter 4

Contributions in Fault Tolerance for Soft-Core Processors

This chapter introduces the different approaches developed in this work that have been focused on improving the fault tolerance of soft-core processors implemented in SRAM based FPGAs. These approaches address different aspects related to fault tolerance. Thus, in some cases they improve specific deficient features of existing techniques and in others propose new alternatives to circumvent known issues.

The first contribution presents the PICDiY, a soft-core processor developed specifically for this work. This soft-processor offers simplicity, modularity and self-sufficiency, making it easily adaptable. Due to those features it has been utilized as an experimental subject to develop and test the fault tolerance approaches presented.

In next contributions two novel approaches that deal with the data management utilizing the bitstream have been developed. While one approached focuses on managing BRAM data the other deals with register's data. Both novel data management approaches have been utilized as a basis to develop new fault tolerance approaches. roaches.

The rest of the contributions deal with different fault tolerance aspects: data content scrubbing, extracting data from BRAM memories with damaged interfaces,

checkpointing and rollback in locktstep schemes and synchronization of repaired soft-core processors in TMR schemes. Although most of the proposed techniques have been applied to the PICDiY soft-core processor, the majority of them can be adapted to other soft-cores and even other types of architectures for SRAM FPGA based designs.

4.1 PICDiY: Target Soft-Core Processor

The amount of logic resources required to implement a soft-core processor depends on several factors, such as, data width, memory size, special features, etc. The more elements and larger data widths selected in the designing of a soft-core processor, the larger usability and processing capability are obtained. Nevertheless, usually the price to pay is a resource overhead and also a performance penalty due to the increase of critical paths' length. In addition to the obvious lower area availability, the resource overhead generally implies several drawbacks: higher power consumption, since logic elements need to be powered; more logic resources susceptible to errors, decreasing the fault tolerance level of the design; and in relation with hardening techniques, more logic resources have to be replicated when implementing redundancy schemes. Thus, from the designers' point of view, a simple and clear design of a small processor composed by the common and basic elements of most common architectures is an interesting platform in order to develop new fault tolerance approaches, where a big processor would bring unnecessary complexity. Furthermore, the simplicity of the target processor comes with a significant reduction of the amount of time demanded by synthesis and implementation tools. Due the presented benefits of utilizing small architectures, an 8-bit small processor has been selected as a candidate to be used with the different proposed tests and approaches.

The first step in the study of new hardening alternatives for soft-core processors has been the definition of the target soft-core processor. After studying the different existing IP soft-core processors it has been decided to design a new soft-core processor from scratch. The main reason to make this decision has been to have a total control of all the aspects of the architecture which would expand the customization potential. Modifying the design of existing IPs is mostly a tedious work that commonly comes with unexpected issues. For instance, in [321] a PicoBlaze IP was adapted for synchronization purposes, as a result, the obtained design offered worse characteristics than the original design. The reason for this negative effect is that the PicoBlaze has such a highly optimized and complex design to fit perfectly in Xilinx's devices and to provide them with high performance and small resource usage, that even small changes could lead the

synthesizer to deteriorate this optimization level. Similar results were obtained in [322, 323] where the original design of the PicoBlaze was modified to obtain a platform independent version of this IP.

In a next step ,it has been decided to base the design of the target soft-core processor on the PIC16 microcontroller due to several reasons: it is a RISC architecture that offers a good code density [43]; its Harvard architecture uses a minimalist design which allows fast access and a different bus width for instruction and data (a 14-bits width for the program memory and 8-bits width for the data memory); it has a good execution speed (each instruction takes 4 clock cycles); it is a simple, efficient and stand-alone device; it works properly with state machines; and it is a well known architecture that can be encountered in many engineering applications [324–326]. Furthermore, the existing free toolkits available for the PIC microcontroller can be used to develop application software like IDEs (Integrated Development Environment). Thus, the PICDiY has adequate characteristics for a number applications, such as, memory peripherals or reconfiguration interface tasks where larger processors would be wasted.

The next subsection describes the architecture of the designed 8-bit and platform-independent soft-core processor, which for the sake of simplicity will be referred to it as PICDiY throughout the document.

4.1.1 PICDiY's Architecture

The design of the PICDiY has been developed from scratch by utilizing VHDL language and basing on PIC16's functionality, its architecture's block diagram and its instruction set. Figure 4.1, obtained from the 16F84A data sheet, shows the block diagram of the PIC16. In order to obtain the simplest design, non essential elements of PIC16's architecture have been discarded. The dismissed components are the timers (Power-up, oscillator Start-up, Watchdog and TMR0) and certain ports (only two ports are included). In addition, several adaptations have been introduced in order to optimize the architecture's characteristics and functionality. Thanks to all these modifications, the design developed in this work is a small processor with flexible functionality and complete self-sufficiency. The main features of the architecture, the different components and the adaptations made are discussed in detail through this section.

As shown in Figure 4.2, the PICDiY consist of different blocks: the Arithmetic Logic Unit (ALU), the program counter, the Instruction Decoder and Controller (IDC), the 8-level stack, program and data memory blocks, address and data multiplexers and the different special function registers. Due to its Harvard architecture, separate storage and buses for instructions and data are implemented,

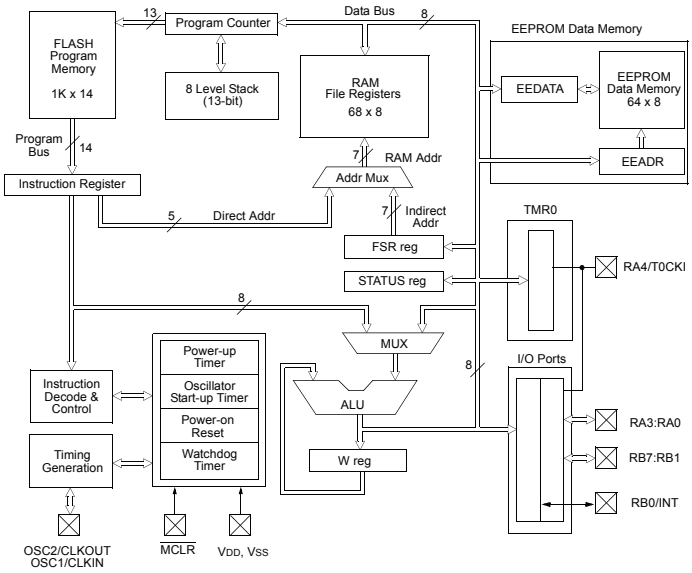


Figure 4.1: Block diagram of the PIC16.

allowing faster access and different bus width for instruction and data. The widths are 14 bits for the program memory and 8 bits for the data memory. The data bus includes a multiplexer structure, not shown in the figure to facilitate the comprehension. Control signals, such as, enables, mux controls, etc. (which mainly are generated by the IDC) are also omitted for the same reason. The processor also has an 8-bit PORTA, clock, reset and interrupt inputs (both clock and reset inputs are also not shown in the figure) and an 8-bit PORTB output directly connected to the PORTB register. In this way, PORTA and PORTB are physical connections between the processor and external peripherals for data transmission. Only two ports are implemented for achieving a minimal usage of FPGA resources, nevertheless, the number of ports can be easily changed by modifying the VHDL files of the design.

The special function registers are an area of data memory dedicated to registers that are required for configuration and control, which can not be used as general purpose registers by the user. Since each of the special function register has a specific functionality, in the case of the PICDiY, each of them have been implemented on separated blocks outside the data memory block. In this way, the PICDiY contains seven register-blocks (FSR, STATUS, PCL, PCLATH, INTCON, PORTB and W) of eight bit width. The registers related to the discarded components

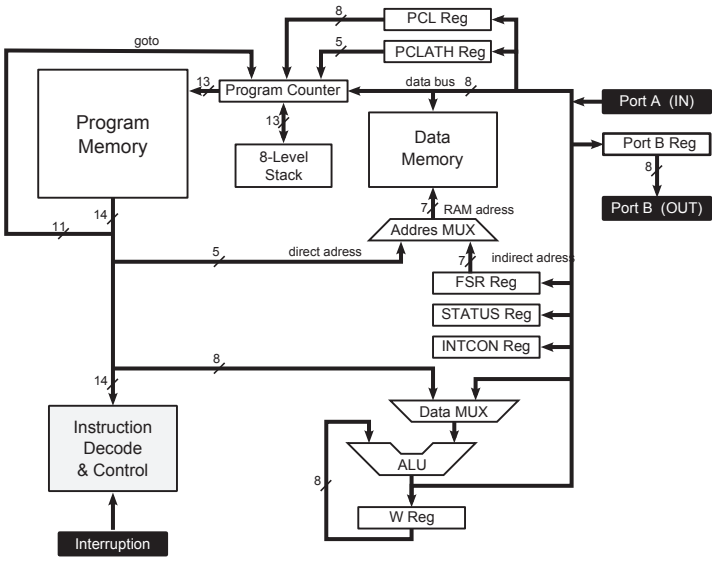


Figure 4.2: Block diagram of the PICDiY.

(TMR, EEDATA, EEADR, OPTION, TRISA, TRISB, EECON1 and EECON2) have not been implemented in the PICDiY due to obvious reasons.

The STATUS register is one of the most relevant registers when programming the processor, since it contains the arithmetic status of the ALU, the RESET status and the bank select bit for the data memory. As all the special function registers, the STATUS register can be the destination for any instruction. Figure 4.3 shows the different bits of the STATUS register. The first three bits are the carry (C), digit carry (DC) and zero (Z) flags of the ALU, respectively and their values vary depending on the results of logical or arithmetic operations. Bits three and four (power down and watchdog timer timeout) are unused since they are related to functions not implemented in the PICDiY. Finally, bit five (RP0) and bit six (RP1) are the bank selection bits. Bearing in mind that PICDiY's data memory has only two banks, RP1 is not used. However, it could be used in future adaptations if a larger data memory is required.

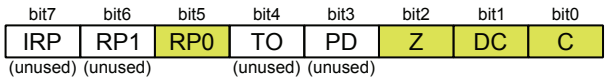


Figure 4.3: STATUS register.

The FSR (File Select Register) is used for indirect addressing to other file registers. If a file register address is loaded into the FSR register, the content of that file register can be indirectly read or written. This register is usually used as a pointer to a block of locations. Reading the FSR itself indirectly results in a 00h, while writing to itself indirectly results in a no-operation (STATUS bits could be altered).

The PCL register (Program Counter Low Byte) and the PCLATH (Program Counter Latch High) registers are used to store instruction's addresses to be loaded in the program counter. Both registers, which are fully readable and writable, are required because the length of the program counter can be up to 13 bits (depending of the depth of the Program Memory). Figure 4.4 depicts how the bits are distributed when data is load to the program counter and, as it can be observed, several bits of the PCLATH register are unused. Since the depth of the program memory can be modified, the quantity of the unused bits may vary. For instance, in the case of a program memory of 256 instructions depth the eight bits of the PCL are sufficient, leaving all the five bits of the PCLATH unused. The adaptation of these characteristics require minor changes in the HDL design.

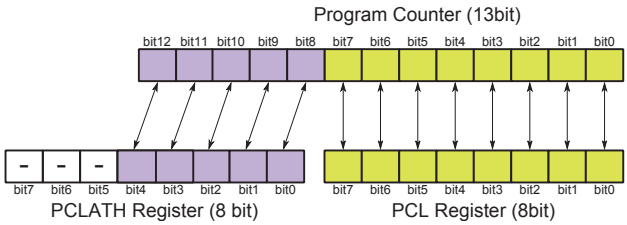


Figure 4.4: PCL and PCLATH registers.

The INTCON is a readable and writable register utilized to control the interruptions. Since a single type of interruption is implemented for the PICDiY, only the seventh bit is used to store the interruption state. The management of this bit is performed by the IDC.

The W register (Working register) is used for ALU operations as an operand. It is not an addressable register. Nevertheless, it can be read and written by using some instructions, because it also can be used to store the results of operations. For instance, `MOVWF` and `MOVF` instructions can move the values from the W register to any location in the data memory, and vice-versa.

The ALU (Arithmetic Logic Unit) is an 8-bit block responsible for performing arithmetic operations such as adding, subtracting, decreasing and increasing, logic operations such as AND, OR, XOR and IOR and other operations like

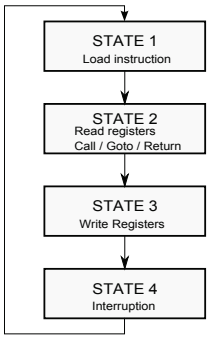


Figure 4.5: State diagram of IDC's FSM.

shifting (right or left within a register), swapping or not operation. Depending on the 8th bit of the instruction word, the result is sent to the W register (when '0') or to another register (when '1') like RAM Data memory, FSR or PORTB. According to which instruction is executed, the ALU can affect values of zero, carry or digit carry bits of the STATUS register.

The IDC is responsible of managing the processor's resources. After decoding each instruction, it controls different elements of the processor, such as, the ALU, the multiplexers, the different registers, the DATA memory and even the program counter. It is implemented as four states Finite States Machine (FSM). This is the reason why the execution of each instruction takes four clock cycles. Figure 4.5 presents the IDS's FSM and as it can be observed, in the first state the IDC activates the instruction reading from the program memory. The second state depending of the instruction, is used to read data or to load the correct next instruction on the program counter, when `CALL`, `GOTO` or `RETURN` instructions are executed. In the third state, if necessary, the IDC enables the writing in the data memory. Finally, the last state is used for data stabilization and for the interruption management.

In PIC16's architecture the data memory is divided into two areas. The first area is composed by the special function registers, while the second is the user-data memory. Since, the PICDiY's special function registers are implemented in separate blocks, its data memory block includes only the user-data memory, which has been implemented utilizing the dedicated BRAMs [327] in order to optimize the utilization of resources when using Xilinx FPGAs. In this way, the synthesizer will use the Xilinx FPGAs dedicated BRAM resources, avoiding the use of distributed RAM that would use more logic cells. Nevertheless, memories can be easily implemented in other vendors' devices by using their specific resources,

thus, maintaining platform independence. The data memory is banked in two blocks to permit a larger memory storage and each bank extends up to 7Fh (128 bytes). As it has been commented before, the bank selection is done by writing the STATUS register's RP0 bit. In this way, Bank 0 is selected by clearing the RP0 bit of the STATUS register and the selection of Bank 1 is done by setting this bit to one. All the data memory addresses can be accessed either directly by utilizing the absolute address of each register or indirectly. When indirect option is selected, using the zero address in the data memory, the processor uses the FSR register to store the data in the correct address. Indirect addressing also utilizes the value of the RP0 bit to access the banked areas of data memory. Figure 4.6 shows the mapping of the memories. As it can be seen, the first twelve locations of each bank are reserved for the special function registers and the remaining locations are used for the user-data memory. The reason to reserve those initial addresses unused is to maintain coherence with PIC16's architecture and its instruction set. In this way, despite the fact that the special function registers are implemented outside the data memory module, they can be accessed using their original addresses thanks to the IDC, which controls the enable port of each register.

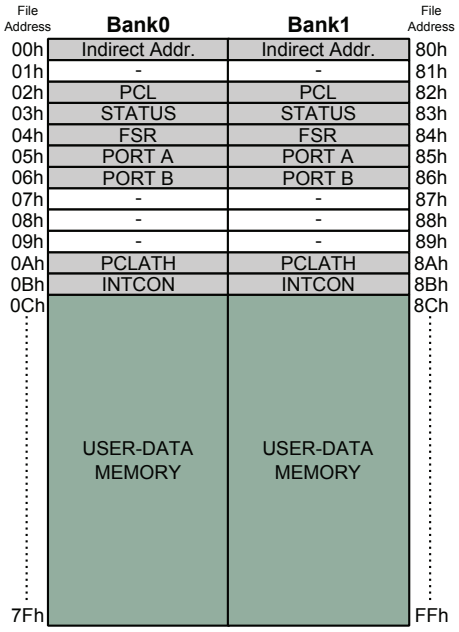


Figure 4.6: Mapping of PICDiY's user-data memory and registers.

The program memory block is responsible to store the instructions to be executed by the processor. Similarly to the data memory block, it is implemented by using BRAMs. However, the data length is 14 bits. Another remarkable difference is that it is implemented as a read only memory. Thus, the instructions must be written in the HDL file. In favour of easing the programming task, the HEX2VHD for PICDiY software has been developed in this work. This software generates a VHDL file of the program memory from a HEX file. This HEX file can be obtained in a straightforward fashion by using a PIC16-compatible IDE, like the MPLAB by Microchip Technology Inc.

The program counter block manages the execution sequence of program memory's instructions, loading the next instruction or jumping to another one when `CALL`, `GOTO`, `RETURN`, indirect instructions or an interrupt are executed. It is a 13 bit block capable of addressing an 8K x 14 memory space.

4.1.2 PICDiY's Instructions

The RISC architecture of the PICDiY, allows higher CPU operating frequencies and implies that it has a reduced number of instructions, which means simplicity in programming, since it is necessary to manage just a small number of instructions. As Table 4.1 shows, the processor uses 33 instructions of a 14 bit width, divided into byte-oriented file register operations (18 instructions), bit-oriented file register operations (4 instructions) and literal and control operations (11 instructions). It has to be remarked that, due to the removal of the timers, PIC16's `SLEEP` and `CLRWDT` instructions are not available. Each instruction takes a single machine cycle consisting of four clock periods for the execution, that is an improvement over the PIC16, since it uses two machine cycles for instructions which modify the value of the program counter. All of the instruction are stored in the program memory.

Figure 4.7 shows the general instruction format for byte-oriented file register operations. As it can be observed, the highest five bits are mainly used to define the opcode, the 7th bit is used to set the direction of result's value (d) for the ALU and the lowest 7 bits are the file registers address.

Figure 4.7: Byte-oriented file register operations instruction format.

Figure 4.8 shows bit-oriented file register operations instruction format. The

Instruction	Description	Instruction Code	Flags STATUS
Byte-Oriented File Register Operation			
ADDWF f, d	f + W	00 0111 dfff ffff	C, DC, Z
ANDWF f, d	f AND W	00 0101 dfff ffff	Z
CLRF f	Clear f	00 0001 1fff ffff	Z
CLRW -	Clear W	00 0001 0fff ffff	Z
COMF f, d	Complement f	00 1001 dfff ffff	Z
DECF f, d	Decrement f	00 0011 dfff ffff	Z
DECFSZ f, d	Decrement f, skip if 0	00 1011 dfff ffff	-
INCF f, d	Increment f	00 1010 dfff ffff	Z
INCFSZ f, d	Increment f, skip if 0	00 1111 dfff ffff	-
IORWF f, d	f OR W	00 0100 dfff ffff	Z
MOVF f, d	Move f	00 1000 dfff ffff	Z
MOVWF f	Move W to f	00 0000 1fff ffff	-
NOP -	No operation	00 0000 0xx0 0000	-
RLF f, d	Rotate left f through carry	00 1101 dfff ffff	C
RRF f, d	Rotate right f through carry	00 1100 dfff ffff	C
SUBWF f, d	f - W	00 0010 dfff ffff	C, DC, Z
SWAPF f, d	Swap nibbles in f	00 1110 dfff ffff	-
XORWF f, d	f OR W	00 0110 dfff ffff	Z
Bit-Oriented File Register Operation			
BCF f, b	Bit clear f	01 00bb bfff ffff	-
BSF f, b	Bit set f	01 01bb bfff ffff	-
BTFSC f, b	Bit test, skip if clear	01 10bb bfff ffff	-
BTFSS f, b	Bit test, skip if set	01 11bb bfff ffff	-
Control and Literal Operations			
ADDLW k	Literal + W	11 111x kkkk kkkk	C, DC, Z
ANDLW k	Literal AND W	11 1001 kkkk kkkk	Z
CALL k	Call a subroutine	10 0kkk kkkk kkkk	-
GOTO k	Go to k address	10 1kkk kkkk kkkk	-
IORLW k	Literal OR W	11 1000 kkkk kkkk	Z
MOVLW k	Move literal to W	11 00xx kkkk kkkk	-
RETFIE -	Return from interrupt	00 0000 0000 1001	-
RETLW k	Return with literal in W	11 01xx kkkk kkkk	-
RETURN -	Return from subroutine	00 0000 0000 1000	-
SUBLW k	Literal - W	11 110x kkkk kkkk	C, DC, Z
XORLW k	Literal XOR W	11 1010 kkkk kkkk	Z

Table 4.1: Instruction set of the PICDiY.

highest four bits are used to define the opcode, the next three bits are used to specify which bit will be written and the lowest 7 bits are the file registers address.

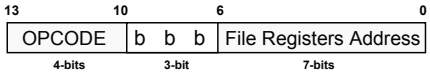


Figure 4.8: Bit-oriented file register operations instruction format.

Figure 4.9 shows the general instruction format for literal operations, where the highest six bits are generally used to define the opcode and the other 8 bits are the literal value.

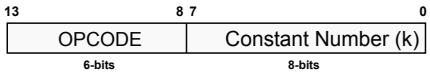


Figure 4.9: Literal operations instruction format.

In the case of `GOTO` and `CALL` control instruction, which are depicted in Figure 4.10, the highest three bits define the opcode and the other bits are used to set the destination address.

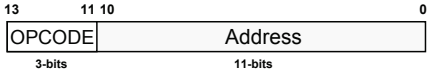


Figure 4.10: Control operations instruction format.

In order to store the actual instruction address when `CALL` instruction or an interrupt are executed, the program counter is connected with the 8-level stack, where the processor can store up to 8 addresses. It has to be noticed that the depth of the stack limits the number of `CALL` instructions to be used, since it can lead to a stack overflow. Nevertheless, the depth of the stack can easily be changed according to the application needs.

The interrupt system works level triggered and due to this, the interrupt signal must have certain specifications. First, the interrupt signal is active high and since the interrupt attention is done in one of the four states of the FSM, it is necessary that the interrupt signal remains stable at least four clock cycles to ensure that the interrupt signal is detected. There is also a maximum duration for the interrupt signal, as long as a signal that lasts for a long time, can call the interruption routine more than a single time. This maximum duration depends on the number of executed instructions in the interruption routine. In the worst

case, after the execution of the `RETURN FROM INTERRUPT` instruction the interrupt signal must be in low level. When, in order to respond to external events, the interrupt input is set to high level and the interrupt conditions are fulfilled, the program counter changes the program flow jumping to the fourth address of the program memory to execute the interrupt routine, after this execution it continues from the previous point on.

As the presented features imply, thanks to the basic and modular architecture of 8-bits, an easily adaptable, self-sufficient and platform-independent soft-core processor has been designed.

4.2 Bitstream Based BRAM Approach: Contribution in BRAM Data Management through the Bitstream in 7 Series

Due to their particular characteristics and the differences in the architectures of the bitstream, the existing methods for Virtex-V devices cannot be utilized to manage data using the bitstream in 7 series devices. No methods able to extract or write user data in BRAM memories nor in registers of 7 series by Xilinx devices have been found in the literature during this research work.

This section introduces the Bitstream Based BRAM Approach (BBBA) to access and manage the content of BRAM based memories using the configuration bitstream. This approach is able to access and manage data in BRAM based memory designs for 7 series FPGA SoC implementations. It access to the configuration bitstream to manage the data content of different BRAMs. Since this method does not add extra elements to the original design, there is no impact on the resource usage. This is relevant because the resource overhead involves longer datapaths, more power consumption, worse susceptibility and less resources available. Besides, although it is not a recommended procedure in the majority of cases, it provides the possibility of accessing the memory even when it is being read or written. Therefore, some of the common drawbacks of classical memory management strategies can be solved by using the proposed BBBA approach, such as, the resource overhead, the appearance of single points of failure or the lower availability of memory blocks. This and several other advantages unleashes a myriad of possibilities when working with BRAM memories. In addition, unlike other existing techniques related to different devices, by virtue of the use of the PCAP it avoids single point of failures that appear when utilizing reconfiguration interfaces that make use of logic resources like the ICAP port.

4.2.1 Proposed Method to Obtain the Bitstream Structure of Data in BRAMs

The aim of this method is to manage different BRAMs of the FPGA by solely resorting to the bitstream. For that purpose it is necessary to read and generate bitstreams containing the data to be loaded in such memories. As it has been presented in Section 2.2.4 Xilinx provides certain information about the bitstream in [4, 120] and in the *.ll* location file generated by Vivado during the write bitstream process. However, the information about the bitstream composition provided by manufacturer is not detailed enough. In addition, the utilization of the data provided by the *.ll* location file demands a huge processing effort and big memory storage to manage big data blocks. For these reasons reverse engineering has been performed so as to find out relevant information about the structure of data in BRAMs. In this section, the utilized setup and the different flows and software programs are described in depth.

As it has been previously introduced, the target device selected for this work is the Xilinx Z-7020 Zynq-7000 All Programmable SoC. Figure 4.11 shows a simplified block diagram of the overall scheme utilized. The I/O peripheral signals of the programmable logic and the Processing System are connected through the Extended Multiplexed I/O (EMIO) interface, allowing the I/O peripheral controller to access the programmable logic. The Processing System reaches the configuration data through the PCAP interface in order to perform a readback, which stands for the process of reading the bitstream data from the configuration memory. The software running in the Processing System used for the readback has been developed using the *XDevCfg* library. During the bitstream processing a DDR memory peripheral has been used to store the bitstream and additional application data. This simple approach permits isolating the configuration memory from any other component in the design.

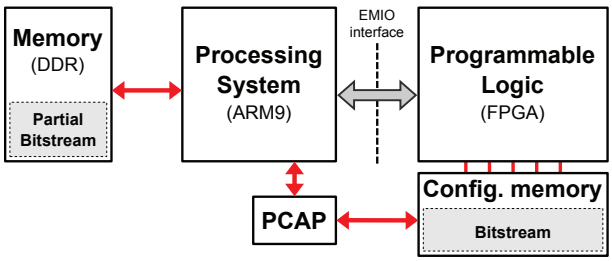


Figure 4.11: Block diagram of the implementation scheme.

The FAR address of each of the 14 BRAM columns has been identical to de-

termine in wich words of the bitstream the data content of BRAMs is stored. This information can be extracted from the *.ll* location file. However, it has been decided to to use reverse engineering for verification purposes. In this way, a reconfigurable region for each BRAM column has been created, and thereafter the generated partial bitstreams have been examined. The FAR addresses of all BRAM columns obtained are shown in table 4.2. In the next step each BRAM column has been studied under different implementations within the FPGA section. Considering that each column of the Z7020 contains 20 18K BRAMs, this set of implementations consists of 20 memories of 512Kb depth and 32 bits each. Using this memory size, each 18K-sized BRAM is entirely utilized. In order to ease the interpretation of the obtained data, all BRAMs have been specifically arranged by following an increasing order by using location constraints. The implementation also includes an FSM based memory filler module to write the data in all addresses in the memories and a multiplexer to selectively activate the write-enable ports.

Table 4.2: FAR addresses of BRAM columns in Z7020.

Resource	FAR (hex)	Resource	FAR (hex)
BRAM1	0x00c20000	BRAM8	0x00c00180
BRAM2	0x00c20080	BRAM9	0x00c00200
BRAM3	0x00c20100	BRAM10	0x00c00280
BRAM4	0x00c20180	BRAM11	0x00800100
BRAM5	0x00c20200	BRAM12	0x00800180
BRAM6	0x00c20280	BRAM13	0x00800200
BRAM7	0x00c00100	BRAM14	0x00800280

Figure 4.12 depicts an example of the utilized flow to determine in which words of the bitstream is located the data of each 18K BRAM. Starting with the first (BRAM which is located in the bottom part of the device), the memory filler module writes zeroes in all data positions within the memory. Right after that, the portion of bitstream of the column defined by the specific FAR and the number of frames of a BRAM column are read by using the readback functionality. This content is stored in the DDR memory. Subsequently, a similar process is done but writing all ones (0xFFFFFFF) instead. The data content obtained in both bitstreams is compared, storing the address and the content of each different word in a spreadsheet. These steps have been performed for every of the 20 memories of each column, after which new implementations have been carried out by changing the location constraints and by using a proper FAR to study all BRAM columns.

Different conclusions have been drawn from the analysis of the obtained information. To begin with, it has been confirmed that the content of each BRAM is distributed along the partial bitstream of its BRAM column. This is probably

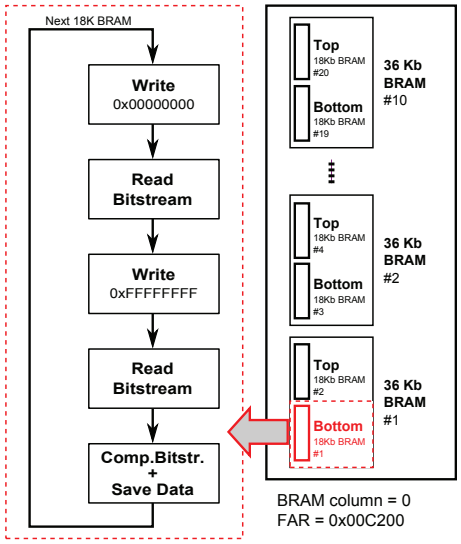


Figure 4.12: Example of the flow used to determine the BRAM data location in the bitstream.

due to the interleaving mechanism. Moreover, there is no direct correspondence between the data bits of an implemented memory and the data bits within the bitstream. Table 4.3 shows an example of data distribution for a single 32-bit data word over the bitstream. In this plot, the memory is constrained to the first 18K BRAM of the first column and the data content written is all ones. The first column of the table indicates the bit position in the data word. The second column shows the relative position of the word in the data portion of the partial bitstream. Finally, the last column denotes which bit of the particular word of the bitstream is affected by changes in the memory. The next memory words follow different distribution rules to place their bits in the bitstream. In some cases changes in a single bit of the user memory may lead to changes in two bits of the bitstream. Regarding the amount of data bits to handle, inferring the complex relation between them requires a big processing effort. Even though working at bit level is interesting in certain applications where the memory space of words to be managed is not so big, in general it is not as practical as working with entire BRAMs. For this reason this approach has focused on managing data content of entire BRAMs instead of controlling changes over single bits or words, hence paving the way towards future developments. Anyway, if working at bit level is demanded, it is recommended to utilize the information from the *.ll* location file.

Table 4.3: Bit distribution example of the first data word of the first 18K BRAM in Z7020.

N. Bit	Word Addr.	N. Bit in Word	N. Bit	Word Addr.	N. Bit in Word
0	0	0	16	2	0
1	0	16	17	2	8
2	1	0	18	1940	0
3	1	16	19	1940	16
4	2	16	20	1941	0
5	3	0	21	1941	16
6	3	16	22	1942	16
7	4	0	23	1943	0
8	0	8	24	1943	16
9	0	24	25	1944	0
10	1	8	26	1940	8
11	1	24	27	1940	24
12	2	24	28	1941	8
13	3	8	29	1941	24
14	3	24	30	1942	24
15	4	8	31	1943	8

The remaining conclusions are related to the location of data words within the bitstream. First, the location scheme of data words in each column is the same. That is to say, all BRAM columns have the same word organization structure to store data within the bitstream. Second, the data content of each 18K BRAM is placed by performing cyclic jumps along the frames of the bitstreams of BRAM columns. These jumps can be noted in Table 4.4, which depicts an example of the data organization within one frame of the two 18K BRAMs (top and bottom) that compose a 36K BRAM. The data content written in the memories for this example is all ones, thus the data content in all words of the frame is composed of 'F's. The jump sequence begins with an address jump defined by the product of the frame number (relative to the partial bitstream of the BRAM column) and by the initial address of the particular BRAM. Table 4.5 displays the initial addresses for each 18K BRAM of a BRAM column in the Z7020. The next four addresses are also consecutively used to store data. Thereafter a jump of 101 words to continue in the next frame. As it can be also observed in Table 4.4, the data organization in the top 18K BRAM differs from that in the bottom 18K BRAM. For instance, referring to Figure 4.12 the data in the first 18K BRAM is arranged differently when compared to the second or the fourth BRAM, but similar to its third or fifth counterpart. In the case of 36K BRAMs, since they are composed by a 18K BRAM pair (top and bottom), the organization in all of them is the same. Albeit feasible the inference of the relationship between the top and bottom 18K BRAM has been discarded due to its complexity (it is necessary to process at bit level) and subsequent lack of practicability for real applications. In this way, this approach supports the inner data exchange between BRAMs at

the same deployment level (i.e. top-top or bottom-bottom).

Table 4.4: Data organization example of one frame of a 18K BRAM column in Z7020.

Data word address	Data in bottom BRAM	Data in top BRAM
Init addr. + (frame number × 101) + 0	FFFFFFFF	FFFF0000
Init addr. + (frame number × 101) + 1	FFFFFFFF	FFFFFFFF
Init addr. + (frame number × 101) + 2	FFFFFFFF	FFFFFFFF
Init addr. + (frame number × 101) + 3	FFFFFFFF	FFFFFFFF
Init addr. + (frame number × 101) + 4	0002FFFF	FFFFFFFF

Table 4.5: Init addresses (hex) of 18K BRAMs in Z7020.

BRAM num.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Init addr.	0	5	A	F	14	19	1E	23	28	2D	33	38	3D	42	47	4C	51	56	5B	60

4.2.2 Managing Data Content of BRAMs with the BBBA

As argued in Section 2.2.5, the *Bitstream Based BRAM Approach* (BBBA) provides a great potential for a number of applications. Apart from data reading and writing, the most essential operations required to implement many common applications are data copy and comparison. This section delves into these two operations, with emphasis on how to apply them using the bitstream.

BRAM Content Copy using the Bitstream

The process to copy the data stored in BRAMs using the bitstream can be held in two main scenarios: 1) data copy of entire BRAM columns and 2) data copy of individual BRAMs. Although the concepts of the latter case are also directly applicable to groups of consecutive BRAMs of 18K or 36K, for the sake of simplicity and clarity, this document will refer to individual 18K BRAMs. At this point it is also important to recall that, in the case of 18K BRAMs, if the source BRAM is a top 18K BRAM the destination must be also a top BRAM (and vice versa). Moreover, it is important to remark that it is not necessary to implement the destination memory in a reconfigurable region of the design, this approach works properly in both, reconfigurable and non configurable areas.

When dealing with single BRAMs, two subcases can be also distinguished depending on whether the data stored in the destination BRAM can be overwritten or not. Overwriting data is useful when the content of all the BRAMs of the column is not relevant, since this method writes the copied data in the destination BRAM and resets the rest of data bits of the other BRAMs within the column.

The second sub-case arises when the content of other BRAMs from the destination column want to be preserved, which comes along with penalties in terms of code and execution overheads. From a global perspective, all the aforementioned cases comply with the flow diagram shown in Figure 4.13, which hinges on reading the partial bitstream of the source BRAM and copying its data content into the destination BRAM column. The main difference between the three cases resides in the frame reading and data processing tasks.

