Victor Manuel Gonçalves Martins

Instrumentação de FPGAs SRAM para Recuperação e Prevenção de Faltas Permanentes Visando Utilização em Aplicações Espaciais

Florianópolis

Maio de 2016

Victor Manuel Gonçalves Martins

INSTRUMENTAÇÃO DE FPGAS SRAM PARA RECUPERAÇÃO E PREVENÇÃO DE FALTAS PERMANENTES VISANDO UTILIZAÇÃO EM APLICAÇÕES ESPACIAIS

Tese submetida ao Programa de Pós-Graduação em Engenharia Elétrica, na área de concentração de Circuitos e Sistemas Integrados, da Universidade Federal de Santa Catarina para a obtenção do Grau de Doutor em Engenharia Elétrica.
Orientador: Prof. Dr. Eduardo Augusto Bezerra

Florianópolis
Maio de 2016

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

Martins, Victor Manuel Gonçalves
Instrumentação de FPGAs SRAM para Recuperação e Prevenção de Faltas Permanentes Visando Utilização em Aplicações Espaciais / Victor Manuel Gonçalves Martins ; orientador, Eduardo Augusto Bezerra - Florianópolis, SC, 2016.
310 p.

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

Inclui referências

1. Engenharia Elétrica. 2. Recuperação de Faltas Permanentes. 3. Prevenção de Faltas Permanentes. 4. Envelhecimento devido ao NBTI e PBTI. 5. Uso de FPGAs em Aplicações Espaciais. I. Bezerra, Eduardo Augusto. II. Universidade Federal de Santa Catarina. Programa de Pós Graduação em Engenharia Elétrica. III. Título.

Victor Manuel Gonçalves Martins

INSTRUMENTAÇÃO DE FPGAS SRAM PARA RECUPERAÇÃO E PREVENÇÃO DE FALTAS PERMANENTES VISANDO UTILIZAÇÃO EM APLICAÇÕES ESPACIAIS

Esta Tese foi julgada adequada para obtenção do Título de "Doutor", e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica.

Florianópolis, 25 de Maio de 2016.

____
Prof. Carlos Galup Montoro, Dr.
Coordenador do Curso

Banca Examinadora:

____
Prof. Eduardo Augusto Bezerra, Dr.
Orientador
Universidade Federal de Santa Catarina - UFSC

____

Prof. Horárico Cláudio de Campos Neto, Dr.
Universidade de Lisboa – IST/UL (Videoconferência)

____

Prof. Fabian Luis Vargas, Dr.
Pontifícia Universidade Católica do RS – PUCRS (Videoconferência)

____

Prof. Jorge Luis Victoria Barbosa, Dr.
Univ. do Vale do Rio dos Sinos – UNISINOS (Videoconferência)

____

Prof. Jorge Filipe Leal Costa Semião, Dr.
Universidade do Algarve – UAlg (Videoconferência)

____

Prof. Héctor Pettenghi Roldán, Dr.
Universidade Federal de Santa Catarina - UFSC

Este trabalho é dedicado a todos os que se dedicam (ou em algum momento se dedicaram) a mim, e ao nitrogénio, fonte principal de inspiração.

AGRADECIMENTOS

Peço desculpas por ser mal agradecido, mas não vou agradecer a ninguém. Vou apenas agradecer à sorte. À sorte por ter os pais que tenho. À sorte de mesmo acordado sentir um sonho. À sorte de ter bons amigos. À sorte de interagir com professores que também são pessoas. À sorte de ter sido orientado por pessoas que também são professores. À sorte de ter colegas que sabem mais que eu mas que me acham inteligente como eles. À sorte de poder ir conhecendo pessoas. À sorte de conseguir entender uma boa parte das pessoas que vou conhecendo. À sorte do tempo me ir dando tempo para observar este tempo. E por fim, para ser incoerente com o início, um nome: agradeço a Miguel Torga, um poeta que bem soube descrever toda a sorte que foi (e é) esta aventura:

"Aparelhei o barco da ilusão
E reforcei a fé de marinheiro.
Era longe o meu sonho, e traiçoeiro
O mar...
(Só nos é concedida
Esta vida
Que temos;
E é nela que é preciso
Procurar
O velho paraíso
Que perdemos).
Prestes, larguei a vela
E disse adeus ao cais, à paz tolhida.
Desmedida,
A revolta imensidão
Transforma dia a dia a embarcação
Numa errante e alada sepultura...
Mas corto as ondas sem desanimar.
Em qualquer aventura,
O que importa é partir, não é chegar."

RESUMO

Os dispositivos reprogramáveis *Field Programmable Gate Arrays* (FPGAs), embora construídos para serem robustos, não são eternos, nem completamente imunes à ocorrência de faltas, sejam elas transitórias ou permanentes. Considerando que o teste após fabricação deteta todas as faltas devidas ao processo de produção, em condições normais ao nível do mar, mesmo com as tecnologias nanométricas recentes, a ocorrência de faltas permanentes numa FPGA durante o seu previsível ciclo de vida é praticamente nula. Já em condições hostis, como no espaço onde o nível de radiação é elevado (ou mesmo ambientes terrestres como centrais nucleares, centros de investigação de física nuclear, aceleradores de partículas, etc.), a ocorrência de faltas permanentes numa FPGA não pode ser desprezada. Para além da radiação, sendo um dispositivo eletrónico, está igualmente sujeito a envelhecimento (*aging*). O *Negative Bias Temperature Instability* (NBTI) e o *Positive Bias Temperature Instability* (PBTI) são dois dos fatores que provocam esse envelhecimento, e que embora não destruam a funcionalidade dos recursos da FPGA, aumentam os seus tempos de propagação. Este envelhecimento pode por isso também originar faltas permanentes a partir de um determinado ponto do ciclo de vida do sistema implementado numa FPGA. A solução para esses casos é a substituição da FPGA ou até mesmo da placa que inclui a mesma. Apesar do facto de que em muitas situações a substituição da FPGA ser considerada uma tarefa simples, em tantas outras, tais como ambientes aeroespaciais onde o acesso é difícil e/ou perigoso para quem tem de realizar a substituição, esta operação poderá ser problemática ou impossível de realizar.

Neste contexto, esta tese propõe o desenvolvimento de soluções, para que um sistema implementado numa FPGA possa autonomamente recuperar da ocorrência de faltas permanentes (evitando utilizar recursos do dispositivo que sofreram essas mesmas faltas), e ao mesmo tempo, atenuar o ritmo de envelhecimento do dispositivo devido ao NBTI (e eventualmente também ao PBTI). Para isso, este trabalho foca em dois objetivos principais: (1) O desenvolvimento de um mecanismo em

hardware, baseado na Reconfiguração Parcial da FPGA, que suporte a implementação de estratégias de recuperação e prevenção de faltas permanentes (minimizando a evolução do envelhecimento causado pelo NBTI). (2) Planear e implementar formas de recuperar ou prevenir da ocorrência de faltas permanentes (*delay faults*), recorrendo ao mecanismo desenvolvido.
O mecanismo apresentado passa por novo fluxo gerador de *bitstreams* parciais, possíveis de realocar em múltiplas partições reconfiguráveis, uma flexibilidade que ultrapassa a proporcionada pelas ferramentas de reconfiguração dinâmica disponibilizadas pelo fabricante. Das estratégias implementadas, uma permite um sistema implementado numa FPGA recuperar de uma falta permanente, sem necessidade de excluir toda a partição. Para atenuação do envelhecimento do dispositivo, outra estratégia altera as partições onde os *bitstreams* se encontram alocados de uma forma cíclica, de forma a que o máximo de recursos dessas partições não estejam configurados da mesma forma um longo período de tempo. É proposto ainda um novo sensor de performance para FPGA e que pode permitir medir também o envelhecimento em cada partição. Com ele é possível a estratégia de alocar módulos (existentes nos *bitstreams* gerados), de modo a uniformizar o envelhecimento e a dissipação de potência pelas várias partições, em função do envelhecimento acumulado, da temperatura atual e da potência consumida por cada módulo.

Palavras-chave: Recuperação de Faltas Permanentes, Prevenção de Faltas Permanentes, Envelhecimento devido ao NBTI e PBTI, FPGAs, SEE.

ABSTRACT

FPGA devices although built to be robust, are not everlasting. They are not completely invulnerable to the occurrence of faults, whether temporary or permanent. Whereas the test after manufacturing detects all faults due to production process, in normal conditions, at sea level, even with the recent nanometric technologies, the manifestation of permanent faults in FPGAs during their expected life cycle is considered to be near zero. However, in hostile conditions, such as in space where radiation levels are higher (or terrestrial environments such as nuclear power plants, nuclear physics research centers, particle accelerators, etc.), the rate of permanent faults in an FPGA device can not be neglected. In addition to the radiation, as the FPGA is an electronic device, it is also susceptible to aging effects. NBTI and PBTI are two of the aging sources and, although they do not damage directly the functionality of the FPGA resources, they are responsible for the increase in the device's propagation times. This aging can therefore also lead to permanent faults in a certain moment in the life cycle of a system implemented on an FPGA. The solution for such cases is to replace the FPGA or even the board where it is on. Despite the fact that in many cases replacing the FPGA can be considered a simple task, in many others, such as in aerospace environments where the access is difficult and / or dangerous for those who have to do the replacement, this operation may be challenging or even impossible to perform.

In this context, this work proposes the development of solutions for a system implemented in an FPGA which can autonomously recover from the manifestation of permanent faults (avoiding use device resources that have suffered these same faults), and at the same time mitigating the rate of aging of the device due to NBTI (and possibly also the PBTI). Therefore, this work focuses on two main objectives: (1) The development of a mechanism in hardware, based on the FPGA Partial Reconfiguration mechanism, which supports the implementation of strategies for recovering and prevention of permanent faults (minimizing the evolution of aging caused by NBTI). (2) The planning

and implementation of ways to recover or to prevent the occurrence of permanent faults (delay faults), using the developed mechanism.
The presented mechanism includes a new flow to generate partial bitstreams, which allows to reallocate multiple reconfigurable partitions. This is a feature does not provided by the dynamic reconfiguration tools delivered by the manufacturers. The implemented strategies allow a system implemented in an FPGA to recover from a permanent fault, with no need to exclude the entire partition. For the device aging mitigation, another strategy changes the partitions where the bitstreams are allocated in a cyclical way, so that the maximum resources of these partitions are not configured in the same way for a long period of time. It is further proposed a new performance sensor for FPGA systems, which also may allow the measuting of aging in each partition. With this sensor the strategy allows the allocation of modules (existing in the generated bitstreams) in order to standardize the aging and power dissipation by the various partitions, as a function of cumulative aging, the current temperature and the power consumed by each module.

Key-words: Permanent Faults Recovery, Permanent Faults Prevention, Aging due to NBTI and PBTI, FPGAs, SEE.

LISTA DE ILUSTRAÇÕES

Figura 1 – Curva da Banheira para tempos de falhas. Modificado de (Klutke et al., 2003). 38
Figura 2 – Representação básica de uma *Look Up Table* (LUT) com três entradas com uma tabela verdade que implementa a função XOR (Hauck and Dehon, 2008). . 52
Figura 3 – Esquema básico de um *slice* com uma LUT de quatro entradas (Hauck and Dehon, 2008). 53
Figura 4 – Representação geral da arquitetura de uma FPGA (Hauck and Dehon, 2008). 54
Figura 5 – Arquitetura de conexão entre os blocos lógicos (Hauck and Dehon, 2008). 55
Figura 6 – Roteamento hierárquico usado para interconexão de grupos de blocos lógicos (Hauck and Dehon, 2008). . 56
Figura 7 – Constituição de uma FPGA Spartan-6. 57
Figura 8 – Composição de uma LUT de seis entradas (extraído do Apêndice A). 59
Figura 9 – Diagrama de um *slice* do tipo SLICEM (extraído do Apêndice A). 60
Figura 10 – Organização dos *slices* dentro de um *Configurable Logic Block* (CLB) (Xilinx Inc., 2010) (Xilinx Inc., 2012c) (Xilinx Inc., 2012d) (Xilinx Inc., 2013a). . . . 61
Figura 11 – Organização dos *slices* nas linhas e colunas de CLBs (Xilinx Inc., 2010) (Xilinx Inc., 2012c) (Xilinx Inc., 2012d) (Xilinx Inc., 2013a). 61
Figura 12 – Organização da memória de configuração ao longo de uma linha (*row*). 63
Figura 13 – Organização da memória de configuração de uma FPGA. 64
Figura 14 – Descrição do registo de endereçamento dos *frames* (*Frame Address Register* (FAR)) (Xilinx Inc., 2015a). 65

Figura 15 – Espaço de endereçamento para *MultiBoot* de uma memória flash *Byte Peripheral Interface* (BPI) (Xilinx Inc., 2015c). 68
Figura 16 – Diagrama básico de reconfiguração parcial (*Partial Reconfiguration* (PR)) (Xilinx Inc., 2013e). 69
Figura 17 – Tipos de *Single-Event Effects* (SEEs) que podem ocorrer numa FPGA (White, 2012). 73
Figura 18 – Resultados experimentais que mostram a relação entre a radiação *Total Ionizing Dose* (TID) e o atraso (Benfica et al., 2011). 81
Figura 19 – Componentes Temporária e Permanente do NBTI (Mishra et al., 2012). 83
Figura 20 – Exemplo de uma implementação usando o fluxo *Isolation Design Flow* (IDF) (Xilinx Inc., 2016c). 92
Figura 21 – Metodologia de teste proposta (Gericota, 2003). . . 95
Figura 22 – Representação da disposição inicial dos Módulos (a) e após realocação dos Módulos *Function C* e *Function D* nas colunas quatro e cinco, respetivamente (b) (Montminy et al., 2007). 97
Figura 23 – Os três modos de interligação entre Módulos (Montminy et al., 2007). 98
Figura 24 – Aceleração de uma Tarefa em Hardware (Iturbe et al., 2011). 101
Figura 25 – Comunicação entre Tarefas via *Internal Configuration Access Port* (ICAP) (Iturbe et al., 2011). 101
Figura 26 – Arquitetura do Sistema Autónomo Tolerante a Faltas (Bolchini et al., 2012). 105
Figura 27 – Sistema com Capacidade de Realocação de *Bitstreams* (Ochoa-Ruiz et al., 2013). 107
Figura 28 – Diagrama Global da Arquitetura TMR-MDR (Espinosa et al., 2012). 109
Figura 29 – Estrutura do *Configurable Transistor* (CT) na Arquitetura PAnDA-Zwei (Lawson et al., 2014). 112

Figura 30 – Estrutura do *Slice* na Arquitetura PAnDA-Zwei (Lawson et al., 2014). 113
Figura 31 – Micro-arquitetura de um *Virtual Hardware Component* (VHC) (Kirischian et al., 2004). 116
Figura 32 – Arquitetura com quatro *slots* para VHCs e infraestrutura de comunicação (Kirischian et al., 2010). 118
Figura 33 – Estrutura do VHC com opção de modo (Dumitriu and Kirischian, 2010). 119
Figura 34 – Arquitetura geral um *Embedded Reconfigurable System* (ERS) com *slots* para VHCs (Dumitriu and Kirischian, 2010). 120
Figura 35 – Arquitetura geral da VHC (Dumitriu et al., 2012). . 122
Figura 36 – Estrutura Geral do Sistema Colaborativo (Dumitriu and Kirischian, 2013). 123
Figura 37 – Arquitetura de um *Collaborative Macro-Functional Unit* (CMFU) (Dumitriu and Kirischian, 2013). . . . 124
Figura 38 – Arquitetura Geral do Sistema implementado numa FPGA (Dumitriu et al., 2014). 126
Figura 39 – Alternativas de Realocação de Módulos na FPGA (Dumitriu et al., 2015a). 128
Figura 40 – Controlo da Disposição dos Recursos Internos de um *Slot* (Dumitriu et al., 2015a). 129
Figura 41 – Arquitetura Geral do *Multimodal Adaptive Collaborative Reconfigurable self-Organized System* (MACROS) (Dumitriu et al., 2015b). 130
Figura 42 – Exemplo de um circuito implementado em apenas 1/4 da FPGA. 142
Figura 43 – Probabilidades $\mathbb{P}(B)$ e $\mathbb{P}(C)$ em função de n. 143
Figura 44 – Gráfico da probabilidade $\mathbb{P}(X \geq n)$. 145
Figura 45 – Exemplo de quatro circuitos implementados em 3/4 da FPGA. 147
Figura 46 – Probabilidades $\mathbb{P}(D)$ e $\mathbb{P}(E)$ para diferentes números de regiões (r). 148

Figura 47 – Ilustração das fases de stress e recuperação (*recovery*) do NBTI (Palermo et al., 2015). 152
Figura 48 – Estrutura geral da implementação da solução desenvolvida uma FPGA. 153
Figura 49 – Diagrama geral da solução desenvolvida. 154
Figura 50 – Fluxograma geral da aplicação desenvolvida. 156
Figura 51 – Comparação entre *Built-In Self-Test* (BIST) "tradicional" e o proposto (Martins et al., 2014a). 167
Figura 52 – Fluxo de implementação do detetor de faltas (Martins et al., 2014a). 168
Figura 53 – Regras de Roteamento 178
Figura 54 – Fluxos FPGA Development e *Assisted Design Flow* (ADF) (Martins et al., 2015a) 183
Figura 55 – The Buffer Options 184
Figura 56 – Buffers Window Example 186
Figura 57 – Distribuição de Recursos nas *Reconfigurable Partitions* (RPs) 195
Figura 58 – Diagrama do *Delay Meter* (DM). 201
Figura 59 – Primitiva *CARRY4* da família Virtex-6 (Xilinx Inc., 2013g). 202
Figura 60 – Distribuição de Recursos nas RPs com *Differential Delay Sensors* (DDSs). 204
Figura 61 – Análise do NBTI do circuito equivalente à Primitiva *CARRY4*. 205
Figura 62 – Gráfico do valor lido pelo DM em função da temperatura. 207
Figura 63 – Fluxograma geral da aplicação desenvolvida com *Triple Modular Redundancy* (TMR) e rotação de recursos. 212
Figura 64 – Diagrama do sistema implementado para validação do BIST com *scan chain* virtual. 219
Figura 65 – Diagrama do Sistema (Martins et al., 2015a). 223
Figura 66 – Diagrama do Sistema (Martins et al., 2015d). 229

Figura 67 – Diagrama do Sistema com DDSs e primitiva *System Monitor*. 237
Figura 68 – Caracterização de um DM na Virtex-6 utilizada. . . 241
Figura 69 – Análise do ruído/erro na leitura do DDS. 243
Figura 70 – Influência da temperatura na corrente I_{CCINT}. . . . 244
Figura 71 – Influência da temperatura na tensão de alimentação V_{CCINT}. 244
Figura 72 – Influência da temperatura na potência consumida $V_{CCINT}.I_{CCINT}$. 245
Figura 73 – Medição da temperatura recorrendo ao *golden* DM existente no DDS. 246
Figura 74 – Conversão dos valores do *golden* DM em temperatura. 247
Figura 75 – Fluxo normal da ferramentas influenciado através de *constraints*. 251
Figura 76 – Constituição de uma FPGA Spartan-6. 278
Figura 77 – Exemplo dos dois tipos de colunas de CLBs. 279
Figura 78 – Diagrama de um slice do tipo SLICEM. 280
Figura 79 – Composição de uma LUT de seis entradas. 280
Figura 80 – Sequência de configuração da Spartan-6 (Xilinx Inc., 2015b). 281
Figura 81 – Vista interna da Spartan-6 XC6SLX45T (Xilinx Inc., 2010). 284
Figura 82 – Registos de endereçamento dos *frames* (Xilinx Inc., 2015b). 285
Figura 83 – Organização de um *bit file* completo. 286
Figura 84 – Distribuição dos *frames* ao longo da primeira linha (ROW 0). 287
Figura 85 – Distribuição dos *frames* numa coluna com *slices* SLICEL e SLICEX. 288
Figura 86 – Distribuição dos *frames* numa coluna com *slices* SLICEM e SLICEX. 289
Figura 87 – Distribuição das palavras de configuração das LUTs num *slice* SLICEX. 290

Figura 88 – Distribuição das palavras de configuração das LUTs num *slice* SLICEL. 291
Figura 89 – Distribuição das palavras de configuração das LUTs num *slice* SLICEM. 292
Figura 90 – Ordem dos bits dos vários tipos de organização. . . . 293
Figura 91 – Organização dos bits do *frame* 25 em colunas de CLBs com SLICEL e SLICEX. 294
Figura 92 – Organização dos bits do *frame* 20 em colunas de CLBs com SLICEL e SLICEX. 295
Figura 93 – Organização dos bits do *frame* 26 em colunas de CLBs com SLICEM e SLICEX. 296
Figura 94 – Organização dos bits do *frame* 20 em colunas de CLBs com SLICEM e SLICEX. 297
Figura 95 – Organização dos bits do *frame* 23 em colunas de CLBs com SLICEM e SLICEX. 299
Figura 96 – Diagrama de um SLICEX (Xilinx Inc., 2010). 300
Figura 97 – Diagrama de um SLICEL (Xilinx Inc., 2010). 301
Figura 98 – Diagrama de um SLICEM (Xilinx Inc., 2010). . . . 302
Figura 99 – Distribuição dos *frames* numa coluna de BRAMs. . 303
Figura 100 – Distribuição das palavras de configuração das BRAMs. 304
Figura 101 – Distribuição geral dos bits num *frame* de configuração de RAMB8s. 305
Figura 102 – Organização dos bits da RAMB8X0Y6 no *frame* 22 em colunas de BRAMs. 306
Figura 103 – Bits de configuração da RAMB8X0Y7 e RAMB8X0Y6 no *frame* 23. 307
Figura 104 – Bits de configuração da RAMB8X0Y7 no *frame* 24. 308
Figura 105 – Bits de configuração da RAMB8X0Y6 no *frame* 24. 309
Figura 106 – Organização do conteúdo das BRAMs no *bit file*. . . 310
Figura 107 – Diferença entre a família Spartan-6 e outras famílias em relação ao carregamento do valor inicial dos flip-flops. 311

LISTA DE TABELAS

Tabela 1 – Recursos existentes num CLB nas várias famílias de FPGAs (Xilinx Inc., 2010) (Xilinx Inc., 2012c) (Xilinx Inc., 2012d) (Xilinx Inc., 2013a) 62
Tabela 2 – Características da organização de configuração em várias famílias de FPGAs (Xilinx Inc., 2015b) (Xilinx Inc., 2012b) (Xilinx Inc., 2015c) (Xilinx Inc., 2015a) 63
Tabela 3 – Comparativo dos trabalhos relativos a Realocação de Módulos e Plataformas com Tolerância a Faltas Permanentes 133
Tabela 4 – Tabela da probabilidade $\mathbb{P}(X \geq n)$. 145
Tabela 5 – Rotinas em software com valores individuais de Memória e Tempos de Execução 172
Tabela 6 – Características do Hardware 172
Tabela 7 – Rotinas em Software com Valores Individuais de Memória e Tempos de Execução 190
Tabela 8 – Características das Partições 191
Tabela 9 – Rotinas em Software com Valores Individuais de Memória e Tempos de Execução 209
Tabela 10 – Características das Partições com DDS incluído . . . 209
Tabela 11 – Comparativo do presente trabalho com os trabalhos relativos a Realocação de Módulos e Plataformas com Tolerância a Faltas Permanentes 215
Tabela 12 – Parâmetros dos quatro módulos testados 220
Tabela 13 – Memória alocada nos quatro módulos testados . . . 220
Tabela 14 – Número de Ciclos de Relógio por cada Vetor de Teste 221
Tabela 15 – Características de cada RP 226
Tabela 16 – Memória e Número de Ciclos de Relógio e Tempo para troca entre RPs 227
Tabela 17 – Características de cada RP 232
Tabela 18 – Memória e Número de Ciclos de Relógio e Tempo para troca entre RPs 233

Tabela 19 – Interferência da lista de *Constraints* nos *Bitstreams* Parciais 234
Tabela 20 – Características de cada RP 238
Tabela 21 – Memória e Número de Ciclos de Relógio consumidos pelas rotinas usadas. 240
Tabela 22 – Configurações dos multiplexadores OUTMUX nos *slices* SLICEL e SLICEM. 297
Tabela 23 – Configurações dos multiplexadores FFMUX nos *slices* SLICEL e SLICEM. 298
Tabela 24 – Configurações dos multiplexadores OUTMUX nos *slices* SLICEX. 298
Tabela 25 – Configurações dos multiplexadores FFMUX nos *slices* SLICEX. 299

LISTA DE ABREVIATURAS E SIGLAS

ADF *Assisted Design Flow*

AES *Advanced Encryption Standard*

AFDIR *Advanced Fault Detection, Isolation and Recovery*

ALU *Arithmetic Logic Unit*

ASIC *Application Specific Integrated Circuit*

ASP *Application Specific Processor*

ASVP *Application Specific Virtual Processor*

BCM *Bit-Stream and Configuration Manager*

BISR *Built-In Self-Recovery*

BIST *Built-In Self-Test*

BPI *Byte Peripheral Interface*

BPSG *Borophospho-Silicate Glass*

BS *Boundary Scan*

BTI *Bias Temperature Instability*

BUFGCTRL *Global Clock MUX Buffer*

CB *Connection Block*

CIF *Communication Interface*

CLB *Configurable Logic Block*

CMFU *Collaborative Macro-Functional Unit*

CMOS *Complementary Metal Oxide Semiconductor*

CPU *Central Processor Unit*

CRC *Cyclic Redundancy Check*

CT *Configurable Transistor*

DCCI *Distributed Communication and Control Interface*

DCM *Digital Clock Manager*

DCT *Discrete Cosine Transform*

DDS *Differential Delay Sensor*

DM *Delay Meter*

DPR *Dynamic Partially Reconfigurable*

DPRC *Dynamically-reconfigurable Platform with Relocatable Components*

DSP *Digital Signal Processing*

ECC *Error Correction Code*

EDIF *Electronic Design Interchange Format*

EPOS *Embedded Parallel Operating System*

ERS *Embedded Reconfigurable System*

ESA *European Space Agency*

FAR *Frame Address Register*

FDIR *Fault Detection, Isolation and Recovery*

FFT *Fast Fourier Transform*

FIFO *First-In First-Out*

FinFET *Fin Field-Effect-Transistor*

FPGA *Field Programmable Gate Array*

GPIO *General Purpose Input Output*

GSR *Global Set-Reset*

HDL *Hardware Description Language*

IC *Integrated Circuit*

ICAP *Internal Configuration Access Port*

IDCT *Inverse Discrete Cosine Transform*

IDF *Isolation Design Flow*

IE *Interface Element*

IGBT *Insulated Gate Bipolar Transistor*

IP *Intellectual Property*

IPIF *Intellectual Property Interface*

ITRS *International Technology Roadmap for Semiconductors*

JTAG *Joint Test Action Group*

LCD *Local Clock Domain*

LFSR *Linear Feedback Shift Register*

LUT *Look Up Table*

MACROS *Multimodal Adaptive Collaborative Reconfigurable self-Organized System*

MARS *Multi-task Adaptive Reconfigurable System*

MBU *Multiple Bit Upset*

MiniCAB *Mini Configurable Analogue Block*

MOSFET *Metal Oxide Semiconductor Field Effect Transistor*

NBTI *Negative Bias Temperature Instability*

NCD *Native Circuit Description*

NGC *Native Generic Circuit*

NGD *Native Generic Database*

NSEU *Neutron Single Event Upset*

OS *Operating System*

PB4MP *Partial Bitstream for Multiple Partitions*

PBTI *Positive Bias Temperature Instability*

PC *Personal Computer*

PCF *Physical Constraints File*

PE *Processing Element*

PR *Partial Reconfiguration*

RAM *Random Access Memory*

RCS *Reconfigurable Computing System*

RP *Reconfigurable Partition*

RPSP *Re-configurable Parallel Stream Processor*

ROM *Read-Only Memory*

ROSC *Ring Oscillator*

RTL *Register-Transfer Level*

RTSNoC *Real-Time Star Network-on-Chip*

SBU *Single Bit Upset*

SEB *Single-Event Burnout*

SEE *Single-Event Effect*

SEFI *Single-Event Function Interrupt*

SEGR *Single-Event Gate Rupture*

SEL *Single-Event Latch-up*

SEM *Soft Error Mitigation*

SET *Single-Event Transient*

SEU *Single-Event Upset*

SMART-FDIR

SoC *System-on-Chip*

SRAM *Static Random Access Memory*

SRL *Shift Register LUT*

TMR *Triple Modular Redundancy*

TID *Total Ionizing Dose*

TORC *Tools for Open Reconfigurable Computing*

TPG *Test Pattern Generator*

VHC *Virtual Hardware Component*

VHDL *Very High Speed Integrated Circuit Hardware Description Language*

XDL *Xilinx Design Language*

XML *Extensible Markup Language*

SUMÁRIO

I REFERENCIAIS TÉCNICO-TEÓRICOS 35

1 INTRODUÇÃO 37
1.1 Motivação 38
1.2 Objetivos 45
1.3 Contribuições 48
1.4 Organização do Texto 50

2 ESTRUTURA DA MEMÓRIA E RECONFIGURAÇÃO DAS FPGAS DA XILINX 51
2.1 Arquitetura Geral de uma FPGA 51
2.1.1 Blocos lógicos (CLBs, *Slices* e LUTs) 51
2.1.2 Matrizes de Interconexão 53
2.1.3 Outros Recursos de uma FPGA 55
2.1.4 Granularidade 57
2.2 Arquiteturas de famílias de FPGAs da Xilinx 58
2.2.1 Constituição das LUTs 58
2.2.2 Constituição dos *Slices* 58
2.2.3 Constituição dos CLBs 59
2.2.4 Secção de Coluna de CLBs 62
2.2.5 Linha de Secções de Recursos 63
2.2.6 Endereçamento da Memória de Configuração . 64
2.3 Capacidades de Reconfiguração 66
2.3.1 Reconfiguração e Múltiplos *Bitstreams* 67
2.3.2 Reconfiguração Parcial 68
2.3.3 Reconfiguração Parcial Baseada em Diferenças 69
2.4 Conclusão 70

3 ESTADO DA ARTE 71
3.1 Teste para Deteção de Faltas Permanentes em FPGAs 72

3.1.1 Faltas Características das FPGAs 72
3.1.2 Modelos de Faltas 77
3.1.3 Conclusão 79
3.2 Influência da Radiação TID nas FPGAs nas Aplicações Espaciais 79
3.2.1 Conclusão 81
3.3 Envelhecimento (*Aging*) devido ao NBTI e PBTI 82
3.3.1 Ameaças e Características do NBTI e PBTI . 82
3.3.2 Conclusão 85
3.4 Realocação de Módulos em FPGA 85
3.4.1 Abordagens para Realocação de Módulos . . . 86
3.4.2 Isolation Design Flow da Xilinx 91
3.4.3 Conclusão 92
3.5 Plataformas em FPGA com Tolerância a Faltas Permanentes 93
3.5.1 Metodologias de Teste para FPGAs Integradas em Sistemas Reconfiguráveis 93
3.5.1.1 Descrição da Metodologia Proposta 93
3.5.1.2 Conclusão, Vantagens e Desvantagens 95
3.5.2 Utilização de *Bit files* Parciais para Tolerância a Faltas 96
3.5.2.1 Parcelamento da FPGA e Funcionamento dos Módulos Realocáveis 96
3.5.2.2 Conclusão, Vantagens e Desvantagens 99
3.5.3 Realocação de Aceleradores em Hardware via ICAP para Execução e Sincronização de Multitarefas em Hardware 100
3.5.3.1 Descrição do Funcionamento do Sistema com Multitarefa em Hardware 100
3.5.3.2 Conclusão, Vantagens e Desvantagens 102
3.5.4 Plataformas em FPGAs com Sistemas Autónomos Tolerantes a Faltas 103

3.5.4.1 Descrição do Funcionamento do Sistema Autónomo Tolerante 103
3.5.4.2 Conclusão, Vantagens e Desvantagens 105
3.5.5 Sistema com Realocação de *Bitstreams* Otimizada para Aplicações com Multi-Módulos Carregados por Reconfiguração Parcial 106
3.5.5.1 Descrição do Funcionamento do Sistema com Capacidade de Realocação de *Bitstreams* 106
3.5.5.2 Conclusão, Vantagens e Desvantagens 108
3.5.6 Projeto de Sistemas Críticos em FPGAs com Tolerância a Faltas para Ambientes Hostis . . . 108
3.5.6.1 Descrição da Arquitetura do Mecanismo de Ocultação e Correção de Faltas 109
3.5.6.2 Conclusão, Vantagens e Desvantagens 110
3.5.7 Matriz Analógica e Digital Programável para Sistema Tolerante a Faltas 111
3.5.7.1 Descrição da Arquitetura da Matriz Analógica e Digital Programável 111
3.5.7.2 Conclusão, Vantagens e Desvantagens 113
3.5.8 Mecanismo Autónomo e Distribuído de Recuperação de Faltas (Transitórias e Permanentes) para Aplicações Espaciais 114
3.5.8.1 Arquitetura Reconfigurável para Processamento Paralelo de *Stream* 115
3.5.8.2 Arquitetura com Proteção Contra Radiação 117
3.5.8.3 Sistema de Computação Reconfigurável 117
3.5.8.4 *Framework* de ERSs com Realocação de VHCs 119
3.5.8.5 *Framework* para Aplicações Espaciais com Autorecuperação de Faltas 121
3.5.8.6 Mecanismos de Auto-Integração de *System-on-Chip* (SoC) para Sistemas de Reconfiguração Dinâmica . . . 122
3.5.8.7 Mecanismo Descentralizado para Recuperação de Faltas para Sistemas Espaciais baseados em FPGAs . . . 125

3.5.8.8 Arquitetura Reconfigurável com Capacidade de Adaptação aos Recursos Disponíveis pelo Sistema 127
3.5.8.9 Arquitetura Auto-Organizada Reconfigurável Ajustável para Recuperar de Faltas Transitórias e Permanentes 129
3.5.8.10 Conclusão, Vantagens e Desvantagens 131
3.6 Resumo da Revisão do Estado da Arte e Conclusão 132

II MATERIAIS E MÉTODOS 139

4 FUNDAMENTAÇÃO E ESTRATÉGIAS PROPOSTAS 141
4.1 Análise Probabilística das Faltas Permanentes numa FPGA 141
4.1.1 Utilizando apenas $1/r$ dos recursos: Maximização da tolerância a faltas permanentes 141
4.1.2 Utilizando $(r-1)/r$ dos recursos: Maximização da utilização da FPGA 146
4.2 Análise da Confiabilidade num sistema em FPGAs 149
4.3 Combate ao Envelhecimento (*aging*) Causado pelo NBTI 151
4.4 Estrutura da Solução Apresentada 153
4.4.1 Módulo de Controlo 154
4.4.2 Mecanismo de Deteção 157
4.4.3 Módulo de Atuação 159
4.4.4 Identificação das Etapas do Fluxo FDIR 160
4.5 Conclusão 161

5 TRABALHOS E METODOLOGIAS DESENVOLVIDAS 163
5.1 Detetor de faltas em módulos implementados em FPGAs 163
5.1.1 Arquitetura e Organização da FPGA Virtex-5 164

5.1.2 Gerenciamento da Configuração da FPGA . . 165
5.1.3 Detetor de faltas *Low Cost* 165
5.1.4 Arquitetura BIST 166
5.1.4.1 BIST com *Scan Chain* Virtual 166
5.1.4.2 Fluxo (*Flow*) de implementação do detetor de faltas . 167
5.1.4.3 Sequência de execução do BIST 169
5.1.4.4 Memória usada pelo BIST e tempo de execução . . . 171
5.1.5 Conclusão 175
5.2 Projeto de *Bistreams* Parciais para Múltiplas Partições em FPGAs da Xilinx 176
5.2.1 Interface e Roteamento na Reconfiguração Parcial 177
5.2.1.1 Interface com as Partições Reconfiguráveis 178
5.2.1.2 Roteamento entre o Sistema e as Partições Reconfiguráveis 180
5.2.2 Metodologia ADF Orientada para PB4MP . . 182
5.2.2.1 Memória usada devido ao uso do PB4MP e tempo de execução 189
5.2.3 Conclusão 192
5.3 Estratégia TMR com Características de *Dependability* Melhoradas Baseada na Metodologia PB4MP 193
5.3.1 TMR baseado em PB4MP 194
5.3.1.1 Arquitetura TMR 194
5.3.1.2 A Adaptação do PB4MP para Incrementar a Confiabilidade 195
5.3.1.3 Memória usada devido à implementação das estratégias e tempo de execução 197
5.3.2 Conclusão 197
5.4 Monitorização e Minimização dos Efeitos de *Aging* nas FPGAs 198
5.4.1 O DDS Desenvolvido 199
5.4.1.1 Arquitetura do *Delay Meter* 201
5.4.1.2 Arquitetura do *Differential Delay Sensor* 203

5.4.1.3 Análise da Primitiva *CARRY4* com a Ferramenta *AgingCalc* 204
5.4.1.4 Uso do *Delay Meter* como Medidor de Temperatura . 206
5.4.1.5 Memória Usada e Tempo de Execução Relativos à Implementação do DDS 208
5.4.2 Conclusão 210
5.5 Conclusões Gerais e Trabalhos Futuros 211

III ETAPA DE VALIDAÇÃO 217

6 EXPERIMENTOS E RESULTADOS 219
6.1 Experimento com Detetor de Faltas Desenvolvido 219
6.1.1 Resultados Experimentais 220
6.1.2 Avaliação dos Resultados Obtidos 220
6.1.3 Conclusões e Trabalho Futuro 222
6.2 Experimento com Implementação do Mecanismo PB4MP Desenvolvido 223
6.2.1 Resultados Experimentais 226
6.2.2 Avaliação dos Resultados Obtidos 227
6.2.3 Conclusões e Trabalho Futuro 228
6.3 Experimento da Implementação de um Sistema com TMR Baseado no Mecanismo PB4MP 228
6.3.1 Resultados Experimentais 233
6.3.2 Avaliação dos Resultados Obtidos 234
6.3.3 Conclusões e Trabalho Futuro 235
6.4 Experimento de um Sistema com Monitorização de Performance e de Envelhecimento Através do Sensor DDS 237
6.4.1 Resultados Experimentais 240
6.4.1.1 Caracterização do *Delay Meter* 241
6.4.1.2 Caracterização do *Differential Delay Sensor* 242

6.4.1.3 Influência da Temperatura na Corrente e Tensão de Alimentação da FPGA 242
6.4.1.4 Uso do DDS como Sensor de Temperatura numa Partição da FPGA 245
6.4.2 Avaliação dos Resultados Obtidos 247
6.4.3 Conclusões e Trabalho Futuro 248
6.5 Conclusões Gerais 250

7 CONCLUSÃO E TRABALHOS FUTUROS . 253

REFERÊNCIAS 257

APÊNDICES 275

APÊNDICE A – XILINX SPARTAN-6 *INSIDE OUT* 277
A.1 Introdução 277
A.2 FPGA Spartan-6 - Visão Global 278
A.2.1 *Configurable Logic Blocks* (CLBs) 278
A.2.2 SLICEX, SLICEL e SLICEM 279
A.2.3 *Look Up Table* (LUT) 279
A.3 Processo de Configuração de uma FPGA Spartan-6 281
A.3.1 Sequência do carregamento da configuração de uma FPGA Spartan-6 281
A.3.2 Organização do endereçamento de uma FPGA Spartan-6 283
A.4 Organização dos dados de Configuração de uma FPGA Spartan-6 285
A.4.1 Organização do *bit file* completo 286
A.4.2 Organização da configuração de uma linha (ROW) 286

A.4.3 Organização da configuração das colunas de 16 CLBs 288
A.4.3.1 Organização dos bits de configuração das LUTs nos *slices* 289
A.4.3.2 Organização dos bits de configuração dos CLBs (excluindo LUTs) 293
A.4.4 Organização da configuração das colunas de 8+4 BRAMs 302
A.4.5 Organização da configuração do conteúdo das BRAMs 307
A.5 Conclusões 310

Parte I

REFERENCIAIS TÉCNICO-TEÓRICOS

1 INTRODUÇÃO

O uso de *Field Programmable Gate Arrays* (FPGAs) para implementação de sistemas eletrónicos é cada vez mais difundido. Assim como qualquer outro componente eletrónico, as FPGAs são devidamente testadas após o seu processo de fabricação (*development phase*) (Avizienis et al., 2004). No entanto, mesmo passando exemplarmente em todos os testes de manufatura, isso não garante imunidade a faltas no futuro. Na verdade, estes dispositivos envelhecem, ou seja, degradam-se fisicamente durante a sua vida útil (Pradhan, 1996). Este processo pode ser mais ou menos lento, em função de vários fatores como: tecnologia usada no seu fabrico, no processo de controlo de qualidade nas etapas do seu fabrico, temperaturas e choques térmicos a que seja exposto (relacionados com o ambiente onde operam), tensões de funcionamento aplicadas e a sua própria dimensão e complexidade (Koren and Krishina, 2007).

Isto significa que após a etapa de fabricação e a partir de uma determinada altura no ciclo de vida do dispositivo, a probabilidade do aparecimento de faltas permanentes aumenta e é necessário verificar se as faltas não alteraram o normal funcionamento das funções implementadas na FPGA. Acrescentando o facto de que os módulos implementados numa FPGA são através da configuração de uma memória interna (Gokhale and Graham, 2005) do dispositivo, isto significa que para além da integridade dos recursos reconfiguráveis, também é preciso garantir a integridade da memória responsável por configurar esses recursos. Por isso, é imperativo munir o sistema implementado numa FPGA de uma capacidade de manutenção (*maintainability*) (Avizienis et al., 2004), de modo a que a sua disponibilidade (*availability*) aumente para lá do que a sua normal degradação física iria à partida implicar. Principalmente em aplicações espaciais onde poderão ter de operar por dezenas de anos, num ambiente extremamente hostil (sujeito a níveis de radiação cósmica superiores aos existentes ao nível do mar), e onde a reparação ou substituição de uma FPGA se torna difícil ou mesmo impossível.

A Figura 1 ilustra a taxa de falhas originadas por faltas que

ocorrem nas FPGAs que segue a designada por curva da banheira (*Bathtub Curve*) (Klutke et al., 2003). Esta curva, delimitada em três secções pelas duas linhas verticais a tracejado, correspondendo essas secções às três fases da vida útil de um componente eletrónico. São elas: mortalidade infantil ou precoce (*Infant Mortality*), falhas aleatórias (*Random Failures*) e falhas de desgaste (*Wearout Failures*).

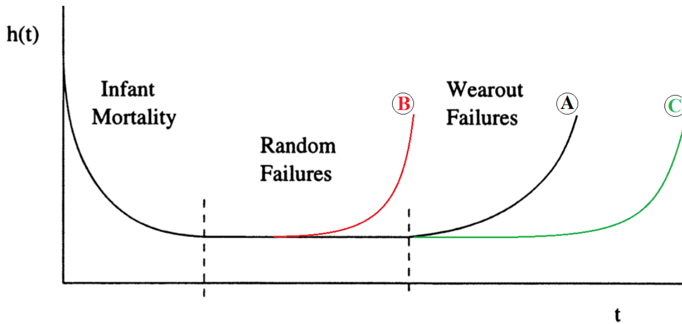


Figura 1 – Curva da Banheira para tempos de falhas. Modificado de (Klutke et al., 2003).

Assumindo o cenário **A** como o natural, ao submeter uma FPGA a uma situação de maior desgaste, devido a um envelhecimento (*aging*) causado pela radiação ou pelo *Bias Temperature Instability* (BTI), as faltas de desgaste ocorrem mais cedo, dando origem a falhas no sistema, e o indesejado cenário **B** passa a ser o real. Como após o fabrico do dispositivo, com uma determinada tecnologia, pouco existe a fazer para evitar a ocorrência de faltas permanentes, resta encontrar formas que impeçam que essas faltas causem falhas no sistema. Dessa forma, pelo menos em determinados tipos de aplicações, será possível ampliar o ciclo de vida de uma FPGA e assim alcançar o cenário **C**.

1.1 MOTIVAÇÃO

Recentemente, as faltas permanentes no hardware têm-se tornado um fator importante para sistemas usados em aplicações espaciais (Dumitriu et al., 2012). Existem várias causas para estas faltas (Xilinx

Inc., 2012a): a) efeitos de radiação, como o *Single-Event Latch-up* (SEL) e o *Total Ionizing Dose* (TID); b) defeitos de fabricação que se encontram escondidos; e c) o envelhecimento do próprio substrato. Todos estes casos devem ser considerados para sistemas com missões críticas e espaciais. O progresso na tecnologia *Complementary Metal Oxide Semiconductor* (CMOS) nanométrica permitiu uma maior concentração de recursos numa FPGA, o que por sua vez, aumentou igualmente a probabilidade de ocorrerem faltas permanentes no dispositivo (Xilinx Inc., 2016b).

Uma forma de combater as causas destas faltas permanentes é recorrer a um processo de fabrico de FPGAs que as tornem mais robustas. Esta abordagem baseia-se no uso de tecnologias de fabricação para o fortalecimento das portas CMOS que podem proporcionar um aumento significativo na tolerância ao TID. No entanto, o custo do dispositivo pode ser uma ordem de grandeza superior do que a mesma FPGA comercial. Outro potencial problema associado à utilização de FPGAs com maior resistência ao TID é a acessibilidade a estas mesmas FPGAs. Muitas vezes, restrições comerciais e de acesso à tecnologia representam um fator limitante para a utilização destas FPGAs em aplicação comerciais.

O uso de FPGAs em aplicações espaciais representa uma opção bastante atrativa devido à sua flexibilidade, que tanto permite futuras alterações/atualizações como aumentar o número de funcionalidades do sistema. No entanto, o seu uso traz consigo uma série de desafios, que exigem esforços redobrados a fim de tornar as aplicações tolerantes a faltas, com capacidade de recuperar o seu normal funcionamento diante da presença de faltas, e ao mesmo tempo prevenir a ocorrência de faltas (Avizienis et al., 2004). As FPGAs, tal como os *Application Specific Integrated Circuits* (ASICs), devido à hostilidade do meio ambiente em que se encontram inseridas quando no espaço, por exemplo, radiações e temperaturas, veem o seu ciclo de vida reduzir-se (Koren and Krishina, 2007). Além disso, no caso das FPGAs é exigida particular atenção pelo facto de as funções nelas existentes serem implementadas através de uma memória de configuração interna *Static Random Access Memory*

(SRAM) (Gokhale and Graham, 2005), que pode ser acidentalmente alterada devido à radiação a que está sujeita (Quinn et al., 2005).

Uma forma que alguns fabricantes encontraram para reduzir o efeito da radiação espacial nas FPGAs foi a criação de famílias com tecnologia *anti-fuse*, onde a memória de configuração interna é substituída por uma forma de programação única e irreversível. Embora isto elimine a ocorrência de *bit flips* na memória de configuração (porque esta não existe), não elimina a ocorrência dos *bit flips* nos flip-flops do circuito (nem nos blocos de memória), e o dispositivo perde a capacidade de ser novamente reprogramado (totalmente ou parcialmente) (Alfke and Padovani, 2004). No caso da Xilinx, este fabricante apostou antes na criação de famílias com uma maior imunidade à radiação (Xilinx Inc., 2012a). Assim, conseguiu manter todas as capacidades associadas às FPGAs e ao mesmo tempo uma boa imunidade à radiação espacial (Alfke and Padovani, 2004). Em todas estas soluções o foco é a prevenção de *bit flips* causados pela radiação (Quinn et al., 2005), não evitando o envelhecimento (*aging*) da FPGA. O *aging* pode ter origem em várias fontes como: a radiação (Kastensmidt et al., 2014) que vai destruindo a estrutura física da eletrónica do dispositivo; ou alguma forma de BTI. Relativamente ao *Bias Temperature Instability*, este ocorre de duas formas: o *Negative Bias Temperature Instability* (NBTI) que afeta os transístores pMOS (Zhang et al., 2014a) e o *Positive Bias Temperature Instability* (PBTI) que lesa os transístores nMOS (Mizubayashi et al., 2015). Ambos, embora não destruam a estrutura física dos transístores de um modo imediato, incrementam os seus respetivos tempos de resposta. Isto deve-se à variação do *threshold voltage* (V_T), que em casos estudados relativos ao NBTI, com uma tecnologia industrial de 45nm pode variar 9% ao fim de dois anos, podendo essa variação ser maior para temperaturas superiores a $125°C$ (Mishra et al., 2012).

Uma FPGA no espaço está sujeita a grandes níveis de radiação ionizante (*Ionizing Radiation*) que podem causar *Single-Event Effects* (SEEs) (White, 2012). Os SEEs que podem ocorrer nas FPGAs dividem-se em dois tipos:

- Os *Soft Errors*, quando a memória de configuração é corrompida, dando origem a faltas que podem ser removidas através da correção da memória correspondente. Existem três tipos de subclasses de *Soft Errors*: *Single-Event Transients* (SETs), *Single-Event Upsets* (SEUs) e *Single-Event Function Interrupts* (SEFIs) (White, 2012);

- Os *Hard Errors*, quando a FPGA fica com algum recurso irremediavelmente danificado, originando faltas que não poderão ser corrigidas (exceto através da substituição do dispositivo). Existem igualmente três tipos de subclasses de *Hard Errors*: SELs, *Single-Event Burnouts* (SEBs) e *Single-Event Gate Ruptures* (SEGRs) (White, 2012).

Relativamente aos *Soft Errors*, como são os que têm uma maior probabilidade de ocorrer, o fabricante de FPGAs Xilinx tem disponibilizado soluções capazes de tratar de forma adequada esse tipo de faltas (Carmichael et al., 2000) (Carmichael and Tseng, 2009) (Xilinx Inc., 2011a). Já em relação aos *Hard Errors*, sejam eles devidos à radiação ou ao BTI, devido a uma probabilidade de ocorrência menos expressiva em relação aos *Soft Errors*, nenhuma solução específica é agregada às FPGAs por parte do fabricante. Neste sentido, o fabricante recomenda apenas o uso de estratégias como, por exemplo, *Triple Modular Redundancy* (TMR) (Pradhan, 1996).

Alguns trabalhos veem sendo propostos neste contexto. No passado, uma abordagem muito interessante foi apresentada em (Gericota et al., 2001) (Gericota et al., 2002). Aproveitando o forte domínio em controlo de baixo nível sobre os recursos da FPGA, liberta sucessivamente cada recurso em utilização, trocando-o por outro igual que não esteja a ser utilizado e testa cada um individualmente. Detetada alguma falta permanente, esse recurso deixa de ser utilizado. Devido à evolução das FPGAs que passaram da arquitetura *Column-based* usada até às Virtex-2, para a arquitetura *Tile-based* usada a partir da Virtex-4 (Shih and Hsiung, 2009), esta abordagem foi impossível de manter. Isto porque o fabricante não continuou a disponibilizar documentação sobre as novas famílias de FPGAs, com o mesmo elevado nível de detalhe técnico. Isto

associado ao facto de o roteamento numa FPGA ter-se tornado mais complexo e que alterá-lo interfere nos caminhos críticos das funções implementadas, fez com que esta abordagem não tivesse evolução para as atuais famílias de FPGAs.

No *Air Force Institute of Technology* em Hobson Way foi desenvolvido um trabalho (Montminy et al., 2007) onde o autor divide os recursos da FPGA em várias partes homogéneas, aloca cada módulo do sistema em cada uma dessas partes, reservando uma ou mais, para realocar módulos que tenham sofrido uma falta permanente na parte dos recursos onde foram inicialmente alocados.

Na última década no *Department of Electrical and Computer Engineering* da Universidade de Ryerson em Toronto, um grupo realizou vários trabalhos onde um dos focos foi uma plataforma que resolvesse o aparecimento de faltas permanentes autonomamente (Kirischian et al., 2004) (Kirischian et al., 2006) (Kirischian et al., 2009) (Kirischian et al., 2010) (Dumitriu and Kirischian, 2010) (Dumitriu et al., 2012). Subdividindo os recursos da FPGA em várias partições reconfiguráveis (do inglês *Reconfigurable Partitions*) a que chamam *slots*, cada *slot* implementa um módulo do sistema e caso algum venha a sofrer de alguma falta permanente, o módulo nele implementado será movido para outro que se encontre livre e isento de qualquer falta permanente. De modo a tornar esta plataforma ainda mais robusta, as operações verificação, reprogramação e sincronização são realizadas com uma política de descentralização, onde cada módulo está preparado para realizar essas tarefas, evitando existir um módulo central para as executar. A evolução mais recente do trabalho deste grupo evidencia cada vez mais o foco em aplicações espaciais, que é igualmente a direção deste trabalho (Dumitriu and Kirischian, 2013) (Dumitriu et al., 2014) (Dumitriu et al., 2015a) (Dumitriu et al., 2015b).

As presentes abordagens recorrem apenas às ferramentas de alto nível usadas na reconfiguração parcial (do inglês *Partial Reconfiguration*). Isso implica que cada módulo tem de ter um ficheiro de reconfiguração parcial (*bit file*) associado para cada zona de recursos (*slot*) onde possa ser alocado. Significa isto que, por exemplo, numa plataforma (Dumitriu

et al., 2012) (Dumitriu and Kirischian, 2013) com 16 *slots*, terá de ter 16 *bit files* para cada módulo. Para um sistema que tenha 10 módulos diferentes, implica ter uma biblioteca de 160 *bit files*, algo que além de exigir memória extra, poderá não ser muito escalável e obrigará também a ter formas de garantir que a memória é imune a faltas.

Este *overhead* que é a obrigatoriedade da existência da biblioteca de *bit files* de reconfigurações parciais e respetiva gestão, deve-se ao facto de o fluxo de funcionamento das ferramentas do fabricante (neste caso a Xilinx) gerarem *bit files* de reconfiguração parcial para localizações absolutas. Ou seja, mesmo que tenhamos duas partições com recursos exatamente iguais (quantidade e distribuição), para um mesmo módulo, é necessário um *bit file* de reconfiguração parcial para cada partição, caso se deseje poder ter a opção de decidir em que partição alocar esse mesmo módulo.

No entanto, o estado de evolução das FPGAs e respetivas ferramentas permite-nos um maior controlo sobre os recursos da FPGA. Se por um lado, a informação técnica de baixo nível disponibilizada pelos fabricantes de FPGAs é cada vez mais reduzida (quer por causa do aumento do nível de abstração, quer por uma questão de segredo industrial), por outro lado, o conjunto de diretivas possíveis de dar através da imposição de restrições (*constraints*) (Xilinx Inc., 2013b) (Xilinx Inc., 2013e) é cada vez maior. Ou seja, existe por parte do fabricante uma biblioteca que permite manipular num baixo nível a geração de um *bit file*, embora muitas vezes não seja direto entender o que uma diretiva provoca no baixo nível no dispositivo, exatamente pela falta de informação técnica por parte do fabricante. É acima de tudo uma biblioteca de largas dezenas de *constraints* para o fabricante usar nas próprias ferramentas de um modo maioritariamente transparente para o projetista. Mas sendo uma biblioteca regularmente atualizada e que vai mantendo a compatibilidade com as *constraints* anteriores, significa que o risco de uma nova família de FPGAs deixar de respeitar essa biblioteca é relativamente reduzido. Normalmente, uma nova família acrescenta variantes das *constraints* já existentes, continuando a respeitar as restantes.

Aproveitando estas *constraints*, alguns trabalhos têm sido fei-

tos para alcançar a compatibilidade de *bit files* entre *Reconfigurable Partitions* (RPs). Usam como base o fluxo da ferramenta PlanAhead da Xilinx (Xilinx Inc., 2013d), incluindo alguns passos manuais extra pelo meio (Ichinomiya et al., 2012b) (Ichinomiya et al., 2012a). Baseado nestes trabalhos um outro foi desenvolvido com algumas melhoras relativamente ao roteamento do restante sistema não cruzar as RPs, para além de reduzir a necessidade de *Look Up Table*s (LUTs) para criar a interface entre módulos e o restante sistema (Drahonovský et al., 2013). No entanto, são soluções que exigem intervenção manual em vários passos, sendo alguns deles demorados. Como ferramenta mais autónoma, foi desenvolvido o GoAhead (Beckhoff et al., 2012). Esta solução baseia-se na manipulação e conversão entre os formatos *Xilinx Design Language* (XDL) e *Native Circuit Description* (NCD), após a etapa de mapeamento das ferramentas da Xilinx. É um processo que exige um elevado processamento para sistemas mais complexos, e que não será possível realizar nas novas ferramentas como o Vivado (Xilinx Inc., 2015d), pelo facto do formato XDL deixar de ser devidamente suportado.

Tudo isto dá espaço para desenvolver novas soluções que permitam dotar sistemas implementados em FPGA de capacidades de autonomamente recuperar de faltas permanentes (através da exclusão dos recursos onde ocorrem as faltas) e prevenir que elas ocorram (evitando um uso exaustivo de certos recursos em detrimento de outros). Além disso, no presente, a fabricante de FPGAs Xilinx[1] tem igualmente cada vez mais características que auxiliam a deteção de faltas transitórias na memória de configuração (por exemplo, primitivas como *Internal Configuration Access Port* (ICAP) ou FRAME_ECC_VIRTEXx) (Xilinx Inc., 2012b) (Xilinx Inc., 2015c), tal como as próprias ferramentas de *Partial Reconfiguration* (PR) vêm permitindo uma flexibilidade cada vez maior ao projetista para gerir os recursos da FPGA (Xilinx Inc.,

[1] Por ser atualmente uma das principais empresas que desenvolvem FPGAs e por ser a que o autor deste tese melhor conhece (tanto a nível técnico das famílias de dispositivos como a nível de ferramentas), a escolha da fabricante das FPGAs usadas neste trabalho recaiu sobre a Xilinx.

2013e) (Xilinx Inc., 2013c), o que pode ser igualmente utilizado para distinguir faltas permanentes das transitórias (*Soft Errors* dos *Hard Errors*), e encontrar formas de as ultrapassar e evitar.

Resumindo, o atual cenário de soluções existentes (ou em desenvolvimento) para esta finalidade é ainda reduzido. Por outro lado, o estado de evolução das FPGAs e do conjunto de diretivas possíveis de dar via *constraints* permite-nos um grande controlo sobre os recursos da FPGA. Tudo isto dá espaço para desenvolver novas soluções que permitam dotar sistemas implementados em FPGA de capacidades de autonomamente prevenir a ocorrência de faltas permanentes, e que quando estas ocorram, recuperarem dessas faltas sem que isso implique inutilizar partições reconfiguráveis. Uma solução como estas ganha ainda mais relevo pelo facto de no Brasil, o Instituto Nacional de Pesquisas Espaciais (INPE) possuir como objetivo o uso de FPGAs em missões futuras. Adicionando a um sistema implementado numa FPGA a capacidade de auto-recuperar não só das faltas transitórias, mas também das permanentes, será possível aproveitar as vantagens do uso destes dispositivos, e ao mesmo tempo, manter algum conservadorismo típico da área espacial. Conservadorismo esse que é inerente ao elevado custo associado a uma missão gorada devido ao uso de uma tecnologia que está a dar os primeiros passos nesta área.

1.2 OBJETIVOS

Esta tese teve como objetivo primordial alcançar uma solução eficiente para recuperação autónoma de faltas permanentes em FPGAs. Solução essa que é flexível, não invasiva, autónoma, bem como economiza os recursos disponíveis. Esta mesma solução permite implementar estratégias que possibilitam igualmente retardar o envelhecimento (*aging*) dos recursos da FPGA causado pelo NBTI e PBTI.

Para isso, esta proposta é sustentada em dois pilares que se podem designar por camadas. O primeiro é a camada de suporte e que consistiu em todo o levantamento e desenvolvimento técnico que possibilitou suportar a implementação do segundo objetivo. O segundo,

a camada de aplicação, consiste em aproveitar o atual conhecimento relativo à realocação de módulos em múltiplas partições e evoluir para uma solução mais eficiente e automática. Essa solução é a proposta de um mecanismo, que permite a um sistema gerir a alocação dos seus próprios módulos em função da ocorrência de faltas permanentes ou de uma estratégia anti-envelhecimento.

Relativamente à camada de suporte, os objetivos foram:

1. Obtenção de informação de baixo nível relativa à memória de configuração da FPGA. Devido ao facto de que a existência de fontes onde se possa obter este género de informação de uma forma direta ser escassa, esta foi alcançada por interpretação e cruzamento de fontes indiretas (algumas notas de aplicação disponibilizadas pelo fabricante e que evidenciam determinados detalhes), e engenharia inversa realizada por intermédio da geração/observação de múltiplos *bitstreams*. No final foi obtido um mapa da memória de configuração da FPGA, com indicação da finalidade de cada bit (ou grupo de bits) e seu respetivo endereço;

2. Identificar as *constraints* relacionadas com o mapeamento e roteamento. Consistiu fundamentalmente na seleção de diretivas existentes que permitem ter controlo sobre a organização dos recursos da FPGA, da suas localizações e dos caminhos que os interligam;

3. Encontrar um novo sensor de performance. Tirando proveito do conhecimento baixo nível da FPGA reunido, foi desenvolvido um sensor que possibilita monitorizar tanto a performance como o nível de envelhecimento de uma determinada RP;

4. Apresentar uma metodologia que permite a criação de um novo mecanismo de realocação de módulos em FPGA. Com a informação obtida nos objetivos anteriores, foi proposto um fluxo automático que implementa um novo mecanismo de realocar módulos numa FPGA, destinado ao desenvolvimento de aplicações que necessitem

de uma gestão de módulos mais flexível do que aquela proporcionada pelas atuais ferramentas disponibilizadas pelo fabricante de FPGAs Xilinx. Este mecanismo permite igualmente a opção de incluir o sensor de performance desenvolvido.

Em relação à camada de aplicação, que consiste na prevenção e recuperação autónoma de faltas permanentes em FPGAs, os objetivos foram:

1. Aplicação para recuperação de faltas permanentes. Com base na camada de suporte, foi proposto um procedimento que acrescenta a um sistema embarcado numa FPGA, a capacidade de auto-reorganizar a sua configuração, de modo a tornar faltas permanentes detetadas em faltas redundantes[2]. Tal auto-reorganização passa pela troca de módulos entre partições de forma a isolar a falta permanente;

2. Método passivo para redução do *aging* causado pelo BTI. Apresentar um procedimento que adiciona a um sistema embarcado numa FPGA, a possibilidade de gerir a utilização dos recursos existentes nas RPs. Esta gestão tem como base a rotação dos módulos entre as várias RPs de modo a variar o uso e configuração dos recursos, minimizando assim o envelhecimento causado pelo BTI. A cadência da rotação dos módulos é obtida por intermédio de um temporizador, sem qualquer observação interna da FPGA;

3. Alcançar um método ativo para redução do *aging* causado pelo BTI. Adicionar ao procedimento descrito no ponto anterior proatividade na gestão do uso dos recursos. O procedimento reúne a informação obtida pelos sensores de performance existentes nos módulos alocados nas RPs, que processada juntamente com as características de cada módulo (como por exemplo a potência dissipada), decide onde e quando cada módulo é realocado.

[2] Faltas redundantes são faltas que embora existam no sistema não originam qualquer erro no mesmo, e por isso não dão origem a qualquer defeito (Avizienis et al., 2004).

Além dos objetivos principais, este trabalho proporcionou a obtenção de resultados secundários durante o seu desenvolvimento. O emergir de ideias, durante o desenvolvimento, para novas aplicações, sejam na mesma área de recuperação/prevenção de faltas permanentes ou noutra. Não significando isso que as respetivas implementações tenham sido realizadas, para evitar a dispersão na investigação. Um exemplo dessas ideias foi o planeamento descrito no trabalho (Martins et al., 2016), onde é estudada uma solução para incluir num CubeSat, com a finalidade de medir a ocorrência de faltas transitórias e permanentes numa FPGA comercial a operar no espaço. No entanto, as ideias ocorridas juntamente com o agregar de informação técnica exigida para a implementação desta proposta, irão fomentar e catalisar outros trabalhos de elementos do mesmo laboratório (ou laboratório parceiro).

1.3 CONTRIBUIÇÕES

O uso de FPGAs SRAM em projetos de sistemas embarcados para aplicações espaciais, mesmo que em módulos menos críticos, é uma opção que suscita receios. O facto deste dispositivo implementar a sua funcionalidade através da configuração de uma memória SRAM torna-o mais vulnerável à ocorrência de faltas. Por outro lado, no Brasil, o Instituto Nacional de Pesquisas Espaciais (INPE) pretende usar FPGAs em lançamentos futuros, exatamente para aproveitar a capacidade de poderem ser reprogramadas. Esta capacidade, para além de reduzir os custos de desenvolvimento, permite corrigir algum *bug* mesmo após o lançamento.

Neste contexto, esta tese apresenta uma metodologia que admite realocar módulos em hardware em múltiplas partições parciais, sem necessidade de múltiplos *bitstreams* parciais, que com uma política assertiva de seleção de recursos da FPGA permite o sistema tolerar a ocorrência de pelo menos uma falta permanente. Esta metodologia tira vantagem do facto da implementação de um sistema numa FPGAs ser feita por intermédio do conteúdo de uma memória SRAM, que corresponde à configuração de uma matriz de recursos com alguma

homogeneidade, e dessa forma manipular a localização dos módulo de modo a isolar os recursos com pelo menos uma falta permanente.

Este trabalho, orientado para sistemas implementados em FPGAs SRAM, apresenta como original o colocar a realocação de módulos em múltiplas *Reconfigurable Partitions* (RPs) em prol da recuperação da faltas permanentes por parte do sistema, contribuindo assim para ajudar a dissipar parte dos receios em relação a estes dispositivos em aplicações espaciais. A capacidade de um sistema conseguir superar a ocorrência de uma falta permanente traz um aumento da confiabilidade (*reliability*), algo primordial neste género de aplicações. O retardar do envelhecimento devido ao NBTI em sistemas que poderão ter de funcionar durante décadas, é outra importante contribuição para elevar o nível de confiabilidade, alcançada através de uma rotação da localização dos módulos nas RPs disponíveis no sistema, que permite a variação do uso dos recursos e da respetiva configuração.

A informação de baixo nível relativa à FPGA reunida, embora não seja uma contribuição direta para a ciência, o seu conhecimento favorece o desenvolvimento de novos projetos que exijam um domínio semelhante sobre os recursos de uma FPGA.

Todo o trabalho foi investigado e desenvolvido tanto no Grupo de Sistemas Embarcados (GSE), pertencente à Universidade Federal de Santa Catarina (UFSC), como no exterior, no laboratório *Electronic System Design and Automation* (ESDA) do grupo de investigação *Embedded Electronic Systems* (EES) existente no INESC-ID em Lisboa. Esta experiência contribuiu assim também para interligar dois importantes polos de conhecimento, que poderão permitir novos e mais complexos projetos em parceria.

Como contribuição científica, os resultados deste trabalho foram submetidos num artigo completo no periódico IEEE *Transactions on Reliability*, e foram publicados até ao momento em cinco conferências internacionais ((Martins et al., 2014a) (Martins et al., 2015c) (Martins et al., 2015b) (Martins et al., 2015a) (Martins et al., 2015d)). Tirando proveito dos conhecimentos técnicos adquiridos durante a pesquisa para

esta tese, e auxiliando o grupo responsável pela missão Floripa-Sat[3], foram ainda publicados mais dois trabalhos em conferências internacionais da área ((Martins et al., 2014b) (Martins et al., 2016)).

1.4 ORGANIZAÇÃO DO TEXTO

A partir deste ponto, este documento está organizado da seguinte maneira:

Capítulo 2 expõe os principais detalhes da arquitetura das FPGAs, como é a organização dos seus recursos internos, em que consiste a capacidade de reconfiguração das FPGAs e como essa capacidade é implementada e operada.

Capítulo 3 é dedicado à aos trabalhos relacionados. Serão apresentados os trabalhos mais relevantes associados a auto-teste em FPGAs e recuperação autónoma de ocorrência de faltas, com principal foco nas faltas permanentes.

Capítulo 4 apresenta as estratégias propostas e fundamenta de uma forma teórico-prática o porquê de as propor.

Capítulo 5 descreve o trabalho realizado em cada uma das etapas desta tese, o processo incremental de valor a cada etapa e as razões que influenciaram a sequência temporal destas mesmas etapas.

Capítulo 6 relata os experimentos feitos de forma a validar os trabalhos realizados no âmbito do tema desta tese.

Finalmente, o **Capítulo 7** inclui algumas apreciações finais do trabalho realizado e indica possíveis linhas para trabalhos futuros.

Apêndice A consiste num estudo detalhado realizado sobre a estrutura da memória de configuração de uma FPGA da família Spartan-6 da Xilinx, e que teve um importante peso na tomada de determinadas decisões relativas à solução proposta.

[3] Floripa-Sat é um CubeSat que se encontra em desenvolvimento por uma equipa de estudantes na Universidade Federal de Santa Catarina (UFSC).

2 ESTRUTURA DA MEMÓRIA E RECONFIGURAÇÃO DAS FPGAS DA XILINX

O conhecimento em detalhe da constituição de uma *Field Programmable Gate Array* (FPGA) é fundamental para que se consiga obter da mesma forma o melhor desempenho e eficiência. Nesta secção será descrito um resumo técnico geral relativo aos vários recursos de hardware que constituem uma FPGA e as suas respetivas funções. Sendo eles desde *Configurable Logic Block*s (CLBs) usados para implementação de funções lógicas, matrizes de interconexão que permitem a interligação dos vários recursos, *Digital Signal Processing*s (DSPs) usados para cálculos matemáticos, passando também pelos blocos de memória BRAMs. São ainda apresentados pormenores relativos a arquiteturas de algumas famílias de FPGAs da Xilinx (Xilinx Inc., 2016a). Por último, são apresentados detalhes sobre programação e reprogramação parcial, seguido de um resumo sobre a organização da memória de configuração de uma FPGA da Xilinx. Resumo que foi alcançado após a pesquisa realizada durante o programa de doutorado e que pode ser consultada no Apêndice A.

2.1 ARQUITETURA GERAL DE UMA FPGA

Uma FPGA pode ser descrita como uma matriz de componentes lógicos, que podem ser configurados e interligados como o projetista pretender. A tal liberdade é acrescentada ainda a possibilidade de poder realizar as vezes que forem necessárias, até implementar o sistema final (Shih and Hsiung, 2009).

2.1.1 Blocos lógicos (CLBs, *Slices* e LUTs)

Ignorando as entradas/saídas e alguns blocos específicos, é possível afirmar que os recursos existentes dentro de uma FPGA podem ser subdivididos em dois grupos: blocos lógicos e matrizes de interconexão (Gokhale and Graham, 2005).

Toda a expressão computacional pode ser expressa através de uma equação Booleana, que por sua vez pode ser representada por uma

tabela verdade. Então, o elemento lógico de maior relevo numa FPGA são as LUTs, cuja construção é baseada no uso de um multiplexador N para 1 (N entradas e uma saída), com N bits de memória e que são capazes de implementar uma tabela verdade (Shih and Hsiung, 2009).

Uma LUT implementa qualquer função lógica de N entradas através da configuração da sua memória, que desta forma realiza a tabela verdade correspondente à função. Tome-se como exemplo a implementação de uma função "ou exclusivo"(XOR) com três entradas, sugerida por (Hauck and Dehon, 2008) na Figura 2. A implementação da função na LUT é feita preenchendo a memória de modo a respeitar a respetiva tabela verdade. Uma combinação de entrada (que fisicamente é o endereço da memória) irá selecionar o valor correspondente para resultado da função implementada. Funções com um maior número de entradas que aquelas que uma LUT possui são implementadas agregando-se diversas LUTs entre si.

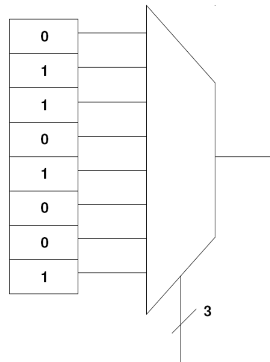


Figura 2 – Representação básica de uma LUT com três entradas com uma tabela verdade que implementa a função XOR (Hauck and Dehon, 2008).

Embora a LUT tenha sido escolhida como a menor unidade computacional nas FPGAs comercialmente disponíveis, o número de entradas de uma LUT, que influencia diretamente o seu tamanho, não é consensual e continua a ser estudado (Hauck and Dehon, 2008).

Se uma FPGA tiver apenas LUTs, ela não poderá manter nenhum

tipo de estado e, portanto, não poderá realizar qualquer tipo de lógica sequencial. Para acrescentar esta capacidade, os fabricantes adicionaram um flip-flop do tipo D à saída de cada LUT, dando origem, por parte da Xilinx (Xilinx Inc., 2016a), ao *slice*. A Figura 3 ilustra de uma maneira genérica como fica um *slice* de FPGA.

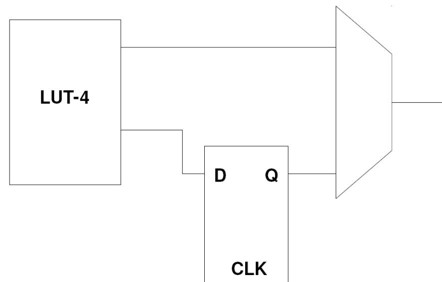


Figura 3 – Esquema básico de um *slice* com uma LUT de quatro entradas (Hauck and Dehon, 2008).

A um grupo de *slices*, a Xilinx designou nas suas famílias de FPGA por *Configurable Logic Block* (CLB) (Xilinx Inc., 2016a). O número de LUTs por *slice* ou por bloco lógico é igualmente investigado, como no caso dos estudos realizados por Dehon (Dehon, 2005), onde evidências empíricas sugerem que agrupamentos com mais de uma LUT-4 (LUT com quatro entradas) podem melhorar os aspetos de área de silício consumida e atrasos em sinais. Muitas FPGAs comerciais adequaram o número de LUT-4s em cada bloco lógico para tirar vantagem destas observações. A partir da família Virtex-5 (Xilinx Inc., 2012c) da Xilinx, as LUTs passaram a ser de seis entradas LUT-6.

2.1.2 Matrizes de Interconexão

As FPGAs atuais são formadas por uma matriz bidimensional de blocos lógicos e uma malha de trilhas que permitem a sua interconexão. Conforme mostra a Figura 4, esta arquitetura oferece uma grande flexibilidade de interconexão aos blocos lógicos (Hauck and Dehon, 2008).

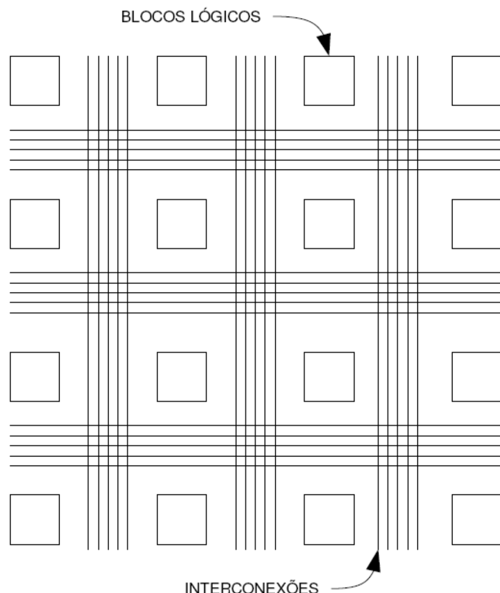


Figura 4 – Representação geral da arquitetura de uma FPGA (Hauck and Dehon, 2008).

