

**UNIVERSIDADE FEDERAL DE SANTA CATARINA  
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA  
ELÉTRICA**

Frederico Ferlini

**PLAESER - PLATAFORMA DE EMULAÇÃO DE *SOFT ERRORS*  
VISANDO A ANÁLISE EXPERIMENTAL DE  
TÉCNICAS DE TOLERÂNCIA A FALHAS:  
UMA PROTOTIPAÇÃO RÁPIDA UTILIZANDO FPGAS**

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina para a obtenção do Grau de Mestre em Engenharia Elétrica.  
Orientador: Prof. Dr. Eduardo Augusto Bezerra

Florianópolis  
2012

Ficha de identificação da obra elaborada pelo autor,  
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Ferlini, Frederico

PLAESER - Plataforma de Emulação de Soft Errors Visando a Análise Experimental de Técnicas de Tolerância a Falhas [dissertação] : uma prototipação rápida utilizando FPGAs / Frederico Ferlini ; orientador, Eduardo Augusto Bezerra - Florianópolis, SC, 2012.

159 p. ; 21cm

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia Elétrica.

Inclui referências

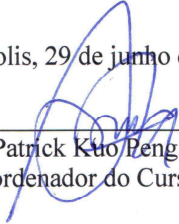
1. Engenharia Elétrica. 2. Tolerância a falhas. 3. Injeção de Falhas em Hardware. 4. FPGA. 5. Reconfiguração Parcial Dinâmica. I. Bezerra, Eduardo Augusto . II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

Frederico Ferlini


**PLAESER - PLATAFORMA DE EMULAÇÃO DE *SOFT ERRORS*  
VISANDO A ANÁLISE EXPERIMENTAL DE  
TÉCNICAS DE TOLERÂNCIA A FALHAS:  
UMA PROTOTIPAÇÃO RÁPIDA UTILIZANDO FPGAS**


Esta Dissertação foi julgada adequada para obtenção do Título de Mestre em Engenharia Elétrica, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina

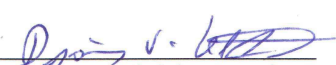
Florianópolis, 29 de junho de 2012.

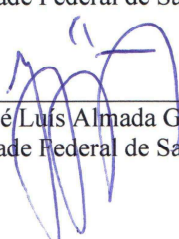
  
\_\_\_\_\_  
Prof. Patrick Kuo Peng, Dr.  
Coordenador do Curso

**Banca Examinadora:**

  
\_\_\_\_\_  
Prof. Eduardo Augusto Bezerra, Dr.  
Presidente (Orientador)  
Universidade Federal de Santa Catarina

  
\_\_\_\_\_  
Prof. Fernando Gehm Moraes, Dr.  
Pontifícia Universidade Católica do Rio Grande do Sul

  
\_\_\_\_\_  
Prof. D'Jones Vinicius Lettnin, Dr.  
Universidade Federal de Santa Catarina

  
\_\_\_\_\_  
Prof. José Luis Almada Güntzel, Dr.  
Universidade Federal de Santa Catarina

*Este trabalho é dedicado aos meus Amigos/Colegas e principalmente aos meus extraordinários Pais/Irmãos que, assim como Deus, sempre acreditaram em mim, inquestionavelmente.*



## AGRADECIMENTOS

Primeiramente, gostaria de agradecer ao Prof. Dr. Eduardo Augusto Bezerra pela oportunidade de realizar o mestrado na UFSC, por prover os recursos necessários para a pesquisa e pela orientação, entusiasmo, compreensão e paciência ao longo do desenvolvimento desse trabalho.

Agradeço principalmente a minha Família que, de forma inquestionável, me apoiou em todas as decisões que tomei até então. Agradeço a minha Mãe a quem amo muito e que, mesmo sentindo muito a falta dos filhos que moram longe, procurou não transparecer a saudade e me ajudou muito com a sua compreensão, conselhos e carinho sempre que precisei. Agradeço ao meu Pai e sua esposa a quem amo muito, por sempre atenderem a qualquer pedido de ajuda de forma inquestionável e acreditarem em mim independente do que eu faça. Não menos importante, eu agradeço aos meus Irmãos a quem amo muito e que me apoiam incondicionalmente, sem nem mesmo saber direito qual a minha área de atuação. Finalmente, não menos distante, agradeço aos meus primos que completam a minha família e a quem, da mesma forma, amo muito.

Agradeço aos meus Amigos/Colegas que, além de me apoiarem, dão à minha vida motivo de viver através da alegria e pelo simples fato de estar junto deles, inclusive pelas inúmeras histórias que escrevemos juntos. Agradeço especialmente àqueles que pegaram a estrada para passarem um tempo comigo, aos que compreenderam os meus momentos de isolamento e, principalmente, àqueles que aturaram as minhas manias e teorias durante nosso convívio. Agradeço também ao Badoo por dar uma ajuda em momentos difíceis.

Agradeço a UFSC e a CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior), que forneceram o suporte para o desenvolvimento deste trabalho.

Enfim, dessa vez não é para pedir, mas sim para agradecer. Muito obrigado Deus, por sempre se mostrar presente quando mais precisei e atender às minhas preces, de maneira que me encorajou a ser uma pessoa melhor.

*“If a problem has no solution, it may not be a problem, but a fact - not to be solved, but to be coped with over time”*

(Shimon Peres)

## RESUMO

O constante avanço na fabricação de circuitos integrados com a miniaturização da tecnologia, o aumento da frequência de operação e a diminuição da tensão de alimentação fazem deles cada vez mais sensíveis à radiação. A preocupação com a sensibilidade de circuitos integrados não é mais restrita a projetos de aplicações espaciais onde o ambiente é mais hostil quanto à radiação. Circuitos fabricados com tecnologias em escala nanométrica são potencialmente sensíveis a partículas que se encontram na atmosfera terrestre e até no nível do mar. A importância da tolerância a falhas em semicondutores existe desde quando anomalias foram observadas no comportamento de dispositivos operando no espaço. A larga presença de circuitos integrados em diversas áreas do nosso cotidiano faz com que técnicas de tolerância a falhas ganhem importância também para aplicações terrestres. Desse modo, formas eficientes de avaliação dessas técnicas de tolerância a falhas são essenciais para lidar com essa demanda. É importante que essa avaliação possa ser realizada em etapas iniciais do projeto de circuitos integrados tolerantes à radiação de forma a reduzir o custo com locação de instalações que utilizam equipamentos de radiação induzida para verificação. Nesse contexto, o trabalho de dissertação apresenta um estudo sobre diferentes técnicas de injeção de falhas. Além do estudo, foi desenvolvida uma plataforma de emulação de *soft errors* (PLAESER) visando a análise experimental de técnicas de tolerância a falhas. A plataforma PLAESER provê suporte ao fluxo proposto para avaliação de técnicas de tolerância a falhas em fase inicial do projeto de circuitos robustos através da prototipação rápida em FPGAs. Os resultados obtidos com os casos de teste utilizados procuram mostrar o emprego do fluxo proposto para análise de técnicas de tolerância a falhas.

**Palavras-chave:** Emulação de falhas. *Soft Errors*. Reconfiguração parcial. Prototipação rápida. FPGA. Circuitos reconfiguráveis. Injeção de falhas. Tolerância a falhas. Circuitos endurecidos.

## ABSTRACT

The continuous improvements in the integrated circuits manufacture process considering the miniaturization of technology, increase of clock frequencies and limitation of power supply, make them more susceptible to radiation. The concern with circuit sensitivity is no longer restricted to space applications, in harsh environment. Integrated circuits manufactured with nanometric technologies are potentially sensitive to particles present in the atmosphere and also at the sea level. Fault tolerance strategies applied to semiconductors have been around since upsets were first experienced in space applications. The large usage of integrated circuits in several areas of everyday life makes fault tolerance techniques important also for terrestrial applications. Therefore, efficient hardness evaluation solutions are essential to deal with this demand. Such evaluation is important and should be performed earlier in hardened integrated circuit designs in order to reduce costs with rental of radiation facilities. In this context, this work presents a evaluation of different fault injection techniques. Moreover, a soft error emulation platform (PLAESER) has been developed in order to analyze fault tolerance techniques experimentally. PLEASER gives support to the flow proposed to evaluate fault tolerance techniques earlier in hardened circuit designs through rapid prototyping. The results obtained with the selected test cases show the employment of the proposed flow to analyze fault tolerance techniques.

**Keywords:** Partial reconfiguration. Soft errors. Reconfiguration circuits. Fault tolerance. Hardened circuit. FPGA. Fault emulation. Fault injection. Rapid prototyping.

## LISTA DE FIGURAS

Figura 1 - Relação entre falha, erro e defeito.....	21
Figura 2 - Ambiente radioativo espacial. ....	25
Figura 3 - Relação entre as fontes de radiação e os efeitos nos componentes.....	26
Figura 4 - Exemplo de um SET. ....	28
Figura 5 - Arquitetura genérica de um FPGA. ....	29
Figura 6 - Memória de configuração do FPGA. ....	30
Figura 7 - Cadeias de <i>scan</i> propostas em (33). ....	39
Figura 8 - (A) Plataforma de emulação AMUSE (34) - (B) Troca de RTL para GL para emulação do efeito SET.....	40
Figura 9 - Os equivalentes em GL de um elemento (A) combinacional e um (B) sequencial instrumenta da AMUSE. ....	41
Figura 10 - Fluxo proposto em (35) para diminuir a intrusão da instrumentação. ....	42
Figura 11 - Controlador de localização e o exemplo de um registrador instrumentado utilizado pela técnica apresentada em (36).....	42
Figura 12 - Fluxo do sistema FLIPPER (37). ....	44
Figura 13 - DTE da FT-UNSHADES (39). ....	45
Figura 14 - Arquitetura do SoC AT94K da Atmel (42). ....	46
Figura 15 - O fluxo (A) e a arquitetura (B) do bloco de injeção de falha desenvolvido em (42). ....	46
Figura 16 - Estrutura do FuSE (44).....	47
Figura 17 - Resumo comparativo das técnicas de injeção. ....	48
Figura 18 - Arquitetura do FPGA XC5VLX110T. ....	52
Figura 19 - Matriz de roteamento. ....	53
Figura 20 - <i>Configurable Logic Block (CLB)</i> . ....	53
Figura 21 - Diagrama detalhado do SLICEL. ....	55
Figura 22 - Cadeia de conexão do JTAG. ....	56
Figura 23 - Arquitetura do JTAG encontrada nos FPGAs da Xilinx. ....	57
Figura 24 - Ferramenta iMPACT em modo de depuração da cadeia JTAG. ....	59
Figura 25 - Exemplo (em hexa) do <i>bitstream</i> em formato texto. ....	60
Figura 26 - Exemplo da utilização da interface UART do V5SC. ....	65
Figura 27 - Significados das coordenadas XY para os CLB. ....	70
Figura 28 - Campos da palavra de 32 bits de endereço.....	70
Figura 29 - Sistema de coordenadas X endereçamento.....	71
Figura 30 - Visão em profundidade da metade inferior da memória de configuração do FPGA. ....	73
Figura 31 - Pilha da coluna (47) que é do tipo CLB. ....	75
Figura 32 - As oito palavras de configuração das LUTs do <i>slice</i> X81Y19.....	76
Figura 33 - Função de conversão da coordenada XY para o endereço da memória de configuração do FPGA XC5VLX110T.....	77
Figura 34 - Trajetória do <i>bitstream</i> de configuração do FPGA. ....	77
Figura 35 - Planos da memória do FPGA da Xilinx. ....	78

Figura 36 - Função que converte as coordenadas de um <i>slice</i> no índice da palavra do <i>bitstream</i> com os 16 bits iniciais da configuração da LUT .....	80
Figura 37 - Sistema Injetor de Falhas. ....	82
Figura 38 - Estrutura de funcionamento do GLIFA (Gerador de Lista de Falhas). ....	87
Figura 39 - Estrutura de informações dos slices da Virtex5 encontrada na biblioteca do GLIFA.....	88
Figura 40 - Amostra dos parâmetros encontrados na biblioteca do GLIFA "FPGA_MODEL.LIB". ....	89
Figura 41 - Arquivo com a lista de falhas gerada pelo GLIFA.....	90
Figura 42 - Arquitetura do SoCIF.....	93
Figura 43 - Fluxo completo do software do SoCIF. ....	102
Figura 44 - Fluxo Básico .....	103
Figura 45 - O topo “Sistema de Injeção de Falhas” (SIF) com os blocos de entrada/saída dos DUTs e a instanciação dos dois DUTs FAULTY/GOLDEN. ....	104
Figura 46 - Modelo do TOPO_SIF.....	105
Figura 47 - Ilustração do registrador gerador do sinal resultante do módulo comparador.....	106
Figura 48 - Módulo Comparador.....	107
Figura 49 - Aplicação do atributo no TOPO_SIF.....	108
Figura 50 - Fluxo de implementação do PlanAhead™ utilizando <i>netlists</i> . ....	109
Figura 51 - Exemplo de restrição de área utilizando a devida sintaxe do UCF. ....	110
Figura 52 - Exemplo do arquivo de descrição do circuito implementado utilizando a linguagem XDL. ....	111
Figura 53 - Exemplo do relatório de resultados gerado ao final do fluxo do PLAESE. ....	113
Figura 54 - O esquemático ilustrativo do caso de teste do contador.....	115
Figura 55 - Ilustração do contador com redundância tripla. ....	116
Figura 56 - Votador majoritário.....	116
Figura 57 - Contador TMR exemplo disponibilizado pela Xilinx. ....	117
Figura 58 - <i>PlanAhead</i> do contador com a área das instâncias GOLDEN e FAULTY estabelecidas. ....	119
Figura 59 - Resultados da implementação do contador com TMR.....	120
Figura 60 - Utilização dos recursos do FPGA XC5VLX110T. ....	120
Figura 61 - Resultado da emulação de falhas no contador TMR da Xilinx. ....	121
Figura 62 - Resultado da emulação de falhas acumulada no contador TMR da Xilinx.....	121
Figura 63 - PlanAhead do contador sem TMR com a lógica de carry. ....	122
Figura 64 - Exemplo do uso da LUT para o acesso ao controle dos multiplexadores. ....	123
Figura 65 - Utilização dos recursos do FPGA XC5VLX110T. ....	124
Figura 66 - Resultado da emulação de falhas no contador sem TMR e com lógica de carry. ....	124

Figura 67 - PlanAhead do contador com TMR sem a lógica de carry. ....	125
Figura 68 - Utilização dos recursos do FPGA XC5VLX110T. ....	126
Figura 69 - Resultado da emulação de falhas no contador com TMR e sem a lógica de carry. ....	126
Figura 70 - Resultado da emulação de falhas acumulada no contador com TMR sem a lógica de carry. ....	127
Figura 71 - <i>PlanAhead</i> do contador sem TMR e sem a lógica de carry.....	128
Figura 72 - Utilização dos recursos do FPGA XC5VLX110T. ....	128
Figura 73 - Resultado da emulação de falhas no contador sem TMR e sem a lógica de carry. ....	129
Figura 74 - Computador de bordo com redundância externa e monitor de barramento. ....	131
Figura 75 - Comandos para geração da "BROM" com a imagem do programa teste para o LEON3. ....	132
Figura 76 - A configuração do SoC do LEON3 e espaço de endereçamento dos periféricos. ....	133
Figura 77 - A reinicialização da execução do software do OBC após a injeção de uma falha.....	134
Figura 78 - PlanAhead da implementação do SIF com a instância do processador LEON3 principal (FAULTY DUT) com restrição de área. ....	135
Figura 79 - Percentual de ocupação do FPGA com o projeto do SIF com o LEON3. ....	136
Figura 80 - Quantidade de recursos do FPGA utilizados no projeto SIF com o LEON3. ....	136
Figura 81 - Quantidade estimada de componentes utilizados no projeto todo (SIF) e pelo SoCIF e pelas duas instâncias do DUT (FAULTY e GOLDEN). ....	137
Figura 82 - Resultado da injeção de falhas no OBC com o monitor de barramento. ....	138
Figura 83 - Pseudocódigo do algoritmo executado no VisuAlg. ....	151
Figura 84 - A biblioteca descrita no arquivo "fpga.bib". ....	158

## LISTA DE TABELAS

Tabela 1 - Especificação das características do FPGA XC5VLX110T. ....	51
Tabela 2 - Resumo de medidas (palavras/quadro/ <i>bitstream</i> /FPGA). ....	61
Tabela 3 - Comandos do V5SC quando em condição de controle. ....	64
Tabela 4 - Descrição dos campos da palavra de endereço do FPGA. ....	70
Tabela 5 - Tamanho da pilha de cada tipo de coluna. ....	73
Tabela 6 - Resumo do número de quadros nos diferentes planos do FPGA. ....	79
Tabela 7 - Campos de cada linha do arquivo "FAULTS.LIST". ....	91
Tabela 8 - Cálculo do intervalo de tempo de cada teste. ....	96
Tabela 9 - Resumo do resultado da injeção de falhas no OBC com o monitor de barramento. ....	137



## LISTA DE ABREVIATURAS E SIGLAS

AMBA – Advanced Microcontroller Bus Architecture  
ASCII – American Standard Code for Information Interchange  
ASIC – Application Specific Integrated Circuit  
AVR – Advanced Virtual RISC  
AXI – Advanced eXtensible Interface  
BCC – Bare-C Cross-Compiler  
BIST – Built-In Self-Test  
BitGen – Bitstream Generator  
BMM – Block RAM Memory Map  
BRAM – Block RAM  
BSDL – Boundary-Scan Description Language  
BSP – Board Support Package  
CLB – Configurable Logic Block  
COTS – Comercial-Off-The-Shelf  
CRC – Cyclic Redundancy Check  
DCM – Digital Clock Manager  
DD – Displacement Damage  
DRP – Dynamic Reconfigure Port  
DSP – Digital Signal Processing  
DUT – Design Under Test  
EDIF – Electronic Design Interchange Format  
EDK – Embedded Development Kit  
ELF – Executable and Linkable Format  
ESA – European Space Agency  
FAT – File Allocation Table  
FIT – Failure in Time  
FPGA – Field-Programmable Gate Array  
GLIFA – Gerador de Lista de Falhas  
GCR – Galactic Cosmic Rays  
GCC – GNU Compiler Collection  
GPqCom – Grupo de Pesquisa em Comunicações  
GSE – Grupo de Sistemas Embarcados  
GTP – Gigabit Transceiver  
HDL – Hardware Description Language  
ICAP – Internal Configuration Access Port  
IDE – Integrated Development Environment  
ILA – Integrated Logic Analyzer  
IOB – Input/Output Block  
IP – Intellectual Property

JTAG – Joint Test Action Group  
LET – Linear Energy Transfer  
LUT – Look-Up Table  
MBU – Multiple Bit Upset  
MDM – MicroBlaze™ Debug Module  
MMCM – Mixed-Mode Clock Manager  
MPU – Microprocessor Unit  
MUT – Module Under Test  
NCD – Native Circuit Description  
OBC – On-Board Computer  
OCD – On-Chip Debug  
PLAESER – Plataforma de Emulação de Soft Erros  
PLB – Programmable Logic Blocks  
PLL – Phase-Locked Loop  
RAM – Random Access Memory  
RISC – Reduced Instruction Set Computer  
RTL – Register Transfer Level  
SBE – Single Bit Error  
SDK – Software Development Kit  
SEE – Single Event Effect  
SEL – Single Event Latchup  
SEM – Soft Error Mitigation  
SER – Soft Error Rate  
SET – Single Event Transient  
SEU – Single Event Upset  
SIF – Sistema Injetor de Falhas  
SoCIF – System-on-a-Chip Injetor de Falhas  
SRAM – Static RAM  
SRL – Shift Register LUT  
TAP – Test Access Port  
TID – Total Ionization Dose  
TMR – Triple Modular Redundancy  
UCF – User Constraints File  
UFSC – Universidade Federal de Santa Catarina  
VHDL – VHSIC Hardware Description Language  
VHSIC – Very High Speed Integrated Circuit  
V5SC – Virtex-5 SEU Controller  
XDL – Xilinx® Description Language  
XMD – Xilinx® Microprocessor Debugger  
XPS – Xilinx® Platform Studio  
XST – Xilinx® Synthesis Technology

## SUMÁRIO

<b>1 INTRODUÇÃO .....</b>	<b>17</b>
1.1 MOTIVAÇÃO .....	18
1.2 OBJETIVOS .....	19
1.3 ORGANIZAÇÃO DO TEXTO .....	20
<b>2 CONCEITOS BÁSICOS .....</b>	<b>21</b>
2.1 CONFIABILIDADE .....	21
2.2 RADIAÇÃO E SEUS EFEITOS .....	23
2.3 MODELOS DE FALHAS .....	27
2.4 CIRCUITOS RECONFIGURÁVEIS .....	29
<b>3 INJEÇÃO DE FALHAS (TRABALHOS RELACIONADOS)....</b>	<b>31</b>
3.1 TÉCNICAS DE INJEÇÃO FÍSICA DE FALHAS .....	32
3.1.1 Método por Radiação .....	32
3.1.2 Método por Raios Laser .....	33
3.1.3 Método de Injeção por Pinos .....	33
3.2 TÉCNICAS DE INJEÇÃO DE FALHAS NO NÍVEL LÓGICO .....	34
3.2.1 Software-Implemented Fault Injection (SWIFI).....	34
3.2.2 Injeção de Falhas Baseada em Simulação.....	35
3.2.3 Injeção de Falhas Utilizando Recursos de Depuração .....	36
3.2.4 Injeção de Falhas por Emulação.....	37
3.2.4.1 Instrumentação .....	38
3.2.4.2 Reconfiguração .....	43
3.2.5 Técnicas Híbridas de Injeção de Falhas.....	46
<b>4 INVESTIGAÇÃO DA ARQUITETURA ALVO .....</b>	<b>49</b>
4.1 ARQUITETURA .....	49
4.2 CONFIGURAÇÃO DO FPGA .....	55
4.3 INVESTIGAÇÃO DA MEMÓRIA DE CONFIGURAÇÃO .....	58
4.3.1 Configuration Readback .....	58
4.3.2 Controlador V5SC .....	63
4.3.3 Internal Access Configuration Port (ICAP) .....	68
4.4 COORDENADAS X ENDEREÇO DE QUADROS DE MEMÓRIA .....	69
<b>5 PLATAFORMA DE EMULAÇÃO DE SOFT ERRORS.....</b>	<b>81</b>
5.1 CIRCUITO RECONFIGURÁVEL – FPGA .....	82
5.2 FERRAMENTAS DE SÍNTESE E IMPLEMENTAÇÃO .....	83
5.2.1 Ferramenta de Síntese Lógica.....	83
5.2.2 Ferramenta de Síntese Física .....	84
5.2.3 Gerador da Configuração do FPGA .....	85
5.2.4 Tradutor do Descritor de Hardware.....	85
5.2.5 Compilador do Software .....	86
5.3 GLIFA – GERADOR DE LISTA DE FALHAS .....	86
5.4 SOCIF – SYSTEM-ON-A-CHIP INJETOR DE FALHAS .....	92
5.4.1 Arquitetura – Hardware .....	92
5.4.1.1 Controlador da SRAM .....	93
5.4.1.2 Controlador do ICAP .....	95

5.4.1.3 Controlador do <i>CompactFlash</i> .....	95
5.4.1.4 Controlador do Tempo de Teste .....	95
5.4.1.5 Controlador das Portas do SoCIF .....	96
5.4.1.6 Controlador de Interrupção.....	96
5.4.1.7 Controlador da Comunicação Serial.....	97
<b>5.4.2 Funcionalidade – Software.....</b>	<b>97</b>
5.4.2.1 Inicializar Módulos.....	99
5.4.2.2 Carregar Lista de falhas.....	100
5.4.2.3 Injetar as Falhas.....	100
5.4.2.4 Gravar os Resultados.....	101
<b>6 FLUXO DA PLATAFORMA DE EMULAÇÃO - PLAESER ...</b>	<b>103</b>
6.1 PREPARO DO DUT .....	104
6.2 INTEGRAÇÃO .....	105
6.3 IMPLEMENTAÇÃO.....	108
6.4 GERAÇÃO DA LISTA DE FALHAS .....	110
6.5 INJEÇÃO DAS FALHAS .....	112
6.6 LEITURA DOS RESULTADOS .....	113
6.7 LIMITAÇÕES.....	114
<b>7 RESULTADOS OBTIDOS .....</b>	<b>115</b>
7.1 EXEMPLO – CONTADOR .....	115
7.1.1 Contador com TMR com a Lógica de Carry (Original).....	118
7.1.2 Contador sem TMR com a Lógica de Carry .....	122
7.1.3 Contador com TMR sem a Lógica de Carry .....	125
7.1.4 Contador sem TMR e sem a Lógica de Carry .....	127
7.2 PROCESSADOR LEON3 COM MONITOR DE BARRAMENTO.....	129
<b>8 TRABALHOS FUTUROS .....</b>	<b>139</b>
<b>9 CONCLUSÃO .....</b>	<b>141</b>
<b>REFERÊNCIAS.....</b>	<b>143</b>
<b>APÊNDICE A – (PSEUDOCÓDIGO) .....</b>	<b>151</b>

## 1 INTRODUÇÃO

Sistemas digitais estão presentes em todo lugar do nosso cotidiano, desde aparelhos domésticos como microondas e lavadoras de roupas até aplicações mais complexas como componentes de automóveis, sistemas de controle de aeronaves e equipamentos médicos. Esses sistemas digitais proporcionam mais produtividade e flexibilidade ao dia a dia, mas é de conhecimento comum que eles não estão livres de falhas. Algumas dessas falhas podem ser atribuídas a imperfeições durante a fabricação, enquanto outras são de natureza externa como defeitos de produção ou estresse causado pelo ambiente. Além disso, a miniaturização dos dispositivos aumenta a incidência de erros transientes (*Soft Errors*) e conseqüentemente diminui a confiabilidade do sistema (1). Logo, sistemas digitais utilizados em aplicações críticas ou ligados diretamente à segurança do ser humano requerem um alto grau de confiabilidade.

Tolerância a falhas em semicondutores ganhou mais importância desde que começaram a ser observadas anomalias no comportamento de aplicações espaciais (2). De lá pra cá, o constante avanço em tecnologias de fabricação fez com que os circuitos integrados (CI) atingissem a escala nanométrica. As características desses CIs, por exemplo, a dimensão de transistores e frequência de operação, os qualificam como potencialmente sensíveis a perturbações causadas por partículas encontradas na atmosfera terrestre ou até no nível do mar. A grande presença de circuitos integrados em diversas áreas faz com que a questão de tolerância a falhas transientes, que era uma preocupação restrita de aplicações críticas, principalmente espaciais, ganhe cada vez mais importância no desenvolvimento de dispositivos terrestres.

Nesse contexto, erros induzidos por radiação são uma ameaça crescente que acompanha o avanço da tecnologia de fabricação de CIs e por isso diversos esquemas de tolerância a falhas vêm sendo desenvolvidos para enfrentar esse desafio. Portanto, a avaliação dessas técnicas de tolerância a falhas diante do acontecimento de *soft errors* tem um papel fundamental. O uso de *benchmarks* ou de métodos convencionais de teste para análise de desempenho de sistemas não se aplicam para avaliação de confiabilidade. No entanto, é preciso mecanismos específicos que permitam observar o comportamento do sistema na presença de falhas (1). Contudo, esse tipo de avaliação só é viável se houver técnicas que consigam acelerar artificialmente o acontecimento dessas falhas, considerando que para dispositivos

eletrônicos os valores típicos da taxa de incidência de *soft errors* são na ordem de anos (3).

Injeção de falhas surge como uma solução viável e tem sido profundamente explorada em pesquisas e pela indústria para a análise de confiabilidade de sistemas (4). Diversas técnicas são utilizadas para prover experimentos através da injeção de falhas. Entre essas técnicas se encontra a injeção de falhas baseada em prototipação (5)(6) que utiliza um circuito reconfigurável FPGA (do inglês, *Field Programmable Gate Arrays*) para emular o acontecimento das falhas.

A pesquisa desenvolvida nesse trabalho propõe uma plataforma de emulação de *soft errors* que busca possibilitar a análise experimental de técnicas de tolerância a falhas aplicadas em projetos de circuitos críticos. A plataforma proposta, chamada PLAESER, utiliza recursos de reconfiguração específicos de FPGAs da Xilinx para permitir a injeção de falhas de forma menos intrusiva. O trabalho desenvolvido visa definir um fluxo que permita a análise experimental da técnica de tolerância a falhas ainda no início do projeto do circuito através de uma prototipação rápida. O quanto antes essa análise puder ser feita no projeto, menor será o tempo de locação de instalações para o teste com aceleradores de partículas que custam em média US\$ 100K por dia (7).

## 1.1 MOTIVAÇÃO

A inserção do Brasil no grupo de países auto-suficientes em relação à tecnologia aeroespacial, ou seja, com capacidade de desenvolvimento e do lançamento de satélites passa impreterivelmente pela formação de recursos humanos na área de efeitos causados por radiações. Essa formação é fundamental para possibilitar a fabricação nacional de circuitos robustos ao ambiente espacial e que atendam às normas já existentes como as da Agência Espacial Européia (ESA) para qualificação desses circuitos. Além disso, estudos sobre técnicas de tolerâncias aplicadas a dispositivos COTS (do inglês, *Commercial Off The Shelf*) são importantes para reduzir o impacto do embargo sofrido pelo Instituto Nacional de Pesquisa Espacial (INPE), e pelo Brasil em geral, na aquisição de componentes tolerantes a radiação. Esse embargo, devido à restrição governamental que regula o comércio internacional de armas (ITAR – *International Traffic in Arms Regulation*) faz com que a aquisição de dispositivos para área espacial seja custosa em tempo e dinheiro, e também, com tecnologias defasadas (8).

Nesse contexto, a plataforma de emulação de falhas proposta nesse trabalho pode ser empregada como uma das etapas do fluxo de

teste de projeto para o desenvolvimento nacional de circuitos eletrônicos tolerantes à radiação. Além disso, a constante miniaturização dos dispositivos concomitantemente ao aumento da frequência de operação e a diminuição da tensão de alimentação ampliam a suscetibilidade à radiação dos dispositivos. A previsão feita em (9) mostra uma barreira em que a taxa de erros devido aos efeitos da radiação inviabilizará a criação de CIs sem alguma técnica de tolerância, mesmo para aplicações operando em nível do mar. Dessa forma, a plataforma de emulação de falhas proposta nesse trabalho busca proporcionar ao desenvolvedor de CIs uma ferramenta para auxiliar na escolha de uma técnica de tolerância a falhas adequada, através de uma análise experimental guiada por um fluxo de prototipação rápida.

O uso de FPGAs tornou-se popular na verificação de circuitos de aplicação específica (3). Aliado a isso, a flexibilidade e a capacidade de reconfiguração parcial dos FPGAs motivaram o uso desse tipo de dispositivo na proposta da PLAESER para dar suporte à injeção de falhas em circuitos visando à implementação tanto para ASIC (do inglês, *Application Specific Integrated Circuit*) quanto para FPGAs.

## 1.2 OBJETIVOS

O objetivo principal deste trabalho é realizar um estudo das técnicas de injeção de falhas existentes, além de definir uma plataforma de emulação de *soft errors*. Além disso, pretende-se que o uso da plataforma seja guiado através do fluxo proposto, que procura permitir a análise experimental de um circuito supostamente tolerante a falhas. Ainda, deseja-se que através da prototipação rápida seja possível aplicar o fluxo da plataforma PLAESER em etapas iniciais de projetos de circuitos tolerantes buscando diminuir o custo de projeto gasto com o tempo de locação de instalações para verificação através de radiação induzida.

Os objetivos específicos deste trabalho abrangem diversas áreas de conhecimento. Esses objetivos são:

- Aprender conceitos de suscetibilidade de circuito diante a radiação;
- realizar um levantamento das técnicas de injeção de falhas existentes e descobrir as suas características de forma a possibilitar a escolha de uma técnica para ser implementada;
- implementar uma plataforma de emulação de *soft errors* para dar suporte ao fluxo de injeção de falhas proposto;

- avaliar experimentalmente casos de teste utilizando a plataforma desenvolvida guiada pelo fluxo proposto;
- descrever o conhecimento adquirido durante o desenvolvimento do projeto com o objetivo de detalhar as adversidades encontradas devido a não documentação (proposital) de recursos existentes nos FPGAs da Xilinx.

### 1.3 ORGANIZAÇÃO DO TEXTO

O presente trabalho está organizado da seguinte forma: nas seções 2 e 3 são apresentados conceitos básicos e uma revisão das técnicas de injeção de falhas necessários para o entendimento deste trabalho; na seção 2 são abordados conceitos que caracterizam uma falha que se origina de um dos efeitos causados pela radiação. Na seção 3 são apresentadas diversas técnicas de injeção de falhas e algumas dessas são discutidas através do detalhamento de exemplos do estado da arte.

As seções 4, 5 e 6 apresentam o trabalho desenvolvido e as contribuições desse trabalho. A seção 4 mostra a investigação da arquitetura do FPGA utilizado para a emulação de falhas. Nessa seção são destacadas as dificuldades encontradas devido a não documentação de certos recursos do FPGA. O conhecimento adquirido nessa investigação é detalhado de forma a gerar uma documentação desses recursos.

A plataforma de emulação de *soft errors* (PLEASER) desenvolvida é descrita na seção 5. Cada componente que integra a plataforma PLEASER é detalhado de forma a ajudar o entendimento do fluxo de emulação de falhas proposto.

O fluxo proposto é seguido na seção 6 com o objetivo de descrever cada etapa. O mesmo fluxo é aplicado em casos de teste e seus resultados são mostrados na seção 7.

O direcionamento para trabalhos futuros e as conclusões são apresentadas nas seções 8 e 9. Na seção 8 são apresentadas diversas sugestões de melhorias a serem feitas para que a plataforma PLEASER possa se consolidar como uma ferramenta de análise experimental de técnicas de tolerância a falhas. Nessas sugestões se encontram inúmeras oportunidades de trabalhos de pesquisas que podem ser realizados.



## 2 CONCEITOS BÁSICOS

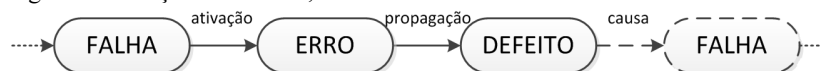
Conceitos básicos relacionados ao desenvolvimento deste trabalho são apresentados nesta seção. As definições relativas a confiabilidades utilizadas nesse trabalho estão de acordo com (10).

### 2.1 CONFIABILIDADE

O serviço disponibilizado por um sistema se refere ao comportamento desse sistema quando utilizado pelo usuário. A especificação desse sistema define a sua função. O serviço entregue por esse sistema é dito correto quando esse serviço implementa exatamente a função para qual o sistema foi designado. O termo confiabilidade se refere à capacidade do sistema de prover um serviço de forma correta. Um defeito no sistema é o evento que ocorre quando o serviço entregue diverge do seu funcionamento correto, ou seja, um sistema é dito com defeito quando é incapaz de prover um serviço como especificado.

O sistema é regido por uma série de elementos internos que controlam os resultados gerados e a sequência desses resultados representa o serviço prestado pelo sistema. Um defeito no serviço corresponde à variação em um ou mais resultados dessa sequência. Essa variação é derivada de algum erro nos elementos que controlam o sistema. A causa do erro em um elemento do sistema é chamada de falha. Portanto, o erro é a manifestação de uma falha que provocou a interrupção na geração correta de resultados desse elemento. Esse erro conduzirá a um defeito no sistema se o elemento errôneo for requisitado durante a interrupção do seu funcionamento correto. É importante perceber que muitos erros não terão influência no resultado do sistema, ou seja, nem todos os erros acarretarão defeitos do sistema. A Figura 1 mostra a relação entre, falha, erro e defeito.

Figura 1 - Relação entre falha, erro e defeito.



Fonte: AVIZIENIS, A. et al. (2004) (10).

Com relação a sistemas que requerem uma alta confiabilidade, foram desenvolvidas diversas técnicas ao longo dos últimos cinquenta anos com o objetivo de atingir os requisitos de projeto de sistemas confiáveis. Essas técnicas podem ser divididas em quatro grandes grupos, que são:

- prevenção de falhas – busca impedir a ocorrência ou a introdução de falhas através de metodologias de projeto e pela escolha de componentes com tecnologia apropriada;
- tolerância a falhas – visa prover o serviço de forma correta mesmo com a presença de falhas e para isso utiliza técnicas como: mascaramento, confinamento, recuperação do sistema, tratamento da falha, entre outros;
- remoção de falhas – verifica a presença de falhas com o objetivo de reduzir o número de ocorrências e a severidade das falhas;
- previsão de falhas – estima o número atual e a incidência futura de falhas de forma a prever as consequências.

A especificação de sistemas que implementam diversas funções pode aceitar que o sistema permaneça operando mesmo que com defeito em algumas de suas funções. Dessa forma é mais interessante que o sistema continue operando ainda que de forma degradada, com limitações no serviço, mais lento ou em modo de emergência por exemplo. Nesses tipos de sistema, as técnicas de remoção e de previsão de falhas são mais aplicadas visando alcançar a qualidade de serviço e a disponibilidade. As técnicas de prevenção e tolerância são empregadas em sistemas críticos, onde defeitos no serviço entregue podem ser catastróficos.

Particularmente para o projeto de hardware em aplicações espaciais o aspecto de confiabilidade requer atenção especial. A confiabilidade nesses casos geralmente é relacionada à habilidade do circuito tolerar falhas induzidas pelo ambiente que podem conduzir a defeitos no sistema (11). Nesse contexto espacial, a falha é definida como um mau comportamento de um componente interno do sistema. A ativação da falha pode propagar para a saída desse componente, o resultado desse comportamento, e dessa forma, acarretar em um erro. Finalmente, um defeito ocorre se os resultados errados gerados por esse componente interrompem o correto funcionamento do serviço prestado pelo sistema.

Em aplicações espaciais críticas um defeito pode provocar um término prematuro da missão. Contudo, falhas e erros podem ser tolerados dependendo das especificações do sistema, desde que defeitos possam ser mascarados para evitar que aconteçam ou possam ser detectados de forma a colocar o sistema em um estado seguro. Nesse

contexto, diversas técnicas de tolerância se aplicam, mas antes disso é preciso analisar a origem das falhas.

As falhas podem ser classificadas de diversas formas, que dependem do critério considerado, como mostra a lista abaixo:

- momento – na fase de desenvolvimento do sistema ou durante operação;
- origem – interna ou externa ao sistema;
- fenômeno – natural ou provocada por um ser humano;
- dimensão – de hardware ou de software;
- objetivo – maliciosa ou sem propósito malicioso;
- intenção – deliberado a partir de uma decisão ruim ou não;
- forma – acidental ou por mau uso;
- persistência – transiente ou permanente.

No contexto de aplicações espaciais, as técnicas de tolerância geralmente visam falhas que ocorrem devido à radiação do ambiente em que o sistema opera. A persistência é o critério mais utilizado para classificação de falhas induzidas por radiação uma vez que basicamente todas as falhas desse tipo são externas ao hardware e acontecem de forma natural durante a operação. Os erros provenientes de falhas permanentes são chamados de *Hard Errors*, enquanto que, para falhas transientes os erros são chamados de *Soft Errors*, sendo que estes últimos geralmente são revertidos com a reinicialização do sistema.

## 2.2 RADIAÇÃO E SEUS EFEITOS

As falhas introduzidas no sistema pelo ambiente podem ser causadas não somente pela radiação, mas podem ser também de outra natureza. Por exemplo, falhas mecânicas podem ser originadas por vibrações e choques ou até alterações de temperatura ou pressão. Falhas elétricas podem ser causadas por interferência eletromagnética (EMI – *Electromagnetic Interference*) ou por descargas eletrostáticas (ESD – *Electrostatic Discharge*). Além disso, o envelhecimento do dispositivo pode também causar falhas. Contudo, a radiação é um dos aspectos naturais mais críticos para as tecnologias modernas de dispositivos eletrônicos (11). Dessa forma, será dada mais atenção à radiação.

A radiação pode ser generalizada como um grupo de partículas carregadas ou não que podem interagir com um dispositivo eletrônico através da troca de energia. No ponto de vista da presença de radiação, o espaço é o ambiente mais inóspito. Muitos tipos de radiação e de

partículas existem no espaço e são basicamente geradas por reações nucleares das estrelas. As partículas se movem facilmente no vácuo, mas quando chegam perto da Terra essas partículas se chocam com átomos e moléculas de oxigênio, nitrogênio e outros gases presentes na atmosfera. Dessa forma, as partículas perdem grande parte de sua energia e acabam dando origem a outras partículas. Logo, a atmosfera atua como um escudo natural da Terra que nos protege da radiação do espaço. Conseqüentemente, circuitos eletrônicos destinados a aplicações aeroespaciais estão mais expostos à radiação do que aqueles que operam em nível do mar, que estão sujeitos aos efeitos de partículas que não perderam toda sua energia.

Entre os diversos tipos de radiação presentes no espaço, as principais são os raios cósmicos, partículas subatômicas (*mesons*) e partículas alfa. As partículas com mais potencial para causar falhas presente nos raios cósmicos galácticos, também chamados de GCRs (do inglês, *Galactic Cosmic Rays*), são os núcleos de íons pesados altamente energizados. A origem desse tipo de radiação ainda não foi verdadeiramente comprovada (12). O outro tipo de radiação que pode causar o defeito de um sistema se chama *meson* que é produzido basicamente pela interação de partículas altamente energizadas dos raios cósmicos com a atmosfera terrestre. Por último, as partículas alfa são produzidas do decaimento de elementos radioativos, e possuem bem menos energia se comparadas com um íon pesado, e por isso, dificilmente conseguem vencer ao empacotamento de um CI a ponto de produzir uma falha. Contudo, outras partículas alfa podem ser produzidas pela interação de outras partículas, como nêutrons ou prótons, com o silício do dispositivo e dessa forma podem ser geradas dentro do próprio dispositivo e assim podendo acarretar em uma falha.

Todos esses tipos de radiação podem viajar pelo espaço e acabar sendo presas em campos magnéticos. As partículas que viajam perto do campo magnético em volta da Terra e ficam presas formam uma nuvem radioativa chamada de cinturão de Van Allen. As erupções gigantes de gases e plasma na superfície do Sol produzem uma distorção forte nos campos magnéticos em volta da Terra e com isso aumentando consideravelmente a absorção de raios cósmicos (11). A Figura 2 ilustra o cinturão de Van Allen.

As radiações que chegam ao nível do solo são geradas basicamente da interação dos raios cósmicos com a atmosfera terrestre. Existem fontes radioativas na Terra, como a crosta terrestre que emite partículas alfa, mas são insignificantes. As partículas primárias que compõem os raios cósmicos quando atingem a atmosfera perdem muito

da sua energia e produzem partículas secundárias que são divididas, principalmente, em três grupos: *hádrons*, *múons* e elementos eletromagnéticos (elétrons, pósitrons e fótons). Dependendo da sua origem essa partícula pode viajar por uma determinada distância até interagir com novos átomos ou atingir o solo. Mesmo que as partículas percam continuamente suas energias, o fato da miniaturização da tecnologia faz com que dispositivos eletrônicos sejam cada vez mais susceptíveis. Em (13) são mostrados alguns casos em que raios cósmicos causaram *soft errors* em aplicações operando na terra.

Figura 2 - Ambiente radioativo espacial.



Quando uma partícula primária incide em um material, ela percorre um caminho aleatório que é determinado pela colisão contra os núcleos do material que desviam essa partícula fazendo-a perder parte da sua energia e produzir partículas secundárias. O termo frequentemente utilizado para descrever a energia transferida para o material, ou seja, a energia que a partícula perde a cada segmento da sua trajetória ao passar pelo material é representado pela sigla LET (do inglês, *Linear Energy Transferred*). A ionização é o mecanismo pelo qual certa carga é liberada internamente no material atravessado pela partícula. Essa ionização pode ser direta, ou seja, a partícula primária já é carregada ou indireta se a partícula primária não é carregada, mas a sua interação com esse material gera novas partículas carregadas. A ionização direta acontece principalmente devido a íons pesados, elétrons, pósitrons e partículas alfa, que quando entram no material podem arrancar elétrons de átomos neutros ou conceder elétrons para átomos ionizados, e dessa forma, provocar o movimento de carga ao

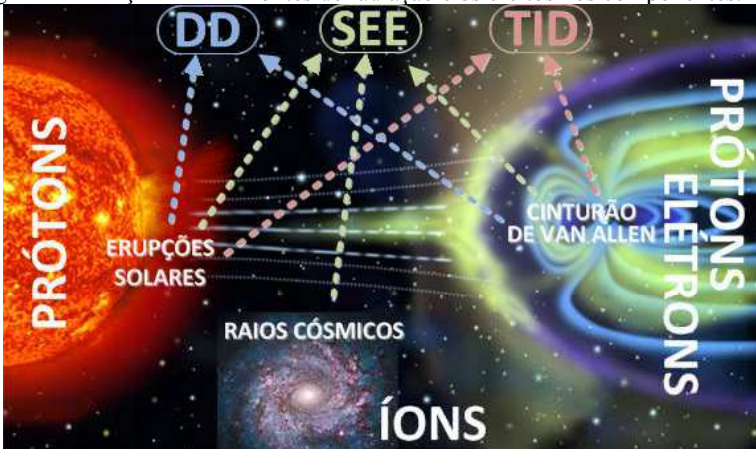
longo da trajetória da partícula. A ionização indireta é derivada basicamente de nêutrons e fótons.

Quando um íon pesado atravessa um semicondutor, ele pode modificar momentaneamente a sua condutividade através da criação de pares elétron/lacuna. Isso pode ocasionar uma concentração de carga em uma determinada região. Caso essa região seja eletricamente ativa pode haver a alteração das características básicas do funcionamento do circuito.

A quantidade de energia depositada no circuito pela partícula define dois efeitos. O efeito cumulativo onde todas as partículas contribuem para uma dose ionizante total absorvida pelo circuito que é chamado de TID (do inglês, *Total Ionization Dose*). As partículas com baixo LET contribuem mais para esse efeito acumulativo uma vez que partículas com alto LET geram uma dose concentrada e causam um efeito diferente que é chamado de efeito de evento único ou SEE (do inglês, *Single Event Effect*). O SEE está relacionado ao fato de uma partícula de alto LET, por exemplo, um íon pesado, depositar grande quantidade de carga em um pequeno volume de material.

Diferentemente de TID e SEE que se referem à carga depositada no dispositivo, o dano por deslocamento DD (do inglês, *Displacement Damage*) ocorre quando uma partícula pesada adentra um semicondutor e consegue alterar o arranjo dos átomos da estrutura cristalina do silício causando um dano duradouro, afetando assim, propriedades das junções do semicondutor. Esse dano pode ser revertido através de técnicas especiais que utilizam temperaturas altas para fornecer energia suficiente para a rede cristalina se recombinar e voltar ao equilíbrio.

Figura 3 - Relação entre as fontes de radiação e os efeitos nos componentes.



A Figura 3 mostra um resumo da relação entre a origem da radiação e o efeito causado nos componentes. A radiação do cinturão de Van Allen, que é composto de prótons e elétrons altamente energizados, contribui para todos os tipos de efeitos, igualmente aos prótons energizados provenientes das erupções solares. Os raios cósmicos são compostos de íons altamente energizados capazes de induzir SEE, mas que são raros e não possuem quantidade suficiente para contribuir para a degradação do componente (12).