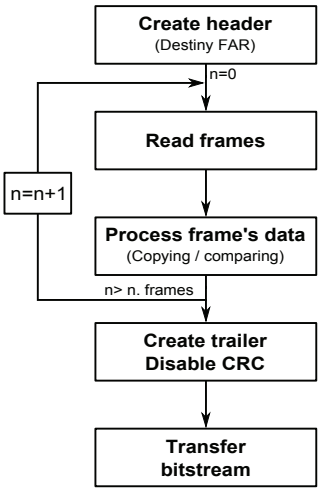
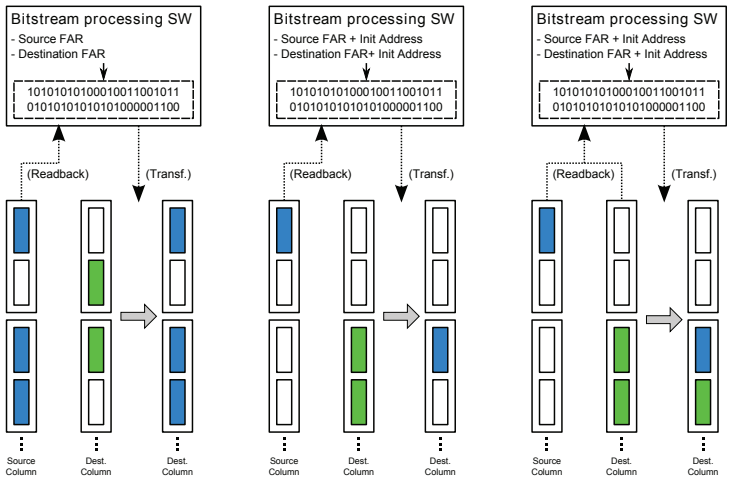


Figure 4.13: Flow diagram of the BRAM copy procedure.

The first task of the flow consists of creating the header of the new partial bitstream that will be transferred with the copied data. The header of the BRAM column's partial bitstream is composed by 30 words of 32 bits. This header is almost the same for each BRAM column, being the difference that the 27^{th} word of this header contains the FAR address of the destination BRAM column.

After creating the header, the data content must be stored. This task is carried out in two steps that are performed for each frame within the partial bitstream of the BRAM column. First, the frames of the source and, if needed (only necessary when the data overwriting has to be avoided), the destination BRAMs have to be read by utilizing the readback functionality. The data content from the source frame has to be processed in order to organize it properly for the destination BRAM. After performing a readback, several bits of the obtained bitstream are set to logic high. Since, these bits are the mask bits that protect the content of BRAMs, they must be reset in order to transfer the bitstream successfully. Each



(a) Entire BRAM column copy. (b) BRAM copy with overwriting. (c) BRAM copy without overwriting.

Figure 4.14: Different copying types.

of these bits is located in the 18th position of the 4^{th}, 14^{th}, 24^{th}, 34^{th}, 44^{th}, 55^{th}, 65^{th}, 75^{th}, 85^{th} and 95^{th} words of the data section of each frame. For this reason, during the processing task all set bits have to be reset by resorting to a binary mask.

The specific data processing performed on the data content differs among application cases. Figure 4.14 exemplifies the three most basic scenarios: entire BRAM column copy, single BRAM copy overwriting the rest of the BRAMs of the column and single BRAM copy without overwriting. However, based on these cases it is possible to implement more complex applications, such as, copying various BRAMs into different locations or swapping data between distinct columns, etc.

The most elementary case is to copy entire BRAM columns. Taking into account 1) that the data organization has found to be equal in all BRAM columns and 2) that the content of the destination BRAM column is not necessary, only the source frames have to be read, copying them directly onto the destination frames. Obviously, in this case all BRAMs from the destination column are overwritten.

In the case of copying single BRAMs with overwriting, the process is slightly different because the data read from the source frame must be reorganized to be placed properly in the destination BRAM. Thus, the exact placing of the source

and destination BRAMs must be known. The use of placement constraints is very useful in this context, but it is also possible to determine the placement by analysing the implementation results. Based on the known positions of the BRAMs in the columns and by using Table 4.5 the initial addresses can be obtained. With this information and following the address jump pattern described in the previous subsection data can be properly arranged and written in the new partial bitstream. It is important to note that if the source BRAM and the destination BRAM are placed in the same column, the content of the source BRAM will be lost.

Copying single BRAMs without overwriting is more complex, especially in terms of program code (in quantitative more than twenty times more lines of code than with overwriting), because it is necessary to take into account several hypotheses. The data of the destination column must be read and copied in the new partial bitstream, and the bit `set` must be reset in all the words where they are set in the readback. By contrast, the cases with overwriting only require resetting the words of the read BRAM because the rest of data words are sent empty. Another singular aspect when copying without overwriting is that if source and destination columns are the same, the program is more efficient since only one readback must be performed for each frame. In any case, the initial addresses of source and destination BRAMs and the address jump pattern must be known beforehand.

After organizing and writing all the data, the trailer must be added in the end of the created partial bitstream. Most of the trailer words mean "no operation" (NOP) words, but it also contains CRC data. Considering that as it has been mentioned the CRC method for 7 series is still unknown [117], these words related to the CRC are written with the NOP word. In this manner, the trailer is the same for all created partial bitstreams. Finally, the produced partial bitstream is transferred to the device via PCAP port, reconfiguring dynamically and partially the specific logic portion.

BRAM Content Comparison using the Bitstream

BBBA can be also used to compare the data content of different BRAM columns or single BRAMs. In both cases, if the data content is changing during the running process it is advisable to halt the system so also avoid data changes during the bitstream reading process and obtain a correct result. However, in those cases where the information stored in BRAMs remains unchanged for certain time (i.e. program memories) the comparison can be performed without stopping the system. Depending on the requirements the comparison can be performed in

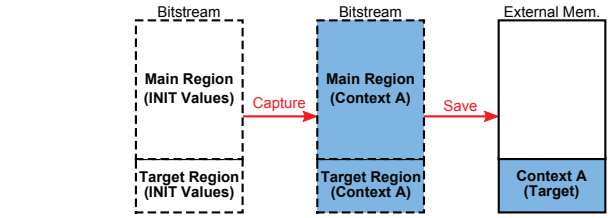
terms of different levels of depth. When the comparison aims at only determining whether the contents of the memories are equal to each other (as in e.g. the detection of SEUs) the detection of the first discrepancy suffices for stopping the comparison process. In other cases, a deeper comparison can be required for e.g. determining the fraction of unequal bits. To perform this comparison between entire BRAM columns the only information required is the FAR of each BRAM column. By using the readback function all the frames can be read and compared directly.

When working tackling single BRAMs different scenarios can be held. When the BRAMs to be compared are placed in the same position of different columns the way to proceed can be the same to the column comparison. The main difference is that is not necessary to compare all words within each frame. Provided that the initial address and the address jump pattern are known it is possible to know which specific words should be compared. If the BRAMs to be compared are placed in different positions within the same (or different) BRAM columns, the data content has to be processed in the same way that the copying cases in order to compare the right data words. In this case also, comparing BRAMs of the same column is more efficient than comparing BRAM from different column, since it is only necessary to read the partial bitstream of a single column. It has to be reminded again that this approach works with top or bottom 18K BRAMs separately. For this reason, top 18K BRAMs are compared with other top BRAMs and vice versa with bottom 18K BRAMs.

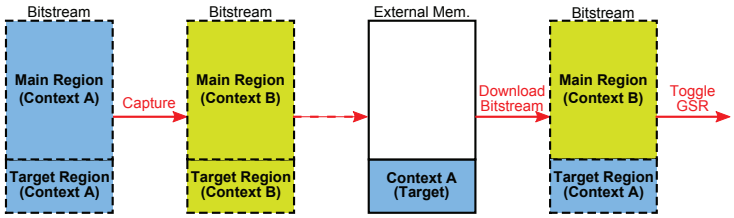
4.3 Approach to Manage Data of Registers with the Bitstream

Similar to what occurs with the process of managing data of BRAMs, the approaches to manage data of register using the bitstream developed for Virtex-V devices are not applicable to 7 series. When dealing with memory modules implemented as distributed memory, the strategy from [8, 61] could be a valuable starting point, making several adaptations. First, `STARTUP_VIRTEX5` and `CAPTURE_VIRTEX5` primitives of Virtex-V devices have to be replaced by `STARTUPEE2` and `CAPTUREE2` primitives of 7 series, respectively. An additional interesting adaptation is to substitute ICAP interface by the PCAP interface, avoiding the utilization of additional resources necessary to implement the ICAP.

Nevertheless, the adaptation of these approaches to 7 series devices presents a relevant issue, since unlike with Virtex-V devices, Xilinx has not released any information about protection/unprotection bits of 7 series. Hence, the only option



(a) Context save.



(b) Context restore.

Figure 4.15: Context save and restore approach for 7 series devices with external memory.

available nowadays is to capture and to restore content of all the registers in the device. This implies a relevant limitation, especially when to read/write data from/to particular areas is needed. In such case, as Figure 4.15 depicts, the process of context saving and restoring requires a relatively tedious approach. The first step to save the context is to capture the state of all registers. In an initial state the bitstream contains the INIT values predefined in the VHDL. In case of triggering the capture signal without protecting any region, the content of all flip-flops is stored in specific locations of the bitstream. Thus, the data of the target region has to be extracted from the entire bitstream and saved in an external memory (i.e. a DDR memory). This action requires to utilize and process the information from the previously generated logic location text file (*.ll). In order to avoid overwriting the actual value of the rest of flip-flops, it is necessary to capture their context in order to merge it with the previously stored one. Once the merge process is done, the created partial bitstream can be downloaded to the FPGA updating the content of the flip-flops memory loaded by the `GSR` signal. It has to be taken into account that the `GSR` signal has no timing specifications and that it spreads across the device slowly. Hence, it is advisable to stall the system for a certain period and/or to cluster the register tightly.

This presented alternative does not require to modify the bitstream and it is

valid for any design (even for the not reconfigurable ones). However, it is a time demanding process, since it requires several operations (i.e. saving the context into the external memory, capturing the actual state, downloading partial bitstreams, etc.) in each context saving/restoring action. This implies a significant time demand. It also requires an external memory and processing the bitstream to extract the data for each capture and restore. To circumvent these drawbacks, this work proposes a method to unprotect as many as need reconfigurable regions. I this way, maintaining the rest of the design protected the necessity of saving the data from the other regions (those can are to be protected) can be avoided.

On the other hand, when utilizing a bitstream based strategy to copy the content of registers from one reconfigurable region to others', one of the biggest drawback is the processing effort demanded. This aspect gains special relevance in designs with a significant amount of registers, such as soft-core processors. The main reason behind this disadvantage is that, even using the same HDL design for different reconfigurable blocks, due to the default resource placement freedom of the implementation tool, it creates a distinct implementation for each block. This results in different locations for the same resource in the different reconfigurable blocks. Hence, the location of each data bit within the bitstream varies, increasing the complexity of the relocating processing when copying data through the bitstream. In this scenario, the source and destination locations of each data bit of each flip-flop must be identified in the *.ll* file in order to utilize this information during the copying process. It also implies that, in the majority of cases, the copying of each data bit requires a combination of a reading and a writing operations. In addition, in several cases the portion of the destination bitstream (the bitstream portion where the captured data is going to be copyied) has to be read in order to preserve the content of other registers that save the content in the same bitstream frame. This case also requires a process to merge the captured content with the saved information of the destination bitstream. All those steps have to be performed in each context saving/restoring action, which results in a remarkable instruction overhead. It also requires a complete study of the *.ll* file.

For this reason, an interesting solution could be to obtain exactly alike implementations in the different regions for each design, placing each resource in the same positions of different columns (reconfigurable regions). This would allow to perform the copying process by copying entire bitstream portions and using a significantly simpler relocation process and avoiding the need of data merging. Hence considerably reducing the number of instruction required. However, obtaining identical implementation of a design in different reconfigurable regions is a tedious task. Therefore, this work proposes a design flow which allows to ease this task by making use of the location constraints.

4.3.1 Proposed Flow to Protect/Unprotect Partial Regions in 7 series

Following the idea suggested in [124], an interesting alternative could be utilized to read and write registers' data of a particular region without affecting the rest of the device. As that work states, a property named `RESET_AFTER_RECONFIG=TRUE` [108] can be utilized with 7 series devices to implement partially reconfigurable modules in order to avoid the manual unprotection/protection actions. When implementing partial reconfigurable designs with this property in 7 series (for UltraScale devices is always enabled), the partial bitstream created by Vivado contains the commands to protect the rest of the device (including static and remaining partial regions), maintaining the actual partial region unprotected. Hence, this property enables to protect a particular partial region in a straightforward fashion. Nevertheless, as Figure 4.16 shows, this method can only unprotect (open padlock) a single partial region at once. As it can be observed, every downloaded partial bitstream unprotects the target reconfigurable module and protects (closed padlock) the rest of the device, even if the a partial reconfigurable module has been previously unprotected. Thus, there is no way to unprotect more than one partial region in the same device.

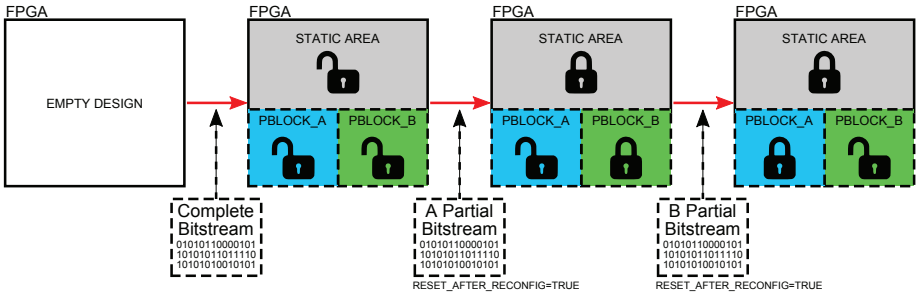


Figure 4.16: Effect of the `RESET_AFTER_RECONFIG=TRUE` property in FPGA protection.

With the aim of solving the issue of unprotecting more than a single partial region in 7 series devices, a novel design flow has been developed in this work. As a first step, two reconfigurable regions (*PblockA* and *PblockB*) have been designed. In the implementation process only *PblockA* has been implemented with the `RESET_AFTER_RECONFIG` property. After that, both generated partial bitstreams have been analysed and compared. The comparison has shown remarkable differences in the content, making it difficult to identify any special bit or command word. However, after a further analysis, a special word in 51st position of the frame with a particular content (0xE00009BC) has been observed

through the partial bitstream. Afterwards, a new comparison has been performed to detect discrepancies in the 0xE00009BC special word. The comparison results indicate that only two 0xE00009BC words appear in the *A* partial bitstream and do not appear in the *B* partial bitstream. Hence, it has been initially assumed that these two words were the protection words for the *B* reconfigurable region. In this way, different tests has been carried out modifying the partial bitstream of *PblockA*, downloading it and performing physical tests. These tests have concluded that erasing one of both special words has an effective protecting/unprotecting effect of *PblockB*.

Using the obtained information from the test the procedure described in Figure 4.17 has been developed. It utilizes the `RESET_AFTER_RECONFIG=TRUE` property in combination with the edition of the partial bitstream, as a straightforward method to protect or unprotect as many as desired partially reconfigurable modules. This edition only requires to erase or to add the proper 0xE00009BC special word of the previously generated partial bitstream.

Once the partial regions have been unprotected, the content of the registers of the unprotected regions can be easy read and written by using the `GCAPTURE` and the `GRESTORE`, respectively without affecting the protected regions. When using the `RESET_AFTER_RECONFIG=TRUE` property it is advisable to also utilize the `SNAPPING_MODE` constraint, which automatically creates legal reconfigurable blocks. This method, combined with a proper bitstream processing, also enables to modify data of registers, for instance for copying the content from one register to another.

The proposed method has a number of advantages and drawbacks that should be taken into account in order to choose the most adequate solution for each design. The most remarkable benefit of this alternative is that it is relatively fast, especially when capturing and restoring the context, because both `GCAPTURE` and `GRESTORE` operations are not time consuming processes. Thus, the time requirements are related to the time needed by the `GSR` signal to spread across the partial reconfigurable circuit, which is design dependant. In addition, this approach does not demand any additional element, neither external memories nor logical resources.

The most relevant drawback of this approach is the need of implementing partially reconfigurable blocks. In fact, when using the `RESET_AFTER_RECONFIG=TRUE` property, the partially reconfigurable module's height must align to clock region boundaries, which means occupying an entire column of resources (there is no block width restriction). Depending on the design, this could limit the available resources for other purposes. It is interesting to mention that in UltraScale devices there is no height requirement, since the `RESET_AFTER_RECONFIG=TRUE`

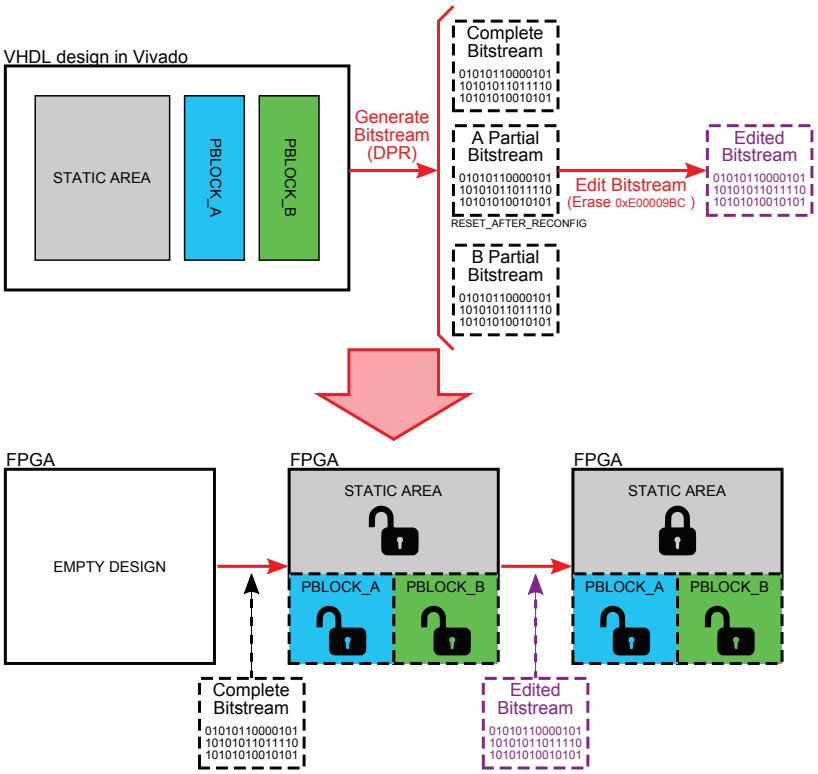


Figure 4.17: Approach to unprotect several regions in 7 series devices based on the `RESET_AFTER_RECONFIG` property.

property is always enabled. Moreover, if the XADC component is used, its interface cannot respond during the partial reconfiguration period, blocking its access. Another remarkable disadvantage of the use of the `RESET_AFTER_RECONFIG=TRUE` property is that, since the partial bitstream created contains the commands to protect the rest of the device and to unprotect the partial region, the size of the bitstream significantly increases.

Another important limitation when using approaches that utilize partial reconfiguration schemes is that the maximum achievable operating frequency of the design can be reduced. Although two identical designs, one static and another reconfigurable, can be logically exactly equal from an RTL description point of view, the way both are synthesized and implemented is significantly different. The partial reconfiguration flow demands to synthesize each reconfigurable module out of context, which limits cross-boundary optimizations. The reason for this is to guarantee that logical interfaces between static and reconfigurable partitions remain fixed. During the implementation, the Pblocks are required to physically divide the static and reconfigurable partitions. This implies layout requirements that also restrict the optimization of the placement process.

4.3.2 Proposed Flow to Generate Equal Implementations of a Design in Different Reconfigurable Regions

This section presents the proposed design flow to obtain identical implementations of a particular HDL design in distinct reconfigurable regions, which for the sake of simplicity, will be referred to it as *Location Constraints Flow*. This flow is specially interesting in some desings that make use of hardware redundancy strategies. The main problem when trying to obtain three alike implementations resides in the implementation tool, which in order to optimize the design process, places each resource of the replicated instances in different locations. This generates totally distinct implementations.

Since the partial bitstream generated for each reconfigurable block only can be used in its particular reconfigurable region, the most suitable solution is to utilize location constraints when implementing each instance. Nevertheless, the utilization of location constraints with a high number of resources, and specially in reconfigurable designs, is a tedious procedure.

The most simple alternative is to place all the resources of each instance manually in the same custom positions of each resource column utilized by the different reconfigurable regions. After synthesizing the entire HDL design, this placement task can be easily done by means of Vivado's graphical interface by selecting each resource within the netlist and dragging it to the chosen resource position.

In designs with a high resource usage, this task can be highly time consuming. An alternative to ease this placement task is to utilize TCL commands of Vivado. Nevertheless, this requires to previously obtain the information of both, the netlist and the device's resources locations. In any case, the designing costs of this alternative are significantly high. But worse still is the fact that using this strategy requires a deep designing knowledge in order to obtain a proper implementation. Bearing in mind the advanced algorithms utilized by Vivado's implementation tools, it is highly unlikely that a designer could obtain such an efficient implementation as Vivado. Hence, using this alternative implies high designing costs and low efficiencies.

In order to overcome these drawbacks, this work proposes the *Location Constraints Flow* shown in Figure 4.18. This flow based on utilizing the resource placement of one of the instances generated by Vivado to place the resources of the rest of the instances.

Figure 4.18: Flow chart of the *Location Constraints Flow*.

The *Location Constraints Flow* starts by implementing the previously synthesized design. This results in an implementation containing the resource placement of each instance. Due to this, it is possible to take advantage of the processing effort carried out by Vivado to generate an optimized implementation. Once a proper implementation has been obtained, the next goal is to utilize the placement of one reference instance to generate new placements for the rest of the instances. Due to this, in a second step, the `write_xdc` TCL command is used in order to obtain the placement of all the resources of a design. This command generates a *.xdc* file that specifies the particular cell utilized by each resource, providing a detailed list that contains the placement of all resources utilized by each instance. Bearing in mind that there is a possibility of trying to relocate a resource in a location occupied by other resource from the original implementation, it is highly advisable to first unplace all resources. For this task, a script can be generated by using the `unplace_cell` TCL command combined with the information extracted from the *.xdc* file. Once the resources have been unplaced, the next step is to place them in their proper locations based on the locations of the reference instance. This step can be also carried out following the same TCL script based strategy. Nevertheless, in this case it is necessary to specify the new locations to the `place_cell` command. This can be carried out in a straightforward fashion by simply changing the number of column (X coordinate) of each SLICE from

the reference instance. Once the placement constraint script has been loaded, the `route_design` and `write_bitstream` commands can be executed to obtain the bitstream and to create the complete and the partial bitstreams.

Thanks to the proposed *Location Constraints Flow*, an implementation with a proper placement can be obtained with minimal designing costs. In addition, it almost preserves the implementation tool's efficiency, barely affecting performance when compared with an unconstrained design.

4.4 Data Content Scrubbing Approach

Related to soft-core processors, performing data content scrubbing is an interesting technique to fix errors in memories. This is because most used techniques to harden memory elements (TMR, ECC, etc.) mask errors and do not correct them. Nevertheless, data content scrubbing is mainly advisable for program memories, because the content remains unchanged. For the rest of memory elements (data memory, registers, etc.) the data scrubbing usually is not practical because the content is constantly updated during operation. However, even with those elements, in certain cases, this method could be an interesting alternative to initialize specific memory positions or registers. For instance, this could help to avoid the need of initialization instructions after resetting the system.

When creating a partial bitstream that includes initialized memory elements the bitstream generated contains the initialization data for such elements. Downloading this partial bitstream would refresh data content of memory elements initialized by design. Despite this procedure could be considered as a data content scrubbing method, it has to be remarked that this implies to reconfigure the entire reconfigurable block. Hence, the reconfiguration will affect all elements of the reconfigurable region, which in many cases could not be feasible. In addition, this practice requires to stop the operation of the reconfigurable block and commonly resetting it. Moreover, since the partial bitstream contains many more frames in addition to the frames with the data content, it increases the time demand of reconfiguration.

In order to circumvent these issues, a *Data Content Scrubbing Approach* for memory modules implemented with BRAMs based on the previously presented BBBA is proposed. It utilizes its simple implementation scheme shown in Figure 4.11. The proposed approach consists in downloading the data content of the BRAM memory by applying the BRAM copying method (with the BBBA). Depending on the number and placement of the BRAMs utilized by the remaining memory modules of the design, the most suitable alternative has to be chosen. This is

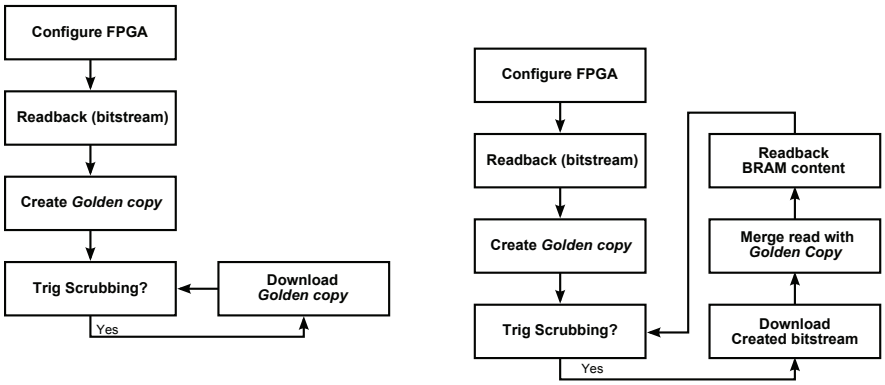
because it has to be determined if the actual content of the remaining BRAMs has to be preserved or not. Thus, while in cases where a BRAM column only contains memory blocks to be scrubbed, all the three proposed alternatives are suitable. In implementations in which program and data memories are placed in the same BRAM column, the BRAM copy without overwriting approach has to be adopted in order to preserve the information of data memory. Bearing in mind that the BRAM copy without overwriting approach demands higher processing effort, and consequently more time, it is advisable to avoid its utilization, when it is possible.

Considering that depending on the design the BRAM column could be overwritten or not, two design flows are proposed to perform data content scrubbing. As it can be seen in Figure 4.19, both flows share the first four steps, which are used to obtain the *golden copy* bitstream. The first step is to configure the FPGA by downloading the bitstream. In a next step, a readback process is performed through the PCAP interface. Thanks to this, the data obtained is properly structured and can be utilized to create the partial *golden copy* bitstream file in a straightforward fashion. These first steps could be substituted by extracting the data from the bitstream file. However, due to the complex data distribution in the bitstream file this process would require a high processing effort. In addition, each design requires specific processing due to the different BRAM locations.

Once the *golden copy* is obtained and stored in an external memory (i.e. a DDR) the scrubbing process can be triggered. In the case that the remaining BRAMs of the BRAM column can be overwritten the *golden copy* can be directly download as is described in Figure 4.19(a). But in cases were they cannot be overwritten, as depicted in 4.19(b), a readback operation has to be performed in order to store the actual data content and to be able of merging it with the *golden copy*. Depending on the design this process may require to stop the clock signal in order to assure stable information.

In this way, in contrast to what [184] states, by using BBBA it is feasible to perform an user memory scrubbing in BRAM based structures by using the bitstream information. Just as it occurs with the configuration scrubbing, neither resource overhead nor performance penalty are incurred by the design. In addition, thanks to this proposed approach, the memory coherence problem described in Section 3.1 can be solved. Besides, in designs where the BRAM column can be overwritten the configuration scrubbing can performed in runtime without stopping the system, thus, without adding any performance penalty.

One limitation of this approach is the scrubbing frequency, since the time demanded by BBBA affects to the time between consecutive scrubbings, which is less the upper bound for the majority of the applications. Another drawback of



(a) Content scrubbing with overwriting.

(b) Content scrubbing without overwriting.

Figure 4.19: Flow charts of the *Data Content Scrubbing Approach*.

this technique is that it demands a Processing System and an external memory.

4.5 Approach to Extract Data From Damaged Memories Using the BBBA

A potential consequence of SEUs is the damage on the interfaces of memory resources. Thus, as Figure 4.20 describes, if such an error disables or affects to relevant ports of the memory it is likely to provoke permanent damages. Moreover, if ports, such as, `address`, `reset`, `clock` or `data output` are affected by the fault, this can eliminate any possibility of recovering the data stored in memories, at least with regular off-the-self methods. This case can be more likely to happen in cases where memories have not been hardened. This situation could be especially critical if the information stored has a special value. The method based on the BBBA technique proposed in this section allows to recover the data content stored in BRAM based memories with damaged interfaces in a straightforward manner.

Due to the simplicity of this approach it could be considered as a specific application of the BBBA. Considering that it only uses the BBBA it shares its implementation scheme shown in Figure 4.11, requiring a Processing System and an external memory. It is mainly based on utilizing the bitstream information

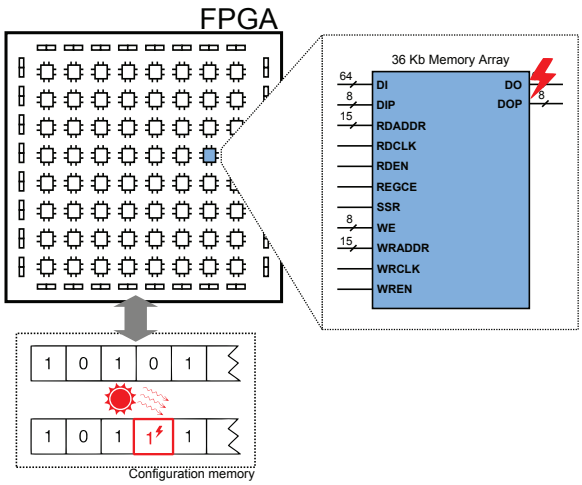
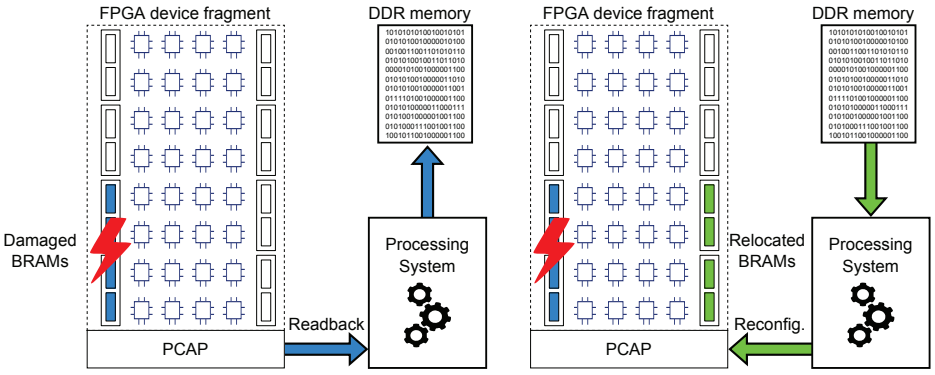


Figure 4.20: Example of an SEU affecting the interface of a BRAM.



(a) Extracting data from BRAMs. (b) Relocating saved data.

Figure 4.21: Simplified example of extracting and relocating data from a damaged memories.

obtained after using the BBBA as described in Figure 4.21(a). After acquiring the bitstream portion with data content, different alternatives can be utilized decode it. Since obtaining data content directly from a bitstream is a complex process, the most effortless way is to relocate the saved bitstream downloading it to a not damaged memory of the design with a similar implementation as shown in Figure 4.21(b). However, since this is an unlikely scenario, an alternative can be to implement the original design (or a simpler design with only the damaged design) in an additional FPGA device and downloading the read bitstream. Logically, this approach is not suitable in cases where the fault affects the bitstream content related to the BRAM data.

4.6 Lockstep Approaches

This section proposes three approaches to deal with the most challenging issues of lockstep schemes, which are related to checkpointing and rollback processes. While the first approach adopts a full hardware solution to provide fast checkpointing and rollback, the second applies the BBBA to reduce resource usage by utilizing bitstream information. Finally, the third proposed approach combines the first two methods to provide a trade-off solution.

All three approaches have been developed focusing on recovering from faults in user memory elements. In this way, a fault fixing of the configuration memory has not been considered. Nevertheless, it is possible to add this aspect to the approaches by performing a configuration scrubbing after the error detection. In this case, it is necessary to mask configuration bits related to user data content elements of the *golden copy* to protect saved data against overwriting during the scrubbing.

The proposed approaches have been applied to the PICDiY designed in this work. However, they can be adapted to other existing soft-core processors in a straightforward manner. In addition, different ideas from proposed techniques can be also applied to other designs that share characteristics with soft-core processors, like memory modules. Hence, these approaches can be valuable tools when hardening soft-core processors with DMR techniques, providing the designers with more alternatives when making trade-off design decisions.

4.6.1 Hardware Based Fast Lockstep Approach

This section presents a novel lockstep approach which is focused on circumventing the problem of the latency in context saving (checkpointing) and reloading

(rollback) processes. This approach has been named as *HW Fast Lockstep*. The reduced latency of the *HW Fast Lockstep* is obtained by introducing hardware changes in the design of the soft-core processor. Following the line of research of this work, the selected soft-core processor to work with has been the PICDiY. Utilizing this design as a basis, several adaptations have been made to achieve a fast context saving and restoring lockstep approach. Furthermore, a Lockstep Controller and a Backup Memory modules (for the context saving) have been added to the architecture. However, the storing elements must also be hardened to achieve an effective fault tolerance level. For this reason, data and program memories are protected by an ECC encoding and decoding implementation.

Figure 4.22 shows the implemented lockstep system in which two units of the adapted processor work at the same time and share program and data memories. As it can be observed, this approach implies a medium grained DMR scheme. The Lockstep Controller block manages the behaviour of the system in presence of errors and the synchronization of both processors. The sharing of memories and registers between the processors is performed using two different mixer blocks. Thus, there is a mixer structure for the program memory that merges the data address and the read enable inputs from each processor. There is also a mixer for the data memory which merges the data address and writes enable inputs and data input / output. The last mixer is used to mix the data off the registers of both processors (FSR, STATUS, PCLATH, INTCON, PORTB and WORK) and its output is connected to the backup memory for a direct context saving. These mixer blocks are continuously comparing the signals from the processors and they report to the Lockstep Controller when a discrepancy is detected. When that happens, the Lockstep Controller forces the context load of the IDC blocks of both processors, loading the last correct context saved in the backup memory.

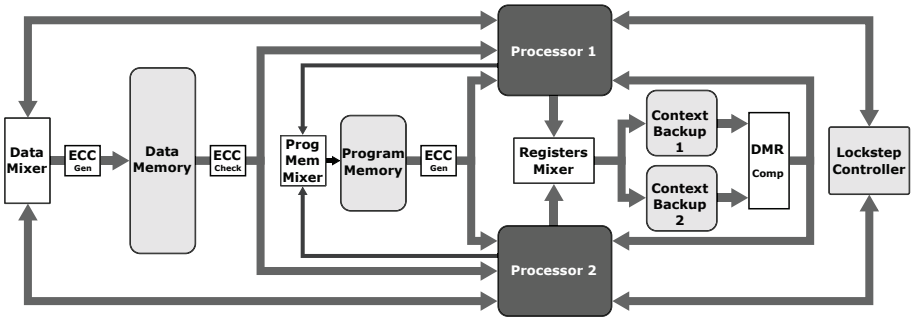


Figure 4.22: Simplified block diagram of the *HW Fast Lockstep* approach.