Na Figura 5 é apresentado com mais pormenor a capacidade de interconexão entre blocos lógicos. Observa-se que as trilhas horizontais e verticais são os recursos de interconexão entre os diversos blocos. Sendo que um bloco lógico acessa o recurso de conexão mais próximo através do bloco de conexão, também designado por *Connection Block* (CB). As caixas de comutação (*SWITCH BOX*) têm a função de realizar a interconexão entre as trilhas horizontais e verticais de roteamento entre os CBs (Hauck and Dehon, 2008).

De modo a reduzir o atraso que pode ocorrer na transmissão de um sinal entre dois blocos lógicos localizados em extremidades opostas da FPGA são utilizadas estratégias como a do agrupamento hierárquico. A Figura 6 ilustra um exemplo genérico de agrupamento hierárquico.

No nível mais baixo da hierarquia, matrizes de 2x2 blocos lógicos são agrupadas formando um único *cluster*. Dentro deste *cluster*, apenas é possível o roteamento entre os blocos vizinhos. Por sua vez,

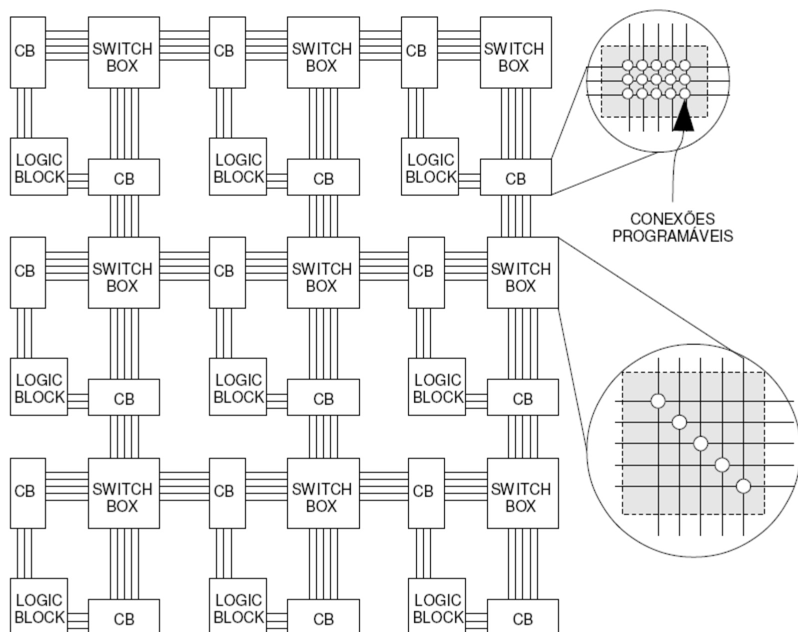


Figura 5 – Arquitetura de conexão entre os blocos lógicos (Hauck and Dehon, 2008).

matrizes de 2x2 destes *clusters* são organizadas em *clusters* maiores, englobando 16 blocos lógicos. Neste nível da hierarquia, é possível um roteamento entre *clusters* de 2x2 blocos lógicos, como se cada *cluster* 2x2 blocos lógicos fosse reduzido a apenas um bloco lógico. Este processo é repetido recursivamente nos níveis mais altos da hierarquia (Hauck and Dehon, 2008).

2.1.3 Outros Recursos de uma FPGA

Com o uso dos blocos lógicos é possível a realização de qualquer circuito lógico combinatório ou sequencial. Porém, o número de blocos lógicos necessários para a implementação de determinados circuitos ou as restrições de área ocupada por tais circuitos podem ser impeditivos para cercas aplicações. As FPGAs passaram por isso a incluir novos recursos (muitas vezes específicos para certas operações) de modo a

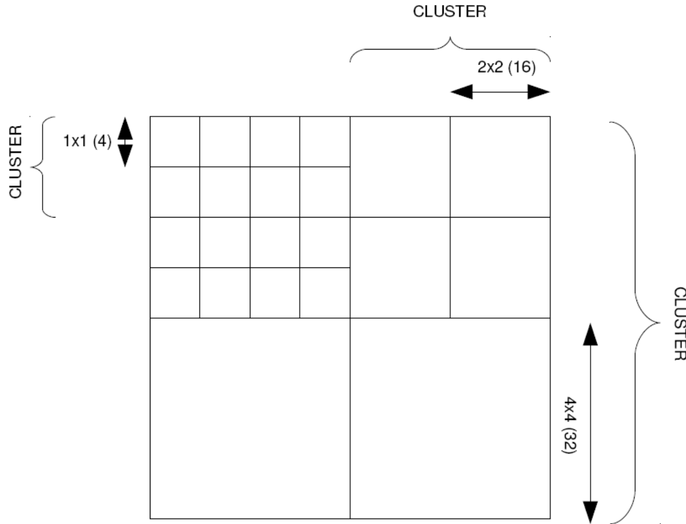


Figura 6 – Roteamento hierárquico usado para interconexão de grupos de blocos lógicos (Hauck and Dehon, 2008).

aumentar o desempenho para determinadas aplicações, o que reduz a utilização de blocos lógicos. Alguns exemplos destes são:

- *Fast carry chain*: normalmente designado por *carry logic*, é adicionado ao bloco lógico e formam caminhos dedicados para conduzir o resultado de operações de soma (do inglês *carry*) para o bloco adjacente de maior ordem numa operação de soma aritmética, evitando assim o uso dos canais de interconexão convencionais e aumentando o desempenho da operação;

- Multiplicadores (DSPs): especialmente projetados para aplicações de processamento digital de sinais onde há o uso intensivo de somas e multiplicações;

- BRAMs (ou *Block RAMs*): blocos de memória que permitem uma capacidade e flexibilidade maior para armazenamento de dados, como a implementação de *Random Access Memory* (RAM), *Read-Only Memory* (ROM) ou *First-In First-Out* (FIFO);

- Processadores: arquiteturas como um Cortex da ARM (Inc., 2016) (Xilinx Inc., 2013h), permitem a implementação de soluções com sistema operacional embarcado e são sistemas projetados para aplicações de alto valor agregado e que exigem desempenho computacional elevado.

Como exemplo, a Figura 7 ilustra a arquitetura de uma FPGA da família Spartan-6 (Xilinx Inc., 2011b) da Xilinx. Para além dos blocos lógicos, que neste dispositivo correspondem aos CLBs (compostos por oito LUTs de seis entradas cada uma), existem DSPs e outros módulos com finalidades bem específicas.

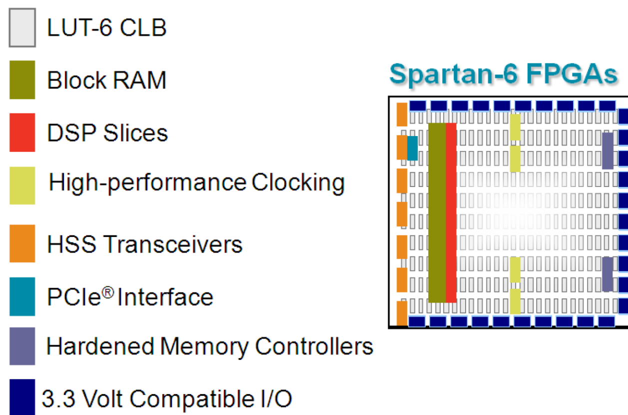


Figura 7 – Constituição de uma FPGA Spartan-6.

2.1.4 Granularidade

Uma das características mais importantes de um sistema baseado numa arquitetura reconfigurável é a estrutura usada para o processamento reconfigurável. Diferentes sistemas apresentam diversos tipos de granularidade. Para autores como (Hauck and Dehon, 2008), estes sistemas variam de estruturas com uma granularidade mais fina onde é possível manipular dados em nível de bit, até estruturas com maior granularidade (granularidade grossa) utilizadas para manipular

grupos de bits através de funções complexas como *Arithmetic Logic Units* (ALUs) ou DSPs. Outros autores, como Phan Cong-Vinh (Cong-Vinh, 2009), subdividem a granularidade em fina, média e grossa (do inglês *fine-grained*, *medium-grained* e *coarse-grained*). Neste caso, estruturas de granularidade média são construídas com blocos lógicos funcionais para processar múltiplos dados com largura de bits diferentes. Este mesmo autor (Cong-Vinh, 2009), defende que em comparação com as estruturas de menor granularidade, as estruturas de granularidade média apresentam um melhor desempenho, uma vez que são otimizadas para executar tarefas específicas. Mesmo assim, a desvantagem desta abordagem, segundo este autor, é o alto custo do processo de síntese.

2.2 ARQUITETURAS DE FAMÍLIAS DE FPGAS DA XILINX

Nesta secção são apresentadas as estruturas base da arquitetura das famílias Spartan-6 (Xilinx Inc., 2010), Virtex-5 (Xilinx Inc., 2012c), Virtex-6 (Xilinx Inc., 2012d) e 7 Series (inclui as famílias Artix-7, Kintex-7 e Virtex-7) (Xilinx Inc., 2013a) da Xilinx. Embora diferentes, as arquiteturas destas famílias têm uma base comum igual entre elas. Associadas a essas estruturas existe uma organização da memória de configuração, que é aqui resumidamente explicada numa visão de baixo para cima *bottom-up*. A maioria desta informação foi obtida no decorrer da investigação reportada no Apêndice A.

2.2.1 Constituição das LUTs

Na anterior secção 2.1.1 foram apresentados alguns detalhes relativos às LUTs, *slices* e CLBs. Iniciando pelo elemento LUT, a partir da família Virtex-5 este passou a ter seis entradas (LUT-6) e a possibilidade de duas saídas. A Figura 8 mostra como é organizada a LUT-6.

2.2.2 Constituição dos *Slices*

Agrupando quatro LUTs, oito ou quatro flip-flops (dependendo da família de FPGAs), multiplexadores e lógica de *carry*, obtemos um

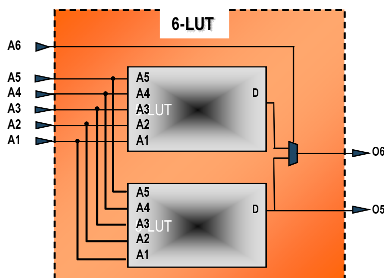


Figura 8 – Composição de uma LUT de seis entradas (extraído do Apêndice A).

slice. Estes elementos são usados por todos os *slices* para fornecer funções lógicas e aritméticas, incluindo o uso como memórias *Read-Only Memory* (ROM).

Existem três tipos de *slices*, onde uns fornecem mais funcionalidades do que outros. Essas funcionalidades são: armazenamento de dados usando memória *Random Access Memory* (RAM) distribuída, deslocadores de dados *Shift Register LUT*s (SRLs) com 32 registadores, multiplexadores maiores para permitirem mais capacidade de interligações internas, e o já mencionado *carry logic*. *Slices* que oferecem todos estes recursos são chamados de SLICEM, os que apenas oferecem multiplexadores maiores e *carry logic* são chamados de SLICEL, e os mais básicos, que não têm nenhuma destas funcionalidades, são chamados de SLICEX. Na Figura 9 é possível observar a constituição de um *slice* do tipo SLICEM, tal e qual como os existentes nas famílias Spartan-6, Virtex-6 e 7 Series.

Na família Spartan-6 existem *slices* dos três tipos, onde os *slices* da coluna da esquerda podem ser SLICEM ou SLICEL, e os da coluna da direita são sempre SLICEX. No caso da Virtex-5, Virtex-6 e 7 Series, apenas existem *slices* do tipo SLICEM e SLICEL.

2.2.3 Constituição dos CLBs

Dois *slices* formam o denominado de *Configurable Logic Block* (CLB) e este é o principal recurso de lógica disponível para a implemen-

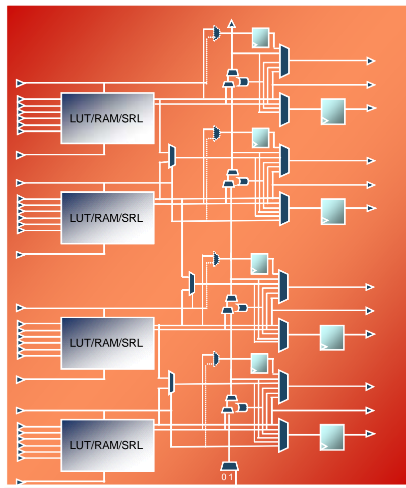
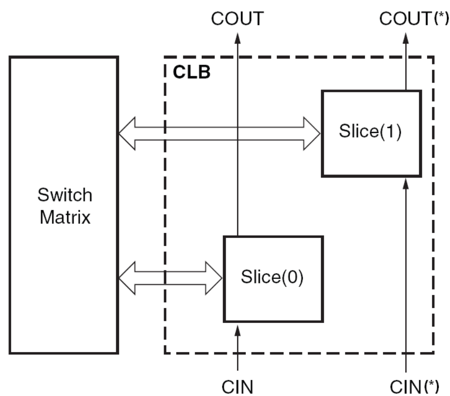


Figura 9 – Diagrama de um *slice* do tipo SLICEM (extraído do Apêndice A).

tação de circuitos combinatórios ou sequenciais numa FPGA. Cada CLB é conectado a uma matriz de interligação para acessar à matriz geral de roteamento, conforme ilustra a Figura 10.

Os dois *slices* existentes em cada CLB não possuem conexão direta entre eles e cada *slice* faz parte de uma coluna de *slices* isolada. Cada *slice* em cada coluna possui um recurso de *carry logic* independente (nos CLBs da família Spartan-6, apenas a coluna de *slices* da esquerda possui este recurso). Os *slices* localizados na base do CLB são chamados de SLICE(0) enquanto os *slices* localizados na parte superior do CLB são chamados de SLICE(1), conforme a denominação do fabricante Xilinx (Xilinx Inc., 2010) (Xilinx Inc., 2012c) (Xilinx Inc., 2012d) (Xilinx Inc., 2013a).

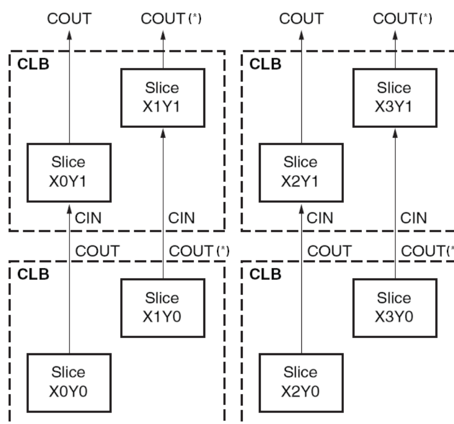
As ferramentas de síntese da Xilinx adotam a seguinte denominação para os *slices*: uma letra "X" seguida de um número identifica a posição de cada *slice* na coluna de *slices*. A letra "Y" seguida de um número identifica a linha onde estão localizados os *slices* (Xilinx Inc., 2010) (Xilinx Inc., 2012c) (Xilinx Inc., 2012d) (Xilinx Inc., 2013a). A



(*) Nos CLBs da família Spartan-6 a coluna de *slices* da direita não possui *carry logic.*

Figura 10 – Organização dos *slices* dentro de um CLB (Xilinx Inc., 2010) (Xilinx Inc., 2012c) (Xilinx Inc., 2012d) (Xilinx Inc., 2013a).

Figura 11 apresenta quatro CLBs localizados no canto inferior esquerdo de uma FPGA, demonstrando as numerações de identificação.



(*) Nos CLBs da família Spartan-6 a coluna de *slices* da direita não possui *carry logic.*

Figura 11 – Organização dos *slices* nas linhas e colunas de CLBs (Xilinx Inc., 2010) (Xilinx Inc., 2012c) (Xilinx Inc., 2012d) (Xilinx Inc., 2013a).

A constituição completa de um CLB depende da gama das FPGAs. A Tabela 1 resume os recursos lógicos disponíveis em cada CLB das famílias de FPGAs Spartan-6, Virtex-5, Virtex-6 e 7 Series. Os CLBs podem ser encontrados em todos os dispositivos destas famílias de FPGAs, em diferentes quantidades (Xilinx Inc., 2010) (Xilinx Inc., 2012c) (Xilinx Inc., 2012d) (Xilinx Inc., 2013a).

Tabela 1 – Recursos existentes num CLB nas várias famílias de FPGAs (Xilinx Inc., 2010) (Xilinx Inc., 2012c) (Xilinx Inc., 2012d) (Xilinx Inc., 2013a)

Família	***Slice***	**LUT**	**Flip-Flop**	***Carry Logic***	**Multiplexador Maior**	**RAM(1) Distribuída**	**SRL(1)**
Spartan-6	2	8	16	1(2)	1(2)	256 bits	128 bits
Virtex-5	2	8	8	2	2	256 bits	128 bits
Virtex-6	2	8	16	2	2	256 bits	128 bits
7 Series	2	8	16	2	2	256 bits	128 bits

(1) Apenas nos SLICEM. SLICEL e SLICEX não possuem RAM distribuída nem registadores de deslocamento SRL.
(2) Apenas nos SLICEM e SLICEL. SLICEX não possui *Carry Logic* nem Multiplexadores maiores.

2.2.4 Secção de Coluna de CLBs

Embora a organização das colunas de CLBs exemplificadas na Figura 11 comecem "Y0" e terminem no topo da matriz de CLBs, estas estão subdivididas em secções que são delimitadas pelas regiões de relógio existentes no dispositivo. O número de CLBs por cada secção depende da família de FPGAs e a sua correspondente memória de configuração é um conjunto de *frames* contíguos que os configura como um todo. Um *frame* é o bloco mínimo de memória possível de endereçar para ler e escrever. A Tabela 2 contém algumas características relativas à formação de um *frame* nas mais recentes famílias de FPGAs da Xilinx.

A última coluna da Tabela 2 indica o número de CLBs em cada secção. Outros tipos de recursos terão um valor diferente, tal e qual como o número de *frames* necessários para configurar cada secção depende da família e do tipo de recurso.

Tabela 2 – Características da organização de configuração em várias famílias de FPGAs (Xilinx Inc., 2015b) (Xilinx Inc., 2012b) (Xilinx Inc., 2015c) (Xilinx Inc., 2015a)

Família	**Bits por Palavra**	**Palavras por *frame***	**Bits por *frame***	**CLBs por Secção**
Spartan-6	16 bits	65	1040	16
Virtex-5	32 bits	41	1312	20
Virtex-6	32 bits	81	2592	40
7 Series	32 bits	101	3232	50

2.2.5 Linha de Secções de Recursos

As secções de recursos são organizadas tal como um grupo de livros ao longo de uma prateleira, onde cada livro é uma secção de recursos. A correspondente memória de configuração é igualmente contígua, começando a configurar a próxima secção após finalizar a anterior. A Figura 12 ajuda a entender esta organização e a correspondência entre secção de recursos e sua memória de configuração.

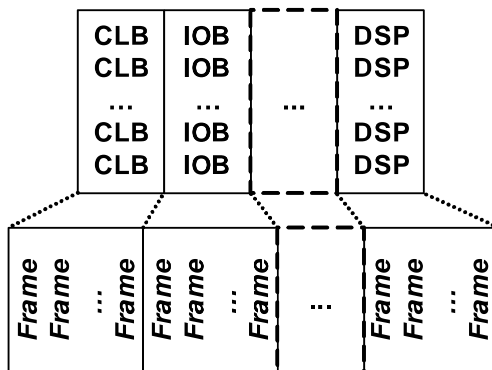


Figura 12 – Organização da memória de configuração ao longo de uma linha (*row*).

Observa-se portanto que uma linha de secções de recursos corresponde a um segmento de memória de configuração contínuo. No total, uma FPGA é constituída por várias destas linhas, numeradas

por ordem crescente, tipicamente a mesma organização de secções de recursos e sempre com o mesmo tamanho de memória de configuração atribuída. A organização geral de toda a memória de configuração de uma FPGA pode ser observada na Figura 13.

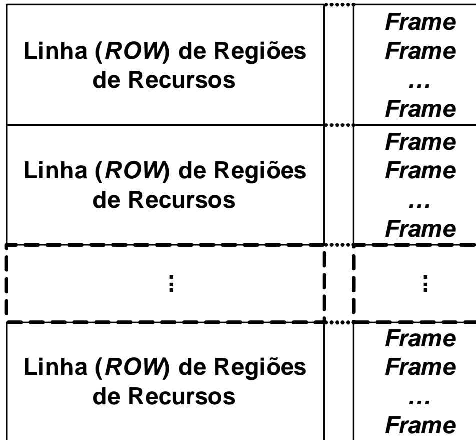


Figura 13 – Organização da memória de configuração de uma FPGA.

Nas famílias Virtex-5, Virtex-6 e 7 Series existe ainda uma hierarquia superior na organização que divide as linhas (*rows*) em dois grupos: o *Top Bit* e o *Bottom Bit* (Xilinx Inc., 2012b) (Xilinx Inc., 2015c) (Xilinx Inc., 2015a).

2.2.6 Endereçamento da Memória de Configuração

O endereçamento da memória de configuração está relacionado com a organização da memória descrita anteriormente. No Apêndice A.3.2 é possível ver com detalhe essa organização numa Spartan-6. Embora semelhante, nas famílias Virtex-5, Virtex-6 e 7 Series, como usam palavras de 32 bits, e têm as linhas de recursos subdivididas em *Top Bit* e *Bottom Bit*, o registo usado para endereçar a memória possui algumas diferenças. Esse registo é designado por *Frame Address Register* (FAR) e pode ser consultado na tabela incluída na Figura 14.

O registo FAR é usado durante as operações de carregamento

Address Type	Bit Index	Description
Block Type	[25:23]	Valid block types are CLB, I/O, CLK (000), block RAM content (001), and CFG_CLB (010). A normal bitstream does not include type 010.
Top/Bottom Bit	22	Select between top-half rows (0) and bottom-half rows (1).
Row Address	[21:17]	Selects the current row. The row addresses increment from center to top and then reset and increment from center to bottom.
Column Address	[16:7]	Selects a major column, such as a column of CLBs. Column addresses start at 0 on the left and increase to the right.
Minor Address	[6:0]	Selects a frame within a major column.

Figura 14 – Descrição do registo de endereçamento dos *frames* (FAR) (Xilinx Inc., 2015a).

de *bit files*, quer sejam de configuração completa ou parcial da FPGA. Numa situação em que se pretenda alterar um determinado parâmetro de configuração de um CLB, o primeiro passo é configurar o campo *Top/Bottom Bit* indicando se o CLB se encontra no grupo superior de linhas ou no inferior. Em seguida colocar no campo *Row Address* o número da linha de secção de recursos correspondente, e no campo *Column Address* a coluna a que a secção de recursos onde se encontra o CLB em questão. Por último, indicar no campo *Minor Address* qual o *frame* onde se encontra a configuração do parâmetro que se deseja alterar. Usando a analogia de uma biblioteca, o campo *Top/Bottom Bit* corresponderia a qual armário (o de cima ou o de baixo), o *Row Address* seria qual prateleira, o *Column Address* qual o livro, e o *Minor Address* qual a página do livro que desejamos ler/escrever.

Quando o objetivo não é ler/escrever um *frame*, mas sim um conjunto de *frames* consecutivos, configura-se o registo FAR com o endereço do primeiro *frame* e procede-se à leitura/escrita desse grupo de *frames* de uma só vez, sem necessidade de configurar o endereço para *frame*.

2.3 CAPACIDADES DE RECONFIGURAÇÃO

Uma das importantes características de uma *Field Programmable Gate Array* (FPGA) é a sua capacidade de atuar como um "hardware em branco". Isto permite-lhe apresentar um maior desempenho do que aplicações em software a ser executadas em processadores de uso geral, e mais flexibilidade do que uma solução *Application Specific Integrated Circuit* (ASIC) com funções fixas. Isto faz com que a capacidade de reconfiguração de uma FPGA seja muito importante quando, por qualquer necessidade técnica, seja preciso proceder a alguma alteração (ou verificação) de alguma secção do seu hardware configurável.

Cada elemento configurável de uma FPGA requer um vetor de bits de armazenamento para manter a configuração definida pelo usuário. Para este fim, são usadas memórias estáticas voláteis. O tamanho desse vetor depende das necessidades de configuração de cada elemento (por exemplo, o número de bits necessários para a seleção de um multiplexador depende do número de entradas que ele possui). Então, uma FPGA é configurada através da carga do aplicativo definido pelo usuário - um *bitstream*[1] (também designado por *bit file*) - na memória de configuração interna da FPGA (Hauck and Dehon, 2008). Uma FPGA pode carregar-se a partir de dispositivos de memória não volátil (normalmente uma memória *flash* existente na periferia da FPGA), ou ainda ser configurada por uma fonte externa, como um microprocessador, microcontroladores, um *Personal Computer* (PC), ou equipamentos de programação e teste. Em qualquer um dos casos, há dois modos genéricos de configurar uma FPGA. O primeiro é recorrer a uma comunicação de dados série, que é usada para minimizar os requisitos de pinos do dispositivo. O segundo é o recurso a uma comunicação paralela de 8 bits, 16 bits ou 32 bits (na família Spartan-6 não existe a opção de 32 bits), que normalmente é utilizada para obter um maior desempenho na operação de configuração (Xilinx Inc., 2015b) (Xilinx Inc., 2012b) (Xilinx Inc., 2015c) (Xilinx

[1] O termo *bitstream* é frequentemente usado para descrever os dados de configuração que devem ser carregados numa FPGA ao iniciar. Os detalhes do formato de um *bitstream* para uma determinada FPGA são normalmente considerados como proprietários de cada fabricante.

Inc., 2015a).

2.3.1 Reconfiguração e Múltiplos *Bitstreams*

Cada fabricante de FPGAs disponibiliza um ou mais mecanismos para efetuar a carga a partir de múltiplos arquivos de *bitstream* (*MultiBoot*) localizados numa mesma memória flash externa. Em alguns fabricantes, este método de carregamento é definido pelo nível lógico de alguns dos seus pinos de entrada e saída, como no caso das FPGAs das famílias Virtex-5, Virtex-6 e 7 Series da Xilinx (no caso da família Spartan-6 esta capacidade tem uma flexibilidade mais reduzida). Nestas famílias de FPGAs a escolha do *bitstream* a ser carregado é feita através dos pinos RS[1:0] (*Revision Select*). Estes pinos da FPGA possuem resistências internas a fazer de *pull-down*, o que provoca que o *bitstream* inicial seja preferencialmente o selecionado pelo endereço RS[1:0]="00", também conhecido como *safe bitstream*. Quando um erro da configuração é detetado, a lógica de configuração gera um sinal interno de reset e o endereço em RS[1:0] é apontado para o valor seguro, "00". A Figura 15 apresenta um exemplo de reconfiguração com múltiplos *bitstreams*. A maneira mais simples de usar os pinos RS com uma memória externa flash (com interface *Byte Peripheral Interface* (BPI)) é conectá-los nas linhas de endereço mais altas dessa mesma memória. No exemplo da Figura 15, o espaço de endereçamento da memória flash é dividido em quatro secções, e cada uma pode ser usada para armazenar um *bitstream* diferente.

Neste género de reconfiguração existem diferentes *bitstreams* armazenados numa memória flash configuraram completamente uma FPGA. A próxima Secção 2.3.2 apresenta a capacidade de Reconfiguração Parcial, com a qual é possível reconfigurar partes da FPGA a partir de múltiplos arquivos de *bitstream*, responsáveis por configurar uma determinada secção de recursos FPGA.

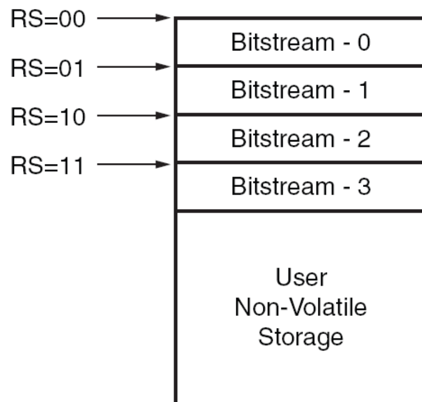


Figura 15 – Espaço de endereçamento para *MultiBoot* de uma memória flash BPI (Xilinx Inc., 2015c).

2.3.2 Reconfiguração Parcial

A reconfiguração parcial, *Partial Reconfiguration* (PR), é a modificação de uma secção de recursos de uma FPGA em operação através do carregamento de um arquivo de configuração, que apenas modifica a secção de recursos desejada. Geralmente é através de um *bit file* (.bit) parcial. Após a carga completa da FPGA, os arquivos *bit file* parciais podem ser carregados para alterar certas regiões dentro da FPGA sem comprometer a integridade de outras aplicações que estão a ser executadas nas demais regiões do dispositivo, conforme apresenta o documento do fabricante Xilinx (Xilinx Inc., 2013e).

A Figura 16 ilustra as premissas adotadas na reconfiguração parcial. Conforme é apresentado, a função implementada no bloco reconfigurável (*Reconfig Block* A) é modificada através da recarga de um dos diversos arquivos .bit parciais disponíveis, A1.bit, A2.bit, A3.bit ou A4.bit. Desta forma, os circuitos lógicos dentro da FPGA são divididos em dois tipos virtuais de lógica: em lógica reconfigurável e em lógica estática. A área em cinzento claro dentro da FPGA da Figura 16 representa a lógica designada como estática, enquanto que a área do bloco de reconfiguração, em cor cinza mais escuro, representa a área de lógica

reconfigurável. Esta área reconfigurável é normalmente designada por partição reconfigurável ou *Reconfigurable Partition* (RP).

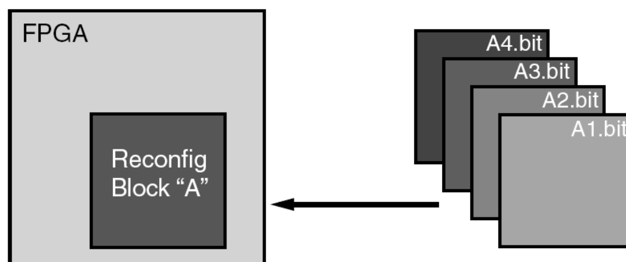


Figura 16 – Diagrama básico de reconfiguração parcial (PR) (Xilinx Inc., 2013e).

As famílias de FPGAs Virtex-5, Virtex-6 e 7 Series têm suporte para reconfiguração parcial através de ferramentas disponibilizados pela Xilinx. As opções podem passar pelo ISE Design Suite, PlanAhead, ou mesmo através de aplicativos que são executados através de linhas de comandos (Xilinx Inc., 2013e). No caso da família Spartan-6, embora permita ser reconfigurada devido às suas diferenças em relação às famílias Virtex, não é suportada pelas ferramentas para a criação de partições reconfiguráveis (RPs) (Xilinx Inc., 2013e).

2.3.3 Reconfiguração Parcial Baseada em Diferenças

A reconfiguração parcial baseada em diferenças é útil para fazer pequenas alterações na configuração de uma FPGA em funcionamento, tais como alterar parâmetros de equações lógicas, alterar parâmetros de filtros digitais e alterar pinos de entrada ou saída do dispositivo, evitando a necessidade de reprogramar toda a FPGA. Este fluxo de projeto não é recomendado para fazer grandes mudanças na funcionalidade ou estrutura de um projeto, como por exemplo, alterar um algoritmo inteiro. Quando existem alterações significativas ou o roteamento deve ser modificado, o fluxo recomendado é realizar as alterações nos arquivos *Hardware Description Language* (HDL) e seguir o fluxo de uma reconfi-

guração parcial apenas, conforme citado no documento do fabricante Xilinx (Xilinx Inc., 2007).

2.4 CONCLUSÃO

Na primeira parte deste capítulo foram apresentados, de uma forma resumida, os aspetos arquiteturais dos dispositivos FPGA (LUTs, *Slices*, CLBs, etc) e mais especificamente, na Secção 2.2, foi apresentada a estrutura (tanto da organização dos recursos como da correspondente memória de configuração), seguida pelas famílias Spartan-6, Virtex-5, Virtex-6 e 7 Series do fabricante Xilinx. Na parte final deste mesmo capítulo foram descritos os modos possíveis de reconfiguração (completa ou parcial) de uma FPGA da Xilinx.

Tendo este trabalho como base o pleno domínio da arquitetura das FPGAs e da sua memória de configuração, de modo a alterar as funcionalidades implementadas numa determinada secção de recursos, faz com que seja importante esta recolha de informações. Isto porque são informações que fornecem conhecimentos técnicos vitais para que neste trabalho se alcancem os melhores resultados para a camada de suporte apresentada na Secção 1.2.

Muita da informação presente neste capítulo foi possível recorrendo à reconfiguração parcial baseada em diferenças (Xilinx Inc., 2007), que permite gerar *bit files* parciais que vão alterar apenas um campo específico de um determinado *slice*. Repetindo de um modo criterioso este processo, foi alcançada uma engenharia inversa, que comparando *bit files* parciais, permitiu observar que bits são responsáveis por configurar o campo alterado e a que *frame* pertencem. Os resultados após realizar esta engenharia inversa são relatados no Apêndice A e foram fundamentais para a clarificação de como criar a camada de suporte, de modo a alcançar o objetivo chave desta proposta.

O próximo capítulo abordará o estado da arte e nele serão apresentados trabalhos em que pelo menos numa parte da abordagem seguida, possuem semelhanças com o que é proposto nesta tese.

3 ESTADO DA ARTE

Ao nível do mar, o tempo médio para que ocorra uma falta permanente, que provoque um erro e origine uma falha (*Mean Time to Failure* - MTTF) (Koren and Krishina, 2007) num sistema implementado numa FPGA, mesmo com as tecnologias nanométricas atuais (Xilinx Inc., 2016b) ultrapassa muitas vezes o período de vida previsto do equipamento comercial que inclui a FPGA. Por essa razão, não existe uma efetiva necessidade de munir as funções implementadas neste tipo de dispositivos de estratégias que as tornem tolerantes a faltas permanentes. Esta política de um relativo desprezo pela ocorrência de faltas permanentes é completamente posta de parte quando a área de aplicações seja a espacial, onde o ambiente de operação é muito mais hostil e daí aumentar a probabilidade de ocorrência de faltas permanentes, para além de que a operação de efetuar uma simples troca de uma FPGA não é, muitas vezes, uma opção.

Nas aplicações espaciais, devido às suas especificidades, o tratamento das faltas é por isso ainda mais rigoroso. Como exemplo disso, a *European Space Agency* (ESA) conta já com dois estudos na área de *Fault Detection, Isolation and Recovery* (FDIR): um chamado SMART-FDIR, coordenado pela Alenia Spazio com a colaboração do Politecnico di Milano (Guiotto et al., 2003), e outro com o nome *Advanced Fault Detection, Isolation and Recovery* (AFDIR), realizado pela ESA em parceria com a Astrium, de França, e da Space Systems Finland Ltd (Holsti and Paakko, 2001).

Tal como a sigla FDIR indica, estes estudos focam-se em técnicas (ou métodos) que permitem realizar a deteção de faltas, o seu isolamento após a sua identificação e recuperação do normal funcionamento de todo o sistema. É com o FDIR em mente (principalmente o isolamento e a recuperação), e sabendo que devido à radiação ou envelhecimento, um problema físico possa dar origem a uma falta permanente (Avizienis et al., 2004), que foi orientado este estudo do estado da arte. A prevenção para evitar o acelerado envelhecimento é outro ponto importante deste estudo.

3.1 TESTE PARA DETEÇÃO DE FALTAS PERMANENTES EM FPGAS

Embora o teste para deteção de faltas seja uma funcionalidade necessária a incluir na solução desenvolvida nesta tese, no decorrer da mesma observou-se que podem existir várias opções para desempenhar esta função, e que dependente da arquitetura de todo o sistema e da cobertura de faltas desejada, assim deverá ser tomada esse decisão. No entanto, na fase inicial da pesquisa e aproveitando os resultados do estudo do Apêndice A, foi desenvolvida uma proposta para teste da área de recursos utilizada por um determinado módulo (Martins et al., 2014a).

No caso dos ASICs já foram desenvolvidas soluções que conduziram à inclusão de estruturas de auto-teste interno e de técnicas de conceção orientadas para a testabilidade que facilitassem a realização do teste e permitissem manter o seu custo dentro de valores aceitáveis (Isermann, 2006). Entretanto, o aparecimento de dispositivos lógicos programáveis veio colocar novos desafios às técnicas existentes. Na realidade, deixámos de ter um componente cuja funcionalidade estava perfeitamente definida para passarmos a ter um conjunto de recursos que podem assumir "qualquer" função. O alvo do teste deixa, neste caso, de ser funcional e passa a ser estrutural. O desconhecimento da estrutura de implementação dos blocos que constituem a FPGA obriga à adoção de um modelo de faltas híbrido para o seu teste estrutural. A necessidade de cobrir todos os modos de operação implica o uso de várias configurações de teste (Dutton and Stroud, 2009a) (Dutton and Stroud, 2009b). A possibilidade de reconfiguração parcial dinâmica veio acrescentar novos desafios ao problema do teste das FPGAs, mas possibilitou igualmente a conceção de novas soluções para resolvê-lo (Jamuna and Agrawal, 2012) (Parreira et al., 2003) (Parreira et al., 2006).

3.1.1 Faltas Características das FPGAs

Fabricantes de FPGA como a Xilinx caracterizaram grande parte das faltas que ocorrem nos seus dispositivos como SEEs. Os SEEs

resultam de uma interação de uma partícula com um nível alto de energia com elementos do circuito num dispositivo. Por outras palavras, quando uma partícula ionizada de elevada energia atinge um circuito integrado, esse evento provoca a libertação de uma série de eletrões à medida que, penetrando o substrato do semicondutor, vai perdendo energia. (White, 2012).

As origens dos SEEs são três. A primeira, que afeta sobretudo os dispositivos que, como as FPGAs, têm por base funcional uma matriz de memória do tipo RAM, é a radiação cósmica oriunda da atmosfera, pois este tipo de memória é bastante suscetível a este tipo de interferências. A segunda é a impureza nos materiais usados no encapsulamento dos circuitos integrados. Por vezes, entre esses materiais existem vestígios de urânio e isótopos de tório, os quais emitem partículas *alpha* que podem provocar SEE no substrato de silício. E a terceira é a impureza do próprio substrato, nomeadamente o elemento boro usado no *Borophospho-Silicate Glass* (BPSG), que é igualmente uma fonte de radiação ionizante (White, 2012).

A Figura 17 mostra os vários tipos de SEEs que existem, e como eles estão divididos em duas amplas categorias. São elas: *Soft Errors* e *Hard Errors*.

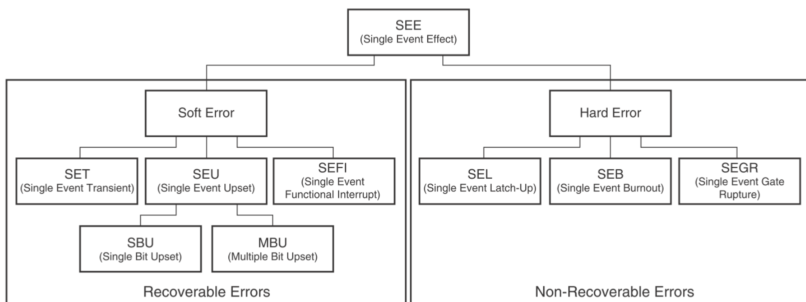


Figura 17 – Tipos de SEEs que podem ocorrer numa FPGA (White, 2012).

Soft Errors são faltas, que embora alterem a normal atividade do circuito, não o danificam fisicamente e é possível restaurar o seu

correto funcionamento. *Hard Errors* são quando ocorre uma destruição física do circuito irreversível e que dá origem a falta permanente.

Relativamente aos *Soft Errors*, estes subdividem-se da seguinte forma:

- *Single-Event Transient* (SET), impulso transitório, com algumas centenas de picosegundos de duração (o valor exato depende das características do circuito e da energia da partícula), originado pelo excesso de carga, quando a zona de impacto se situa perto de ou num nó do circuito. Se o comprimento do impulso for superior ao tempo de transição da lógica presente no circuito, o impulso não é atenuado, mesmo que a partícula possua uma energia baixa. Se o nó de impacto é um nó de dados de um circuito de retenção, a tensão induzida força momentaneamente no circuito um estado instável. Se a lógica falhar, o restabelecimento do estado anterior, o circuito assumirá eventualmente um novo valor lógico (ocorrência de uma falta transitória - *transient fault*) (Rockett, 2001) (Alexandrescu et al., 2004);

- *Single-Event Upset* (SEU), alteração dos valores armazenados em elementos de memória da FPGA (SRAM, flip-flops, memória de configuração), que enquanto não forem corrigidos, o circuito implementado manterá a presença das respetivas faltas. A sensibilidade aos SEUs encontra-se por isso profundamente correlacionada com a quantidade de elementos de memória presentes no circuito, tais como flip-flops e células de memória (Legat et al., 2010). Dada a diminuição do tamanho dos transístores e da sua tensão de alimentação, a quantidade de carga necessária para alterar o seu estado também se reduziu, pelo que a sua suscetibilidade ao impacto de partículas deste género aumentou. Quando um SEU ocorre, este pode apenas provocar a alteração de um bit, *Single Bit Upset* (SBU), ou de vários bits, *Multiple Bit Upset* (MBU);

- *Single-Event Function Interrupt* (SEFI), quando acontece uma interrupção do normal funcionamento do circuito implementado no

dispositivo (para lá do simples corrompimento dos dados existentes no circuito). Quando este tipo de efeito acontece, por norma, exige a reprogramação da FPGA, reiniciar ou mesmo desligar/ligar a alimentação do dispositivo, para que este recupere o seu normal funcionamento.

Responsáveis pelas faltas permanentes, os *Hard Errors* encontram-se subdivididos nas seguintes três subclasses:

- *Single-Event Latch-up* (SEL), ocorre quando um bloqueio do circuito é induzido pela radiação. Este bloqueio tanto poderá ser permanente, como poderá ser eliminado através do ligar/desligar da alimentação do circuito;

- *Single-Event Burnout* (SEB), é um curto-circuito causado pelo impacto de um ião com um nível alto de energia num transístor, que provoca uma polarização indesejada. Estes eventos são tipicamente ameaças para *Metal Oxide Semiconductor Field Effect Transistors* (MOSFETs), embora também o sejam para *Insulated Gate Bipolar Transistors* (IGBTs), díodos de alta voltagem e outros circuitos similares;

- *Single-Event Gate Rupture* (SEGR), consiste num plasma cravado, causado pelo impacto de um ião com um nível alto de energia, resultando na rutura do isolamento de óxido da *gate*.

Para além dos SEEs, existem mais três tipos de efeitos causados pela radiação (Xilinx Inc., 2014b):

- *Total Ionizing Dose* (TID), quando a acumulação de radiação numa determinada parte do circuito atinge um limite, para além do qual essa mesma parte deixa de ser tolerante a este tipo de radiação. Embora os circuitos tenham tolerância à radiação, têm ao mesmo tempo um efeito de memória em relação à quantidade de radiação que podem absorver ao longo do seu ciclo de vida. Quando esse limite é excedido, faltas permanentes começam a

ocorrer nas partes do circuito onde o valor foi ultrapassado (Zhang et al., 2014b);

- *Prompt Dose/Rate Effects*, são efeitos provocados quando uma dose de radiação não natural atinge um determinado patamar. O estudo da influência deste tipo de radiação nos circuitos é realizado pelos laboratórios da Xilinx, em cooperação com agências do governo dos Estados Unidos da América e principais companhias de defesa. Por essa razão, os respetivos dados não são públicos;

- *Neutron Single Event Upset* (NSEU), semelhante ao SEU, com a diferença de que é causado por neutrões atmosféricos. Embora este efeitos não sejam considerados ao nível de radiação espacial, quando se trata de aplicações terrestres ou atmosféricas, são devidamente analisados e caracterizados.

Das duas categorias dos SEEs, a fonte de faltas mais preponderante, a que tem uma maior probabilidade de ocorrer são os *Soft Errors*, em particular a subclasse SEU (os SBUs, por exemplo, são os SEEs mais analisados nas aplicações de aviação).

No passado, este problema era uma preocupação somente em ambientes hostis como o espaço sideral, mas os *Integrated Circuits* (ICs) baseados em tecnologias nanométricas, tornaram-se tão sensíveis, que mesmo a radiação ao nível do mar origina níveis inaceitáveis deste tipo de faltas transitórias (White, 2012).

Duas técnicas para a recuperação de erros provocados por SEUs na memória de configuração das FPGAs, embora com algumas limitações, foram propostas no passado em (Carmichael et al., 2000) e em (Huang and McCluskey, 2001). Essas limitações prendem-se com a possibilidade de as LUTs poderem ser usadas como memórias distribuídas, situação em que, em caso de deteção de um erro, se o vetor para a sua correção abranger uma dessas LUTs, a sua correção obriga à paragem do sistema como única forma de garantir a coerência dos dados nela constantes. Mais recentemente, a fabricante de FPGAs Xilinx disponibilizou soluções como (Carmichael and Tseng, 2009) ou o *LogiCORE IP Soft Error*

Mitigation Controller (SEM) (Xilinx Inc., 2011a). O SEM permite, com a exceção de recursos da FPGA, que estejam a ser utilizados como memória, verificar e recuperar bits errados causados por SEUs, sem que seja perturbado o normal funcionamento do sistema implementado na FPGA. A pensar nos problemas originados pelos SEUs, as FPGAs da Xilinx têm já incluídas duas estratégias para os detetar. São elas o *Readback CRC*, que verifica continuamente em *background* se algum SEU origina um *bit flip* na memória de configuração da FPGA, que provoque uma alteração no respetivo *Cyclic Redundancy Check* (CRC), e a existência de um campo de *Error Correction Code* (ECC) em cada *frame* (secção mínima em que é dividida a memória de configuração), que permite a deteção e localização de *bit flips* no *frame* (Xilinx Inc., 2015b) (Xilinx Inc., 2012b) (Xilinx Inc., 2015c). Para auxiliar a verificação do ECC em cada *frame*, as famílias mais recentes de FPGAs da Xilinx possuem um hardware específico para essa finalidade, que pode ser instanciado pelo projetista através da primitiva FRAME_ECC_VIRTEXx (Xilinx Inc., 2012b) (Xilinx Inc., 2015c).

Embora com uma probabilidade bem mais reduzida de ocorrer, os *Hard Errors* quando acontecem, são para sempre. Significa isto que se a localização do circuito na FPGA for estática e este sofrer de uma falta permanente (*Hard Error*), o circuito poderá nunca mais voltar a desempenhar a(s) função(ões) para a(s) qual(quais) foi projetado. Os *Hard Errors* são por esta razão o principal objeto desta tese.

3.1.2 Modelos de Faltas

A realização de um teste para deteção de faltas tem sempre associado um modelo de faltas. Tendo a tecnologia no processo de fabrico evoluído para escalas nanométricas, aliada ao consequente uso de tensões de alimentação cada vez mais baixas, acaba invariavelmente por tornar a FPGA cada vez mais sensível a pequenos defeitos físicos que possam ocorrer. As previsões expressas no *International Technology*

Roadmap for Semiconductors (ITRS)[1] (for Semiconductors (ITRS), 2011) para os próximos 15 anos acentuam ainda mais esta tendência, na tentativa de responder aos desafios implícitos na Lei de Moore (Moore, 1998). Tal facto traz consigo novos tipos de defeitos físicos (ou altera a probabilidade para um valor mais relevante), que obrigue a vários modelos de faltas. No entanto, considerando neste trabalho que as faltas permanentes podem ser causadas por *Hard Errors* (Figura 17) ou *aging* devido a BTI, estas poderão influenciar o circuito implementado na FPGA de duas formas:

1. Um defeito físico provocar um valor lógico fixo num determinado ponto do circuito digital implementado. Ou seja, o modelo de faltas do tipo *stuck-at-0* e *stuck-at-1*, que tanto podem ter origem na radiação a que o dispositivo foi sujeito, como serem devidas ao envelhecimento do mesmo (Abramovici et al., 1994);

2. Um problema físico atrasar a propagação do sinal ao longo de todo o comprimento da linha, sendo geralmente utilizado para isso o modelo de faltas de atraso de propagação (*delay fault model*). Este modelo estende este conceito da consideração do atraso de transição, introduzido por uma única porta lógica (*gate delay*), à soma de todos os atrasos introduzidos na propagação de um determinado sinal (*path delay*).

Contudo, se o circuito for implementado na FPGA de modo a que o recurso com defeito não seja utilizado, ou o acréscimo de atraso a ele devido não faça violar nenhum atraso máximo permitido, então existe a falta mas não a falha, uma vez que a falta não é suscetível de impedir o funcionamento correto do circuito. Este tipo de faltas podem ser classificadas como não detetáveis, redundantes ou não-testáveis, e podem ou não ser contabilizadas na métrica de deteção de faltas (Avizienis et al., 2004).

[1] O *International Technology Roadmap for Semiconductors* (ITRS) resulta de um consenso entre as indústrias de semicondutores quanto à estimativa das suas necessidades de investigação e desenvolvimento num horizonte de 15 anos.

3.1.3 Conclusão

Muitas das origens das faltas permanentes são comuns tanto num ASIC como numa FPGA. No entanto, no caso da FPGA, existe a particularidade de que a implementação do sistema no dispositivo é realizada através da configuração de uma memória interna. Isto faz com que exista um conjunto de faltas características das FPGAs, e ao mesmo tempo novas formas de realizar o teste nestes dispositivos.

3.2 INFLUÊNCIA DA RADIAÇÃO TID NAS FPGAS NAS APLICAÇÕES ESPACIAIS

Desde que a tecnologia dos dispositivos eletrónicos usados em satélites baixou dos 180nm, estes tornaram-se muito sensíveis ao stress da radiação existente no espaço (Wang et al., 2016). Uma das sensibilidades mais comuns que deve ser tida em conta para que a confiabilidade dos sistemas existentes em satélites não seja colocada em causa é a *Total Ionizing Dose* (TID), algo que levou mesmo a *European Cooperation for Space Standardization* (ECSS) a elaborar um documento nesse sentido (for Space Standardization (ECSS), 2008).

Fisicamente, um circuito integrado ao ser exposto à radiação por um longo período de tempo sofre alterações nas suas características, o que pode resultar numa deterioração paramétrica ou uma falha funcional. Ou seja, pode colocar a sua estabilidade em causa (que tanto poder ser pela via do aumentando o seu tempo de atraso da resposta como pela ocorrência de respostas erradas) (Boudenot, 2007) (Zhang et al., 2014b).

A TID atinge inicialmente as camadas isolantes, que podem prender carga ou apresentar mudanças em sua interface (Leroy and Rancoita, 2009). O cúmulo de cargas nos óxidos de isolação, induzido por radiação ionizante provoca o incremento da corrente de fuga após a irradiação (Zhang et al., 2014b).

Quando um transístor do tipo MOS é sujeito a radiação ionizante de alta energia, são gerados pares de eletrão-buraco no óxido. Esta

geração leva a quase todos os efeitos da TID. Os portadores gerados provocam a acumulação de carga, o que pode levar à deterioração do dispositivo (Huang et al., 2014).

Focado na observação do efeito da radiação nos semicondutores usados em satélites, recorrendo para isso a uma FPGA, uma equipa do Central Laboratory pertencente ao *Beijing San-Talking Testing Engineering Academy Co.* analisou e elaborou um método preditivo relativo à influencia da radiação na confiabilidade de sistemas incluídos em satélites em diferentes orbitas (Wang et al., 2016). Ainda tendo como alvo a análise do efeito da radiação em aplicações espaciais, no decorrer desta tese foi igualmente planeado um experimento para caracterizar a verdadeira influencia da radiação em FPGAs comerciais usadas em CubeSats (Martins et al., 2016). Nesse experimento um circuito implementado na FPGA deteta a ocorrência de SEEs, tanto na memória de configuração como nos flip-flops existentes nos CLBs, e regista o acontecimento em conjunto com a seguinte informação: localização do recurso onde ocorreu a falta; data e hora da ocorrência; posição orbital em relação ao planeta terra; intensidade solar que os painéis solares se encontravam a receber; e níveis de tensão e corrente aos terminais da bateria.

Embora a interferência da radiação nos recursos de uma FPGA seja conhecida (Xilinx Inc., 2012a), num projeto que reuniu grupos da Universidade Federal de Santa Catarina (UFSC) em Florianópolis, da Universidade de São Paulo (USP) e da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) em Porto Alegre, foi desenvolvida uma plataforma para estudar o efeito da radiação TID (que pode ser também conjugada com interferência eletromagnética) numa FPGA da Xilinx, existente nessa mesma plataforma (Benfica et al., 2011) (Benfica et al., 2012).

No processo de estudo, várias FPGAs foram submetidas à irradiação de raios gama, cada uma com um tempo de exposição diferente, de modo a receber diferentes doses de radiação TID. Depois foi analisada a relação entre a radiação TID e o atraso na propagação dos sinais dentro da FPGA. Para isso, foi implementada em cada FPGA uma cadeia com

10000 inversores, de modo a que fosse possível medir diferenças. Os resultados obtidos são apresentados na Figura 18.

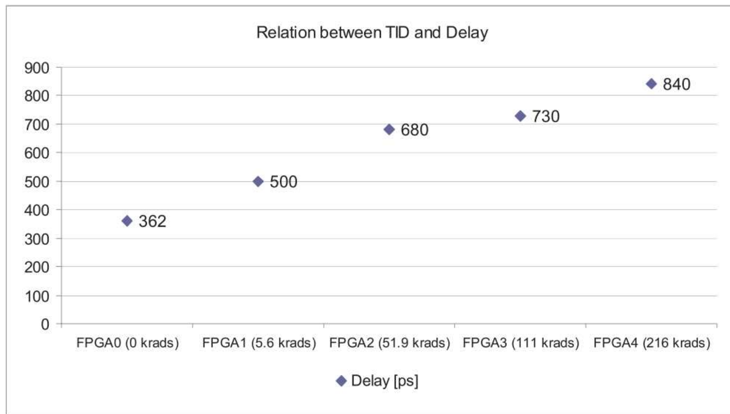


Figura 18 – Resultados experimentais que mostram a relação entre a radiação TID e o atraso (Benfica et al., 2011).

Os resultados obtidos serviram para mostrar que embora a quantidade de radiação TID não tenha sido suficiente para originar qualquer falta permanente, esta mesma radiação foi responsável por um incremento do atraso, que no pior caso atingiu os 132%. Ou seja, esta experiência mostra que a deterioração dos recursos provocados por radiação TID, pode levar a que uma determinada área de recursos de uma FPGA possa ficar impedida de implementar um determinado módulo do sistema, devido a deixar de cumprir os respetivos requisitos temporais.

3.2.1 Conclusão

Sendo um dos objetivos desta tese dotar um sistema da capacidade de este se auto-recuperar de faltas permanentes, trabalhos como estes demonstram a interferência da radiação TID no funcionamento das FPGAs, tanto em laboratório como em orbita do planeta terra. Por esta razão, é importante que uma plataforma tolerante a faltas permanentes tenha em consideração igualmente o problema do aumento do atraso

na propagação.

3.3 ENVELHECIMENTO (*AGING*) DEVIDO AO NBTI E PBTI

Uma das principais ameaças à confiabilidade a ser considerada em tecnologias como o CMOS, ou mais recentemente o *Fin Field-Effect-Transistor* (FinFET), é o *Bias Temperature Instability* (BTI) que é um resultado físico/químico que provoca uma degradação do óxido, resultando em um desvio da tensão de *threshold* (V_T) ao longo do tempo e consequente diminuição na corrente de dreno e na transcondutância (Alam and Mahapatra, 2005) (Shen et al., 2006). As tecnologias nanométricas recentes continuam a lutar contra este efeito (Liao et al., 2008) (Wang and Xu, 2014), e a radiação, que por si só já é uma ameaça devido à destruição que pode causar, acaba por igualmente acelerar o desvio da tensão da *threshold* devido aos seus efeitos (Benfica et al., 2012).

Existem duas formas distintas de BTI: o *Negative Bias Temperature Instability* (NBTI), que afeta os transístores pMOS e o *Positive Bias Temperature Instability* (PBTI) que afeta os transístores nMOS. Faltas originadas pelo BTI não danificam radicalmente os recursos físicos da FPGA, mas eles aumentam o seu tempo de resposta (Mishra et al., 2012). O NBTI tem tido um impacto maior sobre a tecnologia atual (Mishra et al., 2012) (Wang and Xu, 2014), com uma variação que numa tecnologia industrial de 45nm pode atingir os 9% ao fim de dois anos no valor da tensão *threshold* (V_T) dos transístores pMOS durante a operação do circuito (podendo essa variação ser superior para temperaturas mais elevadas na ordem dos $125°C$) (Mishra et al., 2012) (Ceratti et al., 2012). Ainda assim, nas novas tecnologias, a influência do PBTI passou a ser igualmente considerada (Mizubayashi et al., 2015).

3.3.1 Ameaças e Características do NBTI e PBTI

Em relação às mais recentes tecnologias, o NBTI é uma ameaça bem conhecida por ser responsável pela degradação do óxido da porta (*gate*), consequência do aparecimento de novos dadores de eletrões

(conhecidos por *interface traps*) no canal, que são no fundo obstáculos à condução no semicondutor que se formam na interface entre o Silício e o Dióxido de Silício ($Si - SiO_2$) (Alam and Mahapatra, 2005) (Shen et al., 2006) (Liao et al., 2008) (Wang and Xu, 2014). Um *interface trap* é criado quando uma tensão negativa é aplicada ao transístor pMOS durante um tempo prolongado (Mishra et al., 2012). Como mostra a Figura 19, há dois componentes de NBTI com base nos *interface traps* pré-existentes e a criação de novos. Estes são designados como temporário/recuperável (*Recoverable Part*) e permanente (*Permanent Part*). O NBTI temporário é a parte da tensão de *threshold* (V_T) que varia ΔV_{th} e que pode ser reduzido de novo após a alternância da fase de stresse (*Stress Phase*) para a fase de recuperação (*Recovery Phase*).

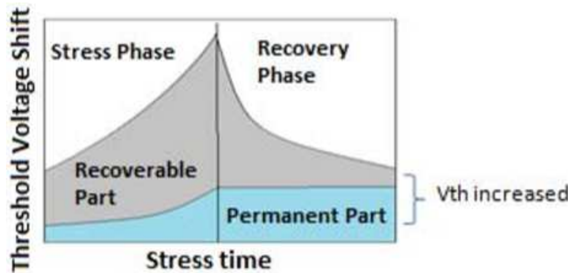


Figura 19 – Componentes Temporária e Permanente do NBTI (Mishra et al., 2012).

Uma variação na tensão de *threshold* (V_T) ΔV_{th} de um transístor pMOS é proporcional à geração de *interface traps* devido ao NBTI, que pode ser expresso como,

$$\Delta V_{th} = (1+m)\frac{qN_{it}(t)}{C_{OX}} \tag{3.1}$$

Em que m representa as alterações V_T equivalentes, devido à degradação da mobilidade (ou parâmetro do modelo), q é a carga elétrica, $N_{it}(t)$ é a geração de *interface traps*, que é o fator mais importante na degradação do desempenho de avaliação devido ao NBTI, e C_{OX} é a capacitância do óxido da porta (*gate*).

Considerando as duas situações de BTI (NBTI nos transístores pMOS e PBTI nos transístores nMOS), ambos os resultados nos atrasos do circuito aumentam com o tempo. Dois modelos importantes da literatura (o modelo *Reaction-Diffusion* (Chakravarthi et al., 2004) e o modelo Charge-Trapping (CT) (Silva and Wirth, 2010)) descrevem a degradação temporal na tensão de *threshold* (como indicado na equação 3.1) nas portas pMOS e nMOS ao longo do tempo t, $\Delta V_{th,p}(t)$ e $\Delta V_{th,n}(t)$ respetivamente, da seguinte forma:

$$\Delta V_{th,p}(t) = K_1\,\xi_1\,f(t) = c_1 f(t) \tag{3.2a}$$

$$\Delta V_{th,n}(t) = K_2\,\xi_2\,f(t) = c_2 f(t) \tag{3.2b}$$

Onde ξ_1 e ξ_2 são a probabilidade de stress das portas pMOS e nMOS (Agarwal et al., 2008) respetivamente, e K_1 e K_2 são constantes que dependem da temperatura e da tensão de alimentação indicadas no correspondente modelo de envelhecimento ((Chakravarthi et al., 2004) ou (Silva and Wirth, 2010)). A função $f(t)$ também depende do modelo e por isso pode ser diferente para portas pMOS e nMOS.

Em relação ao atraso da porta lógica $D(t)$, este pode ser descrito pela função $D(t) = D(t_0) + S\Delta V_{th}(t)$ (para simplificação deixou de se realçar as equações individuais para os casos p e n). Quando $D(t_0)$ é o atraso nominal da porta e S é a sua sensibilidade de atraso para alterações no V_{th} calculado no V_{th} nominal. Usando o ΔV_{th} do par de equações 3.2 e definindo agora que $k = Sc$, temos:

$$D(t) = D(t_0) + kf(t) \tag{3.3}$$

Se as condições de stress de temperatura, tensão de alimentação, e *duty cycle* forem fixas, o atraso é uma função do tempo e é possível calcular a duração do atraso nominal e a sensibilidade à variação de V_{th} para cada porta caracterizada.

3.3.2 Conclusão

Embora tipicamente o envelhecimento (*aging*) devido ao BTI não destrua a funcionalidade dos recursos físicos de uma FPGA, estes tenderão a incrementar o tempo de resposta ao longo do ciclo de vida. No caso do NBTI, que tem sido mais combatido no passado devido à sua maior influência em relação ao PBTI, o incremento da parte permanente (*Permanent Part* na Figura 19) da tensão de *threshold* V_{th} é estimado como sendo de 5-15% por ano (Ceratti et al., 2012). Nas mais recentes tecnologias, a influência do PBTI na ΔV_{th} é análoga (Mizubayashi et al., 2015). Como demonstra a equação 3.3, ao incremento de V_{th} numa porta, corresponde um aumento no tempo de atraso dessa mesma porta. A soma de todos os pequenos incrementos de atraso podem provocar o não cumprimento temporal de um caminho crítico, ou mesmo fazer emergir caminhos críticos que anteriormente não o eram. Portanto, é essencial providenciar estratégias que reduzam a progressão da componente permanente (*Permanent Part*) ao longo do tempo. Para ASICs existem já exemplos de trabalhos que combatem esta evolução não recuperável (Palermo et al., 2015), mas não existem propostas com o mesmo objetivo no universo das FPGAs.

3.4 REALOCAÇÃO DE MÓDULOS EM FPGA

A FPGA permite criar qualquer quantidade de tarefas específicas, desde que elas não excedam a quantidade de recursos existente na FPGA. Como com a *Partial Reconfiguration* (PR) disponível em dispositivos, tais como, as famílias Virtex da Xilinx, é possível alterar componentes do sistema sem interromper o funcionamento dos restantes componentes, significa que com o uso da PR, só a quantidade de tarefas específicas em execução é limitada pela quantidade de recursos. O número total de tarefas pode ultrapassar essa quantidade. A PR fornece por isso uma capacidade de extensão dos recursos existentes num dispositivo.

No entanto, o fluxo permitido pelas ferramentas da Xilinx

gera diferentes *bitstreams* parciais para cada partição reconfigurável, *Reconfigurable Partition* (RP), inclusive no cenário de ser o mesmo componente. Isto significa que são necessários $N \times M$ *bitstreams* parciais para implementar M componentes em N RPs. Esta restrição obriga a uma maior necessidade de memória para albergar todos os *bitstreams* parciais. Por essa razão, é essencial encontrar e automatizar uma forma de gerar *bitstreams* parciais que possam ser alocadas em múltiplas RPs.

Ferramentas de reconfiguração parcial como o PlanAhead da Xinlix (Xilinx Inc., 2013d) permitem criar RPs individuais e gerar um *bitstream* parcial para cada módulo em cada RP. Esta individualidade acontece, basicamente, porque:

- normalmente o roteamento entre recursos externos à RP passa pelo interior da mesma. Tal facto faz com que o *bitstream* parcial possua, para além da implementação do módulo instanciado na RP, roteamento do sistema externo à partição;
- não há controlo sobre o roteamento que faz o interface entre o módulo implementado na RP e o restante sistema implementado na FPGA. Como resultado, dois módulos com o mesmo interface, ficam com roteamentos relativos distintos.

Sendo o objetivo principal desta tese tirar vantagem do facto do hardware implementado numa FPGA SRAM ser feito através do conteúdo de uma memória de configuração para daí o sistema conseguir recuperar de falas permanentes, encontrar uma forma que permita realocar qualquer módulo numa qualquer partição através de uma cópia de uma zona de memória, é algo que pode favorecer a eficiência na construção de uma plataforma focada na recuperação de faltas permanentes.

3.4.1 Abordagens para Realocação de Módulos

De modo a resolver a individualidade de gerar um *bitstream* parcial para cada módulo em cada RP, alguns trabalhos com tópicos relacionados foram já publicados. No artigo (Ichinomiya et al., 2012b) é

resumida uma lista de operações que permitem dotar RPs de flexibilidade no uso dos seus *bitstreams* parciais. Exige que o fluxo **Translation** → **Mapping** → **Placement and Routing** seja percorrido mais duas vezes. Uma para usar no final o comando “PR2UCF” que retira a localização dos *Proxy Logics*[2], e outra para utilizar o Direct Routing Constraints do Xilinx FPGA Editor, em modo gráfico (manual), para retirar o roteamento dos sinais que interligam as RPs ao resto do sistema. Embora apresente uma solução para o roteamento de sinais que ligam saídas das RPs ao restante sistema, essa solução só é aplicável no caso em que o $fanout = 1$. Nada é referido relativamente ao roteamento do restante sistema que cruza as RPs. E não apresenta soluções para o roteamento dos sinais que ligam às entradas das RPs, nem para os sinais de relógio.

Os mesmos autores apresentaram o trabalho (Ichinomiya et al., 2012a). Embora muito semelhante ao anterior (Ichinomiya et al., 2012b), neste o autor refere que o problema do roteamento de sinais que cruzam as RPs é resolvido manualmente através do Xilinx FPGA Editor. Desenvolveram o módulo *Intellectual Property Interface* (IPIF), que obriga todos os sinais de entrada e saída da RP a passarem por uma determinada secção de recursos da FPGA. Isto, para além de permitir o controlo dos sinais de I/O, reduz a probabilidade do roteamento de sinais exteriores ao componente implementado cruzarem a RP. No entanto, este módulo adicionado restringe ainda mais a liberdade das ferramentas usadas no fluxo seguido. Além disso, o problema de sinais cruzarem RPs foi apenas reduzido, não eliminado.

Uma outra abordagem interessante desenvolvida pelo grupo *Hardware/Software Co-Design* do Departamento da Ciência da Computação da Universidade de Erlangen-Nuremberg é o GoAhead (Beckhoff et al., 2012) (Beckhoff et al., 2013b) (Beckhoff et al., 2013a). Esta

[2] *Proxy Logic* é um elemento LUT1 (uma LUT configurada como tendo apenas uma entrada) inserido automaticamente pela ferramenta para cada sinal de interface do módulo implementado na partição reconfigurável (*Pin Partition*), exceto para ligações de rotas dedicadas como sinais de relógio. *Proxy Logic* é necessário para ser um ponto fixo, conhecido como um interface entre a lógica estática e reconfigurável (Xilinx Inc., 2013e).

ferramenta vem no seguimento da anterior ReCoBus-Builder (Koch et al., 2008) que suportava as famílias de FPGA da Xilinx anteriores à Virtex-4. Trata-se efetivamente de uma nova ferramenta e não apenas de uma manipulação no fluxo das ferramentas da Xilinx. No lugar de recorrer a *constraints* para forçar um determinado *Placement and Routing*, manipulam o XDL, obtido no final do *Mapping*, de modo a que a ferramenta *Placement and Routing* só tenha uma opção disponível para uma determinada localização de um recurso ou roteamento. O facto de ser um fluxo com uma ferramenta nova, pode levantar problemas a nível da verificação do próprio fluxo. Na análise do autor do trabalho (Drahonovský et al., 2013), o GoAhead requer repetidas conversões entre formatos XDL e NCD. Estas conversões exigem tempos de implementação elevados quando comparados com os necessários no normal fluxo disponibilizado pelo fabricante, o que faz com que não seja indicada para FPGAs com maior densidade de recursos. Para além disso, em FPGAs recentes, a completa obtenção do formato XDL não é garantida, o que indica que este formato possa estar a ser abandonado pela Xilinx.

No *LE2I Laboratory* da Universidade de Burgundy foi desenvolvida a metodologia OORBIT (*Offline/Online Relocation of Bitstreams*) (Touiza et al., 2012). Baseada na ferramenta ReCoBus-Builder (Koch et al., 2008), esta metodologia, após correr pelo menos uma vez o fluxo **Translation** → **Mapping** → **Placement and Routing**, obtém a localização dos *Proxy Logics* (tudo indica que OORBIT chama o comando "PR2UCF" em *background* para retirar esta informação do ficheiro NCD). Com esta informação gera um novo conjunto de *constraints* onde realoca a posição dos *Proxy Logics* de modo a que tenham todos a mesma localização relativa nas várias RPs, e bloqueia a posição de *Hard Macros* inseridos previamente, para garantir o mesmo roteamento relativo na fronteira entre a parte estática do sistema e as RPs. Relativamente aos sinais de relógio, nada é mencionado nesta metodologia, mas é provável que tal seja feito por intermédio de alguma *Hard Macro*. A solução de *Hard Macros*, embora eficiente, limita logo à partida bastante liberdade no roteamento de todo o sistema. Esta

metodologia não só garante a compatibilidade dos *bitstreams* parciais gerados nas várias RPs, como prepara os *bitstreams* de modo a que o processo de realocação seja mais eficiente, tal como é demonstrado no sistema com realocação multi-módulo descrito da Secção 3.5.5.

Mais recentemente, um outro trabalho (Drahonovský et al., 2013) apresenta um processo para que seja igualmente possível realocar *bitstreams* parciais em várias RPs. É baseado no trabalho (Ichinomiya et al., 2012a) e elimina o problema do roteamento dos sinais que cruzavam as RPs. Em vez de módulos IPIF, adiciona na parte estática do sistema uma LUT a cada sinal de entrada/saída que constitui o interface com o componente implementado na RP. Desta forma, os sinais que constroem o interface entre uma RP e o restante sistema ficam delimitados por pares Proxy Logic-LUT. Estes pares são então mapeados em determinadas localizações através de restrições (*constraints*). Este processo, juntamente com o uso do *Direct Routing Constraints* do Xilinx FPGA Editor que é usado para extrair e forçar o roteamento através de *constraints*, garantem a uniformidade do interface entre qualquer RP e o restante sistema. No entanto, a forma ad hoc como são restringidas essas localizações sacrifica logo ao início a liberdade das ferramentas que compõem o fluxo, o que acaba por introduzir uma degradação extra dos caminhos críticos. Como principal novidade, o autor aproveita o facto das recentes FPGAs da Xilinx possuírem LUTs com duas saídas. Assim, com uma LUT6 física é possível implementar duas LUT1, o que permite reduzir um pouco o *overhead* da adição de hardware. No artigo, não sendo claro, tudo indica que pelo menos algumas partes do fluxo são realizadas manualmente, o que não é prático, e nenhuma solução é apresentada para controlo dos sinais de relógio.

Focado na prototipagem rápida, um novo fluxo chamado *Turbo Flow* (TFlow) (Love et al., 2013) foi desenvolvido no *Bradley Department of Electrical and Computer Engineering*, pertencente à *Virginia Polytechnic Institute and State University*. Este fluxo, orientado para projetar sistemas em FPGA de uma forma modular, visa reduzir o tempo de compilação do sistema a implementar. Para isso realiza uma pré-compilação dos módulos, em várias permutações existentes numa bi-

blioteca. Estes módulos podem depois ser encaixados durante o processo de implementação, o que diminui drasticamente o tempo necessário para obter o mesmo *bitstream*, quando comparado com o tempo necessário para o fluxo disponibilizado pelas ferramentas do fabricante de FPGAs. Para obter tal objetivo, recorrem ao *Tools for Open Reconfigurable Computing* (TORC) (Steiner et al., 2011), que após o normal fluxo de implementação fornecido pelo fabricante (*mapper*, *placer*, *router* e gerador de bit de configuração), extrai da *netlist Electronic Design Interchange Format* (EDIF) os principais atributos dos módulos, tais como os sinais de interface, localização e recursos utilizados, que são depois armazenados num arquivo *Extensible Markup Language* (XML). Com esta informação é depois possível abreviar etapas do normal fluxo de implementação. Uma grande desvantagem deste fluxo, é que como não há nenhuma otimização global, o roteamento dos sinais pode ser mais longo que no fluxo normal, o que pode originar mais caminhos críticos e com tempos de atraso superiores. Se é verdade que o TFlow permite realocar módulos em segundos, esta realocação é feita antes da geração do *bitstream* onde é implementado o sistema e não pelo próprio sistema. Ou seja, é uma boa solução para acelerar o desenvolvimento e prototipagem, mas não para ser usado num sistema implementado numa FPGA.

Uma importante dificuldade associada à realocação de *bitstreams* parciais em várias RPs é o roteamento dos sinais de relógio. O artigo (Flynn et al., 2009) apresenta uma solução para esse problema. Para isso apresentam os *Local Clock Domains* (LCDs), que com o uso e fixação de primitivas BUFR (Xilinx Inc., 2013g), consegue garantir um roteamento relativo dos sinais de relógio igual nas várias RPs. Sugere ainda que exista apenas um sinal de relógio global, e em cada RP gerar outros sinais de relógio necessários através de um *Digital Clock Manager* (DCM). Neste artigo nada é descrito relativamente ao resto do fluxo que garante a realocação de *bitstreams* parciais.

Vários autores pertencentes a grupos da Universidade Técnica de Dresden e da Universidade de Lübeck na Alemanha, após estudarem vários trabalhos relativos à realocação de *bitstreams* que recorrem a

mecanismos que permitem realocar o mesmo *bitstream* em várias RPs, observaram que faltava uma ferramenta para explorar sistematicamente os recursos de uma FPGA, com a finalidade de identificar todas as possibilidades de RPs para um determinado grupo específico de recursos (Backasch et al., 2014). O processo para encontrar as RPs possíveis é dividido em quatro fases. No início são definidas as necessidades de recursos pretendidas para as RPs. Estes dados são utilizados na análise da matriz de recursos da FPGA tal como na geração do padrão de componentes de hardware que compõe o dispositivo. Em função das necessidades de hardware, a análise gera um mapa de recursos adaptado à FPGA selecionada, enquanto a geração de padrões é usado para gerar posicionamentos que preencham os requisitos de hardware pedidos. Em seguida, o algoritmo proposto de *floorplanning* explora a FPGA e encontra todas as possibilidades de RPs que contêm o padrão gerado. Finalmente, as localizações resultantes podem ser exportadas para ser utilizadas no projeto. Esta ferramenta auxilia assim, durante o processo de seleção da FPGA para o projeto, respondendo às seguintes perguntas: Quantas RPs podem coexistir numa determinada FPGA? Quanto recursos pode ter cada módulo para ser possível ter N módulos na FPGA disponível? Quais são as localizações possíveis das RPs e que zonas de recursos da FPGA não são adequadas para a realocação?

