PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE ENGENHARIA
PROGRAMA DE PÓS GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Nícolas Marroni

Desenvolvimento de Uma Metodologia de Injeção de Falhas de Atraso Baseada em FPGA

Porto Alegre – Rio Grande do Sul

Desenvolvimento de Uma Metodologia de Injeção de Falhas de Atraso Baseada em FPGA

Dissertação apresentada como requisito para obtenção do grau de Mestre pelo Programa de Pós-Graduação da Faculdade em Engenharia Elétrica da Pontifícia Universidade Católica do Rio Grande do Sul.

Área de concentração: Sinais, Sistemas e Tecnologia da Informação
Linha de Pesquisa: Sistemas de Computação

Orientadora:
Profa. Dra. Letícia Maria Bolzani Poehls

Co-orientador:
Prof. Dr. Fabian Luis Vargas

Porto Alegre – Rio Grande do Sul
07 de novembro de 2013

DESENVOLVIMENTO DE UMA METODOLOGIA DE INJEÇÃO DE FALHAS DE ATRASO BASEADA EM FPGA

CANDIDATO: NÍCOLAS MARRONI

Esta Dissertação de Mestrado foi julgada para obtenção do título de MESTRE EM ENGENHARIA ELÉTRICA e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Pontifícia Universidade Católica do Rio Grande do Sul.

DRA. LETÍCIA MARIA BOLZANI POEHLS - ORIENTADORA

DR. FABIAN LUIS VARGAS - CO-ORIENTADOR

BANCA EXAMINADORA

DR. ARIEL LUTENBERG - FIUBA - UNIVERSIDADE DE BUENOS AIRES - UBA

DR. AURELIO TERGOLINA SALTON - PPGEE - FENG - PUCRS

Agradecimentos

Agradeço à minha orientadora, Profa. Dra. Letícia Maria Bolzani Poehls, pela confiança em mim depositada e sobretudo a atenção e suporte teórico a mim dedicados sem os quais não seria possível a realização deste trabalho.

Ao meu co-orientador Prof. Dr. Fabian Luis Vargas pela oportunidade de integrar o SiSC e principalmente por todos os ensinamentos em teste e confiabilidade de sistemas, ainda na graduação até a conclusão do presente trabalho de Mestrado.

Ao Prof. Dr. Fernando César Comparsi de Castro, e a Profa. Dra. Maria Cristina Felippetto de Castro pela oportunidade que me foi concedida no centro de pesquisa ainda como bolsista de iniciação científica, onde minha busca por conhecimento e aprimoramento constante teve início.

Agradeço à minha família a qual sempre me apoiou e incentivou na busca por meus objetivos durante todas as etapas da minha vida.

À todos os amigos e colegas do programa de pós-graduação por contribuírem com minha formação acadêmica e humana.

Ao governo brasileiro e a CAPES pela confiança e o aporte financeiro que possibilitaram a realização deste Mestrado através da concessão de bolsas de estudos.

"Victory favors neither the righteous

nor the wicked it favors the prepared."

Autor Desconhecido

Resumo

Com a evolução da tecnologia CMOS, a densidade e a proximidade entre as linhas de roteamento dos Circuitos Integrados (CIs) foram incrementadas substancialmente nos últimos anos. Pequenas variações no processo de fabricação, como ligações indesejadas entre trilhas adjacentes e variações no limiar de tensão dos transistores devido a alterações no processo de litografia podem causar um comportamento anômalo no CI. Assim, o desenvolvimento de novas metodologias de teste capazes de proverem uma elevada capacidade de detecção de falhas, oriundas a partir dos mais variados tipos de defeitos de manufatura tornaram-se essenciais nos dias de hoje. Especificamente diante de CIs fabricados a partir de tecnologias abaixo de 65nm, torna-se fundamental o uso de metodologias de teste que visam a detecção de falhas de atraso, pois as variações no processo de produção não manifestam uma alteração lógica no comportamento do circuito resultante, e sim uma alteração na temporização do circuito.

Neste contexto, esta dissertação de mestrado propõe o desenvolvimento de uma metodologia de injeção de falhas de atraso com a finalidade de extrair a cobertura de falhas e analisar a eficiência de metodologias de teste desenvolvidas para CIs complexos. A metodologia proposta visa nortear a inserção de falhas de atraso em pontos específicos do CI. Esses pontos de inserção são resultados do estudo de variações probabilística do processo de fabricação de CIs em larga escala e podem ser utilizados na modelagem de falhas de atraso decorrentes dessas variações. Através da especificação, implementação, validação e avaliação de uma ferramenta de emulação em *Field Programmable Gate Array* (FPGA), será possível avaliar a robustez de sistemas integrados complexos frente a falhas de atraso, extrair a cobertura de falhas e avaliar a eficiência tanto de metodologias de teste quanto de técnicas de tolerância a falhas.

Palavras-chave: Injeção de falhas, Emulação de falhas, FPGA, Sistemas tolerantes a falhas, Injeção de falhas de atraso.

Abstract

With the evolution of CMOS technology, density and proximity between routing lines of integrated circuits (ICs) have increased substantially in the recent years. Slight variations in the manufacturing process, as the undesired connection between adjacent tracks and variations in threshold voltage due to changes in the lithographic process can cause the IC to behave anomalously. In this context, the development of new test methodologies, which are capable of providing high capacity fault detection in order to identify defects, becomes essential. Specifically when manufacturing ICs using technologies below 65nm, the use of test methodologies that aim at detecting delay faults is crucial, thus the production process does not cause a change in the resulting logic circuit's behaviour, but only a change in the circuit's timing.

Thereby, this master thesis proposes the development of a methodology for the injection of delay faults in order to extract the delay fault coverage and to analyse the efficiency of existing methodologies for complex ICs. The proposed approach aims at guiding the insertion of delay faults into specific points of the IC. Such insertion points are results of the probabilistic variation in the manufacturing process of large-scale integrated circuits and can be used in modelling delay faults arising from such variations. Through the specification, implementation, validation and assessment of an emulation tool in the Field-Programmable Gate Array (FPGA) it will be possible to understand the degree of robustness of complex integrated systems against delay faults, extract the fault coverage and evaluate the efficiency of both test methodologies and techniques for fault tolerance.

Keywords: Fault Injection, Fault Emulation, Delay, Field Programmable Gate Arrays, Fault Tolerant Computing, Dynamic Delay-Fault Injection, Fault-Tolerance System.

Sumário

Página

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

- ANSI - *American National Standards Institute*
- ASIC - *Application-Specific Integrated Circuit*
- BIST - *Built-In Self-Test*
- BRAMS - *Block RAMs*
- CI - *Circuito Integrado*
- CMOS - *Complementary Metal-Oxide-Semiconductor*
- DF - *Delay Faults*
- DFT - *Design for Testability*
- E/S - *Entradas/Saídas*
- EDA - *Electronic Design Automation*
- EMI - *Electromagnetic Interference*
- FC - *Fault Coverage*
- FPGA - *Field-Programmable Gate Array*
- GUI - *Graphic User Interface*
- HDL - *Hardware Description Language*
- ICs - *Integrated Circuits*
- IP - *Intellectual Property*
- JTAG - *Joint Test Action Group*
- LUT - *Look-Up Table*
- NCD - *Native Circuit Description*

- NGD - *Native Generic Database*
- PD - *Path Delay*
- PROM - *Programmable Read Only Memory*
- RTL - *Register Transfer Level*
- SDC - *Synopsys Design Constraints*
- SDF - *Standard Delay Format*
- SDRAM - *Synchronous Dynamic Random Access Memory*
- SEE - *Single Event Effects*
- SET - *Single Event Transient*
- SoCs - *Systems-on-Chip*
- SRL - *Shift Register LUT*
- TCL - *Tool Command Language*
- TD - *Transition Delay*
- TIFs - *Técnicas de Injeção de Falhas*
- VHDL - *VHSIC hardware description language*
- VHSIC - *Very High Speed Integrated Circuit*
- VLSI - *Very-Large-Scale Integration*
- XST - *Xilinx Synthesis Technology*

1 Introdução

O aperfeiçoamento do processo de fabricação de semicondutores e a miniaturização da tecnologia tornou possível a integração de milhões de transistores em um único Circuito Integrado (CI), aumentando assim a complexidade dos mesmos. Essa redução nas dimensões no tamanho dos transistores, permitiu a diminuição do *delay* dos transistores o que, por sua vez, permite o desenvolvimento de sistemas de alta performance e com dimensões cada vez mais reduzidas. Esses sistemas passaram a integrar um enorme número de aplicações nas mais diversas áreas tecnológicas. Assim, aplicações críticas baseadas em *Systems-on-Chip* (SoCs), tais como aeroespaciais, médico-hospitalares, militares, entre outras, foram extremamente beneficiadas pelos avanços anteriormente mencionados. Entretanto, a miniaturização da tecnologia trouxe consigo uma série de desafios, dentre os quais podemos citar a confiabilidade e a testabilidade de SoCs.

A confiabilidade do CI é afetada a cada geração tecnológica uma vez que, a tensão de alimentação de SoCs é reduzida a fim manter sob controle o consumo de energia dinâmica dos mesmos. Essa redução provoca um aumento da suscetibilidade do sistema em relação ao ruído do meio, pois afeta o dimensionamento da tensão de *threshold* do transistor. No espaço, a probabilidade de uma partícula de radiação atingir uma determinada parte do CI e conseguir efetivamente inverter o nível lógico armazenado naquela região é muito superior em tensões mais baixas. No cotidiano, fatores externos como Interferências Eletromagnéticas (*Electromagnetic Interference* - EMI) podem causar instabilidades momentâneas no funcionamento do circuito, podendo gerar até mesmo um comportamento anômalo perceptível ao usuário do sistema, dependendo do nível de robustez agregado ao sistema em uso [6]. Além disso, a constante miniaturização da tecnologia aumenta a ocorrência de diferentes tipos de defeitos durante o processo de manufatura de CIs, devido fundamentalmente ao aumento na densidade e à proximidade das interconexões. Esses defeitos são decorrentes de variações no processo de fabricação, tais como, o desalinhamento de máscaras, a descontinuidade e o acoplamento das interconexões. Assim, observa-se atualmente uma alteração na forma como os defeitos se manifestam, podendo em alguns

casos não haver alteração da função lógica, mas sim uma alteração das características temporais do circuito. Neste sentido, a partir do comportamento funcional desses defeitos, surgem novos tipos de falhas que passam a ser alvo da maioria das metodologias de testes, bem como das técnicas de tolerância a falhas agregadas a fim de aumentarem a robustez de SoCs.

Dentre as principais falhas associadas aos defeitos de manufatura, as falhas de atraso (*Delay Faults* - DF) representam um grande desafio. Uma falha de atraso ocorre quando o valor propagado para um Elemento Sequencial (ES), através da lógica combinacional, é corrompido devido a um atraso no sinal durante o período de amostragem do ES [1]. Falhas de temporização em CIs são de difícil verificação, pois exigem que a mesma seja executada *at speed*, ou seja, a verificação deve ser feita com o sistema trabalhando na sua velocidade normal de funcionamento. As falhas de temporização podem ser divididas em duas classes: falhas de transição (*Transition Delay* - TD) e falhas de caminho (*Path Delay* - PD). TD ocorrem em um nó específico do CI causando um atraso maior do que o período de *clock* especificado. PD levam em consideração os atrasos presentes em todo o caminho de lógica e, dessa forma a falha de atraso é vista como atrasos de menor duração porém distribuídos ao longo do caminho. Nesse caso, a soma dos atrasos é responsável pela falha do CI [1]. Neste cenário, o interesse e desenvolvimento de novas metodologias de teste ou de tolerância a falhas vêm aumentando rapidamente quando consideram-se CIs projetados em tecnologias abaixo de 65nm. A definição da técnica adequada requer a identificação dos tipos de defeitos, e consequentemente do modelo de falha, com uma maior probabilidade de ocorrência durante o processo de manufatura do CI. Assim, a avaliação final do circuito quanto à robustez é classicamente feita através do uso de Técnicas de Injeção de Falhas (TIFs). Em outras palavras, faz-se necessário a utilização de instrumentos e ferramentas específicos para o propósito de injetar falhas, criar defeitos ou erros, e monitorar seus efeitos [3].

TIFs podem ser classificadas em *hardware*, *software*, simulação e emulação. TIFs por *hardware* podem ainda ser classificadas em com contato ou sem contato. TIFs por *hardware* com contato, por exemplo, são capazes de avaliar com fidelidade o comportamento do sistema alvo podendo injetar variações de tensão nos pinos de alimentação. Já as TIFs por *harware* sem contato podem utilizar, por exemplo, a geração de interferências eletromagnéticas para causar falhas no sistema. TIFs por *software* modificam o código do sistema de modo a causar uma falha quando este for acessado pelo *hardware*. TIFs por simulação utilizam modelos de alto nível, geralmente descritos em VHDL, para injetar falhas nos estágios iniciais de desenvolvimento do projeto. Injeção de falhas baseada

em emulação consiste em uma abordagem extremamente adequada para injeção de um grande número de falhas, uma vez que, possibilita a utilização da rapidez do *hardware* com a flexibilidade da simulação sobre um modelo do sistema real [3].

A fim de acelerar o processo de injeção de falhas, nos últimos anos técnicas de injeção de falhas baseadas em emulação tornaram-se amplamente utilizadas. Esses métodos utilizam *Field-Programmable Gate Arrays* (FPGAs) para realizar o protótipo do CI sob teste e incluir mecanismos de injeção de falhas. Na emulação em FPGA, um circuito programável é usado para substituir o simulador. Em mais detalhes, o CI é implementado usando o FPGA, o qual pode emular o comportamento do SoC à nível de registradores e portas lógicas. Assim, ao invés de utilizar um simulador em *software*, um circuito de *hardware* que, por sua vez, é muito mais rápido, é utilizado. Os mecanismos para injeção de falhas e controle do processo como um todo são mais complexos. Entretanto, o aumento da velocidade torna possível a análise de um circuito real em um curto intervalo de tempo [7]. Além disso, TIFs baseadas na emulação do SoC em um dispositivo programável são bastante empregadas devido a facilidade de reconfiguração dos experimentos e ao satisfatório nível de observabilidade. Com isso é possível emular diversos tipos de falhas e explorar em paralelo, a velocidade de execução do *hardware* com a capacidade de observar o comportamento real do circuito na presença de falhas.

Neste contexto, esta dissertação de mestrado propõe a especificação, implementação, validação e avaliação de uma metodologia de injeção de falhas para SoCs de alta complexidade. A técnica proposta neste trabalho caracteriza-se por ser um mecanismo de injeção de falhas baseado em emulação, uma vez que é diretamente implementada em um FPGA. Em mais detalhes, esta dissertação de mestrado visa o desenvolvimento de uma metodologia para injeção de falhas de atraso para análise do comportamento temporal de CIs. Assim, um modelo de tempo quantizado que descreve os atrasos do circuito em intervalos definidos de tempo é definido. Fazendo uso do arquivo que descreve as interconexões das células e seus atrasos durante o processo de instrumentação, a plataforma agrega esses atrasos na emulação do sistema podendo alterar as características de temporização da célula para emular a injeção das falhas. O modelo de quantização aqui descrito baseia-se em um conjunto de ferramentas desenvolvidas na Universidad Carlos III de Madrid em Madri, Espanha, para análise de eventos transientes singulares (*Single Event Transient* - SET). Eventos transientes são causados quando uma partícula atinge uma região sensível do circuito contendo lógica combinacional produzindo um pulso de corrente, tornando-se então uma perturbação da tensão que pode propagar-se através da lógica até atingir a entrada de dados de um ou mais registradores de dados [8]. Convém mencionar que a

plataforma desenvolvida nesta dissertação visa integrar esse conjunto de ferramentas de modo a agregar dois dos modelos de falhas de atraso existentes, ou seja, incorporar ao conjunto de ferramentas existente as falhas de atraso em portas lógicas (*gate delay fault*) e as falhas de atraso em caminhos (*path delay fault*).

A implementação da metodologia proposta baseia-se em registradores de deslocamento encarregados de simular o atraso das células que compõem o CI e suas interconexões, de modo que se tenha uma análise fiel do comportamento temporal do sistema. A arquitetura da injeção de falhas aqui proposta é baseada no uso de sabotadores, responsáveis por alterar o atraso das células no momento da injeção. Assim, quando a injeção de falhas do sistema é ativada, os atrasos originais das células são alterados através dos registradores de deslocamento, dando origem a uma falha no sistema. Para a etapa de validação funcional da técnica é utilizado o arquivo resultante do processo de síntese FPGA. Esse arquivo contém o comportamento original do CI e os componentes responsáveis por emular o atraso e alterar o comportamento do sistema durante a injeção de uma falha, mapeados apenas com os recursos disponíveis no FPGA. A validação do sistema é realizada utilizando o *software* de simulação e verificação de circuitos *Modelsim* da *Mentor Graphics*. Com base nos modelos de falhas alvo os estudos de caso são descritos, levando-se em consideração a quantidade de células com capacidade de injeção de falhas, a duração das falhas e a carga de trabalho sobre o projeto em análise. Concluída a validação, uma avaliação da metodologia frente a sua capacidade de injeção de falhas e *overhead* de área é realizada. Note que devido ao fato da técnica prever o uso de um protótipo em *hardware*, a velocidade do experimento de injeção de falhas e o número de falhas injetadas é significativamente maior quando comparada a técnicas de injeção de falhas baseadas em simulação. Finalmente, a metodologia apresentada permitirá avaliar a robustez de sistemas integrados complexos frente a falhas de atraso e calcular a cobertura de falhas de técnicas de teste e tolerância a falhas.

1.1 Objetivos

Este trabalho de mestrado tem como principal objetivo a especificação, implementação, validação e avaliação de uma metodologia baseada em emulação para a injeção de falhas de atraso em SoCs. Em mais detalhes, a abordagem proposta é voltada para injeção de falhas de atraso em portas lógicas, bem como em caminhos, utilizando emulação em FPGA. Para isso torna-se necessário que o CI seja caracterizado em uma linguagem de descrição de *hardware* (*Hardware Description Language* - HDL) a qual descreva o funci-

onamento do mesmo, a sua concepção e organização, juntamente com as informações de atraso associadas a tecnologia de circuitos integrados de aplicação específica (*Application-Specific Integrated Circuit* - ASIC) alvo. Baseado nas informações de variação do processo de fabricação é possível elaborar uma metodologia que reflita a ocorrência de falhas reais no circuito em análise. Com isso é possível realizar um mecanismo na camada de abstração de portas lógicas que, seguindo a metodologia proposta, seja capaz de introduzir falhas de atraso em pontos específicos de CI complexos. Abaixo segue a lista de objetivos específicos:

1. Especificação, implementação e validação de uma metodologia de inserção de atrasos em circuito complexos;
2. Obtenção dos pontos mais propensos a falhas de atraso utilizando o arquivo que descreve as interconexões do circuito;
3. Elaboração de um mecanismo que realize a transferência do atraso do circuito para a plataforma de emulação;
4. Modificação do circuito através da inserção do elemento de atraso;
5. Desenvolvimento de uma ferramenta que automatize o processo de injeção de falhas de atraso em portas lógicas e caminhos críticos do CI;
6. Emulação do circuito de injeção em plataforma de desenvolvimento FPGA;

1.2 Organização dos Capítulos

Este trabalho estrutura-se em mais 5 capítulos dispostos da seguinte forma:

- Capítulo 2: Apresenta os fundamentos teóricos que estabelecem alguns conceitos para o absoluto entendimento do trabalho proposto, abordando algumas definições associadas a robustez de sistemas integrados. Além disso, critérios necessários para determinar o nível de robustez de sistemas, a caracterização de falhas, e o escopo onde as mesmas encontram-se nos diversos níveis de abstração de um projeto são descritos. Em seguida, uma classificação geral dos mecanismos de injeção de falhas e sua importância frente a diferentes tipos de aplicações críticas, bem como ferramentas de injeção de falhas disponíveis na literatura são apresentadas.

- Capítulo 3: Apresenta com um maior nível de detalhamento os procedimentos adotados na elaboração de cada uma das etapas que constituem o desenvolvimento da plataforma de injeção de falhas, descrevendo em detalhes as ferramentas empregadas e os algoritmos implementados durante sua realização. São vistos no capítulo as etapas que constituem a síntese de CIs e os arquivos gerados a partir desse processo. A elaboração da biblioteca responsável por produzir uma estrutura análoga, que descreva o comportamento desejado para as células do projeto, baseado no modelo de falhas de atraso alvo é descrita. A instrumentação do circuito que tem por objetivo alterar as características originais das células, o processo de síntese no FPGA e a geração do arquivo de configuração do dispositivo para avaliação da plataforma em *hardware* são apresentados.

- Capítulo 4: Apresenta os métodos utilizados durante a validação funcional da metodologia proposta. Em mais detalhes, esse capítulo tem como principal objetivo demostrar que os atrasos extraídos de um determinado projeto, no contexto de uma biblioteca tecnológica, estão corretamente quantizados e representados em uma estrutura de *hardware* reconfigurável. Neste capítulo, é apresentado também a arquitetura desenvolvida para fornecer a controlabilidade necessária às células com capacidade de injeção, a qual é responsável por controlar o momento e a duração das falhas injetadas durante o funcionamento do circuito.

- Capítulo 5: Apresenta 2 estudos de caso utilizados na avaliação da plataforma, sob a perspectiva de desempenho e funcionalidade durante as principais etapas de elaboração do CI sob avaliação. Além disso, esse capítulo prevê uma análise dos *overheads* de área, e performance da TIF e dos resultados obtidos para cada um dos estudos de caso definidos para este trabalho. Os experimentos tem por objetivo analisar a robustez dos CIs frente a falhas de atraso e avaliar a cobertura de falhas de técnicas de tolerância a falhas.