Several aspects of the original PICDiY design have had to be modified. One

of the most important adaptation has been made in the IDC module. Since it controls most of the elements during the execution of instructions, it has been modified to adapt its behaviour. The FSM of four states that manages the IDC has been adapted by adding an new state to perform the context saving and reloading. Figure 4.23 shows the flow diagram of the adapted FSM. As it can be observed, in the absence of errors the execution follows the original sequence of four states. In the first state, the IDC activates the instruction reading from the program memory. The second state, depending on the loaded instruction, is used to read data or to load the next instruction on the program counter when `CALL`, `GOTO` or `RETURN` instructions are executed. In the third state, if necessary, the IDC enables writing data memory or registers and the fourth state is used for data stabilization, interruption attention management and context saving in the backup memory. Thanks to the adaptation, during the fourth state, each processor sends out a signal to the Lockstep Controller to be used in the synchronization of both processors. When an error is detected the state machine changes to the new context reload state, loading the last correct context saved into both processors. The checkpointing is regularly performed every 4 clock cycles without any extra latency for the processors. In the same way, the rollback only takes a 2 extra clock cycles.

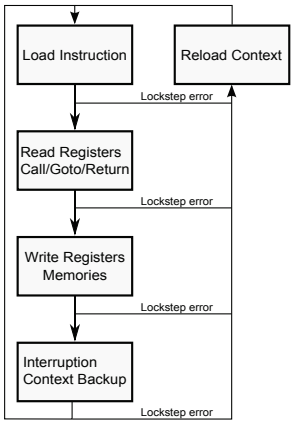
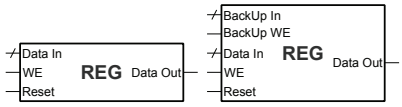


Figure 4.23: Finite state machine diagram of the adapted FSM.

Another significant modification has been made in the processor's registers to permit data load simultaneously. As shown in Figure 4.24. an extra data input controlled by an extra write enable input has been added to each register. Due to this, when the Lockstep Controller sends the reload command, all the information saved in the backup memory can be reloaded simultaneously in the registers

without any latency.



(a) Original register. (b) Modified register.

Figure 4.24: Original and adapted registers.

By reason of their critical importance, a fault strategy has been adopted to protect memory blocks, which as in the original design, are implemented with BRAMs. Due to the extra clock latency addition and the possible errors caused by double-bit faults in unused bits, the Xilinx's built-in ECC fabric has been discarded. These drawbacks have motivated the implementation of custom ECC generation and checking modules particularly adapted to the designed approach. Hamming code has been selected for the implementation of these modules, which is a widely utilized, robust and simple code. Considering that the data word width of the designed processor is 8-bit, the ECC code needs at least 5-bit to detect and correct a single error and detect a double error. In the case of the program memory, the instruction word width is 13 bits, so the ECC code needs at least 6 bits. The 8-level stack memory of the processor has been also adapted by using the same ECC strategy. Different ECC checking modules are used to detect and auto-correct single errors in the data, the 8-level stack and the program memories during reading operation, but they don't correct errors in the memory array. They only present corrected data in the output. Nevertheless, the errors in data and stack memories should be overwritten with the next data store during the normal operation. In addition, a double error detection output is available, which can be used to indicate uncorrectable errors.

In the case of memory elements sensitive to accumulative errors, like data memory positions that are not overwritten during the program execution or program memories, additional strategies have to be adopted. In the case of the program memory, the proposed *Data Content Scrubbing Approach* can be utilized to perform periodic scrubbings. The most adequate scrubbing frequency is application and program dependant. An alternative to harden non overwritten data memory positions is to modify the original program by adding extra software instructions in order to perform a overwriting operation of these positions.

On the other hand, the ECC generation modules are used only by the data and 8-level stack memories. As long as the program memory data has to be initialized

with the correct instructions during the synthesis and implementation processes, a ECC generation module would not be useful in the case of this memory. As a result, a visual basic application has been developed to generate the program memory HDL modules. This application uses the .HEX file generated by the assembler with the source code to create the program memory VHDL file with the ECC encoded data.

The implementation of the backup memory for the processor context saving demands a different method, because a parallel load of the data of all registers is necessary to achieve a low latency context saving. Taking into account that the Xilinx BRAM structure can only write to a single memory address at the same time, the data has been saved using a register structure. Considering this fact, the ECC alternative has been discarded for the hardening of the backup memory, because it will dramatically increases the overhead. So, the strategy adopted has been the DMR implementation. In this case, this is adequate because the probability of two consecutive SEUs is usually very low [174]. So, if an SEU affects a register it is extremely rare that another SEU will affect the backup memory simultaneously. In the same way, if an SEU affects the backup memory it is not probable that one of the processors will be affected by another SEU at the same time, and the wrong data will be overwritten in the next context saving. Nevertheless, if this rare situation occurs it will be detected by the comparators of the backup memory.

Thanks to the proposed architecture, a lockstep approach which performs a fast checkpointing and rollback is obtained. Due to this fact, the saving process can be regularly performed every 4 clock cycles without causing any extra latency, nor affecting the processor's performance. In the same way, a context loading process can be achieved only with 2 extra clock cycles. The main drawback of this approach is that the logic added to the design increases the hardware overhead. For this reason, it also reduces the maximum achievable operating frequency. Finally, it has to be remarked that this design can be implemented in FPGA devices from different vendors and the main ideas of this approach can be adapted to designs based on different soft-core processors.

4.6.2 Bitstream Based Low Overhead Lockstep Approach

The *Bitstream Based Low Overhead Lockstep* approach is focused on performing the rollback and checkpointing processes by making use of the bitstream in order to limit the resource overhead. Figure 4.25 represents the simplified scheme of this approach. As it can be observed, in this minimalistic approach the FPGA fabric contains two soft-core processors connected to a comparator. This scheme also

includes the Processing System, which is responsible for performing the checkpointing and rollback processes. These operations are performed via PCAP interface and controlling both, `STARTUPEE2` and `CAPTUREE2` primitives. In order to stop the clock when it is necessary, the Processing System also controls the design's clock signal by using the `CLOCKING WIZARD` IP. With the aim to protect the content of registers of the rest of the design when utilizing the `STARTUPEE2` primitive a reconfigurable region has been created. While both utilized soft-core processors are contained in the reconfigurable region, the rest of the design remains in the static region. Thanks to the use of a single reconfigurable region with the `RESET_AFTER_RECONFIG=TRUE` property, the generated partial bitstream contains by default the instructions necessary to protect (closed padlock) the static region and unprotect (open padlock) the reconfigurable region.

Although both processors can also be implemented in two different reconfigurable regions, this strategy does not provide significant benefits. Since in DMR schemes it is not possible to determine which is the faulty module, it is not necessary to independently repair nor relocate each processor in the presence of permanent errors. However, in some cases, it could be necessary depending on design requirements. For instance, implementing two independent reconfigurable modules provide more design flexibility when relocating them in case of permanent errors. In this case, the second reconfigurable region has to be unprotected by using the proposed design flow to protect/unprotect partial regions (introduced in Section 4.3).

During the normal operation (absence of errors) the Processing Systems performs a checkpointing by reading the partial bitstream of each soft-core processor copy. In this way, the data content from BRAM memories is extracted utilizing the BBBA and saved in the external memory (DDR). In a similar way, the content of registers can be obtained using the `CAPTUREE2` primitive and processing the read partial bitstream based on the information of the *.ll* file.

Two strategies can be adopted to perform the storage of the context in the external memory. As Figure 4.26(a) shows, the first one consists in reading the context of a single processor. After, the read data has to be processed in order to create a proper partial bitstream for the second processor. Thus, both partial bitstreams with the context of the processors are stored and ready to be used in future rollbacks. Bearing in mind that the bitstream readback's time demand increases with the bitstream size, this alternative results faster. Nevertheless, this approach also requires a high processing effort to create the second partial bitstream. This processing effort depends on the registers quantity and their floorplanning. For instance, if both processors have similar floorplanning, which could be obtained by utilizing placement constraints, the processing effort

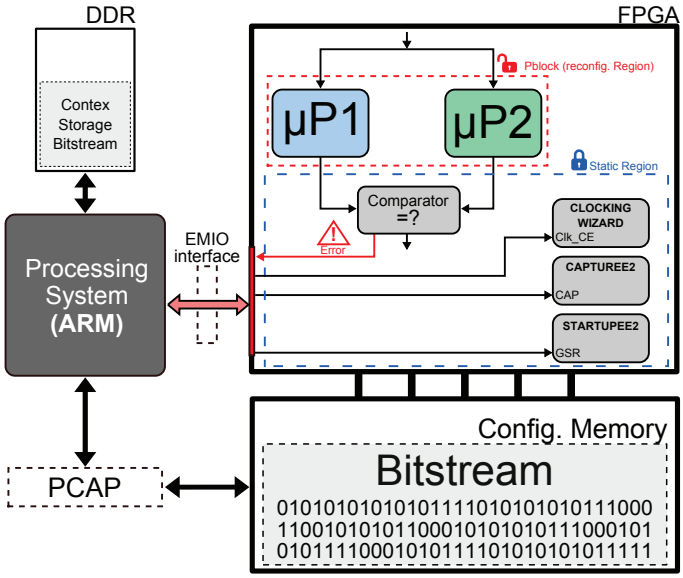
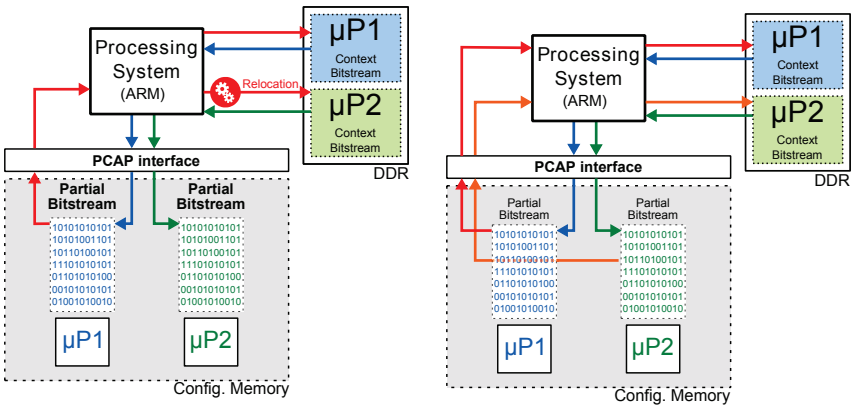


Figure 4.25: Simplified block diagram of the *Bitstream Based Low Overhead Lockstep* approach.

can be lower. The second alternative, as depicted in Figure 4.26(b), is to read partial bitstreams with the context of both processors and store them. Since this approach needs to read double information, the readback takes more time. However, when dealing with small processors like the PICDiY the time increase could be negligible. On the other hand, this strategy does not require to process the bitstream making the storing process faster. Hence, in this case, the second strategy has been adopted.

Storing the context in an external DDR provides higher reliability than storing it the FPGA fabric, since as [328] states the probability of having a radiation-induced error in data stored in the DDR is significantly lower. Besides, it is also possible to duplicate data stored in the DDR in order to increase the reliability, since commonly this software redundancy implies minimal costs in terms of data.

Nevertheless, storing data in a DDR memory demands a significant processing effort. For this reason, to use the configuration memory to store the content of registers is an interesting alternative way to avoid the need of any processing. After running the `GCAPTURE` command, the content of flip-flops is stored in their respective `INIT0/INIT1` positions. Hence, the information is automatically stored in the configuration memory. This implies that after performing the checkpoint-



(a) Context saving of a single processor. (b) Context saving of both processors.

Figure 4.26: Context saving strategies.

ing the context of registers is stored in two way: in the reconfiguration memory and in hardware resources. Bearing in mind this fact, the case of simultaneous errors in `INIT0/INIT1` positions of the bitstream and the flip-flop resources is an unlikely scenario. Hence, considering that it is not necessary to process the bitstream to extract the content of flip-flops and the adequate robustness provided, this strategy has been adopted instead of using the external DDR to store the content of registers.

After detecting a failure due to discrepancy in the outputs of both processors, the comparator asserts an error signal, which is connected with the Processing System via EMIO interface. This signal triggers the rollback recovery process. This process is performed by downloading the data memory content saved in the DDR memory (by using the BBBA) and triggering the `GRESTORE` command with the `STARTUPEE2` primitive to restore the saved content of registers.

This approach allows to perform both, rollback and checkpointing with a minimal impact in terms of resource overhead. Its main handicap is a relatively high time demand, which comes due to the utilization of the BBBA. It also requires the utilization of the Processing System and an external memory to control the different processes and store context, respectively.

4.6.3 Bitstream Based Autonomous Lockstep Approach

In order to circumvent the main drawbacks of both proposed lockstep approaches a third alternative, depicted in Figure 4.27, which in a way is a halfway solution between them has been developed. Despite the low overhead of the *Bitstream Based Low Overhead Lockstep*, it implies the need of external elements (Processing System and a context saving memory) and it demands much time. Due to this, some ideas from the *HW Fast Lockstep* have been added to the *Bitstream Based Autonomous Lockstep Approach*, obtaining a trade-off solution.

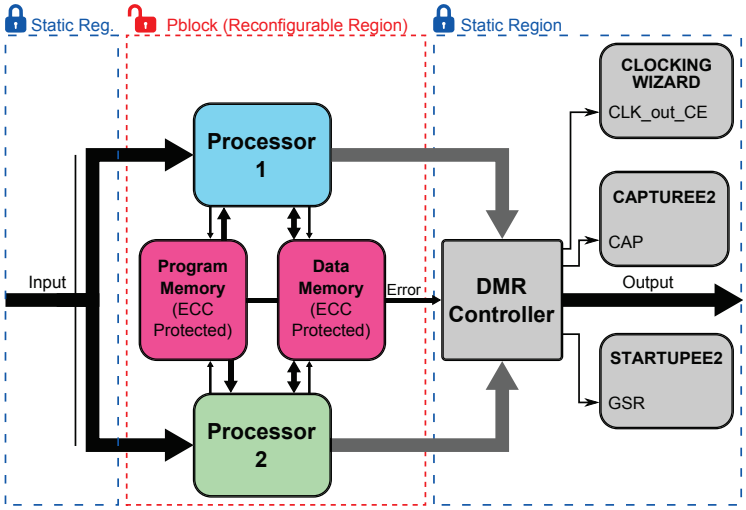


Figure 4.27: Simplified block diagram of the *Bitstream Based Autonomous Lockstep Approach*.

Like in the case of the *HW Fast Lockstep* approach, both processors can be located in separate reconfigurable regions. Also in this case, due to the lack of benefits and the complexity of this option, both soft-core processors have been implemented in the same reconfigurable region. Thanks to that, the generated partial bitstream protects the static area and unprotects the reconfigurable region by default, without requiring bitstream changes.

Bearing in mind that managing BRAM content with the bitstream requires the utilization of the Processing System and an external memory, as proposed in the the *HW Fast Lockstep*, a shared ECC hardened memory strategy has been implemented to protect both, data and program memories.

The most relevant module of this approach is the Lockstep Controller block, which

detects discrepancies between both processors' outputs and controls checkpointing and rollback processes. This module is based on an FSM structure which is shown in Figure 4.28. The first four states are mainly utilized to run synchronized with four states FSM of the PICDiY's. The second state is also used to perform the context saving by triggering the CAP port of the `CAPTUREE2` primitive. In the fourth state, the outputs of both processors are compared. In case of discrepancy, whether the error is unrecoverable or not has to be determined. This is done by utilizing a register to control if the error has been produced also in the previous FSM cycle. In case of a non permanent error the recovering process starts by stepping into the fifth state, which stops clocks of processors and triggers the `GSR` signal from the `STARTUPEE2` primitive. Bearing in mind that the `GSR` is asynchronous and it spreads across the device relatively slow a sixth state is used as a waiting state to ensure that all registers are properly set. The sixth state follows to the second state to synchronize with the four-states FMS of PICDiY processors. In the case of defining the error as unrecoverable the FSM gets stuck in the seventh state. An unrecoverable error may be due to two or more faulty bits in memories (ECC fault) or due to permanent errors. In the case of ECC fault, reseting both processors and the rest of the system can be a feasible solution. On the other hand, a permanent error scenario will require a relocation of both processors in a different fabric logic region.

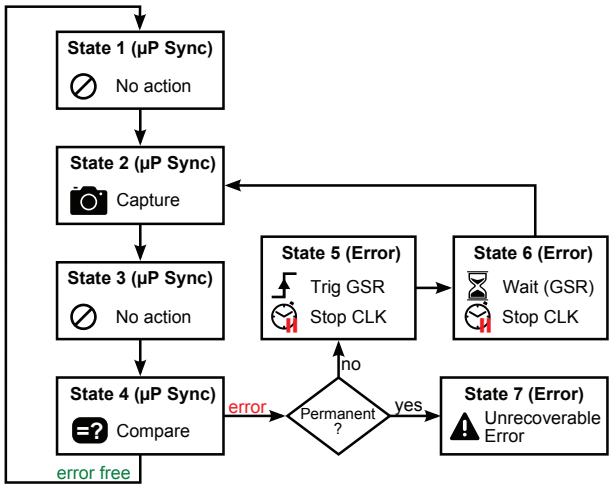


Figure 4.28: State diagram of the Lockstep Controller block.

In addition to the relatively fast lockstep and rollback processes, a relevant benefit of this approach is that, since no bitstream data read, write nor relocation is

required, the registers' context recovering does not need any processing system nor external memory. Hence, this approach can be considered as autonomous in terms of the need of external elements. Nevertheless, due to the ECC implementation to harden the data memory a resource overhead is introduced. Since this hardware overhead also implies the increase of data paths, the maximum achievable operating frequency of the design decreases.

4.6.4 Lockstep Approaches Overview

Three different lockstep approaches have been proposed to deal with the issues that arise with the utilization of the checkpointing and rollback techniques following distinct strategies. Table 4.6 summarizes their most relevant characteristics.

Table 4.6: Lockstep approaches overview.

Characteristic	HW Fast	Low Overhead	Autonomous
HW overhead	high	low	moderate
HW modifications	high	no	moderate
Time demand	low	high	low
Performance penalty	high	low	moderate
Platform independence	yes	relative	relative
PS requirement	no	yes	no

The *HW Fast Lockstep* approach achieves a high checkpoint frequency and a fast rollback recovery with platform independence. It also avoids the need of any bitstream processing. However, since it demands considerable hardware adaptations to add the different elements, it increases the hardware overhead and reduces the maximum achievable operating frequency.

The *Bitstream Based Low Overhead Lockstep* approach utilizes the BBBA developed in this work as a basis to perform the rollback and the checkpointing and does not introduce any resource overhead. Thanks to this fact, it does not affect to the maximum operating frequency of the design. Another relevant advantage of this approach is that it does not require any adaptation of the target processor. Nevertheless, the prize to pay when utilizing this approach is a low checkpointing frequency and a relatively slow rollback recovery due to the time demand of bitstream readback and writing processes. The BBBA also requires to use the Processing System and an external memory to store the context. Despite that in Zynq based designs this could be an negligible issue due to its dual ARM processor, the adaptation of this approach depends on technical aspects

of the hardware platform utilized. Hence, it could be considered as a relatively platform-independent solution.

The *Bitstream Based Autonomous Lockstep* approach combines several aspects of the two previous methods, implying a trade-off solution. In this way, it utilizes the *Approach to Manage Data of Registers with the Bitstream* to save and load the context of registers and protects data and program memories following an ECC strategy. Due to this it also avoids the need of processing the bitstream. In this way, this approach provides an autonomous fast checkpoint and recovery processes with a moderate hardware overhead. The drawbacks of this method are related to the implementation of the ECC logic, which increases the hardware overhead and reduces the maximum frequency. Furthermore, due to the usage of different Xilinx primitives is a platform dependant technique. However, it could be possible to adapt the main ideas to other vendor's technology.

All the proposed approaches provide different solutions, depending on the requirements of the application to be designed. When the availability is a crucial aspect the *HW Fast Lockstep* approach could be more advisable. On the other hand, in applications that don't demand real time response, the *Bitstream Based Low Overhead Lockstep* approach could be a good candidate in order to save resources and power. The *Bitstream Based Autonomous Lockstep* approach is a halfway solution that provides fast context saving and recovering operations but requires small adaptations on designs to be hardened.

4.7 Proposed Synchronization Approaches for Repaired Soft-Core Processors in Hardware Redundancy Based Schemes

This section addresses the scarcity of investigations around synchronization in hardware redundancy based systems by proposing, implementing and evaluating five different synchronization methodologies. Since the triple redundancy method is one of the most established redundancy levels, the approaches in this section will be based on TMR (Triple Modular Redundancy) schemes. Nevertheless, the majority of concepts presented can be applied to other redundancy levels. The five approaches span a broad spectrum of possible alternatives from minimal hardware overhead to completely hardware-based synchronization. This allows balancing the trade-off between implementation cost and synchronization speed, depending on the requirements of the target application at hand. The performance of the proposed techniques is verified and compared to each other on the

PICDiY processor. However, all the approaches are of general nature and can easily be migrated to other processor architectures. The presented methods are furthermore not restricted to a set-up implementing TMR and DPR (Dynamic Partial Reconfiguration). They are applicable to any TMR protected processor system to recuperate a processor element, which was forced out of sync by an SEE.

When implementing a combination of TMR and DPR for the realization of fault tolerant systems, a pure reconfiguration of a faulty module is not sufficient given that the reconfigured module comprises of an internal state. This synchronization is especially critical for processors, because their state is composed of a number of different registers. For the PICDiY processor the following elements need to be synchronized: the program counter, the stack-pointer and the stack content, the special function registers (FSR, STATUS, INTCON, PCL, PCLATH and INTCON) and the data memory. These elements will be referred to as *synchronization objects* in the remainder of this work.

In general, finding an adequate synchronization strategy for a given application implies balancing a trade-off. On the one hand, adding specialized hardware for the processor synchronization will enable a very fast synchronization process. On the other hand, implementing the synchronization with little extra, or none, hardware combined with software will result in less implementation overhead and a lower impact on the critical path of the design.

The structure of the synchronization method impacts the duration of two substeps of the whole recovery process. Firstly, the time required to copy the correct values of the different synchronization elements to a recently reconfigured processor instance in the coarse-grain TMR setup. This time will be called *copy-time*. The second aspect of the synchronization speed is the time from the detection of an error by the voter until the point in time where the system is ready to start the synchronization process. This second time is named *wait-for-sync*. Some approaches can not begin directly with the synchronization, but they first need to finish ongoing calculations before CPU time can be spent for updating a reconfigured processor instance in the system. The time from the detection of an error to the re-synchronization has implications on the overall system robustness, because in this time period the TMR system operates only with two functional instances, making it vulnerable to consecutive SEUs.

The whole SEU recovery process is illustrated in Figure 4.29, where the time requirement is defined by the sum of four components: the time needed to detect the error, the *wait-for-sync* time, the time consumed for repairing the SEU by partial reconfiguration and the *copy-time*. Whereas the time for partial reconfiguration is proportional to the size of the reconfigured partition and the reconfiguration

Table 4.7: General synchronization objects and accessibility for PICDiY, PicoBlaze and MicroBlaze processors.

Synchronization object	PICDiY	PicoBlaze	MicroBlaze
CPU-flags	read/write	no access	read/write
Stack-pointer	no access	no access	read/write
Stack-content	no access	no access	read/write
Program counter	read/write	no access	read/write
CPU registers	read/write	read/write	read/write
Data memory	read/write	read/write	read/write

speed, the *time to detection* is not affected by the synchronization approach, and is only application dependent and hence is considered beyond the scope of this work.

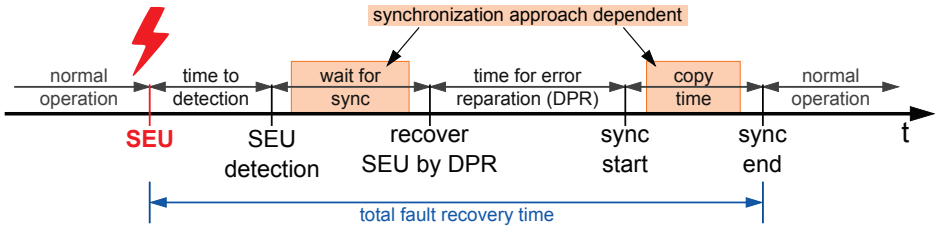


Figure 4.29: SEU recovery process and impact of synchronization times.

In the following, different synchronization approaches are presented for the example of the PICDiY processor. The synchronization objects of this specific architecture are summarized in Table 4.7. This table also contains the synchronization objects of the PicoBlaze, a close alternative and the MicroBlaze processor, a more powerful and complex processor architecture. Despite of the simpler architecture of the PICDiY and the PicoBlaze, they are more demanding in terms of synchronization. For the MicroBlaze all synchronization objects are accessible via software. However, the PICDiY and the PicoBlaze have an inherent need for additional hardware when a complete synchronization is desired because some elements are neither readable nor writable by software.

It needs to be noticed that this work does not consider the program memory as a synchronization object. This is because when a program memory module is repaired by utilizing DPR its content is also initialized by the bitstream. Nevertheless, among the proposed approaches the BBBA based one is the only that could synchronize the program memory since it is implemented as a ROM.

The only way to address the synchronization of this module by the other four approaches would be to implement it as a writable memory block.

4.7.1 Cyclic Resets Based Synchronization Approach

The first synchronization approach proposed is based on applying a cyclic reset to the three processors as indicated in Figure 4.30. This approach will be referred as *Reset Sync*.

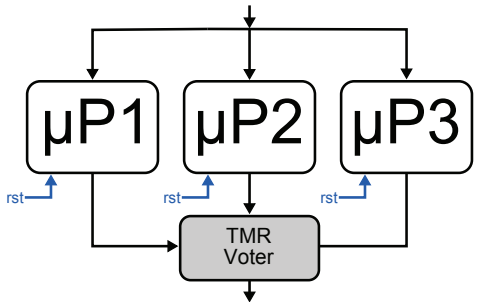


Figure 4.30: Simplified diagram of the *Reset Sync* approach.

This reset based strategy allows bringing all TMR instances of the PICDiY back to a known state and restarting the program execution from the scratch with all three PICDiY in parallel. No changes in the original PICDiY design's implementation are required. In this approach the BRAM memory on Xilinx devices is not cleared by a reset. The software on the PICDiY processors consequently needs to initialize all registers or scratchpad locations before the first usage.

There are different possibilities for triggering the synchronization reset:

- The PICDiY sets a flag in a shared memory upon termination of their calculations. When at least two instances have set the flag, a reset may be issued.
- A timer is implemented programming it to the worst-case runtime of the given software and is started together with the PICDiYs. The expiring timer will trigger a reset.
- The instruction bus addresses of the PICDiYs are supervised to detect the end of the program.

- Very simple applications, where it is admissible to lose one packet, may trigger the reset directly upon detection of an SEU.

For the first three possible trigger methods, the worst-case of the synchronization time *wait-for-sync* is one complete algorithm runtime. On the other hand, the *copy-time* does not exist because no values are copied.

Bearing in mind that this proposed approach does not synchronize the device in its entirety, some aspects must be considered for the software running on the processors. This fact implies that the are software restrictions when utilizing this technique. On this purpose, the two different software flows shown in Figure 4.31 are hereafter presented in order to enable its utilization.

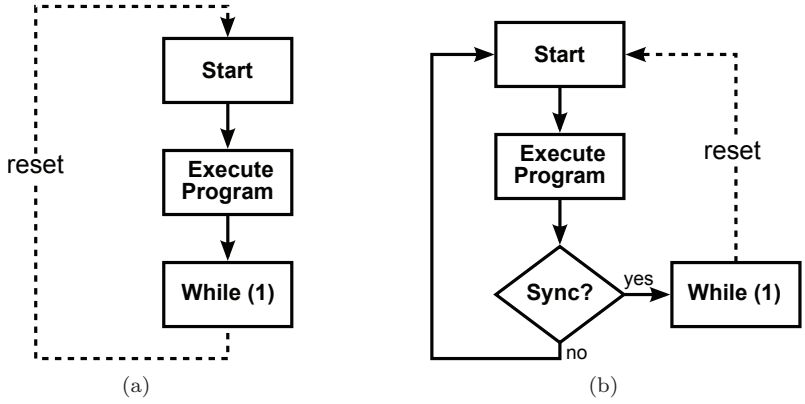


Figure 4.31: Proposed software flows when using *Reset Sync*

The flow chart in Figure 4.31(a) schedules a reconfiguration after each iteration of the algorithm implemented on the processor. After the software finishes its execution a waiting state is entered (e.g. an endless loop). After it has been detected that at least two processors reached this waiting state or after the expiration of a synchronization timer a resynchronization is triggered. This is executed independently of whether a fault occurred or not, in order to leave the waiting state and prepare the following iteration of the algorithm.

A software flow triggering reconfigurations only if necessary is presented in Figure 4.31(b). After finishing one calculation iteration the processors read the TMR-protected information of the system voter if a resynchronization is required. If this is the case, then the software flow proceeds to a waiting state as in Figure 4.31(a) and waits for synchronization by reset. If, on the other hand, no synchronization is required, the processors may directly continue with the next calculation.

4.7.2 Memory and Address Force Based Synchronization Approach

The *Reset Sync* approach implies a high number of restrictions to the software running on the hardware. One important restriction is that it is not possible to leave permanent data in the processor registers or in its internal memory. Although this memory is not affected by the resets, it can not be assumed to be error-free, because it is not synchronized by any means.

This drawback is addressed in this synchronization approach based on memory and address jump forces, which would be referred as *Force Sync*. (Figure 4.32), which builds upon and enhances a synchronization idea presented in [59] based on a shared memory. Adding a TMR-protected shared memory to the system allows a synchronization of the processors by concurrent writing to this memory, followed by a concurrent reading of the data. The actual synchronization of the data is executed when all PICDiYs write concurrently to the TMR memory, where the voters on the input of the TMR memory are able to mask an incorrect input from one processor. The proposed method utilizes a synchronization memory of only one byte (i.e. the datawidth of the PICDiY) yielding in a minimal hardware overhead.

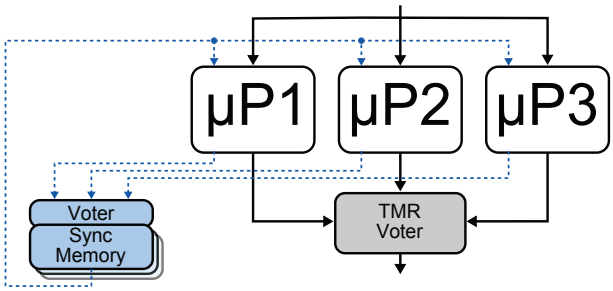


Figure 4.32: Simplified diagram of the *Force Sync* approach.

The synchronization process is triggered by externally forcing a `GOTO` instruction to all three PICDiY instances. The destination of this `GOTO` is the synchronization software sequence, which is executed by all processors simultaneously. At the same time the program counters of the PICDiYs are synchronized utilizing this simple method. In the synchronization sequence all PICDiY memory locations requiring synchronization are first written to and then read from the synchronization memory. It needs to be ensured that the synchronization is only triggered when the processors are not serving any interrupt. The software restriction ac-

companying this approach demands to utilize similar program flows as proposed for *Reset Sync* (in Section 4.7.1). However, in the case of the *Force Sync*, the reset has to be replaced by the execution of the `GOTO` instruction.

The rationale behind forcing a `GOTO` to execute the synchronization code as opposed to using an Interrupt Service Routine (ISR) is that the *Force Sync* approach follows a "black box" approach, which does not require changes in the implementation of the used processor. Using an ISR for synchronization would require a synchronization of the processor stack, which would imply changes to be made to the HDL code of the processor.

The synchronization time *wait-for-sync* in the *Force Sync* approach has a worst-case of the complete algorithm runtime, whereas the *copy-time* equals the best case of 16 clock cycles per each byte of synchronized register or data memory address. Each writing and reading requires two instructions (`MOVWF` and `MOVF`), which each takes four cycles.

4.7.3 Synchronization Approach Based on Using an Interruption and a Synchronization Memory

This third synchronization approach, that will be referred as *Interrupt Sync*, uses a synchronization memory and the synchronization software residing in the ISR.

Under this approach the processors do not need to wait for the current calculation to finish, but it is possible to almost directly react on a detected SEU by assigning the synchronization interrupt. Along with this improvement comes the need for modifications in the PICDiY implementation. When using an ISR for synchronization, it is essential to synchronize the processor stack in order to synchronize the return addresses of all processors, because before entering the ISR the program counter of the reconfigured processor will have a different value than the other two.

In Figure 4.33 the Sync Control block represents the logic block added to the overall system for the synchronization of the stack-pointer and the stack content. As these elements are not software accessible, it is not possible to implement the *Interrupt Sync* in the same "black box" fashion as *Force Sync* and *Reset Sync* approaches. The Sync Control module is a FSM based structure that, once the synchronization has been triggered, synchronizes the stack content. This process is carried out by reading and voting the stack position of each PICDiY and writing the voted data in the three processor instances. This strategy requires to adapt the PICDiY to give access of the stack memory ports to the Sync Control block.

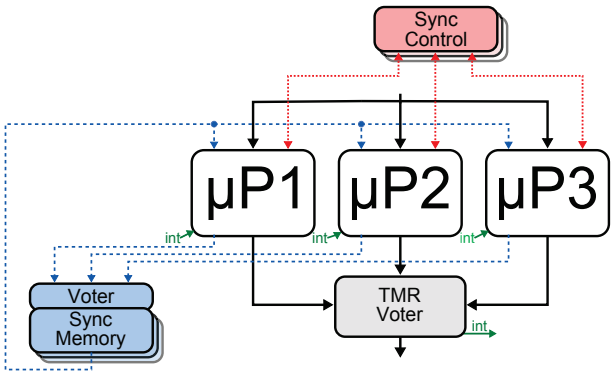


Figure 4.33: Simplified diagram of the *Interrupt Sync* approach.

The advantages of this synchronization approach become visible when focusing on the *wait-for-sync* synchronization time. This is reduced to be only a few cycles long, namely the time needed for triggering the interrupt. The *copy-time* remains almost the same as in *Force Sync*, since the software for the copying is identical and the Sync Control block only takes 16 clock cycles to synchronize the stack content. Nevertheless, due to the demanded adaptations, this approach increases both, the resource overhead and designing costs. In the case of the PICDiY processor the design costs are limited. However, when using other processors these costs depend on the complexity of the soft-core design and the designer skills. For instance, adapting the PicoBlaze requires a substantial designing effort.