### 2.3 MODELOS DE FALHAS

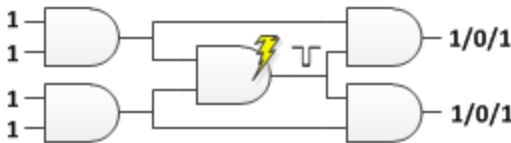
Os efeitos físicos do impacto da radiação em materiais são fenômenos muito complexos e requerem modelos matemáticos e estatísticos específicos. As simulações desses modelos são altamente custosas em tempo devido à complexidade e à quantidade de informação envolvida. Esse quadro agrava extremamente quando consideramos circuitos construídos com as tecnologias modernas que integram bilhões de transistores. Contudo, comportamentos comuns entre os circuitos afetados podem ser percebidos. Esses comportamentos estão ligados ao funcionamento do circuito e não aos princípios físicos do fenômeno da radiação. Nesse contexto, são introduzidos os modelos de falhas que estão relacionados ao comportamento do circuito, ou melhor, à mudança de comportamento quando o circuito é afetado por radiação.

O endurecimento de CIs comerciais, em geral, perante os efeitos duradouros, tem melhorado nos últimos anos como descrito em (12). O contrário tem acontecido com a sensibilidade dos dispositivos diante aos efeitos radioativos transientes, que tem se agravado devido à miniaturização da tecnologia de dispositivos. Dessa maneira, serão apresentados os modelos de falhas do tipo SEE. Esses efeitos podem ser divididos entre *Hard Errors*, que correspondem a falhas destrutivas, e *Soft Errors*, onde o efeito temporário da falha muitas vezes desaparece, principalmente, por meio da simples reinicialização do dispositivo.

- *Soft Errors*
  - *Single Event Upset (SEU)*
  - *Multiple Cell Upset (MCU)*
  - *Single Event Transient (SET)*
  - *Single Event Functional Interrupt (SEFI)*
- *Hard Errors*
  - *Single Event Latch-up (SEL)*
  - *Single Event Gate Rupture (SEGR)*

O efeito chamado de SET é causado quando uma partícula atinge um circuito combinacional do dispositivo. Durante um período de pico/nano segundos a carga coletada pelo campo elétrico injeta elétrons ou lacunas dependendo da polarização do campo elétrico do transistor. Esse fenômeno pode causar um *glitch* na tensão de saída do transistor que é chamado de SET. A quantidade mínima de carga para induzir um SET é chamada de carga crítica. O comportamento lógico de um SET é definido como uma transição dupla ( $0 \rightarrow 1 \rightarrow 0$ ) ou ( $1 \rightarrow 0 \rightarrow 1$ ). A Figura 4 apresenta um exemplo de um SET onde mostra um circuito lógico atingido por uma partícula que ocasiona um *glitch* na saída da porta lógica do centro da figura. O efeito do SET se propaga para a saída do circuito.

Figura 4 - Exemplo de um SET.



O efeito causado por uma partícula que atinge um elemento de memória e inverte o seu estado é chamado de SEU. SEUs não são considerados permanentes porque uma vez que a próxima operação de gravação for executada o valor errado será sobrescrito. Contudo, em casos que o elemento de memória é utilizado somente para leitura pelo sistema então o efeito desse SEU é permanente. Dependendo do tipo do dispositivo, SEUs também podem ser obtidos quando o efeito de um SET se propaga até o próximo elemento de memória e acaba sendo armazenado. Quando mais de um SEU ocorre em um circuito então isso é chamado de MCU e nesses casos se os múltiplos elementos de memórias afetados fazem parte de um registrador maior que armazena uma palavra então pode ser chamado de MBU (do inglês, *Multiple Bit Upset*).

O efeito SEFI acontece quando a partícula atinge um elemento de controle funcional do dispositivo de forma que o seu efeito acaba sendo global. Os SEFIs geralmente estão relacionados a erros temporários que afetam ao sinal de reinicialização ou sinal que desabilita a escrita de uma memória interrompendo a sua funcionalidade.

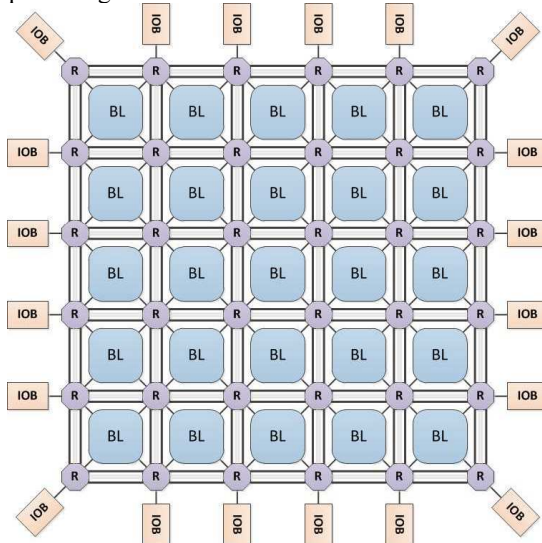
Os efeitos destrutivos de SEEs, que são SEL e SEGR, podem aumentar o fluxo de corrente ou até causar um curto-circuito e dessa forma gerando um sobreaquecimento no dispositivo e podendo até queimá-lo.



## 2.4 CIRCUITOS RECONFIGURÁVEIS

Os circuitos reconfiguráveis, no caso os FPGAs, fazem parte de uma família de dispositivos utilizados para implementação de hardware personalizável. A ação de reconfigurar o FPGA significa mudar a sua funcionalidade de forma a suportar uma nova aplicação. A possibilidade de ter um hardware configurável é o que faz os FPGAs serem tão populares. Softwares que podem ser seccionados para serem executados de forma paralela fazem do FPGA uma ótima solução para implementar o algoritmo em hardware de forma a melhorar o tempo de execução. Em computação de alto desempenho, principalmente no processamento de imagens em que o paralelismo dos algoritmos pode ser explorado se implementado em hardware, os FPGAs são muito visados (14). Dada a capacidade de lógica que pode ser sintetizada pelos modelos de FPGAs atuais, um SoC (do inglês, *System-on-a-Chip*) inteiro pode ser implementado dentro desses dispositivos reconfiguráveis. Além disso, os maiores fabricantes de FPGAs lançaram SoCs customizáveis com processadores ARM integrados em um FPGA (15)(16)(17).

Figura 5 - Arquitetura genérica de um FPGA.

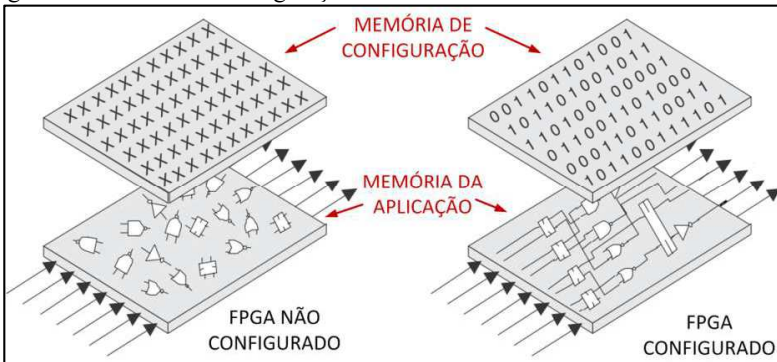


A arquitetura genérica de um FPGA é mostrada na Figura 5. Os FPGAs possuem três elementos básicos que são os Blocos Lógicos (BL), os Blocos de Entrada/Saída (IOB) e os recursos de comunicação (R). Os BLs são os blocos principais do FPGA e contém geradores de função lógica e elementos de armazenamento. Os BLs são configurados

para implementar o circuito combinacional/seqüencial. O IOB (do inglês, *Input/Output Block*) tem a função de interconectar um sinal interno em um pino do encapsulamento do FPGA. Cada pino do FPGA possui um IOB correspondente que pode ser configurado para determinar o sentido da porta se é de entrada, saída ou bidirecional. Os recursos de interconexão (R) são como roteadores programáveis que permitem a conexão entre BLs e IOBs.

O arquivo de configuração do FPGA é chamado de *bitstream*. O carregamento do *bitstream* no FPGA é que define a sua funcionalidade como ilustra a Figura 6. Os tipos de FPGAs mais comuns são os *antifuse*, que são programados somente uma vez, e os baseados em memórias, que podem ser do tipo SRAM ou Flash (18). Considerando isso, os trabalhos sobre efeitos de evento único (SEE) em FPGA geralmente utilizam o termo SEU mesmo quando outros efeitos são analisados como, por exemplo, em (19). É importante ressaltar aqui que esse trabalho utiliza a reconfiguração da memória de configuração do FPGA para inverter a lógica de elementos do circuito com o objetivo de emular SETs.

Figura 6 - Memória de configuração do FPGA.



Os arquivos de configuração dos diversos fabricantes de FPGAs permanecem obscuros por motivos políticos, de segurança e principalmente, devido à propriedade intelectual dos dispositivos. Esses motivos são exaustivamente discutidos em (20).

### 3 INJEÇÃO DE FALHAS (TRABALHOS RELACIONADOS)

A injeção de falhas em hardware é peça fundamental na análise do comportamento de circuitos na presença de falhas, pois é uma abordagem amplamente aceita (3). As técnicas de injeção de falhas buscam providenciar informações sobre a confiabilidade do circuito sendo testado, aqui chamado de DUT (do inglês, *Design Under Test*). Essas informações podem ser sobre: a) validação do cumprimento dos requisitos de confiabilidade; b) detecção de pontos fracos nas técnicas de tolerância a falhas adotadas; c) previsão do comportamento do DUT na presença de falhas.

O termo hardware utilizado juntamente de injeção de falhas, como em *injeção de falhas em hardware*, está relacionado a todas as técnicas que buscam simular o efeito da variação do comportamento do dispositivo na presença de uma falha. Entre essas técnicas existem aquelas aplicadas, principalmente, a sistemas processados, em que alterações no software são feitas com o objetivo de simular uma má execução do sistema devido ao acontecimento de uma falha no hardware do dispositivo. Dessa forma, é importante haver uma distinção entre injeção de falhas para a análise do comportamento do hardware e do software, sendo que este último está fora do escopo desse trabalho.

As técnicas de injeção de falhas não se limitam a forma de injeção propriamente dita. As técnicas envolvem o processo completo necessário para a injeção das falhas. Isso engloba todo o ambiente requerido para inicialização do DUT, a seleção da carga de trabalho apropriada, a captura dos dados pertinentes ao comportamento com falha do DUT, a comparação com os dados do comportamento do DUT livre de falhas, a classificação dos efeitos sofridos pelo DUT e o monitoramento de todo o processo.

O método de injeção de falhas depende do tipo de DUT que está sendo testado. No caso de memórias, o efeito a ser analisado pela injeção de falhas será predominantemente SEUs, enquanto SoCs, por serem circuitos mais complexos, requerem uma injeção de falhas mais específica. Outro fator a ser considerado é a possibilidade de se provocar falhas reais (físicas) no dispositivo ou se somente modelos de falhas (lógicas) serão aplicados. O nível de abstração do tipo de falhas a serem injetadas está relacionado diretamente ao DUT que pode ser um COTS, um protótipo ou um modelo do projeto. A intrusão no DUT é outra questão levantada durante a escolha do método de injeção de falhas, uma vez que a análise deve ser feita com o circuito operando o mais próximo possível do que será a sua aplicação final. Finalmente, o

resultado esperado devido à técnica de injeção escolhida irá avaliar a robustez do circuito perante a presença de falhas. Tipicamente, no contexto de radiação, os resultados das campanhas de falhas são apresentados em valores de FIT (do inglês, *Failures In Time*) que representa o número de vezes que o circuito apresentou defeito em um bilhão de horas ( $10^9$ h) de operação.

Essa seção faz um apanhado das diversas técnicas de injeção de falhas existentes. As técnicas são apresentadas em dois grandes grupos, ou seja, as que trabalham com injeção física de falhas e as técnicas com abordagem no âmbito do mau comportamento funcional causado pela falha. Entre essas técnicas é dada uma atenção especial à emulação de falhas que foi utilizada para o desenvolvimento desse trabalho.

### 3.1 TÉCNICAS DE INJEÇÃO FÍSICA DE FALHAS

Os métodos de injeção física de falhas utilizam fontes externas para permitir ensaios com radiação, ruídos eletromagnéticos e envelhecimento dos CIs. O objetivo desses testes é analisar a robustez de COTS ou qualificar um protótipo como “endurecido” perante os efeitos causados pela radiação. Alguns dos métodos de injeção física de falhas são mostrados a seguir.

#### 3.1.1 Método por Radiação

A radiação cósmica é a fonte principal de SEE em CIs, portanto, realizar testes com o dispositivo em elevadas altitudes ou até no espaço é a forma mais realística de avaliar a sensibilidade à SEE dos CIs. Considerando a baixa probabilidade de erros, seriam necessárias centenas de milhares de amostras do DUT para se obter uma medição válida. Portanto, o tempo e o custo dessa abordagem as tornam inviáveis, principalmente para projetos que visam o lançamento do produto no mercado. Dessa forma, aceleradores do efeito radioativo são utilizados para qualificar os produtos. Esses aceleradores se encontram em instalações específicas para esse tipo de teste e realizam testes que duram horas ou dias em poucas amostras do DUT (3). Uma lista dessas instalações é encontrada em (21).

Nesses tipos de teste são utilizados diversos tipos de partículas com valores de LET distintos visando causar efeitos diferentes. Os tipos de partículas e os valores de LET típicos utilizados para gerar SEE, TID e DD são detalhados em (12). Muitos desses valores que caracterizam o teste do circuito são padronizados de acordo com a aplicação final do

dispositivo. Essa padronização é geralmente feita por agências espaciais ou comitês como o JEDEC que criou o padrão JESD89 (21), onde define os requerimentos e procedimentos para teste da taxa de *soft errors*, chamada de SER (do inglês, *Soft Error Rate*), para CIs utilizados em aplicações terrestres, ou seja, desde aeronaves até dispositivos operando na superfície da Terra. Experimentos realizados com FPGAs da Xilinx com características descritas no padrão JESD89 são apresentados em (22).

### **3.1.2 Método por Raios Laser**

A injeção de falhas através de raios laser é semelhante ao método que utiliza íons pesados no sentido em que o feixe é aplicado diretamente na superfície do silício. No entanto, o raio laser é muito mais preciso, e dessa maneira é possível injetar falhas em lugares específicos de forma muito mais controlada. A incidência do raio laser no silício pode causar efeitos semelhantes àqueles provocados por partículas dos raios cósmicos. Esse método é geralmente associado à mudança do estado de elementos do circuito, para o teste do efeito do tipo SEU. Os trabalhos (23) e (24) mostram os resultados obtidos através da análise de SEE com a injeção física de falhas por raios laser em FPGAs.

Com a ajuda de um microscópio especial, a utilização de raio laser para injeção de falhas provê alta acessibilidade no sentido de localizar elementos do circuito de forma não intrusiva. Desse modo, a injeção de falhas por raio laser utiliza equipamentos mais baratos do que instalações para ensaios de radiação. Além disso, o método por raios laser precisa de um ambiente bem mais simples para a injeção de falhas, pois, não é necessário isolar o DUT, por exemplo, para que os componentes da periferia do DUT que não estão sendo testados não sofram as perturbações causadas pela radiação, o que comprometeria o resultado da análise.

### **3.1.3 Método de Injeção por Pinos**

Diferentemente das outras técnicas de injeção física de falhas apresentadas, o método por indução de valores nos pinos do DUT requer contato físico entre a plataforma de teste e o DUT. O método busca replicar o efeito de uma falha natural através da mudança forçada do valor lógico de um pino do CI. Considerando a complexidade de sistemas modernos a injeção de falhas por pinos fica muito limitada

quanto ao poder de acessibilidade do método. Dessa forma, ela é mais utilizada para analisar diversos efeitos nos terminais do circuito.

O método de injeção de falhas através dos pinos muitas vezes é empregado por ferramentas de teste em conjunto com outras técnicas com o objetivo de ampliar os resultados. Soluções como MESSALINE, MARS, FIST, RIFLE descritas em (25) empregam funcionalidades do método de injeção de falhas nos pinos do circuito.

### 3.2 TÉCNICAS DE INJEÇÃO DE FALHAS NO NÍVEL LÓGICO

Os métodos de injeção física de falhas disponibilizam valores realísticos da taxa de SER e são amplamente utilizados para qualificação de dispositivos como “endurecidos” para aplicações críticas em ambientes hostis como o espaço. Contudo, as instalações com aceleradores de nêutrons, por exemplo, cobram na faixa de trezentos até mil dólares por hora de exposição (26), ou seja, injeção física de falhas é extremamente cara e dessa forma é preciso soluções de análise que possam ser aplicadas mais cedo no projeto de circuitos robustos.

Os métodos de injeção de falhas no nível do funcionamento do circuito exploram recursos lógicos disponíveis para poder inserir o efeito que uma falha em hardware provocaria. Geralmente, esses recursos lógicos têm finalidades diferentes do que a da injeção de falhas, por exemplo, o padrão 1149.1 da IEEE que provê acesso a cadeias de *scan* do circuito ou sistemas processados com recursos de depuração OCD (do inglês, *On-Chip Debug*) que permitem a observação de elementos de memória internos do processador (contadores, registradores, etc.). No caso de dispositivos com hardware programável, os recursos reconfiguráveis que permitem o acesso e o controle de nodos do dispositivo podem ser utilizados para injeção de falhas.

#### 3.2.1 Software-Implemented Fault Injection (SWIFI)

A sigla SWIFI (do inglês, *Software Implemented Fault Injection*) está diretamente ligada à execução de partes específicas de software que modificam elementos internos (acessíveis pelo usuário) provocando o efeito de uma falha ocorrida no hardware. O método SWIFI está relacionado principalmente a sistemas processados, onde de alguma forma o software da aplicação normal é interrompido para a execução de um código de injeção de falhas que deve alterar um elemento, como um registrador, um dado da memória, ou até uma instrução da aplicação. SWIFI aparece também na análise do comportamento do DUT na

ocorrência de problemas de comunicação ou interação com outros sistemas, como, por exemplo, mensagens repetidas/faltando ou com informações erradas, falhas na leitura da memória, entre outros.

Esse tipo de injeção de falha pode ser dividido quanto ao momento da injeção, que pode ser em tempo de compilação ou durante a execução. Em tempo de compilação, modificações na imagem do software são feitas e quando são executadas, ativam a falha que simula o efeito de uma perturbação ocorrida no hardware. Esse tipo de falha não precisa de código adicional para ser executada e é geralmente utilizada para emular falhas permanentes. No caso de falhas injetadas durante a execução, mecanismos de disparo são utilizados para avisar o momento da injeção da falha. Temporizadores ou instruções específicas (de armadilha) no código são utilizadas para a interrupção da aplicação e execução da tarefa de injeção de falha.

Uma avaliação de sistemas baseado em COTS para aplicações espaciais foi desenvolvida em (27). Nesse trabalho o COTS era a placa CETIA com dois processadores PowerPC 750 rodando LynxOS. O trabalho apresenta a utilização da ferramenta XCEPTION para emular o efeito de SEUs no sistema. A ferramenta XCEPTION explora os recursos avançados de monitoramento de desempenho e de erros presentes em processadores atuais e utiliza os mecanismos de exceção do próprio processador para o disparo das falhas. Diferentes ferramentas, como: FERRARI, que utiliza temporizadores para disparar uma rotina de injeção de falhas; FIAT, que pode corromper mensagens, atrasar tarefas ou terminá-las abruptamente; FTAPE, que adiciona *drivers* de injeção de falhas no sistema operacional; entre outras são apresentadas em (25) e fazem parte das que implementam falhas em hardware no nível de software.

### **3.2.2 Injeção de Falhas Baseada em Simulação**

A injeção de falhas baseada em simulação utiliza um modelo do sistema a ser analisado. Os modelos de simulação utilizam descrições de hardware como VHDL e Verilog ou modelagens em mais alto nível de abstração como SystemC. No método de simulação, as falhas podem ser injetadas pela ferramenta de simulação ou pela alteração da descrição do modelo do hardware. Nesse último caso podem ser adicionados “sabotadores” no modelo, que são componentes específicos com o único propósito de injetar falhas, ou o emprego de componentes “mutantes” que contém a descrição do funcionamento livre e na presença de falha. O outro método utiliza recursos ou adapta as ferramentas de simulação

de forma a utilizar comandos para controlar sinais internos do modelo do sistema. Esse método é dependente da funcionalidade dos comandos do simulador, no entanto, não há necessidade da alteração do modelo de hardware.

Em (28), um modelo do processador LEON3 da Gaisler usado pela ESA (do inglês, *European Space Agency*) descrito em SystemC é utilizado para a simulação de falhas. Esse modelo descrito em SystemC tem seu nível de abstração elevado para o modelo TLM (do inglês, *Transaction Level Modeling*) que é baseado em transações. No modelo TLM são colocados mutantes que são utilizados para simular os efeitos de falhas em seções da memória (pilha, instruções, dados, etc.) do processador.

A ferramenta MEFISTO apresentada em (29) utiliza sabotadores, mutantes e comandos de simulação para fazer análise mais apurada do comportamento de falhas em modelos descritos em VHDL. Outras ferramentas como VERIFY que propôs uma extensão da descrição VHDL para adicionar recursos para injeção de falhas, HEARTLESS que desenvolveu sua própria ferramenta para a simulação de falhas permanentes e transientes, entre outros trabalhos que compõem o grupo de injeção de falhas baseado em simulação são apresentados em (25).

### 3.2.3 Injeção de Falhas Utilizando Recursos de Depuração

Processadores atuais possuem recursos específicos para o suporte a testes e também para depuração. Esses recursos, chamados de OCD, habilitam a injeção de falhas e a observação dos efeitos externamente ao processador de forma rápida e eficiente.

A ferramenta FIMBUL (do inglês, *Fault Injection and Monitoring Using Built in Logic*) desenvolvida em (30) utiliza o TAP (do inglês, *Test Access Port*) do OCD do processador Thor para a injeção de falhas. A TAP permite o acesso às cadeias de *scan* interna e da periferia do processador Thor. Dessa forma, as falhas transientes são injetadas em qualquer local em que as cadeias têm acesso. Nesse mesmo trabalho é apresentada uma comparação entre as ferramentas FIMBUL e MEFISTO onde é mostrado que a técnica por simulação (MEFISTO) tem uma cobertura de erros levemente superior. No entanto, a ferramenta FIMBUL que utiliza a técnica SCIFI (do inglês, *Scan Chain Implemented Fault Injection*), mostrou ser cem vezes mais rápida.

O trabalho apresentado em (31) propõe alterações na infraestrutura de depuração (OCD) com o objetivo de possibilitar a injeção de falhas para dar suporte à verificação de mecanismos de



tolerância a falhas. O OCD orientado à injeção de falha é chamado de OCD-FI (do inglês, *OCD-Fault Injection*) é baseado no padrão NEXUS. O OCD-FI proposto consiste em um hardware adicional que insere automaticamente falhas na ocorrência de condições de disparo, ou seja, sem a necessidade de comandos para o OCD. O endereço da instrução que disparará a injeção da falha é gerado aleatoriamente entre um dos valores presentes no espaço de endereçamento da memória de instruções. O mesmo acontece com o endereço da memória de dados que terá o valor do dado alterado quando a injeção da falha tiver sido disparada. Após as falhas terem sido injetadas os resultados são recuperados depois da conclusão de todos os experimentos.

### 3.2.4 Injeção de Falhas por Emulação

A prototipação de hardware baseada em FPGAs ficou popular na verificação de ASICs, e mais recentemente é explorada para injeção acelerada de falhas. O termo “emulação” nesse contexto está relacionado à prototipação do circuito a ser analisado em FPGAs. É importante ressaltar que muitas das técnicas que empregam o FPGA para a emulação de falhas, utilizam-no somente como meio de injeção de falhas, e depois o circuito analisado pode ser implementado em alguma outra tecnologia, como ASIC por exemplo.

A injeção de falhas requer habilidades de controle e observação elevadas. A controlabilidade é alcançada através do recurso de reconfiguração do FPGA que permite o acesso a memória de configuração do circuito. Outra forma de emulação de falhas consiste na adição de blocos de hardware, chamados de “instrumentos”, no protótipo para dar suporte à injeção.

Um dos primeiros trabalhos utilizando FPGA para a emulação de falhas é apresentado em (32). Esse trabalho implementa falhas permanentes do tipo *stuck-at* conectando sinais a valores constantes. Nesse caso, para cada falha injetada o circuito era sintetizado novamente e conseqüentemente o *bitstream* tinha que ser recarregado no FPGA para uma nova emulação. Logo, não era muito eficiente.

Entre os métodos mais novos que utilizam FPGAs para emulação de falhas dois tipos de abordagens podem ser percebidos. A primeira, chamada de “instrumentação”, insere lógica, de alguma maneira no circuito, com o objetivo de aumentar o controle e o poder de observação do DUT durante a sua execução. A segunda utiliza recursos de reconfiguração parcial existente para disponibilizar o controle e poder

de observação necessária para a emulação. Exemplos dessas duas abordagens de emulação de falhas são apresentados a seguir.

#### 3.2.4.1 Instrumentação

O termo instrumentação nesse contexto significa introduzir modificações na descrição HDL (do inglês, *Hardware Design Language*) do circuito com o objetivo de dar suporte à emulação de falhas.

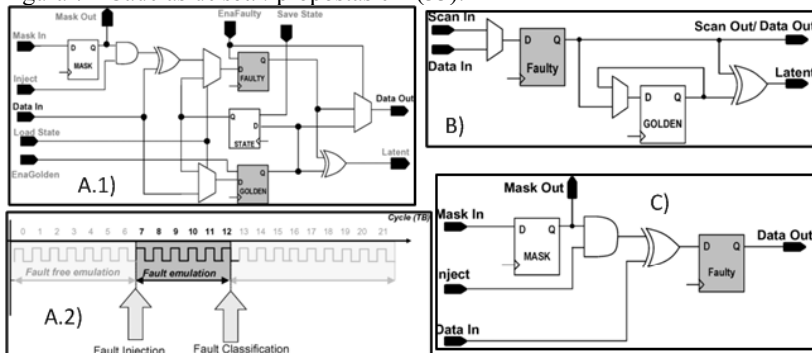
O trabalho desenvolvido em (33) apresenta um sistema de análise de SEUs totalmente autônomo, ou seja, sem a necessidade de interação com um hospedeiro para controlar a emulação. A instrumentação desenvolvida nesse trabalho se refere a cadeias de *scan* específicas para a injeção de falhas. No caso, a cadeia de *scan* é formada por todos os registradores do DUT. São apresentadas três tipos de células da cadeia de *scan* com características de otimização diferentes, como mostra a Figura 7. O sistema apresentado no trabalho carrega as falhas a serem injetadas através da cadeia de *scan*. O teste do DUT é dividido em ciclos de relógio, dessa forma, para cada falha existe um número do ciclo de relógio correspondendo ao momento em que a falha será injetada. As falhas são injetadas em todos os registradores e em todos os ciclos de relógio do total de ciclos do teste. A célula apresentada em A.1 permite armazenar o estado anterior do registrador antes da falha ser injetada. Dessa forma, o sistema de injeção pode recuperar o estado do DUT ao invés de reiniciar o DUT e esperar o número de ciclos de relógio até o momento de injeção. Quando a célula A.1 é utilizada, o DUT opera através dos registradores FAULTY e GOLDEN em ciclos de relógios alternados (multiplexado em tempo). Dessa forma, existe um DUT GOLDEN e um DUT FAULTY sem a necessidade de replicar todo o circuito. A existência desses dois DUTs permite que a emulação possa ser interrompida assim que o efeito da falha injetada desapareça e também possibilita classificar se o efeito da falha permanece latente ao final dos ciclos de relógio da emulação. A forma de onda A.2 da Figura 7 mostra o número de ciclos (5) de relógio utilizados para a emulação de uma falha que começa no ciclo sete e em 12 é possível notar que falha virou um defeito ou seu efeito desapareceu.

A cadeia de *scan* formada pelas células do tipo B da Figura 7 é utilizada para reduzir o impacto na ocupação do FPGA causada pelas células do tipo A.1. Nesse tipo de cadeia de *scan* (B) o sistema de emulação de falhas que fica no hospedeiro gera uma lista com todos os estados dos registradores do DUT para cada ciclo de relógio e armazena em uma memória externa da placa com o FPGA. O estado dos

registradores é recuperado dessa memória externa, o que corresponde ao momento da injeção da falha, ou seja, isso é feito para todos os ciclos de relógio do teste e para cada registrador onde a falha será injetada. No início da campanha um teste completo é executado sem a injeção de falha para armazenar o resultado final no registrador GOLDEN. Com isso, ao final dos testes com injeção de falha os registradores FAULTY e GOLDEN são comparados para poder classificar as falhas que não causaram defeito como latente ou silenciosas.

A célula apresentada em C na Figura 7 busca otimizar ainda mais a ocupação do FPGA e ainda reduzir a necessidade de uma memória externa. Para cada injeção de falha, a cadeia dos registradores MASK é carregada com o padrão que define onde a falha será injetada. Como o estado não é armazenado o teste é executado inteiramente para cada injeção de falha. Nesse caso, somente as saídas do DUT são analisadas, o que permite classificar o efeito das falhas como as que provocaram defeito ou não.

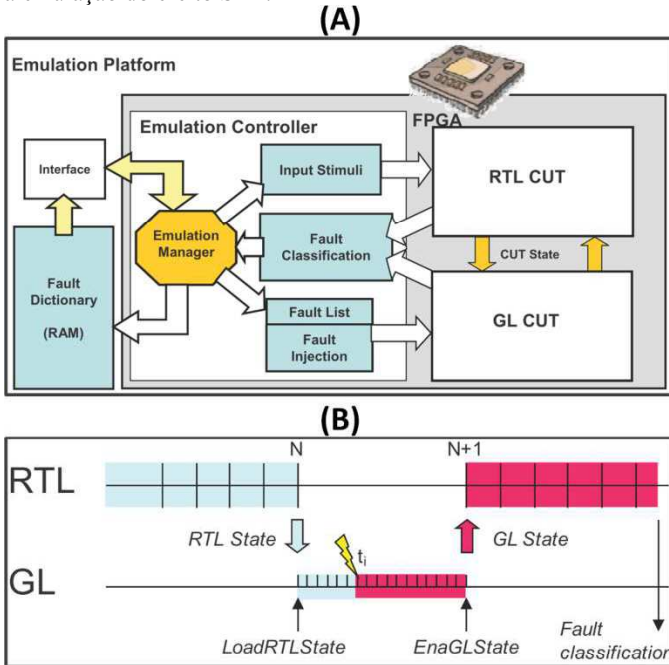
Figura 7 - Cadeias de *scan* propostas em (33).



A abordagem para gerar estimativas do efeito de SETs, chamada de AMUSE (do inglês, *Autonomous Multilevel Emulation System for Soft Error Evaluation*), é apresentada em (34). Esse trabalho é basicamente uma evolução do trabalho anterior através da adição do suporte à análise de SET. AMUSE integra a descrição RTL (do inglês, *Register Transfer Level*) e a descrição em nível de portas lógicas GL (do inglês, *Gate Level*) do DUT de forma que ocorre a troca do nível de abstração de RTL para o GT durante o ciclo de relógio da injeção da falha para permitir a emulação do SET. Após o ciclo de relógio da injeção da falha, o nível RTL é reassumido para propagação do efeito da falha de forma mais eficiente, onde nesse caso, o efeito dessa falha se tornou um SEU. A Figura 8 (A) apresenta a plataforma de emulação AMUSE contendo os dois modelos do DUT, um em RTL e outro em GL

sendo executados no mesmo FPGA. Na Figura 8 (B) é ilustrada a troca de nível de abstração para a injeção de falha em GL.

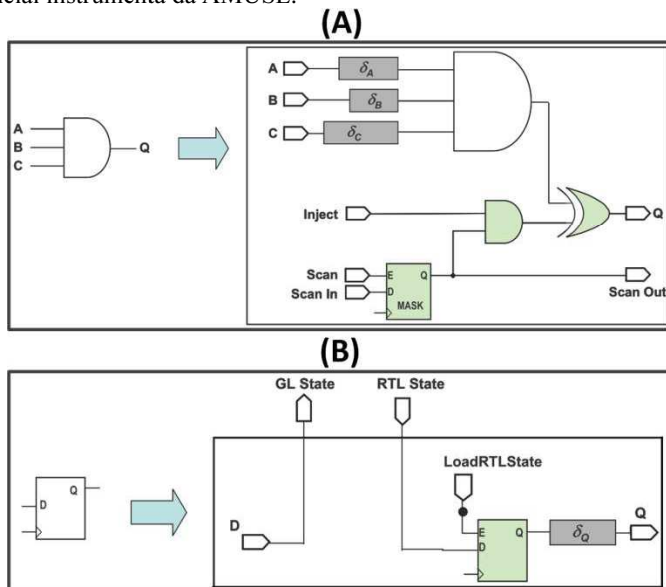
Figura 8 - (A) Plataforma de emulação AMUSE (34) - (B) Troca de RTL para emulação do efeito SET.



O atraso de propagação da variação causada pela falha é emulado no nível GL. Os atrasos disponibilizados pela ferramenta de síntese são arredondados para um múltiplo inteiro do valor de quantização do tempo (TQ) escolhido. Cada atraso é implementado através de um registrador de deslocamento onde a sua profundidade representa o múltiplo de TQ. A Figura 9 mostra os equivalentes de um elemento combinacional (A) e um sequencial (B) descritos em GL da plataforma AMUSE. Os estados dos registradores em RTL são carregados para os registradores GL no ciclo de relógio da injeção de falha. A cadeia de *scan* presente em GL é carregada com o padrão que define qual dos elementos lógicos terá a sua saída invertida durante alguns ciclos em GL. Ao final dos ciclos o valor da entrada dos registrados em GL é gravado nos registradores em RTL. Em RTL as células de instrumentação são semelhantes as do trabalho anterior contendo uma versão FAULTY que recebe o valor vindo de GL e uma GOLDEN. Dessa forma, o efeito da falha é propagado através do modelo em RTL que é mais eficiente. Como no trabalho anterior o

efeito de cada falha pode ser classificado como latente, silencioso ou que causou defeito.

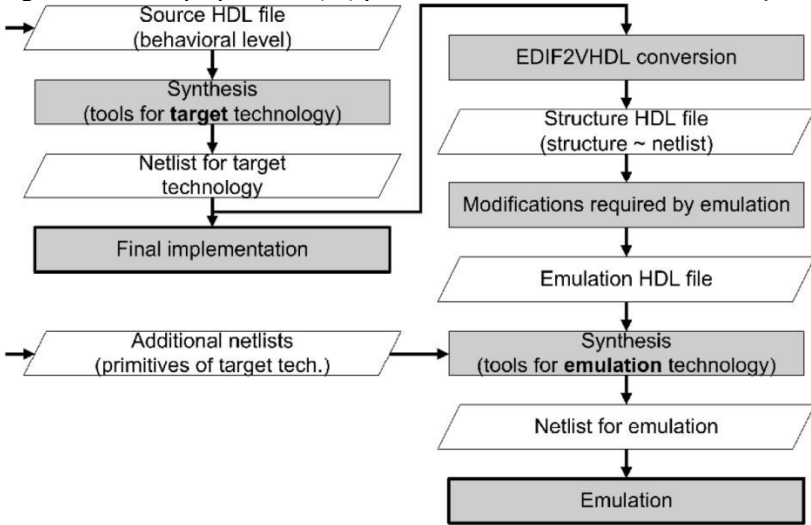
Figura 9 - Os equivalentes em GL de um elemento (A) combinacional e um (B) sequencial instrumentada da AMUSE.



Os FPGAs são largamente utilizados para a prototipagem e verificação de circuitos projetados para ASICs. No entanto, os componentes utilizados durante a síntese desse circuito dependem do dispositivo alvo, onde nesse caso pode ser ASIC para aplicação final ou o FPGA para a fase de prototipagem e verificação. No contexto de aplicações críticas, a lista com as falhas a serem testadas deve ser gerada com base nos componentes do dispositivo da aplicação final visando resultados mais realísticos. Assumindo que a mesma descrição HDL do circuito seja utilizada para síntese tanto para FPGA quanto para o ASIC, a lista de falhas não se aplicaria para os dois devido à diferença de componentes entre as duas tecnologias. Visto isso, o trabalho apresentado em (35) converte o resultado da síntese visando ASIC para uma HDL contendo somente a estrutura dos componentes, ou seja, sem a parte comportamental do circuito. Dessa forma, alguns componentes desse HDL podem ser substituídos por elementos específicos do FPGA para dar suporte à emulação de falhas através de instrumentação sem que a estrutura do circuito seja alterada. O fluxo proposto nesse trabalho é apresentado na Figura 10. Esse trabalho

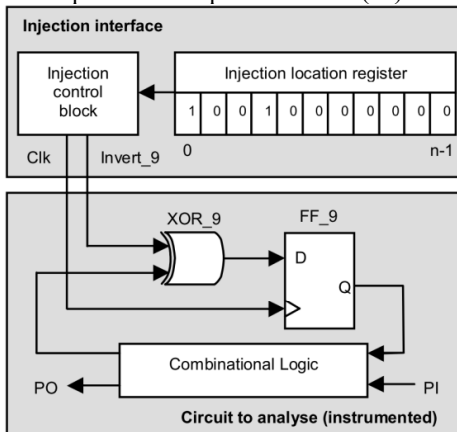
apresenta também a utilização de modelos de atraso semelhantes aos apresentados na plataforma AMUSE.

Figura 10 - Fluxo proposto em (35) para diminuir a intrusão da instrumentação.



A técnica apresentada em (36) faz análise de SEUs através de instrumentação mas sem utilizar cadeias de *scan*. A técnica utiliza uma porta lógica XOR antes de cada registrador com o objetivo de manipular a inversão do sinal de entrada. Todas essas XORs são controladas por uma ou mais centrais de que definem onde a falha será injetada, dessa forma, o local da falha pode ser carregado paralelamente.

Figura 11 - Controlador de localização e o exemplo de um registrador instrumentado utilizado pela técnica apresentada em (36).

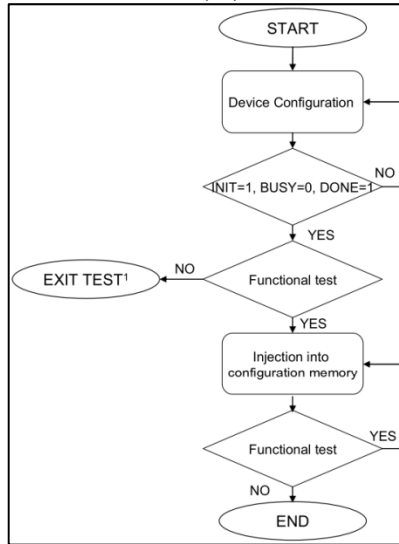


A Figura 11 mostra o exemplo de um registrador instrumentado com uma porta XOR na entrada. O sinal que controla a inversão executada pela porta XOR é gerado pelo controlador de injeção de falhas. O controlador de injeção de falhas decodifica o valor do registrador de localização para determinar onde a falha será injetada. A plataforma de emulação utiliza um FPGA Virtex2PRO contendo um PowerPC que escreve no registrador de localização onde será injetada a próxima falha.

#### 3.2.4.2 Reconfiguração

Ao contrário da instrumentação, as técnicas que utilizam reconfiguração buscam não alterar a descrição HDL do circuito buscando ser menos intrusiva no projeto do DUT. O sistema FLIPPER apresentado em (37) visa a análise de FPGAs da Xilinx quanto a efeitos de SEUs e MCUs. FLIPPER permite a análise da sensibilidade do DUT através da coleta da distribuição de probabilidade do número de falhas injetadas necessárias para causar uma falha funcional. FLIPPER também é utilizado para testar quais os bits da memória de configuração contendo a implementação do DUT são sensíveis ou não interferem no circuito implementado. FLIPPER usa resultados de simulação para geração de vetores de teste contendo valores de entrada para o estímulo do DUT e de saída para comparação. As falhas são injetadas na memória de configuração do FPGA através de um software executado em um computador hospedeiro. A memória de configuração do FPGA é reconfigurada parcialmente (somente onde a falha é injetada) durante a execução do DUT para a emulação de SEUs ou acúmulo deles (MCUs). A Figura 12 mostra o fluxo do sistema FLIPPER. Os resultados obtidos com o sistema FLIPPER são comparados com os gerados por um acelerador de partículas em (38). Os resultados dessa comparação validam FLIPPER como uma ferramenta para a estimação do efeito de falhas nos bits de configuração do FPGA, principalmente para verificação de técnicas de tolerância a falhas aplicadas na memória de configuração do FPGA.

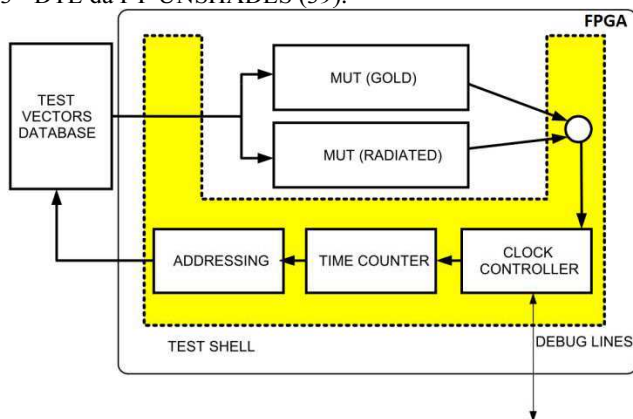
Figura 12 - Fluxo do sistema FLIPPER (37).



A plataforma FT-UNSHADES (do inglês, *Fault Tolerant–University of Sevilla Hardware Debugging System*) é apresentada em (39). Diferentemente do trabalho anterior, o sistema de controle de injeção de falhas é implementado junto com o DUT no FPGA. Esse sistema de controle é chamado de DTE (do inglês, *Design for Test Emulation*) e é apresentado na Figura 13. O DTE funciona exclusivamente em FPGAs da Xilinx. Além do FPGA ilustrado na figura, outro FPGA menor é utilizado para fazer a interface com o computador hospedeiro. As falhas testadas são aplicadas para analisar os efeitos do tipo SEU. Para cada falha no DUT, chamado de MUT (do inglês, *Module Under Test*) no trabalho, o sinal de relógio é interrompido para a injeção através da reconfiguração parcial. As saídas do MUT (RADIATED) onde foi injetada a falha são comparadas com as saídas do MUT (GOLD) livre de falhas. Se ao final da emulação da falha nenhuma diferença entre as saídas é encontrada então os recursos *capture* que amostra os valores de todos registradores do FPGA e *readback* que permite a releitura desses valores são utilizados. Esses recursos permitem comparar os estados dos registradores do MUT GOLD e do MUT RADIATED para analisar se o efeito da falha está latente ou desapareceu. O software executado no computador hospedeiro é quem controla a escolha do registrador onde será feita a injeção da falha. Em (40) e (41) são apresentadas melhorias na FT-UNSHADES de forma a possibilitar a análise de SETs e MBUs.



Figura 13 - DTE da FT-UNSHADES (39).



O trabalho em (42) apresenta uma plataforma de injeção de falhas baseada em um processador embarcado. A técnica desse trabalho é aplicada utilizando FPGAs de vendedores diferentes, como o AT94K da Atmel e alguns FPGAs das famílias Virtex4 e 5 da Xilinx. O dispositivo da Atmel AT94K é um SoC com um micro controlador AVR (do inglês, *Advanced Virtual RISC*) de 8-bits e um FPGA composto por uma matriz  $N \times N$  ( $N=94$ ) de PLBs (do inglês, *Programmable Logic Blocks*) que contém dois geradores de função lógica de três entradas cada, um registrador e um multiplexador. A Figura 14 mostra a arquitetura do SoC AT94K da Atmel. O código do programa de injeção de falhas é gravado na memória de instrução do AVR no SoC AT94K. O processador AVR tem acesso de escrita na memória de configuração do FPGA o que permite a reconfiguração parcial dinâmica. As falhas são injetadas com o objetivo de analisar a capacidade de detecção de BIST (do inglês, *Built-In Self-Test*) configurado no FPGA. Na segunda abordagem foi desenvolvido um bloco embarcado para a emulação de falhas utilizando o recurso ICAP (do inglês, *Internal Configuration Access Port*) dos FPGAs da Xilinx. A lista de falhas é carregada na memória do bloco embarcado que faz releitura de um quadro da memória de configuração correspondente com a localização da falha a ser injetada. Esse quadro sofre a alteração de acordo com a falha da lista e é reescrito na memória de configuração. Após a reconfiguração parcial para a injeção da falha o BIST é executado e o seu resultado é armazenado. Isso é feito para cada falha da lista como mostra o fluxo (A) executado pelo bloco embarcado (B) na Figura 15.

Figura 14 - Arquitetura do SoC AT94K da Atmel (42).

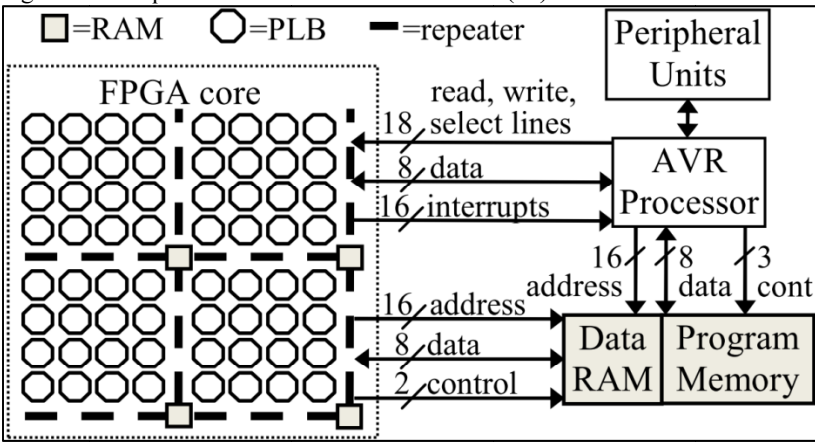
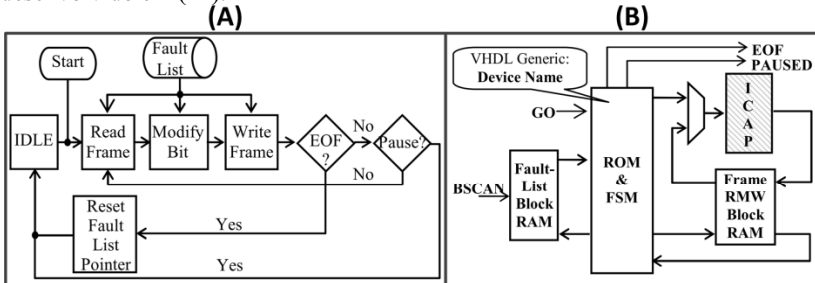


Figura 15 - O fluxo (A) e a arquitetura (B) do bloco de injeção de falha desenvolvido em (42).



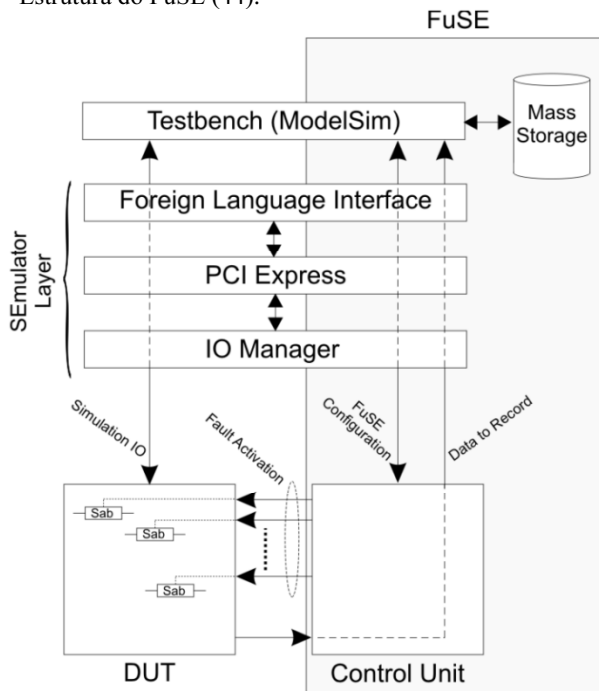
O trabalho desenvolvido em (43) também utiliza o recurso ICAP dos FPGAs da Xilinx juntamente com o processador sintetizável MicroBlaze™ para a injeção de falha. Nesse trabalho o DUT é executado uma vez livre de falhas e seus resultados (GOLDEN) são armazenados para posterior comparação. Para cada injeção de falha o sinal de relógio do DUT é interrompido e a reconfiguração parcial é realizada através do ICAP. Os resultados obtidos de cada falha são armazenados e após o término da campanha eles são enviados para um computador hospedeiro para a geração de resultados.