3.4.2 Isolation Design Flow da Xilinx

Com o intuito de tornar circuitos críticos mais tolerantes a faltas, a Xilinx desenvolveu a metodologia *Isolation Design Flow* (IDF) (Xilinx Inc., 2016c). Este fluxo baseia-se no já existente formado pelas ferramentas que compõem o ISE (ou Vivado no caso da família 7 Series), e é parte integrante do grupo de ferramentas da Xilinx com a certificação IEC61508. A aplicação desta metodologia implica o seguimento de um conjunto de regras, entre as quais todo o sistema deverá estar organizado em partições, embora nem todas tenham de ser reconfiguráveis (Hallett, 2015). Como resultado, todos os módulos do sistema implementados em RPs ficam 100% isolados do restante sistema. Mesmo os sinais de entrada

e saída que interligam cada RP ao resto do circuito são obrigados a seguir algumas regras específicas. A Figura 20 mostra a implementação de uma aplicação de segurança crítica.

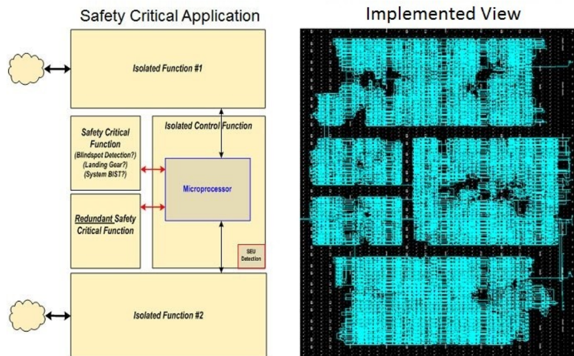


Figura 20 – Exemplo de uma implementação usando o fluxo IDF (Xilinx Inc., 2016c).

3.4.3 Conclusão

A compatibilidade do interface das várias RPs é algo que o normal fluxo das ferramentas ignoram. Por isso é necessário encontrar formas de garantir que essa compatibilidade seja implementada. Uma dessas formas seria um fluxo que respeite as recomendações usadas na metodologia IDF, e que de uma forma automática nos dê a capacidade de troca de módulos entre RPs. Alguns trabalhos já realizados vão nesse sentido, no entanto possuem ainda algumas limitações como: retiram liberdade às ferramentas *Mapping* e *Placement and Routing*; não permitem a possibilidade de desabilitar todos os sinais que constituem o interface com cada RP, o que pode gerar valores transitórios errados; necessitam de repetir fluxo Translation → Mapping → Placement and Routing pelo menos mais duas vezes; e necessitam de passos manuais intermédios.

Focando no objetivo central que é dotar do sistema a capacidade de recuperar de faltas permanentes, alguns dos trabalho analisados possuem detalhes importantes que podem ser usados (e ainda melhora-

dos), no sentido de criar uma plataforma que possa recuperar de faltas permanentes, bastando para isso mover a localização do módulo faltoso para outra RP, através da cópia do respetivo *bitstream*.

3.5 PLATAFORMAS EM FPGA COM TOLERÂNCIA A FALTAS PERMANENTES

3.5.1 Metodologias de Teste para FPGAs Integradas em Sistemas Reconfiguráveis

Na Faculdade de Engenharia da Universidade do Porto, Manuel Gericota propôs, há mais de uma década, uma nova metodologia de teste para FPGAs com capacidade de reconfiguração parcial dinâmica, com o objetivo de assegurar o seu funcionamento continuado (Gericota et al., 2001) (Gericota et al., 2002) (Gericota, 2003).

O objetivo principal é dotar um sistema implementado numa FPGA da realização cíclica de testes ao longo do seu ciclo de vida. Isto porque, por um lado, a própria tecnologia utilizada não está isenta de problemas de fabrico e alguns deles poderão apenas gerar faltas após um determinado período em funcionamento (Shnidman et al., 1998). Por outro, em ambientes em que a FPGA esteja sujeita a um nível considerável de radiação, esta pode igualmente provocar faltas (permanentes e/ou transitórias) nas funções implementadas na FPGA (Huang and McCluskey, 2001) (Alexandrescu et al., 2004) (White, 2012).

3.5.1.1 Descrição da Metodologia Proposta

O autor apresenta um teste que abrange tanto faltas permanentes como transitórias, e a possibilidade de estas se manifestarem e serem corrigidas ao longo de toda a vida útil da FPGA. O teste é realizado de uma forma individual à maioria dos recursos que compõem o dispositivo, não exige nenhuma condicionante durante o processo de implementação do circuito (ou qualquer acréscimo de hardware) e é executado de um modo transparente (o circuito implementado mantém sempre o seu normal funcionamento). É por isso um teste à real estru-

tura dos recursos da FPGA, que é realizado ao mesmo tempo que as funções implementadas no dispositivo se mantêm em funcionamento.

Como para testar os recursos estes não podem estar a ser utilizados, são então libertados antes de serem testados. O mecanismo proposto passa portanto por selecionar uma parcela de recursos, libertá-la de qualquer utilização, testá-la e caso não sejam detetadas faltas, alocar nessa parcela de recursos as funções existentes noutra parcela, libertando essa outra parcela para ser testada. Ou seja, a implementação deste mecanismo exige que, sucessivamente, permita a relocação das funções em execução, replicando a sua funcionalidade em áreas previamente testadas e libertando continuadamente para o teste os recursos que se encontrem ocupados.

A sequência de relocação, como ilustra a Figura 21, implica uma rotação da área sob teste, abrangendo a totalidade da área de configuração da FPGA. A área sob teste é primeiramente testada e posteriormente reconfigurada com a funcionalidade implementada na próxima área a testar. Seguindo esta sequência, a relocação faz-se sempre para uma área previamente testada, pelo que, na eventualidade de durante o teste ser detetada uma falta num determinado recurso da FPGA, a funcionalidade previamente implementada nesse recurso já terá sido relocalizada num recurso isento de faltas.

A metodologia proposta segue o ciclo da Figura 21 e divide-se genericamente nas três partes seguintes:

1. a relocação da funcionalidade, por replicação dos elementos ativos;

2. a rotação da área consignada ao teste, varrendo a totalidade da FPGA num intervalo de tempo determinado;

3. o teste estrutural dessa área.

Estes três passos desenrolam-se sucessiva e ininterruptamente até que a totalidade da FPGA esteja testada.

Resumindo, o teste realiza-se sequencialmente, varrendo a totalidade da FPGA, sem paragens e com um tempo de latência determinado.

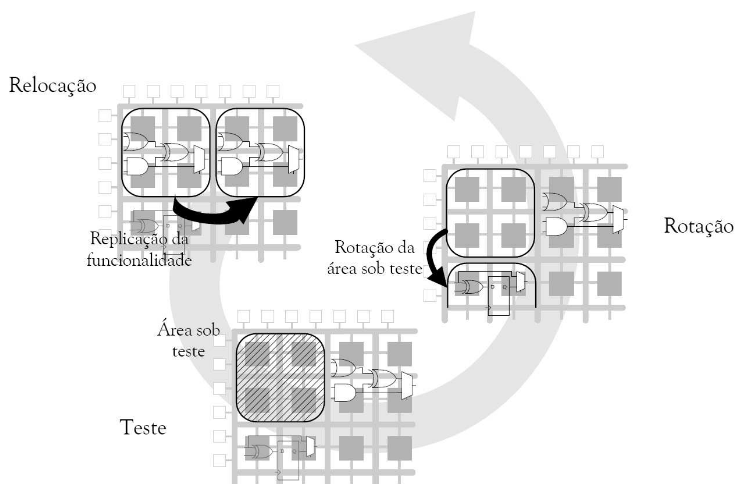


Figura 21 – Metodologia de teste proposta (Gericota, 2003).

Findo o varrimento, o processo repete-se, nos mesmos moldes, num ciclo infinito, para que qualquer falta que ocorra antes, durante ou após a configuração de novas funções, ao longo de toda a vida útil do componente, seja detetada e ultrapassada.

3.5.1.2 Conclusão, Vantagens e Desvantagens

Esta proposta (aplicável às FPGAs disponíveis comercialmente na época em que foi desenvolvida) possui grandes vantagens tais como: não requerer qualquer modificação da arquitetura do sistema implementado, pois todo o processo de teste, deteção e recuperação é controlado do exterior através do *Boundary Scan* (BS); o teste é realizado à totalidade dos CLBs da FPGA, dentro das limitações devidas à impossibilidade de replicação de LUTs utilizadas como memória distribuída; e não afeta o funcionamento de qualquer uma das funções implementadas. Adicionalmente, é possibilitado o teste das interligações, sujeito a restrições, no sentido em que, dependendo da taxa de ocupação da FPGA, pode não ser exequível o teste da totalidade destes recursos.

Como as operações realizadas nesta metodologia são feitas

através do BS, aproveitando a ferramenta *JBits* (Xilinx Inc., 2014a) e a detalhada informação de baixo nível disponibilizada pela fabricante Xilinx, esta metodologia tornou-se refém do próprio fabricante. Como esta deixou de disponibilizar o mesmo tipo de detalhe de informação nas novas famílias de FPGAs e a ferramenta *JBits* deixou de ser suportada depois da família Virtex-2, a migração e a evolução desta metodologia para as famílias de FPGAs mais recentes acabaram por ser inviabilizadas.

Relativamente à política de recuperação de faltas permanentes, esta metodologia apenas exclui o CLB onde ocorreu a falta, o que torna esta estratégia bastante eficiente na gestão de recursos. Já em relação ao teste, como é feito a cada bloco de recursos individualmente, faltas de atraso (*delay faults*) causadas pelo envelhecimento, seja ele devido a radiação ou algum tipo de BTI, não serão detetadas e por isso ficarão por ser tratadas.

3.5.2 Utilização de *Bit files* Parciais para Tolerância a Faltas

Num trabalho desenvolvido no *Air Force Institute of Technology* em Hobson Way foi proposto o uso da reprogramação parcial, *Partial Reconfiguration* (PR), para aumentar a tolerância a faltas permanentes em sistemas digitais implementados em FPGAs (Montminy et al., 2007). A ideia base é dividir o sistema em vários módulos, implementá-los em zonas separadas, e quando uma falta permanente ocorrer num dos módulos, realocar esse módulo para uma zona livre através da reprogramação parcial da FPGA.

3.5.2.1 Parcelamento da FPGA e Funcionamento dos Módulos Realocáveis

Nesta proposta há um aproveitamento quase que direto da estrutura baseada em colunas (*column-based*) existente nas primeiras famílias de FPGAs. A família usada, uma Virtex-2, é uma delas. Nestes dispositivos, os recursos estão organizados em colunas completas, pelo que quando se usa a reconfiguração dinâmica, o volume mínimo de recursos que se consegue isolar e configurar sem interferir com o restante

circuito configurado na FPGA, é uma dessas colunas. Significa por isso que para dividir os recursos de uma FPGA em parcelas, estas terão de ser formadas por um grupo de colunas completas.

A Figura 22 mostra um exemplo em que uma FPGA é dividida em cinco parcelas, com um tamanho igual e sendo constituídas por um conjunto de colunas de recursos adjacentes.

(a) Disposição Inicial.

(b) Disposição após Realocações.

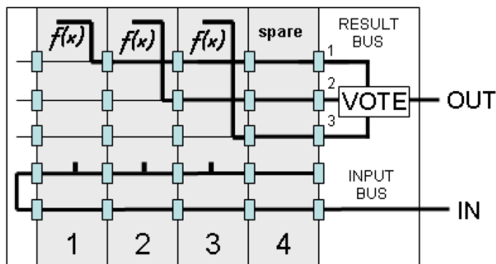
Figura 22 – Representação da disposição inicial dos Módulos (a) e após realocação dos Módulos *Function C* e *Function D* nas colunas quatro e cinco, respetivamente (b) (Montminy et al., 2007).

Na mesma figura, é exemplificado conceptualmente o funcionamento do trabalho proposto. Na Figura 22a, o sistema composto por quatro módulos (*Function A*, *Function B*, *Function C* e *Function D*) é distribuído pelas primeiras quatro parcelas, sobrando uma quinta (*Spare*) para a eventualidade de ocorrer uma falta permanente em algum dos quatro módulos. Caso, por exemplo, ocorra uma falta permanente na parcela 3 que tem nela implementado o módulo *Function C*, este módulo juntamente com o módulo *Function D* serão movidos para a direita, tal como mostra a Figura 22b.

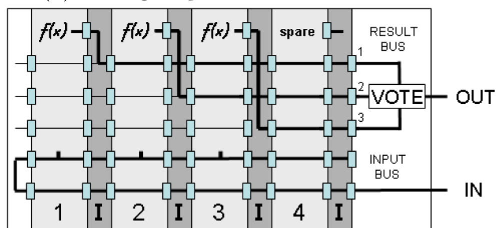
Usando este conceito e assumindo que as faltas ocorrem sempre em parcelas diferentes, um sistema implementado pode sofrer um número de faltas permanentes igual ao número de parcelas livres (*Spare*).

A demonstração do trabalho é realizado com um circuito com redundância no espaço *Triple Modular Redundancy* (TMR), e tal como representado na Figura 23, apresenta três possíveis formas de estruturar a interligação entre módulos, de maneira a que todo o sistema fique

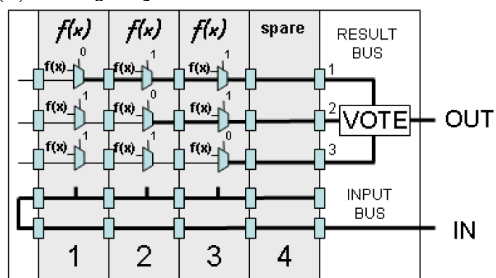
tolerante a faltas permanentes.



(a) Configuração com Conexão Direta.



(b) Configuração com Módulo de Interconexão.



(c) Configuração com Roteamento usando LUTs.

Figura 23 – Os três modos de interligação entre Módulos (Montminy et al., 2007).

A primeira delas, apresentada na Figura 23a, é a configuração com a conexão direta entre módulos. Nesta configuração, embora existam três módulos que desempenham a mesma função, cada um deles tem uma ligação diferente a um barramento de dados, para além de ter igualmente implementada a passagem de barramentos usados por outros módulos. Tal facto, obriga a que para cada uma das quatro parcelas seja

necessário gerar um *bitstream* parcial para cada um dos três módulos, ou seja, 12 *bitstreams*, mais quatro para o caso da parcela livre (*Spare*).

Para reduzir o número de *bitstreams* necessários, o autor sugere a alternativa da configuração com módulos de interconexão, tal como ilustrado na Figura 23b. A diferença em relação à opção anterior, é que nesta parte de interligação é separada da funcionalidade do módulo. Desta forma, basta um *bitstream* de cada módulo para a sua posição inicial, mais um de cada para a parcela livre. Como desvantagem está o facto de necessitar também de *bitstreams* para fazer as várias hipóteses de interligação entre módulos. Mas como o seu tamanho é mais reduzido, isto permite uma redução considerável da memória exigida para a biblioteca dos *bitstreams* parciais (existente no exterior).

A terceira alternativa, esquematizada na Figura 23c, visa alcançar uma redução da memória para armazenar os *bitstreams* parciais ainda superior. Nesse sentido, o conteúdo das LUTs responsáveis pela interligação entre módulos é diretamente alterado para configurar a interligação dos módulos ao barramento desejado. Esta alteração é realizada através de reduzidos *bitstreams* de reconfiguração parcial baseada em diferenças, tal como referido na Secção 2.3.3. Esta alternativa tem como principal desvantagem o aumento do caminho crítico do sistema.

3.5.2.2 Conclusão, Vantagens e Desvantagens

Esta abordagem mostra o potencial do uso da reconfiguração dinâmica para dotar um circuito de tolerância a faltas permanentes. Associada a um sistema com TMR, esta proposta pode ser um bom complemento à redundância existente no circuito, pois por um lado garante que o sistema não tem erros na saída quando ocorre uma falta permanente, e o outro é possível recuperar o correto funcionamento de toda o TMR.

Como limitações deste trabalho, excluindo a alternativa da Figura 23c, exige um volume considerável para armazenar a biblioteca de *bitstreams* parciais. A exclusão de toda a parcela por causa de uma falta permanente num único recurso da FPGA nessa parcela é

outro desperdício que esta abordagem comporta. Para além disso, a proposta aproveita a própria estrutura das FPGAs *column-based*, no entanto, a partir da família Virtex-4 as FPGAs da Xilinx passaram a ter uma organização de recursos *tile-based*. Tal facto trouxe uma maior dificuldade em implementar esta proposta, sendo talvez por isso que os autores não voltaram a publicar nada que mostrasse a evolução desta proposta para as famílias de FPGAs mais recentes.

Relativamente ao processo de realocação, este só é desencadeado quando uma falta permanente é detetada. Não existe por isso qualquer política de gestão das parcelas em uso de modo a gerir a utilização dos recursos da FPGA. Tal ação, que pela descrição técnica do trabalho parece possível, poderia atenuar o envelhecimento dos recursos da FPGA devido ao BTI ao longo do ciclo de vida do sistema implementado no dispositivo.

3.5.3 Realocação de Aceleradores em Hardware via ICAP para Execução e Sincronização de Multitarefas em Hardware

O *System Level Integration Group* da Universidade de Edimburgo, em parceria com *Embedded System-on-Chip Group* da *IKERLAN-IK4 Research Alliance* no País Basco em Espanha desenvolveram uma técnica que simplifica a realocação de tarefas em hardware numa FPGA (Iturbe et al., 2011).

3.5.3.1 Descrição do Funcionamento do Sistema com Multitarefa em Hardware

A aplicação principal deste trabalho é permitir a gestão de multitarefa em hardware. Para isso apresentam o *Communication Interface* (CIF) que é incluído em cada tarefa em hardware existente no sistema, e que permite a comunicação e sincronização entre tarefas. Como mostra a Figura 24, o CIF inclui um *buffer* para a entrada de dados (*Input data BUFFER*) e outro para a saída de dados (*Output results BUFFER*), que são acedidos pelo restante sistema através da

camada de configuração (*Configuration Layer*). Ou seja, tal como indica a Figura 25, o microprocessador principal (*MAIN uP*) implementado na camada funcional (*Functional Layer*) usa o *Internal Configuration Access Port* (ICAP) existente na FPGA para escrever os dados a serem processados nos *buffers* de entrada e para o ler dos *buffers* onde são lançados os resultados por cada tarefa em hardware.

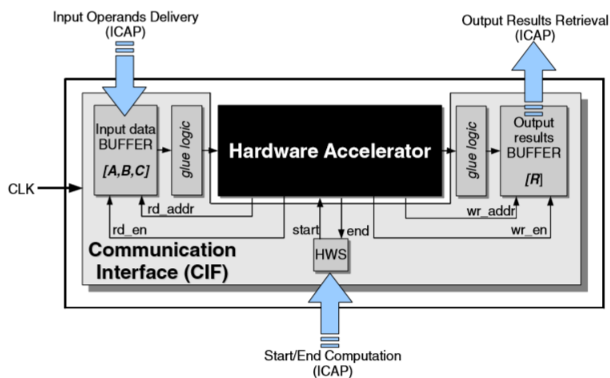


Figura 24 – Aceleração de uma Tarefa em Hardware (Iturbe et al., 2011).

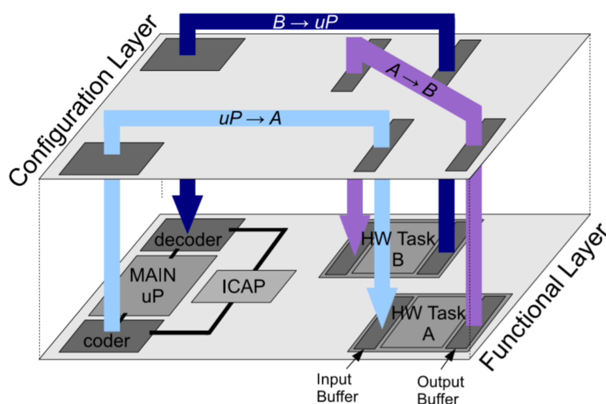


Figura 25 – Comunicação entre Tarefas via ICAP (Iturbe et al., 2011).

O CIF dá assim origem a tarefas em hardware que não necessitam de roteamento de sinais para as entradas e saídas. Desta forma,

entre o módulo acelerador alocado numa partição e o restante sistema, apenas é necessário existir um sinal de relógio, o que simplifica o fluxo para realocação de módulos aceleradores entre *Reconfigurable Partitions* (RPs).

Mais recentemente, o mesmo grupo focou-se no único sinal que cruza verdadeiramente a fronteira entre as partições onde são alocadas as tarefas em hardware e o restante sistema (Iturbe et al., 2012). Nesse sentido, cada RP é delimitada pela região de relógio e usa um relógio local fornecido pela primitiva `BUFGCTRL` (Xilinx Inc., 2013g). Explorando esta capacidade das recentes famílias de FPGA da Xilinx, e dado que a comunicação entre o *MAIN uP* e as tarefas é feita através do ICAP, cada tarefa pode ter uma frequência de relógio local de acordo com as características do seu hardware acelerador, o que permite aumentar a performance de todo o sistema, pois cada tarefa em hardware irá funcionar à sua frequência máxima.

De forma a melhorar a confiabilidade relativamente à distribuição de relógio, este mesmo trabalho (Iturbe et al., 2012) liga a cada RP dois sinais para o mesmo relógio. A igualdade destes dois sinais de relógio é verificada, caso os sinais não sejam iguais, o mesmo módulo que os verifica deteta qual o correto e seleciona para alimentar a correspondente tarefa em hardware. Esta redundância habilita por isso a tolerância a faltas a nível da árvore de relógio.

3.5.3.2 Conclusão, Vantagens e Desvantagens

O facto de usar o ICAP para toda a comunicação permite a este trabalho evitar processos de maior complexidade, como os descritos na Secção 3.4.1, para seja possível uma flexível realocação das tarefas em hardware. Mas se esta é a sua principal vantagem, é igualmente a sua principal desvantagem, pois todos os processos de reconfiguração, comunicação e sincronização passam pelo ICAP. Em aplicações que exijam um considerável fluxo de dados entre tarefas, esta solução terá a sua performance drasticamente reduzida. A duplicação do sinal de relógio permite adicionar tolerância a faltas no único sinal que interliga cada

RP ao restante sistema. Relativamente a tolerância a faltas permanentes nos restantes recursos da FPGA existentes em cada partição, o autor comenta como trabalho futuro tirar partido desta arquitetura para esse fim. Tudo indica que o passo será excluir áreas de recursos onde tenha sido detetada alguma falta permanente. Não é mencionada qualquer forma de minimizar o envelhecimento (*aging*) da FPGA, quer seja devido a radiação, quer seja a algum tipo de BTI.

3.5.4 Plataformas em FPGAs com Sistemas Autónomos Tolerantes a Faltas

No Departamento de Eletrónica e Informação pertencente ao Politécnico de Milão foi projetado um fluxo automático para a implementação de sistemas autónomos tolerantes a faltas em plataformas com FPGA baseadas em SRAM. Esta abordagem consegue recuperar da ocorrência tanto de faltas transitórias como permanentes (Bolchini et al., 2012).

3.5.4.1 Descrição do Funcionamento do Sistema Autónomo Tolerante

A generalidade das abordagens existentes na literatura apenas se foca nas estratégias de recuperação e nos aspetos relacionados com a arquitetura, sem considerar globalmente toda a conceção do sistema confiável. Por essa razão, o fluxo proposto explora durante a etapa de implementação as diferentes alternativas de nível de tolerância a faltas e particionamento.

O desenvolvimento deste sistema autónomo capaz de mitigar faltas permanentes e transitórias foi feito na continuidade de trabalhos anteriores, tanto seguindo estratégias relacionadas com cenários onde somente ocorrem faltas transitórias (Bolchini et al., 2011b) (Bolchini et al., 2011a), quanto explorando as abordagens apresentadas em (Bolchini et al., 2011a) (Bolchini et al., 2011c).

Focando-se na confiabilidade do sistema, a ferramenta criada para gerar o circuito desejado gera automaticamente um conjunto de

bitstreams que seguem estas três etapas:

- identificação do número de faltas permanentes que podem ser gerenciadas, que é relacionado com o número de partições (áreas) do sistema e os recursos disponíveis na arquitetura (por exemplo, o tamanho da memória do *bitstream*);
- identificação cada *bitstream* de recuperação relativamente a cada falha permanente possível de ocorrer;
- procura de uma solução otimizada, alcançada por intermédio de uma exploração do espaço de projeto, que inclui a descrição *Very High Speed Integrated Circuit Hardware Description Language* (VHDL), a FPGA de destino, etc.

Em resumo, após analisar a descrição do sistema em VHDL é decidido qual o número máximo de faltas permanentes que o sistema deverá ser capaz de auto recuperar (`max_faults`), e qual o número de diferentes áreas no trabalho em que o sistema se deverá organizar (`max_areas`).

A arquitetura do sistema gerado pelo fluxo desenvolvido, tal como mostra a Figura 26, necessita de duas FPGAs. Uma SRAM FPGA, onde é implementado o circuito (*Payload*) ao qual se deseja adicionar a capacidade de autonomamente recuperar de faltas (permanentes e transitórias), e outra (*Rad-Hard* FPGA), que controla o processo de reconfiguração e de gestão da biblioteca de *bitstreams*, armazenados numa memória externa.

Na descrição deste trabalho, o autor menciona que recorre ao TMR em determinadas aplicações, para assim facilmente conseguir detetar a ocorrência de faltas e ao mesmo tempo evitar que os erros produzidos por estas não originem falhas no sistema.

No exemplo usado para demonstração, o número máximo de faltas permanentes toleradas (recuperáveis) foi definido como dois (`max_faults=2`). O circuito (*Payload*) é composto por 15 componentes e, sem aplicar um limitação `max_areas`, o número de *bitstreams* diferentes que corresponderiam a todas as sequências possíveis de duas faltas

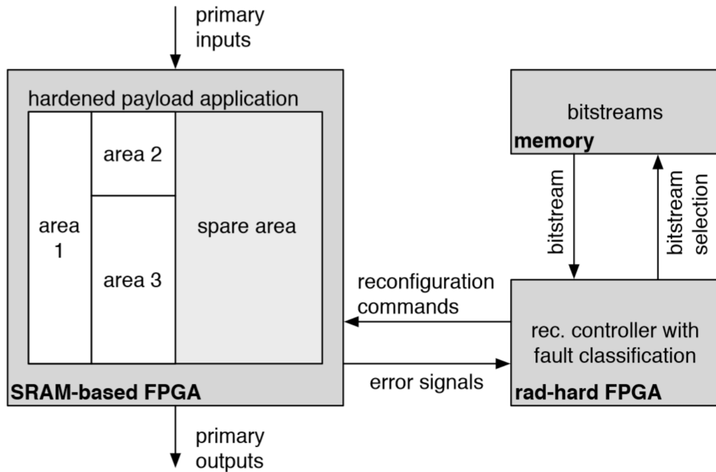


Figura 26 – Arquitetura do Sistema Autónomo Tolerante a Faltas (Bolchini et al., 2012).

permanentes, seria de 241, o que tendo em conta que o tamanho do *bitstream* para o dispositivo selecionado é 2.9 MB, implicaria uma memória de pelo menos 700 MB. Para reduzir a quantidade de memória exigida, foi fixado o parâmetro `max_areas=6`, o que diminuiu a necessidade de memória para 128 MB.

3.5.4.2 Conclusão, Vantagens e Desvantagens

A análise da descrição do circuito em VHDL para assim ajustar a quantidade de faltas permanentes toleráveis em função dos recursos disponíveis é algo importante para o projetista. Como não recorre a nenhuma estratégia de realocação de módulos como os apresentados na Secção 3.4.1, a biblioteca de *bitstreams* exige um volume de memória que atinge facilmente centenas de MB. Para além da exigência deste volume de memória, se faltas permanentes ocorrerem neste hardware, toda a plataforma fica em risco. Igualmente, o facto de necessitar de uma segunda FPGA para gerir o circuito e a proteger contra faltas, mesmo sendo *Rad-Hard*, não está imune a sofrer faltas permanentes. Ou seja, para proteger um circuito contra a ocorrência de faltas foi adicionado

mais hardware, sendo que o novo hardware se encontra desprotegido.

É ainda referido que quando uma falta permanente é detetada, toda a área é assinalada como não utilizável, significando por isso, que basta um recurso da FPGA pertencente a uma determinada área ser destruído, para que todos os restantes recursos existentes nessa área sejam considerados não utilizáveis. Relativamente ao envelhecimento (*aging*) causado por radiação ou algum tipo de BTI, nada é referido, nem em relação às faltas de atraso (*delay faults*), também elas permanentes, e que o envelhecimento pode igualmente originar.

3.5.5 Sistema com Realocação de *Bitstreams* Otimizada para Aplicações com Multi-Módulos Carregados por Reconfiguração Parcial

O *LE2I Laboratory* da Universidade de Burgundy apresentou uma proposta para que um sistema implementado numa FPGA, possa de um modo fácil e eficiente realocar módulos em partições reconfiguráveis (RPs), tal como se de tarefas em software se tratassem (Ochoa-Ruiz et al., 2013). Para isso recorreu à metodologia OORBIT (*Offline/Online Relocation of Bitstreams*) (Touiza et al., 2012), já referida da Secção 3.4.1 e que foi também ela desenvolvida por este mesmo grupo.

3.5.5.1 Descrição do Funcionamento do Sistema com Capacidade de Realocação de *Bitstreams*

O principal foco desta proposta é mostrar uma forma de acelerar o processo de realocação de *bitstreams*, de modo a que a troca de tarefas em hardware possa aproximar-se a uma troca de tarefas em software. Isto no que diz respeito ao peso do tempo necessário para realizar a troca em relação ao tempo de execução total da tarefa em hardware, pois se o tempo de realocação for relevante, poderá atenuar o ganho obtido pela aceleração em hardware. Isto é conseguido através da utilização do módulo *Bitstream Relocator*, como mostra o diagrama da Figura 27.

O *Bitstream Relocator* é responsável por analisar o *bitstream* durante o processo de reconfiguração e modificar as informações relacio-

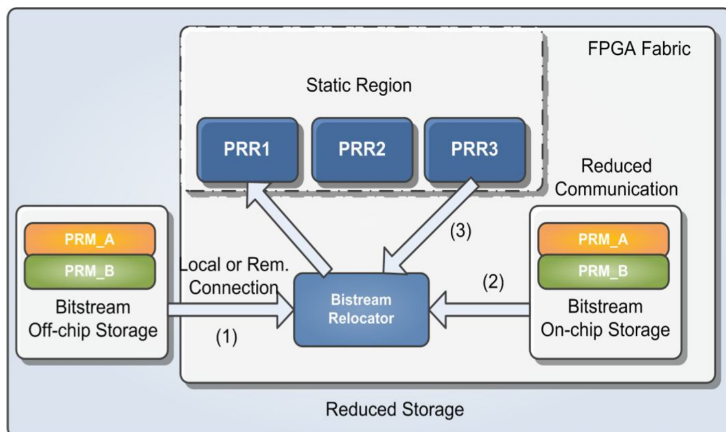


Figura 27 – Sistema com Capacidade de Realocação de *Bitstreams* (Ochoa-Ruiz et al., 2013).

nadas com a colocação do *bitstream*, re-mapeando as suas funcionalidades para uma partição reconfigurável diferente (na Figura 27 designada por PRR). O carregamento dos *bitstreams* parciais correspondentes aos módulos a realocar (na Figura 27 designados por PRMs) pode ser realizado de três formas: (1) de uma memória externa (Off-chip Storage) que pode ser local ou remota; (2) da memória interna (On-chip Storage), que embora implique reservar BRAMs para esta finalidade, reduz os tempos de carregamento dos *bitstreams*; e (3) a troca de *bitstreams* entre diferentes PRRs.

A aceleração no processo de realocação é obtida pelo uso da metodologia OORBIT (Touiza et al., 2012). Numa primeira fase, o OORBIT analisa os *bitstreams* parciais obtidos através do fluxo das ferramentas de reconfiguração parcial, onde obtém os endereços de configuração *Frame Address Register* (FAR) das várias PRRs, de modo a definir todas a possibilidades de realocação no sistema. Numa segunda fase, são calculados os novos endereços FAR e os novos valores CRC (ao alterar o endereço FAR de um *bitstream* parcial, é necessário recalcular o seu novo CRC final), para cada possibilidade de realocação. Esta informação é depois adicionada no final do *bitstream* original corres-

pondente. Desta forma, o processo de realocação apenas precisa alterar os campos do endereço FAR e CRC no *bitstream* do módulo que pretende realocar em função da PRR destino. Como estes campos já foram previamente calculados, o *Bitstream Relocator* não necessita despender processamento no momento em que pretende realocar qualquer módulo.

3.5.5.2 Conclusão, Vantagens e Desvantagens

Este trabalho tira proveito da reconfiguração parcial dinâmica com a finalidade de desenvolver um mecanismo eficiente de troca de contexto num sistema implementado numa FPGA. Para isso apresenta uma metodologia que permite a realocação de um *bitstream* em várias RPs e minimiza o tempo de sobrecarga para realizar a realocação dos *bitstreams*, combatendo assim os principais fatores que podem evitar a adoção da reconfiguração parcial dinâmica em sistemas com multitarefas em hardware, devido à sobrecarga de tempo de reconfiguração e de necessidade de memória para armazenar todas as possibilidades de *bitstreams*.

Embora o OORBIT permita a liberdade de realocação de um *bitstream* em qualquer RP compatível, esta vantagem não é aproveitada para recuperação de faltas permanentes que possam ocorrer devido a radiação, nem existe qualquer política em não sobrecarregar os recursos de uma RP em relação a outras, de forma a minimizar o envelhecimento (*aging*) devido a algum tipo de BTI.

3.5.6 Projeto de Sistemas Críticos em FPGAs com Tolerância a Faltas para Ambientes Hostis

Um outro trabalho, realizado no *Fault-Tolerant Systems Group*, pertencente ao *Instituto de Aplicaciones de las TIC Avanzadas* da *Universitat Politècnica de València* (UPV), apresenta uma abordagem que permite a utilização segura de FPGAs comerciais em ambientes hostis (Espinosa et al., 2012). A proposta combina a redundância espacial e temporal com reconfiguração dinâmica parcial para aumentar a resili-

ência a faltas (transitórias e permanentes) de projetos implementados em FPGAs.

3.5.6.1 Descrição da Arquitetura do Mecanismo de Ocultação e Correção de Faltas

O objetivo do mecanismo desenvolvido é tolerar a ocorrência de faltas, mesmo durante o processo de reconfiguração da FPGA, enquanto minimiza o impacto do processo de recuperação sobre a disponibilidade do sistema. A arquitetura designada por TMR-MDR (de *TMR-Module Discard and Repair*), apresentada na Figura 28, é constituída por dois componentes de hardware principais: uma FPGA comercial com capacidades de reconfiguração parcial dinâmica, e uma memória *FLASH* com elevado nível de tolerância a faltas para armazenar os diferentes *bitstreams* para programar a FPGA.

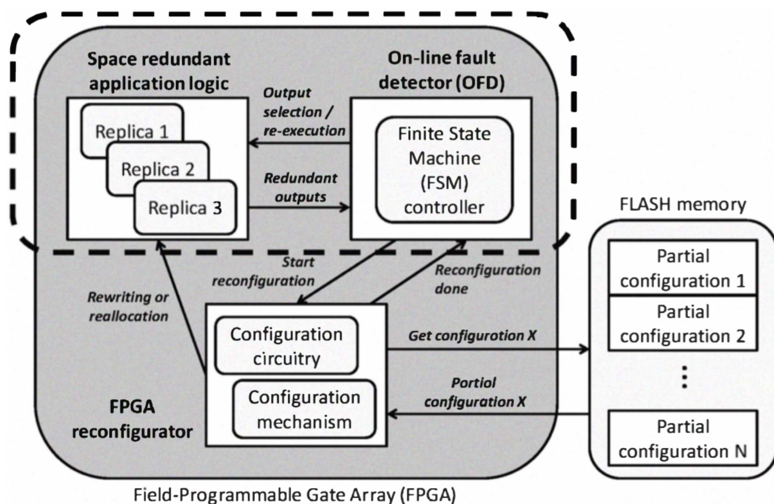


Figura 28 – Diagrama Global da Arquitetura TMR-MDR (Espinosa et al., 2012).

A inclusão do TMR visa mascarar a ocorrência de faltas no sistema, tanto a nível dos recursos reconfiguráveis como da respetiva memória de configuração. O OFD, o detetor de faltas on-line (do inglês

On-line Fault Detector), coleta informações fornecidas pelas aplicações implementadas e gere sinais de correção. Seu *modus operandi* é o seguinte: a primeira falta detetada é considerada transitória e o resultado correto é selecionado em tempo real. A deteção de uma falta repetida no mesmo local é reconhecida como um *soft error* (um *bit flip* na memória de configuração), e vai provocar uma notificação por parte do OFD para o circuito de reconfiguração (do inglês *Configuration circuit*), para que este restaure a respetiva RP (identificada na Figura 28 como "Replica"). Se a falta persistir, significará que se trata de um *hard error* e a aplicação será deslocalizada para uma outra partição reconfigurável isenta de faltas, através do carregamento do *bitstream* correspondente. No caso em que ocorre uma segunda falta numa outra "Replica" durante o tratamento da primeira, o OFD suspende o sistema até que a situação que impede a correta obtenção de resultados seja ultrapassada.

Para alcançar um tempo de reconfiguração rápido é criada uma biblioteca de *bitstreams* parciais (identificados na Figura 28 como *Partial Configuration*). O número de *bitstreams* necessários para lidar com um conjunto de módulos operacionais m e k RPs é $N = (m + k)(m + 1)$, acrescido de um *bitstream* adicional para a parte estática do projeto. Embora N seja um valor que implique um considerável volume de armazenamento na memória *FLASH*, é a forma de obter a maior flexibilidade possível e daí alcançar o melhor nível de ocultar faltas permanentes, que apenas é possível movendo a aplicação para uma nova partição arbitrária que não esteja livre e isenta de faltas permanentes.

3.5.6.2 Conclusão, Vantagens e Desvantagens

Esta proposta adiciona a tolerância a faltas a um sistema implementado numa FPGA comercial baseada em SRAM. Tal facto aumenta o interesse de expandir o uso destes dispositivos em sistemas críticos que operam em ambientes hostis, onde as únicas alternativas atuais, FPGA anti-fusível ou anti-radiação, são soluções caras, ou difíceis de obter (no caso em que os governos dos países colocam barreiras na exportação ou importação).

Como o autor recorre ao normal fluxo de *Partial Reconfiguration* (PR) e dadas as características da arquitetura, o número total de *bitstreams* parciais que compõe a biblioteca de opções para deslocar módulos para outras partições é considerável. Para além da redundância no espaço TMR que implica triplicar a necessidade de recursos da FPGA para a mesma aplicação, a necessidade de ter RPs livres para permitir futuras deslocações incrementa ainda mais a necessidade de recursos da FPGA. Isto deve-se ao facto de uma falta permanente detetada numa RP implicar a exclusão de toda a partição.

Embora esta arquitetura demonstre alguma capacidade técnica para permitir a implementação de uma gestão de recursos com vista à rotação no uso dos mesmos, neste trabalho tal objetivo não é abordado. Tal poderia ajudar a prevenir o envelhecimento dos recursos da FPGA devido ao BTI.

3.5.7 Matriz Analógica e Digital Programável para Sistema Tolerante a Faltas

O *Intelligent Systems Group*, pertencente ao Departamento de Eletrónica da Universidade de Iorque no Reino Unido, tem vindo a desenvolver e propor uma nova arquitetura reconfigurável para novas FPGAs a que designou por *Programmable Analogue and Digital Array* (PAnDA) (Trefzer et al., 2011) (Walker et al., 2013) (Campos et al., 2013) (Lawson et al., 2014). Ou seja, é proposta uma nova arquitetura para fabrico de uma nova FPGA e não uma arquitetura para usar nas FPGAs existentes.

3.5.7.1 Descrição da Arquitetura da Matriz Analógica e Digital Programável

O PAnDA, uma matriz programável digitalmente, é uma nova arquitetura reconfigurável proposta para a fabricação de FPGAs, com opções de configuração abaixo da atual camada digital existente. Esta arquitetura está assente no *Configurable Transistor* (CT) que permite o ajuste da largura efetiva do transístor (nMOS ou pMOS) após a fabri-

cação do dispositivo (Walker et al., 2013). Isto é alcançado através da implementação de um conjunto de transístores em paralelo, que podem ser ativados ou desativados, conforme mostra a Figura 29. Quando um número de transístores em paralelo estão habilitados, a largura efetiva do CT é a soma das larguras dos transístores ativos.

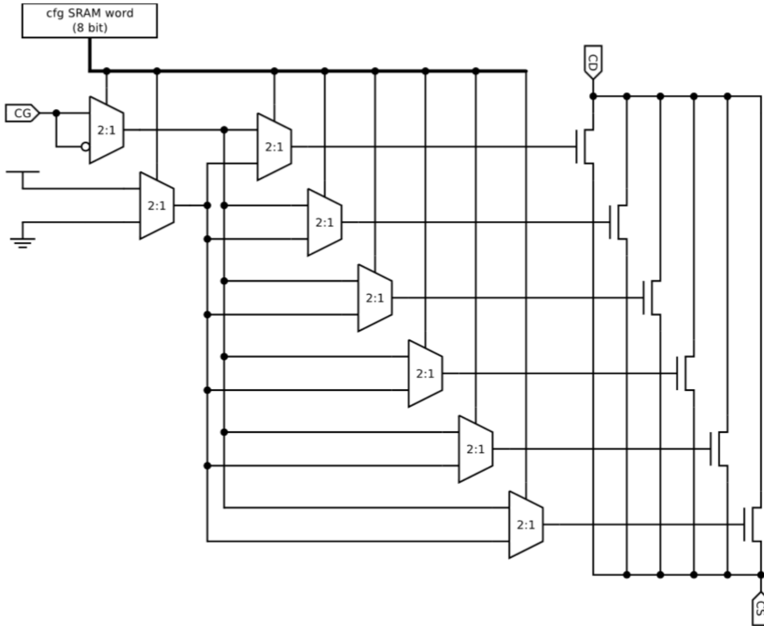


Figura 29 – Estrutura do CT na Arquitetura PAnDA-Zwei (Lawson et al., 2014).

Quatro CTs ligados em série dão origem a um *CT Branch*, e com quatro *CT Branch* é construído um *Mini Configurable Analogue Block* (MiniCAB) que pode ser configurado para executar várias funções lógicas de quatro entradas. O MiniCAB é por isso o equivalente a uma atual LUT. Por sua vez, um *Slice* é composto por quatro MiniCABs. A este nível, tal como esquematizado na Figura 30, as saídas dos quatro MiniCABs podem ser ligados entre si em várias combinações, utilizando de um a quatro MiniCABs para uma única função lógica.

Os MiniCABs também podem ser utilizados como quatro funções

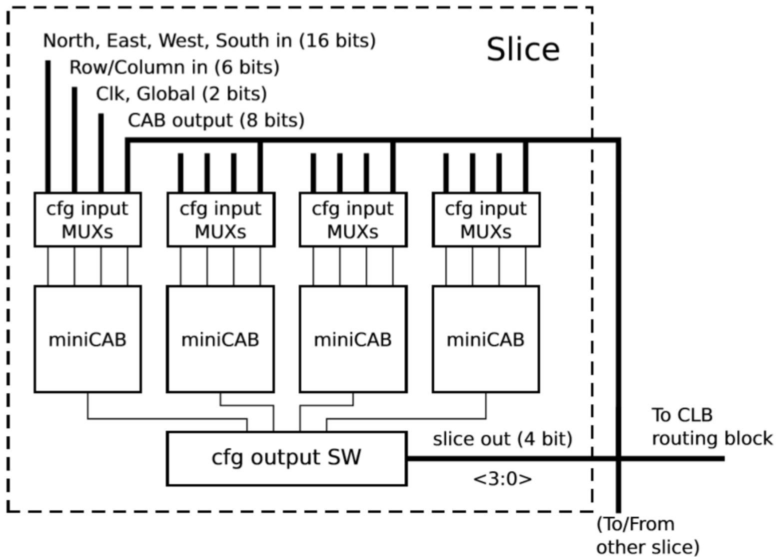


Figura 30 – Estrutura do *Slice* na Arquitetura PAnDA-Zwei (Lawson et al., 2014).

separadas, ou outras combinações, tais como funções que usam dois MiniCABs, etc. Isto é conseguido usando portas de transmissão existentes no **"cfg output SW"**. Tal como nas atuais FPGAs da Xilinx, dois *Slices* formam um CLB.

Resumindo, os autores baseiam-se na estrutura das FPGAs existentes atualmente (LUT, *Slice* e CLB) e criam uma estrutura semelhante que é suportada por *Configurable Transistor*s (CTs). A particularidade do CT cria um novo baixo nível de configuração que permite um sistema implementado numa destas FPGAs recuperar de faltas permanentes, bastando para isso, de uma forma rápida e localizada, desativar os transístores que sofreram as faltas permanentes.

3.5.7.2 Conclusão, Vantagens e Desvantagens

Embora esta investigação não tenha como objetivo uma arquitetura para usar numa FPGA atual, de modo a implementar sistemas com redundância a faltas permanentes, a sugestão apresentada permite

desenhar um novo tipo de FPGA que permita a implementação de um sistema de recuperação de faltas permanentes de baixo nível.

Para além de um sistema tolerante a faltas ser muito bem suportado pela arquitetura PAnDA, devido à hierarquia dos seus componentes e das muitas implementações possíveis para uma mesma função, pode imaginar-se que estratégias semelhantes às que são utilizadas nas existentes FPGAs baseadas em LUTs, poderão ser igualmente adotadas para esta nova arquitetura. Como principal desvantagem em relação às atuais FPGAs, está o forte aumento de hardware, tanto ao nível do silício, como na necessidade de mais memória SRAM para configurar o novo nível onde é preciso configurar os novos CTs.

3.5.8 Mecanismo Autónomo e Distribuído de Recuperação de Faltas (Transitórias e Permanentes) para Aplicações Espaciais

Na última década, investigadores do Departamento de Engenharia Elétrica e Computadores da Universidade de Ryerson em Toronto, em parceria com o grupo *Embedded Systems Robotics and Automation* da *MDA Corporation* em Brampton, têm trabalhado numa plataforma que, para além de acelerar o processamento de *stream*, contorna a ocorrência de faltas (transitórias e permanentes), autonomamente e durante a normal execução (Dumitriu et al., 2015b).

O desenvolvimento desta plataforma orientada para aplicações espaciais já vem desde 2004, tendo a sua evolução sido incremental. Nesta secção irá ser apresentado o resumo dos vários trabalhos que mostram esta evolução até ao presente (Kirischian et al., 2004) (Kirischian et al., 2006) (Kirischian et al., 2009) (Kirischian et al., 2010) (Dumitriu and Kirischian, 2010) (Dumitriu et al., 2012) (Dumitriu and Kirischian, 2013) (Dumitriu et al., 2014) (Dumitriu et al., 2015a) (Dumitriu et al., 2015b).

3.5.8.1 Arquitetura Reconfigurável para Processamento Paralelo de *Stream*

O primeiro trabalho analisado foi uma micro-arquitetura reconfigurável para processamento de *stream* paralelo, com capacidade de auto-construção e auto-recuperação de faltas transitórias e permanentes (Kirischian et al., 2004). A necessidade de elevado processamento e ao mesmo tempo a limitação física de recursos de uma FPGA, levou ao desenvolvimento de uma arquitetura designada por *Re-configurable Parallel Stream Processor* (RPSP). Esta é composta por duas FPGAs e um banco de memória. Na primeira FPGA são reconfiguradas as várias unidades de processamento *Application Specific Virtual Processors* (ASVPs), que são construídas recorrendo a uma biblioteca de componentes *Virtual Hardware Components* (VHCs) já pré-compilados (*bitstreams*). Na segunda irá correr o sistema operativo responsável por gerir a reconfiguração da anterior. O banco de memória é onde são armazenados os *bitstreams* que compõem a biblioteca de ASVPs e de VHCs.

Cada componente VHC, que é na realidade um *bitstream* parcial que pode ser alocado numa determinada RP, é constituído, como ilustra a Figura 31, por dois elementos: o Elemento de Processamento ou *Processing Element* (PE) (por exemplo: adicionador, multiplicador, *shift*, *Fast Fourier Transform* (FFT), etc); e o Elemento de Interface ou *Interface Element* (IE), interliga o PE ao ASVP.

Através dos *Tri-state Buffers*, cada VHC é interligado às Linhas de Roteamento Global (*Global Routing Lines*), em função do processador ASVP que está a utilizar este VHC.

Para implementar, uma aplicação o RPSP carrega na FPGA o *bitstream* correspondente à parte estática do hardware do ASVP, e carrega da biblioteca de VHCs os *bitstreams* parciais de modo a executar a função desejada. Existem portanto duas bibliotecas de *bitstreams*: uma para a parte fixa de cada ASVP e outra para todas as variantes de cada VHC (para cada VHC terão de existir tantos *bitstreams* parciais, como localizações (RPs) onde se pretenda alocar esse VHC).

Além de acelerar em hardware o processamento de *stream*, esta

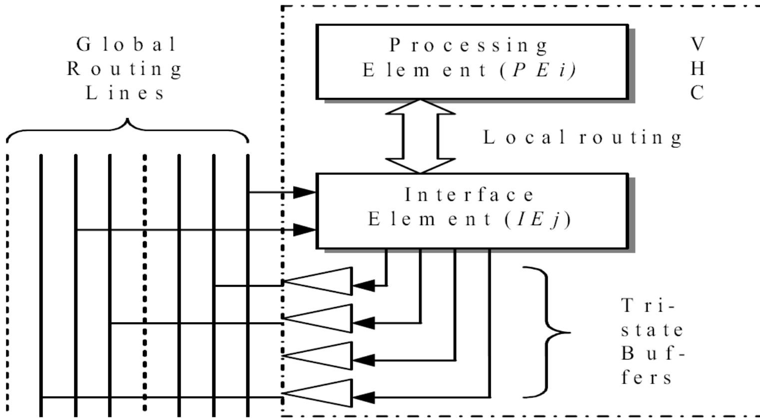


Figura 31 – Micro-arquitetura de um VHC (Kirischian et al., 2004).

arquitetura aproveita a sua estrutura para recuperar da ocorrência de faltas (transitórias e permanentes). O ASVP possui dois níveis de procedimento para recuperar de uma falta:

1. Primeiro nível: dedicado às faltas transitórias (*Soft Errors* como os SEUs) consiste em reprogramar os VHCs usados pelo ASVP ativo nas respetivas RPs (operação normalmente designada por *scrubbing* e que corrige *bit flips* que possam ter ocorrido na memória de configuração da FPGA).

2. Segundo nível: para as faltas permanentes (*Hard Errors* como os SELs, SEBs ou SEGRs), implica realocar o VHC que falha no teste noutra RP que esteja livre (designado pelo autor como troca de *slot*), e sinalizar o atual *slot* como danificado.

Embora o segundo nível implique perda de recursos da FPGA (após ocorrência de uma falta permanente num *slot*), no final obtém-se um considerável incremento de *Reliability* do sistema.

3.5.8.2 Arquitetura com Proteção Contra Radiação

Aproveitando as capacidades da reconfiguração parcial (*Partial Reconfiguration*) usada na arquitetura anteriormente desenvolvida, os autores focam-se na resolução dos problemas causados pela radiação nas FPGAs (Kirischian et al., 2006).

Este trabalho aumentou a performance da plataforma existente. Para isso contribuiu o facto de que quando é detetada uma falta, todo o processo de recuperação da tarefa que sofreu essa falta é realizado sem que aconteça nenhuma interrupção no resto do sistema implementado na FPGA. Passou assim a ser possível efetuar uma recuperação funcional sem qualquer degradação de performance. Além disso, este novo método permite que a arquitetura se adapte às faltas de um modo automático, restabelecendo autonomamente a interligação e sincronismo, minimizando assim a degradação da performance quando uma falta permanente ocorre.

Com vista às aplicações espaciais, esta nova versão torna um sistema implementado numa FPGA mais tolerante à radiação, algo que permitiu um aumento de disponibilidade (*Availability*) e *Reliability* do sistema.

3.5.8.3 Sistema de Computação Reconfigurável

Na seguinte revisão da plataforma desenvolvida ocorreu principalmente uma nova formalização do trabalho feito (Kirischian et al., 2009) (Kirischian et al., 2010). O RPSP passou a ser enquadrado como um Sistema de Computação Reconfigurável ou *Reconfigurable Computing System* (RCS). A demonstração de todo o mecanismo foi feita através da recente plataforma desenvolvida *Multi-task Adaptive Reconfigurable System* (MARS), que inclui agora uma FPGA da família Virtex-4.

A arquitetura RCS, à semelhança do RPSP apresentada na Secção 3.5.8.1, é formada por ASVPs que são compostos por grupos de VHCs consoante a aplicação desejada para cada ASVP. Nesta nova versão, o IE de um VHC, para além de ser responsável pelo interface, passou também a sincronizar o fluxo de dados existente a cada instante

no RCS.

Um melhoramento que a arquitetura RCS anuncia é a infraestrutura de comunicação (*Communication Interconnect*). Esta nova funcionalidade, como mostra a Figura 32, é incluída na parte estática da configuração de um ASVP e tem a função de interligar os vários VHCs e os componentes estáticos (existentes na parte estática) de uma forma mais flexível.

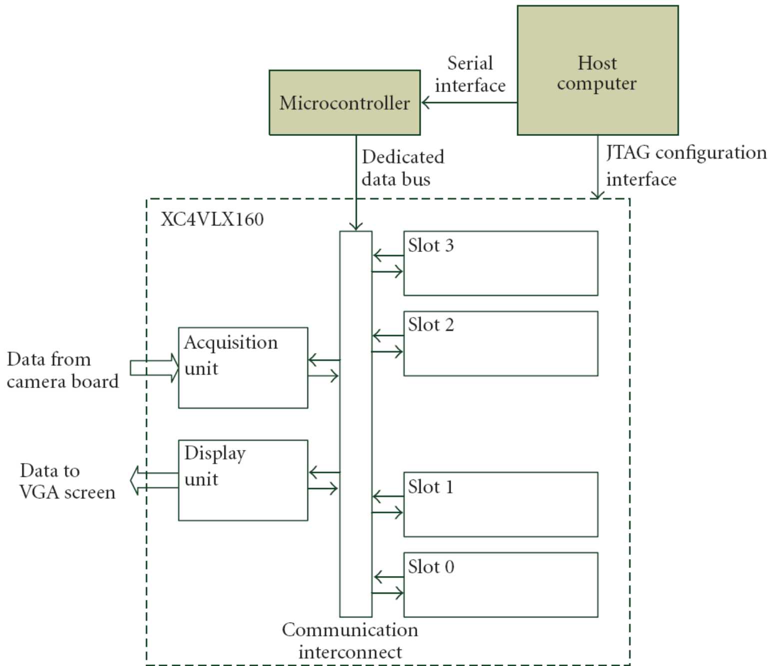


Figura 32 – Arquitetura com quatro *slots* para VHCs e infraestrutura de comunicação (Kirischian et al., 2010).

O *Communication Interconnect* permite assim programar os canais de comunicação entre os VHCs da arquitetura da ASVP. Como o RCS pode receber diferentes ASVPs e como cada uma delas inclui vários VHCs distribuídos pelos seus *slots*, isso obriga a que os *slots* sejam homogéneos (incluírem a mesma quantidade e tipo de recursos), para que qualquer VHC possa ser alocado em qualquer dos *slots*. No entanto, para

cada *slot* é necessário existir um *bitstream* parcial diferente de cada VHC. Esta igualdade de recursos nos *slots* obriga a que os VHCs necessitem sensivelmente da mesma quantidade de recursos (existente num *slot*), ou então existir um considerável desperdício de recursos da FPGA. Quantos mais *slots* livres uma ASVP tiver, mais faltas permanentes que ocorram em VHCs são possíveis de recuperar.

3.5.8.4 *Framework* de ERSs com Realocação de VHCs

Na seguinte iteração da plataforma ocorre uma uniformização da arquitetura anterior (RCS), de modo a que ela possa ser mais escalável (Dumitriu and Kirischian, 2010). O resultado foi um *framework* que acelera a implementação de *Embedded Reconfigurable Systems* (ERSs) baseados em VHCs, e aumenta o número de aplicações possíveis de processar com este trabalho. Fundamentalmente, em vez das várias ASVPs possíveis de reconfigurar na FPGA, em função da aplicação desejada, existe apenas uma ASVP (designada agora ERS), que é muito mais flexível e permite ser composta com VHCs de modo a realizar todas as aplicações necessárias. Relativamente ao elemento VHC, este continua a ser a chave nesta arquitetura, tendo sido alvo de uma importante evolução na sua estrutura, tal como mostra a Figura 33.

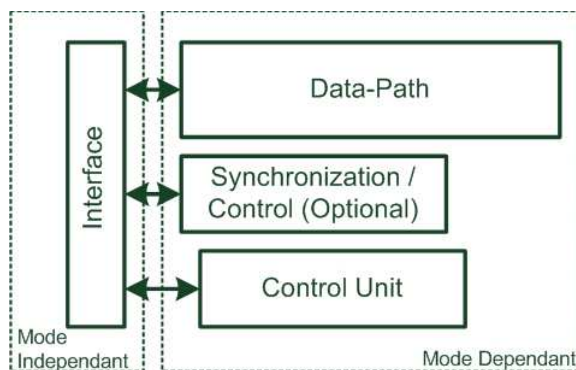


Figura 33 – Estrutura do VHC com opção de modo (Dumitriu and Kirischian, 2010).

Comparando com a micro-arquitetura da Figura 31, a parte funcional (PE) deu lugar a um *Data-Path* que tornou o VHC mais flexível. O *Data-Path* é assim responsável pela execução da(s) função(ões) associada(s) ao VHC. O VHC inclui ainda uma unidade de controlo (*Control Unit*) e a opção de um módulo de sincronização (*Synchronization / Control*) que completam a parte cuja configuração depende do modo como o VHC é configurado (*Mode Dependant*). Não dependendo do modo configurado (*Mode Independant*), continua a existir o módulo Interface que completa a totalidade de um VHC.

O ERS mantém a arquitetura com uma parte fixa e um conjunto de *slots* (de tamanho homogéneo) para alocar VHCs. No entanto, como mostra a Figura 34, a parte estática foi reduzida a um componente estático responsável pela gestão da interligação de *slots* e entradas/saídas, e toda a estrutura total passou ela própria a ser também menos heterogénea.

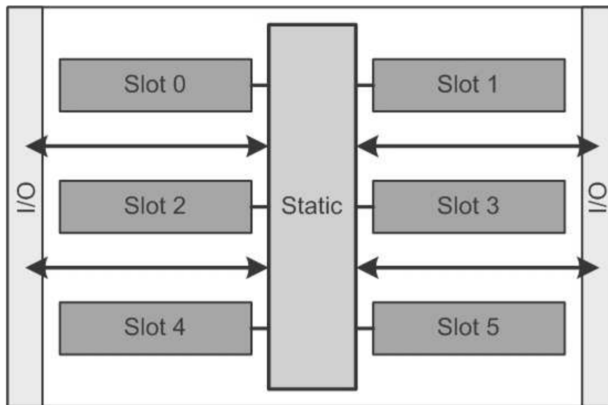


Figura 34 – Arquitetura geral um ERS com *slots* para VHCs (Dumitriu and Kirischian, 2010).

Formalmente tudo passou a ser um componente (VHC) que é interligado com outros componentes através de uma infraestrutura de comunicação. Esta reestruturação deu origem a um *framework* ainda mais indicado para aplicações espaciais. Sendo a arquitetura maioritariamente composta por *slots*, a realocação de VHCs permite a recuperação

de faltas permanentes de uma maior quantidade de recursos da FPGA. Além disso, como a biblioteca dos *bitstreams* das partes fixas deixou de ser necessária, significa menos memória externa para o armazenamento e por isso menos hardware sujeito a radiação. Ou seja, permitiu um novo aumento de disponibilidade (*Availability*) e *Reliability* do sistema.

3.5.8.5 ***Framework* para Aplicações Espaciais com Auto-recuperação de Faltas**

Se nos trabalhos anteriores foram apresentadas formas para ultrapassar a ocorrência de faltas, neste novo *framework*, o foco foi investigar soluções para atenuar o efeito das faltas que possam ocorrer (Dumitriu et al., 2012). Para alcançar este objetivo, o controlo na arquitetura foi ainda mais descentralizado, reduzindo assim a probabilidade de uma falta permanente colocar em risco a operacionalidade do sistema ou parte dele. Ou seja, evitar que qualquer função da arquitetura esteja centralizada num único módulo, que ao sofrer uma falta permanente devido à radiação possa comprometer todo o sistema.

Na reformulação do *framework*, a anterior arquitetura RCS evoluiu para a arquitetura *Dynamically-reconfigurable Platform with Relocatable Components* (DPRC), com um novo método para auto-sincronização das atividades de processamento interno, sem depender de nenhum processador externo auxiliar. Outra importante alteração da arquitetura foi a nível do próprio VHC que passou a ser baseado em um ou vários *Intellectual Property* (IP) *cores*. Juntamente com isto, este novo *framework* adicionou uma unidade de sincronização a cada VHC e, como mostra a arquitetura da Figura 35, delegou-lhe a responsabilidade de autonomamente sincronizar-se com o restante sistema.

Desta forma, se ocorrer uma falta permanente nos recursos de um determinado *slot*, bastará realocar o VHC noutro *slot* compatível disponível e todo o sistema voltará a funcionar devidamente sincronizado sem necessidade de intervenção de nenhum outro módulo.

Hierarquicamente, num dado instante a DPRC implementa um *Application Specific Processor* (ASP) (em versões anteriores designada

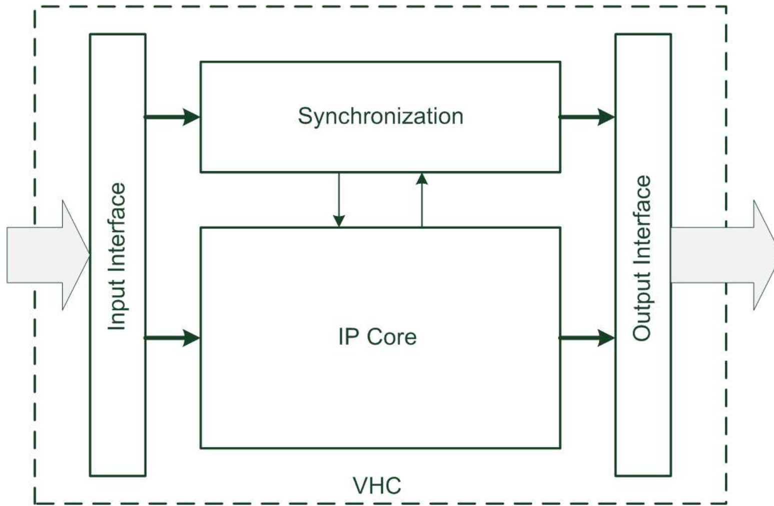


Figura 35 – Arquitetura geral da VHC (Dumitriu et al., 2012).

por ASVP), que estruturalmente significa que um grupo de VHCs é alocado e devidamente interligado através da estrutura de comunicação para implementar uma aplicação. Nesta versão do *framework* foi ainda introduzida a distinção entre VHCs estáticos, que estão sempre presentes fisicamente na FPGA, e dinâmicos, que só são alocados num *slot* quando são necessários.

Resumindo, neste novo passo na evolução do *framework*, o hardware responsável pela comunicação e sincronização foi reorganizado, de modo a que estas operações deixassem de ser centralizadas, evitando assim que uma falta permanente possa provocar falhas nestas operações e com isso inutilizar todo o sistema. Ficou por isso um *framework* ainda mais orientado para aplicações espaciais.

3.5.8.6 Mecanismos de Auto-Integração de *System-on-Chip* (SoC) para Sistemas de Reconfiguração Dinâmica

Continuando o trabalho anteriormente realizado, os autores fizeram evoluir a plataforma DPRC e os componentes VHCs desenvolvidos e apresentados na Secção 3.5.8.5, para um sistema *Dynamic Partially*

Reconfigurable (DPR) e unidades *Collaborative Macro-Functional Units* (CMFUs) respetivamente (Dumitriu and Kirischian, 2013). A unidade CMFU permite a implementação de um mecanismo de reconfiguração distribuída, que desta forma evita que o subsistema de recuperação trave internamente no caso de algum módulo sofrer uma falta. Para isso, cada unidade alocada num *slot*, passou a incluir mais capacidade de auto-sincronização com o resto do sistema. Essa capacidade inclui informar os restantes módulos sobre a necessidade de reconfiguração ou realocação.

Na Figura 36 é observável que a composição do novo sistema segue a organização anterior exposta na Figura 34.

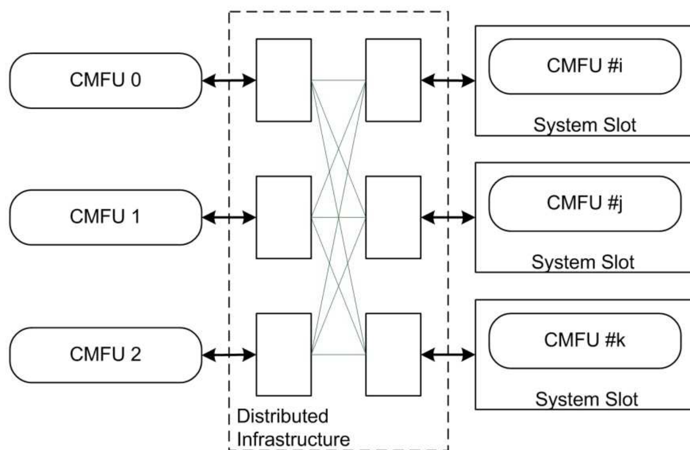


Figura 36 – Estrutura Geral do Sistema Colaborativo (Dumitriu and Kirischian, 2013).

Seguindo a mesma filosofia do DPRC da Secção 3.5.8.5, o sistema é constituído por CMFUs estáticos ou alocados dinamicamente num *System Slot*. A evolução do VHC (Figura 35) para o novo CMFU, esquematizado na Figura 37, é o maior contributo deste trabalho.

O módulo de sincronização do anterior VHC evoluiu para uma unidade de cooperação (*Co-Op Unit*), responsável por todas as atividades colaborativas existentes no CMFU e qualquer necessidade específica para o interface de uma determinada aplicação. A unidade *Co-Op Unit*

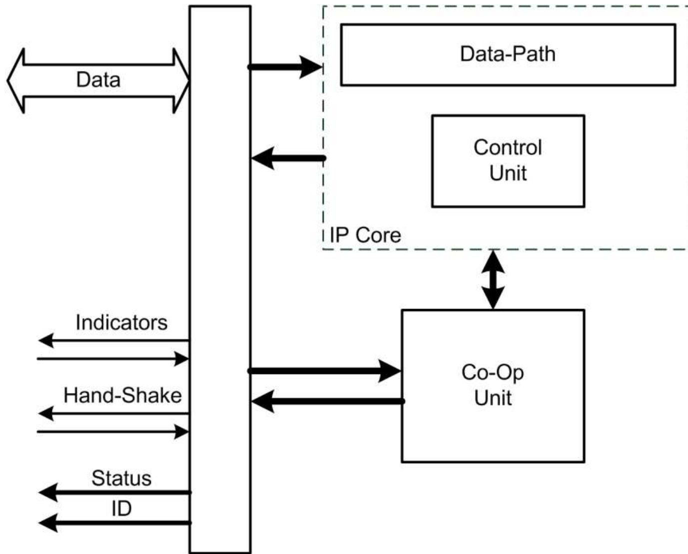


Figura 37 – Arquitetura de um CMFU (Dumitriu and Kirischian, 2013).

tornou por isso o interface com o restante sistema mais completo, tendo a seu cargo a gestão de várias informações no CMFU como: onde o CMFU deve ser conectado no sistema; quando o CMFU pode ser conectado e desconectado do sistema de uma forma segura; ou quando o IP *core* local precisa de ser ativado/desativado para executar a sua operação. Sinais como *Indicators* e *Hand-Shake* existentes no interface são utilizados pela unidade *Co-Op Unit* para determinar quando o CMFU pode ser conectado e desconectado do sistema, e quando o IP *core* deve ficar ativo.

Distribuindo as funções de interligação e sincronização pelas CMFUs, evitando a necessidade de ter um módulo para realizar essas funções de um modo centralizado, o sistema torna-se mais tolerante a faltas permanentes. Isto porque, se ocorrer uma falta permanente num determinado *System Slot*, bastará realocar o CMFU lá alocado para outro *System Slot* que se encontre disponível e livre de faltas permanentes.

3.5.8.7 Mecanismo Descentralizado para Recuperação de Faltas para Sistemas Espaciais baseados em FPGAs

Embora existam FPGAs especialmente desenvolvidas para suportar uma tolerância ao *Total Ionizing Dose* (TID) superior a 1 Mrad (Si) (Xilinx Inc., 2012a), o seu preço também é uma ordem de magnitude mais elevado que as FPGAs concebidas para aplicações terrestres. Para além do preço elevado, a disponibilidade de tais FPGAs para aplicações comerciais é limitada pela regulamentação da produção e da comercialização.

Com o objetivo de usar as FPGAs comerciais em aplicações espaciais, e tendo em consideração que nesse ambiente o nível de radiação pode originar faltas permanentes nos recursos de uma FPGA devido ao TID, os autores decidiram dotar a plataforma anteriormente desenvolvida com um *Built-In Self-Recovery* (BISR) (Dumitriu et al., 2014). Para isso desenvolveram um método e um mecanismo para a recuperação de faltas (transitórias e permanentes) em sistemas *System-on-Chip* (SoC) implementados em FPGAs, usando como base as já desenvolvidas CMFUs. Cada CMFU consiste numa macro-função com um *Data-Path* específico, uma unidade de controlo e circuitos que providenciam funções de auto-integração, auto-sincronização e de auto-recuperação para o CMFU, não sendo por isso necessário qualquer controlo centralizado. Na Figura 38 encontra-se a arquitetura geral do sistema implementado numa FPGA recorrendo a CMFUs.

Embora semelhante à anterior arquitetura da Figura 36, existem duas redundâncias que são visíveis: dois mecanismos de reconfiguração (*Configuration Engines*) e dois armazenamentos de *bitstreams* (*Bitstream Storages*). Para isso esta nova abordagem utiliza os dois *Internal Configuration Access Ports* (ICAPs) existentes nas recentes famílias de FPGA da Xilinx. Apenas um dos dois ICAPs se encontra ativo em qualquer momento, podendo o outro passar a ser utilizado caso seja detetada uma falta no que está atualmente ativo. Com esta duplicação a plataforma aumenta a confiabilidade (*reliability*) até no canal de reconfiguração, que é a base do mecanismo de realocação.

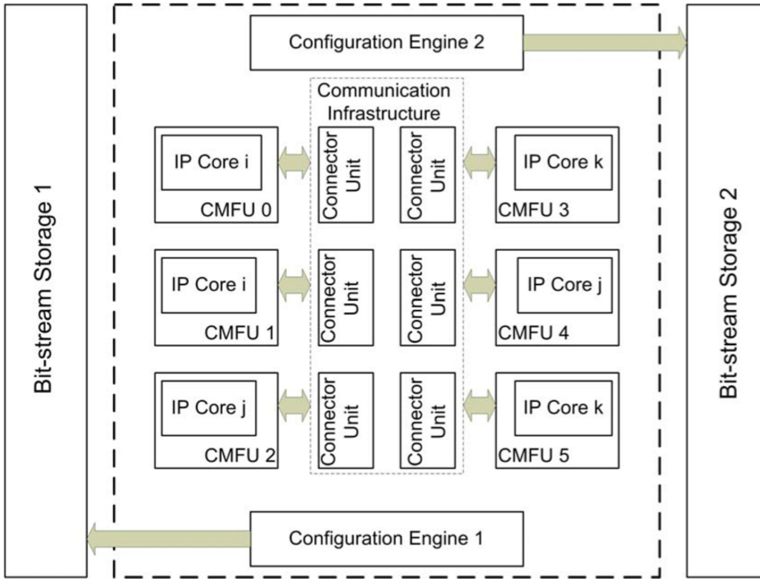


Figura 38 – Arquitetura Geral do Sistema implementado numa FPGA (Dumitriu et al., 2014).

Outra regra apresentada neste novo método, é que cada módulo de reconfiguração parcial utilizado pela plataforma deve incluir um *Built-In Self-Test* (BIST), de modo a que o teste para deteção de faltas seja feito localmente. A gestão de quando o BIST deve ser executado é mais uma tarefa que foi adicionada ao *Co-Op Unit* que faz parte da CMFU (consultar Figura 37).

Quando ocorre uma falta, esta pode ser transitória ou permanente. Normalmente, a solução para as diferenciar passa por reconfigurar uma ou várias vezes o módulo onde foi detetada a falta. Se o módulo voltar a funcionar corretamente, trata-se de uma falta transitória e está a recuperação concluída, caso contrário, será necessário realocar o módulo noutra RP que esteja disponível. No mecanismo proposto, logo que uma falta seja detetada num módulo, esse módulo é imediatamente realocado noutra RP e só depois é feita a triagem entre falta transitória ou permanente. Desta forma o sistema não fica bloqueado enquanto

se reconfigura as vezes necessárias para verificar se não é uma falta transitória.

Resumindo, com este trabalho os autores incrementaram de várias formas o nível de robustez da plataforma desenvolvida face à ocorrência de faltas transitórias e permanentes, abrindo assim ainda mais a porta para o uso de FPGAs comerciais em aplicações espaciais. No entanto, a plataforma continua a não aproveitar as suas capacidades para prevenção de faltas permanentes devido a *aging* (NBTI ou PBTI).

3.5.8.8 Arquitetura Reconfigurável com Capacidade de Adaptação aos Recursos Disponíveis pelo Sistema

Tendo já alcançado um bom grau de disponibilidade (*Availability*) e confiabilidade (*Reliability*) para a plataforma desenvolvida, e uma vez que a potência disponível é igualmente limitada em aplicações espaciais, os autores decidiram pesquisar formas de reduzir o consumo de potência em arquitetura reconfigurável criada (Dumitriu et al., 2015a). Assim, nesta etapa, para um dado algoritmo são geradas diferentes variantes da mesma arquitetura em função de três atributos: utilização de recursos da FPGA; performance; e consumo de potência.

