UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

ANDRÉ FLORES DOS SANTOS

Análise do Uso de Redundância em Circuitos Gerados por Síntese de Alto Nível para FPGA Programado por SRAM sob Falhas Transientes

Porto Alegre
2017

ANDRÉ FLORES DOS SANTOS

Análise do Uso de Redundância em Circuitos Gerados por Síntese de Alto Nível para FPGA Programado por SRAM sob Falhas Transientes

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Microeletrônica, da Universidade Federal do Rio Grande do Sul, como parte dos requisitos para obtenção do título de Mestre em Microeletrônica.

Área de concentração: Tolerância a falhas em circuitos digitais.

ORIENTADOR: Fernanda Gusmão de Lima Kastensmidt.

Porto Alegre

2017

CIP - Catalogação na Publicação

Flores dos Santos, André
Análise do Uso de Redundância em Circuitos Gerados por Síntese de Alto Nível para FPGA Programado por SRAM sob Falhas Transientes / André Flores dos Santos. -- 2017.
103 f.
Orientadora: Fernanda Gusmão de Lima Kastensmidt.

Dissertação (Mestrado) -- Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR-RS, 2017.

1. Tolerância a Falhas. 2. Redundância Modular Tripla. 3. Síntese de Alto Nível. 4. FPGA. 5. Single Event Effects. I. Gusmão de Lima Kastensmidt, Fernanda, orient. II. Título.

DEDICATÓRIA

Dedico este trabalho aos meus pais, João Manoel Ribeiro dos Santos e Vera Flores dos Santos, meu irmão Thiago Flores dos Santos, minha irmã Vanessa Flores dos Santos que me apoiaram em todo esse tempo árduo longe da minha cidade e dos amigos.

AGRADECIMENTOS

Aos meus pais que sempre me apoiaram de todas as formas, emocionalmente, financeiramente e tiveram compreensão em todos os momentos difíceis em que passei. Sempre valorizaram o investimento intelectual e fizeram o máximo esforço para viabilizar meus estudos, mesmo muitas vezes não tendo condições.

A minha irmã Vanessa Flores dos Santos que dividiu moradia comigo por quase todo o tempo em que estive na Pós-graduação, com momentos de amizade e companheirismo.

A UFRGS e o programa de Pós-graduação em Microeletrônica, por viabilizar minha formação acadêmica. A capes pela bolsa de estudos concedida.

A minha orientadora, Prof. Dr. Fernanda Lima Kastensmidt, por me aceitar como aluno, pelos seus ensinamentos, confiança e me incentivar a dar o melhor de mim.

Aos Professores Ricardo Reis, Henri Ivanov Boudinov, Paolo Rech, Renato Perez Ribas, Tiago Balen, José Rodrigo Azambuja, entre outros.

Aos colegas de laboratório Jorge Lucio Tonfat, Lucas Antunes Tambara, Ádria de Barros, Filipe Lins, Genaro Rodrigues, Fabiano Libano, Augusto Wankler, Alexandra Zimpeck, entre outros.

Aos meus professores do Centro Universitário Franciscano Mirkos Ortiz, Ana Paula Canal, Alessandro Mainardi, Alexandre Zamberlan, Reiner Perozzo entre outros que me apoiaram na escolha do meu futuro.

A todos os meus amigos e parentes.

RESUMO

Este trabalho consiste no estudo e análise da suscetibilidade a efeitos da radiação em projetos de circuitos gerados por ferramenta de Síntese de Alto Nível para FPGAs (*Field Programmable Gate Array*), ou seja, circuitos programáveis e sistemas em chip, do inglês *System-on-Chip* (SOC). Através de um injetor de falhas por emulação usando o ICAP (*Internal Configuration Access Port)* localizado dentro do FPGA é possível injetar falhas simples ou acumuladas do tipo SEU (*Single Event Upset*), definidas como perturbações que podem afetar o funcionamento correto do dispositivo através da inversão de um bit por uma partícula carregada. SEU está dentro da classificação de SEEs (*Single Event Effects*), efeitos transitórios em tradução livre, podem ocorrer devido a penetração de partículas de alta energia do espaço e do sol (raios cósmicos e solares) na atmosfera da Terra que colidem com átomos de nitrogênio e oxigênio resultando na produção de partículas carregadas, na grande maioria nêutrons. Dentro deste contexto além de analisar a suscetibilidade de projetos gerados por ferramenta de Síntese de Alto Nível, torna-se relevante o estudo de técnicas de redundância como TMR (*Triple Modular Redundance*) para detecção, correção de erros e comparação com projetos desprotegidos verificando a confiabilidade. Os resultados mostram que no modo de injeção de falhas simples os projetos com redundância TMR demonstram ser efetivos. Na injeção de falhas acumuladas o projeto com múltiplos canais apresentou melhor confiabilidade do que o projeto desprotegido e com redundância de canal simples, tolerando um maior número de falhas antes de ter seu funcionamento comprometido.

Palavras-chaves: Field-Programable Gate Arrays (FPGAs), System-on-Chips (SoCs), Single Event Effects (SEEs), Single Event Upset (SEU), Triple Modular Redundance (TMR).

ABSTRACT

This work consists of the study and analysis of the susceptibility to effects of radiation in circuits projects generated by High Level Synthesis tool for FPGAs Field Programmable Gate Array (FPGAs), that is, system-on-chip (SOC). Through an emulation fault injector using ICAP (Internal Configuration Access Port), located inside the FPGA, it is possible to inject single or accumulated failures of the type SEU (Single Event Upset), defined as disturbances that can affect the correct functioning of the device through the inversion of a bit by a charged particle. SEU is within the classification of SEEs (Single Event Effects), can occur due to the penetration of high energy particles from space and from the sun (cosmic and solar rays) in the Earth's atmosphere that collide with atoms of nitrogen and oxygen resulting in the production of charged particles, most of them neutrons. In this context, in addition to analyzing the susceptibility of projects generated by a High Level Synthesis tool, it becomes relevant to study redundancy techniques such as TMR (Triple Modular Redundancy) for detection, correction of errors and comparison with unprotected projects verifying the reliability. The results show that in the simple fault injection mode TMR redundant projects prove to be effective. In the case of accumulated fault injection, the multichannel design presented better reliability than the unprotected design and with single channel redundancy, tolerating a greater number of failures before its operation was compromised.

Keywords: Field-Programable Gate Arrays (FPGAs), System-on-Chips (SoCs), Single Event Effects (SEEs), Single Event Upset (SEU), Triple Modular Redundance (TMR).

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ADIRU	Air Data Inertial Reference Unit
APSoC	All Programmable System-on-Chip
ASIC	Application Specific Integrated Circuits
AXI	Advanced eXtensible Interface Stream
ASIC	Application Specific Integrated Circuits
BRAM	Block Random Access Memory
CLB	Configurable Logic Block
CME	Coronal Mass Ejector
COTS	Components Off-The-Shelf
CED	Concurrent Error Detection
DMA	Direct Memory Acess
DRC	Design Rule Checking
DSP	Digital Signal Processor
DUT	Design Under Test
DWC	Duplication With Comparison
FI	Fault Injector
FIFO	First in First out
FIT	Failure in Time
FPGAs	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HDR	High Dose Hate
HLS	High Level Synthesis
ICAP	Internal Configuration Acess Port
IDE	Integrated Development Environment
IP	Intellectual Property
ISS	International Space Station
ITAR	International Traffic in Arms Regulations
LET	Linear Energy Transfer
LDR	Low Dose Rate

LUT	Look Up Table
MBU	Multiple Bit Upset
MWBF	Mean Workload Between Failure
RAM	Randon Acess Memory
RTL	Register Transfer Level
SDC	Silent Data Corruption
SEB	Single Event Burnout
SEFI	Single Event Functional Interrupt
SEE	Single Event Effects
SER	Soft Error Rate
SEGR	Single Event Gate Rupture
SEL	Single Event Latchup
SEU	Single Event Upset
SET	Single Event Transient
SRAM	Static Random Acess Memory
TID	Total Ionizing Doze
TMR	Triple Modular Redundance
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Description Language

1 INTRODUÇÃO

A evolução tecnológica da indústria de semicondutores tem seguido um padrão de miniaturização dos transistores e aumento da complexidade lógica. Essa tendência do setor indica a redução contínua das características de operação dos componentes, bem como o baixo consumo de energia. Muitos parâmetros dos componentes são dimensionados à medida que as tecnologias são reduzidas como: comprimento do canal, tamanho da junção, profundidade de junção, concentração de dopagem, espessura de óxido e tensão de alimentação (DOMINICK, 2008). A tendência de miniaturização dos circuitos integrados possibilitou a redução das dimensões das células de memória SRAM (*Static Random Acess Memory*) e suas tensões de operação, conforme a figura 1. A evolução da escala tecnológica acabou tornando os circuitos integrados mais suscetíveis a falhas que podem se originar de diferentes formas, tais como: erros de implementação, perturbação na alimentação do dispositivo, efeitos de envelhecimento (SRINIVASAN, 2008) e interação com o ambiente (interferência eletromagnética e efeitos da radiação) (TAMBARA, 2013).

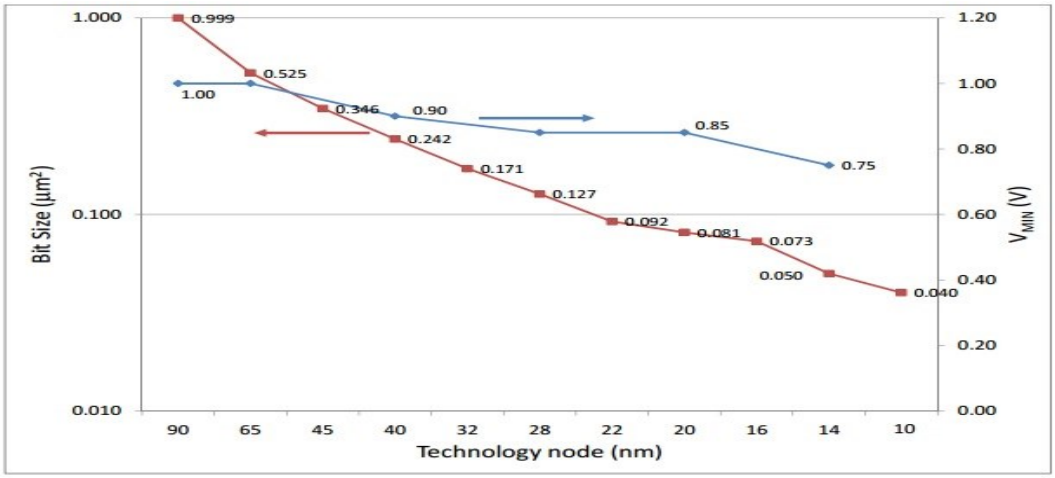


Figura 1 Redução da tensão de alimentação e da área ocupada por uma célula SRAM em relação ao nó tecnológico (ISSCC, 2016)

A evolução dos circuitos programáveis conhecidos como FPGAs (*Field Programmable Gate Array*) programados por SRAM tem acompanhado a tendência de miniaturização dos circuitos integrados e consequentemente, tornaram-se mais suscetíveis a serem perturbados por partículas carregadas.

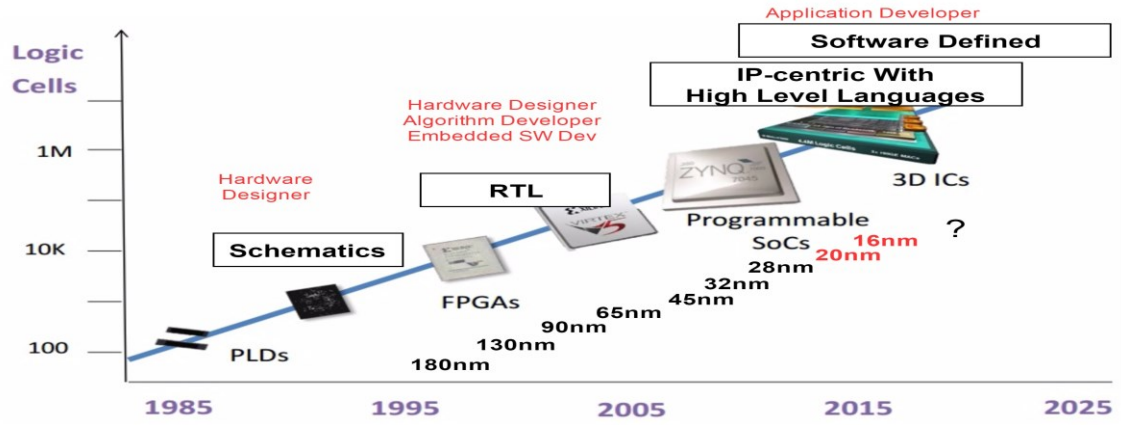


Figura 2 A evolução dos dispositivos programáveis, metodologia de projeto do FPGA, deslocamento de pessoas de desenvolvimento, escala tecnológica em nanômetro. (RAJE; AUTOR, 2015)

A figura 2 ilustra a evolução dos dispositivos programáveis em função da quantidade de células lógicas utilizadas pelo tempo; A evolução da metodologia de projetos em FPGAs (*Hardware Design, Algorithm Developer, Embedded SW Dev, Application Developer*) e evolução da escala do nó tecnológico. Na subseção deste capítulo 2.2 serão apresentados os efeitos da radiação cósmica em FPGAs.

A radiação cósmica está constantemente atingindo a Terra, partículas de alta energia do espaço e do Sol colidem com átomos de nitrogênio e oxigênio na atmosfera da Terra o que resulta na produção de partículas carregadas que podem gerar *Single Event Effects* (SEEs), na grande maioria essas partículas são nêutrons de alta velocidade. Eles viajam em alta velocidade pela atmosfera da Terra até colidirem com gases atmosféricos, objetos na superfície ou objetos que viajam na atmosfera da Terra (ACTEL, 2002).

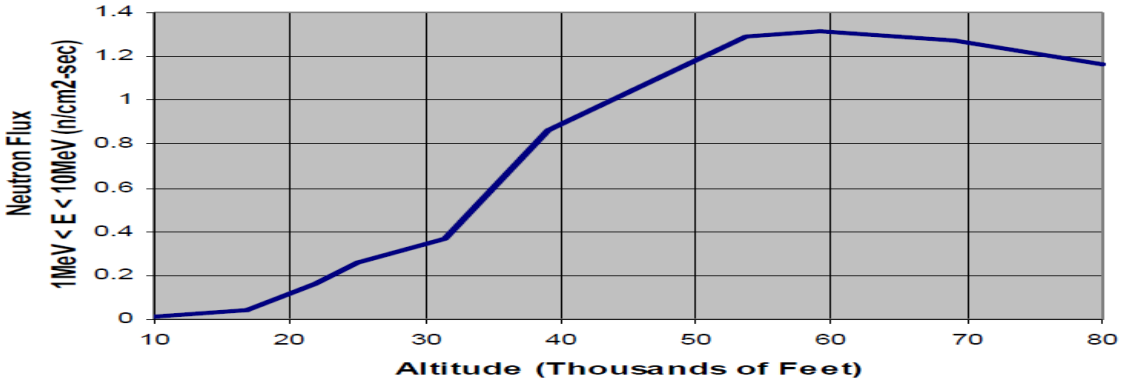


Figura 3 Fluxo de nêutrons em função de altitude (ACTEL, 2002)

O número de nêutrons ou fluxo de nêutrons presente na atmosfera terrestre provém de diversos fatores, sendo a altitude a mais significativa, visto que quanto menor o seu valor,

menor será o valor do fluxo de nêutrons que são atenuados por gases atmosféricos. O pico do fluxo de nêutrons ocorre a 60.000 pés, ilustrado na figura 3.

A Latitude é considerada outro fator importante na influência do valor de fluxo de nêutrons, já que o campo magnético da Terra aprisiona partículas cósmicas e os impede de colidir com gases atmosféricos. Quanto mais próximo da linha do equador se estiver, menor a probabilidade de partículas cósmicas penetrarem na atmosfera terrestre (ACTEL, 2002), na figura 4 é ilustrado o fluxo de nêutrons em diferentes altitudes e latitudes.

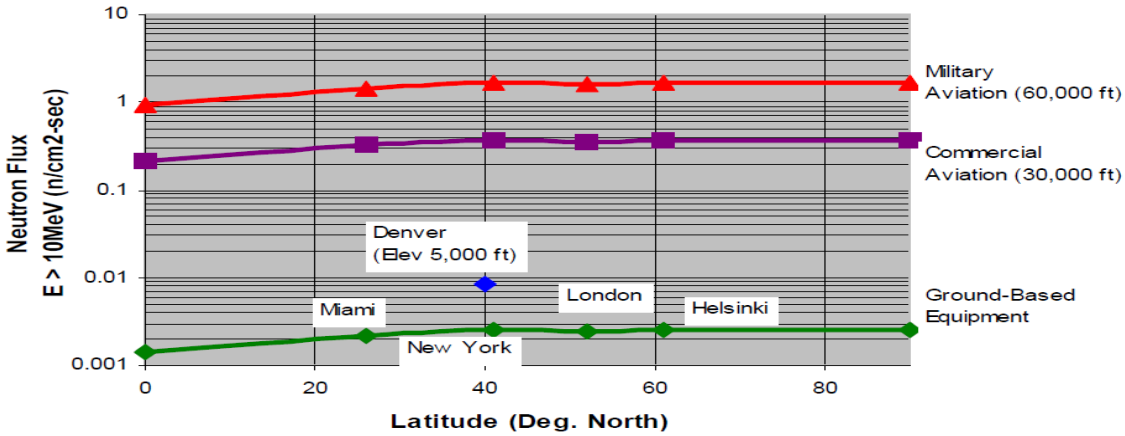


Figura 4 Latitude (Norte) (ACTEL, 2002)

Uma partícula energética pode depositar cargas num dispositivo semicondutor de duas maneiras. A primeira maneira é chamada de ionização direta, ou seja, pela partícula em si, ilustrada na figura 5(a), a segunda maneira é pela ionização indireta, ou seja, por partículas secundárias geradas pela colisão entre a partícula que está incidindo e o material, ilustrado na figura 5(b). Estes são os dois mecanismos físicos que podem levar ao mau funcionamento de um circuito devido à carga absorvida pelo material atingido (GOMES, 2014).

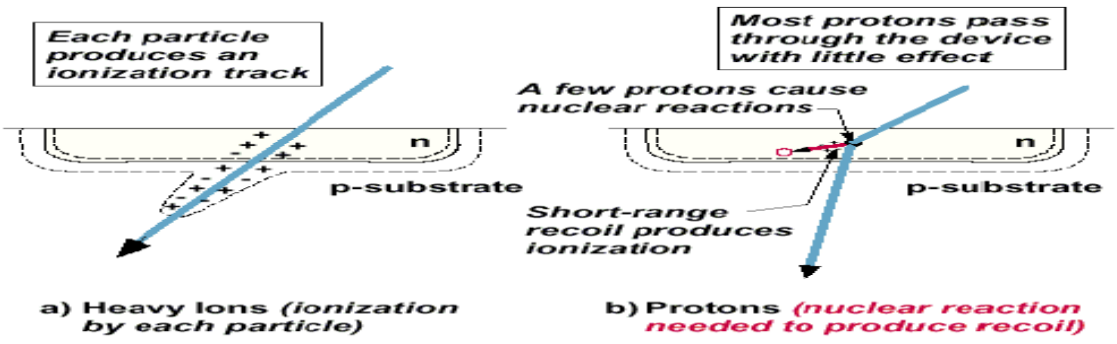
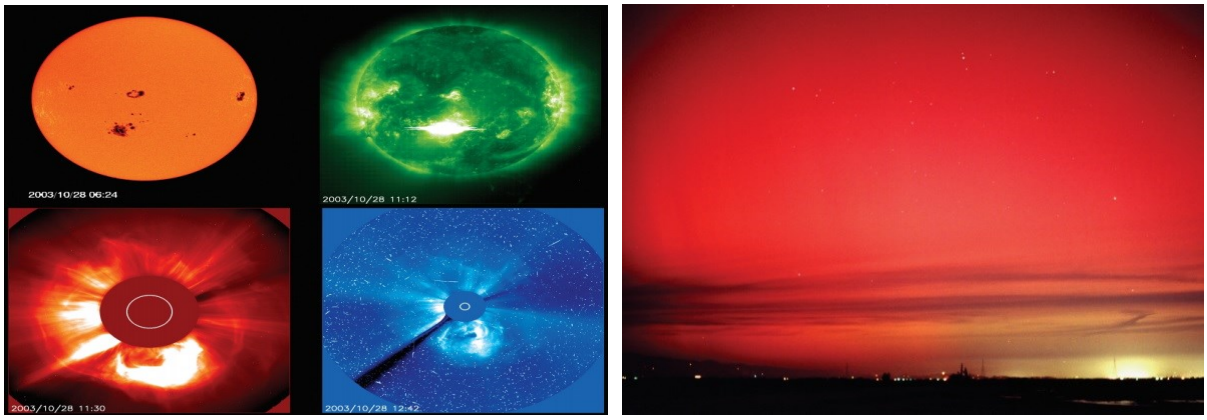


Figura 5 Mecanismos de deposição de carga de um SEE (GOMES, 2014)

A incidência da radiação pode interferir na confiabilidade de sistemas eletrônicos que são utilizados tanto no nível do mar quanto no nível espacial. Atualmente dispositivos eletrônicos são utilizados em larga escala por diversas aplicações, por exemplo, médica, aeroespacial, militar e automotiva, consideradas aplicações críticas. Tornando-se uma séria preocupação para os projetistas quanto a falhas causadas pela radiação. Neste contexto a qualificação de componentes eletrônicos tornou-se uma importante fase na caracterização de circuitos integrados sob os efeitos da radiação. Diversos experimentos têm demonstrado quanto à radiação afeta a confiabilidade das aplicações, por exemplo, experimentos com sistemas aviônicos em altitudes elevadas (NORMAND, 1996) e aplicações espaciais avaliando a dose de energia transferida por partículas em determinadas altitudes (ROTH, 1994). A radiação em altitudes de cruzeiro para aeronaves comerciais pode ser muito perigosa. Por exemplo, em sete de outubro de 2008, um avião a 37 mil pés teve sua altitude alterada de modo brusco sem comando, devido a dados incorretos enviados por uma das unidades inerciais de referência (ADIRU, *Air Data Inertial Reference Unit*) ferindo diversos passageiros, onde a ocorrência de SEU foi uma hipótese não descartada (ATSB, 2011). Em 2003, houve um período de intensa atividade solar onde foram registrados eventos de partículas solares de alta energia, criando tempestades geomagnéticas, ficaram conhecidas como Tempestades do *Haloween* de 2003, ilustradas na figura 6 (a). Em menos de uma hora após a erupção com ejeção de massa coronal (CME, *Coronal Mass Ejector*) e raio X intenso, as partículas energéticas solares aceleradas por uma onda de choque provocaram na terra uma tempestade geomagnética, com consequências como: correntes geomagneticamente induzidas em transformadores de energia elétrica, afetando seu funcionamento; desvio de rotas de aviões em tentativa de evitar as regiões de altas latitudes; proteção adicional para os astronautas da ISS (*International Space Station*). O setor espacial foi um dos mais afetados com estimativa de 59% das missões impactadas (MACHADO, 2014). Estes eventos seguem um ciclo de aproximadamente onze anos e são responsáveis pela variação do clima espacial e do fluxo de partículas energéticas que atingem a Terra. A figura 6 (b) ilustra o fenômeno chamado de Auroras Vermelhas de Baixa Latitude, características das tempestades geomagnéticas (BAKER, 2009).



(a) (b)

Figura 6 Erupção solar e Aurora Vermelha no Vale do Napa Califórnia, tempestade magnética de 2001 (BAKER, 2009)

FPGAs (*Field Programmable Gate Array*) são atrativos para aplicações críticas, pois apresentam alto desempenho, baixo consumo de potência, menor custo de produção em comparação com os circuitos integrados de aplicações específicas ASICS e podem ser reprogramados (ADELL, 2008). FPGAs podem ser programados através das tecnologias de memórias SRAM, FLASH ou EEPROM (MICROSEMI, 2011). Este trabalho enfocará FPGAs baseados em SRAM sem adentrar em mais detalhes sobre as outras tecnologias (FLASH e EEPROM).

Os FPGAs baseados em SRAM são principalmente suscetíveis a *Single Event Upset* (SEU) em seus bits da memória de configuração e células de memória incorporadas (TONFAT, 2016). A proteção de um sistema contra falhas transientes pode ser sucedida através de técnicas de tolerância a falhas aptas a detectar, corrigir ou mascarar as falhas transientes. Ordinariamente estas técnicas quando implementadas em *hardware*, são fundamentadas em redundância espacial ou temporal. O sistema pode sofrer alguns tipos de impactos de acordo com a técnica utilizada, por exemplo, aumento na área, recursos utilizados e no tempo de execução total do circuito. As técnicas de redundância para mitigação e proteção de sistemas mais tradicionais são: Duplicação com Comparação (DWC, *Duplication With Comparison*) e Redundância Modular Tripla (TMR, *Triple Modular Redundancy*), (GOMES, 2014). Trabalhos relacionados aplicam essas técnicas em nível de descrição de *hardware* (HDL) ou no nível de porta lógica. Porém, com o avanço dos circuitos programáveis e das novas gerações de APSoC (*All Programmable System-on-Chip*), faz-se necessário o uso de linguagens de mais alto nível para desenvolver circuitos nessas plataformas com maior agilidade e flexibilidade.

O aumento da necessidade computacional e a eficiência energética de sistemas embarcados têm crescido cada vez mais. Embora esteja disponível uma grande variedade de microprocessadores otimizados para diferentes desempenhos e eficiência energética, nenhuma destas soluções faz uso de todas as capacidades de otimização ao desenvolver um sistema para um cenário de aplicação específica. O uso de circuitos dedicados de aplicação específica projetados e implementados em FPGAs podem ser uma boa alternativa. É possível utilizar núcleos IP (*Intellectual Property*) de microprocessadores em conjunto com aceleradores de *hardware*, otimizados para atingir tanto o desempenho como potencia desejada. Os aceleradores de *hardware* podem ser descritos diretamente em linguagem de descrição de *hardware* HDL, ou mais recentemente, em linguagem de mais alto nível, usando *software* de Síntese de Alto Nível (HLS). Os aceleradores de *hardware* podem ser utilizados em conjunto com núcleos de processadores padrões, resultando em alto desempenho, eficiência energética, aplicações específicas e sistemas programáveis (HEMPEL, 2013).

A questão está em quanto esforço deve ser realizado para gerar bons resultados com o HDL, em termos de área e desempenho. Existem diversas possibilidades de otimizações que são aplicadas no algoritmo C, C++ com efeito direto no RTL. O desenvolvimento de aceleradores de *hardware* em Síntese de Alto Nível é vantajoso por apresentar um tempo menor na criação do RTL. Porém quanto à qualidade do projeto em termos de efetividade e confiabilidade contra falhas, o projetista deve escolher a melhor otimização através de testes e análise de resultados. Os projetos de sistemas para FPGAs requerem o uso específico de ferramentas de criação do fabricante, por exemplo, *Vivado Design Suite Tools* para síntese e roteamento do projeto para FPGAs da empresa Xilinx.

Nos últimos anos diversas abordagens para a geração automatizada de aceleradores de *hardware* baseados em FPGAs foram publicadas. ROCC 2.0 apresenta uma abordagem para acelerar as partes computacionais intensivas dos programas em C (VILLARREAL, 2010). Esta abordagem exige a identificação manual das seções críticas em desempenho do código e define alguns objetivos de projetos para aceleradores de *hardware*, tais como: melhorias em desempenho, minimização ao acesso às memórias e melhorias em utilização de recursos. Outra abordagem chamada Projeto Panda é apresentada em (PILATO, 2011), onde define uma estrutura HLS semiautomática com objetivo de implementar automaticamente os acessos de memória em Síntese de Alto Nível, através de uma especificação comportamental e um conjunto de restrições de projetos.

Xilinx dispõe de uma integração de ferramentas como: Vivado High Level Synthesis (XILINX, 2016) e Vivado Design Suite Tools (XILINX, 2013). Vivado High Level Synthesis

(HLS) é uma ferramenta de Síntese de Alto Nível para geração automatizada de aceleradores de *hardware*, permite a tradução de códigos em C, C++ e System C em código RTL (*Register Tranfer Level*). Em contraste com outras ferramentas de Síntese de Alto Nível, Vivado HLS tem o objetivo de mapear aplicações inteiras em *hardware*. Vivado HLS oferece algumas otimizações de alto nível, por exemplo, *pipeline*, *loop unrolling* e compartilhamento de recursos. Estas diretivas serão apresentadas no capítulo 3, onde expõem suas características de melhorias em termos de desempenho, área e exploração de paralelismo na execução das aplicações.

A Síntese de Alto Nível (HLS), torna-se cada vez mais popular para a criação de projetos de alta performance e eficiência energética reduzindo o tempo de criação. Permitindo que os projetistas trabalhem em um nível mais alto de abstração utilizando um programa em *software* para especificar uma funcionalidade em *hardware* (NANE, 2016).

Ao utilizar redundância na ferramenta HLS da Xilinx é necessário ter precaução quanto à simplificação do circuito pela ferramenta, o que pode tornar a redundância dependente dos parâmetros do HLS. A ferramenta de HLS da Xilinx gera uma máquina de estados para controle e execução da aplicação, sendo considerada uma das partes mais sensíveis do projeto se atingida por uma falha transiente.

Dentro deste contexto, esta dissertação tem como propósito inserir redundância (TMR, *Triple Modular Redundancy*) em projetos gerados por ferramenta de Síntese de Alto Nível e analisar a eficácia sob injeção de falhas por emulação (falhas transientes) no FPGA Xilinx Artix 7. A aplicação utilizada neste trabalho foi multiplicação de matrizes 6x6 e o método de análise de resultados consiste na comparação da porcentagem de bits críticos em relação aos bits de configuração e bits essenciais entre projetos sem e com redundância.