### 3.2.5 Técnicas Híbridas de Injeção de Falhas

A ferramenta FuSE (do inglês, *Fault Injection Using SEmulation*) é apresentada em (44). FuSE integra simulação e emulação em FPGA em um único ambiente. O funcionamento básico da ferramenta consiste

em um simulador onde é executado o teste e o FPGA que implementa o DUT. O simulador (SEmulator®) se comunica com o DUT através da interface PCI Express para o envio de estímulos e captura de valores de sinais internos. O FuSE controla sabotadores inseridos no DUT para a injeção de falhas. A Figura 16 mostra a estrutura do FuSE.

Figura 16 - Estrutura do FuSE (44).



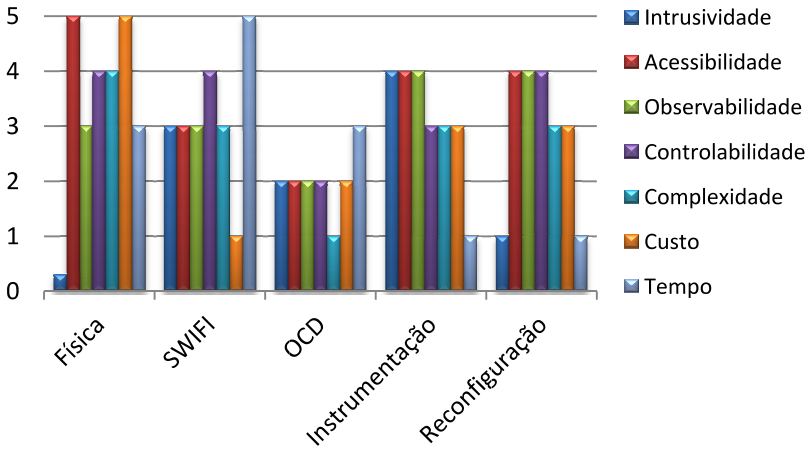
Além do FuSE, o sistema FLIPPER pode ser considerado híbrido uma vez que utiliza simulação para criação de estímulos e geração de valores para comparação.

A Figura 17 mostra de forma geral um resumo das principais características das diferentes técnicas de injeção de falhas. As características avaliadas são:

- Intrusividade – Diferença em relação ao projeto original;
- Acessibilidade – Alcance da injeção de falhas;
- Observabilidade – Principalmente capacidade de classificação de falhas;
- Controlabilidade – Poder de controle da injeção (local, momento, entre outros);

- Complexidade – Dificuldade para execução do experimento;
- Custo – Custo dos equipamentos e ferramentas próprias para o teste;
- Tempo – Tempo do experimento.

Figura 17 - Resumo comparativo das técnicas de injeção.



## 4 INVESTIGAÇÃO DA ARQUITETURA ALVO

A flexibilidade da emulação de falhas alcançada através de hardware reconfigurável apresentada na seção anterior foi fator fundamental para a escolha dessa técnica de injeção para o desenvolvimento desse trabalho. A injeção de falhas requer um alto nível de controlabilidade dos componentes do circuito a fim de modificar o seu estado lógico. Isso pode ser feito por meio dos mecanismos de reconfiguração dos FPGAs para alterar o circuito ou o conteúdo de elementos de memória. Para isso, é necessário um estudo mais detalhado da arquitetura do FPGA e das suas funcionalidades. Esse estudo foi feito sobre o FPGA da Xilinx modelo XC5VLX110T. O kit de avaliação utilizado para o desenvolvimento desse trabalho foi o XUPV5-LX110T da Xilinx que é baseado no FPGA XC5VLX110T da família Virtex5.

No estudo do FPGA XC5VLX110T serão apresentados os seguintes itens: a arquitetura do FPGA juntamente com os principais componentes relevantes para a emulação de falhas; o sistema de coordenadas que define a localização de cada componente; a memória de configuração do FPGA e a respectiva forma de endereçamento; e finalmente, algumas das formas de investigação que permitiram o entendimento da arquitetura.

### 4.1 ARQUITETURA

Na introdução desse trabalho foram apresentados os blocos básicos de um FPGA. Nessa seção serão apresentados os componentes específicos encontrados nos FPGAs da família Virtex-5 da Xilinx. A arquitetura do FPGA XC5VLX110T é apresentada na Figura 18. Na figura estão também destacados alguns blocos desse FPGA. Os principais blocos são:

- CLB (do inglês, *Configurable Logic Block*) – é o elemento lógico básico e é constituído de dois *slices*. O *slice* permite a implementação de lógica combinacional e/ou sequencial e pode ser configurado para operar com um registrador de deslocamento de 32 bits ou uma RAM de 64 bits, chamada de *Distributed RAM*;
- BRAM (do inglês, *Block RAM*) – módulo de armazenamento flexível que pode ser implementado como uma memória 36Kbits ou duas de 18Kbits com

duas interfaces independentes de acesso ou como uma fila;

- DSP48E – é a extensão do módulo DSP (do inglês, *Digital Signal Processing*) da Virtex4. Os *slices* DSP da coluna do FPGA estão arranjados em cascata e são constituídos com um multiplicador de complemento de dois 25x18 e um somador/subtrator/acumulador de 48 bits que dão suporte a algoritmos de DSP;
- IOB – bloco configurável que suporta os mais populares padrões de entrada/saída (LVCMOS, LVTTTL, PCI, LVDS, entre outros). Possui recursos para a otimização de interface síncrona, como controle de escorregamento de sinal, des/serializador, divisor de *clock*, entre outros;
- PCIE – bloco integrado que permite projetos com total funcionalidade de *PCI Express Endpoint* com a mínima utilização de recursos;
- TEMAC (do inglês, *Tri-mode Ethernet Medica Access Controller*) – elemento integrado que permite projetos utilizando conexão Ethernet com a mínima utilização de recursos;
- CMT (do inglês, *Clock Management Tile*) – Possui dois DCMs (do inglês, *Digital Clock Managers*) e um PLL (do inglês, *Phase-Locked Loop*) que buscam prover para o FPGA o sinal de relógio mais otimizado possível;
- GTP\_DUAL – possui dois transceptores seriais gigabit *full-duplex* de 100Mb/s até 3.5Gb/s que servem para dar suporte protocolos, SATA, PCIe, Gigabit Ethernet, entre outros.

Além dos principais componentes contidos no FPGA XC5VLX110T estão destacados mais dois elementos na Figura 18. Esses elementos são:

- IOB\_BANK – cada conjunto de IOBs possui apenas uma tensão de referência configurável que dá o suporte para diferentes padrões de entrada/saída;
- PRIMITIVES – são elementos básicos que permite o acesso do usuário recursos específicos do FPGA. Esses elementos são:
  - ICAP – permite acesso a memória de configuração do FPGA;

- USER\_ACCESS – permite acesso a um registrador de 32 bits escrito durante a configuração do FPGA. Recurso utilizado principalmente para identificação de projeto;
- STARTUP – permite o controle de determinados sinais globais após a configuração do FPGA;
- BSCAN – permite acesso ao controlador do JTAG TAP;
- FRAME\_ECC – permite acesso ao circuito de detecção e correção de erros de quadros de configuração;
- CAPTURE – permite a amostragem dos valores de latches/registradores de todo FPGA;
- KEY\_CLEAR – permite apagar a chave de criptografia da configuração interna do FPGA;
- DCI\_RESET – permite reinicializar a máquina de estados do DCI (do inglês, *Digitally Controlled Impedance*);
- SYSMON – permite acesso a conversores analógico/digital do FPGA.

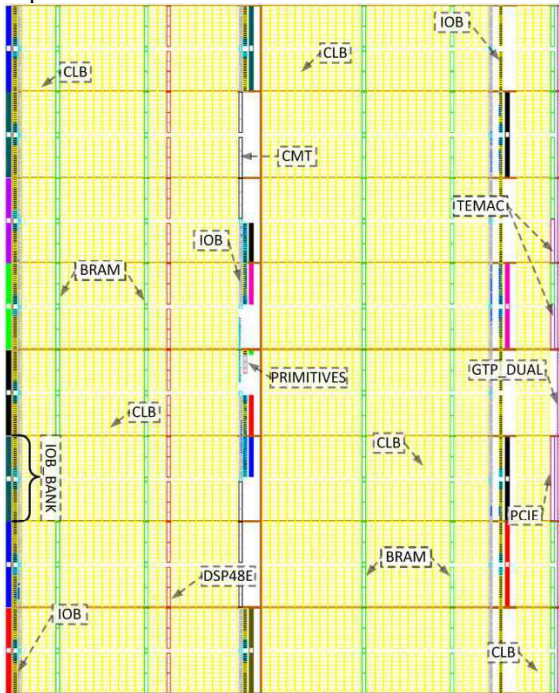
As matrizes de roteamento e os *buffers*, que não estão na Figura 18, completam a lista dos principais componentes presentes no FPGA XC5VLX110T. A Tabela 1 apresenta as características do dispositivo.

Tabela 1 - Especificação das características do FPGA XC5VLX110T.

CLBs	Matriz (linha X coluna)	160 X 54
	Número de <i>Slices</i>	17.280
	Máx. <i>Distributed RAM</i>	1.120 Kb
BRAMs	18 Kb / 36 Kb	296 / 148
	Capacidade máxima de armazenamento	5.328 Kb
<i>Slices</i> DSP48E		64
CMTs (2 x DCMs / 1 x PLL)		6
<i>Endpoint Blocks for PCI Express</i> (PCIE)		1
<i>Ethernet MACs</i> (TEMAC)		4
<i>Gigabit Transceivers</i> (GTP)		16
Conjuntos de IOBs (IO BANKs)		20
IOBs que podem ser usados pelo usuário		680

Fonte: Xilinx Inc. (2009) (45).

Figura 18 - Arquitetura do FPGA XC5VLX110T.



A Figura 19 mostra uma matriz de roteamento do FPGA XC5VLX110T. A matriz é responsável por interconectar as linhas de roteamento com as portas referentes da interface do CLB. A interface do CLB por sua vez faz a interligação das portas do CLB com os respectivos sinais de entrada/saída de cada um dos *slices*. Na Figura 19 destaca-se ainda a linha do sinal de *carry* de cada *slice* que sem passar por uma matriz de roteamento é conectado diretamente no *slice* do CLB adjacente visando otimizar o atraso de circuitos aritméticos.

A Figura 20 apresenta um CLB com os dois tipos de *slices* existentes. Todos os *slices* possuem quatro geradores de funções lógicas (ou tabelas verdades), quatro elementos de armazenamento, multiplexadores e lógica de *carry*. Esses elementos são utilizados para implementar circuitos combinacionais/sequenciais, aritméticos ou então operar como uma ROM. Somado a isso, alguns *slices* suportam duas funções adicionais: armazenamento de dados como uma RAM distribuída e registrador de deslocamento de até 32 bits. *Slices* com essa funcionalidades adicionais são chamados de SLICEM enquanto os outros são os SLICEL.



Figura 19 - Matriz de roteamento.

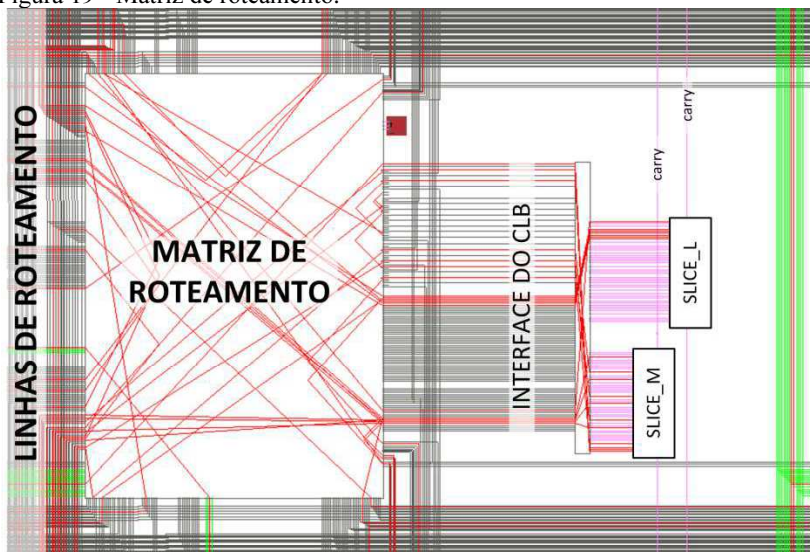
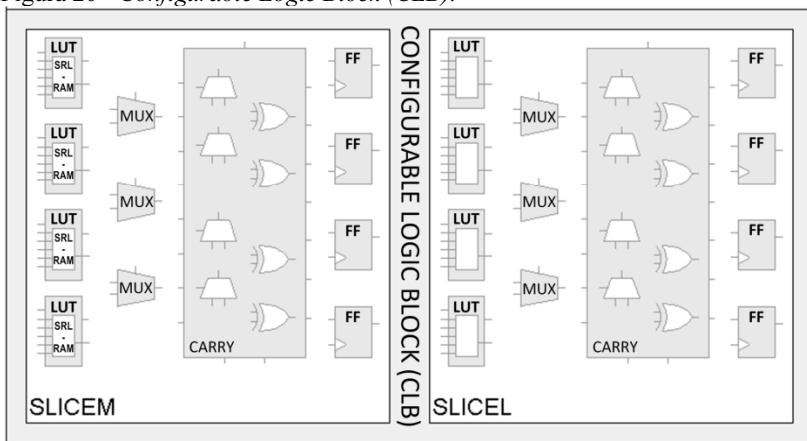


Figura 20 - Configurable Logic Block (CLB).



Os geradores de função lógica dos FPGAs Virtex5, chamados de LUTs (do inglês, *Look-up Table*), possuem seis entradas independentes (A1 até A6) e duas saídas independentes (O5 e O6) em cada uma das quatro LUTs (A, B, C e D) do *slice* como mostra a Figura 21. A figura mostra o diagrama detalhado de um SLICEL. O gerador de função pode implementar qualquer função booleana com seis entradas ou duas funções de cinco entradas desde que as duas compartilhem das mesmas entradas. As portas O5 e O6 das LUTs são conectadas às saídas e a

outros recursos dos *slice* de maneira diferente e são utilizadas conforme a necessidade. Somado a isso, o *slice* possui três multiplexadores (F7AMUX, F7BMUX e F8MUX) ligados nas saídas das LUTs que combinados permitem a implementação de funções booleanas de sete e oito e entradas.

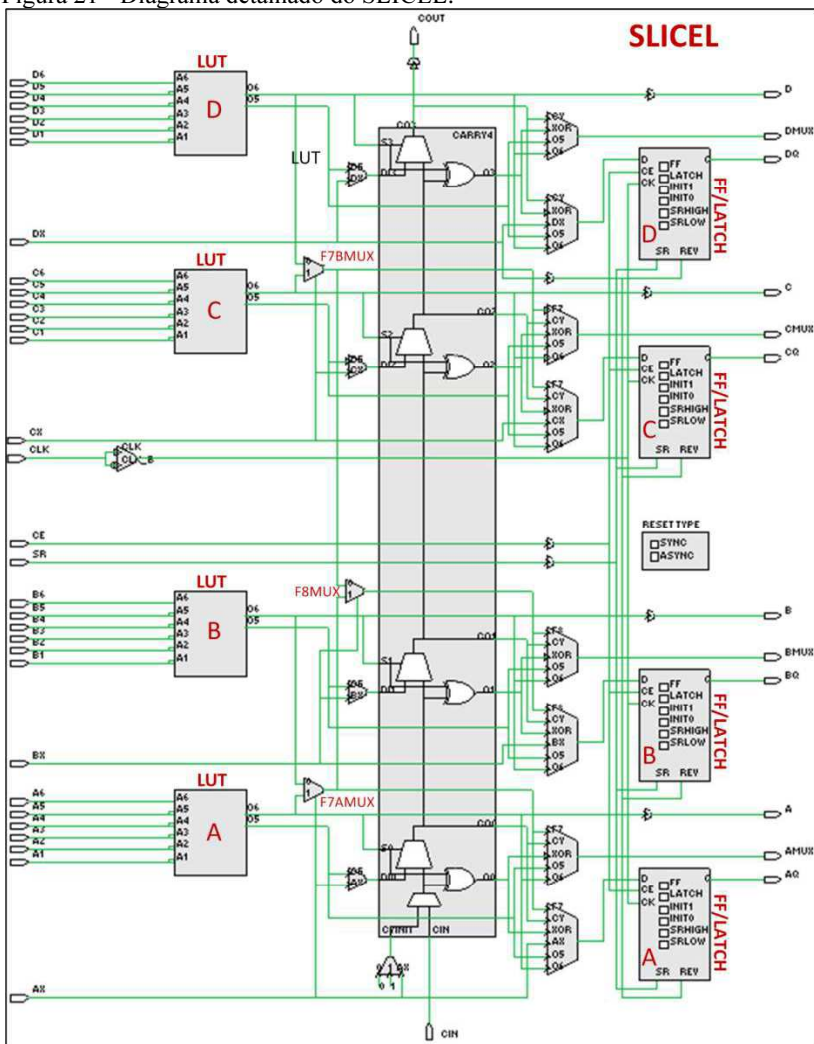
Os slices são compostos de quatro elementos de armazenamento que podem ser configurados como registradores do tipo D sensíveis a borda ou *latches* sensíveis a nível. As entradas dos elementos de armazenamento podem ser derivadas de LUTs ou direto das portas do *slice*. Os sinais de controle, como *clock* (CK), *clock enable* (CE), *set/reset* (SR) e *reverse* (REV) são comuns aos quatro elementos do mesmo *slice*. Os atributos SRHIGH e SRLow definem para qual o nível é forçado o sinal de saída do elemento de armazenamento quando o sinal SR é acionado. Os valores iniciais de cada elemento são determinados pelos atributos INIT0 e INIT1 e podem ser definidos individualmente, ao contrário, da escolha do atributo do SR de síncrono ou assíncrono que é comum aos quatro elementos. A escolha desses atributos permite as seguintes configurações de *latch* ou registrador:

- nem *set* ou *reset*;
- *set*, *reset* ou *set/reset* síncronos;
- *set* (*preset*), *reset* (*clear*) ou *set/reset* (*preset/clear*) assíncronos.

Além disso, cada CLB possui circuito aritmético dedicado para a soma e subtração de forma mais rápida. Esse circuito é constituído de duas cadeias de *carry* separadas, uma em cada *slice*. Essas cadeias podem ser arranjadas em cascata com o circuito de *carry* do CLB adjacente superior para formar um somador/subtrator maior.

Tanto o SLICEL quanto o SLICEM permitem a implementação de ROMs com capacidade de 64, 128 e 256 bits ocupando 1, 2 e 4 LUTs respectivamente. Contudo, diferentemente do SLICEL apresentado na Figura 21, o SLICEM possibilita o uso das LUTs como memórias distribuídas ou registradores de deslocamento de até 32 bits, chamados de SRL (do inglês, *Shift Register LUT*). São diversas as configurações de memória distribuída e SRL possíveis com o SLICEM.

Figura 21 - Diagrama detalhado do SLICEL.



#### 4.2 CONFIGURAÇÃO DO FPGA

Os FPGAs são configurados através do carregamento dos dados específicos da aplicação desenvolvida para a memória interna do FPGA. Esses dados que configuram o FPGA são chamados de *bitstream*. Devido à memória volátil dos FPGAs da Xilinx o carregamento do

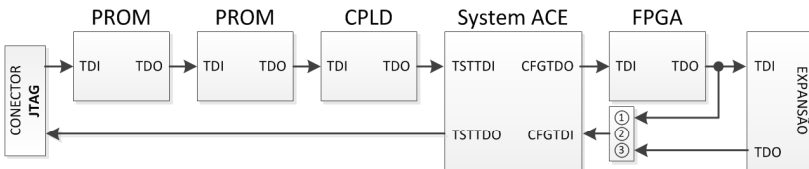
*bitstream* tem que ser realizado toda vez que o dispositivo é ligado. Nos FPGAs Virtex5 esse carregamento é realizado através de pinos dedicados que permitem diferentes modos de configuração:

- *Master* ou *Slave* serial;
- *Master* (x8 e x16) ou *Slave* (x8, x16 e x32) paralelo;
- *Master* SPI/BPI (do inglês, *Serial/Byte Peripheral Interface*) Flash (BPI, x8 e x16);
- *JTAG/Boundary-Scan*.

O kit XUPV5-LX110T é baseado no kit ML505 de Xilinx com a diferença do FPGA que são XC5VLX110T e XC5VLX50T respectivamente. Ambos os kits possuem componentes que possibilitam a exploração dos modos de configuração. Esses componentes são:

- Duas XCF32P *Platform Flash* PROMs da Xilinx que permitem a configuração do FPGA de modo *Master* ou *Slave* e serial ou paralelo;
- uma memória JS28F256P30T95 *NOR Linear Flash* que permite o modo *Master* BPI;
- uma memória M25P32 *SPI Flash Memory* que permite o modo *Master* SPI;
- um controlador System ACE™ que com um cartão de memória CompactFlash contendo até oito *bitstreams* permite a configuração do FPGA através da interface JTAG;
- um XC95144XL CPLD da Xilinx que pode ser usado para controlar uma das memórias para configurar o FPGA; e,
- a própria interface JTAG.

Além do controlador System ACE™ e do FPGA, as duas PROMs e o CPLD completam a cadeia do JTAG. A Figura 22 apresenta a cadeia de componentes conectados na interface JTAG do kit XUPV5-LX110T. Figura 22 - Cadeia de conexão do JTAG.

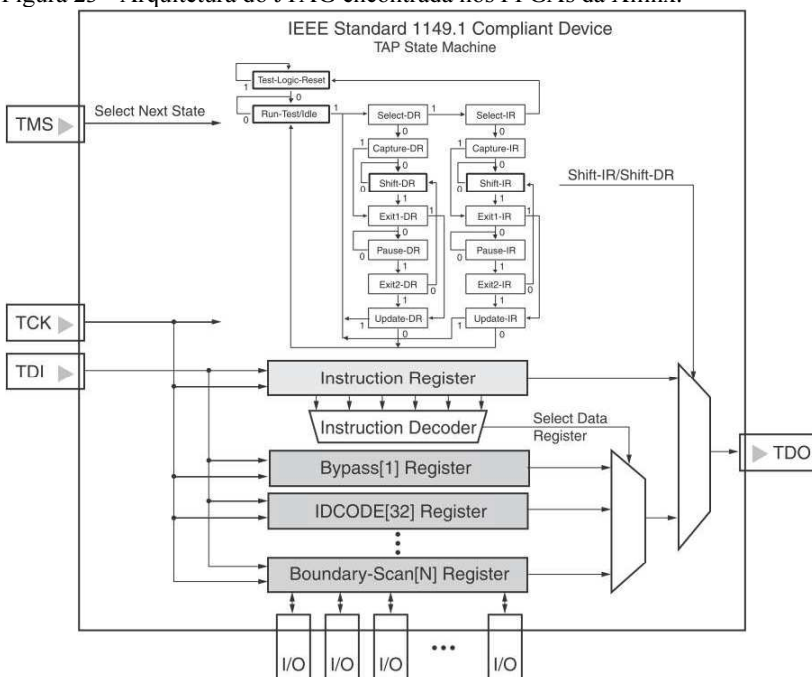


Através do JTAG é possível configurar as duas PROMs, o CPLD e o FPGA. Além disso, as memórias SPI/BPI *Flash* também podem ser configuradas por meio do JTAG, porém, a BPI *Flash* utiliza o FPGA

como ponte para o acesso a memória e a SPI *Flash* pertence à cadeia de outra porta com conector JTAG.

Os FPGAs da Xilinx são compatíveis com padronização IEEE 1149.1 (JTAG). A padronização define os elementos da arquitetura do JTAG que provê suporte ao teste do dispositivo. Os elementos básicos são o TAP, o controlador do TAP, o registrador e o decodificador de instrução, o registrador de *Boundary-Scan* e um registrador (BYPASS), sendo que este liga a entrada direto na saída. Além desses elementos básicos existem os registradores de identificação (IDCODE) e de configuração que são específicos de dispositivos da Xilinx. A Figura 23 resume a arquitetura de JTAG encontrada nos FPGAs da Xilinx com as quatro portas da máquina de estados do controlador TAP (TMS, TCK, TDI e TDO).

Figura 23 - Arquitetura do JTAG encontrada nos FPGAs da Xilinx.



Fonte: Xilinx Inc. (2011) (46).

A arquitetura do JTAG presente nos FPGAs da Xilinx implementa os comandos básicos do padrão da IEEE e permite as instruções específicas dos dispositivos. Essas instruções possibilitam o acesso a registradores internos definidos pelo usuário e principalmente concedem suporte ao carregamento do *bitstream* na memória de

configuração do FPGA e a sua releitura. O processo de releitura do *bitstream* carregado no FPGA, chamado de *readback*, é útil para a verificação de erros na memória de configuração, mas é raramente utilizado para a depuração de projetos devido ao poder de análise concedido pela ferramenta ChipScope™ ILA (do inglês, *Integrated Logic Analyzer*) da Xilinx.

### 4.3 INVESTIGAÇÃO DA MEMÓRIA DE CONFIGURAÇÃO

A definição da funcionalidade do *bitstream* que é configurar o comportamento do FPGA está clara nos manuais dos dispositivos dos fabricantes de circuitos reconfiguráveis. Contudo, o detalhamento completo do que representa cada bit do arquivo de configuração é mantido confidencial pelas empresas por vários motivos que são discutidos em (20). Dessa forma, foram utilizados diversos recursos com o objetivo de entender o *bitstream* de forma a possibilitar a localização exata da configuração de determinados componentes. Componentes estes que serão escolhidos para se injetar as falhas. Essa fase do projeto da dissertação foi a que despendeu mais tempo devido à falta de documentação proposital por parte da empresa fabricante do FPGA utilizado, no caso a Xilinx.

A investigação da memória de configuração do FPGA XC5VLX110T foi feita de três maneiras distintas. As três maneiras são apresentadas nas próximas subseções dessa seção e o conhecimento adquirido em cada uma serviu de forma complementar para o cálculo da localização dos componentes.

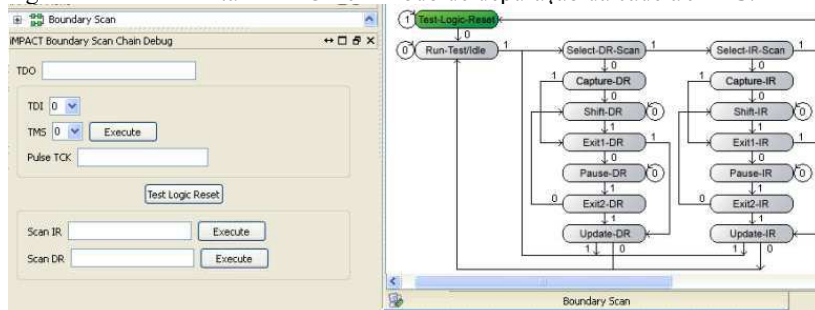
#### 4.3.1 Configuration Readback

Antes do carregamento do *bitstream* no FPGA é preciso configurar todos os componentes presentes na cadeia JTAG. Isso é feito por meio do registrador de instruções do JTAG, chamado de IR (do inglês, *Instruction Register*). Os outros registradores da arquitetura JTAG são chamados de DR (do inglês, *Data Register*). A máquina de estados do controlador JTAG implementa dois fluxos básicos que são o deslocamento de bits através do IR ou do DR. O tamanho dos registradores IR e DR, entre outras características do circuito JTAG de cada componente são encontradas na biblioteca BSDL (do inglês, *Boundary-Scan Description Language*). O tamanho do IR dos FPGAs Virtex5 é de 10bits. Os componentes PROM, CPLD e SystemACE da

XUPV5 têm IRs de tamanho 16, 8 e 8bits respectivamente o que corresponde a um total de 58 bits em toda a cadeia.

A porta TMS do TAP determina o próximo estado do controlador JTAG após a próxima borda de subida do TCK. Os sinais TMS e TCK não estão ligados em cadeia da mesma forma que TDI e TDO como mostrado na Figura 22, ou seja, o estado do controlador JTAG é igual para todos os componentes da cadeia. Para a configuração do IR é preciso colocar o controlador JTAG no estado SHIFT-IR e permanecer com o TMS igual a zero durante 58 pulsos de TCK para o deslocamento dos bits pela porta TDI. Após isso, é a vez do deslocamento de bits através do DR selecionado pela decodificação do valor carregado em IR. Como o objetivo é a configuração do FPGA então o valor do IR escolhido é o referente ao DR de configuração enquanto que o valor do IR dos outros componentes da cadeia é definido para ser o DR de BYPASS; Logo, os 32 bits do DR de configuração do FPGA mais quatro bits de cada DR de BYPASS da cadeia são deslocados pela porta TDI durante os 36 pulsos de TCK em que o controlador JTAG permanece no SHIFT-DR. No caso da leitura dos registradores o processo é o mesmo, enquanto os bits são deslocados por TDI na porta TDO surgem os bits que estavam nos registradores. A ferramenta iMPACT possui uma interface gráfica que permite a entrada manual de comandos para depuração da cadeia JTAG como mostra a Figura 24. A interface gráfica é muito útil para o entendimento do funcionamento do JTAG.

Figura 24 - Ferramenta iMPACT em modo de depuração da cadeia JTAG.



A ferramenta iMPACT permite a geração de arquivos contendo todas as operações executadas na interface gráfica, como verificação e o carregamento da memória de configuração do FPGA. Esses arquivos podem ser salvos no formato SVF (do inglês, *Serial Vector Format*) que descrevem comandos JTAG de forma compacta e independente de dispositivo. Os comandos básicos do formato SVF são SIR e SDR que

são utilizados para o deslocamento de bits para os registradores IR e DR respectivamente, RUNTEST que serve para aguardar TCK pulsos no estado RUN-TEST/IDLE e STATE permite a mudança do estado do controlador para um estado especificado.

Os arquivos SVF foram utilizados para entender o uso das instruções de controle do JTAG para o carregamento e a releitura da memória de configuração do FPGA. Contudo, o JTAG apenas permite o acesso ao barramento do circuito de configuração do FPGA através do registrador de configuração do JTAG. O acesso a esse barramento é feito através das instruções CFG\_IN e CFG\_OUT do JTAG. As instruções do JTAG JSTART, JSHUTDOWN e JPROGRAM permitem o controle do fluxo do circuito de configuração do FPGA.

O circuito de configuração do FPGA possui um banco de registradores que não são os mesmos registradores do JTAG. Esse banco de registradores é conectado a um barramento que é acessível através do registrador de configuração do JTAG de 32 bits. As instruções CFG\_IN e CFG\_OUT definem o sentido da informação passada para o banco de registradores que pode ser de leitura ou escrita. No sentido de escrita do barramento (CFG\_IN), os 32 bits deslocados são chamados de palavras. O conjunto dessas palavras forma um pacote que é processado pelo circuito de configuração do FPGA. Cada primeira palavra do pacote, chamada de cabeçalho, informa o número de palavras desse pacote. O cabeçalho de pacotes de configuração possui também o endereço do registrador do banco e a operação de escrita/leitura a ser executada.

Figura 25 - Exemplo (em hexa) do *bitstream* em formato texto.

```

Xilinx ASCII Bitstream
Created by Bitstream 0.76xd
Design name:   leon3mp.ncd;UserID=0xFFFFFFFF
Architecture:  virtex5
Part:         5vlx110tff1136
Date:        Thu May 10 12:24:11 2012
Bits:       31118848
1 FFFFFFFF 13 AA995566 25 00000000 37 3000C001 49 20000000
  FFFFFFFF 20000000 30026001 00400000 20000000
  FFFFFFFF 30020001 00000000 3000A001 20000000
  FFFFFFFF 00000000 30012001 00400000 20000000
  FFFFFFFF 30008001 000035E5 3000C001 30002001
  FFFFFFFF 00000000 3001C001 00000000 00000000
  FFFFFFFF 20000000 00000000 30030001 30008001
  FFFFFFFF 30008001 30018001 00000000 00000001
  000000BB 00000007 02AD6093 20000000 20000000
  11220044 20000000 30008001 20000000 30004000
  FFFFFFFF 20000000 00000009 20000000 500ED5A0
12 FFFFFFFF 24 30022001 36 20000000 48 20000000 60 00000000

```



Após a síntese física de um circuito a ferramenta BitGen (do inglês, *Bitstream Generator*) da Xilinx é utilizada para a geração do *bitstream* contendo a configuração do FPGA equivalente para o circuito sintetizado. Como o *bitstream* é gerado em formato binário a ferramenta permite a criação de um arquivo de configuração idêntico, porém em formato texto. Nesse arquivo é possível identificar os pacotes de configuração utilizados para o carregamento do *bitstream* no FPGA. Um exemplo desse arquivo é apresentado na Figura 25 onde é importante notar que as instruções para a interface do JTAG são transparentes para o usuário.

A Figura 25 destaca as 60 primeiras palavras do *bitstream* das quais 59 delas são comandos e a última é a primeira palavra a ser carregada na memória de configuração do FPGA. A memória de configuração do FPGA é dividida em quadros de 41 palavras, ou seja, toda operação de escrita/leitura na memória de configuração tem que ser feita em múltiplos de 41 palavras de 32 bits. A Tabela 2 apresenta um resumo das dimensões da memória de configuração e do respectivo *bitstream* para o carregamento do FPGA XC5VLX110T.

Tabela 2 - Resumo de medidas (palavras/quadro/*bitstream*/FPGA).

DIMENSÃO		#	MEDIDA
Palavras		32	bits
Quadro		1.312	bits
		41	palavras
FPGA	Total	24.304	quadros
	Configuração	23.712	quadros
		972.192	palavras
<i>Bitstream</i>	Total	972.464	palavras
	Comandos	272	palavras

No texto descritivo presente no início do arquivo da Figura 25 é informado o tamanho do *bitstream* em bits. Considerando a Tabela 2 é possível perceber que além das 59 palavras de comandos no início do *bitstream* existem mais 213 no final para concluir a configuração.

A ferramenta BitGen permite a geração de outros arquivos no mesmo formato do que foi apresentado na Figura 25 e que são úteis para o entendimento da memória de configuração do FPGA. Esses outros arquivos contêm os bits esperados após uma releitura e comandos para execução do *readback*. Isso permite a verificação da memória de configuração através da comparação do *bitstream* relido e o esperado. Contudo, nem todo o bit do *bitstream* pode ser comparado e para isso existe um terceiro arquivo contendo uma máscara para esses bits. O

componente CAPTURE do FPGA permite a amostragem instantânea de todos os elementos de armazenamento do projeto e para achar os bits capturados dentro do *bitstream*, a ferramenta BitGen possibilita a geração de uma lista com a posição de cada bit, o quadro que se encontra e a posição desse quadro no *bitstream*.

O modelo de descrição VHDL para a instanciação da primitiva de um componente LUT do FPGA Virtex5 é encontrado no manual da ferramenta (47). Uma primitiva corresponde a um elemento nativo do dispositivo, ou seja, a primitiva instanciada na descrição RTL de um projeto será exatamente o componente implementado. No caso do elemento LUT existem 19 primitivas, uma para cada combinação do número de portas de entrada e saída permitidas. Em toda primitiva LUT existe o atributo "INIT" que determina a sua configuração. Esse atributo varia de tamanho de acordo com o número de elementos da tabela verdade, ou seja, o número de entradas da LUT presente nessa primitiva. O exemplo INIT => X"96696996" corresponde a uma primitiva LUT de cinco entradas que implementa a lógica XOR.

Foram criados diversos projetos contendo apenas uma LUT de seis entradas. O atributo INIT da instância das LUTs é o mesmo para todos os projetos e corresponde ao da porta lógica XOR. O valor do INIT foi escolhido considerando o padrão de bits da tabela verdade da lógica XOR que tem menos chance de ser confundido dentro de um *bitstream* se comparado com os padrões das lógicas (N)AND e (N)OR. Em cada projeto a LUT foi implementada num local diferente do FPGA. Para facilitar a procura do valor do INIT nos arquivos de configuração foi utilizada uma versão em formato texto dos *bitstreams* convertida para hexadecimal. O valor de INIT foi encontrado somente em alguns dos *bitstreams* e com os seus oito bytes divididos em quatro linhas distintas do arquivo de configuração. Esses *bitstreams* foram comparados com aqueles em que os bytes do INIT não foram encontrados. A comparação entre os *bitstreams* aponta diversas linhas com diferenças em poucos bits de um byte, no entanto, as linhas que contém a configuração da LUT, apresentam dois bytes diferentes. Com isso foi possível localizar a configuração das LUTs nos *bitstreams* em que o valor do INIT não foi encontrado.

Achado os valores de configuração das LUTs nos *bitstreams*, releituras da memória de configuração do FPGA eram feitas através da programação SVF. A releitura completa do FPGA era comparada com o *bitstream* gerado pela ferramenta BitGen com o objetivo de garantir que o valor de INIT era o mesmo. Releituras de quadros avulsos da memória de configuração eram feitas utilizando a forma de endereçamento

indicada no manual (46) na tentativa de achar os valores de INIT. Considerando que a localização da LUT nos projetos é definida em coordenadas XY do FPGA a conversão para o endereço de quadros não coincidia. Logo, sem o endereço do quadro que contém a configuração da LUT é impossível injetar uma falha no circuito combinacional.

O entendimento da interface JTAG foi imprescindível para o carregamento de um *bitstream* no FPGA e a releitura de quadros da memória de configuração através da programação SVF. A descoberta principal foi a de que os valores do atributo INIT da primitiva da LUT não eram os mesmos encontrados na memória de configuração, mas ainda assim era possível descobrir os valores correspondentes. Contudo, somente com essas informações não é possível achar uma fórmula para encontrar os bits de configuração das LUTs de forma a possibilitar a construção de um emulador de falhas.

#### 4.3.2 Controlador V5SC

Os produtos da Xilinx são desenvolvidos para terem uma baixa suscetibilidade, a SEUs, inerente no dispositivo. Essa afirmação aparece em (48) onde é apresentado uma macro (conjunto de primitivas) que pode ser incluída em um projeto para implementar um circuito de detecção e correção de SEUs. Como estratégia para suportar detecção de SEUs um recurso novo foi introduzido nos FPGAs Virtex5 da Xilinx. Esse recurso, chamado de *Readback CRC* (do inglês, *Cyclic Redundancy Check*), é um circuito intrínseco do FPGA que permite a releitura de forma contínua da memória de configuração do FPGA e calcula seu código CRC32. Após completar a releitura completa da memória pela primeira vez o valor do CRC é guardado para comparação das releituras seguintes de forma que qualquer disparidade pode representar o acontecimento de um SEU.

Considerando que o circuito *Readback CRC* permite somente detecção de erros, um controlador foi disponibilizado pela Xilinx que usa esse recurso com o objetivo de também permitir a correção de erros. A macro desse controlador, introduzida em (48), é chamada de V5SC (do inglês, *Virtex5 SEU Controller*) e utiliza os componentes FRAME\_ECC e ICAP para localização e correção dos erros. Todo quadro de configuração possui 12 bits de paridade (SECDED Hamming) que são calculados na criação do *bitstream* pela ferramenta BitGen. Essa paridade é utilizada pelo FRAME\_ECC que calcula a síndrome do quadro podendo detectar até dois erros e corrigir até um bit invertido. A

leitura e a escrita do quadro corrigido de volta para memória de configuração é feita através do ICAP.

O V5SC é detalhado em (49) e possui um pequeno número de sinais de estado e de controle que permitem a sua integração em um projeto da forma desejada. O controlador V5SC possui uma interface opcional UART que através de um terminal permite o controle da emulação de SEUs e a visualização do processo de detecção e correção do erro. A configuração padrão do V5SC opera a uma frequência de 50MHz (60MHz máx.) e a taxa de transmissão da UART é de 115.200 bits por segundo.

Algumas características do V5SC são dependentes do dispositivo. No XC5VLX110T o V5SC leva 15,72ms para completar uma verificação do FPGA o que equivale a percorrer 19.167 quadros. Esse valor é menor que o tamanho da memória de configuração do FPGA porque o *Readback CRC* não cobre quadros que contenham a configuração SRLs, *Distributed RAM*, BRAM e DRP (do inglês, *Dynamic Reconfigure Port*). Os quadros de configuração desses elementos não são cobertos principalmente porque contêm as inicializações de blocos de memória que geralmente são sobrescritas ao longo da execução do circuito como é explicado em (50).

O número de quadros cobertos (i2) pelo *Readback CRC* aparece no relatório de inicialização do circuito do V5SC enviado para o terminal. Além disso, o relatório informa a versão do V5SC (i1), o código de identificação do FPGA (i3), a situação do código ECC (i5) dos quadros de configuração e valores de alguns sinais do circuito *Readback CRC* (i4). Após o relatório de inicialização o V5SC envia a cada 128 varreduras do FPGA o seu estado onde é informado o número de verificações completadas (e1), o valor do sinal de erro gerado pelo circuito *Readback CRC* (e2), o número de quadros com erro (e3) e o número de bits corrigidos (e4) desde a inicialização do circuito. O estado do V5SC é continuamente mostrado até que se entre na condição de controle através do comando “\*” (\*) como mostra a Figura 26. Nesta figura estão destacadas as informações referenciadas pelos índices dos parênteses.

Com o V5SC na condição de controle existem oito comandos que podem ser usados através da interface UART. Os comandos são explicados na Tabela 3. Alguns desses comandos são mostrados no exemplo da Figura 26 e estão referenciados pelos índices dos parênteses.

Tabela 3 - Comandos do V5SC quando em condição de controle.

s	Leitura da situação atual do V5SC (s1) (s2)
d	Entra em modo de somente detecção de erro (d)

a	Entra em modo de autocorreção (a)
1	Simula erro em um bit aleatório (1)
2	Simula erro em dois bits aleatórios
r	Readback do quadro especificado em "FAddr" (r)
q	Descobre o endereço do quadro a partir do seu número de sequência
t	Inverte um bit escolhido de um quadro especificado

Figura 26 - Exemplo da utilização da interface UART do V5SC.

```

V5SC VA.2 (i1)
IDCODE = 72AD6093 (i2)
RBCRC_EN = 0
RBCRC_EN = 1 (i3)
GLUTMASK_B = 0
ECC Checks (i4)
Size = 4ADF (i5)
000080 0 00 00
000100 0 00 00
000180 0 00 00 (e4)
(e1) : (e2) (e3)
      :
012F80 0 00 06
013000 0 00 06* (*)

>s (s1)
013448 0 00 06 ACM

>d (d)
DOM

>1 (1)
SimSEU 1309 150

>a (a)
ACM

> (bi)
SBE 1309 150 010300 C30

>s (s2)
014DF5 0 00 07 ACM

>r (r)
FAddr = 100881
00000000 00000000 00000000 00000800 00008000
00480800 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000
00000000 00002000 00000800 00400000 00000010
0000058E
00000000 00000000 00000000 00000000 00000000
00000000 00002000 00000000 00002000 00000000
00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00010000 00000000

```

O exemplo apresentado na Figura 26 coloca o V5SC em condição de controle (\*), captura a situação atual de V5SC (s1), entra em modo de operação DOM (do inglês, *Detection Only Mode*), inverte um bit aleatório (1), retorna para o modo de operação ACM (do inglês, *Auto Correction Mode*), automaticamente surge à notificação de correção do bit invertido (bi) e então é feita uma nova leitura da situação do V5SC (s2) onde é possível perceber que o contador de erros corrigidos foi incrementado para 07.

A Figura 26 mostra que após o comando de inversão de um bit aleatório (1), automaticamente surge o aviso de tentativa de simulação de erro. Esse aviso é composto com as informações da tentativa que foi feita (“SimSEU”), o número de sequência do quadro (“1309”) e qual dos seus 1312 bits foi invertido (“150”). Além dessas informações, a notificação de correção do bit invertido (be) apresenta tipo de erro ocorrido (“SBE” do inglês, *Single Bit Error*), o endereço do quadro (“010300”) e o resultado da síndrome (“c30”) calculada pelo FRAMER\_ECC.

Ainda na Figura 26 é apresentada a leitura do quadro de endereço 100881 (24 bits em hexadecimal). O quadro é apresentado com a palavra 21 entre as primeiras e últimas 20 palavras que estão divididas em linhas com cinco palavras (hexadecimal). A palavra 21 contém os 12 bits de paridade do ECC calculado para o quadro (“58E”).

Uma LUT foi prototipada junto com controlador V5SC. Cinco entradas da LUT foram ligadas em chaves do kit XUPV5-LX110T e a saída da LUT e as chaves foram ligadas em LEDs da placa. A tabela verdade da LUT foi definida no VHDL com o valor INIT => X"96696996", ou seja, o conteúdo da memória de configuração da LUT corresponde à função lógica de uma XOR. A localização da LUT foi restringida para a LUT\_D do primeiro *slice* da parte inferior esquerda do FPGA. Com essa configuração, diversas tentativas de inversão de bits foram feitas até que o resultado da LUT fosse afetado. Isso permitiu descobrir que é factível a inversão da lógica de uma LUT através do ICAP durante o funcionamento do circuito. Contudo, ainda não é possível calcular o endereço dos quadros que contém a configuração de todas as LUTs do FPGA.

Um programa utilizando a linguagem Perl foi desenvolvido para automatização dos testes realizados com o V5SC. O programa executado em um computador hospedeiro usa a interface UART do controlador V5SC para a execução dos comandos. Considerando que 24 bits são utilizados para o endereçamento dos quadros nos comandos do V5SC, existem mais de 16 milhões de possibilidades de endereços.

Utilizando o comando “q” do controlador que converte o número de sequência do quadro de configuração para o respectivo endereço foi possível gerar uma lista com todos os endereços possíveis da memória do FPGA. Nessa lista é possível perceber que os endereços são incrementados de um até que ocorre um salto para um valor distante. Entre esses saltos estão segmentos de endereços contínuos. Esses segmentos representam pilhas de quadros. O tamanho da pilha varia de acordo com tipo de componente (CLBs, BRAMs, IOBs, etc.) que os quadros dessa pilha configuram. Esse conceito de pilhas de quadros fica claro na próxima seção onde é utilizado para cálculo do endereço dos quadros que contém a configuração da LUT. No entanto, ainda não é possível determinar exatamente quais os bits dos quadros representam a configuração de uma LUT.

Diferentemente dos controladores de SEU para as famílias seis e sete da Xilinx que podem ser obtidos através da ferramenta *CORE Generator™* da Xilinx, o controlador V5SC é disponibilizado somente sob requisição formal. A aprovação da requisição foi concedida pela Xilinx e permite o acesso ao *Soft Error Mitigation (SEM) Core Lounge* sob o acordo da licença *LogicCORE™ IP* que proíbe a edição do RTL fonte do material disponibilizado a menos que indicado para tal. Contudo, é possível analisar o RTL do V5SC e perceber que utiliza um micro controlador *PicoBlaze™ 8-Bit* para acessar o ICAP e *FRAME\_ECC* e controlar as filas da interface UART. O V5SC é uma macro, ou seja, contém basicamente instâncias de primitivas e quase nada de descrição comportamental e a imagem do programa executado pelo *PicoBlaze* se encontra em forma de inicialização de uma BRAM o que dificultaria a adaptação do V5SC para ser utilizado nesse trabalho, além de ser proibido.