Esta nova abordagem determina a implementação de múltiplas arquiteturas de uma determinada tarefa ou aplicação, onde cada uma delas terá uma estrutura interna diferente, que implicará também características diferentes (tal como o uso de recursos, consumo energia e desempenho). Através da implementação de uma arquitetura específica, um sistema de computação reconfigurável pode atenuar alterações como a redução da energia e recursos disponíveis, ou alterações na taxa de processamento.

Relativamente a situações em que faltas permanentes afetem o sistema, tal como exemplifica a Figura 39, são agora propostas duas abordagens distintas que podem ser utilizadas para isolar as regiões com faltas: **A)** realocação do módulo para outro *Slot* disponível e sem faltas; ou **B)** alterar a estrutura lógica e física da arquitetura, sem mudanças do seu local absoluto.

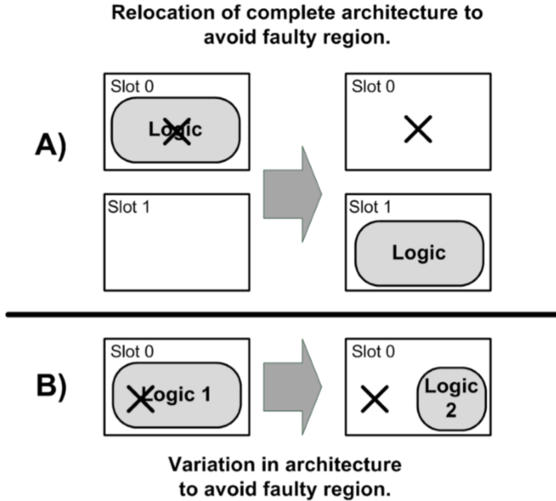


Figura 39 – Alternativas de Realocação de Módulos na FPGA (Dumitriu et al., 2015a).

A abordagem da Figura 39 **B)** é por isso uma nova forma de recuperação de faltas permanentes que este trabalho propõe. Quando é detetada uma falta permanente no sistema, e não há nenhum *Slot* livre para realocar o módulo, então uma versão mais reduzida do mesmo algoritmo pode ser carregada de modo a evitar a região onde ocorreu a falta permanente. Para que este processo seja possível é necessário gerar várias versões, que tal como demonstra o exemplo da Figura 39, utilizem diferentes disposições de recursos existentes em cada *Slot* (que fisicamente corresponde a uma RP).

No exemplo da Figura 39, o módulo possui uma versão para a performance máxima e que por isso precisa de mais recursos disponíveis (*Maximal Variant Slot Layout*). Tem ainda mais duas versões com uma performance menor e que por isso necessita de menos recursos. Como sobra uma quantidade considerável de recursos, uma parte desse excedente é excluído através de *constraints* (*Constrained Logic*), sendo a área excluída diferente nas versões com menor performance (*Minimal Variant Slot Layouts*).

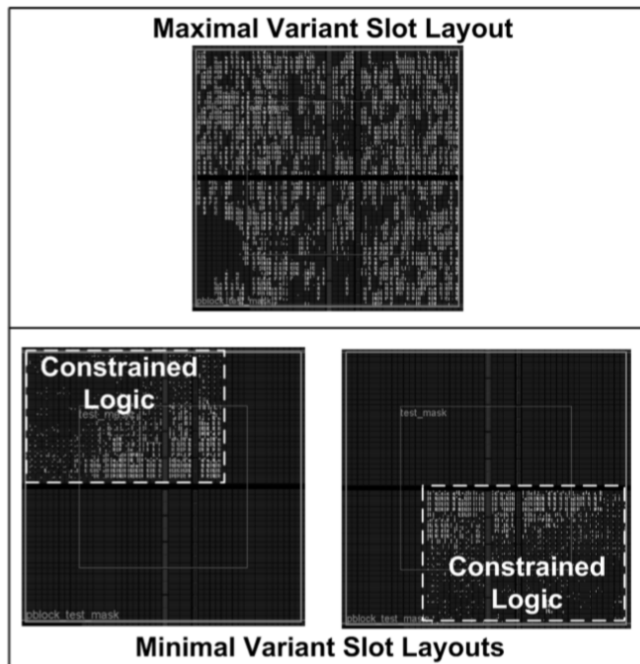


Figura 40 – Controlo da Disposição dos Recursos Internos de um *Slot* (Dumitriu et al., 2015a).

Este novo modo de recuperação (Figura 39 **B)**) tem a importante vantagem de não ser necessário excluir *Slots* sempre que ocorra nele uma falta. No entanto e seguindo este exemplo, se cada módulo tiver três versões para cada *Slot*, o tamanho da biblioteca de *bitstreams* parciais irá triplicar. Em relação a prevenir o envelhecimento devido a alguma forma de BTI, a plataforma continua a não apresentar qualquer tipo de solução.

3.5.8.9 Arquitetura Auto-Organizada Reconfigurável Ajustável para Recuperar de Faltas Transitórias e Permanentes

Na mais recente publicação sobre o desenvolvimento da arquitetura orientada para aplicações espaciais (Dumitriu et al., 2015b), os

autores usam um artigo longo num periódico (*IEEE Transactions on Computers*), para com algumas alterações nas terminologias, descreverem com detalhe o trabalho realizado anteriormente (Dumitriu et al., 2014). A arquitetura e o método *Multimodal Adaptive Collaborative Reconfigurable self-Organized System* (MACROS) são introduzidos tal como mostra a Figura 41.

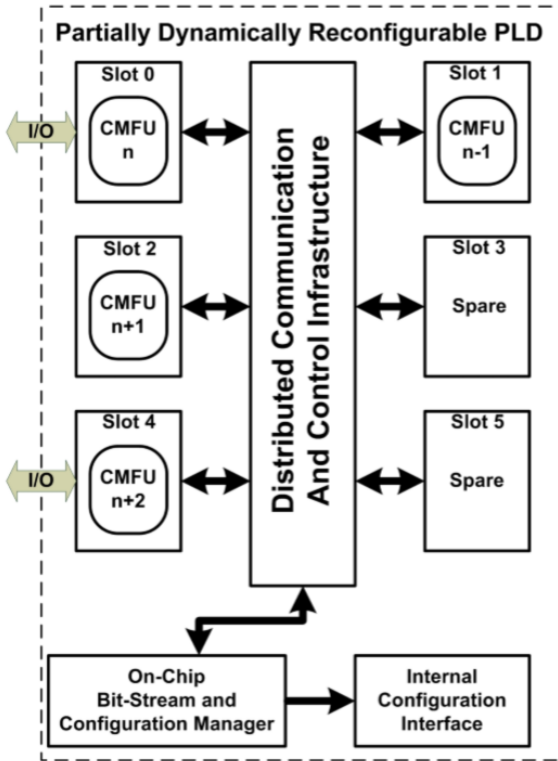


Figura 41 – Arquitetura Geral do MACROS (Dumitriu et al., 2015b).

Em tudo semelhante à arquitetura da Figura 38, um MACROS é composto de várias CMFUs, alguns *Slots* livres para realocação (utilizados para atividades de mitigação), um *Distributed Communication and Control Interface* (DCCI) e um *Bit-Stream and Configuration Manager* (BCM). Para além de executar as aplicações pretendidas, o objetivo

principal desta arquitetura é minimizar pontos críticos onde uma falta permanente possa colocar em causa todo o sistema, distribuindo por isso a funcionalidade do sistema entre os vários componentes. Assim, se em algum ponto ocorrer uma falta permanente, o restante sistema será capaz de recuperar dessa falta. O tamanho da biblioteca de *bitstreams* parciais é que continua a exigir um volume importante de memória externa que precisará de ser igualmente mitigado em relação a faltas no futuro.

3.5.8.10 Conclusão, Vantagens e Desvantagens

Após uma década de investigação, os autores de todas as etapas alcançaram uma arquitetura dotada de um *Built-In Self-Recovery* (BISR). Numa fase inicial possuía duas bibliotecas de *bitstreams*, uma para versões estáticas (*bitstreams* completos) e outra para os módulos reconfiguráveis (*bitstreams* parciais). A partir da versão desenvolvida no trabalho (Dumitriu and Kirischian, 2010) a parte estática passou a ter apenas um *bitstream*, deixando assim de existir a respetiva biblioteca.

Como principal vantagem, devido à descentralização de tarefas como sincronização, comunicação e realocação, a plataforma é tolerante a faltas (transitórias e permanentes), em quase todos os seus módulos. Tal opção torna esta arquitetura mais robusta e preparada para aplicações espaciais do que qualquer outra estudada.

No fluxo de implementação desta arquitetura não é usado qualquer mecanismo para que o mesmo *bitstream* parcial possa ser realocado em mais do que um *Slot*. Devido a isso, o volume de memória necessário para armazenar a biblioteca é N vezes superior, sendo N o número de *Slots* disponíveis para realocar módulos do sistema. Esta é portanto uma importante desvantagem.

Outra desvantagem é o facto de que quando ocorre uma falta permanente num *Slot*, por norma toda a RP correspondente a esse *Slot* fica excluída de ser utilizada pelo sistema, o que obriga a projetar o sistema com *Slots* extra, para que seja possível recuperar de faltas permanentes. Isto implica mais consumo de recursos da FPGA. A única exceção

a esta regra é a abordagem **B)** da Figura 39 no trabalho (Dumitriu et al., 2015a). Nesta alternativa de recuperar de uma falta permanente, implica apenas alterar a estrutura lógica e física da arquitetura, através do carregamento de outro *bitstream* parcial, sem mudar de *Slot*. Embora interessante, esta abordagem tem uma granularidade elevada e implica um volume de memória ainda maior para armazenar todas as variantes de *bitstreams* parciais para cada módulo.

3.6 RESUMO DA REVISÃO DO ESTADO DA ARTE E CONCLUSÃO

Após a exposição de algum estado da arte relativo às ameaças das faltas que podem ocorrer em dispositivos eletrónicos como as FPGAs (Secção 3.1, Secção 3.2 e Secção 3.3), foram pesquisados e examinados de uma forma global diversos trabalhos relacionados com métodos de realocação na Secção 3.4 (relacionados com a camada de suporte referida na Secção 1.2), e na Secção 3.5 (relacionadas com a camada de aplicação), foram procurados e estudados trabalhos relacionados com plataformas com tolerância a faltas permanentes. Com o resultado da análise destas duas últimas secções foi preenchida a Tabela 3.

Para uma melhor interpretação da Tabela 3 várias características foram analisadas. No entanto existem quatro pormenores que distinguem os vários trabalhos e que ajudam a enquadrar a contribuição desta tese. Na seguinte lista encontram-se as características extraídas, encontrando-se a negrito a seleção das quatro com maior relevo.

- Fluxo de Geração de Realocação. Em alguns trabalhos todo o processo é realizado de uma forma manual, noutros foram desenvolvidas ferramentas para automatizar o processo. Possui por isso as opções: (**M**) Manual, (**A**) Automático ou (**-**) Impossível Avaliar;

- Analisa as Possibilidades de Localização. Embora não seja o foco desta tese, num trabalho futuro poderá ser importante desenvolver

Tabela 3 – Comparativo dos trabalhos relativos a Realocação de Módulos e Plataformas com Tolerância a Faltas Permanentes

	Fluxo de Geração de Realocação **M**anual ou **A**utomático	Analisa as Possibilidades de Localização (**S**/**N**)	Alguma Liberdade no Roteamento dos Sinais (**S**/**N**)	Gestão de Sinais de Relógio na Realocação (**S**/**N**)	Realocação **Ú**nica ou **M**últipla de Módulos	Suporta Famílias de FPGAs Recentes (**S**/**N**)	Tolerante a Faltas Transitórias e Permanentes	Recupera de Faltas Permanentes (**S**/**N**)	Exclui Partição para Recuperar de Faltas Permanentes	Controlo de Realocação Descentralizado	Prevenção *Aging*
(Ichinomiya et al., 2012b)	M	N	N	-	M	S	-	-	-	N	N
(Ichinomiya et al., 2012a)	M	N	N	-	M	S	-	-	-	N	N
(Koch et al., 2008)	A	N	N	-	M	N	-	-	-	N	N
(Beckhoff et al., 2012)	A	N	N	-	M	S	-	-	-	N	N
(Beckhoff et al., 2013a)	A	N	N	-	M	S	-	-	-	N	N
(Touiza et al., 2012)	A	N	N	-	M	S	-	-	-	N	N
(Drahonovský et al., 2013)	M	N	N	-	M	S	-	-	-	N	N
(Steiner et al., 2011)	-	-	S	-	-	S	N	N	-	-	N
(Love et al., 2013)	-	-	S	-	-	S	N	N	-	-	N
(Flynn et al., 2009)	-	-	S	S	-	S	N	N	-	-	N
(Backasch et al., 2014)	A	S	S	-	M	S	-	-	-	N	N
(Xilinx Inc., 2013d)	A	N	S	S	U	S	-	-	-	-	N
(Xilinx Inc., 2016c)	-	-	S	-	U	S	S	N	-	-	N
(Gericota, 2003)	-	-	S	-	-	N	S	S	N	-	N
(Montminy et al., 2007)	-	N	S	-	U	N	S	S	S	N	N
(Iturbe et al., 2011)	-	N	S	-	M	S	S	-	-	N	N
(Iturbe et al., 2012)	-	N	S	S	M	S	-	-	-	N	N
(Bolchini et al., 2012)	-	N	S	-	U	S	S	S	S	N	N
(Ochoa-Ruiz et al., 2013)	A	N	N	-	M	S	-	-	-	N	N
(Espinosa et al., 2012)	A	N	S	-	U	S	S	S	S	N	N
(Lawson et al., 2014)	-	-	-	-	-	N	S	S	N	-	N
(Kirischian et al., 2006)	A	N	S	-	U	S	S	S	S	N	N
(Dumitriu and Kirischian, 2010)	A	N	S	-	U	S	S	S	S	N	N
(Dumitriu et al., 2014)	A	N	S	-	U	S	S	S	S	S	N
(Dumitriu et al., 2015a)	A	N	S	-	U	S	S	S	N	S	N
(Dumitriu et al., 2015b)	A	N	S	-	U	S	S	S	S	S	N

ferramentas para estudar previamente a localização das RPs. Com as opções: (**S**) Sim, (**N**) Não ou (**-**) Impossível Avaliar;

- Possui Alguma Liberdade no Roteamento dos Sinais. Determinados trabalhos fixam o roteamento dos sinais na fase inicial, o que retira a liberdade do roteamento nas fases seguintes e acaba por provocar uma deterioração superior do o caminho. Tem as opções: (**S**) Sim, (**N**) Não ou (**-**) Impossível Avaliar;

- Existe Gestão de Sinais de Relógio na Realocação. Quando o autor propõe algum mecanismo extra para gerir os sinais de relógio. Com as opções: (**S**) Sim, (**N**) Não ou (**-**) Impossível Avaliar;

- **Qual o Tipo de Realocação dos Módulos. A realocação pode ser baseada no normal fluxo das ferramentas e implicar um único *bitstream* parcial para cada módulo em cada partição, ou usar um fluxo alternativo que permita um mesmo *bitstream* poder ser realocado em múltiplas partições. Com as opções: (Ú) Única, (M) Múltipla ou (-) Impossível Avaliar;**

- Suporta Famílias de FPGAs Recentes. São consideradas FPGAs recentes as famílias após a família Virtex-6 (inclusive). Com as opções: (**S**) Sim ou (**N**) Não;

- É Tolerante a Faltas Transitórias e Permanentes. Uma característica associada a plataformas desenvolvidas de modo a serem tolerantes a faltas transitórias e permanentes. Com as opções: (**S**) Sim, (**N**) Não ou (**-**) Impossível Avaliar;

- **Recupera de Faltas Permanentes. Se a plataforma proposta possui algum mecanismo que permita o sistema recuperar de faltas permanentes. É um importante detalhe que é relacionado com esta tese e é avaliado com as seguintes opções: (S) Sim, (N) Não ou (-) Impossível Avaliar;**

- **Necessita de Excluir a Partição para Recuperar de Faltas Permanentes. Outra novidade que esta tese apresenta é recuperar de uma falta permanente sem excluir definitivamente uma RP. Por isso também este parâmetro é comparado com os restantes trabalhos e inclui as opções: (S) Sim, (N) Não ou (-) Impossível Avaliar;**

- Inclui Controlo de Realocação Descentralizado. Não sendo uma prioridade nesta fase do trabalho desenvolvido, o descentralizar do controlo da gestão do mecanismo de recuperação de faltas, é igualmente uma área importante para futuras iterações da plataforma desenvolvida. A avaliação foi feita com as opções: (**S**) Sim, (**N**) Não ou (**-**) Impossível Avaliar;

- **Aborda a Prevenção de *Aging*. Embora uma área não abordada nos trabalhos estudados, o prevenir o envelhecimento dos recursos da FPGA, foi também uma área de foco nesta tese. Este ponto foi classificado com as opções: (S) Sim ou (N) Não;**

Embora cada trabalho tenha o seu conjunto próprio de características, na Tabela 3 observa-se três importantes realidades:

- A realocação do mesmo *bitstream* parcial em múltiplas RPs não é utilizado em prol da recuperação de faltas permanentes. As únicas plataformas que talvez poderiam fazê-lo são os trabalhos (Ochoa-Ruiz et al., 2013) e (Iturbe et al., 2011), no entanto nenhuma delas tem o foco para recuperar de qualquer tipo de faltas, e no caso do (Iturbe et al., 2011), como toda a comunicação assenta sobre o ICAP, deixa esta abordagem muito limitada a nível do tipo de aplicações;

- Quando a realocação (permitida pelo normal fluxo das ferramentas de Reconfiguração Parcial disponibilizadas pela Xilinx) é usada para remoção de uma falta permanente, a partição é excluída. A única exceção é a abordagem do trabalho (Dumitriu et al.,

2015a), mas que para além de implicar a redução de performance do sistema, obriga a um maior aumento da memória exigida para armazenar toda a biblioteca de *bitstreams* parciais;

- Em nenhuma das arquiteturas estudadas os autores tentam qualquer aproximação para evitar o envelhecimento devido aos *stress effects* nos transístores da FPGA, principalmente nos MOSFETs pMOS, onde a performance e confiabilidade devida ao NBTI é influenciada (Ioannou and Rosa, 2014).

Após a revisão e pesquisa do estado da arte no que diz respeito a mecanismos de realocação de *bitstreams* parciais e plataformas tolerantes a faltas permanentes, ficou claro o não uso de um mecanismo de realocação numa plataforma tolerante a faltas permanentes, onde a recuperação de uma falta permanente não obrigue à exclusão de toda a partição reconfigurável (RP) e, ao mesmo tempo aproveitar o mecanismo de realocação para minimizar o envelhecimento dos recursos da FPGA devido ao NBTI e PBTI. Ou seja, embora um mecanismo que permita a realocação de *bitstreams* parciais em várias RPs possa tornar mais eficiente uma plataforma que tenha por base recuperar de faltas permanente, pois precisa de apenas um *bitstream* parcial para cada módulo, o que diminui drasticamente o tamanho da biblioteca de *bitstreams* e ao mesmo tempo a realocação pode ser realizada por uma simples cópia interna de memória de configuração). Se a este facto for somado um planeamento da redistribuição dos recursos das RPs não utilizados em cada módulo, a ocorrência de uma falta permanente pode ser ultrapassada pelo sistema sem excluir a RP onde ocorreu a falta, incrementando assim ainda mais a eficiência em relação às soluções existentes.

Por estas razões, foram desenvolvidas nesta tese estratégias que permitem criar uma nova plataforma para aplicações espaciais em FPGA, que usa um mecanismo de realocação múltipla em prol da recuperação de faltas permanentes do sistema, e onde a recuperação de uma falta permanente não implique a exclusão de toda a RP. Assim, foi obtida uma plataforma muito mais eficiente a nível de recursos (biblioteca de *bitstreams* parciais mais reduzida e sem necessidade de excluir a

partição onde ocorreu a falta), incrementando assim a disponibilidade (*availability*) e confiabilidade (*reliability*) dos sistemas implementados.

No próximo capítulo, realizando uma análise de probabilidades e estatística, serão realçadas as vantagens que podem obter-se recorrendo à reconfiguração parcial dinâmica e organizando os recursos da FPGA numa determinada ordem. Após essa análise, a estrutura global desta tese é definida em blocos que serão depois detalhados.

Parte II

MATERIAIS E MÉTODOS

4 FUNDAMENTAÇÃO E ESTRATÉGIAS PROPOSTAS

No capítulo anterior, após uma análise abrangente do estado da arte, concluiu-se que os mecanismos que permitem a realocação de *bitstreams* parciais em múltiplas partições não eram usados em plataformas para aumentar a tolerância a faltas permanentes aos sistemas nelas implementados. Neste capítulo serão demonstradas as vantagens de usar este mecanismo nas estratégias propostas. As vantagens evidenciadas visam demonstrar que é possível um sistema implementado numa FPGA recuperar de uma falta permanente, e ao mesmo tempo reduzir a probabilidade de várias faltas originarem falhas no sistema, o que é de extrema utilidade, principalmente quando se equaciona utilizar FPGAs em missões espaciais, onde os dispositivos eletrónicos estão sujeitos a um maior desgaste devido às caraterísticas do meio de operação. Finalizada a análise, é apresentada e detalhada a arquitetura geral total do que é proposto desenvolver nesta tese.

4.1 ANÁLISE PROBABILÍSTICA DAS FALTAS PERMANENTES NUMA FPGA

A radiação a que uma FPGA está sujeita no espaço não é constante, pois existe uma dependência de acontecimentos como tempestades solares e mesmo da localização do hardware que está longe de ser estática. Visando facilitar a obtenção de uma modelação matemática, vamos considerar que a radiação existente num determinado ambiente hostil provoca uma cadência de faltas permanentes constante.

Com esta simplificação, serão apresentados nas próximas duas secções dois possíveis cenários para aumentar a tolerância a faltas permanentes de um sistema digital implementado numa FPGA.

4.1.1 Utilizando apenas $1/r$ dos recursos: Maximização da tolerância a faltas permanentes

Imagine-se um cenário em que numa FPGA ocorre em média uma falta permanente por cada período de tempo T no dispositivo,

e que nesta é implementado um circuito que usa exatamente 1/4 dos seus recursos. Os recursos utilizados estão concentrados na região **A** do dispositivo, tal como ilustrado na Figura 42.

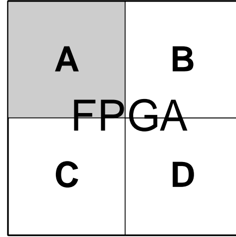


Figura 42 – Exemplo de um circuito implementado em apenas 1/4 da FPGA.

Considerando que a distribuição das faltas permanentes que ocorrem na FPGA é uniforme, assumindo que a FPGA sofreu uma falta permanente ao fim de um período de tempo T e definindo $A =${A região **A** sofreu pelo menos uma falta permanente ao fim de um período de tempo T}, então a probabilidade $\mathbb{P}(A) = \frac{1}{4}$ (J. Susan Milton and Isermann, 2002).

Generalizando para o caso em que a FPGA está dividida em r regiões (de tamanho exatamente igual) e sofreu n faltas permanentes ao fim de um período de tempo T, a probabilidade de pelo menos uma falta permanente ter danificado a região onde se encontra implementado o circuito, definindo $B =${A região onde está implementado o circuito sofreu pelo menos uma falta permanente ao fim de um período de tempo T}, é calculada pela Equação 4.1 (J. Susan Milton and Isermann, 2002).

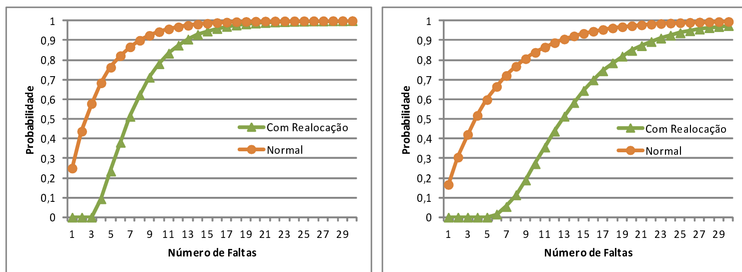
$$\mathbb{P}(B) = 1 - \left(\frac{r-1}{r}\right)^n \tag{4.1}$$

Assumindo agora que cada uma das r regiões da FPGA corresponde a uma partição reconfigurável (RP), e que em caso de uma falta permanente danificar o normal funcionamento do circuito implementado, um mecanismo realoca-o numa outra região isenta de faltas (até esgotar as regiões disponíveis), então para que o circuito seja irreme-

diavelmente danificado devido às n faltas permanentes, é necessário que todas as regiões tenham sofrido pelo menos uma falta permanente. Definindo $C =${Todas as regiões sofreram pelo menos uma falta permanente ao fim de um período de tempo T}, o cálculo de $\mathbb{P}(C)$ é igual a um menos a probabilidade de pelo menos uma região não ter sofrido nenhuma falta permanente e é obtido do seguinte modo (considerando que $n \geq r$) (J. Susan Milton and Isermann, 2002):

$$\begin{aligned}\mathbb{P}(C) &= 1 - \sum_{i=1}^{r-1} \left(\frac{(r-i)^n}{r^n} \binom{r}{r-i} (-1)^{i+1} \right) \\ &= 1 - \frac{1}{r^n} \sum_{i=1}^{r-1} \left((r-i)^n \frac{r!}{(r-i)!(r-(r-i))!} (-1)^{i+1} \right) \\ &= 1 + \frac{r!}{r^n} \sum_{i=1}^{r-1} \left(\frac{(r-i)^n}{(r-i)!i!} (-1)^i \right)\end{aligned} \tag{4.2}$$

Os gráficos da Figura 43 evidenciam a vantagem ao nível da robustez a faltas permanentes que a realocação traz ao circuito. A Figura 43a compara a probabilidade $\mathbb{P}(B)$ e $\mathbb{P}(C)$ para uma FPGA dividida em quatro regiões iguais ($r = 4$), e a Figura 43b compara as mesmas probabilidades para uma FPGA dividida em seis regiões iguais ($r = 6$).



(a) Quatro regiões ($r = 4$). (b) Seis regiões ($r = 6$).

Figura 43 – Probabilidades $\mathbb{P}(B)$ e $\mathbb{P}(C)$ em função de n.

É visível que em ambos os cenários quando $\mathbb{P}(B) \approx 0,5$ o valor de $\mathbb{P}(C)$ é nulo. Ou seja, quando sem possibilidade de realocação o

número de faltas permanentes existentes na FPGA são suficientes para uma probabilidade de 50% do circuito ter sido danificado ao ponto de passar a falhar, com realocação, as mesmas faltas permanentes não conseguem ameaçar o normal funcionamento do circuito.

Como foi assumido que a média é uma falta permanente por período de tempo T, significa que em média, um circuito com realocação resiste $(r-1)T$ sem qualquer perturbação no seu funcionamento, o que evidencia que em quantas mais regiões (r) for dividido o dispositivo, mais este se torna tolerante a faltas permanentes.

Considerando agora que o ciclo de vida previsto para o circuito é igual ao período de tempo T e que nesse intervalo de tempo, em média ocorre apenas uma falta permanente, então, com estes pressupostos, a probabilidade da ocorrência de faltas permanentes segue a distribuição de Poisson dada pela seguinte equação (J. Susan Milton and Isermann, 2002):

$$\mathbb{P}(X = n) = \frac{\lambda^n e^{-\lambda}}{n!}, \tag{4.3}$$

onde e é número de Euler ($e \approx 2,718$), λ é a taxa média de ocorrência de faltas permanentes por período de tempo T e n o número de faltas permanentes ocorridas no período de tempo T. Como foi assumido que no período de tempo T em média ocorre uma falta permanente, então a Equação 4.3 pode ser simplificada e ficar da seguinte forma:

$$\mathbb{P}(X = n) = \frac{1^n e^{-1}}{n!} = \frac{1}{en!} \tag{4.4}$$

Para saber qual a probabilidade de ocorrer pelo menos um determinado número n de faltas permanentes num período de tempo T o cálculo é feito da seguinte forma (J. Susan Milton and Isermann, 2002):

$$\mathbb{P}(X \geq n) = 1 - \sum_{i=0}^{n-1} \mathbb{P}(X = i) = 1 - \sum_{i=0}^{n-1} \frac{1}{ei!} \tag{4.5}$$

Com recurso à Equação 4.5 são calculados os valores para os primeiros sete casos de $\mathbb{P}(X \geq n)$ que preenchem a Tabela 4. A estes

mesmos valores corresponde o gráfico da Figura 44.

Tabela 4 – Tabela da probabilidade $\mathbb{P}(X \geq n)$.

$\mathbb{P}(X \geq 1)$	$\mathbb{P}(X \geq 2)$	$\mathbb{P}(X \geq 3)$	$\mathbb{P}(X \geq 4)$	$\mathbb{P}(X \geq 5)$	$\mathbb{P}(X \geq 6)$	$\mathbb{P}(X \geq 7)$
0,632120	0,264241	0,080301	0,018988	0,003659	0,000594	0,000083

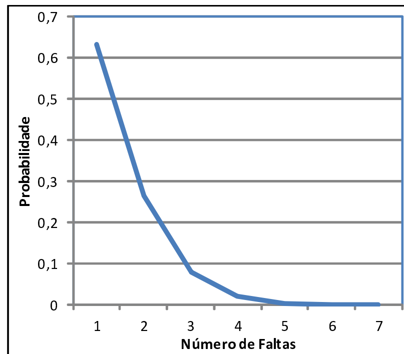


Figura 44 – Gráfico da probabilidade $\mathbb{P}(X \geq n)$.

Na Figura 44 é visível que a probabilidade de ocorrerem quatro ou mais faltas permanentes num período de tempo T é extremamente reduzida, e a probabilidade de ocorrerem seis ou mais faltas é quase nula.

Comparando a Figura 44 com a Figura 43a, observa-se que no caso em que a FPGA é dividida em quatro regiões iguais ($r = 4$), sem possibilidade de realocação, a probabilidade de ocorrer uma falta permanente que danifique o circuito é $\mathbb{P}(B) = \frac{\mathbb{P}(X=1)}{4} = \frac{1}{4e} = 0,09197$. No mesmo cenário, existindo a possibilidade de realocação, é necessário que no mínimo ocorram quatro faltas permanentes para que exista alguma probabilidade diferente de zero, de perturbar o normal funcionamento do circuito. Como $\mathbb{P}(X \geq 4) = 0,018988$, mesmo que ocorram quatro faltas ($n = 4$) durante o período T, como é visível na Figura 43a a probabilidade ronda $0,1$. Ou seja, $\mathbb{P}(C) \approx \frac{0,018988}{10} \approx 0,001898$, o que significa que a probabilidade de ocorrerem quatro faltas durante o ciclo de vida T que causem alguma falha no sistema é inferior a 0,2%. Se

a opção for dividir o dispositivo em seis partes iguais ($r = 6$), então a probabilidade de que a ocorrência de um conjunto de faltas permanentes seja suficiente para perturbar o normal funcionamento do circuito é ainda mais reduzido ($< 20 * 10^{-6}$), contra a opção sem realocação, onde apenas com uma falta $\mathbb{P}(B) = \frac{\mathbb{P}(X=1)}{6} = \frac{1}{6e} = 0,06131$.

Resumindo, dividindo a FPGA em várias partes iguais e dotando o sistema com a capacidade de realocação, é possível reduzir drasticamente a probabilidade de que a ocorrência de faltas permanentes possa colocar em risco o normal desempenho do circuito implementado (apenas numa das partes).

4.1.2 Utilizando $(r-1)/r$ dos recursos: Maximização da utilização da FPGA

Na Secção 4.1.1 foi demonstrado como a realocação permite obter uma drástica redução da probabilidade de que a ocorrência de faltas permanentes possam colocar em risco o circuito. No entanto mostra igualmente um enorme desperdício de recursos da FPGA, pois a obtenção de menor probabilidade (implica aumentar o número de regiões r), é inversamente proporcional à quantidade de recursos úteis da FPGA $\frac{1}{r}$. Ou seja, quanto maior a tolerância a faltas permanentes pretendida, maior é o desperdício de recursos do dispositivo.

Para resolver esta enorme desvantagem, em vez de existir um circuito a ocupar $\frac{1}{r}$ do dispositivo, passam a coabitar r circuitos na FPGA, cada um numa região e a ocupar $\frac{r-1}{r}$ dos recursos dessa região. A Figura 45 mostra o exemplo para $r = 4$.

Nesta nova distribuição, cada fração homogénea de $\frac{1}{4}$ dos recursos da FPGA (A, B, C e D) é sub-dividida noutras quatro partes iguais (identificados por 1, 2, 3 e 4). Dessas quatro partes, apenas três são usadas para implementar um determinado módulo (ou módulos), ficando a restante livre. A distribuição da parte livre de cada uma das frações é feita de modo a que o mesmo número de identificação não se repita, tal como mostra a Figura 45.

Desta forma, se por exemplo, ocorrer uma falta permanente

Figura 45 – Exemplo de quatro circuitos implementados em 3/4 da FPGA.

na zona A_2, a solução será através de realocação, trocar a parte do sistema implementado na parte A com a que se encontra implementada na parte B, pois esta não usa os recursos localizados em B_2. Com esta abordagem, podemos garantir que a probabilidade de uma falta permanente perturbar o normal funcionamento do sistema é zero. Não existindo a capacidade de realocação, numa situação onde $\frac{3}{4}$ dos recursos da FPGA fossem utilizados, esta mesma probabilidade seria $0,75$, o que é um valor muito elevado.

Generalizando, no novo cenário, uma FPGA é dividida em r regiões (de tamanho exatamente igual), caso ocorram duas faltas permanentes ao fim de um período de tempo T, e apenas $\frac{1}{r}$ dos recursos não são utilizados. Na situação em que não exista a opção de realocação, a probabilidade de pelo menos uma das duas faltas permanentes terem danificado a região onde se encontra implementado o circuito, definindo $D =${Ao fim de um período de tempo T a região onde está implementado o circuito sofreu pelo menos uma das duas faltas permanentes que ocorreram na FPGA}, é calculada pela Equação 4.6 (J. Susan Milton and Isermann, 2002).

$$\mathbb{P}(D) = 1 - \frac{1}{r^2} \tag{4.6}$$

Existindo a opção de realocação, para que o sistema seja perturbado no caso de ocorrerem duas faltas permanentes (o valor mínimo de faltas permanentes necessário para que o sistema possa eventualmente falhar), é preciso que elas ocorram na mesma fração e em duas partes

dessa fração distintas (Ex: A_1 e A_2), ou que ocorram em duas frações diferentes mas em duas partes com o mesmo índice (Ex: A_1 e B_1). Portanto, definindo $E =${Uma mesma fração sofreu duas faltas permanentes em duas partes diferentes, ou duas frações diferentes sofreram uma falta permanente na parte com o mesmo índice ao fim de um período de tempo T}, o cálculo de $\mathbb{P}(E)$ é dado pela Equação 4.7 (J. Susan Milton and Isermann, 2002).

$$\begin{aligned}\mathbb{P}(E) &= \frac{2\left(r^2 - r\right) r}{\left(r^2\right)^2} \\ &= \frac{2\left(r-1\right) r^2}{\left(r^2\right)^2} \\ &= \frac{2\left(r-1\right)}{r^2}\end{aligned} \tag{4.7}$$

O gráfico da Figura 46 esboça os resultados obtidos pela Equação 4.6 e pela Equação 4.7. É assim possível comparar a probabilidade de duas faltas permanentes poderem danificar o sistema implementado numa FPGA dividida em r frações (com ou sem realocação). Foram calculadas as probabilidades para r a variar entre dois e dez.

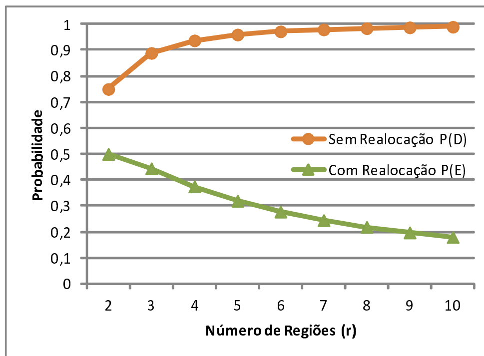


Figura 46 – Probabilidades $\mathbb{P}(D)$ e $\mathbb{P}(E)$ para diferentes números de regiões (r).

Para além da importante vantagem de que com realocação, o sistema fica imune à primeira falta, o gráfico da Figura 46 evidencia

duas outras importantes vantagens:

- Para qualquer valor de r, a probabilidade $\mathbb{P}(D)$ é sempre muito superior à probabilidade $\mathbb{P}(E)$;
- Enquanto probabilidade $\mathbb{P}(D)$ aumenta com o incremento do valor de r, tendendo para um, a probabilidade $\mathbb{P}(E)$ diminui e tende para zero.

Esta última vantagem referida é de grande importância, pois significa que podemos ter uma elevada taxa de ocupação de recursos da FPGA, e ao mesmo tempo dotar o sistema de uma maior tolerância a faltas permanentes.

4.2 ANÁLISE DA CONFIABILIDADE NUM SISTEMA EM FPGAS

Quando se pensa em aplicações espaciais, a confiabilidade (*Reliability*) ganha uma importância redobrada. E se a ameaça são as faltas permanentes, que podem levar o sistema a falhar sem hipótese de ser reparado, o valor (*Mean Time to Failure* - MTTF) (Koren and Krishina, 2007) ganha um relevo ainda maior. Se for assumido que existem vários sistemas iguais implementados em várias FPGAs e que o tempo até cada um dos sistemas falhar segue uma distribuição com uma função de densidade probabilística de $f(t)$, então o $MTTF$ desse grupo de sistemas pode ser calculado matematicamente tal como formalizado na equação 4.8:

$$MTTF = \int_0^\infty tf(t)dt = \int_0^\infty R(t)dt \tag{4.8}$$

onde $R(t)$ corresponde à função de *Reliability*.

Para simplificação, considere-se agora um novo cenário em que um sistema ocupa 100% dos recursos da FPGA onde se encontra implementado e que estatisticamente, num ambiente hostil como no espaço, uma FPGA sofre uma falta permanente a cada T anos. Como todos os recursos se encontram em uso, assume-se ainda que a ocorrência

de uma falta permanente implica o sistema falhar sem possibilidade de reparação. Nestas condições, a taxas de falhas (*Failure Rate*) $\lambda = \frac{1}{T}$.

Num cenário como o da Figura 45 ($r = 4$) em que o sistema ocupa $\frac{3}{4}$ dos recursos do dispositivo, não existindo realocação, a taxas de falhas reduz-se para $\lambda = \frac{3}{4.T}$, pois a falta permanente pode ocorrer no $\frac{1}{4}$ dos recursos não utilizados. Generalizando para o mesmo género de distribuição de recursos, mas com r regiões, o resultado é a equação 4.9:

$$\lambda = \frac{r-1}{r.T} \tag{4.9}$$

Como $MTTF = \frac{1}{\lambda}$, então:

$$MTTF = \frac{r.T}{r-1} \tag{4.10}$$

Assumindo a existência da capacidade de realocação tal como descrito na Secção 4.1.2, o sistema consegue ultrapassar pelo menos a ocorrência da primeira falta permanente. Tal significa que se em média ocorre apenas uma falta permanente nos primeiros T anos, considerando que a probabilidade da ocorrência deste tipo de faltas segue a distribuição de Poisson, então recorrendo à Tabela 4, $\mathbb{P}(X \geq 2) = 0,264241$ corresponde à probabilidade de pelo menos duas faltas permanentes ocorrerem por toda a FPGA durante os primeiros T anos. Comparando com o caso em que não existe realocação, onde uma falta permanente é suficiente para poder provocar a falha do sistema, sendo $\mathbb{P}(X \geq 1) = 0,632120$, então chega-se à conclusão que nos primeiros T anos, ignorando a vantagem que com realocação a probabilidade de duas faltas permanentes levarem o sistema a falhar é inferior, sem realocação o sistema corre quase mais $2,4\times$ risco de falhar.

Considerando agora o cenário com um período de tempo $2T$ anos, para que em média ocorram duas faltas permanentes, e recorrendo às probabilidades da equação 4.6 e da equação 4.7, é possível chegar às aproximações das taxas de falhas λ_1 (sem realocação) e λ_2 (com realocação) presentes nas seguintes equações:

$$\lambda_1 = \frac{\mathbb{P}(D)}{2.T} = \frac{r^2 - 1}{2.T.r^2} \tag{4.11a}$$

$$\lambda_2 = \frac{\mathbb{P}(E)}{2.T} = \frac{2(r - 1)}{2.T.r^2} \tag{4.11b}$$

Calculando os correspondentes MTTF através do λ_1 e λ_2, o resultado são as equações 4.12a e 4.12b respetivamente.

$$MTTF_1 = \frac{2.T.r^2}{r^2 - 1} \tag{4.12a}$$

$$MTTF_2 = \frac{2.T.r^2}{2(r - 1)} \tag{4.12b}$$

Tal como já indicava a Figura 46, com realocação, ao garantir que o sistema pode suportar pelo menos uma falta permanente, caso o número de regiões $r > 2$, então o MTTF será pelo menos o dobro do que sem realocação. Usando como exemplo o caso concreto da Figura 45, com $r = 4$, os resultados seriam os seguintes:

$$MTTF_1 = \frac{2.T.4^2}{4^2 - 1} = \frac{32.T}{15} \tag{4.13a}$$

$$MTTF_2 = \frac{2.T.4^2}{2(4 - 1)} = \frac{32.T}{6} \tag{4.13b}$$

Ou seja, para $r = 4$, o $MTTF_2$ (com realocação) é $2,5\times$ superior ao $MTTF_1$ (sem realocação). Um aumento de 150% no tempo médio de vida do sistema implementado e que aumenta ainda mais se r for superior. Este importante ganho é exatamente o que todo o trabalho desta tese procura alcançar.

4.3 COMBATE AO ENVELHECIMENTO (*AGING*) CAUSADO PELO NBTI

Na Secção 3.3.1 foram descritas as ameaças tanto do NBTI como do PBTI. Embora a importância do PBTI tenha vindo a equiparar-se com a do NBTI, este último tem tido um maior foco. A variação da

tensão de *threshold*, ΔV_{th} nos transístores pMOS provoca igualmente uma variação no tempo de resposta em cada transístor. Isto significa que ao longo do tempo, os recursos que compõem uma FPGA poderão exigir a redução da frequência de funcionamento para que não ocorram faltas permanentes (*gate delay* ou *path delay*). Alguns trabalhos estimam mesmo uma variação que pode atingir 5-15% por ano na tensão *threshold* (V_{th}) (Mishra et al., 2012) (Ceratti et al., 2012).

No entanto trabalhos como (Palermo et al., 2015), demonstram que é possível combater parcialmente o envelhecimento devido ao NBTI gerindo a utilização dos transístores que compõem o circuito. A Figura 47, baseada no modelo (R-D) (Alam and Mahapatra, 2007), mostra que a variação no funcionamento de um transístor (em condução ou ao corte) pode minimizar o efeito do NBTI.

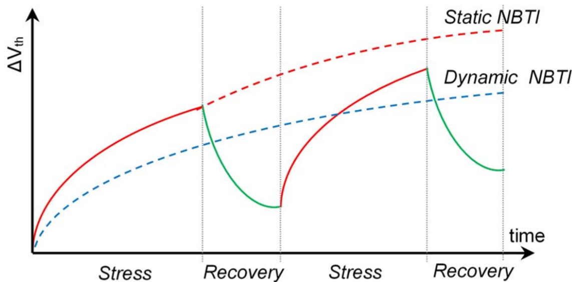


Figura 47 – Ilustração das fases de stress e recuperação (*recovery*) do NBTI (Palermo et al., 2015).

É visível na Figura 47 que se um transístor estiver sempre ativo, o ΔV_{th} é superior (*Static* NBTI) em relação à situação em que o estado do transístor alterna entre ativo (Stress) e não ativo (*Recovery*).

Consultando novamente o exemplo da Figura 45 na Secção 4.1.2, observa-se que para além da recuperação de faltas permanentes, a troca de localização dos vários circuitos implementados pode igualmente variar a utilização dos recursos da FPGA ao longo do tempo, o que indiretamente poderá reduzir o envelhecimento devido ao NBTI. Como a implementação dos circuitos é realizada através do preenchimento da memória SRAM de configuração do dispositivo, esta irá sofrer igualmente

alterações, ficando por isso também com as suas células a variar de valor lógico.

4.4 ESTRUTURA DA SOLUÇÃO APRESENTADA

Sendo o grande objetivo desta tese dotar um sistema implementado numa FPGA de um mecanismo que lhe permita autonomamente recuperar de faltas permanentes, é importante identificar as principais partes que permitem construir as estratégias desejadas. A Figura 48 ilustra genericamente a estrutura do sistema desenvolvido para uma FPGA de maneira a que este permita a implementação do mecanismo investigado.

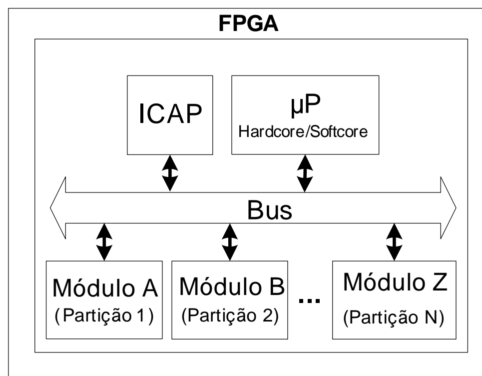


Figura 48 – Estrutura geral da implementação da solução desenvolvida uma FPGA.

O sistema inclui um microprocessador, que pode existir fisicamente no dispositivo (*Hardcore*) ou ser implementado na lógica configurável (*Softcore*), e que para além de desempenhar as funções inerentes à(s) aplicação(ões) embarcada(s), acumula ainda o controlo do mecanismo proposto. O microprocessador quando não se encontra a controlar o novo mecanismo, desempenha as suas "naturais" funções comunicando com os módulos alocados nas N partições parciais através de um *Bus* que o interliga aos vários módulos. Quando o microprocessador cuida dos procedimentos referentes ao controlo do mecanismo de

controlo e recuperação de falhas do sistema, este pode comunicar com os módulos por intermédio do *Bus* ou realizar operações de realocação através do módulo *Internal Configuration Access Port* (ICAP).

Relativamente à arquitetura do mecanismo desenvolvido, modularmente estão identificados e delimitados três módulos, os quais são apresentados no diagrama da Figura 49. São eles: Módulo de Controlo, Mecanismo de Deteção e Módulo de Atuação.

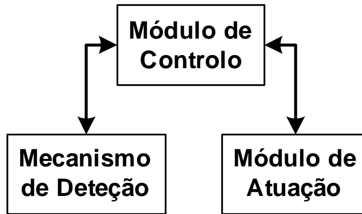


Figura 49 – Diagrama geral da solução desenvolvida.

4.4.1 Módulo de Controlo

Este módulo é hierarquicamente responsável pelo controlo do Mecanismo de Deteção e do Módulo de Atuação. É por isso o nível de topo desta proposta e é essencialmente implementado em software/-firmware. Este Módulo de Controlo é integrado no software do sistema implementado no processador embarcado na FPGA e é ajustado em função da estrutura do próprio sistema e das estratégias adotadas (Ex: número de módulos em hardware, número de partições reconfiguráveis, com ou sem rotação de módulos entre as partições, etc.).

Embora dependa do restante sistema para decidir as suas ações (por exemplo, não ativar o mecanismo de deteção de faltas num módulo, quando este se encontre a realizar uma função que não pode ser interrompida), este módulo é autónomo na execução dos seus procedimentos. Inclui uma tabela de dados onde gere a informação relativa às faltas permanentes já detetadas, a sua localização e em que *bistream* a falta originou uma falha. Desta forma o sistema possui memória do passado

e não volta a realocar o *bitstream* de um módulo numa partição onde já falhou devido a uma falta permanente.

Ainda que pensado para ser fisicamente implementado na própria FPGA através de um processador embebido, pode ser alocado num processador no exterior (existente numa FPGA adjacente ou outro processador físico) e comunicar com o dispositivo através do porto *Joint Test Action Group* (JTAG), ou mapeando uma primitiva ICAP nos pinos da FPGA.

A Figura 50 apresenta o fluxograma geral de toda a solução proposta para aumentar a tolerância a faltas permanentes num sistema implementado numa FPGA.

Inicialmente o módulo responsável pela gestão entra em ciclo a verificar se ocorreu qualquer falta que tenha provocado alguma falha num determinado módulo, nos casos em que o sistema possui mecanismos como TMR (Montminy et al., 2007) (Bolchini et al., 2012) (Espinosa et al., 2012), ou se está no momento de realizar algum teste em algum módulo. Caso se cumpra alguma destas situações, aguarda até que estejam reunidas as devidas condições no sistema para proceder ao teste. Caso nenhuma destas situações ocorra, é verificado se o nível medido de *aging* num determinado módulo, ou um agendamento previamente estabelecido exigem uma troca de módulos de modo a permitir uma rotação no uso dos recursos da FPGA.

A etapa do teste é realizada pelo Mecanismo de Deteção. No final, passado o teste com sucesso, volta a entrar no ciclo inicial, exceto no caso em que o sistema tenha detetado uma falha e essa falha tenha ocorrido recentemente de um modo regular. Neste último cenário, poderá existir uma falta que não seja detetável pelo teste existente, que poderá ser por incapacidade de cobertura ou *path delay* devido ao envelhecimento.

Quando no final do teste a indicação é que existe pelo menos uma falta na zona de recursos onde está localizado o módulo, poderão ser tomadas duas decisões diferentes. Caso seja a primeira vez, como poderá tratar-se de um *bit flip* na correspondente memória de configuração, o mesmo módulo é realocado na mesma posição. Sendo a segunda vez

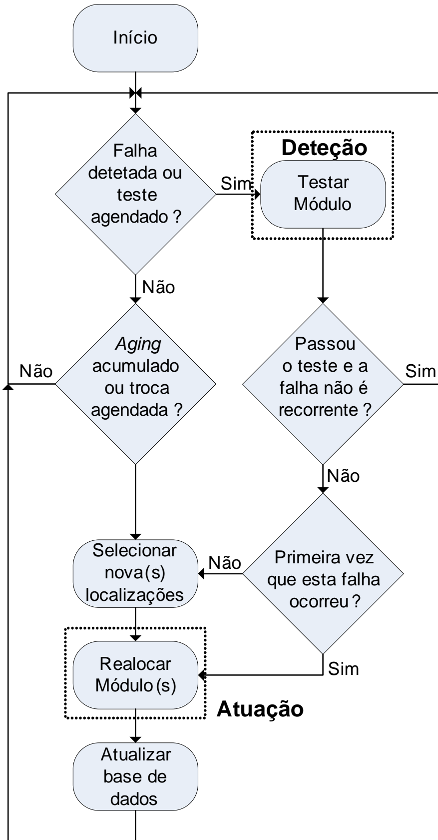


Figura 50 – Fluxograma geral da aplicação desenvolvida.

consecutiva (duas falhas seguidas na mesma posição), significará que ocorreu uma falta permanente, e será então selecionada uma outra RP para alocar o módulo testado. Esta seleção tanto poderá ser uma partição que se encontre livre, ou a troca com outro módulo que esteja numa outra partição e que nunca foi testado na partição onde foi detetada a falta permanente. Igualmente, a partição escolhida não poderá estar registada na tabela de dados como imprópria para alocar o módulo testado, ou seja, no passado, nunca lhe foram detetadas faltas permanentes quando teve alocado o módulo que irá receber.

Após a devida seleção da partição (ou partições no caso da

realização de uma troca de partição ou rotação de recursos), o módulo é então realocado nessa nova área de recursos. Caso essa mesma área já tenha um outro módulo alocado, esse módulo poderá ser simplesmente trocado diretamente com o que foi testado, desde que, como anteriormente referido, na tabela de dados não exista indicação em contrário. Quando uma troca direta ocorre, deverá ser agendado um teste urgente ao módulo que foi agora realocado na zona onde foi detetada pelo menos uma falta permanente. Isto porque, poderá ocorrer que a(s) falta(s) detetada(s) anteriormente possam perturbar igualmente este outro módulo. O processo de realocar um módulo é da competência do Módulo de Atuação.

No final, é atualizada a tabela de dados onde se encontra registado, em que RPs da FPGA um determinado módulo não pode ser alocado, devido à existência de faltas permanentes nessas RPs. Finalizada a atualização, volta ao ciclo inicial.

4.4.2 Mecanismo de Deteção

Este mecanismo é responsável por detetar as faltas existentes em cada um dos módulos do sistema que se deseje testar. Embora tenha sido realizado algum trabalho com vista a este mecanismo (Martins et al., 2014a), devido à diversidade de opções, não ficou definido um mecanismo fixo deteção. A escolha deverá ocorrer e ser integrada manualmente durante a fase de projeto e em função do nível de confiabilidade pretendido para o sistema. Isto porque, o desenvolvedor poderá querer optar por uma arquitetura com TMR (Bolchini et al., 2012) (Espinosa et al., 2012), que para além de garantir o mascarar das faltas (aumentando a disponibilidade do sistema), permite ainda a deteção das mesmas. Ou desejar incluir um BIST (Dumitriu et al., 2015b) em cada módulo, ou mesmo realizar um teste baseado em software quando se trate de módulos baseados em microprocessadores. Nestas duas alternativas, o sistema terá de ficar *offline* durante o teste, pelo que as aplicações do sistema que dependem destes módulos não poderão ser críticas ao ponto de exigir a disponibilidade dos módulos permanentemente. Para além

destas alternativas, o projetista poderá ainda incluir um mecanismo como o *Soft Error Mitigation* (SEM) (Xilinx Inc., 2011a) para detetar e corrigir os *Soft Errors* que ocorram na memória de configuração da FPGA.

Como poderão ser detetadas faltas permanentes ou transitórias, o fluxograma, como mostra a Figura 50, é capaz de realizar essa triagem e tem procedimentos para cada tipo de faltas.

As opções tomadas pelo desenvolvedor relativamente às formas de realizar o processo de deteção deverão sempre ter como foco as seguintes linhas de orientação:

- De modo a evitar ao máximo interferências no circuito de cada módulo a ser testado, a política será reduzir ao máximo o *overhead* de hardware adicionado ao módulo para que seja possível ser testado;

- Deverá ser possível testar cada módulo periodicamente, de modo a detetar a ocorrência das faltas permanentes ao longo do ciclo de vida da FPGA;

- O teste deverá ser individual e orientado ao módulo, de modo a que durante o mesmo, o restante sistema possa manter o seu normal funcionamento (exceto serviços que dependam do módulo sob teste);

- O teste deverá ser funcional, evitando assim que a deteção de uma falta transparente desencadeie uma realocação de um módulo, quando este não sofre qualquer perturbação devido a essa falta detetada;

- O mecanismo responsável pela deteção das faltas deve fornecer ao módulo que o controlará, uma flexibilidade que permita gerir uma política total de teste para cada módulo: agendamento, prioridade, período de teste, etc.

4.4.3 Módulo de Atuação

Quando uma falta é detetada, é necessário tomar medidas para que essa falta deixe de perturbar o normal funcionamento do sistema. De igual modo, quando um módulo se encontra alocado na mesma partição durante um longo período de tempo, isto poderá incrementar o envelhecimento dos recursos da partição, pelo que é igualmente útil rodar a localização dos módulos pelas RPs disponíveis. Esta parte é responsável pelos mecanismos necessários para executar essas medidas.

A tarefa base a cargo deste Módulo de Atuação é portanto a realocação de módulos do sistema, quando estes se encontrem numa zona de recursos onde foi detetada uma nova falta permanente, ou sempre que se pretenda rodar as posições dos módulos entre as partições compatíveis existentes. Como a implementação do hardware numa FPGA é realizada através da escrita da correspondente memória de configuração, então a tarefa de realocação de módulos do sistema passa por operações de manipulação dessa mesma memória de configuração.

Para que fosse possível realizar todos os processos associados a este Módulo de Atuação, foi necessário desenvolver o mecanismo *Partial Bitstream for Multiple Partitions* (PB4MP) (Martins et al., 2015a) e estratégias como as apresentadas no trabalho (Martins et al., 2015d), de modo a alcançar os seguintes objetivos:

- Efetuar as manipulações da memória de reconfiguração da FPGA (associadas ao processo de realocação), sem que seja necessário ter uma biblioteca com todas as combinações de *bitstreams*. Ou seja, implique ter apenas um *bitstream* para cada módulo, independentemente da RP onde se deseja realocar o módulo;

- Incrementar a tolerância a faltas permanentes no sistema implementado na própria FPGA sem que isso implique desperdício relevante de recursos (não obrigue excluir uma partição só porque foi detetada nela uma falta permanente).

Para que fosse possível cumprir estes objetivos foram desenvolvidas ferramentas para gerar o mecanismo PB4MP (Martins et al.,

2015a), que orientam os processos associados à geração dos *bitstreams* do sistema (síntese e *place and route*). Com essas orientações foi possível implementar um nível de tolerância a faltas permanentes que siga uma das aproximações das que foram apresentadas na Secção 4.1.1 e na Secção 4.1.2. Significa isto, que tal como na parte da deteção, também esta implicará igualmente a criação de ferramentas para auxiliar o processo de desenvolvimento.

4.4.4 Identificação das Etapas do Fluxo FDIR

Na Secção 3 foram referidos dois estudos que relatavam as etapas adequadas que deveriam compor um fluxo FDIR. Desses estudos é possível retirar quatro etapas base. São elas: Deteção, Localização, Isolamento e Recuperação.

Embora nem sempre com uma unidade exclusiva para cada uma destas etapas (consultar o fluxograma da Figura 50), as mesmas são realizadas no trabalho desenvolvido da seguinte forma:

1. **Deteção**: Esta operação é realizada na parte que possui o mesmo nome no fluxograma. Como o sistema implementado na FPGA é dividido em módulos, o teste pode ser realizado a cada módulo independente com uma determinada regularidade. Quando um módulo não passa no teste é porque uma falta que está a gerar um erro foi detetada. O teste realizado, como referido na Secção 4.4.2, tanto pode ser baseado num BIST existente em cada módulo, ou recorrendo a uma arquitetura TMR;

2. **Localização**: Esta etapa, devido ao facto de o teste realizado na Deteção ser feito ao módulo, acaba por ser direta. Ou seja, se um módulo não passa no teste, automaticamente é conhecida a área de recursos da FPGA que contém pelo menos uma falta (permanente ou não) que está a causar um erro no módulo. Como existe a possibilidade de faltas serem ou não permanentes (*Hard Error* ou *Soft Error*), será necessário fazer a devida triagem. Essa triagem será feita adotando a abordagem sugerida pela aplicação

da Xilinx (Carmichael and Tseng, 2009), onde a primeira vez que a falha é detetada (Figura 50), o módulo em questão é realocado na mesma localização. Assim, a memória de configuração correspondente à zona de recursos utilizada pelo módulo é reprogramada, anulando assim a possibilidade de por exemplo um SEU ser considerado uma falta permanente. Esta despistagem é realizada na parte Deteção indicada no fluxograma;

3. **Isolamento**: O confinar dos recursos do dispositivo que possuam alguma falta permanente é realizado pelo Módulo de Controlo. Este módulo tem a seu cargo a gestão de uma base de dados onde consta que localizações de recursos se encontram disponíveis para alocar cada um dos módulos do sistema. Quando numa localização é detetada uma falta permanente que provoque erros num determinado módulo, esta localização é devidamente assinalada na base de dados, para que o módulo nunca mais volte a ser lá realocado;

4. **Recuperação**: Gerida pelo Módulo de Controlo, é o Módulo de Atuação que realiza a recuperação do normal funcionamento do módulo onde foi detetada pelo menos uma falta permanente. A Atuação, tal como descrita na Secção 4.4.3, consistirá numa operação de mover o conteúdo da memória de configuração correspondente à área de recursos onde se encontra alocado o módulo, para a memória de configuração correspondente à nova área de recursos onde o módulo será realocado.

Após esta enumeração e explicação de cada etapa, embora não de uma forma rígida, as mesmas etapas estabelecidas nos estudos até aqui realizados em FDIR (Guiotto et al., 2003) (Holsti and Paakko, 2001) são igualmente usadas como forma de orientação deste trabalho.

4.5 CONCLUSÃO

Numa perspetiva macro, o objetivo original desta tese foi implementar o mecanismo esquematizado no diagrama da Figura 49, que é

incluída num sistema global desenvolvido para FPGAs, de modo a dotar esse sistema da capacidade de recuperar de faltas permanentes ou transitórias. Para que este objetivo fosse possível foi necessário desenvolver um conjunto de ferramentas em software, que permitem a implementação do fluxo da Figura 50. Este conjunto de ferramentas implica sempre uma intervenção inicial do projetista e algum trabalho na fase final, aquando da integração do software responsável pelo controlo na parte do fluxo geral do sistema.

Entre as principais vantagens deste trabalho, salienta-se o facto de que o desenvolvedor não necessitará de conhecer os detalhes técnicos da FPGA para conseguir implementar um mecanismo que as ferramentas atuais não permitem. Isto acontece uma vez que consegue adicionar de uma forma automática, após dimensionar e realizar a devida divisão do sistema em módulos, a capacidade de o sistema desenvolvido detetar e recuperar de faltas permanentes.

Durante todo o processo de desenvolvimento das ferramentas necessárias para garantir a implementação das estratégias que compõem este trabalho, foi priorizada a compatibilidade entre famílias de FPGAs da Xilinx. Esta opção deveu-se à familiarização do autor com as FPGAs deste fabricante. Como muito do trabalho foi desenvolvido com base em detalhes técnicos característicos destas famílias, a sua aplicação em FPGAs SRAM de outro fabricante, implicaria igualmente estudar em pormenor o funcionamento desses dispositivos.

No próximo capítulo serão descritos os vários trabalhos desenvolvidos no sentido de implementar os vários módulos que compõem a arquitetura da Figura 49, e aproveitar a análise probabilística na Secção 4.1.2 para uma maior tolerância a faltas permanentes.

5 TRABALHOS E METODOLOGIAS DESENVOLVIDAS

Após o planeamento da estrutura que foi exposto na Secção 4.4, foram desenvolvidos vários trabalhos na direção de implementar toda essa estrutura. Neste capítulo serão descritos numa ordem cronológica todos os trabalhos realizados. Cada um dos trabalhos constituiu uma fase do trabalho total e a sua análise acabou por interferir ou ajustar a(s) fase(s) seguinte(s). A partir de um determinado ponto, a observação do trabalho desenvolvido começou a salientar novas ideias (novas etapas), o que se repetiu até esgotar o tempo disponível para concluir esta tese.

5.1 DETETOR DE FALTAS EM MÓDULOS IMPLEMENTADOS EM FPGAS

O Mecanismo de Deteção caracterizado na Secção 4.4.2, embora seja obrigatório, é sempre uma opção do projetista, em função do nível de tolerância a faltas definido. No início do ciclo de trabalho para esta tese e após a conclusão do estudo incluído no Apêndice A, foi tirado proveito da informação técnica recolhida e foi desenvolvido um novo modo de realizar um teste a um módulo implementado numa FPGA (Martins et al., 2014a).

O módulo de teste construído, baseado no tradicional BIST, é orientado para sistemas em FPGA que já incluam um processador *hardcore* ou *softcore*. Este teste permite verificar a integridade da funcionalidade de cada módulo existente no dispositivo, sem necessidade de hardware adicional relevante, pois não existe necessidade de construção de um *scan chain*. O mecanismo, embora tenha sido desenvolvido com recurso da Virtex-5 XC5VLX50T da Xilinx, pode ser igualmente usado em famílias mais recentes da Xilinx que suportem PR. Os resultados mostram que apenas com um ligeiro incremento na necessidade de memória por parte do processador, é possível verificar se alguma das faltas que venham a ocorrer no dispositivo alteraram o normal funcionamento de algum dos módulos implementados internamente. Este incremento de memória exigida pode tornar-se absolutamente desprezável, quando

o próprio sistema já exige para si uma quantidade numa ordem de grandeza superior.

5.1.1 Arquitetura e Organização da FPGA Virtex-5

A implementação deste BIST com *scan chain* virtual é um processo que requer conhecimentos com bastante detalhe do dispositivo usado. Por essa razão é importante rever os detalhes técnicos importantes da família de FPGAs Virtex-5 (Xilinx Inc., 2012b) (família da FPGA utilizada para implementação deste trabalho), que são fundamentais para implementar este BIST.

Tal como já descrito no Capítulo 2, a arquitetura da família de FPGAs Virtex-5 é em muito semelhante às mais recentes Virtex-6 e 7-series. Tem por isso igualmente a capacidade de suportar a reconfiguração parcial de uma região bidimensional de recursos, o que permite tanto forçar a localização de um módulo, como alterar as suas propriedades (configuração) dinamicamente.

A Virtex-5 é uma FPGA *tile-based* e por isso os recursos estão organizados numa matriz, onde existem linhas (*rows*) e colunas (*columns*). Cada linha é composta por uma secção de cada coluna de recursos, em que no caso dos CLBs são 20. Para configurar cada secção de 20 CLBs são necessários 36 *frames* (numerados de 0 a 35), constituídos por 41 palavras de 32 bits cada um deles. Os 36 *frames* são divididos em diferentes tipos de configuração, sendo eles roteamento (*frames* 0 a 25), conteúdo das LUTs dos 20 CLBs (*frames* 26 a 29 para os *slices* com X ímpar e *frames* 32 a 35 para os *slices* com X par), e bits responsáveis pela seleção de multiplexers e determinadas propriedades de cada um dos *slices* (*frames* 30 e 31). Nestes dois *frames* é onde se encontram os campos INIT responsáveis pelo valor inicial de cada flip-flop, fundamentais para a implementação deste BIST baseado num *scan chain* virtual.

5.1.2 Gerenciamento da Configuração da FPGA

Embora não exista uma reconfiguração parcial (*Partial Reconfiguration*) total, o objetivo de um *scan chain* virtual implicou dominar as ferramentas e especificações associadas a esta capacidade da FPGA, recorrendo ao interface ICAP. Os detalhes técnicos de maior importância foram:

1. o facto de que as ferramentas adicionam uma primitiva *LUT1* (*Proxy Logic*) a cada entrada e saída do módulo implementado na correspondente partição reconfigurável, algo que permite ter controlabilidade sobre as entradas do módulo;

2. o uso dos comandos GRESTORE (que provoca um *Global Set-Reset* (GSR)) e GCAPTURE para controlabilidade e observabilidade dos registos pertencentes a um determinado módulo. O GRESTORE permite carregar os flip-flops com os valores lógicos presentes nos respetivos campos INIT, enquanto que o GCAPTURE permite o inverso, ou seja, transferir os valores lógicos presentes nos flip-flops para os campos INIT correspondentes;

3. o desabilitar de todos os registos da influência do sinal GSR, com exceção dos registos pertencentes ao módulo sob teste (muito importante para que seja possível testar a funcionalidade de um módulo, sem perturbar o normal funcionamento dos restantes serviços do sistema).

5.1.3 Detetor de faltas *Low Cost*

Era importante que este detetor de faltas fosse o menos intrusivo possível, pudesse ser aplicado a cada módulo isoladamente e evitasse a adição de hardware extra no módulo. Estes objetivos foram concretizados implementando um BIST por intermédio de um *scan chain* virtual (sem necessidade de existir fisicamente no hardware), recorrendo para isso, às capacidades da reconfiguração parcial dinâmica de uma FPGA.

5.1.4 Arquitetura BIST

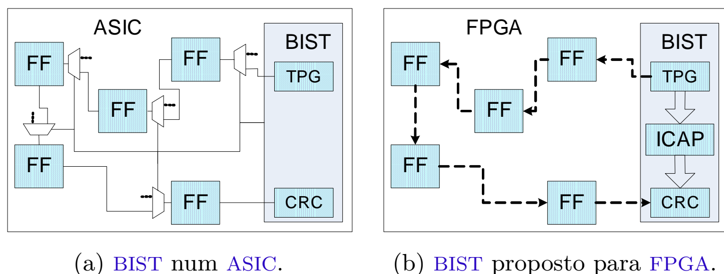
A arquitetura BIST quando usada num ASIC, como ilustrado na Figura 51a, é composta por um *Test Pattern Generator* (TPG), um *scan chain*, um CRC e uma pequena máquina de estados. O TPG é muitas vezes construído por intermédio de um *Linear Feedback Shift Register* (LFSR) e é responsável pela geração de vetores de teste. Estes vetores são inseridos na entrada do *scan chain*, que é constituído por todos os registos (ou uma parte, caso existam vários *scan chain*), ligados em série por intermédio de um multiplexer de duas entradas, que é adicionado ao hardware exatamente para esse efeito. Por outro lado, na saída do *scan chain*, irão sair os resultados do vetor de teste anterior e os mesmos irão ser comprimidos através do CRC. A máquina de estados é responsável por controlar todo o processo. Ativa o *scan chain*, insere um vetor de teste, habilita o sinal de relógio durante um ciclo (ou um número predefinido de ciclos), retira o vetor resultante e comprime-o (ao mesmo tempo que é retirado o resultado, é inserido o próximo vetor de teste). No final, o valor presente no CRC, terá de ser igual a uma assinatura pré-calculada. Se tal não acontecer, é porque o circuito tem imperativamente pelo menos uma falta. Ultrapassado o processo de teste com sucesso, a máquina de estados desativa o *scan chain* e o circuito implementado passa a prover os serviços para o qual foi desenvolvido. Esta arquitetura, embora exija uma pequena incrementação de hardware e degradação do seu caminho crítico, é uma solução simples que traz uma grande vantagem a nível de deteção de faltas após o processo de fabrico e ao longo do ciclo de vida do circuito.

5.1.4.1 BIST com *Scan Chain* Virtual

Se no caso da implementação de BIST num ASIC, o acréscimo de hardware pode até ser aceitável, no caso de uma FPGA, salvo casos muito específicos, certamente não será. Para ultrapassar este *overhead*, a arquitetura foi adaptada para a FPGA. Essa adaptação, representada na Figura 51b, passa por conhecer que registos possui um sistema (ou subsistema), qual a sua localização e com essa informação, estabelecer

um *scan chain*. Com toda a informação necessária, tudo será realizado pelo microprocessador já existente no sistema embarcado, através do interface ICAP, também ele disponível no dispositivo.

Comparando com a arquitetura da Figura 51a, nesta nova abordagem, as funções do TPG, CRC e da máquina de estados, serão realizadas por software. Desta forma, o principal *overhead* será o acréscimo de processamento e de memória para as correspondentes rotinas. Relativamente ao acréscimo de hardware, este limitar-se-á à adição de uma primitiva *Global Clock MUX Buffer* (BUFGCTRL) por cada sinal de relógio usado pelo sistema sob teste, de um pequeno módulo *General Purpose Input Output* (GPIO) para o microprocessador poder controlar cada BUFGCTRL e uma primitiva *LUT1* para cada entrada e saída de uma RP (*Proxy Logic*), adicionadas pelas ferramentas de reconfiguração parcial.



(a) BIST num ASIC. (b) BIST proposto para FPGA.

Figura 51 – Comparação entre BIST “tradicional” e o proposto (Martins et al., 2014a).

5.1.4.2 Fluxo (*Flow*) de implementação do detetor de faltas

Embora tenha sido desenvolvido para que a sua aplicação fosse simples, o fluxo de implementação (Figura 52) deste detetor de faltas exige alguma intervenção humana. No entanto, é apenas uma intervenção no estado inicial do processo de implementação do hardware e outra no processo de desenvolvimento do software.

Tendo um sistema pronto a implementar numa FPGA, o primeiro passo é selecionar que módulos devem ser sujeitos a teste. Ou por

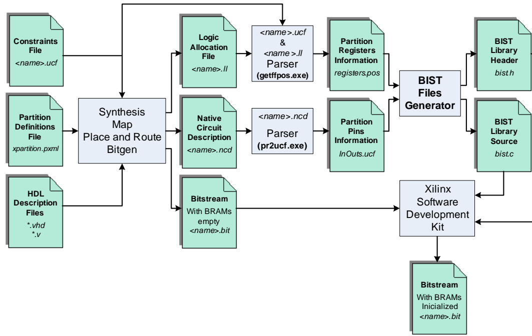


Figura 52 – Fluxo de implementação do detetor de faltas (Martins et al., 2014a).

outras palavras, ao nível da descrição do hardware (VHDL ou Verilog), quais as instâncias no nível de topo (*top level*) devem ser limitadas a uma determinada RP da FPGA. Estas decisões deverão ser devidamente registadas no **Constraints File** através das *constraints* `AREA_GROUP RANGE`, e no **Partition Definitions File**, tal como explicado na documentação da Xilinx (Xilinx Inc., 2013e). A seleção da área de cada partição reconfigurável deverá ter em atenção duas regras: selecionar apenas a quantidade de recursos necessária para cada módulo; selecionar sempre secções completas de colunas (*columns*) pertencentes a uma mesma linha (*row*), e pertencentes à mesma região de relógio (caso uma região de relógio não seja suficiente, deverão ser usadas duas adjacentes). Ainda manualmente, a cada linha de relógio que entra em cada módulo que será testado, será adicionada uma primitiva BUFGCTRL, que será controlada por um módulo GPIO previamente inserido no sistema para desempenhar esta função.

Após a intervenção anterior, a ferramenta procederá de uma forma autónoma a várias etapas (Síntese, Mapeamento, *Place and Route*, e Geração de *Bitstreams*), e gerará três ficheiros identificados na Figura 52 como: **Logic Allocation File**, **Native Circuit Description** e **Bitstream**. O **Logic Allocation File** juntamente com o **Constraints File** serão então interpretados por um *Parser* desenvolvido chamado *getffpos.exe*, e irão originar um outro ficheiro (**Partition Register**

Information) com uma pequena base de dados onde constam todos os registos (ou LUTs usadas como RAMs) utilizados por cada módulo. Com a utilização de outro *Parser*, o utilitário *pr2ucf.exe*, será extraído do ficheiro **Native Circuit Description** a posição das LUTs usadas como *Proxy Logic* e guardadas num outro ficheiro (**Partition Pins Information**). Para executar estes dois *Parser* ambos os comandos são chamados automaticamente da seguinte forma:

```
getffpos.exe -u <name>.ufc -l <name>.ll -o register.pos
pr2ucf.exe -i <name>.ncd -o InOuts.ucf
```

Com a informação contida no **Partition Register Information** (*register.pos*) e no **Partition Pins Information** (*InOuts.ucf*), uma nova ferramenta desenvolvida (**BIST Files Generator**) criará todas as rotinas necessárias para integrar o detetor de faltas proposto no original código a embarcar na FPGA. A chamada automática desta ferramenta é realizada do seguinte modo:

```
bist_files_generator.exe -p register.pos -u InOuts.ucf
```

Incluídas estas rotinas no restante software, a ferramenta **Xilinx Software Development Kit** encarregar-se-á de gerar o *Bitstream*, com os correspondentes blocos de memória devidamente preenchidos. Bastará então carregar o *Bitstream* final na FPGA para que o sistema fique pronto.