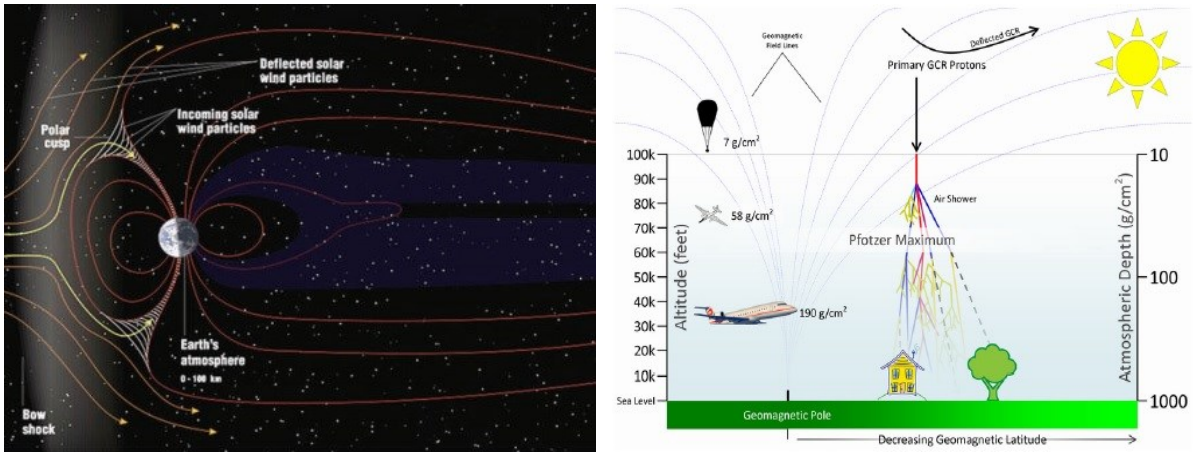
Para alcançar os objetivos propostos, o presente trabalho está organizado da seguinte maneira: o Capítulo Dois aborda o ambiente de radiação e seus efeitos em componentes eletrônicos. Definição de Transferência Linear de Energia (LET, *Linear Energy Transfer*) e Dose Total Ionizante (TID, *Total Ionizing Doze*). Classificações e efeitos causados por falhas transientes (SEEs) em dispositivos eletrônicos, principalmente FPGAs baseados em SRAM. Descrição de parâmetros para medição da radiação no que diz respeito a grandezas aplicáveis a SEEs. O Capítulo Três descreve a ferramenta de Síntese de Alto Nível (HLS) da Xilinx, diretivas de otimizações, síntese e sua integração com a ferramenta Xilinx Vivado Design Suite Tools. O Capítulo Quatro expõe as técnicas de redundância aplicáveis em FPGAs baseados em SRAM e aplicação neste trabalho. O Capítulo Cinco apresenta o estudo de caso e seus propósitos, projetos escolhidos na implementação com injeção de falhas simples e

acumulada. Descreve as duas arquiteturas utilizadas no trabalho. Expõe o número de passos para executar a aplicação. O Capítulo Seis expõe os métodos para injeção de falhas simples e acumulada. Descrição da metodologia utilizada na estrutura do injetor de falhas e na memória de configuração do FPGA, comunicação e controle do injetor de falhas através de um computador hospedeiro, arquiteturas de armazenamento, controle e recebimento dos dados, métricas de estimativa para falhas simples. Trabalhos correlatos com injeção de falhas na memória de configuração do FPGA utilizando projetos com circuitos gerados por ferramenta de HLS. O Capítulo Sete aborda sobre os resultados em termos de área e desempenho, bits críticos (Injeção de falhas simples), *confiabilidade* (injeção de falhas acumulada) e métricas de confiabilidade. O Capítulo Oito descreve a conclusão do trabalho e trabalhos futuros.

2 AMBIENTE DE RADIAÇÃO E SEUS EFEITOS EM COMPONENTES ELETRÔNICOS

O Sol gera constantemente um grande fluxo de partículas energizadas devido a sua atividade, o que faz o espaço ser transpassado por um fluxo de partículas altamente energizadas como prótons e átomos, com a variação de níveis de energia na faixa de 30 keV a 10^{15} MeV (MACHADO, 2014), ou seja, na ordem de milhões de elétrons-volt, com capacidade de ionizar os átomos em materiais semicondutores e dispositivos eletrônicos.

A radiação ionizante tem origem de eventos solares ou raios cósmicos, interatuam com a atmosfera e o campo magnético da Terra. A magnetosfera da Terra é definida por uma região onde o campo magnético estabelece o movimento do plasma, composto por prótons e elétrons.



(a) **(b)**

Figura 7 Interação da radiação solar e cósmica com a Magnetosfera Terrestre (a). Interação de raios cósmicos com átomos e moléculas presentes na atmosfera terrestre criando uma cascata de partículas mais leves (b) (MORISON, 2008; CHIELLE, 2016)

A figura 7 (a) ilustra interação da radiação solar e galáctica atingindo a magnetosfera e a ionosfera terrestre. A figura 7 (b) descreve a interação dos raios cósmicos com átomos e moléculas presentes na atmosfera terrestre originando uma cascata de partículas mais leves, por exemplo, prótons, elétrons, muons, partículas alpha, nêutrons, etc (CHIELLE, 2016). De acordo com alguns fatores do ambiente operacional, por exemplo, espaço, altitude de voo de aviões comerciais e militares, nível do mar, existem diferentes partículas e diferentes fluxos que podem perturbar o circuito integrado. No espaço temos ocorrência de íons, prótons e

elétrons. Em altitudes de voos principalmente nêutrons de alta velocidade. No nível do mar nêutrons.

A interação das partículas ionizantes com os inúmeros sistemas eletrônicos embarcados, por exemplo, satélites e aviões, é profundamente complexo, pode variar de acordo com o tipo de partícula, fluxo e nível de energia (MACHADO, 2014). A unidade de medida pelo qual a partícula incide no material é o eletron-volts (eV), definido como a energia cinética adquirida por um elétron que atravessa uma diferença de potencial de 1volt no vácuo (1eV = $1{,}602.10^{-19}$Joule, aproximadamente) (ARRUDA, 2006). A energia transferida para o dispositivo denominada Transferência Linear de Energia (LET, *Linear Energy Transfer*), é mensurada pela energia incremental por unidade de comprimento (MeV / (mg/cm^2)). O LET mínimo para causar um SEU é denominado de limite LET (*LET Threshold*), existem alguns níveis de robustez para o dispositivo funcionar corretamente, dependendo da quantidade de energia transferida para o silício. Neste trabalho a tecnologia utilizada foi de 28nm (Xilinx Artix 7), onde é possível observar erros com LET *Threshold* a partir de 2 $MeV.cm^2/mg$. Pode-se calcular a probabilidade de uma determinada partícula causar uma perturbação no dispositivo eletrônico, através do número de perturbações e a quantidade de partículas que atravessam o material. O cálculo para estimar a possibilidade de uma partícula causar uma inversão de bit é o número de perturbações dividido pelo número de partículas por cm^2, denominado como seção transversal ou de choque do material (σ, *Cross Section*), usualmente expressado em termos de área (geralmente em cm^2/dispositivo ou cm^2/bit). Portanto, a sensibilidade de um dispositivo a uma perturbação pode ser medida em função da seção transversal em termos de LET (KASTENSMIDT, 2006).

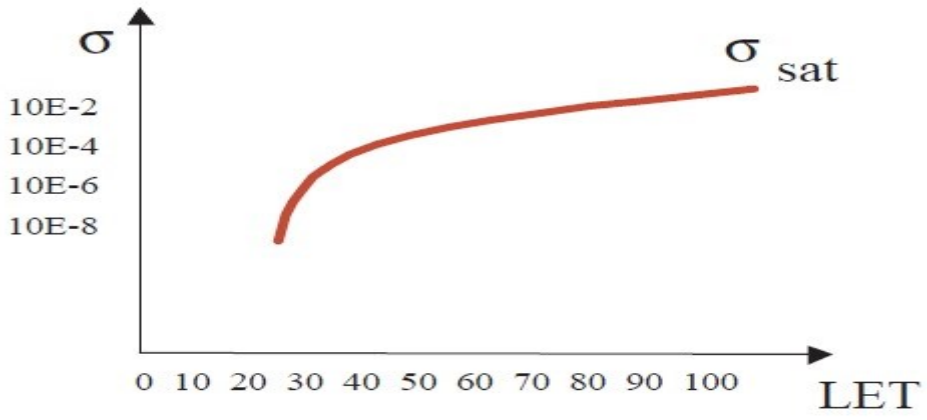


Figura 8 Exemplo de Seção de Choque (σ) pela curva de Transferência de Energia Linear (LET) (KASTENSMIDT, 2006)

O fluxo de partículas pode ser definido por meio do número de partículas que atravessam uma área de 1 cm^2 durante o tempo de 1 segundo [$1/s.cm^2$]. A figura 8 ilustra a curva de transferência de energia linear (LET) em função da seção de choque (σ).

A dose de radiação é definida como a quantidade de energia absorvida por um material no momento que atravessa um ambiente de radiação. A unidade de medida pode ser Gray (Gy) ou rad (1Gy = 100 rads), sendo que 1 rad = 10^{-2} J/kg. Por exemplo, os efeitos da radiação em $1cm^3$ de silício, 1 rad equivale a geração de cerca de 4.10^{13} pares de lacunas-elétrons (ARRUDA, 2006).

Os danos e anomalias sofridos por um componente eletrônico devido à dose total ionizante, do inglês *Total Ionizing Dose* (TID), dependem do tempo total de exposição à radiação. Os efeitos são acumulativos e tem relação à intensidade e o tempo de exposição do dispositivo a radiação (TAMBARA, 2013). A ação da radiação ionizante resulta na geração de pares de elétron-lacunas, estas cargas podem danificar o componente eletrônico ou até levá-lo a uma falha funcional com o passar do tempo e de acordo com a energia ionizante absorvida. O aprisionamento de cargas no óxido de silício (Sio_2) e na interface do transistor resultam em pares de elétron-lacunas, sendo o principal fenômeno capaz de alterar o comportamento principal de um dispositivo eletrônico. As lacunas podem ficar aprisionadas no óxido, na interface do óxido (mau funcionamento) ou se recombinar com elétrons. A taxa de recombinação do elétron com a lacuna depende da taxa da dose. Uma taxa alta da dose (HDR, *High Dose Rate*) induz a uma alta quantidade de pares elétron-lacunas, aumentando a taxa de recombinação (ARRUDA, 2006).

Os efeitos que causam perturbações nos circuitos integrados são chamados de *Single Event Effects* (SEEs), efeitos transitórios em tradução livre, causados pela radiação atmosférica e considerados um problema de projeto em sistemas críticos, como aeroespacial e terrestre que exigem alta confiabilidade como relatados no capítulo 1. Existem vários tipos de SEEs, definidos como o resultado de uma única partícula colocando energia suficiente para causar uma perturbação ou falha no dispositivo eletrônico. Ao medir a susceptibilidade de um sistema para SEEs, deve-se levar em consideração os componentes de fabricação, a probabilidade de exposição e a criticalidade da unidade funcional.

Componentes digitais e sistemas integrados tornaram-se mais suscetíveis a partículas de radiação como SEEs, devido à baixa tensão de alimentação e redução de transistores em escala nanométrica. Esses fenômenos são resultado da interação de raios cósmicos de alta velocidade com componentes à base de silício. Os efeitos do SEEs podem causar várias condições de falha, como corrupção de dados ou até mesmo uma falha do sistema, causando

um resultado errado. Podem ocorrer outros tipos de falhas ou efeitos indesejáveis, como danos no *hardware*, dados corrompidos na memória, paradas e interrupções do microprocessador, etc (DOMINICK, 2008).

Single Event Effects podem ser classificados em erros destrutivos, temporários não destrutivos ou permanentes. SEEs com potencial destrutivo são chamados, do inglês *hard errors*, e os mais comuns com potencial não destrutivo são chamados, do inglês *Soft errors* (O'BRYAN .M, 2015).

Os erros destrutivos podem danificar o dispositivo se não forem corrigidos a tempo. Na classificação destrutiva do SEEs temos: *Single Event Burnout* (SEB), causado pela corrente excessiva no dispositivo, *Single Event Gate Rupture* (SEGR), uma falha permanente no óxido de dispositivo e *Single Event Latchup* (SEL) em que há um curto circuito no dispositivo e também pode ser classificado como evento temporário não destrutivo.

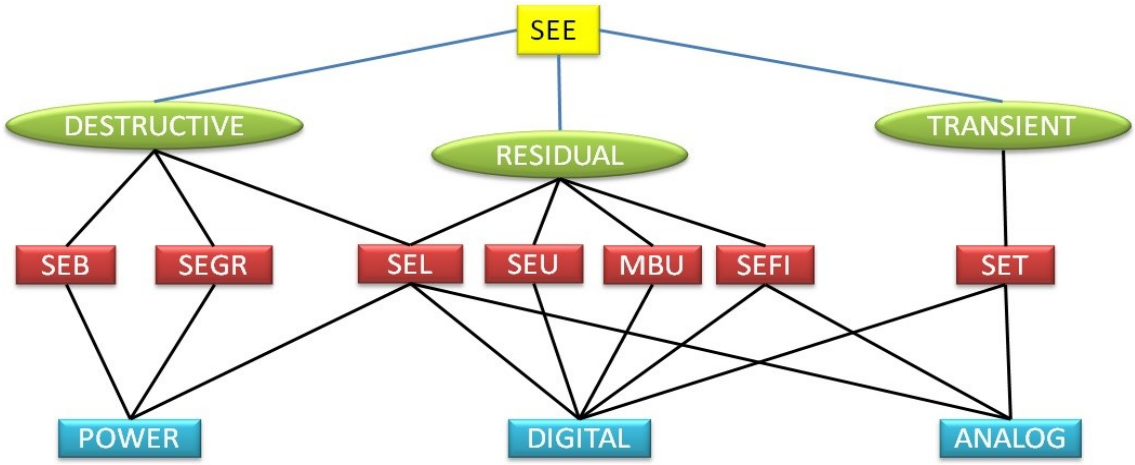


Figura 9 Classificação de SEEs (AUTOR)

SEE pode ser considerado um efeito causado pela ionização de regiões sensíveis de um dispositivo eletrônico causado por uma única partícula, resultando em uma falha de operação, temporária ou permanentemente. A figura 9 ilustra a classificação de SEEs. Na classificação temporária não destrutiva de SEE temos: *Single Event Upset* (SEU) é um tipo mais comum de SEEs não destrutivo e acontece quando uma partícula atinge a lógica sequencial do circuito, memória ou registrador, resultando em uma inversão ou alteração na lógica de bit único, do inglês *bit-flip*. *Multiple Bit Upsets* (MBU) acontecem quando uma partícula perturba mais que um bit em uma área próxima. *Single Event Transients* (SET) acontece quando uma partícula afeta uma porta, do inglês *gate,* da lógica combinacional do circuito criando uma falha e *Single Event Functional Interrupt* (SEFI), que coloca o

dispositivo num tempo de interrupção temporária ou para um estado indefinido. (STURESSON, 2009).

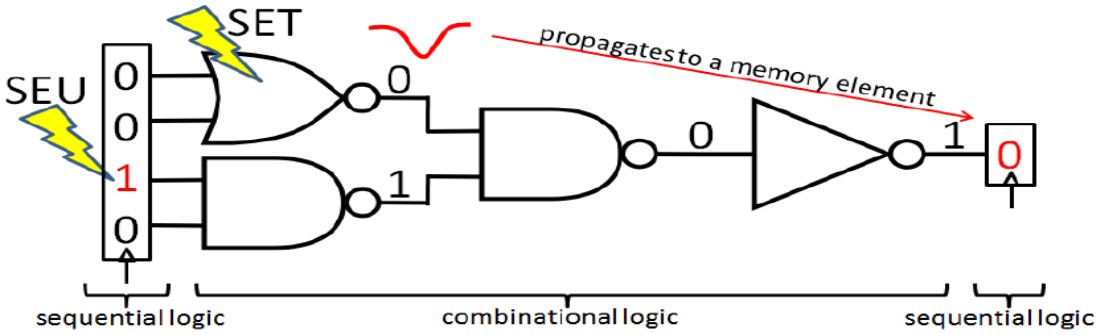


Figura 10 Exemplo de SEU e SET (CHIELLE, 2016)

Na figura 10 é ilustrado um exemplo de SEU e SET. Um elemento de memória é atingido por uma partícula causando um SEU, alterando o valor de '0' para '1', conhecido como *bit-flip*. Neste exemplo, o SEU é mascarado pela porta lógica NAND por causa da outra entrada que é '0'. Se fosse '1', a saída da NAND seria afetada pela falha e haveria propagação do erro. Outra partícula atinge a porta lógica NOR causando um SET mudando temporariamente o valor esperado na saída de '1' para '0', neste exemplo a falha não é mascarada pelas outras portas e se propaga para o elemento de memória, então se o pulso atingir o elemento de memória durante um evento de relógio, do inglês *clock*, um valor errado é armazenado.

A resposta de um componente ao SEE é caracterizada como área sensível, do inglês *Cross Section*, obtida por testes com o componente dimensionando os resultados para o ambiente de aplicação (LAURENCE, 2014).

As Memórias de Acesso Aleatório (RAM) são mais sensíveis a SEUs, pois apresentam volatilidade no armazenamento da informação que é representada por bits lógicos. Por exemplo, se uma carga depositada por uma partícula colidir com o circuito integrado pode ser suficiente para inverter o estado de uma célula de memória. Alguns dispositivos que utilizam incorporado este tipo de memória são suscetíveis como: microprocessadores, ASIC (*Application Specific Integrated Circtuits*), FPGAs (*Field Program Gate Array*).

Um grande conjunto de dispositivos eletrônicos utilizados em aeronaves, aplicações espaciais e terrestres podem ser perturbados por partículas ionizadas. Embora os efeitos mais prevalecentes sejam em altitudes elevadas, como rotas de aviões comerciais e militares. À medida que a susceptibilidade aos efeitos de radiação continua aumentando para essas

tecnologias baseadas em silício, a necessidade de mitigar erros ou falhas aumenta também. Soluções para mitigação de SEEs em nível de *chip*, subsistemas e projetos de sistemas são necessários para lidar com os impactos dos efeitos da radiação, observando os potenciais modos de falhas e os seus efeitos no dispositivo (LAURENCE, 2014).

Soluções de mitigação de falhas incluem o projeto de circuito, arquitetura de sistemas tolerantes a falhas e soluções de tecnologia em nível de componente (DOMINICK, 2008). Para criar um sistema que atenda a todos os requisitos de segurança e confiabilidade durante uma operação em seu ambiente de radiação, é necessário ter uma boa combinação de técnicas de mitigação de falhas, resultando em um sistema com níveis de risco conhecidos e desenvolvido a um custo ideal.

2.2 SINGLE EVENT EFFECTS (SEEs) em FPGAs

FPGAs são constituídos por um arranjo de blocos lógicos programáveis e uma matriz de blocos endereçáveis reconfiguráveis, que os permite serem conectados com inúmeras portas lógicas que podem ser interligadas com diferentes configurações. A memória de configuração do FPGA é organizada quanto ao número de quadros de configuração, do inglês *configuration frames*, que são a menor unidade endereçável e ocupam as células SRAM (*Static Random Access Memory*) do dispositivo (AISHWARYA, MAHENDRAN, 2016).

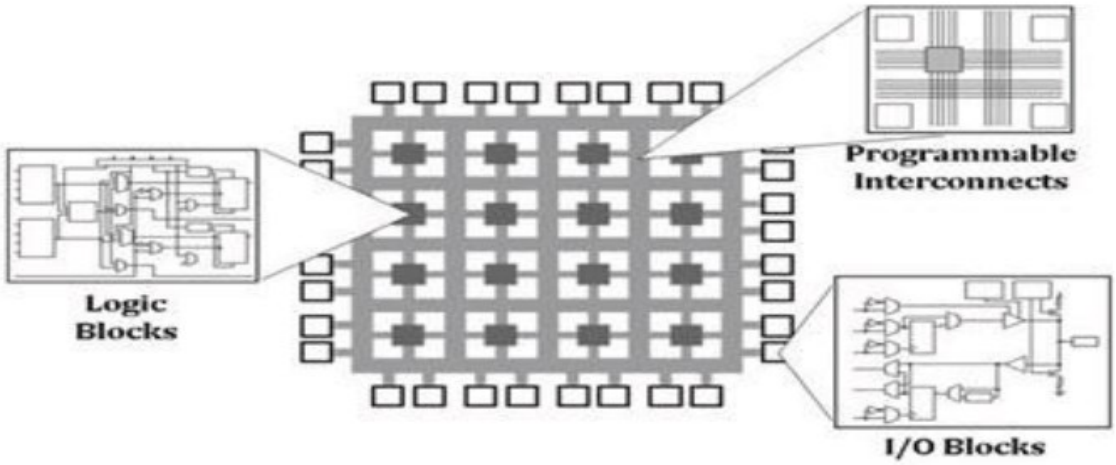


Figura 11 Arquitetura do FPGA (AISHWARYA; MAHENDRAN, 2016).

Os FPGAs baseados em SRAM são principalmente suscetíveis a *Single Event Upset* (SEU) em seus bits de memória de configuração e células de memória incorporadas. Quando

a energia é coletada por uma junção sensível de um transistor de célula SRAM que compõe os bits da memória de configuração do fluxo de bits, do inglês *bitstream* do FPGA, pode ocorrer uma inversão de bit, do inglês *bit-flip*. Este *bit-flip* pode alterar a configuração de uma conexão de roteamento, configuração de uma LUT, do inglês *Look Up Table* (tabela que determina qual a saída para qualquer dado de entrada, representada por uma função), alterar o estado dos *flip-flops* nos CLBs, do inglês *Configurable Logic Block* (permite implementar funcionalidades lógicas no *chip*) ou alterar a configuração de uma BRAMs (*Block Random Access Memory*). O *bit-flip* tem efeito persistente, que só pode ser corrigido quando um novo *bitstream* é carregado para o FPGA.

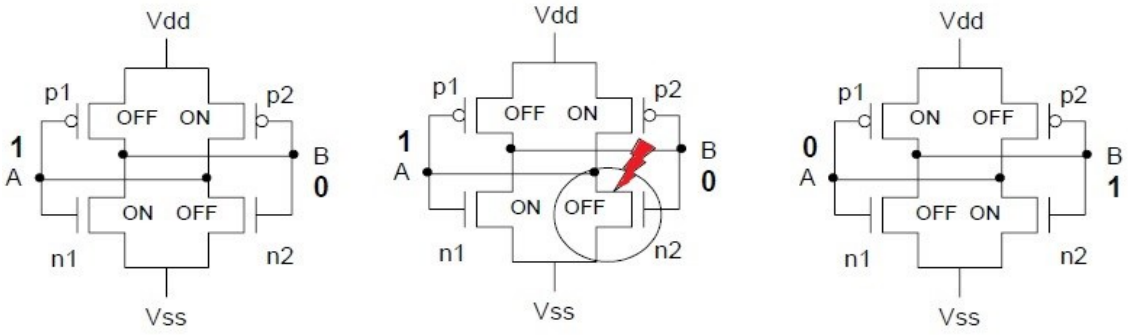


Figura 12 Exemplo de ocorrência de SEU em célula de memória SRAM. (KASTENSMIDT, 2006)

A célula de memória SRAM de arquitetura tradicional é composta por 6 transistores geralmente. A figura 12 ilustra um exemplo de ocorrência de um SEU em uma célula de memória SRAM representado em três etapas. Na etapa 1 a célula SRAM com nodos ‘n1’ e ‘p2’ ativose e nodos ‘p1’ e ‘n2’ desligados. Na etapa 2 o nodo ‘n2’ sofre a influência do SEU. Na etapa 3 o valor armazenado em ‘n2’ foi invertido pelo SEU, tornando o nodo ‘n2’ ativo e o ‘p2’ desligado. Concluindo, quando um dos nodos mais sensíveis de uma célula de memória SRAM é atingido por uma partícula energizada, por exemplo, uma junção PN de um transistor desligado ou dreno de um transistor desligado, é gerado um pulso de corrente transitória que pode inverter o valor armazenado e ativar o canal do transistor oposto, causando a ocorrência de um *bit-flip* (KASTENSMIDT, 2006).

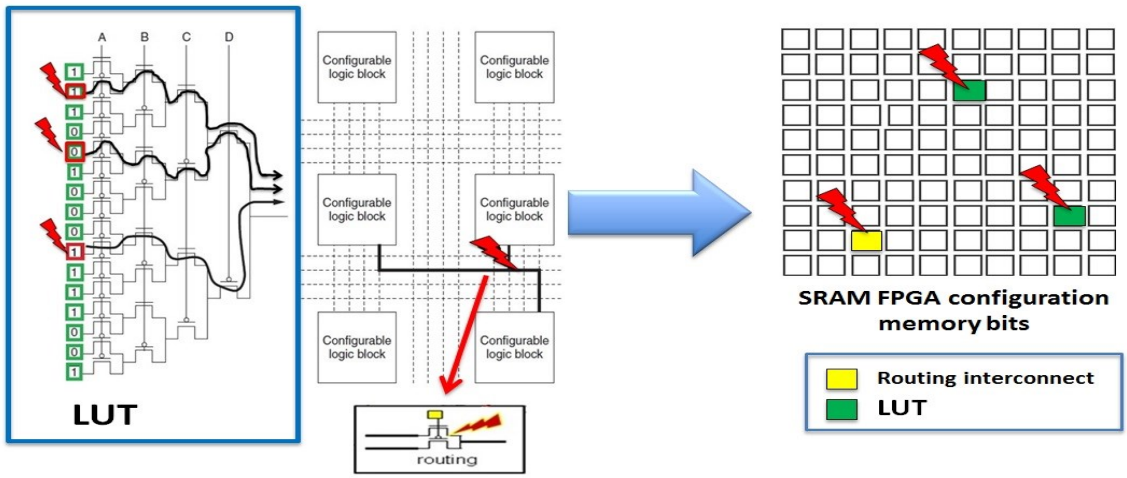


Figura 13 Exemplo de SEU em FPGA baseado em SRAM (AUTOR)

A figura 13 ilustra a ocorrência de um SEU na memória de configuração do FPGA, a imagem deve ser analisada como se cada *bit-flip* acontecesse um após o outro em separado. Pois se todos ocorrecem juntos estaria simulando um MBU (*Multiple Bit Upset*) (QUIN, 2005). Neste caso o SEU esta atingindo interconexões e LUTS. *Bit-flips* também podem ocorrer nos *flip-flops* dos CLBs com efeito transitório. Assim, a maioria dos erros observados em ambientes agressivos vêm de *bit-flips* (SEU) nos bits da memória de configuração. A injeção de falhas no *bitstream* pode ser realizada em laboratório e é útil para analisar a vulnerabilidade lógica de um projeto para *bit-flips* na memória de configuração.

Dependendo da arquitetura do projeto implementado em FPGAs baseados em SRAM, mais ou menos bits de configuração são usados e bits suscetíveis podem ser responsáveis por provocar um erro na saída do projeto.

No entanto, não é somente o número de bits usados que determinam a sensibilidade, o efeito de mascaramento do algoritmo de aplicação desempenha um papel importante, outros fatores também podem contribuir diretamente para a análise de taxa média de erros em FPGAs como: área, desempenho, tempo de execução em ciclos de relógio e tipos de recursos utilizados que podem contribuir diretamente na análise de taxa média de erros em FPGAs (TONFAT, 2016).

Existem alguns parâmetros para medição da radiação no que diz respeito a grandesas aplicáveis a SEEs e foram calculados com base em resultados experimentais. A sensibilidade de um circuito a eventos singulares tem a definição em termos de área sensível total do dispositivo através da grandeza com denominação de seção de choque (σ), do inglês *Cross Section*. Existem dois tipos de seção de choque: Seção de choque estático (σ*static*), e dinâmico (σ*dynamic*).

A seção de choque estática (σ*static*) é um parâmetro intrínseco do dispositivo normalmente expresso em termos de área (usalmente cm^2 / dispositivo ou cm^2 / *bit*) e está relacionado com a mínima área suscetível do dispositivo a uma espécie de partículas (por exemplo, neutrôns, prótons, íons pesados, etc.). A expressão para obter a seção de choque estática (σ*static*) por *bit* de um dispositivo é:

$$\sigma_{static} = \frac{N_{SEU}}{\Phi_{particle}} \qquad \sigma_{static-bit} = \frac{N_{SEU}}{\Phi_{particle} \times N_{bit}} \quad (1, 2)$$

$$\sigma_{static} Circuit = 6{,}99E-15 * ConfigurationBitsCircuit \quad (3)$$

O parâmetro N_{SEU} é o número de SEU nos bits de memória de configuração, $\Phi_{particle}$ é a fluência de partículas, medida por partículas por cm^2 e é calculada multiplicando o fluxo de partículas pelo tempo que o dispositivo foi exposto a esse fluxo. A seção de choque por *bit* é calculada dividindo a seção de choque estática pelo número total de bits no dispositivo (Nbit).

A seção de choque estática do dispositivo FPGA Xilinx Artix 7 para nêutrons (*Static Cross Section Device*) é definida com valor 6,99E-15, disponibilizada pelo fabricante (XILINX, 2015b). A seção de choque estática para a área do circuito (cm^2) é calculada pela multiplicação da seção de choque estática do dispositivo pelo número de bits de configuração. Os bits de configuração no circuito dependem da área total do projeto implementado. A seção de choque dinâmica (σ*dynamic*) é definida como a probabilidade de uma partícula gerar um erro na saída do circuito. A expressão para obter a seção de choque dinâmica é:

$$\sigma_{dynamic} = \frac{N_{ERROR}}{\Phi_{particle}} \quad (4)$$

Onde o parâmetro N_{ERROR} é o número de erros observados no comportamento do circuito ou aplicação e Φ*particle* é a fluência de partículas.

Em (VELAZCO, 2010), o autor apresenta uma abordagem combinando a seção de choque estática (*Static Cross Section*) de FPGAs baseados em SRAM com resultados de campanhas de injeção de falhas por emulação para prever a taxa de erros de qualquer aplicação implementada. Os resultados obtidos em testes de Íons carregados são comparados com as previsões para validar a metodologia.

Um erro pode ser definido como alguma mudança de comportamento esperado de um projeto. Os erros são classificados da seguinte maneira: erro de corrupção silenciosa de dados (SDC, *Silent Data Corruption*), ou seja, dados errados na saída do projeto; Classificação de erro de tempo limite, por exemplo, ausência de dados na saída do projeto após um determinado tempo (*Timeout*).

Os bits críticos são definidos pela Xilinx (XILINX, 2012), como a quantidade de bits de configuração que uma vez invertidos, podem causar um erro no comportamento esperado do projeto (SDC ou *Timeout*). Os bits críticos são obtidos através de injeção de falhas no projeto sobre teste (DUT). Neste trabalho foram implementadas duas formas de injeção de falhas: Injeção de falhas simples e injeção de falhas acumuladas. Na injeção de falhas simples os bits críticos de cada projeto são obtidos por uma campanha de injeção de falhas exaustiva e sequencial, na qual todos os bits de configuração da área de injeção são invertidos um de cada vez. Na injeção de falhas acumuladas é analisada a confiabilidade de cada projeto, ou seja, quantos *bit-flips* acumulados são necessários para gerar um erro na saída do projeto, sendo sorteados bits aleatórios da área de injeção de falhas para serem invertidos. Os detalhes para cada modo de injeção de falhas são descritos no capítulo 6 e os resultados no capítulo 7.

3 HIGH LEVEL SYNTHESIS TOOLS

O Xilinx Vivado High-Level Synthesis Tools (HLS) transforma uma especificação C em uma implementação de nível de transferência de registro (RTL) que pode ser sintetizada em um FPGA Xilinx. A especificação pode ser escrita em linguagens de programação C, C++ e System C. O FPGA tem uma arquitetura paralela e oferece benefícios em termos de desempenho, custo e consumo de energia em comparação com os processadores tradicionais. O uso do HLS para projetistas de *hardware* oferece vários benefícios, como um maior nível de abstração na criação de *hardware* de alto desempenho.