Antes de seguir adiante no trabalho, para não haver confusão é de fundamental importância perceber que na documentação da Xilinx é utilizado somente o termo SEU devido ao fato do FPGA ser basicamente uma memória SRAM e as falhas que ocorrem nesse dispositivo são quase sempre nas células de armazenamento. Esse trabalho foi desenvolvido com o objetivo de utilizar a reconfiguração da memória do FPGA para possibilitar a emulação de falhas tanto na parte combinacional (SETs) quanto na parte sequencial (SEUs) dos *slices* implementados nos circuitos testados.

### 4.3.3 Internal Access Configuration Port (ICAP)

Descoberto que é possível alterar a lógica das LUTs e a forma como estão organizados os quadros contendo a configuração das LUTs falta somente achar a posição exata dos seus 64 bits. Isso possibilita a inversão completa dos 64 bits e consequentemente da equação lógica implementada pela LUT.

O uso do V5SC para a descoberta de um algoritmo que resultasse na posição dos 64 bits de configuração da LUT foi descartado devido à ineficiência de utilizar a interface UART, além do que o V5SC usa a porta ICAP com frequência de 50MHz que corresponde à metade da frequência máxima de operação descrita no manual do dispositivo. Por isso, a ferramenta EDK (do inglês, *Embedded Development Kit*) da Xilinx foi utilizada para montar um sistema embarcado simples com um processador com acesso ao componente ICAP. EDK é um ambiente de desenvolvimento de projetos de sistemas embarcados processados que integra as ferramentas XPS (do inglês, *Xilinx Platform Studio*) e SDK (do inglês, *Software Development Kit*). A XPS possui uma interface gráfica que guia a montagem de sistemas embarcados complexos contendo processadores, barramentos e periféricos, em minutos. Com o sistema embarcado arquitetado e implementado é a vez do desenvolvimento do software que é auxiliado pelo SDK e que inclui compiladores, bibliotecas, ferramentas de depuração, terminais, editor de texto, entre outros.

O sistema embarcado construído é baseado no processador Microblaze™ que através de barramentos permite a associação de vários periféricos presentes no catálogo de IPs da Xilinx. Um desses periféricos é o XPS\_HWICAP que controla o componente ICAP possibilitando o acesso à memória de configuração do FPGA. O *driver* desse periférico possui diversas funções prontas do tipo leitura/escrita de um quadro que facilitam em muito o uso do ICAP. Juntamente com o *driver* existem alguns exemplos de código em linguagem C utilizando as funções para o XPS\_HWICAP. A utilização do XPS e do SDK aliado a maiores detalhes do processador e do periférico são apresentados nas seções 5 e 6.

Algumas LUTs foram instanciadas junto ao sistema embarcado da mesma forma que foi feita com o controlador V5SC e na releitura do FPGA pelo JTAG. Dois dos exemplos disponibilizados junto ao XPS\_HWICAP realizam a leitura e a escrita na memória de configuração. O primeiro exemplo mostra o uso da função leitura de um quadro para a releitura do *slice* inferior esquerdo do FPGA. Quatro



LUTs foram instanciadas no *slice* inferior esquerdo do FPGA e o quadro lido pelo exemplo não continha a informação da configuração de nenhuma das LUTs. Porém, com a facilidade da programação da MicroBlaze através do SDK foi possível fazer diversos testes até que se descobriu que os 256 bits das quatro LUTs de cada *slice* estão dispostos em quatro quadros consecutivos. Cada quadro (1.312 bits ou 41 palavras) contém 64 bits de configuração das quatro LUTs de um *slice* e estão divididos em duas palavras. Cada metade da palavra contém 16 bits de configuração de uma LUT, ou seja, os próximos 16 bits dessa LUT estão a 41 palavras de distância, e assim por diante. Logo, para inverter a lógica de uma LUT é preciso negar os 16 bits da palavra de cada um dos quatro quadros. Dessa forma foi construída a base do emulador de falhas desenvolvido nesse trabalho.

O segundo exemplo mostra o uso de uma função que inverte os 64 bits de uma LUT de uma única vez, ou seja, sem a necessidade de realizar quatro vezes a leitura de um quadro, inversão dos 16 bits da palavra da LUT e escrita desse quadro. Contudo, essa função nunca retornou uma resposta esperada mesmo quando os parâmetros do exemplo anterior, onde foi obtido sucesso, foram utilizados. O código fonte dessa função é disponibilizado pré-compilado, diferentemente das outras funções da biblioteca de *drivers* do SDK. Dessa forma, não há como descobrir a maneira que a Xilinx encontrou para acessar os 64 bits de uma LUT de uma única vez, ainda mais, se considerar que o acesso a memória de configuração é feito através da leitura/escrita de no mínimo 1.312 bits (um quadro). Há inúmeras reclamações no fórum da Xilinx relacionadas ao não funcionamento da função e nenhuma com uma solução. O funcionamento dessa função teria influência relevante no desempenho do emulador de falhas desenvolvido nesse trabalho.

#### 4.4 COORDENADAS X ENDEREÇO DE QUADROS DE MEMÓRIA

Os elementos do FPGA estão distribuídos em forma de uma matriz. Coordenadas XY são utilizadas para a localização desses elementos no projeto. No caso dos CLBs, as coordenadas XY determinam a posição dos *slices* como mostra a Figura 27. As coordenadas XY dos *slices* são diferentes das coordenadas XY utilizadas para localizar as BRAMs, IOBs entre outros. A Figura 27 mostra a parte inferior esquerda do FPGA. Nessa figura fica clara a linha vertical de *carry* que permite o encadeamento dos circuitos aritméticos dos *slices*.

Figura 27 - Significados das coordenadas XY para os CLB.

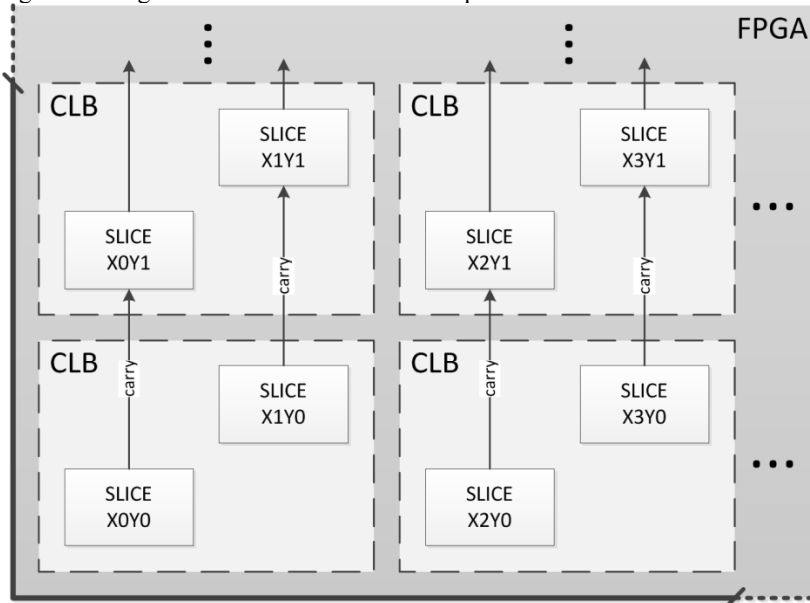


Figura 28 - Campos da palavra de 32 bits de endereço.

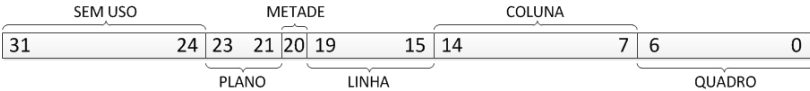


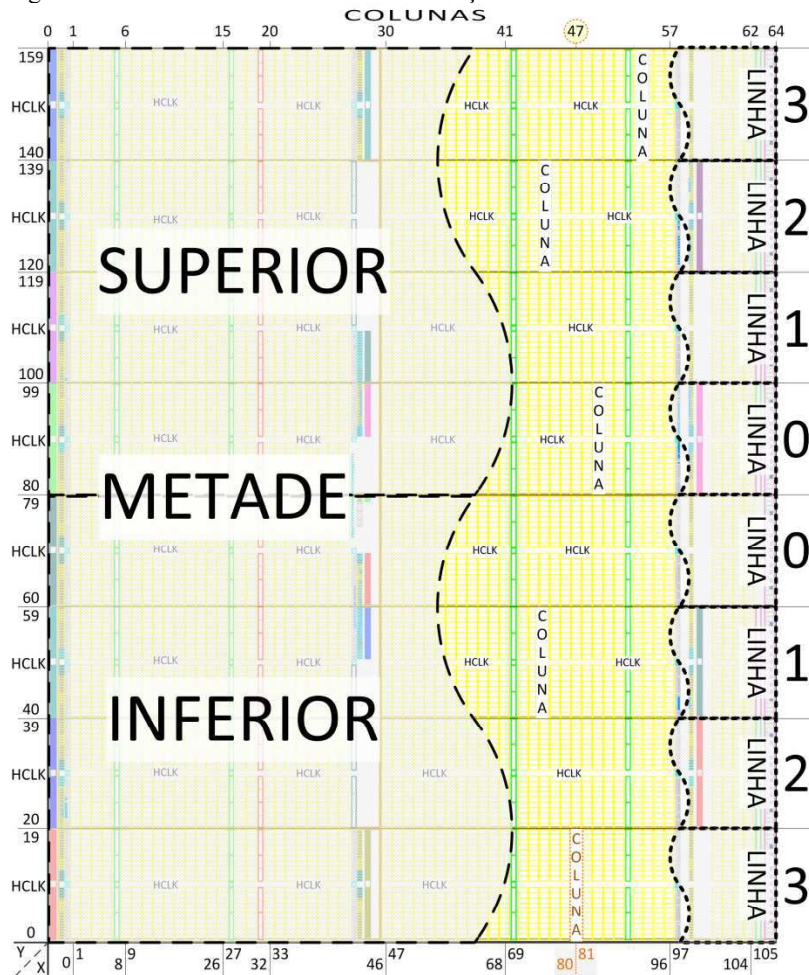
Tabela 4 - Descrição dos campos da palavra de endereço do FPGA.

CAMPOS	BITS	DESCRIÇÃO
PLANO	[23:21]	(000) – Configuração da matriz de roteamento e dos parâmetros dos componentes do FPGA. (001) – Conteúdo das memórias BRAMs (010) – Quadros especiais utilizados para reconfiguração parcial (RECONFIG_P) (011) – Reservado. Quadros das BRAMs
METADE	[20]	(0/1) SUPERIOR / INFERIOR
LINHA	[19:15]	Linha do FPGA. Começa em zero (meio) e cresce em direção as extremidades
COLUNA	[14:7]	Coluna da linha. Começa em zero (esquerda) e cresce para esquerda
QUADRO	[6:0]	Número do quaro na pilha. Começa em zero.

Diferentemente do sistema de localização por coordenadas a memória de configuração do FPGA utiliza um sistema de endereçamento de quadros. As palavras de endereço da memória de

configuração têm tamanho 32 bits que são divididos em campos como mostra a Figura 28. A Tabela 4 descreve o significado de cada campo e seus respectivos números de bits.

Figura 29 - Sistema de coordenadas X endereçamento.



A Figura 29 mostra a representação do sistema de endereçamento no FPGA e compara com as coordenadas XY. A figura mostra o eixo X (abaixo) e o eixo Y (esquerda) do sistema de coordenadas de *slices* apresentado na Figura 27 para o FPGA XC5VLX110T. Como visto no início dessa seção (Tabela 1), esse FPGA possui 160 CLBs de “altura” e 54 de “largura” o que corresponde a um eixo Y de zero até 159 e o X de

zero até 105 *slices*. No lado direito e acima da Figura 29 está a representação utilizada pelo sistema de endereçamento na forma de linhas (à direita) e colunas (acima). Diferentemente das coordenadas XY onde os eixos variam de acordo com o tipo de elemento o sistema de LIN/COL acessa todos os componentes.

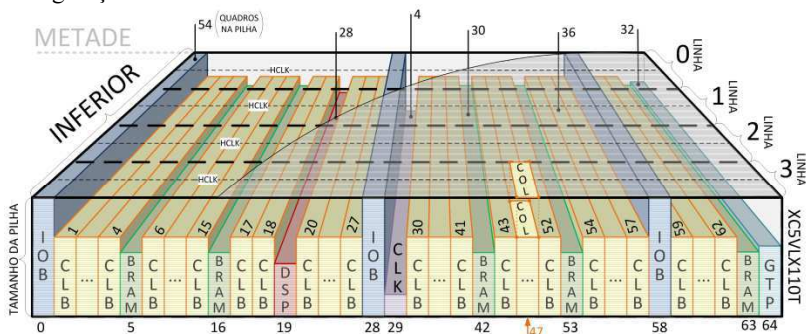
Na visão da memória de configuração, o FPGA está repartido em metades superior e inferior que por sua vez estão divididas em linhas. A contagem das linhas é feita do centro do FPGA para as extremidades. Cada coluna da linha abrange um número determinado de componentes (20 CLBs, 40 IOBs, 8 DSPs, 4 BRAMs, etc.). No meio de cada coluna se encontra HCLK que é uma divisória horizontal das regiões de sinal de relógio como mostra a Figura 29.

Na parte de cima da Figura 29 é mostrado o número da coluna de CLBs referente aos dois valores do eixo X do sistema de coordenadas dos *slices*. A primeira (0) e a última (64) coluna não possuem um valor de X respectivo, pois são compostas por IOBs e GTPs, respectivamente, ou seja, os *slices* da extrema direita do eixo X, por exemplo, se encontram na coluna 62.

Os FPGAs de uma mesma família (Virtex4, Virtex5, Spartan6, entre outros) da Xilinx diferem de acordo com a aplicação para a qual foram desenvolvidos. A diferença maior entre os modelos, além da quantidade de recursos, é a existência de componentes específicos para aplicação como um processador integrado dentro do FPGA, interfaces de comunicação com alto desempenho ou dez vezes mais recursos de DSP. No caso do FPGA XC5VLX110T a terminação ‘T’ se refere à disponibilidade dos componentes GTPs. Considerando que os componentes do FPGA estão dispostos em longas colunas, o índice das colunas é também utilizado para endereçar um tipo de componente. Logo, o endereçamento de CLBs, por exemplo, é afetado diretamente pela quantidade de recursos do modelo e principalmente pela existência de componentes específicos de aplicação.

O último campo do endereço da memória de configuração não pode ser observado na visão que a Figura 29 proporciona. A Figura 30 ilustra o que seria a projeção em perspectiva da metade inferior do FPGA para explicar o conceito de pilhas de quadros. A mesma coluna (47) destacada na Figura 29 é apontada na Figura 30 buscando mostrar a diferença de perspectiva entre as duas figuras. Na Figura 30 as colunas de CLB consecutivas estão agrupadas para mostrar os índices de todas as colunas do FPGA XC5VLX110T.

Figura 30 - Visão em profundidade da metade inferior da memória de configuração do FPGA.



A Figura 30 destaca a diferença do tamanho das pilhas de quadros de cada coluna. O tamanho da pilha varia com o tipo de elemento que constitui a coluna. A Tabela 5 apresenta os tamanhos das pilhas para todos os tipos de colunas encontradas no FPGA XC5VLX110T. Esta tabela mostra também que os primeiros 26 quadros da pilha, exceto o da coluna de CLK, contém informações de interconexão. Nos quadros 26 e 27 das colunas do tipo IOB, BRAM, DSP e GTP estão os bits referentes à interface da coluna. Nos quadros restantes estão as configurações específicas dos elementos que constituem a coluna. A pilha abrange a configuração de um número diferente de componentes que depende do tipo de coluna. O número de componentes é mostrado na última coluna da Tabela 5.

Tabela 5 - Tamanho da pilha de cada tipo de coluna.

TIPO	TAMANHO DA PILHA	ROTEAMENTO [0-25]	INTERFACE [26-27]	NÚMERO DE COMPONENTES
IOB	54	X	X	40
CLB	36	X	-	20 (40 SLICES)
BRAM	30	X	X	4
DSP	28	X	X	8
CLK	4	-	-	-
GTP	32	X	X	2

A pilha da coluna (47) destacada nas Figura 29 e Figura 30 está dissecada na Figura 31. A coluna 47 é do tipo CLB e por isso os 26 primeiros quadros [0-25] da pilha são referentes à interconexão da coluna. Os 10 quadros restantes que contém os bits de configuração dos 20 CLBs (40 *slices*) são divididos de acordo com o que configuram. Os

quatro quadros [26-29] configuram as 80 LUTs dos 20 *slices* com coordenada X ímpar enquanto os últimos quatro quadros da pilha [32-35] configuram os *slices* com coordenada X par. No caso da coluna 47 os quadros [26-29] e [32-35] da pilha configuram as LUTs dos *slices* com coordenadas X 81 e 80, respectivamente. Os dois quadros que sobram 29 e 30 configuram os outros elementos dos *slices* como, FF/LATCH, CARRY, etc.

A palavra de número 20 dos quadros da pilha configura a divisória HCLK. O significado dos bits da palavra de configuração de HCLK é igual para todos os quadros da pilha e por isso é mostrado duas vezes na Figura 31. Os 12 bits iniciais [11:0] são da paridade ECC, os quatro bits [15:12] são particulares da configuração do circuito do HCLK e o restante dos bits não são utilizados. A Figura 31 mostra a posição dos bits de HCLK referente aos 1312 bits do quadro. Os 10 CLBs abaixo da divisória HCLK são configurados pelas vinte [0-19] palavras do quadro antes da palavra de HCLK [20], logo, os 640 bits que restam no quadro configuram os outros 10 CLBs que estão acima de HCLK. Ainda na Figura 31 estão destacadas as oito palavras que configuram as quatro LUTs (256 bits) do *slice* de coordenada X81Y19. Essas oito palavras são as de número 39 e 40 dos quatro quadros [26-29].

A Figura 32 detalha a distribuição dos bits de configuração das quatro LUTs do *slice* X81Y19. Os intervalos de bits [31:16] e [15:0] da palavra 39 do quadro 26 correspondem aos 16 bits iniciais da configuração das LUTs B e A, respectivamente. A palavra de número 39 dos quadros subsequentes (27, 28 e 29) completam os 64 bits de configuração da LUT B e os 64 bits da LUT A do *slice* X81Y19. O mesmo se aplica para os 64 bits de configuração da LUT C e os 64 bits da LUT D que se encontram na última palavra (40) dos quadros 26, 27, 28 e 29 como mostra a figura.

O código em Perl da função que realiza a conversão das coordenadas XY em um endereço de memória é apresentado na Figura 33. A função tem como parâmetros as coordenadas X e Y e a letra referente a uma das LUTs do *slice* e retorna o endereço do primeiro quadro que contém os 16 bits iniciais de configuração da LUT. Além do endereço do primeiro quadro a função retorna a máscara para separar os 16 bits da LUT e o número da palavra do quadro em que os 16 bits estão. O pseudocódigo possui valores específicos para o FPGA XC5VLX110T, contudo, pode ser facilmente generalizado com uma biblioteca contendo as características de cada modelo de FPGA.

Figura 31 - Pilha da coluna (47) que é do tipo CLB.

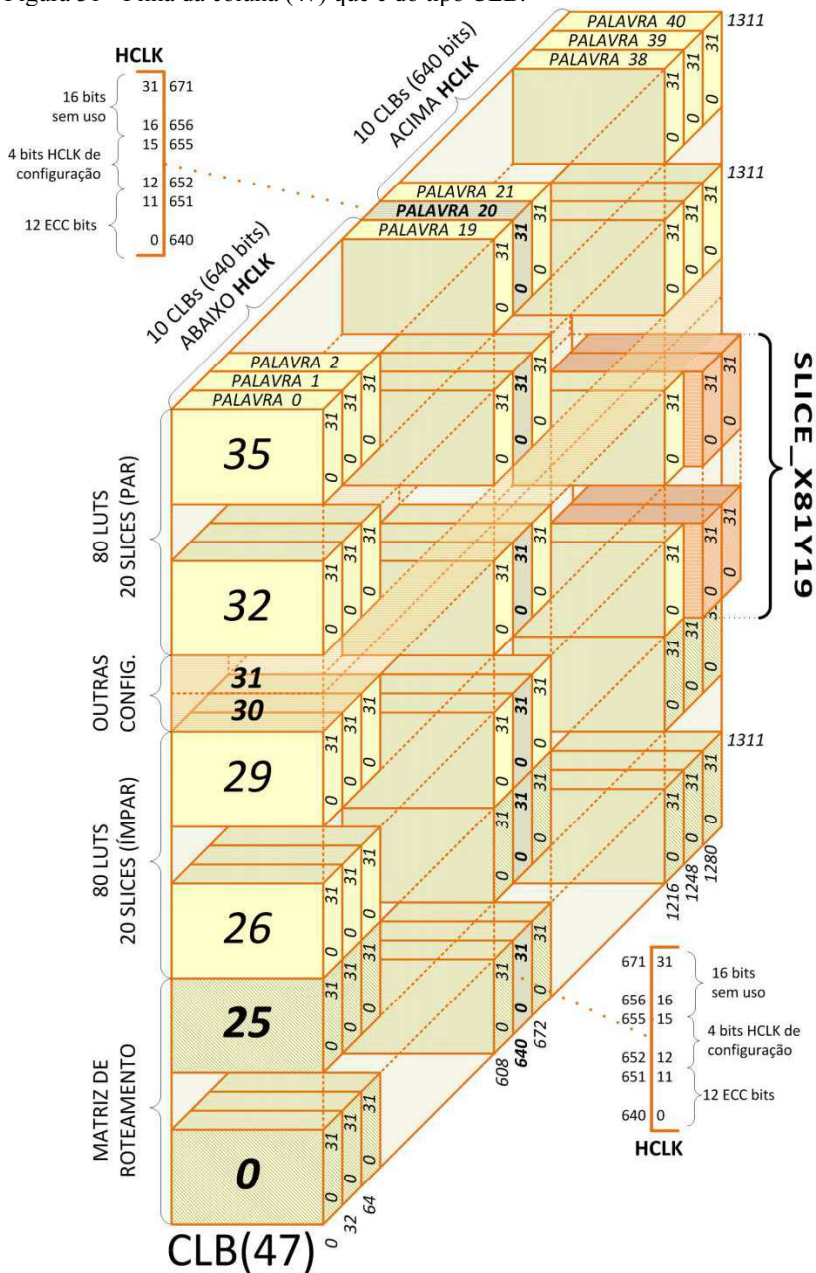
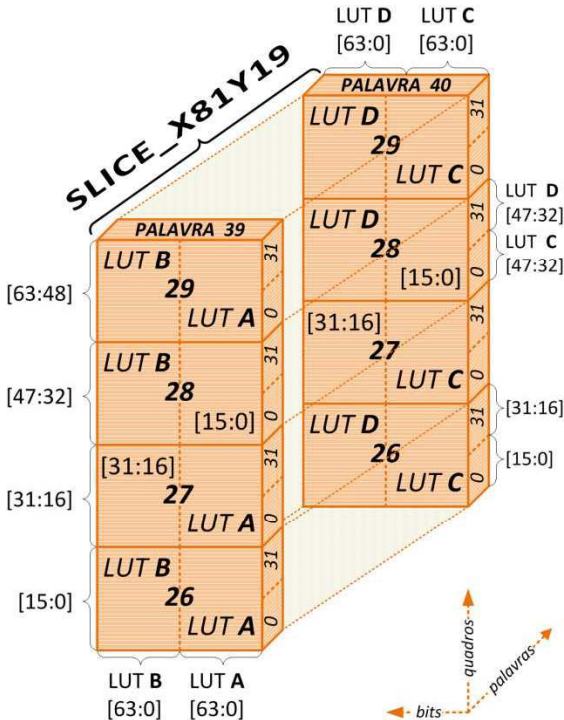




Figura 32 - As oito palavras de configuração das LUTs do *slice* X81Y19.

O carregamento do *bitstream* no FPGA segue a ordem da contagem crescente do valor do endereço, ou seja, começa no meio do FPGA e segue para o canto superior direito e então volta para o meio continuando na direção da ponta inferior direita do FPGA. O incremento do endereço é feito automaticamente pelo circuito de configuração do FPGA que busca otimizar longas escritas/leituras em rajadas. A ordem em que a memória de configuração do FPGA é carregada pelo *bitstream* é apresentada na Figura 34. A figura destaca o SALTO que acontece na mudança para a próxima linha, ou seja, ocorre uma descontinuidade grande na sequência do endereçamento. Entre esses saltos existem dois quadros nulos no *bitstream*. Esses quadros existem para que o circuito de configuração limpe as informações da linha anterior e acerte o ponteiro para a execução leitura/escrita da próxima linha como descrito em (49). Ao final da escrita dos quadros de configuração do FPGA começa a transferência do conteúdo das BRAMs.



Figura 33 - Função de conversão da coordenada XY para o endereço da memória de configuração do FPGA XC5VLX110T.

```

sub slice_xy2faddr{
  my ($x, $y, $lut) = @_ ;
  my @clb_cols = (1 .. 4,6 .. 15,17 .. 18,20 .. 27,
                  30 .. 41,43 .. 52,54 .. 57,59 .. 62) ;

  ### METADE ###
  if($y < 160/2){ $faddr = (1 << 20); } #INFERIOR
  else{ $faddr = 0; } #SUPERIOR

  ### LINHA ###
  if( ($y < 20) || ($y >= 140)){ $faddr |= (3 << 15); } #linha 3
  elsif(($y < 40) || ($y >= 120)){ $faddr |= (2 << 15); } #linhas 2
  elsif(($y < 60) || ($y >= 100)){ $faddr |= (1 << 15); } #linha 1
  #else{ $faddr |= (0 << 15); } #linha 0

  ### COLUNA ###
  $faddr |= ($clb_cols[($x & ~0x1)/2] << 7);

  ### QUADRO ###
  if($x & 0x1){ $faddr |= 26; } #coordenada X impar
  else{ $faddr |= 32; } #coordenada X par

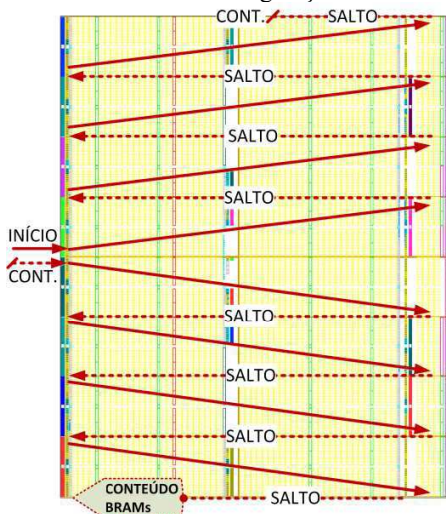
  ### PALAVRA ###
  $word = (($y-(floor($y/20)*20)) * 2); #[0-19]CLBs * 2 (slices)
  $word += (($word >= 20) ? 1 : 0); #pula palavra HCLK
  $word += ((($lut eq 'C')||($lut eq 'D')) ? 1 : 0);

  ### MASCARA DE 16 BITS ###
  $mask = (((($lut eq 'A')||($lut eq 'C')) ? 0x0000FFFF : 0xFFFF0000);

  return ($faddr, $word, $mask);
}

```

Figura 34 - Trajetória do *bitstream* de configuração do FPGA.



Enquanto a definição dos parâmetros das BRAMs fica no plano de configuração (000) o conteúdo das BRAMs fica em outro plano. Nesse plano existe uma coluna para cada coluna do tipo BRAM existente no plano de configuração. As colunas do plano (001) possuem pilhas com o tamanho de 128 quadros contendo o conteúdo das quatro BRAMs da respectiva coluna. No FPGA XC5VLX110T existem cinco colunas de BRAMs por linha, logo, são 640 quadros com o conteúdo das BRAMs por linha. Além do plano de configuração (000) e do plano com o conteúdo das BRAMs (001) existem mais dois planos. Os quatro planos de memória do FPGA são mostrados na Figura 35 que apresenta o que seria a visão de um corte transversal do FPGA.

Figura 35 - Planos da memória do FPGA da Xilinx.

XC5VLX110T	
0	0 1 ... 4 5 6 ... 15 16 17 18 19 20 ... 27 28 29 30 ... 41 42 43 ... 52 53 54 ... 57 58 59 ... 62 63 64
1	0 1 ... 4 5 6 ... 15 16 17 18 19 20 ... 27 28 29 30 ... 41 42 43 ... 52 53 54 ... 57 58 59 ... 62 63 64
2	0 1 ... 4 5 6 ... 15 16 17 18 19 20 ... 27 28 29 30 ... 41 42 43 ... 52 53 54 ... 57 58 59 ... 62 63 64
3	0 1 ... 4 5 6 ... 15 16 17 18 19 20 ... 27 28 29 30 ... 41 42 43 ... 52 53 54 ... 57 58 59 ... 62 63 64

O último plano (011) do FPGA também possui quadros referentes às BRAMs, contudo, esses quadros não são configuráveis, ou seja, somente leitura. Diferentemente do plano (001), o tamanho da pilha das colunas do último do plano (011) é de apenas um quadro de memória cada. O plano remanescente (010) contém um quadro para cada coluna do FPGA contendo os bits de configurações utilizados para projetos com reconfiguração parcial. Os quadros dessa coluna têm o mesmo modelo dos quadros de HCLK do plano de configuração (12 bits de ECC, 4 bits específicos e 16 sem uso).

Os quadros dos planos (010) e (011) não contribuem para o tamanho do *bitstream*. A Tabela 6 mostra um resumo do número de quadros que constituem cada plano. No plano de configuração (000) a tabela apresenta o número de quadros das pilhas para cada tipo de coluna de componentes. Os dois quadros de SALTO adicionais para a leitura/escrita do FPGA em modo rajada existem para todos os planos e estão destacados na tabela. Na coluna mais a direita da tabela é mostrado o total de quadros de cada plano que contribuem para o tamanho do *bitstream*. A última linha da tabela mostra o total de quadros encontrados no FPGA e o número presente no *bitstream*.

As informações da Tabela 6 permitem o cálculo da localização dos 64 bits de configuração das LUTs no *bitstream*. A Figura 36 mostra a função para cálculo da posição da palavra contendo os 16 bits iniciais de configuração da LUT. A função retorna o índice dessa palavra no *bitstream* e a máscara para poder separar os 16 bits de configuração.

Essa função trabalha com o endereço do quadro de memória então os parâmetros que são as coordenadas X e Y e a letra da LUT são passados para a função da Figura 33 que retorna o endereço do quadro da LUT. A função utiliza os valores da Tabela 6 e repete a trajetória do *bitstream* apresentado na Figura 34 e vai acumulando os quadros que estão antes do quadro com a configuração da LUT. O valor acumulado de quadros é convertido para o número de palavras multiplicando-se por 41 (número de palavras em um quadro). A posição da palavra no quadro que contém os 16 bits é adiciona ao índice. Antes de retornar o índice da palavra com os 16 bits de configuração da LUT no *bitstream* são adicionadas as 59 palavras de comandos de configuração do FPGA. A função apresentada na Figura 36 retorna o índice da palavra e a máscara para separar os 16 bits de configuração da LUT. Incrementando-se o índice retornado pela função em 41 é possível encontrar os outros 48 bits de configuração da LUT.

Tabela 6 - Resumo do número de quadros nos diferentes planos do FPGA.

#	PLANO/TIPO	TAMANHO DA PILHA	TOTAL DE COLUNAS	QUADROS POR LINHA	TOTAL DE QUADROS	TOTAL BITSTREAM
0	IOB	54	3	162	1296	18576
	CLB	36	54	1944	15552	
	BRAM	30	5	150	1200	
	DSP	28	1	28	224	
	CLK	4	1	4	32	
	GTP	32	1	32	256	
	SALTO	2	1	2	16	
1	CONTEÚDO	128	5	640	5120	5136
	SALTO	2	1	2	16	
2	RECONFIG_P	1	65	65	520	-
	SALTO	2	1	2	16	-
3	RESERVADO	1	5	5	40	-
	SALTO	2	1	2	16	-
<b>TOTAL DE QUADROS DA MEMÓRIA DO FPGA</b>					<b>24304</b>	<b>23712</b>

Algumas adaptações nas funções apresentadas na Figura 33 e na Figura 36 devem ser feitas para abranger FPGAs de famílias diferentes,

mas a lógica de endereçamento da memória de configuração é basicamente a mesma para os FPGAs da Xilinx. O pseudocódigo mais genérico está disponibilizado no apêndice desse trabalho.

Figura 36 - Função que converte as coordenadas de um *slice* no índice da palavra do *bitstream* com os 16 bits iniciais da configuração da LUT

```

sub slice_xy2getbits{
    my ($x, $y, $lut) = @_;

    ### CONVERTE COORDENADA XY PARA ENDEREÇO DE MEMÓRIA ###
    my ($faddr, $word, $mask) = slice_xy2faddr($x, $y, $lut);

    ### TOTAL DE QUADROS DO PLANO (000) DO FPGA ###
    my $total_frames_0 = 18576;
    my $frames_per_half = $total_frames_0/2;
    my $frames_per_row = $frames_per_half/4;

    ### POSICAO DE CADA TIPO DE COLUNA DO FPGA ###
    my @iob_cols = (0, 28, 58);
    my @brm_cols = (5, 16, 42, 53);
    my @dsp_cols = (19);
    my @clk_cols = (29);
    my @gtp_cols = (64);
    my @clb_cols = (1 .. 4,6 .. 15,17 .. 18,20 .. 27,
                   30 .. 41,43 .. 52,54 .. 57,59 .. 62);

    ### INDICE DA PRIMEIRA PALAVRA DE CONFIGURACAO DA LUT ###
    my $bitstream_index = 0;

    ### ACUMULA METADE DO TOTAL DE QUADROS SE A LUT FOR DA METADE INFERIOR ###
    $bitstream_index += (($faddr >> 20) & 0x1) * $frames_per_half;

    ### ACUMULA O TOTAL DE QUADROS POR LINHA ###
    $bitstream_index += (($faddr >> 15) & 0x3) * $frames_per_row;

    ### ACUMULA OS QUADROS DAS PILHAS DE CADA COLUNA ###
    $col = (($faddr >> 7) & 0xFF);
    foreach (@iob_cols){ $bitstream_index += (($col > $_) ? 54 : 0); }
    foreach (@brm_cols){ $bitstream_index += (($col > $_) ? 30 : 0); }
    foreach (@dsp_cols){ $bitstream_index += (($col > $_) ? 28 : 0); }
    foreach (@clk_cols){ $bitstream_index += (($col > $_) ? 4 : 0); }
    foreach (@gtp_cols){ $bitstream_index += (($col > $_) ? 32 : 0); }
    foreach (@clb_cols){ $bitstream_index += (($col > $_) ? 36 : 0); }

    ### ACUMULA OS QUADROS DA PILHA EM ANTES DO QUADRO DE CONFIG. DA LUT ###
    $bitstream_index += ($faddr & 0x7F);

    ### CONVERTE A QUANTIDADE DE QUADROS PARA O NUMERO DE PALAVRAS ###
    $bitstream_index *= 41;

    ### ADICIONA A POSICAO DA PALAVRA NO QUADRO QUE CONTEM A CONFIG. DA LUT ###
    $bitstream_index += $word;

    ### ADICIONA AS 59 PALAVRAS DE CONFIGURACAO DO INICIO DO BITSTREAM ###
    $bitstream_index += 59;

    ### RETORNA A POSICAO EXATA DA PRIMEIRA PALAVRA DE CONFIG. DA LUT ###
    return ($bitstream_index, $mask); ### RETORNA A MASCARA DOS 16 BITS ###
}

```

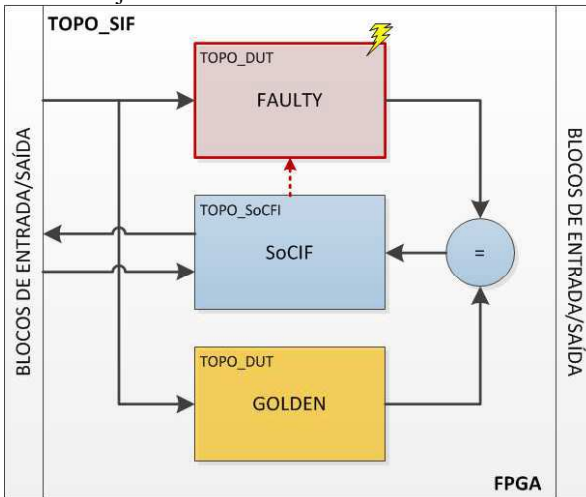
## 5 PLATAFORMA DE EMULAÇÃO DE SOFT ERRORS

Este capítulo apresenta os elementos que compõem a PLAESER (Plataforma de Emulação de *Soft Errors*). A compreensão da descrição da PLAESER contida nessa seção auxilia no entendimento do fluxo proposto para a prototipação rápida da plataforma de emulação de *soft error*. O fluxo proposto será detalhado na seção 6.

A sensibilidade à radiação de circuitos com tecnologias cada vez menores trazem a necessidade da utilização de técnicas de tolerância a falhas até mesmo para dispositivos eletrônicos que operam em nível do mar. Devido a essa miniaturização dos circuitos integrados, novas técnicas de projeto visando maior confiabilidade vêm sendo desenvolvidas. Para que esses circuitos possam apresentar características de tolerância a falhas, e as respectivas técnicas aplicadas se consolidem é preciso testar esses projetos para avaliar seu comportamento na presença de falhas. O projeto a ser testado é denominado DUT.

A plataforma PLEASER tem a finalidade de avaliar de forma experimental técnicas de tolerância a falhas projetadas para circuitos críticos. A metodologia utilizada pela PLAESER é a simulação de um ambiente hostil por emulação de falhas utilizando circuitos reconfiguráveis, os FPGAs. A capacidade de reconfiguração dos FPGAs é utilizada pelo PLEASER para injeção das falhas. As campanhas de injeção buscam verificar o comportamento lógico do circuito na presença de falhas. Isso é feito invertendo a função lógica de partes do circuito. O efeito causado pela falha injetada no resultado do circuito é que define a tolerância do projeto e determina se a falha se tornou um erro ou não. Para isso são utilizadas duas instâncias do DUT para comparação de seus resultados, sendo que as falhas são injetadas somente em um deles, chamado aqui de FAULTY, enquanto o outro opera livre de falhas, apelidado de GOLDEN. O SoC injetor de falha, nomeado SoCIF, baseado no *soft-processor* MicroBlaze™ da Xilinx, é responsável pelo controle e execução das campanhas de falhas do PLAESER. A geração das campanhas de falhas do PLAESER é tarefa do aplicativo desenvolvido GLIFA (Gerador de Lista de Falhas). A Figura 37 ilustra o sistema injetor de falhas (SIF) carregado no FPGA para análise experimental do DUT pela emulação das falhas geradas pelo GLIFA e injetadas pelo SoCIF na instância FAULTY do DUT.

Figura 37 - Sistema Injetor de Falhas.



A PLAESER pode ser dividida em elementos de acordo com as suas funções que permitem a análise de técnicas de tolerância aplicadas a circuitos críticos por meio de emulação de falhas. Os elementos são:

- Circuito reconfigurável – FPGA;
- Ferramentas de síntese e implementação;
- GLIFA – Gerador de lista de falhas;
- SoCIF – *System-on-a-chip* Injetor de Falhas
  - Arquitetura - Hardware
  - Funcionalidade – Software

Nas subseções que constituem essa seção são detalhadas as funções de cada elemento e suas respectivas importâncias para a PLAESER.

### 5.1 CIRCUITO RECONFIGURÁVEL – FPGA

O conceito de emulação de falhas do PLAESER só existe devido à possibilidade de reconfiguração inerente dos dispositivos baseados em tecnologia FPGA. A maneira em que a emulação de falhas é executada pela PLAESER restringe o universo dos circuitos reconfiguráveis para somente aqueles que permitem o acesso a memória de reconfiguração do FPGA, internamente, ou seja, sem a utilização de componentes externos.

Este trabalho foi desenvolvido utilizando os recursos disponíveis no Laboratório de Comunicações e Sistemas Embarcados da UFSC (Universidade Federal de Santa Catarina). Esse laboratório é composto

pelos grupos de pesquisa GPqCom (Grupo de Pesquisa em Comunicações) e GSE (Grupo de Sistemas Embarcados) que disponibilizaram os kits de desenvolvimento da Xilinx, ML402, ML403, ML507 e XUPV5. Esses kits têm como base os FPGAs de modelo Virtex4 e Virtex5 da Xilinx, que possuem um componente de acesso a memória de configuração do FPGA, denominado ICAP. Componente esse que permite a modificação do conteúdo da tabela verdade de uma LUT modificando assim a sua função lógica. Essa modificação é realizada internamente ao FPGA e durante o funcionamento do mesmo.

A flexibilidade e a capacidade lógica e de recursos encontrada nos FPGAs atuais possibilitam a análise experimental de técnicas de tolerância a falhas aplicadas a circuitos críticos pela plataforma PLAESER. A importância desses dois fatores para o PLAESER é explicada pela necessidade de comportar o SoCIF e ainda duas instâncias do DUT que podem ser alterados de acordo com o projeto.

## 5.2 FERRAMENTAS DE SÍNTESE E IMPLEMENTAÇÃO

Além de realizarem sua função original, que é transformar uma descrição de hardware em um circuito equivalente na forma de um arquivo de dados de configuração do FPGA, as ferramentas de desenvolvimento permitem também a aplicação de configurações necessárias para a correta injeção de falhas de forma que os resultados gerados pela PLEASER sejam válidos.

Nas próximas seções são descritas as ferramentas utilizadas pela PLEASER, na ordem em que aparecem no seu fluxo.

### 5.2.1 Ferramenta de Síntese Lógica

A ferramenta de síntese lógica pode visar o dispositivo alvo ou não. De forma geral a síntese lógica é encarregada de traduzir uma descrição de hardware, tipo HDL ou esquemático, para um nível de portas lógicas e posteriormente para elementos específicos da arquitetura alvo. No caso de FPGAs da Xilinx os elementos em questão podem ser LUTs, *Buffers*, lógica de vai-um (*carry*), entre outros componentes específicos da tecnologia. O resultado da síntese lógica é uma lista, chamada de *netlist*, com os elementos específicos da arquitetura contidos no circuito sintetizado.

A ferramenta da Xilinx responsável pela síntese lógica é chamada de XST (do inglês, *Xilinx Synthesis Technology*). Além de ter sido utilizada para síntese lógica do próprio SoCIF, a ferramenta tem sua

participação no fluxo da PLEASER no momento da geração da *netlist* do DUT e do SIF.

### 5.2.2 Ferramenta de Síntese Física

Seguindo o fluxo de projeto para FPGA, após a síntese lógica é realizada a síntese física que converte a descrição lógica do circuito em seu equivalente físico na forma de um arquivo para configuração do dispositivo alvo selecionado. O que inicialmente era uma lista de elementos genéricos da arquitetura de uma família de FPGAs agora é um arquivo de configuração que contém a especificação, localização, parâmetros e interface de componentes particulares do dispositivo escolhido e conectados através de matrizes de roteamento. Esse arquivo de configuração do FPGA é chamado de *bitstream*.

O processo de síntese física da Xilinx é denominado “implementação”. Diferentemente da síntese lógica, onde somente a XST é empregada, na síntese física são utilizados diversos programas. A ferramenta *PlanAhead* da Xilinx utiliza o conjunto de programas da síntese física, e programas adicionais, para o projeto e análise de circuitos para configuração de FPGAs. Além do fluxo tradicional de síntese física a ferramenta oferece a oportunidade de se trabalhar com uma metodologia hierárquica que suporta a preservação de projetos e fluxo baseado em equipe de desenvolvimento. O objetivo da preservação de projetos é permitir que o usuário implemente componentes críticos do seu circuito e preserve-os de forma que outros módulos não tão restritos possam ser desenvolvidos sem afetar o que foi preservado. O fluxo baseado em equipe utiliza o conceito de “preservação” para permitir que membros distintos da equipe possam implementar paralelamente seus módulos.

Visando permitir que diversos DUTs possam ser analisados sem a necessidade de implementar um SoCIF para cada projeto, a PLEASER emprega o conceito de “preservação” buscando a prototipação rápida do SIF para execução da campanha de falhas. Os blocos do circuito que são escolhidos para serem preservados são chamados de partições. No caso do PLEASER o SoCIF foi selecionado para ser uma partição uma vez que ele não carece de alteração para ser aplicado na análise de DUTs padrões.

A ferramenta *PlanAhead* pode ser empregada para a execução da síntese lógica de um projeto utilizando o XST. Porém, a utilização de partições em projetos em nível de RTL não é suportada para FPGAs anteriores à sexta geração, como Virtex 5/4/II, Spartan 3, entre outros.



Deste modo, a síntese lógica dos módulos do DUT e SIF é realizada separadamente.

A aplicação do PLEASER para a análise experimental de um circuito supostamente tolerante a falha somente é factível se as falhas puderem ser injetas exclusivamente na instância FAULTY do DUT. Para isso, restrições de área especificando a localização distinta de cada instância do DUT são indispensáveis. A criação dessas restrições é realizada utilizando a interface gráfica do *PlanAhead* que permite a definição de áreas para cada módulo do projeto. Para o fluxo do PLEASER, um modelo foi desenvolvido, contendo as restrições do SoCIF e as restrições relacionadas às áreas ocupadas pelas instâncias do DUT, com o objetivo de otimizar o fluxo.

### 5.2.3 Gerador da Configuração do FPGA

O último passo da implementação é a geração do arquivo de configuração do FPGA. O *bitstream* é a tradução da lista de elementos do circuito para padrão que será escrito na memória de configuração do FPGA. A Lista de elementos gerada na síntese lógica é atualizada com a localização, interface e parâmetros diversos dos componentes físicos do dispositivo FPGA selecionado, junto com a interconexão desses componentes, definidas pela configuração das matrizes de roteamento. O programa da Xilinx responsável por essa tradução é o BitGen.

O conteúdo do arquivo de configuração do FPGA não é legível, contudo, o programa BitGen permite a geração de um arquivo representado em ASCII (do inglês, *American Standard Code for Information Interchange*) idêntico ao *bitstream*. Esse arquivo contém os valores reais que configuram a tabela verdade das LUTs e, conseqüentemente a sua função lógica. Essa informação não é pertinente ao fluxo da PLEASER, uma vez que o valor da LUT é lido direto da configuração e é invertido a cada injeção de falha. Contudo, essa informação foi muito utilizada para a depuração do SoCIF e pode ser utilizada para análise dos resultados. Logo, o dado de configuração da LUT é escrito no relatório gerado ao final de cada campanha de falhas.

### 5.2.4 Tradutor do Descritor de Hardware

A lista de elementos resultante da síntese lógica é atualizada ao longo da implementação com informações que vão completando a configuração dos componentes que serão utilizados no FPGA escolhido. Cada programa do fluxo de síntese física da Xilinx atualiza a *netlist* e

escreve seu resultado em um arquivo que serve de entrada para o próximo programa do fluxo de projeto. Os arquivos contendo as atualizações da *netlist* utilizam um formato proprietário da Xilinx que não é legível. Contudo, um programa não documentado da Xilinx traduz o *netlist* para uma linguagem, também proprietária da Xilinx, porém, legível. Tanto o programa quanto a linguagem legível de descrição da Xilinx tem o nome de XDL (do inglês, *Xilinx Description Language*).

No final da implementação do SIF fica definida a localização de todos os componentes do projeto, inclusive das LUTs. A localização das LUTs é crucial para geração da lista de falhas pelo GLIFA. A localização das LUTs da instância FAULTY do DUT é recuperada do *netlist* pós-implementado através do programa XDL. Além da localização, informações como o nome dado para a instância da respectiva LUT, função booleana, entre outros, são utilizadas para complementar o relatório de resultados.

### 5.2.5 Compilador do Software

Sistemas embarcados construídos visando FPGAs da Xilinx contam com o suporte de um ambiente de desenvolvimento de software, chamado de Xilinx SDK. Esse ambiente integra as habilidades de edição, compilação, execução e depuração para aplicações desenvolvidas para os processadores ARM® *dual-core* Cortex™-A9 MPCore™, MicroBlaze™, e PowerPC™, e seus respectivos *drivers* e bibliotecas. A conectividade pré-configurada com os dispositivos alvo permite a depuração através do JTAG e também como console serial.