4.7.4 Complete Hardware-Based Synchronization Approach

The fourth proposed approach is a completely hardware based synchronization method that follows a similar strategy to the *HW Fast Lockstep*. This approach with the largest hardware overhead is illustrated in Figure 4.34 and it will be referred as *Hw Sync*. In this case, the Sync Control module is expanded to control the synchronization of the registers and the data memory as well without requiring any specific synchronization software.

This approach requires to increase the FSM structure of the Sync Control module in order to be able to synchronize the stack memory, the registers and the data memory. It also requires notable adaptations in the processor design to give access to the Sync Control module to the data and control ports of all memory elements (registers and memories). With these modifications, all registers can be synchronized at once. On the other hand, synchronizing both, stack and data

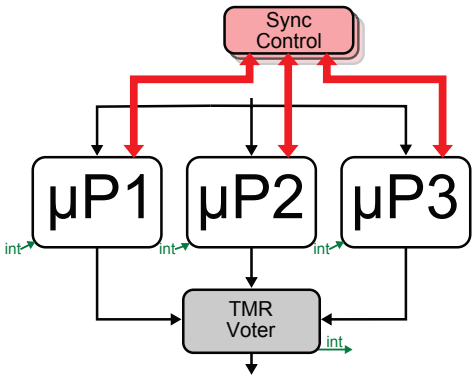


Figure 4.34: Simplified diagram of the *Hw Sync* approach.

memories, requires to access all data positions during the reading and writing processes.

When the hardware based synchronization is executed, simultaneous access to registers or memories must be avoided. As almost all PICDiY instructions use registers in their operations, this work proposes to stall the processor while the hardware based synchronization takes place. An easy implementation of this stalling can be achieved by externally forcing all PICDiY instruction addresses to the fourth position of the program memory. This address is the interrupt entrance point and thus always implies an unconditional jump to the ISR. Another alternative is to use a dedicated buffered clock output of the `Clocking Wizard`.

A tailored optimization of the stall time can be achieved in a variety of ways. If e.g. the usage of the memory in the first part of the ISR is forbidden, the stalling can be as short as the register synchronization time. The stalling can also be shortened, when only a subset of data memory locations require synchronization.

The *Hw Sync* approach has the same short *wait-for-sync* time as the *Interrupt Sync*, but it has a significantly reduced synchronization time for the registers and data content (*copy-time*). Whereas software based synchronization in the PICDiY takes 16 clock cycles (two `MOV` instructions for reading and other two for writing) per memory cell, hardware based synchronization needs only two cycles (single cycle for reading and another for storing). In this case, considering a 128 address depth data program and the 7 special registers to be synchronized this approach takes 272 clock cycles for this task. An additional benefit of hardware synchronization is that the copying of registers and data memory can be executed in parallel. The most significant drawbacks of this approach are the impact in the

hardware overhead and the performance penalty introduced by the adaptation. Another relevant aspect to be considered is the high designing effort required to modify the design.

4.7.5 Complete Bitstream-Based Synchronization

The fifth and last proposed synchronization approach, utilizes the *Bitstream Based BRAM Approach* (BBBA) and the *Approach to Manage Data of Registers with the Bitstream* presented in this work to synchronize BRAM based memories and registers data respectively. Due to this approach, that will be referred as *Bitstream Sync*, data memory and registers are synchronized with no software nor hardware restrictions and without increasing the resource overhead.

As it can be observed in Figure 4.35, this approach shares several aspects with the previously proposed *Bitstream Based Autonomous Lockstep Approach*. Nevertheless, in this case the synchronization is done by extracting the context from the partial bitstream of one the two correct processors and copying it to the partial bitstream of the repaired module. Considering that the `CAPTUREE2` primitive only captures content of flip-flops from non protected areas, it is mandatory to edit created partial bitstream following the proposed method based on the `RESET_AFTER_RECONFIG=TRUE` property to unprotect (open padlock) the three reconfigurable modules (*PblockA*, *PblockB*, and *PblockC*) and protect the static area (closed padlock).

Figure 4.36 explains the flow of the synchronization routine. Initially as Figure 4.36(a) shows, each of the three replicas is connected to the *TMR Controller* module, which detects discrepancies and identifies the faulty processor. After detecting an error (Figure 4.36(b)) this information is sent to the *Processing System* which triggers the partial reconfiguration of the faulty module (Figure 4.36(c)). Once the synchronization routine has started, as shown in Figure 4.36(d), the first step is to stop the clocks of processor (which in this case is done by enabling the `Clk_Out_CE` from the `Clocking Wizard`) IP to assure a complete stall. Next, the context of registers is captured by triggering the `CAP` port from the `CAPTUREE2` primitive. Then the context of the repaired processor has to be updated. This process is summarized in Figure 4.36(e). In order to obtain the captured context and data from BRAMs a bitstream readback of the partial bitstream of one of the correct processors is performed. This information is processed to relocate the context data creating a new partial bitstream for the repaired processor. Once downloaded the created bitstream, in the final step depicted in Figure 4.36(f), the context of all processors is restored by triggering the `GRS` signal of the `STARTUPEE2` primitive. After waiting for the propagation of the `GSR` signal the clock can be

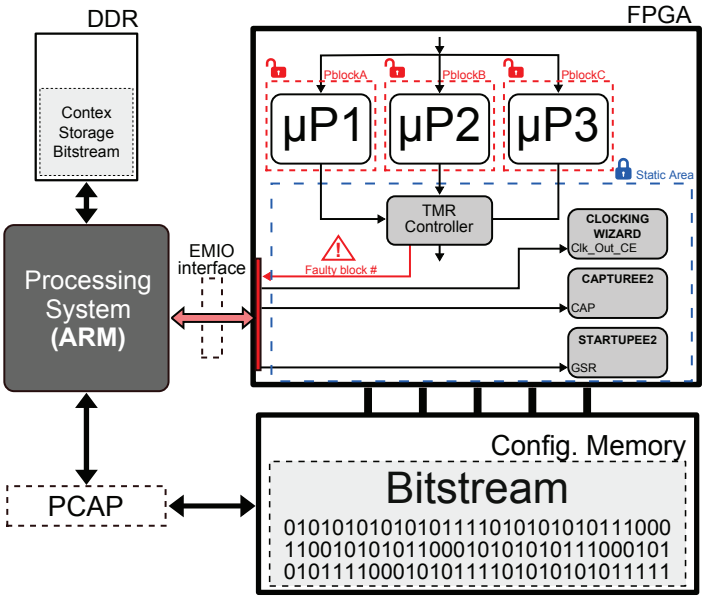
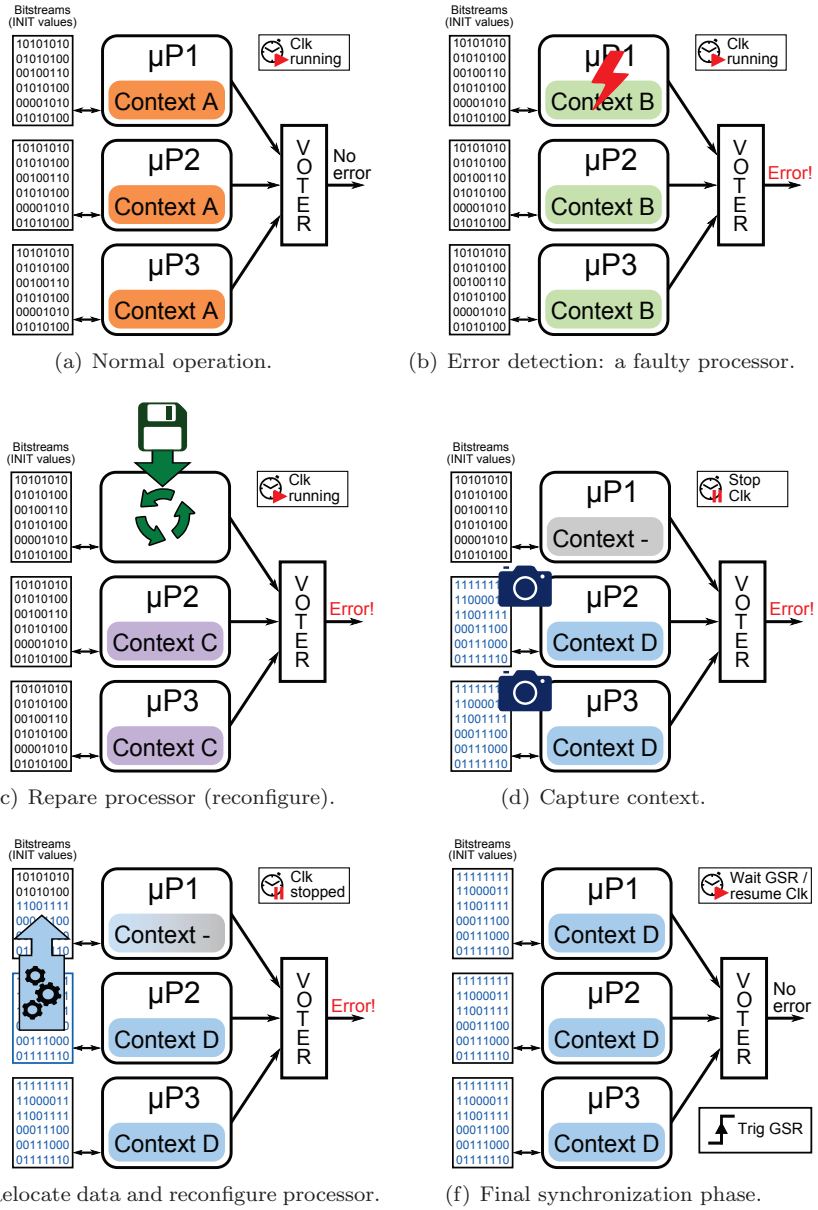


Figure 4.35: Simplified diagram of the *Bitstream Sync* approach.

activated to resume to normal operation.

Considering that by default each processor commonly presents distinct implementations due to the default resource placement freedom of the implementation tool this implies a complex process of copying and relocating the bitstream data. This drawback gains importance bearing in mind the size of soft-core processors which gather a significant quantity of memory elements to be synchronized. Hence, it requires a big processing effort, especially taking into account that each design implementation requires to the Processing System to specifically adapt the data relocation processes. For this reason, the BRAMs of each soft-core processor have located in the same position of each BRAM column by using placement constraints. In a similar way, the registers of each of the three replicas have been implemented following the proposed *Location Constraints Flow*. Thanks to both strategies three alike implementations in each reconfigurable region can be obtained. This strategy reduces complexity of the bitstream manipulation process because, in this case, each replica of the tripled memory resources presents similar data structure and locations within the bitstream.

This approach provides the remarkable benefits of not requiring hardware adaptations. Hence, the utilization of the *Bitstream Sync* does not increase the hardware



(a) Normal operation.

(b) Error detection: a faulty processor.

(c) Repare processor (reconfigure).

(d) Capture context.

(e) Relocate data and reconfigure processor.

(f) Final synchronization phase.

Figure 4.36: Synchronization routine of the *Bitstream Sync* approach.

overhead. Nevertheless, the price to pay is a high time demand and the necessity of the Processing System and an external memory. In addition, the utilization of the `RESET_AFTER_RECONFIG=TRUE` property increases the size of generated partial bitstreams, which results in higher reconfiguration times and requires more memory space to store the bitstreams.

Despite that in the case of this work the processing and bitstream manipulation tasks are carried out by the Zynq's ARM processor, in devices without hard-processors this bitstream manipulation could be developed utilizing an additional soft-core processor or and FSM based module. However, these kind of alternatives could limit the advantages in terms of hardware overhead described initially.

4.7.6 Synchronization Approaches Overview

An overview of all the five proposed approaches is given in Table 4.8. It summarizes the synchronization strategy for all synchronization elements of the PICDiY processor. The entry *SW* represents the synchronization via synchronization-memory, *HW* represents a modification to the processor, Bitstream represents the synchronization via *Bitstream* and a dash indicates that the element is not synchronized. The synchronization methods for the program counter manifest the biggest differences for the first four approaches. Whereas *Reset Sync* relies on resetting this register, *Force Sync* works by forcing a `GOTO` instruction to update this value. On the contrary, *Interrupt Sync* and *Hw Sync* only synchronize the stack while being in the ISR and the program counter is updated upon returning from the interrupt. On the other hand, *Bitstream Sync* synchronizes all memory elements by utilizing the bitstream. Thanks to this, neither hardware modifications nor the addition of new software to the soft-core processor are required.

Table 4.8: Synchronization methods overview.

Sync. object	Reset	Force	Interrupt	HW	Bitstream
ALU-flags	-	-	HW	HW	Bitstream
Stack-pointer	-	-	HW	HW	Bitstream
Stack-content	-	-	HW	HW	Bitstream
CPU registers	-	SW	SW	HW	Bitstream
Data memory	-	SW	SW	HW	Bitstream
Program counter	reset	forced `GOTO`	on ISR return	on ISR return	Bitstream

Chapter 5

Validation of Fault Tolerance Approaches

After introducing the different contributions focused on improving existing fault tolerance techniques for SRAM-based FPGAs, this chapter introduces a detailed analysis of the proposed approaches. Aspects such as, impact on resource overhead, effects in terms of performance penalty, availability and functionality are evaluated utilizing a hardware platform. The results extracted from these experiments provide relevant information for fault-tolerance designers in order to make trade-off decisions and to choose the best solution to fit with design specifications.

In Section 5.1 the basis of the experimental setup is introduced, by means of describing the hardware platform and the employed test application. Section 5.2 validates the designed soft-core processor and compares it with two representative soft-core processor IPs. Section 5.3 evaluates the proposed BBBA approach to manage BRAM data through bitstream and compares it with a basic FSM based implementation. In next Section, Section 5.4, the proposed *Approach to Manage Data of Registers with the Bitstream* is validated by using basic implementations which include 8-bit registers in different scenarios. In a next step, the test to refute the *Data Content Scrubbing Approach* is described in Section 5.5. Section 5.6 follows describing the evaluation of the *Approach to Extract Data From Damaged Memories Using the BBBA*. Furthermore, in Section 5.7 the validation of the three proposed lockstep approaches is detailed. Finally, Section 5.8 deals with the validation of the different proposed approaches to synchronize repaired soft-core processors in hardware redundancy based schemes.

5.1 Experimental Setup

All experimental setups of this research work have been implemented in a general purpose development board from AVNET named ZedBoard [329], which is depicted in depicted in Figure 5.1. Zedboard includes a Xilinx Zynq-7000 AP SoC XC7Z020-CLG484 device that contains tightly coupled 7-series programmable logic and dual-core ARM Cortex-A9 centric processing-system. The leading position of Xilinx in the field of all programmable FPGAs and SoCs devices makes this device a remarkable candidate to be used as a validation platform. Due to its advanced technology, Xilinx's products are widely used in several fields, such as, medical devices, industrial control, etc. For instance, NASA has recently utilized Xilinx technology to implement the scope of CubeSat Launch initiative [141].

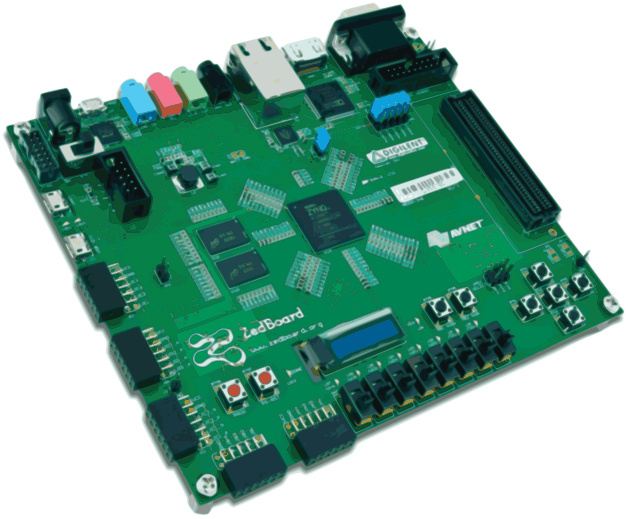


Figure 5.1: ZedBoard Zynq-7000 ARM/FPGA SoC development board.

The software utilized to develop and implement the different FPGA designs has been Vivado Design Suite (16.2). It provides the tools needed to develop complex designs, optimizing the implementation for Xilinx devices, providing IP subsystems reuse and accelerating the design process by enabling to work at a high level of abstraction.

Due to the PIC16 instruction set compatibility of the PICDiY processor, the programming has been done using MPLAB IDE8.76 IDE by Microchip Technology Inc. to obtain the .HEX files with the application programs. After that,

the processors program memory block has been generated using a visual basic application developed for this work combined with previously generated .HEX files to generate the VHDL file of the programmed program memory.

Additional software like PuTTY [330] has also been utilized to perform serial communications between ZedBoard and computer. PuTTY is a free implementation of Telnet and SSH for Windows and Unix platforms.

In order to obtain reliable and comparable results in the different validation tests the methodology shown in Figure 5.2 has been mainly followed. Once the VHDL code of the design has been properly written and checked, the first validation step has been to synthesize and simulate it. This step has been done by using Vivado 16.2 and its simulator. Different configurations have been thoroughly simulated, starting with independently checking of each module or IP that constitute the system and ending with full design simulations. In several cases the debugging of this physical implementation process has been performed with the help of Vivado's Integrated Logic Analyzer (ILA), which permits to visualize internal signals of the FPGA design. After debugging and improving the designs, the next step has been to implement and generate the bitstream to be download into the FPGA. Also Vivado 16.2 suite was employed to carry out these processes.

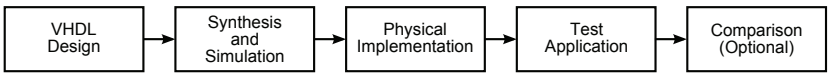


Figure 5.2: Flow chart of the utilized validation strategy.

In a further phase, the designs have been validated by running real case applications on the soft-core processors under test. This have provided the insight into correctness of the results obtained in a real world scenario. Despite that in some cases specific applications have utilized, the mainly used one has been a RS232 serial transmission based application, which hereinafter will be referred to as *TestApp*. This application sends a sequence of ASCII characters stored in the program memory thanks to a finite state machine based program. The communication setup uses 8-bits data word without parity and a single stop bit. The baud rate of 9600 is generated with a special block in the FPGA but outside the processor design, is based on a counter based structure. The signal generated on the baud generation block is connected to the processors interrupt inputs and the transmission process is performed by a specific interrupt routine. Thanks to that, with the correct setting in the baud rate generation block, the transmission can work with different baud rates and with distinct clock frequencies. The basic experimental setup (a stand-alone processor) is presented in Figure 5.3. The FPGA transmits the information generated by the processor through output port

to the computer, where PuTTY is running, via COM serial port. *TestApp* has been considered adequate for test purposes because, in spite of being a relatively simple mechanism, it makes use of distinct elements of processors (registers, data memory, ports, interruption, etc.), its instructions and programming structures. In addition, a transmission application requires high availability to maintain specifications of the protocol which makes it an interesting validation example. On the other hand, the lack of complexity of the application allows to avoid using extended external elements, which has helped to focus on the characteristics of the processor.

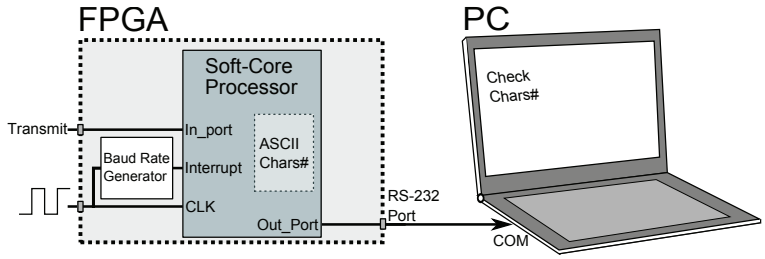


Figure 5.3: *TestApp* experimental setup.

Finally, the majority of proposed designs have been compared with similar existing designs or solutions in order to refute obtained results. Which in each particular case has required to analyse distinct existing solutions in order to choose the most representative alternatives as comparison subjects. With the aim of obtaining comparable results, all designs have been implemented selecting the same synthesis and implementation strategies oriented to optimize the performance: *Flow_PerfOptimized_high (Vivado Synthesis 2016)* for syntheses and *Performance_ExplorePostRoutePhys-Opt(Vivado Implementation 2016)* for implementations.

5.2 Validation of PICDiY Soft-Core Processor

The first step of the design validation has been to simulate all elements of the PICDiY (registers, IDC, memories, etc.). Then, the entire processor design has been checked by simulating all the instructions have been validated in a physical implementation by utilizing the ILA analyser. Finally, the processor has been validated for distinct applications implemented in ZedBoard, including the *TestApp*.

After the validation process the PICDiY has been compared with other 8-bit soft-core processors in order to evaluate it. Two processors have been chosen for this task: a widely used soft-core processor and a PIC based one.

The first selected soft-core processor has been the PicoBlaze, since is a prevailing IP for Xilinx based designs [58, 67, 69, 70]. Being optimized for 7-series devices, it presents a very reduced size (small as 26 slices depending on device family) and high performance (up to 240 MHz). In addition, it is highly integrated for implementing non-time critical state machine and presents a predictable fast interrupt response. All these characteristics make the PicoBlaze an ideal reference for a comparison.

Regarding the second soft-core processor, different IP cores based on the PIC16's architecture are available (CQPIC, PPX16mcu, RISC5X, RISC16F84, MINI-RISC, Synthetic PIC, UMASS, etc.). All of these cores differ slightly in their architectures, but they all implement the main features of a PIC16 microcontroller. The selected PIC16 based soft-core processor has been the PPX16mcu [331]. The main reason to select the PPX16mcu has been that it is the closest alternative to the PICDiY, since PPX16mcu is a minimalist open source version that does not has neither Watchdog nor EEPROM. PPX16mcu is a single cycle and, four times faster, VHDL implementation of the 16F84.

For the comparison tests, the three soft-core processors have been implemented with the same timing constraints (100 MHz) and without including the logic resources necessary to implement the *Testapp*. Additional implementations have been also performed in order to determine the maximum frequency achievable with each soft-core processor. Implementation results (at 100 MHz) are shown in Table 5.1. As it can be observed, PICDiY presents a significantly lower device utilization compared to PPX16mcu, while PicoBlaze is the one with the lowest overhead. As this table shows, those results are closely related with the dynamic power consumption of each soft-core. Due to the limited resource usage of the analysed soft-cores, there is only a slight difference between them. In any case, these results prove that the power consumption of the three soft-cores is moderate. Regarding the maximum achievable frequencies, the tests have shown that PicoBlaze provides the best results (150 MHz). On the other hand, the maximum frequencies for PICDiY and PPX16mcu are similar.

Furthermore, Table 5.2 shows a detailed list of the primitives utilized by each implementation. Apart from the resource overhead information given by this table, it is interesting to note that PICDiY uses distributed memory resources (`RAMS64E` and `RAMS32` primitives) instead of BRAM blocks to implement the data memory module. This is because Vivado changes the initial BRAM based implementation defined by the VHDL design in order to meet the timing constraints. Additional

tests with more moderate timing constraints (e.a. 90 MHz) have shown that Vivado implements the data memory by using a `RAMB18E1` primitive. A similar situation has been observed in case of PPX16mcu. While in a 100 MHz design all memories are implemented with distributed memory resources, a design with lower frequency utilizes a `RAMB18E1` primitive.

Table 5.1: Implementation results summary of the soft-core processors (@100MHz).

Resource	PICDiY	PPX16mcu	PicoBlaze
Slice LUTs	266	478	130
Slice Registers	82	223	114
F7 Muxes	17	2	16
F8 Muxes	8	-	8
Block RAM Tile	0.5	-	0.5
f_{max} (MHz)	107	110	150
Dynamic p. (W)	0.129	0.120	0.123
Static p. (W)	0.125	0.125	0.125

Table 5.2: Primitive utilization of the soft-core processors (@100MHz).

Primitive	PICDiY	PPX16mcu	PicoBlaze
FDRE	80	119	111
FDCE	-	66	3
FDSE	2	-	-
FDPE	-	38	-
CARRY4	8	9	10
LUT1	1	16	1
LUT2	32	53	1
LUT3	11	54	1
LUT4	18	55	1
LUT5	37	100	42
LUT6	131	173	76
MUXF7	17	2	16
MUXF8	8	-	8
RAMB18E1	1	-	1
RAMD32	-	12	24
RAMS32	7	4	8
RAMD64E	-	24	-
RAMS64E	32	-	32

Further aspects must be considered when comparing three soft-core processors. Although the functionality and programming of PPX16mcu is quite similar to PICDiY's, PPX16mcu requires extra instructions to configure ports, since they can work as input or output. In this way, compared to PICDiY's, the program code of PPX16mcu needs six extra instructions to configure both, input and output ports. Besides, while PPX16mcu needs eight clock cycles to execute instructions with conditional jumps, PICDiY needs only 4 clock cycles for all the

instructions, which increases the performance. This aspect is especially relevant in FSM based designs.

On the other hand, there are several differences between PICDiY and PicoBlaze that deserve to be remarked. One of the most relevant is that PicoBlaze is platform dependent and is only suitable for Xilinx's devices, while PICDiY is platform independent and can be implemented in FPGAs from any vendor. Considering that the design of PicoBlaze is highly optimized for Xilinx devices, any attempt to export its architecture results in an increase of resource utilization and a lower maximum frequency. This aspect can be observed by analysing the different efforts made to obtain platform independent soft-cores based on the PicoBlaze [26, 322, 323]. Another interesting aspect is that, as Figure 5.4 shows, commonly the output port of PicoBlaze has to be registered in order to synchronize the reading with the *strobe* signal, causing an increase in resource utilization. Analysing programming aspects, PicoBlaze only uses two clock cycles to execute each instruction and the data loading is done with a single instruction (`LOAD`), while PICDiY needs two (`MOVW + MOVWF`). However, PICDiY offers better characteristics to deal with multiple conditional branches. This is because none of PicoBlaze instructions gives access to the program counter. The example presented in Table 5.3 demonstrates PicoBlaze needs more instructions than PICDiY to carry out these programming structures. In this example, PicoBlaze needs to go through the complete list of instructions to reach the last case. Depending on the number of programmed cases, this can be a critical aspect especially in certain applications. For instance, a program containing a FSM with many states would demand a huge number of instructions, affecting the performance.

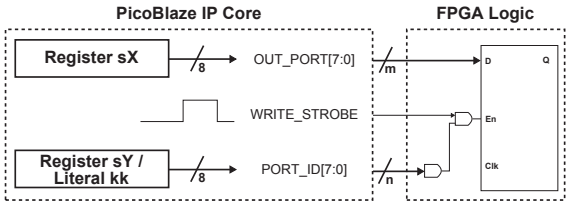


Figure 5.4: Output operation and FPGA interface for PicoBlaze.

Considering these aspects, it has been concluded that the PICDiY is an adequate candidate to be used as a target processor for different approaches. Specially taking into account that it is an adaptation-friendly processor thanks to its modularity.

Table 5.3: Comparison of FSM coding examples for PICDiY and PicoBlaze processors.

N. Instr.	PICDiY		PicoBlaze	
1	MOVF	state, w	COMP	state,0
2	ADDWF	PLC, F	JUMP	Z, state0
3	GOTO	state0	COMP	state,1
4	GOTO	state1	JUMP	Z, state1
5	GOTO	state2	COMP	state,2
6	GOTO	state3	JUMP	Z, state2
7	GOTO	state4	COMP	state,3
8	GOTO	state5	JUMP	Z, state3
9	GOTO	state6	COMP	state,4
10	GOTO	state7	JUMP	Z, state4
11	GOTO	state8	COMP	state,5
12	GOTO	state9	JUMP	Z, state5
13	GOTO	state10	COMP	state,6
14			JUMP	Z, state6
15			COMP	state,7
16			JUMP	Z, state7
17			COMP	state,8
18			JUMP	Z, state8
19			COMP	state,9
20			JUMP	Z, state9
21			COMP	state,10
22			JUMP	Z, state10

5.3 Validation of the Bitstream Based BRAM Approach

In order to assess in practice the performance of the proposed *Bitstream Based BRAM Approach* (BBBA), a ZedBoard has been utilized as an experimental platform. The validation of the proposed approach has been carried out by means of several tests based on the scheme presented in Figure 4.11. The systems implemented for these tests are aiming at reading, copying, writing and comparing different data blocks from different BRAM columns, single 18K BRAMs and 36K BRAMs. The performance scores analysed in this study are the resource overhead, the execution time and the availability.

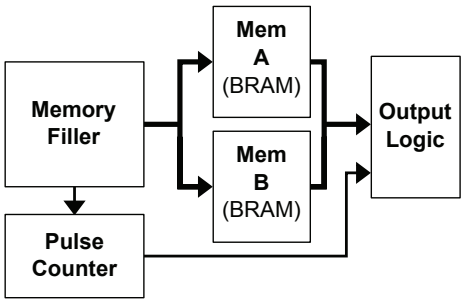
In the first step the bitstream approach has been compared with a common implementation. Most of the memory controllers are based on the use of processors, previously generated generic IP cores or specifically built FSMs. In order to account for the worst case scenario in terms of execution time, the content written in both memories has been equal. As a consequence, the comparing procedure traverses the entire memory space and, hence, the highest execution time is achieved. A dedicated soft-core processor would be a waste of resources; there-

fore, in this case, a FSM based controller is the most efficient and fastest method to control the memories.

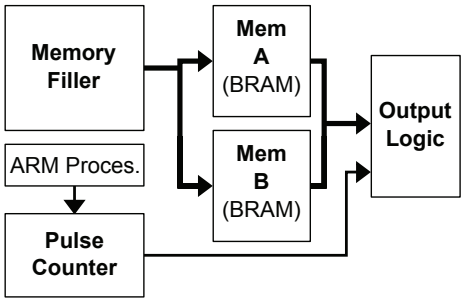
The implemented test applications are memory copying and memory comparing. For each application, different tests have been performed including entire BRAM column and single BRAM experiments. Both applications have been designed to be commanded by a `ready` signal, which permits choosing when the action at hand is executed over the second memory. In this way, both approaches (the FSM based and the bitstream based) feature the same functionality. However, in all tests the `ready` signal has been set to *active*, so as to avoid the effect of the waiting time in the measurement of processing time. All applications begin by fully writing the source memory, which has not been taken into account when measuring the processing time. Thereafter, the next step to perform the data copy process is to read the source memory, for which the application waits the `ready` signal from the destination memory. Upon receiving this signal, the data content is copied to the destination memory and cross-checked with the original data of the source memory in order to detect data discrepancies. To this end, once the writing process is done the cross-check starts by reading the first memory. After receiving the `ready` signal from the second memory, the data therein are read and compared to the previously read memory. In order to account for the worst case scenario in terms of execution time and evaluate the highest execution time case, the content written in both memories has been equal. No errors have been detected and all the memory addresses have been analysed during the comparing processes.

Figure 5.5 depicts the simplified block diagrams of the different testing designs implemented in the programmable logic part of the Zynq device. The original system has been implemented as a reference in order to evaluate the resource overhead. As shown in 5.5(a), it consists of different modules where the most relevant ones are the two memory blocks, the FSM based memory filler, the pulse counter and the output logic. The two memory blocks are the target memories, which are equally implemented with different sizes depending on the test to be addressed. The memory filler is an FSM based module used to fill the memories with specific test data. The pulse counter, which is composed by a counter and an FSM, measures the time spent by each approach. Finally, the output logic selects the signals to be displayed as outputs.

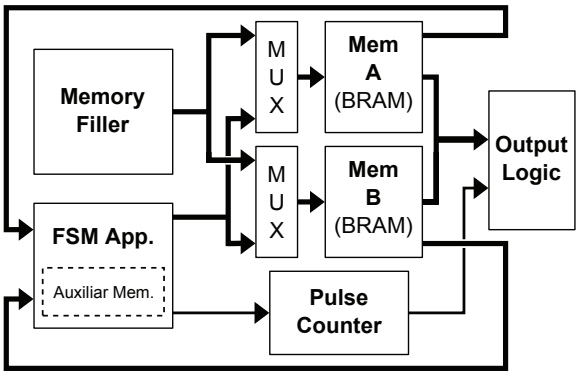
As evinced Figure in 5.5(b), the proposed BBBA is similar to the original system scheme. This is because it does not need any logic resource. The only difference is that the pulse counter is controlled by the ARM processor available in the processing system part of the Zynq device, which is only necessary to measure the time. The number of instructions to be executed by the ARM vary depending



(a) Original system.



(b) System with the BBBA.



(c) System with the FSM applications.

Figure 5.5: Block diagrams of the implemented approaches.

on the location of the BRAMs and the possibility of data overwriting. Both applications, memory copying and memory comparing, have been implemented with source and destination BRAMs in the same or different column and with or without overwriting.

Figure 5.5(c) presents the scheme for the FSM based memory copying and memory comparing operations. In this case, further modifications have been incorporated with respect to the original scheme. The most remarkable one is the addition of the FSM application module. For each test (copying and comparing) a specific FSM has been implemented, minimizing the number of states for an efficient coupling. In order to wait the `ready` signal, an auxiliary BRAM memory has been allocated in these modules. Both FSM applications also manage the pulse counter module. Finally, additional multiplexer modules have been added to manage the inputs of the target memories. As opposed to BBBA, when using FSMs there is no functional difference related to the location of the BRAMs, nor any the problem of data overwriting of the rest of BRAMs from the same BRAM column.

Table 5.4 summarizes the results of copying and comparing tests. In both tests, the analysed parameters are the same. The first five columns (Slice Registers, Slice LUTs, Occupied Slices, LUT FF pairs and 18K BRAMs) quantify the usage of FPGA resources. The number in parenthesis indicates the relative resource overhead with respect to the original system. The remaining parameter is the time used to perform each application extracted from the pulse counter module. It should be noted that, since the Processing System presents a non-deterministic response (due to uncontrollable side conditions e.g. hardware interruptions and hardware-software communication), there are small variations in the measured time for each experiment. Consequently, the provided scores represent averaged values after 10 experiments. In the same table, the first two rows correspond to the parameters of the original system without any additional element with the aim of being used as a reference. In this case, since no data operation is involved, there is no information for the time parameter. The next four rows are the values of the monitored parameters corresponding to the FSM based applications. Last but no least, the results of the bitstream manipulation based applications are shown in the last parameter set.

The conclusions drawn by observing the above table support the research hypothesis stared in Section 4.2. The BBBA based applications obtain significantly better results in terms of resources overhead. While the overhead in BBBA is essentially zero, the overheads of the FSM based approaches are between 50% and 180%. However, the results also confirm that the BBBA based applications are the most time consuming options in the benchmark. Different performed

Table 5.4: Implementation results summary of BRAM data copy and comparison tests (@100MHz).