A validação e correção do projeto funcional são realizadas mais rapidamente do que com a linguagem de *hardware* tradicional (VHDL). O uso de diretivas de otimização facilita criar implementações específicas de *hardware* de alto desempenho com possibilidade de utilizar no mesmo código C diferentes diretivas para uma implementação otimizada. Reutilização do código C em diferentes projetos e dispositivos (XILINX, 2016). O processo de aplicação de diretivas de otimização é realizado dependendo da descrição do código C, C++ e suas características. É necessário observar o que se destina ao projeto final antes de aplicá-las. Algumas métricas de análise importantes são: os ciclos de relógio (*Clock Cycle)*, latência e a área total ocupada pelo circuito. A Síntese de Alto Nível (HLS) inclui as seguintes fases:

• Programação ou agendamento, do inglês *Scheduling* - Determina quais operações ocorrem em cada ciclo de relógio, com base no comprimento do ciclo de relógio ou da frequência de ciclo de relógio, o tempo necessário para a conclusão da operação definida pelo dispositivo de destino e as diretivas de otimizações são definidas pelo projetista. A figura 15 ilustra a programação e fases de ligação para o código da função foo_1 descrito na figura 14.

```
int foo_1(char x, char a, char b, char c) {
        char y;
        y = x*a+b+c;
        return y
}
```

Figura 14 Pseudocódigo para função Foo_1 (XILINX, 2016)

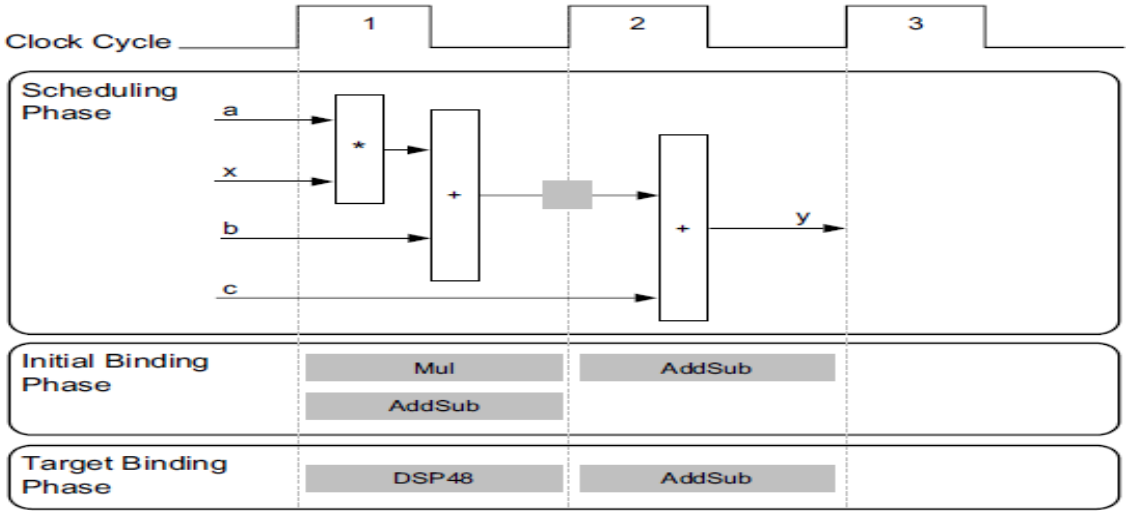


Figura 15 Fase de agendamento e fase de ligação (XILINX, 2016)

Na fase de programação ilustrada na figura 15 (*Scheduling Phase*), a Síntese de Alto Nível determina o que deve ocorrer em cada ciclo de relógio. O quadrado entre o primeiro e segundo ciclo de relógio indica quando um registro interno armazena uma variável. Neste exemplo, Síntese de Alto Nível requer apenas que a saída da adição seja registrada através de um ciclo de relógio.

• Ligação, do inglês *Binding* - Determina quais recursos de *hardware* implementam cada operação agendada para criar a solução ideal. Síntese de Alto Nível usa informações sobre o dispositivo de destino.

• Extração da lógica de controle, do inglês *Control Logic Extraction* - Através da lógica de controle é criada uma máquina de estados finitos (FSM) que são as sequências de operações no projeto RTL.

```
void foo_2(int in[3], char a, char b, char c, int out[3]) {
        int x, y;
        for (int i = 0; i < 3; i++) {
        x = in[i];
        y = a*x + b + c;
        out [i] = y;
        }
}
```

Figura 16 Pseudocódigo para função foo_2 (XILINX, 2016)

A figura 17 ilustra a extração da lógica de controle e a implementação das portas de entrada e saída, do inglês *input/output* (I/O) para o código de função 'foo_2' exibido na figura 16.

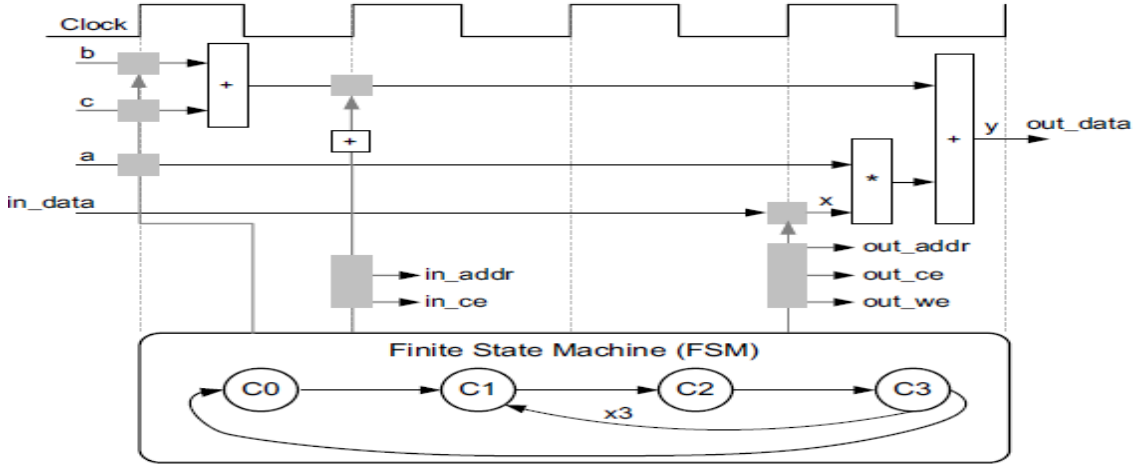


Figura 17 Função Foo_2 - Extração de lógica de controle e implementação I/O (XILINX, 2016)

O exemplo de código da função foo_2 executa as mesmas operações que a função 'foo_1', mas com a diferença na execução de operações dentro de um laço de repetição '*for*' e dois argumentos de função são vetores. A Síntese de Alto Nível (HLS) extrai automaticamente a lógica de controle usada no código C, cria uma máquina de estados (FSM) e implementa os argumentos de função como portas no final do projeto RTL.

As variáveis escalares do tipo '*char*' são mapeadas para as portas de barramento de dados padrão de 8 bits. Os argumentos das matrizes como '*in*' e '*out*' contém uma coleção de dados. Em Síntese de Alto Nível, as matrizes são sintetizadas como blocos RAM (*Random Access Memory*), mas são possíveis outras opções, como FIFO (*First in First out*), RAM distribuída e registradores individuais. Ao usar vetores como argumentos na função topo, Síntese de Alto Nível assume que o bloco RAM está fora da função e cria portas de acesso fora do projeto para o bloco RAM, como portas de endereço e portas de dados e sinais de habilitação de gravação.

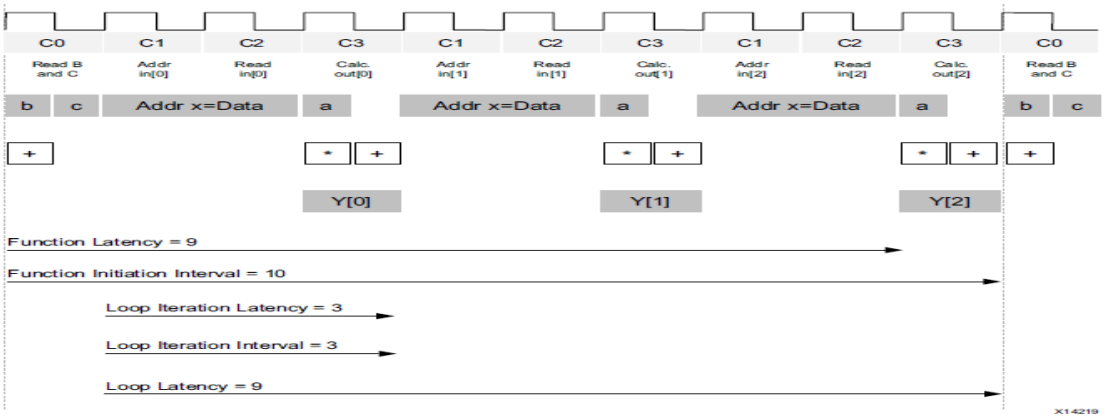


Figura 18 Função Foo_2 - Latência e intervalo inicial (XILINX, 2016)

A figura 18 ilustra a execução completa ciclo a ciclo do código da função foo_2, incluindo os estados para cada ciclo de relógio, operações de computação, operações de leitura e operações de gravação.

• Latência (*Latency*): A função usa 9 ciclos de relógio para produzir todos os valores de resultado.

• O intervalo de inicialização da função, do inglês *Interval Initiation Function:* tem valor 10, o que significa que leva 10 ciclos de relógio antes que a função possa iniciar uma nova leitura para dados de entrada e começar a processar o próximo conjunto de dados.

• Latência de iteração de laço de repetição, do inglês *Loop Iteration Latency*: A latência de cada iteração do laço de repetição é de 3 ciclos de relógio.

• Intervalo de iteração de laço de repetição, do inglês *Loop Iteration Interval:* Tem valor 3, que significa que leva 3 ciclos de relógio para iniciar uma nova execução do laço de repetição.

• Latência do laço de repetição, do inglês *Loop Latency*: Tem valor de 9 ciclos de relógio, que significa que leva 9 ciclos de relógio para completar todo o laço de repetição.

3.1 DIRETIVAS DE OTIMIZAÇÃO

As diretivas de otimização estão disponíveis no Vivado HLS para ajudar o projetista de *hardware* a construir uma micro-arquitetura que satisfaça as metas de desempenho e área desejadas. As diretivas podem ser configuradas pela IDE (*Integrated Development Environment)* gráfica do HLS ou por linha de comando, segue alguns exemplos de diretivas utilizadas pelo HLS:

• *Allocation*: especifica um limite para o número de operações, núcleos e funções utilizadas. Isso pode forçar o compartilhamento ou recursos de *hardware* e pode aumentar a latência. Por exemplo na função foo_3 ilustrada na figura 19.

```
int foo_3 (int a, int b) {
    int c, d;
    c = a*b;
    d = a*c;
    return d;
}
```

Figura 19 Pseudocódigo para função foo_3 (XILINX, 2016)

Usando a diretiva '*Set_directive_allocation -limit 1 -type operation foo mul*', limita a implementação a uma operação mul (*multiplication*) (por padrão, não há limite). Isso força Síntese de Alto Nível para usar um único multiplicador para a função "foo_3".

Tabela 1. Exemplos de operadores disponíveis para diretiva Allocation no HLS (XILINX, 2016)

Operator	**Description**
add	Integer Addition
ashr	Arithmetic Shift-Right
dadd	Double-precision floating point addition
dcmp	Double -precision floating point comparison
ddiv	Double -precision floating point division
dmul	Double -precision floating point multiplication
drecip	Double -precision floating point reciprocal
drem	Double -precision floating point remainder
drsqrt	Double -precision floating point reciprocal square root
dsub	Double -precision floating point subtraction
dsqrt	Double -precision floating point square root
fadd	Single-precision floating point addition
fcmp	Single-precision floating point comparison
fdiv	Single-precision floating point division
fmul	Single-precision floating point multiplication
frecip	Single-precision floating point reciprocal
frem	Single-precision floating point remainder
frsqrt	Single-precision floating point reciprocal square root
fsub	Single-precision floating point subtraction
fsqrt	Single-precision floating point square root
icmp	Integer Compare
lshr	Logical Shift-Right
mul	Multiplication
sdiv	Signed Divider
shl	Shift-Left
srem	Signed Remainder
sub	Subtraction
udiv	Unsigned Division
urem	Unsigned Remainder

• *Inline*: remove a função como uma entidade separada na hierarquia do projeto. É importante investigar se esta opção está desativada, quando o trabalho esta relacionado a injeção de falhas, caso contrário alguns módulos podem ficar separados no projeto final, o que pode não ser considerado um modelo ideal.

• *Array Map*: combina múltiplas matrizes menores em um único conjunto de dados (*array*) grande para ajudar a reduzir os recursos de RAM de bloco, exibido na figura 20.

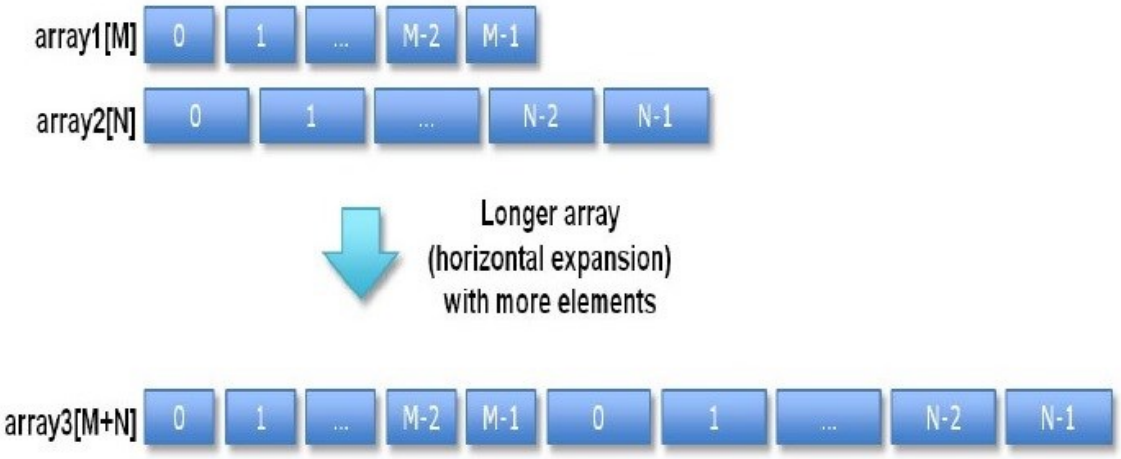


Figura 20 Exemplo da diretiva de otimização Array Map combinado em Array horizontal (XILINX, 2016)

• *Array Partition*: particiona grandes conjuntos de dados, do inglês *arrays* em vários conjuntos de dados menores ou em registradores individuais, para melhorar o acesso aos dados e remover gargalos da memória RAM. As memórias têm apenas uma quantidade limitada de portas de leitura e portas de escrita que podem limitar a taxa de transferência de um algoritmo intensivo de carregamento ou armazenamento, isso pode ser melhorado dividindo a matriz original em vários *arrays* menores, efetivamente aumentando o número de portas. Existem três modos de operação do *Array Partition*:

- Bloco: a matriz original é dividida em blocos de tamanho igual de elementos consecutivos da matriz original.
- Cíclico: a matriz original é dividida em blocos de tamanho igual, intercalando os elementos da matriz original.
- Completo: a operação padrão é dividir a matriz em seus elementos individuais. Este modo corresponde à resolução de uma memória em registradores, o que pode aumentar o número de portas e o desempenho.

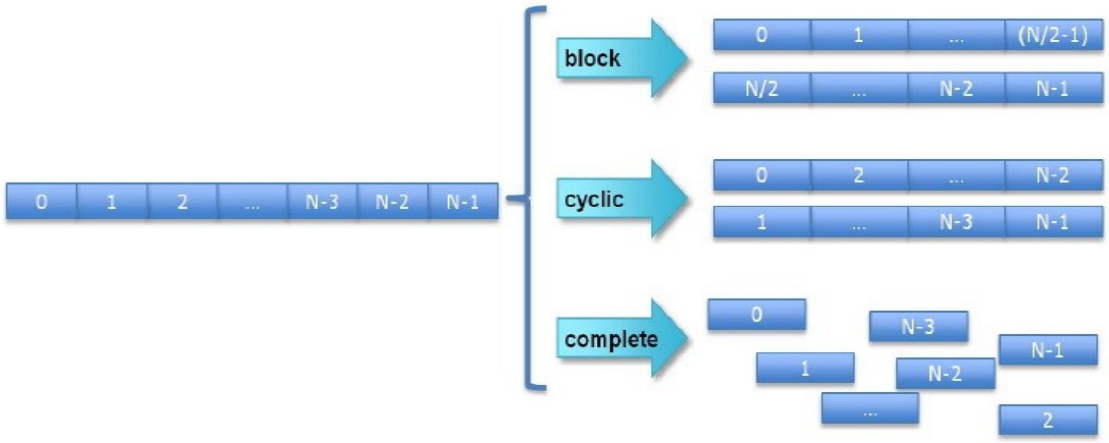


Figura 21 Exemplo da diretiva de otimização Array partition (XILINX, 2016)

• *Array Reshape*: combina a diretiva de otimização *Array Partition* com a diretiva *Array Map* no modo vertical, é utilizada para reduzir o número de blocos de RAM. Tem como vantagem a utilização dos benefícios do particionamento e acesso paralelo aos dados. A diretiva *Array Reshape* permite que mais dados sejam acessados em um único ciclo de relógio, por exemplo, laços de repetição podem ser totalmente ou parcialmente desenrolados para criar um *hardware* suficiente para consumir os dados adicionais em um único ciclo de relógio.

```
void foo_4 (...) {
  int array1[N];
  int array2[N];
  int array3[N];
#pragma HLS ARRAY_RESHAPE variable=array1 block factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array2 cycle factor=2 dim=1
  #pragma HLS ARRAY_RESHAPE variable=array3 complete dim=1
}
```

Figura 22 Pseudocódigo para função foo_4, utilizando diretiva de otimização Array_reshape (XILINX, 2016)

A função foo_4 demonstra o uso da diretiva *Array Reshape* em três arranjo de dados: *block, cycle* e *complete* que são as opções disponíveis para esta otimização ilustradas na figura 23. O fator, do inglês *factor*, é 2 por padrão do HLS, mas é possível utilizar outros valores no modo *block* e *cycle* apenas. O modo completo é o mais vantajoso quando o objetivo do projetista de *hardware* é ter um bom desempenho.

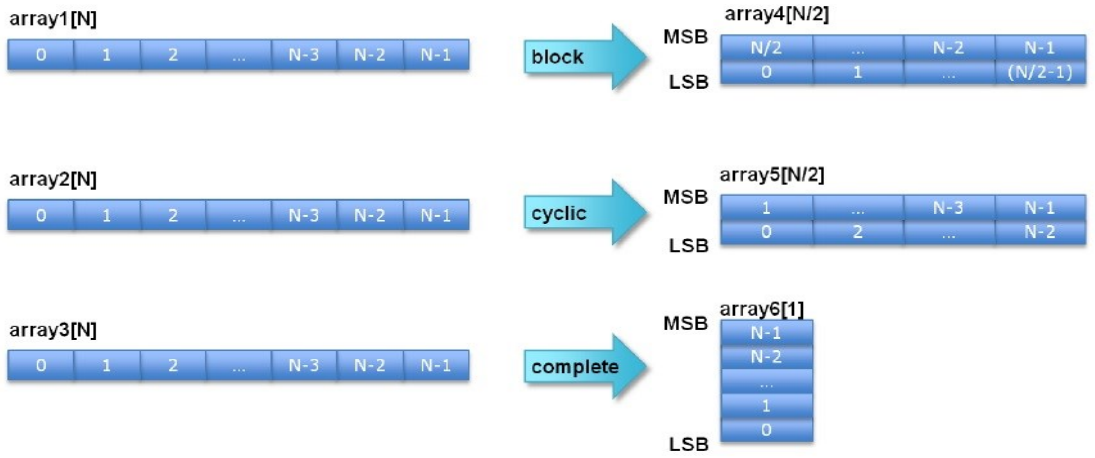


Figura 23 Exemplo da diretiva de otimização Array Reshaping (XILINX, 2016)

• *Pipeline*: pode ser utilizado para otimizar o fluxo de dados, funções e laços de repetição. O *Pipeline* transforma uma descrição funcional sequencial em uma arquitetura de processo paralela. Pode reduzir o intervalo de iniciação permitindo a execução concorrente ou paralela de operações dentro de um laço de repetição ou função.

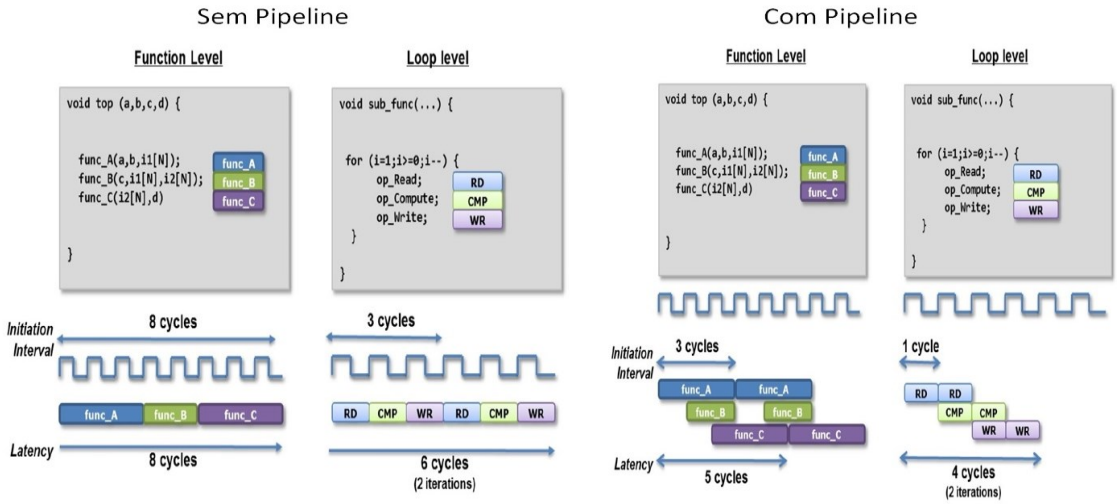


Figura 24 Exemplo da diretiva de otimização Pipeline em nível de função e laço de repetição (XILINX, 2016)

A figura 24 ilustra dois exemplos de execuções sem e com *Pipeline*. No exemplo sem *Pipeline* temos a função chamada '*top( )*' que leva 8 ciclos de relógio para ser executada e a função '*sub_func( )*' leva 6 ciclos. Aplicando-se a otimização *Pipeline* o valor do tempo de execução da função '*top( )*' cai para 5 ciclos de relógio e da função '*sub_func( )*' para 4 ciclos de relógio.

• *Loop Unrolling* (*Unroll*)*:* é uma otimização que desenrola os laços de repetição (*loops*), parcialmente ou completamente para paralelizar a execução. Por padrão o HLS mantém os laços de repetição enrolados, ou seja, tratados como uma única entidade onde todas as operações são implementadas utilizando os mesmos recursos de *hardware* para a iteração do laço. Ao desenrolar laços de repetição com a otimização *Loop Unrolling* devem ser levadas algumas considerações como: verificar se os conjuntos de dados (*arrays*) estão mapeados ou não para BRAMs; Se o laço de repetição é mapeado para elementos sequenciais como, por exemplo, ilustrado na figura 25 podemos ter três modos de configuração: laço enrolado (*Rolled Loop*), laço parcialmente desenrolado (*Partially Unrolled Loop*), laço desenrolado (*Unrolled Loop*). Se não for mapeados para elementos sequenciais o número de ciclos seria determinado pelo atraso do multiplicador neste exemplo.

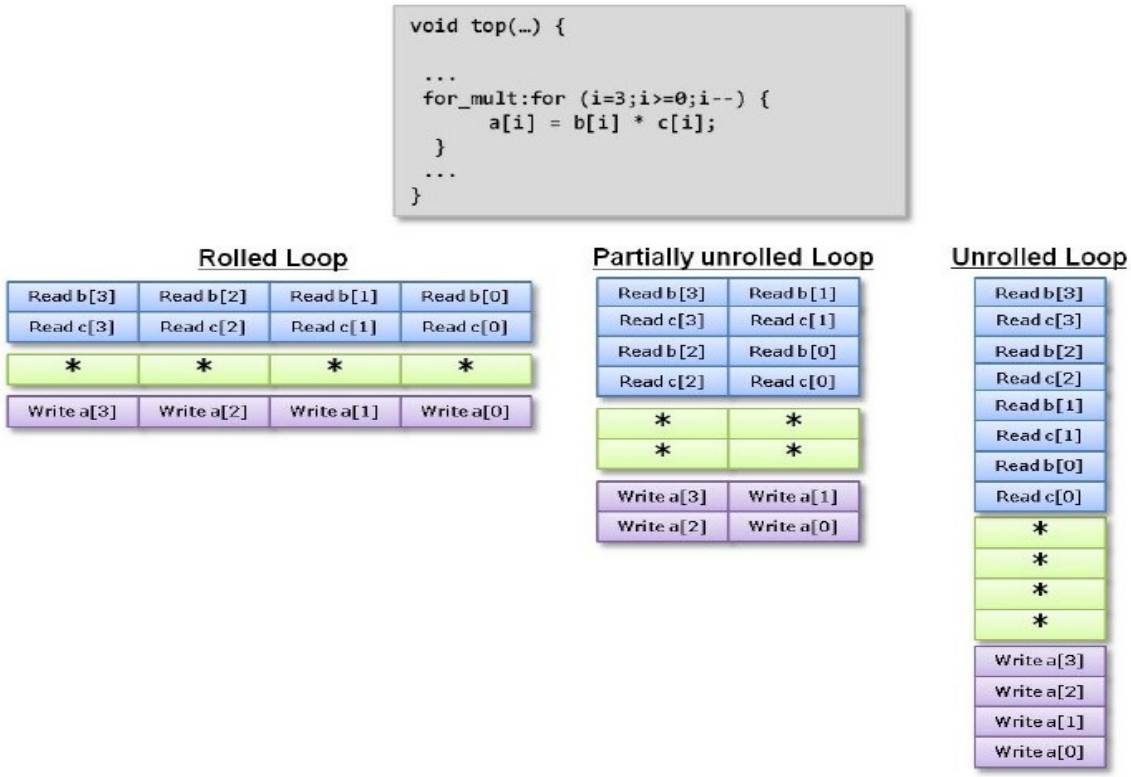


Figura 25 Exemplo de diretiva de otimização *Loop Unrolling* (*Unroll*) para função top (Xilinx, 2013b)

3.2 SÍNTESE

A ferramenta de Síntese de Alto Nível (HLS) da Xilinx chamada Vivado High Level Synthesis (HLS) sintetiza o código da seguinte maneira:

• Argumentos de função de nível superior sintetizam em portas RTL I/O (*Input/Output*, Entrada/Saída).
• As funções C sintetizam em blocos na hierarquia RTL (*Register Transfer Level*).
• Laços de repetição nas funções C são mantidos enrolados por padrão.
• Quando os laços de repetição são enrolados a síntese cria a lógica para uma iteração do laço e o projeto RTL executa essa lógica para cada iteração em sequência. Ao usar diretivas de otimização, torna-se possível desenrolar os laços, o que permite que todas as iterações ocorram em paralelo otimizando o desempenho e possivelmente aumentando a área.
• Arranjo de dados, do inglês *arrays*, no código C sintetizam em bloco RAM no projeto final FPGA. Se o arranjo de dados estiver na interface da função de nível superior, Síntese de Alto Nível implementa-os como portas para acessar um bloco RAM fora do projeto.

Síntese de Alto Nível cria a implementação ideal com base no comportamento padrão, restrições e diretivas de otimização especificadas pelo usuário. O projetista pode usar diretivas de otimização para modificar e controlar o comportamento padrão da lógica interna e das portas de entrada e saida I/O (*input/Output*), gerar uma implementação de *hardware* e variações sobre o mesmo código C. Os seguintes formatos de saída gerados pela Síntese de Alto Nível (HLS) são:
• Arquivos de implementação RTL em formatos de linguagem de descrição de *hardware* (HDL). Esta é a saída primária do Vivado HLS. Usando a Síntese de Alto Nível é possível sintetizar o RTL numa implementação a nível de porta, do inglês *gate-level*, e um arquivo de fluxo de bits (*bitstream*) FPGA. O RTL está disponível nos seguintes formatos padrões da indústria:

° VHDL (IEEE 1076-2000)
° Verilog (IEEE 1364-2001)

A ferramenta Xilinx Vivado High Level Synthesis sintetiza uma função C em um bloco de propriedade intelectual, do inglês IP (*Intellectual Property*) que pode se integrar num sistema de *hardware*. Ela é fortemente integrada com o resto das ferramentas de projeto Xilinx e oferece suporte e recursos de linguagem abrangentes para criar a implementação ideal para o algoritmo C.

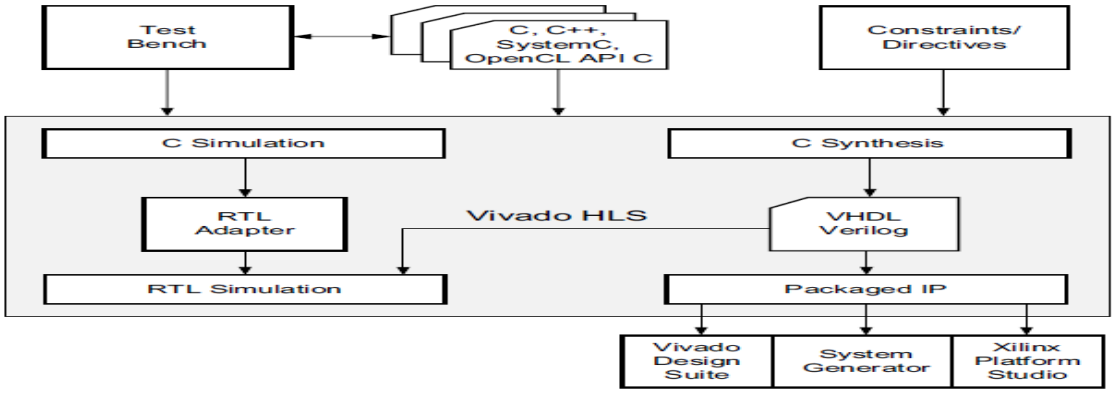


Figura 26 Xilinx Vivado HLS Design Flow (XILINX, 2016)

Usando a síntese lógica, é possível sintetizar o IP gerado em um fluxo de bits FPGA, do inglês *bitstream* (XILINX, 2016). A figura 26 ilustra o fluxo de projeto do Xilinx Vivado HLS.

3.3 VIVADO DESIGN SUITE TOOLS

O Vivado Design Suite Tools oferece várias maneiras de realizar as tarefas envolvidas no projeto e verificação Xilinx FPGA. Além do tradicional fluxo de projeto RTL para fluxo de bits (*bitstream*), o Vivado Design Suite Tools fornece novos fluxos de integração no nível do sistema que se concentram no projeto de propriedade intelectual (IP). Vários IP's podem ser instanciados, configurados e interativamente conectados em projetos de bloco de subsistema IP dentro do ambiente do integrador IP Vivado. A análise e verificação de projeto é ativada em cada estágio do fluxo. Os recursos de análise de projeto incluem simulação lógica, análise de consumo de energia (*Power*), definição de restrições (*Constraints*), análise de tempo (*Timing*), verificação de regras de projeto (DRC, *Design Rule Checking*) e visualização da lógica de projeto. O Vivado IDE (*Integrated Development Environment*) fornece uma interface para montar, implementar e validar o projeto e o IP. Ao usar a ferramenta de projeto Xilinx Vivado Design Suite Tools, é possível obter as seguintes métricas: área em termos de utilização de recursos após o projeto ser sintetizado e roteado para o dispositivo alvo, desempenho do projeto em termos de ciclos de relógio, tempo de execução e também os bits essenciais do projeto específico. Os bits essenciais são calculados usando um algoritmo proprietário da ferramenta Xilinx após o fluxo de bits (*bitstream*) ser gerado (XILINX, 2013).