O software do processador do SoCIF foi desenvolvido utilizando o ambiente SDK da Xilinx que utiliza um compilador baseado no GCC (do inglês, *GNU Compiler Collection*). O executável do SoCIF está pronto para ser executado, mas pode ser facilmente compilado utilizando o ambiente SDK.

O ambiente SDK da Xilinx é empregado durante o fluxo do PLEASER para a configuração do FPGA com *bitstream* contendo o projeto do SIF. Após a configuração do FPGA, o SDK é utilizado para o carregamento da memória do SoCIF com o software de injeção de falhas.

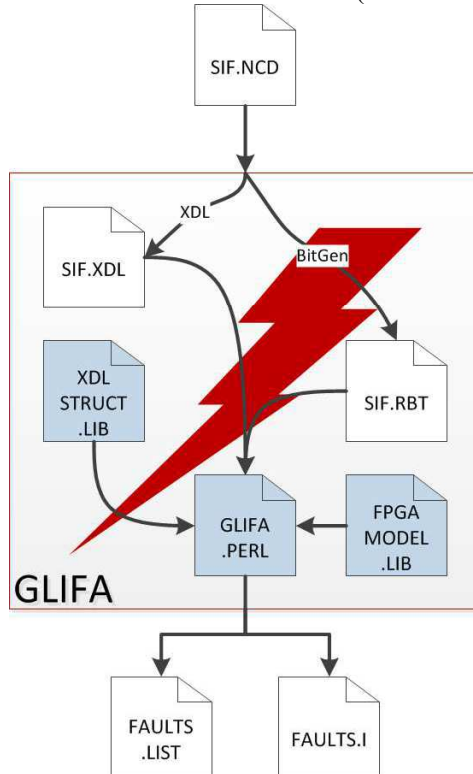
## 5.3 GLIFA – GERADOR DE LISTA DE FALHAS

Os blocos lógicos configuráveis (CLBs) são os recursos lógicos em que os FPGAs da Xilinx são baseados e que permitem a

implementação de circuitos combinacionais e sequenciais. A cada nova geração de dispositivos reconfiguráveis da Xilinx atualizações na arquitetura dos FPGAs são feitas, contudo, o conceito de CLB tem sido mantido e está presente na mais nova família da empresa, a série sete de FPGAs.

Os CLBs são divididos em *slices* que são compostos por elementos básicos, que são: LUTs; registradores; lógica de *carry*; e multiplexadores. Esses elementos permitem que um *slice* possa ser configurado como um circuito combinacional, aritmético e/ou de armazenamento. Cada modelo de FPGA possui uma quantidade diferente de CLBs que por sua vez contém um número de *slices* determinado pela família do modelo juntamente com a quantidade de elementos básicos de cada *slice*. O tamanho das LUTs, ou seja, o seu número de entradas varia de acordo com certas famílias de FPGAs da Xilinx. Consequentemente, a localização e o tamanho da tabela verdade das LUTs difere de acordo com o FPGA utilizado.

Figura 38 - Estrutura de funcionamento do GLIFA (Gerador de Lista de Falhas).



O Gerador de Lista de Falhas, proposto e denominado nesse trabalho de GLIFA, foi construído de forma que seja possível adicionar o suporte para novos modelos de FPGAs além do XC5VLX110T utilizado como plataforma de prototipação. O conceito de bibliotecas é utilizado para que as informações particulares de cada FPGA possam ser adicionadas futuramente. O GLIFA foi desenvolvido em Perl script em ambiente Unix e é executável através da camada de emulação do Cygwin em sistemas operacionais Microsoft Windows. A Figura 38 ilustra a funcionamento do GLIFA com seus respectivos arquivos de entrada e saída.

Figura 39 - Estrutura de informações dos slices da Virtex5 encontrada na biblioteca do GLIFA.

```

struct( SLICE_INFO =>
[
  inst_name   => '$', #2
  slice_type  => '$', #3
  clb_type    => '$', #4
  clb_x       => '$', #4
  clb_y       => '$', #4
  slice_x     => '$', #5
  slice_y     => '$', #5
  lut5        => '@', #10,19,28,40 (A,B,C,D)
  lut6        => '@', #11,20,29,41 (A,B,C,D)
  cy0         => '@', #12,21,30,43 (A,B,C,D)
  ff          => '@', #13,22,32,44 (A,B,C,D)
  fffinit     => '@', #14,23,33,45 (A,B,C,D)
  ffmux       => '@', #15,24,34,46 (A,B,C,D)
  ffsr        => '@', #16,25,35,47 (A,B,C,D)
  outmux      => '@', #17,26,37,48 (A,B,C,D)
  used        => '@', #18,27,39,49 (A,B,C,D)
  ce_used     => '$', #31
  clk_inv     => '$', #36
  out_used    => '$', #38
  bel_prop    => '$', #42
  precyinit   => '$', #50
  rev_used    => '$', #51
  sr_used     => '$', #52
  sync_attr   => '$', #53
];

```

O GLIFA tem como entrada a *netlist* do Sistema Injetor de Falhas completamente implementado, representado na Figura 38 pelo “SIF.NCD”. A partir do “SIF.NCD” é gerado o arquivo “SIF.XDL” contendo a tradução da *netlist* para o formato XDL e o arquivo “SIF.RBT” contendo a configuração do FPGA (*bitstream*) de forma legível. Conforme colocado anteriormente, em FPGAs de famílias distintas, os *slices* dos CLBs terão elementos diferentes e consequentemente isso afetará a composição do arquivo SIF.XDL. Visando a utilização do GLIFA em novos dispositivos, foi desenvolvida

uma biblioteca (“XDL\_STRUCT.LIB”) contendo a estrutura da informação dos CLBs utilizada pelo formato XDL. A Figura 39 apresenta a estrutura presente na biblioteca do GLIFA para os *slices* da Virtex5.

Nos comentários das linhas (precedidos pelo caractere #) se encontra a ordem em que os campos de informação do *slice* da Virtex5 aparecem no formato XDL. Observando os comentários dos campos “lut6” e “ff” na Figura 39 é possível destacar a coerência em relação à existência de quatro LUTs e quatro registradores nos *slices* da Virtex5.

As coordenadas XY determinam a posição dos *slices* dentro de um FPGA. O comprimento de X e de Y é determinado pelo “tamanho” dos FPGAs. A variedade de tamanhos, ou seja, a capacidade de lógica configurável dos modelos de FPGAs da Xilinx é bastante significativa. Essa variedade é que torna imprescindível um banco de dados contendo os parâmetros necessários para a conversão das coordenadas XY para o endereço dos quadros de configuração que guardam a tabela verdade da LUT. Para isso foi desenvolvida a biblioteca “FPGA\_MODEL.LIB” contendo as informações que mapeiam o FPGA e assim permitindo a geração da lista de falhas com os endereços das LUTs. A Figura 40 ilustra alguns dos parâmetros encontrados na biblioteca “FPGA\_MODEL.LIB”.

Figura 40 - Amostra dos parâmetros encontrados na biblioteca do GLIFA "FPGA\_MODEL.LIB".

```

struct( FrameAddr =>
[
  Unused => '$', # 8-Unused      [31-24]
  BlkType => '$', # 3-BlockType  [23-21]
  TopBott => '$', # 1-Top/Bottom [20]
  RowAddr => '$', # 5-Row Addr  [19-15]
  MajAddr => '$', # 8-Major Addr [14-7]
  MinAddr => '$', # 7-Minor Addr [6-0]
]);

#fpga information
$fpga_model = "vx5lxt110t";
$stam_x = 108;
$stam_y = 160;
@iob_cols = (0, 28, 58);
@clb_cols = (1 .. 4, 6 .. 15, 17 .. 18, 20 .. 27, 30 .. 41, 43 .. 52, 54 .. 57, 59 .. 62);
@brm_cols = (5, 16, 42, 53, 63);
@clk_cols = (29);
@dsp_cols = (19);
@mix_cols = (64);
$clb_stack = 36; #size of clb stack
$dsp_stack = 28; #size of dsp stack
$brm_stack = 30; #size of bram stack
$iob_stack = 54; #size of iob stack
$clk_stack = 4; #size of clk stack
$mix_stack = 32; #size of mix stack (PCI e, TEMAC, GTPs...)

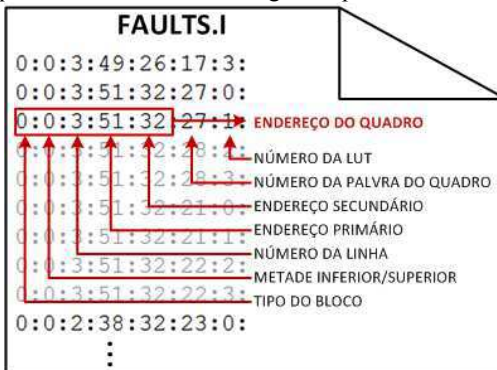
```

O carregamento do FPGA com os dados do *bitstream* é feito através de comandos que fazem com que o FPGA entre em modo de configuração. O endereço do primeiro quadro de configuração do FPGA é passado através desses comandos. Esse endereço é auto incrementado conforme os quadros forem sendo escritos na memória de configuração do FPGA. O arquivo de configuração do FPGA segue essa lógica e, dessa forma, é possível achar os valores da tabela verdade das LUTs no arquivo “SIF.RBT”. Conforme colocado anteriormente essa informação não é fundamental para a execução da emulação de falhas, porém, pode servir para uma futura otimização do PLEASER, além de ser fundamental para a depuração da plataforma e, portanto, o valor real das LUTs é gravado para a geração do relatório final.

O resultado do GLIFA é a geração de dois arquivos, o “FAULTS.LIST” e o “FAULTS.I”. O arquivo “FAULTS.I” contém somente as informações necessárias para o SoCIF poder executar a campanha de falhas. Essas informações são:

- número que determina a LUT alvo, selecionada a partir das LUTs existentes em um *slice*;
- endereço do quadro contendo o valor da LUT;
- índice da palavra do quadro de configuração onde se encontra o valor da LUT.

Figura 41 - Arquivo com a lista de falhas gerada pelo GLIFA.



A Virtex5 é composta de LUTs com seis entradas. Isso representa 64 bits de configuração para cada LUT. Esses 64 bits estão divididos em quatro quadros consecutivos. Os quadros de configuração da Virtex5 são compostos de 41 palavras de 32 bits. Cada bloco de 16 bits se encontra na mesma posição e na mesma palavra em cada um dos quatro quadros. Logo, cada linha do arquivo “FAULTS.I” contém o endereço do primeiro quadro e o número da palavra de 32 bits onde se encontram os

16 bits iniciais que configuram a LUT, mascarados utilizando-se o número da LUT. A Figura 41 destaca as informações, para cada falha, contidas em um exemplo de arquivo FAULTS.I gerado pelo GLIFA.

A informação do tipo do bloco não tem utilidade, uma vez que as falhas são injetadas somente em LUTs, ou seja, blocos do tipo CLB. Contudo, o tipo de bloco será importante quando a PLEASER suportar injeção de falhas no conteúdo dos blocos de memórias (BRAMs).

O segundo arquivo gerado pelo GLIFA, "FAULTS.LIST", contém informações mais detalhadas de cada falha. A Tabela 7 apresenta os campos contidos em cada linha do arquivo "FAULTS.I" que descrevem uma falha. As letras da coluna mais a esquerda da Tabela 7 indicam a ordem dos campos na linha do arquivo gerado pelo GLIFA.

Tabela 7 - Campos de cada linha do arquivo "FAULTS.LIST".

#	CAMPOS DE CADA LINHA	EXEMPLO
A	Identificador da falha	35
B	Coordenada X do <i>slice</i>	75
C	Coordenada Y do <i>slice</i>	39
D	Número da LUT do <i>slice</i>	0
E	Endereço do quadro de configuração em hexadecimal	0x0011161A
F	Índex da primeira palavra do quadro na ordem do <i>bitstream</i>	636361
G	Índex da palavra no quadro	39
H	Nome do sinal da saída da LUT	faulty_inst/.../i0.C2/L0[3].if_cy4_false.XOR_X_inst/XCY
I	Nome da instância da LUT	faulty_inst/counter_inst/counter/i0.C2/SUM[3]
J	Equação Lógica da LUT	O6=(A2@A4)
K	Máscara dos 16 bits	0x0000FFFF
L	As quatro palavras conteúdo	(0000CCCC,0000CCCC,00003333,00003333)

O GLIFA encontra as LUTs onde serão injetadas as falhas pelo nome da instância FAULTY do DUT. Na Tabela 7 é possível observar que o nome dado para a instância FAULTY do DUT é "*faulty\_inst*". O arquivo XDL preserva o nome completo de todas as instâncias do

projeto, logo, o nome da instância FAULTY do DUT estará presente no nome de todas as instâncias de seus componentes.

Ao final do fluxo da PLEASER é gerado um arquivo com o resultado da injeção de falhas, chamado de “FAULTS.O”. Esse arquivo é utilizado para completar com o resultado a “FAULTS.LIST” para geração do relatório da campanha de injeção de falhas.

#### 5.4 SOCIF – SYSTEM-ON-A-CHIP INJETOR DE FALHAS

A ferramenta XPS permite que desenvolvedores criem sistemas embarcados altamente customizáveis que podem ser implementados para FPGAs da Xilinx. Percorrendo o fluxo da ferramenta é possível montar sistemas mono ou multi-processados utilizando os processadores MicroBlaze™ ou PowerPC® baseados na arquitetura de barramentos PLB. Atualmente a Xilinx suporta também a terceira geração da arquitetura de barramento AMBA (do inglês, *Advanced Microcontroller Bus Architecture*), chamada AXI (do inglês, *Advanced eXtensible Interface*) para o processador MicroBlaze™. Isso tudo aliado a um extenso catálogo de periférico e IPs (do inglês, *Intellectual Property*).

A utilização da ferramenta XPS teve como objetivo a criação de um SoC capaz de carregar uma lista de falhas, de tamanho considerável, e aplicá-las utilizando a porta interna de acesso à configuração do FPGA, denominada ICAP. O SoC também deve ser capaz de reinicializar o DUT para cada injeção de falha e ser interrompido na medida em que houvesse alguma diferença entre as saídas das instâncias do DUT.

O SoC injetor de falhas construído, apelidado aqui de SoCIF, é baseado no processador totalmente sintetizável para os FPGAs da Xilinx, chamado MicroBlaze™. O MicroBlaze é um processador RISC de 32 bits com um conjunto de instruções otimizadas para aplicações embarcadas.

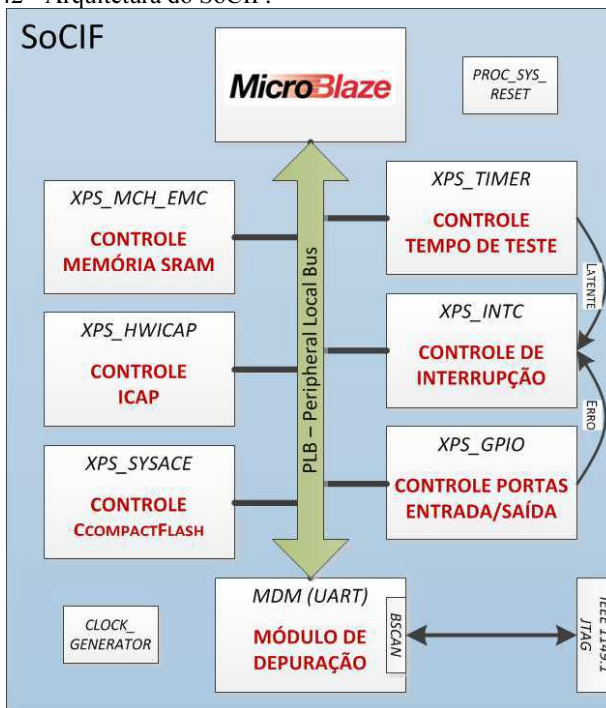
Para descrição do SoCIF foi feita sua divisão entre a parte hardware e software que são detalhadas nas seções que seguem.

##### 5.4.1 Arquitetura – Hardware

A Figura 42 apresenta os periféricos que compõem a arquitetura do SoC Injetor de Falhas (SoCIF). A função de cada um dos periféricos será apresentada nessa seção.



Figura 42 - Arquitetura do SoCIF.



Os periféricos conectam-se com a MicroBlaze através do barramento PLB que opera a uma frequência de 125MHz. Essa frequência de operação rege o sinal de relógio de todos os periféricos do barramento, incluindo o processador. O módulo *CLOCK\_GENERATOR* utiliza DCMs, PLLs ou MMCM (do inglês, *Mixed-Mode Clock Manager*) de acordo com suas disponibilidades em cada modelo de FPGA para a geração do sinal de relógio do SoCIF. O módulo *CLOCK\_GENERATOR* encontra-se na parte esquerda inferior da Figura 42.

#### 5.4.1.1 Controlador da SRAM

Os trabalhos de injeção de falhas baseados em FPGAs destacam o fato de que a comunicação entre o computador hospedeiro e a placa contendo o circuito reconfigurável é o gargalo de desempenho dos emuladores de falhas (33)(39)(40)(51). Considerando isso a PLAESER foi pensada de forma que todo o controle de injeção de falhas fosse realizado inteiramente pelo SoCIF internamente no FPGA, sem a necessidade da comunicação com o hospedeiro durante a execução da

emulação. Contudo, técnicas de tolerância a falhas empregadas em circuitos para aplicações críticas, como missões espaciais, geralmente implicam em uma maior utilização de recursos quando comparado com um circuito padrão. Técnicas como TMR (do inglês, *Triple Modular Redundancy*), *on-line checkers*, DWC (do inglês, *Duplication With Comparison*), codificações para detecção/correção de erros, entre outros, requerem adição de recursos lógicos para serem implementados.

O módulo de comunicação para computadores de bordo de satélites (OBC – *On-Board Computer*) desenvolvido em (52) utiliza os algoritmos de detecção/correção de erros Bose-Chaudhuri-Hocquenghem (BCH), Reed-Solomon (RS) e/ou Convolutacional de forma adaptativa. Os algoritmos de codificação de dados são aplicados de acordo com sua necessidade que é determinada pela qualidade do canal de comunicação. O módulo de comunicação foi implementado para o FPGA da Actel modelo A3PE1500. O elemento básico dos FPGAs da família ProASIC3E da Actel é chamado de VersaTile e possuem uma LUT com três entradas. A versão completa do módulo de comunicação implementada no A3PE1500 teve uma ocupação de 58% do FPGA, ou 22.133 VersaTiles utilizados. Um *slice* da família Virtex4 da Xilinx contém duas LUTs de quatro entradas e dois registradores enquanto um VersaTile pode implementar uma LUT de três entradas ou um registrador como comparado em (53). De qualquer forma nem toda a lógica implementada para um FPGA da Actel será sintetizada utilizando plenamente as LUTs dos FPGAs da Xilinx.

A ferramenta XPS permite que sejam utilizados blocos específicos de armazenamento de dados, chamados de BRAMs, para serem utilizados como memórias de dados/instruções do processador MicroBlaze. Cada BRAM da família Virtex5 da Xilinx, comporta a configuração de uma memória de 36 bits de até 1K de profundidade. A estrutura de dados utilizada pelo software do SoCIF para representar cada falha ocupa 96 bits de memória, logo cada BRAM pode armazenar um pouco mais de 300 falhas. Portanto, a utilização do fluxo da PLAESER para a análise experimental do módulo de comunicação adaptativo para OBCs, desenvolvido em (52) iria implicar em uma lista com mais de dez mil falhas a serem emuladas, ou seja, algo que seria inviável utilizando BRAMs.

O kit de desenvolvimento da Xilinx XUPV5-LX110T utilizado para o desenvolvimento desse trabalho contém uma memória SRAM (do inglês, *Static Random Access Memory*) com capacidade de até 256K palavras de 36 bits. O acesso síncrono da memória SRAM do tipo ZBT (do inglês, *Zero Bus Turnaround*) é realizado pelo controlador *Xilinx*

*Multi-channel External Memory Controller (XPS\_MCH\_EMC)*. Deste modo, essa memória foi escolhida para armazenar o software do SoCIF e ainda assim suportar um número significativo de falhas.

#### 5.4.1.2 Controlador do ICAP

O acesso a memória de configuração do FPGA para a emulação da falha pelo SoCIF é realizada através do componente ICAP. O controle desse componente é realizado pelo periférico XPS\_HWICAP que permite a modificação da estrutura e funcionalidade do circuito durante a execução do FPGA.

A frequência máxima de operação do ICAP varia de acordo com o modelo de FPGA. O modelo XC5VLX110T utilizado suporta até 100MHz para o sinal do relógio do ICAP. Para isso, o módulo *CLOCK\_GENERATOR* é configurado para gerar mais um sinal de relógio que será utilizado pelo XPS\_HWICAP para controlar o componente ICAP.

#### 5.4.1.3 Controlador do *CompactFlash*

Procurando diminuir a comunicação entre o FPGA e o hospedeiro optou-se pela utilização de um cartão de memória como forma de disponibilizar a lista de falhas a serem injetadas para o SoCIF. O contrário é verdadeiro, ou seja, o resultado da campanha de falhas gerado pelo SoCIF é passado para o hospedeiro também utilizando um cartão de memória.

O kit de desenvolvimento XUPV5-LX110T possui um controlador System ACE que permite o uso de cartões de memória com o formato *Compact Flash* (CF) para configuração do FPGA através da porta de JTAG. Esse controlador tem múltiplas interfaces, incluindo JTAG, *Compact Flash* e MPU (do inglês, *Microprocessor Unit*). Essa última interface é utilizada pelo periférico XPS\_SYSACE para permitir a leitura/escrita do cartão CF pela MicroBlaze.

#### 5.4.1.4 Controlador do Tempo de Teste

O controle do tempo de cada teste é realizado pelo periférico XPS\_TIMER. O periférico é um temporizador/contador de 32 bits parametrizável que permite vários modos de operação. O XPS\_TIMER é configurado como um contador regressivo que ao chegar a zero aciona o seu sinal de interrupção que está ligado no controlador de interrupção. O valor de início determina o intervalo de tempo entre o começo da contagem até o disparo da interrupção. Esse valor é definido no nível de software e é calculado conforme apresentado na Tabela 8.

Tabela 8 - Cálculo do intervalo de tempo de cada teste.

INTERVALO	Intervalo de tempo para o contador chegar à zero
VALOR_INICIO	Valor de início do contador
FREQ_PLB	Frequência de operação do barramento do SoC
$INTERVALO = (VALOR\_INICIO + 2) \times FREQ\_PLB$	

O cálculo apresentado na Tabela 8 é utilizado no software do SoCIF para determinar o tempo limite de cada teste, ou seja, o tempo aguardado para que a falha injetada se manifeste como erro gerando diferença entre as saídas das instâncias do DUT.

#### 5.4.1.5 Controlador das Portas do SoCIF

O periférico XPS\_GPIO permite ao processador o acesso a portas de propósito geral. O periférico é parametrizável e permite a configuração de até 32 portas que podem ser definidas dinamicamente como entradas ou saídas. Essas 32 portas também podem ser divididas em dois canais distintos. O controlador das portas do SoCIF possui 10 portas divididas igualmente em dois canais. O primeiro canal está configurado como somente entrada enquanto o segundo pode ter suas cinco portas utilizadas como entrada ou saída. O periférico também está conectado ao controle de interrupção.

O sinal que indica diferença dos DUTs deve ser o resultado da comparação das saídas das instâncias GOLDEN e FAULTY. Esse sinal está conectado no primeiro canal do periférico. Uma mudança do valor desse sinal resulta no acionamento do módulo de interrupção. O sinal de interrupção gerado pelo módulo XPS\_GPIO está destacado na Figura 42.

O canal de número dois do periférico XPS\_GPIO é utilizado para controle do sinal de reinicialização do DUT. Esse sinal é imprescindível para o funcionamento do PLAESER na medida em que permite colocar o DUT em um estado inicial, conhecido e livre de falhas. Dessa forma o DUT estará funcionando igualmente para cada injeção de falha, ou seja, a ordem que as falhas são injetadas não compromete o resultado da campanha. Antes da injeção de cada falha esse sinal é ativado para colocar as instâncias FAULTY e GOLDEN do DUT em um mesmo estado inicial.

As portas restantes do XPS\_GPIO são utilizadas para depuração do SoCIF.

#### 5.4.1.6 Controlador de Interrupção

O processador MicroBlaze possui uma única porta de recepção de pedidos de interrupções externas. Essa porta é conectada a saída do

módulo XPS\_INTC que concentra os sinais de interrupção gerados pelos outros periféricos. O módulo possui várias opções configuradas no nível de software que atribuem prioridades e habilitam/desabilitam até 32 sinais de interrupção.

Os módulos XPS\_GPIO e XPS\_TIMER são os únicos periféricos do SoCIP que estão conectados no controlador de interrupção.

#### 5.4.1.7 Controlador da Comunicação Serial

Nas versões mais atuais do XPS da Xilinx o módulo de depuração MDM (do inglês, *MicroBlaze™ Debug Module*) não é mais uma opção no fluxo da ferramenta, ou seja, o periférico é automaticamente adicionado no barramento do processador. O módulo MDM habilita a depuração de até oito processadores MicroBlaze através do uso do JTAG. A sua remoção tem que ser feita “manualmente”, contudo, a sua contribuição para o total de recursos utilizados é de apenas quatro LUTs, de acordo com o seu manual. Além disso, o módulo suporta o uso do JTAG para emulação de um módulo UART (do inglês, *Universal Asynchronous Receiver/Transmitter*).

Considerando o fato de que o número de falhas interfere diretamente no tempo de execução do SoCIP foi desenvolvida a opção de acompanhar o progresso de emulação durante a análise experimental do DUT. A porcentagem e o número de falhas injetadas que se tornaram erro ou permaneceram latentes é reportada para o computador hospedeiro por meio da UART do módulo de depuração. Através do JTAG a ferramenta SDK da Xilinx permite a emulação de um terminal serial onde é possível visualizar o andamento da campanha de injeção de falhas.

#### 5.4.2 Funcionalidade – Software

A Figura 43 apresenta o fluxo completo do software desenvolvido para a emulação do efeito de um SET para proporcionar a análise experimental de um circuito (DUT) no qual tenha sido aplicada alguma técnica de tolerância a falhas. O efeito de um SET é definido pela dupla variação do valor da lógica combinacional de um elemento do circuito durante um curto intervalo de tempo. Essa variação é realizada através da porta de acesso a configuração interna do FPGA. A porta ICAP por sua vez é controlada pelo módulo XPS\_HWICAP que conectado ao barramento da MicroBlaze permite que a emulação de um SET seja realizada no nível de software.

O kit de desenvolvimento de software (SDK) da Xilinx foi utilizado para a escrita do código fonte do SoCIF. O SDK é baseado na IDE (do inglês, *Integrated Development Environment*) padrão do Eclipse somado a *plug-ins* de desenvolvimento de software embarcado, construtores (em código aberto) e ferramentas específicas da Xilinx. Além disso, o ambiente integra o BSP (do inglês, *Board Support Package*) que é uma coleção de bibliotecas de código fonte configuráveis e *drivers* do extenso catálogo de periféricos presentes no XPS. Uma vez que a plataforma de hardware de um projeto é montada no XPS, sua descrição pode ser exportada para o ambiente SDK que identifica os módulos presentes no projeto e carrega o respectivo BSP.

O BSP do projeto do SoCIF contém os seguintes *drivers*:

- *bram\_v3\_00\_a*: tem a única função de permitir testes aplicados a memória do tipo BRAM;
- *emc\_v3\_01\_a*: tem a única função de permitir testes aplicados a memória do tipo SRAM;
- *gpio\_v3\_00\_a*: contém a função de inicialização, definição dos sentidos das portas, definição da função de interrupção desse módulo, entre outros;
- *hwicap\_v6\_00\_a*: contém a função de inicialização, leitura/escrita de frames, entre outros;
- *intc\_v2\_03\_a*: contém a função de inicialização, definição das funções destino de cada interrupção. (des)habilitação mestre das interrupções, entre outros;
- *standalone\_v3\_02\_a*: é a camada mais inferior de software da MicroBlaze e permite o acesso às funções básicas do processador na ausência de um sistema operacional;
- *sysace\_v2\_00\_a*: contém a função de inicialização, configuração do modo de operação, leitura/escritas de setores da *CompactFlash*, entre outros;
- *tmrctr\_v2\_04\_a*: contém a função de inicialização, configuração do modo de operação, início/pausa do contador, definição da função de interrupção desse módulo, entre outros;
- *uartlite\_v2\_00\_a*: contém a função que informa a ocupação das filas de envio e de recepção, leitura/escrita na fila de transmissão, entre outros;

- *xilfats\_v1\_00\_a*: contém funções de manuseio de arquivos em sistema de arquivos do tipo FAT (do inglês, *File Allocation Table*).

A última biblioteca listada foi adicionada no BSP do SoCIF para permitir o uso do sistema de arquivos FAT compatível com sistemas operacionais populares.

O *link script* é utilizado pelo SDK para controlar a alocação na memória das diferentes seções de um executável. Nesse arquivo encontra-se a informação de que tanto a seção de dados quanto a seção do código de um programa estão alocadas na SRAM. A seção de pilha *stack/heap* é utilizada para a passagem de parâmetros entre funções e para alocação dinâmica de memória. O tamanho dessa seção é ampliado considerando o fato que a lista de falhas é carregada em uma lista encadeada dinâmica. O novo tamanho da seção de pilha foi padronizado para suportar uma lista com mais de oito mil falhas. Obviamente, esse valor pode ser ampliado para o caso de um número maior de falhas.

O fluxo do software do SoCIF pode ser dividido em quatro estágios. Esses quatro estágios são detalhados nas seções que seguem.

#### 5.4.2.1 Inicializar Módulos

Os módulos do SoCIF são inicializados e configurados nesse estágio. As configurações relevantes estão destacadas no fluxo apresentado na Figura 43 e são listados a seguir:

- Inicialização do módulo de controle de interrupção;
- Configuração do XPS\_INTC para atender somente a interrupção corrente, sem empilhar interrupções;
- Inicialização do módulo XPS\_TIMER;
- Configuração do modo de operação do XPS\_TIMER para um contador regressivo com recuperação automática do valor inicial;
- Definição do valor inicial do contador. Padrão equivalente a 200ms;
- Definição da função chamada na ocorrência de interrupção do módulo XPS\_TIMER;
- Inicialização do módulo XPS\_GPIO;
- Definição da função chamada na ocorrência de interrupção do módulo XPS\_GPIO;
- Habilitação do canal de número um do módulo XPS\_GPIO para a geração de interrupções (sinal do comparador das instâncias do DUT);

- Configuração do sentido das portas do canal de número dois para saída (controle do sinal de reinicialização do DUT);
- Inicialização do módulo XPS\_HWICAP. Definição da arquitetura do FPGA;
- Autoteste do módulo XPS\_HWICAP, por garantia;

#### 5.4.2.2 Carregar Lista de falhas

A segunda etapa do software do SoCIF tem o objetivo de criar uma lista encadeada dinâmica de acordo com a quantidade de falhas presentes no arquivo “FAULTS.I”. Como visto anteriormente cada linha do arquivo gerado pelo GLIFA representa uma falha.

A utilização da *CompactFlash* é limitada a sua formatação. O módulo XPS\_SYSACE reconhece CFs com formatação FAT 12/16/32 contendo apenas uma seção de segurança. Esse tipo de formatação não é realizado pelas formas padrão dos sistemas operacionais populares e então, para tal foi utilizado o aplicativo *mkdosfs*. Um exemplo de parâmetros utilizados para formatação do CF é apresentada a seguir:

```
"mkdosfs.exe -R 1 -F 16 -n ML50X -s 64 J:"
```

#### 5.4.2.3 Injetar as Falhas

O estágio da injeção de falhas é constituído de um laço que é repetido para cada falha. As funções executadas nesse laço são:

- Inicialização do DUT;
- HABILITAÇÃO das interrupções dos módulos XPS\_GPIO, XPS\_TIMER;
- As três funções que seguem são executadas duas vezes com o objetivo de emular um SET;
  - Leitura dos quatro quadros contendo a configuração da LUT;
  - Inversão da tabela verdade da LUT;
  - Escrita dos quatro quadros de configuração da LUT;
- É dada a partida no contador;
- O software entra em um laço aguardando uma interrupção;
- Se a interrupção foi gerada pelo módulo XPS\_TIMER significa que o contador chegou a zero e durante esse tempo o efeito da falha injetada não se propagou para as saídas do DUT.



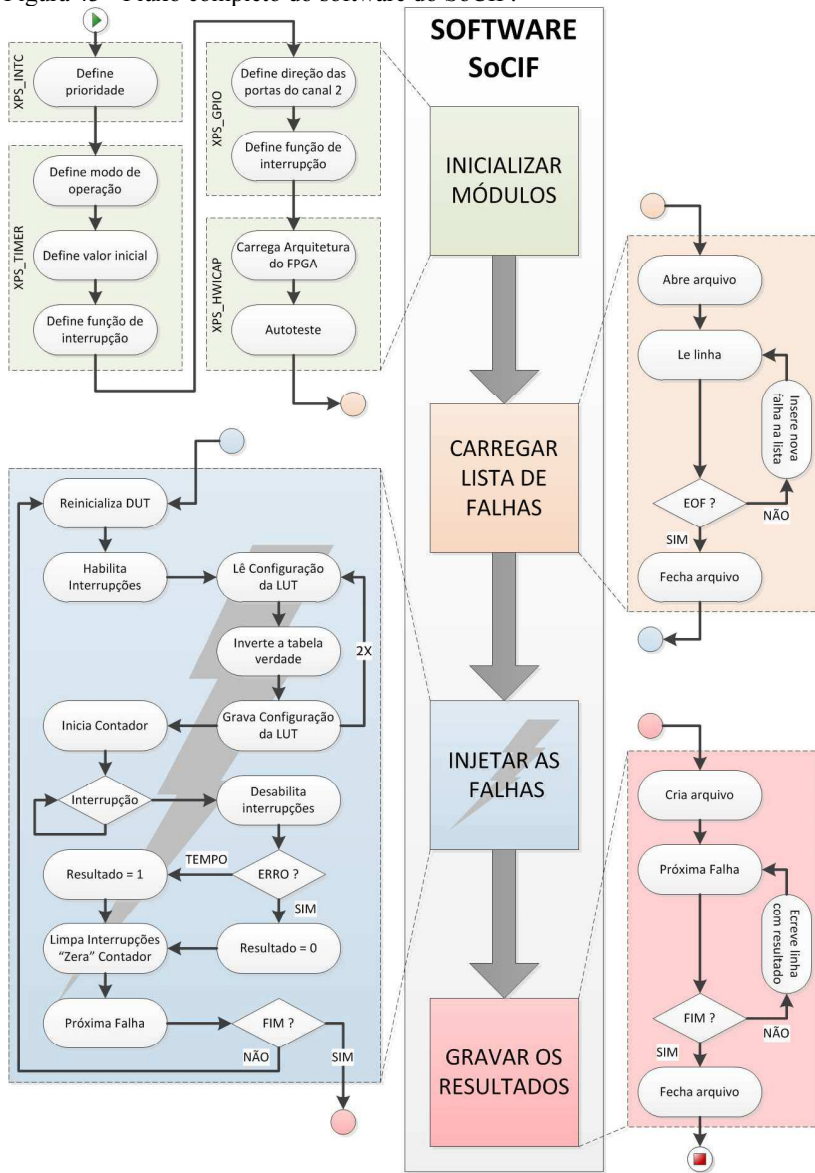
- Caso contrário, as saídas das instâncias do DUT divergiram em algum momento provocando a interrupção do módulo XPS\_GPIO;
- As interrupções são desabilitadas e reconhecidas. O valor inicial do contador é restabelecido;
- A próxima falha é carregada.

#### 5.4.2.4 Gravar os Resultados

O último estágio do software do SoCIF é responsável por gravar o resultado da injeção de falhas no cartão de memória. Para isso, o arquivo “FAULTS.O” é criado, e cada linha do arquivo representa o resultado de cada falha injetada.

O arquivo “FAULTS.O” é utilizado para completar com o arquivo “FAULT.LIST” com os resultados, visando a geração do relatório final. O arquivo “FAULTS.O” segue a mesma ordem das falhas dos arquivos gerados pelo GLIFA.

Figura 43 - Fluxo completo do software do SoCIF.

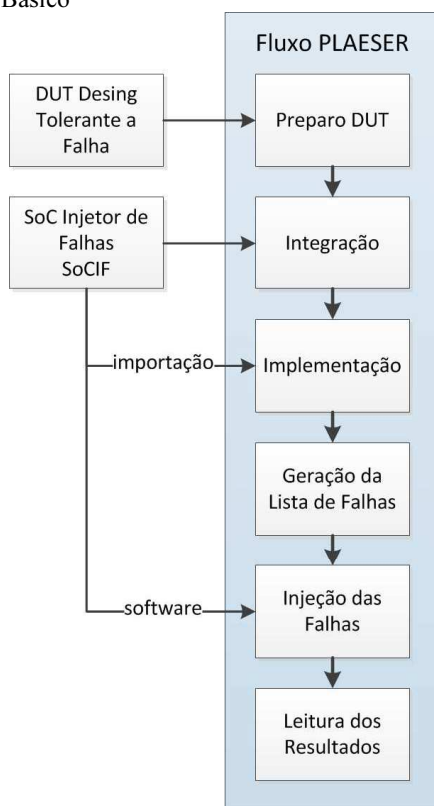


## 6 FLUXO DA PLATAFORMA DE EMULAÇÃO - PLAESER

Esta seção apresenta a principal contribuição do trabalho, introduzindo o fluxo proposto para a prototipação rápida de um sistema de emulação de SEs em FPGAs. O fluxo proposto é executado pela PLAESER e o kit de desenvolvimento da Xilinx XUPV5-LX110T foi utilizado para a prototipação. Inicialmente, é apresentado o diagrama básico do fluxo proposto. Posteriormente, o fluxo é desmembrado para elucidação de cada etapa. Finalmente, a aplicação desse fluxo será demonstrada na seção de resultados obtidos.

O fluxo proposto é composto por seis etapas fundamentais que resultam na injeção de falhas em cada gerador de função (LUT) do projeto tolerante a falhas, ou seja, o circuito sendo testado (DUT), e a geração de relatórios com os resultados obtidos. A Figura 44 apresenta o diagrama de fluxo da PLAESER.

Figura 44 - Fluxo Básico



## 6.1 PREPARO DO DUT

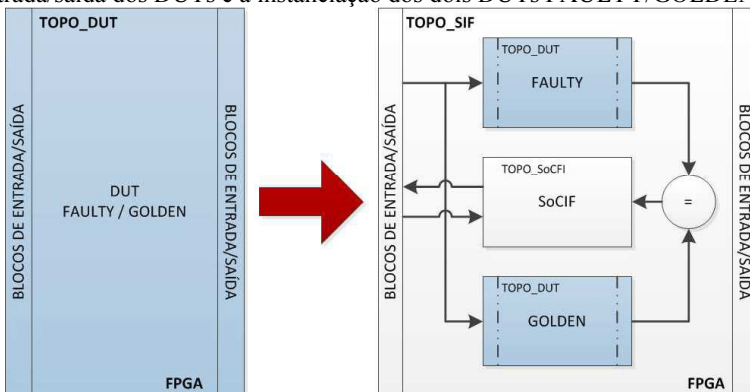
A primeira etapa do fluxo da PLAESER é o preparo do circuito tolerante a falhas (DUT) para que possa ser integrado no sistema de injeção de falhas que conterá o SoCIF e duas instâncias do DUT, sendo que uma servirá de referência para comparação com a outra onde as falhas serão injetadas.

Basicamente, dois cuidados devem ser tomados para que o DUT possa ser analisado pelo PLAESER, são eles:

- Remoção de todos os componentes específicos da interface do FPGA instanciados explicitamente no DUT.
- Configuração da ferramenta de síntese para que não sejam inferidos outros componentes de interface.

Nas “bordas” do FPGA (periferia) estão localizados componentes específicos para a interface entre os pinos e a configuração lógica interna. Esses componentes são chamados de blocos de entrada/saída (*I/O Blocks*) e são compostos basicamente de *buffers*. Quando um circuito é sintetizado as portas declaradas no módulo de mais alto nível hierárquico, também chamado de “topo do projeto”, são conectadas a esses *buffers* que fazem a interface com os pinos do FPGA. Ao tentar instanciar esse circuito sintetizado com os *buffers* de entrada/saída dentro de um novo projeto resultará em erros na hora da síntese desse novo projeto. Logo, os *buffers* de entrada/saída instanciados explicitamente dentro do DUT (TOPO\_DUT) devem ser movidos para o topo do projeto do sistema de injeção de falhas (TOPO\_SIF). A Figura 45 elucida a instanciação dos dois DUTs com seus respectivos blocos de entrada/saída movidos para o topo do SIF (TOPO\_SIF).

Figura 45 - O topo “Sistema de Injeção de Falhas” (SIF) com os blocos de entrada/saída dos DUTs e a instanciação dos dois DUTs FAULTY/GOLDEN.

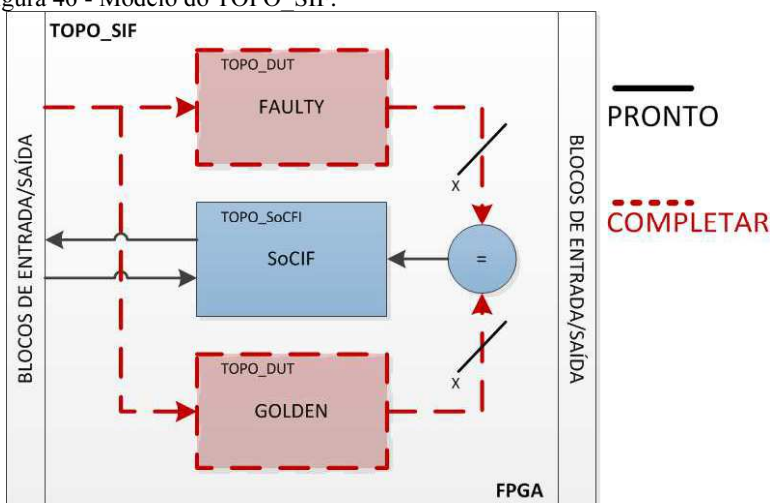


Além disso, a ferramenta de síntese infere automaticamente blocos de entrada/saída e para que isso não ocorra é necessário configurá-la. Na ferramenta de síntese lógica da Xilinx (XST) é necessário que a opção “*iobuf*” seja alterada para que o XST não insira automaticamente *buffers* de entradas/saída.

## 6.2 INTEGRAÇÃO

A etapa de integração é realizada utilizando um modelo do TOPO\_SIF para auxiliar na integração do SoCIF com o DUT. O modelo do TOPO\_SIF contém a declaração do componente e a instanciação do SoCIF com suas respectivas portas ligadas ao TOPO\_SIF. Também, contém a instanciação do módulo comparador já conectado ao SoCIF.

Figura 46 - Modelo do TOPO\_SIF.

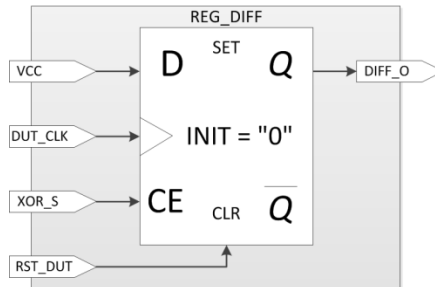


O modelo do TOPO\_SIF deve ser completado manualmente com a declaração do DUT e suas respectivas instanciações GOLDEN e FAULTY cujas saídas devem ser conectadas no módulo comparador. O módulo comparador é parametrizável para se adequar ao número de sinais a serem comparados. A Figura 46 ilustra o modelo e TOPO\_SIF destacando as partes que devem se complementadas.

O SoCIF deve ser capaz de colocar o DUT em um estado conhecido, livre de falhas, para que o resultado de cada injeção não seja influenciado por injeções anteriores, muito menos pela ordem da campanha de injeção. Para isso, o sinal de reset do DUT deve ser conectado ao SoCIF que reiniciará o DUT a cada falha a ser injetada.

O módulo comparador trabalha sincronizado com os sinais de saída das instâncias do DUT, ou seja, eles devem utilizar o mesmo sinal de relógio. Por isso, o módulo comparador é considerado assíncrono ao sinal de relógio do SoCIF pela ferramenta de implementação. Com isso, a ferramenta procurará alguma relação entre os dois domínios de relógio, e sendo impossível resultará em erros na análise de tempo, a menos que a ferramenta seja informada que o caminho entre o comparador e o SoCIF não precisa ser analisado. O sinal resultante do comparador, chamado “DIFF\_O” na Figura 47, muda para nível alto e assim se conserva quando ocorre uma diferença, representada pelo sinal “XOR\_S”, entre os sinais de saída das instâncias dos DUTs, e retorna ao seu estado inicial (“INIT = “0””) apenas quando o sinal “RST\_DUT” do DUT é acionado, sinal este que é controlado pelo SoCIF. Portanto, dado o conhecimento do comportamento do sinal “DIFF\_O” fica claro que esse caminho pode ser restringido para ser ignorado pela análise de tempo, permitindo assim a conclusão da implementação do projeto sem erros provenientes desse caso.

Figura 47 - Ilustração do registrador gerador do sinal resultante do módulo comparador.

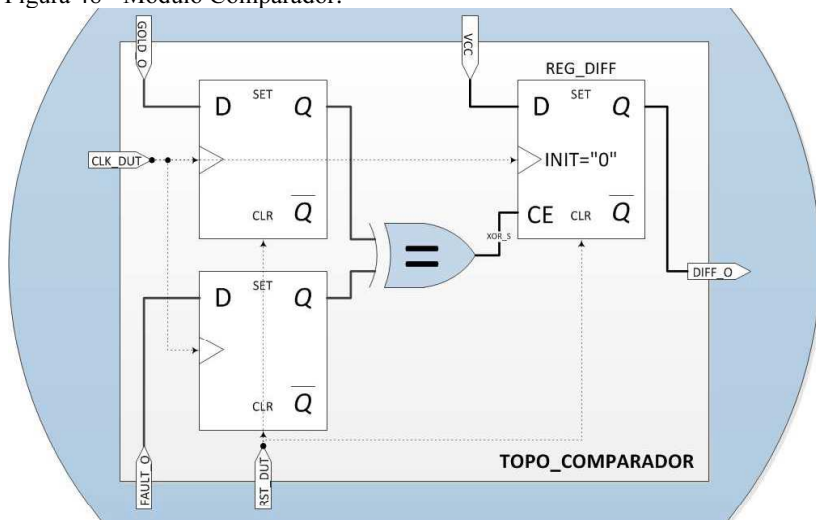


A mudança de domínio de relógio, no sentido do SoCIF para o DUT, faz com que a mesma situação comentada antes com sinal “DIFF\_O” aconteça com o sinal “RST\_DUT” que também deve ser ignorado na hora da análise de tempo.

Além desses cuidados com a mudança de domínio de relógio, ainda existe a questão da adição de atraso nas portas de saída dos DUTs devido à lógica de comparação. Por exemplo, um DUT com saídas não registradas e restrições de tempo apertadas, faz com que o atraso adicionado pela lógica combinacional do comparador, ainda que pequena, possa ser o suficiente para que os requisitos temporais consigam ser atendidos pela ferramenta. Nesse caso, esses caminhos não podem ser ignorados pela análise de tempo, pois, atrasos muito discrepantes entre os sinais de saída das instâncias do DUT podem

ocasionar uma diferença temporal no comparador, e não lógica, portanto comprometendo o resultado da injeção de falhas. Devido a isso, os sinais de entrada do módulo comparador são registrados, assim, separando a lógica de comparação do resto do circuito. A Figura 48 apresenta o módulo comparador para um bit.