5.1.4.3 Sequência de execução do BIST

De uma forma completamente autónoma, o procedimento responsável pela verificação do(s) módulo(s) executa a seguinte sequência de sub-tarefas:

1. Desativar todas as entradas de relógio do(s) módulo(s) a ser(em) testado(s).

2. Guardar todo o contexto existente em flip-flops, LUTs usadas como RAM e BRAMs usadas pelo(s) módulo(s) a ser(em) testado(s).

3. Carregar as LUTs usadas como RAM e BRAMs com um conjunto de valores predefinido.

4. Calcular através de um LFSR implementado em software o(s) vetor(es) de teste necessários por intermédio do procedimento responsável pelo TPG.

5. Ler *frame* (o primeiro ou *frame* seguinte) da memória de configuração da FPGA (correspondente a conteúdo de flip-flops usados ou *Proxy Logic*).

6. Se os bits do *frame* corresponderem ao conteúdo de LUTs (*Proxy Logic*), alterá-los de modo a inserir nas *LUT1* usadas como entradas o(s) novo(s) vetor(es) de teste, e nas *LUT1* usadas como saídas, fixar o seu valor, para não perturbar o resto do sistema.

7. Se bits do *frame* corresponderem ao valor de flip-flops, ler os atuais valores, guardá-los no(s) vetor(es) de resultado(s) e alterá-los de modo a inserir neles o(s) novo(s) vetor(es) de teste.

8. Escrever *frame* alterado na memória de configuração da FPGA.

9. Voltar ao ponto 5) até ler/alterar todos os *frames* que contenham o valor de flip-flops usados ou *Proxy Logic*.

10. Ativar todas as entradas de relógio do(s) módulo(s) a ser(em) testado(s) durante um ciclo de relógio (ou outro número de ciclos predefinido).

11. Calcular o(s) CRC(s) com o(s) vetor(es) de resultado(s) formado(s) pelos valores do flip-flops retirados dos *frames* no ponto 7).

12. Voltar ao ponto 4) caso ainda não tenha sido atingido o número de vetores de teste desejado.

13. Restaurar todo o contexto salvaguardado em 2), tal como voltar a restaurar a função de todos os *Proxy Logic* (entradas e saídas).

14. Ativar todas as entradas de relógio do(s) módulo(s) testado(s).

No final desta sequência, caso o valor final do CRC calculado seja igual à assinatura calculada previamente *offline*, significa que o módulo passou no seu teste de verificação funcional. Caso contrário, o sistema ficará com conhecimento que aquele módulo apresenta pelo menos uma falta quando implementado na área atual de recursos da FPGA, e em função disso decidirá que providências deverá tomar (de referir que as faltas existentes poderão ser do tipo SEU, e nesse caso a solução passará por corrigir a memória de configuração da FPGA e não uma operação de realocação do módulo).

Esta sequência apresentada é a versão mais simples de entender o algoritmo e é a que foi usada na implementação usada para a sua validação na Secção 6.1. É possível proceder a alterações, de modo a obter consideráveis ganhos a nível de eficiência (ex. usar o tempo de espera pelos resultados da leitura de *frames* para calcular o próximo vetor ou preparar o próximo *frame*). Relativamente ao tempo necessário para executar toda a sequência, o seu cálculo é detalhado na próxima secção.

5.1.4.4 Memória usada pelo BIST e tempo de execução

A memória usada pelo BIST e o tempo de execução em software dependem de vários detalhes do algoritmo, alguns deles correlacionados com os valores fixos dos parâmetros do software e outros dependentes dos parâmetros de hardware dos módulos sob teste.

De modo a avaliar o consumo de memória e o tempo de execução, um conjunto de equações foi definido com base na Tabela 5 e na Tabela 6. A Tabela 5 descreve todas as funções em software e indica quanta memória é utilizada (compilado com a *flag* de otimização -O3) e os requisitos de processamento. A coluna de memória utilizada mostra a quantidade exigida pela parte do código (M_{CODE}). A coluna Execução pormenoriza todos os parâmetros que influenciam o tempo de execução indicando o número de ciclos de relógio. A Tabela 6 descreve todos os parâmetros de hardware que influenciam o tempo de execução e a memória usada (M_{DATA}).

Tabela 5 – Rotinas em software com valores individuais de Memória e Tempos de Execução

Descrição da Rotina de Software	Bytes	Execução (Ciclos)
BIST_main(): Função principal que chama todas as restantes rotinas para implementar todo o algoritmo do BIST.	614	_ _ _
XHwICAP Base Driver: Driver base para comunicar com o XHwICAP (inclui as funções Init(), SelfTest() e GetConfigReg()).	7012	_ _ _
XHwICAP DeviceReadFrame(): Rotina para ler um *frame* da memória de configuração da FPGA.	554	NC_{RF} (2750)
XHwICAP DeviceWriteFrame(): Rotina para escrever um *frame* na memória de configuração da FPGA.	784	NC_{WF} (3020)
TPG(): Rotina para gerar um vetor de teste de 32-bit.	56	NC_{TPG32} (6)
CRC(): Rotina para calcular o CRC de um vetor de resultado de 32-bit.	62	NC_{CRC32} (7)
PutGetBitsInFrame(): Rotina para colocar (*Put*) e ler (*Get*) o valor de estado de um flip-flop num *frame* selecionado presente na memória de processamento, onde é lido o estado resultante após aplicar um vetor de teste e é colocado o valor do bit correspondente do próximo vetor de teste.	406	NC_{PGbit} (30)
PutInputInLUT1(): Rotina para colocar o valor de uma *LUT1* (*Proxy Logic*) usada como entrada (*input*), a qual é configurada pelo *frame* que se encontra presente na memória de processamento, de modo a configurar o bit do vetor de teste selecionado.	478	NC_{PLUT1} (117)
Total M_{CODE}	**9966**	_ _ _

Tabela 6 – Características do Hardware

Parâmetros do Hardware	Descrição dos parâmetros para cada módulo em hardware
N_{FFbits}	Número de flip-flops.
$N_{FFframes}$	Número de *frames* que possuem bits INIT de flip-flops usados pelo módulo.
N_{Inputs}	Número de bits usados em todas as entradas.
$N_{ColSlices}$	Número de colunas de *slices* (na mesma linha - *row*) que contêm as LUTs usadas como entradas (*Proxy Logic*).
$N_{RAMbits}$	Número de bits usados como conteúdo de LUTs usadas como memória RAM.
$Frequency$	Valor da frequência do relógio do sistema (usado pela unidade de processamento).

Devido ao facto de se utilizar o IP Core *XHwICAP* da Xilinx, recorreu-se igualmente ao uso do respetivo *driver*, também ele existente na ferramenta EDK. Por esta razão as operações realizadas através

do uso direto deste *driver* não foram otimizadas para este trabalho implementado, perdendo por isso eficiência tanto a nível de tempos de execução como de memória necessária.

A memória exigida para adicionar o BIST é dividida em duas partes: Código (M_{CODE} detalhada na Tabela 5) e dados (M_{DATA}). O volume de memória usada como dados é ainda subdividida em duas partes: permanente e temporária. A memória para dados permanentes (M_{Dperm}) contém a base de dados da localização dos recursos necessários para cada módulo. Isto requer dois bytes para cada estado de um flip-flop utilizado (para localização do respetivo INIT no *frame*), quatro bytes para cada *frame* usado para configurar o estado dos flip-flops ($N_{FFframes}$), para guardar o endereço do *frame* na memória de configuração, e três bytes para cada bit usado como entrada (N_{Inputs}), para mapear a *LUT1* usada como *Proxy Logic* em cada entrada. Por outro lado, a memória de dados temporária (M_{Dtmp}) guarda o contexto dos módulos sob teste e necessita de: 64 bits para o conteúdo de cada LUT (usada como *Proxy Logic* - M_{Inputs}), um bit para cada estado original de cada flip-flop (M_{FFbits}), e os dados do conteúdo de todas as LUTs usadas como RAM, correspondendo ao original contexto das mesmas ($N_{RAMbits}$) (equação 5.1b). Com a equação 5.2 obtém-se a memória total usada como dados.

$$M_{Dperm} = N_{FFbits} \times 2 + N_{FFframes} \times 4 + N_{Inputs} \times 3 \tag{5.1a}$$

$$M_{Dtmp} = N_{Inputs} \times 8 + \lceil \frac{N_{FFbits}}{8} \rceil + \lceil \frac{N_{RAMbits}}{8} \rceil \tag{5.1b}$$

$$M_{DATA} = M_{Dperm} + M_{Dtmp} \tag{5.2}$$

A equação 5.3 retorna a quantidade de memória requerida para o teste de um único módulo do sistema, ao passo que a equação 5.4 calcula a memória total necessária para que seja possível testar todos os módulos desejados do sistema (MEM_{Max}). Este valor total final de memória depende de todos os flip-flops e entradas existentes em todos os módulos do sistema sob teste (M_{Dperm}). A influência dos

dados temporários no valor MEM_{Max}, como no final do teste de cada módulo esta parte de memória é libertada, a mesma poderá ser utilizada pelo próximo módulo a ser testado. Significa isto, que o módulo que exigir a maior quantidade de memória para dados temporários (M_{Dtmp}) decide qual o volume de memória exigido para esta parcela, ou seja, $max(M_{Dtmp})$.

$$MEM_{Total} = M_{CODE} + M_{DATA} \tag{5.3}$$

$$\begin{aligned} MEM_{Max} = M_{CODE} + max(M_{Dtmp}) \\ + \sum_{i=1}^{N} (M_{Dperm\ i}) \end{aligned} \tag{5.4}$$

O número total de ciclos de relógio exigidos para executar um vetor de teste completo (NCV_{Total}) é determinado por seis fatores (5.6):

- NCV_{TPG}, número total de ciclos consumidos pela função *TPG()*, que depende do número de bits necessários para todos os flip-flops e entradas, os quais decidem quantas vezes esta função é chamada (equação 5.5a) e que gera 32 bits a cada vez que é executada;

- NCV_{CRC}, número de ciclos gastos pela função *CRC()*, que depende do número de bits necessários para todos os flip-flops, os quais determinam quantas vezes esta função é chamada (equação 5.5b);

- NCV_{PGbit}, número total de ciclos de relógio exigidos pela função *PutGetBitsInFrame()*, que depende do número de bits usados por todos os flip-flops, por serem estes que decidem quantas vezes esta função é usada (equação 5.5c);

- NCV_{LUT1}, número total de ciclos dedicados à função *PutInputInLUT1()*, que varia em função do número de bits necessários para todas as entradas, os quais determinam o número de vezes que esta função é executada (equação 5.5d);

- NCV_{RF}, número total de ciclos consumidos pela função *XHwICAP DeviceReadFrame()*, a qual depende diretamente do parâmetro $N_{FFframes}$, que decide quantas vezes esta função é chamada (equação 5.5e);

- NCV_{WF}, número total de ciclos de relógio despendidos pela função *XHwICAP DeviceWriteFrame()* que depende dos parâmetros $N_{FFframes}$ e $N_{ColSlices}$ (multiplicado por quatro, porque para escrever todo o conteúdo de uma LUT numa Virtex-5 é necessário escrever em quatro *frames*), que decide quantas vezes esta função é chamada (equação 5.5f).

$$NCV_{TPG} = \lceil \frac{N_{FFbits} + N_{Inputs}}{32} \rceil \times NC_{TPG32} \tag{5.5a}$$

$$NCV_{CRC} = \lceil \frac{N_{FFbits}}{32} \rceil \times NC_{CRC32} \tag{5.5b}$$

$$NCV_{PGbit} = N_{FFbits} \times NC_{PGbit} \tag{5.5c}$$

$$NCV_{LUT1} = N_{Inputs} \times NC_{PLUT1} \tag{5.5d}$$

$$NCV_{RF} = N_{FFframes} \times NC_{RF} \tag{5.5e}$$

$$NCV_{WF} = (N_{FFframes} + N_{ColSlices} \times 4) \times NC_{WF} \tag{5.5f}$$

$$\begin{aligned} NCV_{Total} = {} & NCV_{TPG} + NCV_{CRC} + NCV_{PGbit} \\ & + NCV_{LUT1} + NCV_{RF} + NCV_{WF} \end{aligned} \tag{5.6}$$

Depois de obter o número total de ciclos de relógio, a equação 5.7 é usada para obter o número de vetores de teste aplicados em cada segundo V_{Rate}.

$$V_{Rate} = \frac{Frequency}{NCV_{Total}} \tag{5.7}$$

5.1.5 Conclusão

Este mecanismo de teste baseado num BIST com um *scan chain* virtual está validado na Secção 6.1. Embora tenha demonstrado a

vantagem de poder testar um módulo alocado numa RP sem necessidade de adicionar hardware extra, comporta algumas dificuldades que levou à sua não evolução para uma versão mais eficiente. A principal é a necessidade de ferramentas que garantam uma determinada cobertura de faltas. Se é verdade que a função *TPG()* gera um conjunto de vetores usando um LFSR parametrizável, não existe qualquer garantia em relação à cobertura de faltas que esse conjunto permite. Para isso seria necessário desenvolver um outro fluxo que analisasse cada módulo (tendo em conta os recursos da FPGA usados), e sugerisse os melhores parâmetros para o LFSR existente na função *TPG()*. Também o facto de que as faltas devido a envelhecimento (por causa da radiação ou de alguma forma de BTI), apenas aumentam o atraso na propagação dos sinais, faz com que este tipo de faltas se tornem mais difíceis de detetar por este teste. No entanto, todo o desenvolvimento deste trabalho fomentou o incremento de conhecimentos técnicos relativos às FPGAs e clarificou o foco para o restante trabalho desta tese.

5.2 PROJETO DE *BISTREAMS* PARCIAIS PARA MÚLTIPLAS PARTIÇÕES EM FPGAS DA XILINX

As FPGAs comerciais atuais baseadas em SRAM, cuja estrutura foi explanada no Capítulo 2, possuem uma memória de configuração que controla uma matriz de funções lógicas e as suas interligações. Com a reconfiguração parcial (*Partial Reconfiguration*) é possível alterar parte da FPGA sem ser necessário reconfigurar toda a sua memória de configuração, pois a reconfiguração é realizada sem ser necessário interromper o restante sistema, podendo aliás, até ser feita por ele.

As FPGAs, que podem ser adaptadas para processamento paralelo, permitiam já que o projetista pudesse criar qualquer quantidade de tarefas específicas, desde que elas não excedessem a quantidade de recursos existente na FPGA. Com a reconfiguração parcial passou a ser possível alterar componentes do sistema sem interromper o funcionamento dos restantes componentes. Tal evolução significou que só a quantidade de tarefas específicas em execução é limitada pela quantidade

de recursos. O número total de recursos necessários para todas as tarefas existentes pode por isso ultrapassar todos os recursos disponíveis pelo dispositivo. A reconfiguração parcial fornece por isso uma capacidade de extensão dos recursos existentes num dispositivo.

No entanto, o fluxo permitido pelas ferramentas da Xilinx geram diferentes *bitstreams* parciais para cada RP, inclusive no cenário de ser o mesmo componente. Isto significa que são necessários $N \times M$ *bitstreams* parciais para implementar M componentes em N RPs. Esta restrição obriga a uma maior necessidade de memória para albergar todos os *bitstreams* parciais e reduz a flexibilidade na realocação dos mesmos, o que é particularmente penalizador quando se pretende desenvolver o Módulo de Atuação caracterizado na Secção 4.4.3. Por esta razão, foi essencial desenvolver e automatizar uma forma de gerar *bitstreams* parciais que possam ser alocadas em múltiplas RPs.

O resultado é um novo fluxo ao qual se deu o nome *Assisted Design Flow* (ADF) e que implementa o mecanismo PB4MP (Martins et al., 2015a). O ADF tem como base o fluxo de reconfiguração parcial disponibilizado pelas ferramentas da Xilinx (Xilinx Inc., 2013d), integra regras que fazem parte do IDF referenciado na Secção 3.4.2 (Xilinx Inc., 2016c), e tira proveito de determinadas *constraints* (Xilinx Inc., 2013b). Este mecanismo será a base para implementar o Módulo de Atuação.

5.2.1 Interface e Roteamento na Reconfiguração Parcial

Para que seja possível utilizar o mesmo *bitstream* parcial em diferentes RPs, é necessário cumprir um conjunto de regras. Há por isso cuidados a ter nos recursos existentes em cada RP, no interface de cada RP com o restante sistema e no roteamento dos sinais que constituem as respetivas fronteiras.

Relativamente à zona de recursos para cada RP é necessário seguir estes requisitos:

- Todas as RPs têm de incluir exatamente os mesmo recursos da FPGA com a mesma distribuição física;

- Os limites verticais de cada RP (superior e inferior) precisam de coincidir com as fronteiras das regiões de relógio da FPGA;
- As RPs só podem incluir recursos físicos de entrada/saída do dispositivo se estes não forem usados pelo sistema.

Respeitando estes requisitos e conhecendo a quantidade de recursos que se pretende atribuir a cada RP, definir a área de recursos usando a ferramenta PlanAhead (Xilinx Inc., 2013d) é um processo automático.

5.2.1.1 Interface com as Partições Reconfiguráveis

A compatibilidade da interface das várias RPs é algo que o normal fluxo das ferramentas ignoram. Por isso é necessário encontrar formas de garantir que essa compatibilidade seja implementada. A Figura 53 **a)** e **b)** mostra dois exemplos de roteamento que impedem a criação de interfaces relativamente iguais, enquanto a Figura 53 **c)** mostra como deve ser estruturado o roteamento entre as RPs e o restante sistema.

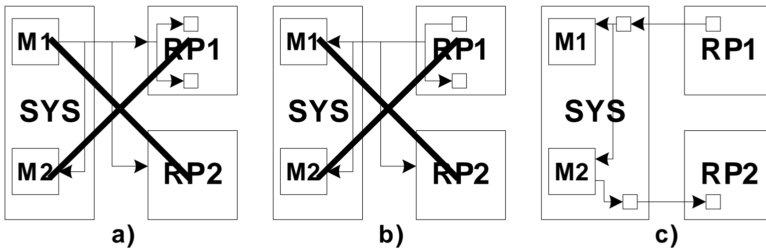


Figura 53 – Regras de Roteamento

A lista de situações, devido ao uso das atuais ferramentas de reconfiguração parcial, que geram incompatibilidade no roteamento dos sinais que constituem o interface entre as RPs e o sistema é a seguinte:

1. Situações em que um sinal de output de uma partição possua um *fanout* maior que um. Neste cenário o sinal tem vários destinos. Alguns deles poderão existir no interior da RP - Figura 53 **b)**;

2. Situações em que a fonte do sinal de entrada de uma partição tenha um *fanout* maior que um. Nesta situação, a entrada do módulo é apenas um dos vários destinos do sinal. Frequentemente o mesmo sinal é a entrada de várias partições, ou pode ser a entrada em mais do que um sub-módulo na mesma RP - Figura 53 **a)**;

3. Como cada região de relógio de uma FPGA possui múltiplas árvores de relógio, durante o *Place and Route*, diferentes árvores de relógio poderão ser selecionadas para as várias RPs.

Em relação às situações 1) e 2), se as partições forem definidas como reconfiguráveis, estas ficam resolvidas do lado das RPs. Configurando o campo *Reconfigurable="true"* no ficheiro *xpartition.pxml* gerado pelo PlanAhead, as ferramentas irão adicionar uma primitiva *LUT1* em cada entrada e saída de cada RP. Estas adições, designadas por *Proxy Logic*, podem ser ancoradas em recursos específicos da FPGA adicionando as seguintes instruções ao ficheiro de *constraints* associado ao projeto:

```
PIN "<rp_name.net_name>" LOC=SLICE_XxYy;
PIN "<rp_name.net_name>" BEL=x6LUT;
```

Onde `rp_name` é o nome do módulo instanciado no *top level* da descrição do hardware (VHDL ou Verilog), `net_name` é o nome do sinal que faz parte do interface entre o módulo e o resto do sistema, `LOC=SLICE_XxYy` indica qual o *slice* onde é fixado e `BEL=x6LUT` força qual das LUTs do *slice* é usada como respetivo *Proxy Logic*. Desta forma, do lado de cada RP, os recursos onde são conectados os sinais das entradas e saídas é fixo e controlado.

Adotando algumas regras do fluxo IDF (Xilinx Inc., 2016c), recorre-se à semelhante *constraint SCC_BUFFER*. Tal como no *Proxy Logic*, esta *constraint* força a adição de uma primitiva *LUT1* em cada entrada e saída, agora do lado do sistema, que interliga com as RPs. A sua localização pode ser igualmente forçada e um exemplo desta *constraint* é:

```
PIN "<system.net_name>" SCC_BUFFER=SLICE_XxYy;
PIN "<system.net_name>" BEL=x6LUT;
```

Onde `system` é o nome atribuído à instância do restante sistema no *top level*, `net_name` é o nome do sinal (do lado do sistema), que faz parte do interface entre o sistema e um dos módulos existentes, `SCC_BUFFER=SLICE_XxYy` indica qual o *slice* onde é fixado e `BEL=x6LUT` força qual das LUTs do *slice* é usada como respetivo *SCC_BUFFER*. Desta forma, também do lado do restante sistema, os recursos onde são conectados os sinais das entradas e saídas é fixo e controlado.

Em relação ao problema da situação 3), recorrendo à primitiva *BUFHCE* (Xilinx Inc., 2013g) com uma *constraint* associada, é possível garantir qual(ais) a(s) árvore(s) de relógio que são utilizada(s) numa determinada RP. Para além da fixação da árvore de relógio, a primitiva *BUFHCE* permite habilitar/desabilitar a entrada do sinal de relógio na respetiva RP. Isto é uma importante exigência no processo de realocação de módulos que necessitem da suspensão do sinal de relógio. Um exemplo de como incluir uma destas *constraints* é o seguinte:

```
INST "<BUFHCE_name_of_instance>" LOC=BUFHCE_XxYy;
```

Onde `BUFHCE_name_of_instance` é o nome da instância da primitiva *BUFHCE* no *top level* do sistema e `LOC=BUFHCE_XxYy` indica qual as coordenadas do *BUFHCE* onde este é fixado. Assim, ancorando estas primitivas numa mesma posição relativa, também a compatibilidade o roteamento dos sinais de relógio é garantida.

5.2.1.2 Roteamento entre o Sistema e as Partições Reconfiguráveis

Na secção anterior foram descritas técnicas que garantem compatibilidade dos recursos da FPGA entre RPs e respetiva localização. Seguem-se as técnicas que garantam compatibilidade no roteamento do sistema.

Uma garantia necessária é que o roteamento de cada RP pertença unicamente ao módulo implementado e não possua sinais do

restante sistema a cruzar a partição. Isso é conseguido adicionando o parâmetro *Isolated="true"* a cada RP na correspondente declaração no ficheiro *xpartition.pxml*.

Outro importante ponto é garantir que o roteamento dos sinais que interligam as RPs ao restante sistema sejam compatíveis. Após a aplicação das técnicas apresentadas na Secção 5.2.1.1, este processo fica possível de realizar recorrendo à opção *Directed Routing Constraints* existente na ferramenta *fpga_edline.exe*. Com esta opção, após executar a ferramenta *Placement and Routing*, é possível extrair a descrição de como foi realizado o roteamento de um conjunto de sinais selecionados. É escolhido então o roteamento existente no interface entre o sistema e uma RP, o qual através da criação de um grupo de *constraints*, é replicado para o interface das restantes RPs, garantindo assim a uniformidade desejada do roteamento do mesmo conjunto de sinais após uma segunda execução do fluxo *Translation* → *Mapping* → *Placement and Routing* (Xilinx Inc., 2013b). Uma *Directed Routing Constraint* com *Absolute Placement Constraint* fornecida pela ferramenta *fpga_edline.exe* é escrita do seguinte modo:

```
NET "net_name"
ROUTE="{3;1;6vlx240tff1156;9aab57fd!-1;"
"42256;-156792;S!0;-683;-224!1;-9745;1703!2;"
"-10236;-2730!3;1789;1251!4;843;392;L!}";
INST "rp_name/rp_net_name_PROXY" LOC=SLICE_X96Y71;
INST "rp_name/rp_net_name_PROXY" BEL="D6LUT";
INST "SCC_BUFFER_sys/sys_net_name" LOC=SLICE_X91Y71;
INST "SCC_BUFFER_sys/sys_net_name" BEL="C6LUT";
```

Esta descrição é dividida em duas partes. A primeira identifica o nome do sinal e apresenta o seu roteamento (`ROUTE=`). A segunda indica quais os recursos que são origem e destino do sinal. Neste exemplo, o sinal tem origem num *Proxy Logic* de uma RP e tem como destino um *SCC_BUFFER* existente no restante sistema.

Na segunda e terceira linha da `ROUTE` *constraint* encontra-se detalhado o roteamento do sinal. Os primeiros dois argumentos definem a coordenada xy absoluta da origem do sinal. Por essa razão é sucedido

por um terceiro argumento `S!0`, que indica que é a fonte (***S**ource*) do sinal e o ponto de conexão **0**. Os restantes grupos de três argumentos definem coordenadas relativas dos seguintes pontos de conexão. O argumento final, `L!`, indica que é o ponto destino do sinal, ou seja, uma entrada num recurso da FPGA. Para obter o valor absoluto em cada ponto de conexão, basta ir somando o valor relativo ao valor absoluto anterior da seguinte forma:

```
 42256;-156792;S!0; (posição absoluta da fonte do sinal)
  -683;   -224!1;
 41573;-157016   (posição absoluta de conexão 1)
 -9745;   1703!2;"
 31828;-155313   (posição absoluta de conexão 2)
-10236;  -2730!3;
 21592;-158043   (posição absoluta de conexão 3)
  1789;1251!4;
 23381;-156792   (posição absoluta de conexão 4)
   843;392;L!}";
 24224;-156400   (posição absoluta do destino do sinal)
```

Na descrição do roteamento de um sinal apenas as coordenadas da origem são absolutas. Por isso, para alcançar o roteamento compatível de dois sinais pertencentes ao interface de duas RPs com o sistema, bastará usar a mesma descrição do sinal, alterando devidamente apenas as coordenadas da origem.

5.2.2 Metodologia ADF Orientada para PB4MP

A capacidade de realocação, primordial para implementar o Módulo de Atuação da Secção 4.4.3, implicou definir uma metodologia com vista a incrementar o nível de confiança do sistema (*dependability*) e que facilite significativamente a gestão dos módulos em hardware na FPGA. O fluxo que representa a metodologia criada foi designado de *Assisted Design Flow* (ADF) e implicou o desenvolvimento de ferramentas a que se chamou PB4MP. Contudo, este fluxo usa igualmente as capacidades de reconfiguração parcial e IDF (Xilinx Inc., 2016c) para realizar a desejada implementação.

A Figura 54 mostra o fluxo de desenvolvimento completo de um sistema em FPGA baseado no PB4MP.

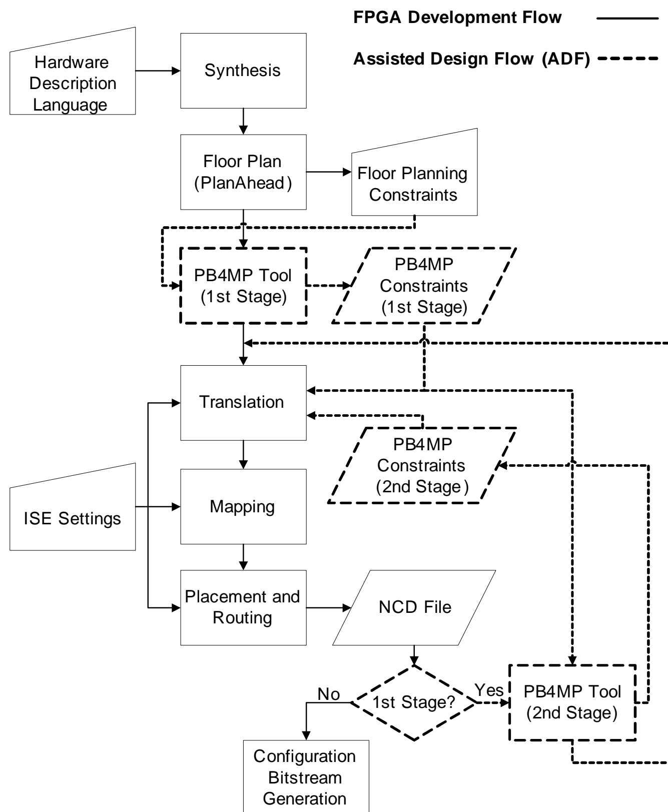


Figura 54 – Fluxos FPGA Development e *Assisted Design Flow* (ADF) (Martins et al., 2015a)

Todo este processo usa o normal fluxo das ferramentas da Xilinx (ISE e PlanAhead) e requer apenas uma intervenção na fase inicial do projeto do hardware, tal como sucede no processo IDF (Xilinx Inc., 2013f). As restantes fases que compõem a metodologia são automáticas.

1. Estruturar corretamente a HDL instanciando os módulos pretendidos no *top level* de modo a que fiquem incluídos nas devidas

RPs. Isto inclui adicionar primitivas BUFHCE para que seja possível habilitar/desabilitar o(s) relógio(s) que entram em casa módulo;

2. Tomar a decisão em relação aos sinais que compõem os interfaces entre as RPs e o restante sistema, se a cada um deles é adicionada uma primitiva SCC_BUFFER, como na Figura 55 **a)**, ou uma primitiva LUT2 que permite desligar o sinal, tal como ilustrado na Figura 55 **b)**. Na segunda opção cada *LUT6* física implementa duas primitivas LUT2 tal como proposto no trabalho (Drahonovský et al., 2013), o que contribui para economizar recursos da FPGA.

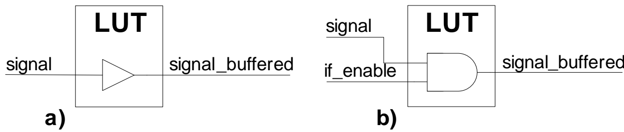


Figura 55 – The Buffer Options

Para auxiliar a inserção das primitivas LUT2 de uma forma automática é usada a seguinte ferramenta desenvolvida (quando se trata da opção das primitivas SCC_BUFFER estas serão adicionadas automaticamente pelas ferramentas de reconfiguração parcial devido à inclusão do parâmetro *Isolated="true"*):

```
pb2mp.exe -B -i <system name>.<vhd/v> [-v]
   -u buffer.ucf
```

Onde o parâmetro -B indica que se trata da etapa de adicionar os *buffers* aos sinais que compõem o interface, que serão neste caso primitivas LUT2. O *top level* da descrição do hardware `<top_level_name>.<vhd/v>` será percorrido (parâmetro -v é adicionado quando se trata de um ficheiro em *verilog*), passará a incluir os novos *buffers* e ao mesmo tempo o ficheiro de *constraints* buffer.ucf será gerado indicando a lista das primitivas adicionadas e o nome dos sinais que entram/saem nas respetivas primitivas. Estas *constraints* serão necessárias numa etapa futura;

3. Sintetizar (*Synthesis*) o HDL do sistema e dos vários módulos incluídos no sistema para o *Register-Transfer Level* (RTL) equivalente, que na Xilinx é designado por *Native Generic Circuit* (NGC). Implica simplesmente usar a ferramenta de síntese `xst.exe` do ISE sem qualquer interferência extra. O resultado dado por esta ferramenta será a geração de um ficheiro `<system>.ngc` para o sistema e a produção de um ficheiro NGC para cada módulo;

4. Definir as RPs (*Floor Plan*) recorrendo à ferramenta PlanAhead. É imperativo seguir os requisitos referenciados na Secção 5.2.1 no momento de decidir quais os recursos de cada RP e a sua localização. No final desta etapa serão gerados os ficheiros `xpartition.pxml`, com a descrição da estrutura de partições, e `<system name>.ucf`, com as respetivas *constraints*;

5. Gerar o primeiro grupo complementar de *constraints* por intermédio da nova ferramenta PB4MP. Estas *constraints* serão responsáveis pela inclusão dos *buffers* (`SCC_BUFFER` ou `LUT2`) no sistema. A forma de usar a ferramenta desenvolvida é a seguinte:

   ```
   pb2mp.exe -F [-B] -i <system name>.<vhd/v> [-v]
      -u <system name>.ucf -o <system name>_f.ucf
   ```

   Onde o parâmetro `-F` informa que se trata do primeiro (***F**irst*) estágio de geração de *constraints* e o parâmetro opcional `-B` indica se a opção dos *buffers* é o uso da primitiva `LUT2` (caso não seja usado este parâmetro, a opção recairá pela primitiva `SCC_BUFFER`). O uso do parâmetro `-B` indica ainda que a informação presente no ficheiro `buffer.ucf` gerado anteriormente deve ser usado.

   Nesta fase as *constraints* não bloqueiam a posição das primitivas usadas como *buffers*. Apenas é definida uma janela (*Buffer Window*) de possíveis localizações para essas primitivas, tal como ilustra a Figura 56. Desta forma é permitido que as ferramentas usadas nas próximas fases possuam alguma liberdade para obter uma melhor colocação de recursos e roteamento (*Mapping* e *Placement and Routing*).

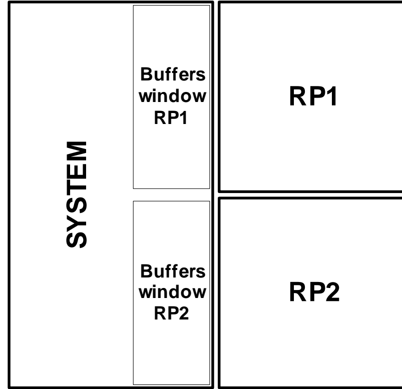


Figura 56 – Buffers Window Example

O resultado desta etapa será o ficheiro de *contraints* `<system name>_f.ucf` que será usado na etapa seguinte;

6. Traduzir (*Tranlation*) o NGC do sistema e dos vários módulos incluídos no sistema para *Native Generic Database* (NGD) equivalente. Requer unicamente utilizar a ferramenta `ngdbuild.exe` do ISE, usando como entradas os ficheiros `<system name>.ngc` e `<system name>_f.ucf`, e como resultado são obtidos os ficheiros `<system name>.ngd` e `<system name>.pcf`. O ficheiro *Physical Constraints File* (PCF) é utilizado para transmitir as *constraints* necessárias para as duas etapas seguintes;

7. Mapear (*Mapping*) os recursos necessários para implementar todo o sistema (incluindo os módulos) nos recursos disponíveis na FPGA destino. Utiliza para isso a ferramenta `map.exe` do ISE, tendo como entradas os ficheiros `<system name>.ngd` e `<system name>.pcf`, e obtendo como resultado o ficheiro `<system name>_map.ncd`, que será usado na próxima etapa;

8. Localizar e rotear (*Placement and Routing*) os recursos necessários para implementar todo o sistema (incluindo os módulos) e as suas interligações na FPGA selecionada. Recorre para isso à ferramenta `par.exe` do ISE e usa como entradas os ficheiros

`<system name>_map.ncd` e `<system name>.pcf`. O resultado final é o ficheiro `<system name>.ncd`, que será analisado no próximo passo;

9. Analisar o ficheiro NCD resultante `<system name>.ncd`. Para isso a nova ferramenta PB4MP volta a ser utilizada e esta por sua vez recorre à opção *Directed Routing Constraints* da ferramenta `fpga_edline.exe` do ISE. Tal como descrito na Secção 5.2.1.2, o roteamento dos sinais que compõem os interfaces entre os módulos alocados nas RPs e o sistema é extraído. Após esta extração existe um grupo de *constraints* `ROUTE` para cada RP, que incluem a correspondente localização dos recursos. A ferramenta PB4MP seleciona então um desses grupos, e usa-o para substituir os restantes, de modo a que todos tenham roteamentos relativos iguais. Com esta informação é gerado um segundo grupo de *constraints* que é gravado no ficheiro `<system name>_s.ucf`. O modo de usar a nova ferramenta desenvolvida é a que se segue:

   ```
   pb2mp.exe -S -i <system name>.ncd
      -u <system name>_f.ucf -o <system name>_s.ucf
   ```

   Onde o parâmetro `-S` informa que se trata do segundo (***S**econd*) estágio de geração de *constraints*.

   Finalizada esta etapa será necessário repetir os processos relatados nas fases 6), 7) e 8) (Translation → Mapping → Placement and Routing);

10. Traduzir (*Tranlation*) novamente o NGC do sistema e dos vários módulos incluídos no sistema para NGD equivalente. Em relação à etapa 6), a única alteração que existe é o ficheiro de *constraints* que deixa de ser o primeiro `<system name>_f.ucf` e passa a ser o novo `<system name>_s.ucf`;

11. Mapear (*Mapping*) outra vez os recursos necessários para implementar todo o sistema (incluindo os módulos) nos recursos disponíveis na FPGA destino. A ferramenta utilizada e forma de a usar é igual ao descrito na etapa 7);

12. Localizar e rotear (*Placement and Routing*) de novo os recursos necessários para implementar todo o sistema (incluindo os módulos) e as suas interligações na FPGA selecionada. A ferramenta `par.exe` do ISE é novamente usada com os mesmos parâmetros que na fase 8);

13. Gerar *bitstreams* de configuração e reconfiguração parcial. Utiliza para esse fim a ferramenta `bitgen.exe` do ISE, tendo como entrada o ficheiro `<system name>.ncd` e produzindo vários *bitstreams* (um para o sistema, o ficheiro `<system name>.bit`, e um *bitstream* parcial para cada um dos módulos);

14. Obter informações dos *bitstreams* parciais necessárias para futuras operações de realocação. Embora os *bistreams* parciais gerados sejam fisicamente compatíveis entre as várias RPs, é necessário obter deles os endereços os endereços FAR (Consultar Secção 2.2.6) e qual o número de palavras que é necessário escrever na memória de configuração para carregar cada um dos módulos (este volume de memória terá de ser igual em todos os *bitstreams* parciais). Para realizar esta tarefa é usada uma outra ferramenta desenvolvida chamada `bit_inform.exe` e que é usada da seguinte forma:

    ```
    bit_inform.exe -i <parcial bitstream name>.bit
       [-S|-V|-7] -o <parcial bitstream name>.h
    ```

    Onde os parâmetros `-S`, `-V` e `-7` indicam se a FPGA usada pertence à família Spartan-6, Virtex-5(6) e 7-Series respetivamente.

    O ficheiro `<parcial bitstream name>.h` gerado para cada *bitstream* parcial será incluído no software que constitui o Módulo de Controlo referido na Secção 4.4.1.

Embora a metodologia esteja descrita por etapas distintas, cada uma usando uma determinada ferramenta, da fase 2) à fase 14) um *script* chamado `pb4mp.cmd` é responsável por executar todas as etapas incluídas de uma forma automática.

Esta metodologia origina por isso um novo fluxo, designado por ADF, que consegue de uma forma automática gerar *bitstreams*

parciais, passíveis de serem trocados diretamente entre RPs. Baseado no fluxo das ferramentas de reconfiguração parcial da Xilinx, utiliza ainda recomendações usadas na metodologia IDF e aproveita algumas técnicas de trabalhos de outros autores automatizando-as. Permite uma maior liberdade às ferramentas *Mapping* e *Placement and Routing*, e acrescenta a possibilidade de desabilitar todos os sinais que constituem o interface de cada módulo alocado numa RP. O fluxo é completo, sem necessidade de qualquer passo manual intermédio, e onde apenas é necessário repetir fluxo *Translation* → *Mapping* → *Placement and Routing* mais uma vez (nas propostas (Ichinomiya et al., 2012b) (Ichinomiya et al., 2012a) (Drahonovský et al., 2013) é repetido mais duas vezes).

5.2.2.1 Memória usada devido ao uso do PB4MP e tempo de execução

O incremento de memória devido ao uso do PB4MP e o tempo de execução das respetivas rotinas em software que serão incluídas no Módulo de Controlo (Secção 4.4.1) depende de vários detalhes da estratégia adotada. A base da estratégia é sempre a mesma, a troca de módulos entre RPs, mas pode ser feita de duas maneiras diferentes:

- Os *bitstreams* parciais dos módulos ficam no exterior da FPGA, numa memória extra existente para o efeito (caso exista capacidade, poderá ser a mesma onde é armazenado o *bitstream* do sistema), e os módulos são (re)alocados carregando os *bitstreams* parciais da memória externa para as RPs pretendidas;
- A troca é realizada movendo a correspondente memória de configuração da FPGA, sem necessidade de uma memória externa para guardar constantemente os *bitstreams* parciais.

A opção preferencial é a segunda porque não precisa de nenhum hardware extra. Mesmo assim, esta via exige mais ou menos memória, ou tempo de execução, em função da performance pretendida/possível. Se o microprocessador existente no sistema tiver alguma memória livre ao seu dispor, poderá ser possível ler e escrever de uma só vez toda

a memória de configuração correspondente a um módulo. Mas se a memória livre for muito limitada, a opção possível será ler e escrever *frame* por *frame*. Procurando o impacto no hardware o mais reduzido possível, a opção *frame* por *frame* foi a adotada para este trabalho.

A caraterização do consumo de memória e do tempo de execução é feita através de um conjunto de equações que são definidas tendo em conta os parâmetros listados na Tabela 7 e na Tabela 8. As funções em software da Tabela 7, a respetiva memória usada (compilado usando *gcc* para a arquitetura MIPS32), e os correspondentes tempos de execução, foram obtidos usando uma FPGA da família Virtex-6. A segunda coluna indica a quantidade de memória requerida para a parte do código (M_{CODE}). A última coluna detalha todos os parâmetros que influenciam o tempo de execução indicado o número de ciclos de relógio necessários.

Tabela 7 – Rotinas em Software com Valores Individuais de Memória e Tempos de Execução

Descrição da Rotina em Software	Memória (Bytes)	Execução (Ciclos)
RP_swap(rp1, rp2): Função que usa todas as restantes rotinas para implementar o algoritmo que troca duas RPs.	418	_ _ _
XHwICAP Base Driver: Base do driver usado para comunicar com o ICAP (inclui as funções *Init()*, *SelfTest()* e *GetConfigReg()*).	7012	_ _ _
XHwICAP DeviceReadFrame(): Rotina usada para ler um *frame* da memória de configuração da FPGA.	554	NC_{RF} (3720)
XHwICAP DeviceWriteFrame(): Rotina usada para escrever um *frame* na memória de configuração da FPGA.	784	NC_{WF} (4010)
Total M_{CODE}	**8768**	_ _ _

A Tabela 8 apresenta todos os parâmetros do hardware que influenciam o tempo de execução e a memória usada para dados (M_{DATA}).

Tal como na Secção 5.1.4.4 a memória necessária para implementar este mecanismo é dividida em duas partes: código (M_{CODE} discriminado na Tabela 7) e temporária para dados (M_{DATA}). A secção de dados é a memória necessária para guardar o endereço inicial das

Tabela 8 – Características das Partições

Parâmetros da RP	Descrição dos Parâmetros dos Módulos em Hardware
$Size_{Frame}$	Número de bytes num *frame*.
N_{RPs}	Número de RPs no sistema.
$N_{RPFrames}$	Número de *frames* que possui a configuração de uma RP.
$Frequency$	Frequência do relógio do sistema (usado pelo CPU).

RPs, quatro bytes para cada uma, e o armazenamento temporário do conteúdo da memória de configuração da FPGA, durante a troca de RPs. No modo mais económico, onde a operação leitura/escrita é feita ao *frame*, é necessário o dobro do tamanho de um *frame* $Size_{Frame}$ (equação 5.8a). No caso mais exigente, quando a troca é feita num bloco só, é necessário o dobro do tamanho da configuração de memória de uma RP. Isto significa duas vezes o tamanho do *frame* $Size_{Frame}$ por cada um dos $N_{RPFrames}$ *frames* (equação 5.8b). Tal duplicação deve-se ao facto de ser sempre necessário salvaguardar o conteúdo de ambas as RPs que são trocadas. Caso a estratégia seja simplesmente carregar um *bitstream* parcial de uma memória externa, escrevendo-o por cima da memória de configuração referente a uma RP (sem salvaguardar dados), então não existe qualquer necessidade deste armazenamento temporário e pode inclusive existir mais módulos do que RPs disponíveis.

$$M_{DATA} = N_{RPs} \times 4 + Size_{Frame} \times 2 \tag{5.8a}$$

$$M_{DATA} = N_{RPs} \times 4 + Size_{Frame} \times N_{RPFrames} \times 2 \tag{5.8b}$$

A equação 5.9 calcula a quantidade de memória necessária, para implementar o mecanismo de troca de dois módulos entre duas RPs tirando proveito da compatibilidade produzida pelo PB4MP. Este total de memória final dependerá sempre da família de FPGA, porque o tamanho do *frame* $Size_{Frame}$ varia, e também da quantidade de *frames* que inclui cada bloco usado em cada operação leitura/escrita.

$$MEM_{Total} = M_{CODE} + M_{DATA} \tag{5.9}$$

O número total de ciclos de relógio exigidos para executar uma troca completa entre duas RPs (NC_{Total}) é determinado por dois fatores (equação 5.10a): família de FPGA, por causa do tamanho do *frame* $Size_{Frame}$, que influencia o número de ciclos consumidos pelas rotinas *XHwICAP DeviceReadFrame()* (NC_{RF}) e *XHwICAP DeviceWriteFrame()* (NC_{WF}); e o tamanho de cada RP $N_{RPFrames}$, que decide quantos *frames* precisam de ser lidos/escritos. Se em vez da troca, a gestão pretendida for fazer o *overwrite* de um *bitstream* parcial existente numa memória externa, não será necessário ler os *frames* que configuram no presente a RP, e os tempos em que a rotina *XHwICAP DeviceWriteFrame()* (NC_{WF}) aguarda pela confirmação da escrita de um *frame*, podem ser usados para carregar o próximo *frame* da memória externa. Neste caso o valor de ciclos necessários será menor e será dado pela equação 5.10b.

$$NC_{Total} = (NC_{RF} + NC_{WF}) \times N_{RPFrames} \times 2 \tag{5.10a}$$

$$NC_{Total} = NC_{WF} \times N_{RPFrames} \tag{5.10b}$$

Após obter o número total de ciclos de relógio, com a equação 5.11 obtém-se o tempo requerido para trocar dois módulos entre RPs (ou carregar um módulo de uma memória externa) em segundos (T_{Swap}).

$$T_{Swap} = \frac{NC_{Total}}{Frequency} \tag{5.11}$$

5.2.3 Conclusão

O principal objetivo deste trabalho foi criar um novo processo automático baseado no fluxo das ferramentas do ISE da Xilinx, que gere *bitstreams* parciais que podem ser re(alocados) em múltiplas partições reconfiguráveis. Desta forma é facilitada a gestão de módulos (ou tarefas) em hardware no sistema, pois a liberdade na escolha da RP destino passa a depender apenas se uma qualquer se encontra livre. Para além de automático, o novo fluxo permite ainda uma maior liberdade às

ferramentas de *place* (em relação a trabalhos anteriores), e consagra a capacidade habilitar/desabilitar o interface de cada RP.

A razão que está na base do desenvolvimento do mecanismo PB4MP foi mais do que a necessidade de obter a liberdade em (re)alocar um módulo em qualquer RP. Na verdade este foi um importante passo para alcançar um objetivo maior: conseguir gerir os recursos da FPGA de modo a ultrapassar a ocorrência de faltas permanentes sem sacrificar RPs completas, e tentar combater problemas de envelhecimento (*aging*), como os causados pelo NBTI. O trabalho descrito na Secção 5.3 é por isso a etapa seguinte natural realizada.

5.3 ESTRATÉGIA TMR COM CARACTERÍSTICAS DE *DEPENDABILITY* MELHORADAS BASEADA NA METODOLOGIA PB4MP

Na Secção 5.2 foi apresentado o fluxo ADF que permite gerar *bitstreams* parciais que podem ser (re)alocados em qualquer RP. Embora com o PB4MP seja possível alterar a posição dos módulos por entre as RPs disponíveis, o que pode permitir isolar uma falta permanente existente numa determinada RP, para isso é necessário detetar a ocorrência de uma falta permanente. Ao mesmo tempo, é importante evitar que ao isolar o recurso da FPGA onde ocorreu a falta, tal implique excluir toda a partição, como sucede em outros trabalhos como (Bolchini et al., 2012), (Espinosa et al., 2012) e (Dumitriu et al., 2015b).

Além disso, o uso de FPGAs em aplicações críticas (como é o caso de aplicações espaciais), onde a ocorrência de uma falta pode resultar numa falha do sistema, tem vindo a aumentar e não há forma de garantir que o sistema seja perpetuamente imune a faltas (transitórias ou permanentes). A tal facto soma-se que no caso de aplicações espaciais estes dispositivos são sujeitos a níveis de radiação mais elevados do que os existentes na superfície terrestre, e que por isso corre um maior risco de sofrer faltas (para além de uma aceleração no seu envelhecimento), então a atenção ao tratamento de faltas deve ser ainda mais reforçado.

Apostando no melhoramento da disponibilidade (*availability*),

confiabilidade (*reliability*) e manutenção (*maintainability*), importantes características de *Dependability*, a capacidade de recuperação de faltas permanentes em sistemas implementados em FPGA foi melhorada, ao mesmo tempo que se previne igualmente a ocorrência dessas faltas causadas por envelhecimento devido ao NBTI (e eventualmente PBTI). Para isso, duas estratégias foram desenvolvidas e aplicadas a um sistema com *Triple Modular Redundancy* (TMR) (Koren and Krishina, 2007). A arquitetura TMR possui as capacidades para FDIR (Guiotto et al., 2003) e foi implementada com recurso à metodologia ADF orientada para PB4MP (Martins et al., 2015d). Numa das fases desta metodologia é estabelecida uma distribuição dos recursos da FPGA em cada RP de forma a que o mecanismo implementado possa desempenhar as duas estratégias desenvolvidas.

5.3.1 TMR baseado em PB4MP

O sistema investigado recorre a um TMR sendo assim possível garantir que o sistema consegue ultrapassar a ocorrência de uma falha que ocorra em algum dos três módulos e, ao mesmo tempo, detetar qual o módulo onde a falha ocorreu, permitindo assim saber qual a RP onde existe pelo menos uma falta que provocou a falha.

Para a implementação do sistema foi usada a metodologia ADF apresentada na Secção 5.2.2, para a qual foi desenvolvida e acrescentada uma nova ferramenta, que força a utilização de conjuntos de recursos da FPGA diferentes entre as várias RPs.

5.3.1.1 Arquitetura TMR

Basicamente, o mecanismo TMR implica implementar três vezes o mesmo módulo/função. Todos os módulos funcionam em paralelo e fazem exatamente o mesmo. Os resultados dos três módulos são analisados por um outro módulo, neste caso um *Central Processor Unit* (CPU), que vota num resultado correto. Deste modo, mesmo que um módulo produza um resultado errado, os outros dois garantem um resultado correto. O ponto fraco desta estratégia é o módulo que vota,

que se falhar, todo o sistema TMR falha. A principal vantagem do TMR é a capacidade de detetar e corrigir erros, sendo a principal desvantagem a necessidade de usar três vezes mais hardware.

5.3.1.2 A Adaptação do PB4MP para Incrementar a Confiabilidade

Para tirar vantagem do uso do PB4MP para incrementar a confiabilidade do sistema, uma lista extra de restrições (*constraints*) é acrescentada à metodologia ADF. Esta lista é elaborada usando a restrição `CONFIG PROHIBIT` que nos permite a opção de selecionar os recursos da FPGA que pretendemos que não sejam usados. Adotando como exemplo um sistema com três RPs, uma possível política de seleção é excluir uma coluna completa de recursos a cada três colunas, delimitadas pela linhas (*rows*) que organização a FPGA (Xilinx Inc., 2015c). Para alcançar a confiabilidade desejada, o processo de exclusão é feito sem nunca excluir uma coluna com a mesma posição relativa de outra já excluída noutra RP, tal como é esquematizado na Figura 57.

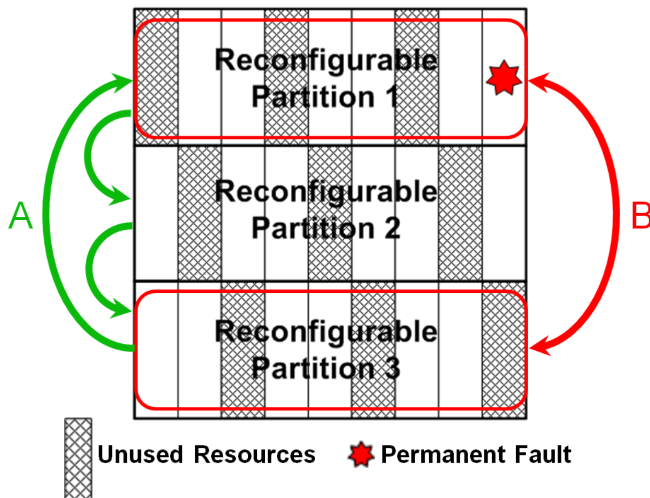


Figura 57 – Distribuição de Recursos nas RPs

A razão para optar por forçar esta distribuição onde uma parte

dos recursos das RPs são excluídos foi anteriormente fundamentada na Secção 4.1.2. Semelhante à distribuição esquematizada na Figura 45, neste caso prático com $r = 3$, para além de suportar a ocorrência de uma falta permanente sem ter de excluir toda a RP, recorrendo à Equação 4.7, a probabilidade de duas faltas permanentes perturbarem o normal funcionamento do sistema é de 44,4%, tal como demonstra o gráfico da Figura 46. Tendo em consideração que um TMR mesmo com um dos módulos a produzir falhas consegue continuar a responder corretamente, significa que apenas após ocorrerem três faltas permanentes, com uma determinada distribuição, é que o sistema como um todo poderá gerar falhas. Sendo a probabilidade de ocorrência de uma falta permanente já muito reduzida, a ocorrência de três faltas será ainda muito mais reduzida. Se a isso se somar o facto de que apenas certas distribuições dessas faltas poderão comprometer a funcionalidade do sistema (terá de ocorrer uma em cada partição e em posições relativas diferentes), então a probabilidade de três faltas permanentes destruírem o sistema tenderá para zero.

Para produzir a lista de restrições que permite forçar esta distribuição é necessário realizar um novo passo na fase 5) da metodologia descrita na Secção 5.2.2. No início da fase 5), a nova ferramenta desenvolvida `gen_constraints.exe` deve ser usada da seguinte forma:

```
gen_constraints.exe -i <system name>.ucf
 -o <system name>_g.ucf
```

Este novo comando percorre o ficheiro de *constraints* gerado pelo PlanAhead (`<system name>.ucf`), analisa a área de recursos atribuída a cada RP declarada e acrescenta a lista de restrições `CONFIG PROHIBIT` de forma a implementar a distribuição da Figura 57. Após este novo passo, segue-se o normal passo que componha anteriormente toda a fase 5), com a única diferença de que agora o ficheiro de *constraints* passou a ser o gerado pelo passo anterior (`<system name>_g.ucf`), tal como exemplifica o seguinte comando:

```
pb2mp.exe -F [-B] -i <system name>.<vhd/v> [-v]
 -u <system name>_g.ucf -o <system name>_f.ucf
```

A nova distribuição permite dois processos da gestão dos recur-

sos em função da finalidade desejada. O processo **A** que permite a troca dos recursos em uso, ajudando a minimizar a degradação devida ao NBTI (Zhang et al., 2014a). O processo **B** onde, mesmo que um módulo sofra uma falta permanente, é possível trocar a RP com outro módulo que não use o mesmo recurso onde a falta permanente foi detetada. A restrição `CONFIG PROHIBIT` exclui recursos (CLB, DSP, BRAMs, etc.) mas não exclui os blocos de roteamento associados. No entanto, se o recurso não é usado, não existe roteamento para esses recursos. Isto significa que o bloco de roteamento não é usado, ou é usado de uma maneira diferente.

Esta distribuição pode parecer um aumento extra de 33% de recursos da RP, no entanto, num sistema implementado numa FPGA, devido ao congestionamento do roteamento isto não é um real *overhead*. Quando o roteamento é limitado ao tamanho de uma RP, este congestionamento é ainda maior. Portanto e na prática, esta política pode nem implicar um incremento no consumo de hardware da FPGA.

5.3.1.3 Memória usada devido à implementação das estratégias e tempo de execução

As rotinas usadas para implementar as duas estratégias ilustradas na Figura 57, usando uma FPGA da família Virtex-6 da Xilinx, são as mesmas e possuem os mesmos parâmetros que as presentes na Tabela 7. De igual modo, os parâmetros que caracterizam o hardware usado por cada módulo, são os enunciados na Tabela 8.

Por estas razões, tanto a memória usada pelo mecanismo que permite os dois processos quanto o seu respetivo tempo de execução, seguem as mesmas equações descritas na Secção 5.2.2.1.

5.3.2 Conclusão

O principal objetivo desta etapa do trabalho foi criar um sistema TMR usando a metodologia da Secção 5.2.2 que permite a implementação PB4MP. Para incrementar alguns atributos relacionados com a *dependability* do sistema duas estratégias foram desenvolvidas. Uma

focada na prevenção relativamente a faltas permanentes que possam ser provocadas por envelhecimento devido ao NBTI, e outra orientada para recuperação de uma falta permanente. Ambas as estratégias estão assentes num prévio planeamento na distribuição dos recursos de cada RP, algo que é realizado por intermédio da inserção de um passo extra na metodologia já existente, e que é feito através do uso de uma nova ferramenta desenvolvida para a mesma finalidade.

Como o módulo que vota o resultado é implementado em software e é por isso o ponto fraco que pode produzir falhas no sistema, será importante dotar o CPU do sistema da funcionalidade de desfazer/-recuperar (*rollback/recovery*) da ocorrência de erros (Li et al., 2013).

Com o processo para recuperar de uma falta permanente, tal como é demonstrado mesmo matematicamente, é possível excluir um recurso com uma falta permanente sem que isso implique excluir toda a partição reconfigurável, o que é uma importante vantagem em relação a trabalhos semelhantes. Relativamente ao processo que se foca na prevenção do envelhecimento dos recursos do dispositivo é necessária mais investigação sobre a influência do NBTI (e mesmo PBTI) na origem de faltas em sistemas implementados em FPGAs, nomeadamente as faltas de atraso (*delay faults*). A próxima secção abordará exatamente essa investigação

5.4 MONITORIZAÇÃO E MINIMIZAÇÃO DOS EFEITOS DE *AGING* NAS FPGAS

Na Secção 3.3 foi demonstrado que os circuitos eletrónicos, onde se incluem as FPGAs, sofrem de envelhecimento (*aging*), o que resulta num processo natural de degradação física. A velocidade a que ocorre este envelhecimento depende de vários fatores como a tecnologia usada, o nível de qualidade no processo de fabrico, temperatura e choque termal (relacionado com o ambiente onde o dispositivo opera), tensão de alimentação, dimensões e a complexidade do próprio circuito (Pradhan, 1996).

As principais ameaças à confiabilidade a serem consideradas

na tal tecnologia incluem os efeitos da radiação (Benfica et al., 2012), e também o BTI, que é o resultado físico/químico que provoca uma degradação do óxido, resultando em um desvio da tensão de *threshold* (V_T) ao longo do tempo. O BTI quando afeta os transístores pMOS é designado por *Negative Bias Temperature Instability* (NBTI), e quando afeta os transístores nMOS é designado por *Positive Bias Temperature Instability* (PBTI). As faltas originadas por estas duas formas de BTI não destroem drasticamente os recursos da FPGA mas aumentam o respetivo tempo de resposta (Mishra et al., 2012). Em determinadas tecnologias estima-se que o NBTI pode provocar uma variação de 5-15% por ano no V_T dos transístores pMOS (Mishra et al., 2012) (Ceratti et al., 2012). Nas tecnologias mais recentes, a influência do PBTI tem vindo a ganhar também um peso relevante (Mizubayashi et al., 2015).

Sendo a manutenção um importante atributo de *dependability*, que sempre contribui para melhorar a disponibilidade e confiabilidade de um sistema (Avizienis et al., 2004), é importante efetuar uma prevenção a faltas que possam resultar devido ao processo de degradação provocado por estas formas de envelhecimento. Este trabalho é exatamente nesse sentido. Desenvolvendo um novo sensor de *aging*, designado por *Differential Delay Sensor* (DDS), e aproveitando a metodologia desenvolvida anteriormente, é realizada uma prevenção da ocorrência de faltas permanentes devidas ao atraso (*delay faults*). O sensor DDS permite igualmente medir a performance da RP e a sua leitura é afetada pelo *aging*, temperatura, tensão de alimentação (VDD) ou variações de processo.

5.4.1 O DDS Desenvolvido

Na equação 3.3 (Secção 3.3.1), observa-se que o atraso num transístor depende do envelhecimento, temperatura, tensão de alimentação e ciclo de trabalho (*duty cycle*). Tal significa que quando o atraso cresce, o aumento pode ser devido ao envelhecimento ou a outra razão, nomeadamente da temperatura que pode facilmente sofrer fortes oscilações. Considerando tal cenário, o sensor DDS foi desenvolvido.

Neste DDS, constituído por dois sensores de atraso *Delay Meter* (DM), são realizadas duas leituras de atraso em simultâneo. Um primeiro DM é o sensor de referência (*gold*) e usa recursos da FPGA que nunca se encontram em utilização (exceto quando são configurados como sensor de referência). Um segundo DM usa recursos sempre que um módulo específico está alocado numa específica RP. Com as duas medições ao mesmo tempo, a diferença de valores entre ambos os DMs corresponde à diferença de performance entre os dois DMs, podendo dar indicações sobre o envelhecimento e sobre as diferenças de temperatura.

Para uma FPGA pouco envelhecida, é previsível que a diferença de valores nos dois DMs não seja devida ao envelhecimento, mas às diferenças de temperatura existentes no silício (conhecidas como *Temperature Hotspots*, ou pontos quentes), provocadas pela maior dissipação de potência média na localização física de um DM em relação ao outro DM (*gold*). No entanto, à medida que a FPGA vai envelhecendo, vão sendo introduzidas alterações nos atrasos dos DMs que serão cumulativos com os atrasos devido à dissipação de potência e aquecimento dos transístores. Porém, como não é conhecida a arquitetura interna dos módulos e da FPGA ao nível de transístor, é impossível prever nesta fase qual o DM que irá ter atrasos maiores provocados pelo envelhecimento. Por exemplo, numa primeira análise poder-se-á supor que será o DM de referência (*gold*) que apresenta atrasos menores, e o DM localizado no módulo em funcionamento apresenta os atrasos maiores. Mas, se considerarmos que o DM de referência, embora não estando a ser configurado e estimulado, estejam os seus correspondentes transístores ligados, e que o outro DM está sendo sempre configurado e estimulado, então poderá também o DM de referência a envelhecer mais e, consequentemente a apresentar os maiores atrasos, devido ao maior envelhecimento dos transístores que estão ligados e estão sempre no estado de *stress* (em oposição aos transístores do outro DM que estarão constantemente a comutar entre *stress* e *recovery*). Assim, é necessário proceder a vários testes de envelhecimento na FPGA para aferir como envelhecem os dois DMs (que poderá variar de FPGA para FPGA, em função da estrutura e funcionamento interno). De qualquer modo, é importante referir que

o DDS pode medir as diferenças de performance entre os dois DM, e sendo feito o registo dos valores lidos e a análise da evolução dos atrasos ao longo do tempo, podemos ter uma noção do envelhecimento de um bloco e da sua temperatura.

O DM usado para construir o DDS é baseado na linha de *carry* existente nos recursos da FPGA, de um modo similar ao sensor de *aging* desenvolvido em (Leong et al., 2015).

5.4.1.1 Arquitetura do *Delay Meter*

Cada DM é composto por uma coluna de multiplexers, conectados em série e com todas as saídas registadas como mostra a Figura 58. Normalmente, a entrada encontra-se sempre no nível lógico '0'. Para medir o atraso, no flanco ascendente do relógio a entrada é forçada ao nível lógico '1', e no seguinte flanco ascendente do relógio todas as saídas dos multiplexers são registadas. Como cada multiplexer implica um pequeno atraso, qualquer variação do atraso implicará um diferente vetor registado nos *flip-flops* usados para registar as saídas dos multiplexers.

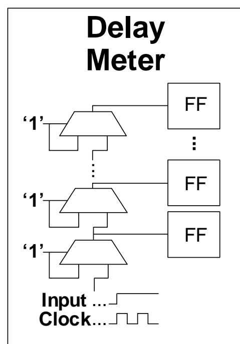


Figura 58 – Diagrama do *Delay Meter* (DM).

A construção do DM é baseada na primitiva *CARRY4* existente em FPGAs da Xilinx (Xilinx Inc., 2013g) e nos *flip-flops* existentes no mesmo *slice* da FPGA. Como mostra a Figura 59, cada primitiva *CARRY4* possui quatro multiplexers, e na família Virtex-6 da Xilinx, o seu conjunto implica um atraso de 68ps. Num sistema com uma

frequência de relógio de 200MHz, uma primitiva *CARRY4* permite detetar uma variação de frequência de relógio em torno de 2,7MHz. Olhando para as quatro saídas dos multiplexers, cada um tem associado um atraso de cerca de 17ps, o que significa uma variação do relógio de apenas 0,7MHz (num mesmo sistema a 200MHz).

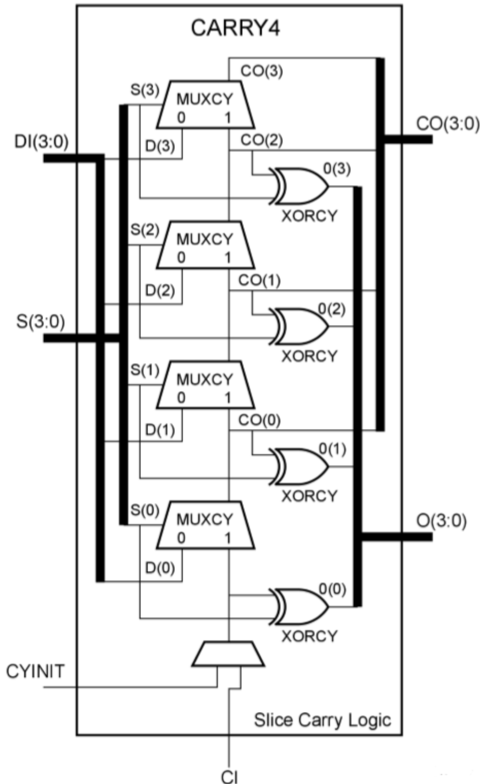


Figura 59 – Primitiva *CARRY4* da família Virtex-6 (Xilinx Inc., 2013g).

Para implementar a funcionalidade desejada do DM, as entradas **DI** e **S** da primitiva *CARRY4* são colocadas no nível lógico ‘1’, a entrada **CINIT** é usada na primeira primitiva *CARRY4* da cadeia para ser a entrada do DM, e a saída **CO(3)** é conectada com a entrada **CI** da seguinte primitiva *CARRY4*. As saídas **O** são ignoradas.

5.4.1.2 Arquitetura do *Differential Delay Sensor*

Para que o sensor DDS meça o verdadeiro *aging*, todos os módulos precisam de possuir dois pequenos grupos de recursos da FPGA reservados. O primeiro, onde o *Delay Meter* de referência é alocado, fornece o valor de referência e usa a mesma posição relativa na FPGA. O segundo grupo possui um conjunto exclusivo de recursos reservados para implementar o outro DM. De forma a reduzir os efeitos de envelhecimento nos recursos usados pelo DM de referência, a correspondente memória de configuração da FPGA é configurada para ficar o menos sujeita a NBTI possível (como se os recursos não sejam usados pelo sistema). Só quando o sistema realiza uma medição é que os recursos são configurados como DM. O segundo grupo, por contrário, outro DM possui os seus recursos configurados de modo a provocar envelhecimento no canal de multiplexers. Quando o sistema decide medir o *aging* e as diferenças de performance, o DM de referência (*gold*) é implementado temporariamente e ambos os atrasos são registados.

Na distribuição de recursos por RPs da Figura 60 (onde é acrescentada uma coluna em relação ao esquema de distribuição presente na Figura 57), a primeira coluna representa os recursos usados pelos DDSs (um em cada RP) e como eles são distribuídos pelos vários DMs.

É visível que o *gold* DM (usado como referência) se encontra sempre alocado na mesma região relativa de recursos do dispositivo. O outro DM que mede o atraso acumulado no módulo (*Module Delay Meter*) usa uma região exclusiva num módulo, sendo a mesma marcada para não ser utilizada noutro módulo (*Unused Resources*).

A opção de recorrer às primitivas *CARRY4* e não a outros recursos da FPGA, como por exemplo LUTs, é porque desta forma a implementação dos DDSs praticamente não necessita de *switch boxes* para interligar os recursos usados (apenas para controlar a entrada do sinal de entrada). Isto traduz-se num melhor controlo da posição e interligação dos recursos, o que permite alcançar a construção de dois DM idênticos fisicamente. Além disso, a fabricante Xilinx não disponibiliza informação técnica relativa à constituição e configuração dos *switch*

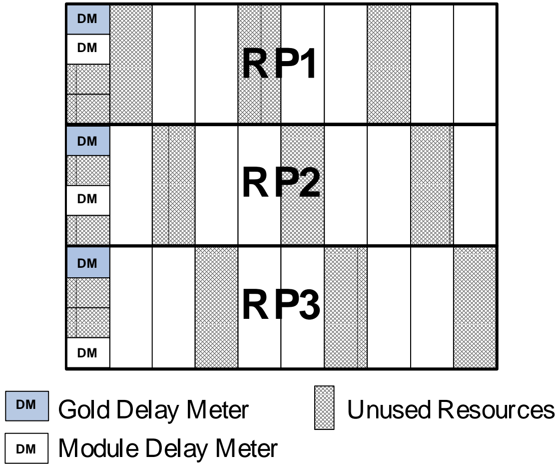


Figura 60 – Distribuição de Recursos nas RPs com DDSs.

boxes, tornando difícil o seu uso e controlo.

Embora existam outros sensores de temperatura ou *aging* implementados em outros trabalhos como (Sengupta and Sapatnekar, 2014), tipicamente são baseados em *Ring Oscillator* (ROSC) o que implica um constante funcionamento em torno de uma determinada frequência. Com o novo DDS, o sensor encontra-se parado (o *gold Delay Meter* nem se encontra implementado 99.9% do tempo), o que significa menos consumo de potência.

5.4.1.3 Análise da Primitiva *CARRY4* com a Ferramenta *AgingCalc*

Para ajudar a demonstrar o funcionamento do DDS, um circuito equivalente da primitiva *CARRY4*, como o da Figura 59, foi desenhado e analisado utilizando a ferramenta *AgingCalc* (dos Santos Pachito, 2012). Cada multiplexer foi construído recorrendo apenas à porta lógica *NAND* de duas entradas existentes nas bibliotecas da ferramenta. Como as FPGAs da família Virtex-6 são produzidas com a tecnologia de 40nm (Xilinx Inc., 2009), e o *AgingCalc* não tem esta opção, duas

bibliotecas CMOS de 32nm foram selecionadas: *32nm_bulk_library* e *32nm_HP_library*. Estas duas bibliotecas do *Predictive Technology Model* (PTM) ((ASU), 2012) foram desenvolvidas pelo grupo *Nanoscale Integration and Modeling* (NIMO), da *Arizona State University* (ASU). A ferramenta *AgingCalc* foi desenvolvida no âmbito de uma tese de Mestrado (dos Santos Pachito, 2012) realizada na Universidade do Algarve, em Faro, e permite analisar o *aging* devido ao NBTI de um circuito SPICE que use as bibliotecas incluídas na ferramenta.

As bibliotecas usadas por esta ferramenta baseiam-se nos modelos detalhados em (Vattikonda et al., 2006), onde é possível prever a evolução do valor da tensão *threshold* (V_T) de um transístor pMOS em função do NBTI a que está sujeito. Com a informação do V_T ao longo dos anos, calculada para cenários de temperatura diferentes, é assim possível simular e observar o aumento do atraso de propagação num circuito causado por este género de envelhecimento.

Após realizar várias simulações, com todas as entradas asseridas a '1', é quando a primitiva *CARRY4* manifesta mais efeitos de *aging* na propagação da cadeia de *carry*, sendo por isso a configuração onde o atraso mais aumenta. Dois cenários de temperatura foram analisados para ambas as bibliotecas: 50°C (122°F) e 100°C (212°F). A Figura 61 mostra os resultados do *AgingCalc* para os primeiros dez anos.

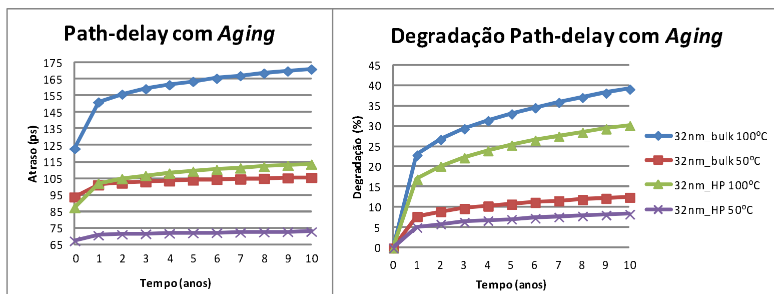


Figura 61 – Análise do NBTI do circuito equivalente à Primitiva *CARRY4*.

Há duas observações importantes sobre estes dados: para a temperatura de 100°C, o envelhecimento faz com que o atraso ao longo

da cadeia de portas *NAND* que compõem os multiplexers aumente cada vez mais; e uma grande parte da degradação ocorre no primeiro ano (mínimo 5% e máximo 22%). Isto significa que, quando não é possível assegurar uma temperatura correta (inferior a 30°C), é imperativo combater os efeitos devidos ao NBTI desde o primeiro momento.

Os resultados dos gráficos da Figura 61 foram obtidos a partir de um circuito equivalente à primitiva *CARRY4*, aplicando às suas entradas o vetor que mais induzia envelhecimento devido ao NBTI. Quando foi aplicado o vetor que menos envelhecimento provocava (com todas as entradas atribuídas a '0'), em todas as simulações, a degradação após dez anos foi entre 0% e 1%. Isto indica que será possível evitar que os recursos da FPGA atribuídos ao *golden Delay Meter* envelheçam devido ao NBTI, permitindo assim alcançar o objetivo principal do DDS, que é medir ao longo do tempo o envelhecimento sem interferência da temperatura (ou tensão de alimentação). Como não é conhecido o verdadeiro circuito da primitiva *CARRY4* (ao nível da porta lógica e transístores), nem a tecnologia exata que é usada pelo fabricante, tal impede chegar a conclusões mais sustentadas. No entanto, mesmo que no caso real os valores tenham discrepâncias, dificilmente o fenómeno causado pelo NBTI não se manifestará (com uma dimensão maior ou menor).

5.4.1.4 Uso do *Delay Meter* como Medidor de Temperatura

Embora o DM tenha sido desenvolvido para, em par, implementar um sensor DDS, o seu valor isolado pode ser convertido num valor de temperatura local dentro da FPGA. Isto porque, para uma mesma tensão de alimentação e envelhecimento, se a temperatura subir, o atraso medido pelo DM irá igualmente aumentar.