- Capítulo 6: Apresenta as conclusões relacionadas a cada uma das etapas de desenvolvimento da técnica proposta nesta dissertação de mestrado.

2 Fundamentação Teórica

Este capítulo apresenta os principais conceitos e definições necessários para o completo entendimento deste trabalho de mestrado. Segundo [9], sistemas embarcados são caracterizados por cinco propriedades fundamentais: funcionalidade, desempenho, robustez, custo e segurança. A função de um sistema está relacionada ao que o sistema se destina a realizar e é descrita pela especificação funcional. O comportamento do sistema é aquilo que o sistema faz para implementar suas funções e é descrito através de uma sequência de estados. O estado total de um dado sistema é o conjunto dos seguintes estados: computação, comunicação, armazenamento de informação, interconexões e condição física. A robustez de um sistema de computação é a capacidade de oferecer um serviço que pode justificadamente ser confiável. Dependendo da aplicação pretendida para o sistema, a ênfase pode ser colocada em diferentes pontos de robustez. Uma definição alternativa que prove um critério para decidir se um serviço fornece determinado nível de robustez é sua habilidade de evitar falhas no serviço, que por sua vez, são mais frequentes e severas que o aceitável. Assim, robustez (*dependability*) é um conceito integrador que engloba os seguintes atributos [9]:

- Disponibilidade (*availability*): leitura correta do serviço;
- Confiabilidade (*reliability*): continuidade do serviço correto;
- Segurança (*safety*): ausência de consequências catastróficas no ambiente;
- Integridade (*integrity*): ausência de alterações indevidas da informação;
- Manutenção (*maintainability*): capacidade de sofrer reparações e evoluções.

Neste contexto, surgem os conceitos de falha (*fault*), erro (*error*) e falha do sistema (*failure*). A falha (*fault*) do serviço é um evento que ocorre quando o serviço prestado desvia do funcionamento correto. Um serviço falha ou porque não está de acordo com a especificação funcional, ou porque esta especificação não descreve adequadamente a função

do sistema. Uma falha no serviço é uma transição do serviço correto para o incorreto, isto é, a não execução da função do sistema. O período de prestação de serviço incorreto é uma interrupção do serviço. A transição do serviço incorreto para o serviço correto é chamado de restauração do serviço. O desvio do serviço correto pode assumir diferentes formas que são chamados de modos de falha de serviço e são classificados de acordo com a severidade da falha. Uma vez que um serviço é uma sequência de estados externos do sistema, uma falha no serviço significa que pelo menos um estado (ou mais) externos ao sistema desviam do estado correto do serviço, e o desvio desse funcionamento é chamado de erro (*error*) [10].

Na maioria dos casos uma falha (*fault*) primeiramente causa um erro (*error*) no estado do serviço de um componente que é uma parte do estado interno do sistema e o estado externo não é imediatamente afetado. Por este motivo, a definição de um erro (*error*) é a seguinte: A parte do estado total do sistema, que pode levar à falhas (*failure*) nos serviços subsequentes. É importante notar que muitos erros não atingem o estado externo do sistema a ponto de causar uma falha no sistema (*failure*). A falha (*fault*) está ativa quando ela causa um erro, caso contrário, é considerada latente. Quando a especificação funcional de um sistema inclui um conjunto de várias funções, a falha de um ou mais dos serviços que desempenham as funções podem deixar o sistema num modo degradado que ainda apresenta um subconjunto de serviços necessários para o utilizador. É possível dizer então que o sistema sofreu uma falha (*failure*) parcial de sua funcionalidade ou desempenho [10].

Diversos meios para se atingir os atributos de robustez têm sido desenvolvidos ao longo dos anos. Esses meios podem ser agrupados em quatro categorias [9]:

- Prevenção de falhas (*fault prevention*): meios para prevenir a ocorrência ou introdução de falhas;
- Tolerância a falhas (*fault tolerance*): meios para evitar falhas de serviço na presença de falhas;
- Remoção de falhas (*fault removal*): significa reduzir o número e a gravidade das falhas;
- Previsão de falhas (*fault forecasting*): meios para estimar o número atual, a incidência futura, e as prováveis consequências de falhas.

A prevenção e a tolerância a falhas buscam prover meios para entregar um serviço confiável. Já a remoção e a previsão de falhas buscam alcançar confiança nesta habilidade justificando que as especificações de robustez, funcionalidade e segurança são adequadas e que o sistema justificadamente tem capacidade de atender essas especificações [9].

A tolerância a falhas como medida para garantir a robustez de sistemas, pode ser mais amplamente descrita através de uma série de técnicas que auxiliam na composição de um sistema tolerante a falhas. Algumas das técnicas que possibilitam a obtenção de sistemas tolerantes a falhas são: Mascaramento de falhas, detecção de erros, diagnóstico de erros, recuperação de erros [9].

Remoção de falhas durante a fase de desenvolvimento do sistema consiste em três etapas: verificação, diagnóstico e correção. Verificação é o processo de investigar se o sistema adere às propriedades especificadas, chamada de condições de verificação. Se o sistema não corresponde às especificações, então é necessário realizar mais duas etapas, diagnosticar as falhas que impedem as condições de verificação sejam alcançadas, e então realizar as correções necessárias. Verificar as especificações é geralmente referenciado como validação. Falhas não cobertas na especificação do sistema podem ocorrer em qualquer estágio do desenvolvimento, seja durante a própria fase de especificação, ou durante fases subsequentes quando evidências são encontradas que o sistema não irá implementar alguma de suas funções. Remoção de falhas durante o uso do sistema pode ser caracterizado como manutenção corretiva, a qual visa remover falhas que produziram um ou mais erros e foram reportadas com êxito, ou preventiva, a qual visa revelar e remover falhas antes que estas causem erros durante a operação normal do sistema [10].

A previsão de falhas é conduzida através da realização e experimentação do comportamento do sistema em relação à ocorrência ou ativação de falhas. A previsão de falhas pode ser vista sobre dois aspectos: avaliação qualitativa, a qual visa identificar, classificar, e ranquear os modos de falha, ou a combinação de eventos (falhas de componentes ou condições de ambiente) que levariam a falhas no sistema; avaliação quantitativa, ou probabilística, a qual visa avaliar em termos de probabilidades o valor numérico em que alguns dos atributos são satisfeitos. Os métodos para avaliação quantitativa e qualitativa podem ser específicos, ou podem ser utilizados para realizar ambas as formas de avaliação. As duas abordagens principais para previsão de falhas utilizando análise probabilística, visando derivar estimativas probabilísticas, são modelamento e experimentação (avaliação). Estas abordagens são complementares uma vez que a modelagem necessita de dados sobre os processos básicos modelados (processo de falha, o processo de manutenção, o processo

de ativação do sistema, etc), que podem ser obtidos quer por experimentação, ou pelo processamento de dados de falhas [10].

2.1 Escopo de Falhas

Conforme anteriormente definido, uma falha é um evento que ocorre quando o serviço prestado desvia do funcionamento correto [10]. Em detalhes, uma falha e suas consequências podem ser observadas em diferentes níveis de abstração de um projeto, a começar pelo nível mais atômico até o mais abstrato [11]. Neste sub-capítulo serão apresentados os níveis de abstração e citadas algumas causas e técnicas que ocasionam ou impedem a transição entre esses níveis.

Impedimentos à robustez (*dependability*) podem ser vistos sob seis níveis de abstração diferentes [11]:

- Nível de defeitos ou nível de componentes, lidando com desvios do padrão em partes atômicas.
- Nível de falha ou nível lógico, lidando com alterações dos valores de sinais ou seleções de caminho.
- Nível de erro ou nível de informação, lidando com dados ou estados internos divergentes.
- Nível de mau funcionamento ou nível de sistema, lidando com o comportamento funcional divergente.
- Nível de degradação ou de nível de serviço, lidando com o desempenho divergente.
- Nível de falha de sistema, lidando com saídas ou ações que divergem do funcionamento normal.

Afirma-se que todos estes níveis são úteis, no sentido de que as técnicas que comprovadamente aumentam a robustez do sistema podem ser aplicadas em cada um deles. Segundo [12], um sistema pode estar em um dos sete estados: Ideal (*ideal*), defeituoso (*defective*), com falha (*faulty*), errôneo (*erroneous*), mau funcionamento (*malfunctioning*), degradado (*degraded*), ou sistema falho (*failed*). Inicialmente, um sistema pode começar em qualquer um dos sete estados, dependendo da conveniência e rigor dos esforços de validação. Uma vez no estado inicial, o sistema se move de um estado para outro, como

resultado do desvio de um padrão ou norma, ou da restauração da condição natural ou apropriada. O desvio de um padrão ou norma leva o sistema a um estado inferior (menos desejável), enquanto que a restauração da condição apropriada através de alguma medida permite que um sistema faça a transição para um estado mais elevado.

A figura 1 mostra que a observabilidade do estado do sistema (facilidade de reconhecimento externo que o sistema está em um estado particular) aumenta a medida que o nível de abstração do sistema também aumenta. A inferência de que um sistema é "ideal" só pode ser feita através de técnicas de provas formais, uma proposição que é impraticável para sistemas de computação modernos devido a complexidade dos mesmos. No outro extremo, um sistema falho pode ser geralmente reconhecido com pouco ou nenhum esforço, sendo uma prática comum forçar um sistema para um estado mais baixo (por exemplo, a partir de um estado falho para um estado errôneo, utilizando injeção de falhas), a fim de deduzir o seu estado inicial [12].

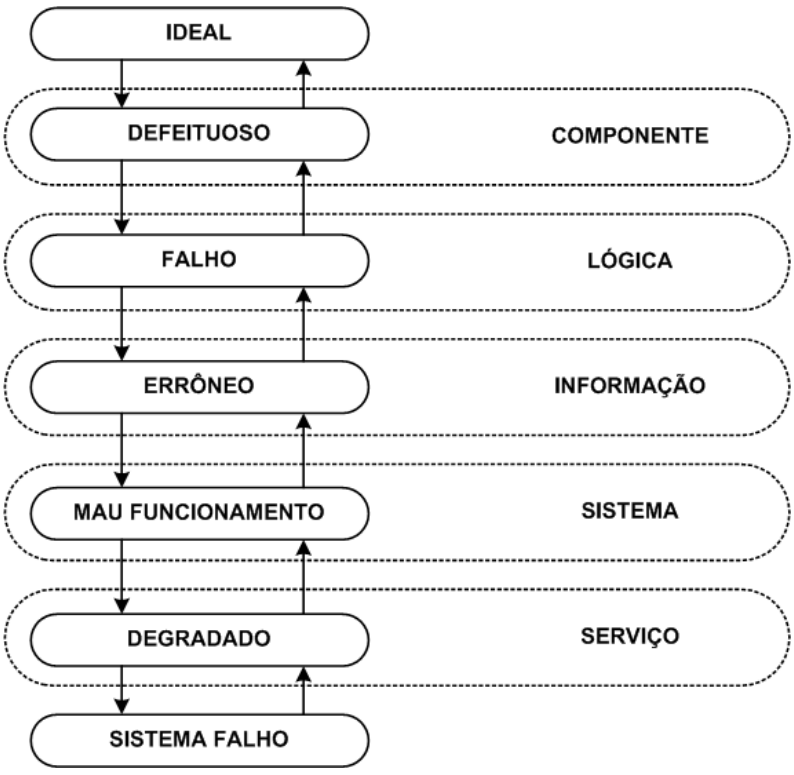


Figura 1: Os estados do sistema e suas transições envoltos pelos níveis de abstração aos quais se relacionam.

Erros no projeto físico podem resultar em componentes defeituosos no sistema, pela concepção inadequada ou até mesmo pelo surgimento de defeitos decorrentes do desgaste dos componentes, envelhecimento, e das condições operacionais mais severas do que inicialmente previstas. Um defeito pode estar dormente ou ineficaz por muito tempo após a sua ocorrência. Durante este período de dormência, a detecção externa do defeito é

impossível ou, pelo menos, extremamente difícil. Apesar dos esforços para evitar ou remover, defeitos que podem estar presentes em um produto normalmente nada é feito até que os mesmos se transformem em falhas.

Um sistema faz a transição a partir de um estado defeituoso (*defective*) para um estado falho (*faulty*), quando um defeito dormente é acordado e dá origem a uma ou mais falhas. Projetistas tentam impedir esta transição, fornecendo margens de segurança adequadas para os componentes ou empregando métodos de tolerância a defeitos. Ironicamente, pode-se ocasionalmente tentar facilitar esta transição para o propósito de expor defeitos, uma vez que as falhas (*faults*) são mais facilmente observáveis do que os defeitos (*defects*). Falhas baseadas em defeitos (*defects*) podem ser classificadas de acordo com a duração (permanente, intermitente/recorrente, ou transitória), extensão (local ou distribuída) e efeito (dormente ou ativa) [12].

Detecção de falhas e técnicas de remoção são partes integrantes de toda a concepção lógica e metodologias de implementação. A transição de um estado falho (*faulty*) a um estado errôneo (*erroneous*) ocorre quando uma falha afeta o estado de algum elemento de armazenamento ou de saída. Projetistas tentam impedir essa transição usando métodos de tolerância a falhas. Outra abordagem é a de controlar esta transição, de modo que esta seja conduzida a um estado incorreto, porém seguro. Pode-se também tentar facilitar esta transição para o propósito de expor as falhas do sistema, uma vez que os erros são mais facilmente observáveis do que falhas, sendo este precisamente o objetivo dos métodos de injeção de falhas [11].

Padrões de entrada especiais são aplicados ao circuito ou sistema em teste, enquanto se observa possíveis erros nas saídas ou estados internos. Um erro é qualquer desvio do estado de um sistema a partir do estado de referência, tal como definido pela sua especificação. Um sistema passa do estado errôneo (*erroneous*) ao estado de mau funcionamento (*malfunctioning*) quando um erro (*error*) afeta o comportamento funcional de algum subsistema. Esta transição pode ser evitada empregando técnicas de tolerância de erro, como recuperação a nível de sistema, redundância de recursos e diversidade de projetos. A detecção do mau funcionamento (complementada por um mecanismo de recuperação) constitui a principal estratégia no projeto de sistemas de computadores confiáveis de hoje. No caso de detecção do mau funcionamento em *hardware*, a unidade pode ser isolada do resto do sistema. Isto reduz a quantidade de recursos de *hardware* (memória, poder de processamento, largura de banda, etc) disponíveis para os cálculos, levando a uma degradação do desempenho correspondente. A disponibilidade de recursos de *backup*

podem adiar essa transição, assim como a capacidade de substituir ou reparar os módulos defeituosos, sem a necessidade de desligar ou perturbar o sistema [11].

Considerando que o desempenho geral do sistema possa estar degradado, usuários individuais ou processos não precisam experimentar o mesmo nível de degradação do serviço (por exemplo, devido à reavaliação de prioridades). Na verdade, quando a degradação é grave, somente processos críticos podem ser autorizados a executar, e somente os processos mais críticos podem ser agendados para a recuperação e execução continuada, enquanto outros processos poderão receber suporte reduzido ou nenhum suporte (com as devidas advertências). Em muitos casos, pode ser possível escrever os programas de aplicação, de tal modo que a sua conclusão bem sucedida não dependa da disponibilidade de qualquer recurso particular. Um sistema falho é percebido quando a capacidade do sistema de tolerância à degradação é esgotada e, como resultado, o seu desempenho cai abaixo de um limite aceitável [11].

2.2 Modelos de Falhas

Para avaliar de forma eficaz a qualidade de metodologias de testes, bem como a eficiência de abordagens de *Built-In-Self-Test* (BIST), são necessários modelos de falhas. O primeiro requisito de um bom modelo de falha é que o mesmo reflita com precisão o comportamento dos defeitos reais que podem ocorrer durante o processo de fabricação, bem como o comportamento de falhas que podem ocorrer durante a operação do sistema. A segunda exigência de um bom modelo de falha, e tão importante quanto a primeira, é que o mesmo deve ser computacionalmente eficiente no que diz respeito ao ambiente de simulação de falhas [13]. Estas duas exigências estão, muitas vezes, em oposição uma a outra. Atualmente, os modelos de falha mais utilizados incluem falhas em nível de portas lógicas, transistores, entre outros. Todos esses modelos de falha podem ser emulados em um ambiente de simulação e, na sua maior parte, aproximam-se bastante ao comportamento de defeitos e falhas reais que podem ocorrer durante o processo de fabricação ou em campo. A fim de detectar um dado modelo de falha, faz-se necessário compreender o comportamento do circuito, uma vez que o circuito defeituoso deve produzir um erro em uma das suas saídas primárias para que o defeito seja detectado [13].

Requisitos de tempo agressivos em projetos de alta velocidade introduziram a necessidade de testar falhas de atraso menores e falhas distribuídas causadas por variações estatísticas no processo de fabricação. O objetivo do teste de atraso é detectar defeitos

de tempo e garantir que o projeto atende as especificações de desempenho para o qual foi projetado. A necessidade de testes de atraso evoluiu a partir do problema comum enfrentado pela indústria de semicondutores: projetos que funcionam corretamente em baixas frequências de *clock*, mas falham na velocidade nominal. Diversos experimentos mostram que testes que não visam especificamente as falhas de atraso têm um sucesso limitado na detecção de defeitos de atraso. A crescente necessidade de testes de atraso é um resultado dos avanços na tecnologia VLSI e do aumento da velocidade dos projetos. Com a introdução da tecnologia sub-mícron, os efeitos do ruído estão se tornando um componente significativo nas falhas de atraso [14].

Na literatura, o modelo de falha *stuck-at* é um dos mais explorados e conhecidos, podendo ser modelado como uma única linha dentro do circuito lógico tendo assumido um valor lógico constante independente das entradas aplicadas. Uma falha *stuck-at-0* ou *stuck-at-1* pode resultar de várias classes de defeitos físicos (conexão aberta, por exemplo, ou de curto-circuito), dependendo da tecnologia. Alternativamente, segundo [1], esta situação pode ser descrita dizendo que o sinal vai levar uma quantidade infinita de tempo para subir do nível lógico 0 ao nível lógico 1. Assim, uma falha *stuck-at* é uma falha de atraso infinito e, de fato, um circuito que passa em testes de *stuck-at* é provável que não tenha qualquer falha de atraso infinito [1]. Para os sistemas digitais que trabalham em qualquer velocidade considerável, isso não é suficiente. A operação de tais sistemas é geralmente sincronizada por sinais de relógio e é necessário que todos os elementos lógicos combinatórios atinjam um estado estacionário após um determinado período de relógio especificado. Algumas entradas e saídas podem ser variáveis de estado ligadas aos *flip-flops* (FF) e outras podem ser entradas e saídas primárias. Todas as alterações da entrada são sincronizadas com um sinal de relógio e todas as saídas, como pode ser visto na figura 2, devem atingir o seu estado lógico final constante dentro de um período de relógio após a mudança das entradas. Assim, para um funcionamento correto o atraso da lógica combinacional não deve exceder o período de relógio [1].

Segundo [14], três modelos de falha de atraso podem ser consideradas: Falhas de atraso por transição (*transition delay fault*), falhas de atraso em portas lógicas (*gate delay fault*) e falhas de atraso em caminhos (*path delay fault*). Assume-se que cada porta lógica tem um atraso arbitrário de queda (ou ascensão) de cada pino de entrada para o pino de saída. As interconexões também possuem atrasos de queda (ou ascensão) arbitrários. Modelos de atraso de transição e portas lógicas são usadas para representar as falhas de atraso concentrados na lógica combinacional, enquanto no modelo de falhas de atraso de caminho os atrasos estão distribuídos por várias portas lógicas através de elementos sequenciais

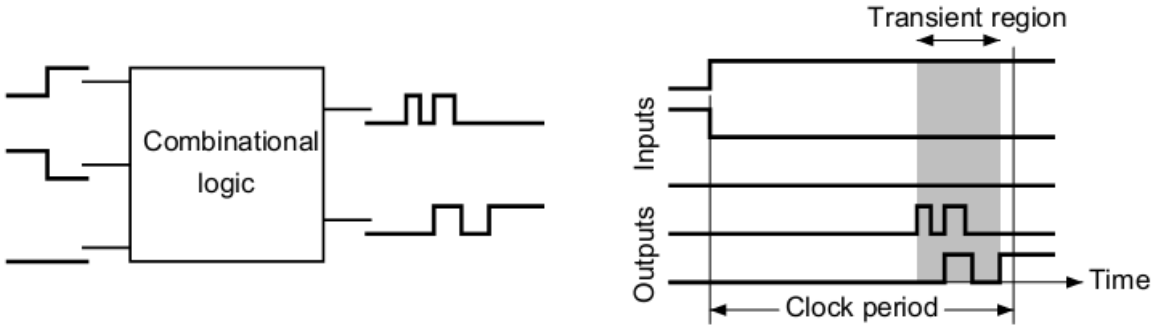
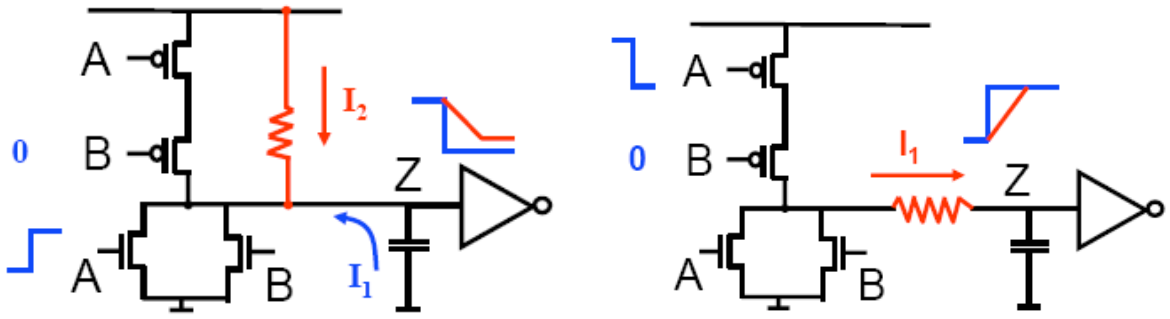


Figura 2: Uma falha de atraso ocorre quando um ou mais sinais excedem o período de *clock* determinado do sistema [1].

pertencentes ao caminho.