Figura 48 - Módulo Comparador.



Para a etapa de implementação é preciso que todo o projeto esteja sintetizado, devido ao fato da ferramenta PlanAhead™ da Xilinx não permitir a utilização da metodologia de design hierárquico (do inglês, *Hierarchical Design Methodology*) no nível de RTL para FPGAs anteriores a sexta geração de FPGAs da Xilinx. A metodologia de design hierárquico permite a quebra de circuitos complexos em blocos isolados e mais simples utilizando o conceito de partições. Esse conceito permite preservar a implementação de um bloco, inclusive com roteamento, para a exportação dessa partição e reinserção em outros circuitos.

Dado que o DUT foi sintetizado na primeira etapa e o SoCIF é uma partição a ser importada, a síntese do TOPO\_SIF não precisa sintetizar novamente esses módulos. Para que isso seja possível é utilizado o atributo, chamado “BOX\_TYPE”, para que a ferramenta de síntese trate as duas instanciações, do DUT e a do SoCIF, como uma caixa preta. Esse atributo é especificado dentro do VHDL do TOPO\_SIF, como exemplifica a Figura 49.

Figura 49 - Aplicação do atributo no TOPO\_SIF.

```

architecture RTL of SIF is
...
    attribute BOX_TYPE : STRING;
    attribute BOX_TYPE of SoCIF : component is "user_black_box";
    attribute BOX_TYPE of DUT : component is "user_black_box";
...
begin

```

### 6.3 IMPLEMENTAÇÃO

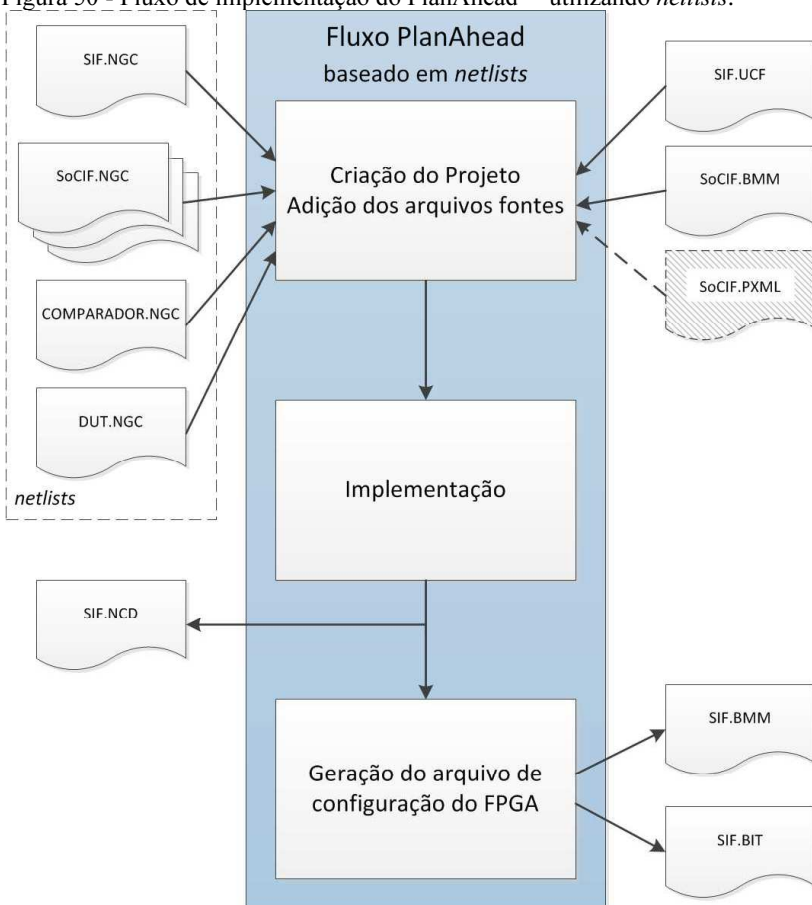
A etapa de implementação consiste em juntar todos os módulos sintetizados do sistema de injeção de falhas (SIF) e atualizar o arquivo de restrições (UCF) para então poder implementar o SIF. O resultado dessa etapa é a descrição final do projeto, com o roteamento e a localização de todos os recursos utilizados do FPGA, que servirá para geração da lista de falhas descrita na próxima etapa e também para a formação do arquivo de configuração do FPGA que será fundamental na etapa de injeção de falhas.

A implementação SIF é realizada pela ferramenta PlanAhead™ da Xilinx. Para isso, é criado um projeto no PlanAhead incluindo todos os arquivos que contém a descrição dos circuitos sintetizados que compõem o SIF. Esses arquivos são chamados de *netlists*, e possuem uma extensão “NGC” quando sintetizado pela ferramenta XST ou “EDIF” (do inglês, *Electronic Design Interchange Format*) que é um formato padrão de descrição de circuito também aceito pelo PlanAhead. A Figura 50 mostra o fluxo de implementação do PlanAhead tendo como entrada os *netlists* do SoCIF, do comparador, do DUT e do topo do projeto, o “SIF.NGC”, gerados na etapa anterior.

Além dos *netlists* é passado para o PlanAhead o arquivo “SoCIF.BMM”, de extensão “BMM” (do inglês, *Block RAM Memory Map*), que contém a descrição sintática de como BRAMs individuais constituem um mapa de memória contíguo. Como o projeto ainda não foi implementado não são conhecidas quais BRAMs foram utilizadas para montar o mapa de memória. Ao final da implementação, junto com a geração do arquivo de configuração do FPGA, nomeado de “SIF.BIT”, o arquivo “SoCIF.BMM” é atualizado com as coordenadas das BRAMs do FPGA utilizadas para montar o mapa de memória do SoCIF.



Figura 50 - Fluxo de implementação do PlanAhead™ utilizando *netlists*.



Finalmente, o arquivo “SIF.UCF” contendo as restrições definidas pelo usuário é adicionado ao projeto. As restrições pertinentes ao módulo do SoCIF já estão no arquivo “SIF.UCF”, que são:

- Pinos das portas utilizadas pelo SoCIF, com suas respectivas tensões de referência;
- Frequência de entrada do sinal de relógio da placa utilizado pelo SoCIF e possivelmente pelo DUT também;
- Restrições para ignorar a análise de tempo dos sinais conectados ao DUT ou ao comparador, como explicado na etapa dois;

- Restrição de área que determina a localização SoCIF no FPGA.

O arquivo “SIF.UCF” deve ser atualizado com as restrições pertencentes ao DUT e também é preciso que sejam criadas restrições de áreas para as instâncias do DUT. As instâncias *golden* e *faulty* do DUT devem ser implementadas em áreas distintas do FPGA para que não haja compartilhamento de recursos e também para que as falhas sejam injetadas somente na instância *faulty* para que a comparação possa ser feita. Restrições de área padrão para as instâncias do DUT já estão declaradas no “SIF.UCF”, mas podem ser facilmente alteradas graficamente no PlanAhead™ ou diretamente no “SIF.UCF” modificando a restrição de área (“AREA\_GROUP”). A Figura 51 exemplifica a descrição da restrição de área do SoCIF e das instâncias do DUT utilizando a devida sintaxe para o arquivo de restrições (UCF).

Figura 51 - Exemplo de restrição de área utilizando a devida sintaxe do UCF.

```
INST "socif_inst" AREA_GROUP = "pblock_socif_inst";
AREA_GROUP "pblock_socif_inst" RANGE=SLICE_X28Y60:SLICE_X81Y99;
AREA_GROUP "pblock_socif_inst" RANGE=DSP48_X0Y24:DSP48_X0Y39;
AREA_GROUP "pblock_socif_inst" RANGE=RAMB36_X2Y12:RAMB36_X2Y19;
INST "golden_inst" AREA_GROUP = "pblock_golden_inst";
AREA_GROUP "pblock_golden_inst" RANGE=SLICE_X0Y100:SLICE_X105Y159;
AREA_GROUP "pblock_golden_inst" RANGE=DSP48_X0Y40:DSP48_X0Y63;
AREA_GROUP "pblock_golden_inst" RANGE=RAMB36_X0Y20:RAMB36_X3Y31;
INST "faulty_inst" AREA_GROUP = "pblock_faulty_inst";
AREA_GROUP "pblock_faulty_inst" RANGE=SLICE_X0Y0:SLICE_X105Y59;
AREA_GROUP "pblock_faulty_inst" RANGE=DSP48_X0Y0:DSP48_X0Y23;
AREA_GROUP "pblock_faulty_inst" RANGE=RAMB36_X0Y0:RAMB36_X3Y11;
```

Caso não haja necessidade de alteração de qualquer restrição relacionada à localização do módulo SoCIF, então é possível utilizar o arquivo SoCIF.PXML que contém a descrição da partição para possibilitar a importação da implementação pronta do SoCIF. A utilização de partição para reutilização do SoCIF já implementado justifica a expressão “prototipação rápida” aplicado ao fluxo do PLAESER.

#### 6.4 GERAÇÃO DA LISTA DE FALHAS

O resultado da implementação do SIF é o arquivo “SIF.NCD” que corresponde a um banco de informações relacionadas à síntese física do projeto. Ao contrário da síntese lógica resultante da ferramenta XST, o arquivo NCD contém os recursos reais do FPGA, como IOBs, CLBs, DCMs, PLLs, por exemplo, que serão utilizados, suas localizações, parâmetros, interface, configuração das matrizes de roteamento, entre outros. Logo, contém toda a informação necessária para a geração da

lista de falhas a serem injetadas pelo SoCIF. Contudo, esse arquivo não é legível e utiliza um padrão proprietário da Xilinx.

A recuperação das informações relevantes das falhas a serem injetadas é feita convertendo o formato binário de descrição do circuito (NCD) para uma descrição legível. Para isso foi utilizado o aplicativo XDL da Xilinx para converter o formato NCD para uma descrição legível com a linguagem XDL. A Figura 52 mostra um exemplo da descrição do circuito após a implementação utilizando a linguagem XDL.

Figura 52 - Exemplo do arquivo de descrição do circuito implementado utilizando a linguagem XDL.

```

inst "golden_inst/counter_inst/counter/C0/QINV[3]" "SLICEL",placed CLBIM_X44Y119 SLICE_X76Y119 ,
cfg " A$LUT:::#OFF A$LUT:golden_inst/counter_inst/counter/C0/L0[3].if_tmr_true_VQ_inst:$LUT:06=(-A4*(A1+A2))+(A4*(A1+A2))
ACYO:::#OFF AFF:::#OFF AFFINIT:::#OFF AFPMUX:::#OFF AFFSR:::#OFF AOUTMUX:::#OFF
AUSED:::0 B$LUT:::#OFF B$LUT:::#OFF BCYO:::#OFF BFF:::#OFF BFFINIT:::#OFF
BFPMUX:::#OFF BFPSR:::#OFF BOUTMUX:::#OFF BUSED:::#OFF C$LUT:::#OFF C$LUT:::#OFF
CCYO:::#OFF CUSED:::#OFF CFF:::#OFF CFFINIT:::#OFF CFPMUX:::#OFF CFPSR:::#OFF
CLKINV:::#OFF COUTMUX:::#OFF COUTUSED:::#OFF CUSED:::#OFF D$LUT:::#OFF
D$LUT:::#OFF DCYO:::#OFF DFF:::#OFF DFFINIT:::#OFF DFFMUX:::#OFF DFFSR:::#OFF
DOUTMUX:::#OFF DUSED:::#OFF PRECINIT:::#OFF REVUSED:::#OFF SRUSED:::#OFF
SYNC_ATTR:::#OFF "
;

inst "faulty_inst/counter_inst/counter/I0.C2/SUM[3]" "SLICEL",placed CLBLL_X43Y39 SLICE_X76Y39 ,
cfg " A$LUT:::#OFF A$LUT:faulty_inst/counter_inst/counter/I0.C2/L0[3].if_cy4_false_XOR_X_inst/KCY:$LUT:06=(A2&A4)
ACYO:::#OFF AFF:::#OFF AFFINIT:::#OFF AFPMUX:::#OFF AFFSR:::#OFF AOUTMUX:::#OFF
AUSED:::0 B$LUT:::#OFF B$LUT:::#OFF BCYO:::#OFF BFF:::#OFF BFFINIT:::#OFF
BFPMUX:::#OFF BFPSR:::#OFF BOUTMUX:::#OFF BUSED:::#OFF C$LUT:::#OFF C$LUT:::#OFF
CCYO:::#OFF CUSED:::#OFF CFF:::#OFF CFFINIT:::#OFF CFPMUX:::#OFF CFPSR:::#OFF
CLKINV:::#OFF COUTMUX:::#OFF COUTUSED:::#OFF CUSED:::#OFF D$LUT:::#OFF
D$LUT:::#OFF DCYO:::#OFF DFF:::#OFF DFFINIT:::#OFF DFFMUX:::#OFF DFFSR:::#OFF
DOUTMUX:::#OFF DUSED:::#OFF PRECINIT:::#OFF REVUSED:::#OFF SRUSED:::#OFF
SYNC_ATTR:::#OFF _ROUTETHROUGH:A:AMUX "
;

```

O exemplo apresentado na Figura 52 contém as informações utilizadas para a geração da lista de falhas, que são:

- O *slice* utilizado que é dado pela sua coordenada dentro do FPGA;
- A LUT do *slice* que está sendo ocupada com alguma lógica, ou seja, o seu atributo é diferente de “#OFF”; e
- Se a LUT está sendo empregada como uma função lógica e não como um SRL ou como uma memória distribuída, chamada de LUT RAM.

Outras informações são utilizadas para formar o relatório de resultados, essas informações são:

- O nome da instância que foi dada para o respectivo *slice*;
- O nome dado para o componente utilizado do respectivo *slice*;
- A equação booleana atribuída a cada LUT.

A partir dessas informações duas listas são geradas. A primeira, chamada de “FAULTS.I”, contém o mínimo de informações necessárias para injeção de falhas. Uma segunda lista é gerada contendo as mesmas informações da primeira, só que mais detalhadas e com mais dados,

denominada “FAULTS.LIST”, para posterior geração de relatório com os resultados.

A geração das listas é feita por *scripts* escritos em Perl que convertem o arquivo “SIF.NCD” para o formato XDL utilizando a ferramenta também chamada XDL. Para cada LUT dentro da instância “FAULTY” do DUT é atribuída uma falha. A configuração do FPGA é dividida em quadros de 41 palavras de 32 bits e não em coordenadas de *slice*. Além disso, a função lógica de cada LUT é escrita em 64 bits que são divididos em 16 bits igualmente localizados dentro de quatro quadros de configuração sequenciais. Logo, o script converte as coordenadas XY dos *slices* para o respectivo endereço do quadro de configuração e calcula o índice da palavra que contém os 16 bits da LUT. Dado que os 64 bits de cada LUT estão em quatro quadros em sequência, então o arquivo “FAULTS.I” contém somente o endereço do primeiro quadro, o índice da palavra de 32 bits e o número respectivo a LUT do *slice* onde será injetada a falha. O número da LUT é utilizado para poder mascarar os 16 bits iniciais do conteúdo da LUT.

O arquivo “FAULTS.I” é então copiado para o cartão de memória que será lido pelo SoCIF para injetar as falhas. Esse mesmo cartão de memória será utilizado pelo SoCIF para gravar o resultado das injeções de falha e finalmente a geração dos relatórios com os resultados na última etapa.

## 6.5 INJEÇÃO DAS FALHAS

Antes de configurar o FPGA com o arquivo “SIF.BIT”, o mesmo contém as BRAMs do SoCIF atualizado com um programa de inicialização, cuja função é permanecer aguardando até que algum software seja carregado para então poder executá-lo. Isso é feito automaticamente pelo ambiente de desenvolvimento de software embarcado da Xilinx (SDK) que utiliza a ferramenta Data2MEM, também da Xilinx, para transferir dados de um programa para a forma de BRAMs contínuas. Além disso, o ambiente SDK é responsável por compilar o software que é executado pelo SoCIF junto com os *drivers* e as bibliotecas necessárias.

O software é escrito na memória do SoCIF através da ferramenta de depuração da Xilinx, XMD (do inglês, *Xilinx Microprocessor Debugger*), que utiliza JTAG para se comunicar com o processador do SoCIF. Os comandos necessários para o carregamento do software na memória do SoCIF são invisíveis para o usuário, uma vez que o ambiente SDK automatiza esse processo.

O software de injeção de falhas que é executado pelo SoCIF é chamado SoCIF.ELF. Esse arquivo é um executável com o formato “ELF” (do inglês, *Executable and Linkable Format*) compatível com o processador do SoCIF. Devido à ampla quantidade de recursos que um DUT pode utilizar, correspondendo a uma lista significativa de falhas, tanto a memória de instruções quanto a memória de dados do processador do SoCIF ficam em uma memória externa do FPGA para que o uso de BRAMs para esse fim não inviabilize a sua prototipação.

A ocupação da memória, o tamanho dos arquivos de entrada e saída e o tempo de execução do software de injeção de falha estão diretamente ligados ao número de falhas injetadas e, conseqüentemente, à quantidade de recursos do DUT. Como a execução pode ser demorada, a evolução da campanha de injeção de falhas pode ser acompanhada pelo terminal do SDK. Os dados são transmitidos para o terminal através do JTAG que emula uma comunicação serial com o SoCIF.

Concluída a execução da injeção das falhas o resultado é escrito no cartão de memória, no arquivo “FAULTS.O”. A informação contida no arquivo “FAULTS.O” indica se a falha injetada causou alguma diferença na saída, ou se após um intervalo de tempo (determinado no software do SoCIF, padrão igual a um segundo) nenhuma diferença entre o GOLDEN e o FAULTY DUT foi evidenciada.

## 6.6 LEITURA DOS RESULTADOS

A última etapa do fluxo PLEASER tem como objetivo juntar toda a informação pertinente à campanha de injeção de falhas e gerar um relatório final, com os resultados obtidos, de forma que seja de fácil entendimento.

Figura 53 - Exemplo do relatório de resultados gerado ao final do fluxo do PLAESE.

ID	X	Y	LUT	FRAME_ADDR	WORD_IDX	OFFS	SIGNAL_NAME	INSTANCE_NAME	LUT_EQUATION	LUT_MASK	ORD_VALU	ERR	TIME	
0	76	58	0	0x001096A0	542881	37/L0[0].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR2[3]	O6=(A6**A6) *(( 0x0000FFFF	(00330017,	0	1				
1	76	58	1	0x001096A0	542881	37/L0[1].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR2[3]	O6=(A6**A6) *(( 0xFFFF0000	(00330017,	0	1				
2	76	58	2	0x001096A0	542881	38/L0[2].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR2[3]	O6=(A6**A6) *(( 0x0000FFFF	(033F0003,	0	1				
3	76	58	3	0x001096A0	542881	38/L0[2].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR2[3]	O6=(("A5*(A6*A 0xFFFF0000	(033F0003,	0	1				
4	77	58	0	0x0010969A	542635	37/L0[0].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR1[3]	O6=(A6**A6) *(( 0x0000FFFF	(00170055,	0	1				
5	77	58	1	0x0010969A	542635	37/L0[1].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR1[3]	O6=(A6**A6) *(( 0xFFFF0000	(00170055,	0	1				
6	77	58	2	0x0010969A	542635	38/L0[2].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR1[3]	O6=(A6**A6) *(( 0x0000FFFF	(1717003F,	0	1				
7	77	58	3	0x0010969A	542635	38/L0[3].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR1[3]	O6=(("A5*(A4*A 0xFFFF0000	(1717003F,	0	1				
8	74	58	0	0x00109620	541405	37/L0[0].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR0[3]	O6=(A6**A6) *(( 0x0000FFFF	(003F0055,	0	1				
9	74	58	1	0x00109620	541405	37/L0[1].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR0[3]	O6=(A6**A6) *(( 0xFFFF0000	(003F0055,	0	1				
10	74	58	2	0x00109620	541405	38/L0[2].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR0[3]	O6=(A6**A6) *(( 0x0000FFFF	(77770017,	0	1				
11	74	58	3	0x00109620	541405	38/L0[3].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR0[3]	O6=(("A4*(A3*A 0xFFFF0000	(77770017,	0	1				
												TOTAL	0	12

O cartão de memória contendo o arquivo “FAULTS.O” é utilizado para completar as informações do arquivo “FAULTS.LIST” com os resultados de cada injeção de falhas. Com todas as informações

reunidas é criado o arquivo “RESULT.CSV” relatando as informações da campanha de injeções. A Figura 53 mostra um exemplo de relatório gerado ao final do fluxo PLEASER.

## 6.7 LIMITAÇÕES

Algumas restrições devem ser atendidas pelo DUT para que o fluxo de injeção de falhas do PLEASER possa ser aplicado corretamente. As restrições são:

- Circuito de *reset* em todo o DUT.
- Elementos de memórias especiais, implementados em LUTs, como SRLs e RAMs, não podem ser utilizados pelo DUT ou devem ser restringidos para colunas em que não haja outras LUTs com funcionamento padrão.

A primeira restrição é originada do fato de que antes de toda injeção de falha o DUT deve partir de um estado conhecido para que as falhas sejam aplicadas de forma igual em todo o circuito. Para que isso possa acontecer é preciso que todo registrador e elemento de memória tenham um estado de *reset* definido e que possa ser controlado pelo SoCIF.

As LUTs da Xilinx além de serem geradores de funções lógicas podem ser implementadas como elementos especiais, SRLs ou LUT RAMs. Tanto SRL ou LUT RAM que estiverem na mesma coluna que ocorrer uma leitura/escrita da memória de configuração podem perder o seu estado no momento da operação.

## 7 RESULTADOS OBTIDOS

O uso do fluxo da plataforma PLAESER foi validado utilizando-se um exemplo simples antes de ser empregado para a análise experimental de uma aplicação real. Buscando não comprometer os resultados foi escolhido um circuito desenvolvido por terceiros e que possuísse uma técnica de tolerância a falhas. Após a utilização desse exemplo para verificação e depuração da PLEASER foi então aplicado seu fluxo no objetivo maior do projeto, o computador de bordo desenvolvido em (54).

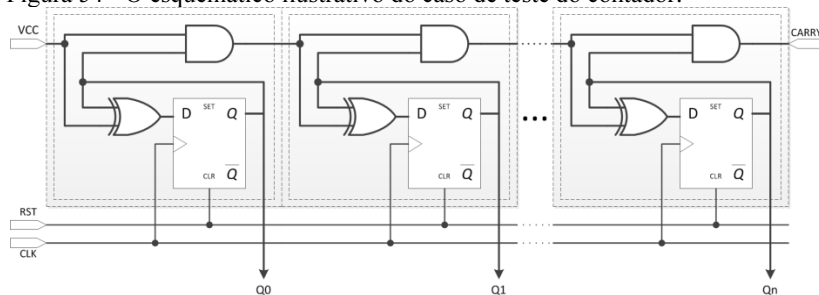
O exemplo escolhido foi desenvolvido para exemplificar a metodologia recomendada pela Xilinx para construção de circuitos que implementam a técnica de redundância tripla. A metodologia apresentada em (55) requer uma cópia tripla completa do circuito, uma vez que todos os caminhos, e não somente os registradores são suscetíveis aos *soft errors*. Dessa forma, votadores majoritários são utilizados recebendo como entrada as saídas dos três circuitos (triplicação), e fornecendo como resultado a entrada mais votada (maior incidência).

A seguir são apresentados os casos de teste analisados com o PLAESER e respectivos resultados.

### 7.1 EXEMPLO – CONTADOR

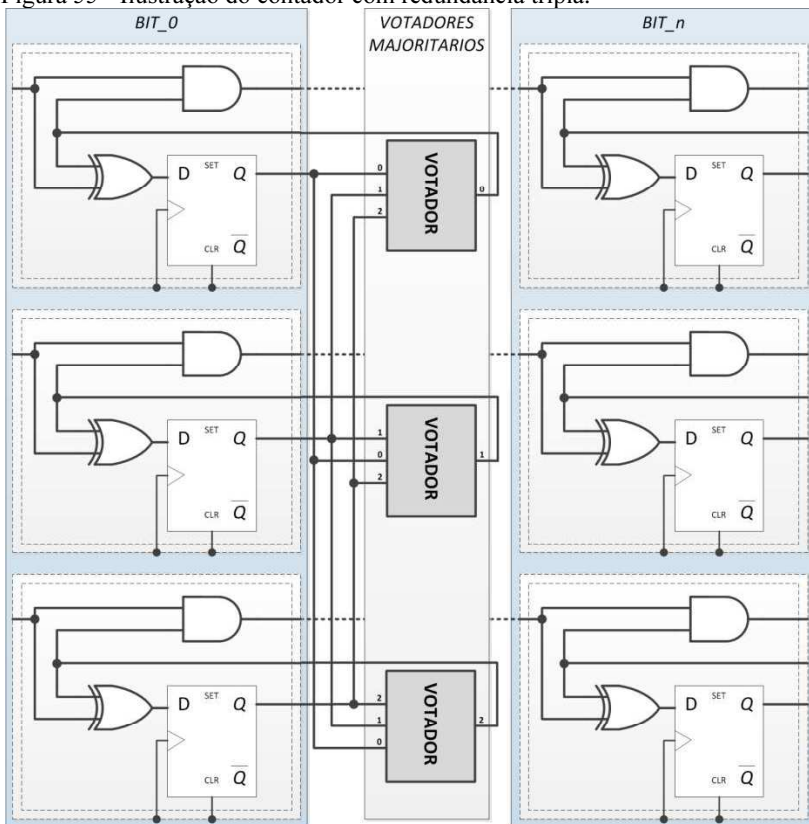
A Figura 54 mostra o esquemático do contador binário síncrono com *ripple-carry* utilizado como caso de teste. Esta figura destaca também a quantidade de recursos utilizados pra cada bit do contador.

Figura 54 - O esquemático ilustrativo do caso de teste do contador.



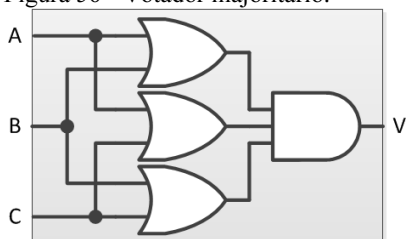
A metodologia recomendada pela Xilinx aplicada ao contador faz com que todo o seu circuito seja triplicado e junto a isso são adicionados os votadores majoritários. A Figura 55 ilustra o método de redundância tripla aplicado ao contador.

Figura 55 - Ilustração do contador com redundância tripla.



A Figura 56, abaixo, apresenta o circuito do votador majoritário utilizado em (55).

Figura 56 - Votador majoritário.

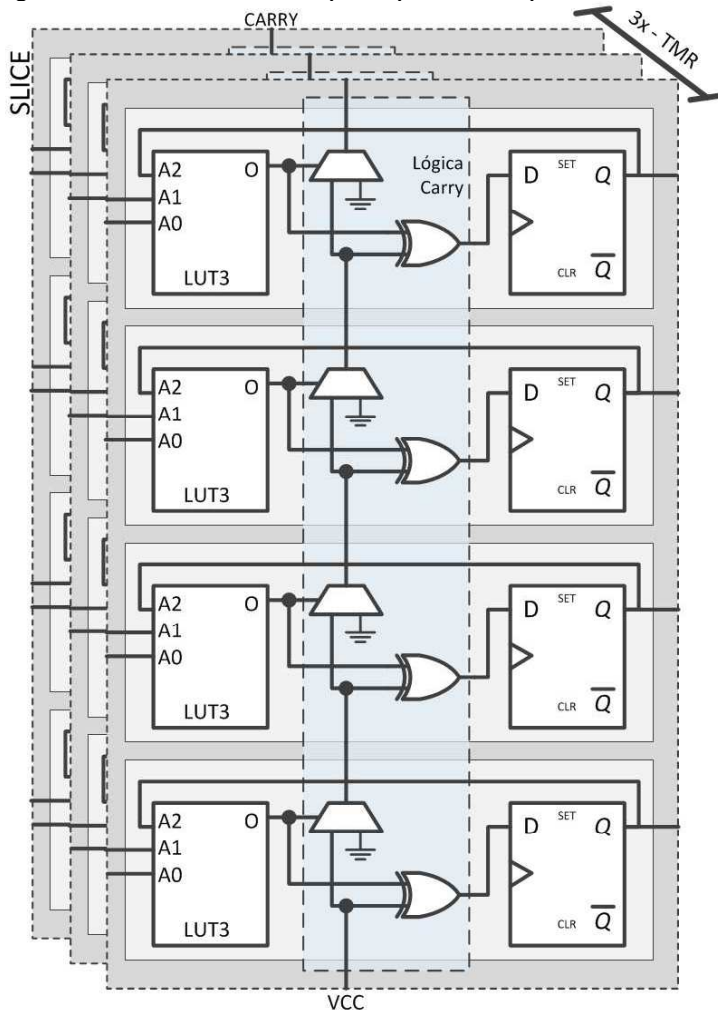


A descrição de hardware dos contadores é disponibilizada pela Xilinx na linguagem VHDL. O número de bits dos contadores é parametrizável que, para esse caso de teste, foi definido um contador de quatro bits. Observando as duas últimas figuras é possível determinar o



número de portas lógicas utilizadas para cada bit do contador. Contudo, a lógica da soma foi implementada utilizando-se primitivas da biblioteca de componentes da Xilinx. As primitivas são multiplexadores e portas XOR presentes na lógica de carry dos CLBs. Dessa forma, somente três LUTs, uma para cada votador, são utilizadas para implementar um bit do contador. Porém, o acesso ao controle dos multiplexadores da lógica de carry do *slice* do CLB é feito através de uma LUT, que mesmo sem lógica a LUT é ocupada.

Figura 57 - Contador TMR exemplo disponibilizado pela Xilinx.



A Figura 57 ilustra o contador com redundância tripla, exemplo disponibilizado pela Xilinx e implementado utilizando os recursos aritméticos dos *slices* do FPGA. A LUT implementa a lógica do votador enquanto a porta lógica XOR e o multiplexador (que faz a função da porta AND da Figura 55) desempenham a função lógica do contador. Com o objetivo de comparar os resultados com um contador sem TMR foi gerada uma adaptação do exemplo com apenas uma instância do contador e logicamente sem o votador. Contudo, a adaptação do contador de quatro bits utiliza quatro LUTs “nulas” para acessar o controle do multiplexador da lógica de vai-um. Por isso foram geradas mais duas adaptações do exemplo, uma com e outra sem TMR, onde foi utilizado uma LUT para implementar a lógica da porta XOR do contador.

O sinal de relógio utilizado no contador é o mesmo que alimenta o módulo *CLOCK\_GENERATOR* do SoCIF, de 100MHz. Os resultados obtidos com o exemplo do contador de quatro bits com TMR, e suas três adaptações são apresentados a seguir.

### 7.1.1 Contador com TMR com a Lógica de Carry (Original)

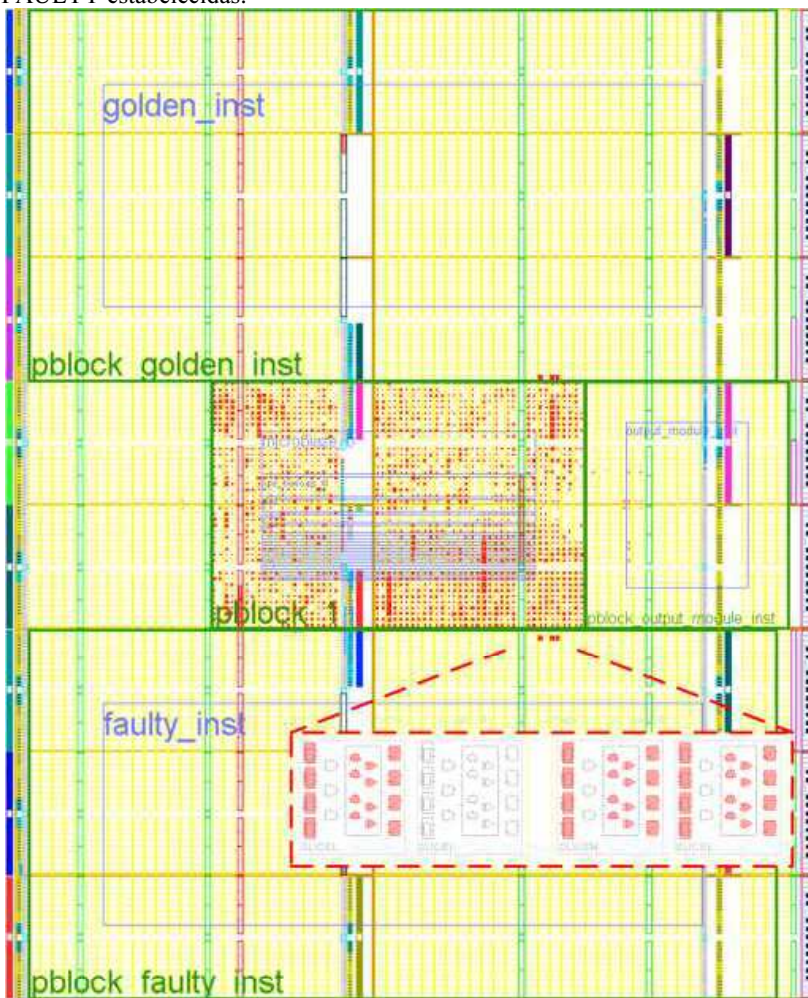
A Figura 58 mostra o resultado da implementação do contador de quatro bits com redundância tripla originado do exemplo da Xilinx juntamente com o SoCIF que se encontra no centro da figura. As instâncias FAULTY e GOLDEN do contador estão restringidas na parte inferior e superior respectivamente. Na Figura 58 também está destacado o resultado da implementação da instância FAULTY do DUT onde é possível observar a utilização de 12 LUTs, 12 registradores e 12 pares multiplexador/XOR que corresponde a três lógicas de *carry*, uma de cada *slice*.

A porcentagem de utilização dos recursos do FPGA modelo XC5VLX110T é apresentada na Figura 59. Na mesma figura é exibida a máxima frequência de operação do circuito implementado calculada pela ferramenta de síntese física, onde “FMax” corresponde a um valor maior do que os 125MHz do processador do SoCIF.

Dado a ínfima quantidade de recursos utilizados somando as duas instâncias do DUT podemos chamar a atenção para a ocupação do FPGA onde os 10% de *slices* representam quase que em sua totalidade o “tamanho” do SoCIF. É importante destacar ainda o uso de um dos dois componentes ICAP presentes no FPGA e a quantidade de portas utilizadas pelos módulos XPS\_SRAM e XPS\_SYSACE além do sinal de relógio e de reinicialização. A Figura 60 é parte do relatório de

implementação da ferramenta *PlanAhead* e apresenta em números a utilização do FPGA.

Figura 58 - *PlanAhead* do contador com a área das instâncias GOLDEN e FAULTY estabelecidas.



A análise experimental do contador com TMR da Xilinx utilizando o fluxo da PLEASER para a emulação de SETs é relatada pelo arquivo “RESULT.CSV” gerado pela plataforma e é apresentado na Figura 61. O relatório gerado é baseado no arquivo “FAULT.LIST” e contém o resultado de cada falha injetada. Considerando a Figura 58 os três *slices* utilizados pela instância FAULTY possuem a mesma

coordenada Y, ou seja, a coluna “Y” do arquivo de resultados da Figura 61 deve conter o mesmo valor, que no caso é 58. O que difere na localização dos três *slices* é a coordenada X que possui os valores 74,76 e 77 como mostra a coluna “X” do relatório gerado pela plataforma PLAESER.

Figura 59 - Resultados da implementação do contador com TMR.

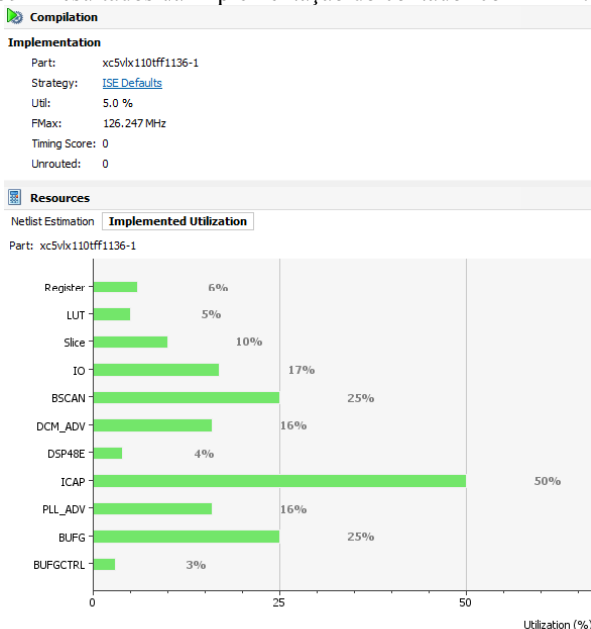


Figura 60 - Utilização dos recursos do FPGA XC5VLX110T.

Number of BSCANs	1 out of 4	25%
Number of BUFGs	8 out of 32	25%
Number of BUFGCTRLs	1 out of 32	3%
Number of DCM_ADVs	2 out of 12	16%
Number of DSP48Es	3 out of 64	4%
Number of ICAPs	1 out of 2	50%
Number of ILOGICs	16 out of 800	2%
Number of External IOBs	109 out of 640	17%
Number of LOCed IOBs	109 out of 109	100%
Number of OLOGICs	59 out of 800	7%
Number of PLL_ADVs	1 out of 6	16%
Number of RAMB18X2SDPs	1 out of 148	1%
Number of RAMB36_EXPs	2 out of 148	1%
Number of Slices	1842 out of 17280	10%
Number of Slice Registers	4189 out of 69120	6%
Number used as Flip Flops	4105	
Number used as Latches	0	
Number used as LatchThrus	4	
Number of Slice LUTs	3567 out of 69120	5%
Number of Slice LUT-Flip Flop pairs	5540 out of 69120	8%

Figura 61 - Resultado da emulação de falhas no contador TMR da Xilinx.

ID	X	Y	LUT_FRAME_ADDR	WORD_IDX	OFFS	SIGNAL_NAME	INSTANCE_NAME	LUT_EQUATION	LUT_MASK	ORD	VALU	ERR	TIME
0	76	58	0	0x001096A0	542881	37/[0][0].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR2[3]	O6=(A6**A6)*((0x0000FFFF	(00330017,	0	1			
1	76	58	1	0x001096A0	542881	37/[0][1].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR2[3]	O6=(A6**A6)*((0xFFFF0000	(00330017,	0	1			
2	76	58	2	0x001096A0	542881	38/[0][2].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR2[3]	O6=(A6**A6)*((0x0000FFFF	(033F0003,	0	1			
3	76	58	3	0x001096A0	542881	38/[0][2].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR2[3]	O6=(("A5"A6*A0x0FFFFFF0000	(033F0003,	0	1			
4	77	58	0	0x0010969A	542635	37/[0][0].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR1[3]	O6=(A6**A6)*((0x0000FFFF	(00170055,	0	1			
5	77	58	1	0x0010969A	542635	37/[0][1].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR1[3]	O6=(A6**A6)*((0xFFFF0000	(00170055,	0	1			
6	77	58	2	0x0010969A	542635	38/[0][2].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR1[3]	O6=(A6**A6)*((0x0000FFFF	(1717003F,	0	1			
7	77	58	3	0x0010969A	542635	38/[0][3].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR1[3]	O6=(("A5"A4*A0x0FFFFFF0000	(1717003F,	0	1			
8	74	58	0	0x00109620	541405	37/[0][0].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR0[3]	O6=(A6**A6)*((0x0000FFFF	(003F0055,	0	1			
9	74	58	1	0x00109620	541405	37/[0][1].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR0[3]	O6=(A6**A6)*((0xFF+0000	(003F0055,	0	1			
10	74	58	2	0x00109620	541405	38/[0][2].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR0[3]	O6=(A6**A6)*((0x0000FFFF	(77770017,	0	1			
11	74	58	3	0x00109620	541405	38/[0][3].if_tmr_true.VQ_inst_faulty_inst/COUNT_TR0[3]	O6=(("A4"A3*A0x0FFFFFF0000	(77770017,	0	1			
											TOTAL	0	12

As falhas emuladas foram aplicadas em cada uma das LUTs da instância FAULTY do DUT. Cada LUT representa a implementação de um votador. Considerando que há um votador para cada bit do contador de quatro bits, que por sua vez está triplicado, chega-se ao número de 12 falhas injetadas. O relatório da Figura 61 mostra o esperado para um circuito com redundância tripla, ou seja, todas as falhas injetadas foram mascaradas pelos votadores.

Uma alteração no software do SoCIF foi feita para emular a capacidade de tolerância do DUT quanto ao acúmulo de falhas, ou seja, cada lógica da LUT invertida não era revertida para a injeção da próxima falha. A Figura 62 apresenta o resultado da emulação de falhas acumuladas.

Figura 62 - Resultado da emulação de falhas acumulada no contador TMR da Xilinx.

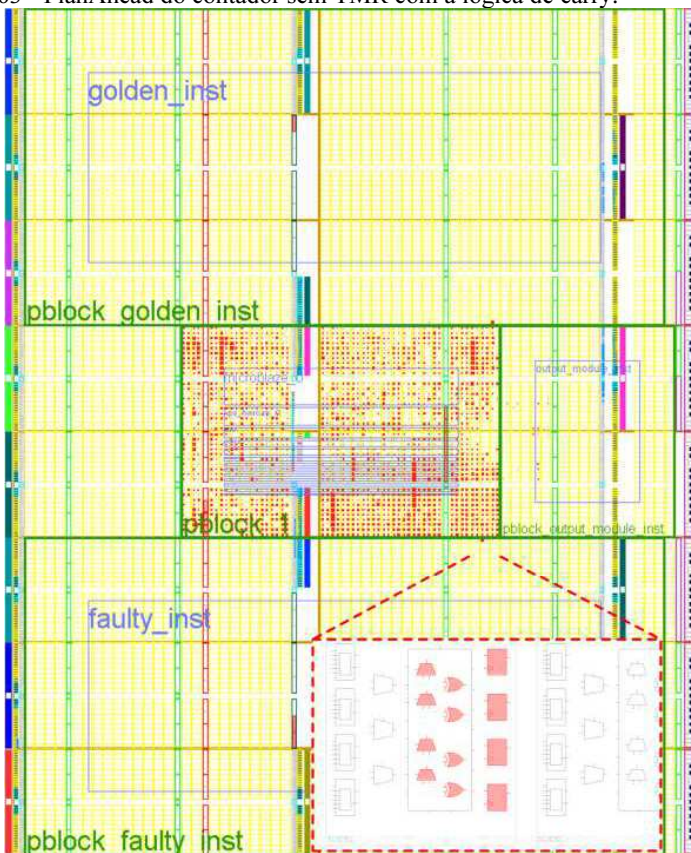
ID	X	Y	LUT_FRAME_ADDR	WORD_IDX	OFFS	SIGNAL_NAME	INSTANCE_NAME	LUT_EQUATION	LUT_MASK	WORD_VALUES	ERR	TIME	
0	76	58	0	0x001096A0	542881	37 if_tmr_true.VQ_inst_faulty_inst/COUNT_TR2[3]	O6=(A6**A6)*((0x0000FFFF	(00330017,0055C	0	1			
1	76	58	1	0x001096A0	542881	37 if_tmr_true.VQ_inst_faulty_inst/COUNT_TR2[3]	O6=(A6**A6)*((0xFFFF0000	(00330017,0055C	0	1			
2	76	58	2	0x001096A0	542881	38 if_tmr_true.VQ_inst_faulty_inst/COUNT_TR2[3]	O6=(A6**A6)*((0x0000FFFF	(033F0003,033F	0	1			
3	76	58	3	0x001096A0	542881	38 if_tmr_true.VQ_inst_faulty_inst/COUNT_TR2[3]	O6=(("A5"A6*A0x0FFFFFF0000	(033F0003,033F	0	1			
4	77	58	0	0x0010969A	542635	37 if_tmr_true.VQ_inst_faulty_inst/COUNT_TR1[3]	O6=(A6**A6)*((0x0000FFFF	(00170055,0017C	1	0			
5	77	58	1	0x0010969A	542635	37 if_tmr_true.VQ_inst_faulty_inst/COUNT_TR1[3]	O6=(A6**A6)*((0xFFFF0000	(00170055,0017C	1	0			
6	77	58	2	0x0010969A	542635	38 if_tmr_true.VQ_inst_faulty_inst/COUNT_TR1[3]	O6=(A6**A6)*((0x0000FFFF	(1717003F,1717C	1	0			
7	77	58	3	0x0010969A	542635	38 if_tmr_true.VQ_inst_faulty_inst/COUNT_TR1[3]	O6=(("A5"A4*A0x0FFFFFF0000	(1717003F,1717C	1	0			
8	74	58	0	0x00109620	541405	37 if_tmr_true.VQ_inst_faulty_inst/COUNT_TR0[3]	O6=(A6**A6)*((0x0000FFFF	(003F0055,003F	1	0			
9	74	58	1	0x00109620	541405	37 if_tmr_true.VQ_inst_faulty_inst/COUNT_TR0[3]	O6=(A6**A6)*((0xFFFF0000	(003F0055,003F	1	0			
10	74	58	2	0x00109620	541405	38 if_tmr_true.VQ_inst_faulty_inst/COUNT_TR0[3]	O6=(A6**A6)*((0x0000FFFF	(77770017,7777C	1	0			
11	74	58	3	0x00109620	541405	38 if_tmr_true.VQ_inst_faulty_inst/COUNT_TR0[3]	O6=(("A4"A3*A0x0FFFFFF0000	(77770017,7777C	1	0			
											TOTAL	8	4

A lista de falhas é gerada no passo de *slices*, ou seja, um *slice* com três LUTs ocupadas resulta na adição de três falhas na lista. Devido ao algoritmo da ferramenta de síntese física, cada contador de quatro bits foi implementado por completo dentro de um único *slice* do CLB. Logo, as quatro primeiras falhas foram injetadas em cada votador de um contador das três instâncias. Por isso, o teste feito com o acúmulo de falhas apresentou erros na saída do FAULTY a partir da quinta falha injetada. Fato que pode ser observado na Figura 62.

### 7.1.2 Contador sem TMR com a Lógica de Carry

O contador com TMR do exemplo original da Xilinx foi alterado para um circuito sem redundância, ou seja, foi deixada apenas uma instância do contador de quatro bits. A Figura 63 apresenta o resultado da implementação e também destaca os recursos utilizados pela instância FAULTY do DUT. É possível perceber que a instância FAULT do DUT foi implementada ocupando apenas um *slice*, e a sua ocupação consistiu apenas nos registradores e na lógica aritmética utilizada. Contudo, como havia sido dito anteriormente, as LUTs desse *slice* são utilizadas para o acesso ao controle dos multiplexadores.

Figura 63 - PlanAhead do contador sem TMR com a lógica de carry.