		Slice Registers	Slice LUTs	BRAMs	Time (ms)
Orig.	BRAM	88	84	2	-
	BRAM Columns	110	529	40	-
FSM	BRAM Copy	135 (53%)	235 (80%)	3 (50%)	0.001
	BRAM Column Copy	165 (50%)	703 (33%)	60 (50%)	0.02
	BRAM Comp.	111 (26%)	198 (35%)	3 (50%)	0.001
	BRAM Column Comp.	141 (28%)	614 (16%)	60 (50%)	0.02
Bitstream	BRAM Copy(Same column / Overwr.)	88 (0%)	84 (0%)	2 (0%)	2.5
	BRAM Copy(Same column / No Overwr.)	88 (0%)	84 (0%)	2 (0%)	10.7
	BRAM Copy(Dif. column / Overwr.)	88 (0%)	84 (0%)	2 (0%)	2.8
	BRAM Copy(Dif. column / No Overwr.)	88 (0%)	84 (0%)	2 (0%)	12.1
	BRAM Column Copy	110 (0%)	529 (0%)	40 (0%)	10,2
	BRAM Comp.(Same column)	88 (0%)	84 (0%)	2 (0%)	2.5
	BRAM Comp.(Dif. column)	88 (0%)	84 (0%)	2 (0%)	4.7
	BRAM Column Comp.	110 (0%)	529 (0%)	40 (0%)	13.1

test have shown that the most time consuming phase is the bitstream processing necessary to read, create or compare partial bitstreams. Since this approach utilizes Xilinx general purpose functions, an optimization of these functions to fit with the bitstream management approaches could provide significantly better timing results. Another limitation is imposed by the maximum PCAP frequency (100 MHz). This limitation has a direct impact on the velocity of the bitstream writing and reading processes, since it limits the communication speed between the bitstream processing block and FPGA. Increasing this maximum frequency can be an interesting improvement, since it could increment the execution speed of the approach.

Moreover, it has been also confirmed via additional off-line experiments that the processing time varies depending on the relative location of the BRAMs and the possibility of overwriting the information of the destination memory, hence elucidating that the quickest cases are copying and comparing BRAMs allocated in the same column.

5.4 Validation of the Approach to Manage Data of Registers with the Bitstream

The validation of the *Approach to Manage Data of Registers with the Bitstream* has been performed in a simple manner and most of tests have been performed during the developing process of the technique. Non-essential elements have been avoided to elude any possible impact in bitstream's content and structure. Considering that it is a very particular approach no alternatives have been found to be compared with.

The development and validation design is shown in Figure A.1, which has been obtained from the utilized Vivado's project. In this design, the reset ports of the two 8-bit register modules are managed by the Processing System via EMIO interface. The Processing System also controls the `CAP` and `GSR` ports from `CAPTUREE2` and `STARTUPEE2` primitives, respectively. With the aim of maintaining simplicity and a faithful physical reading, the data input and output ports are connected directly to ZedBoard's switches and LEDs, respectively. The switch connected to the first data bit also controls the multiplexer, providing the possibility of visualizing the connection between both registers in real time. The bitstream is readback from the FPGA combining the PCAP and the Processing System, and stored in the DDR memory. The content of INIT values are obtained from the bitstream thanks to the location of flip-flop's content provided by a previously generated *.ll* file.

In a first validation step, the functionally of `GRESTORE` and `GCAPTURE` commands have been checked in a non-reconfigurable design with successful results. In addition, several data content copying experiments have been conducted with this setup. The basis of these tests has been a basic procedure. First, as Figure 5.6 shows, specific placement constraints have been used in order to utilize the same flip-flops of different resource columns for both registers. As showed in segments of .ll files represented in Figure 5.7, it has been confirmed that same structures (same frame offsets) are contained in the bitstream for the data of both registers, differing only in the FARs. Utilizing this information, the content of one registers has been copied to the other with the bitstream confirming the viability of the proposed approach.

After the first experiment, one of the registers has been implemented as reconfigurable module enabling the `RESET_AFTER_RECONFIG=TRUE` property, maintaining the other register and the rest of the design in the static region. In this second experiment, the combination of information extracted from read bitstreams, the *.ll* file and LEDs has proved that the utilization of the `RESET_AFTER_RECONFIG=TRUE` property protects the static area from the effect of the `GCAPTURE` command. In ad-

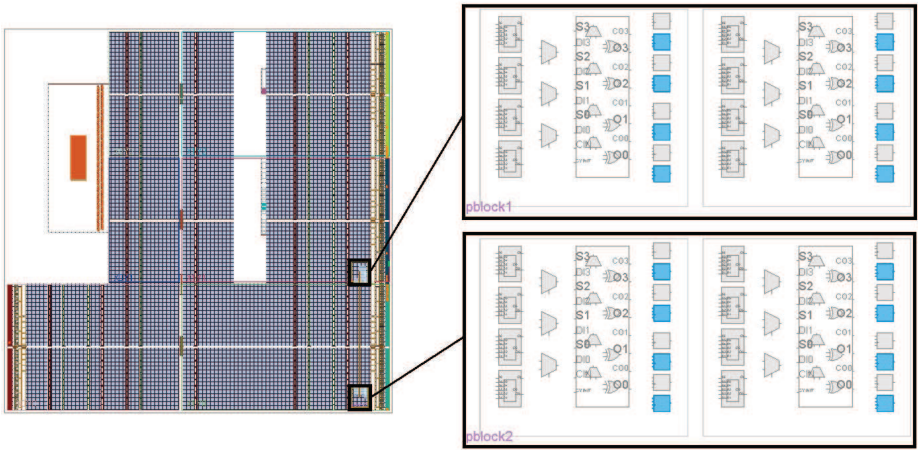


Figure 5.6: Device image of the design with placement constraints from Vivado.

```
REGISTER_A
    <offset> <FAR>      <frame offset> <information>
Bit 15982243 0x0040221f  3              Block=SLICE_X106Y50 Latch=AQ Net=Two_regs_i/reg8_black_box_0/U0/inst/o_Data[0]
Bit 15982244 0x0040221f  4              Block=SLICE_X107Y50 Latch=AQ Net=Two_regs_i/reg8_black_box_0/U0/inst/o_Data[4]
Bit 15982268 0x0040221f  28             Block=SLICE_X106Y50 Latch=BQ Net=Two_regs_i/reg8_black_box_0/U0/inst/o_Data[1]
Bit 15982269 0x0040221f  29             Block=SLICE_X107Y50 Latch=BQ Net=Two_regs_i/reg8_black_box_0/U0/inst/o_Data[5]
Bit 15982273 0x0040221f  33             Block=SLICE_X106Y50 Latch=CQ Net=Two_regs_i/reg8_black_box_0/U0/inst/o_Data[2]
Bit 15982274 0x0040221f  34             Block=SLICE_X107Y50 Latch=CQ Net=Two_regs_i/reg8_black_box_0/U0/inst/o_Data[6]
Bit 15982298 0x0040221f  58             Block=SLICE_X106Y50 Latch=DQ Net=Two_regs_i/reg8_black_box_0/U0/inst/o_Data[3]
Bit 15982299 0x0040221f  59             Block=SLICE_X107Y50 Latch=DQ Net=Two_regs_i/reg8_black_box_0/U0/inst/o_Data[7]

REGISTER_B
Bit 24275555 0x0042221f  3              Block=SLICE_X106Y0 Latch=AQ  Net=Two_regs_i/reg8_black_box_1/U0/inst/o_Data[0]
Bit 24275556 0x0042221f  4              Block=SLICE_X107Y0 Latch=AQ  Net=Two_regs_i/reg8_black_box_1/U0/inst/o_Data[4]
Bit 24275580 0x0042221f  28             Block=SLICE_X106Y0 Latch=BQ  Net=Two_regs_i/reg8_black_box_1/U0/inst/o_Data[1]
Bit 24275581 0x0042221f  29             Block=SLICE_X107Y0 Latch=BQ  Net=Two_regs_i/reg8_black_box_1/U0/inst/o_Data[5]
Bit 24275585 0x0042221f  33             Block=SLICE_X106Y0 Latch=CQ  Net=Two_regs_i/reg8_black_box_1/U0/inst/o_Data[2]
Bit 24275586 0x0042221f  34             Block=SLICE_X107Y0 Latch=CQ  Net=Two_regs_i/reg8_black_box_1/U0/inst/o_Data[6]
Bit 24275610 0x0042221f  58             Block=SLICE_X106Y0 Latch=DQ  Net=Two_regs_i/reg8_black_box_1/U0/inst/o_Data[3]
Bit 24275611 0x0042221f  59             Block=SLICE_X107Y0 Latch=DQ  Net=Two_regs_i/reg8_black_box_1/U0/inst/o_Data[7]
```

Figure 5.7: Fragment of the *.ll* file from the design with placement constraints.

dition, it has been demonstrated that when generating bitstreams, the size of the bitstream considerably increases by enabling the `RESET_AFTER_RECONFIG=TRUE` property. In this example in particular, it has been observed that while the size of a regular *.bit* file is 88.492 bytes, the size of a *.bit* file generated the `RESET_AFTER_RECONFIG=TRUE` property is 240.108 bytes. As a consequence, an increase of 271.3% in the size of the bitstream has been detected. The main reasons for this elevated percentage is that, considering the small size of the partial bitstream, the instructions added to the bitstream to protect the reconfigurable region imply a significant data overhead. In bigger reconfigurable designs the impact of these instruction is be lower.

In a further step, the proposed approach to protect/unprotect partial regions in 7 series devices by editing the partial bitstream has been validated. In this case, both registers have been implemented in each reconfigurable Pblock. After editing one of the generated partial bitstreams by erasing the proper 0xE00009BC special words, it has been downloaded so as to reconfigure the FPGA. The information provided by the LEDs and bitstream readbacks in different `GCAPTURE` and `GRESTORE` tests have successfully demonstrated that both reconfigurable regions are unprotected while the static region remains protected.

The final validation step has been to test the proposed flow to generate equal implementations of a design in different reconfigurable regions. In this case, a TMR design with three instances of the PICDiY soft-core processor have been implemented in three reconfigurable regions (Pblock0, Pblock1 and Pblock2). In this first implementation all resources of each instance have been freely placed by the tool. Thus, as can be observed in Figure 5.8 obtained from the device diagram of Vivado, the three resultant implementations present distinct placements. Afterwards, the *Location Constraints Flow* has been utilized to generate three equal implementation (selecting the Pblock0 as a reference). As it can be observed in Figure 5.9 the resource placement of the three entities within three Pblocks. In a similar way, the generated partial bitstreams of the three Pblocks have been compared in order to check if the location of data bits within the bitstream is the same. Furthermore, as Tables 5.5 and 5.6 demonstrate the resource usage of both implementations is almost the same. Table 5.5 also confirms that the proposed design flow does not affect the implementation in terms of power consumption and maximum operation frequency.

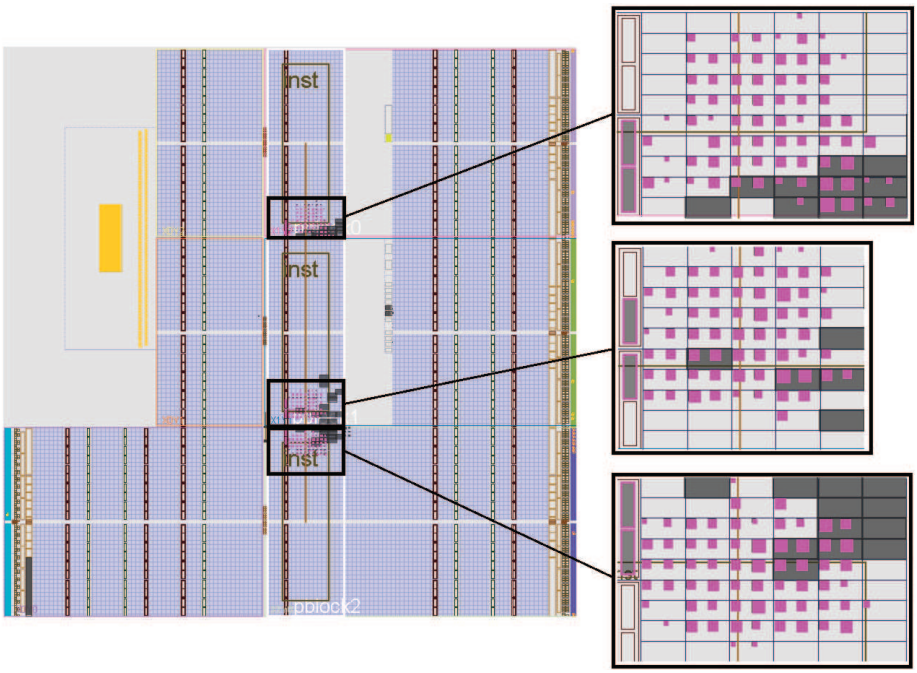


Figure 5.8: Device image of the TMR design without placement constraints from Vivado.

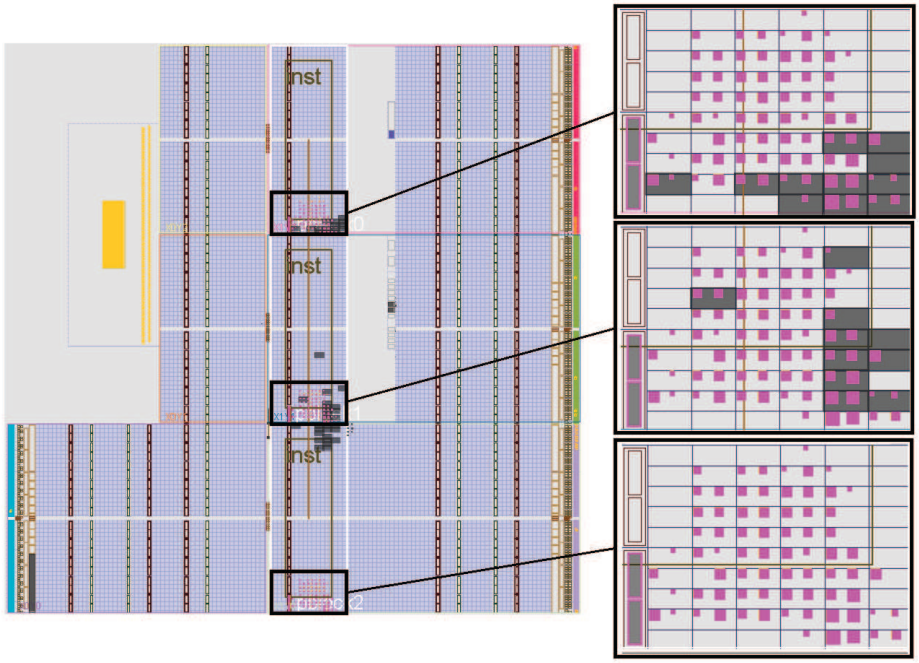


Figure 5.9: Device image of the TMR design witht placement constraints from Vivado.

Table 5.5: Implementation results summary of a reconfigurable TMR implementation with and without placement constraints (@60MHz).

Resource	Bitstream	Bitstream w/place
Slice LUTs	678	679
Slice Registers	187	187
F7 Muxes	3	3
Block RAM Tile	3	3
f_{max} (MHz)	80	80
Dynamic p. (W)	1.668	1.668
Static p. (W)	0.161	0.161

Table 5.6: Primitive utilization of a reconfigurable TMR implementation with and without placement constraints (@60MHz).

Resource	Bitstream	Bitstream w/place
FDRE	181	181
FDSE	6	6
CARRY4	24	24
LUT1	3	3
LUT2	90	90
LUT3	47	47
LUT4	55	55
LUT5	109	109
LUT6	405	405
MUXF7	3	3
RAMB18E1	6	6
RAMS32	21	21
CAPTUREE2	1	1
STARTUPE2	1	1
PS7	1	1
PLLE2_ADV	1	1

5.5 Validation of the Data Content Scrubbing Approach

Bearing in mind that the *Data Content Scrubbing Approach* is directly based on the previously validated *Bitstream Based BRAM Approach* (BBBA), its validation has been a straightforward operation. A PICDiY IP has been utilized to implement the *TestApp*. In a first step, the content of the program memory BRAM of the processor has been readback and stored as a *golden copy* in the external DDR memory. After that, several corrupted versions of the *golden copy* have been generated, by flipping different data-related bits. The corrupted partial bitstreams have been utilized to perform several error injection tests.

Thanks to performed tests it has been proved that this approach successfully performs a scrubbing of the data content of BRAMs without affecting the rest of resources. Hence, neither resource overhead increase nor performance penalty

has been observed. Due to the lack of data content scrubbing alternatives this approach has not been compared against other approaches.

5.6 Validation of Approach to Extract Data From Damaged Memories Using the BBBA

Similar to what happens in the case of the *Data Content Scrubbing Approach*'s validation, the validation of the *Approach to Extract Data From Damaged Memories Using the BBBA* is based on the validation of the *Bitstream Based BRAM Approach* (BBBA). Figure A.2 shows the block diagram of the utilized implementation extracted from Vivado. The design contains two memory blocks, with a depth of 256 8-bit words, extracted from the PICDiY design. Both memories can be written or read by the FSM based BRAM Filler block. An additional enable port has been used to select, via external switch, if each memory has to be written. The outputs of both memories are connected to a multiplexer module in order to select which ones are shown on the LEDs. The validation design also includes the Processing System to perform readbacks and reconfigure the FPGA through PCAP interface.

The blue module in Figure A.2 represents the memory block that has been implemented as a reconfigurable module, while the rest of modules, including the green memory block, has been located in the static region. The reconfigurable module has been utilized as a target memory block. In this way, the generated partial bitstream has been utilized to create several corrupted files by flipping different bits. After distinct error injection attempts the interface of the target memory block has been damaged impeding to read data. Next, with the aim of extracting data content of the damaged memory, the BBBA has been used to copy the bitstream portion with the required information to the memory block on the static area. This have provided the possibility of successfully extract all the stored information.

This validation test has proved the efficacy of the *Data Content Scrubbing Approach*, which can recover data from BRAM based data memories without affecting the design in terms of resource usage and performance.

5.7 Validation of the Lockstep Approaches

In order to evaluate the correct functionality of the lockstep approaches, the three proposed methods have been physically tested by running the *TestApp* in a ZedBoard.

Despite partial reconfiguration based injection provides a wider range of results, especially in terms of robustness, it increases design and implementation times. Furthermore, considering the vast existing literature the lockstep has been proven to be a consolidated and thoroughly analysed technique in terms of robustness. Thus, bearing in mind these aspects and that proposed approaches are focused on improving both checkpointing and rollback processes, fault simulation tests have been carried out by manually flipping data bits from user registers rather than partial reconfiguration based fault injections. This strategy has considerably simplified test tasks, enabling to focus them on the correct behaviour of the approaches.

As depicted in Figure 5.10, one of the processors has been corrupted by modifying certain registers by an additional input to force errors. The most significant signals, such as outputs of lockstep controller, W register or IDC have been monitored by Integrated Logic Analyzer (ILA) module from Vivado.

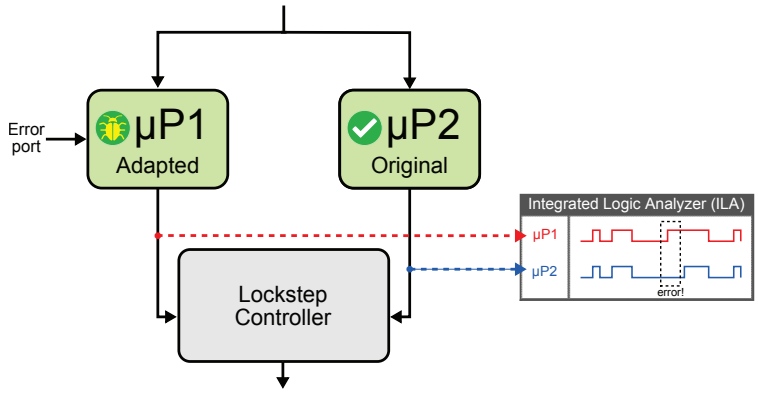


Figure 5.10: Test procedure for the lockstep approaches validation.

In both, *Bitstream Based Low Overhead Lockstep* and *Bitstream Based Autonomous Lockstep* approaches, the duplicated PICDiY processors have been implemented in a single same reconfigurable region. Thus, reducing the complexity of checkpointing and rollback processes. In addition, this permits to avoid the need of editing partial bitstreams and the utilization of the

RESET_AFTER_RECONFIG=TRUE property to protect static regions.

The validation tests have been executed following these steps:

1. Configure the entire device and constantly run the *TestApp*.
2. Observe the context before the error injection with ILA.
3. Emulate a failure by triggering the adapted register's error signal.
4. Check correct rollback (compare with pre-fault injection context) using ILA.

This validation flow has been utilized in all three proposed approaches. It is important to observe the context before injecting the fault in order to be able to check if the rollback has been performed successfully. For this reason, fault injections and pre-error context checkings have been carried out taken into account the checkpointing frequency of each method. Due to its time demand, monitoring the entire rollback process of the *Bitstream Based Low Overhead Lockstep* approach would require a huge resource usage. Thus, this approach has been partially monitored with ILA by utilizing the Processing System and the EMIO interface in order to define trigger events related to specific moments and only capture signals in required instants.

During different performed tests all three proposed lockstep approaches have been able to successfully recover from induced errors, proving the validity of all proposed methods.

The obtained results are summarized in Table 5.7. As it can be observed, the first column includes the results for a single PICDiY. This data has been used as the reference value for the percentage information. The next three columns contain the results of each of the three proposed approaches. The last column includes the results for a coarse-grained TMR implementation included for comparison purposes. Regarding the information provided, while the first five rows of the table are a summary of the utilized resources, the next three rows contain timing related information (maximum operating frequency and the time requirements for both checkpointing and rollback processes). Finally, the last two rows provide information of the power consumption (static and dynamic). The implementations utilized to obtain these results do not include the logic resources used for error-injections and the *TestApp*, and the employed ILA module. For a fair comparison, all the results have been obtained implementing the designs with the same clocking constraints (60 MHz). Table 5.8 includes primitive utilization results, providing with further information in terms of resource utilization.

As results reveal the *HW Fast Lockstep* approach provides very fast checkpointing and rollback operations. Nevertheless, the hardware increase introduced is sub-

Table 5.7: Implementation results summary of the lockstep approaches, a single PICDiY and a coarse grained TMR (@60MHz).

Resource	PICDiY	HW Fast	Low Overhead	Autonomous	TMR
Slice LUTs	232 (100%)	690 (297%)	400 (172%)	435 (188%)	726 (313%)
Slice Registers	74 (100%)	272 (368%)	130 (176%)	141 (191%)	193 (261%)
F7 Muxes	1 (100%)	28 (2800%)	2 (200%)	34 (3400%)	3 (300%)
F8 Muxes	(100%)	2 (-%)		14 (-%)	
Block RAM Tile	1 (100%)	1 (200%)	2 (100%)	1 (200%)	3 (600%)
f_{max} (MHz)	107 (100%)	64 (60%)	85 (79 %)	70 (65%)	90 (84%)
Checkpoint time (ms)	-	3.33×10^{-8}	2.64	1.67×10^{-8}	- -
Rollback time (ms)	-	6.67×10^{-8}	0.03	3.33×10^{-8}	- -
Dynamic p. (W)	0.133 (100%)	0.139 (105%)	1.638 (1232%)	0.125 (92%)	0.139 (105%)
Static p. (W)	0.125 (100%)	0.125 (100%)	0.160 (100%)	0.125 (100%)	0.125 (100%)

Table 5.8: Primitive utilization of the lockstep approaches, a single PICDiY and a coarse grained TMR (@60MHz).

Resource	PICDiY		HW Fast		Low Overhead		Autonomous		TMR	
FDRE	72	(100%)	266	(369 %)	126	(175%)	136	(189%)	187	(260%)
FDSE	2	(100%)	6	(300 %)	4	(200%)	5	(250%)	6	(300%)
CARRY4	8	(100%)	19	(238 %)	16	(200%)	16	(200%)	24	(300%)
LUT1	1	(100%)			2	(200%)	2	(200%)	3	(300%)
LUT2	30	(100%)	61	(203 %)	60	(200%)	64	(213%)	90	(300%)
LUT3	13	(100%)	99	(762 %)	26	(200%)	25	(192%)	47	(362%)
LUT4	17	(100%)	36	(212 %)	34	(200%)	31	(182%)	56	(329%)
LUT5	36	(100%)	159	(442 %)	72	(200%)	141	(392%)	111	(308%)
LUT6	132	(100%)	315	(239 %)	267	(202%)	204	(155%)	402	(305%)
MUXF7	1	(100%)	28	(2800%)	2	(200%)	34	(3400%)	3	(300%)
MUXF8			2				14	(-%)		
RAMB18E1	2	(100%)	2	(100 %)	4	(200%)	2	(100%)	6	(300%)
RAMS32	7	(100%)	26	(371 %)	14	(200%)	14	(200%)	21	(300%)
CAPTUREE2					1	(-%)	1	(-%)		
STARTUPE2					1	(-%)	1	(-%)		
PS7					1	(-%)				
PLLE2_ADV	1	(100%)	1	(100 %)	1	(100%)	1	(100%)	1	(100%)

stantial, reaching almost the same overhead as the TMR implementation (over the 300%). Due to this similar resource overhead in both implementations the power consumption is the same. Although this is not a significant overhead and power consumption reduction for a lockstep approach (compared to the TMR), these results could be improved when applying the approach to larger processors. The reason for this overhead in the *HW Fast Lockstep* approach resides in the relation between the number of resources used to implement the processor and additional resources needed for its adaptation. This additional logic implies a significant increase in the overhead due to required modifications of registers and IDC module, and the implementation of context backup and lockstep modules. Nevertheless, applying the approach to a larger processor should not imply a proportional increase of resources usage. Hence, the obtained results should be improved in comparison with a TMR implementation, which will always increase its overhead proportionally. Another relevant drawback of this approach is the performance penalty introduced in the design. In this case, due to the increase of data paths the maximum operation frequency reached decreases a 40% comparing with the design with a single PICDiY.

Moreover, it has been demonstrated that *Bitstream Based Low Overhead Lockstep* is the less resource consuming approach. Related with this fact, the performance penalty introduced is moderate, decreasing only the 21%. However, the weakest point of this approach is the time demand of the checkpointing operation since it utilizes the BBBA. Another handicap of this approach is its the power consumption. The utilization of the Processing System increases drastically the power consumption, specially the dynamic power. However, in designs where the Processing System is already utilized, this power increase can be considered inherent to the design. Hence, applying this hardening approach in such scenarios will not imply any increase of the power consumption.

The *Bitstream Based Autonomous Lockstep* provides interesting trade-off results. Its hardware overhead is comparable to the results of the *Bitstream Based Low Overhead Lockstep*. On the other hand, the results in terms of checkpointing and rollback speed are even better that the *HW Fast Lockstep*. The main drawback of this technique is that several data paths are increased, specially due to the logic introduced for the ECC hardening of memories. Hence, it noticeably decreases the maximum operation frequency. Related with the power consumption this approach presents remarkable results. After performing a deeper analysis of the power results it has been observed that the power consumption is related to the `MMCM` primitive used by the `Clocking Wizard`. Unlike the other implementations this one requires to configure the `Clocking Wizard` to generate a second clocking signal (used to be able to halt the design). This different configuration appears to reduce the power consumption of the `MMCM` primitive.

In general terms, all proposed lockstep approaches have demonstrated to be successful. Although each implementation provides different benefits and present distinct drawbacks it can be concluded that the most balanced solution is *Bitstream Based Autonomous Lockstep*. Finally, compared to the TMR all the proposed lockstep approaches introduce lower hardware overhead. Nevertheless, they also add higher performance penalty, obtaining lower maximum operation frequencies.

5.8 Validation of the Synchronization Approaches

Similar to when validating the lockstep approaches, the fault emulation of the synchronization approaches has been focused on checking the correct synchronization behaviour. Considering that the synchronization issue appears after repairing damaged modules, partial reconfiguration based fault injection techniques enable a very straightforward testing method for the proposed synchronization approaches. Opposed to a complete fault injection (as the experiments executed for the PicoBlaze in [332]), the simplified injection method of altered partial bitstreams proposed in this work focuses mainly on the correct error detection and recovery, rather than determining the robustness of the complete setup.

In Figure 5.11 the test approach is showcased for the processor instance *μP1*. Several corrupted versions of the PICDiY processor instance have been created by editing the original partial bitstream. In this way, a correct bitstream for each partition containing a PICDiY processor and some corrupted bitstreams are created to force the processor out of sync. Due to this, the fault injection and the repairing processes can be performed in straightforward manner.

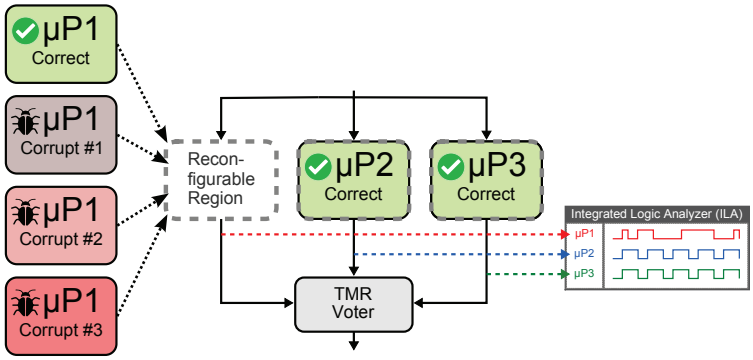


Figure 5.11: Test procedure for the synchronization validation.

A synchronization test is executed as follows:

1. Configure the entire device.
2. Emulate a failure by loading a *corrupt* partial bitstream.
3. Repair the failure by loading the *correct* partial bitstream.
4. Trigger synchronization routine.
5. Check correct synchronization using ILA.

The use of ILA permits to validate the synchronization process on the actual FPGA by monitoring the external processor signals, such as the instruction address for observation of the program counter and also internal signals like the stack pointer for validation of *Interrupt Sync* and *Hw Sync* approaches. In order to circumvent possible issues with the trigger settings when using ILA for both partial reconfiguration and logic analysis, the synchronization has been triggered manually instead of invoking this step automatically after a partial reconfiguration. Due to the time required by the BBBA technique, in the case of *Bitstream Sync* the utilization of ILA has required a special treatment. Consequently, each test has consisted in two captures, one triggered with the synchronization and the other after applying the BBBA approach. This is because monitoring the entire process would require a huge resource usage (especially BRAMs) to implement the ILA in the FPGA.

All five proposed synchronization methods have been implemented on the ZedBoard to validate their correct operation. Fault injection tests have been executed as defined previously. After repairing faulty modules the synchronization has been triggered. Functional behaviours have been validated by observing the synchronization processes of the different approaches with the ILA.

Results of the different synchronization approaches are unveiled in Table 5.9, where the values of the different implementations are summarized as absolute values and in relation to *Reset Sync* (in brackets). As it can be observed in the first five rows that contain a resource utilization summary, the *Hw Sync* approach introduces the biggest hardware overhead. Besides, while *Force Sync* and *Interrupt Sync* approaches increase the resource utilization moderately, both, *Reset Sync* and *Bitstream Sync* approaches, do not affect the hardware overhead. Similar conclusions can be obtained from Table 5.10, which includes the utilization of primitives. Considering the proposed approaches, the increase in resource requirements is significant, especially if a high degree of hardware synchronization is supported. When implementing a processor as small as the PICDiY, adding modules such as the synchronization memory or the Sync Control block has a stronger relative weight as compared to bigger processors. Regarding power con-

sumption, *Bitstream Sync* presents the worst results due to the use of the Processing System. The power consumption results of the rest of implementations are closely related with the resource overhead: higher resource utilization comes with higher power consumption. Furthermore, although the are not big differences in the possible operation frequency for all designs (90 MHz), the *Bitstream Sync* approach presents the lowest value (80 MHz).

Another relevant information included in Table 5.9 is the synchronization time demanded for each approach implementation. As it can be observed, despite its limitations, the *Reset Sync* approach synchronizes the system instantaneously. On the contrary, the *Bitstream Sync* is the most time consuming method. This is because of the time needed to read, process and write the bitstream. The synchronization times for the rest of approaches depends on the number of registers and memory positions to be synchronized. It has to be remarked that the results of these approaches are closely dependant of the system clocking frequency. Besides, for these tests the worst case scenario has been considered, synchronizing all the registers and the entire data memory. In this way, the *Force Sync* needs 0.036 ms (112 clock cycles for the 7 special registers and 2048 clock cycles for 128 positions of the data memory) to synchronize the design. Similarly, the *Interrupt Sync* needs 0.0363 ms. This slight difference resides in the need of 16 clock cycles to synchronize the stack. Finally, *Hw Sync* significantly improves the synchronization time to 0.0045 ms (272 clk cycles).

Based on the trade-off between the three factors of the synchronization approaches (the FPGA resource overhead, the synchronization speed and the elements which support synchronization) different applications might benefit from different synchronization techniques. A simple algorithm, like the Advanced Encryption Standard (AES), does not require a very elaborate synchronization. If the key does not need to be stored, a cyclic reset (*Reset Sync*) might be a sufficient solution. If only a small amount of data needs to be kept on the processor (for example in cyclic sensor readouts) *Force Sync* is a suitable option. The last readouts can be kept in the scratchpad memory and the rest of the system can be forced to a sync whenever this is required. Both, *Interrupt Sync* and *Hw Sync* approaches, represent almost no restrictions to the device software and only minor considerations need to be taken into account for the development of the ISR. *Bitstream Sync* is able to synchronize all memory elements of a design without any software restrictions. However, its time demand makes it suitable only for applications with loose time constraints.

Table 5.9: Implementation results summary of the synchronization approaches (@60MHz).

Resource	Reset		Force		Interrupt		HW		Bitstream	
Slice LUTs	675	(100%)	684	(101%)	760	(113%)	1008	(149%)	679	(100%)
Slice Registers	187	(100%)	205	(110%)	239	(128%)	437	(234%)	187	(100%)
F7 Muxes	3	(100%)	3	(100%)	36	(1200%)	57	(1900%)	3	(100%)
F8 Muxes					6	(-%)	21	(-%)		
Block RAM Tile	3	(100%)	3	(100%)	3	(100%)	3	(100%)	3	(100%)
f_{max} (MHz)	90	(100%)	90	(100%)	90	(100%)	90	(100%)	80	(89%)
Sync. time (ms)	-		0.036		0.0363		0.0045		12.9	
Dynamic p. (W)	0.139	(100%)	0.139	(100%)	0.140	(101%)	0.147	(106%)	1.668	(1200000%)
Static p. (W)	0.125	(100%)	0.125	(100%)	0.125	(100%)	0.125	(100%)	0.161	(129%)

Table 5.10: Primitive utilization of the synchronization approaches (@60MHz).