Modelos de falha de transição assumem que o erro de atraso afeta somente uma porta lógica no circuito. Existem duas falhas de transição associados a cada porta lógica, falhas do tipo *slow-to-rise*, onde o sinal demora a alterar o estado lógico de 0 para 1 e *slow-to-fall* onde o sinal demora a alterar o estado lógico de 1 para 0. Na figura 3(a) é possível observar que o defeito (*resistive bridge*) causado no circuito faz com que uma transição de subida em A seja percebida com atraso na transição do sinal de saída do nível lógico 0 para o nível lógico 1. Entretanto, a transição do nível lógico 1 para o nível lógico 0 é acelerada. Na figura 3(b) observa-se um outro tipo de defeito (*resistive open*) responsável por causar um retardo na transição do sinal que, por sua vez, causa um atraso em ambas as bordas de transição do sinal de saída [2]. Supõe-se que, no circuito livre de falhas cada porta tem um determinado atraso nominal. Falhas de atraso resultam em um aumento ou diminuição desse atraso. Sob o modelo de falha de transição o atraso adicional causado pela falha é assumido ser suficientemente grande para impedir a transição de alcançar qualquer saída primária no momento da amostragem. Além de ser um modelo para falhas de atraso, modelos de falhas de transição também podem ser utilizados como um modelo para teste de falhas *stuck-at* em circuitos CMOS. Para detectar uma falha de transição em um circuito combinacional, é necessário aplicar dois vetores de entrada (v_1) e (v_2). O primeiro vetor (v_1) inicializa o circuito, enquanto que o segundo vetor (v_2) ativa a falha e propaga o seu efeito para as saídas primárias. O vetor (v_2) pode ser encontrado usando ferramentas de geração de falha de teste, e é durante a aplicação deste segundo vetor que a falha se comporta como uma falha do tipo *stuck-at* [14].

A suposição de que a falha afeta apenas uma porta lógica no circuito não é suficientemente realista. Uma falha de atraso pode afetar mais de uma porta lógica e, embora nenhuma das falhas individuais de atraso possa ser grande o suficiente para afetar o



(a) *Resistive bridge.* (b) *Resistive open.*

Figura 3: Defeitos de fabricação que podem causar violações de tempo [2].

desempenho do circuito, várias falhas juntas podem resultar em uma degradação do desempenho. O modelo de falha de transição anteriormente descrito não pode ser usado para circuitos sequenciais se o relógio for aplicado a uma velocidade nominal, pois o modelo não leva em conta o tamanho do defeito. O modelo de falha de atraso de um circuito sequencial é caracterizado pelo local da falha, tipo de falha e o tamanho do defeito. O tipo de falha pode ser *slow-to-rise* ou *slow-to-fall* e o tamanho da falha representa a quantidade de atraso extra provocado pelo defeito. Em circuitos sequenciais, tamanhos diferentes de falha irão resultar em diferentes falhas nos estados posteriores do circuito. Modelos de falha de atraso em portas lógicas assumem que a falha de atraso é acumulada em uma única porta lógica no circuito. No entanto, ao contrário do modelo de transição, falhas de atraso em portas lógicas assumem que o incremento no atraso não irá afetar o desempenho independente do caminho de propagação através do local da falha. Assume-se que o local da falha através dos caminhos mais longos pode causar a degradação do desempenho. As limitações do modelo de falha de atraso em portas lógicas são similares àquelas para o modelo de falha de transição. Principalmente, devido à suposição do atraso em uma única porta lógica, um teste pode falhar na detecção de falhas de atraso, resultado da soma de vários pequenos defeitos de atraso. A principal vantagem deste modelo é que o número de defeitos é linear com o número de portas no circuito.

Em modelos de falha de atraso em caminhos, um circuito combinacional é considerado defeituoso se o atraso de qualquer um dos seus caminhos exceder um limite especificado. Um defeito de atraso em um caminho pode ser observado através da propagação de uma transição através do caminho. Portanto, a verificação de uma falha de atraso no caminho consiste de um caminho físico e de uma transição, que será aplicada no início do caminho. O atraso ou o comprimento do caminho representa a soma dos atrasos das

portas e interconexões sobre esse caminho. Testes para o modelo de falha de atraso de caminho podem detectar pequenos defeitos de atraso distribuídos causados por variações estatísticas do processo. Assim, a principal limitação deste modelo de falha é que o número de caminhos no circuito pode ser muito grande, possivelmente exponencial ao número de portas. Por esta razão, testar todas as falhas de atraso de caminho no circuito não é nada prático. Duas estratégias são normalmente utilizadas para selecionar o conjunto de defeitos de atraso de caminho para o teste. Uma consiste em selecionar um conjunto mínimo de caminhos de tal forma que para cada sinal s no circuito contendo o caminho mais longo, s é selecionado para o teste. A outra é selecionar todos os caminhos com atrasos esperados maiores do que um limite especificado [14].

2.3 Técnicas de Injeção de Falhas

A seguir, serão apresentadas as principais técnicas de injeção de falhas propostas na literatura. É importante salientar que essas técnicas são utilizadas fundamentalmente nos seguintes contextos: Análise de robustez de sistemas integrados frente a diferente modelos de falhas; Análise e cálculo da cobertura de falhas de técnicas de teste e de tolerância a falhas definidas a fim de aumentar a confiabilidade de sistemas integrados frente a diferentes modelos de falhas; Evidenciar a importância da escolha correta do mecanismo de injeção de falhas com base nos tipos de falhas que se deseja introduzir, demonstrar um típico ambiente de injeção de falhas e por fim, destacar algumas vantagens e desvantagens das diferentes categorias de injeção de falhas e as ferramentas mais utilizadas em cada categoria, são objetivos deste sub-capítulo.

Injeção de falhas é uma técnica que consiste na realização de experiências controladas onde a observação do comportamento do sistema na presença de falhas é induzida explicitamente pela introdução de escrita (injeção) de falhas no sistema. A injeção de falhas tenta determinar se a resposta do sistema coincide com as suas especificações, na presença de falhas em um intervalo definido. Técnicas de injeção de falhas fornecem uma maneira para a remoção e previsão de falhas [6].

É importante mencionar que sistemas grandes e complexos em conjunto com restrições de tempo fazem a inserção exaustiva impraticável. Portanto, apenas um subconjunto cuidadosamente escolhido de todas as possíveis falhas pode ser analisado. A inserção deve ser controlada de modo que o tipo, a localização, o tempo e a duração de cada falha, ou a distribuição estatística correspondente, são, pelo menos, aproximadamente

conhecidos. Durante cada experimento, o sistema deve ser utilizado com uma carga de trabalho representativa de modo a obter uma resposta realista. O efeito de cada falha inserida é precisamente monitorado e registrado com instrumentação. Experimentos de injeção de falha provêem um meio para compreender como o sistema se comporta na presença de falhas, e tal conhecimento acaba por levar a projetos de melhor qualidade e de maior confiabilidade. A maioria das falhas que ocorrem antes da implantação completa do sistema são descobertas através de testes. Falhas que não são removidas podem reduzir a confiabilidade do sistema quando o mesmo estiver em campo [15].

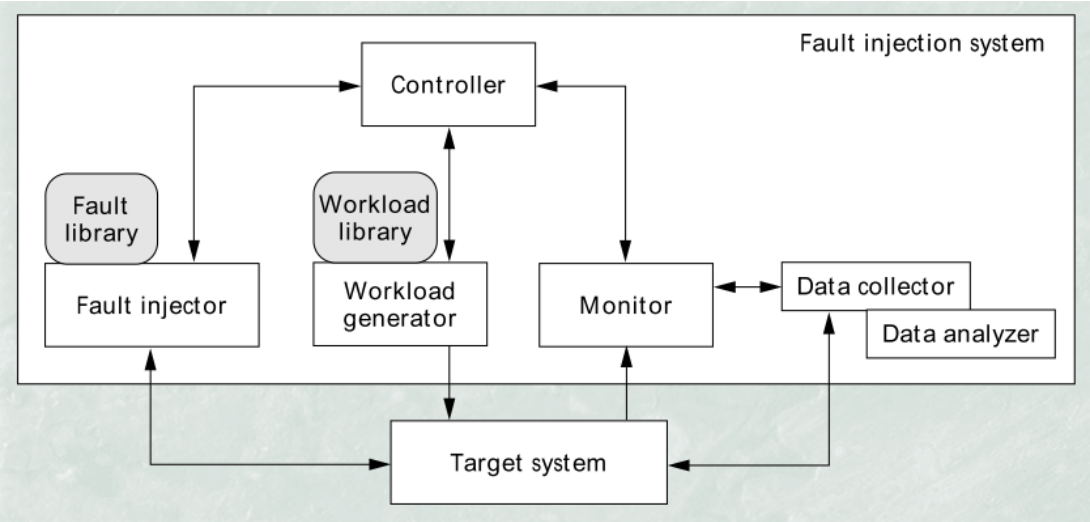


Figura 4: Componentes básicos de um ambiente de injeção de falhas genérico [3].

Na figura 4 é possível observar um ambiente de injeção de falhas genérico, que consiste tipicamente de: Sistema alvo, injetor de falhas, biblioteca de falhas, gerador de carga de trabalho, biblioteca de carga de trabalho, controlador, monitor, coletor de dados e analisador de dados. O injetor de falhas aplica falhas no sistema alvo e executa comandos do gerador de carga de trabalho (aplicações, padrões de referência). O monitor rastreia a execução dos comandos e inicia a coleta de dados quando necessário. O coletor de dados realiza a coleta de dados com o sistema em execução, o analisador de dados, o qual pode ser executado sem a necessidade do sistema estar em operação, realiza o processamento de dados e análise. O controlador governa os experimentos. Fisicamente, o controlador é um programa que pode ser executado no sistema sobre análise ou em um computador a parte. A injeção de falha pode ser personalizada ao *hardware* ou realizada em *software*. O injetor de falhas pode suportar diferentes tipos, locais e durações de falhas, cujos valores podem ser recuperados a partir de uma biblioteca de falhas [3].

2.3.1 Injeção de Falhas Baseadas em *Hardware*

Injeções de falhas baseadas em *hardware* utilizam *hardware* adicional para introduzir falhas no sistema alvo. Dependendo das falhas e suas localizações, os métodos de injeção de falhas implementados em *hardware* podem ser divididos em duas categorias: Injeção de falhas de *hardware* com contato, onde o injetor tem contato direto com o sistema alvo, produzindo variações na tensão ou corrente de alimentação externas ao *chip* do circuito. E injeções de falhas de *hardware* sem contato, onde o injetor não tem contato físico direto com o sistema alvo. Em vez disso, uma fonte externa produz um fenômeno físico, tal como *heavy-ion radiation* ou interferência eletromagnética, causando assim correntes espúrias dentro do *chip* do circuito [6]. Em injeções de falhas com contato as modificações necessárias para a introdução da falha ou para a captura do efeito que estas acarretam podem causar alterações na temporização ou em outras características do circuito, portanto são intrusivas.

2.3.1.1 Benefícios da injeção de falhas por *hardware*:

- Injeções de falhas por *hardware* podem acessar locais de difícil acesso através de outros modos. Por exemplo, o método de *heavy-ion radiation* pode injetar falhas em circuitos VLSI em locais impossíveis de atingir através de outros métodos.
- Esta técnica funciona bem para sistemas que necessitam de grande resolução temporal para disparar ou capturar falhas.
- A avaliação experimental por injeção em *hardware* é muitas vezes a única maneira prática de estimar cobertura e latência de forma realmente precisa.
- Esta técnica é mais adequada em modelos de falhas de baixo nível comportamental (portas lógicas).
- Os experimentos podem ser executados próximos ao tempo real de funcionamento do sistema, permitindo a execução de um grande número de injeções de falhas.
- Executando os experimentos de injeção de falhas no *hardware* definitivo funcionando com o *software* que será executado em campo tem a vantagem de identificar quaisquer falhas de concepção que possam estar presentes no *hardware* e no desenvolvimento do *software*.
- Possui a facilidade de modelar falhas permanentes a nível de pinos do *chip*.

2.3.1.2 Desvantagens na utilização de injeção de falhas por *hardware*:

- A injeção de falhas por *hardware* pode apresentar alto risco de danos para o sistema alvo.
- O alto nível de integração dos dispositivos, circuitos híbridos e tecnologias de encapsulamento cada vez mais densos limitam o acesso à injeção.
- A portabilidade e observabilidade são baixas.
- O tempo de configuração para cada experimento, pode não compensar o tempo ganho pela capacidade de realizar os experimentos em tempo quase real.

2.3.1.3 Principais Ferramentas Disponíveis na Literatura:

- FIST (*Fault Injection System for Study of Transient Fault Effect*): Desenvolvido na Universidade Chalmers de Tecnologia da Suécia, emprega tanto métodos de contato quanto sem contato para criar falhas transitórias no interior do sistema alvo. Esta ferramenta utiliza *heavy-ion radiation*, utilizando o Californium-252 (Cf-252) como fonte de radiação, para criar falhas transitórias em localizações aleatórias dentro de um circuito.

  A injeção de falhas por *heavy-ion radiation* não permite determinar o momento e a duração da falha injetada, portanto quando o *chip* é exposto à radiação pode causar um único ou múltiplos *bit flips*. Devido à impossibilidade de conhecer o local e a duração das falhas a análise de resultados pode ser um problema. Para solucionar este problema pode ser utilizado um *chip* de referência executando em sincronismo as mesmas instruções do *chip* sobre radiação. Comparando a saída de ambos os *chips* a cada ciclo de relógio é possível detectar a ocorrência de erros nos pinos de saída do sistema. A preparação dos experimentos para cada tipo de circuito requer um trabalho considerável e juntamente com isso é necessário realizar uma investigação preliminar da sensibilidade do circuito à radiação. Portanto, a técnica de radiação deve ser utilizada principalmente para circuitos complexos VLSI, tais como microprocessadores, controladores de acesso direto à memória, e semelhantes [16].

- MESSALINE: Um sistema que introduz falhas no nível de pinos desenvolvido no LAAS-CNRS em Toulouse, França. MESSALINE usa ambas as sondas ativas e

soquetes para conduzir injeções de falha no nível de pinos. A ferramenta é composta de quatro módulos, um módulo de injeção de falhas responsável por ativar os elementos de injeção, um módulo de ativação do sistema alvo o qual garante a correta inicialização e carga de trabalho no sistema, um módulo que coleta a reação do sistema após a injeção de falha e um módulo gerenciador de sequência de testes responsável por elaborar os testes com base nos parâmetros de entrada.

O módulo de injeção permite a injeção de até 32 pontos de injeção podendo injetar falhas do tipo *stuck-at*, *open*, *bridge* e falhas lógicas complexas. A ferramenta pode controlar a duração da existência da falha e a frequência com que esta ocorre. A frequência máxima do sinal a ser injetado tem o limite de 10MHz, e o sensor de detecção tem a resolução mínima para sinalizar a ocorrência de um erro em 300ns, portanto erros com duração menor que este valor não são detectados pela ferramenta [17].

- FOCUS: Um ambiente de automação de projeto desenvolvido na Universidade de Illinois em Urbana-Champaign, utilizado para analisar um controlador de motor a jato baseado em um microprocessador usado nas aeronaves Boeing 747 e 757. FOCUS utiliza um ambiente de simulação hierárquica baseada em SPLICE para rastrear o impacto de falhas transientes.

  Um modelo de transição de estado é construído de modo a descrever a propagação de erros entre as unidades funcionais de microprocessador e as distribuições subsequentes nos pinos de E/S. O modelo é utilizado para identificar e isolar os caminhos de propagação de falha críticas, a unidade funcional mais sensível a propagação de falhas, e a unidade com o maior potencial de causar erros nos pinos externos. O processo de injeção de falha é implementado como uma modificação em tempo de execução do circuito, através do qual uma fonte de corrente é adicionada ao nó de destino, alterando assim o nível de tensão do nó ao longo do intervalo de tempo na forma de onda de corrente injetada. O ambiente experimental permite a injeção falhas transitórias e permanentes (simples ou múltiplas).

  As falhas resultantes da injeção podem ser caracterizadas em um transiente de tensão que causa erros em saídas com circuitos sequenciais biestáveis; injeções de falhas que resultam em transientes de tensão causando erros nos pinos de E/S do *chip*; injeções de falhas que resultam em uma alteração das funções de saída de controle do processador; erros que ocorre durante o primeiro ciclo do *clock* na sequência de uma falha transitória; e erros que ocorrem durante o segundo ou subsequentes ciclos

de *clock* [18].

- RIFLE: Um sistema de injeção de falhas no nível de pino para validação de robustez desenvolvido na Universidade de Coimbra, Portugal. Este sistema pode ser adaptado a uma vasta gama de sistemas alvo e as falhas são injetadas principalmente nos pinos do processador. A injeção das falhas é determinística e pode ser reproduzida, se necessário. O injetor de falhas é capaz de detectar circunstâncias particulares em que as falhas injetadas não afetam o sistema alvo e gerar conjuntos de falhas com impacto específico no sistema.

  Falhas de diferentes naturezas podem ser injetadas, a natureza das falhas pode ser definida pelo tipo de falha (*stuck-at*, *bit inversion*, etc), quantidade de pinos afetados durante a injeção e a duração dessas falhas. A idéia principal do RIFLE é combinar o disparo da injeção e técnicas de monitoramento tradicionalmente utilizadas em analisadores lógicos digitais com a lógica necessária para a inserção de falhas no nível de pino do sistema. Uma característica inovadora do RIFLE é a capacidade de definir conjuntos específicos de falhas depois de estabelecidos critérios individuais para a injeção[19].

2.3.2 Injeção de Falhas Baseadas em *Software*

Falhas de *software* são provavelmente a principal causa de quedas dos sistemas. Métodos de injeção de falhas são uma forma de avaliar as consequências de erros encobertos durante o desenvolvimento. Técnicas de injeção de falhas em *software* são bastante atraentes, pois não requerem *hardware* de custo elevado e podem ser utilizadas para injeção de falhas em sistemas operacionais, o que é difícil de realizar com injeção de falhas em *hardware*. Tradicionalmente, injeções de falhas baseadas em *software* envolvem a modificação do *software* de execução do sistema em análise a fim de fornecer a capacidade de modificar o estado do sistema [3].

2.3.2.1 Benefícios da injeção de falhas por *software*:

- Esta técnica pode ser direcionada para aplicativos e sistemas operacionais, o que é difícil de ser feito utilizando injeção de falhas em *hardware*.

- Os experimentos podem ser executados em tempo quase real, permitindo a execução de um grande número de injeções de falhas.

- Executar os experimentos de injeção de falha no *hardware* real que está funcionando com o *software* real tem a vantagem de incluir todas as falhas de *design* que podem estar presentes no *hardware* e no projeto de *software*.
- Não requer qualquer *hardware* de propósito especial; baixa complexidade, baixo nível de desenvolvimento e baixo custo de implementação.
- Nenhum tipo de desenvolvimento ou validação de modelos é necessária.
- Pode ser expandido para novas classes de falhas.

2.3.2.2 Desvantagens na utilização de injeção de falhas por *software*:

- Conjunto limitado de instantes de injeção.
- Não pode injetar falhas em locais inacessíveis ao *software*.
- Exige uma modificação do código fonte para suportar a injeção de falha, o que significa que o código em execução durante a injeção de falha não é o mesmo código que será executado em campo.
- A observabilidade e controlabilidade são limitadas. Na melhor das hipóteses, é possível danificar os registradores internos do processador (bem como locais dentro do mapa de memória) que são visíveis ao programador. Assim, as falhas não podem ser injetadas no *pipeline* do processador ou na fila de instruções, por exemplo.
- Difícil modelagem de falhas permanentes.

É possível classificar os métodos de injeção de *software* com base no momento em que as falhas são injetadas: Durante o tempo de compilação ou durante o tempo de execução. Para injetar falhas em tempo de compilação, a instrução do programa deve ser modificada antes da imagem do programa ser carregada e executada. Ao invés de injetar falhas no *hardware* do sistema alvo, este método injeta erros no código-fonte ou código *assembly* do programa alvo para emular o efeito de *hardware*, *software*, e falhas transientes. O código modificado altera as instruções do programa alvo, causando a injeção. A injeção gera uma imagem de *software* errônea, e quando o sistema executa a imagem, ativa a falha. Em falhas em tempo de execução, é necessário um mecanismo para acionar a injeção de falhas. Os mecanismos de disparo comumente utilizados incluem:

- *Time-out*: A mais simples das técnicas, um temporizador expira em um tempo predeterminado, provocando a injeção. O controlador de tempo pode ser um temporizador por *hardware* ou *software*.