Como o valor lido por um DM corresponde a um atraso e não a uma temperatura, é necessário chegar a uma função que permita a conversão valor de atraso $\rightarrow$ temperatura. A conversão esperada pode ser aproximada a uma conversão linear (para uma primeira aproximação), seguindo o exemplo do gráfico da Figura 62.

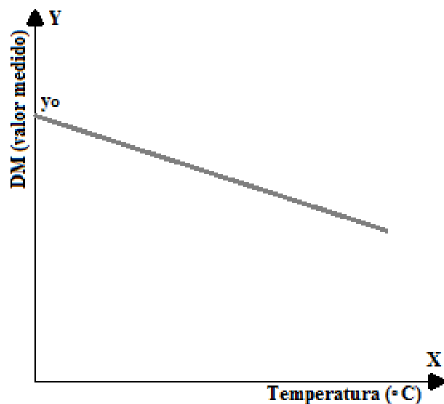


Figura 62 – Gráfico do valor lido pelo DM em função da temperatura.

No entanto, é preciso notar que esta conversão aproximada só é válida para um DM que não sofra de influências de envelhecimento (ou que sejam reduzidas) e considerando que não existem flutuações da tensão de alimentação (normalmente provocadas pela operação do dispositivo). Para além disso, a utilização prática do DM como sensor de temperatura carece ainda de validação prática numa FPGA que envelhece, uma vez que pode ser necessário avaliar a evolução das medições à medida que o DM vai envelhecendo.

Para a temperatura 0°C o sensor DM irá obter um valor y_o. Consoante a temperatura for subindo o atraso vai aumentando, e como explicado na Figura 58, a saída dos vários multiplexers é registado num dado instante. Com o aumento do atraso, significa que menos *flip-flops* irão registar o valor lógico '1', pelo que o valor obtido pelo DM tenderá a diminuir linearmente. A inclinação da reta presente na Figura 62 dependerá da tecnologia usada em cada FPGA, mas assumindo que a reta tem uma inclinação m, a equação 5.12 representa a função que converte o valor da temperatura no valor lido pelo DM.

$$y_{(x)} = -m.x + y_o \tag{5.12}$$

Tendo o valor lido pelo DM, para o sistema o converter para

obter a temperatura é fazer o cálculo inverso, tal como indica a equação 5.13.

$$x_{(y)} = \frac{y_o - y}{m} \tag{5.13}$$

Para que seja possível o sistema realizar estas conversões, é necessário que previamente seja feita uma caracterização do sensor DM. Tal processo foi feito parcialmente no experimento da Secção 6.4 e deve der repetido para qualquer família de FPGA (caso na mesma família sejam usadas tecnologias diferentes, é igualmente necessário preceder a esta caracterização).

5.4.1.5 Memória Usada e Tempo de Execução Relativos à Implementação do DDS

Sendo a implementação dos DDSs incluída nos módulos construídos pela metodologia que possibilita o PB4MP, então, no sistema final, as rotinas usadas serão as mesmas que as da Tabela 7 presente na Secção 5.2.2.1. No entanto, o uso do DDS obrigou ao acréscimo de uma nova rotina que prepara e realiza a medição dos valores dos dois *Delay Meter*s incluídos em cada DDS. A nova rotina, `DDS_measure(rp)`, junta-se assim às restantes e completa a Tabela 9.

(*) No caso da rotina `DDS_measure(rp)`, após detalhado estudo da estrutura da memória de configuração da FPGA adotada (Virtex-6 family), observou-se que cada coluna de CLBs (que são configurados em grupo) contém 40 CLBs. Para configurar por completo todos estes 40 elementos lógicos são necessários 36 *frames*. No entanto, apenas 12 *frames* são usados para configurar as propriedades dos CLBs ($N_{CLBsConfFrames} = 12$). Os restantes 24 são usados para configurar os *switch boxes* que não são necessários para implementar os DMs, e por essa razão, não há necessidade de modificar a região da memória correspondente. Para configurar a zona de memória requerida pelos 12 *frames*, não é necessário ter memória para salvaguardar todos os *frames* completos. É suficiente possuir as máscaras para alterar apenas os bits necessários. Assim é possível evitar que seja necessário reservar

Tabela 9 – Rotinas em Software com Valores Individuais de Memória e Tempos de Execução

Descrição da Rotina em Software	Memória (Bytes)	Execução (Ciclos)
RP_swap(rp1, rp2): Função que usa todas as restantes rotinas para implementar o algoritmo que troca duas RPs.	418	_ _ _
DDS_measure(rp): Função que usa as restantes rotinas XHwICAP abaixo para medir os valores presentes no DDS numa RP (*).	2376	NC_{DDS}
XHwICAP Base Driver: Driver usado para comunicar com o ICAP (inclui as funções *Init()*, *SelfTest()* e *GetConfigReg()*).	7012	_ _ _
XHwICAP DeviceReadFrame(): Rotina usada para ler um *frame* da memória de configuração da FPGA.	554	NC_{RF} (3720)
XHwICAP DeviceWriteFrame(): Rotina usada para escrever um *frame* na memória de configuração da FPGA.	784	NC_{WF} (4010)
Total M_{CODE}	**11144**	_ _ _

mais memória do sistema para esta finalidade. No entanto, a necessidade de incluir a informação relativa às máscaras, faz com que a rotina `DDS_measure(rp)` exija um certo volume de memória. No final, com a inclusão desta nova rotina, a quantidade de memória M_{CODE}, mantendo o mesmo compilador que anteriormente, sofreu um acréscimo de 2376 bytes para os 11144 bytes.

No que se refere aos parâmetros que caracterizam o hardware usado por cada módulo, também a Tabela 8 passou a incluir um novo parâmetro $N_{CLBsConfFrames}$, tal como mostra a nova Tabela 10.

Tabela 10 – Características das Partições com DDS incluído

Parâmetros da RP	Descrição dos Parâmetros dos Módulos em Hardware
$Size_{Frame}$	Número de bytes num *frame*.
$N_{CLBsConfFrames}$	Número de *frames* necessários para configurar uma coluna de CLBs.
N_{RPs}	Número de RPs no sistema.
$N_{RPFrames}$	Número de *frames* que possui a configuração de uma RP.
$Frequency$	Frequência do relógio do sistema (usado pelo CPU).

Como já referido, o novo parâmetro $N_{CLBsConfFrames}$ corres-

ponde ao número de *frames* exigidos para configurar uma coluna de CLBs delimitada pela região de relógio (apenas as configurações dos CLBs, não incluindo os *frames* necessários para os *switch boxes* realizarem o roteamento).

Tendo em consideração a Tabela 9, o cálculo da memória total exigida pela inclusão do mecanismo PB4MP com DDSs obedece à equação 5.9, descrita na Secção 5.2.2.1. Também o número de ciclos exigidos para fazer uma troca de RP, após consultar a Tabela 10, continua a seguir a equação 5.10b.

A operação realizada pela rotina `DDS_measure(rp)`, implica ler cada um dos $N_{CLBsConfFrames}$ *frames*, proceder à alteração dos bits em função da máscara correspondente, de modo a implementar o *gold Delay Meter* (ou desfazer a mesma implementação após a realização da leitura do DDS), e escrever de volta o *frame* alterado. A equação 5.14 formaliza o cálculo do número de ciclos necessários para que a rotina `DDS_measure(rp)` realize uma operação completa de medição.

$$NC_{DDS} = (NC_{RF} + NC_{WF}) \times N_{CLBsConfFrames} \tag{5.14}$$

As dimensões do DM permitem que seja implementado numa única coluna de CLBs. Devido a isso, o valor NC_{DDS} apenas depende da família de FPGAs, a qual tem impacto em todos parâmetros (NC_{RF}, NC_{WF} e $N_{CLBsConfFrames}$).

5.4.2 Conclusão

Esta nova etapa do trabalho teve como objetivo adicionar mais um pouco de confiabilidade a um sistema implementado numa FPGA. Mas ao contrário das fases anteriores do trabalho, em que através da metodologia apresentada se dotava principalmente o sistema da capacidade de recuperar de faltas permanentes, nesta etapa o foco foi a prevenção de faltas permanentes, acima de tudo *delay faults* que possam ser originadas por influência do *aging* causado pelo NBTI (e eventualmente também pelo PBTI). Nesse sentido um novo sensor

de *aging* e de performance, o *Differential Delay Sensor* (DDS), foi desenvolvido para que seja possível monitorizar simultaneamente a performance da RP e obter uma indicação do possível NBTI acumulado numa determinada secção de recursos dessa mesma partição.

Neste trabalho, apenas o NBTI foi analisado. Presentemente a ferramenta *AgingCalc* ainda não suporta análises ao PBTI, e por essa razão não é possível realizar uma análise equivalente para ambas as formas de BTI. No entanto, o *AgingCalc* usa os modelos *Predictive Technology Model*, os quais terão certamente algumas diferenças em relação às tecnologias utilizadas no fabrico da FPGA selecionada. O *Triple-Oxide Approach* (Xilinx Inc., 2009) é um exemplo usado pela Xilinx para alcançar uma redução significativa no consumo de potência no modo estático. A verdadeira construção da primitiva *CARRY4* (em portas lógicas) também poderá ter diferenças em relação ao circuito usado na análise relatada na Secção 5.4.1.3, mas com uma menor ou maior influência, o aumento do atraso no caminho crítico será inevitável. Será por isso enriquecedor refazer a mesma análise com a mesma ferramenta *AgingCalc*, quando esta suportar também o PBTI e incluir bibliotecas da tecnologia FinFET (2nm e 7nm), de modo a avaliar o feito devido ao BTI nas famílias mais recentes de FPGAs.

5.5 CONCLUSÕES GERAIS E TRABALHOS FUTUROS

Neste capítulo foram descritos os trabalhos realizados no sentido de alcançar o planeado no diagrama da Figura 49. Embora inicialmente o fluxograma previsto fosse o da Figura 50, após os vários trabalhos desenvolvidos este acabou por evoluir para o fluxograma presente na Figura 63.

A grande alteração foi a adoção do mecanismo TMR que para além de permitir superar a ocorrência de uma falha num módulo, identifica qual a RP onde existe uma falta que gerou essa falha. Assim, deixou de ser necessário realizar um teste periódico a cada módulo quando este se encontra alocado numa RP (operação que seria da responsabilidade do Mecanismo de Deteção). No entanto, não existe a

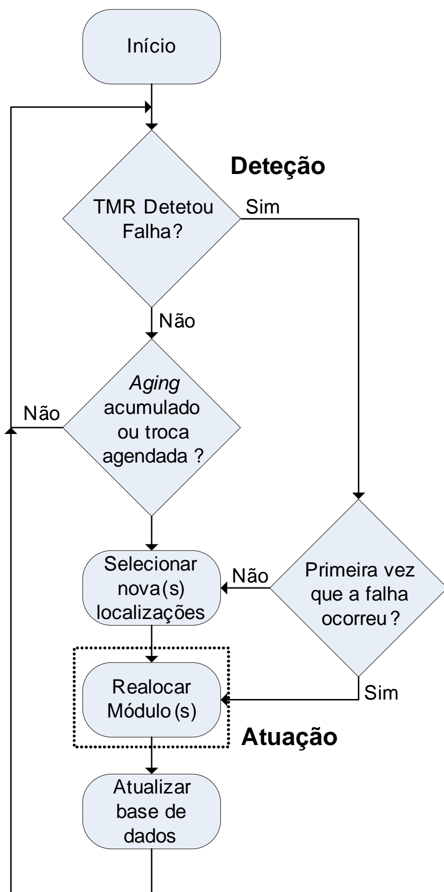


Figura 63 – Fluxograma geral da aplicação desenvolvida com TMR e rotação de recursos.

obrigação de recorrer ao TMR, podendo o projetista optar pelo fluxo planeado originalmente.

Após o desenvolvimento das várias etapas descritas neste capítulo, é agora possível comparar a produção do presente trabalho com os outros trabalhos anteriormente estudados e que foram organizados na tabela existente na Secção 3.6. De forma a facilitar esta comparação à mesma tabela foi acrescentado o presente trabalho, dando origem à Tabela 11.

Tal como observado na Secção 3.6, o presente trabalho apresenta uma nova metodologia que permite a realocação de módulos (o mecanismo PB4MP), e a utiliza para elaboração de uma plataforma com tolerância a faltas permanentes, dotando o sistema da capacidade de auto-recuperar-se da ocorrência de faltas. A esta fusão, através de uma política cuidada de distribuição de recursos da FPGA, é igualmente possível tornar pelo menos uma falta permanente que provoca falhas num módulo numa falta transparente. Assim, é evitado o desperdício resultante da exclusão total da partição onde foi detetada uma falta permanente, procedimento usual noutras plataformas com tolerância a faltas permanentes.

A contribuição desta tese é por isso aproveitar as vantagens demonstradas nos trabalhos desenvolvidos por outros autores dedicados à realocação de múltipla módulos, e inclui-las numa plataforma onde o sistema possa recuperar de pelo menos uma falta permanente. Isto porque nos trabalhos encontrados relativos a plataformas orientadas para recuperar de faltas permanentes, a realocação depende de uma biblioteca que tem de incluir obrigatoriamente um *bitstream* parcial para cada combinação módulo-partição. Para além disso, nesses mesmos trabalhos de plataformas com recuperação a faltas permanentes, a forma de recuperar de uma falta permanente tipicamente passa por realocar o módulo noutra RP e excluir a atual RP. Resumidamente, este trabalho propõe uma nova plataforma que tira partido da realocação múltipla para ganhar maior flexibilidade, reduzir biblioteca de *bitstreams* parciais, e soma a isso a eficiência de poder recuperar de pelo menos uma falta permanente sem necessitar de excluir uma RP.

Para além das inovações do parágrafo anterior, a metodologia apresentada permite ainda uma rotação da utilização dos recursos da FPGA (existentes nas RPs), o que vai ao encontro do desejo de minimizar o envelhecimento desses recursos devido ao NBTI (e eventualmente PBTI). Este tipo de envelhecimento aumenta, por regra, o tempo de atraso dos transístores que compõem os recursos do dispositivo, o que a uma dada altura no ciclo de vida da FPGA começa a provocar *delay faults*. Na última etapa de desenvolvimento, foi ainda desenvolvido e adicionado um

sensor de performance e de envelhecimento. Este novo sensor, designado por *Differential Delay Sensor*, faz uma medição baseada no diferencial entre os valores obtidos de dois *Delay Meter*.

No próximo capítulo deste documento serão descritos os vários experimentos realizados com a finalidade de validar os vários trabalhos apresentados no presente capítulo.

Tabela 11 – Comparativo do presente trabalho com os trabalhos relativos a Realocação de Módulos e Plataformas com Tolerância a Faltas Permanentes

	Fluxo de Geração de Realocação **M**anual ou **A**utomático	Analisa as Possibilidades de Localização (**S**/N)	Alguma Liberdade no Roteamento dos Sinais (**S**/N)	Gestão de Sinais de Relógio na Realocação (**S**/N)	Realocação **Ú**nica ou **M**últipla de Módulos	Suporta Famílias de FPGAs Recentes (**S**/N)	Tolerante a Faltas Transitórias e Permanentes	Recupera de Faltas Permanentes (**S**/N)	Exclui Partição para Recuperar de Faltas Permanentes	Controlo de Realocação Descentralizado	Prevenção *Aging*
(Ichinomiya et al., 2012b)	M	N	N	-	M	S	-	-	-	N	N
(Ichinomiya et al., 2012a)	M	N	N	-	M	S	-	-	-	N	N
(Koch et al., 2008)	A	N	N	-	M	N	-	-	-	N	N
(Beckhoff et al., 2012)	A	N	N	-	M	S	-	-	-	N	N
(Beckhoff et al., 2013a)	A	N	N	-	M	S	-	-	-	N	N
(Touiza et al., 2012)	A	N	N	-	M	S	-	-	-	N	N
(Drahonovský et al., 2013)	M	N	N	-	M	S	-	-	-	N	N
(Steiner et al., 2011)	-	-	S	-	-	S	N	N	-	-	N
(Love et al., 2013)	-	-	S	-	-	S	N	N	-	-	N
(Flynn et al., 2009)	-	-	S	S	-	S	N	N	-	-	N
(Backasch et al., 2014)	A	S	S	-	M	S	-	-	-	N	N
(Xilinx Inc., 2013d)	A	N	S	S	U	S	-	-	-	-	N
(Xilinx Inc., 2016c)	-	-	S	-	U	S	S	N	-	-	N
(Gericota, 2003)	-	-	S	-	-	N	S	S	N	-	N
(Montminy et al., 2007)	-	N	S	-	U	N	S	S	S	N	N
(Iturbe et al., 2011)	-	N	S	-	M	S	S	-	-	N	N
(Iturbe et al., 2012)	-	N	S	S	M	S	-	-	-	N	N
(Bolchini et al., 2012)	-	N	S	-	U	S	S	S	S	N	N
(Ochoa-Ruiz et al., 2013)	A	N	N	-	M	S	-	-	-	N	N
(Espinosa et al., 2012)	A	N	S	-	U	S	S	S	S	N	N
(Lawson et al., 2014)	-	-	-	-	-	N	S	S	N	-	N
(Kirischian et al., 2006)	A	N	S	-	U	S	S	S	S	N	N
(Dumitriu and Kirischian, 2010)	A	N	S	-	U	S	S	S	S	N	N
(Dumitriu et al., 2014)	A	N	S	-	U	S	S	S	S	S	N
(Dumitriu et al., 2015a)	A	N	S	-	U	S	S	S	N	S	N
(Dumitriu et al., 2015b)	A	N	S	-	U	S	S	S	S	S	N
Presente Trabalho	A	N	S	S	M	S	S	S	N	N	S

Parte III

ETAPA DE VALIDAÇÃO

6 EXPERIMENTOS E RESULTADOS

No capítulo anterior, sequencialmente, foram apresentados os trabalhos associados a cada etapa de evolução desta tese. Neste capítulo, na mesma ordem, serão detalhados os vários experimentos que foram feitos para validar cada uma dessas etapas. Desses mesmos experimentos foram obtidos resultados que após interpretados ajudaram a comprovar o respetivo trabalho e levaram a tomadas de decisão que influenciaram os trabalhos seguintes.

6.1 EXPERIMENTO COM DETETOR DE FALTAS DESENVOLVIDO

Para validar o detetor de faltas descrito na Secção 5.1 foi implementado um sistema embarcado composto por um MicroBlaze e quatro módulos, tal como esquematizado na Figura 64.

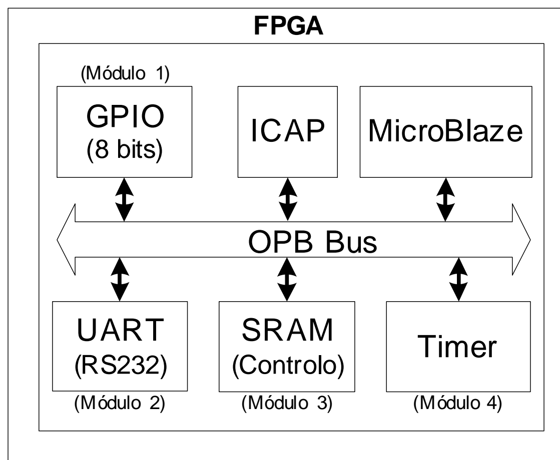


Figura 64 – Diagrama do sistema implementado para validação do BIST com *scan chain* virtual.

Os quatro módulos selecionados são: um GPIO de 8 bits (Módulo 1); uma UART RS232 (Módulo 2); um controlador SRAM (Módulo

3); e um temporizador (*Timer*) (Módulo 4). As características destes quatro módulos estão indicadas na tabela 12.

Tabela 12 – Parâmetros dos quatro módulos testados

Parâmetros	**Módulo 1**	**Módulo 2**	**Módulo 3**	**Módulo 4**
N_{FFbits}	124	145	570	246
$N_{FFframes}$	6	13	18	14
N_{Inputs}	222	183	355	186
$N_{ColSlices}$	6	15	17	13
$N_{RAMbits}$	512	1216	2048	0
$Frequency$	100MHz	100MHz	100MHz	100MHz

6.1.1 Resultados Experimentais

A Tabela 13 e a Tabela 14 apresentam os resultados baseados nas equações descritas na Secção 5.1.4.4. Para cada módulo, na Tabela 13, foi calculada a percentagem da memória usada em cada uma das componentes que perfazem a memória total. A coluna Média mostra a média das quatro percentagens. Semelhante cálculo foi realizado na Tabela 14 em relação ao número de ciclos de relógio para cada parâmetro em relação ao seu total.

Tabela 13 – Memória alocada nos quatro módulos testados

Memória	**Módulo 1**	**Módulo 2**	**Módulo 3**	**Módulo 4**	**Média**
M_{Dperm}	938	891	2277	1106	9,52%
M_{Dtmp}	1857	1635	**3168**	1520	15,01%
M_{DATA}	2795	2526	5445	2626	24,53%
M_{CODE}	9966	9966	9966	9966	75,47%
MEM_{Total}	12761	12492	15411	12592	100%

6.1.2 Avaliação dos Resultados Obtidos

Os resultados mostram que o maior valor de M_{Dtmp} dos quatro módulos é 3168, o que significa que $MEM_{Max} = 18346$ (equação 5.4).

Tabela 14 – Número de Ciclos de Relógio por cada Vetor de Teste

Ciclos	Módulo 1	Módulo 2	Módulo 3	Módulo 4	Média
NCV_{TPG}	71	68	179	87	0,04%
NCV_{CRC}	34	39	132	61	0,02%
NCV_{PGbit}	3720	4350	17100	7380	2,92%
NCV_{LUT1}	25974	21411	41535	21762	11,50%
NCV_{RF}	16500	35750	49500	38500	13,15%
NCV_{WF}	90600	220460	259720	199320	72,38%
NCV_{Total}	136899	282077	368166	267110	100%
V_{Rate}	730	355	272	374	

Relativamente ao número de vetores de teste por segundo, V_{Rate}, este varia entre 272 e 730. O acréscimo de hardware exigido para incluir este teste no sistema, serão as *LUT1s* normalmente adicionadas pela ferramenta que implementa a reconfiguração parcial, quatro primitivas BUFGCTRL, 103 flip-flops, e 55 LUTs para o processador controlar as primitivas BUFGCTRL.

Analisando os resultados e a equação 5.4, se apenas um módulo for testado, os incrementos de memória M_{Dperm} e M_{Dtmp} são cerca de 10% e 15% respetivamente. Quando vários módulos poderão ser testados, a componente de memória M_{Dperm} passa a ter um relevo maior e a parte de memória M_{Dtmp} passa a ter menos impacto no total de memória. A memória referente ao código (M_{CODE}) é sempre a principal parcela do requisito total de memória, e a principal contribuição para isso deve-se às rotinas usadas do driver *XHwICAP*.

Em conjunto, as operações de escrita e leitura de *frames* da memória de configuração ($NCV_{WF} + NCV_{WF}$) são responsáveis por cerca de 85% do processamento necessário. Este é o principal limitador da performance desta implementação, mas é possível ser minimizado otimizando para este detetor as rotinas usadas do driver *XHwICAP*. Independentemente disso, o V_{Rate} será sempre inferior ao de um "tradicional" BIST implementado num ASIC. O processamento necessário pelas rotinas TPG e CRC é nesta versão negligenciável.

Os resultados e as equações mostram que o número de flip-flops e o número de entradas do módulo têm uma forte influência na eficiência do detetor de faltas (M_{Dperm} and NCV_{LUT1}). No que diz respeito ao número de flip-flops, não há solução para reduzir a sua necessidade. Já relativamente ao *interface* dos módulos (entradas/saídas), outra opção de barramento, que implique menos sinais de interligação, trará uma redução no número de entradas, o que permite obter uma melhor eficiência no que ao processamento diz respeito.

A cobertura de faltas e a eficiência na deteção das mesmas para esta proposta de BIST, é sempre dependente do conjunto de vetores de teste gerados pela rotina TPG. No entanto, nesta implementação, o objetivo principal era definir uma nova estratégia de teste baseada num BIST e verificar se era possível implementá-la. Os resultados mostram que tecnicamente esta abordagem pode ser uma opção para testar faltas do tipo *stuck-at-0* e *stuck-at-1* que ocorram em recursos usados por um módulo alocado numa determinada RP.

6.1.3 Conclusões e Trabalho Futuro

Sendo o alvo deste detetor de faltas testar funcionalmente, de um modo autónomo, um determinado módulo numa determinada RP de uma FPGA, quando esta se encontra em ambientes hostis que lhe possam causar faltas (transitórias ou permanentes), então o objetivo foi alcançado. Se por um lado, o número de vetores de teste por segundo não permite um teste muito rápido (algo esperado), por outro, tirando algum incremento de memória no sistema embarcado, o restante hardware adicionado é residual.

Este mesmo trabalho é passível de ser otimizado, desenvolvendo principalmente um *driver* mais eficiente que substitua o IP Core *XHwI-CAP* da Xilinx. Tal é possível de alcançar porque como o driver existente foi desenvolvido para ser generalista, existem várias sub-funções desnecessárias para a finalidade desejada. Como se observou que 70% do código existente é desnecessário e que cerca de 80% dos ciclos de relógio consumidos são a realizar operações sem utilidade para este trabalho,

estimam-se por isso uma reduções semelhantes para a memória necessária para o código e de processamento nas funções de ler e escrever os *frames* na memória de configuração.

Como trabalho futuro, seria importante desenvolver uma ferramenta que simule/emule o circuito de modo a obter um polinómio para o LFSR que permita verificar cada módulo com uma boa de cobertura de faltas, gerando o menor número de vetores possível.

6.2 EXPERIMENTO COM IMPLEMENTAÇÃO DO MECANISMO PB4MP DESENVOLVIDO

De modo a validar o mecanismo PB4MP, foi implementado numa FPGA da família Virtex-6 (XC6VLX240T) da Xilinx um sistema embarcado com um *softcore* e várias RPs (Mück and Fröhlich, 2013), tal como esquematizado na Figura 65.

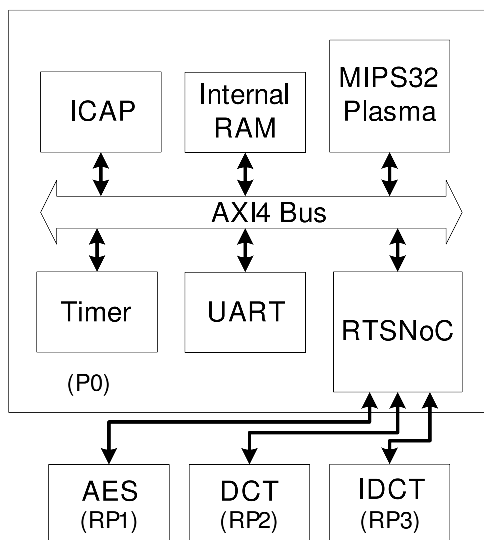


Figura 65 – Diagrama do Sistema (Martins et al., 2015a).

A plataforma de teste é baseada numa implementação de um MIPS32, o *softcore* Plasma, que é livre e encontra-se disponível no Open-

cores (OpenCores, 2014). A estrutura interna de comunicação do CPU é baseada na família de protocolos AXI4, que se tornaram o standard na indústria para a interconexão baseada em *bus*. A reconfiguração do hardware é realizada pelo interface ICAP interligado ao AXI4, que é gerido pelo *Operating System* (OS).

O OS que corre no CPU Plasma é o *Embedded Parallel Operating System* (EPOS) (Project, 2014), que providencia o suporte necessário para implementar as necessidades do sistema e a comunicação com os restantes módulos. RP1, RP2 e RP3 representam as RPs que contêm neste caso os componentes *Advanced Encryption Standard* (AES), *Discrete Cosine Transform* (DCT) e *Inverse Discrete Cosine Transform* (IDCT) respetivamente.

Devido à familiaridade no seu uso no laboratório onde foi feito o experimento, foi utilizado um esquema de interconexão baseado na *Real-Time Star Network-on-Chip* (RTSNoC) (Berejuck, 2011) para as partições reconfiguráveis. A RTSNoC consiste em *routers* com uma topologia em estrela que podem ser organizados de modo a formar uma malha 2-D. Cada *router* tem oito canais bidirecionais que podem ser conectados a componentes em hardware ou a canais de outros *routers*. Cada componente presente numa RP é conectado a uma porta de um *router* RTSNoC e um porto do *router* é conectado a uma ponte AXI4.

A síntese do módulo maior, o AES, reportou a necessidade de 4172 LUT *Flip-Flop pairs*. Como na família Virtex-6 cada *Slice* tem quatro LUT *Flip-Flop pairs*, um CLB tem dois *Slices*, e uma secção de coluna de CLBs possui 40 CLB, então cada uma destas secções de coluna possui 320 LUT *Flip-Flop pairs*. Isto significa que seriam precisas pelo menos 13 secções destas colunas. No entanto, devido ao congestionamento do roteamento durante a operação de *Placement and Routing*, foi necessário aumentar a disponibilidade de recursos para 20 secções de colunas de CLBs.

Durante o processo da aplicação da Metodologia da Secção 5.2.2, a opção na fase 2) foi pelo uso da primitiva `SCC_BUFFER` e por isso foi usado o seguinte comando:

```
pb2mp.exe -i system_platform.vhd -u buffer.ucf
```

Na fase 4) da mesma Metodologia, através da ferramenta PlanAhead, foram definidas as três RPs de modo a incluir 20 colunas de CLBs pertencentes a uma mesma região de relógio. As áreas de recursos fixadas no final foram as seguintes:

```
INST "AES" AREA_GROUP = "RP0";
AREA_GROUP "RP0" RANGE=SLICE_X114Y40:SLICE_X151Y79;
INST "DCT" AREA_GROUP = "RP1";
AREA_GROUP "RP1" RANGE=SLICE_X114Y80:SLICE_X151Y119;
INST "IDCT" AREA_GROUP = "RP2";
AREA_GROUP "RP2" RANGE=SLICE_X114Y120:SLICE_X151Y59;
```

No final do processo automático, para gerar o ficheiro `parcial_system_platform.h` necessário para incluir no software que irá correr no CPU, o seguinte comando foi executado:

```
bit_inform.exe -i system_platform.bit -V
-o parcial_system_platform.h
```

Sendo o sistema constituído por três RPs, cada uma com um módulo alocado, optou-se por uma gestão básica neste experimento, onde é implementada uma troca periódica dos módulos entre as RPs de uma forma circular. O seguinte pseudo-código exemplifica como é a sequência da aplicação usada para esta demonstração.

```
void main_application(){
  ...
  init_swap_counter();
  while (1){
    ...
    if (read_swap_counter() >= SWAP_COUNTER_LIMIT) {
      RP_swap(&RP1, &RP2);
      RP_swap(&RP2, &RP3);
      reset_swap_counter();
    }
    ...
  }
}
```

Existe um contador que é inicializado no início, e enquanto a aplicação é executada, o sistema verifica periodicamente o valor desse contador. Se o contador ultrapassa o limite definido (`SWAP_COUNTER_LIMIT`), duas operações de troca são executadas, de modo a implementar a sequência RP1 $\rightarrow$ RP2 $\rightarrow$ RP3 $\rightarrow$ RP1... Antes

de realizar as alterações, todos os sinais do interface são desligados em ambas as RPs.

Todos os módulos estão alocados numa RP e cada uma delas tem as características presentes na Tabela 15.

Tabela 15 – Características de cada RP

Parâmetros da RP	Valores dos Parâmetros
$Size_{Frame}$	324
N_{RPs}	3
$N_{RPFrames}$	1392 (880 para CLBs + 512 para BRAMs)
$Frequency$	50MHz

6.2.1 Resultados Experimentais

Tal como registado na Tabela 7 da Secção 5.2.2.1, a memória usada pelo código (M_{CODE}) é fixa. Relativamente à memória temporária para dados (M_{DATA}) esta é obtida diretamente através da equação 5.8a. A soma destas duas partes, como a equação 5.9 indica, corresponde à memória extra total necessária para implementar este mecanismo de troca rotativo.

O tempo de execução em ciclos de relógio para a troca entre duas RPs (NC_{Total}) é calculado pela equação 5.10a, e o tempo em segundos (T_{Swap}) é obtido pela equação 5.11. Na Tabela 16 encontram-se todos este valores calculados.

Este foi um exemplo simples para testar a capacidade de realocar módulos em qualquer RP, sem necessidade de ter uma biblioteca de *bitstreams* parciais com todas as combinações dos módulos para cada RP. Num sistema mais realista irão existir mais módulos que RPs. Nesse caso a geração dos restantes *bitstreams* parciais será feita pelo PlanAhead, aproveitando a capacidade desta ferramenta de recuperar e fixar todo o roteamento do sistema anterior (exceto nas RPs onde serão alocados os novos *bitstreams* parciais), e assim desta forma, automaticamente os *bitstreams* parciais serão igualmente compatíveis.

Tabela 16 – Memória e Número de Ciclos de Relógio e Tempo para troca entre RPs

Variáveis	Valores Calculados
M_{CODE}	8768 Bytes
M_{DATA}	660 Bytes
MEM_{Total}	9428 Bytes
NC_{Total}	21520320 Ciclos
T_{Swap}	0,430 Segundos

6.2.2 Avaliação dos Resultados Obtidos

Da Tabela 16 observa-se que a implementação desta versão simplista do mecanismo precisa apenas de 9428 bytes de memória extra do sistema principal, sendo a parte do código (M_{CODE}) a principal responsável por este incremento.

Relativamente ao tempo de execução para a troca entre duas RPs, NC_{Total}, são necessários 20159840 ciclos de relógio, ao que corresponde uma frequência de 50MHz 0,403 segundos. Para implementar a rotação completa, tal como indica o pseudo-código anteriormente apresentado, são necessárias duas trocas. Isto significa cerca de 0,806 segundos para realizar uma rotação completa. Embora este valor tenha sido avaliado, a sua importância sempre depende do sistema implementado (nuns casos o sistema estar *offline* 0,8 segundos poderá ser desprezível e noutros ser catastrófico). No entanto, o objetivo principal desta tese é acima de tudo o sistema recuperar de uma falta permanente (evitando que a ocorrência de uma destas faltas possa destruir para sempre o sistema implementado numa FPGA), pelo que este tempo, embora deva ser o mais reduzido possível, não é um fator importante. Na literatura científica estudada relativamente a plataformas com igualmente com recuperação de faltas permanentes, este fator é mesmo ignorado.

Resumindo, sendo esperado que o incremento de memória no sistema fosse de apenas alguns KBytes, o principal objetivo deste experimento foi verificar que os *bitstreams* parciais gerados pela Metodologia

desenvolvida poderiam ser realocados livremente pelas três RPs disponíveis. Tal foi validado sem que qualquer erro devido a *timmings* tenha sido detetado. Tal era igualmente esperado pelo facto de ter sido seguida a política de registar as entradas e saídas de cada módulo, tornando assim indiferente qual a RP onde cada módulo é realocado.

6.2.3 Conclusões e Trabalho Futuro

Sendo o principal objetivo deste teste validar a metodologia que permite implementar a estratégia PB4MP, tal foi validado usando três módulos distintos que foram ciclicamente alternando entre as três RPs disponíveis, sem nunca ter sido detetado nenhum erro na execução das suas funções.

Se a cada módulo for adicionada a capacidade de teste, este mecanismo permite implementar a capacidade de recuperação de faltas permanentes ao sistema, bastando para isso não realocar um módulo numa RP onde lhe foi detetada uma falta permanente. Tendo em conta os objetivos desta tese, este será o foco do próximo experimento.

Ainda como trabalho futuro, a investigação de como o PB4MP pode prolongar o ciclo de vida da FPGA através da redução do efeito de stress causado pelo NBTI é outra prioridade, sendo ainda a migração do ADF para o fluxo do Vivado da Xilinx um passo natural.

6.3 EXPERIMENTO DA IMPLEMENTAÇÃO DE UM SISTEMA COM TMR BASEADO NO MECANISMO PB4MP

A validação do sistema com TMR detalhado na Secção 5.3 foi feita aproveitando o mesmo sistema embarcado implementado na Secção 6.2. Continuando a usar uma plataforma com uma FPGA da família Virtex-6 (XC6VLX240T) da Xilinx o sistema sofreu poucas alterações, como mostra o diagrama da Figura 66.

A plataforma de teste continua assim a incluir a implementação de um MIPS32, o *softcore* Plasma, uma estrutura interna de comunicação baseada na família de protocolos AXI4 e a reconfiguração do hardware é

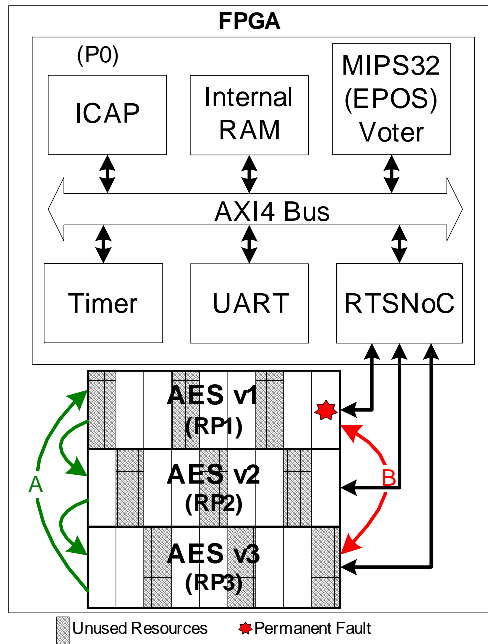


Figura 66 – Diagrama do Sistema (Martins et al., 2015d).

realizada pelo interface ICAP, também ele interligado ao AXI4 e gerido pelo OS.

Tal como no experimento anterior, RP1, RP2 e RP3 representam as RPs, no entanto agora cada uma contém uma versão do componente AES. Os mesmos dados de entrada são enviados pelo CPU para cada um dos três módulos que compõem o TMR e após o devido processamento, todos os resultados são recolhidos pelo mesmo CPU. Os três resultados são analisados por uma função que realiza a votação implementada em software. Se alguma diferença é detetada, o módulo onde ocorreu uma falha é identificado e outro módulo presente noutra RP é selecionado para trocar de RP com ele. Antes de realizar a troca, as entradas de sinal de relógio são desativadas em ambas as RPs.

Cada módulo AES ocupa 4172 LUT *Flip-Flop pairs*, o que na família Virtex-6 (Xilinx Inc., 2015c) significa que são necessários pelo

menos 522 CLBs. Sabendo que cada coluna de CLBs (delimitada pela sua região de relógio, que corresponde às fronteiras superior e inferior de cada RP), contém 40 CLBs, então no mínimo serão exigidas 14 destas colunas. No entanto como já tinha sido verificado no experimento da Secção 6.2, devido ao congestionamento do roteamento durante a operação de *Placement and Routing* (ainda sem qualquer *constraint* `CONFIG PROHIBIT`), foi necessário aumentar a disponibilidade de recursos para 20 secções de colunas de CLBs. Este facto resultou na oportunidade de implementar as estratégias esquematizadas na parte inferior do diagrama da Figura 66, sem implicar um incremento extra de 33% de recursos da FPGA.

Seguindo o mesmo procedimento da Secção 6.2, na fase 2) da aplicação da metodologia da Secção 5.2.2, a opção foi pelo uso da primitiva `SCC_BUFFER` e por isso foi usado este mesmo comando:

```
pb2mp.exe -i system_platform.vhd -u buffer.ucf
```

Até chegar ao valor final de colunas de CLBs por RP foram necessárias várias iterações a partir da fase 4) até chegar ao valor mínimo sem dar congestionamento no roteamento. Na iteração final foram definidas pela ferramenta PlanAhead as três RPs de modo a incluir 23 colunas de CLBs pertencentes a uma mesma região de relógio. As áreas de recursos fixadas no final foram as seguintes:

```
INST "AES" AREA_GROUP = "RP0";
AREA_GROUP "RP0" RANGE=SLICE_X106Y40:SLICE_X151Y79;
INST "DCT" AREA_GROUP = "RP1";
AREA_GROUP "RP1" RANGE=SLICE_X106Y80:SLICE_X151Y119;
INST "IDCT" AREA_GROUP = "RP2";
AREA_GROUP "RP2" RANGE=SLICE_X106Y120:SLICE_X151Y59;
```

Para que seja possível executar as estratégias da parte inferior do diagrama da Figura 66 é então necessário acrescentar um novo passo na fase 5) da metodologia associada ao fluxo ADF. Neste passo, responsável por criar a lista de restrições `CONFIG PROHIBIT`, a ferramenta desenvolvida para esta função é então chamada do seguinte modo :

```
gen_constraints.exe -i system_platform.ucf
-o system_platform_g.ucf
```

De seguida, o normal comando que já faz parte da mesma metodologia é executado da seguinte forma:

```
pb2mp.exe -F -i system_platform.vhd
-u system_platform_g.ucf -o system_platform_f.ucf
```

Finalizado o processo automático, é gerado o ficheiro `parcial_system_platform.h`, que será incluído no software que irá correr no CPU. Tal é feito por intermédio do seguinte comando:

```
bit_inform.exe -i system_platform.bit -V
-o parcial_system_platform.h
```

O seguinte pseudo-código exemplifica de uma forma básica como funciona o algoritmo da aplicação de validação, que desempenha as duas estratégias delineadas na Figura 66 em conjunto com o TMR.

```
void main_application(){
  ...
  init_swap_counter();
  first_AES_error = 0;
  while (1){
    ... // Processo A
    if ((read_swap_counter() >= SWAP_COUNTER_LIMIT) && (first_AES_error == 0)) {
      RP_swap(&RP1, &RP2);
      RP_swap(&RP2, &RP3);
      reset_swap_counter();
    }
    ...
    send_data(&AES1, input_data);
    send_data(&AES2, input_data);
    send_data(&AES3, input_data);
    ...
    get_data(&AES1, output_data[1]);
    get_data(&AES2, output_data[2]);
    get_data(&AES3, output_data[3]);

    // Processo B
    AES_error = vote_AES(output_data, result_out);

    if (AES_error) {
      switch () {
      case 1:
        if (last_AES_error == 1) RP_swap(&RP1, &RP3);
        else RP_swap(&RP1, &RP2);
        break;
      case 2:
        if (last_AES_error == 2) RP_swap(&RP2, &RP1);
        else RP_swap(&RP2, &RP3);
        break;
      case 3:
        if (last_AES_error == 3) RP_swap(&RP3, &RP2);
        else RP_swap(&RP3, &RP1);
        break;
      default:
        break;
      }
```

```
            last_AES_error = AES_error;
            first_AES_error++;
        }
        ...
    }
}
```

No processo A, existe um contador que é inicializado no início, e enquanto a aplicação é executada, o sistema verifica periodicamente o valor desse contador. Se o contador ultrapassa o limite definido (`SWAP_COUNTER_LIMIT`), duas operações de troca são executadas, de modo a implementar a sequência RP1 → RP2 → RP3 → RP1... Antes de realizar as alterações, todos os sinais do interface são desligados em ambas as RPs.

Para o processo B, os dados de entrada são enviados para os três módulos AES. Após o tempo de processamento necessário, todos os resultados são lidos dos mesmos módulos. Os três resultados são analisados pela função `vote_AES()`, a qual se detetar alguma diferença em algum dos módulos, retorna a indicação do módulo que falhar. Nesse cenário haverá uma troca de módulos entre RPs. Quando o processo B deteta pela primeira vez uma falha, a variável `first_AES_error` é incrementada, e ao deixar de ser nula, bloqueia indefinidamente o processo A. Este bloqueio é importante, pois poderia ocorrer que uma falta permanente fosse detetada e isolada pelo processo B, e depois numa futura rotação executada pelo processo A, essa mesma falta poderia voltar a produzir falhas num dos módulos.

Relativamente às RPs, cada uma delas possui as características indicadas na Tabela 17.

Tabela 17 – Características de cada RP

Parâmetros da RP	**Valores dos Parâmetros**
$Size_{Frame}$	324
N_{RPs}	3
$N_{RPFrames}$	1524 (1012 para CLBs + 512 para BRAMs)
$Frequency$	50MHz

6.3.1 Resultados Experimentais

Ignorando o software associado à aplicação (incluindo a função `vote_AES()`) e tendo apenas em consideração o incremento de memória devido às rotinas necessárias para desempenhar as duas estratégias implementadas, este é o mesmo que no experimento anterior, que se encontra mencionado na Secção 6.2.1. Ou seja, a memória usada pelo código (M_{CODE}) é fixa e encontra-se discriminada na Tabela 7 da Secção 5.2.2.1, enquanto a memória temporária para dados (M_{DATA}) é obtida diretamente através da equação 5.8a. O total de memória, corresponde à soma das duas partes como evidencia a equação 5.9.

O tempo de execução em ciclos de relógio para a troca entre duas RPs (NC_{Total}) é calculado pela equação 5.10a, e o tempo em segundos (T_{Swap}) é obtido pela equação 5.11. Na Tabela 18 encontram-se todos este valores calculados.

Tabela 18 – Memória e Número de Ciclos de Relógio e Tempo para troca entre RPs

Variáveis	**Valores Calculados**
M_{CODE}	8768 Bytes
M_{DATA}	660 Bytes
MEM_{Total}	9428 Bytes
NC_{Total}	23561040 Ciclos
T_{Swap}	0,471 Segundos

Comparando com os resultados da Tabela 16 na Secção 6.2.1, as exigências de memória mantêm-se ao passo que o número de ciclos necessários para fazer uma troca (NC_{Total}) subiu de 21520320 para 23561040 ciclos. Tal aumento deve-se ao facto de que cada RP passou a conter uma área maior de recursos o que implica um maior número de *frames* a ser transferidos.

De modo a auxiliar a análise da influência da política de distribuição de recursos ilustrada na parte inferior do diagrama da Figura 66 nos *bitstreams* parciais, a ferramenta `bit_inform.exe`, previamente

desenvolvida, foi melhorada de modo a analisar os três *bitstreams* (dos três módulos), e realizar a comparação entre os correspondentes *frames* de memória de configuração. Os resultados da comparação estão listados na Tabela 19.

Tabela 19 – Interferência da lista de *Constraints* nos *Bitstreams* Parciais

Detalhe	**Sem *Constraints***	**Com *Constraints***
Média Total Bits 0	1689700	1685917
Média Total Bits 1	280219	284002
Total Posições Comuns Bits 0	1396224 (-18%)	1342436 (-21%)
Posições Comuns Roteamento Bits 0	1055328 (-15%)	1025529 (-17%)
Posições Comuns Configuração Bits 0	340896 (-76%)	316907 (-78%)
Total Posições Comuns Bits 1	59063 (-79%)	14728 (-95%)
Posições Comuns Roteamento Bits 1	18706 (-87%)	4755 (-97%)
Posições Comuns Configuração Bits 1	40357 (-72%)	9973 (-93%)

As primeiras duas linhas mostram a média do número de bits 0 ou bits 1 nos três *bitstreams* parciais analisados. As restantes linhas reportam a quantidade de bits (0 e 1) que possuem o mesmo valor lógico na mesma posição relativa nos três *bitstreams*. Estas linhas são divididas em dois grupos: bits de roteamento, usados para configurar os *switching boxes* responsáveis pelo roteamento; e bits de configuração, responsáveis pela configuração do interior dos CLBs. Para cada valor é indicada a percentagem de redução em relação ao valor da média total (por exemplo: a redução de 18% no total de posições comuns com o bit 0 sem *constraints*, corresponde à redução do valor da média total bits 0, 1689700, para o valor 1396224).

6.3.2 Avaliação dos Resultados Obtidos

O novo passo acrescentado à metodologia (com o comando `gen_constraints.exe`) implicou um aumento de 20 para 23 colunas de CLBs. Significou por isso, que o dotar do sistema TMR com as duas novas estratégias, trouxe um incremento de 15% de recursos em cada

RP, o que é menos de metade em relação aos 33% que o diagrama da Figura 66 poderia induzir. Devido a este aumento de recursos o número de ciclos necessários para fazer uma troca (NC_{Total}) também teve um incremento de 9,48%.

Da análise do conteúdo dos *bitstreams* parciais feita pela ferramenta `bit_inform.exe` desenvolvida, observa-se que a redução de bits 0 em comum é pouco expressiva. Isto deve-se ao facto de que por norma, se um recurso não é usado a sua correspondente memória de configuração é preenchida com zeros. Como resultado, todos os bits a 0 usados para configurar um CLB irão manter-se a 0 noutro *bitstream* onde o mesmo CLB não seja usado.

Já em relação aos bits 1, os resultados mostram que com as *constraints* `CONFIG PROHIBIT` que conduzem à desejada distribuição, o número de posições de memória desses bits que mantiveram o mesmo valor 1 teve uma redução de cerca de 95% (97% nos bits de roteamento e 93% nos bits de configuração). Esta redução nunca poderá ser de 100% porque sempre existirá o interface entre cada módulo e o restante sistema na FPGA, o que devido à metodologia desenvolvida para implementar o PB4MP, precisa de ser obrigatoriamente igual.

A redução no número de posições comuns relacionadas com o roteamento mostra que com a mesma distribuição de recursos, a memória de configuração dos *switching boxes* é afetada praticamente do mesmo modo, mesmo não existindo *constraints* que o forcem diretamente.

6.3.3 Conclusões e Trabalho Futuro

O principal objetivo deste experimento foi implementar um sistema numa FPGA com TMR, recorrendo à metodologia desenvolvida que permite obter o funcionamento PB4MP. Com a adição de um novo passo a esta metodologia, que através de uma lista de restrições estabelece uma distribuição de recursos utilizáveis diferente para cada RP, foi possível realizar as duas estratégias apresentadas na Figura 66.

O resultado foi um sistema que para além de possuir um TMR que lhe permite detetar e ultrapassar uma falha num dos módulos,

consegue ainda recuperar de pelo menos uma falta permanente que ocorra em uma das RPs. Tudo isto sem que no sistema total ocorra uma falha (exceto se ocorrer uma falha na função `vote()`), o que incrementa consideravelmente alguns atributos associados à *dependability* do sistema, tais como confiabilidade e disponibilidade.

Sendo a função `vote()` que corre no CPU o ponto fraco do sistema, onde se ele falhar todo o sistema falha, este poderá ser igualmente implementado em hardware (necessitando de incluir a capacidade de comunicar a ocorrência de falhas dos módulos ao CPU), seguindo as regras sugeridas pelo FDIR (Holsti and Paakko, 2001). Em alternativa, no futuro, uma solução baseada em trabalhos como (Li et al., 2013) poderá melhorar a confiabilidade relativa ao que corre no CPU, o que incluirá a função `vote()`.

Com vista na análise da estratégia **A** da Figura 66, os resultados da Tabela 19 mostram que a política de forçar uma distribuição de recursos distinta para cada partição, para além de induzir a utilização diferente dos recursos entre módulos, também a respetiva memória de configuração possui valores lógicos diferentes. Tal não permite concluir de uma forma direta que irá ocorrer uma redução na progressão do envelhecimento devido ao NBTI ou PBTI. A verdade é que medir e analisar esta provável redução sem acesso à devida informação técnica do fabricante é impossível. Mas considerando que quando os recursos são usados de diferentes formas, a memória de configuração que controla os elementos da FPGA (células de memória e multiplexers) também é alterada, então estes resultados indicam que a polarização de muitos transístores existentes nesses elementos é alterada, o que alguma influência terá na missão de redução dos efeitos de BTI no envelhecimento desses elementos. No entanto não foi possível quantificar o valor desta influência.

6.4 EXPERIMENTO DE UM SISTEMA COM MONITORIZAÇÃO DE PERFORMANCE E DE ENVELHECIMENTO ATRAVÉS DO SENSOR DDS

O teste do DDS descrito na Secção 5.4 foi feito realizando uma nova iteração ao sistema embarcado implementado na Secção 6.3 (Figura 66). Mantendo a mesma política de distribuição de recursos, foi acrescentada uma nova coluna onde se encontram os DDSs, e, para tornar possível a caracterização dos *Delay Meter*s e monitorizar o sistema, foi incluída a primitiva *System Monitor* (Xilinx Inc., 2014d), que permite medir a temperatura, a tensão de alimentação (V_{CCINT}) e a respetiva corrente (I_{CCINT}) do dispositivo. Após estes incrementos o sistema ficou tal como o diagrama da Figura 67.

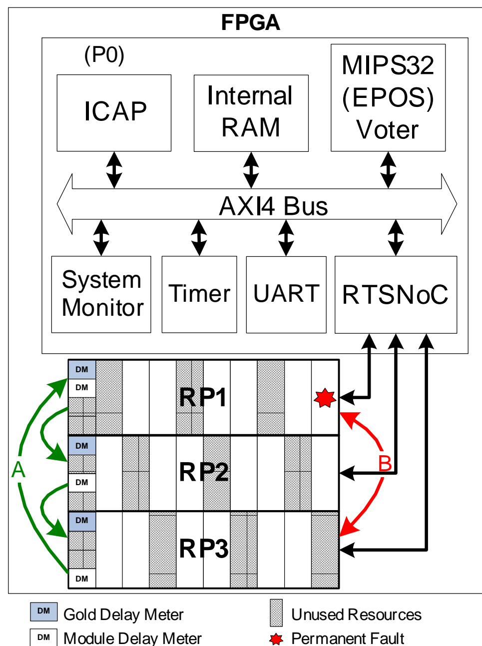


Figura 67 – Diagrama do Sistema com DDSs e primitiva *System Monitor*.

Embora nos dois programas desenvolvidos o foco tenha sido testar e validar apenas o sensor desenvolvido, o hardware implementado

continua a reunir as condições exigidas para implementar os processo **A** e **B** mencionados na Secção 5.3.1.2, tal como indicado na Figura 67.

A arquitetura continua com três RPs tendo cada uma alocado um módulo AES. No entanto a funcionalidade do TMR foi deixada para segundo plano, sendo o foco apenas sobre o funcionamento do DDS e dos DMs que o compõem. Relativamente ao experimento da Secção 6.3, onde cada RP incluía 23 colunas de CLBs (delimitadas pela região de relógio), com a inclusão dos DDSs cada partição passou a possuir 24 dessas colunas.

Embora a implementação do sistema tenha sido realizado através da mesma metodologia que no anterior experimento, nesta fase, como ainda não foi desenvolvida nenhuma ferramenta para esse fim, a inclusão dos DDSs foi realizada manualmente (tanto a nível do RTL como no ficheiro de *constraints*).

No final, como a FPGA usada pertence à família Virtex-6, cada RP ficou com as características indicadas na Tabela 20.

Tabela 20 – Características de cada RP

Parâmetros da RP	**Valores dos Parâmetros**
$Size_{Frame}$	324
$N_{CLBsConfFrames}$	12
N_{RPs}	3
$N_{RPFrames}$	1568 (1056 para CLBs + 512 para BRAMs)
$Frequency$	50MHz

Relativamente aos testes feitos, foram escritos dois programas com o objetivo de analisar o comportamento do DM e do DDS. O primeiro, para estudar o comportamento do DM e a medição diferencial do DDS encontra-se descrito no seguinte pseudo-código.

```
void main_application_DM_characterization(){
  ...
  init_counter ();
  while (1) {
    ...
    send_data(&AES3, input_data);
    get_data(&AES3, output_data[3]);
```

```
    if (read_counter() >= COUNTER_LIMIT) {
      DDS_measure(3, dds[3]);
      Get_Sysmonitor(&sysmon);
      printf ("Golden DM(RP3):%f, Module DM:%f", dds[3].dm_golden, dds[3].dm_module);
      printf ("Temperature:%f, Vccint:%f, Iccint:%f", sysmon.temperature, sysmon.vccint,
        sysmon.iccint);
      reset_counter();
    }
  }
}
```

Em suma, o processador só usa o módulo AES que está alocado na RP3 e ciclicamente (cerca de três vezes por segundo) são lidos os valores do DDS e alguns valores da primitiva *System Monitor* (a temperatura, a tensão de alimentação V_{CCINT} e a corrente I_{CCINT}). A opção da partição RP3 acontece por ser a que se encontra fisicamente mesmo junto da primitiva *System Monitor* dentro do dispositivo, obtendo assim o valor de temperatura mais rigoroso existente nesta partição. A utilização deste módulo é feito de uma forma moderada de modo a que não sobreaqueça muito a RP onde se encontra. No entanto, para que fosse mais rápida a observação da evolução da temperatura, a ventoinha de refrigeração acoplada à FPGA foi desligada durante o ensaio.

O segundo programa, orientado para medir diferenças de temperatura entre RPs, está exemplificado no próximo pseudo-código.

```
void main_application_DDS_characterization(){
  ...
  init_counter();
  count100 = 0;
  enable_AES_RP3 = 1;
  while (1){
    if (enable_AES_RP3) {
      send_data(&AES3, input_data);
      get_data(&AES3, output_data[3]);
    }

    if (read_counter() >= COUNTER_LIMIT) {
      DDS_measure(1, dds[1]);
      DDS_measure(3, dds[3]);
      Get_Sysmonitor(&sysmon);
      printf ("Golden DM(RP1):%f, Golden DM(RP3):%f", dds[1].dm_golden,
        dds[3].dm_golden);
      printf ("Temperature:%f, Vccint:%f, Iccint:%f", sysmon.temperature, sysmon.vccint,
        sysmon.iccint);
      count100++;
      if (count100 >= 100) {
        enable_AES_RP3 = (enable_AES_RP3++) & 0x01;
        count100 = 0;
      }
      reset_counter();
    }
```

```
  }
}
```

Neste algoritmo, o processador continua a comunicar apenas com o módulo alocado na RP3, agora de um modo intermitente, mas para além de ler os valores do DDS na RP3 e da primitiva *System Monitor*, também obtém os valores presentes no DDS existente na RP1. Por este meio demonstra-se que é possível monitorizar a temperatura em diferentes RPs, o que abre portas para também com esta informação minimizar o envelhecimento da FPGA devido ao NBTI.

6.4.1 Resultados Experimentais

Embora nos testes feitos com a plataforma implementada não tenha sido construído o mecanismo de TMR, as mesmas rotinas da Tabela 9 foram incluídas no sistema. Por essa razão, o acréscimo de memória associado a esta inclusão é o que se encontra discriminado nas primeiras duas linhas da Tabela 21, sendo o seu total, respeitando a equação 5.9, o valor na terceira linha.

Na quarta linha da Tabela 9 é indicado o número de ciclos despendidos pela rotina `DDS_measure(rp)` (NC_{DDS}), calculado por intermédio da equação 5.14, e nas restantes linhas é apresentado o número de ciclos de relógio NC_{Total} (equação 5.10a) para realizar uma troca entre duas RPs e o seu equivalente em segundos (T_{Swap}).

Tabela 21 – Memória e Número de Ciclos de Relógio consumidos pelas rotinas usadas.

Variáveis	Valores Calculados
M_{CODE}	11144 Bytes
M_{DATA}	660 Bytes
MEM_{Total}	11804 Bytes
NC_{DDS}	92760 Ciclos
NC_{Total}	24241280 Ciclos
T_{Swap}	0,412 Segundos

6.4.1.1 Caracterização do *Delay Meter*

Uma vez que os testes foram feitos numa FPGA não envelhecida, foi possível avaliar o DM como sensor de temperatura.

Executando o código referente ao primeiro programa e com a ventoinha responsável pela refrigeração da FPGA desligada, foi possível observar a evolução da temperatura no interior do dispositivo e a sua influência nos DMs de um DDS. Tirando vantagem da primitiva *System Monitor* foi possível correlacionar os valores obtidos através do DDS com os valores da temperatura, a tensão de alimentação V_{CCINT} e a corrente I_{CCINT} medidos pela primitiva. A primeira relação, presente na Figura 68, foi a caracterização de um *golden* DM.

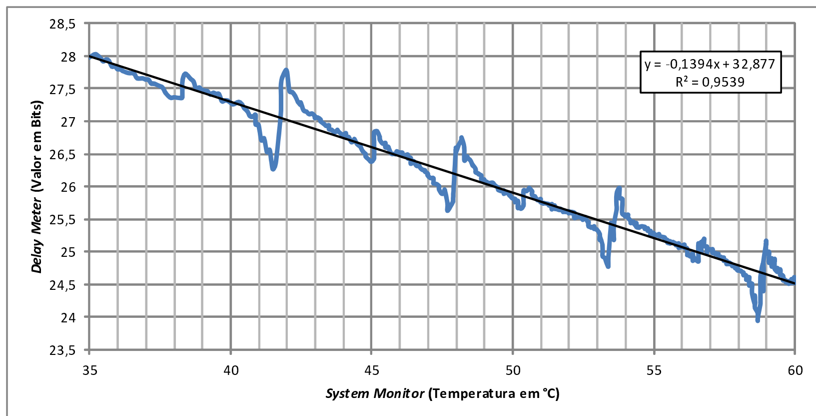


Figura 68 – Caracterização de um DM na Virtex-6 utilizada.

De modo a minimizar o risco de obter uma leitura errada, cada valor do *golden* DM presente na Figura 68, corresponde a uma média de 100 medições consecutivas realizadas pela rotina `DDS_measure(rp)` sempre que esta é chamada. O gráfico resultante foi de encontro ao deduzido na equação 5.12, sendo neste caso $m = 0,1394$ e $y_o = 32,877$.

Embora a tendência do gráfico seja linear e até possua um Coeficiente de Determinação $R^2 = 0,9539$, é igualmente observável que ao longo do incremento da temperatura o valor obtido do DM sofre fortes oscilações. Estas oscilações devem-se aos próprios recursos de

relógio da FPGA (Xilinx Inc., 2014c), que ao detetarem automaticamente que o sinal de relógio sofreu uma alteração devido ao aumento de temperatura, compensa essa alteração. No entanto, mesmo com este mecanismo existente no dispositivo, o valor lido por um DM mantém-se tendencialmente decrescente com o aumento da temperatura, seguindo uma linha linear. Ou seja, após esta caracterização, é possível usar este DM também como sensor de temperatura.

6.4.1.2 Caracterização do *Differential Delay Sensor*

Validada a utilidade de um DM como sensor de temperatura (considerando constante a tensão de alimentação VDD e o *aging*), foi necessário validar se um par destes sensores consegue realmente medir o envelhecimento dos recursos da FPGA. Usando os mesmos dados obtidos pelo teste anterior (resultantes do primeiro código) foi desenhado um primeiro gráfico com a evolução dos valores de ambos os DMs que constituem o mesmo DDS (*Golden* DM e *Module* DM), seguindo-se o esboço de um gráfico de barras com a indicação do resultado $GoldenDM - ModuleDM$. Ambos os gráficos encontram-se na Figura 69.

É visível que embora com uma ligeira diferença, em média inferior a meio bit (positivo ou negativo), os dois DMs sempre se acompanham, inclusive quando acontecem as fortes oscilações originadas pelos recursos de relógio.

Devido à conjugação de fatores como falta de tempo neste ponto da tese, falta de meios técnicos e a falta de recursos (FPGAs para sacrificar ao envelhecimento), não foi possível observar o DDS a medir uma diferença $GoldenDM - ModuleDM$ superior a um bit, de modo a que fosse possível comprovar a sua eficácia.

6.4.1.3 Influência da Temperatura na Corrente e Tensão de Alimentação da FPGA

Aproveitando os mesmos dados obtidos através da execução do primeiro código, foi analisada a interferência da variação da temperatura

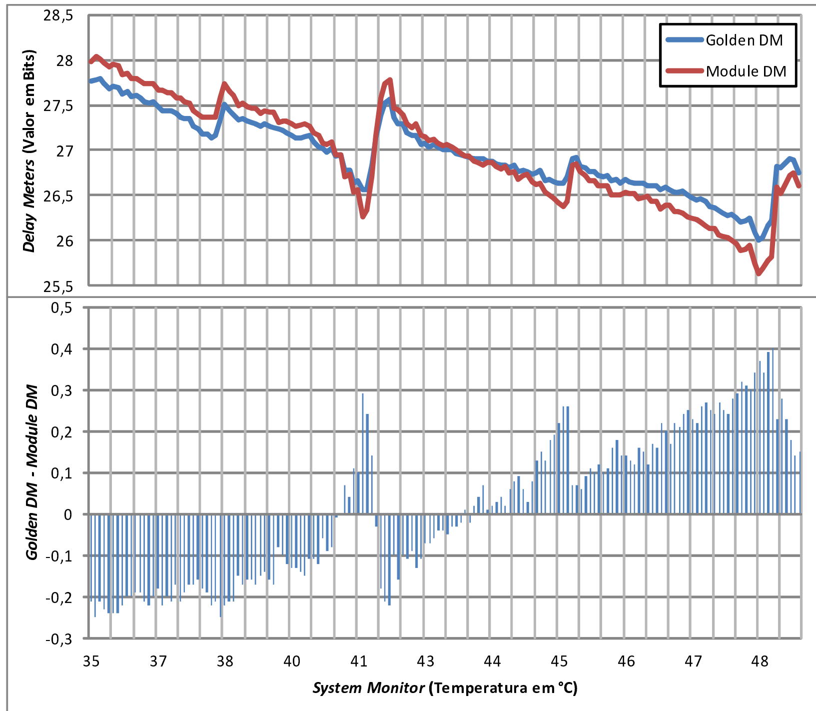


Figura 69 – Análise do ruído/erro na leitura do DDS.

no valor da corrente I_{CCINT} que percorre a FPGA. A relação entre estas duas grandezas originou o gráfico da Figura 70.

Quando um módulo encontra-se em funcionamento, a comutação dos transístores usados da respetiva RP implica uma dissipação de potencia junto desses mesmos transístores. Esta dissipação provoca um aumento de temperatura que necessita de ser combatida por meio de algum mecanismo de refrigeração. Não sendo controlada a temperatura, esta irá incrementar os tempos de atraso dos mesmos transístores, o que por consequência provoca um aumento da corrente durante a transição dos seus estados. Este aumento de corrente implica uma nova subida na dissipação de potência, criando assim um ciclo vicioso.

É bem visível que o aumento da temperatura implica igualmente um aumento na corrente que circula pelo dispositivo. Entre os $35°C$ e

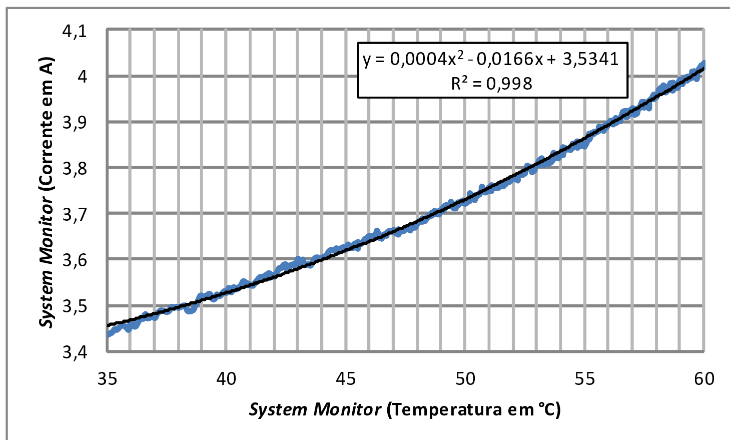


Figura 70 – Influência da temperatura na corrente I_{CCINT}.

os 60°C o incremento é de quase 250 mA por cada 10°C, sendo que esse incremento tem tendência a aumentar ainda mais com o aumento da temperatura.

A mesma relação foi realizada entre a temperatura e a tensão V_{CCINT} que alimenta a FPGA, e deu origem ao gráfico da Figura 71.

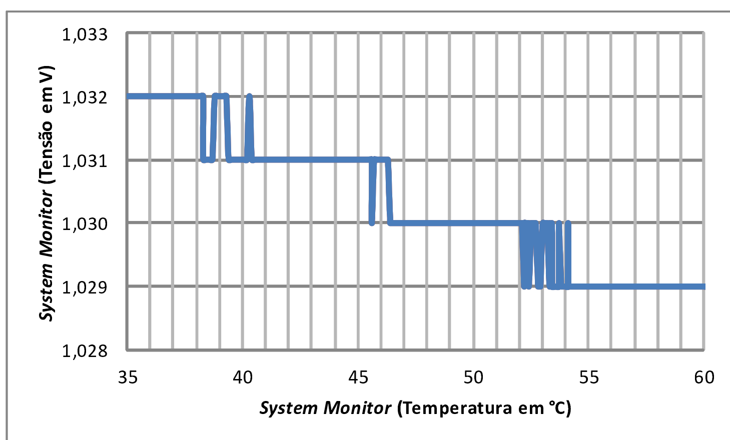


Figura 71 – Influência da temperatura na tensão de alimentação V_{CCINT}.

Em sentido diferente do comportamento da corrente I_{CCINT}, a tensão de alimentação decresce com a temperatura. Com um decremento de cerca de 1 mV por cada 10°C, embora se note que a temperatura influencie esta tensão, não tem a mesma preponderância como a que tem sobre a corrente.

Multiplicando a tensão pela corrente, foi igualmente analisada a evolução do consumo de potência com o aumento da temperatura. A Figura 72 mostra o gráfico resultante.

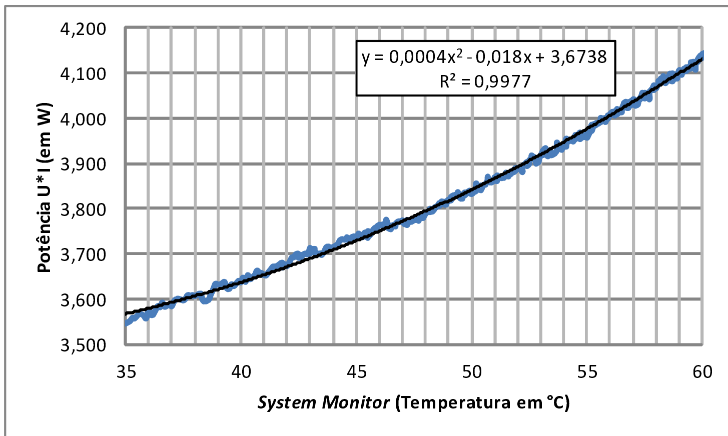


Figura 72 – Influência da temperatura na potência consumida $V_{CCINT}.I_{CCINT}$.

Com uma influência devido à temperatura semelhante à corrente I_{CCINT}, o consumo de potência aumenta cerca de 250 mW por cada 10°C (no intervalo entre 35°C e os 60°C). Ou seja, após uma subida de 25°C na temperatura, o consumo de potência no dispositivo aumentou 16%.

6.4.1.4 Uso do DDS como Sensor de Temperatura numa Partição da FPGA

Correndo o código do segundo programa, onde são usados apenas os *golden* DMs dos DDSs existentes nas partições RP1 e R2, foi analisada a possibilidade de usar o DDS também como sensor de tempe-

ratura intra-partição. Os resultados registados permitiram desenhar os gráficos da Figura 73.

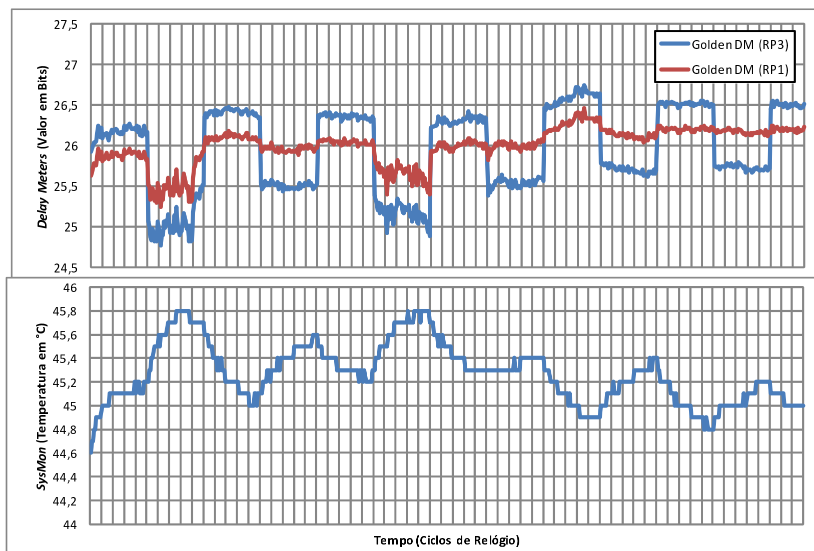


Figura 73 – Medição da temperatura recorrendo ao *golden* DM existente no DDS.

Na primeira metade do ensaio, a ventoinha de refrigeração que se encontra por cima da FPGA encontrava-se desligada. Na segunda metade esta foi ligada. Tal diferença é visível de duas formas: as oscilações da temperatura lidas pela primitiva *System Monitor* (*SysMon*) atingem valores superiores na primeira metade; e o valor recolhido do *golden* DM da RP1 sofre oscilações com maior amplitude também na primeira metade. Ou seja, sem refrigeração ativa, o dispositivo não só aquece mais, como esse calor é em parte transferido para o resto da FPGA, afetando principalmente as áreas adjacentes.

Mesmo com a refrigeração ativa, o *SysMon* mantém sempre alguma oscilação, o que se justifica, como já mencionado, pelo facto da partição RP3 se encontrar fisicamente junto da primitiva *System Monitor*, o que acaba por ser sempre muito influenciada pelo aquecimento que ocorre nesta partição.

Com o intuito de poder comparar a temperatura entre RP1 e RP3, e em simultâneo comprovar que a caracterização registada na Figura 68 é válida, a equação da linha de tendência $y_{(x)} = -0,1394x + 32,877$ foi convertida na equação 5.13 apresentada na Secção 5.4.1.4. A equação resultante foi: $x_{(y)} = \frac{32{,}877 - y}{0{,}1394}$. Aplicando esta equação aos dados da Figura 73, o resultado foi o gráfico da Figura 74.

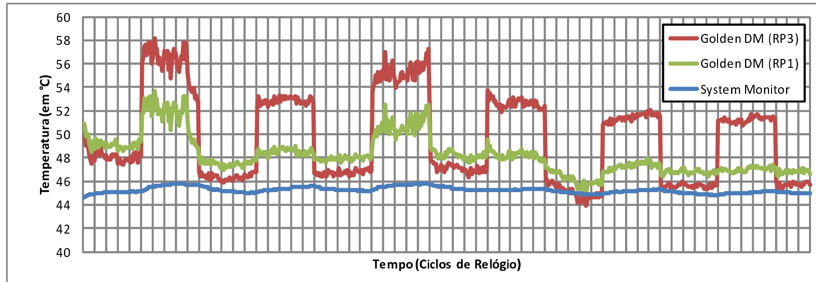


Figura 74 – Conversão dos valores do *golden* DM em temperatura.

Neste novo gráfico é possível observar e comparar as temperaturas na RP1, RP3 e a medida pelo *SysMon*. De novo se observa a diferença entre a primeira e a segunda parte. Na segunda parte, com refrigeração, as variações de temperatura na partição RP3 têm a amplitude mais atenuada, e as variações de temperatura na partição RP1 ficam quase sem sofrer interferência das alterações que ocorrem na partição RP3.