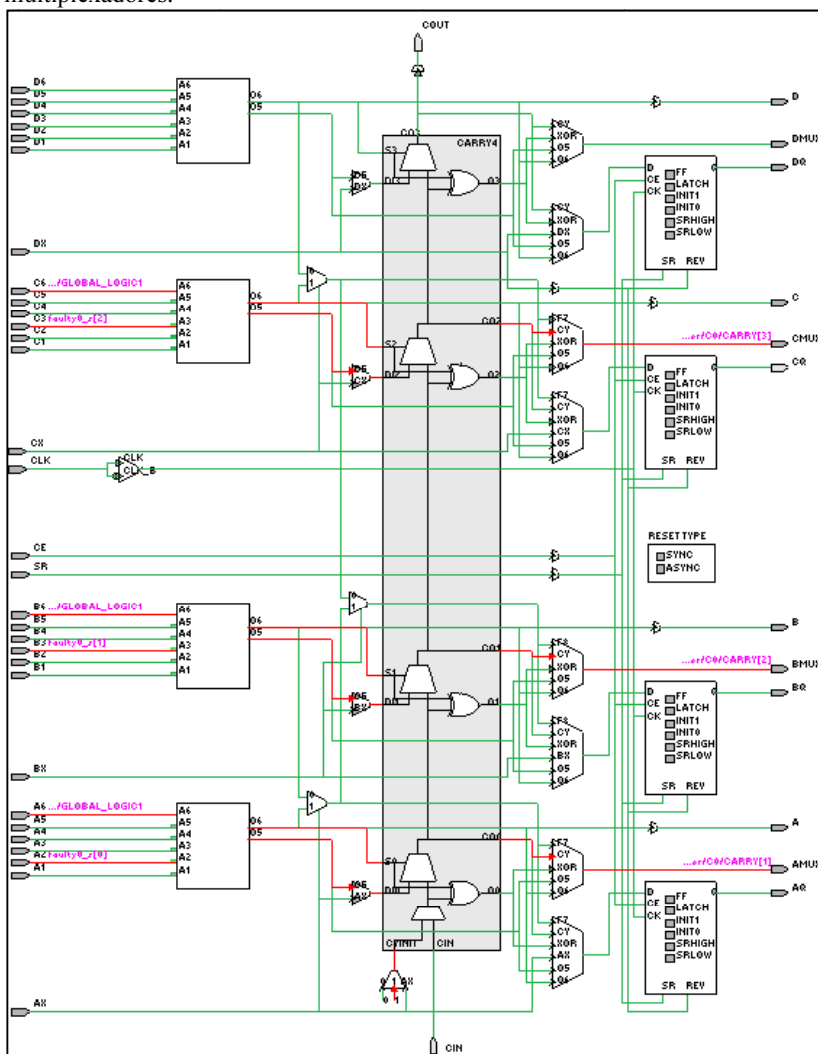


A Figura 64 ilustra um circuito, não o mesmo do contador, mas que também utiliza as LUTs para acessar o sinal de controle dos



multiplexadores da lógica aritmética do *slice*. Esta figura contém o circuito de um *slice* do SoCIF e foi feita utilizando a ferramenta FPGA Editor da Xilinx que permite a visualização do roteamento do circuito. A figura ilustra com detalhes a composição de um *slice* do FPGA da família Virtex5.

Figura 64 - Exemplo do uso da LUT para o acesso ao controle dos multiplexadores.



No relatório de utilização do FPGA é possível observar quatro *slices* a menos que o caso de teste anterior. Isso é devido à ausência das duas instâncias redundantes dos contadores de cada um dos DUTs. A Figura 65 destaca a diferença de quatro *slices* a menos no relatório de utilização.

Figura 65 - Utilização dos recursos do FPGA XC5VLX110T.

Number of BSCANs	1 out of 4	25%
Number of BUFGs	8 out of 32	25%
Number of BUFGCTRLs	1 out of 32	3%
Number of DCM_ADVs	2 out of 12	16%
Number of DSP48Es	3 out of 64	4%
Number of ICAPs	1 out of 2	50%
Number of ILOGICs	16 out of 800	2%
Number of External IOBs	109 out of 640	17%
Number of LOCed IOBs	109 out of 109	100%
Number of OLOGICs	59 out of 800	7%
Number of PLL_ADVs	1 out of 6	16%
Number of RAMB18X2SDPs	1 out of 148	1%
Number of RAMB36_EXPs	2 out of 148	1%
Number of Slices	1838 out of 17280	10%
Number of Slice Registers	4173 out of 69120	6%
Number used as Flip Flops	4169	
Number used as Latches	0	
Number used as LatchThrus	4	
Number of Slice LUTs	3551 out of 69120	5%
Number of Slice LUT-Flip Flop pairs	5524 out of 69120	7%

A Figura 66 apresenta o resultado da emulação de falhas gerado pela PLEASER. Como toda lógica do contador é implementada pela lógica de *carry* do *slice*, as falhas foram injetadas nas quatro LUTs de acesso ao controle dos multiplexadores. O resultado foi que toda a falha injetada resultou em erro.

Figura 66 - Resultado da emulação de falhas no contador sem TMR e com lógica de carry.

ID	X	Y	LUT	FRAME_ADDR	WORD_IDX	OFFS	SIGNAL_NAME	INSTANCE_NAME	LUT_EQUATION	LUT_MASK	WORD_VALUES	ERR	TIME
0	78	59	0	0x000109720	544357	39st	COUNT_TR0[0].rt.faulty_inst/COUNT_TR0[3]	O6=(A6**A6)*(A4)	0x0000FFFF	{00330033,0033}	1	0	
1	78	59	1	0x000109720	544357	39st	COUNT_TR0[1].rt.faulty_inst/COUNT_TR0[3]	O6=(A6**A6)*(A4)	0xFFFF0000	{00330033,0033}	1	0	
2	78	59	2	0x000109720	544357	40st	COUNT_TR0[2].rt.faulty_inst/COUNT_TR0[3]	O6=(A6**A6)*(A4)	0x0000FFFF	{33330033,3333}	1	0	
3	78	59	3	0x000109720	544357	40st	COUNT_TR0[3].rt.faulty_inst/COUNT_TR0[3]	O6=A4	0xFFFF0000	{33330033,3333}	1	0	
TOTAL												4	0

As equações lógicas das LUTs descritas no arquivo de resultados “RESULT.CSV” mostram que as LUTs são utilizadas apenas como lógica de cola (“de passagem”), pois não implementa lógica alguma. Dado que o sinal de adição “+”, o asterisco “\*” e acento fonético “~” representam as operações lógicas AND, OR e NOT respectivamente, podemos concluir que a saída da LUT equivale ao sinal da porta A4. As LUTs de seis entradas da Xilinx suportam a implementação de duas LUTs de “tamanho” cinco e as suas respectivas saídas serem usadas

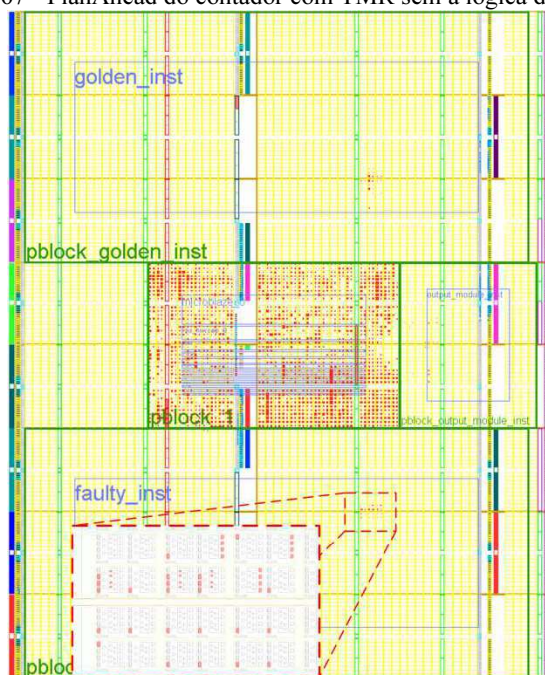


separadamente como mostrado na Figura 64. O uso da equação “ $(A6+\sim A6)$ ”, que equivale ao estado lógico um, provavelmente é utilizado para referenciar o uso das duas portas de saída da LUT o que justifica o fato da equação da última LUT, na Figura 66, ser diferente, uma vez que o sinal de *carry* do último bit não é utilizado.

### 7.1.3 Contador com TMR sem a Lógica de Carry

A operação lógica XOR foi retirada de dentro da lógica de *carry* do *slice* com o objetivo de aumentar o número de falhas a serem injetadas. Para isso, foi instanciada uma LUT de duas entradas e sua tabela verdade foi definida com o valor correspondente de uma porta lógica XOR. Além disso, foi utilizada a restrição, chamada de “LUT\_MAP”, direto na descrição do circuito (no VHDL), com o objetivo de forçar o mapeamento da entidade em uma LUT, ou seja, não permitindo otimizações que possam fazer com que a porta lógica XOR volte para o circuito de *carry*.

Figura 67 - PlanAhead do contador com TMR sem a lógica de carry.



A Figura 67 mostra o resultado da implementação com o destaque para a quantidade e a forma como os recursos lógicos do

FPGA foram utilizados pelo DUT. É importante ressaltar o uso de *slices* com apenas uma LUT ocupada na figura. Isso implica diretamente no relatório de utilização de recursos da Figura 68 onde se encontra destacado o número de *slices* ocupados.

Figura 68 - Utilização dos recursos do FPGA XC5VLX110T.

Number of BSCANS	1 out of 4	25%
Number of BUFGs	8 out of 32	25%
Number of BUFCTRLs	1 out of 32	3%
Number of DCM_ADVs	2 out of 12	16%
Number of DSP48Es	3 out of 64	4%
Number of ICAPs	1 out of 2	50%
Number of ILOGICs	16 out of 800	2%
Number of External IOBs	109 out of 640	17%
Number of LOCed IOBs	109 out of 109	100%
Number of OLOGICs	59 out of 800	7%
Number of PLL_ADVs	1 out of 6	16%
Number of RAMB18X2SDPs	1 out of 148	1%
Number of RAMB36_EXPs	2 out of 148	1%
Number of Slices	1877 out of 17280	10%
Number of Slice Registers	4189 out of 69120	6%
Number used as Flip Flops	4185	
Number used as Latches	0	
Number used as LatchThrus	4	
Number of Slice LUTs	3591 out of 69120	5%
Number of Slice LUT-Flip Flop pairs	5588 out of 69120	8%

Como não poderia ser diferente do resultado do contador com TMR original da Xilinx, o arquivo “RESULT.CSV” na Figura 69 mostra que nenhuma das falhas injetadas implicou em erro, ou seja, nenhuma falha gerou disparidade entre os sinais de saída das duas instâncias do DUT.

Figura 69 - Resultado da emulação de falhas no contador com TMR e sem a lógica de carry.

ID	X	Y	LUT	FRAME_ADDR	WORD_IDX	OFs	SIGNAL_NAME	INSTANCE_NAME	LUT_EQUATION	LUT_MASK	WORD_VALUES	ERR	TIME
0	75	39	0	0x0011616A	636361	39	i.e.XOR_X_inst/XCY	faulty_inst/counter_inst/counter/i0.C2/SUM[3]	O6=(A2@A4)	0x0000FFFF	(0000CCCC,0000	0	1
1	77	40	0	0x0010969A	542635	0	e.XOR_X_inst/XCY	faulty_inst/counter_inst/counter/i0.C2/SUM[2]	O6=(A1@A3)	0x0000FFFF	(00000000,0000F	0	1
2	74	39	0	0x00111620	636607	39	i.e.XOR_X_inst/XCY	faulty_inst/counter_inst/counter/i0.C2/SUM[1]	O6=(A3@A6)	0x0000FFFF	(000055AA,0000	0	1
3	76	39	0	0x001116A0	638083	39	i.e.XOR_X_inst/XCY	faulty_inst/counter_inst/counter/i0.C2/SUM[0]	O6=*A4	0x0000FFFF	(000055CC,0000	0	1
4	75	40	0	0x0010961A	541159	0	t_mr_true.VQ_inst	faulty_inst/counter_inst/counter/i0.C2/QINV[2]	O6=(A6*A6)*((10x0000FFFF	(00550077,000F	0	1	
5	75	40	1	0x0010961A	541159	0	t_mr_true.VQ_inst	faulty_inst/counter_inst/counter/i0.C2/QINV[2]	O6=(A6*A6)*((10x0000FFFF	(00550077,000F	0	1	
6	75	40	2	0x0010961A	541159	1	t_mr_true.VQ_inst	faulty_inst/counter_inst/counter/i0.C2/QINV[2]	O6=(A6*A6)*((10x0000FFFF	(00000117,0000	0	1	
7	74	38	0	0x00111620	636607	37	t_mr_true.VQ_inst	faulty_inst/counter_inst/counter/i0.C2/QINV[3]	O6=((*A2*(A5*A0x0000FFFF	(000055F5,0000	0	1	
8	77	38	0	0x0011169A	637837	37	e.XOR_X_inst/XCY	faulty_inst/counter_inst/counter/i0.C1/SUM[3]	O6=(A1@A6)	0x0000FFFF	(000055AA,0000	0	1
9	74	41	0	0x00109620	541405	2	e.XOR_X_inst/XCY	faulty_inst/counter_inst/counter/i0.C1/SUM[2]	U8=(A6@A3)	0x0000FFFF	(000055AA,0000	0	1
10	77	39	0	0x0011169A	637837	39	e.XOR_X_inst/XCY	faulty_inst/counter_inst/counter/i0.C1/SUM[1]	O6=(A1@A3)	0x0000FFFF	(00000000,0000F	0	1
11	76	41	3	0x001096A0	542881	3	e.XOR_X_inst/XCY	faulty_inst/counter_inst/counter/i0.C1/SUM[0]	O6=*A5	0x0000FFFF	(F0F0000, F0F0	0	1
12	74	40	0	0x00109620	541405	0	t_mr_true.VQ_inst	faulty_inst/counter_inst/counter/i0.C1/QINV[2]	O6=(A6*A6)*((10x0000FFFF	(00550077,000F	0	1	
13	74	40	1	0x00109620	541405	0	t_mr_true.VQ_inst	faulty_inst/counter_inst/counter/i0.C1/QINV[2]	O6=(A6*A6)*((10x0000FFFF	(00550077,000F	0	1	
14	74	40	2	0x00109620	541405	1	t_mr_true.VQ_inst	faulty_inst/counter_inst/counter/i0.C1/QINV[2]	O6=(A6*A6)*((10x0000FFFF	(00000117,0000	0	1	
15	76	38	0	0x001116A0	638083	37	t_mr_true.VQ_inst	faulty_inst/counter_inst/counter/i0.C1/QINV[3]	O6=((*A1*(A3*A0x0000FFFF	(00000F0F,0000	0	1	
16	72	38	3	0x001115A0	635131	38	e.XOR_X_inst/XCY	faulty_inst/counter_inst/counter/CO/SUM[3]	O6=(A1@A6)	0x0000FFFF	(A4550000,55A2	0	1
17	72	39	0	0x001115A0	635131	39	e.XOR_X_inst/XCY	faulty_inst/counter_inst/counter/CO/SUM[2]	O6=(A1@A3)	0x0000FFFF	(0000FFFF,0000	0	1
18	73	40	0	0x0010959A	539683	0	e.XOR_X_inst/XCY	faulty_inst/counter_inst/counter/CO/SUM[1]	O6=(A2@A5)	0x0000FFFF	(0000F0F0,0000	0	1
19	73	39	0	0x0011159A	634885	39	e.XOR_X_inst/XCY	faulty_inst/counter_inst/counter/CO/SUM[0]	O6=*A2	0x0000FFFF	(00000000,0000	0	1
20	72	40	0	0x001095A0	539929	0	t_mr_true.VQ_inst	faulty_inst/counter_inst/counter/CO/QINV[2]	O6=(A6*A6)*((10x0000FFFF	(00030005,0003	0	1	
21	72	40	1	0x001095A0	539929	0	t_mr_true.VQ_inst	faulty_inst/counter_inst/counter/CO/QINV[2]	O6=(A6*A6)*((10x0000FFFF	(00030005,0003	0	1	
22	72	40	2	0x001095A0	539929	1	t_mr_true.VQ_inst	faulty_inst/counter_inst/counter/CO/QINV[2]	O6=(A6*A6)*((10x0000FFFF	(00000117,0000	0	1	
23	75	38	0	0x0011161A	636361	37	t_mr_true.VQ_inst	faulty_inst/counter_inst/counter/CO/QINV[3]	O6=((*A2*(A5*A0x0000FFFF	(000055F5,0000	0	1	
TOTAL												0	24

Exatamente o que foi feito na análise experimental da técnica de TMR do contador da Xilinx foi repetido para esse exemplo. A Figura 70 apresenta os resultados obtidos com a injeção de falhas de forma acumulada. A lista de falhas foi montada pelo GLIFA seguindo a ordem em que os *slices* foram encontrados no arquivo XDL. Coincidentemente, as oito primeiras falhas foram injetadas em LUTs da redundância de número dois do contador, o que resultou na ocorrência do primeiro erro somente a partir da nona falha.

Figura 70 - Resultado da emulação de falhas acumulada no contador com TMR sem a lógica de carry.

ID	X	Y	LUT	FRAME_ADDR	WORD_IDX	OFS	SIGNAL_NAME	INSTANCE_NAME	LUT_EQUATION	LUT_MASK	WORD_VALUES	ERR	TIME
0	75	39	0	0x0011161A	636361	39	XOR_X_inst/XCY faulty_inst/counter_inst/counter/i0.C2/SUM[3]	O6=(A2@A4)	0x0000FFFF	0000CCCC,000C	0	1	
1	77	40	0	0x0010969A	542555	0	XOR_X_inst/XCY faulty_inst/counter_inst/counter/i0.C2/SUM[2]	O6=(A1@A3)	0x0000FFFF	00000000,0000	0	1	
2	74	39	0	0x00111620	636607	39	XOR_X_inst/XCY faulty_inst/counter_inst/counter/i0.C2/SUM[1]	O6=(A3@A6)	0x0000FFFF	000055AA,000C	0	1	
3	76	39	0	0x001116A0	638083	39	XOR_X_inst/XCY faulty_inst/counter_inst/counter/i0.C2/SUM[0]	O6=A4	0x0000FFFF	000000CC,000C	0	1	
4	75	40	0	0x0010961A	541159	0	mr_true_VQ_inst faulty_inst/counter_inst/counter/i0.C2/QINV[2]	O6=(A6+A6)*!(0x0000FFFF)	00550077,000F	0	1		
5	75	40	1	0x0010961A	541159	0	mr_true_VQ_inst faulty_inst/counter_inst/counter/i0.C2/QINV[2]	O6=(A6+A6)*!(0x0000FFFF)	00550077,000F	0	1		
6	75	40	2	0x0010961A	541159	1	mr_true_VQ_inst faulty_inst/counter_inst/counter/i0.C2/QINV[2]	O6=(A6+A6)*!(0x0000FFFF)	00000017,0000	0	1		
7	74	38	0	0x00111620	636607	37	mr_true_VQ_inst faulty_inst/counter_inst/counter/i0.C1/QINV[3]	O6=(A2*AS)*0x0000FFFF	00005F5F,0000	0	1		
8	77	38	0	0x0011169A	637837	37	XOR_X_inst/XCY faulty_inst/counter_inst/counter/i0.C1/SUM[3]	O6=(A1@A6)	0x0000FFFF	000055AA,000C	1	0	
9	74	41	0	0x00109620	541405	2	XOR_X_inst/XCY faulty_inst/counter_inst/counter/i0.C1/SUM[2]	O6=(A8@A3)	0x0000FFFF	000055AA,000C	1	0	
10	77	39	0	0x0011169A	637837	39	XOR_X_inst/XCY faulty_inst/counter_inst/counter/i0.C1/SUM[1]	O6=(A1@A3)	0x0000FFFF	00000000,0000	1	0	
11	76	41	3	0x001096A0	542881	3	XOR_X_inst/XCY faulty_inst/counter_inst/counter/i0.C1/SUM[0]	O6=A5	0x0000FFFF	F0F00000,F0F0	1	0	
12	74	40	0	0x00109620	541405	0	mr_true_VQ_inst faulty_inst/counter_inst/counter/i0.C1/QINV[2]	O6=(A6+A6)*!(0x0000FFFF)	00550077,000F	1	0		
13	74	40	1	0x00109620	541405	0	mr_true_VQ_inst faulty_inst/counter_inst/counter/i0.C1/QINV[2]	O6=(A6+A6)*!(0x0000FFFF)	00550077,000F	1	0		
14	74	40	2	0x00109620	541405	1	mr_true_VQ_inst faulty_inst/counter_inst/counter/i0.C1/QINV[2]	O6=(A6+A6)*!(0x0000FFFF)	00000017,0000	1	0		
15	76	38	0	0x001116A0	638083	37	mr_true_VQ_inst faulty_inst/counter_inst/counter/i0.C1/QINV[3]	O6=(A1*A3)*0x0000FFFF	000000F0,0000	1	0		
16	72	38	3	0x001115A0	635131	38	XOR_X_inst/XCY faulty_inst/counter_inst/counter/CO/SUM[3]	O6=(A1@A6)	0x0000FFFF	AA550000,55A	1	0	
17	72	39	0	0x001115A0	635131	39	XOR_X_inst/XCY faulty_inst/counter_inst/counter/CO/SUM[2]	O6=(A1@A3)	0x0000FFFF	0000FFFF,0000	1	0	
18	73	40	0	0x0010959A	539583	0	XOR_X_inst/XCY faulty_inst/counter_inst/counter/CO/SUM[1]	O6=(A2@A5)	0x0000FFFF	0000F0F0,0000	1	0	
19	73	39	0	0x0011159A	634885	39	XOR_X_inst/XCY faulty_inst/counter_inst/counter/CO/SUM[0]	O6=A2	0x0000FFFF	00000000,0000	1	0	
20	72	40	0	0x001095A0	539929	0	mr_true_VQ_inst faulty_inst/counter_inst/counter/CO/QINV[2]	O6=(A6+A6)*!(0x0000FFFF)	00030005,0003	1	0		
21	72	40	1	0x001095A0	539929	0	mr_true_VQ_inst faulty_inst/counter_inst/counter/CO/QINV[2]	O6=(A6+A6)*!(0x0000FFFF)	00030005,0003	1	0		
22	72	40	2	0x001095A0	539929	1	mr_true_VQ_inst faulty_inst/counter_inst/counter/CO/QINV[2]	O6=(A6+A6)*!(0x0000FFFF)	00000017,0000	1	0		
23	75	38	0	0x0011161A	636361	37	mr_true_VQ_inst faulty_inst/counter_inst/counter/CO/QINV[3]	O6=(A2*AS)*0x0000FFFF	00005F5F,0000	1	0		
											TOTAL	16	8

### 7.1.4 Contador sem TMR e sem a Lógica de Carry

O exemplo do contador utilizando LUTs para implementar a função da porta lógica XOR foi repetida, mas dessa vez, com apenas uma instância do contador, ou seja, sem a técnica de TMR. A Figura 71 apresenta o resultado da implementação do contador sem TMR e sem o uso da porta lógica XOR do circuito de *carry* do *slice* que foi substituída por uma LUT.

A ocupação de recursos do FPGA pelo SIF contendo as duas instâncias do DUT é apresentada na Figura 72. Nessa mesma figura está destacado o número de *slices* utilizados. É possível observar o número menor de *slices* comparados ao exemplo anterior, contudo, a utilização é maior do que em relação ao exemplo equivalente, que utiliza somente o circuito de *carry* para implementar a lógica aritmética do contador.

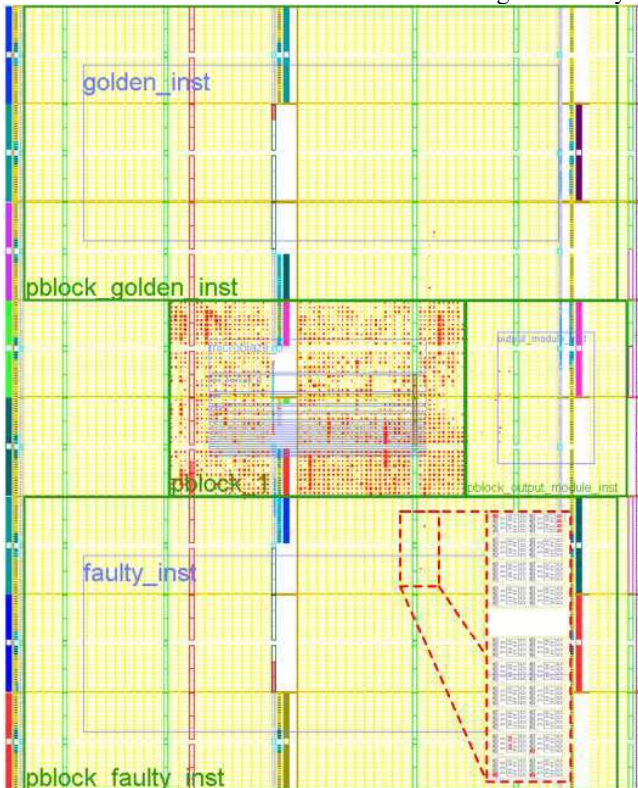
Figura 71 - *PlanAhead* do contador sem TMR e sem a lógica de carry.

Figura 72 - Utilização dos recursos do FPGA XC5VLX110T.

Number of BSCANs	1 out of 4	25%
Number of BUFGs	8 out of 32	25%
Number of BUFGCTRLs	1 out of 32	3%
Number of DCM_ADVs	2 out of 12	16%
Number of DSP48Es	3 out of 64	4%
Number of ICAPs	1 out of 2	50%
Number of ILOGICs	16 out of 800	2%
Number of External IOBs	109 out of 640	17%
Number of LOCed IOBs	109 out of 109	100%
Number of OLOGICs	59 out of 800	7%
Number of PLL_ADVs	1 out of 6	16%
Number of RAMB18X2SDPs	1 out of 148	1%
Number of RAMB36_EXPs	2 out of 148	1%
Number of Slices	1847 out of 17280	10%
Number of Slice Registers	4173 out of 69120	6%
Number used as Flip Flops	4169	
Number used as Latches	0	
Number used as LatchThrus	4	
Number of Slice LUTs	3557 out of 69120	5%
Number of Slice LUT-Flip Flop pairs	5538 out of 69120	8%

A diferença no número de *slices* em relação ao exemplo equivalente (sem TMR) pode ser explicada somente olhando para as equações booleanas implementadas pelas LUTs mostradas na Figura 73. As três primeiras LUTs implementam a função lógica da porta XOR. A função lógica da quarta LUT foi otimizada, uma vez que a porta lógica XOR do primeiro bit do contador tem uma de suas portas presa ao nível lógico um, o que equivale à negação do sinal da outra porta. Dado que somente a lógica da porta XOR foi retirada do circuito de carry, acarreta que o acesso aos multiplexadores que fazem o papel da porta AND do contador é realizado pelas três últimas LUTs da lista. Não há o quarto multiplexador, assim a LUT não é utilizada, uma vez que não há o sinal de *carry* do último bit do contador.

Figura 73 - Resultado da emulação de falhas no contador sem TMR e sem a lógica de carry.

ID	X	Y	LUT	FRAME_ADDR	WORD_IDX	OFFS	SIGNAL_NAME	INSTANCE_NAME	LUT_EQUATION	LUT_MASK	WORD_VALUES	ERR	TIME
0	72	53	3	0x001095A0	539929	28	ie.XOR_X_inst/XCY_faulty_inst/counter_inst/counter/c0/SUM[3]	O6=(A2@A4)	0xFFFF0000	(33330000,3333)	1	0	
1	72	44	0	0x001095A0	539929	8	ie.XOR_X_inst/XCY_faulty_inst/counter_inst/counter/c0/SUM[2]	O6=(A1@A5)	0x0000FFFF	(0000A5A5,0000)	1	0	
2	73	45	0	0x0010959A	539683	10	ie.XOR_X_inst/XCY_faulty_inst/counter_inst/counter/c0/SUM[1]	O6=(A2@A6)	0x0000FFFF	(0000F0F0,0000)	1	0	
3	73	44	0	0x0010959A	539683	8	ie.XOR_X_inst/XCY_faulty_inst/counter_inst/counter/c0/SUM[0]	O6="A2"	0x0000FFFF	(00000000,0000)	1	0	
4	72	45	0	0x001095A0	539929	10	/COUNT_TR0[0]_rt_faulty_inst/counter_inst/counter/c0/CARRY[3]	O6=(A6**A6)*(A2)	0x0000FFFF	(00550000,0055)	1	0	
5	72	45	1	0x001095A0	539929	10	/COUNT_TR0[1]_rt_faulty_inst/counter_inst/counter/c0/CARRY[3]	O6=(A6**A6)*(A3)	0xFFFF0000	(00550000,0055)	1	0	
6	72	45	2	0x001095A0	539929	11	/COUNT_TR0[2]_rt_faulty_inst/counter_inst/counter/c0/CARRY[3]	O6=(A6**A6)*(A3)	0x0000FFFF	(00000055,0000)	1	0	
											TOTAL	7	0

A contagem final dos resultados da emulação mostra que sem uma técnica de tolerância a falhas, todas as falhas injetadas tornaram-se erros.

## 7.2 PROCESSADOR LEON3 COM MONITOR DE BARRAMENTO

O processador LEON3 é um modelo descrito em VHDL completamente sintetizável de um processador 32 bits baseado na arquitetura SPARC V8. O modelo é altamente configurável e seu código é disponibilizado livremente para pesquisa ou educação. O LEON3 e outros dispositivos são disponibilizados pela Aeroflex Gaisler. Algumas características do LEON3 são listadas a seguir:

- Conjunto de instruções SPARC V8 com extensões V8e;
- Sete estágios de *pipeline*;
- Arquitetura de memória Harvard;
- Interface de barramento AHB AMBA2.0;
- Suporte para OCD;
- Suporta multi-processamento simétrico (SMP);
- Modo de economia de energia e *clock gating*;
- Até 125MHz em implementações em FPGAs
- Projeto inteiramente síncrono e utiliza apenas a borda do sinal de relógio.

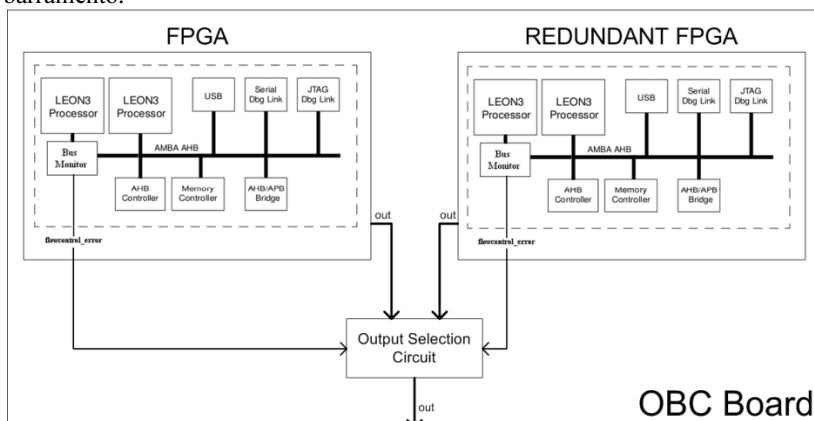
O LEON3 é completamente parametrizável e sua configuração pode ser definida através de uma interface gráfica que, por meio de *generics* da linguagem VHDL, permitem a construção de SoCs complexos a partir do uso de padrões pré-estabelecidos para periféricos. A biblioteca GRLIB possui um catálogo extenso de IPs reutilizáveis e independentes de vendedor. GRLIB é baseada no conceito de “centrado em barramento” o que permite o projeto de SoCs complexos através da interface de barramento AMBA 2.0 (AHB ou APB).

O fluxo completo de implementação é feito através de scripts em ambiente Linux que suportam as principais ferramentas de síntese e simulação do mercado. Além disso, existe o suporte com modelos pré-prontos para a implementação do LEON3 visando ASICs e para diversos kits de desenvolvimento com FPGAs das principais marcas do mercado. Nessa lista de kits suportados se encontram a XUPV5 utilizada para o desenvolvimento desse trabalho e a ML403 utilizada para o desenvolvimento do trabalho do OBC.

O trabalho realizado em (54) utiliza o processador LEON3 para controlar um computador de bordo, chamado de OBC (do inglês, *On-board Computer*), de um satélite. O OBC desenvolvido atende os requisitos fornecidos pelo Instituto Nacional de Pesquisa Espacial (INPE) e utiliza um monitor de barramento como a técnica de tolerância a falhas não intrusiva, uma vez que o OBC é um circuito crítico, dado que todo o processamento do satélite passa pelo computador de bordo. Na especificação do INPE existem dois núcleos que realizam a função de OBC, um principal e outro redundante. Além da redundância externa, em (54) é proposto o uso de FPGAs implementando dois processadores LEON sendo que um deles é o principal e o segundo é de uso exclusivo do monitor de barramento, ou seja, não serve o resto do SoC. Os dois processadores possuem as mesmas entradas e suas saídas são utilizadas pelo monitor de barramento para detecção de divergência dos resultados o que indica erro no fluxo de controle do OBC. Uma vez detectado o erro, o monitor de barramento avisa um circuito seletor externo que troca o fluxo de dados do FPGA principal para o redundante que agora passa ser o principal enquanto o outro é reconfigurado com o objetivo de corrigir o erro. A Figura 74, retirada de (54), apresenta a proposta do uso do monitor de barramento como técnica de tolerância a falhas para um computador de bordo.

O fluxo da plataforma PLAESER foi utilizado para análise experimental do OBC com a técnica de tolerância a falhas não-intrusiva. Analisando a forma em que a técnica de tolerância utilizando o monitor de barramento trabalha, foi considerada outra forma de aplicação do

SIF. Dado que a comparação entre as saídas dos dois processadores é realizada pelo monitor de barramento, foi escolhido então o processador LEON3 principal como sendo a instância FAULTY do DUT enquanto a instância do LEON3 utilizada pelo monitor de barramento é considerada a GOLDEN. A idéia geral é ligar a saída do monitor de barramento no SoCIF e verificar o seu comportamento na presença de falhas utilizando a análise experimental através de emulação com o fluxo da plataforma PLAESER. Os resultados da emulação de falha poderão ser confrontados com os obtidos através de simulação apresentados em (54). Figura 74 - Computador de bordo com redundância externa e monitor de barramento.



Fonte: F. Ferlini, F. Silva, E. Bezerra e D. Lettnin (2012).

Algumas adequações tiveram que ser realizadas com o objetivo de atender as restrições para o uso do fluxo PLAESER. Além disso, periféricos não utilizados pelo computador de bordo foram retirados do barramento do LEON3. Os ajustes foram:

- Remoção de periféricos não utilizados:
  - DDR SDRAM;
  - Módulo I2C;
  - Módulo Ethernet.
- Remoção de periféricos já ocupados pelo SoCIF;
  - Módulo JTAG para depuração;
  - Memória SRAM;
  - Módulo SystemAce;
- Troca da memória de instruções/dados do LEON3 para uma memória ROM e RAM utilizando BRAMS;

A versão original do OBC desenvolvido em (54) utiliza a memória SRAM do kit ML403 da Xilinx. Antes da implementação do



OBC para o kit XUPV5 utilizado nesse trabalho foi optado por substituir a memória de dados/instruções da SRAM para BRAMs que implementam a função de ROM/RAM para o uso do OBC. Os periféricos ROM e RAM da Gaisler são chamados de AHBROM e AHBROM respectivamente. Dessa forma, foi possível atender o requisito do fluxo da PLEASER de poder reinicializar o DUT e colocá-lo em um estado conhecido antes de toda injeção de falha.

Um software simples que realiza operações aritméticas básicas e envia resultados através da serial foi desenvolvido para simular o funcionamento do OBC. O código executável do software é gerado utilizando-se o compilador BCC (do inglês, *Bare-C Cross-Compiler*). O BCC é um compilador cruzado para processadores LEON2 e LEON3 baseado no GCC. O utilitário MKPROM2 é utilizado para gerar imagens de inicialização a partir de programas compilados com BCC. A imagem de inicialização é então traduzida por um script da Gaisler para forma de BRAMs descritas em VHDL que serão implementadas junto ao projeto do OBC. A imagem de inicialização do programa é implementada como uma ROM utilizando-se BRAMs, onde essas BRAMs armazenam as instruções a serem executadas pelo processador LEON3. A Figura 75 apresenta os comandos utilizados para geração do periférico AHBROM a partir do software teste desenvolvido em C.

Figura 75 - Comandos para geração da "BROM" com a imagem do programa teste para o LEON3.

```
sparc-elf-gcc -O2 -g -msoft-float send.c -o send.exe
mkprom2 -freq 80 -baud 115200 -msoft-float -ramsize 64 send.exe -o prom.out
cp prom.out ../../grrlib/designs/leon3-xilinx-ml509/
cd ../../grrlib/designs/leon3-xilinx-ml509/
rm ahbrom*
make ahbrom.vhd FILE=prom.out
make ise
```

O projeto desenvolvido em (54), carrega o programa na memória através do monitor de depuração do LEON, chamado de GRMON, que utiliza a interface JTAG, Serial ou o USB para se conectar com o processador. Considerando o fato de que o SoCIF precisa reiniciar o DUT toda vez que injeta uma falha, logo, foi providencial o uso da AHBROM contendo a imagem do programa para o mesmo poder ser re-executado para a injeção de cada nova falha. Durante a fase de configuração dos parâmetros do SoC do LEON, o espaço de endereçamento da SRAM foi atribuído para o do periférico AHBROM que faz o papel da RAM do processador, e assim concluindo a troca das memórias. A Figura 76 mostra a utilização do módulo de depuração da



Graisler (GRMON) para a leitura das informações do SoC com o LEON3 e os espaços de endereçamento dos periféricos.

Figura 76 - A configuração do SoC do LEON3 e espaço de endereçamento dos periféricos.

```

./grmon-eval -uart /dev/ttyUSB0 -u
GRMON LEON debug monitor v1.1.52 evaluation version

Copyright (C) 2004-2011 Aeroflex Gaisler - all rights reserved.
For latest updates, go to http://www.gaisler.com/
Comments or bug-reports to support@gaisler.com

This evaluation version will expire on 10/10/2012
using port /dev/ttyUSB0 @ 115200 baud

Device ID: : 0x509
GRLIB build version: 4113

initialising .....
detected frequency: 80 MHz

Component          Vendor
LEON3 SPARC V8 Processor      Gaisler Research
AHB Debug UART              Gaisler Research
AHB/APB Bridge               Gaisler Research
LEON3 Debug Support Unit     Gaisler Research
AHB ROM                      Gaisler Research
AHB static ram              Gaisler Research
Generic APB UART            Gaisler Research
Multi-processor Interrupt Ctrl Gaisler Research
Modular Timer Unit          Gaisler Research
General purpose I/O port    Gaisler Research
AHB status register         Gaisler Research

Use command 'info sys' to print a detailed report of attached cores

grlib> info sys
00.01:003 Gaisler Research LEON3 SPARC V8 Processor (ver 0x0)
          ahb master 0
01.01:007 Gaisler Research AHB Debug UART (ver 0x0)
          ahb master 1
          apb: 00000700 - 00000000
              baud rate 115200, ahb frequency 80.00
01.01:006 Gaisler Research AHB/APB Bridge (ver 0x0)
          ahb: 00000000 - 00100000
02.01:004 Gaisler Research LEON3 Debug Support Unit (ver 0x1)
          ahb: 00000000 - a0000000
              AHB trace 128 lines, 32-bit bus, stack pointer 0x4000ffff
              CPU#0 win 8, hwbp 2, itrace 128, lddel 1
              lcache 2 * 8 kbyte, 32 byte/line lru
              dcache 2 * 8 kbyte, 16 byte/line lru
06.01:01b Gaisler Research AHB ROM (ver 0x0)
          ahb: 00000000 - 00100000
07.01:00e Gaisler Research AHB static ram (ver 0x10)
          ahb: 40000000 - 40100000
01.01:00c Gaisler Research Generic APB UART (ver 0x1)
          irq 2
          apb: 00000100 - 00000200
              baud rate 38401, DSU mode (FIFO debug)
02.01:00d Gaisler Research Multi-processor interrupt Ctrl (ver 0x3)
          apb: 00000200 - 00000300
03.01:011 Gaisler Research Modular Timer Unit (ver 0x0)
          irq 8
          apb: 00000300 - 00000400
              8-bit scaler, 2 * 32-bit timers, divisor 80
08.01:01a Gaisler Research General purpose I/O port (ver 0x1)
          irq 7
          apb: 00000000 - 00000900
0f.01:052 Gaisler Research AHB status register (ver 0x0)
          irq 7
          apb: 00000f00 - 00001000
grlib>

```

O projeto do sistema injetor de falhas (SIF) para análise experimental do OBC desenvolvido em (54) tem as seguintes diferenças do OBC original:

- Migração do OBC do kit da ML403 para o XUPV5;
- Troca da memória de dados/instruções de SRAM para ROM/RAM utilizando BRAMS;
- Frequência de operação de 50MHz para 80MHz na XUPV5;
- Software do OBC simplificado;
- Remoção de módulos não utilizados.

O OBC executando o software desenvolvido no kit XUPV5 é ilustrado na Figura 77. A figura mostra o momento em que o OBC é reinicializado após a injeção de uma falha e a imagem do programa é automaticamente recarregada para a retomada da execução.

O utilitário MKPROM2 insere na imagem do programa algumas características do sistema, como a versão do utilitário, frequência de operação, espaços de endereçamento, tamanho das memórias PROM e SRAM, entre outras. Essas características são carregadas no início da execução da imagem do programa pelo processador LEON3 que envia essas informações através da saída padrão, nesse caso para serial, como mostra a Figura 77.

Figura 77 - A reinicialização da execução do software do OBC após a injeção de uma falha.

```
send(1307)
send(1308)
send(1309)
send(1310)
send(1311)
send(1312)
send(1313)
send(1314)

MkProm2 boot loader v2.0
Copyright Gaisler Research - all rights reserved

system clock   : 80.0 MHz
baud rate      : 114942 baud
prom           : 512 K, (2/2) ws (r/w)
sram           : 64 K, 1 bank(s), 0/0 ws (r/w)

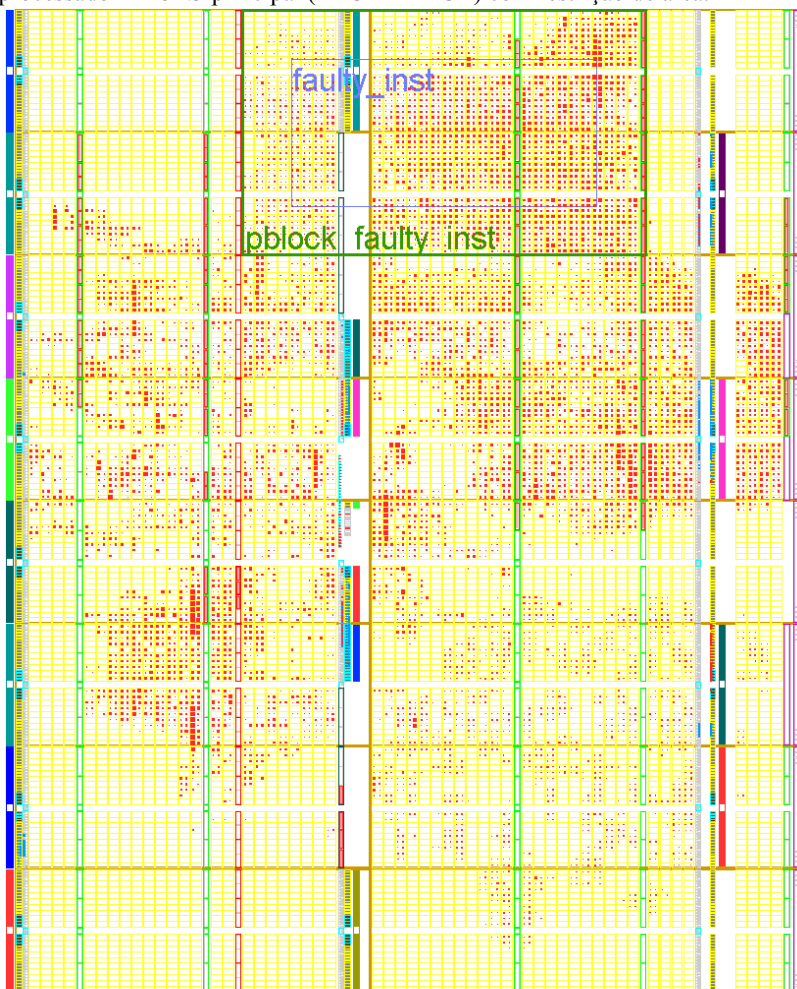
decompressing .text to 0x40000000
decompressing .data to 0x4000ac40

starting send.exe

send(0)
send(1)
send(2)
send(3)
send(4)
send(5)
send(6)
send(7)
send(8)
send(9)
send(10)
send(11)
send(12)
send(13)
send(14)
send(15)
send(16)
send(17)
send(18)
send(19)
```

A Figura 78 apresenta o resultado da implementação do SIF contendo o SoCIF, as instâncias FAULTY e GOLDEN do processador LEON3 do OBC (DUT) juntamente com o barramento AMBA e seus periféricos. Diferentemente do projeto dos contadores o SoCIF não foi importado, ou seja, foi implementado juntamente com a instância GOLDEN do DUT e o resto dos periféricos enquanto o FAULTY do DUT foi implementado separadamente, na parte superior do FPGA, para que as falhas nele não possam interferir no resto do circuito.

Figura 78 - PlanAhead da implementação do SIF com a instância do processador LEON3 principal (FAULTY DUT) com restrição de área.



A ocupação do FPGA com o projeto do SIF com o LEON3 é detalhada no gráfico da Figura 79 com a porcentagem de utilização de cada componente. Na mesma figura é apresentada a frequência máxima de operação do projeto do PlanAhead que no caso é do circuito do SoCIF, porque, a frequência do LEON3 é de 80MHz. Na Figura 80 se encontra o número de componentes utilizados do total disponível no FPGA destacando a quantidade de *slices* utilizados.

Figura 79 - Percentual de ocupação do FPGA com o projeto do SIF com o LEON3.

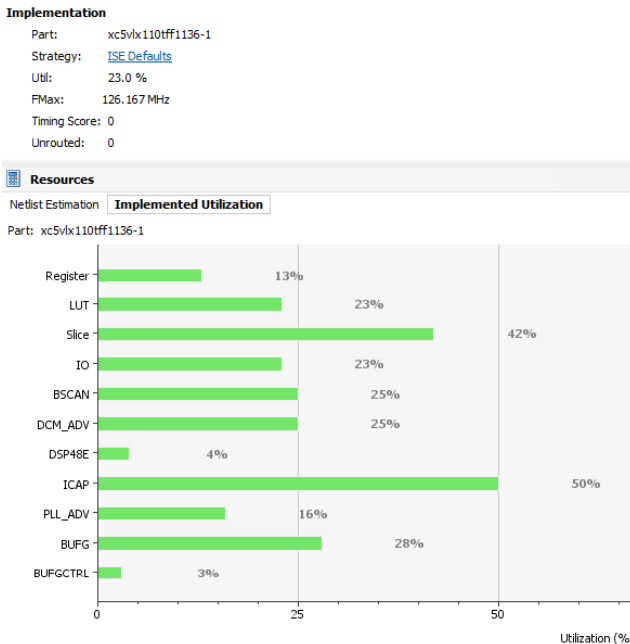
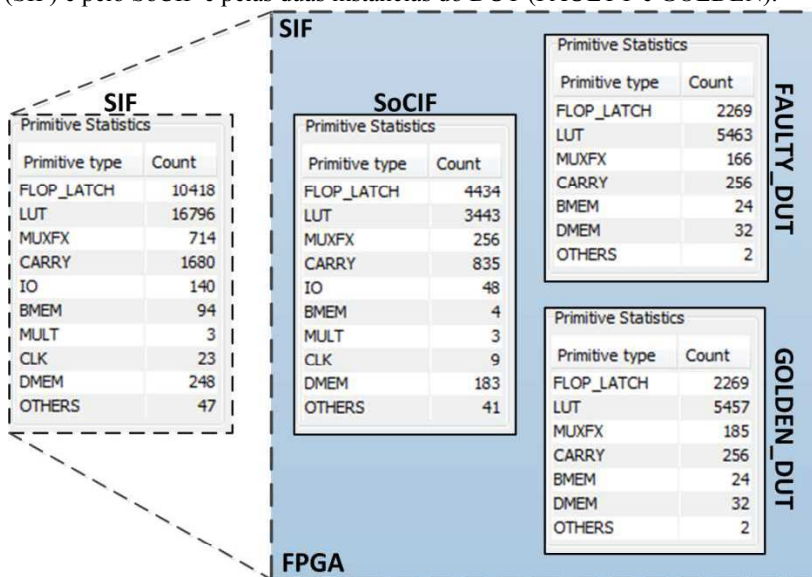


Figura 80 - Quantidade de recursos do FPGA utilizados no projeto SIF com o LEON3.