- *Exception/Trap*: Neste caso, uma exceção de *hardware* ou um evento no *software* (*Trap*) transfere o controle para o injetor de falhas. Ao contrário do tempo limite, a exceção pode injetar a falha sempre que determinados eventos ou condições ocorrerem no código.

- Inserção de Código: Nesta técnica, instruções são adicionadas ao programa alvo permitindo que a injeção de falhas ocorra antes das instruções específicas, de modo semelhante aos métodos de modificação de código citadas anteriormente. Ao contrário das técnicas de modificação de código, a inserção de código executa a injeção de falhas durante a execução e adiciona instruções ao invés de modificar as instruções originais [3].

2.3.2.3 Ferramentas Disponíveis na Literatura:

- XCEPTION: Desenvolvido na Universidade de Coimbra, Portugal, usa a depuração avançada e recursos de monitoramento de desempenho existentes na maioria dos processadores modernos para injetar falhas mais realistas por *software*, e para monitorar a ativação das falhas e seu impacto sobre o comportamento do sistema alvo. A aplicação alvo não é modificada, nenhuma armadilha de *software* é inserida, e não há necessidade de executá-la em modo de rastreamento (o aplicativo é executado em plena velocidade).

  Conjuntos de falhas podem ser definidos pelo utilizador de acordo com vários critérios, incluindo a emulação de falhas em unidades funcionais do processador alvo. Falhas pode ser injetadas quando a busca de uma instrução num endereço específico é realizada ou quando os dados armazenados em algum endereço são acessados [20].

- NFTAPE: Uma ferramenta desenvolvida na Universidate de Illinois em Urbana-Champaign, para compor experimentos de injeção de falhas automatizados, a ferramenta ajuda a resolver problemas comuns em algumas ferramentas como a incapacidade de injetar todos os modelos de falha necessárias no modelo em estudo e a dificuldade de portar a ferramentas para novos sistemas. Um dos fatores de motivação para o desenvolvimento do NFTAPE foi o fato de não se encontrar uma ferramenta de injeção automática de falhas que sustentasse o conjunto de recur-

sos necessários para o Laboratório de Propulsão a Jato para avaliar um sistema de computador desenvolvido para executar experimentos científicos no espaço.

NFTAPE é capaz de avaliar sistemas que utilizam uma grande variedade de métricas de confiabilidade, incluindo disponibilidade, confiabilidade, cobertura, a latência do erro, entre outros. Ao invés de fornecer apenas uma métrica genérica para avaliar os efeitos de todas as injeções de falhas, NFTAPE oferece a flexibilidade de avaliação de dados experimentais, utilizando quaisquer métodos de avaliação que sejam mais apropriados para determinado experimento [21].

- GOOFI (*Generic Object-Oriented Fault Injection*): Um sistema projetado para ser adaptável a diferentes sistemas e técnicas de injeção de falhas desenvolvido no departamento de engenharia de computação na Universidade de Tecnologia Chalmers, na Suécia. A ferramenta é altamente portável entre diferentes plataformas, uma vez que depende somente da linguagem de programação Java e um banco de dados SQL compatível.

  GOOFI é capaz de injetar únicas ou múltiplas falhas transientes e sua arquitetura pode ser dividida em uma arquitetura de três camadas. Na camada superior encontra-se a interface gráfica do usuário (*Graphical User Interface* - GUI). A partir dos menus, campanhas de injeção de falhas podem ser configuradas e iniciadas utilizando a técnica de injeção de falhas escolhida. Na camada do meio estão as classes principais em Java que incluem os algoritmos de injeção de falhas, bem como as classes de interface para o sistema alvo. Os algoritmos de injeção de falhas definidas na classe utilizam métodos abstratos que devem ser implementados em cada classe de interface do sistema de destino. O banco de dados e a interface para o banco de dados estão na camada mais baixa. O banco de dados armazena informações sobre o sistema de destino, as campanhas de injeção de falhas e dados registrados a partir dos experimentos de injeção de falhas [22].

- G-SWFIT: Uma nova técnica para gerar falhas de *software* por mutações orientadas introduzidas a nível de código de máquina, desenvolvida na Universidade de Coimbra, Portugal. O método proposto consiste em encontrar estruturas de programação fundamentais a nível do código de máquina, onde falhas de alto nível de *software* podem ser emuladas. A principal vantagem de emular falhas de *software* no nível de código de máquina é que as falhas de *software* podem ser injetadas, mesmo quando o código-fonte do aplicativo de destino não está disponível.

  G-SWFIT consiste em modificar o código binário de módulos de *software* através da

introdução de alterações específicas, que correspondem ao código que seria gerado pelo compilador se a falha de *software* estivesse no código fonte de alto nível. Uma biblioteca de mutações previamente definida para a plataforma de destino orienta a injeção das alterações no código. O código do aplicativo alvo é analisado para determinados padrões de baixo nível de instrução e as mutações são realizadas sobre os padrões para emular falhas relacionadas ao alto nível. Diversas fontes foram analisadas e construiu-se uma lista de *bugs* que podem ser razoavelmente esperados para ocorrer com frequência. Essas falhas são chamadas de "mutações orientadas" para enfatizar que incluem entradas de experiência de dados sobre falhas de *softwares* reais [23].

	Hardware		*Software*	
	Com contato	Sem contato	Compilação	Execução
Custo	Alto	Alto	Baixo	Baixo
Perturbação	Nenhum	Nenhum	Baixo	Alto
Risco de Dano	Alto	Baixo	Nenhum	Nenhum
Resolução	Alto	Alto	Alto	Baixo
Acessibilidade	Pino	Interno	Registradores	Registradores e E/S
Controlabilidade	Alto	Baixo	Alto	Alto
Gatilho	Sim	Não	Sim	Sim
Repetibilidade	Alto	Baixo	Alto	Alto

Tabela 1: Resumo comparativo entre as técnicas de injeção de falhas por *hardware* e por *software* [3].

2.3.3 Técnicas de Injeção de Falhas Baseadas em Simulação

TIFs baseadas em simulação envolvem a construção de um modelo de simulação do sistema em análise, incluindo um modelo de simulação detalhada do processador em uso, se o projeto fizer uso deste recurso. Os modelos de simulação podem ser desenvolvidos utilizando uma linguagem de descrição de *hardware*, como o VHDL. As entradas do modelo são estimuladas por um conjunto de padrões de entrada e em seguida as falhas são injetadas no projeto.

Em injeções de falhas baseadas em simulação duas categorias principais podem ser identificadas, aquelas que necessitam de modificação do código VHDL e aquelas que usam os comandos internos do simulador. Uma primeira abordagem, com base na modificação do código VHDL, altera a descrição do sistema através da adição de componentes de injeção de falha dedicados chamados sabotadores. Já a mutação (alteração) da descrição

de componentes já existentes no modelo VHDL gera descrições de componentes modificados chamados mutantes.

Um sabotador é um componente adicionado ao modelo VHDL com o único propósito de injetar falhas. Este permanece inativo durante a operação normal do sistema, entretanto quando ativo, ou seja, quando uma falha está sendo injetada altera as características de valor ou tempo de um ou mais sinais. Um mutante é um modelo que contém blocos de código dormentes dentro da descrição de portas lógicas normal. Estes blocos de código são ativados por injeção de falhas, alterando o funcionamento da lógica do dispositivo. No entanto, a utilização de mutantes requer que os modelos de portas lógicas originais sejam substituídos pelos novos modelos mutantes.

A principal vantagem deste método é a sua completa independência do simulador adotado, apesar disso proporcionam um desempenho bastante baixo, devido ao elevado custo para a modificação e, possivelmente, recompilação para cada falha a ser injetada. A abordagem por mutantes oferece a maior quantidade de modelos de falhas das técnicas de injeção de falhas apresentadas, sabotadores são geralmente menos eficientes, pois a manipulação de sinais é mais adequada para implementação de modelos de falhas mais simples.

A segunda abordagem utiliza ferramentas de simulação modificadas (comandos internos dos simuladores VHDL), que suportam recursos de injeção e observação. Esta abordagem normalmente proporciona a melhor consistência entre a descrição original do sistema e a injeção de falhas, pois não exige a modificação do código VHDL. As falhas são injetadas alterando os valores dos sinais utilizados para conectar os componentes do modelo VHDL, isto é feito forçando o sinal a um novo valor através de comandos disponíveis no simulador [24].

2.3.3.1 Benefícios da injeção de falhas baseadas em simulação:

- Injeção de falhas simuladas pode suportar diversos níveis de abstração do sistema, elétrico, lógico, funcional e arquitetural. Fornecendo a máxima flexibilidade em termos de modelos de falha suportados.
- Controle total dos modelos de falhas e dos mecanismos de injeção.
- Fornece respostas em tempo útil para os engenheiros de projeto do sistema.
- Os experimentos de injeção de falhas podem ser realizados utilizando o mesmo *software* que será executado em campo.

- Injeções de falhas simuladas podem ser facilmente integradas ao fluxo de *design* já existente.
- Máxima capacidade de observabilidade e controlabilidade. Essencialmente, com um nível de detalhamento suficiente do modelo, qualquer sinal pode ser corrompido da forma desejada, com os resultados das perturbações facilmente observáveis independentemente da localização do sinal corrompido dentro do modelo.
- Permite realizar a avaliação da robustez em diferentes estágios no processo de *design*, bem antes de um protótipo estar disponível.
- Capaz de modelar falhas transientes e permanentes.

2.3.3.2 Desvantagens da injeção de falhas baseadas em simulação:

- Os modelos não estão prontamente disponíveis.
- Esforços de desenvolvimento amplo para realizar o modelo, especialmente em diferentes níveis de abstração.
- Precisão dos resultados depende da fidelidade do modelo utilizado; confiar na precisão do modelo.
- Não é possível injetar falhas em tempo real.
- O modelo pode não incluir algum dos defeitos de concepção que podem estar presentes no *hardware* real.

2.3.3.3 Ferramentas Disponíveis na Literatura:

- MEFISTO (*Multi-level Error/Fault Injection Simulation Tool*): Uma ferramenta de injeção de falhas baseada em VHDL desenvolvida no LAAS-CNRS em Toulouse, França, em parceria com a Universidade de Tecnologia Chalmers, na Suécia. A ferramenta utiliza a vantagem do simulador VHDL e injeta falhas através de comandos do simulador em variáveis e sinais definidos no modelo VHDL. A ferramenta oferece ao usuário uma variedade de modelos de falhas predefinidos.

  A campanha de injeção de falhas consiste em três fases: uma fase de instalação, uma fase de simulação, e uma fase de processamento de dados. Os dois objetivos principais da fase de instalação são: gerar um modelo executável do sistema, e uma lista de experimentos contendo os comandos necessários para controlar o simulador

para cada experimento na campanha. Na fase de simulação, o modelo de simulação criado na fase de instalação em conjunto com a lista de controle do experimento é usado para programar a execução dos experimentos de injeção. A fase de processamento de dados envolve a extração dos resultados do experimento partir dos dados brutos produzidos no decorrer da simulação, e o processamento dos dados da experiência em um formato conveniente, necessários para gerar os resultados experimentais como cobertura de detecção de erros e medidas de latência [24].

- VERIFY (*VHDL-based Evaluation of Reliability by Injection Faults EfficientlY*): Desenvolvido na Universidade de Erlangen-Nurnberg, Alemanha. VERIFY utiliza uma extensão do VHDL para descrever falhas relacionadas a um componente, permitindo que os fabricantes de *hardware*, que fornecem as bibliotecas do projeto, expressem seu conhecimento sobre o comportamento de falha de seus componentes.

  Para a ferramenta VERIFY foi desenvolvido um compilador que é capaz de lidar com as extensões descritas para VHDL e um simulador para a execução dos experimentos de injeção de falhas. Os sinais de injeção de erro são extraídos automaticamente pelo compilador e fornecidos ao simulador que está ligado ao executável. Apenas uma execução de referência (sistemas sem falhas) é realizada. Durante esta simulação *checkpoints* são salvos em alguns pontos no tempo. Esses pontos são usados para injeção de falhas, onde o estado de cada execução defeituosa é comparado dinamicamente contra o estado do percurso de referência simulado em paralelo. Desta maneira é possível avaliar a taxa de falhas no sistema, recuperações e a distribuição do tempo de recuperação relacionado com a localização, tipo, data e duração das falhas injetadas. A extensão proposta para a linguagem VHDL é muito interessante, mas, infelizmente, exige a modificação da linguagem VHDL em si [25].

- HEARTLESS (*Hierarchical Register-Transfer-Level Fault-Simulator for Permanent & Transient Faults*): Um simulador de falhas a nível de abstração entre registradores (*Register Transfer Level* - RTL) para falhas permanentes e transientes desenvolvido pelo grupo CE, BTU Cottbus, na Alemanha. Pode ser utilizado para simular o comportamento de projetos sequenciais complexos como núcleos de processadores em caso de falhas transientes e permanentes. HEARTLESS suporta o uso de construções de alto nível e funções para acelerar a simulação, podendo suportar como formatos de entrada VHDL e ISCAS.

  A ferramenta foi desenvolvida em ANSI C++, sendo de fácil portabilidade entre plataformas que suportem a linguagem. Funções descritas em C e C++ podem ser

utilizadas para substituir grandes blocos VHDL que são uma parte do circuito, mas não são de interesse na simulação corrente. Os modelos de falha suportados nesta abordagem são: *stuck-at-0/1*, falha de transição e *bit-flip* [26].

- FSFI: Um sistema completo de injeção falhas baseada em simulação desenvolvido no Instituto de Tecnologia de Harbin em parceria com a Universidade de Ciência e Tecnologia de Beijing. FSFI foi projetado em cima de um código-fonte aberto do simulador SAM, desenvolvido pela SUN. O sistema simula vários processadores SPARC V9, incluindo CPU, memória, rede e disco rígido. Quatro módulos foram adicionados ao simulador SAM: Injetor de falhas, monitor, analisador, e controlador.

  Falhas transientes são injetadas em diferentes componentes do processador SPARC, como ULA e decodificador de endereços. Os modelos de falha mais utilizados, *stuck-at-0/1* (para falhas permanentes) e *bit-flip* (para falhas transientes), são adotados no FSFI. O registro de falhas contém as seguintes informações: modelo de falha, o componente falho, resultado da execução da carga de trabalho, informações de falhas reportadas pelo sistema operacional, propagações de falhas, e sintomas capturados pelo monitor. O monitor é especialmente projetado para capturar os sintomas a nível de sistema de modo a sinalizar um comportamento anormal do sistema. Ele também rastreia o fluxo de execução e os estados do processador [27].

2.3.4 Técnicas de Injeção de Falhas Baseada em Emulação

De modo a lidar com as limitações de tempo impostas pela simulação e incluir os efeitos devido ao ambiente do circuito na aplicação são utilizados protótipos de *hardware* em FPGA. O circuito sobre análise é implementado em FPGA usando uma síntese clássica, seguindo o fluxo de projeto a partir da descrição do circuito em alto nível até a geração do arquivo de configuração. A placa de desenvolvimento está ligada a um computador hospedeiro, utilizado para definir a campanha de injeção de falhas, controlar os experimentos de injeção e exibir os resultados.

Em geral, a abordagem utilizando o FPGA visa tirar proveito da alta velocidade de execução de um protótipo de *hardware* para reduzir o tempo de injeção de falhas em relação as simulações. Em geral, entradas de controle adicionais e elementos específicos são introduzidos na descrição do circuito em alto nível ou na descrição de portas lógicas de modo que as falhas possam ser injetadas no protótipo. Isso muitas vezes é chamado de "instrumentar" a descrição do circuito.

De modo a evitar qualquer instrumentação do circuito, outra abordagem, chamada de injeção de falhas baseada em reconfiguração em tempo de execução é utilizada. Em vez de injetar as falhas por meio de sinais externos específicos que controlam a lógica adicional, esta abordagem depende da capacidade de reconfiguração dos dispositivos FPGA. Isto significa que a reconfiguração em tempo de execução tem de ser feita para cada falha a ser injetada, no entanto, isso evita o tempo adicional gasto na preparação das versões instrumentadas. A modificação dos arquivos de configuração necessários para realizar as reconfigurações é um processo muito rápido em comparação com a síntese. Além disso, o tempo de reconfiguração globalmente gasto durante a execução de uma campanha de injeção de falhas no emulador de *hardware* (FPGA) pode ser reduzido por meio de uma reconfiguração parcial do emulador quando tal capacidade estiver disponível.

Uma grande vantagem da reconfiguração em tempo de execução é evitar qualquer sobrecarga de *hardware* na injeção de falhas, permitindo que o desenvolvedor execute a emulação em um FPGA com menos recursos lógicos. Além disso, realizar as modificações diretamente no dispositivo reconfigurável pode levar apenas uma fração de segundo, se a reconfiguração parcial puder ser alcançada. Ganhos de tempo tão perceptíveis podem ser esperados com relação a técnicas "clássicas" de injeção de falhas, apesar de uma nova reconfiguração ser necessária para cada configuração de falha a ser injetada [6].

2.3.4.1 Benefícios da injeção de falhas baseadas em emulação:

- O tempo de injeção é imensamente mais rápido em comparação com técnicas baseadas em simulação
- A emulação do sistema permite que o projetista avalie mais precisamente o comportamento que pode ser esperado do circuito em um ambiente real.
- Especialmente interessante no contexto de sistema em um único *chip* (SoC), uma vez que pode conduzir a análise eficiente, mas de baixo custo da robustez de componentes reutilizáveis (mais frequentemente chamados blocos IP), antes de serem utilizadas em um dado circuito.
- O tempo de experimentação pode ser reduzido através da geração parcial ou total dos padrões de entrada no FPGA. Estes padrões já são conhecidos quando o circuito a ser analisado é sintetizado.

2.3.4.2 Desvantagens da injeção de falhas baseadas em emulação:

- A descrição VHDL deve ser sintetizável e preferencialmente otimizada para evitar a necessidade de um emulador demasiadamente grande e para reduzir o tempo total de execução durante a campanha de injeção.
- O custo de um sistema genérico de emulação em *hardware* e a complexidade de implementação em uma plataforma FPGA dedicada a emulação podem ser bastante elevados.
- Descrições algorítmicas ainda não são amplamente aceitas por ferramentas de síntese em fluxos de projeto clássicos, a abordagem empregando a emulação muitas vezes pode ser aplicada somente a partir do nível de descrição entre registradores.
- Problemas com E/S: Ao usar uma placa de desenvolvimento baseada em FPGA, a principal limitação torna-se o número de E/S do *hardware* programável. A quantidade de sinais que podem ser conectados entre o FPGA e o computador hospedeiro pode restringir o número de injeções de falhas, e o número de sinais monitorados.
- Necessidade de uma ligação de alta velocidade de comunicação entre o computador hospedeiro e a placa de emulação, sendo esta uma das partes mais críticas do emulação.

2.3.4.3 Ferramentas Disponíveis na Literatura:

- FADES (*FPGA-based framework for the Analysis of the Dependability of Embedded Systems*): ferramenta para a avaliação prematura e rápida da confiabilidade de sistemas VLSI. Desenvolvida na Universidade Técnica de Valencia, Espanha, FADES controla a placa de prototipagem através dos métodos inclusos no *software* JBits, este *software* consiste em uma série de funções para gerenciar a memória de configuração de dispositivos Virtex. JBits pode ser usado para analisar o arquivo de configuração do dispositivo para obter a quantidade e localização dos elementos combinacionais e sequenciais utilizados na implementação. A plataforma de *hardware* escolhida para demonstrar a viabilidade da abordagem utilizada é um Celoxica RC1000PP, que suporta JBits e possui um FPGA XCV1000.

  FADES suporta um conjunto de modelos de falha como *stuck-at*, *bit-flip*, *bridging*, *indetermination*, entre outros. A configuração do FPGA é lida e analisada para gerar um arquivo refletindo as alterações de modo a emular a ocorrência de falhas.

O sistema executa uma vez sem falha e esta execução de referência é armazenada para posterior comparação. Uma interface simples permite ao usuário comparar a execução de referência com os sinais defeituosos para determinar o efeito global da falha injetada [28].

- FITVS (*Fault Injection Tool for Validating SEE*): A ferramenta utiliza uma placa baseada em FPGA para emulação de sistemas com injeção de falhas. A placa de desenvolvimento utilizada é equipada com uma FPGA Xilinx XCV2000E e dois HY57V641620 SDRAMs. O sistema desenvolvido no Instituto de Tecnologia em Microeletrônica de Beijing, China, não requer a reconfiguração do FPGA e suporta a inserção e observação de comportamentos defeituosos de forma eficaz.

  Os resultados das experiências mostram a qualidade da proteção dos circuitos e aponta os nós mais sensíveis a SEE. As principais características do FITVS são: Um programa C# no PC hospedeiro chamado de gerenciador de injeção que cria a biblioteca de módulos de falha e controla a injeção automática de falhas, e o controlador de emulação da FPGA que possui a função de controlar quando e quais nodos com falha devem ser disparados.

  O método de injeção de falhas do FITVS pode ser classificado como inserção de sabotadores. FITVS adiciona uma porta *XOR* com controle em cada porta lógica básica (ou seja: *NAND*, *NOR*, *NOT*, etc), esta porta lógica será utilizada como controlador de injeção nos nós do circuito. Quando o sinal de controle do nó e injeção é igual a "0", não há injeção de falhas ativa, no entanto, quando o sinal de controle é igual a "1", a falha é ativada e o valor lógico do nó é invertido, emulando assim um SEE [29].