Resource	Reset		Force		Interrupt		HW		Bitstream	
FDRE	181	(100%)	199	(110%)	233	(129%)	430	(238%)	181	(100%)
FDSE	6	(100%)	6	(100%)	6	(100%)	7	(117%)	6	(100%)
CARRY4	24	(100%)	24	(100%)	24	(100%)	27	(113%)	24	(100%)
LUT1	3	(100%)	3	(100%)	9	(300%)	56	(1867%)	3	(100%)
LUT2	90	(100%)	90	(100%)	78	(87%)	85	(94%)	90	(100%)
LUT3	47	(100%)	48	(102%)	94	(200%)	270	(574%)	47	(100%)
LUT4	55	(100%)	64	(116%)	45	(82%)	60	(109%)	55	(100%)
LUT5	108	(100%)	109	(101%)	148	(137%)	188	(174%)	109	(101%)
LUT6	403	(100%)	402	(100%)	392	(97%)	423	(105%)	405	(100%)
MUXF7	3	(100%)	3	(100%)	36	(1200%)	57	(1900%)	3	(100%)
MUXF8					6	(-%)	21			
RAMB18E1	6	(100%)	6	(100%)	6	(100%)	6	(100%)	6	(100%)
RAMD32					72	(-%)	72	(-%)		
RAMS32	21	(100%)	21	(100%)	24	(114%)	24	(100%)	21	(-%)
CAPTUREE2									1	(-%)
STARTUPE2									1	(-%)
PS7									1	(-%)
PLLE2_ADV	1	(100%)	1	(100%)	1	(100%)	1	(100%)	1	(-%)

Chapter 6

Conclusions and future work

6.1 Conclusions

In this work, a basis for fault tolerance of SRAM-based FPGA implementations, especially focusing on soft-core processor designs, has been presented. In order to provide a context, the technology of SRAM FPGAs has been introduced emphasizing certain aspects of 7 series Xilinx devices. In direct relation with fault tolerance of SRAM FPGAs, the issue of radiation induced faults (mostly single events upsets) has been also discussed, studying their adverse effects.

The spotlight of the state of the art pivots around hardening techniques for SRAM based designs, focusing on soft-core processors. Considering that induced faults can affect both user data and configuration memories, different strategies have been studied in this field. Most of existing hardening techniques are based on providing different redundancy types, such as, hardware, data, software or time. Existing hardening techniques improve reliability levels but they also come with drawbacks, such as, resource overhead, performance penalty, appearance of single points of failure or availability. It has to be remarked that resource overhead also yields lower maximum frequencies (longer data paths), higher power consumption (more active elements), worst fault tolerance (more susceptible elements) and less resources available for implementation purposes. Among the studied redundancy types the prevalent one is the hardware redundancy due to the reliability and availability levels that it provides. The most extended redundancy levels are the Triple Modular Redundancy (TMR) and the Dual Modular Redundancy (DMR). While DMR techniques provide less resource usage, TMR approaches

present higher reliability and availability levels. When applying DMR strategies to soft-core processors they are commonly implemented combining checkpointing and rollback techniques. That introduces either hardware overhead and performance penalty to designs. On the other hand, one of the most recurrent issues when utilizing TMR schemes to harden soft-core processor designs is the synchronization after repairing modules with reconfiguration. Due to this, in this work the need for synchronization techniques has been underlined when using hardware redundancy techniques and partial reconfiguration for building reliable systems. Furthermore, bearing in mind that TMR strategies only mask errors and do not fix them, the accumulation of masked errors is a recurrent potential problem. The most utilized technique to address this issue is to perform bitstream scrubbings. This technique consists in reloading correct partial bitstreams in order to correct masked configuration errors. The state of the art have highlighted an important gap in user data content scrubbing.

Additionally, another area of interest in the state of the art has been addressed in order to investigate the possibility of managing user data content through the bitstream. This possibility has been considered because its potential to provide useful features to existing hardening techniques. Distinct approaches related to this topic have been published. Nevertheless, the state of the art has highlighted a scarcity of methods addressing the features of newer FPGA devices like 7 series by Xilinx.

Based on the knowledge acquired in the state of the art, distinct approaches haven been developed. Five different synchronization approaches for TMR (or higher) and partial reconfiguration protected soft-core processor designs have been proposed so as to cover a wide spectrum of synchronization complexity levels. In a similar way three different lockstep approaches have been developed to provide distinct solutions able to fit to specific DMR scenarios.

On the other hand, two approached to manage user data content through the bitstream have been proposed. While one is suitable to manage the content of BRAM based memories, the other targets managing content of registers. Both approaches represent a new way to manage user data without introducing hardware overhead in designs, opening new application paths. In addition, thanks to the use of dynamic partial reconfiguration these new data management strategies can be performed in runtime. For this reason, despite its higher time demand, proposed techniques does not degrade the system overall performance in a number of applications. Thanks to bitstream based data management it is feasible to modify entire programs or single instructions without making any new synthesis or implementation. Both techniques have been applied to distinct fault tolerance schemes proposing new approaches, such as, user data content scrubbing

and data extraction from memories with damaged interfaces. These bitstream based techniques have been also utilized in two of the three proposed locksted approaches. In a similar way, the bitstream based user data management has been utilized in one of the five proposed approaches to synchronize repaired soft-core processors in hardware redundancy schemes.

All proposed approaches have been validated in both simulations and in actual FPGA implementation, by executing different kinds of applications. The selected hardware platform has been a Zynq SoC based board. In addition, the PICDiY, a specifically designed 8-bit modular soft-core processor, has been developed for this work. The PICDiY has been utilized as a target unit in both designing and validation processes of proposed soft-core processor hardening approaches. The correct operation of all suggested methods has been proven and the advantages and disadvantages have been thoroughly discussed. Different application scenarios have been suggested depending on the implementation and features of the proposed approaches.

6.2 Main Contributions

This section lists the main contributions of this thesis, providing a brief description of each point and including a reference to the corresponding section of the manuscript where the topic is discussed in detail.

1. **Soft-core processor.** The PICDiY, an 8-bit soft-core processor has been developed in this work. Its minimalist design and complete self-sufficiency provide interesting characteristics to be used in several scenarios without significantly impacting the design's resource usage. In addition its modularity makes it a great candidate for adaptations, enabling it to fit closely to specific designs. Thanks to its platform-independence it can be implemented in devices from any vendor. In addition, free Basic based tools have been developed to generate the program memory HDL modules of the soft-core processor. The architecture of PICDiY is introduced in Section 4.1.

2. **BRAM content management through bitstream.** The proposed approach is able to read and write the data content from BRAMs in FPGA based designs by reading and processing the information stored in the bitstream. Thanks to this method it is possible to extract, load, copy or compare the information of BRAMs without neither resource overhead nor performance penalty in the design. It can also be applied to existing designs without the need of re-synthesizing. These advantages makes it an interest-

ing tool to carry out several existing applications improving relevant aspects and it also opens the doors to the design of cutting-edge applications. For this reason it has been utilized in several of the proposed approaches. Despite this approach has been developed focusing on Zynq, it is possible to migrate several of this ideas to other Xilinx devices. Section 4.2 describes the approach in depth.

3. **Registers data management through bitstream.** This method enables to manage data content of registers utilizing the bitstream. Thanks to the proposed method the content of registers can be captured or written without significantly affecting the resource usage nor design's performance. A relevant benefit of this technique is that in several scenarios can be utilized autonomously, with no need of any processing module. This approach includes two design flows to carry out this data management. While the first proposed flow allows to protect/unprotect partial regions, the second permits to generate equal implementations of a design in different reconfigurable regions. Developed for 7 series devices by Xilinx, several concepts proposed in this approach can be adapted to other Xilinx devices. Details of this method are presented in Section 4.3.

4. **Data content scrubbing.** This technique permits to performs data content scrubbings in BRAM based memories. Bearing in mind that memory hardening techniques mainly mask errors and they do no fix them, it is an interesting alternative to correct masked errors in memories. Related with soft-core processors it is especially interesting to repair program memory modules. Section 4.4 provides the relevant aspects of this approach.

5. **Extract data from damaged memories.** Induced faults can affect memory interfaces impeding to obtain the stored information. Despite that the probability of such scenario is not especially high, this situation can be critical when the stored data is sensitive information. Section 4.5 presents the proposed method to recover data from damaged BRAM memories.

6. **Lockstep approaches for soft-core processors.** Three approaches are proposed providing different solutions for the common issues of checkpoint and rollback techniques when hardening soft-core processors in lockstep schemes. Each approach focuses on improving specific features, such as, reduced hardware overhead, minimum checkpoint frequency and fast rollback. Features of each proposed approaches are detailed in Section 4.6.

7. **Synchronization of hardware redundancy protected soft-core processors.** Five approaches are proposed to deal with the synchronization problem that arises after partially reconfiguring a faulty soft-copre proces-

sor module. Each approach provides different solutions in terms of resource usage, self-sufficiency, synchronization and availability. These techniques are valuable tools that could help designers when trade-off decisions have to be made. Section 4.7 discusses the main characteristics of proposed techniques.

6.3 Scientific Publications in the Context of this Work

In this Section all scientific publications published during the development process of this work are presented. The publication are divided into journal publications and conference publications.

Journal publications

J1) I. Villalta, U. Bidarte, **J. Gomez-Cornejo**, J. Jiménez and J. Lázaro. *"SEU Emulation in Industrial SoCs Combining Microprocessor and FPGA"*, Reliability Engineering & System Safety, vol. 170, pages 53 - 63, 2018. Impact Factor (JCR): 3.153. Ranking: Q1.

J2) I. Villalta, U. Bidarte, **J. Gomez-Cornejo**, J. Jiménez and A. Astarloa. *"Estimating the SEU Failure Rate of Designs Implemented in FPGAs in Presence of MCUs"*, Microelectronics Reliability, vol. 78, pages 85 - 92, 2017. Impact Factor (JCR): 1.371. Ranking: Q3.

J3) **J. Gomez-Cornejo**, A. Zuloaga, I. Villalta, J. Del Ser, U. Kretzschmar and J. Lázaro. *"A Novel BRAM Content Accessing and Processing Method based on FPGA Configuration Bitstream"*, Microelectronic Engineering, vol. 49, pages 64 - 67, 2017. Impact Factor (JCR): 1.025. Ranking: Q3.

J4) U. Kretzschmar, **J. Gomez-Cornejo**, A. Astarloa, U. Bidarte and J. Del Ser. *"Synchronization of Faulty Processors in Coarse-Grained TMR Protected Partially Reconfigurable FPGA Designs"*, Reliability Engineering & System Safety, vol. 151, pages 1 - 9, 2016. Impact Factor (JCR): 3.153. Ranking: Q1.

Conference publications

C1) I. Villalta, U. Bidarte, **J. Gomez-Cornejo**, A. Zuloaga, and C. Cuadrado, *"Emulation of Multiple Cell Upsets in FPGAs"*, Conference on Design of

Circuits and Integrated Systems (DCIS), Barcelona (Spain), 2017.

C2) I. Villalta, U. Bidarte, **J. Gomez-Cornejo**, J. Jiménez and C. Cuadrado, *"Effect of Different Design Stages on the SEU Failure Rate of FPGA Systems"*, Conference on Design of Circuits and Integrated Systems (DCIS), Granada (Spain), 2016.

C3) I. Villalta, U. Bidarte, **J. Gomez-Cornejo**, J. Lázaro and C. Cuadrado, *"Dependability in FPGAs, a Review"*, Conference on Design of Circuits and Integrated Systems (DCIS), Estoril (Portugal), 2015.

C4) U. Kretzschmar, **J. Gomez-Cornejo**, N. Moreira, U. Bidarte and A. Astarloa. *"A versatile FPGA Demonstration Platform for academic Use"*, Tecnologías, Aprendizaje y Enseñanza de la Enseñanza de la Electrónica (TAEE), 2014.

C5) **J. Gomez-Cornejo**, A. Zuloaga, U. Kretzschmar, U. Bidarte, J. Jiménez. *"Study of Implementation Alternatives for a Lockstep Approach in FPGA Soft Core Processors"*, Conference on Design of Circuits and Integrated Systems (DCIS), Donostia (Spain), 2013.

C6) **J. Gomez-Cornejo**, A. Zuloaga, U. Kretzschmar, U. Bidarte, A. Astarloa. *"Fast Context Reloading Lockstep Approach for SEUs Mitigation in a FPGA Soft Core Processor"*, Conference of the IEEE Industrial Electronics Society (IECON), Vienna (Austria), 2013.

C7) **J. Gomez-Cornejo**, A. Zuloaga, U. Kretzschmar, U. Bidarte, A. Astarloa. *"Interface Tasks Oriented 8-bit Soft-Core Processor"*. FPGAworld, Tampere (Finland), 2012.

C8) **J. Gomez-Cornejo**, A. Zuloaga, U. Kretzschmar, U. Bidarte, A. Astarloa. *"Implementación en FPGA de un Procesador Soft-Core PIC16"*. Seminario Anual de Automática, Electrónica Industrial e Instrumentación (SAAEI), Guimarães (Portugal), 2012.

6.4 Future work

This Section proposes some lines of research in order to give continuity to the work presented in this thesis. These lines are the following:

- **Further analysis of proposed fault tolerance approaches.** The experimental work presented in this manuscript has been focused on validating proposed approaches in terms of functionality. In particular, the aim

of validation tests carried out for synchronization and lockstep have been focused on determining correct error detection and recovery behaviours, rather than to evaluate the robustness level of designs. An interesting further step could be applying complete fault injection emulation tests in order to obtain more accurate results in terms of fault tolerance. Similarly, applying physical fault injection campaigns could provide additional valuable information. Another interesting alternative is to apply the proposed hardening approaches to additional soft-core processors. The case of more complex processors, such as, LEON3 [334] or Mico32 [333] is an interesting option to obtain complementary information, especially in terms of impact in the resource overhead.

- **Improve time specifications of the bitstream based approaches.** The most remarkable drawback of the approaches based on manipulating bitstream is their time demand. The main reason for this time consumption resides in the bitstream reading, manipulation and transferring processes. The programs utilized for these approaches are based in Xilinx functions that can be optimized to obtain a faster responses. Hence, an interesting path is to optimize the programs used in the proposed approaches by customizing Xilinx functions. For instance, the bitstream reading is done frame by frame, and consequently is it affects the overall processing time of the proposed method. Therefore, the optimization of the functions to read more than one frame at once (i.e. bulk read) could make the technique faster. Another, aspect that limits the speed of the approaches is the limited speed of the configuration port. Thus, an interesting research line could be to investigate new ways to enhance the reconfiguration speed. A remarkable example in this direction is the ZyCAP [114]. It is reported that the ZyCAP achieves a reconfiguration throughput of 382 MB/s, improving over PCAP by almost 3x.

- **Adapt the bitstream based approaches to different technologies.** Proposed approaches have been closely developed for Xilinx 7 series device. An interesting research line can be to adapt these approaches to other families from Xilinx and even to other vendors' devices, such as, Intel (Altera), Microsemi, etc.

- **Expand the application scope of bitstream based approaches.** Besides fault tolerance related scenarios, the bitstream based approaches proposed in this paper pave the way for other avant garde applications. For instance, it may serve as a novel method to store data under privacy requirements. Since by using proposed approaches based on the bitstream, memories can be managed without using any port, the existence of such

memory resources can be hidden in the design, implementing them as isolated memory blocks that can be accessed with no buses. Another example of a possible application is the management with multiprocessor systems shared memory. In order to guarantee the memory coherence and minimize the performance overhead, different protocols have been proposed [335, 336]. Proposed bitstream based approaches could represent an interesting alternative to avoid the use of these complex protocols, buses and interconnections. This could simplify designs by enabling new ways of exchanging data between processors. It even makes it possible for one processor of the system to change or adjust the functionality of another by only changing a number of instructions of its program memory.

Appendix A

Hardware Implementation details of the Proposed Approaches

This appendix contains the block diagrams of the validation designs extracted from Vivado, which provide further implementation details. In these designs additional logic resources, such as, error-injection and *TestApp* logic or the ILA are not included.

In these Vivado block designs, certain specific components, such as, `STARTUPEE2` and `CAPTUREE2` primitives, and the Processing System, are highlighted in purple. Similarly, in the implementations that make use of different PICDiY instances, the PICDiY modules and their output signals are highlighted with different colours (blue, green or red).

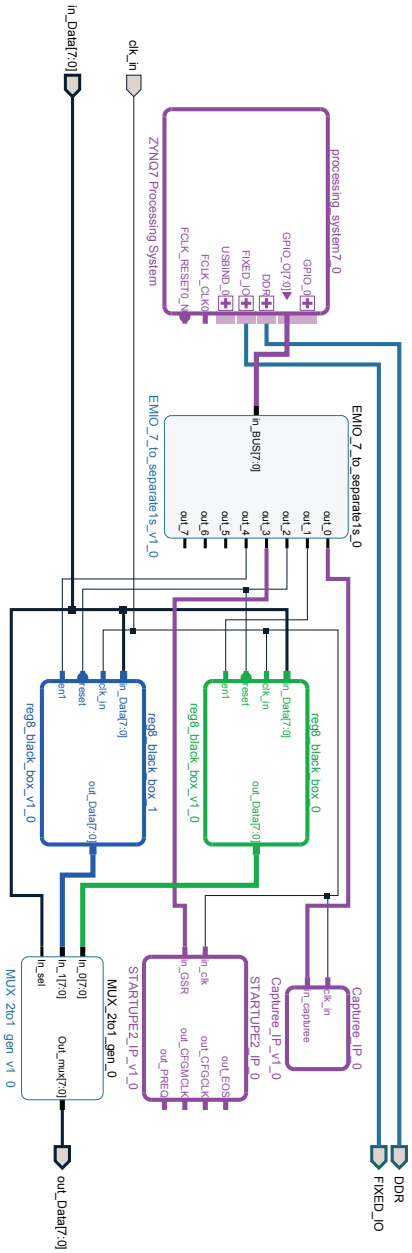


Figure A.1: Vivado block design of the validation setup for the *Approach to Manage Data of Registers with the Bitstream.*

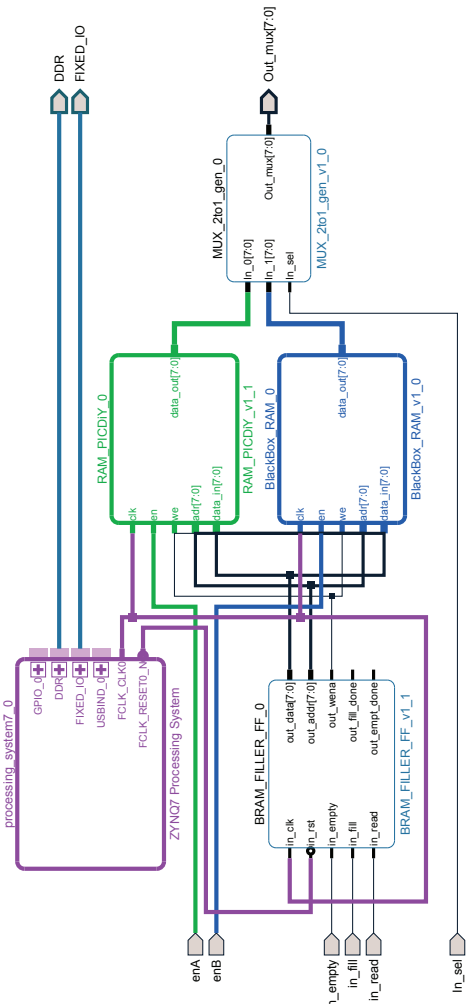


Figure A.2: Vivado block design of the validation setup for the *Approach to Extract Data From Damaged Memories Using the BBBA*.

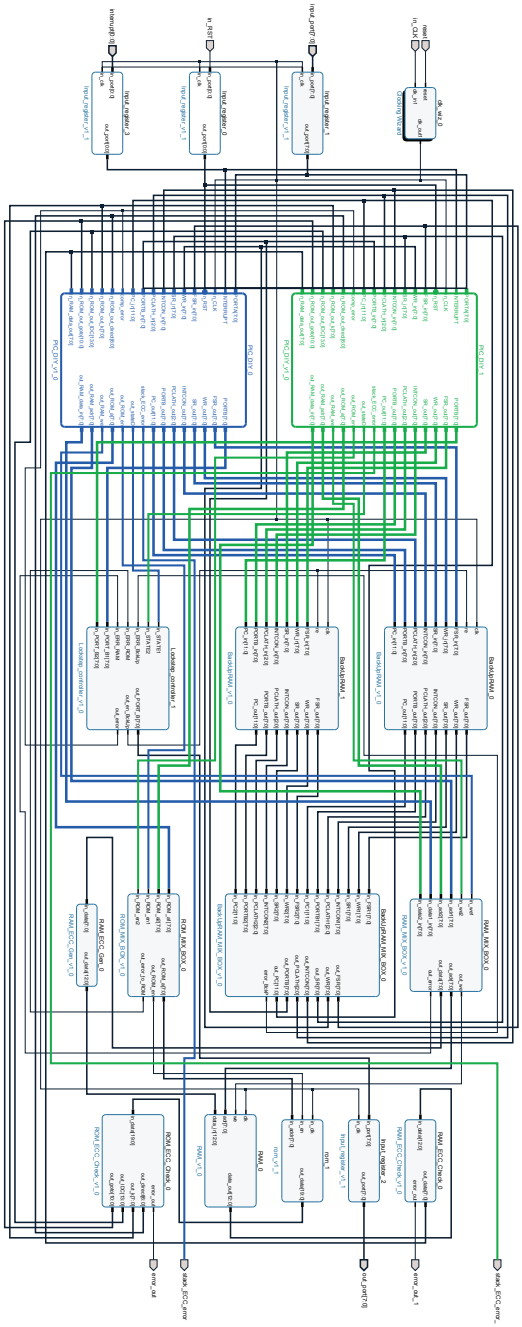


Figure A.3: Vivado block design of the validation setup for the *HW Fast Lockstep*.

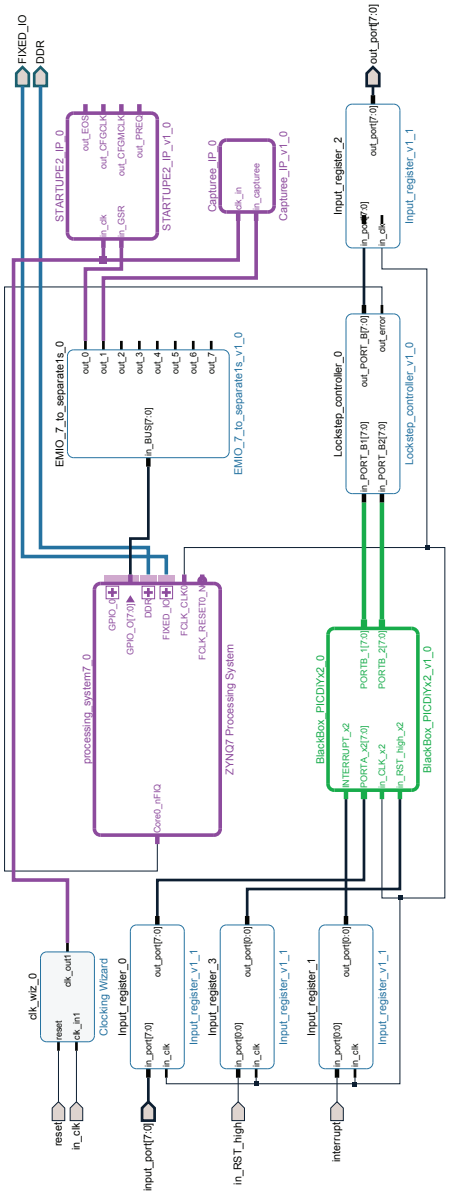


Figure A.4: Vivado block design of the validation setup for the *Bitstream Based Low Overhead Lockstep*.

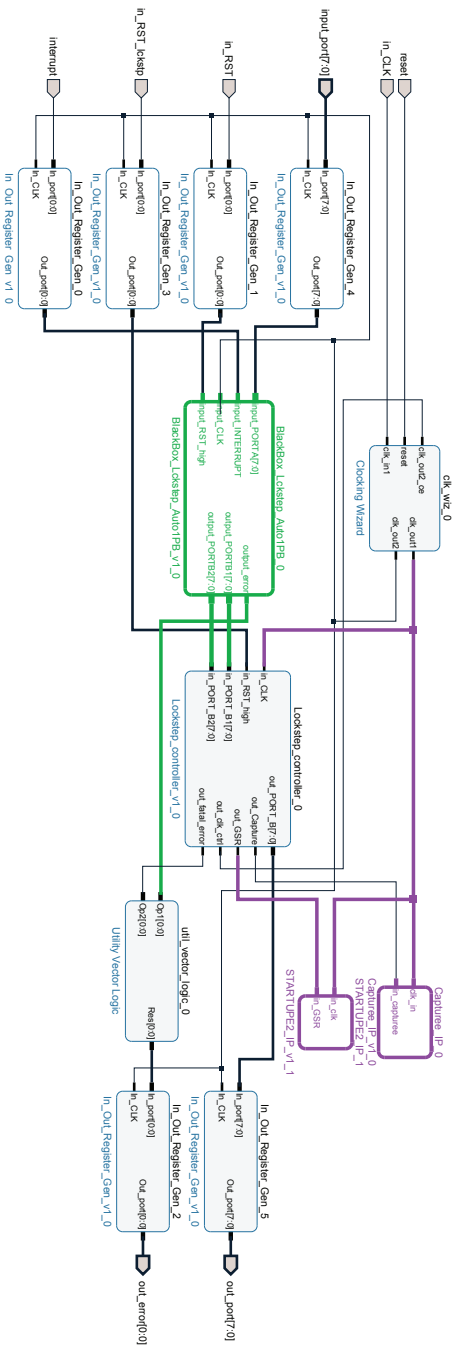


Figure A.5: Vivado block design of the validation setup for the *Bitstream Based Autonomous Lockstep*.

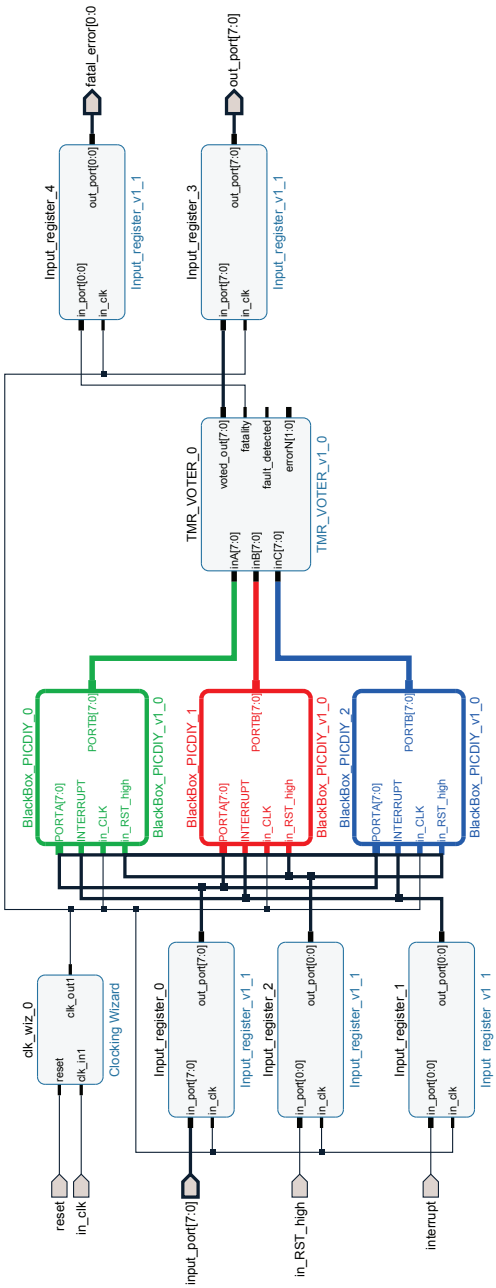


Figure A.6: Vivado block design of the validation setup for the *Reset Sync*.

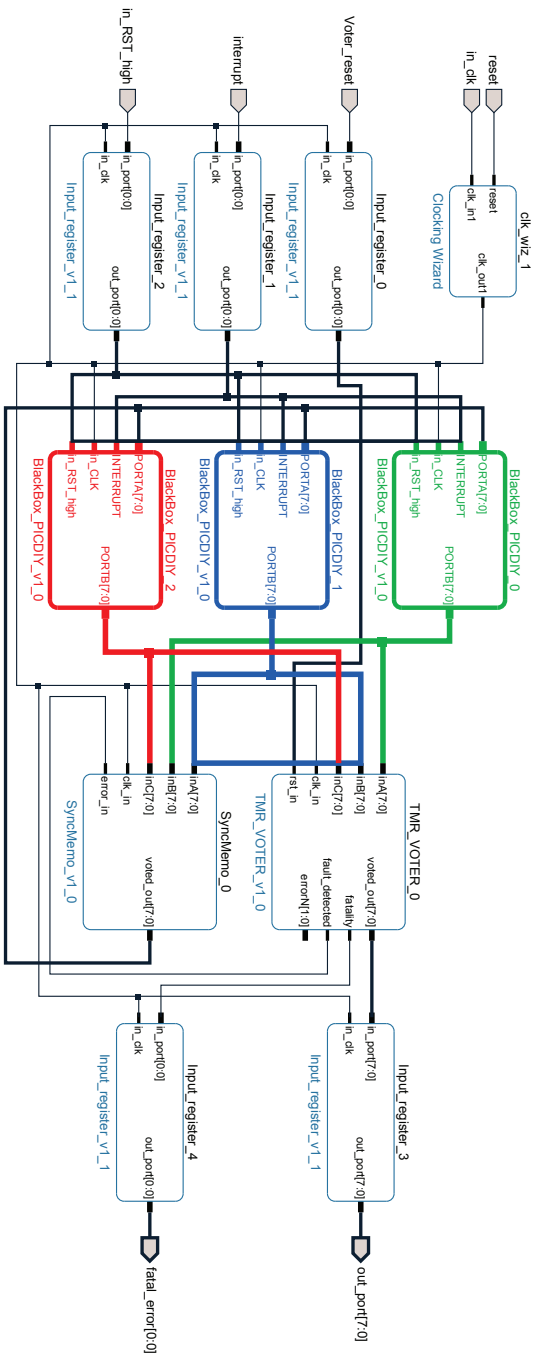


Figure A.7: Vivado block design of the validation setup for the *Force Sync*.

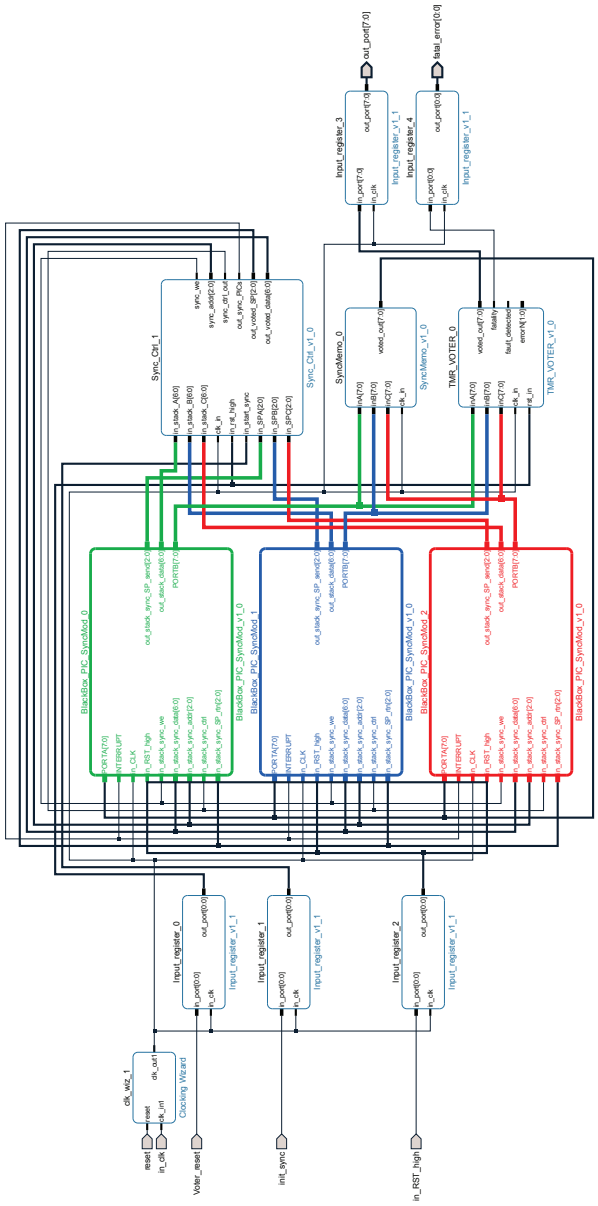


Figure A.8: Vivado block design of the validation setup for the *Interrupt Sync*.

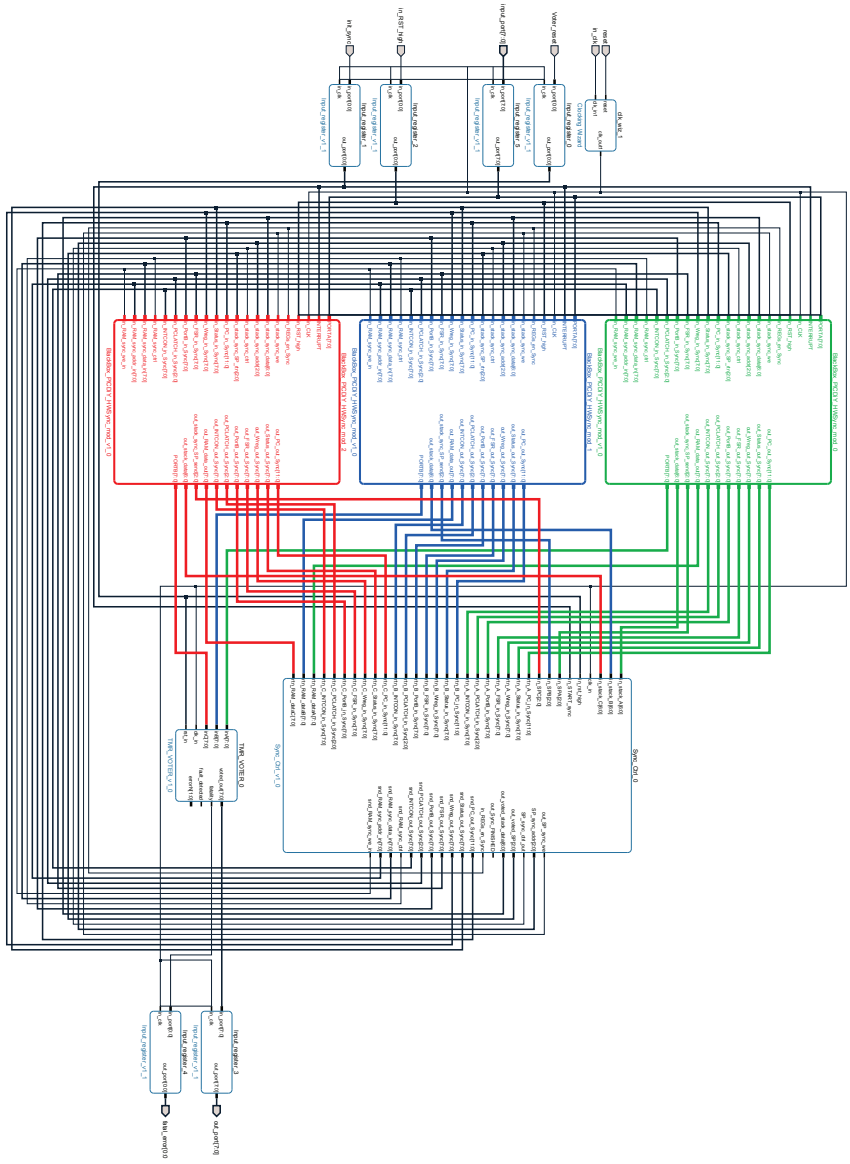


Figure A.9: Vivado block design of the validation setup for the *Hw Sync*.