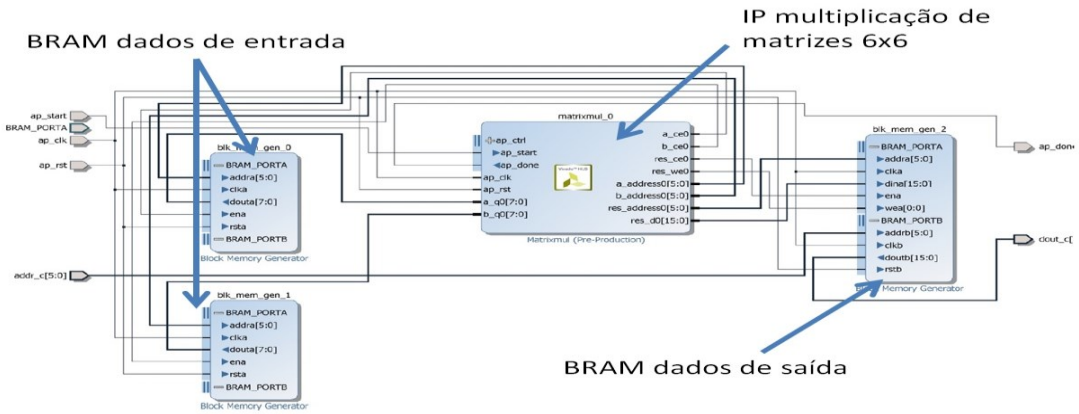


Figura 27 Exemplo para integração de projeto utilizando IP gerado pelo HLS e BRAMs no Vivado design Suite Tools (AUTOR)

A figura 27 ilustra um exemplo de arquitetura de projeto desenvolvido no Xilinx Vivado Design Suite Tools, utilizando um IP desenvolvido na ferramenta Xilinx Vivado High Level Synthesis (HLS). A aplicação utilizada foi multiplicação de matrizes 6x6 com dados de entrada de 8 bits e dados de saída de 16 bits. Neste exemplo, a interface de comunicação e transferência de dados foi configurada para BRAMs, instanciadas do catálogo de IPs do Vivado Design Suite Tools, com função de armazenar os dados de entrada e saída. O controle dos sinais de início (*Start*), resetar (*Reset*) e leitura de endereços das memórias é realizado por uma máquina de estados finitos (FSM) implementado num arquivo HDL adicionado ao projeto.

Figura 28 Simulação IP HLS multiplicação de matrizes 6x6 no Vivado Design Suite Tools (AUTOR)

A figura 28 exibe a simulação do IP HLS, com sinais e pinos destacados como: iniciar (*start_mat*), resetar (*reset_mat*), dados de entrada (a_q0, b_q0) e saída (res_d0). Os dados de entrada utilizados neste exemplo foram o valor ‘85’. O resultado da primeira operação de multiplicação linha x coluna é (85*85 = 7225) destacado na imagem.

É possível implementar outras formas de controle e envio e recepção dos dados. Por exemplo, usar um núcleo de microprocessador implementado na memória, do inglês *Soft Microprocessor Core* como o *Microblaze* (XILINX, 2008), que foi a arquitetura utilizada neste trabalho, apresentada no capítulo 5.

4 TÉCNICAS DE TOLERÂNCIA A FALHAS EM FPGAS PROGRAMADOS POR SRAM

Atualmente sistemas eletrônicos utilizados em aeronaves comerciais ou espaciais contém uma grande quantidade de componentes digitais e analógicos que são altamente sensíveis aos efeitos da radiação, por consequência disso devem ser qualificados ou protegidos para atuarem nesses ambientes. Por exemplo, nos projetos de aplicações espaciais os componentes tolerantes a radiação são largamente utilizados, porém com custos elevados.

A minimização de custos e tempo de desenvolvimento do projeto são fatores que levaram a uma tendência de utilização de componentes com qualificações militares em sistemas aeroespaciais e aviônicos que requerem alta confiabilidade, chamados de *Components Off-The-Shelf* (COTS), quando comparados a dispositivos tolerantes a radiação. Alguns FPGAs modernos que possuem componentes COTS podem facilitar o tempo de colocação no mercado, em virtude da confiabilidade e certificações apresentadas pelos microprocessadores embarcados (*Softcores*) pelos fabricantes de FPGAs (por exemplo Microblaze da Xilinx, Nios II da altera) e outros fabricantes de propriedade intelectual (IP) como a ARM (DU B.; DESOGUS M.; STERPONE L, 2015; TAMBARA, 2013).

As últimas gerações de semicondutores tem apresentado uma maior complexidade, o que torna necessário o aprimoramento das técnicas de tolerância às falhas, isso porque o dispositivo alvo é fortemente associado à técnica utilizada, o que torna necessário investigar os efeitos de uma ocorrência de inversão de bit na arquitetura utilizada. A técnica de tolerância a falhas contra efeitos de radiação de um circuito ou sistema depende da sua fase de concepção e pode ser aplicada em três níveis distintos durante o desenvolvimento de uma aplicação: nível de processo, ocorre quando determinado processo ou tecnologia de fabricação é alterada com objetivo de dar características de tolerância a radiação para um dispositivo. Em nível de projeto, ocorre quando a estrutura ou lógica de um circuito alvo é alterada com objetivo de implementar técnicas de tolerância a SEEs ou a dose total ionizante, do inglês *Total Ionizing Dose* (TID), que decorrem da exposição do componente eletrônico a radiação por um certo período de tempo e pode ter por consequências a alteração das características elétricas de partes do dispositivo pelo motivo das cargas elétricas acumuladas nas interfaces dos semicondutores induzidas pela radiação. Em nível de sistema, ocorre quando a implementação do sistema é alterada com objetivo de se obter tolerância a falhas (TAMBARA, 2013; KASTENSMIDT, 2004).

O uso de FPGAs comerciais (COTS) em aplicações que requerem alto grau de confiabilidade tem aumentado nos últimos tempos e tem relação com técnicas de proteção em nível de sistema, pois nesses dispositivos não existe nenhuma intervenção nas etapas de fabricação e projetos de componentes para a inserção de recursos de tolerância a falhas. Outro ponto importante no desenvolvimento de técnicas de tolerância a falhas em nível de sistema refere-se às regulamentações internacionais, do inglês *International Traffic in Arms Regulations* (ITAR), impostas pelos Estados Unidos no comércio de componentes eletrônicos. O ITAR apresenta algumas regras de restrições de exportação para fabricantes de circuitos integrados dos Estados Unidos, pois considera que componentes eletrônicos tolerantes a radiação podem ser destinados a fabricação de artefatos bélicos (COOK, 2010).

Algumas das principais técnicas de tolerância a falhas com objetivo de detectar ou mitigar erros em nível de sistema apresentam em sua composição algum tipo de redundância e se baseiam em replicar componentes chamado de redundância espacial (redundância de *hardware*) ou replicar o tempo de execução de uma tarefa conhecida por redundância temporal (TAMBARA, 2013).

As técnica de redundância temporal são habitualmente utilizadas para detectar eventos simples transitórios (SETs) em circuitos de lógica combinacional (KASTENSMIDT, 2006). Técnicas de redundância espacial são utilizadas para identificar SEUs em circuitos de lógica sequencial. Exemplos do uso dessas técnicas podem ser encontrados em (NICOLADIS,1999; ANGHEL, 2000; DUPONT, 2002)

O objetivo da redundância temporal é dispor das características do pulso transitório gerado pela colisão de uma partícula para comparar a saída de um circuito em dois momentos distintos, isto é, capturar a saída de um circuito de lógica combinacional em dois momentos diferentes, no qual a borda de um sinal de relógio é deslocada por um tempo ‘d’, ilustrado na figura 29. Grande parte das técnicas de detecção de falhas em *hardware* consiste na duplicação de componentes, no qual a saída das cópias é enviada a um circuito comparador, responsável por identificar a existência ou não de uma falha.

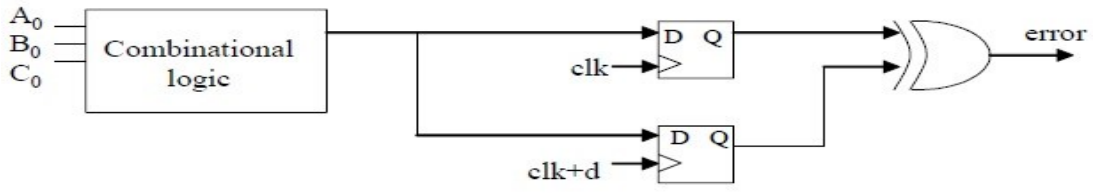


Figura 29 Exemplo de esquema para detecção de SETs presentes em uma lógica combinacional utilizando redundância temporal, onde o pulso de relógio é representado por clk, clk+d (KASTENSMIDT, 2006)

A técnica de duplicação com comparação (DWC, *Duplication With Comparasion*) tem como propósito duplicar o circuito em conjunto com um comparador na saída dos dois módulos para verificar ou detectar falhas representadas por um erro, não corrigindo a falha. Porém a correção da falha pode ser representada, por exemplo, com reinicialização do sistema. Esta técnica também pode ser combinada com outras técnicas como a CED (*Concurrent Error Detection*) que seleciona qual a saída esta correta após a comparação entre as duas saídas dos módulos, ilustrada na figura 30.

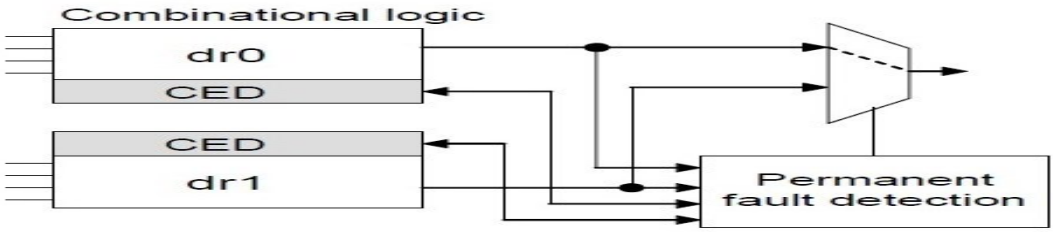
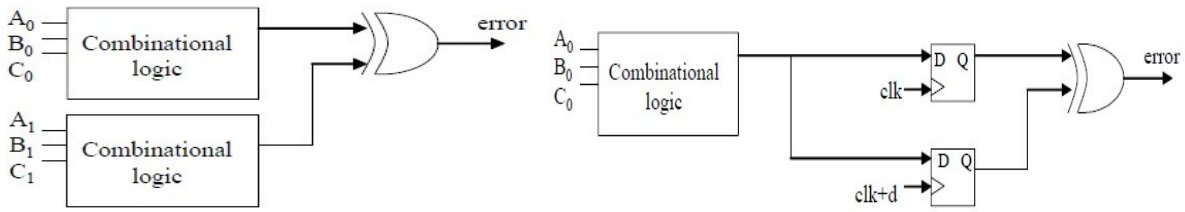


Figura 30 Duplicação com comparação combinada com CED (KASTENSMIDT, 2006)

A duplicação com comparação (DWC) pode ser utilizada em ambas as lógicas. Na lógica combinacional para detectar SETs e na lógica sequencial para detectar SEUs, ilustrados na figura 31 (a,b) respectivamente.

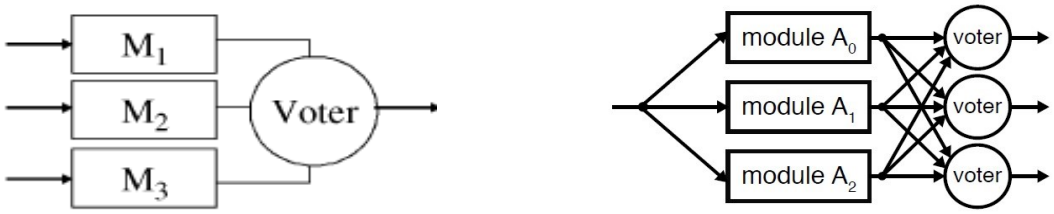


(a) (b)

Figura 31 Exemplo de esquema para detecção de SET na lógica combinacional e SEU na lógica sequencial (KASTENSMIDT, 2006)

Os FPGAs baseados em SRAM possuem matrizes densas de células de memória que são muito sensíveis a SEUs. Alguns métodos de mitigação de erros são necessários para proteger circuitos em FPGAs contra SEUs induzidos por radiação. Uma abordagem de mitigação em FPGAs bem comum utilizada é a Redundância Modular Tripla do inglês, *Triple Modular Redundancy* (TMR) (CARMICHAEL, 2001), que consiste na redundania espacial e ou temporal de componentes, ou seja, na triplicação do circuito ou partes do circuito onde as

saídas das cópias são aplicadas a um votador de maioria. Uma falha única em qualquer um dos módulos redundantes não produzirá erro na saída, como o votador majoritário irá selecionar o resultado correto dos outros dois módulos. O votador majoritário também pode ser triplicado para evitar um único ponto de falha (JOHNSON J, 2010), como ilustrado na figura 32 (a,b).



(a) (b)

Figura 32 Redundância Modular Tripla com um votador (a) e três votadores (b) (STERPONE, 2005)

A técnica TMR combinada com *Scrubbing* (recuperação) de configuração é utilizada como técnica de mitigação em ambientes de alta radiação para memórias configuráveis específicas Block RAMs Xilinx (MILLER, 2006). Alguns métodos utilizados para alcançar tolerância a falhas em FPGAs baseados em SRAM em CLBs e recursos de roteamento foram desenvolvidos atualmente em (STOTT, 2008). Em (LAHRACH, 2010) desenvolveu uma técnica de tolerância a falhas que reúne as capacidades de reconfiguração parcial do FPGA baseado em SRAM combinada com TMR. Em (KASTENSMIDT, 2005) investiga um projeto ideal para a lógica TMR, inserindo os votadores majoritários em posições escolhidas visando à robustez. Outra abordagem sobre a técnica TMR baseado em módulo de Grão Fino com técnica de recuperação, avaliada através de campanha de injeção de falhas num FPGA Xilinx Artix 7 é apresentada em (ZHAO, 2016).

Existe uma classificação para a técnica TMR ao que se refere à granularidade e localização dos votadores majoritários. Quando os votadores são colocados apenas nas saídas do projeto, pode ser chamado de TMR de Grão Grosso e quando os votadores são colocados nas saídas de todos os registradores (*flip-flops*) selecionados e/ou lógica combinacional dependendo dos requisitos de concepção, pode ser chamado de TMR de Grão Fino.

Embora TMR possa ser eficaz esta técnica pode representar um alto custo de sobrecarga de recursos do dispositivo, geralmente pelo menos três vezes o tamanho da área do circuito original. O TMR pode ser implementado em *hardware* em nível RTL, por exemplo, onde cada módulo é triplicado e os votadores majoritários de saída são adicionados. TMR também

pode ser implementado em *Software*, onde partes do código, módulos e funções são triplicadas, porém se ocorrer falha em alguma das cópias replicadas, por consequência a mesma falha poderá se manifestar nas outras cópias.

Nesta dissertação, o TMR foi implementado em C para uso no HLS e estudo de caso. A primeira implementação foi TMR de Grão Grosso Paralela (TMR *Coarse Grain Parallel*), que faz três chamadas da função de multiplicação de matrizes na função principal. Por consequência são gerados três módulos de multiplicação de matrizes sendo executados em paralelo.

A segunda versão é TMR de Grão Grosso Serial (TMR *Coarse Grain Serial*), que executa três vezes a multiplicação de matrizes dentro da função principal de forma sequencial, sem chamadas de função. Desta forma, a execução do segundo bloco multiplicador tem início um ciclo de relógio depois do primeiro e assim sucessivamente. Por consequência o HLS não gera blocos separados para as multiplicações e o RTL fica mais espalhado de certa forma, com mais ciclos de tempo de execução se comparado ao projeto CGP, pois é necessário esperar o último bloco multiplicador terminar para fazer a votação.

A terceira versão é TMR de Grão Fino Paralela (TMR *Fine Grain Parallel*), onde são colocados votadores majoritários dentro da função de multiplicação de matrizes a nível de registradores e também são colocados votadores majoritários nas saídas dos módulos. São realizadas três chamadas da função de multiplicação de matrizes de forma similar ao TMR CGP, porém são executadas uma após a outra e não de forma paralela, o que aumentou o tempo de execução do projeto.

Todas as versões foram implementadas nas arquiteturas propostas *Single Stream* e *Multiple Stream* que serão apresentadas no capítulo 5.

5 ESTUDO DE CASO: TMR DE GRÃO GROSSO SERIAL E PARALELO E TMR DE GRÃO FINO PARALELO PARA MULTIPLICAÇÃO DE MATRIZES

O propósito deste trabalho é aplicar a técnica de redundância TMR diretamente no algoritmo baseado na linguagem C da aplicação, neste caso multiplicação de matrizes 6x6. Sintetizar o projeto pela ferramenta de Síntese de Alto Nível (HLS) da Xilinx e verificar a eficácia da redundância sob injeção de falhas por emulação (falhas transientes) no FPGA Xilinx Artix 7. Os modos de injeção de falhas utilizados foram os seguintes: Injeção de Falhas Simples (SEU) e Injeção de Falhas Acumuladas (SEU acumulados).

Para aceleradores de *hardware* a interface responsável por receber o fluxo de dados de carga de trabalho é muito importante. Nos dispositivos Xilinx, geralmente os aceleradores de *hardware* de alto desempenho são conectados a processadores *soft-core* ou *hard-core* através de um interface DMA (*Direct Memory Acess*) e AXI (*Advanced eXtensible Interface Stream*). Essa infraestrutura de interconexão fornece um controle com *pipeline*, permitindo que o *software* executado no processador coloque várias solicitações de tarefas, reduzindo a latência. A interface de transferência de dados do Microblaze para o IP HLS é realizada pelo barramento AXI (TAMBARA, 2017; SANTOS, 2017).

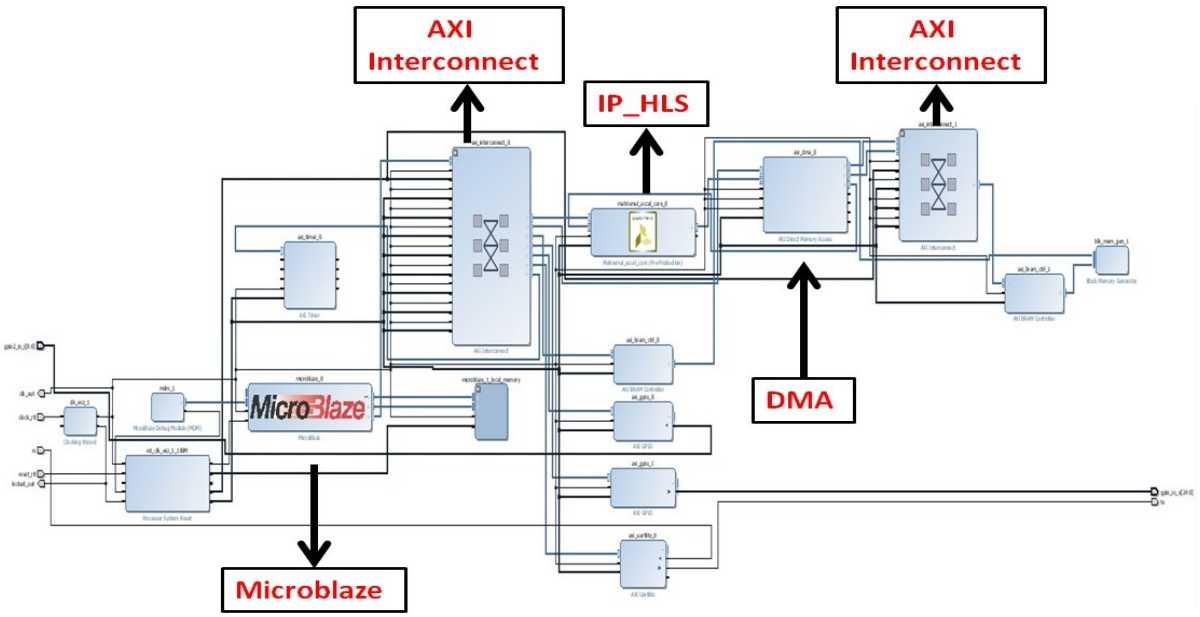


Figura 33 Arquitetura de implementação do projeto Single Stream utilizando um núcleo de microprocessador Microblaze (AUTOR)

Este trabalhou utilizou duas arquiteturas para implementação dos projetos, chamadas de *Single Stream* e *Multiple Stream*. A figura 33 ilustra o modelo de projeto (*Block Design*) da arquitetura chamada *Single Stream*. Esta arquitetura opera apenas um barramento para enviar os dados de entrada e receber os resultados entre o Microblaze e o IP HLS. O microblaze envia os dados de entrada para o IP HLS e recebe os resultados que são comparados com uma execução correta, com objetivo de verificar se a execução da aplicação foi realizada sem erros. É utilizado apenas um IP DMA para a transferência de dados entre microblaze/memória para o IP HLS, o que pode ocasionar problemas no restante da transferência se algum barramento for atingido por um SEU.

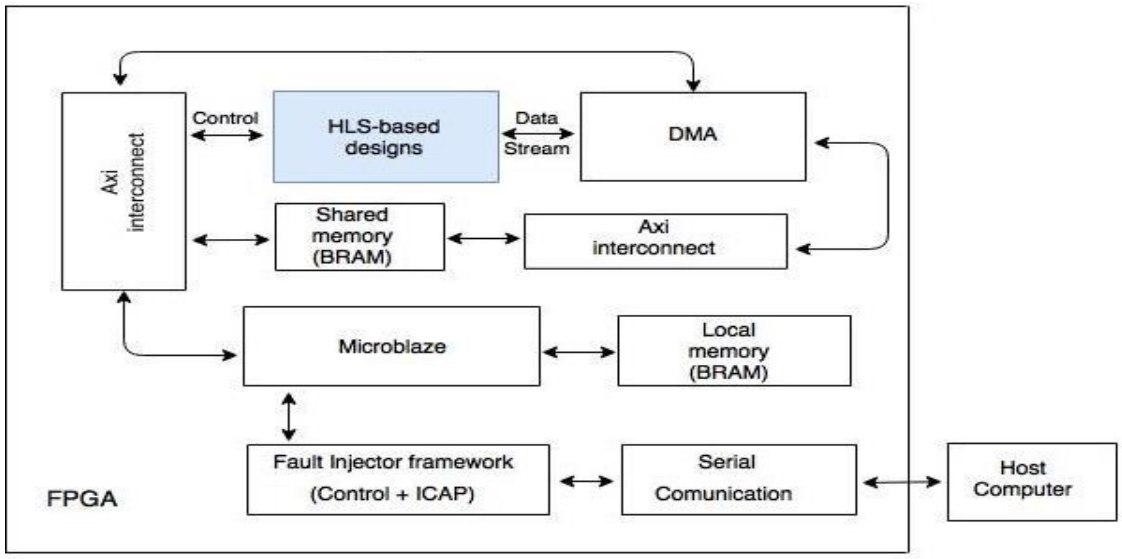


Figura 34 Esquema diagrama da arquitetura Single Stream implementada no FPGA (AUTOR)

O diagrama de esquema para a implementação da arquitetura *Single Stream* no FPGA comunicando-se via serial com o computador hospedeiro é ilustrado na figura 34. O computador envia os sinais de controle para o injetor de falhas através de um *Script* em *Python* e aguarda os resultados para serem salvos em arquivos de log.

O modelo de projeto (*Block Design*) da arquitetura *Multiple Stream* é exibido na figura 35. Esta arquitetura possui canais independentes para o envio e recepção dos dados de entrada e saída. São utilizados IPs DMAs independentes para cada canal de transferência de dados do microblaze/memória para o IP HLS, o que torna o projeto mais confiável, pois se algum dos barramentos for atingido por um SEU os outros ainda poderão ter o seu funcionamento correto e somente parte dos dados serão perdidos.

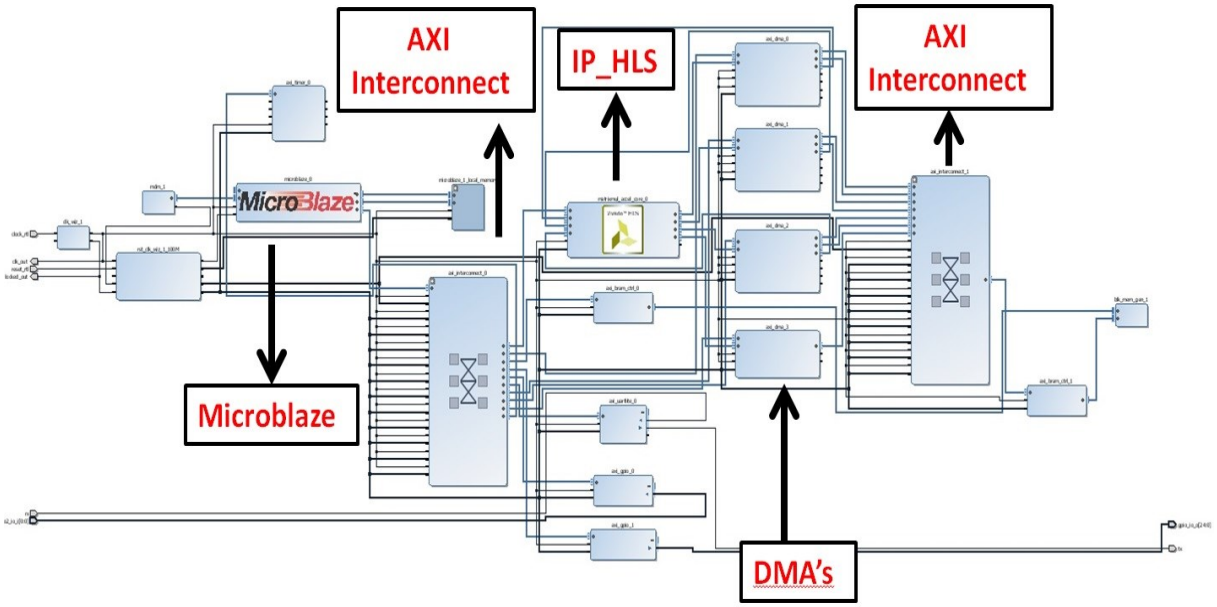


Figura 35 Arquitetura de implementação do projeto Multiple Stream utilizando um núcleo de microprocessador Microblaze (AUTOR)

O diagrama de esquema para a implementação da arquitetura *Multiple Stream* no FPGA, comunicando-se via serial com o computador hospedeiro é exibido na figura 36. Pode-se observar que cada canal DMA possui um barramento independente para transferência de dados entre o microblaze e o IP HLS.

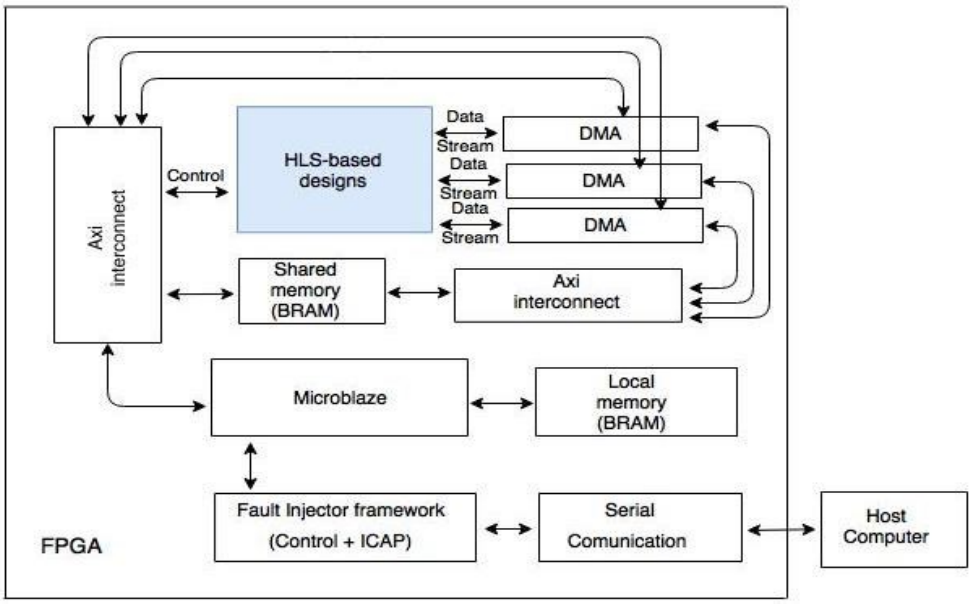


Figura 36 Esquema diagrama da arquitetura Multiple Stream implementada no FPGA e comunicando-se com o computador hospedeiro (Host Computer) (AUTOR)

As duas arquiteturas utilizadas neste trabalho, demonstradas acima, apresentam pequenos custos adicionais em termos de recursos (*overhead*) de área e desempenho. No capítulo 7 serão apresentados resultados de área, desempenho nos modos de injeção de falhas simples (SEU) e injeção de falhas acumulada (SEU acumulados).

Por padrão o HLS gera o IP com pinos de ligar/ativar (*Enable*), endereço (*Address*), pulso do relógio (*Clock*), ativar escrita (*Write enable*), entre outros. Através da ferramenta de projeto Xilinx Vivado Design Suite Tools, podemos obter as seguintes métricas: área em termos de utilização de recursos após o projeto ser sintetizado e roteado para o FPGA alvo, o desempenho do projeto em termos de ciclos de relógio e tempo de execução e também os bits essenciais de um projeto específico.

O tamanho dos dados de entrada e saída utilizados neste trabalho foram definidos da seguinte maneira: dados de entrada 8 bits (*Unsigned Char*) e dados de saídas 16 bits (*Unsigned Short*), representados na figura 37.

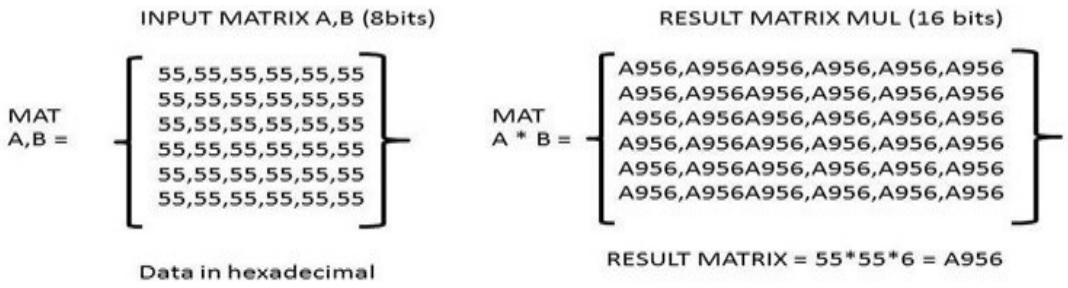


Figura 37 Dados de entrada e saída para matrizes 6x6 (AUTOR)

A representação do número de passos para executar a aplicação de multiplicação de matrizes 6x6, sem utilizar redundância é ilustrada na figura 38. A leitura das entradas e saída de resultados utiliza arquitetura de canal simples (*Single Stream*).

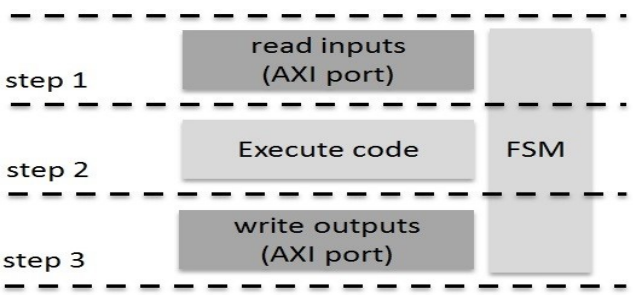


Figura 38 Representação do número de passos para executar a multiplicação de matrizes sem redundância (*matrix multiplication unhardened*) (AUTOR)

Existem diferentes maneiras de implementar redundância em C, de acordo com as chamadas de funções dentro da função principal. Para a aplicação de multiplicação de matrizes a função pode ser chamada três vezes, ou ser executada três vezes sequencialmente, o que pode interferir diretamente na geração do RTL.

Existe uma classificação para a técnica de redundância TMR ao que se refere à granularidade e localização dos votadores majoritários. Quando os votadores são colocados apenas nas saídas do projeto pode ser chamado de TMR de Grão Grosso e quando os votadores são colocados nas saídas de todos os registradores (*flip-flops*) selecionados e/ou lógica combinacional, dependendo dos requisitos de concepção, pode ser chamado de TMR de Grão Fino. Neste trabalho foram implementadas três versões de Redundância Modular Tripla (TMR) chamadas de: TMR de Grão Grosso Paralela (TMR *Coarse Grain Parallel, CGP*), TMR de Grão Grosso Serial (*TMR Coarse Grain Serial, CGS*) e TMR de Grão Fino Paralela (*TMR Fine Grain Parallel, FGP*), utilizando as arquiteturas *Single* Stream e *Multiple Stream* com e sem otimizações, apresentados na tabela 2.