- FuSE (*Fault injection using SEmulation*): Desenvolvida na Universidade de Tecnologia de Viena, Áustria, é uma ferramenta de injeção de falhas que combina o desempenho de um protótipo implementado em *hardware* e a flexibilidade, bem como a visibilidade de uma simulação em HDL. A plataforma de desenvolvimento HMX2-AS2 baseado em FPGA é equipada com duas Altera Stratix II.

  Na emulação o módulo sobre testes é parcialmente (ou totalmente) implementado no FPGA. Assim, as mudanças de sinal dentro da simulação são enviados através da interface PCIexpress para o "Gerenciador de IO", que controla e observa todos os pinos de teste do *hardware*.

  Para o processo real de injeção de falha o projeto sobre testes deve ter acrescentado em sua descrição alguns dispositivos sabotadores, as portas de ativação de

falhas correspondentes, bem como portas de gravação de dados. Cada sabotador está equipado com uma porta de controle diretamente ligada as linhas de ativação correspondentes as falhas a serem inseridas. A ferramenta compreende três dispositivos sabotadores genéricos: um sabotador para lógica *stuck-at-0/1* e *bit-flip*, um sabotador de *bridging* para *AND*, *OR*, e um sabotador que simula falhas de atraso do sinal [30].

	Simulação	Emulação
Custo de implementação	Baixo	Alto
Visibilidade	Total	Limitada
Resolução	Alta	Alta
Acessibilidade	Total	Baixa
Controlabilidade	Alta	Baixa
Repetibilidade	Alta	Alta
Custo de alterações	Baixo	Alto
Injeção de falhas/segundo	Baixa	Muito Alta

Tabela 2: Resumo comparativo entre as técnicas de injeção de falhas por simulação e por emulação.

3 Proposta

Com a miniaturização dos transistores, a densidade dos CIs é incrementada a cada nova geração, e com isso o número e a proximidade entre as linhas de roteamento também é incrementada. Pequenas variações no processo de fabricação, tais como ligação entre trilhas adjacentes, descontinuidade de trilhas, variação da tensão de disparo devido a alterações no processo de litografia podem causar defeitos no circuito final. Assim, novos modelos de falhas são necessários para refletir os possíveis defeitos na nova tecnologia. Alguns dos defeitos resultantes da variação no processo de fabricação não manifestam-se através de uma alteração lógica no comportamento do circuito resultante, e sim em uma alteração na temporização do mesmo. Esse tipo de defeito, com origem na tecnologia submicron, necessita especial atenção dos engenheiros de testes, pois exige que o sistema seja executado na velocidade normal de funcionamento a fim de garantir a detecção desse tipo de falha, bem como garantir que o circuito satisfaça as suas especificações de desempenho.

Neste contexto, uma metodologia de injeção de falhas de atraso para análise do comportamento de CIs complexos é proposta. A metodologia proposta visa orientar a inserção de falhas de atraso em pontos específicos do CI sob análise afim de: (1) verificar o comportamento/robustez de CIs na presença de falhas de atraso, e (2) calcular a cobertura de falhas de metodologias de teste, bem como de técnicas de tolerância a falhas desenvolvidas para detectarem esse tipo de falha. Os pontos de inserção são resultados do estudo de variações probabilística do processo de fabricação de CIs em larga escala e podem ser utilizados na modelagem de falhas de atraso decorrentes dessas variações. Assim, um mecanismo que se utiliza dessas estatísticas pode ser elaborado de forma a estimar o comportamento de um CI complexo na presença de falhas de atraso. Em mais detalhes, o mecanismo proposto utiliza a descrição de portas lógicas do circuito juntamente com o arquivo que descreve o atraso das células que compõem o mesmo, para realizar um estudo dos pontos mais suscetíveis a esse tipo de falha. Entende-se por propensão a falhas de atraso as regiões do circuito com uma menor margem de funcionamento no que diz respeito a variações do período de *clock* utilizado.

Dado o incremento exponencial na complexidade de CIs, o aumento na quantidade de pinos utilizados e a velocidade de funcionamento, inúmeras regiões para análise podem existir. Portanto, é necessário definir alguns critérios na seleção dos pontos do circuito que serão analisados. A plataforma desenvolvida busca fornecer suporte a qualquer mecanismo que pretenda realizar experimentos associados a temporização de uma célula ou a um conjunto de células de um CI que se deseja analisar bastando-se somente, que seja fornecido o conjunto de células gerado a partir dos critérios do algoritmo empregado. Com a plataforma desenvolvida será possível alterar a temporização do circuito adicionando ou subtraindo atrasos às células. Note que a plataforma proposta permite a utilização de valores distintos para bordas de subida e descida de modo a abranger diferentes modelos de falhas de atraso. Além disso, a plataforma proposta procura explorar três aspectos distintos na seleção das células alvo da injeção de falhas: (1) células pertencentes a um caminho crítico; (2) o tipo de função lógica realizada pela célula e (3) o número de entradas lógicas que uma célula é capaz de acionar com precisão (*fanout* da célula).

Convém mencionar que a velocidade de um sistema baseia-se principalmente no seu caminho crítico, ou seja, o caminho lógico entre dois elementos sequenciais com o maior atraso do circuito. O modelo de falhas de atraso em caminho é o mais preciso dos modelos de falhas de atraso, pois modela tanto defeitos distribuídos como localizados. Os caminhos com os maiores atrasos, denominados caminhos críticos, são os mais importantes para os testes de falhas de atraso já que um defeito na temporização do circuito é mais provável que cause uma violação de tempo nestes caminhos [31]. Seguindo esta construção a plataforma busca por informações de caminhos críticos gerados pela ferramenta de síntese para extrair as células que compõem esta estrutura combinacional, de modo a agregar estruturas de testes específicas nestas células.

A extração das células realizadas por função lógica e *fanout* buscam explorar o modelo de falhas de atraso em portas lógicas, o qual assume que o erro de atraso é acumulado em uma porta do circuito. A seleção deste conjunto de células difere da extração por caminho crítico, dado que as células escolhidas não necessariamente pertencerão a um mesmo caminho lógico.

Já a falha de atraso em portas lógicas não assume que o aumento no atraso irá afetar o desempenho independente do caminho de propagação através do local da falha. O modelo de falha de atraso em portas lógicas é um modelo quantitativo, uma vez que leva em conta os atrasos do circuito de modo que os atrasos das portas são representados como intervalos [14]. Assume-se que os caminhos de comprimento mais longo, através do local

da falha, podem efetivamente causar a degradação do desempenho. Neste contexto, esta dissertação apresenta a realização de um modelo de tempo quantizado o qual descreve os atrasos do circuito em intervalos definidos de tempo, fazendo uso do arquivo que descreve as interconexões das células e seus atrasos durante o processo de instrumentação, de modo a agregar o atraso inerente de cada porta lógica levando em conta a sua posição dentro circuito como um todo.

No modelo de falha de atraso em portas lógicas, o atraso através de uma porta depende dos valores lógicos aplicados na mesma. Neste modelo de falhas, várias cópias de uma mesma porta lógica podem ter atrasos diferentes devido a variações de fabricação, e todas as portas lógicas tem certa inércia na resposta a mudanças em suas entradas. Transientes de curta duração, nas entradas da porta lógica são filtrados podendo não se propagarem para a saída. Agregar os valores de temporização ao elaborar os testes para falhas de atraso em portas lógicas permite a aplicação de alguns testes que caso contrário não seriam considerados [14].

Assim, a metodologia proposta utiliza a descrição funcional do circuito que se deseja analisar em linguagem de descrição de *hardware* (HDL). Essa descrição pode então ser utilizada no processo de síntese para fabricação do CI juntamente com as bibliotecas tecnológicas disponíveis. O processo de síntese gera dois arquivos, um contendo a descrição funcional do circuito a nível de portas lógicas mapeadas para uma tecnologia específica e outro, contendo a relação completa dos atrasos destas portas e suas interconexões. Note que após o mapeamento as portas lógicas passam a ser referenciadas como células. As informações expressas neste segundo arquivo serão úteis ao mapear as informações de atraso das células para a plataforma de emulação durante o processo denominado instrumentação. O processo de instrumentação visa extrair e quantizar o atraso das células bem como agregar esses valores na etapa de síntese no FPGA que, por sua vez, realiza a tradução das células para elementos particulares da tecnologia de *hardware* reconfigurável, no caso, recursos disponíveis no FPGA.

A simulação de falhas de atraso em portas lógicas deve levar em conta os atrasos do circuito, de modo a determinar a menor falha de atraso detectada. Simulação de falhas de atraso em portas lógicas consiste em injetar um atraso adicional nos possíveis locais de falha, determinando se esse atraso extra se propaga em termos de novos tempos de chegada e estabilização e, em seguida, se é detectado em uma saída primária ou em uma pseudo saída primária no momento da observação [32].

A arquitetura da injeção de falhas aqui proposta é baseada no uso de sabotadores,

responsáveis por alterar o atraso das células no momento da injeção. A estrutura proposta é descrita em linguagem de descrição de *hardware* composta por registradores de deslocamento encarregados de simular o atraso das células e suas interconexões de modo que se tenha uma análise fiel do comportamento temporal do sistema. Quando a injeção de falhas do sistema é ativada, os atrasos originais das células são alterados através dos registradores de deslocamento dando origem a uma falha no sistema.

Durante o processo de instrumentação todas as células do circuito tem agregadas a sua descrição os valores de atraso quantizados, os quais serão utilizados no processo de síntese no FPGA. Entretanto, para isso é necessário descrever funcionalmente o comportamento que as células terão ao serem transpostas para a plataforma FPGA. Dessa forma, é proposta a criação de bibliotecas funcionais as quais descreverão não só a função lógica da célula mas também a estrutura de injeção de falhas necessária. Com isso, é possível desenvolver uma plataforma flexível onde mesmo com o circuito já instrumentado, é possível redefinir a arquitetura de injeção de falhas agregando diferentes funcionalidades. Com o arquivo do circuito preparado para a instrumentação e as bibliotecas descritas é possível realizar a síntese no FPGA com o objetivo de gerar o arquivo *bitstream* instrumentado.

Para a etapa de validação da plataforma é utilizado o arquivo resultante da síntese lógica. Os arquivo da síntese lógica após ser instrumentado é utilizado juntamente com os valores quantizados dos atrasos e a descrição das bibliotecas como entrada para produzir o arquivo que contém o comportamento original do circuito porém mapeado apenas com os recursos do FPGA. Este arquivo pode então, ser utilizado na validação do sistema utilizando o *software Modelsim* da *Mentor Graphics*. Após a validação pode-se conduzir os testes para o FPGA tirando proveito da alta velocidade de execução da mesma afim de reduzir o tempo de injeção de falhas em relação as simulações. É possível então, através da metodologia empregada, obter uma indicação do nível de robustez do circuito, extrair a cobertura de falhas de metodologias de teste, bem como nortear a inserção de métodos de tolerância a falhas que tem como alvo falhas de atraso.

3.1 Implementação

A plataforma de injeção de falhas de atraso desenvolvida nesta dissertação busca empregar as características associadas a injeção de falhas por emulação tirando proveito da observabilidade e da velocidade de execução obtida através do desenvolvimento utilizando uma plataforma de *hardware* reconfigurável. A plataforma foi desenvolvida utilizando fer-

ramentas de síntese lógica voltadas para a fabricação de ASICs que, por sua vez, foram sintetizados utilizando uma biblioteca tecnológica de 65nm fornecida pela STMicroelectronics. A instrumentação e extração dos atrasos de cada célula do circuito resultante do processo de síntese lógica do circuito original foram realizadas utilizando a linguagem de alto nível TCL (*Tool Command Language*). TCL é uma linguagem de *script* e um interpretador desta mesma linguagem, o interpretador de comandos foi portado do UNIX para diversos ambientes entre eles o DOS e Macintosh portanto um único código executa em diversas plataformas diferentes sem necessidade alguma de adaptação do código. Semelhante a outras linguagens de *shell* do Unix, como o Bourne Shell (sh), o Shell C (csh), e Perl, TCL permite executar outros programas de dentro de suas próprias rotinas além de acessar o sistema de arquivos sem necessidade de extensas compilações [33]. O TCL possui pouca sintaxe, o que o torna uma linguagem de rápida aprendizagem e eficiente na interpretação de comandos, qualidade necessária quando se está lidando com arquivos de interconexões de tamanhos bastante grandes. Além disso, TCL é uma linguagem padrão na indústria de semicondutores para interfaces de programação de aplicativos, muito popular principalmente na indústria de automação de projetos (*Electronic Design Automation* - EDA). As bibliotecas para instrumentação são responsáveis por descreverem a comportamento funcional das células da biblioteca tecnológica é também por agregarem elementos de atraso à emulação do sistema. Ambas as bibliotecas, tanto para instrumentação do comportamento funcional, quanto a biblioteca de injeção de falhas, são descritas em VHDL e são empregadas durante a síntese do FPGA.

A metodologia desenvolvida consiste na utilização de uma descrição em linguagem de *hardware*, a qual sofre o processo de síntese lógica com o objetivo de transformar a descrição comportamental em uma descrição equivalente a nível de portas lógicas. Durante esse processo, informações relacionadas as interconexões e atraso entre portas lógicas são extraídas. Com a descrição das interconexões entre as portas lógicas e os seus atrasos é possível eleger as células que serão utilizadas na instrumentação, apresentando assim, a capacidade de injetar falhas no circuito. De posse de todos os arquivos (descrição das interconexões, atraso das portas lógicas e conjuntos de células escolhidas para injeção de falhas) é dado início ao processo de instrumentação do circuito original, conforme a figura 5.

O procedimento de instrumentação consiste na inserção de entradas de controle e elementos específicos introduzidos na descrição de portas lógicas de modo que as falhas possam ser injetadas no circuito de modo controlado. O resultado do processo de instrumentação é a descrição das interconexões do circuito sem nenhuma alteração porém,

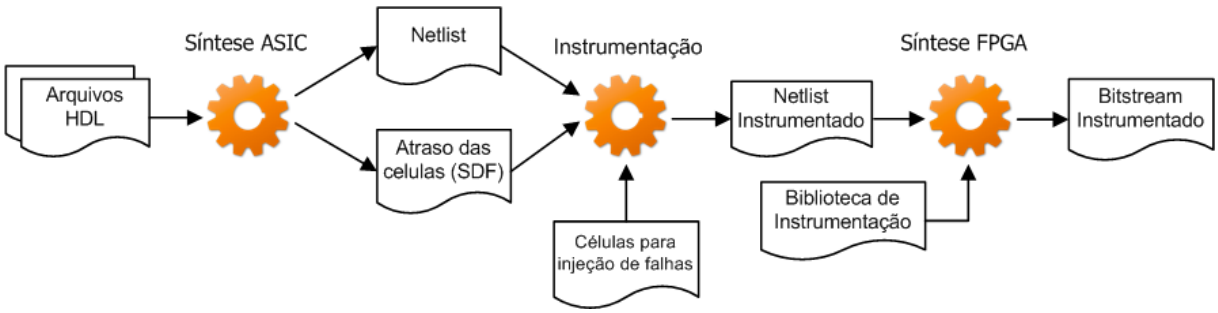


Figura 5: Visão global da plataforma de injeção de falhas de atraso.

com a adição de sinais de controle para cada célula presente na descrição. Essa *netlist* instrumentada, juntamente com o conjunto de bibliotecas que descrevem o mecanismo de instrumentação e habilitam a injeção de falhas, são utilizados no processo de síntese no FPGA a fim de gerar uma descrição funcional do circuito agora mapeado para a tecnologia de *hardware* reconfigurável utilizando portanto, apenas recursos disponíveis no dispositivo. O arquivo proveniente da síntese no FPGA é então utilizado na etapa de validação junto com outros arquivos que descrevem o controle e verificação da execução das campanhas de injeção de falhas.

Nas seções seguintes serão apresentados com um maior nível de detalhamento os procedimentos adotados na elaboração da plataforma de injeção de falhas, descrevendo em mais detalhes as ferramentas empregadas e os algoritmos implementados durante sua realização.

3.2 Síntese ASIC

A primeira etapa na obtenção de uma *netlist* instrumentada é o processo de síntese voltado para a fabricação de Circuitos Integrados de Aplicação Específica (*Application-Specific Integrated Circuits* - ASICs). O fluxo normal de fabricação de CIs de aplicação específica segue uma sequência de etapas que pode ser observada na figura 6.

1. Codificação RTL: O *design* de *chips* começa com a concepção de uma ideia ditada pelo mercado. Essas ideias são então traduzidas em especificações arquitetônicas e elétricas. A fase seguinte envolve a implementação dessas especificações utilizando linguagens de descrição de *hardware* (HDL). Este tipo de codificação é utilizada para descrever as funcionalidades do circuito e durante a síntese para formar uma *netlist* estrutural que compreende os componentes e as suas respectivas ligações a partir de uma biblioteca alvo.

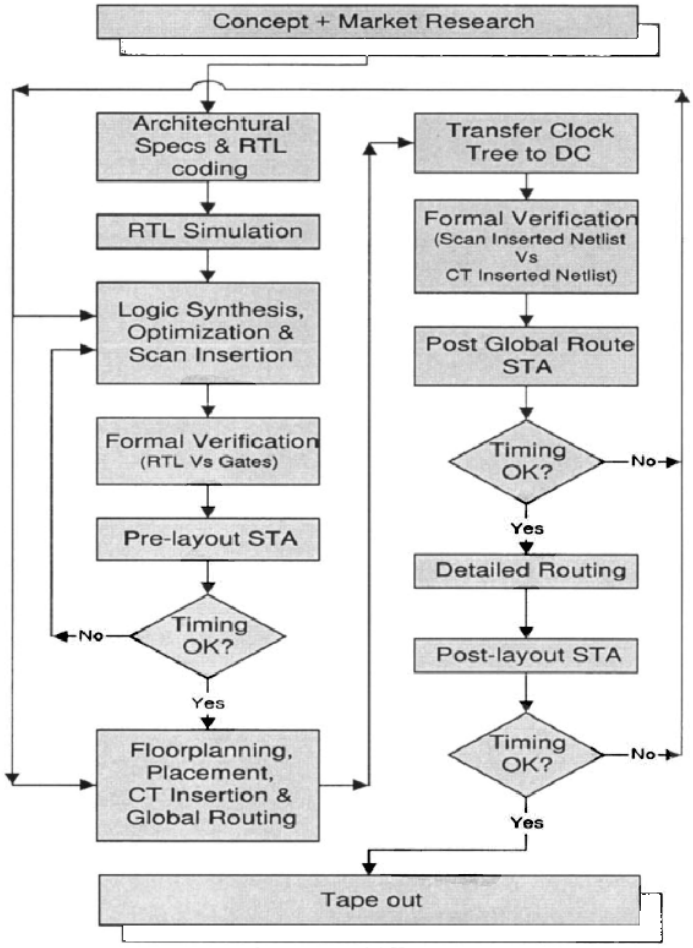


Figura 6: Fluxo típico na elaboração de um ASICs [4].

2. Simulação RTL: O próximo passo é verificar a funcionalidade do projeto simulando o código RTL. Todos os simuladores disponíveis atualmente são capazes de simular o nível comportamental, bem como o nível de codificação entre registradores. Além disso, eles também podem ser usados para simular o projeto a nível de portas lógicas mapeadas para a tecnologia alvo.

3. Inserção de *Scan*: A maioria dos projetos hoje incorporam a técnica de projeto visando testabilidade (*Design for Testability* - DFT) inserindo lógica para testar a funcionalidade do sistema, depois que o *chip* é fabricado. A DFT consiste de lógica e memórias com BIST, varredura de lógica (*scan logic*) e varredura de lógica periférica (*boundary scan*). O processo de inserção das cadeias de registradores, por exemplo, é capaz de mapear o RTL diretamente para os registradores de teste antes mesmo de construir por completo a cadeia de registradores.

4. Restrições (*Constraints*): A síntese de um projeto é um processo iterativo e começa com a definição das restrições de tempo para cada bloco do projeto. Essas restrições

de tempo definem a relação de cada um dos sinais com relação à entrada de *clock*, e a análise estática de tempo tem a função de relatar essas informações de tempo do projeto. A ferramenta de síntese tenta otimizar o projeto para atender as restrições de tempo especificadas, porém, outras medidas (como mudança da frequência de *clock* utilizada) podem ser necessárias se os requisitos de tempo não forem atingidos.

5. Síntese Lógica: A partir de um algoritmo descrito em linguagem de descrição de *hardware* (HDL), o objetivo da síntese é refinar o código implementado, através da transformação de um código em nível de transferência entre registradores (RTL) para o nível de portas lógicas.

6. Verificação Formal: O conceito de verificação formal é relativamente novo para a comunidade de ASIC. Técnicas de verificação formal realizam a validação de um projeto usando métodos matemáticos sem a necessidade de considerações tecnológicas, como a temporização e os efeitos físicos. Eles verificam as funções lógicas de um projeto, comparando-o contra o projeto de referência.

7. Análise Estática de Tempo: A análise estática de tempo (*Static Timing Analysis* - STA) é um dos passos mais importantes no processo de *design* ASIC. Esta análise permite ao usuário analisar exaustivamente todos os caminhos críticos do projeto e expressá-los em um relatório. Além disso, o relatório também pode conter informações de depuração como *fanout* das células utilizadas.

8. *Floorplanning*: A qualidade do *floorplanning* e do posicionamento é mais crítica do que o próprio roteamento. Um posicionamento ideal das células, não só acelera o roteamento final, mas também produz resultados superiores em termos de temporização e redução dos congestionamentos. O arquivo de restrição pode ser usado para indicar o posicionamento das células com base na temporização (*timing driven placement*).