O algoritmo do segundo programa, ao provocar uma forte atividade intermitente no módulo alocado na partição RP3, provoca igualmente uma alternância entre dois patamares de temperaturas. Sem refrigeração ativa, alterna entre $48°C$ e $56°C$, e com refrigeração ativa alterna entre sensivelmente $45°C$ e $51°C$.

6.4.2 Avaliação dos Resultados Obtidos

Os resultados obtidos foram praticamente todos ao encontro ao esperado. Em relação ao DM, embora com algum erro e uma fraca resolução (superior a $1°C$), é possível ser usado como sensor de temperatura local dentro da FPGA. Esta qualidade reduzida (em relação a erro

e resolução), deve-se em grande parte aos recursos de relógio existentes na FPGA, que ao compensarem o atraso do sinal de relógio devido ao próprio aquecimento, acabam por provocar as oscilações visíveis na Figura 68. No entanto, no que ao sensor DDS diz respeito, até mesmo em relação a estas oscilações este demonstrou-se imune. Isto porque o mesmo fenómeno sempre ocorre em ambos os DM que compõem o DDS, e como a leitura deste sensor funciona calculando a diferença entre DMs, também a oscilação causada por este fenómeno é anulada (tal como acontece em relação à variação da temperatura ou tensão de alimentação).

Da análise dos resultados obtidos, o menos esperado foi a forte influência que a temperatura tem sobre a intensidade de corrente que percorre a FPGA, o que por sua vez influencia o consumo de potência por parte do dispositivo. No experimento realizado, como mostra a Figura 72, o consumo de potência subiu cerca de 250 mW por cada $10°C$.

Embora nenhum dos testes tenha permitido validar a ideia original do DDS, medir o envelhecimento devido ao NBTI (e eventualmente também do PBTI), por outro lado, o teste reportado nos gráficos da Figura 73 provou que usando apenas um DM de cada DDS que existe em cada um dos módulos alocados, é possível monitorizar a temperatura em cada uma das RPs. Tal funcionalidade pode permitir o sistema velar pelas RPs, combatendo o seu excessivo aquecimento, algo que a ferramenta *AgingCalc* na Secção 5.4.1.3 demonstrou que acelerava o *aging* devido ao NBTI. Para além disso, como referenciado no parágrafo anterior, o aquecimento provoca também um aumento no consumo de potência, o que é igualmente um fenómeno a combater.

6.4.3 Conclusões e Trabalho Futuro

Este duplo experimento teve como objetivo avaliar e validar o sensor de performance e envelhecimento desenvolvido (*Differential Delay Sensor*). A principal inovação deste sensor é usar e comparar o atraso em duas linhas de recursos da FPGA (DMs), uma que é configurada de

modo a evitar envelhecimento dos recursos devido ao NBTI, e outra que é continuamente sujeita aos efeitos do NBTI. Esta técnica permite medir a diferença entre os atrasos registados por ambos os DMs, reduzindo assim interferência da temperatura ou da tensão de alimentação, conseguindo assim monitorizar a performance e eventualmente a evolução dos efeitos do NBTI nos recursos constantemente usados.

A inclusão deste novo sensor DDS na metodologia desenvolvida que permite o PB4MP, incrementa ainda mais o combate à ocorrência de faltas originadas pelo envelhecimento dos recursos num sistema implementado numa FPGA. Assim, o objetivo não é só providenciar a capacidade de recuperação de faltas num sistema, mas também tentar a prevenção de faltas permanentes causadas pelo envelhecimento.

Embora não tenha sido completamente validada a capacidade de medir a evolução do *aging* nos recursos do dispositivo, o DDS é apenas mais um novo passo de modo a agregar mais valor à metodologia desenvolvida, onde o objetivo primordial é um sistema implementado numa FPGA recuperar de pelo menos uma falta permanente. Mas mesmo no cenário em que após um teste real, com o envelhecimento de várias FPGAs, o DDS demonstre que não consegue desempenhar a sua principal função de medir o *aging*, o facto de conseguir medir a temperatura em cada RP, e sabendo que o aumento desta grandeza está relacionada com um aumento de consumo de potência, não deixa de ser um sensor bastante útil para combater o aceleramento do envelhecimento devido ao NBTI.

Como trabalho futuro, com o que já se encontra validado em relação ao DDS, será possível através de uma análise prévia, atribuir um nível de temperatura e consumo de potência a cada módulo existente no sistema. Com essa informação, para além das estratégias **A** e **B** ilustradas na Figura 67, o sistema poderá ainda decidir a RP onde irá realocar um determinado módulo, de modo a evitar concentrações de calor no dispositivo, o que não só reduz a velocidade de envelhecimento dos recursos sobre grande stress térmico, como ao manter a temperatura geral da FPGA mais baixa, reduz o consumo de potência por parte do sistema.

6.5 CONCLUSÕES GERAIS

O foco inicial (e principal) desta tese foi dotar um sistema implementado numa FPGA da capacidade de recuperar o seu normal funcionamento após ocorrer uma falta permanente. Na Secção 6.2 é validada a capacidade de realocação múltipla (PB4MP) que permite a movimentação de módulos entre várias RPs sem necessidade de uma extensa biblioteca de *bitstreams* parciais. No estágio seguinte, a Secção 6.3, é demonstrado como com uma política de seleção dos recursos não utilizados em cada RP é possível mascarar pelo menos uma falta permanente sem obrigar a exclusão da RP onde é detetada a falta. Ainda nesta secção é adotada uma arquitetura TMR que para além de ocultar falhas que ocorram num dos três módulos, o que aumenta a confiabilidade do sistema, habilita o próprio sistema saber em que RP aconteceu uma falta que originou a falha que foi ocultada.

Para além do objetivo inicial, que permite dotar o sistema com um maior ciclo de vida, colateralmente, o seu desenvolvimento acabou por proporcionar vantagens extras. Assim, para além de ser possível o sistema ultrapassar a ocorrência de pelo menos uma falta permanente, este também pode rodar o uso dos recursos existentes em cada RP com o objetivo de que o envelhecimento desses recursos aconteça de uma forma uniforme (em vez de determinados recursos envelhecerem mais rapidamente que outros), realizando assim uma prevenção às faltas causadas pelo envelhecimento causado pelo NBTI. Com a adição do sensor DDS, Secção 6.4, o sistema pode mesmo observar a performance de cada módulo e poderá dessa informação obter uma previsão do envelhecimento na RP onde se encontra alocado. Por limitações de várias ordens, a verificação e a quantificação da influência do envelhecimento nos recursos do dispositivo foram apenas realizado por simulação. Significa isto que é acima de tudo uma previsão e que certamente terá desvios em relação à realidade. A devida validação através da aceleração do envelhecimento de várias FPGAs (ex: sujeitando-as as temperaturas ou tensões de alimentação mais elevadas que provoquem essa aceleração), é por isso uma etapa natural como trabalho futuro. No entanto,

relativamente ao DM usado para construir o DDS, este foi devidamente caracterizado e mostrou que no pior caso, pode ser usado como sensor de temperatura local (no interior de um RP). Essa funcionalidade poderá ser igualmente útil numa outra etapa de trabalho no futuro, onde por exemplo se pretenda escalonar os módulos em função da potência que consomem, de forma a distribuir o aquecimento por toda a FPGA. Uma abordagem deste tipo poderá reduzir a temperatura média do dispositivo e ao mesmo tempo evitar que determinada secção de recursos acelere o seu envelhecimento em relação os restantes, o que poderá com o tempo dar origem a *delay faults* nessa mesma secção.

Embora as ferramentas tenham sido desenvolvidas em software, o que não garante qualquer imunidade em relação a erros durante a sua criação, grande parte da função delas é analisar informação e gerar pacotes de *constraints*. Essas *constraints* são depois usadas para influenciar as ferramentas disponibilizadas pelo fabricante, tal como mostra o diagrama da Figura 75.

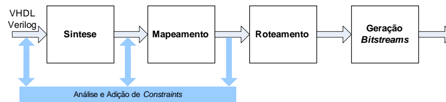


Figura 75 – Fluxo normal da ferramentas influenciado através de *constraints*.

Assim, e considerando que o controlo de qualidade no desenvolvimento das ferramentas por parte do fabricante é elevado e implicou a validação das mesmas, então parte do fluxo das ferramentas encontra-se logo à partida validado. Relativamente a erros causados devido à geração errada de *constraints*, o fluxo IDF apresentado na Secção 3.4.2 inclui *scripts* que no final do roteamento verificam se cada RP se encontra devidamente isolada do resto do sistema, tolerando que apenas os sinais que interligam o módulo ao sistema possam cruzar a fronteira entre cada RP e o restante do sistema. Os erros mais potenciais de ocorrer poderão ser nas *constraints* que garantem a compatibilidade do interface dos vários módulos, para que estes possam ser realocados em qualquer

RP, mas optando pelo TMR, um sinal de um endereço ou dado que não possua o mesmo roteamento relativo implicará logo num primeiro momento uma avalanche de falhas nesse mesmo módulo. Significa tudo isto, que embora não exista a garantia de ausência de *bugs* em todos os passos que interferem com o normal fluxo das ferramentas, todo o ambiente associado e o resultado produzido proporcionam um grau de confiança aceitável por parte do autor.

7 CONCLUSÃO E TRABALHOS FUTUROS

Ao longo das várias secções deste documento foram escritas diversas conclusões e mencionadas várias direções para novas iterações em trabalhos futuros. No entanto é importante realçar determinados pontos chave caracterizam todo o conjunto do trabalho descrito.

Todo o projeto associado a esta tese trouxe diretamente e indiretamente um incremento importante de conhecimento para o todo o universo que envolveu o desenvolvimento do mesmo. Numa aplicação mais direta, o plano de fundo deste trabalho é aumentar o grau de confiança sobre o uso de FPGAs comerciais em aplicações espaciais. O dotar um sistema implementado numa FPGA da capacidade de ultrapassar a ocorrência de faltas permanentes (relativamente a faltas transitórias já existem soluções), e ao mesmo tempo minimizar a evolução do seu envelhecimento, são incontornavelmente áreas importantes para o grupo onde existe uma equipa dedicada a missões espaciais como o Floripa-Sat. É por isso um projeto que contribui diretamente para o Grupo de Sistemas Embarcados (GSE) do qual o autor faz parte.

Numa perspetiva mais centrada no autor, todo o desenvolvimento implicou um forte acréscimo científico nas áreas associadas e um enriquecimento técnico relativo aos detalhes de funcionamento das FPGAs. Ao contrário do conhecimento científico que passou pelo estudo do estado da arte existente e disponível nas tradicionais fontes de informação da área, grande parte da informação técnica passou por cruzamento da informação disponível no website do fabricante, ou mesmo através de experimentos onde foi realizada alguma engenharia inversa.

Outra importante consequência de todo o desenvolvimento realizado, foi que várias tarefas foram feitas com interação com o Laboratório de Integração de Software e Hardware (LISHA), também pertencente à UFSC, com o grupo *Electronic System Design and Automation* (ESDA), pertencente ao INESC-ID em Lisboa, e com Laboratório de Excelência em Eletrônica, Automação e Sistemas Embarcados de Alta Confiabilidade (LABEASE) em Porto Alegre, o que constituiu uma contribuição orientada para a criação da rede científica.

Relativamente à estrutura geral apresentada na Secção 4.4, embora a proposta proporcionasse uma maior tolerância a faltas permanentes, não garantia que quando as faltas ocorressem não originassem falhas no sistema. Ou seja, garantia-se que o sistema iria voltar a ter a sua integridade original, mas durante o processo de recuperação do sistema aquando da ocorrência de uma falta, o sistema poderia ficar pontualmente instável. No entanto, no desenvolvimento da proposta existiu algum desvio face ao planeamento original. Procurando o máximo de confiabilidade de um sistema, optou-se por uma arquitetura com TMR (Secção 5.3) que para além de garantir que uma falta num módulo não provoca uma falha no sistema, permite em simultâneo detetar em que partição ocorreu a falta que levou o módulo falhar. Devido a esta opção, o Mecanismo de Deteção (Secção 4.4.2) passou a estar implícito no próprio algoritmo da arquitetura com TMR, deixando de existir a necessidade de realizar um teste periódico a cada par módulo/partição. Para além da alteração da própria arquitetura global, esta opção garante que todas as faltas não transparentes são detetadas, inclusive as que são devidas ao envelhecimento que provocam o aumento do atraso na resposta dos recursos da FPGA (*delay faults*). No entanto, é de ressalvar que embora o TMR permita detetar e mascarar uma falha que ocorra num dos módulos, na validação realizada, a votação é efetuada pelo microprocessador que não está imune a faltas e que por isso continua desprotegido. Por esse motivo, a função de votação pode ser igualmente implementada em hardware num módulo exclusivo, podendo mesmo nem ser implementado na mesma FPGA. Outra forma, é dotar o microprocessador de tolerância a faltas é uma tarefa prioritária para o futuro e que se encontra a ser estudada no âmbito de outra tese de doutoramento por um aluno pertencente ao mesmo grupo.

Relembrando que os recursos de uma FPGA SRAM são configurados e interligados através do conteúdo de uma memória de configuração interna, significa isto que tanto os recursos quanto a memória que os configura podem sofrer uma falta permanente. Em qualquer dos casos, o módulo que esteja a usar ativamente esses recursos faltosos irá reportar uma falha e o seu tratamento é realizado do mesmo modo,

independentemente se a falta é no recursos ou na respetiva memória.

Realizando esta divisão da FPGA em recursos e memória de configuração, também será importante no futuro analisar não só o efeito do envelhecimento devido ao BTI nos recursos (incluindo os usados para o roteamento), como também avaliar o que este efeito pode causar na células que compõem a memória de configuração. Com esse conhecimento, dados como os que compõem a Tabela 19 poderão ter uma importância reforçada na hora de prever e planear a adequada política de rotação de recursos para mitigar o *aging* causado por alguma forma de BTI.

Embora o objetivo principal e original fosse um sistema recuperar de pelo menos uma falta permanente, devido à flexibilidade fornecida pela metodologia desenvolvida, novos objetivos foram emergindo. Por essa mesma razão acredita-se que a tese realizada poderá continuar a contribuir em futuros trabalhos que desejem explorar a mobilidade de módulos entre RPs permitida pelo mecanismo PB4MP.

Ainda como trabalho futuro e a pensar em famílias recentes de FPGAs como a Zynq (baseada na arquitetura da 7 Series), será importante migrar o fluxo que permite o PB4MP para integrá-lo no Vivado, a nova ferramenta base da Xilinx. Um sistema implementado numa Zynq, um processo mais fácil de realizar através do Vivado, permite utilizar os processadores físicos ARM (Inc., 2016) como CPU central e os recursos programáveis como RPs. Embora esta plataforma continuasse a poder sofrer faltas nos recursos associados ao CPU, como não é *softcore*, ficaria imune a *bit flips* que pudessem alterar a memória de configuração responsável pela sua implementação. Para além deste fluxo responsável pela geração do hardware, também será importante automatizar a integração do software responsável pelo controlo do mecanismo desenvolvido. Talvez o caminho seja optar por um determinado tipo de aplicações (preferencialmente usadas em missões espaciais), onde são identificados os parâmetros que influenciam a implementação do sistema e alcançar um *framework* para essa finalidade.

Numa visão mais futurista, se a granularidade de um módulo evoluir ao ponto de incluir igualmente um microprocessador e que este pode aceder à memória de configuração responsável pela implementação

de ele próprio, significa que o microprocessador poderá eventualmente conseguir auto replicar-se. Dito noutros termos, poderá ser possível evoluir para o equivalente a um *fork()* em hardware.

Por último, e justificando o título desta tese, todo este desenvolvimento foi orientado para que o uso de FPGAs SRAM comerciais possam ser uma real opção em aplicações espaciais. Opção esta que transporta consigo não só a flexibilidade inerente a este tipo de dispositivos como a sua fácil disponibilidade comercial e o seu menor custo. Nesse sentido foi já detalhado o experimento (Martins et al., 2016) a realizar numa FPGA incluída num CubeSat e que se enquadra em mais um importante trabalho futuro.

REFERÊNCIAS

Abramovici et al., 1994 Abramovici, M., Breuer, M. A., and Friedman, A. D. (1994). *Digital Systems Testing and Testable Design*. Wiley-IEEE Press, New York. 78

Agarwal et al., 2008 Agarwal, M., Balakrishnan, V., Bhuyan, A., Kim, K., Paul, B., Wang, W., Yang, B., Cao, Y., and Mitra, S. (2008). Optimized circuit failure prediction for aging: Practicality and promise. In *Proceedings of the IEEE International Test Conferenc (ITC)*, pages 1–10, Santa Clara, USA. 84

Alam and Mahapatra, 2005 Alam, M. A. and Mahapatra, S. (2005). A comprehensive model of pmos nbti degradation. *Microelectronics Reliability*, 45(1):71–81. 82, 83

Alam and Mahapatra, 2007 Alam, M. A. and Mahapatra, S. (2007). A comprehensive model for pmos nbti degradation: Recent progress. *Microelectronics Reliability*, 47(6):853–862. 152

Alexandrescu et al., 2004 Alexandrescu, D., Anghel, L., and Nicolaidis, M. (2004). Simulating single event transients in vdsm ics for ground level radiation. *Proceedings of Journal of Electronic Testing: Theory and Application*, 20(4):413–421. 74, 93

Alfke and Padovani, 2004 Alfke, P. and Padovani, R. (2004). Radiation tolerance of high-density fpgas. <http://japan.xilinx.com/esp/mil_aero/collateral/RadiationEffects/radiation_tolerance.pdf>. 40

(ASU), 2012 (ASU), A. S. U. (2012). Predictive technology model (ptm) website. <http://ptm.asu.edu/>. 205

Avizienis et al., 2004 Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. In *Proceedings of the IEEE Transactions on Dependable and Secure Computing*, pages 11–33. 37, 39, 47, 71, 78, 199

Backasch et al., 2014 Backasch, R., Hempel, G., Werner, S., Groppe, S., and Pionteck, T. (2014). Identifying homogenous reconfigurable regions in heterogeneous fpgas for module relocation. In *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGAs (ReConFig '14)*, pages 1–6, Cancun, Mexico. 91, 133, 215

Beckhoff et al., 2012 Beckhoff, C., Koch, D., and Torresen, J. (2012). Goahead: A partial reconfiguration framework. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44, Toronto, Canada. 44, 87, 133, 215

Beckhoff et al., 2013a Beckhoff, C., Koch, D., and Torresen, J. (2013a). Automatic floorplanning and interface synthesis of island style reconfigurable systems with goahead. *Architecture of Computing Systems – ARCS 2013*, 7767:303–316. 87, 133, 215

Beckhoff et al., 2013b Beckhoff, C., Wold, A., Fritzell, A., Koch, D., and Torresen, J. (2013b). Building partial systems with goahead. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, page 1, Porto, Portugal. 87

Benfica et al., 2012 Benfica, J., Poehls, L. M. B., Vargas, F., Lipovetzky, J., Lutenberg, A., García, S. E., Gatti, E., and Hernandez, F. (2012). Evaluating the effects of combined total ionizing dose radiation and electromagnetic interference. *IEEE Transactions on Nuclear Science*, 59(4):1015–1019. 80, 82, 199

Benfica et al., 2011 Benfica, J., Poehls, L. M. B., Vargas, F., Lipovetzky, J., Lutenberg, A., García, S. E., Gatti, E., Hernandez, F., and Calazans, N. L. V. (2011). Evaluating the use of a platform for combined tests of total ionizing dose radiation and electromagnetic immunity. In *Proceedings of the Radiation and Its Effects on Components and Systems (RADECS)*, pages 473–478, Sevilla, Spain. 14, 80, 81

Berejuck, 2011 Berejuck, M. D. (2011). Dynamic Reconfiguration Support for FPGA-based Real-time Systems. Technical report, Federal University of Santa Catarina, Florianópolis, Brazil. PhD qualifying report. 224

Bolchini et al., 2011a Bolchini, C., Miele, A., and Sandionigi, C. (2011a). Automated resource-aware floorplanning of reconfigurable areas in partially-reconfigurable fpga systems. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 532–538, Chania, Greece. 103

Bolchini et al., 2011b Bolchini, C., Miele, A., and Sandionigi, C. (2011b). A novel design methodology for implementing reliability-aware

systems on sram-based fpgas. *IEEE Transactions on Computers*, 60(12):1744–1758. 103

Bolchini et al., 2012 Bolchini, C., Miele, A., and Sandionigi, C. (2012). Increasing autonomous fault-tolerant fpga-based systems' lifetime. In *Proceedings of the 17th IEEE European Test Symposium (ETS)*, pages 1–6, Annecy, France. 14, 103, 105, 133, 155, 157, 193, 215

Bolchini et al., 2011c Bolchini, C., Sandionigi, C., Fossati, L., and Codinachs, D. M. (2011c). A reliable fault classifier for dependable systems on sram-based fpgas. In *Proceedings of the IEEE On-Line Testing Symposium (IOLTS)*, pages 92–97, Athens, Greece. 103

Boudenot, 2007 Boudenot, J.-C. (2007). Radiation space environment. *Radiation Effects on Embedded Systems*, pages 1–9. 79

Campos et al., 2013 Campos, P. B., Lawson, D. M. R., Bale, S. J., Walker, J. A., Trefzer, M. A., and Tyrrell, A. M. (2013). Overcoming faults using evolution on the panda architecture. In *Proceedings of the IEEE International Congress on Evolutionary Computation (CEC)*, pages 613–620, Cancun, Mexico. 111

Carmichael et al., 2000 Carmichael, C., Caffrey, M., Salazar, A., and Laboratories, L. A. N. (2000). Correcting single-event upsets through virtex partial configuration v1.0 - Xilinx inc. inc. application note (xapp216). <http://www.xilinx.com/support/documentation/application_notes/xapp216.pdf>. 41, 76

Carmichael and Tseng, 2009 Carmichael, C. and Tseng, C. W. (2009). Correcting single-event upsets in virtex-4 fpga configuration memory v1.0 - Xilinx inc. inc. application note (xapp1088). <http://www.xilinx.com/support/documentation/application_notes/xapp1088.pdf>. 41, 76, 161

Ceratti et al., 2012 Ceratti, A., Copetti, T., Bolzani, L., and Vargas, F. (2012). Investigating the use of an on-chip sensor to monitor nbti effect in sram. In *Proceedings of the IEEE 13th Latin America Test Workshop (LATW)*, pages 1–6, Quito, Ecuador. 82, 85, 152, 199

Chakravarthi et al., 2004 Chakravarthi, S., Krishnan, A., Reddy, V., Machala, C., and Krishnan, S. (2004). A comprehensive framework for predictive modeling of negative bias temperature instability. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS)*, pages 273–282, Monterey, USA. 84

Cong-Vinh, 2009 Cong-Vinh, P. (2009). *Dynamic Reconfigurability in Reconfigurable Computing Systems: Formal Aspects of Computing*. VDM Verlag Dr. Müller, USA. 58

Dehon, 2005 Dehon, A. (2005). Nanowire-based programmable architectures. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 1(2):109–162. 53

dos Santos Pachito, 2012 dos Santos Pachito, J. (2012). Aging Prediction Methodology for Digital Circuits. Technical report, University of Algarve, Faro, Portugal. MSc Thesis, <http://w3.ualg.pt/~jsemiao/port/_files/Jackson_thesis_final.pdf>. 204, 205

Drahonovský et al., 2013 Drahonovský, T., Rozkovec, M., and Novák, O. (2013). Relocation of reconfigurable modules on xilinx fpga. In *Proceedings of the IEEE Annual International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 175–180, Karlovy Vary, Czech Republic. 44, 88, 89, 133, 184, 189, 215

Dumitriu and Kirischian, 2010 Dumitriu, V. and Kirischian, L. (2010). A framework of embedded reconfigurable systems based on re-locatable virtual components. *International Journal of Embedded Systems*, 4(3/4):182–194. 15, 42, 114, 119, 120, 131, 133, 215

Dumitriu and Kirischian, 2013 Dumitriu, V. and Kirischian, L. (2013). Soc self-integration mechanism for dynamic reconfigurable systems based on collaborative macro-function units. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '13)*, pages 1–7, Cancun, Mexico. 15, 42, 43, 114, 123, 124

Dumitriu et al., 2012 Dumitriu, V., Kirischian, L., and Kirischain, V. (2012). A framework for adaptive reconfigurable space-borne computing platforms for run-time self-recovery from transient and permanent hardware faults. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 280–287, Erlangen, Germany. 15, 38, 42, 114, 121, 122

Dumitriu et al., 2014 Dumitriu, V., Kirischian, L., and Kirischain, V. (2014). Decentralized run-time recovery mechanism for transient and permanent hardware faults for space-borne fpga-based computing systems. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 47–54, Leicester, England. 15, 42, 114, 125, 126, 130, 133, 215

Dumitriu et al., 2015a Dumitriu, V., Kirischian, L., and Kirischain, V. (2015a). Mitigation of variations in environmental conditions by sopc architecture adaptation. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 1–8, Montreal, Canada. 15, 42, 114, 127, 128, 129, 132, 133, 135, 215

Dumitriu et al., 2015b Dumitriu, V., Kirischian, L., and Kirischain, V. (2015b). Run-time recovery mechanism for transient and permanent hardware faults based on distributed, self-organized dynamic partially reconfigurable systems. *IEEE Transactions on Computers*, PP(89):1–14. 15, 42, 114, 129, 130, 133, 157, 193, 215

Dutton and Stroud, 2009a Dutton, B. F. and Stroud, C. E. (2009a). Built-in self-test of configurable logic blocks in virtex-5 fpgas. In *Proceedings of the 41st Southeastern Symposium System Theory (SSST)*, pages 230–234, Tullahoma, Tennessee. 72

Dutton and Stroud, 2009b Dutton, B. F. and Stroud, C. E. (2009b). Soft core embedded processor based built-in self-test of fpgas. In *Proceedings of the Defect and Fault Tolerance in VLSI Systems(DFT '09). 24th IEEE International Symposium*, pages 29–37, Chicago, Illinois. 72

Espinosa et al., 2012 Espinosa, J., de Andres, D., Ruiz, J. C., and Gil, P. (2012). Tolerating multiple faults with proximate manifestations in fpga-based critical designs for harsh environments. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 292–299, Oslo, Norway. 14, 108, 109, 133, 155, 157, 193, 215

Ferlini, 2012 Ferlini, F. (2012). Plaeser - plataforma de emulação de soft errors visando a análise experimental de técnicas de tolerância a falhas: Uma prototipação rápida utilizando fpgas. Master's thesis, Universidade Federal de Santa Catarina. 277

Flynn et al., 2009 Flynn, A., Gordon-Ross, A., and George, A. D. (2009). Bitstream relocation with local clock domains for partially reconfigurable fpgas. In *Proceedings of the IEEE Annual International Conference on Design, Automation and Test in Europe (DATE)*, pages 300–303, Nice, France. 90, 133, 215

for Semiconductors (ITRS), 2011 for Semiconductors (ITRS), I. T. R. (2011). Itrs reports and ordering information. <http://www.itrs.net/Links/2011ITRS/Home2011.htm>. 78

for Space Standardization (ECSS), 2008 for Space Standardization (ECSS), E. C. (2008). Ecss-e-st-10-12c space engineering. <http://www.ecss.nl/forums/ecss/dispatch.cgi/standards/showFile/100701/d20081115091445/No/ECSS-E-ST-10-12C(2815November2008)29.pdf>. 79

Gericota, 2003 Gericota, M. G. O. (2003). Metodologias de teste para fpgas (field programmable gate arrays) integradas em sistemas reconfiguráveis. In *PhD Thesis, Faculdade de Engenharia da Universidade do Porto*, Porto, Portugal. 14, 93, 95, 133, 215

Gericota et al., 2001 Gericota, M. G. O., Alves, G. R., Silva, M. L., and Ferreira, J. M. (2001). Draft: an on-line fault detection method for dynamic and partially reconfigurable fpgas. In *Proceedings of the On-Line Testing Workshop (IOLTW), Seventh International*, pages 34–36, Taormina, Italy. 41, 93

Gericota et al., 2002 Gericota, M. G. O., Alves, G. R., Silva, M. L., and Ferreira, J. M. (2002). A novel methodology for the concurrent test of partial and dynamically reconfigurable sram-based fpgas. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Paris, France. 41, 93

Gokhale and Graham, 2005 Gokhale, M. B. and Graham, P. S. (2005). *Reconfigurable Computing - Accelerating Computation with Field-Programmable Gate Arrays*. Springer, Berlin. 37, 40, 51

Guiotto et al., 2003 Guiotto, A., Martelli, A., and Paccagninis, C. (2003). Smart-fdir: use of artificial intelligence in the implementation of a satellite fdir. <ftp://ftp.estec.esa.nl/pub/wm/anonymous/wme/Web/SmartFDIR2003.pdf>. Projecto coordenado por Alenia Spazio (ALS) com participação Politecnico di Milano (POLIMI). 71, 161, 194

Hallett, 2015 Hallett, E. (2015). Isolation design flow for xilinx 7 series fpgas or zynq-7000 ap socs (ise tools) v1.3.1 (xapp1086). <http://www.xilinx.com/support/documentation/application_notes/xapp1086-secure-single-fpga-using-7s-idf.pdf>. 91

Hauck and Dehon, 2008 Hauck, S. and Dehon, A. (2008). *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing*. Morgan Kaufmann, Burlington, MA. 13, 52, 53, 54, 55, 56, 57, 66

Holsti and Paakko, 2001 Holsti, N. and Paakko, M. (2001). Towards advanced fdir components.

<ftp://ftp.estec.esa.nl/pub/wm/anonymous/wme/Web/DASIA_2001_Towards20Advanced20FDIR20Components.pdf>. Estudo na Space Systems Finland Ltd. 71, 161, 236

Huang et al., 2014 Huang, R., An, X., Weikang, W., Feng, H., Huang, L., Fan, J., Xing, Z., Wang, Y. B., and Tan, F. (2014). Total ionizing dose (tid) effect and single event effect (see) in quasi-soi nmosfets. *Semiconductor Science and Technology*, 29(1). 80

Huang and McCluskey, 2001 Huang, W.-J. and McCluskey, E. J. (2001). A memory coherence technique for online transient error recovery of fpga configurations. In *Proceedings of the ACM/SIGDA 9th International Symposium on Field Programmable Gate Arrays*, pages 183–192, New York, USA. 76, 93

Ichinomiya et al., 2012a Ichinomiya, Y., Amagasaki, M., Iida, M., Kuga, M., and Sueyoshi, T. (2012a). A bitstream relocation technique to improve flexibility of partial reconfiguration. In *Proceedings of the IEEE International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 139–152, Fukuoka, Japan. 44, 87, 89, 133, 189, 215

Ichinomiya et al., 2012b Ichinomiya, Y., Usagawa, S., Amagasaki, M., Iida, M., Kuga, M., and Sueyoshi, T. (2012b). Designing flexible reconfigurable regions to relocate partial bitstreams. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, page 241, Toronto, Canada. 44, 86, 87, 133, 189, 215

Inc., 2016 Inc., A. (2016). Datasheet dos vários processadores arm. <http://www.arm.com>. 57, 255

Ioannou and Rosa, 2014 Ioannou, D. P. and Rosa, G. L. (2014). Mechanical stress effects on p-channel mosfet performance and nbti reliability. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS)*, pages XT.19.1–XT.19.4, Waikoloa, USA. 136

Isermann, 2006 Isermann, R. (2006). *Fault-Diagnosis Systems*. Springer, Berlin. 72

Iturbe et al., 2011 Iturbe, X., Benkrid, K., Arslan, T., Torrego, R., and Martinez, I. (2011). Methods and mechanisms for hardware multitasking: Executing and synchronizing fully relocatable hardware tasks in xilinx fpgas. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 295–300, Chania, Greece. 14, 100, 101, 133, 135, 215

Iturbe et al., 2012 Iturbe, X., Benkrid, K., Torrego, R., Ebrahim, A., and Arslan, T. (2012). Online clock routing in xilinx fpgas for high-performance and reliability. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 85–91, Erlangen, Germany. 102, 133, 215

J. Susan Milton and Isermann, 2002 J. Susan Milton, J. A. and Isermann, R. (2002). *Introduction to Probability and Statistics: Principles and Applications for Engineering and the Computing Sciences*. McGraw-Hill, USA. 142, 143, 144, 147, 148

Jamuna and Agrawal, 2012 Jamuna, S. and Agrawal, V. K. (2012). Implementation of bistcontroller for fault detection in clb of fpga. In *Proceedings of the Devices, Circuits and Systems (ICDCS), 2012 International Conference*, pages 99–104, Coimbatore, India. 72

Kastensmidt et al., 2014 Kastensmidt, F. L., Tonfat, J., Both, T., Rech, P., Wirth, G., Reis, R., Bruguier, F., Benoit, P., Torres, L., and Frost, C. (2014). Aging and voltage scaling impacts under neutron-induced soft error rate in sram-based fpgas. In *Proceedings of the 19th IEEE European Test Symposium (ETS)*, pages 1–2, Paderborn, Germany. 40

Kirischian et al., 2009 Kirischian, L., Dumitriu, V., and Chun, P. W. (2009). Virtualization of computing resources in rcs for multi-task stream applications. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '09)*, pages 362–373, Cancun, Mexico. 42, 114, 117

Kirischian et al., 2010 Kirischian, L., Dumitriu, V., Chun, P. W., and Okouneva, G. (2010). Research article - mechanism of resource virtualization in rcs for multitask stream applications. *International Journal of Reconfigurable Computing*, 2010(159367):1–13. 15, 42, 114, 117, 118

Kirischian et al., 2006 Kirischian, L., Geurkov, V., Terterian, I., and Kirischain, V. (2006). Multilevel radiation protection of partially reconfigurable field programmable gate array devices. *Journal of Spacecraft and Rockets*, 43(3):523–529. 42, 114, 117, 133, 215

Kirischian et al., 2004 Kirischian, L., Terterian, I., Chun, P. W., and Geurkov, V. (2004). Re-configurable parallel stream processor with self-assembling and self-restorable micro-architecture. In *Proceedings of the International Conference on Parallel Computing in Electrical*

Engineering (PARELEC), pages 165–170, Dresden, Germany. 15, 42, 114, 115, 116

Klutke et al., 2003 Klutke, G.-A., Kiessler, P. C., and Wortman, M. A. (2003). A critical look at the bathtub curve. *IEEE Transactions on Reliability*, 52(1):125–129. 13, 38

Koch et al., 2008 Koch, D., Beckhoff, C., and Teich, J. (2008). Recobus-builder - a novel tool and technique to build statically and dynamically reconfigurable systems for fpgas. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 119–124, Heidelberg, Germany. 88, 133, 215

Koren and Krishina, 2007 Koren, I. and Krishina, C. M. (2007). *Fault-Tolerant System*. Morgan Kaufmann, San Francisco. 37, 39, 71, 149, 194

Lawson et al., 2014 Lawson, D. M. R., Walker, J. A., Trefzer, M. A., Bale, S. J., and Tyrrell, A. M. (2014). A hierarchical fault tolerant system on the panda device with low disruptio. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 69–76, Leicester, UK. 14, 15, 111, 112, 113, 133, 215

Legat et al., 2010 Legat, U., Biasizzo, A., and Novak, F. (2010). Automated seu fault emulation using partial fpga reconfiguration. In *Proceedings of the Design and Diagnostics of Electronic Circuits and Systems (DDECS). IEEE 13th International Symposium*, pages 24–27, Vienna, Austria. 74

Leong et al., 2015 Leong, C., Semiao, J., Santos, M., Teixeira, I., and Teixeira, J. (2015). Fault-tolerance in fpga focusing power reduction or performance enhancement. In *Proceedings of the IEEE 16th Latin America Test Test Symposium (LATS)*, pages 1–6, Puerto Vallarta, Mexico. 201

Leroy and Rancoita, 2009 Leroy, C. and Rancoita, P.-G. (2009). Principles of radiation interaction in matter and detection. *World Scientific*, 2. 79

Li et al., 2013 Li, T., Shafique, M., Rehman, S., Ambrose, J. A., Henkel, J., and Parameswaran, S. (2013). DHASER: Dynamic heterogeneous adaptation for soft-error resiliency in ASIP-based multi-core systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 646–653, San Jose, USA. 198, 236

Liao et al., 2008 Liao, C., Gan, Z., Wu, Y., Zheng, K., Guo, R., Ju, J., Ning, J., He, A., Ye, S., Liu, E., and Wong, W. (2008). Factors for negative bias temperature instability improvement in deep sub-micron cmos technology. In *Proceedings of the IEEE International Conference on Solid-State and Integrated-Circuit Technology (ICSICT)*, pages 612–615, Beijing , RPC. 82, 83

Love et al., 2013 Love, A., Zha, W., and Athanas, P. (2013). In pursuit of instant gratification for fpga design. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Porto, Portugal. 89, 133, 215

Martins et al., 2014a Martins, V. M. G., Ferlini, F., Lettnin, D. V., and Bezerra, E. A. (2014a). Low cost fault detector guided by permanent faults at the end of fpgas life cycle. In *Proceedings of the IEEE 15th Latin America Test Test Workshop (LATW)*, pages 1–6, Fortaleza, Brasil. 16, 49, 72, 157, 163, 167, 168

Martins et al., 2015a Martins, V. M. G., Reis, J. G., Neto, H. C. C., and Bezerra, E. A. (2015a). Designing partial bitstreams for multiple xilinx fpga partitions. In *Proceedings of the IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 256–259, Vancouver, Canada. 16, 49, 159, 177, 183, 223

Martins et al., 2015b Martins, V. M. G., Reis, J. G., Neto, H. C. C., and Bezerra, E. A. (2015b). Fpga redundancy recovery based on partial bitstreams for multiple partitions. In *Proceedings of the IEEE 16th Latin America Test Symposium (LATS)*, pages 1–4, Puerto Vallarta, Mexico. 49

Martins et al., 2015c Martins, V. M. G., Reis, J. G., Neto, H. C. C., and Bezerra, E. A. (2015c). Redundância modular tripla baseada em bitstreams parciais para múltiplas partições da fpga. In *Atas das XI Jornadas sobre Sistemas Reconfiguráveis (REC2015)*, pages 55–61, Porto, Portugal. 49

Martins et al., 2016 Martins, V. M. G., Slongo, L. K., Villa, P. R. C., and Bezerra, E. A. (2016). Soft errors analysis on fpgas for cubesat missions. In *Proceedings of the II Latin American IAA CubeSat Workshop (LACW)*, pages 1–10, Florianópolis, Brasil. 48, 50, 80, 256

Martins et al., 2015d Martins, V. M. G., Villa, P. R. C., Neto, H. C. C., and Bezerra, E. A. (2015d). A tmr strategy with enhanced

dependability features based on a partial reconfiguration flow. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 161–166, Montpellier, France. 16, 49, 159, 194, 229

Martins et al., 2014b Martins, V. M. G., Villa, P. R. C., Slongo, L. K., Salamanca, J. J. L., Sabino, F. L., Martínez, S. V., L. Mariga, I. V., Eiterer, B. V. B., Baldini, M., Felix, M., Spengler, A. W., Melo, F. E. N., Lettnin, D. V., and Bezerra, E. A. (2014b). The experience of designing and developing the on-board electronics of a cubesat in brazil. In *Proceedings of the I Latin American IAA CubeSat Workshop (LACW)*, pages 1–8, Brasília, Brasil. 50

Mück and Fröhlich, 2013 Mück, T. R. and Fröhlich, A. A. (2013). Seamless integration of hw/sw components in a hls-based soc design environment. In *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*, pages 109–115, Montreal, Canada. 223

Mishra et al., 2012 Mishra, R. K., Pandey, A., and Alam, S. (2012). Analysis and impacts of negative bias temperature instability (nbti). In *Proceedings of the IEEE Students Conference on Electrical, Electronics and Computer Science (SCEECS)*, pages 1–4, Bhopal, India. 14, 40, 82, 83, 152, 199

Mizubayashi et al., 2015 Mizubayashi, W., Mori, T., Fukuda, K., Liu, Y. X., Matsukawa, T., Ishikawa, Y., Endo, K., O'uchi, S., Tsukada, J., Yamauchi, H., Morita, Y., Migita, S., Ota, H., and Masahar, M. (2015). Pbti for n-type tunnel finfets. In *Proceedings of the International Conference on IC Design & Technology (ICICDT)*, pages 1–4, Leuven, Belgium. 40, 82, 85, 199

Montminy et al., 2007 Montminy, D. P., Baldwin, R. O., Williams, P. D., and Mullins, B. E. (2007). Using relocatable bitstreams for fault tolerance. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 701–708, Edinburgh, Scotland. 14, 42, 96, 97, 98, 133, 155, 215

Moore, 1998 Moore, G. E. (1998). Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85. 78

Ochoa-Ruiz et al., 2013 Ochoa-Ruiz, G., Touiza, M., Bourennane, E.-B., Guessoum, A., Messaoudi, K., and Hajjaji, M. A. (2013). A novel approach for accelerating bitstream relocation in many-core partially reconfigurable applications. In *Proceedings of the IEEE International Conference on Control, Decision and Information*

Technologies (CoDIT), pages 271–271, Hammamet, Túnisia. 14, 106, 107, 133, 135, 215

OpenCores, 2014 OpenCores (2014). Opencores. <http://opencores.org>. 224

Palermo et al., 2015 Palermo, N., Tihhomirov, V., Copetti, T. S., Jenihhin, M., Raik, J., Kostin, S., Gaudesi, M., Squillero, G., Reorda, M. S., Vargas, F., and Poehls, L. B. (2015). Rejuvenation of nanoscale logic at nbti-critical paths using evolutionary tpg. In *Proceedings of the IEEE 16th Latin America Test Test Symposium (LATS)*, pages 1–6, Puerto Vallarta, Mexico. 16, 85, 152

Parreira et al., 2006 Parreira, A., dos Santos, M. B., and Teixeira, J. P. C. (2006). Bist architectures and fault emulation. In *Proceedings of the IEEE Latin America Test Workshop (LATW)*, Buenos Aires, Argentina. 72

Parreira et al., 2003 Parreira, A., Teixeira, J. P. C., dos Santos, M. B., and de Sousa, J. T. (2003). Fault simulation using partially reconfigurable hardware. In *Proceedings of the International Conference Field Programmable Logic and Applications (FPL)*, pages 839–848, Lisboa, Portugal. 72

Pradhan, 1996 Pradhan, D. K. (1996). *Fault-Tolerant Computer System Design*. Prentice Hall. 37, 41, 198

Project, 2014 Project, T. E. (2014). Embedded parallel operating system. <http://epos.lisha.ufsc.br>. 224

Quinn et al., 2005 Quinn, H., Graham, P., Krone, J., Caffrey, M., and Rezgui, S. (2005). Radiation-induced multi-bit upsets in sram-based fpgas. *IEEE Transactions on Nuclear Science*, 52(6):2455–2461. 40

Rockett, 2001 Rockett, L. R. (2001). A design based on proven concepts of an seu-immune cmos configuration data cell for reprogrammable fpgas. *Proceedings of Microelectronics Journal*, 32(2):99–111. 74

Sengupta and Sapatnekar, 2014 Sengupta, D. and Sapatnekar, S. S. (2014). Predicting circuit aging using ring oscillators. In *Proceedings of the IEEE 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 430–435, Singapore. 204

Shen et al., 2006 Shen, C., Li, M., Foo, C., Yang, T., Huang, D., Yap, A., Samudra, G., and Yeo, Y. (2006). Characterization and physical origin of fast vth transient in nbti of pmosfets with sion dielectric. In *Proceedings of the IEEE Annual International Electron Devices Meeting (IEDM)*, pages 1–4, San Francisco, USA. 82, 83

Shih and Hsiung, 2009 Shih, K.-J. and Hsiung, P.-A. (2009). Reconfigurable computing technologies overview. In *Encyclopedia of Information Science and Technology, Second Edition*, pages 3241–3250, New York, USA. 41, 51, 52, 312

Shnidman et al., 1998 Shnidman, N. R., Mangione-Smith, W. H., and Potkonjak, M. (1998). On-line fault detection for bus-based field programmable gate arrays. *Proceedings of the IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(4):656–666. 93

Silva and Wirth, 2010 Silva, R. D. and Wirth, G. I. (2010). Logarithmic behavior of the degradation dynamics of metal oxide semiconductor devices. *Journal of Statistical Mechanics: Theory and Experiment*, P04025:1–12. 84

Steiner et al., 2011 Steiner, N., Wood, A., Shojaei, H., Couch, J., Athanas, P., and French, M. (2011). Torc: Towards an open-source tool flow. In *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays*, pages 41–44, New York, USA. 90, 133, 215

Touiza et al., 2012 Touiza, M., Ochoa-Ruiz, G., Bourennane, E.-B., Guessoum, A., and Messaoudi, K. (2012). A novel methodology for accelerating bitstream relocation in partially reconfigurable systems. *Microprocessors and Microsystems*, 37(3):358–372. 88, 106, 107, 133, 215

Trefzer et al., 2011 Trefzer, M. A., Walker, J. A., and Tyrrell, A. M. (2011). A programmable analogue and digital array for bio-inspired electronic design optimization at nanoscale silicon technology nodes. In *Proceedings of the Conference on Signals, Systems and Computers (ASILOMAR)*, pages 1537–1541, Pacific Grove, USA. 111

Vattikonda et al., 2006 Vattikonda, R., Wang, W., and Cao, Y. (2006). Memory-efficient and fast run-time reconfiguration of regularly structured designs. In *Proceedings of the ACM/IEEE International 43th Design Automation Conference (DAC)*, pages 1047–1052, San Francisco, USA. 205

Walker et al., 2013 Walker, J. A., Trefzer, M. A., Bale, S. J., and Tyrrell, A. M. (2013). Panda: A reconfigurable architecture that adapts to physical substrate variations. *IEEE Transactions on Computers*, 62(8):1584–1596. 111, 112

Wang et al., 2016 Wang, Q., Chen, D., and Bai, H. (2016). A method of space radiation environment reliability prediction. In *Proceedings of the IEEE Annual Reliability and Maintainability Symposium (RAMS)*, pages 1–6, Tucson, USA. 79, 80

Wang and Xu, 2014 Wang, T. and Xu, Q. (2014). On the simulation of nbti-induced performance degradation considering arbitrary temperature and voltage variations. In *Proceedings of the IEEE Annual Design Automation Conference (DAC)*, pages 1–6, San Francisco, USA. 82, 83

White, 2012 White, D. (2012). Considerations surrounding single event effects in fpgas, asics, and processors v1.0.1 - Xilinx inc. inc. white paper (wp402). <http://www.xilinx.com/support/documentation/white_papers/wp402_SEE_Considerations.pdf>. 14, 40, 41, 73, 76, 93

Xilinx Inc., 2007 Xilinx Inc. (2007). Difference-based partial reconfiguration v2.0 (xapp290). <http://www.xilinx.com/support/documentation/application_notes/xapp290.pdf>. 70

Xilinx Inc., 2009 Xilinx Inc. (2009). Power consumption at 40 and 45nm - white paper: Spartan-6 and virtex-6 devices v1.0. <http://www.xilinx.com/support/documentation/white_papers/wp298.pdf>. 204, 211

Xilinx Inc., 2010 Xilinx Inc. (2010). Spartan-6 fpga configurable logic block user guide v1.1 (ug384). <http://www.xilinx.com/support/documentation/user_guides/ug384.pdf>. 13, 17, 18, 19, 58, 60, 61, 62, 277, 284, 300, 301, 302

Xilinx Inc., 2011a Xilinx Inc. (2011a). Logicore ip soft error mitigation controller user guide v3.1 (ug764). <http://www.xilinx.com/support/documentation/ip_documentation/sem/v3_1/ug764_sem.pdf>. 41, 77, 158

Xilinx Inc., 2011b Xilinx Inc. (2011b). Spartan-6 family overview v2.0 (ds160). <http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf>. 57

Xilinx Inc., 2011c Xilinx Inc. (2011c). Spartan-6 fpga block ram resources user guide v1.5 (ug383). <http://www.xilinx.com/support/documentation/user_guides/ug383.pdf>. 304, 306

Xilinx Inc., 2012a Xilinx Inc. (2012a). Radiation-hardened, space-grade virtex-5qv family overview data sheet v1.3 (ds192). <http://www.xilinx.com/support/documentation/data_sheets/ds192_V5QV_Device_Overview.pdf>. 38, 40, 80, 125

Xilinx Inc., 2012b Xilinx Inc. (2012b). Virtex-5 fpga configuration user guide v3.11 (ug191). <http://www.xilinx.com/support/documentation/user_guides/ug191.pdf>. 19, 44, 63, 64, 66, 77, 164

Xilinx Inc., 2012c Xilinx Inc. (2012c). Virtex-5 fpga user guide v5.4 (ug190). <http://www.xilinx.com/support/documentation/user_guides/ug190.pdf>. 13, 19, 53, 58, 60, 61, 62

Xilinx Inc., 2012d Xilinx Inc. (2012d). Virtex-6 fpga configurable logic block user guide v1.2 (ug364). <http://www.xilinx.com/support/documentation/user_guides/ug364.pdf>. 13, 19, 58, 60, 61, 62

Xilinx Inc., 2013a Xilinx Inc. (2013a). 7 series fpgas configurable logic block user guide v1.5 (ug474). <http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf>. 13, 19, 58, 60, 61, 62

Xilinx Inc., 2013b Xilinx Inc. (2013b). Constraints guide v14.5 (ug625). <http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/cgd.pdf>. 43, 177, 181

Xilinx Inc., 2013c Xilinx Inc. (2013c). Hierarchical design methodology guide user guide v14.5 (ug748). <http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/Hierarchical_Design_Methodology_Guide.pdf>. 45

Xilinx Inc., 2013d Xilinx Inc. (2013d). Partial reconfiguration tutorial: Planahead design tool v14.5. <http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/PlanAhead_Tutorial_Partial_Reconfiguration.pdf>. 44, 86, 133, 177, 178, 215

Xilinx Inc., 2013e Xilinx Inc. (2013e). Partial reconfiguration user guide v14.5 (ug702). <http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug702.pdf>. 14, 43, 44, 68, 69, 87, 168, 277

Xilinx Inc., 2013f Xilinx Inc. (2013f). Single chip crypto lab using pr/iso flow with the virtex-5 family for ise design suite 12.1 v1.1.2 (xapp1105). <http://www.xilinx.com/support/documentation/application_notes/xapp1105_V5SCC_PRISO.pdf>. 183

Xilinx Inc., 2013g Xilinx Inc. (2013g). Virtex-6 libraries guide for hdl designs v14.7 (ug623). <http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/virtex6_hdl.pdf>. 16, 90, 102, 180, 201, 202

Xilinx Inc., 2013h Xilinx Inc. (2013h). Zynq-7000 all programmable soc overview v1.6 (ds190). <http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf>. 57

Xilinx Inc., 2014a Xilinx Inc. (2014a). Jbits 3.0 sdk for virtex-ii. <http://www.xilinx.com/labs/projects/jbits>. 96

Xilinx Inc., 2014b Xilinx Inc. (2014b). Space radiation effects. <http://www.xilinx.com/esp/aerospace-defense/space/radiation-effects.htm>. 75

Xilinx Inc., 2014c Xilinx Inc. (2014c). Virtex-6 fpga clocking resources user guide v2.3 (ug362). <http://www.xilinx.com/support/documentation/user_guides/ug362.pdf>. 242

Xilinx Inc., 2014d Xilinx Inc. (2014d). Virtex-6 fpga system monitor user guide v1.2 (ug370). <http://www.xilinx.com/support/documentation/user_guides/ug370.pdf>. 237

Xilinx Inc., 2015a Xilinx Inc. (2015a). 7series fpgas configuration user guide v1.10 (ug470). <http://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf>. 13, 19, 63, 64, 65, 66

Xilinx Inc., 2015b Xilinx Inc. (2015b). Spartan-6 fpga configuration user guide v2.8 (ug380). <http://www.xilinx.com/support/documentation/user_guides/ug380.pdf>. 17, 19, 63, 66, 77, 277, 281, 283, 285

Xilinx Inc., 2015c Xilinx Inc. (2015c). Virtex-6 fpga configuration user guide v3.9 (ug360). <http://www.xilinx.com/support/documentation/user_guides/ug360.pdf>. 14, 19, 44, 63, 64, 66, 68, 77, 195, 229

Xilinx Inc., 2015d Xilinx Inc. (2015d). Vivado design suite user guide v2015.4 (ug893). <http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug893-vivado-ide.pdf>. 44

Xilinx Inc., 2016a Xilinx Inc. (2016a). All programmable technologies and devices. <http://www.xilinx.com>. 51, 53

Xilinx Inc., 2016b Xilinx Inc. (2016b). Continuing experiments of atmospheric neutron effects on deep submicron integrated circuits v2.0 (wp286). <http://www.xilinx.com/support/documentation/white_papers/wp286.pdf>. 39, 71

Xilinx Inc., 2016c Xilinx Inc. (2016c). Isolation design flow. <http://www.xilinx.com/applications/isolation-design-flow.html>. 14, 91, 92, 133, 177, 179, 182, 215

Zhang et al., 2014a Zhang, Y., Chen, S., and Chen, L. P. S. (2014a). Mitigating NBTI degradation on FinFET GPUs through exploiting device heterogeneity. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 577–582, Tampa, Finland. 40, 197

Zhang et al., 2014b Zhang, Y., Huang, H., Bi, D., Tang, M., and Zhang, Z. (2014b). Investigation of unique total ionizing dose effects in 0.2 μm partially-depleted silicon-on-insulator technology. *Nuclear Instruments and Methods in Physics Research*, 745:128–132. 76, 79

Apêndices

APÊNDICE A – XILINX SPARTAN-6 *INSIDE OUT*

A.1 INTRODUÇÃO

Logo nas primeiras hipóteses mais gerais para o tipo de solução que se procura houve um forte interesse em averiguar as reais capacidades das FPGAs. Para isso, foi necessário fazer um esforço para estudá-las ao máximo e tentar conhecê-las ao detalhe. Esse estudo teria de passar pela forma como é programada a FPGA e como é organizada a informação digital que a programa (*bit files*). Nesse propósito, foi feito este estudo e a opção da família de FPGA para o fazer recaiu na Spartan-6 da Xilinx, uma gama bastante madura e economicamente mais acessível.

A tarefa de obtenção de informações da Spartan-6 selecionada focou-se nos bits de configuração (e na organização dos mesmos) usados nos CLBs, com especial atenção nos bits responsáveis pela configuração das LUTs e nos blocos de memória BRAM (configuração de propriedades e dados). Como ponto de partida, foi aproveitada muita da informação já obtida na realização de uma tese de mestrado no programa de pós-graduação de Engenharia Elétrica na Universidade Federal de Santa Catarina (Ferlini, 2012).

Este foi, acima de tudo, um trabalho de engenharia inversa, sendo orientado por alguma informação da fabricante de FPGAs Xilinx. Devido à opção da família Spartan-6, a primeira fonte de informação foi o guia de configuração desta família (Xilinx Inc., 2015b), ao que se seguiu o estudo em detalhe dos CLBs através do guia correspondente (Xilinx Inc., 2010).

Após o entendimento da constituição de uma FPGA, o próximo passo foi obter informações relativas ao processo de configuração/reconfiguração do dispositivo. Essas informações foram obtidas por intermédio do guia do usuário de reconfiguração parcial da Xilinx (Xilinx Inc., 2013e).

A.2 FPGA SPARTAN-6 - VISÃO GLOBAL

Tal como qualquer outra FPGA atual, a Spartan-6 possui um leque de diferentes recursos: CLBs com LUTs de 6 entradas, blocos de memória BRAM, unidades de processamento digital de sinal DSPs, módulos de gestão de relógios, etc. A Figura 76 dá-nos uma boa ideia dos recursos existentes e da sua distribuição.

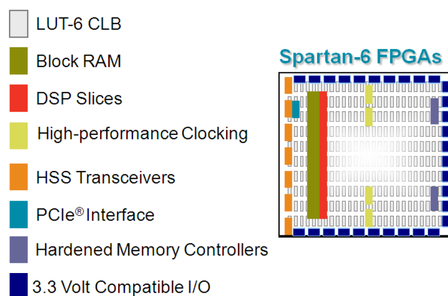


Figura 76 – Constituição de uma FPGA Spartan-6.

Neste estudo, o foco incidirá sobre os bits de configuração dos CLBs, suas respetivas LUTs e dos blocos de memória BRAM.

A.2.1 ***Configurable Logic Blocks*** **(CLBs)**

Os CLBs são a constituição base de uma FPGA, pois são eles os responsáveis pela implementação do circuito digital (combinatório e/ou sequencial).

Na Spartan-6 cada CLB é constituído por dois *slices*. E se em famílias mais antigas a constituição interna destes era homogénea, nas mais recentes e por uma questão de otimização de recursos, tal deixou de verificar-se. Isto deve-se ao facto de que em certas partes do circuito digital implementado são necessários certos tipos específicos de recursos, e noutras partes outros tipos de recursos, mas raramente ambos. Por causa desta realidade, o dispositivo é constituído por colunas intercaladas de CLBs com *slices* do tipo SLICEM/SLICEX e SLICEL/SLICEX. Tal facto é devidamente ilustrado no diagrama da Figura 77.

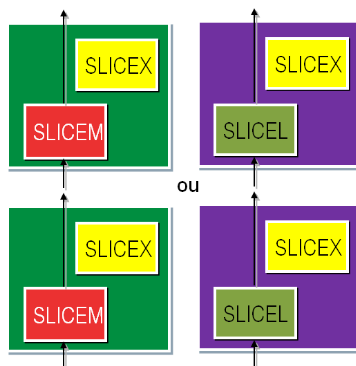


Figura 77 – Exemplo dos dois tipos de colunas de CLBs.

As linhas verticais a preto presentes na Figura 77 simbolizam a cadeia de *carry* que apenas os *slices* tipo SLICEM e SLICEL possuem.

A.2.2 SLICEX, SLICEL e SLICEM

Como referido em A.2.1, devido à política de obtenção da melhor eficiência dos recursos existentes em cada CLB, existe mais do que um tipo de *slice*. O mais básico, o SLICEX, que possui apenas as quatro LUTs (designadas por LUTA, LUTB, LUTC e LUTD), os oito flip-flops e alguns multiplexadores para efetuar as interligações internas. O SLICEL, que soma às capacidades do SLICEX as capacidades lógicas associadas à cadeia de *carry* (muito importante para a implementação de hardware que necessite de realizar operações aritméticas). E o SLICEM, o mais completo, que para além dos recursos no SLICEL, possui todas as capacidades para que as LUTs possam ser também utilizadas como memória distribuída ou como SRL.

A Figura 78 mostra um *slice* com todos os recursos possíveis de alocar internamente.

A.2.3 *Look Up Table* (LUT)

Responsáveis por implementar as mais variadas funções lógicas, as LUTs são porventura o recurso principal existente numa FPGA.

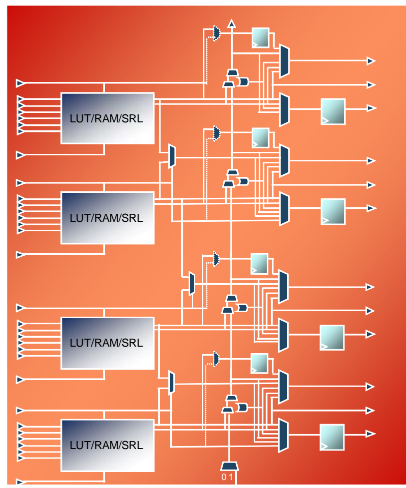


Figura 78 – Diagrama de um slice do tipo SLICEM.

Nas versões anteriores da família Spartan, cada LUT era constituída por uma memória de 16 posições de um bit cada (16 bits no total), o que permitia implementar em cada uma destas memórias, uma função lógica de até quatro entradas. Na versão estudada, a Spartan-6, esta memória possui 64 posições de um bit cada (64 bits) e permite implementar funções de até seis entradas.

Isto pode ser melhor entendido consultando a Figura 79.

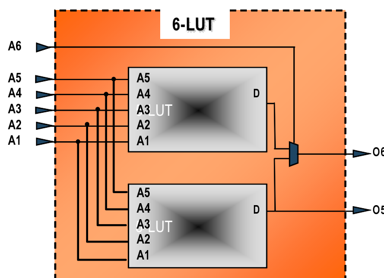


Figura 79 – Composição de uma LUT de seis entradas.

Embora no total cada LUT seja de seis entradas, estas permitem ser separadas em duas de cinco entradas, elevando assim a capacidade

de otimização de recursos da FPGA.

A.3 PROCESSO DE CONFIGURAÇÃO DE UMA FPGA SPARTAN-6

Após o processo de elaboração de um sistema digital numa linguagem de descrição de hardware (VHDL ou Verilog) e respetiva simulação, o processo normal seguinte é gerar um *bit file* e programar a FPGA para a qual foi gerado.

Este procedimento de programação da FPGA é dividido em oito etapas tal como o ilustrado na Figura 80.

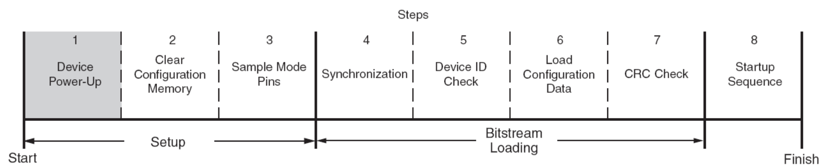


Figura 80 – Sequência de configuração da Spartan-6 (Xilinx Inc., 2015b).

As primeiras três etapas constituem o procedimento de *setup* da Spartan-6. Seguindo-se mais quatro etapas (4 a 7) que compõem o procedimento de carregamento do *bit file*. E por fim, uma última etapa que liberta o dispositivo do processo de configuração e inicia o normal funcionamento da FPGA, depois de devidamente configurada.

Neste trabalho será focado apenas o procedimento de carregamento do *bit file*. No entanto, mais informação relativa a todo o processo pode ser encontrada na literatura da Xilinx (Xilinx Inc., 2015b).

A.3.1 Sequência do carregamento da configuração de uma FPGA Spartan-6

O ficheiro '.bit' gerado pelas ferramentas da Xilinx é mais do que um simples conjunto de bits que configura a FPGA. É que ele é na realidade um conjunto de comandos que desempenha as etapas 4 a 8 enumeradas na Figura 80.

A sequência contida num *bit file* gerado para a Spartan-6 usada neste estudo (6SLX45) é a que se encontra descrita abaixo:

(1) Async Word Detect AA995566.
CMD (Command Register): RCRC: Resets CRC: Resets the CRC register.
FLR (Frame Lenght Register) = 1576.
COR1 (Configuration Option Register 1) = 0x3d08.
COR2 (Configuration Option Register 2) = 0x09ee.
(2) DCODE (Devide ID Register) = 0x04008093 (XC6SLX45).
MASK (Masking Register for CTL) = 0x00cf.
CTL (Control Register) = 0x0081.
CCLK_FREQ (CCLK Frequency Select for Master Mode) = 0x3cc8.
PWRDN_REG (Power-down Option Register) = 0x0881.
EYE_MASK (Mask Pins for Multi-Pin Wake-Up) = 0x0000.
HC_OPT_REG (House Clean Option Register) = 0x001f. CWDT (Configuration Watchdag Timer) = 0xffff.
PU_GWE (GWE Cycle During Wake-up From Suspend) = 0x0005.
PU_GTS (GTS Cycle During Wake-up From Suspend) = 0x0004.
MODE_REG (Reboot Mode Register) = 0x0100.
GENERAL1 (Power-up Self Test or Loadable Program Address) = 0x0000.
GENERAL2 (Power-up Self Test or Loadable Program Address and new SPI Opcode) = 0x0000.
GENERAL3 (Golden Bitstream Address) = 0x0000.
GENERAL4 (Golden Bitstream Address and new SPI Opcode) = 0x0000.
GENERAL5 (User-defined Register for Fail-safe Scheme) = 0x0000.
SEU_OPT (SEU Frequency, Enable and Status) = 0x1be2.
EXP_SIGN (Expected Readback Signature for SEU Detection) = 0x00000000.
(3) FAR_MAJ (Frame Address Register Block and Major)
(3) - BLK=0;
(3) - ROW=0;
(3)- MAJOR=0; - Column Major.
(3) FAR_MIN (Frame Address Register Minor)
(3) - Block RAM=0;
(3) - MINOR=0; - Column Minor.
(3) CMD (Command Register): WCFG: Writes Configuration Data: Used prior configuration data to the FDRI.
(3) FDRI (Frame Data Register, Input Register (write configuration data)) = 742057 words 16bits.
CMD (Command Register): GRESTORE: Pulses the GRESTORE signal.
CMD (Command Register): LFRM: Last Frame.
CMD (Command Register): GRESTORE: Pulses the GRESTORE signal.
CMD (Command Register): START: Begins the Startup Sequence: Initiates the startup sequence.
MASK (Masking Register for CTL) = 0x00ff.
CTL (Control Register) = 0x0081.
(4) CRC (Cyclic Redundancy Check) = 0x000540ff.
(4) CMD (Command Register) : DESYNC: Resets the DALIGN Signal: Used at the end of configuration to desynchronize the device.

A primeira etapa, identificada com (1), que compõe o procedimento de carregamento do *bit file*, a etapa 4, resume-se à procura no *bit file* da palavra de sincronismo (0xAA995566). Após várias informações extra contidas no cabeçalho do ficheiro, esta sequência de 32 bits permite

identificar onde começa a sequência de comandos a enviar para a FPGA.

Na secção identificada com (2), o comando DCODE verifica se a configuração que irá ser carregada foi gerada para o dispositivo em questão.

Após outra sequência de comandos, que podem ser devidamente entendidos consultando a correspondente literatura da Xilinx (Xilinx Inc., 2015b), ocorre a etapa 6 (3) e é carregado o conteúdo da configuração da FPGA.

Os comandos para esta etapa são o FAR_MAJ e FAR_MIN, que registam o endereçamento inicial onde começará a ser carregada a configuração (neste caso será a primeira posição interna da FPGA (BLK=0, ROW=0, MAJOR=0, Block RAM=0 e MINOR=0)), e os comandos CMD(WCFG) e FDRI que transferem todas as 742057 palavras de 16 bits para o interior do dispositivo.

Por último, identificado com (4), o comando CRC verifica se o valor de CRC presente no *bit file* corresponde ao calculado internamente pela FPGA durante o carregamento da configuração. E para finalizar, é enviado o comando CMD(DESYNC) para sinalizar que terminou o processo de carregamento da configuração e, deste modo, libertar o dispositivo para a sua normal operação.

A.3.2 Organização do endereçamento de uma FPGA Spartan-6

Antes de explicar a forma como são organizados os dados de configuração numa Spartan-6, é importante referir que estes dados estão organizados em blocos de 65 palavras de 16 bits (130 bytes). A cada bloco destes é dado o nome de *frame*.

A Figura 81 apresenta-nos uma 'radiografia' da constituição da FPGA usada, enquanto que a Figura 82 mostra-nos a composição dos parâmetros enviados juntamente com os comandos que escrevem nos registos FAR_MAJ e FAR_MIN (referidos na etapa 6 do processo descrito em A.3.1).

Existem oito linhas (*ROWs*) assinaladas por rectângulos ama-

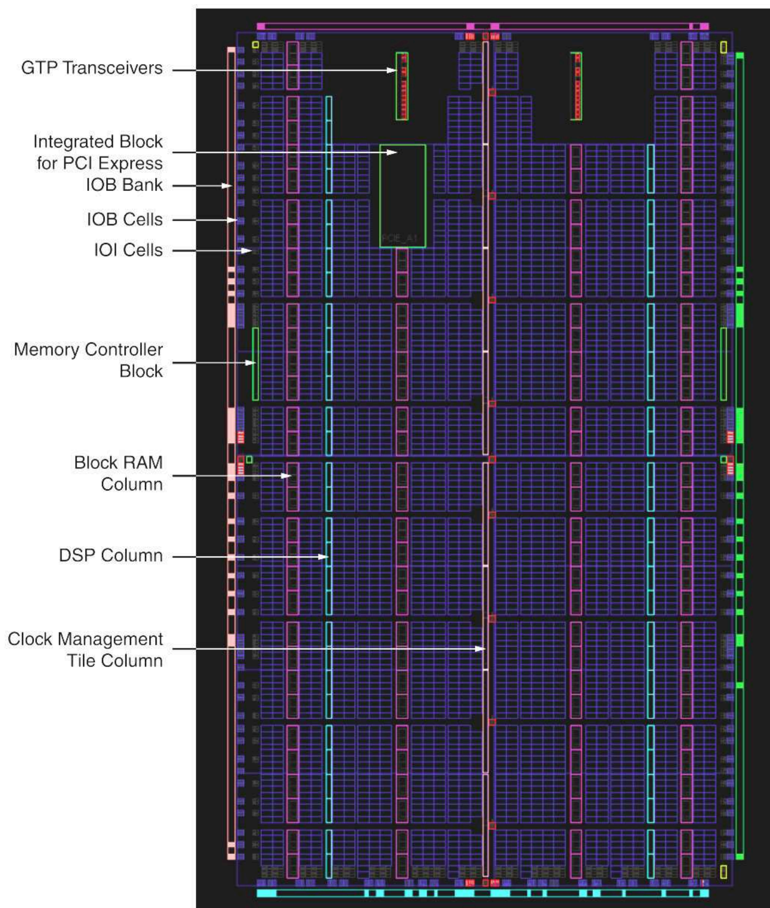


Figura 81 – Vista interna da Spartan-6 XC6SLX45T (Xilinx Inc., 2010).

relos, sendo que cada uma delas é constituída por 40 colunas (*MAJOR*), algumas delas, para ilustração, delineadas com rectângulos verdes na linha número zero.

O tamanho destas colunas não é homogéneo, porque o volume de bits de configuração necessário, varia em função do tipo de recursos da FPGA correspondentes. No entanto, o seu tamanho é sempre um número inteiro de *frames*, sendo a identificação de cada um dado pelo campo MINOR do registo FAR_MIN, presente na Figura 82.

Address Type	Bit Index	Description
Block Type	[25:23]	Valid block types are CLB, I/O, CLK (000), block RAM content (001), and CFG_CLB (010). A normal bitstream does not include type 010.
Top/Bottom Bit	22	Select between top-half rows (0) and bottom-half rows (1).
Row Address	[21:17]	Selects the current row. The row addresses increment from center to top and then reset and increment from center to bottom.
Column Address	[16:7]	Selects a major column, such as a column of CLBs. Column addresses start at 0 on the left and increase to the right.
Minor Address	[6:0]	Selects a frame within a major column.

Figura 82 – Registos de endereçamento dos *frames* (Xilinx Inc., 2015b).

Resumindo, para ler e/ou escrever num determinado *frame* da FPGA, é necessário colocar nos registos FAR_MAJ e FAR_MIN os campos referentes à linha (ROW), coluna (MAJOR) e índice do *frame* na coluna (MINOR).

Para além das oito linhas de configuração até aqui mencionadas, existem ainda dados responsáveis pelo conteúdo das memórias BRAMs existentes na FPGA. Estes valores vêm no *bit file* após toda a parte correspondente à configuração das oito linhas. Para acesso a esta secção, são usados os restantes campos (BLK e Block RAM). No entanto, ainda não foi possível identificar o modo exacto de como são utilizados estes parâmetros.

A.4 ORGANIZAÇÃO DOS DADOS DE CONFIGURAÇÃO DE UMA FPGA SPARTAN-6

Após uma breve introdução da constituição de uma FPGA Spartan-6 e da sua organização interna, seguem-se os detalhes da organização interna do *bit file* e da própria FPGA.

Observando de novo a Figura 81, e com o auxílio da informação fornecida pela Xilinx, sabemos que o dispositivo alvo de estudo possui 30 colunas de CLBs (60 colunas de *Slices*), quatro colunas de BRAMs, duas de DSPs e uma com módulos de gestão de sinais de relógio (DCMs e PLL_ADV).

O acesso a toda esta matriz de recursos será seguidamente

apresentada numa lógica *top-down*.

A.4.1 Organização do *bit file* completo

Tendo por base o dispositivo XC6SLX45, este necessita de 1136 *frames* para configurar cada uma das oito linhas (ROW) e cerca de 2328 *frames* para configurar o conteúdo das BRAMs.

A Figura 83, para além dos *frames* necessários para cada secção, mostra-nos que grupos de recursos são configurados com esses mesmos *frames*.

Row	0	1	2	3	4	5	6	7	
Full Bitfile	SlicesX[0:59]Y[15:0] BRAMsX[3:0]Y[7:0] DSPsX[1:0]Y[3:0] DCMsX[0]Y[1:0](*)	SlicesX[0:59]Y[31:16] BRAMsX[3:0]Y[15:8] DSPsX[1:0]Y[7:4] PLL_ADVsX[0]Y[0](*)	SlicesX[0:59]Y[47:32] BRAMsX[3:0]Y[23:16] DSPsX[1:0]Y[11:8] DCMsX[0]Y[3:2](*)	SlicesX[0:59]Y[63:48] BRAMsX[3:0]Y[31:24] DSPsX[1:0]Y[15:12] PLL_ADVsX[0]Y[1](*)	SlicesX[0:59]Y[79:64] BRAMsX[3:0]Y[39:32] DSPsX[1:0]Y[19:16] DCMsX[0]Y[5:4](*)	SlicesX[0:59]Y[95:80] BRAMsX[3:0]Y[47:40] DSPsX[1:0]Y[23:20] PLL_ADVsX[0]Y[2](*)	SlicesX[0:59]Y[111:96] BRAMsX[3:0]Y[55:48] DSPsX[1:0]Y[27:24] DCMsX[0]Y[7:6](*)	SlicesX[0:59]Y[125:112] BRAMsX[3:0]Y[63:56] DSPsX[1:0]Y[31:28] PLL_ADVsX[0]Y[3](*)	BRAMs - DATA
Frames	1136	1136	1136	1136	1136	1136	1136	1136	2328

Figura 83 – Organização de um *bit file* completo.

Cada linha é responsável por configurar 16 *slices* de cada coluna, oito BRAMs RAMB8 (e quatro BRAMs RAMB16) de cada coluna de BRAMs, quatro DSPs, e dois DCMs ou um PLL_ADV, dependendo se a linha é par ou ímpar.

Embora como ilustrado pela Figura 81, a FPGA tenha visivelmente zonas sem recursos em algumas linhas, na realidade o número de *frames* mantém-se sempre o mesmo. Tal sucede porque embora não existam aí os recursos expectáveis correspondentes às colunas de *slices*, BRAMs e DSPs, outros recursos específicos da FPGA utilizam os mesmos *frames* e desse modo mantêm a composição de *frames* por linha constante.

A.4.2 Organização da configuração de uma linha (ROW)

Embora cada linha possua 40 colunas, a distribuição dos 1136 *frames* não é uniforme. O tamanho de cada coluna depende dos recursos

que esta configura. A Figura 84 mostra a distribuição dos *frames* ao longo da primeira linha (nas outras linhas a distribuição é igual).