Na implementação do TMR de Grão Grosso Serial (TMR CGS, *TMR Coarse Grain Serial*) são três execuções de multiplicação de matrizes (*Execute code*) em série dentro da função principal (sem chamada de função). Para a arquitetura de canal único de transferência de dados (*Single Stream*) temos a aplicação de apenas um votador (*Majority voter*) na saída dos resultados e para a arquitetura de múltiplos canais de transferência de dados (*Multiple Stream*) temos a aplicação de três votadores na saída dos resultados, ilustrados nas figuras 39 e 40 respectivamente.

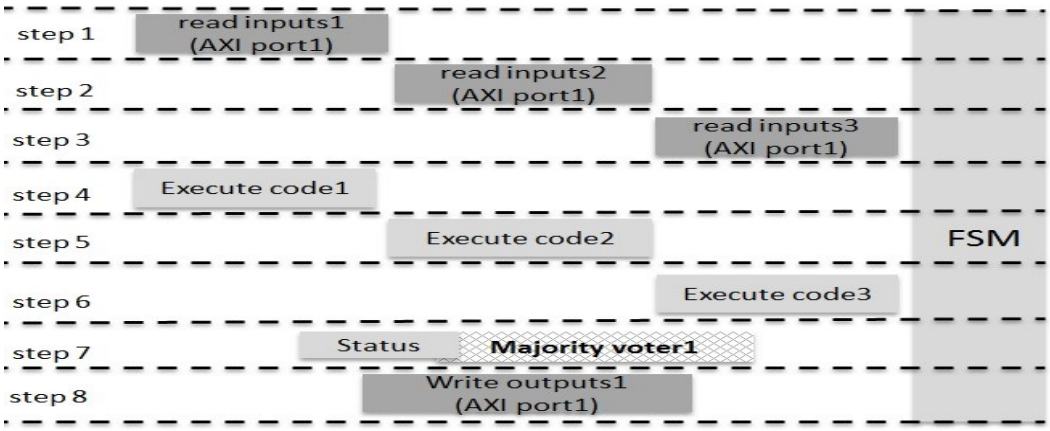


Figura 39 Representação do número de passos para executar a multiplicação de matrizes *TMR CGS Single Stream* (AUTOR)

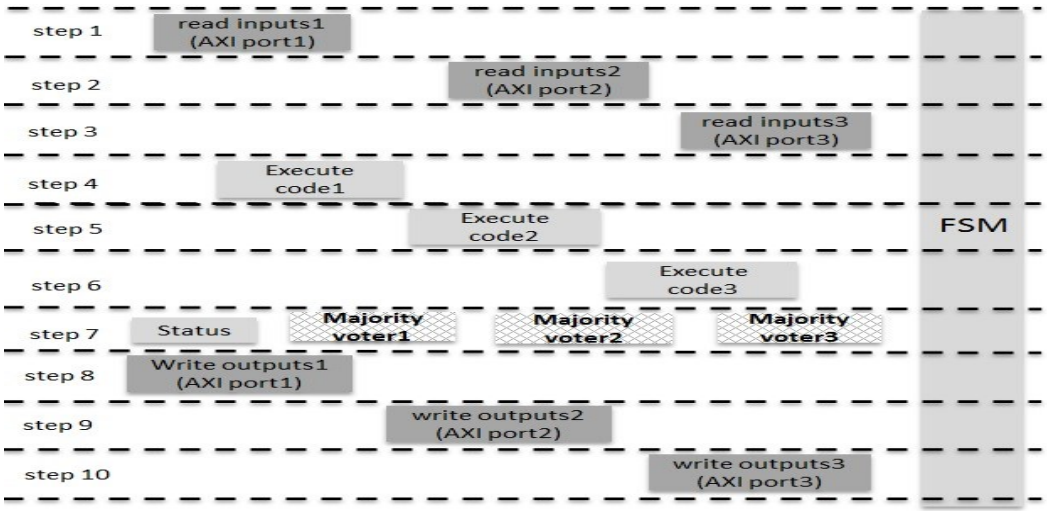


Figura 40 Representação do número de passos para executar a multiplicação de matrizes *TMR CGS Multiple Stream* (AUTOR)

A leitura das entradas (*Read inputs*) e saídas de resultados (*Write outputs*) na arquitetura *Single Stream* é realizada apenas por um canal (*AXI port1*) demonstrado na figura 39. Diferente da versão *Multiple Stream* que utiliza canais independentes (*AXI port1, AXI port2, AXI port3*) exibido na figura 40.

Na implementação do projeto TMR de Grão Grosso Paralelo (TMR CGP, TMR *Coarse Grain Parallel*) são feitas três chamadas da função de multiplicação de matrizes dentro da função principal, o que resultará em três módulos sendo executados em paralelo (*Execute code*). A configuração de votadores majoritários (*Majority voters*) para TMR CGP segue o mesmo padrão do TMR CGS. Para a arquitetura de canal único de transferência de dados (*Single Stream*) temos a aplicação de apenas um votador (*Majority voter*) na saída dos resultados e para a arquitetura de múltiplos canais de transferência de dados (*Multiple Stream*) temos a aplicação de três votadores na saída dos resultados, ilustrados nas figuras 41 e 42 respectivamente. A função '*Status'* verifica a existência de algum valor diferente entre os módulos.

A leitura de entradas (*Read inputs*) e saída (*Write outputs*) de resultados na arquitetura *Single Stream* é realizada apenas por um canal (*AXI port*), ilustrado na figura 41. Ao contrário da arquitetura *Multiple Stream*, que utiliza canais independentes exibido na figura 42 (*AXI port1, AXI port2, AXI port3*).

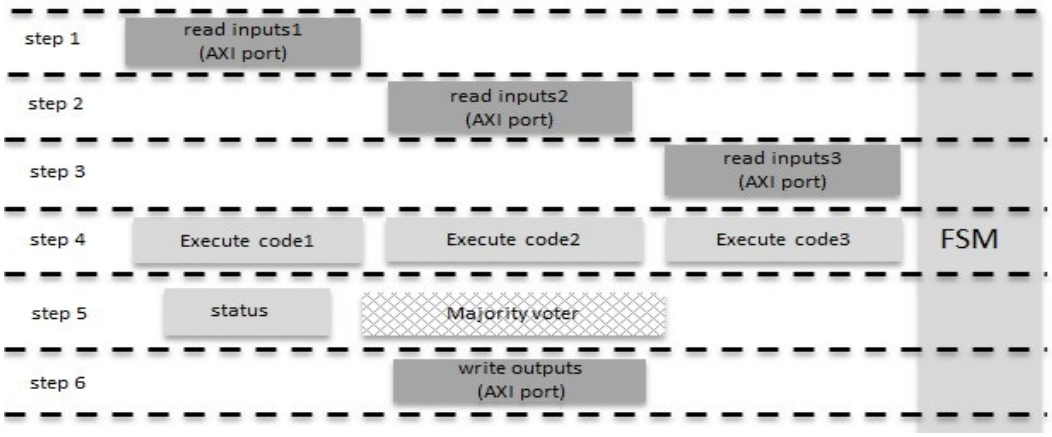


Figura 41 Representação do número de passos para executar a aplicação de multiplicação de matrizes *TMR CGP Single Stream* (AUTOR)

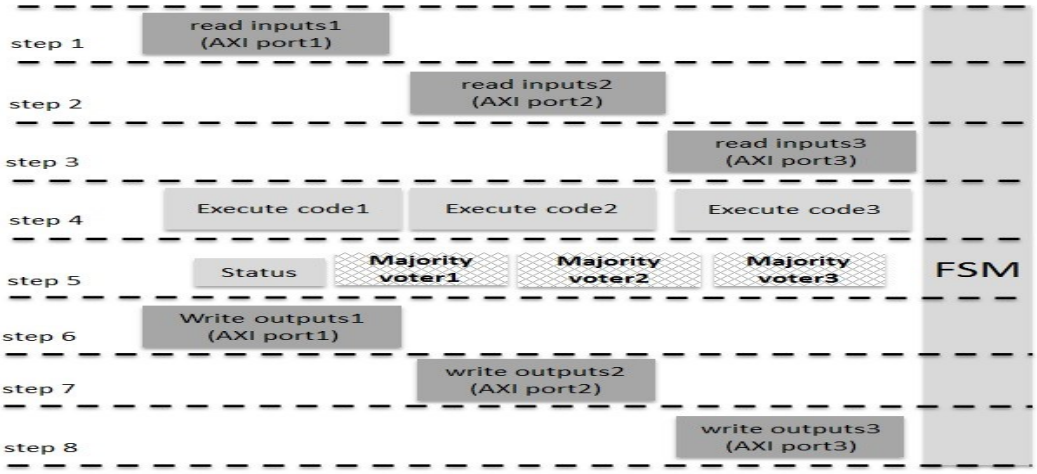


Figura 42 Representação do número de passos para executar a aplicação de multiplicação de matrizes *TMR CGP Multiple Stream* (AUTOR)

A implementação do projeto de Grão Fino Paralelo (FGP, *TMR Fine Grain Parallel*), além de utilizar votadores majoritários nas saídas de forma similar aos outros projetos, faz votação majoritária dentro da função de multiplicação de matrizes (nível de registradores), que tem uma versão específica para este projeto.

A ordenação das chamadas de função dentro da função principal no projeto de TMR Grão Fino Paralelo ocorre de forma similar à implementação de TMR Grão Grosso Paralelo. A diferença esta na forma que os dados são enviados para a função e são votados durante a multiplicação de matrizes.

Na implementação do TMR de Grão Fino Paralelo (TMR FGP, *TMR Fine Grain Parallel*) são realizadas três chamadas da função de multiplicação de matrizes dentro da função principal que resultará em três módulos sendo executados, um após o outro, o que

aumenta o tempo de execução da aplicação, principalmente na versão sem otimização. Além dos votadores majoritários colocados dentro da função de multiplicação de matrizes, temos a configuração similar às outras arquiteturas: para a arquitetura de canal único de transferência de dados (*Single Stream*) temos a aplicação de apenas um votador (*Majority voter*) na saída dos resultados e para a arquitetura de múltiplos canais de transferência de dados (*Multiple Stream*) temos a aplicação de três votadores na saída dos resultados, ilustrados nas figuras 43 e 44 respectivamente. A função '*Status'* verifica a existência de algum valor diferente entre os módulos.

Nesta versão TMR FGP são seis matrizes sendo multiplicadas dentro da função de multiplicação de matrizes ao invés de duas, o que difere das outras duas implementações.

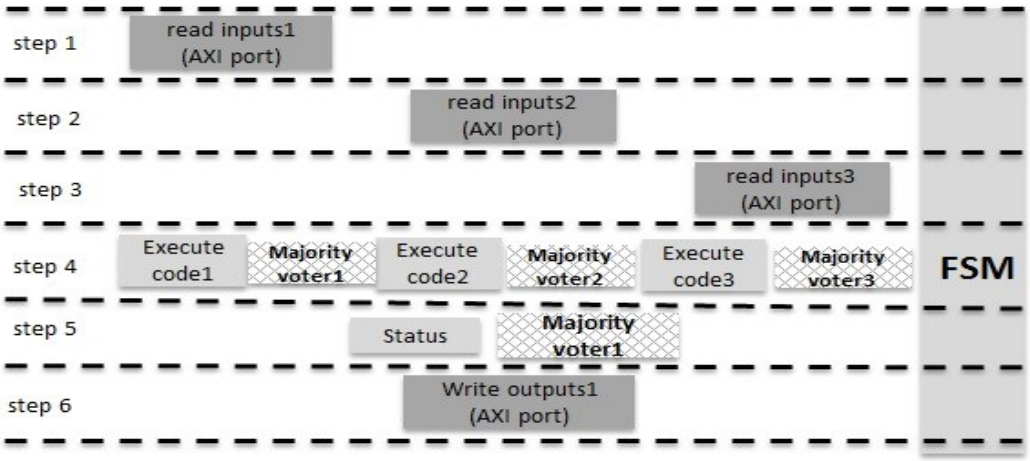


Figura 43 Representação do número de passos para executar a aplicação de Multiplicação de Matrizes *TMR FGP Single Stream* (AUTOR)

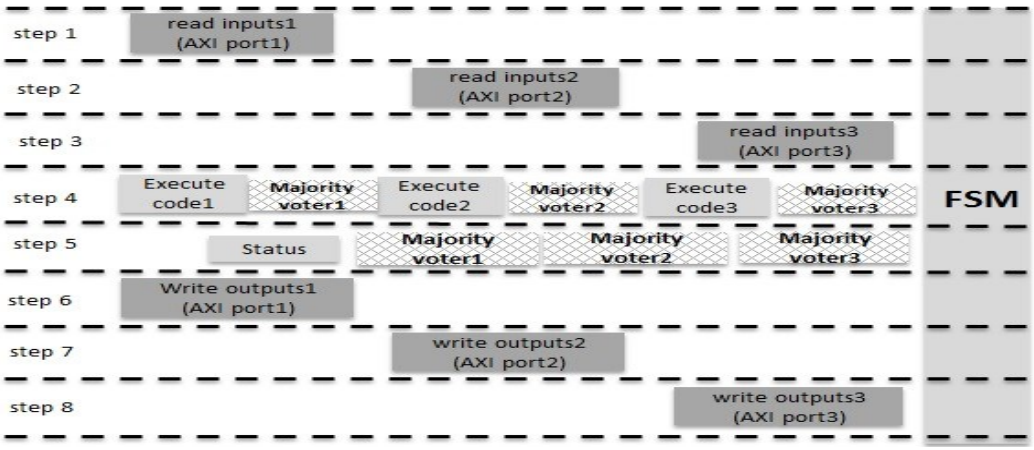
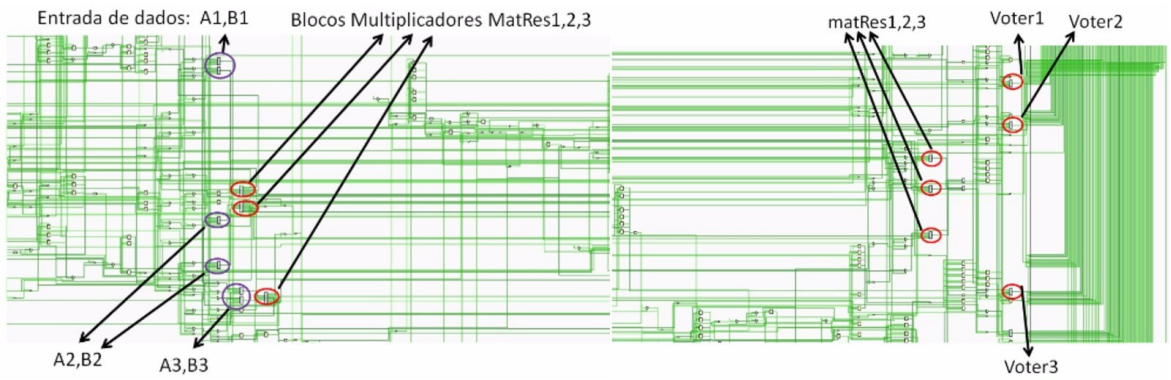


Figura 44 Representação do número de passos para executar a aplicação de multiplicação de matrizes *TMR FGP Multiple Stream* (AUTOR)

A leitura das entradas (*Read inputs*) e saídas (*Write outputs*) de resultados na arquitetura *Single Stream* é realizada apenas por um canal (*AXI port*) demonstrado na figura43. A versão *Multiple Stream* utiliza canais independentes (*AXI port1, AXI port2, AXI port3*) exibido na figura 44.

Pode-se observar na figura 45 (a,b) o RTL gerado para o Projeto TMR CGP pela ferramenta de Síntese de Alto Nível e sintetizado para o FPGA Artix 7 no Vivado Design Suite Tools. Para a entrada de dados temos: a1, b1, a2, b2, a3, b3.



(a) **(b)**

Figura 45 RTL gerado para o projeto TMR Grão Grosso Paralelo (*TMR CGP Multiple Stream*) (AUTOR)

Os blocos multiplicadores são triplicados pela razão de haver três chamadas da função multiplicação de matrizes na função principal, no algoritmo do projeto TMR CGP em linguagem C. Logo após o resultado das multiplicações de matrizes são aplicados três votadores majoritários nas saídas, ilustrados na figura 45 (b).

Em resumo, os projetos implementados para os dois modos de injeção de falhas são apresentados na tabela 2.

Tabela 2 Projetos escolhidos para implementação nos modos de Injeção de Falhas Simples e Falhas Acumuladas

Lista de Projetos Implementados	Injeção de Falhas Simples	Injeção de Falhas Acumuladas
Unhardened	**X**	**X**
Unhardened Pipeline	**X**	
Unhardened Unroll	**X**	
TMR CGP Single Stream	**X**	**X**
TMR CGP Single Stream Pipeline	**X**	

TMR CGP Multiple Stream	x	x
TMR CGP Multiple Stream Pipeline	x	
TMR CGP Multiple Stream Unroll	x	
TMR CGS Single Stream	x	
TMR CGS Single Stream Pipeline	x	
TMR CGS Multiple Stream	x	
TMR CGS Multiple Stream Pipeline	x	
TMR CGS Multiple Stream Unroll	x	
TMR FGP Single Stream	x	
TMR FGP Single Stream Pipeline	x	
TMR FGP Multiple Stream	x	
TMR FGP Multiple Stream Pipeline	x	
TMR FGP Multiple Stream Unroll	x	

6 INJETOR DE FALHAS NO BITSTREAM

A injeção de falhas, do inglês *Fault Injection (FI)*, por emulação é um método bem conhecido para analisar a confiabilidade de um projeto implementado num FPGA baseado em SRAM. O fluxo de bits (*Bitstream*) original configurado no FPGA pode ser modificado por um circuito ou uma ferramenta de computador, invertendo, do inglês *flipping*, um dos bits do fluxo de bits, um de cada vez. Essa inversão de bit emula um SEU nas células de memória de configuração. O injetor de falhas tem proveito sob as capacidades de reconfiguração parcial dinâmica dos FPGAs baseados em SRAM para reduzir o tempo da aplicação de inversão de bits, do inglês *bit-flips*.

Alguns parâmetros devem ser utilizados para caracterizar um projeto em FPGA baseado em SRAM em erros transitórios, do inglês *Soft Errors*. A área de um projeto implementado pode ser expressa em termos do número de recursos usados, tais como LUTs, registradores (*flip-flops*), blocos BRAM (memória configurável), blocos processadores de sinais digitais DSP (*Digital Signal Processor),* etc. Além disso, é possível expressar a área em termos de quadros de configuração, do inglês *Configuration Frames*. Os quadros de configuração de um FPGA são definidos como o menor segmento de memória endereçável da memória de configuração (*Bitstream*). Cada frame está relacionado a um recurso e posição específica na área disponível do FPGA (XILINX, 2015), sendo possível calcular o número de frames de configuração usados por um projeto. O desempenho de um projeto pode ser expresso em termos de tempo de execução, freqüência operacional e carga de trabalho processada. O tempo de execução pode ser definido pelo número de ciclos de relógio (*clock cycles*) para executar a operação. De acordo com o FPGA e arquitetura de um projeto, uma frequência de relógio máxima é ajustada. Outro parâmetro importante é a carga de trabalho processada pelo projeto, definida como a quantidade de dados computados em uma execução. Em termos de confiabilidade de um projeto, as informações de desempenho são úteis para saber quanto tempo o projeto é exposto a falhas transientes durante a execução da função implementada até o final, apresentando as saídas dos resultados.

Os bits essenciais são definidos em (XILINX, 2012) e referem-se à quantidade de bits de configuração associados a um projeto mapeado num determinado FPGA. Os bits essenciais são um subconjunto dos bits de configuração total e seu valor depende da área do projeto implementado.

Os erros são definidos como qualquer comportamento inesperado nos resultados. Eles podem ser classificados como: erros de corrupção silenciosa de dados, do inglês *Silent Data Corruption* (SDC); ou erros de travamentos do processamento, do inglês *Hangs*; ou erros de tempo excedido, do inglês *Timeout Errors*. Os bits críticos são definidos como a quantidade de bits de configuração que, uma vez invertidos, causam um erro no comportamento do projeto esperado.

Os bits críticos são um subconjunto dos bits essenciais. O valor de bits essenciais é gerado pela ferramenta Vivado Design Suite Tools, através da linha de comando de restrição: "set_property bitstream.SEU.ESSENTIALBITS yes [current_design] " (TONFAT, 2016).

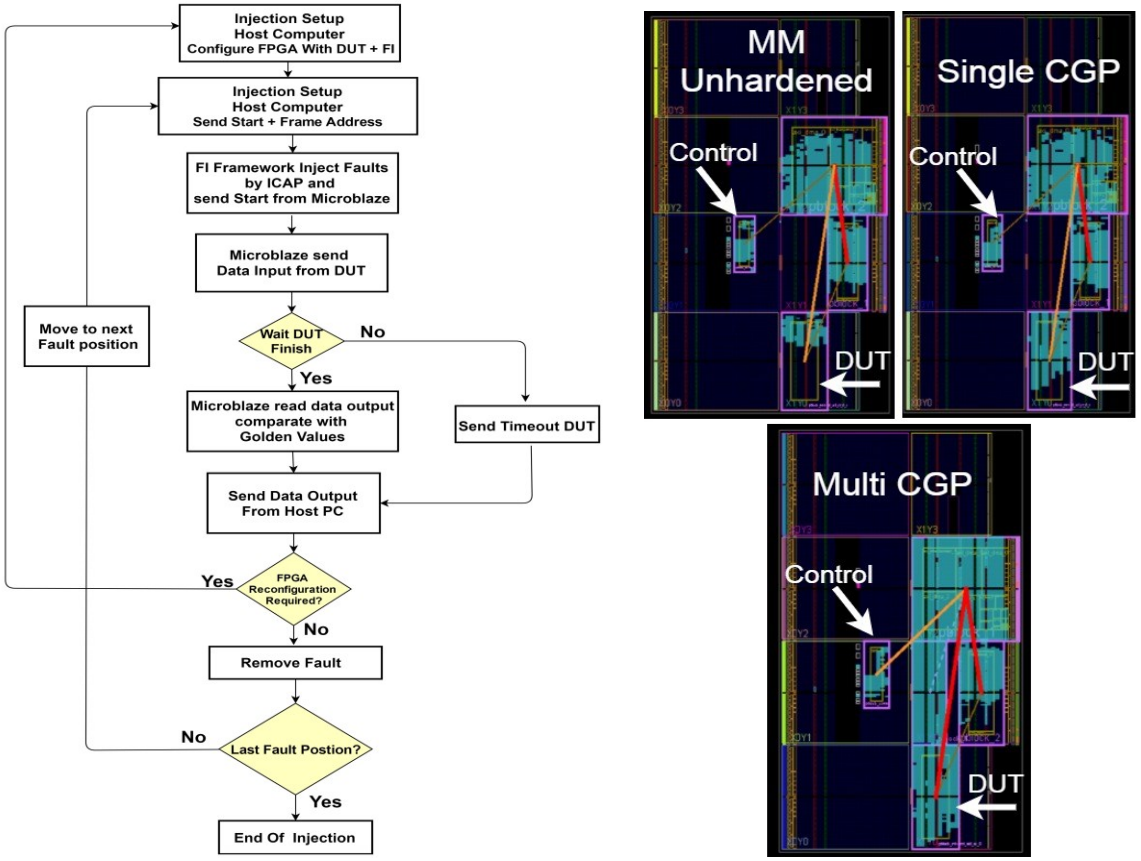


Figura 46 Definição para metodologia de injeção de falhas simples e o planejamento da área ocupada do FPGA (b) (AUTOR)

A saída do projeto sob teste, do inglês *Design Under Test* (DUT) pode ser constantemente monitorada para analisar o efeito da falha injetada no projeto. Se um erro for detectado, isso significa que esse bit da memória de configuração é um bit crítico. É possível injetar falhas em todos os bits de configuração e obter os bits críticos do projeto, este modo é

chamado de injeção de **falhas simples** ilustrado na figura 46 (a,b). A metodologia de injeção de falhas é controlada por um *Script* na linguagem de programação *Python* no computador hospedeiro que faz a comunicação serial com o FPGA. Após o *bitstream* ser carregado é enviado o endereço da posição do quadro de configuração (*Configuration Frame*), onde a falha será injetada e qual a posição do bit do quadro de configuração que a falha será injetada (*Bit Position*). Por exemplo, no FPGA Artix 7 utilizado neste trabalho, cada quadro de configuração tem 3232 bits. A seguir, é enviado o sinal de início (*Start*) para o injetor de falhas, intendente pela comunicação com o ICAP (*Internal Configuration Access Port*) e controle dos sinais de iniciar e resetar a arquitetura. O Microblaze é responsável pelo controle do IP HLS, envio dos dados de entrada e recepção dos resultados das multiplicações de matrizes. Todo o controle de recebimento de dados, envio de resultados via serial para o computador é realizado por uma máquina de estados finitos (FSM) no injetor de falhas. Os resultados são armazenados em arquivos de log com o endereço do quadro de configuração, posição do bit do quadro de configuração, sinal, do inglês *Flag*, com valor '0' ou '1' sinalizando se houver erro ou não na saída da aplicação.

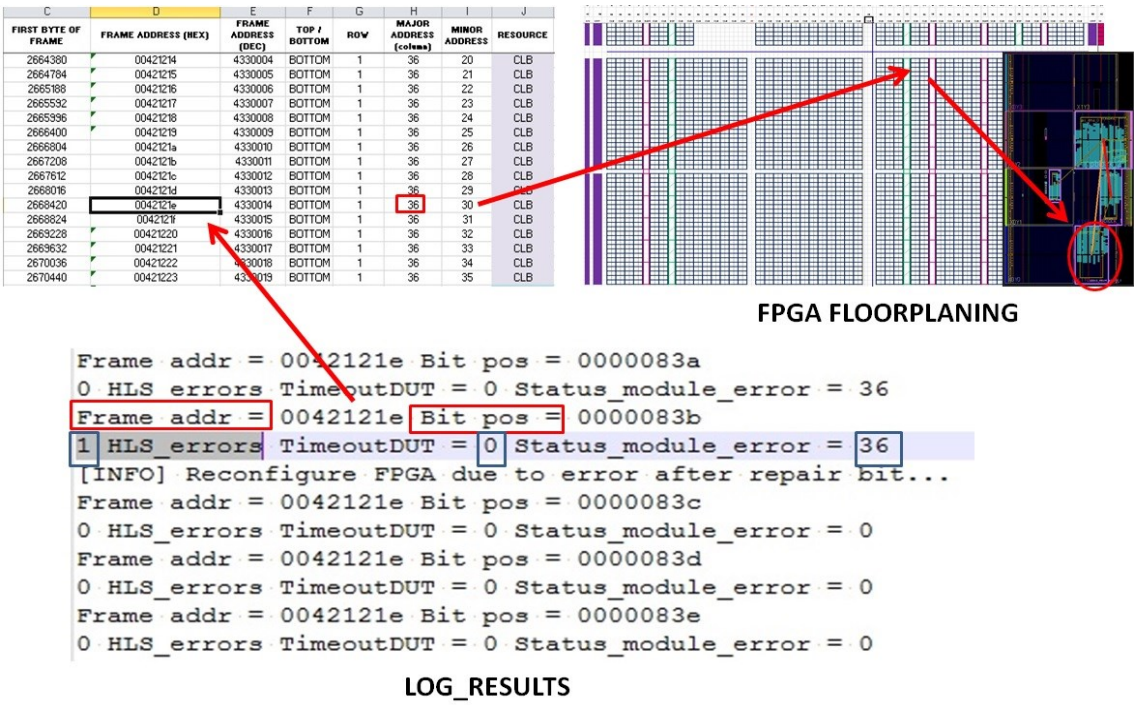


Figura 47 Posição do quadro da memória de configuração (*configuration frame*) onde a falha é injetada na área disponível do FPGA e exemplo do arquivo LOG gerado com resultados de um projeto composto de redundância modular tripla (TMR) (AUTOR)

A figura 47 ilustra o endereço do quadro da memória de configuração (*Frame addr*) onde a falha é injetada, bit por bit. Os resultados são analisados através de arquivos de log. '**0** HLS_errors', significa que não houve erros na saída da aplicação, caso contrário '**1** HLS_errors' define-se como um erro e a posição do bit é considerada como bit crítico. 'TimeoutDUT=0' significa que não foi excedido o tempo configurado para a transferência de algum bloco de dados do IP HLS para o Microblaze. Por exemplo, os dados de entrada para multiplicação de matrizes são as matrizes 'A1' e 'B1', os dados de resposta são representados pela matriz 'RES1'. Dependendo do projeto que esta sendo executado, o projetista vai definir quais blocos pretende analisar ou esperar um determinado tempo para transferência completa. Neste caso a figura 47 demonstra um exemplo de projeto com uso de redundância TMR. O que esta sendo analisado é o tempo de recebimento do bloco de resultados, cada um é definido com tempo de 'um' segundo, controlado por um temporizador, do inglês, *Watchdog*, no código executado pelo Microblaze. Se este tempo for excedido em mais de dois blocos é considerado um 'TimeoutDUT', especificamente para o projeto com redundância TMR definido por três blocos iguais. Pode ser que parte dos dados tenha sido transferido no tempo de 'um' segundo, ou o bloco todo tenha sido perdido, isto vai depender qual parte do circuito o SEU atingiu, a lógica do circuito, controle ou roteamento.

```
//get results from the vivado hls block 'mVoter' by TMR
status = XAxiDma_SimpleTransfer(&AxiDma1, (unsigned int)mCHH, dma_size_res,
XAXIDMA_DEVICE_TO_DMA);
XTmrCtr_Reset(&timer_dev, XPAR_AXI_TIMER_0_DEVICE_ID); //reset timer
begin_time_mCHH = XTmrCtr_GetValue(&timer_dev, XPAR_AXI_TIMER_0_DEVICE_ID);
   while(XAxiDma_Busy(&AxiDma1, XAXIDMA_DEVICE_TO_DMA)){
      end_time_mCHH = XTmrCtr_GetValue(&timer_dev, XPAR_AXI_TIMER_0_DEVICE_ID);
       if( (end_time_mCHH-begin_time_mCHH)>100000000){ //1 seconds
            timeoutmCHH = 1;
       break;
       }
   }
```

Figura 48 Pseudocódigo para transferência de um bloco de dados do IP HLS para o Microblaze com controle de tempo de 1 segundo, executado no Microblaze. (AUTOR)

A figura 48 demonstra o controle de tempo para a transferência de dados do IP HLS para o Microblaze via barramento AXI. O canal utilizado foi *'&AxiDma1'*. O arranjo de dados está armazenado na variável '*mCHH*' neste exemplo. É utilizado um periférico com o nome de *Axi_Timer* do catálogo de IP's do Vivado Design Suite Tools (XILINX, 2016c), para controlar o tempo antes e depois da transferência do bloco. O cálculo é obtido subtraindo-se o tempo final do tempo inicial e verificando se é maior do que 'um' segundo. Se o tempo for

maior, segue para o próximo bloco, sendo considerado '*timeout*' do bloco específico, neste exemplo '*mCHH*'. O '*status_module_error*' é utilizado para definir quantos valores diferentes houveram na execução das três multiplicações, comparando-se a resposta dos módulos por posições. O valor pode variar de 0 a 36, porque neste trabalho foram utilizadas matrizes de entradas 6x6.

Toda a campanha de injeção de falhas pode passar de algumas horas ou dias, dependendo da quantidade de bits que serão invertidos e da conexão ao circuito de controle de injeção de falha. O método de injeção de falhas nos fornece a análise de erros, classificação de erros e os bits críticos. Após a gravação dos resultados e verificação da remoção da falha, se não for possível remove-la uma reconfiguração total é realizada e o *bitstream* é novamente carregado para o FPGA. O projeto é resetado e todo o fluxo de injeção de falhas inicia novamente até chegar a posição final do quadro de configuração especificado.