9. *Placement*: O *placement* utiliza a *netlist* do circuito em conjunto com uma biblioteca de tecnologia para produzir um *layout* de posicionamento válido. O método de posicionamento das células com base na temporização, citado no item anterior, obriga a ferramenta de *layout* a colocar as células de acordo com o grau de importância da temporização entre as células. O *layout* é otimizado de acordo com esse propósito, dando início ao redimensionamento das células e posicionamento dos *buffers*, sendo este último um passo essencial para garantir o sincronismo e a integridade dos sinais. Alguns blocos podem ainda ter locais dedicados previamente definidos em um

processo de *floorplanning* anterior, o que limita a etapa de *placement* em atribuir locais apenas para as células restantes.

10. Árvore de *Clock*: Após a colocação das células, a árvore de *clock* é inserida no projeto pela ferramenta de *layout*. O método de inserção da árvore de *clock* é opcional e depende unicamente do projeto e da preferência do usuário. É possível optar por utilizar métodos mais tradicionais de roteamento da árvore de *clock* a fim de reduzir o total de atraso e o desvio (*skew*) do *clock*.

11. Roteamento Global: A ferramenta de *layout*, geralmente executa o roteamento em duas fases, roteamento global e roteamento detalhado. Após o posicionamento, o projeto é completamente roteado para determinar a qualidade do posicionamento e fornecer os atrasos estimados. Se o posicionamento de célula não é otimizado, o roteamento global irá demorar mais tempo para ser concluído, afetando também o tempo total do projeto. Portanto, para minimizar o número de iterações e melhorar a qualidade do posicionamento, as informação de tempo são extraídas a partir do *layout*, após a fase de roteamento global. Embora, estes valores de atraso não sejam tão precisos como os números extraídos após o roteamento detalhado, eles fornecem uma boa noção da temporização depois do circuito completamente roteado.

12. *Tape out*: O projeto então experimenta uma última análise estática de tempo e em seguida passa por uma verificação do *layout* (*Layout Versus Schematic* - LVS) e das regras de verificação do projeto (*Design Rule Checking* - DRC) antes de efetivamente ocorrer o *tape out*. Um arquivo GDSII é fornecido para a fábrica de semicondutores juntamente com alguns detalhes requeridos para fabricação. A fábrica então produz as máscaras de projeto com base nesses arquivos, que em seguida dão origem ao *chip* do sistema.

O processo de síntese lógica adotado nesta dissertação conserva as mesmas etapas de um fluxo de produção típico da indústria de CIs em larga escala. Entretanto, arquivos gerados em estágios anteriores a conclusão do projeto são utilizados de modo a permitir a elaboração de conjuntos de testes em paralelo com a sequência normal de produção. A utilização desse arquivos tem por objetivo acelerar a detecção de possíveis falhas de atraso que possam se manifestar ao longo da utilização do circuito, de modo a prever a margem de funcionamento do circuito em pontos específicos.

O processo de síntese lógica utiliza como entrada três arquivos, são eles: (1) biblioteca tecnológica CMOS de 65nm fornecida pela STMicroelectronics, a qual descreve as células

disponíveis na elaboração da síntese; (2) os arquivos que detalham a funcionalidade do projeto a nível de RTL, geralmente descritos em VHDL ou Verilog, e (3) o arquivo de restrições (*Synopsys Design Constraints* - SDC), o qual descreve as restrições necessárias ao projeto.

As restrições podem ser relacionadas ao projeto ou relativas aos recursos disponíveis e são empregadas pelo projetista em diversas etapas durante a implementação de circuitos VLSI, como síntese lógica, síntese da árvore de *clock*, *placement*, roteamento, e principalmente na análise estática de tempo (STA). As restrições definem quais as alterações as ferramentas podem ou não realizar no projeto ou como a ferramenta deve proceder em determinadas situações. O tempo de transição é uma das principais elementos observados durante o desenvolvimento de projetos VLSI. O tempo de transição de um sinal é o maior tempo necessário para o seu pino de condução alterar os valores lógicos e é decidido com base no tempo de subida e descida baseando-se nos dados da biblioteca tecnológica utilizada. Para o modelo de atraso não linear (*Non Linear Delay Model* - NLDM), o tempo de transição de saída é uma função da transição de entrada e da carga de saída [34]. Outro fator importante durante a análise de tempo é o *fanout* máximo de uma saída, o qual mede a sua capacidade de condução de carga, ou seja, é o maior número de portas de entrada para o qual a saída pode ser ligada de forma segura. Em mais detalhes, o *fanout* de uma célula é o máximo de entradas que pode ser ligada a uma saída, antes da exigência de corrente por qualquer das entradas exceder a corrente que pode ser entregue pela saída ao mesmo tempo, mantendo ainda assim o nível lógico correto [34].

Durante a síntese lógica algumas restrições tiveram que ser elaboradas de modo a compatibilizar seu funcionamento com as bibliotecas de instrumentação previamente descritas. Dentre as restrições formuladas estão alguns comandos para restringir as células que serão utilizadas na elaboração do circuito final de modo a fornecer essa compatibilização do arquivo gerado a nível de portas lógicas com as portas lógicas instrumentadas (descritas na biblioteca VHDL) que serão substituídas na etapa de síntese no FPGA. Dada a natureza das falhas que estão sendo alvo de análise, o sistema deve ser síncrono, não pode possuir múltiplas frequências de *clock*, técnicas de *clock gating*, *tristates* ou portas bidirecionais.

Como resultado da síntese lógica diversos arquivos são gerados, entre eles o arquivo que descreve a comportamento do sistema a nível de portas lógicas genéricas, o arquivo que descreve a comportamento do sistema agora a nível de portas lógicas mapeadas para a tecnologia específica, neste caso 65nm, o arquivo que especifica os atrasos de cada porta

lógica presente no circuito (*Standard Delay Format* - SDF) e alguns relatórios. Com base nos relatórios é possível obter:

1. Estimativas da utilização de área, com base na quantidade de células e o tamanho de cada célula.
2. Estimativas de temporização, a qual emprega uma aproximação do atraso de cada célula e suas interconexões para encontrar os caminhos de dados com maior atraso.

Alguns dos arquivos serão de especial importância nas etapas seguintes do desenvolvimento como o arquivo Verilog com as células mapeadas para tecnologia específica, o arquivo SDF e o arquivo com os caminhos críticos, pois estes serão utilizados como dados de entrada para a instrumentação da *netlist*.

3.3 Elaboração das Bibliotecas

Com a conclusão da etapa de síntese lógica, a *netlist* resultante da descrição comportamental do circuito de entrada é criada. Essa descrição é baseada em um conjunto de células e suas interconexões no contexto de um projeto desenvolvido com uma biblioteca de 65nm da STMicroelectronics, possuindo portanto apenas as células específicas desta biblioteca. Durante a etapa de síntese no FPGA, todas as células instanciadas na *netlist* precisam ser transpostas para o FPGA, e com esta função é elaborada a biblioteca de células, descrita em detalhes nesta seção. Essa biblioteca é responsável por produzir uma estrutura análoga que descreve o comportamento desejado para uma determinada célula baseado no modelo de falhas de atraso alvo da implementação.

As células da biblioteca são descritas em VHDL utilizando o nível de abstração comportamental, fazendo uso principalmente de componentes específicos da plataforma FPGA para emular o atraso das células. As células são descritas em duas bibliotecas diferentes, uma para emular o comportamento normal do circuito e outra para incluir a capacidade de injeção de falhas. Após concluir a descrição de todas as células para ambas a bibliotecas é possível incluir estes arquivos no processo de síntese no FPGA com a finalidade de substituir as instâncias da *netlist* pela novas estruturas, garantindo assim que o circuito projetado e as células adotadas durante a fase de síntese ASIC sejam preservadas durante a emulação do sistema.

O processo de elaboração das bibliotecas tem início a partir do manual do usuário fornecido pela STMicroelectronics que descreve as características de cada uma das células

presentes na biblioteca abrangendo entre outros atributos especificações físicas, elétricas, atraso de propagação, tempo de transição, restrições de tempo e dissipação de energia. Para descrever comportamentalmente a célula presente na biblioteca a principal característica que se busca é a função lógica executada por esta célula, descrita pela equação dos pinos de saída ou através da tabela verdade. A biblioteca da STMicroelectronics possui um total de 866 células em sua descrição, das quais 163 realizam uma função lógica exclusiva dentro da biblioteca, o restante dessas células descrevem diferentes características físicas de uma mesma função lógica. A partir das 163 funções lógicas fornecidas na biblioteca, entre lógica combinacional e sequencial, 88 são agora transpostas para uma descrição comportamental em VHDL. Grande parte das células que não foram incluídas na descrição em VHDL não o foram por conterem elementos instáveis a uma implementação tipicamente síncrona (*latches*) ou por conterem componentes de teste agregados em sua concepção os quais não são utilizados durante a etapa de síntese ASIC (*scan flip-flop*).

Com a definição do conjunto de células que será descrito representa-se cada um dos elementos do conjunto, caracterizando a entidade da célula em linguagem de descrição de *hardware* com base nos pinos de entrada e saída. Para os elementos sequenciais como registradores de dados um sinal extra, denominado en_FF, é adicionado na descrição da entidade de todos os registradores de dados da biblioteca com a função de habilitar a carga nos registradores. Este sinal é quantizado a partir do *clock* original do sistema, de forma que a referência de tempo se mantenha constante para ambos os elementos combinacionais e sequenciais. A arquitetura das células combinacionais é descrita então com base na função lógica realizada (obtida na descrição da biblioteca), agregando também o componente responsável por emular o atraso das células de modo a fornecer um comportamento temporal do sistema o mais próximo da implementação final do circuito.

O componente empregado na reprodução dos atrasos característicos de cada célula é um registrador de deslocamento com tamanho variável. O comprimento deste registrador de deslocamento pode ser modificado durante a elaboração dos casos de teste dependendo da resolução deseja durante a análise do circuito, bastando para isso a modificação de um dos arquivos que descreve o elemento de atraso em todas as células da biblioteca. É possível observar na figura 7 a arquitetura utilizada na implementação de um registrador de deslocamento de 16 *bits* em dispositivos Xilinx (SRL16). Estes componentes da Xilinx são uma alternativa eficiente na utilização das LUTs como registradores de deslocamento sem a necessidade de utilizar os registradores (*flip-flops*) específicos do FPGA. Estes componentes são ainda otimizados para utilizar uma única porção (*slice*) do FPGA para implementar múltiplos registrador de deslocamento (dependendo da tecnologia utilizada)

em série garantindo uma implementação eficiente e com baixa latência entre componentes.

O registrador de deslocamento tem a função de receber um valor inteiro representando a quantidade de retardos (em ciclos de *clock*) desejados até que uma variação do sinal de entrada seja percebido na saída, bastante semelhante ao atraso inerente das células lógicas. Dessa forma cada valor unitário de atraso é realizado compelindo o sinal através dos registradores de dados conectados em linha e conectando cada uma das saídas dos registradores a um multiplexador de dados, assim, dependendo do atraso desejado o sinal é desviado antes de atingir o final da linha de registradores.

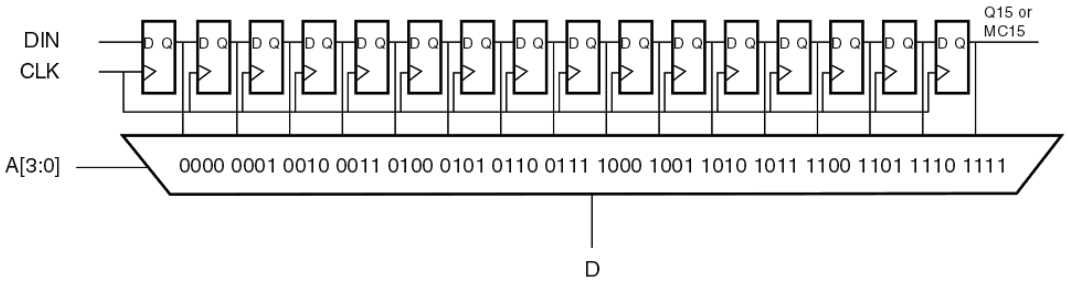


Figura 7: Arquitetura de um registrador de deslocamento de 16 *bits* em um dispositivo Xilinx.

Cada uma das células do conjunto é então descrita sobre este mesmo conceito, utilizando a função lógica da célula transposta para uma linguagem de *hardware* e agregando ainda um elemento de atraso para emulação do atraso da célula. O atraso de cada uma dessas células é calculado através do processo de quantização onde são utilizados os valores reais de atrasos obtidos durante a síntese ASIC (gravados no arquivo SDF). Esses valores são então utilizados como parâmetros para determinar a largura dos atrasos nos registradores de deslocamento, definindo assim um atraso específico para cada uma das células presente no projeto. O sistema quantizado realiza um modelo de tempo no qual descreve os atrasos do circuito em intervalos definidos de tempo determinados com base na largura do registrador de deslocamento utilizado.

Os procedimentos acima descritos são executados para todas as células pertencentes ao subconjunto da biblioteca original, e um arquivo com extensão VHDL representando esta biblioteca de células é gerado. Esta biblioteca de células descreve apenas o comportamento normal do sistema sobre uma perspectiva temporal, sem no entanto agregar instrumentos que possibilitem verificar a robustez do circuito a variações no tempo de resposta de uma célula ou um determinado grupo de células. Com esta finalidade é desenvolvida uma segunda biblioteca em linguagem de descrição de *hardware* visando inserir elementos que possibilitem alterar o funcionamento do circuito durante sua execução de modo a analisar

a robustez do sistema tirando proveito da velocidade de comutação e execução de testes de uma plataforma reconfigurável.

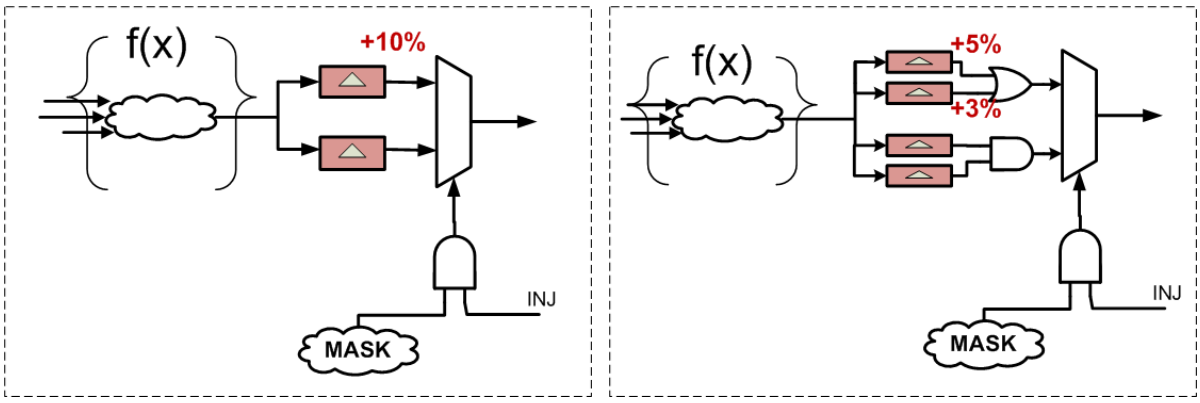
3.3.1 Arquitetura de Injeção de Falhas

A biblioteca de injeção de falhas aqui descrita tem a função de agregar elementos que possibilitem a injeção de falhas no sistema, de modo que seja possível controlar o momento e a duração da injeção de falha desejada. Para isso é necessário a modificação de algumas características elementares, tais como a quantidade de entradas e saídas das células descritas na biblioteca anterior. A arquitetura VHDL que descreve a injeção de falha implementada é adicionada em todas as células combinacionais com a mesma instância, de modo que uma alteração na arquitetura de injeção de falhas possa ser irradiada para todas as células pertencentes a biblioteca de injeção.

A arquitetura voltada para injeção de falhas segue a mesma abordagem das descrições das células sem capacidade de injeção de falhas. Em mais detalhes, para as células com capacidade de injeção de falhas, um elemento de atraso adicional é incluído em conjunto com componentes que possibilitem a troca de contexto durante o funcionamento do sistema.

Além do registrador de deslocamento já presente em todas as células, é adicionado outro registrador com o objetivo de representar uma variação na transição do sinal em relação ao sinal original. Na figura 8(a) é possível ver uma representação da arquitetura utilizada para injeção de falhas utilizando um único valor de atraso para ambas as bordas de transição. A transição de sinal produzida a partir de uma função lógica qualquer é conduzida para a entrada dos registradores de deslocamento, sendo que um deles possui o valor normal extraído do arquivo SDF e o outro possui este mesmo valor, porém com o acréscimo (ou decréscimo) de uma determinada porcentagem. Em seguida, uma única saída é selecionada através do multiplexador controlado pelo sinal de injeção, causando a modificação do sistema que ao invés de utilizar a saída do registrador com o atraso padrão passa a utilizar o registrador de deslocamento com atraso modificado.

Foi desenvolvido ainda outra arquitetura com atrasos individuais, um para borda de subida e outro para borda de descida, de modo a fornecer uma emulação do sistema com maior precisão. A escolha por uma das arquiteturas disponíveis é um compromisso entre a área de prototipação disponível e o nível de detalhamento desejado nos ensaios. Na figura 8(b) observa-se a arquitetura utilizada na implementação da estrutura com valores diferentes de atrasos para bordas de transição de subida e descida. É possível notar na figura que a saída da função lógica agora está conectada a um conjunto de quatro



(a) Atraso de mesma grandeza em ambas as bordas de transição.

(b) Atrasos com percentuais diferentes para cada borda de transição do sinal.

Figura 8: Arquiteturas implementadas para modificar o instante da transição do sinal propagado.

registradores de deslocamento, onde os dois registradores de deslocamento (um para cada borda de transição do sinal) superiores formam o conjunto que atua no momento de injeção da falha no sistema e os dois registradores de deslocamento inferiores formam o conjunto que atua no funcionamento normal do sistema.

A saída de cada um dos dois conjuntos (funcionamento normal e sobre injeção de falha) de registradores de deslocamento e conectada através de uma porta lógica AND ou OR a qual será escolhida de acordo com os valores quantizados de transição das bordas de subida e descida durante a síntese no FPGA. Se o valor da borda de descida é menor, ou seja, ocorre antes da transição de subida a porta lógica utilizada é uma AND garantindo que somente quando existir uma transição de subida e consequentemente o nível lógico alto em ambos os registradores de deslocamento a saída será considerada válida. De forma semelhante, quando o valor de transição de subida é inferior ao de descida a porta lógica OR considera uma saída válida enquanto qualquer um dos registradores de deslocamento apresentar um nível lógico alto em sua saída.

A arquitetura utilizada adiciona quatro sinais de controle na célula instrumentada que está sendo descrita, *scanin*, *scanout*, *shift*, *injector*. Os sinais de controle utilizados para injeção de falhas em uma determina célula são detalhados abaixo:

- *scanin*: tem a função de selecionar quais células terão seus valores alterados durante a injeção de falha através da alteração do estado lógico do sinal no tempo.
- *scanout*: tem a função de garantir que a ligação das células foi executada correta-

mente durante a seleção das células.

- *shift*: habilita o deslocamento dos *bits* na ligação *scanin* através dos registradores de modo a selecionar a célula.

- *injector*: realiza a alteração do contexto de funcionamento das células durante seu período de ativação.

Os sinais de controle acima descritos são responsáveis por selecionar qual célula (ou agrupamento de células) pertencente ao conjunto de injeção será escolhida para apresentar uma alteração dos valores de transição durante o funcionamento do sistema. Durante a instrumentação do sistema (seção 3.4) algumas das instâncias das células originais são sinalizadas como células com capacidade de injeção de falhas e durante este processo esses sinais são adicionados em suas instâncias.

Cada vez que uma das células do projeto é sinalizada como tendo a habilidade de injetar falhas no sistema, essa é conectada através de uma cadeia de registradores com a célula anterior. É possível observar na figura 9 a ligação dos sinais durante a elaboração da cadeia de registradores, onde cada uma das células da biblioteca de injeção tem o seu sinal de *scanout* conectado ao *scanin* da próxima célula. Cada uma das células possui um registrador de dados e a carga neste registrador é associada ao sinal de *shift*, que junto com o sinal de *clock* tem a função de compelir os valores da entrada para a saída do registrador de dados passando por cada uma das células pertencentes a cadeia de registradores.

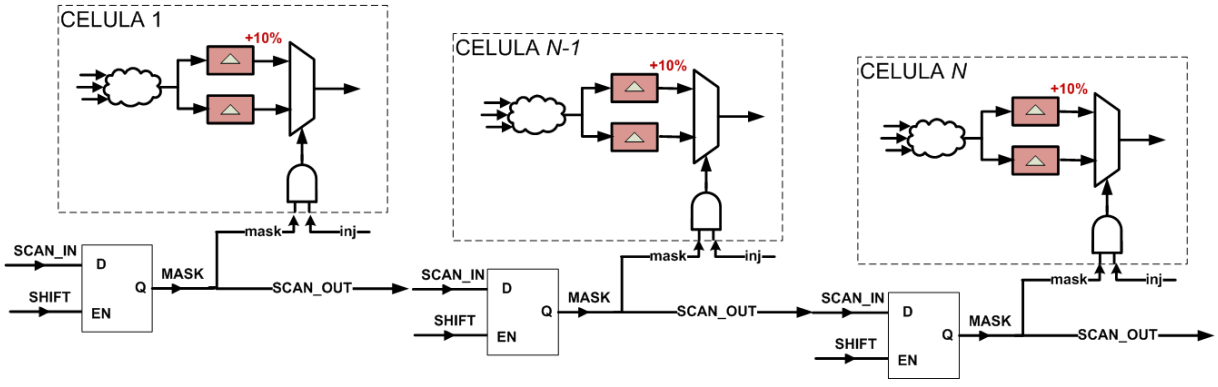


Figura 9: Construção da cadeia de registradores com o conjunto de células com capacidade de injetar falhas no sistema.