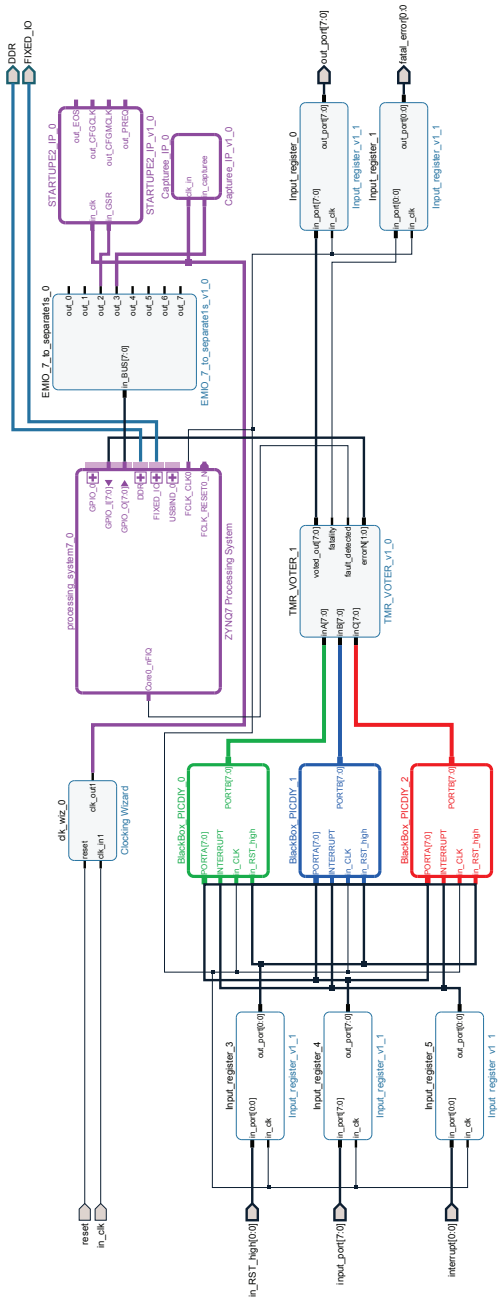


Figure A.10: Vivado block design of the validation setup for the *Bitstream Sync*.

Bibliography

[1] Xilinx Corp., "7 series FPGAs configurable logic block UG474 (v1.8)," Xilinx Documentation, http://www.xilinx.com, 2016.

[2] Xilinx Corp., "7 series FPGAs memory resources UG473 (v1.12)," Xilinx Documentation, http://www.xilinx.com, 2016.

[3] Xilinx Corp., "Vivado design suite 7 series FPGA and Zynq-7000 all programmable SoC libraries guide UG953 (v2014.4)," Xilinx Documentation, http://www.xilinx.com, 2014.

[4] Xilinx Corp., "7 series FPGAs configuration UG470 (v1.9)," Xilinx Documentation, http://www.xilinx.com, 2014.

[5] Xilinx Corp., "Integrated logic analyzer PG172 (v6.2)," Xilinx Documentation, http://www.xilinx.com, 2016.

[6] Xilinx Corp., "Zynq-7000 all programmable SoC technical reference manual UG585 (v1.9.1)," Xilinx Documentation, http://www.xilinx.com, 2014.

[7] H. Kalte and M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems," in *International Conference on Field Programmable Logic and Applications*. IEEE, 2005, pp. 223 – 228.

[8] A. Morales-Villanueva and A. Gordon-Ross, "HTR: On-chip hardware task relocation for partially reconfigurable FPGAs," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer Berlin Heidelberg, 2013, vol. 7806, pp. 185 – 196.

[9] A. Kanamaru, H. Kawai, Y. Yamaguchi, and M. Yasunaga, "Tile-based fault tolerant approach using partial reconfiguration," in *International Workshop on Applied Reconfigurable Computing*. Springer, 2009, pp. 293 – 299.

[10] R. Yao, J. Wu, M. Wang, X. Zhong, P. Zhu, and J. Liang, "State synchronization technique based on present input and healthy state for repairable TMR systems," *IEICE Electronics Express*, pp. 20 161 000 – 20 161 000, 2016.

[11] J. Azambuja, M. Altieri, J. Becker, and F. Kastensmidt, "Heta: Hybrid error-detection technique using assertions," *IEEE Transactions on Nuclear Science*, pp. 2805 – 2812, 2013.

[12] B. Tietche, O. Romain, B. Denby, and F. Dieuleveult, "FPGA-based simultaneous multichannel FM broadcast receiver for audio indexing applications in consumer electronics scenarios," *IEEE Transactions on Consumer Electronics*, pp. 1153 – 1161, 2012.

[13] C. Ttofis, C. Kyrkou, and T. Theocharides, "A low-cost real-time embedded stereo vision system for accurate disparity estimation based on guided image filtering," *IEEE Transactions on Computers*, pp. 2678 – 2693, 2016.

[14] M. Komorkiewicz, K. Turek, P. Skruch, T. Kryjak, and M. Gorgon, "FPGA-based Hardware-in-the-Loop environment using video injection concept for camera-based systems in automotive applications," in *Design and Architectures for Signal and Image Processing (DASIP)*, 2016, pp. 183 – 190.

[15] A. Melzer, F. Starzer, H. Jager, and M. Huemer, "Real-time mitigation of short-range leakage in automotive FMCW radar transceivers," *IEEE Transactions on Circuits and Systems II: Express Briefs*, pp. 1 – 5, 2016.

[16] M. Schirmer, F. Stradolini, S. Carrara, and E. Chicca, "Fpga-based approach for automatic peak detection in cyclic voltammetry," in *International Conference on Electronics, Circuits and Systems (ICECS)*, 2016, pp. 65 – 68.

[17] E. Min, Y. Jung, H. Lee, J. Jang, K. Kim, S. Joo, and K. Lee, "Development of a multipurpose gamma-ray imaging detector module with enhanced expandability," *IEEE Transactions on Nuclear Science*, pp. 1 – 7, 2017.

[18] W. Pawgasame, "Evaluation of digital codings on the soc-based software-defined radio for the military communication," in *Asian Conference on Defence Technology (ACDT)*, 2017, pp. 81 – 87.

[19] H. Irwanto, "Development of instrumentation, control and navigation (ICON) for anti tank guided missile (ATGM)," in *International Conference on Science in Information Technology (ICSITech)*, 2016, pp. 137 – 141.

[20] A. Schmidt and T. Flatley, "Radiation hardening by software techniques on FPGAs: Flight experiment evaluation and results," in *IEEE Aerospace Conference*, 2017.

[21] 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," *IEEE Transactions on Computers*, pp. 2835 – 2847, 2016.

[22] C. Stoif, M. Schoeberl, B. Liccardi, and J. Haase, "Hardware synchronization for embedded multi-core processors," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2011, pp. 2557 – 2560.

[23] B. H. Fletcher, "FPGA embedded processors: Revealing true system performance," in *Embedded Systems Conference*, 2005.

[24] A. Ben Salem, S. Ben Othman, and S. Ben Saoud, "Hard and soft-core implementation of embedded control application using RTOS," in *IEEE International Symposium on Industrial Electronics (ISIE)*, 2008, pp. 1896 – 1901.

[25] J. Tong, I. Anderson, and M. Khalid, "Soft-core processors for embedded systems," in *International Conference on Microelectronics (ICM)*, 2006, pp. 170 – 173.

[26] F. Merchant, S. Pujari, and M. Manish Patil, "Platform independent 8-bit soft-core for SoPC," in *International MultiConference of Engineers and Computer Scientists*, 2009, pp. 1541 – 1544.

[27] E. Stott, P. Sedcole, and P. Cheung, "Fault tolerant methods for reliability in FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2008, pp. 415 – 420.

[28] Y. Z. Xu, H. Puchner, A. Chatila, O. Pohland, B. Bruggeman, B. Jin, D. Radaelli, and S. Daniel, "Process impact on SRAM alpha-particle SEU performance," in *IEEE International Reliability Physics Symposium*, 2004, pp. 294 – 299.

[29] Xilinx Corp., "Device reliability report, fourth quarter 2010," Xilinx Documentation, UG116, http://www.xilinx.com, feb. 2011.

[30] Xilinx Corp., "Xilinx-5 FPGA Configuration User Guide," Xilinx Documentation, UG191, http://www.xilinx.com, aug. 2010.

[31] Xilinx Corp., "Virtex-5 Family Overview," Xilinx Documentation, DS100, http://www.xilinx.com, feb. 2009.

[32] M. Schutti, M. Pfaff, and R. Hagelauer, "VHDL design of embedded processor cores: The industry-standard microcontroller 8051 and 68HC11," in *Annual IEEE International ASIC Conference 1998*, 1998, pp. 265 – 269.

[33] E. Ayeh, K. Agbedanu, Y. Morita, O. Adamo, and P. Guturu, "FPGA implementation of an 8-bit simple processor," in *IEEE Region 5 Conference*, 2008, pp. 1 – 5.

[34] S. de Pablo, J. Cebrian, L. C. Herrero, and A. B. Rey, "A soft fixed-point digital signal processor applied in power electronics," in *FPGAworld Conference, Stockholm*. Electrum-Kista, 2005.

[35] A. Le Masle, W. Luk, and C. Moritz, "Parametrized hardware architectures for the lucas primality test," in *International Conference on Embedded Computer Systems (SAMOS)*, 2011, pp. 124 – 131.

[36] A. Gour, A. Raj, R. Behera, N. Murali, and S. Murty, "Design and development of soft-core processor based remote terminal units for nuclear reactors," in *International Conference on Field-Programmable Technology (FPT)*, 2011, pp. 1 – 4.

[37] L. Anghel, R. Velazco, S. Saleh, S. Deswaertes, and A. El Moucary, "Preliminary validation of an approach dealing with processor obsolescence," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003, pp. 493 – 500.

[38] K. Chapman, "Creating embedded microcontrollers (programmable state machines)," in *The Authoritative Journal For Programmable Logic Users*, 2004.

[39] Z. Hajduk, "An FPGA embedded microcontroller," *Microprocessors and Microsystems*, vol. 38, no. 1, pp. 1 – 8, 2014.

[40] OpenCores, "Official website," http://www.OpenCores.org, 2012.

[41] Wikipedia, "GNU lesser general public license," http://en.wikipedia.org/wiki/LGPL, 2012.

[42] Wikipedia, "BSD licenses," http://en.wikipedia.org/wiki/BSD_licenses, 2012.

[43] M. Palmer, "A comparison of 8-bit microcontrollers," in *Microchip Technology Inc.*, 2002.

[44] D. Mattsson and M. Christensson, "Evaluation of synthesizable CPU cores," Master's thesis, Chalmers University Of Technology, 2004.

[45] D. Gomez-Prado and M. Ciesielski, "Embedded microcontrollers and FPGAs soft-cores," *Electronica UNMSM*, 2006.

[46] P. Borisonv and V. Stoianova, "Implementation of soft-core processors in FPGAs," in *Unitech International Scientific Conference*, 2007.

[47] D. Gallegos, B. Welch, J. Jarosz, V. Houten, and M. Learn, "Soft-core processor study for node-based architectures," Sandia National Laboratories, Tech. Rep., 2008.

[48] J. Nade and R. Sarwadnya, "The soft core processors: A review," *International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering*, pp. 197 – 203, 2015.

[49] F. Schmidt, "Fault tolerant design implementation on radiation hardened by design SRAM-based FPGAs," Ph.D. dissertation, Massachusetts Institute of Technology, 2013.

[50] S. Tanoue, T. Ishida, Y. Ichinomiya, M. Amagasaki, M. Kuga, and T. Sueyoshi, "A novel states recovery technique for the TMR softcore processor," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 543 – 546.

[51] D. Sheldon, R. Kumar, R. Lysecky, F. Vahid, and D. Tullsen, "Application-specific customization of parameterized FPGA soft-core processors," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2006, pp. 261 – 268.

[52] A. Sari, M. Psarakis, and D. Gizopoulos, "Combining checkpointing and scrubbing in FPGA-based real-time systems," in *IEEE VLSI Test Symposium (VTS)*, 2013, pp. 1 – 6.

[53] S. Rezgui, G. Swift, K. Somervill, J. George, C. Carmichael, and G. Allen, "Complex upset mitigation applied to a re-configurable embedded processor," *IEEE Transactions on Nuclear Science*, pp. 2468 – 2474, 2005.

[54] C. Hong, K. Benkrid, X. Iturbe, and A. Ebrahim, "Design and implementation of fault-tolerant soft processors on FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 683 – 686.

[55] M. Learn, "Evaluation of soft-core processors on a Xilinx Virtex-5 field programmable gate array," Sandia National Laboratories, Tech. Rep., 2011.

[56] M. Hubner, D. Gohringer, J. Noguera, and J. Becker, "Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx

FPGAs," in *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1 – 8.

[57] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, "FPGA partial reconfiguration via configuration scrubbing," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 99 – 104.

[58] S. M. Borawake and P. G. Chilveri, "Implementation of wireless sensor network using microblaze and picoblaze processors," in *International Conference on Communication Systems and Network Technologies*, 2014, pp. 1059 – 1064.

[59] Y. Ichinomiya, S. Tanoue, M. Amagasaki, M. Iida, M. Kuga, and T. Sueyoshi, "Improving the robustness of a softcore processor against SEUs by using TMR and partial reconfiguration," in *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2010, pp. 47 – 54.

[60] H. Pham, S. Pillement, and S. Piestrak, "Low overhead fault-tolerance technique for dynamically reconfigurable softcore processor," *IEEE Transactions on Computers*, pp. 1179 – 1192, 2013.

[61] A. Morales-Villanueva and A. Gordon-Ross, "On-chip context save and restore of hardware tasks on partially reconfigurable FPGAs," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2013, pp. 61 – 64.

[62] F. Veljkovic, J. Mora, T. Riesgo, L. Berrojo, R. Regada, A. Sanchez, and E. De la Torre, "Prospection of reconfiguration capabilities using space qualified SRAM-based FPGAs for a satellite communications application," in *AIAA International Communications Satellite Systems Conference*, 2013.

[63] A. Ebrahim, K. Benkrid, X. Iturbe, and C. Hong, "A novel high-performance fault-tolerant ICAP controller," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2012, pp. 259 – 263.

[64] D. Antonio-Torres, D. Villanueva-Perez, E. Sanchez-Canepa, N. Segura-Meraz, D. Garcia-Garcia, D. Conchouso-Gonzalez, J. Miranda-Vergara, J. Gonzalez-Herrera, A.-M. de Ita, B. Hernandez-Rodriguez, R.-E. de los Monteros, F. Garcia-Chavez, V. Tellez-Rojas, and A. Bautista-Hernandez, "A picoblaze-based embedded system for monitoring applications," in *International Conference on Electrical, Communications, and Computers (CONIELECOMP)*, 2009, pp. 173 – 177.

[65] F. Restrepo-Calle, A. Martinez-alvarez, H. Guzman-Miranday, F. Palomoy, and S. Cuenca-Asensi, "Application-driven co-design of fault-tolerant industrial systems," in *IEEE International Symposium on Industrial Electronics (ISIE)*, 2010, pp. 2005 – 2010.

[66] I. Safarulla and K. Manilal, "Design of soft error tolerance technique for FPGA based soft core processors," in *International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*, 2014, pp. 1036 – 1040.

[67] C. Lung, S. Sabou, and A. Buchman, "Emergency radio communication network controller implemented in FPGA," in *IEEE International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 2014, pp. 193 – 196.

[68] J. Mathew and R. Dhayabarani, "Fault tolerance technique for dynamically reconfigurable processor," *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering*, pp. 6656 – 6663.

[69] C. Lung, S. Sabou, and A. Buchman, "Modelling and implementation of intelligent sensor networks with applications in emergency situations management," in *IEEE International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 2015, pp. 315 – 318.

[70] J. Tarrillo, J. Tonfat, L. Tambara, F. L. Kastensmidt, and R. Reis, "Multiple fault injection platform for SRAM-based FPGA based on ground-level radiation experiments," in *Latin-American Test Symposium (LATS)*, 2015, pp. 1 – 6.

[71] J. Adair, "Pushing picoblaze - part1," *Xilinx.com - PicoBlaze User Resources*, 2005.

[72] M. Niknahad, O. Sander, and J. Becker, "QFDR-an integration of quadded logic for modern FPGAs to tolerate high radiation effect rates," in *European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, 2011, pp. 119 – 122.

[73] U. Legat, A. Biasizzo, and F. Novak, "SEU recovery mechanism for SRAM-based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 59, no. 5, pp. 2562 – 2571, 2012.

[74] K. Chapman, "SEU strategies for Virtex-5," in *Application Note: Virtex-5 Family*, 2009.

[75] B. F. Dutton and C. E. Stroud, "Single event upset detection and correction

in Virtex-4 and virtex-5 FPGAs," in *ISCA International Conference on Computers and Their Applications*, 2009, pp. 57 – 62.

[76] L. Claudiu, S. Sebastian, and B. Cristian, "Smart sensor implemented with picoblaze multi-processors technology," in *IEEE International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 2012, pp. 241 – 245.

[77] F. Plavec, B. Fort, Z. Vranesic, and S. Brown, "Experiences with soft-core processor design," in *International Parallel and Distributed Processing Symposium*, 2005, pp. 4 – pp.

[78] J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC v8 architecture," in *International Conference on Dependable Systems and Networks (DSN)*, 2002, pp. 409 – 415.

[79] A. Jordan, C. Hafer, J. Mabra, S. Griffith, J. Nagy, M. Lahey, and D. Harris, "SEU data and fault tolerance analysis of a Leon 3FT processor," in *European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, 2009, pp. 617 – 619.

[80] M. Reorda, M. Violante, C. Meinhardt, and R. Reis, "A low-cost SEE mitigation solution for soft-processors embedded in systems on pogrammable chips," in *Design, Automation Test in Europe Conference Exhibition*, 2009, pp. 352 – 357.

[81] M. Amin, A. Ramazani, F. Monteiro, C. Diou, and A. Dandache, "A self-checking hardware journal for a fault-tolerant processor architecture," *Int. J. Reconfig. Comput.*, pp. 11 – 11, 2011.

[82] J. Becker, A. Thomas, M. Vorbach, and V. Baumgarten, "An industrial/academic configurable system-on-chip project (CSoC): Coarse-grain XXP-/Leon-based architecture integration." IEEE Computer Society, 2003, pp. 1120 – 1121.

[83] M. Reorda, M. Violante, C. Meinhardt, and R. Reis, "An on-board data-handling computer for deep-space exploration built using commercial-off-the-shelf SRAM-based FPGAs," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2009, pp. 254 – 262.

[84] M. Learn, "Evaluation of the Leon3 soft-core processor within a Xilinx radiation-hardened field-programmable gate array," Sandia National Laboratories, Tech. Rep., 2012.

[85] R. Velazco, W. Mansour, F. Pancher, G. Marques-Costa, D. Sohier, and A. Bui, "Improving SEU fault tolerance capabilities of a self-converging

algorithm," in *European Conference on Radiation and its Effects on Components and Systems (RADECS)*, 2011, pp. 138 – 143.

[86] A. Morillo, A. Astarloa, J. Lazaro, U. Bidarte, and J. Jimenez, "Known-blocking. Synchronization method for reliable processor using TMR & DPR in SRAM FPGAs," in *Southern Conference on Programmable Logic (SPL)*, 2011, pp. 57 – 62.

[87] M. Ebrahimi, S. Miremadi, H. Asadi, and M. Fazeli, "Low-cost scan-chain-based technique to recover multiple errors in TMR systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 8, pp. 1454 – 1468, 2013.

[88] L. Sterpone and A. Ullah, "On the optimal reconfiguration times for TMR circuits on SRAM based FPGAs," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2013, pp. 9 – 14.

[89] N. H. Rollins and M. J. Wirthlin, "Reliability of a softcore processor in a commercial SRAM-based FPGA," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 171 – 174.

[90] A. Ioannis and M. Dimitrios, "Replacing the SPARC-based core of the Leon3 HDL microprocessor model with a MIPS-based core," in *SECE Conference*, 2010.

[91] R. Velazco, G. Foucard, F. Pancher, W. Mansour, G. Marques-Costa, D. Sohier, and A. Bui, "Robustness with respect to SEUs of a self-converging algorithm," in *Latin American Test Workshop (LATW)*, 2011, pp. 1 – 5.

[92] J. Becker and A. Thomas, "Scalable processor instruction set extension," *IEEE Design Test of Computers*, 2005.

[93] A. Mohammadi, M. Ebrahimi, A. Ejlali, and S. Miremadi, "SCFIT: A FPGA-Basedased Fault Injection Technique for SEU Fault Model," in *Design Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 586 – 589.

[94] N. H. Rollins and M. J. Wirthlin, "Software fault-tolerant techniques for softcore processors in commercial SRAM-based FPGAs," *ARCS Workshops*, 2011.

[95] A. Sari and M. Psarakis, "Scrubbing-based SEU mitigation approach for systems-on-programmable-chips," in *International Conference on Field-Programmable Technology (FPT)*, 2011, pp. 1 – 8.

[96] W. Mansour and R. Velazco, "Seu fault-injection in VHDL-based processors: A case study," *Journal of Electronic Testing*, pp. 87 – 94, 2013.

[97] P. Huerta Pellitero, "Sistemas de multiprocesamiento simetrico sobre FPGA," Ph.D. dissertation, Universidad Rey Juan Carlos, 2009.

[98] S. Morioka, "VHDL implementation of the PIC16F84 in FPGA," in *Transistor Gijutsu Magazine*, 1999.

[99] S. Yuan, C. P., and S. Liao, "The power stability of FPGA-based microcontroller design and measurement," in *Asia-Pacific Symposium on Electromagnetic Compatibility (APEMC)*, 2010, pp. 1096 – 1099.

[100] T. Lobo, S. Pinto, V. Silva, S. Lopes, J. Cabral, A. Tavares, S. Yoowattana, W. Sritriratanarak, and M. Ekpanyapong, "LP805X: A customizable and low power 8051 soft core for FPGA applications," in *IEEE International Symposium on Industrial Electronics (ISIE)*, 2013, pp. 1 – 7.

[101] G. Fernandez, *Conceptos Basicos de Arquitectura y Sistemas Operativos.* Ed. Syserso., 1998.

[102] P. Leong, P. Tsang, and T. Lee, "A FPGA based forth microprocessor," in *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998, pp. 254 – 255.

[103] H. Y. Cheah, S. Fahmy, and D. Maskell, "iDEA: A DSP block based FPGA soft processor," in *International Conference on Field-Programmable Technology (FPT)*, 2012, pp. 151 – 158.

[104] P. Gaillardon, M. Ben-Jamaa, G. Beneventi, F. Clermidy, and L. Perniola, "Emerging memory technologies for reconfigurable routing in FPGA architecture," in *IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2010, pp. 62 – 65.

[105] K. Huang, Y. Ha, R. Zhao, A. Kumar, and Y. Lian, "A low active leakage and high reliability phase change memory (PCM) based non-volatile FPGA storage element," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 2605 – 2613, 2014.

[106] M. Klein and S. Kol, "Leveraging power leadership at 28 nm with Xilinx 7 series FPGAs," 2013.

[107] J. Gomez-Cornejo, A. Zuloaga, U. Kretzschmar, U. Bidarte, and J. Jimenez, "Fast context reloading lockstep approach for SEUs mitigation in a FPGA soft core processor," in *Conference of the IEEE Industrial Electronics Society, IECON*, 2013, pp. 2261 – 2266.

[108] Xilinx Corp., "Vivado design suite user guide. partial reconfiguration UG909 (v2015.1)," Xilinx Documentation, http://www.xilinx.com, 2015.

[109] J. Correa and K. Ackermann, "Leveraging partial dynamic reconfiguration on Zynq SoC FPGAs," in *International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2014, pp. 1 – 6.

[110] S. Liu, N. Pittman, and A. Forin, "Energy reduction with run-time partial reconfiguration," Microsoft, Tech. Rep., 2009.

[111] S. Liu, R. Pittman, A. Form, and J. Gaudiot, "On energy efficiency of reconfigurable systems with run-time oartial reconfiguration," in *IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2010, pp. 265 – 272.

[112] I. Yoon, H. Joung, and J. Lee, "Zynq-based reconfigurable system for real-time edge detection of noisy video sequences," *Journal of Sensors*, vol. 2016, 2016.

[113] M. Al Kadi, P. Rudolph, D. Gohringer, and M. Hubner, "Dynamic and partial reconfiguration of Zynq 7000 under Linux," in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2013, pp. 1 – 5.

[114] K. Vipin and S. Fahmy, "ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq," *Embedded Systems Letters, IEEE*, vol. 6, no. 3, pp. 41 – 44, 2014.

[115] A. Megacz, "A library and platform for FPGA bitstream manipulation," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2007, pp. 45 – 54.

[116] F. Benz, A. Seffrin, and S. Huss, "Bil: A tool-chain for bitstream reverse-engineering," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 735 – 738.

[117] R. Chakraborty, I. Saha, A. Palchaudhuri, and G. Naik, "Hardware trojan insertion by direct modification of FPGA configuration bitstream," *IEEE Design Test*, vol. 30, no. 2, pp. 45 – 54, 2013.

[118] R. Soni, N. Steiner, and M. French, "Open-source bitstream generation," in *EEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2013, pp. 105 – 112.

[119] C. Morford, "BitMaT - bitstream manipulation tool for Xilinx FPGAs,"

Master's thesis, Faculty of the Virginia Polytechnic Institute and State University, 2005.

[120] Xilinx Corp., "Partial reconfiguration user guide UG702 (v14.1)," Xilinx Documentation, http://www.xilinx.com, 2012.

[121] Xilinx Corp., "Data2MEM. user guide UG658 (v2012.4)," Xilinx Documentation, http://www.xilinx.com, 2012.

[122] D. L. S. Guccione and P. Sundararajan, "JBit: Java based interface for reconfigurable computing," in *Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999, pp. 28 – 30.

[123] M. Koester, H. Kalte, and M. Porrmann, "Relocation and defragmentation for heterogeneous reconfigurable systems," in *ERSA*, 2006, pp. 70 – 76.

[124] A. Morales-Villanueva, R. Kumar, and A. Gordon-Ross, "Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable FPGAs," in *Design, Automation & Test in Europe (DATE)*. IEEE, 2016, pp. 1505 – 1508.

[125] J. Srour and J. McGarrity, "Radiation effects on microelectronics in space," *Proceedings of the IEEE*, pp. 1443 – 1469, 1988.

[126] R. Ladbury, "Radiation hardening at the system level," in *IEEE NSREC Short Course*, 2007, pp. 1 – 94.

[127] J. Wang, "Radiation effects in FPGAs," in *Workshop on Electronics for LHC Experiments*, 2003.

[128] C. Färber, U. Uwer, D. Wiedner, B. Leverington, and R. Ekelhof, "Radiation tolerance tests of SRAM-based FPGAs for the potential usage in the readout electronics for the lhcb experiment," *Journal of Instrumentation*, p. C02028, 2014.

[129] S. Clark, K. Avery, and R. Parker, "TID and SEE testing results of altera cyclone field programmable gate array," in *IEEE Radiation Effects Data Workshop*, 2004, pp. 88 – 90.

[130] N. Rezzak, J. J. Wang, D. Dsilva, and N. Jat, "TID and SEE characterization of microsemi's 4th generation radiation tolerant RTG4 flash-based FPGA," in *IEEE Radiation Effects Data Workshop (REDW)*, 2015, pp. 1 – 6.

[131] G. Allen, G. Madias, E. Miller, and G. Swift, "Recent single event effects re-

sults in advanced reconfigurable field programmable gate arrays," in *IEEE Radiation Effects Data Workshop (REDW)*, 2011, pp. 1 – 6.

[132] G. R. Allen and G. M. Swift, "Single event effects test results for advanced field programmable gate arrays," in *IEEE Radiation Effects Data Workshop*, 2006, pp. 115 – 120.

[133] P. Dodd and L. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Transactions on Nuclear Science*, pp. 583 – 602, 2003.

[134] H. Wei, W. Yueke, X. Kefei, and D. Wei, "SEE vulnerability bit analysis method for switch matrix of SRAM-based FPGA circuits," in *IEEE International Conference on Mechatronics and Automation*, 2016, pp. 2355 – 2359.

[135] J. Nunes, J. Cunha, and M. Zenha-Rela, "On the effects of cumulative SEUs in FPGA-based systems," in *European Dependable Computing Conference (EDCC)*, 2016, pp. 89 – 96.

[136] C. Du, X.and He, S. Liu, Y. Zhang, Y. Li, and W. Yang, "Measurement of single event effects induced by alpha particles in the Xilinx Zynq-7010 System-on-Chip," *Journal of Nuclear Science and Technology*, pp. 1 – 6, 2016.

[137] J. Fu and C. Zhang, "The fault-tolerant design in space information processing system based on COTS," in *International Workshop on Computer Science and Engineering (WCSE)*, 2009, pp. 568 – 571.

[138] D. Binder, E. C. Smith, and A. B. Holman, "Satellite anomalies from galactic cosmic rays," *IEEE Transactions on Nuclear Science*, pp. 2675 – 2680, 1975.

[139] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam, "Mitigation of radiation effects in SRAM-based FPGAs for space applications," *ACM Computing Surveys (CSUR)*, 2015.

[140] M. Violante, C. Meinhardt, R. Reis, and M. Reorda, "A low-cost solution for deploying processor cores in harsh environments," *IEEE Transactions on Industrial Electronics*, pp. 2617 – 2626, 2011.

[141] X. Iturbe, D. Keymeulen, P. Yiu, D. Berisford, R. Carlson, K. Hand, and E. Ozer, "On the use of system-on-chip technology in next-generation instruments avionics for space exploration," in *IEEE International Conference on Very Large Scale Integration, VLSI-SoC*. Springer International Publishing, 2016, pp. 1 – 22.

[142] P. Adell, G. Allen, G. Swift, and S. McClure, "Assessing and mitigating radiation effects in Xilinx SRAM FPGAs," in *European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, 2008, pp. 418 – 24, radiation effect mitigation;Xilinx SRAM;FPGA;.

[143] E. Normand, "Single event upset at ground level," *IEEE Transactions on Nuclear Science*, pp. 2742 – 2750, 1996.

[144] J. T. Wallmark and S. M. Marcus, "Minimum size and maximum packing density of nonredundant semiconductor devices," *Proceedings of the IRE*, vol. 50, no. 3, pp. 286 – 298, 1962.

[145] T. May and M. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Transactions on Electron Devices*, pp. 2 – 9, 1979.

[146] Xilinx Corp., "Ultrascale architecture and product overview," Xilinx Documentation, http://www.xilinx.com, 2016.

[147] I. Corp., "Stratix 10 device datasheet," Altera Documentation, http://www.altera.com, 2016.

[148] M. McCormack, "Trade study and application of symbiotic software and hardware fault-tolerance on a microcontroller-based avionics system," Ph.D. dissertation, Massachusetts Institute of Technology, 2011.

[149] T. Phillips, "Cosmic rays hit space age high," NASA, Tech. Rep., 2009.

[150] D. Matthiä, M. Meier, and G. Reitz, "Numerical calculation of the radiation exposure from galactic cosmic rays at aviation altitudes with the PANDOCA core model," *Space Weather*, vol. 12, no. 3, pp. 161 – 171, 2014.

[151] D. White, "Considerations surrounding single event effects in FPGAs, ASICs, and processors," Xilinx Corp., Tech. Rep., 2012.

[152] "Introduction to single-event upsets," Altera Documentation, http://www.altera.com, Altera, Tech. Rep., 2013.

[153] 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*, vol. 52, no. 6, pp. 2455 – 2461, 2005.

[154] "Understanding single event functional interrupts in FPGA designs," Altera Documentation, http://www.altera.com, Altera, Tech. Rep., 2013.

[155] L. Entrena, A. Lindoso, M. Valderas, M. Portela, and C. Ongil, "Analysis of SET effects in a PIC microprocessor for selective hardening," *IEEE Transactions on Nuclear Science*, vol. 58, no. 3, pp. 1078 – 1085, 2011.

[156] A. Aloisio, V. Bocci, R. Giordano, V. Izzo, L. Sterpone, and M. Violante, "Power consumption versus configuration SEUs in Xilinx Virtex-5 FPGAs," 2013, pp. 1 – 1.

[157] R. Le, "Soft error mitigation using prioritized essential bits," Xilinx Documentation, http://www.xilinx.com, 2012.

[158] M. Ceschia, M. Violante, M. Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, and A. Candelori, "Identification and classification of single-event upsets in the configuration memory of SRAM-based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 50, no. 6, pp. 2088 – 2094, 2003.

[159] M. Bellato, P. Bernardi, D. Bortolato, A. Candelori, M. Ceschia, A. Paccagnella, M. Rebaudengo, M. Reorda, M. Violante, and P. Zambolin, "Evaluating the effects of SEUs affecting the configuration memory of an SRAM-based FPGA," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, 2004, pp. 584 – 589.

[160] W. Vigrass, "Calculation of semiconductor failure rates," *Harris Semiconductor*, 2010.

[161] B. Rahbaran and A. Steininger, "Is asynchronous logic more robust than synchronous logic?" *IEEE Transactions on Dependable and Secure Computing*, pp. 282 – 294, 2009.

[162] H. Jahanirad, K. Mohammadi, and P. Attarsharghi, "Single fault reliability analysis in FPGA implemented circuits," in *International Symposium on Quality Electronic Design (ISQED)*, 2012, pp. 49 – 56.

[163] G. Asadi and M. Tahoori, "Soft error rate estimation and mitigation for SRAM-based FPGAs," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2005, pp. 149 – 160.

[164] A. Souari, C. Thibeault, Y. Blaquiere, and R. Velazco, "Optimization of SEU emulation on SRAM FPGAs based on sensitiveness analysis," in *IEEE International On-Line Testing Symposium (IOLTS)*, 2015, pp. 36 – 39.

[165] I. Villata, U. Bidarte, U. Kretzschmar, A. Astarloa, and J. Lazaro, "Fast and accurate SEU-tolerance characterization method for Zynq SoCs," in *24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1 – 4.