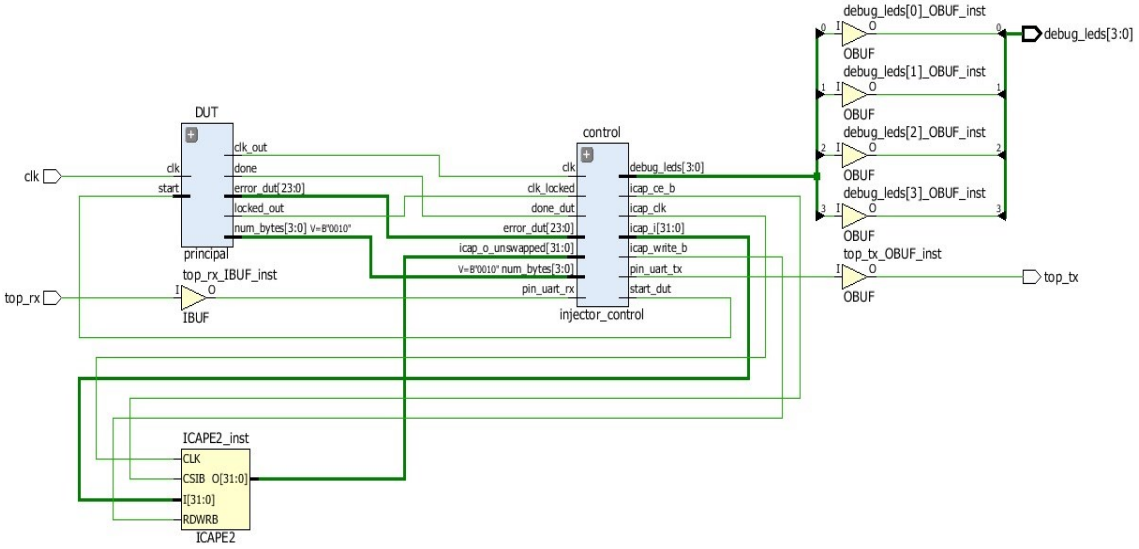


Figura 49 RTL padrão implementado no FPGA(Injetor de falhas + DUT) e comunicação com o computador hospedeiro através da serial (AUTOR)

O RTL do projeto descrito nas figuras 33 e 35, é exibido na figura 49, onde pode-se observar os blocos como: ICAP, controle do injetor de falhas, DUT (controle e Microblaze) e os pinos de comunicação serial '*top_rx*', '*top_tx*' para o computador.

O segundo modo de injeção de falhas por emulação utilizado neste trabalho é chamado de **injeção de falhas acumuladas** (SEU acumulados).

A metodologia de injeção de falhas acumulada tem o controle similar a injeção de falhas simples (sequencial), através de um *Script* em *Python* no computador hospedeiro responsável por realizar a comunicação serial com o FPGA. Porém, a diferença está em

sortear o valor aleatório do quadro de configuração onde a falha vai ser injetada e não recuperar o bit flipado (invertido), configurando-se uma injeção de falhas acumuladas onde a confiabilidade do projeto vai diminuindo ao decorrer do tempo até ocasionar um erro.

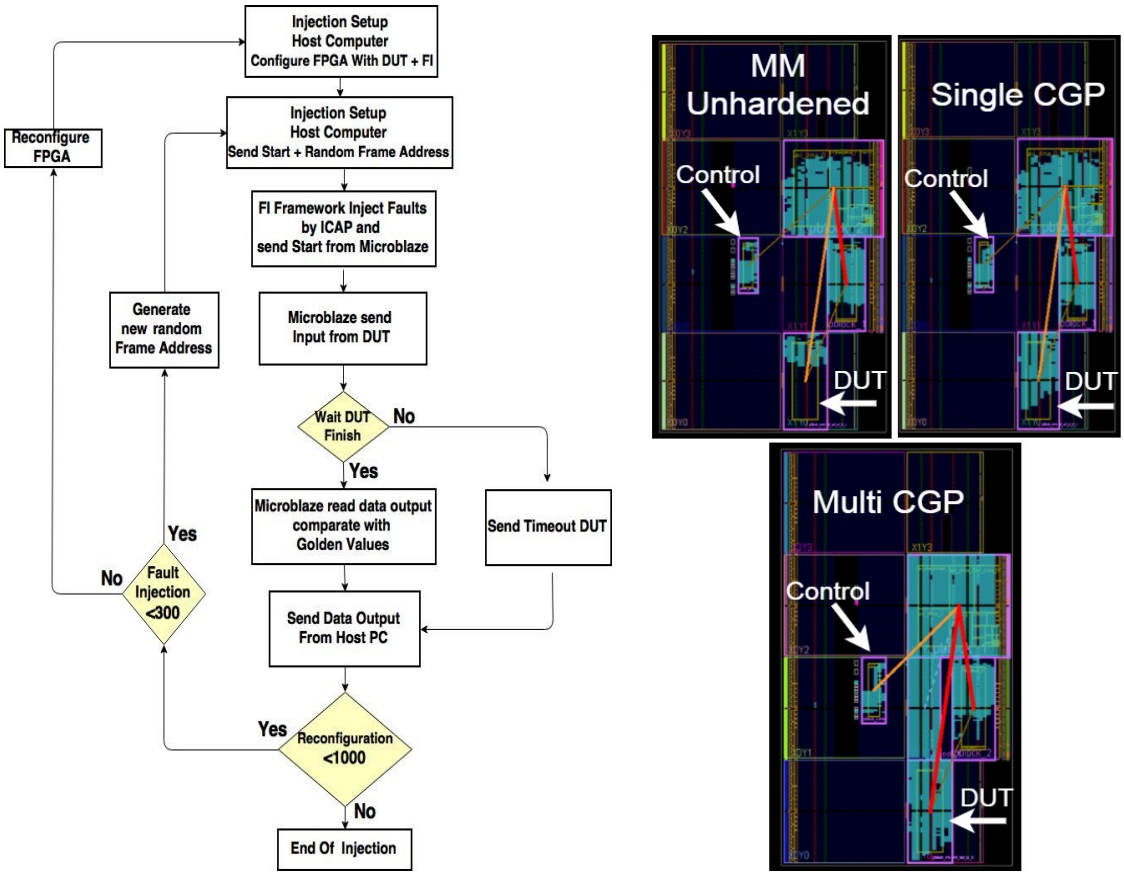


Figura 50 Definição para metodologia da injeção de falhas acumuladas (a) e o planejamento da área ocupada do FPGA (b) (AUTOR)

No modo de injeção de falhas acumuladas são injetadas falhas aleatórias sem remoção da falha dentro da área do projeto até gerar um erro. A reconfiguração de um novo fluxo de dados (*Bitstream*) somente é realizada quando acontecer um erro na saída da aplicação ou atingir o limite de falhas definido no projeto. Neste trabalho este limite foi definido em 300 falhas acumuladas, ilustrado na figura 50 (a). O objetivo deste modo de injeção de falhas acumuladas é observar o limite de falhas injetadas suportado pelo projeto até gerar um erro, assim determinando a confiabilidade do projeto.

A figura 51 demonstra o sorteio aleatório do quadro de configuração da memória do FPGA (*Configuration Frame*) para o modo de injeção de falhas acumuladas utilizado neste

trabalho. Além do *quadro de configuração da memória* , é sorteado a posição do bit que varia de 0 a 3232 no dispositivo alvo Xilinx Artix 7.

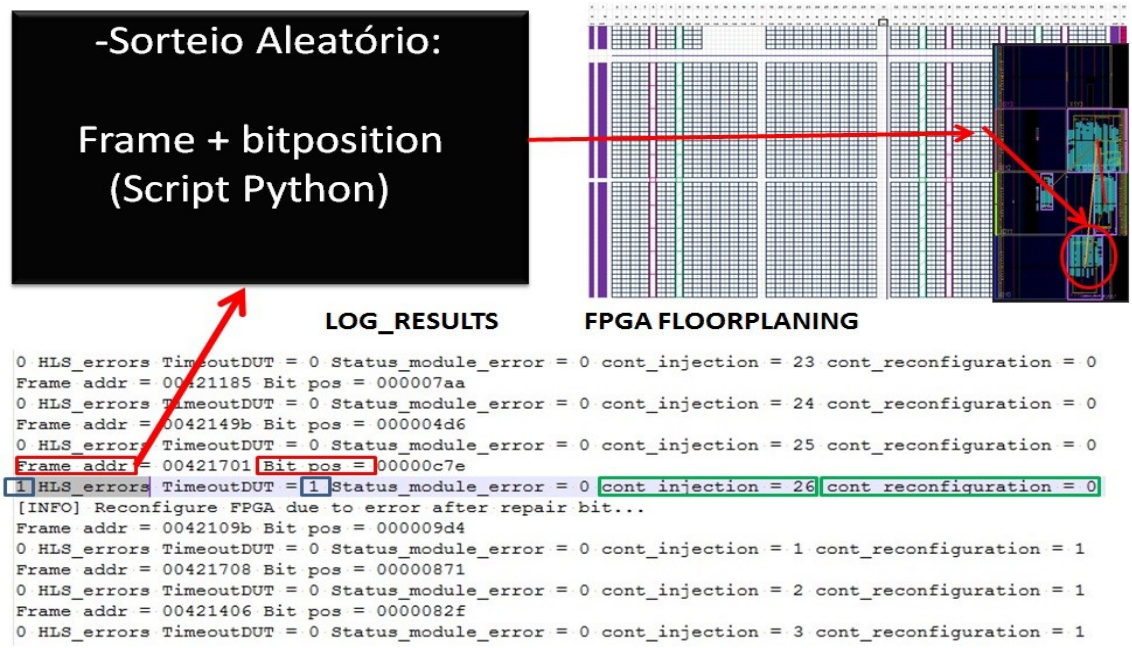


Figura 51 Posição (*Frame addr*) onde a falha é injetada na área disponível do FPGA e exemplo do arquivo LOG gerado com resultados da Injeção de falhas acumulada (TMR) (AUTOR)

As variáveis '*HLS_errors*', '*TimeoutDUT*', '*status_module_error*', seguem o mesmo padrão de funcionamento explicado na injeção de falhas simples neste mesmo capítulo. Através do arquivo de 'log' de resultados é possível analisar a quantidade de falhas necessárias para gerar um erro na saída do projeto. Segundo a variável '*cont_injection*', neste exemplo foram necessárias 26 falhas para fazer o projeto falhar. As duas variáveis '*cont_injection*' e '*cont_reconfiguration'* podem ser calibradas conforme o nível de mitigação escolhido pelo projetista. Neste trabalho foi escolhida a seguinte opção: 1000 reconfigurações e 300 falhas injetadas. Se nenhum erro for gerado na saída do projeto até atingir o limite de falhas injetadas o FPGA é reconfigurado novamente com um novo fluxo de dados (*Bitstream*).

Os resultados para os dois modos de injeção de falhas utilizados neste trabalho serão apresentados no capítulo 6.

6.1 ANALISANDO MÉTRICAS PARA A ESTIMATIVA DE SUSCEPTIBILIDADE A FALHAS SIMPLES

Alguns experimentos de injeção de falhas simples por emulação em projetos gerados pela ferramenta Xilinx High Level Synthesis para FPGA Xilinx Artix 7 tem sido realizados como em (TONFAT, 2016), com objetivo de verificar a sucetibilidade a SEU, analisando a informação dos bits essenciais fornecidos pela Xilinx com os bits críticos obtidos pela injeção de falhas em projetos sem redundância. As otimizações testadas nos projetos foram *pipeline* em alguns laços de repetição e o uso ou não de DSPs na lógica da implementação. Este trabalho baseou-se na arquitetura ilustrada na figura 27, com interface para armazenamento de dados através de BRAMs e controle por uma máquina de estados finitos (FSM).

O método de caracterização proposto usado tem relação para ajudar projetistas a selecionar a arquitetura mais efeciente quanto a suscetibilidade de perturbações como SEU, e eficiência ao desempenho.

A seção de choque estática da área do circuito (*Static Cross Section Circuit Area*) é a mesma definida pela expressão 3, no capítulo 2. A seção de choque dinâmica (σ*dynamic*) é calculada antes da injeção de falhas levando em consideração o valor do parâmetro *essencial bits*, definidos pela Xilinx como bits que podem se tornar críticos se atingidos por alguma falha transiente.

$$\sigma_{dynamic} = Staticcross\,\mathrm{sec}\,tioncircuitarea * \frac{(essentialbits)}{configurationbits} \quad (5)$$

Após a injeção de falhas a seção de choque dinâmica (σ*dynamic*) leva em consideração os bits críticos (*Critical Bits*), definido na expressão 6. Os resultados para os dois casos são apresentados na tabela 4.

$$\sigma_{dynamic} = Staticcross\,\mathrm{sec}\,tioncircuitarea * \frac{(criticalbits)}{configurationbits} \quad (6)$$

Os seguintes parâmetros incluem as métricas utilizadas com base nas medições de radiação e estimativas no laboratório. A Taxa de erros suave, do inglês *Soft Error Rate* (SER) é expresso em unidades *Failure in Time* (FIT) e é definido como a quantidade esperada de erros por 10^9 horas de operação do dispositivo em um determinado ambiente de radiação. Pode ser obtido multiplicando a seção de choque dinâmica (σ*dynamic*) com o fluxo de partículas no ambiente de radiação.

$$SER = \sigma_{dynamic} * \Phi_{particle} * 10^9 \quad (7)$$

O parâmetro Carga Média de Trabalho entre falhas, do inglês *Mean Workload Between* Failure (MWBF) foi proposta em (RECH, 2014) e avalia a quantidade de dados (carga de trabalho) processados corretamente pelo projeto antes do aparecimento de um erro de saída. Ele é definido como:

$$MWBF = \frac{W}{\sigma_{dynamic} * flux * t_{exec}} \quad (8)$$

O parâmetro *W* é a carga de trabalho processada em uma execução, σ*dynamic* é a seção de choque dinâmica, *flux* é a fluência de partículas por unidade de tempo e *Texec* é o tempo de execução do projeto.

A carga de trabalho (*Workload*) para este estudo de caso foram duas matrizes 6x6 com dados de entrada de 8 bits, resultando em 576 (36x8x2) bits de carga de trabalho por execução. As versões com *pipeline* utilizaram mais recursos e apresentaram um tempo de execução melhor, uma vez que a ferramenta HLS desenrola e canaliza os laços de repetição, gerando mais instâncias de *hardware* e incrementando o paralelismo. A tabela 3 apresenta os dados de área e desempenho utilizados.

Tabela 3. Dados de área e performance (TONFAT, 2016)

	Area				**Performance**	
Matrix Mult. Version	**# LUTs**	**# FFs**	**# DSP**	**#Configuration bits**	**Exec. time (clk cycles)**	**Workload (bits)**
No opt., No DSP48E	155 0.24%	70 0.06%	0 0%	694,224 2.99%	733	576
No opt., with DSP48E	50 0.08%	38 0.03%	1 0.42%	784,216 3.38%	733	576
Pipeline opt., No DSP48E	18,910 29.83%	5,735 4.52%	0 0%	6,479,424 27.91%	36	576
Pipeline opt., with DSP48E	1,117 1.76%	1,327 1.05%	198 82.50%	7,559,328 32.56%	36	576

Na tabela 4 os bits críticos e os bits essenciais foram comparados. Através dos resultados de injeção de falhas, pode-se observar que há uma pequena fração (menos de 18%) dos bits essenciais apresentaram-se como bits críticos.

Tabela 4. Comparação entre bits essenciais e bits críticos (TONFAT, 2016)

Matrix Mult. Version	**Essential bits from Xilinx Tool (% of config. bits in design area)**	**Critical bits from FI (% of Essential bits)**
No opt., No DSP48E	34,862 (5.02 %)	2,434 (6.98 %)
No opt., with DSP48E	18,208 (2.32 %)	1,740 (9.56 %)
Pipeline opt., No DSP48E	3,494,735 (53.94 %)	170,929 (4.89 %)
Pipeline opt., with DSP48E	834,967 (11.05 %)	66,758 (8.00 %)

A versão implementada com a otimização *pipeline* e sem utilizar recursos DSPs apresentou mais bits essenciais e críticos, uma vez que utiliza mais recursos que os outros projetos.

A Seção de Choque Dinâmica (σ, *Dinamic Cross Section*), Taxa de Erros Suaves (SER) e Carga Média de Trabalho entre falhas (MWBF) foram estimadas com base nos bits essenciais e bits críticos.

Tabela 5. Seção de choque dinâmica, SER, MWBF estimados com base nos bits essenciais e críticos (TONFAT, 2016)

	From Xilinx report	**From essential bits**			**From critical bits**		
Matrix Mult. Version	**σ_{static} (cm^2)**	**$\sigma_{dynamic}$ (cm^2)**	**SER @ NYC (FIT)**	**MWBF (bits)**	**$\sigma_{dynamic}$ (cm^2)**	**SER @ NYC (FIT)**	**MWBF (bits)**
No opt., No DSP48E	4.6E-08	2.4E-10	3.17	8.9E+19	1.701E-11	0.22	1.3E+21
No opt. with DSP48E	4.9E-08	1.3E-10	1.65	1.7E+20	1.216E-11	0.16	1.8E+21
Pipeline opt., No DSP48E	4.6E-08	2.4E-08	317.57	1.8E+19	1.195E-09	15.53	3.7E+20
Pipeline opt. with DSP48E	5.4E-08	5.8E-09	75.87	7.6E+19	4.666E-10	6.07	9.5E+20

O uso de bits essenciais para estimar a suscetibilidade de um projeto em termos de seção de choque dinâmica ou MWBF fornece uma boa aproximação comparando as métricas calculadas utilizando os bits críticos da injeção de falhas. O trabalho correlato demonstrou que utilizando DSPs a seção de choque dinâmica diminuiu de 0,4x para 0,7x de acordo com a arquitetura e utilizando a otimização *pipeline* a seção de choque dinâmica aumentou de 40x a 70x de acordo com a utilização de DSPs. O MWBF apresenta uma análise de suscetibilidade considerando os bits da área do projeto e o desempenho (tempo de execução e carga de trabalho) da aplicação. O melhor projeto em termos de MWBF é o com maior valor na versão

sem otimização e com DSP, pois apresentou o menor número de bits essenciais e o menor número de bits críticos.

Embora através do número de bits essenciais seja possível determinar o comportamento ou tendência de um projeto a erros transientes (*Soft Errors*), a injeção de falhas é necessária para classificar os erros. Os erros são classificados em corrupção silenciosa de dados (SDC, *Silent Data Corruption*) e tempo limite (*Timeout*). O SDC pode afetar uma simples palavra dos resultados ou um conjunto maior deles.

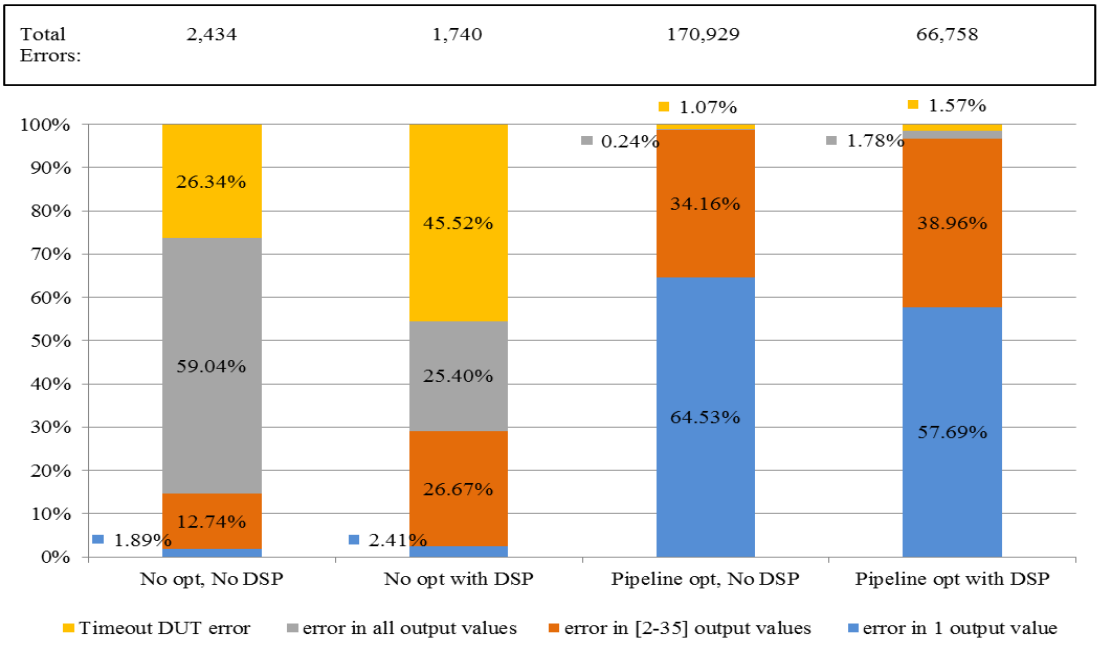


Figura 52 Classificação de erros para os quatro projetos analisados (TONFAT, 2016)

A figura 52 ilustra a classificação dos erros obtidos por injeção de falhas por emulação em projetos sem redundância. Os erros de SDC são classificados em três categorias: Erros em todos os valores da matriz, erros em 2 a 35 valores e erros apenas em 1 valor. Pode-se observar que as versões não otimizadas apresentam um valor de erro diferente das versões com a otimização *pipeline*. As versões não otimizadas apresentam uma maior porcentagem de erros relacionados ao fluxo de controle do algoritmo, porque a máquina de estados (FSM) de controle ocupa uma área maior da arquitetura e tende a ser mais sensível. O Resultado nestas arquiteturas sem otimizações é representado por um número elevado de erros de tempo limite (*Timeout*) e erros em todos os valores de resultados da matriz. As versões com otimização *pipeline* representam um maior número de erros relacionados ao fluxo de dados, elevando o

valor de erros em dados únicos ou poucos valores errados na matriz de resultados. Os resultados são representados principalmente pela arquitetura utilizada no projeto.

7 RESULTADOS

Neste capítulo serão apresentados os resultados de área, desempenho e bits críticos para o modo de Injeção de Falhas Simples (SEU). Para o modo de Injeção de Falhas Acumuladas (SEU acumulados) serão apresentados os resultados de área, desempenho e confiabilidade. A área de um projeto é representada em termos de recursos utilizados, tais como: LUTS, registradores (*flip-flops*), blocos de memória configuráveis (BRAM) e blocos processadores de sinais digitais (DSPs, *Digital Signal Processor*). A área também pode ser representada em número de quadros de configuração (*Configuration Frames*). O desempenho é indicado em termos de tempo de execução medido em ciclos de relógio (*Clock* Cycles) e tempo quando um relógio de 100 MHz é utilizado. A carga de trabalho é representada em bits, sendo duas matrizes de entrada 6x6 (a,b) com 36 elementos e cada elemento com 8 bits de tamanho (2x8x36=576 bits de carga de trabalho) para os projetos sem redundância. Para os projetos com Redundância Modular Tripla (TMR) são 6 matrizes de entrada (a1,b1,a2,b2,a3,b3) com tamanhos de 8 bits (2x8x36 = 576x3 = 1728 bits de carga de trabalho).

7.1 INJEÇÃO DE FALHAS SIMPLES (SEU) ÁREA E DESEMPENHO

A tabela 6 exibe os resultados da área de implementação e análise de desempenho para os projetos propostos no modo de Injeção de Falhas Simples (SEU). O número de quadros de configuração (*Configuration Frames*) utilizado para todos os projetos foi de 316, pois ocuparam a mesma área de implementação, com base na área do maior projeto.

Tabela 6. Detalhes da área utilizada por cada projeto em termos de recursos utilizados do FPGA e bits de configuração. Resultados de desempenho em termos de tempo de execução e carga de trabalho.

	Area			**Desempenho**		
Versão do projeto	**# 6-input LUTs**	**#User FFs**	**BRAM**	**Exec. time (clk cycles)**	**Exec. time (ns) @ 100 MHz**	**Work load (bits)**
Unhardened	303	159	1	849	8,490	576
Unhardened Pipeline	289	178	1	336	3,360	576
Unhardened Unroll	486	285	1	489	4,890	576
CGP Single Stream	888	420	4	1154	11,540	1728
CGP Single Pipeline	902	477	4	641	6,410	1728
CGP Multiple Stream	1084	493	4	1228	12,280	1728
CGP Multiple Stream Pipeline	1098	550	4	715	7,150	1728

CGP Multiple Stream Unroll	2088	967	4	868	8,680	1728
CGS Single Stream	1066	485	4	2618	26,180	1728
CGS Single Stream Pipeline	930	466	4	1076	10,760	1728
CGS Multiple Stream	1066	485	4	2692	26,920	1728
CGS Multiple Stream Pipeline	1132	539	4	1150	11,500	1728
CGS Multiple Stream Unroll	2284	1061	4	1720	17,200	1728
FGP Single Stream	1097	425	3	3488	33,490	1728
FGP Single Stream Pipeline	1071	469	3	1088	10,880	1728
FGP Multiple Stream	1490	542	3	3562	35,620	1728
FGP Multiple Stream Pipeline	1260	542	3	1162	11,620	1728
FGP Multiple Stream Unroll	1722	744	3	1618	16,180	1728

Os projetos que utilizaram otimizações como *pipeline* e *unroll* apresentam uma área maior do que os projetos sem otimização, por ocupar mais recursos para a implementação. Porém, o tempo de execução é melhor nos projetos com otimizações, uma vez que o HLS desenrola laços de repetição, faz a leitura paralela de valores, gerando mais instâncias de *hardware* incrementando o paralelismo.

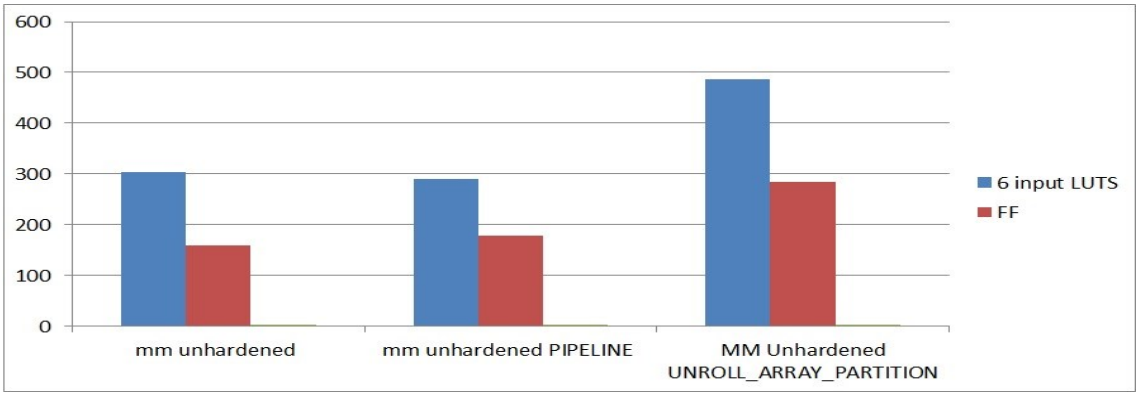


Figura 53 Dados de área ocupada em termos de recursos para os projetos sem redundância (*Unhardened*)

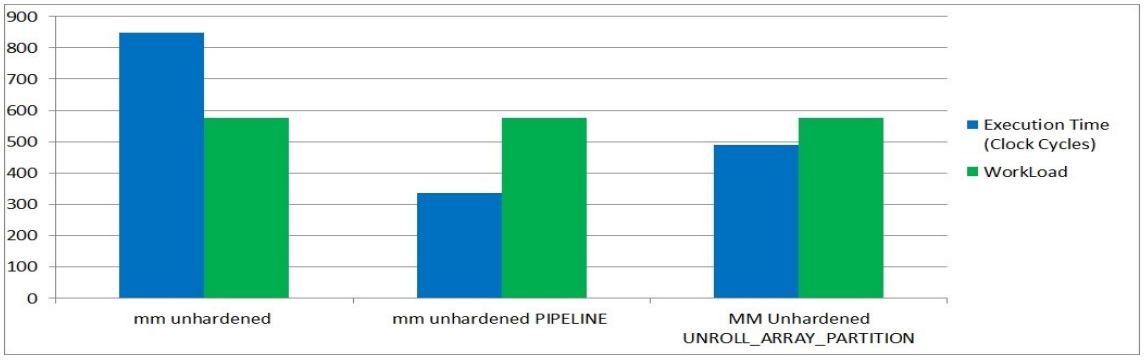


Figura 54 Dados de desempenho representados por tempo de execução e carga de trabalho para os projetos sem redundância (*Unhardened*)

As figuras 53 e 54 exibem os dados de área e desempenho dos projetos sem redundância. Pode-se observar um acréscimo na área conforme a otimização se torna mais agressiva, enquanto o tempo de execução vai diminuindo. A versão com melhor desempenho em termos de tempo de execução foi a otimização *pipeline* efetuando paralelização na execução e desenrolando laços de repetição. A versão com otimização *unroll*, responsável por desenrolar os laços de repetição e paralelizar os dados de entrada para a execução, ficou intermediária entre a a versão sem otimização e com otimização *pipeline*, representando um maior emprego de recursos por não usar o mesmo *hardware* para todas as operações. Os projetos que utilizaram Redundância Modular Tripla (TMR) tiveram um acréscimo em torno de 2x ou 3x na área e no tempo de execução. As diretivas de otimizações utilizadas nos projetos com redundância foram: *pipeline* e *unroll* de forma similar aos projetos sem redundância, aplicadas apenas no laço de repetição (*loop*) mais interno da função de multiplicação de matrizes.

A arquitetura *Single Stream* é composta por um único canal para a transferência dos dados de entrada e saída. Existe um canal para os dados de entrada chamado *Input Stream* e um canal para os resultados chamado *Output Stream*. O barramento de controle é chamado *Control_Bus* e possui uma máquina de estados finitos (FSM) para controle e transferência de dados entre o microblaze e o IP HLS. Os barramentos e a máquina de estados finitos também apresentam suscetibilidade a SEU e podem ocasionar um erro na saída ou corromper parte dos dados.

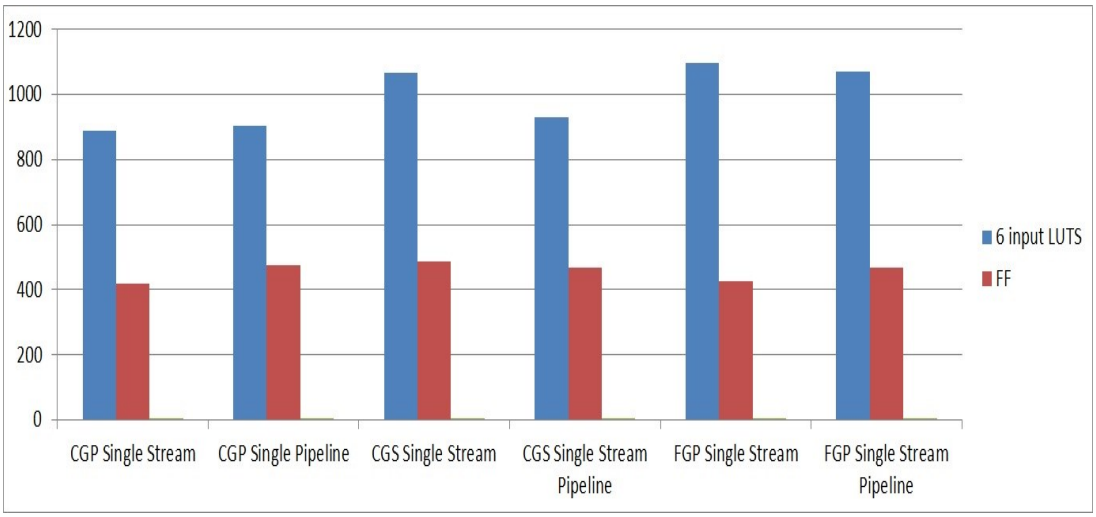


Figura 55 Dados de área ocupada em termos de recursos para projetos com redundância TMR utilizando a arquitetura *Single Stream*

A figura 55 ilustra os dados de área para os projetos com redundância TMR e arquitetura *Single Stream*. Pode-se observar pouca variação na utilização de recursos, a implementação CGS utilizou um número de recursos intermediário entre os projetos CGP e FGP. O projeto com maior área utilizada foi FGP *Single Stream*, pois necessita de mais recursos para implementação de elementos de memória em razão de usar votadores majoritários dentro da função de multiplicação de matrizes. Os projetos apresentaram valores próximos entre 888 a 1097 LUTs e 420 a 485 FF. Os recursos utilizados referem-se aos barramentos de transferência de dados, parte lógica do circuito (aplicação), implementação de memória para alguns casos e barramentos de controle.