Number of BSCANS	1 out of 4	25%
Number of BUFGs	9 out of 32	28%
Number of BUFGCTRLs	1 out of 32	3%
Number of DCM_ADVs	3 out of 12	25%
Number of DSP48Es	3 out of 64	4%
Number of ICAPs	1 out of 2	50%
Number of ILOGICs	62 out of 800	7%
Number of External IOBs	150 out of 640	23%
Number of LOCed IOBs	129 out of 150	86%
Number of OLOGICs	108 out of 800	13%
Number of PLL_ADVs	1 out of 6	16%
Number of RAMB18X2s	34 out of 148	22%
Number of RAMB18X2SDPs	3 out of 148	2%
Number of RAMB36_EXPs	24 out of 148	16%
Number of Slices	<b>7279</b> out of 17280	<b>42%</b>
Number of Slice Registers	9148 out of 69120	13%
Number used as Flip Flops	9132	
Number used as Latches	0	
Number used as LatchThrus	16	
Number of Slice LUTs	16545 out of 69120	23%
Number of Slice LUT-Flip Flop pairs	19598 out of 69120	28%

A contagem estimada pós síntese lógica de componentes primitivos utilizados pelo projeto inteiro (SIF) é mostrada na Figura 81, onde se encontram também a estimativa para a instância do SoCIF e para as instâncias do DUT. Essa estimativa serve para informar a importância de cada instância na utilização dos recursos do FPGA. É importante destacar que a diferença entre os valores da Figura 80 e a quantidade de primitivas estimadas ocorre devido a otimizações feitas durante a implementação.

Figura 81 – Quantidade estimada de componentes utilizados no projeto todo (SIF) e pelo SoCIF e pelas duas instâncias do DUT (FAULTY e GOLDEN).



A lista de falhas gerada pelo GLIFA contém as 5425 LUTs da instância FAULTY do DUT encontradas no arquivo (XDL) com a lista de componentes pós-implementação. O relatório gerado no final do fluxo da plataforma PLAESER com o resultado obtido da análise do monitor de barramento do OBC é apresentado na Figura 82 e o seu resumo na Tabela 9.

Tabela 9 - Resumo do resultado da injeção de falhas no OBC com o monitor de barramento.

Total de falhas injetadas	5425
Falhas detectadas pelo monitor de barramento	3916
Falhas que permaneceram latentes ou silenciosas	1509

Na Tabela 9 é possível notar que o total de falhas injetadas é menor que a quantidade de LUTs estimadas mostradas na Figura 81 devido a otimizações.

Figura 82 - Resultado da injeção de falhas no OBC com o monitor de barramento.

ID	X	Y	LUT	ADDR	WORD_IDX	OFFS	SIGNAL_NAME	INSTANCE_NAME	LUT_EQUATION	LUT_MASK	WORD_VALUES	ERR	TIME
5379	91	139	2	0x0000011A	267361	40	/io/rlu/rin_e_op2101_1	faulty_inst/p0/lu0/r.e.op2_10	0b{("A4"~("A1"~("A2"~A1)0x0000FFFF (55F55505,0555	1	0	0	
5380	91	139	3	0x0000011A	267361	40	/io/rlu/rin_e_op2101_1	faulty_inst/p0/lu0/r.e.op2_10	0b{("A2"~("A5"~A6)~A5)1}0x0000FFFF (0005F7F,0000	1	0	0	
5381	91	140	0	0x0000019A	362563	0	inst/p0/lu0/rf_ren1	faulty_inst/rf0/s1.dp.v0/r.renable	0b{("A1"~("A5"~A6)~A5)1}0x0000FFFF (0005F7F,0000	1	0	0	
5382	91	141	0	0x0000019A	362563	2	icache0/crami_data(1)	faulty_inst/crami_cramin_data(3)	0b{("A5"~A2)~A5)~A4}0x0000FFFF (7777888,7777	1	0	0	
5383	91	141	1	0x0000019A	362563	2	icache0/crami_data(2)	faulty_inst/crami_cramin_data(3)	0b{("A5"~A4)~A5)~A2}0x0000FFFF (000F3F3,0000	1	0	0	
5384	91	141	2	0x0000019A	362563	3	icache0/crami_data(3)	faulty_inst/crami_cramin_data(3)	0b{("A5"~A2)~A5)~A4}0x0000FFFF (000F3F3,0000	1	0	0	
5385	91	144	0	0x0000019A	362563	8	icache0/crami_data(7)	faulty_inst/p0/npic_mux0005(8)10	0b{("A2"~A6)~A2)~A1}0x0000FFFF (0000555,0000	1	0	0	
5386	91	144	3	0x0000019A	362563	9	/io/rlu/npic_mux0005(8)10	faulty_inst/p0/npic_mux0005(8)10	0b{("A1"~("A2"~A1"~("A1"~("A5"~A6)~A5)1}0x0000FFFF (1100000,2100	1	0	0	
5387	91	146	0	0x0000019A	362563	12	inst/p0/lu0/rf_ren2	faulty_inst/rf0/s1.dp.v1/r.renable	0b{("A4"~A6)~A4)~("A6"~0x0000FFFF (000000F,0000	1	0	0	
5388	91	147	0	0x0000019A	362563	14	/io/data0_mux0000(8)13	faulty_inst/p0/lu0/data0_mux0000(8)13	0b{("A5"~A3)~("A4"~A4)~A5}0x0000FFFF (0000008,0000	1	0	0	
5389	91	148	1	0x0000019A	362563	16	inst/p0/lu0/dbgo_bpmis1	faulty_inst/p0/lu0/bpmis1	0b{("A5"~A5)~A6}0x0000FFFF (0005000,0000	0	1	0	
5390	91	148	2	0x0000019A	362563	17	inst/p0/lu0/dbgo_bpmis1	faulty_inst/p0/lu0/bpmis1	0b{("A4"~("A1"~("A3"~A6)~A6)~A6}0x0000FFFF (0002F0F,0000	1	0	0	
5391	91	149	0	0x0000019A	362563	18	inst/p0/lu0_d_pc_not0011	faulty_inst/cmem0/tdatain(0)20	0b{("A3"~("A4"~A5)~A3)1}0x0000FFFF (0200005,5100	1	0	0	
5392	91	149	1	0x0000019A	362563	18	inst/cmem0_mux001(1)	faulty_inst/cmem0/tdatain(0)20	0b{("A5"~A3)~("A1"~A6)~A6}0x0000FFFF (0200005,5100	1	0	0	
5393	91	150	0	0x0000019A	362563	21	icache/wtag_mux001(7)	faulty_inst/p0/lu0/p.v0_and0000	0b{("A2"~A6)~A5)~("A1"~A2)0x0000FFFF (2020220,0200	1	0	0	
5394	91	150	1	0x0000019A	362563	21	icache/wtag_mux001(8)	faulty_inst/p0/lu0/p.v0_and0000	0b{("A1"~A4)~A5)~("A2"~A2)0x0000FFFF (2020220,0200	1	0	0	
5395	91	150	2	0x0000019A	362563	22	icache/wtag_mux001(9)	faulty_inst/p0/lu0/p.v0_and0000	0b{("A2"~A6)~A5)~("A1"~A2)0x0000FFFF (0100400,0200	1	0	0	
5396	91	150	3	0x0000019A	362563	22	inst/p0/lu0_and0001	faulty_inst/p0/lu0/p.v0_and0000	0b{("A5"~A1)~("A4"~A6)~A6}0x0000FFFF (0100400,0200	1	0	0	
5397	91	151	0	0x0000019A	362563	23	icache/wtag_mux001(2)	faulty_inst/crami_cramin_tag(0)14	0b{("A2"~A6)~A5)~("A4"~0x0000FFFF (400400A,AD00	1	0	0	
5398	91	151	1	0x0000019A	362563	23	icache/wtag_mux001(4)	faulty_inst/crami_cramin_tag(0)14	0b{("A2"~A6)~A5)~("A4"~0x0000FFFF (400400A,AD00	1	0	0	
5399	91	151	2	0x0000019A	362563	24	icache/wtag_mux001(3)	faulty_inst/crami_cramin_tag(0)14	0b{("A4"~A3)~("A5"~A4)~A5}0x0000FFFF (5000500,6000	1	0	0	
5400	91	151	3	0x0000019A	362563	24	icache/wtag_mux001(1)	faulty_inst/crami_cramin_tag(0)14	0b{("A3"~("A5"~A4)~A5)~A2}0x0000FFFF (5000500,6000	1	0	0	
5401	91	152	0	0x0000019A	362563	25	eo/vmask_mux0004(1)63	faulty_inst/cmem0/tdatain(0)23	0b{("A3"~A2)~A1)~A6}0x0000FFFF (11515F0,AA2A	1	0	0	
5402	91	152	1	0x0000019A	362563	25	eo/vmask_mux0004(1)50	faulty_inst/cmem0/tdatain(0)23	0b{("A1"~A6)~("A3"~A5)~A5}0x0000FFFF (11515F0,AA2A	1	0	0	
5403	91	152	3	0x0000019A	362563	26	inst/cmem0_mux0011	faulty_inst/cmem0/tdatain(0)23	0b{("A6"~("A4"~A6)~A2)~A5}0x0000FFFF (7F0F000,70F0	1	0	0	
5404	91	152	3	0x0000019A	362563	26	inst/cmem0_mux0011	faulty_inst/cmem0/tdatain(0)23	0b{("A2"~("A6"~A4)~A5)~A5}0x0000FFFF (7F0F000,70F0	1	0	0	
5405	91	153	0	0x0000019A	362563	27	eo/vmask_mux0004(1)14	faulty_inst/p0/dmmu/cache0/vmask_mux0004(5)10	0b{("A5"~A2)~("A3"~A3)~A5}0x0000FFFF (7F7F001,5555	1	0	0	
5406	91	153	1	0x0000019A	362563	27	io/npic_mux0005(4)87	faulty_inst/p0/dmmu/cache0/vmask_mux0004(5)10	0b{("A4"~("A5"~A5)~A5)~A5}0x0000FFFF (7F7F001,5555	1	0	0	
5407	91	153	2	0x0000019A	362563	28	eo/vmask_mux0004(5)10	faulty_inst/p0/dmmu/cache0/vmask_mux0004(5)10	0b{("A5"~A2)~A6)~A1)~A1}0x0000FFFF (0000F11,0000	1	0	0	
5408	91	154	0	0x0000019A	362563	29	eo/vmask_mux0004(1)74	faulty_inst/p0/lu0/p.v0_and0006	0b{("A3"~A4)~A2)~A8}0x0000FFFF (3303011,3300	1	0	0	
5409	91	154	1	0x0000019A	362563	29	inst/p0/lu0/p.v0_and0006	faulty_inst/p0/lu0/p.v0_and0006	0b{("A4"~A6)~A5)~A2}0x0000FFFF (3303011,3300	1	0	0	
5410	91	155	0	0x0000019A	362563	31	inst/p0/lu0_rin_d_inu16	faulty_inst/p0/lu0/r.d.inu11	0b{("A6"~("A1"~A3)~("A1"~A5)0x0000FFFF (0000338,0000	1	0	0	
5411	91	155	1	0x0000019A	362563	31	inst/p0/lu0_rin_d_inu18	faulty_inst/p0/lu0/r.d.inu11	0b{("A1"~A5)~("A1"~A4)~A4}0x0000FFFF (0000338,0000	1	0	0	
5412	91	155	2	0x0000019A	362563	32	_valid_mux0000(0)_SW0	faulty_inst/p0/lu0/r.d.inu11	0b{("A1"~A2)~("A7"~A7)~A}0x0000FFFF (3201011,5501	1	0	0	
5413	91	155	3	0x0000019A	362563	32	_valid_mux0000(1)_SW0	faulty_inst/p0/lu0/r.d.inu11	0b{("A2"~("A3"~A4)~A5)~A5}0x0000FFFF (3201011,5501	1	0	0	
5414	91	156	0	0x0000019A	362563	33	eo/vmask_mux0004(4)63	faulty_inst/p0/dmmu/cache0/vmask_mux0004(7)41	0b{("A1"~A3)~A4)~A2)~A2}0x0000FFFF (00F30F0,0073	1	0	0	
5415	91	156	1	0x0000019A	362563	33	eo/vmask_mux0004(4)50	faulty_inst/p0/dmmu/cache0/vmask_mux0004(7)41	0b{("A5"~("A1"~A3)~A1)~A5}0x0000FFFF (00F30F0,0073	1	0	0	
5416	91	156	2	0x0000019A	362563	34	eo/vmask_mux0004(1)73	faulty_inst/p0/dmmu/cache0/vmask_mux0004(7)41	0b{("A3"~("A5"~A1)~A1)~A1}0x0000FFFF (555500F,AAAA	1	0	0	
5417	91	156	3	0x0000019A	362563	34	eo/vmask_mux0004(1)74	faulty_inst/p0/dmmu/cache0/vmask_mux0004(7)41	0b{("A1"~("A4"~A1)~A6)~A2}0x0000FFFF (555500F,AAAA	1	0	0	
5418	91	157	0	0x0000019A	362563	35	the0r_valid_mux0006(1)	faulty_inst/p0/dmmu/cache0/valid_7	0b{("A1"~("A2"~A3)~A3)~A1}0x0000FFFF (3303011,3300	1	0	0	
5419	91	157	1	0x0000019A	362563	35	the0r_valid_mux0006(1)	faulty_inst/p0/dmmu/cache0/valid_7	0b{("A1"~A6)~A3)~A1}0x0000FFFF (1555555,2A7F	1	0	0	
5420	91	157	2	0x0000019A	362563	36	the0r_valid_mux0006(1)	faulty_inst/p0/dmmu/cache0/valid_7	0b{("A6"~A4)~A5)~A6}0x0000FFFF (05D0D37,05E	1	0	0	
5421	91	157	3	0x0000019A	362563	36	the0r_valid_mux0007(1)	faulty_inst/p0/dmmu/cache0/valid_7	0b{("A6"~A3)~A5)~A6}0x0000FFFF (05D0D37,05E	1	0	0	
5422	91	158	0	0x0000019A	362563	37	the0r_valid_mux0006(1)	faulty_inst/p0/dmmu/cache0/valid_not0001	0b{("A6"~A3)~A5)~A6}0x0000FFFF (1F5F555,1F5F	1	0	0	
5423	91	158	1	0x0000019A	362563	37	io/rlu_rin_d_cmt0	faulty_inst/p0/dmmu/cache0/valid_not0001	0b{("A2)~("A5"~A1)~A3)~A3}0x0000FFFF (FFF5F5F,FFF	1	0	0	
5424	91	158	2	0x0000019A	362563	38	icache0/valid_not0001	faulty_inst/p0/dmmu/cache0/valid_not0001	0b{("A1"~A5)~A4)~A1}0x0000FFFF (000000F,0000	1	0	0	
TOTAL												3916	1509

A execução completa do software do SoCif com a injeção das 5425 falhas levou menos de 20 minutos. Tempo consideravelmente menor do que as simulações realizadas em (54) que levavam de 20 horas até mais de um dia para simular os 1090 sinais do projeto.

O resultado obtido mostra que mais de 72% das falhas injetadas provocaram mudança na saída do monitor de barramento. Resultado quase que inverso ao do encontrado por simulação, onde somente 18%, em média, das falhas foram detectadas pelo monitor de barramento.

A simulação da injeção de falhas em (54) foi realizada em três momentos diferentes da execução do software do OBC. Isso não foi repetido com o SoCif pela falta da abstração de tempo na emulação de falhas. Contudo, o software do SoCif foi alterado para o caso do OBC. As falhas só são injetadas após a imagem do programa do OBC ter sido carregada e estar em execução. Isso foi feito adicionando um laço nulo no código do software do SoCif que gastasse tempo suficiente para que a falha fosse injetada após a execução completa da inicialização do OBC.

## 8 TRABALHOS FUTUROS

O uso de dispositivos reconfiguráveis tem se mostrado um grande aliado na área de confiabilidade de circuitos endurecidos. Trabalhos explorando FPGAs para emulação de falhas têm aparecido com frequência na literatura. O fluxo da PLAESER busca que os usuários possam fazer uma análise experimental básica de seus projetos tolerantes a falhas de forma rápida e sem que para isso tenham que desenvolver seu próprio injetor de falhas. Posteriormente, o usuário pode fazer ajustes na plataforma PLAESER para melhorar a cobertura de análise de seu projeto sem a necessidade de um conhecimento profundo de técnicas de injeção de falhas. Porém, existe muito trabalho a ser desenvolvido para que a PLAESER se torne uma ferramenta genérica de análise experimental de *soft errors*.

Além de aumentar os casos de testes e depurar novos erros, algumas modificações devem ser feitas em curto prazo para consolidar essa versão da PLAESER. As modificações são:

- Integração de todo fluxo da PLAESER através de uma única interface para automatizar e agilizar todo o processo até a geração de resultados.
- Automatização da geração do TOPO\_SIF com as instâncias do DUT e do SoCIF;
- Geração de um comparador genérico, que possa ser empregado na comparação das duas versões de qualquer tipo de DUT, independente do número de saídas;
- Criação de uma forma de configurar a ferramenta através de parâmetros relacionados à injeção de falhas como, o momento de cada injeção, o tipo de falha emulada, os sinais de saída que devem ser comparados, o tipo de carga de trabalho do DUT, entre outros.
- Adição da medição de tempo de execução do software pelo SoCIF através do recurso de criação de perfil de desempenho presente na ferramenta SDK. Essa medição é útil para comparação da PLAESER em relação a outras técnicas de injeção.

Buscando diminuir as limitações da plataforma PLAESER, o desenvolvimento de novas funções deve ser feita em médio prazo. O objetivo dessas funções é abranger um maior número de DUTs e aumentar a quantidade de resultados obtidos por análise. Essas novas funções são:

- Substituição do módulo XPS\_HWICAP por um desenvolvido. O módulo desenvolvido será específico para emulação de falhas. Com acesso/controlado do sinal de relógio do DUT poderá ser definido o momento e a duração de cada injeção da falha possibilitando o uso de ciclos de relógio como abstração de tempo para PLEASER.
- Ampliação do uso da porta de acesso a configuração do FPGA para observar e controlar outros componentes além de LUTs como registradores, BRAMs, matrizes de roteamento, componentes especiais, entre outros;
- Adição de novos modelos de falhas além do SET, para que seja possível a emulação de efeitos de SEUs, MBUs, SELs entre outros;
- Adição de interfaces que substituam o uso de componentes específicos, como *CompactFlash* e memória SRAM.
- Suporte para mais modelos de FPGAs da Xilinx. Bibliotecas com suporte aos principais kits de desenvolvimento da Xilinx.

A redução da ocupação do SoCIF é fundamental para permitir o uso da PLEASER em FPGAs menores. O desenvolvimento de um módulo de hardware específico, contendo o mínimo de recursos necessários para emulação de falhas deve ser feito em longo prazo. Otimizações apresentadas em (56) (57) suportam frequências maiores de operação do ICAP e podem ser agregadas no módulo a ser desenvolvido para acelerar ainda mais a emulação de falhas. O objetivo desse módulo é possibilitar de forma menos intrusiva analisar experimentalmente técnicas de tolerâncias a falha, e principalmente permitir a validação de projetos endurecidos visando FPGAs. Dessa forma, o módulo poderá complementar o esquema de BIST de um projeto tolerante a falhas agregando a funcionalidade de emulação de falhas.



## 9 CONCLUSÃO

No presente trabalho foi realizado um estudo abrangendo as técnicas de injeção de falhas existentes. A utilização de circuitos reconfiguráveis para a emulação de *soft errors* se mostrou bastante versátil e promissora comparada com outras técnicas de injeção de falhas. Contudo, o uso de FPGAs para emulação de falhas não dispensa o uso de outras técnicas de injeção de falhas para a validação de circuitos tolerantes a radiação, mas sim complementa o processo de análise.

Um fluxo para a emulação de falhas visando à análise experimental de técnicas de tolerância a falhas aplicadas a circuitos críticos foi apresentado nesse trabalho. Uma plataforma de emulação de *soft errors*, chamada PLAESER, foi desenvolvida para dar suporte a esse fluxo que busca permitir a prototipação rápida de um circuito crítico para análise da técnica de tolerância a falhas aplicada através da emulação de SETs. A plataforma PLAESER usa duas instâncias do circuito sendo analisado de forma que enquanto as falhas são injetadas em uma das instâncias a outra permanece livre de falhas para a comparação de seus resultados. O resultado dessa comparação determina se a falha injetada repercutiu em um erro ou foi mascarado pela técnica de tolerância a falhas. As falhas são injetadas utilizando o componente ICAP dos FPGAs da Xilinx que permite o acesso interno a memória de configuração do FPGA.

Os resultados obtidos da emulação de falhas no contador com redundância tripla apresentado em (55) mostram a eficiência da técnica de tolerância a falhas TMR. Versões modificadas do contador com TMR foram desenvolvidas com o objetivo de mostrar o funcionamento e integridade dos resultados gerados pela plataforma PLAESER. O fluxo proposto nesse trabalho foi aplicado a um caso de teste mais expressivo. A análise experimental do computador de bordo com monitor de barramento desenvolvido em (54) através de emulação de mais de cinco mil falhas se mostrou significativamente mais rápida do que com a injeção de falhas em 1090 sinais por meio de simulação. Além disso, os resultados da plataforma PLAESER mostraram um comportamento do circuito do OBC na presença de falhas quase que contrário ao obtido por simulação funcional. Espera-se que novos resultados sejam obtidos do OBC desenvolvido em (54) através de simulações temporais pós-implementação. Dessa forma, será possível realizar uma comparação mais realista perante os resultados obtidos com a aplicação da PLAESER para a análise do OBC.

A consolidação da plataforma PLAESER como uma ferramenta de análise experimental através de emulação de falhas, requer que seu desenvolvimento permaneça em andamento. O trabalho apresentado nessa dissertação pode servir de base para novas frentes de pesquisas que busquem tornar a plataforma PLAESER em uma ferramenta mais abrangente. Para isso, pesquisas relacionadas a modelos de falhas, formas de injeção de falhas menos intrusivas, recursos físicos do FPGA para emulação falhas, comparação de técnicas de injeção de falhas e principalmente a aplicação do fluxo da PLAESER em novos casos de teste servirão para o desenvolvimento de uma plataforma mais genérica de injeção de falhas nacional.

## REFERÊNCIAS

- (1) BENSO, A. e PAOLO, P. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. 1. ed. Boston: Kluwer Academic Publishers, 2004. v. 23p. 245
- (2) KASTENSMIDT, F.;; CARRO, L. e REIS, R. *Fault-Tolerance Techniques for SRAM-based FPGAs*. Boston, MA: Springer US, 2006. p. 198
- (3) NICOLAIDIS, M. *Soft Errors in Modern Electronic Systems*. Boston, MA: Springer US, 2011. v. 41p. 335
- (4) HSUEH, M.-C.;; TSAI, T. K. e IYER, R. K. Fault Injection Techniques and Tools. *Computer*, v. 30, n. 4, p. 75-82, doi:10.1109/2.585157, 1997.
- (5) ANTONI, L.;; LEVEUGLE, R. e FEHER, B. Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes. *Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, p. 405-413, doi:10.1109/DFTVS.2000.887181, 2000.
- (6) CIVERA, P. et al. Exploiting FPGA-based Techniques for Fault Injection Campaigns on VLSI Circuits. In: PROCEEDINGS 2001 IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS. *Anais...* [S.l.]: IEEE Comput. Soc, 2001.
- (7) SCHUMACHER, P. *WP414 - SEU Emulation Environment*. . [S.l.: s.n.], 2012.
- (8) MURAOKA, I. Development on Electronic Components for Brazilian Satellite Program. In: INTERNATIONAL SCHOOL ON THE EFFECTS OF RADIATION ON EMBEDDED SYSTEMS FOR SPACE APPLICATIONS - SERESSA. *Anais...* São José dos Campos, Brasil: [s.n.], 2010.
- (9) NORMAND, E. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, v. 43, n. 6, p. 2742-2750, doi:10.1109/23.556861, 1996.

- (10) AVIZIENIS, A. et al. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, v. 1, n. 1, p. 11-33, doi:10.1109/TDSC.2004.2, 2004.
- (11) BATTEZZATI, N.;; STERPONE, L. e VIOLANTE, M. *Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications*. New York, NY: Springer New York, 2011.
- (12) VELAZCO, R.;; FOUILLAT, P. e REIS, R. *Radiation Effects on Embedded Systems*. Dordrecht: Springer Netherlands, 2007.
- (13) MUKHERJEE, S. S. et al. Measuring architectural vulnerability factors. *Micro, IEEE*, v. 23, n. 6, p. 70-75, doi:10.1109/MM.2003.1261389, 2003.
- (14) HSIUNG, P.-A.;; SANTAMBROGIO, M. e HUANG, C.-H. *Reconfigurable System Design and Verification*. 1st. ed. Boca Raton, FL, USA: CRC Press, 2009. p. 268
- (15) XILINX. *Zynq-7000*. Disponível em: <[http://www.xilinx.com/publications/prod\\_mktg/zynq7000/Product-Brief.pdf](http://www.xilinx.com/publications/prod_mktg/zynq7000/Product-Brief.pdf)>. Acesso em: 23 maio. 2012.
- (16) ACTEL. *SmartFusion cSoC*. Disponível em: <[http://www.actel.com/documents/SmartFusion\\_DS.pdf](http://www.actel.com/documents/SmartFusion_DS.pdf)>. Acesso em: 23 maio. 2012.
- (17) ALTERA. *SoC FPGA*. Disponível em: <<http://www.altera.com/literature/br/br-soc-fpga.pdf>>. Acesso em: 23 maio. 2012.
- (18) ZEIDMAN, B. *Designing with FPGAs and CPLDs*. 1. ed. Lawrence, Kansas, USA: C M P Books, 2002. p. 220
- (19) GEORGE, J. et al. Single event upsets in xilinx virtex-4 FPGA devices. In: RADIATION EFFECTS DATA WORKSHOP, 2006 IEEE. *Anais...* [S.l.]: IEEE, 2006.
- (20) DRIMER, S. Security for volatile FPGAs. *University of Cambridge Thesis*, v. UCAM-CLTR, n. 763, p. 1-169, 2009.

(21) PARTICLE, A. e DEVICES, S. Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices. *JEDEC STANDARD*, v. JESD89A, p. 1-94, 2006.

(22) LESEA, A. et al. The Rosetta Experiment: Atmospheric Soft Error Rate Testing in Fiffering Technology FPGAs. *IEEE Transactions on Device and Materials Reliability*, v. 5, n. 3, p. 317-328, doi:10.1109/TDMR.2005.854207, 2005.

(23) MAKOWSKI, D. The impact of radiation on electronic devices with the special consideration of neutron and gamma radiation monitoring. *Technilcal University of Lodz*, p. 1-151, 2006.

(24) BOCQUILLON, a. et al. Highlights of laser testing capabilities regarding the understanding of SEE in SRAM based FPGAs. In: 2007 9TH EUROPEAN CONFERENCE ON RADIATION AND ITS EFFECTS ON COMPONENTS AND SYSTEMS. *Anais...* [S.I.]: IEEE, 2007.

(25) ZIADE, H.;; AYOUBI, R. e VELAZCO, R. A Survey on Fault Injection Techniques. *Int. Arab J. Inf. Technol.*, v. 1, n. 2, p. 171-186, doi:10.1.1.167.966, 2004.

(26) MUKHERJEE, S. *Architecture Design For Soft Errors*. 1. ed. Burlington, USA: Morgan Kaufmann, 2008. p. 360

(27) MADEIRA, H. et al. Experimental evaluation of a COTS system for space applications. In: PROCEEDINGS INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS. *Anais...* [S.I.]: IEEE Comput. Soc, 2002.

(28) SILVA, A. DA e SANCHEZ, S. LEON3 ViP: A Virtual Platform with Fault Injection Capabilities. *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, p. 813-816, doi:10.1109/DSD.2010.34, 2010.

(29) JENN, E. et al. Fault injection into VHDL models: the MEFISTO tool. *Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing*, v. 2, p. 66-75, doi:10.1109/FTCS.1994.315656, 1994.

(30) FOLKESSON, P.;; SVENSSON, S. e KARLSSON, J. A comparison of simulation based and scan chain implemented fault injection. *Digest of*

*Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*, p. 284-293, doi:10.1109/FTCS.1998.689479, 1998.

(31) FIDALGO, A. V.;; ALVES, G. R. e FERREIRA, J. M. Real Time Fault Injection Using a Modified Debugging Infrastructure. In: 12TH IEEE INTERNATIONAL ON-LINE TESTING SYMPOSIUM (IOLTS'06). *Anais...* [S.l.]: IEEE, 2006.

(32) CHENG, K.-ting;; HUANG, S.-yu e DAI, W.-jin. Fault emulation: a new approach to fault grading. *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, p. 681-686, doi:10.1109/ICCAD.1995.480203, 1995.

(33) LOPEZ-ONGIL, C. et al. Autonomous Fault Emulation: A New FPGA-Based Acceleration System for Hardness Evaluation. *IEEE Transactions on Nuclear Science*, v. 54, n. 1, p. 252-261, doi:10.1109/TNS.2006.889115, 2007.

(34) ENTRENA, L. et al. Soft Error Sensitivity Evaluation of Microprocessors by Multilevel Emulation-Based Fault Injection. *IEEE Transactions on Computers*, n. 99, p. 1-1, doi:10.1109/TC.2010.262, 2010.

(35) KAFKA, L.;; DANEK, M. e NOVAK, O. A Novel Emulation Technique that Preserves Circuit Structure and Timing. *2007 International Symposium on System-on-Chip*, p. 1-4, doi:10.1109/ISSOC.2007.4427437, 2007.

(36) VANHAUWAERT, P.;; LEVEUGLE, R. e ROCHE, P. Reduced Instrumentation and Optimized Fault Injection Control for Dependability Analysis. *2006 IFIP International Conference on Very Large Scale Integration*, p. 391-396, doi:10.1109/VLSISOC.2006.313220, 2006.

(37) ALDERIGHI, M. et al. Experimental validation of fault injection analyses by the FLIPPER tool. In: 2009 EUROPEAN CONFERENCE ON RADIATION AND ITS EFFECTS ON COMPONENTS AND SYSTEMS. *Anais...* [S.l.]: IEEE, 2009.

(38) ALDERIGHI, M. et al. Evaluation of Single Event Upset Mitigation Schemes for SRAM based FPGAs using the FLIPPER Fault Injection Platform. In: 22ND IEEE INTERNATIONAL SYMPOSIUM ON DEFECT

AND FAULT-TOLERANCE IN VLSI SYSTEMS (DFT 2007). *Anais...* [S.l.]: IEEE, 2007.

(39) AGUIRRE, M. et al. An FPGA based hardware emulator for the insertion and analysis of Single Event Upsets in VLSI Designs. In: RADIATION EFFECTS ON COMPONENTS AND SYSTEMS WORKSHOP (RADECS). *Anais...* [S.l.: s.n.], 2004.

(40) AGUIRRE, M. a. et al. A New Approach to Estimate the Effect of Single Event Transients in Complex Circuits. *IEEE Transactions on Nuclear Science*, v. 54, n. 4, p. 1018-1024, doi:10.1109/TNS.2007.895549, 2007.

(41) NAPOLES, J. et al. A Complete Emulation System for Single Event Effects Analysis. In: PROGRAMMABLE LOGIC, 2008 4TH SOUTHERN CONFERENCE ON. *Anais...* [S.l.]: IEEE, 2008.

(42) DUTTON, B. et al. Embedded Processor Based Fault Injection and SEU Emulation for FPGAs. In: PROC. INT. CONF. ON EMBEDDED SYSTEMS AND APPLICATIONS. *Anais...* [S.l.: s.n.], 2009.

(43) ZHANG, Q.;; ZHOU, J. e YU, X. A Kind of Low-cost Non-intrusive Autonomous Fault Emulation System. *Computer and Information Science*, v. 4, n. 1, p. 2549-2557, 2011.

(44) JEITLER, M.;; DELVAI, M. e REICHOR, S. FuSE - a hardware accelerated HDL fault injection tool. In: 2009 5TH SOUTHERN CONFERENCE ON PROGRAMMABLE LOGIC (SPL). *Anais...* [S.l.]: IEEE, 2009.

(45) XILINX. Virtex-5 Family Overview. *Xilinx Inc.*, v. 5.0, n. DS100, p. 1-13, 2009.

(46) XILINX. Virtex-5 FPGA Configuration User Guide. *Xilinx Inc.*, v. 3.10, n. UG191, p. 1-166, 2011.

(47) XILINX. Virtex-5 Libraries Guide for HDL Design. *Xilinx Inc.*, v. 13.3, n. UG621, p. 1-383, 2011.

- (48) CHAPMAN, K. SEU Strategies for Virtex-5 Devices. *Xilinx Inc.*, v. 2.0, n. XAPP864, p. 1-16, 2010.
- (49) CHAPMAN, K. New Generation Virtex-5 SEU Controller. *Xilinx Inc.*, v. A, n. 2, p. 1-46, 2010.
- (50) CHAPMAN, K. Virtex-5 SEU Critical Bit Information. *Xilinx Inc.*, v. 1, p. 1-67, 2010.
- (51) CIVERA, P. et al. An FPGA-based approach for speeding-up fault injection campaigns on safety-critical circuits. *Journal of Electronic Testing*, v. 18, n. 3, p. 261–271, 2002.
- (52) BEZERRA, E. A.;; ALMEIDA, G. M. e AZEVEDO, L. R. An adaptive communications module for on-board computers of satellites. In: 2010 NASA/ESA CONFERENCE ON ADAPTIVE HARDWARE AND SYSTEMS. *Anais...* [S.l.]: IEEE, 2010.
- (53) ALEX. FPGA Logic Cells Comparison. *1-CORE, Technologies*, p. 1-5, 2009.
- (54) SILVA, F. A. et al. Non-intrusive fault tolerance in soft processors through circuit duplication. In: LATIN-AMERICAN TEST WORKSHOP (LATW). *Anais...* [S.l.: s.n.], 2012.
- (55) CARMICHAEL, C. Application Note : Virtex Series Triple Module Redundancy Design Techniques for Virtex FPGAs TMR in FPGAs. *Xilinx Inc.*, v. 1.0.1, n. XAPP192, p. 1-37, 2006.
- (56) BONAMY, R. et al. UPaRC — Ultra-Fast Power-aware Reconfiguration Controller. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, n. March, p. 1373-1378, 2012.
- (57) HANSEN, S. G.;; KOCH, D. e TORRESEN, J. High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro. *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, p. 174-180, doi:10.1109/IPDPS.2011.139, 2011.



(58) MORGADO DE SOUZA, C. *VisuAlg 2.0 - Apoio Informática*. Disponível em: <[www.apoioinformatica.inf.br/o-visualg](http://www.apoioinformatica.inf.br/o-visualg)>. Acesso em: 28 maio. 2012.



## APÊNDICE A – (PSEUDOCÓDIGO)

O pseudocódigo do algoritmo de conversão das coordenadas XY para a forma de endereçamento e do cálculo do índice da palavra com os 16 bits iniciais da configuração da LUT foi desenvolvido utilizando-se a ferramenta VisuAlg que é disponibilizada gratuitamente pela empresa Apoio Informática (58). A ferramenta tem o objetivo de auxiliar o ensino de algoritmos de programação introduzindo o conceito de pseudocódigos que são escritos utilizando-se palavras-chaves intuitivas e em português sem acentuação. VisuAlg possui um editor de texto para a criação dos pseudocódigos e permite também a interpretação do algoritmo de forma que é possível executá-lo.

O pseudocódigo criado é apresentado na Figura 83 e foi executado para diferentes valores de x, y e lut\_n. Os resultados foram verificados através da comparação com os obtidos com o script em Perl desenvolvido e apresentado nesse trabalho. A ferramenta VisuAlg foi utilizada para representar o algoritmo desenvolvido através de um pseudocódigo independente de linguagem, mas que seu funcionamento pudesse ser verificado através da execução do algoritmo.

A interação com o algoritmo é realizada através dos comandos “escreva” e “leia”, que escreve na tela e lê do teclado, respectivamente. VisuAlg permite que os dados passados para o algoritmo sejam carregados de um arquivo, onde os dados estão dispostos na ordem que são requisitados a cada execução do comando “leia”. Esse arquivo foi utilizado como uma biblioteca, chamada de “fpgas.bib”, que contém as características da família e do modelo de FPGA utilizado. O conteúdo da biblioteca “fpgas.bib” é mostrado na Figura 84.

Figura 83 - Pseudocódigo do algoritmo executado no VisuAlg.

```

1 algoritmo "pseudocodigo"
2 // Autor : Frederico Ferlini
3 // Data : 30/05/2012
4
5 //*****//
6 //##### BIBLIOTECA DE FPGAS #####
7 arquivo "fpgas.bib"
8
9 //*****//
10 //##### VARIAVEIS GLOBAIS #####
11 var
12
13 //vetor com os indices dos diferentes tipos de colunas
14 iob_cols, brm_cols, dsp_cols, clk_cols, gtp_cols, clb_cols : caractere

```

```

15
16 //total de colunas no FPGA
17 total_cols : inteiro
18
19 //total de quadros das pilhas de cada tipo de coluna
20 piobs, pbrams, pdsp, pclk, pgtps, pcls : inteiro
21
22 //total de quadros no FPGA, em cada metade e em cada linha
23 total_quadros, quadros_metade, quadros_linha : inteiro
24
25 //total de linhas por metade
26 linhas_metade : inteiro
27
28 //total de clbs por coluna
29 clbs_coluna : inteiro
30
31 //tamanho das coordenadas x e y
32 tam_x, tam_y : inteiro
33
34 //total de palavras de configuracao no inicio do bitstream
35 palavras_ini_cfg : inteiro
36
37 //total de palavras por quadro
38 palavras_quadro : inteiro
39
40 //indices do quadro na pilha para slices com a coordenada x par ou impar
41 x_impar, x_par : inteiro
42
43 //numero de bits de deslocamento para os campos do enderecamento de quadro
44 des_metade, des_linha, des_coluna : inteiro
45
46 //*****//
47 //##### MACROS #####//
48
49 //procedimento para o calculo do numero de linhas por metade
50 procedimento CALC_LINHAS_METADE
51 Inicio
52 linhas_metade <- int((tam_y/clbs_coluna)/2) + (int(tam_y/clbs_coluna) % 2)
53 fimprocedimento
54
55 //procedimento para o calculo do vetor com os indices de colunas de CLBs
56 procedimento CALC_CLB_COLS
57 var
58 i, p : inteiro
59 n : caractere

```

```

60 inicio
61 para i de 0 ate (total_cols-1) faca
62   n <- (" "+numpcarac(i)+",")
63   p <- pos(n,(",","+iob_cols)) + pos(n,(",","+brm_cols)) + pos(n,(",","+dsp_cols))
64   p <- p + pos(n,(",","+clk_cols)) + pos(n,(",","+gtp_cols))
65   se p = 0 entao
66     clb_cols <- (clb_cols + numpcarac(i) + ",")
67   fimse
68 fimpara
69 fimprocedimento
70
71 //*****//
72 //##### FUNCOES #####//
73
74 //Converte numero decimal para hex (= vetor de caracteres)
75 funcao dectohex(dec : inteiro) : caractere
76 var
77 hex : caractere
78 tmp : inteiro
79 inicio
80 enquanto dec > 0 faca
81   tmp <- dec mod 16
82   se tmp < 10 entao
83     hex <- numpcarac(tmp) + hex
84   senao
85     hex <- carac(tmp + 55) + hex
86   fimse
87   dec <- dec div 16
88 fimenquanto
89 retorne hex
90 fimfuncao
91
92 //Carrega da biblioteca as caracteristicas da familia de FPGAs
93 funcao carrega_familia(familia : caractere) : logico
94 var
95 linha : caractere
96 resultado : logico
97 inicio
98 resultado <- falso
99 repita
100   leia(linha)
101   se linha=familia entao
102     leia(des_metade)
103     leia(des_linha)
104     leia(des_coluna)

```

```
105 leia(palavras_quadro)
106 leia(palavras_ini_cfg)
107 leia(clbs_coluna)
108 leia(x_impair)
109 leia(x_par)
110 leia(piobs)
111 leia(pbrams)
112 leia(pdsp)
113 leia(pclks)
114 leia(pgtps)
115 leia(pclbs)
116 resultado <- verdadeiro
117 interrompa
118 fimse
119 ate linha="
120 retorne resultado
121 fimfuncao
122
123 //Carrega da biblioteca as características do modelo de FPGA escolhido
124 funcao carrega_fpga(fpga : caractere) : logico
125 var
126 linha : caractere
127 resultado : logico
128 inicio
129 resultado <- falso
130 repita
131 leia(linha)
132 se linha=fpga entao
133 leia(total_quadros)
134 leia(tam_x)
135 leia(tam_y)
136 CALC_LINHAS_METADE
137 quadros_metade <- int(total_quadros / 2)
138 quadros_linha <- int(quadros_metade / linhas_metade)
139 leia(iob_cols)
140 leia(brm_cols)
141 leia(dsp_cols)
142 leia(clk_cols)
143 leia(gtp_cols)
144 leia(total_cols)
145 CALC_CLB_COLS
146 resultado <- verdadeiro
147 interrompa
148 fimse
149 ate linha="EOF" 150 retorne resultado
```

```
151 fimfuncao
152
153 //Calcula o numero de colunas anteriores ao indice "col" informado
154 funcao calc_cols_antes(vet_cols : caractere
155 var
156 str: caractere
157 i, m : inteiro
158 inicio
159 para i de 1 ate compr(vet_cols) faca
160     str <- str + copia(vet_cols,i,1)
161     se copia(str,compr(str),1)="," entao
162         se col > caracpnum(copia(str,1,(compr(str)-1))) entao
163             m <- m + 1
164         fimse
165     str <- ""
166     fimse
167 fimpara
168 retorne m
169 fimfuncao
170
171 //Retorna o numero da coluna da posicao "idx" do vetor "vet_col"
172 funcao pega_col(vet_cols : caractere
173 var
174 str: caractere
175 cnt, i : inteiro
176 inicio
177 para i de 1 ate compr(vet_cols) faca
178     se copia(vet_cols,i,1)="," entao
179         cnt <- cnt + 1
180     se cnt > idx entao
181         interrompa
182     fimse
183     str <- ""
184     senao
185         str <- str + copia(vet_cols,i,1)
186     fimse
187 fimpara
188 retorne caracpnum(str)
189 fimfuncao
190
191 //Calcula o indice da palavra no quadro que contem os bits da "lut"
192 funcao calc_palavra_idx(y, lut : inteiro) : inteiro
193 var
194 palavra_idx : inteiro
195 inicio
```

```
196 palavra_idx <- ((y - int(y/clbs_coluna)*clbs_coluna) * 2)
197 se palavra_idx >= clbs_coluna entao
198   palavra_idx <- palavra_idx + 1
199 fimse
200 se lut >= 2 entao
201   palavra_idx <- palavra_idx + 1
202 fimse
203 retorne palavra_idx
204 fimfuncao
205
206 //Calcula o valor do campo [metade] da palavra de enderecamento
207 funcao calc_metade(y : inteiro) : inteiro
208 var r : inteiro
209 inicio
210   se (y < (tam_y/2)) entao
211     r <- 1
212   fimse
213   retorne r
214 fimfuncao
215
216 //Calcula o valor do campo [linha] da palavra de enderecamento
217 funcao calc_linha(y : inteiro) : inteiro
218 var r,i,met : inteiro
219 inicio
220   para i de 1 ate (linhas_metade-1) faca
221     se (y < (i * clbs_coluna)) ou (y >= (tam_y - (i * clbs_coluna))) entao
222       r <- (linhas_metade - i)
223       interrompa
224   fimse
225   fimpara 226   retorne r
227 fimfuncao
228
229 //Calcula o valor do campo [coluna] da palavra de enderecamento
230 funcao calc_coluna(x : inteiro) : inteiro
231 inicio
232   retorne pega_col(clb_cols,int(x/2))
233 fimfuncao
234
235 //Calcula o valor do campo [quadro] da palavra de enderecamento
236 funcao calc_quadro(x : inteiro) : inteiro
237 var r : inteiro
238 inicio
239   r <- x_par
240   se (x % 2) = 1 entao
241     r <- x_imp
```



```

242 fimse
243 retorne r
244 fimfuncao
245
246 //Calcula a palavra de endereçamento
247 funcao calc_faddr(x, y : inteiro) : inteiro
248 var faddr : inteiro
249 inicio
250 faddr <- int(calc_metade(y) * (2^des_metade))
251 faddr <- faddr + int(calc_linha(y) * (2^des_linha))
252 faddr <- faddr + int(calc_coluna(x) * int(2^des_coluna))
253 faddr <- faddr + calc_quadro(x)
254 retorne faddr
255 fimfuncao
256
257 //Calcula o indice da palavra no bitstream com os primeiros 16bits da lut
258 funcao calc_bitstream_idx(x, y, lut : inteiro) : inteiro
259 var col, idx : inteiro
260 inicio
261 idx <- int(calc_metade(y) * quadros_metade)
262 idx <- idx + int(calc_linha(y) * quadros_linha)
263 col <- calc_coluna(x)
264 idx <- idx + int(calc_cols_antes(iob_cols,col) * piobs)
265 idx <- idx + int(calc_cols_antes(brm_cols,col) * pbrams)
266 idx <- idx + int(calc_cols_antes(dsp_cols,col) * pdsp)
267 idx <- idx + int(calc_cols_antes(clk_cols,col) * pclk)
268 idx <- idx + int(calc_cols_antes(gtp_cols,col) * pgtps)
269 idx <- idx + int(calc_cols_antes(clb_cols,col) * pclbs)
270 idx <- idx + calc_quadro(x)
271 idx <- idx * palavras_quadro
272 idx <- idx + calc_palavra_idx(y,lut)
273 idx <- idx + palavras_ini_cfg
274 retorne idx
275 fimfuncao
276
277 //*****//
278 //##### PARAMETROS DO ALGORITMO #####//
279
280 //familia e modelo do fpga
281 familia, fpga : caractere
282 //coordendas xy e o numero da lut do slice
283 coord_x, coord_y, lut_n : inteiro
284
285 //*****//
286 //##### EXEMPLO #####//

```

```

287 inicio
288
289 //define parametros (familia, fpga, coordenadas xy e a lut)
290 familia <- "VIRTEX5"
291 fpga <- "XC5VLX110T"
292 coord_x <- 0
293 coord_y <- 0
294 lut_n <- 0
295
296 //verifica se o fpga esta na biblioteca e carrega as caracteristicas
297 se carrega_familia(familia) e carrega_fpga(fpga) entao
298 //calcula escreve a palavra de enderecamento
299 escreval("faddr = ",dectohex(calc_faddr(coord_x,coord_y)))
300 //calcula o indice da palavra no bitstream com os 16bits iniciais da LUT
301 escreval("bit_idx = ",calc_bitstream_idx(coord_x,coord_y,lut_n))
302 senao
303 //informa se FPGA/Familia nao for encontrada na biblioteca
304 escreva("FPGA/Familia nao encontrada!")
305 fimse
306
307 fimalgoritmo

```

Figura 84 - A biblioteca descrita no arquivo "fpga.bib".

```

1  CONSTANTES_FAMILIA
2  VIRTEX5
3  20
4  15
5  7
6  41
7  59
8  20
9  26
10 32
11 54
12 30
13 28
14 4
15 32
16 36
17
18 CONSTANTES_MODELO
19 XC5VLX110T
20 18576
21 108
22 160

```

```
23 0,28,58,  
24 5,16,42,53,  
25 19,  
26 29,  
27 64,  
28 65  
29  
30 XC5VLX30T  
31 18576  
32 60  
33 80  
34 0,17,32,  
35 5,27,37,  
36 8,  
37 18,  
38 38,  
39 65  
40 EOF
```