MAJOR	1 Row 1136 Frames	Frames
0	IOs A.0(1)	17
1	IOs A.1(1)	17
2	SliceX[0:1]Y[15:0]	30
3	SliceX[2:3]Y[15:0]	31
4	BRAMsX[0]Y[7:0]	25
5	SliceX[4:5]Y[15:0]	30
6	SliceX[6:7]Y[15:0]	31
7	DSPsX[0]Y[3:0]	24
8	SliceX[8:9]Y[15:0]	30
9	SliceX[10:11]Y[15:0]	31
10	SliceX[12:13]Y[15:0]	30
11	SliceX[14:15]Y[15:0]	31
12	SliceX[16:17]Y[15:0]	30
13	BRAMsX[1]Y[7:0]	25
14	SliceX[18:19]Y[15:0]	31
15	SliceX[20:21]Y[15:0]	30
16	SliceX[22:23]Y[15:0]	31
17	SliceX[24:25]Y[15:0]	30
18	SliceX[26:27]Y[15:0]	31
19	SliceX[28:29]Y[15:0] (3)	31
20	SliceX[30:31]Y[15:0]	31
21	SliceX[32:33]Y[15:0]	30
22	SliceX[34:35]Y[15:0]	31
23	SliceX[36:37]Y[15:0]	30
24	SliceX[38:39]Y[15:0]	31
25	SliceX[40:41]Y[15:0]	30
26	BRAMsX[2]Y[7:0]	25
27	SliceX[42:43]Y[15:0]	30
28	SliceX[44:45]Y[15:0]	31
29	SliceX[46:47]Y[15:0]	30
30	SliceX[48:49]Y[15:0]	31
31	SliceX[50:51]Y[15:0]	30
32	DSPsX[1]Y[3:0]	31
33	SliceX[52:53]Y[15:0]	31
34	SliceX[54:55]Y[15:0]	30
35	BRAMsX[3]Y[7:0]	25
36	SliceX[56:57]Y[15:0]	31
37	SliceX[58:59]Y[15:0]	30
38	IOs B.0(2)	16
39	IOs B.1(2)	16

Figura 84 – Distribuição dos *frames* ao longo da primeira linha (ROW 0).

Testes indicaram que as duas primeiras colunas juntas (0 e 1) e as duas últimas (38 e 39) possuem 34 e 32 *frames* respetivamente. No entanto, não foi identificado qual o verdadeiro número de *frames* por cada uma destas colunas individualmente, pelo que se optou por dividir provisoriamente em duas colunas de 17 *frames* **(1)** e duas de 16 *frames* cada **(2)**.

É visível na mesma figura que existem dois tipos de colunas dominantes e que são responsáveis por configurarem secções de colunas de *slices*. Com uma cor roxa e que usam 30 *frames* cada, estão as configurações de colunas com *slices* SLICEL e SLICEX. Com uma cor verde e com 31 *frames* cada, estão os dados responsáveis pela configuração das colunas com *slices* SICEM e SLICEX.

Na coluna (MAJOR) 19 **(3)** existe uma pequena exceção. Embora seja uma coluna constituída com *slices* SLICEL e SLICEX, no final existe um *frame* a mais responsável pela configuração de DCMs ou PLL_ADV (dependendo se é linha par ou ímpar).

Em relação às quatro colunas que configuram o funcionamento das BRAMs (RAMB8 e RAMB16), estas possuem 25 *frames* cada.

As restantes colunas (7 e 32) são responsáveis pela configuração dos DSPs. Estas possuem uma quantidade diferente de *frames* entre elas (24 vs 31), diferença essa que não foi investigada e por isso ainda não é possível justificar.

A.4.3 Organização da configuração das colunas de 16 CLBs

Como descrito anteriormente em A.4.2, existem dois tipos de colunas de *frames* responsáveis pela configuração dos CLBs: colunas com 16 pares de *slices* SLICEL e SLICEX e colunas com 16 pares de *slices* SLICEM e SLICEX.

As colunas compostas com *slices* SLICEL e SLICEX possuem a distribuição de *frames* indicada na Figura 85.

MINOR	0 - 19	20	21 - 24	25	26 - 29
SLICEL + SLICEX Column 30 Frames	Switchs (20 Frames)	CLB.Settings (1 Frame)	SLICEL.LUTs (4 Frames)	CLB.Settings (1 Frame)	SLICEX.LUTs (4 Frames)

Figura 85 – Distribuição dos *frames* numa coluna com *slices* SLICEL e SLICEX.

Os primeiros 20 *frames* (0 a 19) têm como função configurar as interligações dos próprios *slices* com os restantes recursos da FPGA.

Os *frames* 20 e 25 configuram o funcionamento dos CLBs (excluindo conteúdo das LUTs), ficando os *frames* 21 a 24 responsáveis pelo conteúdo das LUTs pertencentes a *slices* SLICEL, e os *frames* 26 a 29 responsáveis pelos dados que preenchem as LUTs que fazem parte dos *slices* do tipo SLICEX.

Mais detalhes sobre os *frames* 21 a 24 e 26 a 29 serão apresentados na Secção A.4.3.1. Em relação aos *frames* 20 e 25, mais informações serão expostas na Secção A.4.3.2.

Focando agora nas colunas compostas por *slices* SLICEM e SLICEX, estas possuem a distribuição de *frames* indicada na Figura 86.

Os primeiros 20 *frames* (0 a 19) possuem a mesma função tal e qual como no tipo de colunas anterior. O mesmo acontecendo com o *frame* 20.

Os *frames* 21, 22, 24 e 25 correspondem ao conteúdo das LUTs

MINOR	0 - 19	20	21 - 22	23	24 - 25	26	27 - 30
SLICEM + SLICEX Column 31 Frames	Switchs (20 Frames)	CLB.Settings (1 Frame)	SLICEM.LUTs (2 Frames)	SLICEM.Settings (1 Frame)	SLICEM.LUTs (2 Frames)	CLB.Settings (1 Frame)	SLICEX.LUTs (4 Frames)

Figura 86 – Distribuição dos *frames* numa coluna com *slices* SLICEM e SLICEX.

pertencentes a *slices* SLICEM. E os frames 27 a 30 configuram as LUTs que fazem parte dos *slices* do tipo SLICEX.

Embora com outro índice (MINOR), o *frame* 26 é em tudo idêntico ao *frame* 25 das colunas com *slices* SLICEL e SLICEX. Sobrando por isso o *frame* 23 como a principal diferença entre estes dois tipos de colunas de CLBs.

Este *frame* extra tem como responsabilidade configurar características de funcionamento que apenas os *slices* do tipo SLICEM possuem.

Detalhes sobre os *frames* 21, 22, 24, 25, 27, 28, 29 e 30 serão apresentados na próxima secção. Em relação aos *frames* 20, 23 e 26, mais informações serão detalhadas na Secção A.4.3.2.

A.4.3.1 Organização dos bits de configuração das LUTs nos *slices*

Descrevendo agora a organização dos bits de configuração das LUTs existentes nos *slices*, tal como já mencionado neste estudo, cada coluna configura 16 CLBs, que por sua vez incluem 32 *slices* (16+16). Cada um destes *slices* possui quatro LUTs, que obrigam a ter quatro palavras de 16 bits (64 bits) para configurar o conteúdo de cada uma delas.

A organização de cada grupo destes 64 bits depende do tipo de

slices (SLICEX, SLICEL ou SLICEM).

No caso das LUTs pertencentes a *slices* do tipo SLICEX, a Figura 87 mostra como estão organizadas as palavras de configuração das LUTs pertencentes aos *slices* SLICEX existentes na primeira coluna de 16 CLBs (ROW=0 e MAJOR=2). Esta coluna, tal como mostrou anteriormente a Figura 85, usa os *frames* 26 a 29 para configurar as correspondentes LUTs.

Frames	26 (27)												
SLICEX (LUTs) 4 Frames	X1Y15.LUTC.0	X1Y15.LUTA.0	X1Y14.LUTC.0	X1Y14.LUTA.0	...	X1Y8.LUTC.0	X1Y8.LUTA.0	16 bits	X1Y7.LUTC.0	X1Y7.LUTA.0	...	X1Y0.LUTC.0	X1Y0.LUTA.0
32 bits Order Type	Type 0.0	Type 1.0	Type 0.0	Type 1.0	...	Type 0.0	Type 1.0	16 bits	Type 0.0	Type 1.0	...	Type 0.0	Type 1.0

Frames	27 (28)												
SLICEX (LUTs) 4 Frames	X1Y15.LUTC.1	X1Y15.LUTA.1	X1Y14.LUTC.1	X1Y14.LUTA.1	...	X1Y8.LUTC.1	X1Y8.LUTA.1	16 bits	X1Y7.LUTC.1	X1Y7.LUTA.1	...	X1Y0.LUTC.1	X1Y0.LUTA.1
32 bits Order Type	Type 0.1	Type 1.1	Type 0.1	Type 1.1	...	Type 0.1	Type 1.1	16 bits	Type 0.1	Type 1.1	...	Type 0.1	Type 1.1

Frames	28 (29)												
SLICEX (LUTs) 4 Frames	X1Y15.LUTD.0	X1Y15.LUTB.0	X1Y14.LUTD.0	X1Y14.LUTB.0	...	X1Y8.LUTD.0	X1Y8.LUTB.0	16 bits	X1Y7.LUTD.0	X1Y7.LUTB.0	...	X1Y0.LUTD.0	X1Y0.LUTB.0
32 bits Order Type	Type 0.0	Type 1.0	Type 0.0	Type 1.0	...	Type 0.0	Type 1.0	16 bits	Type 0.0	Type 1.0	...	Type 0.0	Type 1.0

Frames	29 (30)												
SLICEX (LUTs) 4 Frames	X1Y15.LUTD.1	X1Y15.LUTB.1	X1Y14.LUTD.1	X1Y14.LUTB.1	...	X1Y8.LUTD.1	X1Y8.LUTB.1	16 bits	X1Y7.LUTD.1	X1Y7.LUTB.1	...	X1Y0.LUTD.1	X1Y0.LUTB.1
32 bits Order Type	Type 0.1	Type 1.1	Type 0.1	Type 1.1	...	Type 0.1	Type 1.1	16 bits	Type 0.1	Type 1.1	...	Type 0.1	Type 1.1

Figura 87 – Distribuição das palavras de configuração das LUTs num *slice* SLICEX.

Os *frames* 26 e 27 (27 e 28 quando é uma coluna de *slices* SLICEM e SLICEX) configuram as LUTs LUTA e LUTC, ao passo que os *frames* 28 e 29 (29 e 30 no caso de *slices* SLICEM e SLICEX) configuram o conteúdo das LUTs LUTB e LUTD.

No início do *frame* 26 (ou 27), os primeiros 32 bits (duas palavras de 16 bits) configuram, com uma organização de bits *Type 0.0*, metade da LUTC pertencente ao *slice* X1Y15. Os 32 bits seguintes configuram metade da LUTA do mesmo *slice*, com uma organização de bits *Type 1.0*.

Seguem-se os 32 bits que configuram metade da LUTC do *slice* seguinte (X1Y14) e seguindo a mesma lógica, os seguintes 32 bits configuram a LUTA do mesmo *slice*.

Este encadeado repete-se até ao *slice* X1Y8, onde existe uma palavra de 16 bits de controlo, antes de continuar com os restantes oito *slices*. Estes 16 bits, embora ainda não detalhados neste estudo, são responsáveis por bits de paridade (ECC), que ajudam a garantir a integridade dos bits de configuração de cada *frame*.

Com 32 bits correspondentes à LUTC e LUTA do *slice* X1Y0, o *frame* 26 (ou 27) fica completo. Segue-se então o *frame* 27 (ou 28) que

irá ser responsável pelos restantes 32 bits das LUTCs e LUTAs.

O encadeado é o mesmo que no *frame* anterior (LUTC/LUTA/-LUTC...), sendo que neste *frame* as organizações dos bits serão *Type 0.1* e *Type 1.1*, para a LUTC e LUTA respetivamente.

Seguindo semelhante encadeamento, os *frames* 28 e 29 (ou 29 e 30) configuram as LUTs LUTB e LUTD. Sendo que no *frame* 28 (ou 29) cada grupo de 32 bits referente a uma LUTD segue uma organização de bits *Type 0.0*, e cada grupo de 32 bits relacionados com a LUTB segue uma organização de bits *Type 1.0*.

No *frame* 29 (ou 30), as organizações dos 32 bits é de *Type 0.1* para a outra metade da configuração das LUTDs e *Type 1.1* para as LUTBs.

Se em vez de ser ROW=0 e MAJOR=2 fosse ROW=1 e MAJOR=5, os *slices* configurados seriam de X5Y31 a X5Y16.

A Figura 88 mostra a organização dos *frames* 21 a 24, responsáveis pela configuração das LUTs dos *slices* tipo SLICEL.

Frames	21												
SLICEL (LUTs) 4 Frames	X0Y15.LUTD.0	X0Y15.LUTB.0	X0Y14.LUTD.0	X0Y14.LUTB.0	...	X0Y8.LUTD.0	X0Y8.LUTB.0	16 bits	X0Y7.LUTD.0	X0Y7.LUTB.0	...	X0Y0.LUTD.0	X0Y0.LUTB.0
32 bits Order Type	Type 1.0	Type 1.0	Type 1.0	Type 1.0	...	Type 1.0	Type 1.0	16 bits	Type 1.0	Type 1.0	...	Type 1.0	Type 1.0

Frames	22												
SLICEL (LUTs) 4 Frames	X0Y15.LUTD.1	X0Y15.LUTB.1	X0Y14.LUTD.1	X0Y14.LUTB.1	...	X0Y8.LUTD.1	X0Y8.LUTB.1	16 bits	X0Y7.LUTD.1	X0Y7.LUTB.1	...	X0Y0.LUTD.1	X0Y0.LUTB.1
32 bits Order Type	Type 1.1	Type 1.1	Type 1.1	Type 1.1	...	Type 1.1	Type 1.1	16 bits	Type 1.1	Type 1.1	...	Type 1.1	Type 1.1

Frames	23												
SLICEL (LUTs) 4 Frames	X0Y15.LUTC.0	X0Y15.LUTA.0	X0Y14.LUTC.0	X0Y14.LUTA.0	...	X0Y8.LUTC.0	X0Y8.LUTA.0	16 bits	X0Y7.LUTC.0	X0Y7.LUTA.0	...	X0Y0.LUTC.0	X0Y0.LUTA.0
32 bits Order Type	Type 0.0	Type 0.0	Type 0.0	Type 0.0	...	Type 0.0	Type 0.0	16 bits	Type 0.0	Type 0.0	...	Type 0.0	Type 0.0

Frames	24												
SLICEL (LUTs) 4 Frames	X0Y15.LUTC.1	X0Y15.LUTA.1	X0Y14.LUTC.1	X0Y14.LUTA.1	...	X0Y8.LUTC.1	X0Y8.LUTA.1	16 bits	X0Y7.LUTC.1	X0Y7.LUTA.1	...	X0Y0.LUTC.1	X0Y0.LUTA.1
32 bits Order Type	Type 0.1	Type 0.1	Type 0.1	Type 0.1	...	Type 0.1	Type 0.1	16 bits	Type 0.1	Type 0.1	...	Type 0.1	Type 0.1

Figura 88 – Distribuição das palavras de configuração das LUTs num *slice* SLICEL.

Tal como na distribuição ilustrada na Figura 87, este exemplo corresponde à primeira linha (ROW=0) e primeira coluna de CLBs que possuem *slices* SLICEL e SLICEX.

Seguindo a mesma lógica de encadeamento, o *frame* 21 começa com 32 bits dedicados à LUTD do *slice* X0Y15, com uma organização de bits *Type 1.0*, seguido por 32 bits dedicados à LUTB do mesmo *slice*, igualmente com uma organização *Type 1.0*.

No *frame* 22, o encadeamento mantém-se LUTD/LUTB/-LUTD... Mas para os restantes 32 bits de configuração de cada LUT (LUTB e LUTD), é organizada segundo a ordem *Type 1.1*.

Os *frames* 23 e 24 configuram as LUTs LUTA e LUTC, seguindo o encadeamento LUTC/LUTA/LUTC... Para a primeira metade dos 64 bits, tanto a LUTA como a LUTC, têm os bits organizados segundo *Type 0.0*, e os restantes 32 bits seguem a organização *Type 0.1*.

Exposto na Figura 89, o terceiro tipo de coluna de *slices*, o SLICEM, segue o mesmo encadeamento que o SLICEL (LUTD/LUTB/-LUTD... e LUTC/LUTA/LUTC...). No entanto, possuem organizações dos bits completamente diferentes (*Type 3.0* no *frame* 21, *Type 3.1* no *frame* 22, *Type 2.0* no *frame* 24 e *Type 2.1* no *frame* 25).

Frames	SLICEM (LUTs) 5 Frames	32 bits Order Type
21	X2Y15.LUTD.0	Type 3.0
	X2Y15.LUTB.0	Type 3.0
	X2Y14.LUTD.0	Type 3.0
	X2Y14.LUTB.0	Type 3.0
	...	...
	X2Y8.LUTD.0	Type 3.0
	X2Y8.LUTB.0	Type 3.0
	16 bits	16 bits
	X2Y7.LUTD.0	Type 3.0
	X2Y7.LUTB.0	Type 3.0
	...	...
	X2Y0.LUTD.0	Type 3.0
	X2Y0.LUTB.0	Type 3.0
22	X2Y15.LUTD.1	Type 3.1
	X2Y15.LUTB.1	Type 3.1
	X2Y14.LUTD.1	Type 3.1
	X2Y14.LUTB.1	Type 3.1
	...	...
	X2Y8.LUTD.1	Type 3.1
	X2Y8.LUTB.1	Type 3.1
	16 bits	16 bits
	X2Y7.LUTD.1	Type 3.1
	X2Y7.LUTB.1	Type 3.1
	...	...
	X2Y0.LUTD.1	Type 3.1
	X2Y0.LUTB.1	Type 3.1
23	1 Frame	
24	X2Y15.LUTC.0	Type 2.0
	X2Y15.LUTA.0	Type 2.0
	X2Y14.LUTC.0	Type 2.0
	X2Y14.LUTA.0	Type 2.0
	...	...
	X2Y8.LUTC.0	Type 2.0
	X2Y8.LUTA.0	Type 2.0
	16 bits	16 bits
	X2Y7.LUTC.0	Type 2.0
	X2Y7.LUTA.0	Type 2.0
	...	...
	X2Y0.LUTC.0	Type 2.0
	X2Y0.LUTA.0	Type 2.0
25	X2Y15.LUTC.1	Type 2.1
	X2Y15.LUTA.1	Type 2.1
	X2Y14.LUTC.1	Type 2.1
	X2Y14.LUTA.1	Type 2.1
	...	...
	X2Y8.LUTC.1	Type 2.1
	X0Y8.LUTA.1	Type 2.1
	16 bits	16 bits
	X2Y7.LUTC.1	Type 2.1
	X2Y7.LUTA.1	Type 2.1
	...	...
	X2Y0.LUTC.1	Type 2.1
	X2Y0.LUTA.1	Type 2.1

Figura 89 – Distribuição das palavras de configuração das LUTs num *slice* SLICEM.

Tal como já referido, devido à necessidade extra de configurações dos CLBs com *slices* SLICEM, é fundamental existir mais um *frame* para configurar este género de colunas de CLBs. Por esta razão existe o *frame* 23 que será detalhado adiante.

Se o número de bits necessários para preencher uma LUT é bem conhecido (64 bits), a ordem destes não o é. Sendo que como é visível nas Figuras 87, 88 e 89, a ordem é diferente para cada tipo de *slice* e LUT.

A Figura 90 mostra a ordem dos bits para cada um dos tipos de organização dos bits de configuração das LUTs.

Existem quatro tipos de organização de bits (*Type 0*, *Type 1*, *Type 2* e *Type 3*), sendo cada um deles composto por duas palavras de 32 bits.

É possível observar que qualquer tipo de organização não segue qualquer ordem linear e que nem sequer é possível separar os bits menos significativos dos mais significativos, pois tanto uns como outros estão dispersos pelas duas palavras de 32 bits de cada tipo de configuração.

TYPE																																
TYPE 0.0	50	51	54	55	62	63	58	59	34	35	38	39	46	47	42	43	18	19	22	23	30	31	26	27	2	3	6	7	14	15	10	11
TYPE 0.1	48	49	52	53	60	61	56	57	32	33	36	37	44	45	40	41	16	17	20	21	28	29	24	25	0	1	4	5	12	13	8	9
TYPE 1.0	11	10	15	14	7	6	3	2	27	26	31	30	23	22	19	18	43	42	47	46	39	38	35	34	59	58	63	62	55	54	51	50
TYPE 1.1	9	8	13	12	5	4	1	0	25	24	29	28	21	20	17	16	41	40	45	44	37	36	33	32	57	56	61	60	53	52	49	48
TYPE 2.0	2	3	7	6	14	15	11	10	18	19	23	22	30	31	27	26	34	35	39	38	46	47	43	42	50	51	55	54	62	63	59	58
TYPE 2.1	0	1	5	4	12	13	9	8	16	17	21	20	28	29	25	24	32	33	37	36	44	45	41	40	48	49	53	52	60	61	57	56
TYPE 3.0	58	59	63	62	54	55	51	50	42	43	47	46	38	39	35	34	26	27	31	30	22	23	19	18	10	11	15	14	6	7	3	2
TYPE 3.1	56	57	61	60	52	53	49	48	40	41	45	44	36	37	33	32	24	25	29	28	20	21	17	16	8	9	13	12	4	5	1	0

Figura 90 – Ordem dos bits dos vários tipos de organização.

A.4.3.2 Organização dos bits de configuração dos CLBs (excluindo LUTs)

Excluindo os bits responsáveis pelo conteúdo das LUTs, existe ainda uma quantidade considerável de configurações de cada CLB, que tem igualmente a sua própria organização.

Na Figura 85 é indicado que os *frames* 20 e 25 eram responsáveis pela configuração de 16 CLBs. As Figuras 91 e 92 indicam a função de cada bit conhecido nos *frames* 25 e 20 respetivamente (*ROW=0 MAJOR=2*).

Os primeiros 64 bits do *frame* 25 possuem bits de configuração do CLB que inclui os *slices* X0Y15 (SLICEL) e X1Y15 (SLICEX), os 64 bits seguintes do CLB com os *slices* X0Y14 e X1Y14, e assim sucessivamente até ao CLB com os *slices* X0Y0 e X1Y0. Existe entre o CLB que possui o par de *slices* X0Y8/X1Y8 e o CLB que possui o par X0Y7/X1Y7, 16 bits para controlo de erros de todo o *frame* 25.

Os bits assinalados com (*) pertencem a um vetor de bits, mas faltam encontrar os restantes bits. É o caso dos multiplexadores OUTMUX do SLICEX.

A função de cada bit repete-se portanto em cada grupo de 64 bits, para cada CLB.

Muitos dos bits do *frame* 20 não estão identificados (o mesmo

Bit														
31	X0Y15<DOUTMUX>	X1Y15<SRUSED>	X0Y14	X1Y14		X0Y8	X1Y8		X0Y7	X0Y7		X0Y1	X0Y0	X0Y0
30	X0Y15<DOUTMUX>	X1Y15<SYNC_ATTR>	X0Y14	X1Y14		X0Y8	X1Y8		X0Y7	X0Y7		X0Y1	X0Y0	X0Y0
29	X0Y15<DOUTMUX>	X1Y15<FF_LATCH>	X0Y14	X1Y14		X0Y8	X1Y8		X0Y7	X1Y7		X1Y1	X0Y0	X1Y0
28	X0Y15<DOUTMUX>	X0Y15<CLKINV>	X0Y14	X0Y14		X0Y8	X0Y8		X0Y7				X0Y0	
27	X0Y15<DFFMUX>	X0Y15<BFFMUX>	X0Y14	X0Y14		X0Y8	X0Y8		X0Y7	X0Y7		X0Y1	X0Y0	X0Y0
26	X0Y15<DFFMUX>	X0Y15<BFFMUX>	X0Y14	X0Y14		X0Y8	X0Y8		X0Y7	X0Y7		X0Y1	X0Y0	X0Y0
25	X0Y15<DFFMUX>	X0Y15<BFFMUX>	X0Y14	X0Y14		X0Y8	X0Y8		X0Y7	X0Y7		X0Y1	X0Y0	X0Y0
24	X0Y15<DFFMUX>	X0Y15<BFFMUX>	X0Y14	X0Y14		X0Y8	X0Y8		X0Y7	X0Y7		X0Y1	X0Y0	X0Y0
23	X0Y15<DCY0><O5>	X0Y15<BOUTMUX>	X0Y14	X0Y14	...	X0Y8	X0Y8		X1Y7	X0Y7	...	X0Y1	X1Y0	X0Y0
22	X1Y15<DOUTMUX>(*)	X0Y15<BOUTMUX>	X1Y14	X0Y14		X1Y8	X0Y8		X0Y7	X0Y7		X0Y1	X0Y0	X0Y0
21	X1Y15<COUTMUX>(*)	X0Y15<BOUTMUX>	X1Y14	X0Y14		X1Y8	X0Y8		X0Y7	X0Y7		X0Y1	X0Y0	X0Y0
20	X0Y15<DFFSRINIT>	X0Y15<BOUTMUX>	X0Y14	X0Y14		X0Y8	X0Y8		X0Y7	X0Y7		X0Y1	X0Y0	X0Y0
19	X0Y15<CFFSRINIT>	X1Y15<BFFMUX>	X0Y14	X1Y14		X0Y8	X1Y8		X0Y7	X0Y7		X0Y1	X0Y0	X0Y0
18	X1Y15<DFFSRINIT>	X1Y15<AFFMUX>	X1Y14	X1Y14		X1Y8	X1Y8		X1Y7	X0Y7		X0Y1	X1Y0	X0Y0
17	X0Y15<CCY0><O5>	X1Y15<BFFSRINIT>	X0Y14	X1Y14		X0Y8	X1Y8		X1Y7	X0Y7		X0Y1	X1Y0	X0Y0
16	X1Y15<BOUTMUX>(*)	X1Y15<AOUTMUX>(*)	X1Y14	X1Y14		X1Y8	X1Y8		X1Y7	X0Y7		X0Y1	X1Y0	X0Y0
15	X0Y15<COUTMUX>	X0Y15<BCY0><O5>	X0Y14	X0Y14		X0Y8	X0Y8	X0Y7	X1Y7	X0Y6		X0Y0	X1Y0	
14	X0Y15<COUTMUX>	X0Y15<PRECYINIT><AX>	X0Y14	X0Y14		X0Y8	X0Y8	X0Y7	X1Y7	X0Y6		X0Y0	X1Y0	
13	X0Y15<COUTMUX>	X1Y15<AFFSRINIT>	X0Y14	X1Y14		X0Y8	X1Y8	X0Y7	X1Y7	X0Y6		X0Y0	X1Y0	
12	X0Y15<COUTMUX>		X0Y14			X0Y8		X0Y7	X0Y7	X0Y6		X0Y0	X0Y0	
11	X0Y15<CFFMUX>	X0Y15<PRECYINIT><VCC>	X0Y14	X0Y14		X0Y8	X0Y8	X0Y7	X0Y7	X0Y6		X0Y0	X0Y0	
10	X0Y15<CFFMUX>	X0Y15<BFFSRINIT>	X0Y14	X0Y14		X0Y8	X0Y8	X0Y7	X0Y7	X0Y6		X0Y0	X0Y0	
9	X0Y15<CFFMUX>	X0Y15<ACY0><O5>	X0Y14	X0Y14		X0Y8	X0Y8	X0Y7	X0Y7	X0Y6		X0Y0	X0Y0	
8	X0Y15<CFFMUX>	X0Y15<AFFSRINIT>	X0Y14	X0Y14	...	X0Y8	X0Y8	X0Y7	X0Y7	X0Y6	...	X0Y0	X0Y0	
7	X1Y15<CLKINV>	X0Y15<AFFMUX>	X1Y14	X0Y14		X1Y8	X0Y8	X0Y7	X0Y7	X0Y6		X0Y0	X0Y0	
6	X0Y15<FF_LATCH>	X0Y15<AFFMUX>	X0Y14	X0Y14		X0Y8	X0Y8	X1Y7	X0Y7	X1Y6		X1Y0	X0Y0	
5	X0Y15<SRUSED>	X0Y15<AFFMUX>	X0Y14	X0Y14		X0Y8	X0Y8	X1Y7	X0Y7	X1Y6		X1Y0	X0Y0	
4	X0Y15<SYNC_ATTR>	X0Y15<AFFMUX>	X0Y14	X0Y14		X0Y8	X0Y8	X0Y7	X0Y7	X0Y6		X0Y0	X0Y0	
3	X0Y15<CEUSED>	X0Y15<AOUTMUX>	X0Y14	X0Y14		X0Y8	X0Y8	X0Y7	X1Y7	X0Y6		X0Y0	X1Y0	
2	X1Y15<DFFMUX>	X0Y15<AOUTMUX>	X1Y14	X0Y14		X1Y8	X0Y8	X1Y7	X1Y7	X1Y6		X1Y0	X1Y0	
1	X1Y15<CFFMUX>	X0Y15<AOUTMUX>	X1Y14	X0Y14		X1Y8	X0Y8	X0Y7	X1Y7	X0Y6		X0Y0	X1Y0	
0	X1Y15<CEUSED>	X0Y15<AOUTMUX>	X1Y14	X0Y14		X1Y8	X0Y8	X1Y7	X1Y7	X1Y6		X1Y0	X1Y0	

Figura 91 – Organização dos bits do *frame* 25 em colunas de CLBs com SLICEL e SLICEX.

sucede em todos os *frames*). Isto não significa forçosamente que eles não sejam utilizados, apenas poderá significar que ainda não foram devidamente identificados.

Na Figura 86, os *frames* 20, 23 e 26 configuram 16 CLBs com *slices* do tipo SLICEM e SLICEX. As tabelas presentes nas Figuras 93 e 94 indicam a função de cada bit conhecido nos *frames* 26 e 20 respetivamente, sendo os bits de configuração conhecidos próprios dos *slices*

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																X0Y15<D5FFSRINIT>	X1Y15<D5FFSRINIT>				X0Y15<C5FFSRINIT>	X1Y15<CFFSRINIT>	X1Y15<C5FFSRINIT>								
								X0Y15<B5FFSRINIT>		X1Y15<B5FFSRINIT>				X1Y15<A5FFSRINIT>	X0Y15<A5FFSRINIT>																
																X0Y14	X1Y14				X0Y14	X1Y14	X1Y14								
								X0Y14		X1Y14				X1Y14	X0Y14																
								.															.								
								.															.								
								.															.								
																X0Y8	X1Y8				X0Y8	X1Y8	X1Y8								
								X0Y8		X1Y8				X1Y8	X0Y8																
X0Y7	X1Y7				X0Y7	X1Y7	X1Y7																	X0Y7		X1Y7				X1Y7	X0Y7
								.															.								
								.															.								
								.															.								
X0Y0	X1Y0				X0Y0	X1Y0	X1Y0																	X0Y0		X1Y0				X1Y0	X0Y0

Figura 92 – Organização dos bits do *frame* 20 em colunas de CLBs com SLICEL e SLICEX.

SLICEM apresentados na Figura 95.

Nestas tabelas, como exemplo, foram usados os CLBs existentes na linha zero e coluna três (*ROW=0 MAJOR=3*).

A constituição do *frame* 26 é quase igual à do *frame* 25 exposto na Figura 91. A única diferença é o bit 8 da segunda palavra de 32 bits que configura o valor inicial do flip-flop A (AFFSRINIT) do SLICEL, e que é movido para o bit 18 da segunda palavra de 32 bits do *frame* 20, onde irá configurar o mesmo flip-flop existente no SLICEM.

A melhor forma de perceber cada um dos parâmetros de configuração de cada *slice* presentes nestes *frames* é consultar as Figuras 96,

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X2Y15<DOUTMUX>	X2Y15<DOUTMUX>	X2Y15<DOUTMUX>	X2Y15<DOUTMUX>	X2Y15<DFFMUX>	X2Y15<DFFMUX>	X2Y15<DFFMUX>	X2Y15<DFFMUX>	X2Y15<DCY0><O5>	X3Y15<DOUTMUX>(*)	X3Y15<COUTMUX>(*)	X2Y15<DFFSRINIT>	X2Y15<CFFSRINIT>	X3Y15<DFFSRINIT>	X2Y15<CCY0><O5>	X3Y15<BOUTMUX>(*)	X2Y15<COUTMUX>	X2Y15<COUTMUX>	X2Y15<COUTMUX>	X2Y15<COUTMUX>	X2Y15<CFFMUX>	X2Y15<CFFMUX>	X2Y15<CFFMUX>	X2Y15<CFFMUX>	X3Y15<CLKINV>	X2Y15<FF_LATCH>	X2Y15<SRUSED>	X2Y15<SYNC_ATTR>	X2Y15<CEUSED>	X3Y15<DFFMUX>	X3Y15<CFFMUX>	X3Y15<CEUSED>
X3Y15<SRUSED>	X3Y15<SYNC_ATTR>	X3Y15<FF_LATCH>	X2Y15<CLKINV>	X2Y15<BFFMUX>	X2Y15<BFFMUX>	X2Y15<BFFMUX>	X2Y15<BFFMUX>	X2Y15<BOUTMUX>	X2Y15<BOUTMUX>	X2Y15<BOUTMUX>	X2Y15<BOUTMUX>	X3Y15<BFFMUX>	X3Y15<AFFMUX>	X3Y15<BFFSRINIT>	X3Y15<AOUTMUX>(*)	X2Y15<BCY0><O5>	X2Y15<PRECYINIT><AX>	X3Y15<AFFSRINIT>		X2Y15<PRECYINIT><VCC>	X2Y15<BFFSRINIT>	X2Y15<ACY0><O5>		X2Y15<AFFMUX>	X2Y15<AFFMUX>	X2Y15<AFFMUX>	X2Y15<AFFMUX>	X2Y15<AOUTMUX>	X2Y15<AOUTMUX>	X2Y15<AOUTMUX>	X2Y15<AOUTMUX>
X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X3Y14	X3Y14	X2Y14	X2Y14	X3Y14	X2Y14	X3Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X3Y14	X2Y14	X2Y14	X2Y14	X2Y14	X3Y14	X3Y14	X3Y14
X3Y14	X3Y14	X3Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X3Y14	X3Y14	X3Y14	X3Y14	X2Y14	X2Y14	X3Y14		X2Y14	X2Y14	X2Y14		X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14	X2Y14
								.															.								
								.															.								
								.															.								
X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X3Y8	X3Y8	X2Y8	X2Y8	X3Y8	X2Y8	X3Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X3Y8	X2Y8	X2Y8	X2Y8	X2Y8	X3Y8	X3Y8	X3Y8
X3Y8	X3Y8	X3Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X3Y8	X3Y8	X3Y8	X3Y8	X2Y8	X2Y8	X3Y8		X2Y8	X2Y8	X2Y8		X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8	X2Y8
																X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X3Y7	X3Y7	X2Y7	X2Y7	X3Y7	X2Y7	X3Y7
X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X3Y7	X2Y7	X2Y7	X2Y7	X2Y7	X3Y7	X3Y7	X3Y7	X3Y7	X3Y7	X3Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X3Y7	X3Y7	X3Y7	X3Y7
X2Y7	X2Y7	X3Y7		X2Y7	X2Y7	X2Y7		X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y7	X2Y6	X2Y6	X2Y6	X2Y6	X2Y6	X2Y6	X2Y6	X2Y6	X2Y6	X3Y6	X1Y6	X2Y6	X2Y6	X3Y6	X2Y6	X3Y6
								.															.								
								.															.								
								.															.								
X2Y1	X2Y1	X3Y1		X2Y1	X2Y1	X2Y1		X2Y1	X2Y1	X2Y1	X2Y1	X2Y1	X2Y1	X2Y1	X2Y1	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X3Y0	X3Y0	X2Y0	X2Y0	X3Y0	X2Y0	X3Y0
X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X3Y0	X2Y0	X2Y0	X2Y0	X2Y0	X3Y0	X3Y0	X3Y0	X3Y0	X3Y0	X3Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X3Y0	X3Y0	X3Y0	X3Y0
X2Y0	X2Y0	X3Y0		X2Y0	X2Y0	X2Y0		X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0	X2Y0																

Figura 93 – Organização dos bits do *frame* 26 em colunas de CLBs com SLICEM e SLICEX.

97 e 98.

Para um melhor entendimento, recomenda-se o recurso à ferramenta *FPGA Editor* da Xilinx.

As próximas tabelas contêm as combinações de bits de seleção dos dois principais multiplexadores existentes em cada *slice*.

Na Tabela 22, estão definidas as combinações para selecionar a saída dos multiplexadores OUTMUX.

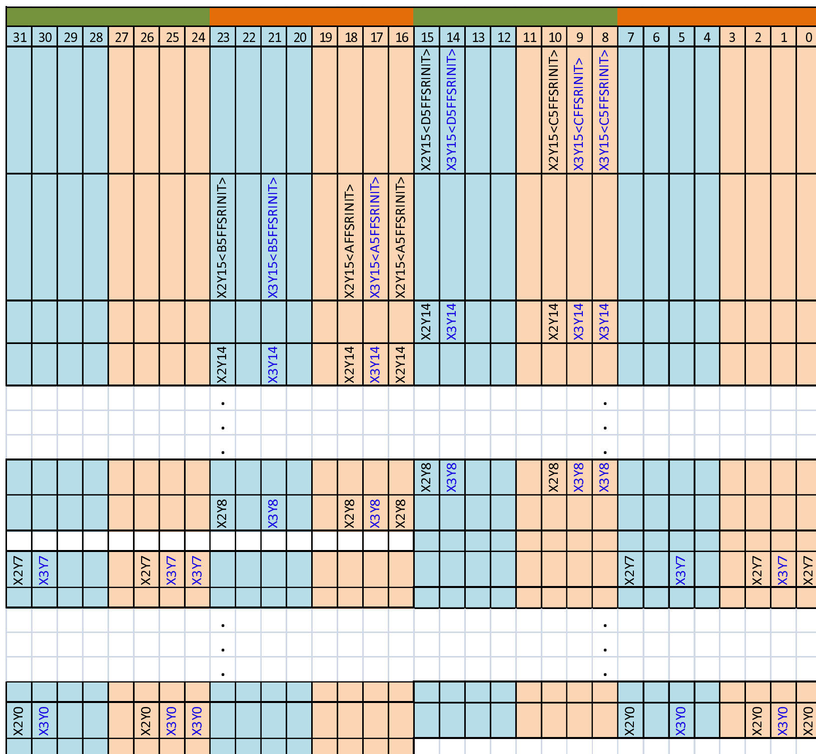


Figura 94 – Organização dos bits do *frame* 20 em colunas de CLBs com SLICEM e SLICEX.

Tabela 22 – Configurações dos multiplexadores OUTMUX nos *slices* SLICEL e SLICEM.

AOUTMUX		BOUTMUX		COUTMUX		DOUTMUX	
Saída	**Seleção**	**Saída**	**Seleção**	**Saída**	**Seleção**	**Saída**	**Seleção**
NS	"0000"	NS	"0000"	NS	"0000"	NS	"0000"
A5Q	"1000"	B5Q	"0100"	C5Q	"0010"	D5Q	"0001"
F7	"1100"	F8	"1100"	F7	"0011"	- -	- -
CY	"0110"	CY	"1010"	CY	"1001"	CY	"0110"
XOR	"0010"	XOR	"0010"	XOR	"1000"	XOR	"0100"
O5	"0101"	O5	"1001"	O5	"0101"	O5	"1010"
O6	"0001"	O6	"0001"	O6	"0100"	O6	"1000"

Em relação às combinações usadas para selecionar a saída dos

multiplexadores FFMUX estas estão organizadas na Tabela 23.

As combinações presentes nestas duas tabelas (Tabela 22 e Tabela 23) são válidas tanto para *slices* do tipo SLICEL como do tipo SLICEM. Ou seja, embora exista diferença internamente (entre A, B, C ou D), no global são iguais entre estes dois tipos de *slices*. Quando o multiplexador não é utilizado fica com a configuração NS (*No Select*).

Tabela 23 – Configurações dos multiplexadores FFMUX nos *slices* SLICEL e SLICEM.

AFFMUX		**BFFMUX**		**CFFMUX**		**DFFMUX**	
Saída	**Seleção**	**Saída**	**Seleção**	**Saída**	**Seleção**	**Saída**	**Seleção**
NS	"0000"	NS	"0000"	NS	"0000"	NS	"0000"
F7	"1011"	F8	"0111"	F7	"1110"	- -	- -
CY	"1101"	CY	"1110"	CY	"0111"	CY	"1011"
XOR	"1100"	XOR	"1100"	XOR	"0011"	XOR	"0011"
AX	"1010"	BX	"0101"	CX	"1010"	DX	"0101"
O5	"0001"	O5	"0010"	O5	"0100"	O5	"1000"
O6	"0000"	O6	"0000"	O6	"0000"	O6	"0000"

A Tabela 24 e a Tabela 25 contêm as opções de seleção da saída dos multiplexadores OUTMUX e FFMUX respetivamente, no caso dos *slices* SLICEX.

Tabela 24 – Configurações dos multiplexadores OUTMUX nos *slices* SLICEX.

AOUTMUX		**BOUTMUX**		**COUTMUX**		**DOUTMUX**	
Saída	**Seleção**	**Saída**	**Seleção**	**Saída**	**Seleção**	**Saída**	**Seleção**
NS	"?1?"	NS	"?1?"	NS	"?1?"	NS	"?1?"
A5Q	"?0?"	B5Q	"?0?"	C5Q	"?0?"	D5Q	"?0?"
O5	"?1?"	O5	"?1?"	O5	"?1?"	O5	"?1?"
O6	"?1?"	O6	"?1?"	O6	"?1?"	O6	"?1?"

A Tabela 25 contém as opções de seleção dos multiplexadores FFMUX, também para o SLICEX.

A Figura 95 mostra os bits identificados no *frame* 23 (SLICEM).

Tabela 25 – Configurações dos multiplexadores FFMUX nos *slices* SLICEX.

AFFMUX		**BFFMUX**		**CFFMUX**		**DFFMUX**	
Saída	**Seleção**	**Saída**	**Seleção**	**Saída**	**Seleção**	**Saída**	**Seleção**
NS	‘0’	NS	‘0’	NS	‘0’	NS	‘0’
AX	‘1’	BX	‘1’	CX	‘1’	DX	‘1’
O6	‘0’	O6	‘0’	O6	‘0’	O6	‘0’

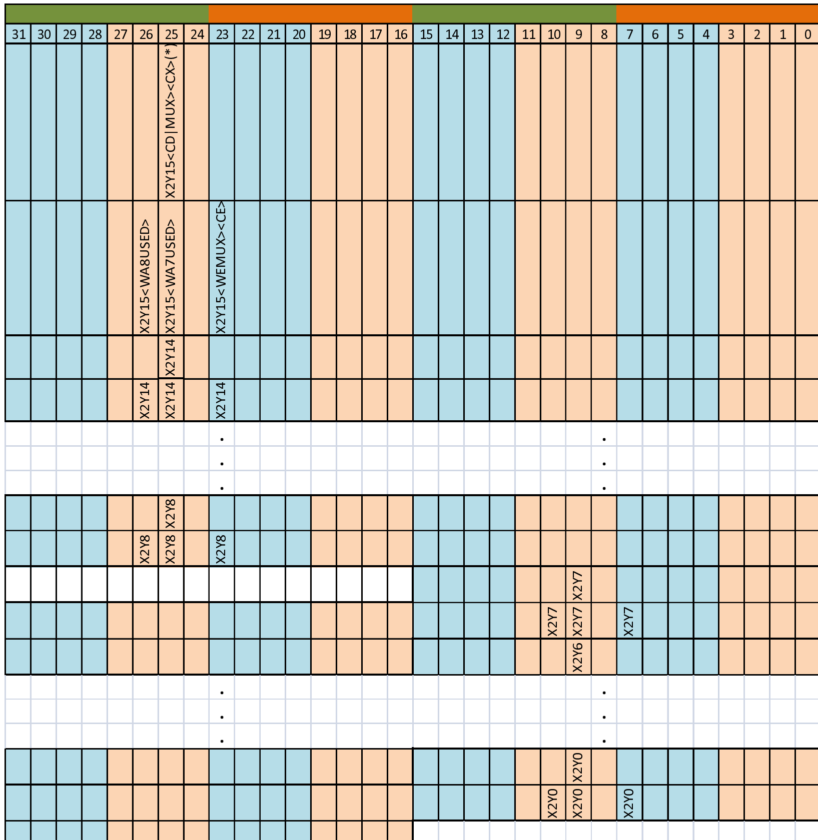


Figura 95 – Organização dos bits do *frame* 23 em colunas de CLBs com SLICEM e SLICEX.

A Figura 96 contém o diagrama interno de um SLICEX.

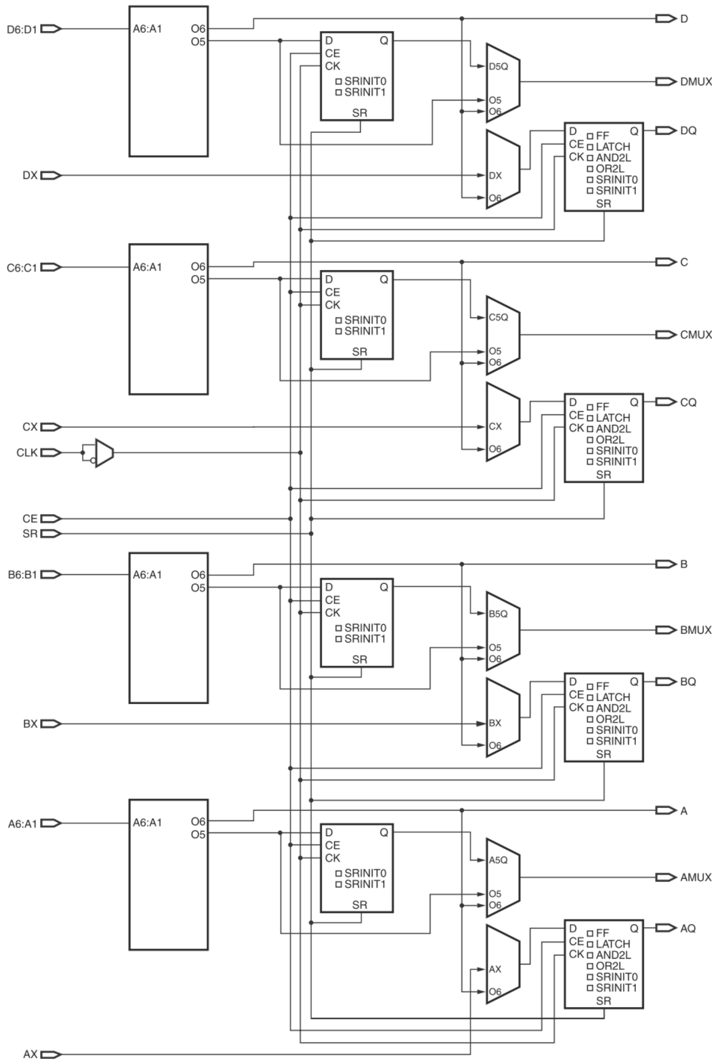


Figura 96 – Diagrama de um SLICEX (Xilinx Inc., 2010).

O diagrama do SLICEL (Figura 97), em relação ao do SLICEX (Figura 96), possui também uma linha de *carry* e capacidade de operações aritméticas (XOR).

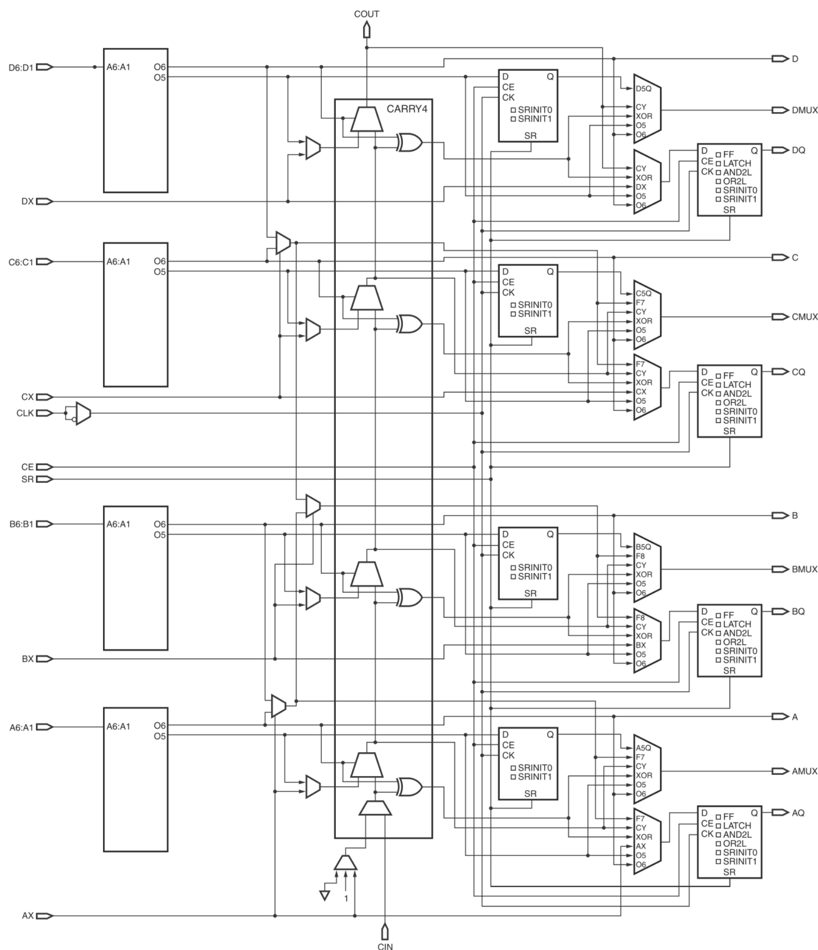


Figura 97 – Diagrama de um SLICEL (Xilinx Inc., 2010).

O diagrama do SLICEM (Figura 98), em relação ao do SLICEL (Figura 97), possui também a capacidade de usar as LUTs como memória (ROM ou RAM), ou SRL.

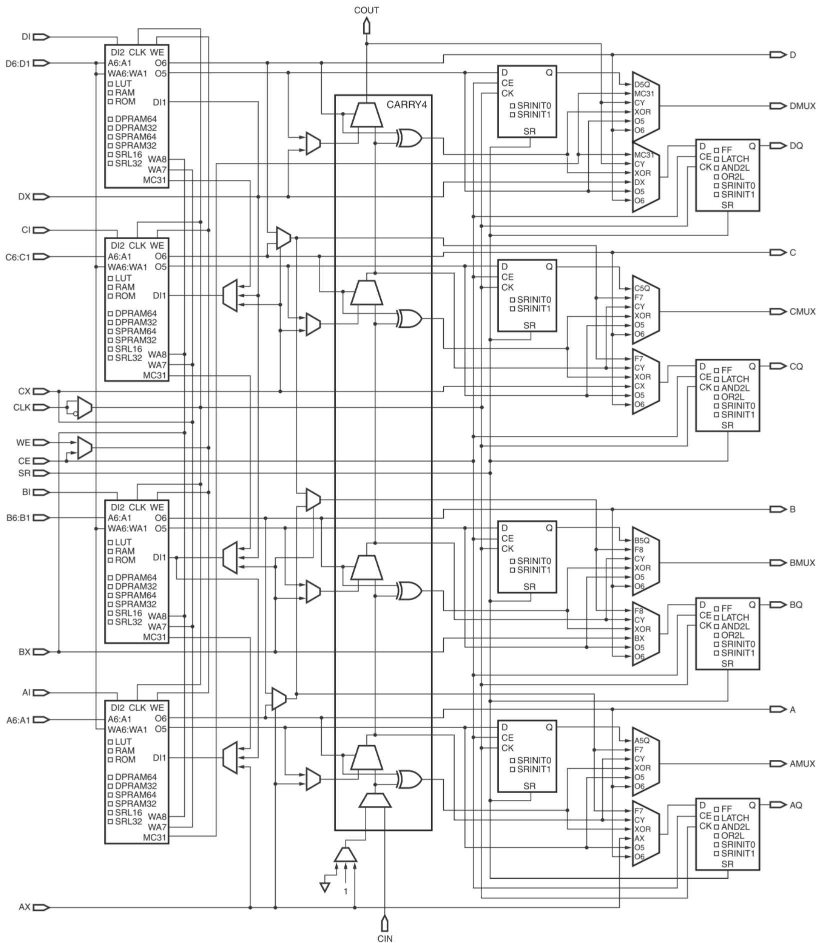


Figura 98 – Diagrama de um SLICEM (Xilinx Inc., 2010).

A.4.4 Organização da configuração das colunas de 8+4 BRAMs

Outras colunas que também foram estudadas em pormenor foram as colunas responsáveis pela configuração das BRAMs.

As colunas apresentadas nesta secção, apenas configuram as características e comportamentos das BRAMs, sendo o seu conteúdo configurado noutra zona de memória do *bit file*, como irá ser detalhado

na Secção A.4.5.

A Figura 99 mostra-nos a distribuição dos 25 *frames* desta coluna.

MINOR	0 - 19	20 - 21	22 - 24
BRAM Column 25 Frames	Switchs (20 Frames)	RAMB16.Settings (2 Frames)	RAMB8.Settings (3 Frames)

Figura 99 – Distribuição dos *frames* numa coluna de BRAMs.

Tal como as colunas que configuram os CLBs, estas colunas têm os primeiros 20 *frames* reservados para configurar as interligações das próprias BRAMs com os restantes recursos da FPGA.

Os *frames* 20 e 21 configuram quatro RAMB16s existentes em cada pedaço de coluna de BRAMs, ao passo que os *frames* 22 a 24 configuram oito RAMB8s.

Detalhes sobre a organização dos bits que compõem os *frames* 20 a 24 serão descritos mais abaixo.

Usando a primeira secção de BRAMs (linha zero (ROW=0) e coluna quatro ($MAJOR$=4)) para exemplificar a organização dos bits de configuração das BRAMs, esta é responsável por configurar as características de funcionamento de quatro RAMB16s e oito RAMB8s.

No caso das RAMB16s, cada uma usa 256 bits em cada *frame* utilizado para este tipo de BRAMs. Sendo que os primeiros 256 bits são para o *slice* de RAMB16 X0Y3, os segundos 256 bits para o X0Y2, até aos últimos 256 bits que são usados pela RAMB16 com o *slice* X0Y0.

Em relação às RAMB8s, cada uma tem para si reservados 128 bits de cada *frame* usado para o seu grupo de BRAMs. E segue o mesmo tipo de distribuição que as RAMB16s, ou seja, agora em grupos de 128 bits, primeiro o *slice* de RAMB8 com o índice Y superior (X0Y7), e

depois por ordem decrescente até ao X0Y0.

A Figura 100 mostra em que BRAMs são usados os *frames* 20 a 24.

Frames	RAMBs Settings 5 Frames
20	X0Y3.RAMB16 (256 bits)
	X0Y2.RAMB16 (256 bits)
	16 bits
	X0Y1.RAMB16 (256 bits)
	X0Y0.RAMB16 (256 bits)
21	X0Y3.RAMB16 (256 bits)
	X0Y2.RAMB16 (256 bits)
	16 bits
	X0Y1.RAMB16 (256 bits)
	X0Y0.RAMB16 (256 bits)
22	X0Y7.RAMB8 (128 bits)
	X0Y6.RAMB8 (128 bits)
	X0Y5.RAMB8 (128 bits)
	X0Y4.RAMB8 (128 bits)
	16 bits
	X0Y3.RAMB8 (128 bits)
	X0Y2.RAMB8 (128 bits)
	X0Y1.RAMB8 (128 bits)
	X0Y0.RAMB8 (128 bits)
23	X0Y7.RAMB8 (128 bits)
	X0Y6.RAMB8 (128 bits)
	X0Y5.RAMB8 (128 bits)
	X0Y4.RAMB8 (128 bits)
	16 bits
	X0Y3.RAMB8 (128 bits)
	X0Y2.RAMB8 (128 bits)
	X0Y1.RAMB8 (128 bits)
	X0Y0.RAMB8 (128 bits)
24	X0Y7.RAMB8 (128 bits)
	X0Y6.RAMB8 (128 bits)
	X0Y5.RAMB8 (128 bits)
	X0Y4.RAMB8 (128 bits)
	16 bits
	X0Y3.RAMB8 (128 bits)
	X0Y2.RAMB8 (128 bits)
	X0Y1.RAMB8 (128 bits)
	X0Y0.RAMB8 (128 bits)

Figura 100 – Distribuição das palavras de configuração das BRAMs.

A Figura 101 apresenta de um modo geral a distribuição dos bits num *frame* para cada *slice* de RAMB8. Como sempre, a meio de cada *frame*, existem sempre 16 bits de controlo de erros.

De ressalvar que para cada tipo de recurso da FPGA existe uma matriz de coordenadas distinta. Quer isto dizer que o *slice* X0Y0 de uma RAMB8 nada tem que ver com o *slice* X0Y0 de uma RAMB16 ou de CLB.

No momento em que foi redigido este estudo, ainda não tinha sido identificado nenhum bit relativo à configuração das RAMB16s, pelo que ainda não existem detalhes sobre os *frames* 20 e 21.

Na Figura 102 são apresentados os bits conhecidos para o *slice* RAMB8X0Y6. Este *frame* é preenchido com os valores que cada RAMB8 tem nos registos à sua saída após o carregamento do *bit file* e os valores que esses mesmos registos tomam após efetuar um *reset* à RAMB8.

Para melhor compreensão destes e de outros bits, é fortemente recomendada a consulta da informação disponibilizada pela Xilinx (Xilinx Inc., 2011c).

Já foi referenciado que no caso do *frame* 22, todas a BRAM8s de índice Y par possuem a mesma distribuição dos bits que a apresentada

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																													RAMB8X0Y7		
																													RAMB8X0Y7		
																													RAMB8X0Y7		
																													RAMB8X0Y7		
																													RAMB8X0Y6		
																													RAMB8X0Y6		
																													RAMB8X0Y6		
																													RAMB8X0Y6		
																													RAMB8X0Y5		
																													RAMB8X0Y5		
																													RAMB8X0Y5		
																													RAMB8X0Y5		
																													RAMB8X0Y4		
																													RAMB8X0Y4		
																													RAMB8X0Y4		
																													RAMB8X0Y4		
																													RAMB8X0Y3		
																													RAMB8X0Y3		
																													RAMB8X0Y3		
																													RAMB8X0Y3		
																													RAMB8X0Y2		
																													RAMB8X0Y2		
																													RAMB8X0Y2		
																													RAMB8X0Y2		
																													RAMB8X0Y1		
																													RAMB8X0Y1		
																													RAMB8X0Y1		
																													RAMB8X0Y1		
																													RAMB8X0Y0		
																													RAMB8X0Y0		
																													RAMB8X0Y0		
																													RAMB8X0Y0		

Figura 101 – Distribuição geral dos bits num *frame* de configuração de RAMB8s.

na Figura 102, no entanto falta ainda garantir se nos de índice ímpar a distribuição é a mesma, ou se existe alguma inversão na ordem, como acontece nos outros *frames* usados pelas RAMB8s.

O *frame* 23 possui exatamente essa particularidade de ter os bits de configuração dos *slices* com índice Y ímpar com a ordem inversa dos que possuem um índice Y par. Tal característica é bem visível na Figura 103.

Significa portanto que a RAMB8X0Y5 (ou outra de índice Y ímpar) terá exatamente a mesma distribuição dos bits no *frame* 23 que a RAMB8X0Y7, acontecendo o mesmo com as RAMB8s com índice Y par, neste caso usando a mesma distribuição que a RAMB8X0Y6.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RAMB8X0Y6<INIT_B[17]>		RAMB8X0Y6<INIT_B[17]>	RAMB8X0Y6<INIT_A[17]>		RAMB8X0Y6<SRVAL_A[17]>	RAMB8X0Y6<INIT_B[16]>		RAMB8X0Y6<SRVAL_B[16]>	RAMB8X0Y6<INIT_A[16]>		RAMB8X0Y6<SRVAL_A[16]>	RAMB8X0Y6<INIT_B[15]>		RAMB8X0Y6<SRVAL_B[15]>	RAMB8X0Y6<INIT_A[15]>		RAMB8X0Y6<SRVAL_A[15]>	RAMB8X0Y6<INIT_B[14]>		RAMB8X0Y6<SRVAL_B[14]>		RAMB8X0Y6<INIT_A[14]>	RAMB8X0Y6<SRVAL_A[14]>	RAMB8X0Y6<INIT_B[13]>		RAMB8X0Y6<SRVAL_B[13]>	RAMB8X0Y6<INIT_A[13]>		RAMB8X0Y6<SRVAL_A[13]>	RAMB8X0Y6<INIT_B[12]>	
RAMB8X0Y6<SRVAL_B[12]>	RAMB8X0Y6<INIT_A[12]>		RAMB8X0Y6<SRVAL_A[12]>		RAMB8X0Y6<INIT_B[11]>		RAMB8X0Y6<SRVAL_B[11]>	RAMB8X0Y6<INIT_A[11]>	RAMB8X0Y6<SRVAL_A[11]>	RAMB8X0Y6<INIT_B[10]>			RAMB8X0Y6<SRVAL_B[10]>	RAMB8X0Y6<INIT_A[10]>	RAMB8X0Y6<SRVAL_A[10]>	RAMB8X0Y6<INIT_B[9]>		RAMB8X0Y6<SRVAL_B[9]>	RAMB8X0Y6<INIT_A[9]>		RAMB8X0Y6<SRVAL_A[9]>										
									RAMB8X0Y6<INIT_B[8]>	RAMB8X0Y6<SRVAL_B[8]>	RAMB8X0Y6<INIT_A[8]>		RAMB8X0Y6<SRVAL_A[8]>		RAMB8X0Y6<INIT_B[7]>	RAMB8X0Y6<SRVAL_B[7]>	RAMB8X0Y6<INIT_A[7]>		RAMB8X0Y6<SRVAL_A[7]>	RAMB8X0Y6<INIT_B[6]>		RAMB8X0Y6<SRVAL_B[6]>	RAMB8X0Y6<INIT_A[6]>		RAMB8X0Y6<SRVAL_A[6]>	RAMB8X0Y6<INIT_B[5]>			RAMB8X0Y6<SRVAL_B[5]>	RAMB8X0Y6<INIT_A[5]>	RAMB8X0Y6<SRVAL_A[5]>
RAMB8X0Y6<INIT_B[4]>		RAMB8X0Y6<SRVAL_B[4]>	RAMB8X0Y6<INIT_A[4]>		RAMB8X0Y6<SRVAL_A[4]>		RAMB8X0Y6<INIT_B[3]>	RAMB8X0Y6<SRVAL_B[3]>	RAMB8X0Y6<INIT_A[3]>		RAMB8X0Y6<SRVAL_A[3]>		RAMB8X0Y6<INIT_B[2]>		RAMB8X0Y6<SRVAL_B[2]>	RAMB8X0Y6<INIT_A[2]>	RAMB8X0Y6<SRVAL_A[2]>		RAMB8X0Y6<INIT_B[1]>		RAMB8X0Y6<SRVAL_B[1]>	RAMB8X0Y6<INIT_A[1]>	RAMB8X0Y6<SRVAL_A[1]>	RAMB8X0Y6<INIT_B[0]>		RAMB8X0Y6<SRVAL_B[0]>	RAMB8X0Y6<INIT_A[0]>			RAMB8X0Y6<SRVAL_A[0]>	

Figura 102 – Organização dos bits da RAMB8X0Y6 no *frame* 22 em colunas de BRAMs.

Tanto o *frame* 23 como 24 incluem um conjunto variado de características passíveis de serem atribuídas a cada uma das BRAM8 (de um modo completamente independente entre elas). Tal como já mencionado, a sua verdadeira compreensão exige a leitura da informação correspondente da Xilinx (Xilinx Inc., 2011c). Sendo este estudo mais focado na localização de cada bit no *bit file*.

As próximas Figuras 104 e 105 mostram a posição de cada bit conhecido na RAMB8X0Y7 e RAMB8X0Y6 respetivamente.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	RAMB8X0Y7<WEAWEL0INV><WEAWEL0>	RAMB8X0Y7<REGCEAINV><REGCEA>	RAMB8X0Y7<WEAWEL1INV><WEAWEL1>	RAMB8X0Y7<RSTAINV><RSTA_B>	RAMB8X0Y7<CLKAWRCLKINV><CLKAWRCLK>		RAMB8X0Y7<CLKBRDCLKINV><CLKBRDCLK>		RAMB8X0Y7<WEBWEU1INV><WEBWEU1>	RAMB8X0Y7<RSTBRSTINV><RSTBRST>	RAMB8X0Y7<WEBWEU0INV><WEBWEU0>	RAMB8X0Y7<REGCEBREGCEINV><REGCEBREGCE>			RAMB8X0Y7<RAM_MODE><TDP>	RAMB8X0Y7<RAM_MODE><SDP>															
																													RAMB8X0Y7		
																													RAMB8X0Y7		
																													RAMB8X0Y7		
																													RAMB8X0Y6		
																													RAMB8X0Y6		
																													RAMB8X0Y6		
															RAMB8X0Y6<RAM_MODE><SDP>	RAMB8X0Y6<RAM_MODE><TDP>			RAMB8X0Y6<REGCEBREGCEINV><REGCEBREGCE>	RAMB8X0Y6<WEBWEU0INV><WEBWEU0>	RAMB8X0Y6<RSTBRSTINV><RSTBRST>	RAMB8X0Y6<WEBWEU1INV><WEBWEU1>		RAMB8X0Y6<CLKBRDCLKINV><CLKBRDCLK>		RAMB8X0Y6<CLKAWRCLKINV><CLKAWRCLK>	RAMB8X0Y6<RSTAINV><RSTA_B>	RAMB8X0Y6<WEAWEL1INV><WEAWEL1>	RAMB8X0Y6<REGCEAINV><REGCEA>	RAMB8X0Y6<WEAWEL0INV><WEAWEL0>	

Figura 103 – Bits de configuração da RAMB8X0Y7 e RAMB8X0Y6 no *frame* 23.

Tal como no *frame* 23 (e possivelmente no *frame* 22), os *slices* de RAMB8s possuem uma ordem dos bits para os que possuem um índice Y ímpar e outra para o par. Sendo sempre que uma ordem dos bits é o espelho da outra.

A.4.5 Organização da configuração do conteúdo das BRAMs

Nas secções anteriores foram estudados detalhes das oito linhas (*ROW*) que preenchem os 9088 *frames* base da configuração da

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			RAMB8X0Y7<WRITE_MODE_B><READ_FIRST>	RAMB8X0Y7<WRITE_MODE_A><READ_FIRST>	RAMB8X0Y7<WRITE_MODE_B><NO_CHANGE>	RAMB8X0Y7<WRITE_MODE_A><NO_CHANGE>	RAMB8X0Y7<DATA_WIDTH_B[2]>	RAMB8X0Y7<DATA_WIDTH_A[2]>	RAMB8X0Y7<DATA_WIDTH_B[1]>	RAMB8X0Y7<DATA_WIDTH_A[1]>	RAMB8X0Y7<DATA_WIDTH_B[0]>	RAMB8X0Y7<DATA_WIDTH_A[0]>	RAMB8X0Y7<RST_PRIOITY_B><CE>	RAMB8X0Y7<RST_PRIOITY_A><CE>	RAMB8X0Y7<RSTTYPE><ASYNC>(*)	RAMB8X0Y7<RSTTYPE><ASYNC>(*)	RAMB8X0Y7<DOB_REG>	RAMB8X0Y7<DOA_REG>							RAMB8X0Y7<EN_RSTRAM_A>						
																						RAMB8X0Y7<EN_RSTRAM_B>									RAMB8X0Y7<ENAWRENINV><ENAWREN>
RAMB8X0Y7<ENBRDENINV><ENBRDEN>																															

Figura 104 – Bits de configuração da RAMB8X0Y7 no *frame* 24.

FPGA. Para além desta matriz principal da configuração, existe outra responsável pelo conteúdo das BRAMs.

Esta parte do estudo ainda se encontra bastante incompleta, no entanto já foi obtida alguma informação, que se encontra registada na Figura 106.

Após os *frames* que preenchem as oito linhas de configurações, o *bit file* possui 256 bytes com o valor 'F'. Não é conhecido ainda o porquê destes 256 bytes, mas é regra eles aparecerem em todos os *bit files*.

Bit	Linha 1	Linha 2	Linha 3	Linha 4
31		RAMB8X0Y6<ENAWRENINV><ENAWREN>		
30				
29				
28				
27				
26				
25				RAMB8X0Y6<EN_RSTRAM_A>
24				
23				
22		RAMB8X0Y6<EN_RSTRAM_B>		
21				
20				
19				
18				RAMB8X0Y6<DOA_REG>
17				RAMB8X0Y6<DOB_REG>
16				RAMB8X0Y6<RSTTYPE><ASYNC>(*)
15				RAMB8X0Y6<RSTTYPE><ASYNC>(*)
14				RAMB8X0Y6<RST_PRIOITY_A><CE>
13				RAMB8X0Y6<RST_PRIOITY_B><CE>
12				RAMB8X0Y6<DATA_WIDTH_A[0]>
11				RAMB8X0Y6<DATA_WIDTH_B[0]>
10				RAMB8X0Y6<DATA_WIDTH_A[1]>
9				RAMB8X0Y6<DATA_WIDTH_B[1]>
8				RAMB8X0Y6<DATA_WIDTH_A[2]>
7				RAMB8X0Y6<DATA_WIDTH_B[2]>
6				RAMB8X0Y6<WRITE_MODE_A><NO_CHANGE>
5				RAMB8X0Y6<WRITE_MODE_B><NO_CHANGE>
4				RAMB8X0Y6<WRITE_MODE_A><READ_FIRST>
3				RAMB8X0Y6<WRITE_MODE_B><READ_FIRST>
2				
1				
0	RAMB8X0Y6<ENBRDENINV><ENBRDEN>			

Figura 105 – Bits de configuração da RAMB8X0Y6 no *frame* 24.

O conteúdo da RAMB8X0Y0 vem logo a seguir. Inclui 1152 bytes (1024 bytes + 128 bytes para paridade) que é precedido e sucedido de 18 bytes, para os quais ainda não foi identificada a sua função.

Aos bytes usados pela RAMB8X0Y0, sucedem-se os bytes aferidos à RAMB8X0Y1 e assim sucessivamente até à última RAMB8 da primeira coluna de BRAMs (RAMB8X0Y7).

Embora a mesma coluna possua quatro RAMB16, os bytes seguintes são preenchidos com o conteúdo das RAMB8s existentes nas outras três colunas de BRAMs da primeira linha (ROW=0).

Figura 106 – Organização do conteúdo das BRAMs no *bit file*.

A disposição dos bytes após os que são ocupados pelo conteúdo da RAMB8X3Y7 não é ainda conhecida. Poderá ocorrer que se continue a usar os bytes seguintes para RAMB8 das linhas seguintes, ou por outro lado, que se proceda ao uso do próximo espaço de memórias para o conteúdo das RAMB16s da mesma linha, antes de saltar para as colunas de BRAMS da linha seguinte.

A.5 CONCLUSÕES

Embora bastante trabalhosa, a obtenção das informações registadas neste documento foram direta ou indiretamente útil.

A nível de conhecimento de detalhes da configuração de CLBs, embora ainda incompleto, este já agrupa uma grande quantidade de informação relativa à organização dos bits que os configuram.

Em relação à configuração das BRAMs, por decisão de alteração do foco de trabalho, esta encontra-se algo incompleta, principalmente na parte do conteúdo das mesmas. No entanto, o processo para obtenção dessas informações já se encontra identificado, pelo que num projeto futuro, em que essa informação venha a ser exigida, a sua obtenção será bem mais direta e rápida, do que seria caso este estudo não tivesse sido realizado.

No decorrer deste estudo foi-se evidenciando que a família Spartan-6 distingue-se das restantes famílias em pormenores e características que são muito importantes para este trabalho. A diferença que mais sobressaiu e que de certo modo fez lamentar a escolha da família Spartan-6 como objeto de estudo é a falta das capacidades de

controlabilidade e observabilidade do valor dos flip-flops existentes nos CLBs. A Figura 107 ajuda a compreender melhor esta diferença.

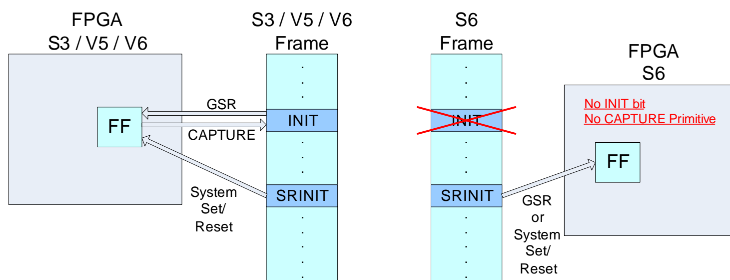


Figura 107 – Diferença entre a família Spartan-6 e outras famílias em relação ao carregamento do valor inicial dos flip-flops.

Talvez para conseguir uma família de mais baixo custo, a Xilinx retirou parte das capacidades de reconfiguração dinâmica, e nessa redução decidiu retirar o hardware que permitia ler e escrever os valores presentes nos flip-flops. Nas famílias Virtex-5, Virtex-6 e até mesmo na antiga família Spartan-3, as FPGAs possuem para cada flip-flop um bit correspondente na memória de configuração, que é designado por INIT. Quando se recorre ao comando GSR, os valores presentes nos bits INIT são transferidos para o estado dos flip-flops. Em sentido inverso, quando é dado à FPGA um comando CAPTURE, o presente estado nos flip-flops é transferido para os bits INIT da memória de configuração. Com estes dois comandos e lendo/escrevendo devidamente na memória de configuração, é possível controlar e observar os estados de todos os flip-flops existentes no dispositivo.

Na família Spartan-6 não existe INIT bit nem o comando CAPTURE. Apenas existe o bit SRINIT, que é o bit que é carregado no flip-flop quando acontece um normal *Set* ou *Reset* no sistema que está implementado na FPGA. Enquanto nas outras famílias existe o INIT para o valor inicial do flip-flop e o SRINIT para o valor de *Set* ou *Reset*, na Spartan-6 o valor SRINIT é igualmente o valor inicial.

Por esta razão, após a conclusão deste estudo, a família Spartan-6 foi abandonada, sendo substituída pela família Virtex-5. No entanto,

as famílias Spartan-6, Virtex-5, Virtex-6 e mesmo as recentes famílias 7 Series têm em comum uma estrutura *Tile-based* (Shih and Hsiung, 2009). Isto significa que em todas elas, a organização da memória de configuração e respetiva forma de endereçamento segue as mesmas regras básicas, pelo que a informação obtida neste estudo mantém-se útil para as famílias mais recentes. Será no entanto sempre necessário fazer as devidas adaptações, tais como: o tamanho do *frame*, quantos *frames* são necessários para configurar um segmento de CLBs (ou outro tipo de recursos), ou quantos CLBs estão incluídos num segmento.