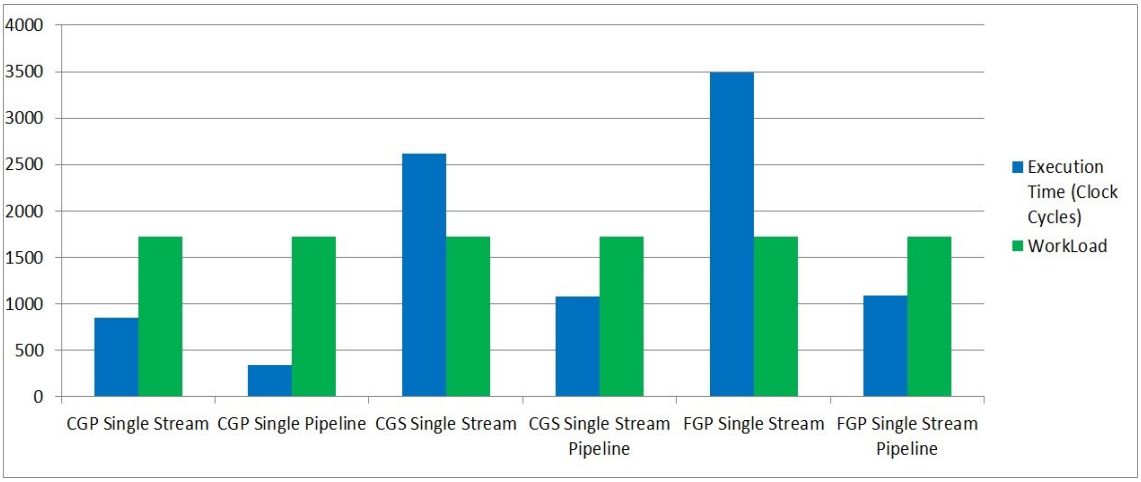


Figura 56 Dados de desempenho representados por tempo de execução e carga de trabalho para os projetos com arquitetura *Single Stream*

Os dados de desempenho para os projetos com redundância TMR e arquitetura *Single Stream* são exibidos na figura 56. Os projetos que utilizaram otimizações apresentaram um melhor desempenho para o tempo de execução. O projeto com maior tempo de execução foi o FGP, pois utiliza uma implementação diferente para a função de multiplicação de matrizes. Houve maior variação em termos de tempo de execução do que área utilizada entre os projetos. Os projetos apresentaram valores entre 336 a 3488 ciclos de relógio (*Clock Cycles*).

A arquitetura *Multiple Stream* é composta por canais individuais para a transferência dos dados. Existe um canal individual para cada par de matrizes de entrada chamados: *Input Stream1, Input Stream2, Input Stream3.* Existe um canal individual para os resultados chamados: *Output Stream1*, *Output Stream2, Output Stream3*. Também existe o barramento de controle chamado *Control_Bus*, o qual possui uma máquina de estados finitos (FSM) para controle e transferência de dados entre o microblaze e o IP HLS. Estes barramentos e a

máquina de estados finitos também apresentam suscetibilidade a SEU. A figura 57 exibe os dados de área para os projetos com redundância TMR e arquitetura *Multiple Stream*.

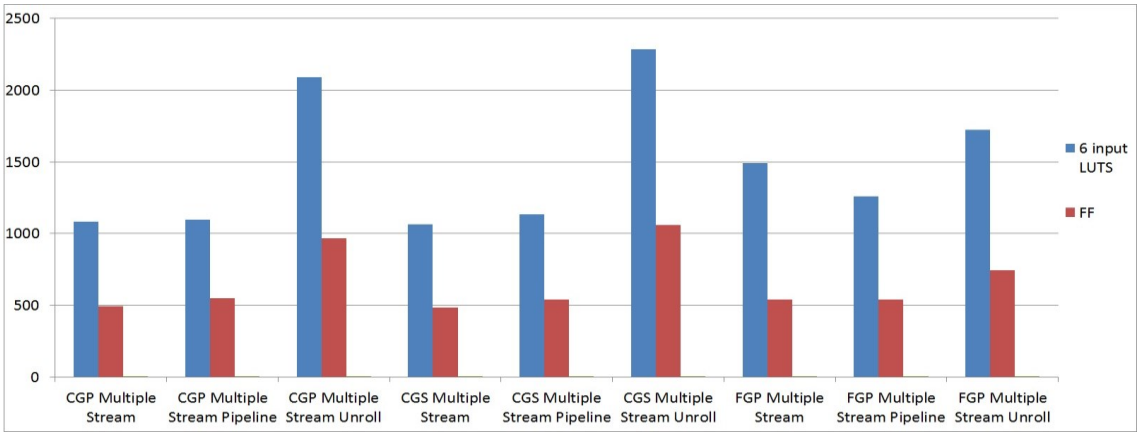


Figura 57 Dados de área ocupada em termos de recursos para os projetos com redundância TMR utilizando a arquitetura *Multiple Stream*

É possível observar predomínio na utilização de recursos nos projetos que usaram otimizações, exibidos na figura 57. O projeto com maior área foi CGS *Multiple Stream Unroll*. Os projetos apresentaram valores entre 1066 a 2284 LUTs e 485 a 1061 FF. Os recursos utilizados referem-se aos barramentos de transferência de dados, parte lógica do circuito (aplicação), implementação de memória em alguns casos e barramentos de controle. Neste caso, os barramentos de controle são independentes e podem representar uma pequena vantagem em comparação a arquitetura *Single Stream*, pois se algum barramento for atingido por uma falha transiente, somente partes dos dados poderão ser comprometidos, ao invés de todos os dados serem afetados.

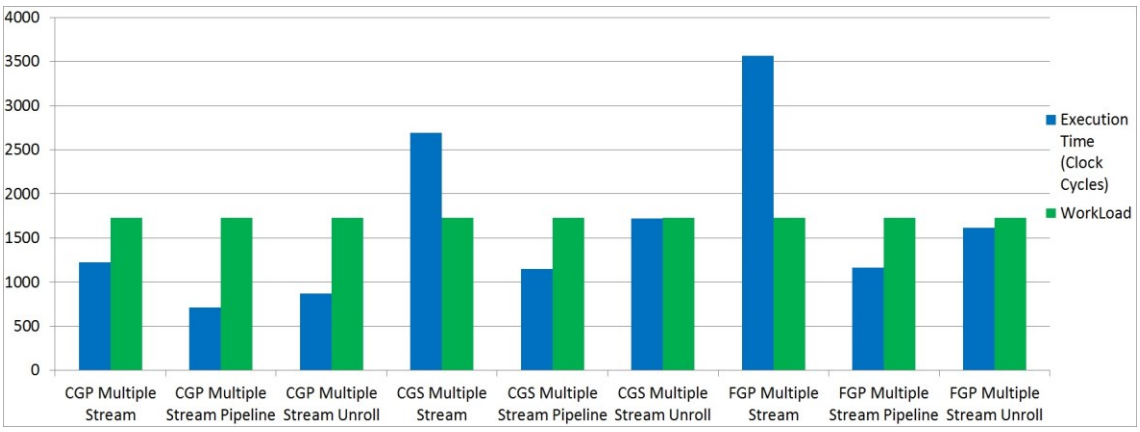


Figura 58 Dados de desempenho representados por tempo de execução e carga de trabalho para os projetos com arquitetura *Multiple Stream*

Os dados de desempenho para os projetos com redundância TMR e arquitetura *Multiple Stream* são exibidos na figura 58. Os projetos que utilizaram otimizações apresentaram um melhor desempenho para o tempo de execução. É possível observar uma maior variação em termos de tempo de execução e área utilizada entre os projetos, se comparados com a arquitetura *Single Stream*. O projeto com maior tempo de execução foi o FGP *Multiple Stream* sem otimização, pelo motivo de implementar votadores dentro da função de multiplicação de matrizes.

7.1.1 INJEÇÃO DE FALHAS SIMPLES (SEU) E BITS CRÍTICOS

A Injeção de Falhas Simples tem como objetivo injetar falhas em toda a área definida para o projeto HLS, definida em 316 quadros de memória de configuração (*Configuration Frames*) com total de 1,021,312 bits. Foram realizadas campanhas de injeção de falhas em projetos com e sem redundância no objetivo de analisar a probabilidade a erros. A efetividade dos projetos com TMR ao ser afetado por uma falha é definida na relação (*criticalbits / configurationbits*), onde é verificado qual a perspectiva de chances deste projeto gerar um erro se atingido por uma falha, ou seja, é demonstrado a porcentagem de bits de configuração que se tornaram bits críticos ao ser atingido por uma falha transiente. As informações dos bits de configuração foram comparadas com os bits críticos fornecidos pela injeção de falhas. Os resultados demonstram que a técnica TMR aplicada em HLS apresenta porcentagem semelhante de valor de bits críticos em relação aos projetos sem redundância. A explicação esta no fato deste trabalho estar utilizando a mesma área de injeção de falhas para todos os projetos, com base na área do maior projeto. Os projetos sem redundância TMR possuem um tamanho três vezes menor do que os projetos com redundância, consequentemente a maioria dos bits da área considerada não foram utilizados.

Tabela 7 Bits críticos em relação aos bits de configuração da área utilizada por todos os projetos

Matrix Mult. Version	Critical bits (SDC - data output)	Critical bits (Hangs)	Critical bits from FI SDC + Hangs (% of Configuration bits)	Dynamic Cross Section Circuit Area
MM Unhardened	3617	149	3766 (0.369%)	2.6324E-11
MM Unhardened Pipeline	4513	234	4747 (0.465%)	3.3182E-11
MM Unhardened Unroll	6036	284	6320 (0.619%)	4.4177E-11
CGP Single Stream	6485	251	6736 (0.660%)	4.7085E-11
CGP Single Stream Pipeline	6055	212	6267 (0.614%)	4.3806E-11

CGP Multiple Stream	6192	189	6381 (0.625%)	4.4603E-11
CGP_Multiple Stream Pipeline	6012	289	6301 (0.617%)	4.4044E-11
CGP Mutiple Stream Unroll	6971	195	7166 (0.702%)	5.0090E-11
CGS Single Stream	6497	185	6682 (0.654%)	4.6707E-11
CGS Single Stream Pipeline	6155	225	6380 (0.625%)	4.4596E-11
CGS Multiple Stream	6135	212	6347 (0.621%)	4.4366E-11
CGS Multiple Stream Pipeline	5766	299	6065 (0.594%)	4.2394E-11
CGS Multiple Stream Unroll	7608	332	7940 (0.777%)	5.5501E-11
FGP Single Stream	6491	220	6711 (0.657%)	4.6910E-11
FGP Single Stream Pipeline	6188	188	6376 (0.624%)	4.4568E-11
FGP Multiple Stream	6472	244	6716 (0.658%)	4.6945E-11
FGP Multiple Stream Pipeline	5975	271	6246 (0.612%)	4.3660E-11
FGP Multiple Stream Unroll	5339	219	5558 (0.544%)	3.8850E-11

Com base nos resultados da relação *criticalbits/configurationbits* utilizando a mesma área de injeção de falhas, procurou-se explorar uma maneira de apresentar os resultados apenas em relação aos bits realmente utilizados por cada projeto chamados de bits essenciais (*Essential Bits*) fornecidos pela Xilinx, ou seja, não são considerados os bits não utilizados na área do DUT (*Design Under Test*). A relação utilizada definida por (*criticalbits/essentialbits*) comprova a efetividade do uso de redundância TMR em ferramenta de Síntese de Alto Nível, pois a porcentagem de bits críticos nos projetos sem redundância é maior do que nos projetos com redundância TMR demonstrado na tabela 8. É possível verificar esta relação também nos resultados da *Dynamic Cross Section Circuit Area*, definido como a probabilidade do projeto gerar um erro nos resultados se atingido por uma falha transiente.

Tabela 8 Bits críticos em relação aos bits essenciais fornecidos pela ferramenta Xilinx Vivado Design Suite Tools

Matrix Mult. Version	**Essential bits from Xilinx Tool (% of Config. Bits in design area)**	**Critical bits (SDC - data output)**	**Critical bits (Hangs)**	**Critical bits from FI SDC + Hangs (% of Essential bits)**	**Dynamic Cross Section Circuit Area**
MM Unhardened	66810 (6.542%)	3617	149	3766 (5.637%)	4.0242E-10
MM Unhardened Pipeline	65565 (6.420%)	4513	234	4747 (7.240%)	5.1687E-10
MM Unhardened Unroll	103584 (10.142%)	6036	284	6320 (6.101%)	4.3557E-10
CGP Single Stream	161503 (15.813%)	6485	251	6736 (4.171%)	2.9775E-10
CGP Single Stream Pipeline	166068 (16.260%)	6055	212	6267 (3.774%)	2.6941E-10
CGP Multiple Stream	212466 (20.803%)	6192	189	6381 (3.003%)	2.1441E-10

CGP_Multiple Stream Pipeline	216252 (21.174%)	6012	289	6301 (2.914%)	2.0801E-10
CGP Mutiple Stream Unroll	411678 (40.309%)	6971	195	7166 (1.741%)	1.2427E-10
CGS Single Stream	167770 (16.427%)	6497	185	6682 (3.983%)	2.8433E-10
CGS Single Stream Pipeline	181571 (17.778%)	6155	225	6380 (3.514%)	2.5085E-10
CGS Multiple Stream	211560 (20.715%)	6135	212	6347 (3.000%)	2.1418E-10
CGS Multiple Stream Pipeline	219202 (21.463%)	5766	299	6065 (2.767%)	1.9752E-10
CGS Multiple Stream Unroll	449849 (44.046%)	7608	332	7940 (1.765%)	1.2601E-10
FGP Single Stream	196129 (19.204%)	6491	220	6711 (3.422%)	2.4428E-10
FGP Single Stream Pipeline	209844 (20.547%)	6188	188	6376 (3.038%)	2.1691E-10
FGP Multiple Stream	277268 (27.148%)	6472	244	6716 (2.422%)	1.7292E-10
FGP Multiple Stream Pipeline	247439 (24.228%)	5975	271	6246 (2.524%)	1.8021E-10
FGP Multiple Stream Unroll	326308 (31.950%)	5339	219	5558 (1.703%)	1.2160E-10

Os valores dos bits essenciais em relação aos bits de configuração dos projetos variaram de 6.42% a 44.05%. Os valores de bits críticos de maneira geral foram próximos ou tiveram pequena variação pelo fato da matriz de entrada ser considerada pequena (6x6), trabalhos correlatos (TAMBARA, 2017) que utilizaram matrizes maiores (32x32), por exemplo, apresentaram maior variação nos resultados de bits críticos. A arquitetura *Multiple Stream* indiferente do projeto ou otimização escolhida, sempre apresentou um valor maior de bits essenciais, devido aos barramentos independentes utilizados para a transferência de dados, o que exige um maior número de recursos para a implementação. Os bits essenciais são definidos pela Xilinx como bits que se forem atingidos por um SEU podem se tornar bits críticos. Através da relação entre bits críticos e bits essenciais temos a probabilidade do projeto ser mais ou menos suscetível a falhas transientes e gerar um erro na saída. A definição do cálculo para porcentagem dos bits críticos em relação aos bits de configuração ou bits essenciais é representada da seguinte forma:

$$P = (\text{total bits críticos} / \#\text{bits de configuração}\ \ \#\text{bits essenciais}) \qquad (9)$$

Onde ‘P’ é a porcentagem de bits críticos em relação aos bits de configuração ou bits essenciais do projeto, ‘total bits críticos’ é o total de bits críticos do projeto obtido pela injeção de falhas, ‘#bits de configuração’ são os bits de configuração da área do projeto e ‘#bits essenciais’ são o valor gerado pela ferramenta Xilinx Vivado Design Suite Tools.

Concluindo que os projetos com a utilização de redundância TMR são mais efetivos que os projetos sem redundância, porque apresentam valores de porcentagem de bits críticos menores em relação à porcentagem dos projetos sem utilização de redundância. O projeto com redundância TMR de melhor resultado em termos de porcentagem de bits críticos em relação aos bits essenciais foi FGP *Multiple Stream Unroll*, com 1,7033%. Os projetos sem redundância apresentaram valores entre 5,6369% a 7,2401%. E os projetos com redundância TMR apresentaram valores entre 1,7033% a 4,1708%. A Seção de Choque Estática do Circuito (*Static Cross Section Circuit Area*) é calculada através da expressão (3) no capítulo 2, subseção 2.2, representa o valor da menor área sensível para o circuito, através da multiplicação da Seção de Choque do Dispositivo Alvo (*Static Cross Section Device*) pelo total de bits de configuração da área do circuito. Neste trabalho foi utilizada a mesma área de injeção de falhas para todos os projetos com base no maior projeto, em consequência desta configuração o valor da seção de choque estática é 7.1390×10^{-9} pata todos os projetos. A Seção de Choque Dinâmica do Circuito (*Dynamic Cross Section Circuit Area*), definida como a probabilidade de uma partícula gerar um erro no projeto, pode ser considerada usando a informação dos bits essenciais ou bits críticos indicada pela seguinte expressão:

$$\sigma_{dynamic-est} = \sigma_{static} * \frac{\#bits_{essential} \vee \#bits_{critical}}{bits_{injected}} (10)$$

Pode-se concluir que as melhores descrições utilizando redundância foram os projetos de múltiplos canais com a otimização *unroll*, pois apresentaram os menores valores de bits críticos em relação aos bits essenciais. Os projetos que fizeram uso da otimização *unroll* expressaram maior valor no uso de recursos como LUTS e *flip-flops*, também apresentaram valor intermediário para o tempo de execução entra a versão sem otimização e otimização *pipeline*, para os projetos TMR CGP, TMR CGS e TMR FGP de múltiplos canais. O que indica maior utilização de recursos e menor dependência de dados se comparada à otimização *pipeline*, que no caso de uma partícula afetar algum elemento de memória ou aritmética compartilhado desencadeará um problema para o resto da execução. Finalizando, temos na otimização *unroll* um maior gasto em termos de área, porém um tempo de execução intermediário e maior garantia de não ser afetado por uma partícula comparado às versões sem otimização e otimização *pipeline*. O que demostra ao projetista de aceleradores de *hardware* gerados por ferramenta de HLS a necessidade de investigação na escolha da otimização adequada ao seu projeto.

7.2 MODO DE INJEÇÃO DE FALHAS ACUMULADAS (SEU ACUMULADOS) E O CÁLCULO DE CONFIABILIDADE

A Injeção de Falhas Acumuladas apresentada no capítulo 6, tem como objetivo analisar o acúmulo de *bit-flips* na memória de configuração do FPGA baseado em SRAM. Foram testados projetos sem e com redundância TMR com arquiteturas *Single Stream* e *Multiple Stream* de entrada e saída de dados em um bloco de 388 quadros de memória de configuração, com o total de bits na área de injeção de 1,254,016 bits. O modo de injeção de falhas utilizado foi calibrado da seguinte forma: 300 falhas injetadas e 1000 reconfigurações com objetivo de analisar a confiabilidade dos projetos, ou seja, quantas falhas seriam necessárias para fazer o projeto gerar um erro na saída. Ao atingir 300 falhas injetadas e o projeto não gerar erro na saída, o fluxo de bits (*Bitstream*) é carregado novamente. Como o número de falhas injetadas é pequeno se comparado ao número total de bits de configuração na área de injeção de falhas, a probabilidade de o mesmo bit ser atingido mais de uma vez é pequena, permitindo que a taxa de erros seja estimada como a média de erros sobre o total de falhas injetadas. Trabalhos futuros com um maior número de falhas ou sem limitar o número de falhas até o projeto falhar serão testados, de modo que se obtenha melhores resultados.

Tabela 9 Recursos utilizados e análise de desempenho para os casos de estudo no modo de injeção de falhas acumuladas

	Area					Desempenho		
Versão do projeto	**# 6-input LUTs**	**#User FFs**	**DSP 48E**	**#Config. frames in design area**	**#Config. Bits in design area**	**Exec. time (clk cycles)**	**Exec. time (ns) @ 100 MHz**	**Workload (bits)**
CGPTMR SingleStream	1216	692	3	388	1,254,016	1,154	11,540	1728
CGPTMR MultiStream	1791	1122	3	388	1,254,016	1,228	12,280	1728
Unhardened	497	340	1	388	1,254,016	849	8,490	576

Em termos de desempenho cada projeto TMR apresenta um tempo maior que a versão do projeto sem redundância. O tempo de execução é calculado pelo número de ciclos necessários para ler as matrizes de entrada, executar a aplicação, votar os resultados das multiplicações e escrever os resultados das matrizes de saída. A sobrecarga de desempenho entre os projetos com TMR é considerada pequena, mas existe pelo motivo das entradas e saídas serem triplicadas no tempo.

Embora cada projeto utilize uma quantidade diferente de recursos, conforme detalhados na tabela 9, as campanhas de injeções de falhas consideram a mesma área de injeção de falhas para todos os projetos. Estabelecendo assim uma condição semelhante para todos os projetos, emulando uma mesma fluência de partículas na sua superfície.

Três implementações de projetos para multiplicação de matrizes foram realizadas: versão sem redundância (*Unhardened*), versão com TMR de Grão Grosso Paralela com canal único de transferência (TMR CGP, *Coarse Grain Parallel Single Stream*) e TMR de Grão Grosso Paralelo de múltiplos canais (TMR CGP, *Coarse Grain Parallel Multiple Stream*).

A arquitetura *Multiple Stream* faz a leitura das entradas e a escrita das saídas em canais separados. O que pode ser vantajoso no caso da injeção de falhas acumuladas, pois se um barramento for atingido por *bit-flip*, talvez os outros barramentos não sejam afetados e a transferência dos dados seja concluída com sucesso sem afetar os resultados.

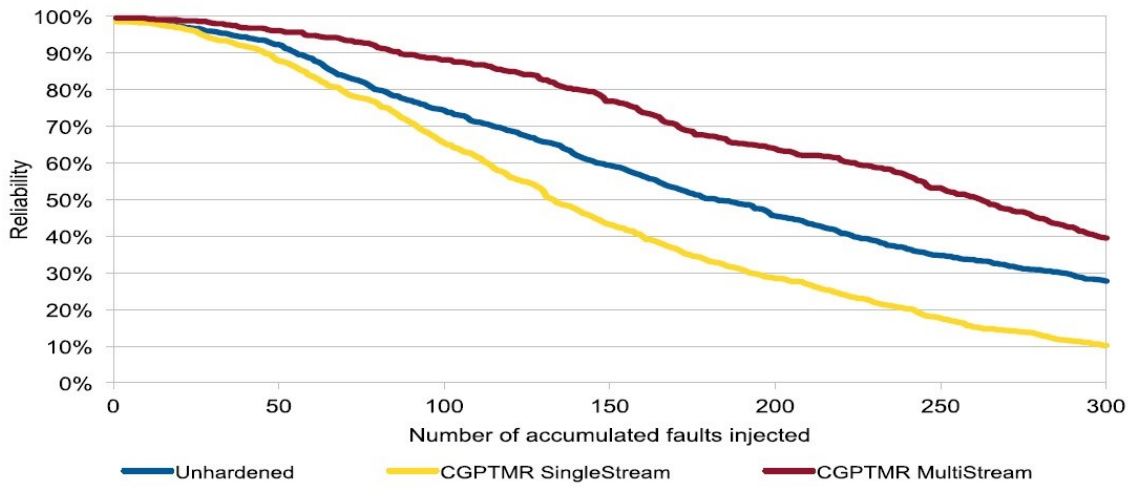


Figura 59 Confiabilidade em função do número de falhas acumuladas para os projetos sem e com redundância TMR

O gráfico de confiabilidade em função do número de falhas injetadas é exibido na figura 59. Os resultados obtidos por injeção de falhas demonstram que os projetos com arquitetura *Single Stream* tiveram sua confiabilidade comprometida preliminarmente aos outros projetos. A arquitetura *Multiple Stream* apresentou um melhor desempenho, pois suporta um número maior de falhas até exibir um erro na saída, observou-se a possibilidade de apenas parte dos dados transferidos serem perdidos, ao contrário da arquitetura *Single Stream* que pode ficar com a sua confiabilidade comprometida quando um *bit-flip* atinge o canal de transferência de dados (*Input Stream* ou *Output Stream*), com relação direta a quantidade de dados que é serializada através do fluxo ilustrado na figura 41.

Considerando o fluxo de nêutrons em Nova York como referência (13 n / cm^2.h) (JEDEC, 2006) e a seção de choque estática do dispositivo (*Static Cross Section Device*) para FPGA Xilinx Artix 7 ($7x10^{-15}$ cm^2/bit) (XILINX, 2015b) podemos estimar a seção de choque estática para a área alvo (*Static Cross Section Target Area*) (388 quadros * 3232 bits = 1254016 bits) que é $8,78x10^{-9}$ cm^2. A expressão para obter a seção de choque estática para a área alvo é:

$$\sigma_{static}\ target\ area = \sigma_{static}\ device * Bits\ target\ area \quad (11)$$

Onde, 'σ *static target area*' é a seção de choque estática da área alvo, 'σ *static device*' é a seção de choque estática do dispositivo fornecida pela Xilinx e '*Bits target area*' é o total de bits da área (*Configuration bits*).

A Taxa de Falhas (*Failure Rate*) é uma métrica de confiabilidade bem conhecida, pode ser dependente de tempo ou não (D. CROWE, 2001). A Taxa de Falhas para a área alvo é $1,14x10^{-7}$, calculada através do valor da multiplicação da seção de choque estática da área alvo (σ*Static Cross Section Target Area*) pelo fluxo de nêutrons de Nova York, como segue:

$$Failure\ Rate\ target\ area = \sigma_{static}\ target\ area\ x\ Fluxo \quad (12)$$

Onde, '*Failure Rate target area*' é a taxa de falhas da área alvo, 'σ *static target area'* é a seção de choque estática da área alvo e '*Fluxo*' é o Fluxo de nêutrons.

O tempo médio entre falhas (MTBF, *Mean Time Between Failure*) é definido como o tempo médio em que um dispositivo ou produto funciona antes de falhar. O MTBF de falhas para área alvo é $8,7x10^{6}$ h, calculado da seguinte forma:

$$MTBF\ target\ area = (1/\ Failure\ Rate\ target\ area) \quad (13)$$

Onde, '*MTBF target area*' é o tempo médio entre falhas, '*Failure rate target area*' é a taxa de falhas para a área alvo. Então o MBTF para o projeto é calculado da seguinte forma:

$$MTBF\ design = MTBF\ target\ area\ x\ Accumulated\ bits \quad (14)$$

Onde, '*Accumulated bits*' é o número de falhas necessário para causar uma falha.

Tabela 10 Confiabilidade em relação ao número de bit-flips acumulados e MTBF

Confiabilidade	Bit-flips acumulados		MTBF projeto	
	Unhardened	CGP TMR	Unhardened	CGP TMR
99%	10	17	$8.7x10^7$	$1.48x10^8$ (+70%)
95%	41	61	$3.6x10^8$	$5.4x10^8$ (+50%)
90%	55	87	$4.8x10^8$	$7.6x10^8$ (+58%)

A tabela 10 demonstra que para uma confiabilidade de 99% a versão sem redundância necessita de 10 falhas acumuladas para apresentar um erro com seu valor de MTBF de $8.7x10^7$ horas. O projeto com redundância TMR CGP precisa de 17 falhas acumuladas para gerar um erro com MTBF de $1.48x10^8$ horas, com 70% a mais de tolerância se comparado com o projeto desprotegido. O projeto com redundância TMR CGP de múltiplos canais foi efetivo no seu comportamento se comparado com o projeto desprotegido, tolerando um maior número de falhas para ter seu funcionamento comprometido. Para uma confiabilidade ser definida como 100% nenhuma falha transiente poderá ter atingido o circuito. O cálculo para definir a confiabilidade (*Reliability*) é realizado da seguinte forma:

$$Confiabilidade = (1 - \%Hangs - \%SDCs) \quad (15)$$

Onde '%Hangs' é representado pela porcentagem de erros de interrupções, descrita na expressão 16, sendo calculada dividindo-se o número de falhas necessárias para a ocorrência de uma interrupção (#Hang) pelo número total de falhas injetadas. Este cálculo é realizado a cada ocorrência de um novo evento de maneira que este valor será acrescido das outras ocorrências e consequentemente irá aumentar.

$$\%Hangs = (\#Hang / TotalFalhasInjetadas) \quad (16)$$

O cálculo da porcentagem de SDCs ou erros de corrupção silenciosa de dados (%SDCs) descrito na expressão 17 é realizado de forma similar ao cálculo da expressão anterior. Divide-se o número de falhas necessárias para a ocorrência de um SDC (#SDC) pelo número total de falhas injetadas. Este cálculo é realizado a cada ocorrência de um novo evento e consequentemente irá aumentar.

$$\%SDCs = (\#SDC / TotalFalhasInjetadas) \quad (17)$$

Concluindo, quanto maior for o número de falhas acumuladas menor será a porcentagem de confiabilidade, num determinado ponto o sistema irá parar de funcionar, servindo como referência para o projetista ter uma expectativa de reconfiguração e funcionamento do sistema.

8 CONCLUSÕES

Este trabalho compreendeu um estudo sobre efeitos da radiação em circuitos programáveis em FPGAs baseados em SRAM e análise de circuitos gerados por ferramenta de Síntese de Alto Nível empregando redundância. Este trabalho tem como objetivo analisar o emprego de redundância em C nas ferramentas de HLS da Xilinx e testar os circuitos gerados sob falhas simples e falhas acumuladas no FPGA. Os resultados se mostraram muito promissores. Pode-se observar que as áreas de maior vulnerabilidade dos circuitos gerados por HLS são: A máquina de estados (FSM) de controle do acelerador de *hardware*, barramentosde controle (*Control BUS*) e transferência de dados (*Input and output Stream*), votadores, interconexões entre as BRAMs que armazenam os dados de entrada e os resultados das multiplicações. Nota-se também que as redundâncias não foram simplificadas pela ferramenta de síntese, o que sugere que a aplicação de técnicas de redundância em C na síntese de alto nível pode ser usada para mascarar falhas em FPGAs.

Com base nos resultados de injeção de falhas simples, pode-se concluir que a melhor configuração de técnicas de tolerância a falhas aplicadas neste trabalho foi TMR de Grão Fino Paralelo com otimização Unroll de Múltiplos Canais (*TMR Fine Grain Parallel Unroll Multiple Stream*), pois apresentou o menor valor de bits críticos em relação aos bits essenciais e o menor valor para Seção de Choque Dinâmica do Circuito (*Dynamic Cross Section Circuit Area*), ou seja, a menor probabilidade de gerar um erro na saída do projeto se atingido por uma partícula (SEU). Com base nos resultados de falhas acumuladas, pode-se concluir que a melhor configuração de técnica de tolerância a falhas foi TMR de Grão Grosso Paralelo de Múltiplos Canais (*TMR Coarse Grain Parallel Multiple Stream*), pois tolerou um número maior de falhas até ter sua confiabilidade comprometida.