[166] A. Lesea, S. Drimer, J. Fabula, C. Carmichael, and P. Alfke, "The Rosetta experiment: Atmospheric soft error rate testing in differing technology

FPGAs," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 317 – 328, 2005.

[167] A. Lesea, "Continuing experiments of atmospheric neutron effects on deep submicron integrated circuits," Xilinx Documentation, http://www.xilinx.com, 2013.

[168] "Continuing experiments of atmospheric neutron effects on deep submicron integrated circuits. WP286 (v2.0)," Xilinx Documentation, http://www.xilinx.com, Xilinx Corp., Tech. Rep., 2016.

[169] Xilinx Corp., "Radiation-hardened, space-grade virtex-5qv family overview ds192(v1.4)," Xilinx Documentation, http://www.xilinx.com, 2014.

[170] Microsemi Corp., "RTG4 FPGA, DS0131 datasheet," Microsemi Documentation, http://www.microsemi.com, 2016.

[171] Microsemi Corp., "RTAX-S/SL and RTAX-DSP Radiation-Tolerant FPGAs, Datasheet," Microsemi Documentation, http://www.microsemi.com, 2015.

[172] Actel Corp., "HiRel SX-A Family FPGAs, Datasheet," Actel Documentation, http://www.actel.com, 2006.

[173] Actel Corp., "Radiation-Tolerant ProASIC3 FPGAs Radiation Effects," Actel Documentation, http://www.actel.com, 2010.

[174] J. Hussein and G. Swift, "Mitigating single-event upsets," Xilinx Documentation, http://www.xilinx.com, 2015.

[175] S. Habinc, "Lessons learned from FPGA developments," Gaiser Research, Tech. Rep., 2002.

[176] S. Habinc, "Suitability of reprogrammable FPGAs in space applications," *Gaisler Research," Feasibility Report*, 2002.

[177] W. J. Huang and E. J. McCluskey, "A memory coherence technique for online transient error recovery of fpga configurations," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2001, pp. 183 – 192.

[178] M. Berg, C. Poivey, D. Petrick, D. Espinosa, A. Lesea, K. LaBel, M. Friendlich, H. Kim, and A. Phan, "Effectiveness of internal versus external seu scrubbing mitigation strategies in a Xilinx FPGA: Design, test, and analysis," *IEEE Transactions on Nuclear Science*, pp. 2259 – 2266, 2008.

[179] N. Imran, R. A. Ashraf, and R. DeMara, "On-demand fault scrubbing using adaptive modular redundancy," in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2013, p. 1.

[180] M. Garvie and A. Thompson, "Scrubbing away transients and jiggling around the permanent: Long survival of FPGA systems through evolutionary self-repair," in *IEEE International On-Line Testing Symposium*, 2004, pp. 155 – 160.

[181] C. Carmichael and C. Wei Tseng, "Correcting single-event upsets in Virtex-4 FPGA configuration memory," Xilinx Documentation, http://www.xilinx.com, Xilinx Corp., Tech. Rep., 2009, xAPP1088 (v1.0).

[182] J. Heiner, N. Collins, and M. Wirthlin, "Fault tolerant ICAP controller for high-reliable internal scrubbing," in *IEEE Aerospace Conference*, 2008, pp. 1 – 10.

[183] M. Kumar, D. Digdarsini, N. Misra, and T. Ram, "SEU mitigation of rad-tolerant Xilinx FPGA using external scrubbing for geostationary mission," in *India Conference (INDICON), 2016 IEEE Annual*. IEEE, 2016, pp. 1 – 6.

[184] N. Rollins, M. Fuller, and M. Wirthlin, "A comparison of fault-tolerant memories in SRAM-based FPGAs," in *IEEE Aerospace Conference*, 2010, pp. 1 – 12.

[185] N. H. Rollins, "Hardware and software fault-tolerance of softcore processors implemented in SRAM-based FPGAs," Ph.D. dissertation, Brigham Young University, 2012.

[186] Xilinx Corp., "Soft error mitigation controller. PG036 (v4.1)," Xilinx Documentation, http://www.xilinx.com, 2014.

[187] G. Vera, S. Ardalan, X. Yao, and K. Avery, "Fast local scrubbing for field-programmable gate array's configuration memory," *Journal of Aerospace Information Systems*, pp. 144 – 153, 2013.

[188] G. Asadi and M. Tahoori, "Soft error mitigation for SRAM-based FPGAs," in *IEEE VLSI Test Symposium*, 2005, pp. 207 – 212.

[189] A. Vavousis, A. Apostolakis, and M. Psarakis, "A fault tolerant approach for FPGA embedded processors based on runtime partial reconfiguration," *Journal of Electronic Testing: Theory and Applications (JETTA)*, pp. 1 – 19, 2013.

[190] E. Kamanu, P. Reddy, K. Hsu, and M. Lukowaik, "A new architecture

for single-event detection and reconfiguration of SRAM-based FPGAs," in *IEEE High Assurance Systems Engineering Symposium (HASE)*, 2007, pp. 291 – 298.

[191] X. Iturbe, M. Azkarate, I. Martinez, J. Perez, and A. Astarloa, "A novel SEU, MBU and SHE handling strategy for Xilinx Virtex-4 FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 569 – 573.

[192] M. Palmer, P. Zicari, F. Frustaci, S. Perri, and P. Corsonello, "An efficient and low-cost design methodology to improve SRAM-based FPGA robustness in space and avionics applications," in *International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, 2009, pp. 74 – 84.

[193] N. Avirneni and A. Somani, "Low overhead soft error mitigation techniques for high-performance and aggressive designs," *IEEE Transactions on Computers*, pp. 488 – 501, 2012.

[194] Z. Qian, Y. Ichinomiya, M. Amagasaki, M. Iida, and T. Sueyoshi, "A novel soft error detection and correction circuit for embedded reconfigurable systems," *IEEE Embedded Systems Letters*, pp. 89 – 92, 2011.

[195] Y. Ichinomiya, M. Amagasaki, M. Iida, M. Kuga, and T. Sueyoshi, "Improving the soft-error tolerability of a soft-core processor on an FPGA using triple modular redundancy and partial reconfiguration," *Journal of Next Generation Information Technology*, pp. 35 – 48, 2011.

[196] A. Fort, M. Mugnaini, V. Vignoli, V. Gaggii, and M. Pieralli, "Fault tolerant design of a field data modular readout architecture for railway applications," *Reliability Engineering & System Safety*, pp. 456 – 462, 2015.

[197] K. Morgan, D. McMurtrey, B. Pratt, and M. Wirthlin, "A comparison of TMR with alternative fault-tolerant design techniques for FPGAs," *IEEE Transactions on Nuclear Science*, pp. 2065 – 2072, 2007.

[198] U. Kretzschmar, A. Astarloa, J. Lazaro, and G. M., "Robustness of different TMR granularities in shared wishbone architectures on SRAM FPGA," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2012.

[199] M. Niknahad, O. Sander, and J. Becker, "Fine grain fault tolerance- a key to high reliability for FPGAs in space," in *IEEE Aerospace Conference*, 2012, pp. 1 – 10.

[200] F. Lahrach, A. Abdaoui, A. Doumar, and E. Chatelet, "A novel SRAM-based FPGA architecture for defect and fault tolerance of configurable logic blocks," in *IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2010, pp. 305 – 308.

[201] M. Niknahad, O. Sander, and J. Becker, "FGTMR - fine grain redundancy method for reconfigurable architectures under high failure rates," in *North-East Asia Symposium on Nano, Information Technology and Reliability (NASNIT)*, 2011, pp. 186 – 191.

[202] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin, "Improving FPGA design robustness with partial TMR," in *Annual IEEE International Reliability Physics Symposium Proceedings*, 2006, pp. 226 – 232.

[203] X. Wang, "Partitioning triple modular redundancy for single event upset mitigation in FPGA," in *International Conference on E-Product E-Service and E-Entertainment (ICEEE)*, 2010, pp. 1 – 4.

[204] J. Johnson and M. Wirthlin, "Voter insertion techniques for fault tolerant FPGA design," *NSF Center for High Performance Reconfigurable Computing (CHREC)*, 2009.

[205] F. Kastensmidt, L. Sterpone, L. Carro, and M. Reorda, "On the optimal design of triple modular redundancy logic for SRAM-based FPGAs," in *Design, Automation and Test in Europe*, 2005, pp. 1290 – 1295.

[206] C. Bolchini, D. Quarta, and M. Santambrogio, "SEU mitigation for SRAM-based FPGAs through dynamic partial reconfiguration," in *ACM Great Lakes symposium on VLSI*, 2007, pp. 55 – 60.

[207] V. Tiwari and P. S. Patwal, "Design and analysis of software fault-tolerant techniques for soft-core processors in reliable sram-based FPGA," *Int. J. Compo Tech. Appl.*, pp. 1812 – 1819, 2006.

[208] K. M. Chandy and C. V. Ramamoorthy, "Rollback and recovery strategies for computer programs," *IEEE Transactions on Computers*, pp. 546 – 556, 1972.

[209] A. L. Sartor, A. Lorenzon, L. Carro, F. Kastensmidt, S. Wong, and A. Beck, "Exploiting idle hardware to provide low overhead fault tolerance for VLIW processors," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2017.

[210] D. K. Pradhan and N. H. Vaidya, "Roll-forward and rollback recovery: Performance-reliability trade-off," in *IEEE International Symposium on Fault- Tolerant Computing*, 1994, pp. 186 – 195.

[211] F. Abate, L. Sterpone, and M. Violante, "A new mitigation approach for soft errors in embedded processors," *IEEE Transactions on Nuclear Science*, pp. 2063 – 2069, 2008.

[212] F. Abate, L. Sterpone, C. Lisboa, L. Carro, and M. Violante, "New techniques for improving the performance of the lockstep architecture for SEEs mitigation in FPGA embedded processors," *IEEE Transactions on Nuclear Science*, pp. 1992 – 2000, 2009.

[213] J. Arm, Z. Bradac, and R. Stohl, "Increasing safety and reliability of roll-back and roll-forward lockstep technique for use in real-time systems," *IFAC Conference on Programmable Devices and Embedded Systems*, pp. 413 – 418, 2016, iFACPDES.

[214] M. Zheng, Z. Wang, and L. Li, "Fault injection method and voter design for dual modual redundancy FPGA hardening," in *International Conference on Electronics Information and Emergency Communication (ICEIEC)*, 2016, pp. 109 – 112.

[215] P. Reviriego, M. Demirci, J. Tabero, A. Regadío, and J. Maestro, "Dmr+: An efficient alternative to TMR to protect registers in xilinx FPGAs," *Microelectronics Reliability*, pp. 314 – 318, 2016.

[216] S. Fouad, F. Ghaffari, M. Benkhelifa, and B. Granado, "Context-aware resources placement for SRAM-based FPGA to minimize checkpoint recovery overhead," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2014, pp. 1 – 6.

[217] J. van Neumann, "Probabilistic logics and synthesis of reliable organisms from unreliable components, automata studies," *Annals of Mathematical Studies*, pp. 43 – 98, 1956.

[218] S. Venkataraman, R. Santos, and A. Kumar, "A flexible inexact TMR technique for SRAM-based FPGAs," in *Conference on Design, Automation & Test in Europe*, 2016, pp. 810 – 813.

[219] C. Bolchini, A. Miele, and M. Santambrogio, "TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGAs," in *IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*, 2007, pp. 87 – 95.

[220] K. Siozios and D. Soudris, "A low-cost fault tolerant solution targeting commercial FPGA devices," *Journal of Systems Architecture*, pp. 1255 – 1265, 2013.

[221] D. Agiakatsikas, N. T. Nguyen, Z. Zhao, T. Wu, E. Cetin, O. Diessel, and

L. Gong, "Reconfiguration control networks for TMR systems with module-based recovery," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2016, pp. 88 – 91.

[222] M. Psarakis, "Reliability-aware overlay architectures for FPGAs: Features and design challenges," *arXiv preprint arXiv:1606.06452*, 2016.

[223] F. Brosser and E. Milh, "SEU mitigation techniques for advanced reprogrammable FPGA in space," *Chalmers University of Technology*, 2014.

[224] M. Sullivan, H. Loomis, and A. Ross, "Employment of reduced precision redundancy for fault tolerant FPGA applications," in *IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2009, pp. 283 – 286.

[225] J. Johnson, "Synchronization voter insertion algorithms for FPGA designs using triple modular redundancy," Ph.D. dissertation, 2010.

[226] F. Veljkovic, T. Riesgo, and E. de la Torre, "Adaptive reconfigurable voting for enhanced reliability in medium-grained fault tolerant architectures," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2015, pp. 1 – 8.

[227] M. Krstic, M. Stojcev, G. Djordjevic, and I. Andrejic, "A mid-value select voter," *Microelectronics Reliability*, pp. 733 – 738, 2005.

[228] H. Kim, H. Lee, and K. Lee, "The design and analysis of avtmr (all voting triple modular redundancy) and dual–duplex system," *Reliability Engineering & System Safety*, pp. 291 – 300, 2005.

[229] M. Amiri and V. Přenosil, "Design of a simple reliable voter for modular redundancy implementations," *Distance Learning, Simulation and Communication*, p. 8, 2013.

[230] S. D'Angelo, C. Metra, S. Pastore, A. Pogutz, and G. Sechi, "Fault-tolerant voting mechanism and recovery scheme for TMR FPGA-based systems," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 1998, pp. 233 – 240.

[231] R. Kshirsagar and R. Patrikar, "Design of a novel fault-tolerant voter circuit for TMR implementation to improve reliability in digital circuits," *Microelectronics Reliability*, pp. 1573 – 1577, 2009.

[232] T. Ban and L. de Barros Naviner, "A simple fault-tolerant digital voter circuit in TMR nanoarchitectures," in *IEEE International NEWCAS Conference*, 2010, pp. 269 – 272.

[233] P. Balasubramanian, K. Prasad, and N. Mastorakis, "A fault tolerance improved majority voter for TMR system architectures," *arXiv preprint arXiv:1605.03771*, 2016.

[234] M. Gericota, L. Lemos, G. Alves, and J. Ferreira, "A framework for self-healing radiation-tolerant implementations on reconfigurable FPGAs," in *IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2007, pp. 1 – 6.

[235] U. Legat, A. Biasizzo, and F. Novak, "Self-reparable system on FPGA for single event upset recovery," in *International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2011, pp. 1 – 6.

[236] A. Upegui, J. Izui, and G. Curchod, "Fault mitigation by means of dynamic partial reconfiguration of Virtex-5 FPGAs," in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2012, pp. 1 – 6.

[237] Xilinx Corp., "Xilinx tmrtool user guide.UG156 (v9.0)," Xilinx Documentation, http://www.xilinx.com, 2009.

[238] "No room for error: Creating highly reliable, high-availability fpga designs," Synopsys Documentation, http://www.Synopsys.com, Synopsys Inc., Tech. Rep., 2012.

[239] A. Jacobs, G. Cieslewski, and A. George, "Overhead and reliability analysis of algorithm-based fault tolerance in FPGA systems," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 300 – 306.

[240] S. Sharma and P. Vijayakumar, "An HVD based error detection and correction of soft errors in semiconductor memories used for space applications," in *International Conference on Devices, Circuits and Systems (ICDCS)*, 2012, pp. 563 – 567.

[241] R. Hentschke, F. Marques, F. Lima, L. Carro, A. Susin, and R. Reis, "Analyzing area and performance penalty of protecting different digital modules with Hamming code and triple modular redundancy," in *Symposium on Integrated Circuits and Systems Design*. IEEE, 2002, pp. 95 – 100.

[242] S. Liu, G. Sorrenti, P. Reviriego, F. Casini, J. Maestro, M. Alderighi, and H. Mecha, "Comparison of the susceptibility to soft-errors of SRAM-based FPGA error correction codes implementations," *IEEE Transactions on Nuclear Science*, pp. 619 – 24, 2012.

[243] G. Prakash and M. Muthamizhan, "FPGA implementation of Bose Chaudhuri Hocquenghem Code (BCH) encoder and decoder for multiple error

correction control," in *International Journal of Innovative Research in Science, Engineering and Technology (IJIRSET)*, 2016.

[244] S. Park, D. Lee, and K. Roy, "Soft-error-resilient FPGAs using built-in 2-D Hamming product code," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 2, pp. 248 – 56, 2012.

[245] S. Esmaeeli, M. Hosseini, B. Vosoughi Vahdat, and B. Rashidian, "A multi-bit error tolerant register file for a high reliable embedded processor," in *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2011, pp. 532 – 537.

[246] Xilinx Corp., "Virtex-5 user guide UG190 (v5.4)," Xilinx Documentation, http://www.xilinx.com, 2012.

[247] S. Sarode and A. Patil, "Implementation of fault tolerant soft processor on FPGA," *IJAR*, pp. 781 – 784, 2016.

[248] Xilinx Corp., "Virtex-6 fpga configuration UG360 (v3.9)," Xilinx Documentation, http://www.xilinx.com, 2015.

[249] W. Yang, L. Wang, and X. Zhou, "CRC circuit design for SRAM-based FPGA configuration bit correction," in *IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, 2010, pp. 1660 – 1664.

[250] V. Savani and N. Gajjar, "Development of SEU monitor system for SEU detection and correction in Virtex-5 FPGA," in *Nirma University International Conference on Engineering (NUiCONE)*, 2011, pp. 1 – 6.

[251] S. Aishwarya and G. Mahendran, "Multiple bit upset correction in SRAM based FPGA using Mutation and Erasure codes," in *International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*, 2016, pp. 202 – 206.

[252] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. R. Sonza, and M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors," *IEEE Transactions on Nuclear Science*, pp. 2231 – 2236, 2000.

[253] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, pp. 63 – 75, 2002.

[254] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, pp. 1491 – 1501, 1985.

[255] M. Rebaudengo, M. Reorda, and M. Violante, "A new software-based technique for low-cost fault-tolerant application," in *Reliability and Maintainability Symposium*, 2003, pp. 25 – 28.

[256] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. August, and S. Mukherjee, "Software-controlled fault tolerance," *ACM Transactions on Architecture and Code Optimization (TACO)*, pp. 366 – 396, 2005.

[257] S. Rehman, "Reliable software for unreliable hardware-a cross-layer approach," Ph.D. dissertation, Karlsruhe, Karlsruher Institut für Technologie (KIT), 2015.

[258] P. Popov and L. Strigini, "Assessing asymmetric fault-tolerant software," in *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2010, pp. 41 – 50.

[259] B. Johnson, J. Aylor, and H. Hana, "Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder," *IEEE Journal of Solid-State Circuits*, pp. 208 – 215, 1988.

[260] K. Shin and H. Kim, "A time redundancy approach to TMR failures using fault-state likelihoods," *IEEE Transactions on Computers*, pp. 1151 – 1162, 1994.

[261] M. Nicolaidis, "Time redundancy based soft-error tolerance to rescue nanometer technologies," in *IEEE VLSI Test Symposium (Cat. No.PR00146)*, 1999, pp. 86 – 94.

[262] L. Anghel, D. Alexandrescu, and M. Nicolaidis, "Evaluation of a soft error tolerance technique based on time and/or space redundancy," in *Symposium on Integrated Circuits and Systems Design (Cat. No.PR00843)*, 2000, pp. 237 – 242.

[263] J. Yoon and H. Kim, "Time-redundant recovery policy of TMR failures using rollback and roll-forward methods," *IEEE Proceedings - Computers and Digital Techniques*, pp. 124 – 132, 2000.

[264] D. Pradhan, *Fault Tolerant Computer System Design*. Prentice-Hall, 1996.

[265] R. Gosheblagh and K. Mohammadi, "Hybrid time and hardware redundancy to mitigate SEU effects on SRAM-FPGAs: Case study over the MicroLAN protocol," *Microelectronics Journal*, pp. 870 – 879, 2014.

[266] D. Burlyaev, P. Fradet, and A. Girault, "Time-redundancy transformations for adaptive fault-tolerant circuits," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2015, pp. 1 – 8.

[267] A. Balachandran, N. Veeranna, and B. Schafer, "On time redundancy of fault tolerant C-based MPSoCs," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2016, pp. 631 – 636.

[268] S. Pontarelli, M. Ottavi, V. Vankamamidi, G. Cardarilli, F. Lombardi, and A. Salsano, "Analysis and evaluations of reliability of reconfigurable FPGAs," *Journal of Electronic Testing*, pp. 105 – 116, 2008.

[269] A. Astarloa, J. Lazaro, U. Bidarte, J. Jimenez, and A. Zuloaga, "A FPGA based platform for autonomous fault tolerant systems," in *Trends in Applied Intelligent Systems*. DCIS Proceedings, 2010, pp. 234 – 239.

[270] S. D'Angelo, C. Metra, and G. Sechi, "Transient and permanent fault diagnosis for FPGA-based TMR systems," in *International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, 1999, pp. 330 – 338.

[271] C. Bolchini, C. Sandionigi, L. Fossati, and D. Codinachs, "A reliable fault classifier for dependable systems on SRAM-based FPGAs," in *IEEE International On-Line Testing Symposium (IOLTS)*, 2011, pp. 92 – 97.

[272] J. Lach, W. Mangione-Smith, and M. Potkonjak, "Efficiently supporting fault-tolerance in FPGAs," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1998, pp. 105 – 115.

[273] Xilinx Corp., "Vivado design suite. using constraints. tutorial.UG945 (v2013.3)," Xilinx Documentation, http://www.xilinx.com, 2013.

[274] W. Huang and E. McCluskey, "Column-based precompiled configuration techniques for FPGA," in *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2001, pp. 137 – 146.

[275] S. Mitra, W. Huang, N. Saxena, S. Yu, and E. McCluskey, "Reconfigurable architecture for autonomous self-repair," *IEEE Design Test of Computers*, pp. 228 – 240, 2004.

[276] R. Salvador, A. Otero, J. Mora, E. de la Torre, L. Sekanina, and T. Riesgo, "Fault tolerance analysis and self-healing strategy of autonomous, evolvable hardware systems," in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2011, pp. 164 –169.

[277] M. Jing and L. Woolliscroft, "A fault-tolerant multiple processor system for space instrumentation," in *International Conference on Control*. IET, 1991, pp. 411 – 416.

[278] S. Yu and E. McCluskey, "Permanent fault repair for FPGAs with limited

redundant area," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2001, pp. 125 – 133.

[279] S. Yu and E. McCluskey, "On-line testing and recovery in TMR systems for real-time applications," in *Proceedings International Test Conference*, 2001, pp. 240 – 249.

[280] M. Ebrahimi, S. Miremadi, and H. Asadi, "ScTMR: A scan chain-based error recovery technique for TMR systems in safety-critical applications," in *Design, Automation Test in Europe*, 2011, pp. 1 – 4.

[281] J. Azambuja, F. Sousa, L. Rosa, and F. Kastensmidt, "Evaluating large grain TMR and selective partial reconfiguration for soft error mitigation in SRAM-based FPGAs," in *IEEE International On-Line Testing Symposium (IOLTS)*, 2009, pp. 101 – 106.

[282] C. Pilotto, J. R. Azambuja, and F. L. Kastensmidt, "Synchronizing triple modular redundant designs in dynamic partial reconfiguration applications," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2008, pp. 199 – 204.

[283] E. Cetin and O. Diessel, "Guaranteed fault recovery time for FPGA-based TMR circuits employing partial reconfiguration," in *International Workshop on Computing in Heterogeneous. Autonomous 'N'Goal-oriented Environments*, 2012.

[284] N. Goel and K. Paul, "Hardware controlled and software independent fault tolerant FPGA architecture," in *International Conference on Advanced Computing and Communications (ADCOM)*, 2007, pp. 497 – 502.

[285] Z. Feng, N. Jing, and L. He, "IPF: In-place X-Filling Algorithm for the reliability of modern FPGAs," 2013, pp. 1 – 1.

[286] Y. Liu and W. Li, "Study on hardware implementation of artificial immune system," in *International Conference on Information Engineering and Computer Science (ICIECS)*, 2010, pp. 1 – 4.

[287] Z. Wang, L. L. Ding, Z. B. Yao, H. X. Guo, H. Zhou, and M. Lv, "The reliability and availability analysis of SEU mitigation techniques in SRAM-based FPGAs," in *European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, 2009, pp. 497 – 503.

[288] A. Ramos, A. Ullah, P. Reviriego, and J. Maestro, "Efficient protection of the register file in soft-processors implemented on xilinx fpgas," *IEEE Transactions on Computers*, no. 99, pp. 1 – 1, 2017.

[289] Altera, "Robust SEU mitigation with stratix III FPGAs," Altera Documentation, http://www.altera.com, 2007.

[290] C. Rousselle, M. Pflanz, A. Behling, T. Mohaupt, and H. Vierhaus, "A register-transfer-level fault simulator for permanent and transient faults in embedded processors," in *Conference and Exhibition Design, Automation and Test in Europe.*, 2001, p. 811.

[291] P. Folkesson, S. Svensson, and J. Karlsson, "A comparison of simulation based and scan chain implemented fault injection," in *International Symposium on Fault-Tolerant Computing*, 1998, pp. 284 – 293.

[292] V. Sieh, O. Tschache, and F. Balbach, "VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions," in *International Symposium on Fault Tolerant Computing*, 1997, pp. 32 – 36.

[293] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcia, and L. Entrena, "Autonomous fault emulation: A new FPGA-based acceleration system for hardness evaluation," *IEEE Transactions on Nuclear Science*, pp. 252 – 261, 2007.

[294] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela, and C. Lopez-Ongil, "Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection," *IEEE Transactions on Computers*, pp. 313 – 322, 2012.

[295] J. Walters, K. Zick, and M. French, "A practical characterization of a NASA SpaceCube application through fault emulation and laser testing," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1 – 8.

[296] M. Bellato, M. Ceschia, M. Menichelli, A. Papi, J. Wyss, and A. Paccagnella, "Ion beam testing of SRAM-based FPGA's," in *European Conference on Radiation and Its Effects on Components and Systems*, 2001, pp. 474 – 480.

[297] T. Vanat, J. Pospiil, F. Kriek, J. Ferencei, and H. Kubatova, "A system for radiation testing and physical fault injection into the fpgas and other electronics," in *Euromicro Conference on Digital System Design*, 2015, pp. 205 – 210.

[298] M. Violante, L. Sterpone, A. Manuzzato, S. Gerardin, P. Rech, M. Bagatin, A. Paccagnella, C. Andreani, G. Gorini, A. Pietropaolo, G. Cardarilli, S. Pontarelli, and C. Frost, "A new hardware/software platform and a

new 1/e neutron source for soft error studies: Testing FPGAs at the ISIS facility," *IEEE Transactions on Nuclear Science*, pp. 1184 – 1189, 2007.

[299] E. Fuller, M. Caffrey, A. Salazar, C. Carmichael, and J. Fabula, "Radiation testing update, SEU mitigation, and availability analysis of the Virtex FPGA for space re-configurable computing," in *International Conference on Military and Aerospace Programmable Logic Devices*, 2000.

[300] M. French, P. Graham, M. Wirthlin, and L. Wang, "Cross functional design tools for radiation mitigation and power optimization of FPGA circuits," in *NASA Earth Science Technology Conference*. Citeseer, 2006.

[301] P. Maillard, M. Hart, J. Barton, P. Jain, and J. Karp, "Neutron, 64 mev proton, thermal neutron and alpha single-event upset characterization of Xilinx 20nm UltraScale Kintex FPGA," in *IEEE Radiation Effects Data Workshop (REDW)*, 2015, pp. 1 – 5.

[302] M. Berg, "Field Programmable Gate Array (FPGA) Single Event Effect (SEE) radiation testing," *NASA Electronic Parts and Packaging (NEPP)*, 2012.

[303] W. Moreno, J. Samson, and F. Falquez, "Laser injection of soft faults for the validation of dependability design," *Journal of Universal Computer Science (J-UCS)*, pp. 712 – 729, 1999.

[304] V. Pouget, A. Douin, G. Foucard, P. Peronnard, D. Lewis, P. Fouillat, and R. Velazco, "Dynamic testing of an SRAM-based FPGA by time-resolved laser fault injection," in *IEEE International On-Line Testing Symposium*. IEEE, 2008, pp. 295 – 301.

[305] G. Saggese, N. Wang, Z. Kalbarczyk, S. Patel, and R. Iyer, "An experimental study of soft errors in microprocessors," *IEEE Micro*, vol. 25, no. 6, pp. 30 – 39, 2005.

[306] J. Azambuja, G. Nazar, P. Rech, L. Carro, F. Lima Kastensmidt, T. Fairbanks, and H. Quinn, "Evaluating neutron induced SEE in SRAM-based FPGA protected by hardware- and software-based fault tolerant techniques," *IEEE Transactions on Nuclear Science*, pp. 4243 – 4250, 2013.

[307] U. Kretzschmar, A. Astarloa, J. Lazaro, J. Jimenez, and A. Zuloaga, "An automatic experimental set-up for robustness analysis of designs implemented on SRAM FPGAS," in *International Symposium on System on Chip (SoC)*, 2011, pp. 96 – 101.

[308] Z. Jing, L. Zengrong, C. Lei, W. Shuo, W. Zhiping, C. Xun, and Q. Chang,

"An accurate fault location method based on configuration bitstream analysis," in *NORCHIP*, 2012, pp. 1 – 5.

[309] I. Villalta, U. Bidarte, J. Gomez-Cornejo, J. Jimenez, and C. Cuadrado, "Effect of different design stages on the SEU failure rate of FPGA systems," in *Conference on Design of Circuits and Integrated Systems (DCIS)*, 2016, pp. 1 – 6.

[310] U. Kretzschmar, "Estimating the resilience against single event upsets in applications implemented on SRAM based FPGAs," Ph.D. dissertation, Basque Country University UPV/EHU, 2014.

[311] M. Alderighi, F. Casini, S. D'Angelo, M. Mancini, A. Marmo, S. Pastore, and G. Sechi, "A tool for injecting SEU-like faults into the configuration control mechanism of Xilinx Virtex FPGAs," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003, pp. 71 – 78.

[312] N. Harward, M. Gardiner, L. Hsiao, and M. Wirthlin, "A fault injection system for measuring soft processor design sensitivity on virtex-5 FPGAs," in *FPGAs and Parallel Architectures for Aerospace Applications*. Springer, 2016, pp. 61 – 74.

[313] M. Shokrolah-Shirazi and S. Miremadi, "FPGA-based fault injection into synthesizable Verilog HDL models," in *International Conference on Secure System Integration and Reliability Improvement*, 2008, pp. 143 – 149.

[314] M. Alderighi, F. Casini, S. D'Angelo, M. Mancini, D. Codinachs, S. Pastore, C. Poivey, G. Sechi, G. Sorrenti, and R. Weigand, "Experimental validation of fault injection analyses by the FLIPPER tool," *IEEE Transactions on Nuclear Science*, pp. 2129 – 2134, 2010.

[315] M. Jeitler, M. Delvai, and S. Reichor, "FuSE - a hardware accelerated HDL fault injection tool," in *Southern Conference on Programmable Logic (SPL)*, 2009, pp. 89 – 94.

[316] B. Rahbaran, A. Steininger, and T. Handl, "Built-in fault injection in hardware - the FIDYCO example," in *IEEE International Workshop on Electronic Design, Test and Applications (DELTA)*, 2004, pp. 327 – 332.

[317] W. Mansour and R. Velazco, "An automated SEU fault-injection method and tool for HDL-based designs," *IEEE Transactions on Nuclear Science*, pp. 2728 – 2733, 2013.

[318] L. Naviner, J. Naviner, G. dos Santos, E. Marques, and N. Paiva, "FIFA: A

Fault-Injection–Fault-Analysis-based tool for reliability assessment at RTL level," *Microelectronics Reliability*, pp. 1459 – 1463, 2011.

[319] J. Mogollon, H. Guzman-Miranda, J. Napoles, J. Barrientos, and M. Aguirre, "FTUNSHADES2: A novel platform for early evaluation of robustness against SEE," in *European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, 2011, pp. 169 – 174.

[320] N. Harward, "Measuring soft error sensitivity of fpga soft processor designs using fault injection," 2016.

[321] U. Kretzschmar, J. Gomez-Cornejo, A. Astarloa, U. Bidarte, and J. D. Ser, "Synchronization of faulty processors in coarse-grained TMR protected partially reconfigurable FPGA designs," *Reliability Engineering & System Safety*, pp. 1 – 9, 2016.

[322] OpenCores - Meziti ElIbrahimi, Abdallah and Zied, ABBASSI, "Copy-Blaze," http://opencores.org, 2011.

[323] OpenCores - Bleyer, P., "PacoBlaze," http://bleyer.org/pacoblaze/, 2007.

[324] W. M. El-Medany, A. Alomary, R. Al-Hakim, S. Al-Irhayim, and M. Nousif, "Implementation of GPRS-based positioning system using PIC microcontroller," in *International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN)*, 2010, pp. 365 – 368.

[325] P. N. Rivera-Arzola, J. C. Ramos-Fernandez, J. M. O. Franco, M. Villanueva-Ibanez, and M. A. Flores-Gonzalez, "A PIC microcontroller embedded system for medical rehabilitation using ultrasonic stimulation through controlling planar X-Y scanning trajectories," in *IEEE Electronics, Robotics and Automotive Mechanics Conference (CERMA)*, 2011, pp. 307 – 310.

[326] S. Genovesi, A. Monorchio, M. B. Borgese, S. Pisu, and F. M. Valeri, "Frequency-reconfigurable microstrip antenna with biasing network driven by a PIC microcontroller," *IEEE Antennas and Wireless Propagation Letters*, vol. 11, pp. 156 – 1 59, 2012.

[327] M. Bales, "Xilinx implementation tutorial," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2007.

[328] T. Santini, L. Carro, F. R. Wagner, and P. Rech, "Reliability analysis of operating systems for embedded SoC," in *European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, 2015, pp. 1 – 5.

[329] Xilinx Corp., "Zynq-7000 all programmable SoC technical reference manual UG585 (v1.9.1)," AVNET Documentation, http://www.zedboard.org, 2014.

[330] T. Simon, *PuTTY*, http://www.putty.org, 2000.

[331] OpenCores - Wallner, Daniel, "PPX16 MUC project," http://opencores.org, 2002.

[332] U. Kretzschmar, A. Astarloa, J. Lazaro, J. Jimenez, and A. Zuloaga, "An Automatic Experimental Set-up for Robustness Analysis of Designs Implemented on SRAM FPGAs," in *International Symposium on System on Chip (SoC)*, nov. 2011, pp. 96 –101.

[333] Lattice Semiconductor Corporation, "LatticeMico32 processor user's guide," Lattice Semiconductor Corporation Documentation, http://www.latticesemi.com/, 2011.

[334] Gaisler, "Leon3 product sheet," Gaisler Documentation, http://www.gaisler.com/, 2010.

[335] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," *SIGARCH Comput. Archit. News*, pp. 348 – 354, 1984.

[336] J. Archibald and J. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. Comput. Syst.*, pp. 273 – 298, 1986.