No momento em que um nível lógico é estabelecido entre dois registradores da cadeia este sinal é conduzido para uma das entradas de uma porta lógica AND aguardando o momento da injeção de falha, o qual é indicado através do sinal *injector* presente na outra entrada dessa mesma porta lógica. No instante em que o sinal é ativado o nível

lógico presente na cadeia de registradores juntamente com o sinal de injeção causam a modificação do sistema que ao invés de utilizar a saída do conjunto de registradores com os atrasos padrão passa a utilizar o conjunto de registradores de deslocamento com atrasos modificados.

3.4 Instrumentação

No contexto desta dissertação, a instrumentação do circuito tem por objetivo alterar as características originais das células descritas na *netlist*, de modo a agregar funcionalidades que emulem com maior fidelidade o comportamento do circuito final, possuindo elementos de controlabilidade para comutar o contexto de funcionamento do circuito entre normal e defeituoso manifestando assim, as possíveis falhas de atraso. A instrumentação a nível de portas lógicas é uma maneira eficiente de alterar a representação original do circuito, devido principalmente ao padrão na geração do arquivo *netlist* resultante da síntese e da uniformidade na descrição de suas interconexões. O arquivo *netlist* é sempre descrito da mesma forma seguindo o padrão com entidade principal com suas portas de entrada e saída, seguida do sentido destas portas (entradas ou saídas), conexões internas e instância das células e suas interconexões. Com isso é possível elaborar algoritmos que realizem a tarefa de modificar a *netlist* original do circuito para um formato de saída desejado que contemple as características necessárias a injeção de falhas de atraso no circuito, sem contudo alterar as estruturas lógicas já existentes no projeto.

O processo de instrumentação tem como principal função incluir as informações de atraso das células no processo de emulação do circuito de modo a fornecer uma análise mais precisa do circuito em relação a temporização do sistema. Dado que o atraso de uma célula é composto em função da biblioteca tecnologia que a descreve, a utilização da biblioteca de 65nm no processo de síntese irá fornecer os atrasos relativos a essa tecnologia mantendo entretanto, o padrão na geração do arquivo que descreve esses atrasos. Desta forma a plataforma desenvolvida é capaz de utilizar diferentes bibliotecas tecnológicas sem a necessidade de alterações na sua concepção.

A instrumentação pode ser dividida em etapas menores de modo a facilitar a descrição das funções e modularizar a entrada e saída de dados entre cada etapa com a finalidade de permitir uma maior flexibilidade nos formatos de dados utilizados na plataforma. Na figura 10 pode-se observar a divisão das etapas de instrumentação e suas interações com o dados de entrada e saída. É possível notar na figura a existência de cinco grandes blocos

funcionais, onde um deles pertence a etapa anterior a síntese lógica, e os outros quatro pertencem a etapa de instrumentação (extração de células, instrumentação, extração de atrasos e geração de constantes). Nos parágrafos seguintes são descritos com detalhes cada uma das etapas que englobam o processo de instrumentação.

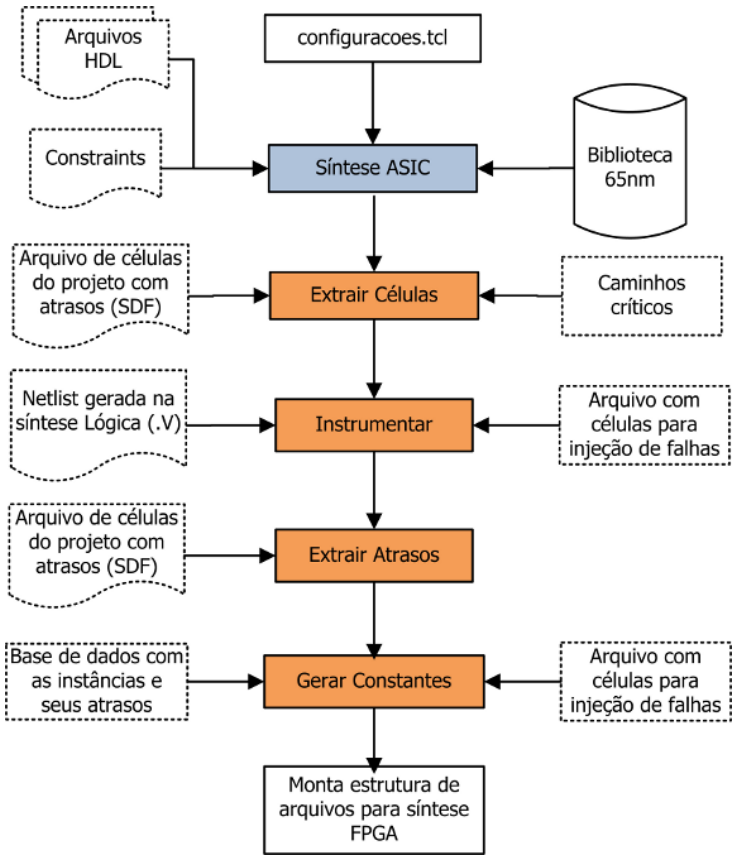


Figura 10: Visão geral do processo de instrumentação do circuito.

3.4.1 Extração de Células

O primeiro bloco pertencente a instrumentação do circuito tem a função de extrair as células utilizando determinadas características ao conjunto de células que se deseja analisar. Esta é a etapa do desenvolvimento onde se tem a maior flexibilidade em termos de análise do circuito, pois é nesta etapa que a triagem das células pertencentes ao circuito que apresentarão a capacidade de injeção de falhas é executada. Diversos métodos podem ser utilizados na seleção das células, a metodologia proposta dispõe de três formas de seleção das células para injeção de falhas: Através do caminho crítico, do tipo de célula (AND2, NOR3, AO211) e do *fanout* da célula (X3, X7).

A extração das células utilizando o caminho crítico se da através do arquivo gerado durante a síntese lógica do circuito. Este arquivo descreve os caminhos críticos (podendo

haver mais de um caminho combinacional com atrasos muito próximos ao da largura de *clock* utilizada) do circuito através das células e dos pinos destas células que compõem este caminho de dados. O arquivo contendo os caminhos críticos não possui nenhuma extensão singular que o identifique como tal, entretanto todo o arquivo gerado segue o mesmo padrão de montagem o que facilita a elaboração de um algoritmo para extrair as informações necessárias. O padrão utilizado no arquivo segue a estrutura: cabeçalho, contendo nome do projeto e data de geração do arquivo, seguido da identificação do caminho e algumas colunas que especificam o nome da instância da célula, o pino da célula utilizada, o tipo de célula para a qual a lógica foi mapeada, o *fanout* e o atraso que esta célula agrega ao caminho, entre outras informações. Para a extração foi utilizado um *script* TCL em conjunto com expressões regulares atuando sobre palavras específicas do arquivo de maneira a delimitar os caminhos contidos no arquivo e facilitar a extração das células que obrigatoriamente seguem determinados padrões na nomenclatura de suas instâncias.

Os caminhos contidos no arquivo são necessariamente delimitados pelas palavras chave "path" e "End-point" as quais estabelecem o início e fim da descrição de um caminho respectivamente. Entre estas palavras chave estão contidas as células (e seus pinos) que pertencem a um determinado caminho. Aplica-se então a expressão regular 3.1 de modo a selecionar somente os nomes das instâncias das células e os pinos utilizados, descartando as outras informações. A sequência de fixação dos limites dos caminhos e extração das células se repete até que todos os caminhos presentes no arquivo sejam processados.

```
^[ a-Z]+[a-Z0-9/_]*?\[?[0-9]*\]?\[?[0-9]*\]?/+[a-Z0-9]*        (3.1)
```

Como resultado da otimização temporal executada pela ferramenta de síntese a distribuição dos atrasos nos caminhos pode ser alterada fazendo com que uma grande porcentagem de caminhos estejam muito próximos do caminho com atraso máximo. Isso resulta em uma sensibilidade muito maior do circuito a defeitos de atraso. O atraso introduzido por variações no processo de fabricação pode causar várias diferenças entre o atraso concebido para um caminho e o atraso realmente existentes no caminho do circuito fabricado [35].

O modelo de falha de atraso em caminhos é o mais realista na modelagem de defeitos físicos de atraso porque ele pode também detectar pequenos defeitos de atraso distribuídos causados por variações no processo, ou a combinação de atrasos locais e distribuídos. No entanto, a principal limitação deste modelo é que o número de caminhos no circuito (e,

portanto, o número de falhas de atraso de caminho) pode ser exponencial do número de portas.

Devido à variação do processo de fabricação, qualquer destes caminhos pode se tornar o caminho mais longo. Portanto, um conjunto de caminhos críticos deve ser selecionado para o teste. Na prática, o critério de seleção dos caminhos pode ser baseado no atraso nominal do caminho sendo superior a um certo limiar, por exemplo, 80% do atraso máximo especificado do circuito [36].

Seguindo a concepção de que possam existir diversos caminhos muito próximos ao atraso do caminho crítico, após a aquisição de todos os caminhos calcula-se uma porcentagem do atraso máximo do circuito com base na variação da margem de tempo fornecida pela ferramenta para cada caminho crítico, de modo a selecionar quais caminhos efetivamente serão instrumentados. Com isso realiza-se a união das células pertencentes a todos os caminhos selecionados para instrumentação para em seguida gravar este conjunto de células em arquivo. Este arquivo de saída será utilizado na fase de instrumentação para designar as células que terão suas características alteradas.

Para a extração das células através do tipo de lógica realizada utiliza-se o arquivo SDF o qual descreve todas as células presentes no circuito. O processo é semelhante ao anterior pois faz uso de expressões regulares para extração das células, sem entretanto aplicar qualquer tipo de delimitação entre as células, já que todas as células do mesmo tipo devem ser extraídas. A expressão regular 3.2 realiza esta etapa garantindo que a células descritas no arquivo começam com o prefixo da biblioteca utilizada. A primeira parte da expressão ("-A 1") garante que ao encontrar uma célula que corresponde a sequência de caracteres empregados, esta retornará uma linha abaixo de onde encontrou a igualdade, essa linha contém o nome da instância atribuído a esta célula. Dessa maneira ao encontrar uma equivalência nos caracteres que formam o tipo de célula é extraído também o nome da instância a qual esta célula representa. Após a remoção de alguns caracteres que não são úteis a esta etapa o nome da instância de cada uma das células é gravada em um arquivo.

```
tipo = {NAND2, XNOR3, XOR2, AND4, NOR3, ...}
                                -A 1 HS65_GS_${tipo}X        (3.2)
```

A extração de células pelo *fanout* (expressão 3.3) utiliza uma expressão bastante semelhante a 3.2 com o mesmo sufixo de biblioteca atuando na expressão regular sobre o arquivo SDF. Entretanto neste caso o tipo de célula lógica implementada é desprezada,

sendo levada em consideração apenas a carga de *fanout* suportada pela célula instanciada. Com a extração de todas as instâncias de célula com o *fanout* desejado esta informação e gravada em um arquivo para ser utilizado nas etapas seguintes.

```
fanout = {X2,X3,X4,...,X9,X10,X11,...}
                    -A 1 HS65_GS_.*${fanout}          (3.3)
```

O arquivo produzido anteriormente, o qual será utilizado nas etapas de instrumentação e geração dos arquivos de constantes, é gerado seguindo um padrão contendo o nome das instâncias das células e os valores percentuais com os quais se deseja alterar as células, cada um deles separados por vírgula. Assim, o arquivo possui o nome da instância da célula que será alterada, o valor percentual (positivo ou negativo) da borda de subida e o valor percentual (positivo ou negativo) da borda de descida em cada uma das linhas do arquivo. O valor percentual escolhido para alterar a temporização da célula pode ser selecionado para todas as células, ou individualmente para cada uma das células.

3.4.2 Instrumentação da *Netlist*

Após selecionado o conjunto de células que serão capazes de injetar falhas no sistema é possível prosseguir para a etapa principal de instrumentação, a qual transforma efetivamente o arquivo *netlist* original no arquivo instrumentado com capacidade de injeção de falhas. O processo de instrumentação começa pela leitura da *netlist* mapeada para tecnologia de 65nm, em seguida, busca-se no arquivo a linha que aponta a entidade de maior hierarquia no projeto, pois esta linha contém os sinais que descrevem as entradas e saídas primárias do circuito. É nesta mesma linha que são adicionados os sinais de controle necessários para o gerenciamento das campanhas de injeção de falhas. A seguir, a finalidade de cada um dos sinais adicionados é detalhada:

- *scanin*: tem a função de selecionar quais células terão seus valores alterados durante a injeção de falha através da alteração do estado lógico do sinal no tempo.
- *scanout*: tem a função de garantir que a ligação das células foi executada corretamente durante a seleção das células.
- *shift*: habilita o deslocamento dos *bits* na ligação *scanin* através dos registradores de modo a selecionar a célula.

- *injector*: realiza a alteração do contexto de funcionamento das células durante seu período de ativação.

- *en_FF*: sinal quantizado do *clock* original que habilita a carga nos registradores.

Com a adição destas portas na entidade de maior hierarquia no projeto procura-se agora por instâncias de células contidas no arquivo. No caso como está sendo utilizado a biblioteca da STMicroeletronics, o padrão na nomenclatura das instâncias é "$HS65\_GS\_$" seguida do tipo de célula e seu valor de *fanout*. Portanto, para uma porta lógica AND de três entradas com *fanout* cinco teríamos "$HS65\_GS\_AND3X5$" e com a remoção do valor de *fanout* teríamos "$HS65\_GS\_AND3$". A necessidade da remoção do *fanout* das células se deve ao fato de haver uma simplificação durante a descrição das bibliotecas de instrumentação em VHDL utilizadas durante a fase de síntese no FPGA, onde as células são descritas somente com base em sua função lógica e número de entradas. Entretanto, nenhuma informação de atraso é extinta pois as mesmas células contidas neste arquivo juntamente com seus *fanouts* e atrasos são descritas no arquivo SDF. Simultaneamente com a remoção do *fanout* das células são acrescentados os parâmetros genéricos de tempo de subida, descida e tipo de arquitetura utilizada em todas as células do circuito independente da injeção de falhas.

A figura 11 apresenta o fluxograma de instrumentação, o qual ilustra as principais modificações necessárias sobre a *netlist* original para produzir a *netlist* instrumentada com capacidade de injeção de falhas. É possível observar na figura 11 a etapa subsequente na descrição do algoritmo onde é verificado se a célula em questão é um elemento sequencial, do tipo registrador de dados. Esta verificação é realizada pois sinais diferentes são adicionados para células combinacionais e sequenciais. Para células combinacionais é adicionado o sinal de *clock* do sistema o qual será responsável por compelir os dados através do registrador de deslocamento responsável por emular o atraso das células. Para células sequenciais a instanciação da mesma durante o processo de síntese já insere o sinal de *clock*, entretanto, nestas células é necessário adicionar o sinal que habilita a carga nos registradores (en_FF) o qual é quantizado a partir do *clock* original do sistema, de forma que a referência de tempo se mantenha constante para ambos os elementos combinacionais e sequenciais. Se todas as células experimentaram este processo a instrumentação das células com capacidade de injeção de falha é executada, se não, procura-se a célula seguinte do arquivo *netlist* executando o processo acima descrito até que todas as células tenham sido instrumentadas.

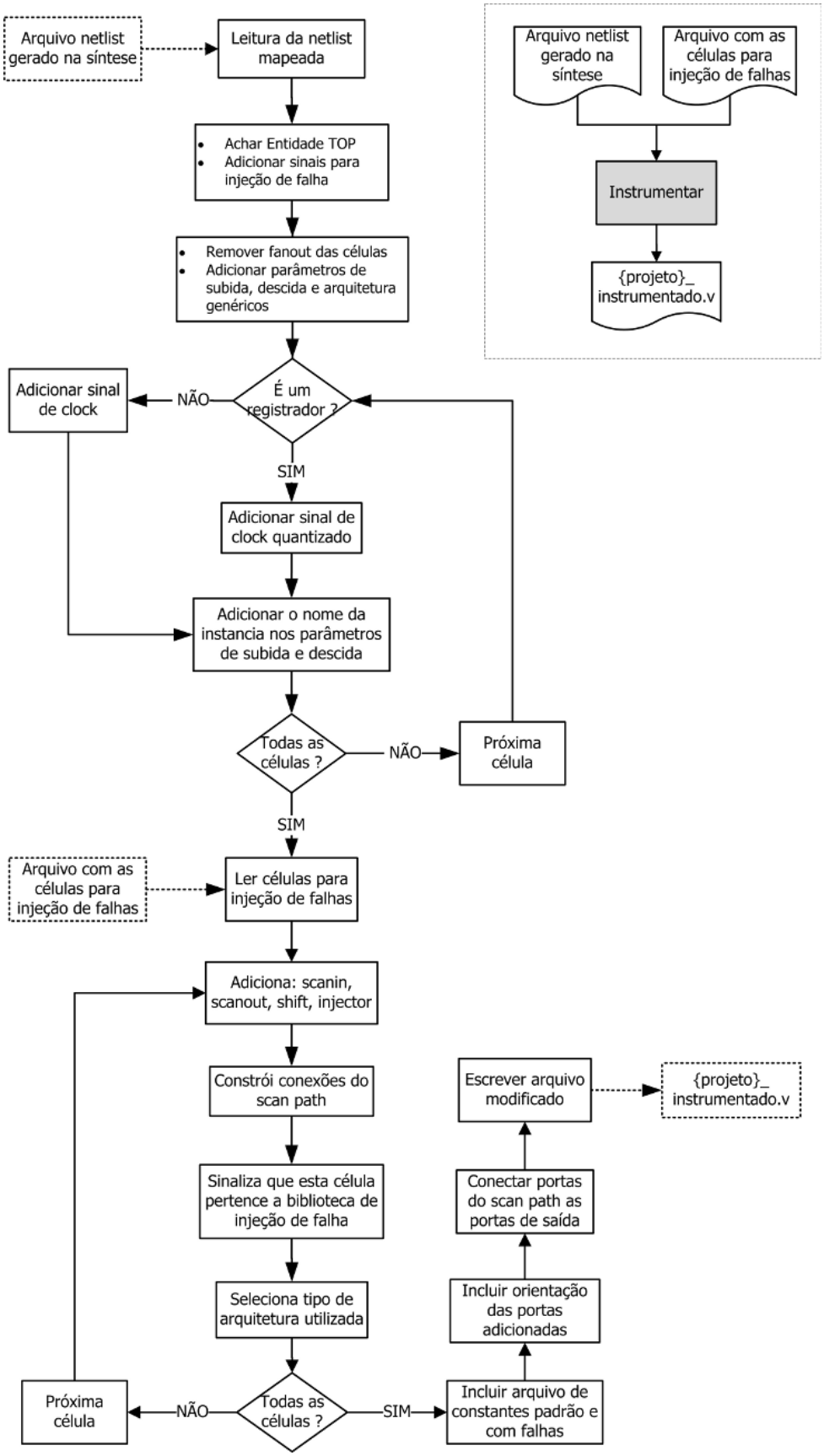


Figura 11: Fluxograma do algoritmo de instrumentação

O processo a seguir contempla as células que possuirão a capacidade de inserir falhas no circuito em análise, para isso é necessário que exista previamente descrito um arquivo contendo as células selecionadas para instrumentação que serão capazes de injetar falhas no sistema. Neste caso, este arquivo foi gerado durante a primeira etapa da instrumentação na fase de extração de células e o conjunto de células selecionadas estão gravadas em arquivo.

A partir deste ponto é realizada a leitura das células que se deseja instrumentar adicionando a capacidade de injetar falhas na circuito. A primeira linha do arquivo é lida e o nome da primeira instância é pesquisada no arquivo *netlist*. Ao encontrar a linha a qual corresponde a instância desta célula é obtido também as portas de entrada e saída desta célula. Os sinais (*scanin*, *scanout*, *shift*, *injector*) responsáveis por controlar o local da falha e o momento e a duração das injeções são então adicionados. Além de adicionar as portas para injeção de falhas neste momento é também criado as conexões lógicas entre as células com capacidade de alterar a temporização do sistema com o propósito de construir uma cadeia de registradores. De modo a selecionar, durante a injeção de falhas, qual a célula (ou células) que terão seus valores alterados é construída uma cadeia de registradores com as células com capacidade de injeção. A cadeia de registradores é responsável por deslocar os *bits* existes na porta *scanin* através de todo o circuito até a porta *scanout*, formando com isso um vetor com todas as células que devem ser ativadas em uma determinada campanha de injeção de falha.

Em seguida, todas as células com capacidade de injeção são sinalizadas. Assim, durante a etapa de síntese no FPGA ao realizar a tradução da *netlist* para um circuito mapeado para FPGA o processo ao invés de buscar pelo circuito descrito na biblioteca de instrumentação irá buscar pelo circuito lógico especificado na biblioteca de injeção de falhas, também descrita em VHDL. Neste momento da instrumentação é realizada também a escolha do tipo de arquitetura utilizada por esta célula, tendo a possibilidade de adotar a arquitetura que emula o comportamento da célula com um único atraso para ambas as bordas (subida e descida) ou a arquitetura com atrasos individuais, um para borda de subida e outro para borda de descida. A escolha por uma das arquiteturas disponíveis é um compromisso entre a área de prototipação disponível e o nível de detalhamento desejado nos ensaios. Ao finalizar esta modificação na célula procura-se pela célula seguinte para executar novamente o processo descrito acima até que todas as células presentes no arquivo de células para injeção tenham sido modificadas no arquivo *netlist*.