Neste trabalho preliminarmente foi realizado um estudo sobre os efeitos da radiação espacial em circuitos integrados e suas consequências na evolução tecnológica da indústria de semicondutores. Posteriormente foram apresentadas as classificações dos efeitos da radiação espacial ao que se referem aos danos sofridos pelos dispositivos eletrônicos e circuitos integrados a fim de evidenciar a necessidade de mitigação e proteção. No primeiro capítulo foi enfatizada a sensibilidade de FPGAs baseados em SRAM sob SEEs e as tendências para novas tecnologias, observando os potenciais modos de falhas e seus efeitos nos dispositivos. Desse modo o primeiro capítulo contribui com uma revisão inclusiva na área de tolerância a falhas a circuitos digitais e pode servir como referência a pesquisadores e estudantes que ambicionam ingressar na área.

O segundo capítulo apresenta o ambiente de radiação e seus efeitos em componentes eletrônicos. A interação da radiação solar e cósmica com a magnetosfera terrestre. Definições para fluxo de partículas e unidades de medidas utilizadas. Definição para Transferência Linear de Energia (LET), Dose Total Ionizante (TID) e seus efeitos em circuitos integrados. Classificações e efeitos causados por SEEs em dispositivos programáveis, principalmente FPGAs baseados em SRAM.

O terceiro capítulo retrata a contextualização deste trabalho no que se refere à geração automática de aceleradores de *hardware* pela ferramenta Xilinx High Level Synthesis Tools. Através da possibilidade de transformar uma especificação/algoritmo código C em *hardware* RTL, desse modo expondo as vantagens e possibilidades para os projetistas trabalharem em conjunto com o FPGA. Na primeira subseção são expostas as diretivas de otimização disponíveis pela ferramenta para contribuir com o projetista de *hardware* na construção de uma arquitetura que atenda as metas de desempenho e área almejadas. Na segunda subseção é descrito o fluxo de síntese e implementação dos padrões utilizados pela ferramenta Vivado HLS. Na terceira subseção são evidenciadas as características da ferramenta Vivado Design Suite Tools, através de exemplos de projetos e arquiteturas utilizadas neste trabalho. Deste modo evidenciando a interação e colaboração entre as duas ferramentas para um projeto com portabilidade em um FPGA Xilinx.

O quarto capítulo enfatiza o uso de FPGAs comerciais que requerem alto grau de confiabilidade, expõe sobre técnicas de tolerância a falhas em FPGAs baseados em SRAM e mitigação de erros em nível de sistema, DWC, TMR. No final do capítulo são apresentadas as versões de TMR implementadas neste trabalho.

O quinto capítulo evidencia o estudo de caso: projetos implementados, arquiteturas utilizadas, representação do número de passos necessários para a execução da aplicação correspondente a cada projeto.

O sexto capítulo expõe a metodologia do injetor de falhas utilizado neste trabalho para preparação de um ambiente de injeção de falhas na memória de configuração do FPGA. São apresentados alguns parâmetros para caracterizar um projeto em FPGA baseado em SRAM em erros suaves. A metodologia utilizada na injeção de falhas simples e acumulada. A forma de obtenção de resultados e suas classificações. Na primeira subseção é apresentado um trabalho correlato com análise de métricas para a estimativa de suscetibilidade a falhas simples.

O sétimo capítulo retrata os resultados de área, desempenho e bits críticos para os projetos no modo de injeção de falhas simples. Para os projetos no modo de injeção de falhas

acumuladas são apresentados os resultados de área, desempenho e confiabilidade. Os resultados no modo de injeção de falhas simples demonstram a efetividade dos projetos com redundância TMR, exibindo valores de bits críticos menores em relação aos bits essenciais dos projetos sem redundância. No caso da injeção de falhas acumuladas somente a arquitetura com redundância TMR de múltiplos canais demonstrou ser mais confiável que o projeto sem redundância, pois suporta mais SEUs acumulados antes de falhar ou comprometer o funcionamento correto do projeto.

Existem muitas possibilidades a serem investigadas para obter um bom projeto, representado por uma boa relação entre área, desempenho e confiabilidade. Trabalhos futuros serão realizados como:

- Realizar testes com implementações de novas aplicações de estudo de caso.
- Testar novas otimizações.
- Realizar testes de radiação para comparação com os resultados de injeção de falhas.

Por fim, pode-se dizer que os trabalhos nesta dissertação agregaram resultados interessantes à área de tolerância a falhas causadas pela radiação em circuitos programáveis utilizando projetos gerados por ferramenta de Síntese de Alto Nível (HLS) pelo fato de provar a efetividade do uso da redundância, tornando os circuitos protegidos menos suscetíveis a falhas. É importante destacar a necessidade de utilizar outras aplicações para testes e comparações com a aplicação de multiplicação de matrizes, definida como dependente de fluxo de dados. Em (TAMBARA, 2017) foram testadas e comparadas algumas aplicações diferentes, porém sem uso de redundância. Estes resultados deverão servir de base para comparações futuras com novos resultados das aplicações utilizando redundância TMR.

REFERÊNCIAS

HEMPEL, GERALD; JAN HOYER; PIONTECK, THILO; HOCHBERGER, CHRISTIAN; Register allocation for high-level synthesis of hardware accelerators targeting FPGAs, **RECONFIGURABLE AND COMMUNICATION-CENTRIC SYSTEMS-ON-CHIP** (ReCoSoC), 2013. DOI: 10.1109/ReCoSoC.2013.6581522

VILLARREAL, J; PARK, A; NAJJAR, W; HALSTEAD, R. Designing modular hardware accelerators in c with roccc 2.0, Field-Programmable Custom Computing Machines (FCCM), 18th IEEE Annual International Symposium, pp. 127 –134, 2010. DOI: 10.1109/FCCM.2010.28

PILATO, C; F. FERRANDI, F; SCIUTO, D. A design methodology to implement memory accesses in high-level synthesis, SEVENTH IEEE/ACM/IFIP INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, ,pp. 49–58, 2011. Disponível em: <http://doi.acm.org/10.1145/2039370.2039381>.

XILINX. Vivado Design Suite User Guide High-Level Synthesis UG902, 2016. Disponível em: <www.xilinx.com>. Acesso em: Jan. 2017.

XILINX. Vivado Design Suite User Guide Design Flows Overview UG892, 2013. Disponível em: <www.xilinx.com>. Acesso em: Jan. 2017.

XILINX. Vivado Design Suite User Guide High-Level Synthesis UG902, 2013b. Disponível em: <www.xilinx.com>. Acesso em: Jan. 2017.

NANE, R; SIMA, V; PILATO, C; FORT, B. Survey and Evaluation of FPGA High-Level Synthesis Tools, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, vol.35, pg. 1591 – 1604, 2016.

O'BRYAN, M. Radiation Effects & Analysis, Nov.2015. Disponível em: <http://radhome.gsfc.nasa.gov/radhome/see.htm>. Acessado em: Jan. 2017.

SRINIVASAN, S. Toward Increasing FPGA Lifetime. **IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING**, Los Alamitos, V.5, n.2, pg.115-127, Abril/Jun 2008.

TAMBARA, L. Caracterização de Circuitos Programáveis e Sistemas em Chip sob Radiação, PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA, UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL, pg 17-18, 2013.Disponível em: <http://hdl.handle.net/10183/86477>. Acessado em: Mar. 2017.

Internacional Solid-State Circuits Conference. **ISSCC Trends 2016**. Disponível em: <http://isscc.org/doc/2016/ISSCC2016_TechTrends.pdf>. Acessado em: Abril. 2017.

RAJE, S. **Extending the Power of FPGAs**, 2015. Disponível em: <https://www.xilinx.com/ publications/prod_mktg/club_vivado/presentation-2015/korea/Seoul_Salil_Raje.pdf>.

KASTENSMIDT, F. Designing Fault-Tolerant Techniques for SRAM-based FPGAs, UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL, 2004. Disponível em: < http://hdl.handle.net/10183/27593>. Acesso em: Fev. 2017.

ACTEL, Effects of Neutrons on Programmable Logic White Paper, pg 13, 2002.

GOMES, I. A. C. Uso de Redundância Modular Tripla Aproximada para Tolerância a Falhas em Circuitos Digitais, PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA, UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL, pg 20-21, 2014. Disponível em: <http://hdl.handle.net/10183/99056>. Acessado em: Mar. 2017.

NORMAND, E. Single Event Effects in Avionics. **IEEE Transactions on Nuclear Science**, vol. 43, pag. 461-474, Abril. 1996.

ROTH, R D. Solid State Microdosimeter for Radiation Monitoring in Spacecraft and Avionics, **IEEE Transactions on Nuclear Science**, vol.41, pg. 2118-2124, Dez. 1994.

AUSTRALIAN TRANSPORT SAFETY BUREAU (ATSB) – **Aviation occurrence investigation AO-2008-070 final -** In-flight upset 154 km west of Learmonth, WA 7 October 2008 VH-QPA Airbus A330-303. Canberra, 2011. Commonwealth of Australia.

MACHADO, S. R. F. Estudo de um Processo de Garantia da Confiabilidade de Sistemas Eletrônicos Embarcados a Single Event Upsets causados por Partículas Ionizantes, **MINISTÉRIO DA CIÊNCIA E TECNOLOGIA E INOVAÇÃO - INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS**, Maio. 2014.

BAKER, D. N. Severe Space Weather Events - Understanding Societal and Economic Impacts**, Committee on the Societal and Economic Impacts of Severe Space Weather Events:** A Workshop, National Research Council, pg. 1-33, 2009.

ADELL, P; ALLEN, G; Assessing and Mitigating Radiation Effects in Xilinx SRAM FPGAs, **Radiation and Its Effects on Components and Systems (RADECS)**, 2008.

TONFAT, J; KASTENSMIDT, F; RECH, P. Analyzing the Effectiveness of a Frame-Level Redundancy Scrubbing Technique for SRAM-based FPGAs , **IEEE Transactions on Nuclear Science**, v62, pg.3080-3087, Dez. 2015.

MICROSEMI, **Understanding Single Event Effects in FPGAs**, 2011. Disponível em: <www.actel.com> Acessado em Abril. 2017.

TONFAT, J. TAMBARA, L. SANTOS A. KASTENSMIDT, F. Method Analyze the Susceptibility of HLS Designs in SRAM-based FPGAs Under Soft Errors, **LECTURE NOTES IN COMPUTER SCIENCE,** vol. 9625, pg. 132-143, Mar. 2016.

XILINX. Vivado Design Suite User Guide Design Getting Started UG910, 2016b. Disponível em: <www.xilinx.com>. Acesso em: Jan. 2017.

CHIELLE, E. Selective Software-Implemented Hardware Fault Tolerance Techniques to Detect Soft Errors In Processors with Reduced Overhead, PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA, UNIVERSIDADE FEDERAL DO RIO

GRANDE DO SUL, pg 34-35, 2016. Disponível em: < http://hdl.handle.net/10183/142568>. Acessado em: Mar. 2017.

MORISON, I. Introduction to Astronomy and Cosmology. John Wiley & Sons, Dec. 2008.

ARRUDA, T. M. Análise da Influência da radiação em circuitos eletrônicos. Dissertação (Mestrado em Engenharia Aeronáutica e Mecânica e Área de Sistemas Aeroespaciais e Mecatrônica) – Instituto Tecnológico da Aeronáutica, São José dos Campos – SP, 2006. Disponível em: <http://www.bd.bibl.ita.br/tde_busca/arquivo.php?codArquivo=290> . Acessado em: Abril. 2017.

KASTENSMIDT, F; REIS, R; CARRO, L. Fault-Tolerant Techniques for SRAM-Based FPGAs, Dordrecht: Springer, 2006.

STERPONE, L; VIOLANTE, M. Analysis of the Robustness of the TMR Architecture in SRAM-Based FPGAs, **IEEE Transactions on Nuclear Science**, Volume. 52, pg. 1545-1549, Oct. 2005.

STURESSON, F. Single Event Effects (SEE) Mechanism and Effects. **Space Radiation and its Effects on EEE Components**, Jun.2009.

LAURENCE, H. MUTUEL. Appreciating the Effectiveness of Single Event Effect. **IEEE/AIAA 33rd DIGITAL AVIONICS SYSTEMS CONFERENCE (DASC)**, pg 5B1-1 – 5B1-11, 2014.

AISHWARYA, S. MAHENDRAN, G. Multiple Bit Upset Correction in SRAM based FPGA using Mutation and Erasure codes, INTERNATIONAL CONFERENCE ON ADVANCED COMMUNICATION CONTROL AND COMPUTING TECHNOLOGIES (ACACCCT), pg 202-206, 2016.

VELAZCO, R; FOUCARD, PAUL PERONNARD, P. Combining Results of Accelerated Radiation Tests and Fault Injections to Predict the Error Rate of an Application Implemented in SRAM-Based FPGAs, IEEE TRANSACTIONS ON NUCLEAR SCIENCE, vol. 57, 2010. DOI: 10.1109/TNS.2010.2087355

XILINX. Soft Error Mitigation Using Prioritized Essential Bits, XAPP538 (v1.0), 2012. Disponível em: <www.xilinx.com>. Acesso em: Fev. 2017.

XILINX. Microblaze Processor Reference Guide , UG081 (v9.0) , 2008. Disponível em: <https://www.xilinx.com>.Acesso em: Jan 2017.

DU B.; DESOGUS M.; STERPONE L. Analysis and Mitigation of SEUs in ARM-based SoC on Xilinx Virtex-V SRAM-based FPGAs, **Ph.D. Research in Microelectronics and Electronics (PRIME), 2015 11th Conference on**, Jun. 2015.

COOK, K. L. B The ITAR and You - What You Need to Know about the International Traffic in Arms Regulations. IEEE AEROSPACE CONFERENCE, pg1-12, Mar. 2010.

ANGHEL, L.; ALEXANDRESCU, D.; NICOLAIDIS, M. Evaluation of a soft error tolerance technique based on time and/or space redundancy. **SYMPOSIUM ON INTEGRATED**

CIRCUITS AND SYSTEMS DESIGN, SBCCI. IEEE Computer Society, 2000. p. 237-242.

DUPONT, E.; NICOLAIDIS, M.; ROHR, P. Embedded robustness IPs for transient-error-free ICs. IEEE Design & Test of Computers, New York, v.19, n.3, p. 54-68, May-June 2002.

CARMICHAEL, C. TRIPLE MODULE REDUNDANCY DESIGN TECHNIQUES FOR VIRTEX FPGAS, Xilinx Application Note XAPP197, vol. 1, 2001.

JOHNSON, J; WIRTHLIN M. Voter Insertion Algorithms for FPGA Designs Using Triple Modular Redundancy, 18th annual ACM/SIGDA international symposium on Field programmable gate arrays, pg 249-258, Feb. 2010.

MILLER, G; CARMICHAEL, C; SWIFT, G. SINGLE-EVENT UPSET MITIGATION FOR XILINX FPGA BLOCK MEMORIES, XILINX Application Note XAPP962 v1, vol. 1, 2006.

STOTT, E; SEDCOLE, P; CHEUNG, P. Fault tolerant methods for reliability in FPGAs, Field Programmable Logic and Applications, FPL, pp. 415–420, 2008.

LAHRACH, F; DOUMAR, A; CHÂTELET, E; ABDAOUI, A. Master-Slave TMR Inspired Technique for Fault Tolerance of SRAM-based FPGA, VLSI (ISVLSI), IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM, 2010. DOI: 10.1109/ISVLSI.2010.38

FOUCARD, G; PERONNARD, P; VELAZCO. R. Reliability Limits of TMR implemented in a SRAM-based FPGA: Heavy Ion Measures vs. Fault injection Predictions, Test Workshop (LATW), 11th Latin American, 2010.
DOI: 10.1109/LATW.2010.5550337

KASTENSMIDT, F; STERPONE, L; CARRO, L; REORDA, M. On the Optimal Design of Triple Modular Redundancy Logic for SRAM-based FPGAs, DESIGN AUTOMATION AND TEST IN EUROPE, 2005. DOI: 10.1109/DATE.2005.229

ZHAO, Z; AGIAKATSIKAS, D; NGUYEN, N; CETIN, E; DIESSEL, O. Fine-grained Module-based Error Recovery in FPGA-based TMR Systems, FIELD-PROGRAMMABLE TECHNOLOGY (FPT) INTERNATIONAL CONFERENCE, 2016.
DOI: 10.1109/FPT.2016.7929433

TAMBARA, L; TONFAT, J; SANTOS A; KASTENSMIDT, F. Analyzing Reliability and Performance Trade-Offs of HLS-Based Designs in SRAM-Based FPGAs Under Soft Errors, **IEEE TRANSACTIONS ON NUCLEAR SCIENCE**, Vol. 64. NO.2. Fev.2017.

SANTOS, A.F.; TAMBARA, L.; BENEVENUTI, F.; TONFAT, J.; KASTENSMIDT, F. Applying TMR in Hardware Accelerators Generated by High-Level Synthesis Design Flow for Mitigating Multiple Bit Upsets in SRAM-Based FPGAs, **LECTURE NOTES IN COMPUTER SCIENCE**, vol. 10216, pg. 202-213, Mar. 2017. DOI:10.1007/978-3-319-56258-2_18.

XILINX. 7 Series FPGAs Configuration - User Guide, UG470 (v1.10), 2015. Disponível em: <www.xilinx.com>. Acesso em: Jan. 2017.

AISHWARYA S; MAHENDRAN G. Multiple Bit Upset correction in SRAM based FPGA using self repairable Erasure codes, **International Conference on Emerging Engineering Trends and science**, pg 356-362, 2016.

RECH, P; PILLA, L.L; NAVAUX, P.O.A; CARRO, L. Impact of GPUs Parallelism Management on Safety-Critical and HPC Applications Reliability. **44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks,** pg. 455-466, 2014.

JEDEC (2006). Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices JEDEC Standard [Online]. Disponível em: http://www.jedec.org/sites/default/files/docs/jesd89a.pdf.

XILINX. Device Reliability Report, UG116 (v9.4), 2015b. Disponível em: <Xilinx.com>. Acesso em: Fev. 2017.

D. CROWE, A. FEINBERG, "Design for Reability", International Standard Book Number: 13:978-1-4200-4084-5, Disponível em: https://www.crcpress.com/, 2001.

XILINX. AXI TIMER LogiCORE IP product Guide, PG079 (v2.0), 2016c. Disponível em: <https://www.xilinx.com>. Acesso em: Jan 2017.

ANEXOS:

Pseudocódigo em C para a implementação de multiplicação de matrizes 6x6.

```
read data from the input stream: a, b
     void matrixmul(mat_a_t a[6][6], mat_b_t b[6][6], result_t res[6][6])
     {
          int i, j, k;
          result_t accum;

          row: for(i=0; i<MAT_DIM; i++){
              col: for(j=0; j<MAT_DIM; j++){
                                    accum = 0;
                   prod: for(k=0; k<MAT_DIM; k++){
                              accum += a[i][k] * b[k][j];
                              res[i][j] = accum;
                   }
              }
          }
      }
write data from the output stream: res
```

Pseudocódigo em C para TMR de Grão Grosso Serial de Canal Simples (*Single Stream*):

```
read the data from input the stream: a0, b0,a1 ,b1, a2, b2

// do matrix multiplication TMR Coarse Grain Serial
Row_res1: for(i=0; i<MAT_DIM; i++) {
  Col_res1: for(j=0; j<MAT_DIM; j++) {
   accum1 = 0;
   Prod_res1: for(k=0; k<MAT_DIM; k++) {
         accum1 += mat_a1[i][k] * mat_b1[k][j];
          mat_res1[i][j] = accum1;
} } }

Row_res2: for(i=0; i<MAT_DIM; i++) {
     Col_res2: for(j=0; j<MAT_DIM; j++) {
     Accum2 = 0;
     Prod_res2: for(k=0; k<MAT_DIM; k++) {
        Accum2 += mat_a2[i][k] * mat_b2[k][j];
         mat_res2[i][j] = accum2;
} } }

Row_res3: for(i=0; i<MAT_DIM; i++) {
     Col_res3: for(j=0; j<MAT_DIM; j++) {
     Accum3 = 0;
     Prod_res3: for(k=0; k<MAT_DIM; k++) {
        Accum3 += mat_a3[i][k] * mat_b3[k][j];
         mat_res3[i][j] = accum3;
} } }

// Voter
voter: for(i=0; i<MAT_DIM; i++){
   voter_: for(j=0; j<MAT_DIM; j++){
       voter_func(&mat_res1[i][j], &mat_res2[i][j], &mat_res3[i][j],
       &voterOutput);
        mVoter[i*MAT_DIM+j] = voterOutput.mVoter;
        status_func(&mat_res1[i][j], &mat_res2[i][j], &mat_res3[i][j],
       &voter_status);
      status[i*MAT_DIM+j] = voter_status.mStatus;
} }
write the data to the stream: mVoter
write the data to the stream: mStatus
```

Pseudocódigo em C para TMR de Grão Grosso Serial de Múltiplos Canais (*Multiple Stream*):

```
read the data from input the stream0: a0, b0
read the data from input the stream1: a1, b1
read the data from input the stream2: a2, b2

// do matrix multiplication TMR Coarse Grain Serial
Row_res1: for(i=0; i<MAT_DIM; i++) {
  Col_res1: for(j=0; j<MAT_DIM; j++) {
   accum1 = 0;
   Prod_res1: for(k=0; k<MAT_DIM; k++) {
         accum1 += mat_a1[i][k] * mat_b1[k][j];
          mat_res1[i][j] = accum1;
} } }

Row_res2: for(i=0; i<MAT_DIM; i++) {
     Col_res2: for(j=0; j<MAT_DIM; j++) {
     Accum2 = 0;
     Prod_res2: for(k=0; k<MAT_DIM; k++) {
        Accum2 += mat_a2[i][k] * mat_b2[k][j];
         mat_res2[i][j] = accum2;
} } }

Row_res3: for(i=0; i<MAT_DIM; i++) {
     Col_res3: for(j=0; j<MAT_DIM; j++) {
     Accum3 = 0;
     Prod_res3: for(k=0; k<MAT_DIM; k++) {
        Accum3 += mat_a3[i][k] * mat_b3[k][j];
         mat_res3[i][j] = accum3;
} } }

// Voter
voter: for(i=0; i<MAT_DIM; i++){
   voter_: for(j=0; j<MAT_DIM; j++){
       voter_func(&mat_res1[i][j], &mat_res2[i][j], &mat_res3[i][j],
       &voterOutput1);
        voter_func(&mat_res1[i][j], &mat_res2[i][j], &mat_res3[i][j],
       &voterOutput2);
        voter_func(&mat_res1[i][j], &mat_res2[i][j], &mat_res3[i][j],
       &voterOutput3);

        mVoter1[i*MAT_DIM+j] = voterOutput1.mVoter;
        mVoter2[i*MAT_DIM+j] = voterOutput2.mVoter;
        mVoter3[i*MAT_DIM+j] = voterOutput3.mVoter;

        status_func(&mat_res1[i][j], &mat_res2[i][j], &mat_res3[i][j],
       &voter_status);
      status[i*MAT_DIM+j] = voter_status.mStatus;
} }
write the data to the stream0: mVoter1
write the data to the stream1: mVoter2
write the data to the stream2: mVoter3
write the data to the stream3: mStatus
```

Pseudocódigo em C para TMR de Grão Grosso Paralelo de Canal Simples (*Single Stream*):

```
read the data from input the stream: a0, b0, a1, b1, a2, b2

// do matrix multiplication TMR Coarse Grain Parallel
matrixmul(mat_a1, mat_b1, mat_res1);
matrixmul(mat_a2, mat_b2, mat_res2);
matrixmul(mat_a3, mat_b3, mat_res3);

// Votador
for(i=0; i< MAT_DIM; i++){
   for(j=0; j< MAT_DIM; j++){
 voter_func(&mat_res1[i][j], &mat_res2[i][j], &mat_res3[i][j], &voterOutput1);
 mVoter[i*MAT_DIM+j] = voterOutput.mVoter;
 status_func(&mat_res1[i][j], &mat_res2[i][j], &mat_res3[i][j],
 &voter_status);
 status[i*MAT_DIM+j] = voter_status.mStatus;
  }
}

write the data to the stream: mVoter
write the data to the stream: mStatus
```

Pseudocódigo em C para TMR de Grão Grosso Paralelo de Múltiplos Canais (*Multiple Stream*):

```
read the data from input the stream0: a0, b0
read the data from input the stream1: a1, b1
read the data from input the stream2: a2, b2

// do matrix multiplication TMR Coarse Grain Parallel
matrixmul(mat_a1, mat_b1, mat_res1);
matrixmul(mat_a2, mat_b2, mat_res2);
matrixmul(mat_a3, mat_b3, mat_res3);

// Votador
for(i=0; i< MAT_DIM; i++){
   for(j=0; j< MAT_DIM; j++){
 voter_func(&mat_res1[i][j], &mat_res2[i][j], &mat_res3[i][j], &voterOutput1);
 voter_func(&mat_res1[i][j], &mat_res2[i][j], &mat_res3[i][j], &voterOutput2);
 voter_func(&mat_res1[i][j], &mat_res2[i][j], &mat_res3[i][j], &voterOutput3);

 mVoter1[i*MAT_DIM+j] = voterOutput1.mVoter;
 mVoter2[i*MAT_DIM+j] = voterOutput2.mVoter;
 mVoter3[i*MAT_DIM+j] = voterOutput3.mVoter;

 status_func(&mat_res1[i][j], &mat_res2[i][j], &mat_res3[i][j],
 &voter_status);
 status[i*MAT_DIM+j] = voter_status.mStatus;
  }
}

write the data to the stream0: mVoter1
write the data to the stream1: mVoter2
write the data to the stream2: mVoter3
write the data to the stream3: mStatus
```

Pseudocódigo em C da Função de Multiplicação de Matrizes específica para TMR de Grão Fino Paralelo:

```
void matrixmul_2({
     Row: for(int i=0; i<MAT_DIM; i++) {
        Col: for(int j=0; j<MAT_DIM; j++) {
          accum1 = 0;
          accum2 = 0;
          accum3 = 0;
          accumv1= 0;
          accumv2= 0;
          accumv3= 0;

         Prod: for(int k=0; k<MAT_DIM; k++) {
          accum1 += a1[i][k] * b1[k][j];
          accum2 += a2[i][k] * b2[k][j];
          accum3 += a3[i][k] * b3[k][j];

          status_func(&accum1, &accum2, &accum3, &voter_status);
          status[i*MAT_DIM+j] = voter_status.mStatus;

          accumv1 = ( (accum2 & accum3) | (accum1 &accum3) | (accum1 & accum2) |
          (accum1  & accum2 & accum3) );
          accumv2 = ( (accum2 & accum3) | (accum1 &accum3) | (accum1 & accum2) |
          (accum1 & accum2 & accum3) );
          accumv3 = ( (accum2 & accum3) | (accum1 &accum3) | (accum1 & accum2) |
          (accum1 & accum2 & accum3) );

          accum1 = accumv1;
          accum2 = accumv2;
          accum3 = accumv3;

          res1[i][j] = accum1;
          res2[i][j] = accum2;
          res3[i][j] = accum3;
         }

         voter_func(&res1[i][j], &res2[i][j], &res3[i][j], &voterOutput);
         mVoter[i*MAT_DIM+j] = voterOutput.mVoter;
        }
      }
      }
```

Pseudocódigo em C para TMR de Grão Fino Paralelo de Canal Simples (*Single Stream*):

```
read the data from input the stream: a0, b0, a1, b1, a2, b2

// do matrix multiplication TMR Fine Grain
Matrixmul_2 (mat_a1, mat_b1, mat_a2, mat_b2, mat_a3, mat_b3, mVoter1, status);
Matrixmul_2(mat_a1, mat_b1, mat_a2, mat_b2, mat_a3, mat_b3, mVoter2, status);
Matrixmul_2(mat_a1, mat_b1, mat_a2, mat_b2, mat_a3, mat_b3, mVoter3, status);

// Voter 2
for(i=0; i<MAT_DIM; i++){
  for(j=0; j<MAT_DIM; j++){
     voter_func(&mVoter1[i*MAT_DIM+j], &mVoter2[i*MAT_DIM+j],
     &mVoter3[i*MAT_DIM+j], &voterOutput);

     mVoter_out[i*MAT_DIM+j] = voterOutput.mVoter;
  }
}
write the data to the stream: mVoter
```

Pseudocódigo em C para TMR de Grão Fino Paralelo de Múltiplos Canais (*Multiple Stream*):

```
read the data from input the stream0: a0, b0
read the data from input the stream1: a1, b1
read the data from input the stream2: a2, b2

// do matrix multiplication TMR Fine Grain
Matrixmul_2 (mat_a1, mat_b1, mat_a2, mat_b2, mat_a3, mat_b3, mVoter1, status);
Matrixmul_2(mat_a1, mat_b1, mat_a2, mat_b2, mat_a3, mat_b3, mVoter2, status);
Matrixmul_2(mat_a1, mat_b1, mat_a2, mat_b2, mat_a3, mat_b3, mVoter3, status);

// Voter 2
for(i=0; i<MAT_DIM; i++){
  for(j=0; j<MAT_DIM; j++){
     voter_func(&mVoter1[i*MAT_DIM+j], &mVoter2[i*MAT_DIM+j],
     &mVoter3[i*MAT_DIM+j], &voterOutput1);
     voter_func(&mVoter1[i*MAT_DIM+j], &mVoter2[i*MAT_DIM+j],
     &mVoter3[i*MAT_DIM+j], &voterOutput2);
     voter_func(&mVoter1[i*MAT_DIM+j], &mVoter2[i*MAT_DIM+j],
     &mVoter3[i*MAT_DIM+j], &voterOutput3);

     mVoter_out1[i*MAT_DIM+j] = voterOutput1.mVoter;
     mVoter_out2[i*MAT_DIM+j] = voterOutput2.mVoter;
     mVoter_out3[i*MAT_DIM+j] = voterOutput3.mVoter;
  }
}
write the data to the stream0: mVoter1
write the data to the stream1: mVoter2
write the data to the stream2: mVoter3
write the data to the stream3: mStatus
```

PUBLICACÕES:

ARC 2016:
TONFAT, J. TAMBARA, L. SANTOS A. KASTENSMIDT, F. Method Analyze the Susceptibility of HLS Designs in SRAM-based FPGAs Under Soft Errors, **LECTURE NOTES IN COMPUTER SCIENCE,** vol. 9625, pg. 132-143, Mar. 2016.

ARC 2017:
SANTOS, A.F.; TAMBARA, L.; BENEVENUTI, F.; TONFAT, J.; KASTENSMIDT, F. Applying TMR in Hardware Accelerators Generated by High-Level Synthesis Design Flow for Mitigating Multiple Bit Upsets in SRAM-Based FPGAs, **LECTURE NOTES IN COMPUTER SCIENCE**, vol. 10216, pg. 202-213, Mar. 2017. DOI:10.1007/978-3-319-56258-2_18.

NSREC 2016:
TAMBARA, L; TONFAT, J; SANTOS A; KASTENSMIDT, F. Analyzing Reliability and Performance Trade-Offs of HLS-Based Designs in SRAM-Based FPGAs Under Soft Errors, **IEEE TRANSACTIONS ON NUCLEAR SCIENCE**, Vol. 64. NO.2. Fev.2017.

LASCAS 2017:
Evaluating the Efficiency of using TMR in the High-Level Synthesis Design Flow of SRAM-based FPGA. Aguardando publicação. Publicado no LASCAS 2017.