Ao finalizar a modificação das portas das células descritas na *netlist* é incluído o nome dos arquivos de constantes, os quais serão referenciados como arquivos externos a *netlist* gerados em uma etapa posterior (geração de constantes, seção 3.4.4). Estes arquivos contêm os valores dos parâmetros genéricos de borda de subida e descida adicionados a cada célula do projeto, cujo o valor descrito no arquivo representa o atraso quantizado da respectiva célula no circuito. Dois arquivos são incluídos na *netlist*, um para descrever os atrasos normais de todas as células do circuito e outro que contém apenas os atrasos com valores alterados das células com capacidade de injeção de erro. Incluir os atrasos quantizados em arquivos diferentes facilita a geração de múltiplos casos de teste sobre o mesmo conjunto de células sem a necessidade de alterar o arquivo *netlist*.

Após a conexão das portas a *netlist* modificada é gravada em um novo arquivo ({projeto}_instrumentado.v) o qual será utilizado posteriormente na etapa de síntese no FPGA.

3.4.3 Extração dos Atrasos

A próxima etapa tem a função de obter os atrasos reais de cada célula calculados durante o processo de síntese lógica do projeto com base na biblioteca tecnológica e nas interconexões das células no projeto. Estes atrasos serão utilizados na etapa de geração de constantes, onde os atrasos de cada célula são transformados em valores discretos pelo processo de quantização. A extração dos atrasos utiliza o arquivo SDF para obter todos os atrasos necessários, e ao extrair os atrasos das células estes dados são colocados em um formato de melhor manipulação, contendo apenas as informações essenciais de temporização do circuito.

O arquivo SDF armazena os dados de tempo gerados por ferramentas de EDA para uso em diversas fases do processo de *design*. Os dados no arquivo SDF são representados de uma forma independente de ferramenta e podem incluir: Atrasos de módulos, interconexões e portas; restrições de tempo como *setup*, *hold*, *skew* e caracterizações como escala utilizada, parâmetros relativos ao meio (tensão, temperatura, etc) e tecnologia utilizada. Para esta etapa são utilizados sobretudo os valores de atrasos de subida e descida das células presentes no projeto.

O processo de extração inicia pela leitura do arquivo SDF previamente gerado durante a síntese lógica, seguido da aplicação da expressão regular 3.4 de modo a condensar a informações contidas no arquivo em uma estrutura mais adequada para manipulação. Após empregar a expressão regular conserva-se apenas as informações das instâncias e

os valores de transição de subida e descida das portas, simplificando a iteração sobre o conjunto de células do projeto. As instâncias de células são designadas pela palavra chave "INSTANCE" seguidas do nome que as identifica de forma exclusiva em todo o projeto. O nome da instância é armazenado em uma variável e os valores de subida e descida de cada uma das portas que compõem esta célula são extraidos.

```
INSTANCE.*[^)]|::-?[0-9]{1,5}[^.)]
```
(3.4)

O atraso das portas segue logo abaixo da instância da célula, cada uma das portas possui um atraso referente ao tempo que uma alteração no estado desta porta demora a ser percebido na saída. Esta alteração pode ser uma variação do estado lógico 0 para 1 (subida) ou do estado lógico 1 para 0 (descida), e pode ser dependente ou não do valor lógico nas outras portas desta célula. Cada atraso de subida de uma porta é disposto em uma linha seguido obrigatoriamente do atraso de descida (na linha abaixo), cada uma das portas é então descrita através de seu valor de subida e descida até que não se tenha mais nenhuma porta da célula a ser analisada. A figura 12 apresenta o processo de obtenção dos atrasos através de um fluxograma, evidenciando o fluxo descrito anteriormente em que cada uma das portas de uma célula tem seu valor extraído para ambas as bordas de transição até que não haja mais portas para serem extraídas.

Após todas as portas de uma célula serem analisadas calcula-se o valor máximo do atraso da borda de transição de subida (entre todos os valores obtidos para bordas de subida daquela célula) e o mesmo para a borda de transição de descida. Dessa forma é possível obter os dois valores que serão utilizados no processo de quantização da etapa seguinte. Ao finalizar este estágio verifica-se se ainda há células a serem processadas, se houver executa-se novamente o processo acima descrito começando pela etapa de busca de uma nova instância. Quando todas as células do circuito tiverem seus valores extraídos calcula-se então o valor máximo de atraso entre todos os valores extraídos (usando tanto bordas de subida quanto de descida) para determinar o nível de resolução máximo alcançado para um determinado projeto. Na última etapa os nomes das instâncias e seus respectivos atrasos de subida e descida são gravados em um arquivo com cada um dos elementos descritos em uma linha e separados por vírgulas. A escrita dos dados neste formato padroniza a leitura facilitando o processamento dos elementos na etapa seguinte.

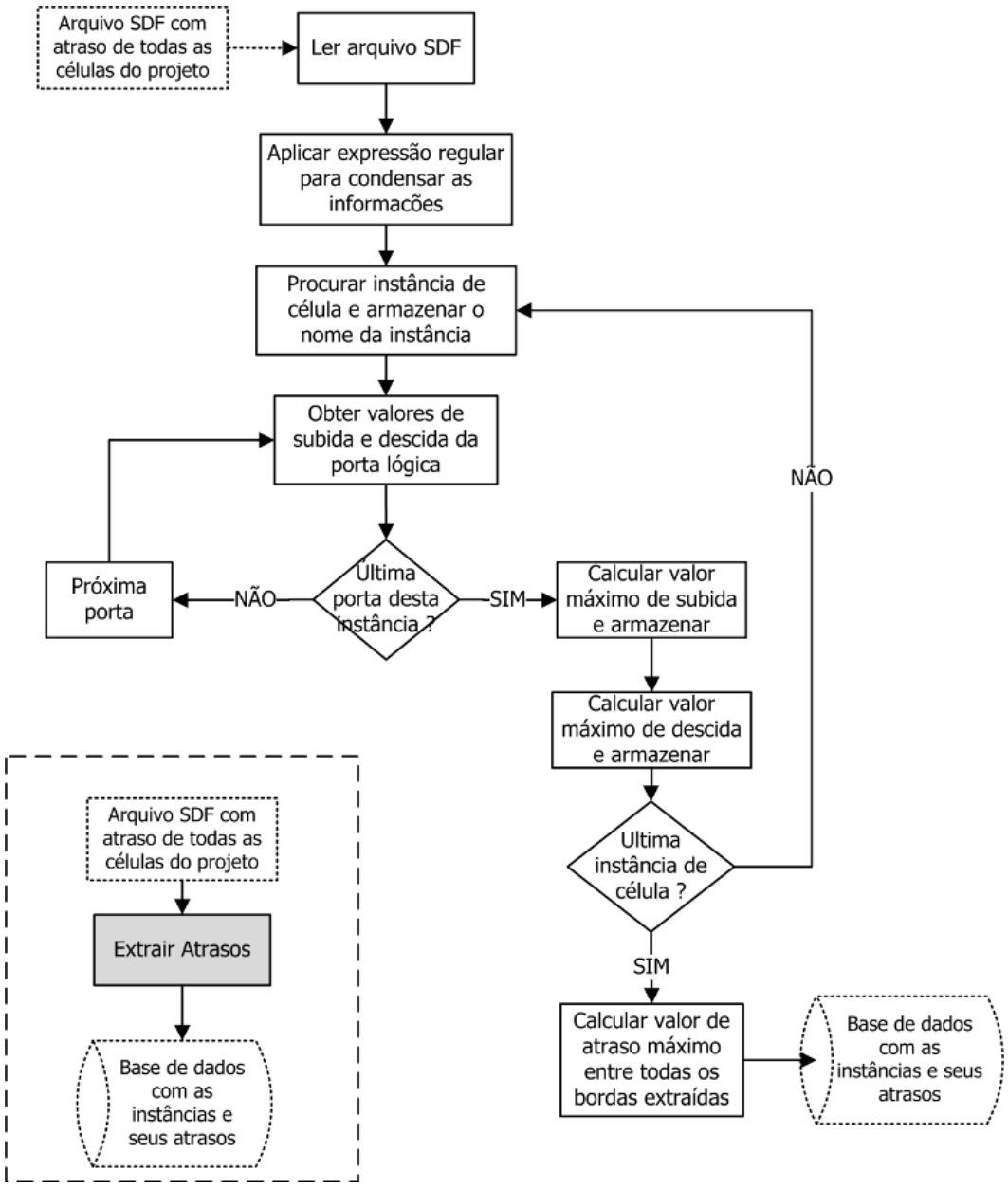


Figura 12: Fluxograma do algoritmo de extração de atrasos.

3.4.4 Geração das Constantes

A etapa a seguir tem o objetivo de elaborar os arquivos contendo os valores de referência passados na etapa de instrumentação do *netlist* (seção 3.4.2) como parâmetros para as células do arquivo *netlist*. O processo recebe como entrada dois arquivos, um contendo a relação de todas as células e seus atrasos e outro que possui o conjunto de células específicas para injeção de falhas. Dois arquivos são gerados ao final desta etapa, um contendo os atrasos extraídos do arquivo SDF agora quantizados sem quaisquer alterações, e outro contendo o valor dos atrasos quantizados com alterações percentuais nos valores de transição das bordas de subida e descida do sinal de modo a emular uma

variação desses valores no funcionamento do sistema.

O processo se inicia através da leitura do arquivo contendo todas as células do projeto e seus valores de transição para bordas de subida e descida, os quais serão utilizados no processo de quantização, essas informações são então transferidas para uma estrutura de dados no formato de lista de modo a auxiliar a manipulação desses elementos. Em seguida é efetuada a leitura do conjunto de células com capacidade de injetar falhas no sistema juntamente com seus respectivos valores percentuais que se deseja agregar em cada célula (para cada uma das bordas de transição). Este conjunto de células será utilizado especialmente para gerar o arquivo de constantes com falhas e também é colocado em uma estrutura de dados do tipo lista de modo a favorecer o acesso aos elementos.

Com todas as células do projeto a disposição, procura-se pela primeira célula da lista e seus valores de transição para calcular os valores quantizados. Com o valor máximo de atraso entre todos os valores extraídos (usando tanto bordas de subida quanto de descida) obtido na etapa anterior é possível determinar a resolução temporal máxima alcançada para o projeto. Para isso é utilizada a equação 3.5 a qual descreve a relação entre o atraso máximo do projeto sobre os níveis de quantização (representados pelos estágios do registrador de deslocamento utilizado) e a resolução temporal máxima obtida para um determinado sistema.

$$resolução = atraso\_máximo / níveis\_quantização \tag{3.5}$$

A resolução obtida pode então ser utilizada na determinação dos valores quantizados das bordas de transição de subida e descida conforme as equações 3.6 e 3.7, respectivamente. É possível observar que durante o processo de quantização os valores são submetidos a uma etapa de arredondamento do valor quantizado (devido a necessidade de múltiplos inteiros da largura do registrador de deslocamento utilizado) dando origem a um erro de quantização inerente ao processo de arredondamento.

$$\begin{aligned} borda\_positiva'_{qz} &= borda\_positiva / resolução \\ borda\_positiva_{qz} &\approx arred(borda\_positiva'_{qz}) \end{aligned} \tag{3.6}$$

$$\begin{aligned} borda\_negativa'_{qz} &= borda\_negativa / resolução \\ borda\_negativa_{qz} &\approx arred(borda\_negativa'_{qz}) \end{aligned} \tag{3.7}$$

Após o cálculo dos valores quantizados é possível verificar se a célula sobre análise é um

registrador de dados, identificada através do sufixo "_reg" adicionado à instância da célula durante o processo de síntese. Se a célula for realmente um registrador o valor gravado em arquivo é igual a zero, pois neste modelo de falhas não emprega-se o atraso das células sequenciais. Se a célula não for um registrador, o valor quantizado calculado anteriormente é então armazenado para posteriormente ser gravado em arquivo. Em seguida é possível verificar se a célula pertence ao conjunto de células para injeção de falhas, se a célula não pertencer ao conjunto de injeção realiza-se a busca pela próxima célula na lista, porém, se esta pertencer ao conjunto busca-se novamente os valores originais de transição juntamente com os valores percentuais que se deseja agregar nas bordas de transição do sinal. Utilizando as equações 3.8 e 3.9 é possível estabelecer os novos valores de transição que serão utilizados no processo de quantização descritos anteriormente pelas equações 3.6 e 3.7 porém, agora contendo os valores modificados de transição.

$$borda\_positiva' = borda\_positiva * (1 + porcentagem\_pos/100) \tag{3.8}$$

$$borda\_negativa' = borda\_negativa * (1 + porcentagem\_neg/100) \tag{3.9}$$

Com os novos valores quantizados do conjunto de células para injeção esses valores são armazenados para posteriormente serem gravados em arquivo. Após os novos valores terem sidos armazenados observa-se se esta é a última instância da lista, se não for o processo é executado sobre a instância seguinte da lista. A partir deste ponto o algoritmo retorna ao momento de busca do nome da instância e seus valores de transição realizando novamente os estágios descritos anteriormente até que todas as células do projeto tenham seus valores quantizados. A figura 13 ilustra o fluxo de dados utilizado na elaboração do algoritmo de geração dos parâmetros de entrada para arquivo *netlist*.

Ao se verificar que não há mais células pendentes para calcular o atraso quantizado é iniciado o estágio que irá gravar as variáveis armazenadas durante todo o processo de iteração sobre as células em um arquivo. Ambos os arquivos são gravados sobre a extensão de um cabeçalho Verilog (*verilog header*) o qual pode ser incluído na *netlist* através de um comando específico para esta função (*include*). Dessa forma é possível separar a descrição das instâncias e suas interconexões dos valores normais e dos valores com falhas utilizados durante a síntese no FPGA. Isto facilita a depuração do código e a geração de múltiplos casos de testes sobre um mesmo *netlist* instrumentado. Visando auxiliar a automação do processo, após a conclusão da etapa de geração das constantes, todos os arquivos necessários ao estágio de síntese no FPGA são copiados para uma estrutura de diretórios, onde será realizado o processo de elaboração do arquivo de configuração do

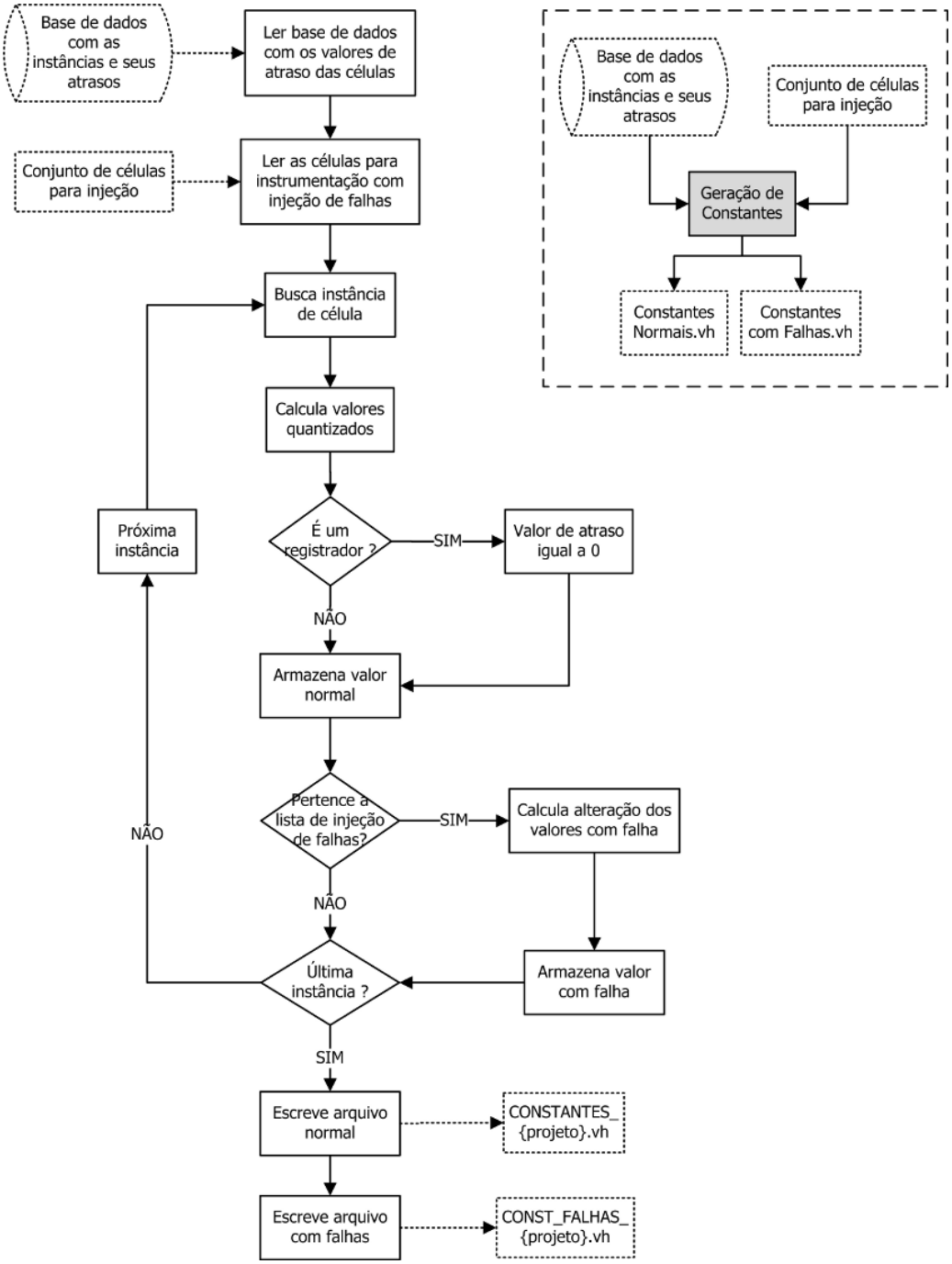


Figura 13: Fluxograma do algoritmo de geração de constantes.

FPGA. Dessa forma, todos os arquivos do projeto original utilizados na síntese ASIC são copiados para essa estrutura, juntamente com a *netlist* instrumentada e seus arquivos de constantes com e sem falhas finalizando assim, o processo aqui denominado de forma mais ampla de instrumentação.

3.5 Síntese no FPGA

O processo de síntese no FPGA corresponde a etapa responsável por gerar o arquivo contendo toda a lógica necessária para a validação dos conceitos apresentados e a geração do arquivo de configuração do dispositivo para avaliação da plataforma proposta. Para realizar esta etapa uma série de arquivos gerados nas etapas anteriores são utilizados, e de posse desses arquivos é executado o processo de síntese no FPGA com o objetivo de gerar o arquivo empregado nas simulações para validar os conceitos apresentados nesta dissertação. Após a validação, o fluxo de projeto normal do FPGA é executado com a finalidade de elaborar o arquivo de configuração do dispositivo, de modo a avaliar e extrair os resultados da plataforma de injeção de falhas em *hardware*.

O fluxo de projeto de FPGAs possui três estágios fundamentais: (1) síntese, (2) implementação e (3) geração do arquivo de configuração. A implementação pode ainda ser dividida em três segmentos menores, são eles: (1) tradução, (2) mapeamento e (3) posicionamento e roteamento (sendo estas duas etapas definidas sobre um mesmo segmento devido a proximidade de suas funcionalidades já que em um FPGA as estruturas de roteamento seguem um padrão, o posicionamento em especial irá definir o roteamento). Cada uma das etapas da síntese no FPGA executa uma função específica descrita em detalhes a seguir:

1. Síntese: O processo de síntese converte o código HDL (VHDL / Verilog) em uma *netlist* a nível de portas lógicas (representada nos termos da biblioteca de componentes UNISIM, uma biblioteca Xilinx que contém as primitivas básicas). Por padrão, a Xilinx utiliza o sintetizador *Xilinx Synthesis Technology* (XST) já associado ao ISE. O relatório de síntese contém diversas informações úteis, entre elas uma estimativa da frequência máxima alcançada para o projeto e alguns avisos e notificações sobre o resultado final da síntese indicando eventuais problemas ou questões que necessitem uma atenção especial por parte do desenvolvedor. Após uma síntese bem-sucedida, é possível observar o esquemático de tecnologia através da *netlist Native Generic Circuit* (NGC) gerada. Esse esquemático contém os componentes do dispositivo e suas interconexões utilizadas na elaboração do projeto.

2. Tradução: A tradução é executada pelo programa NGDBUILD. Durante a fase de tradução o arquivo NGC é convertido para uma nova *netlist*, a qual recebe a *netlist* de entrada e as restrições de *design* e gera uma *netlist* com extensão *Native Generic Database* (NGD). A principal diferença entre os dois arquivos é a biblioteca utili-

zada como referência. Onde a *netlist* NGC baseia-se na biblioteca de componentes UNISIM, concebidos para a simulação comportamental, e o arquivo NGD baseia-se na biblioteca SIMPRIM contendo algumas informações aproximadas sobre a temporização do sistema.

3. Mapeamento: O mapeamento é executado pelo programa MAP. O programa recebe como entrada o NGD gerado na etapa anterior executando sobre este arquivo a verificação das regras de projeto. Durante a fase de mapeamento as primitivas SIMPRIM de uma *netlist* NGD são mapeadas em recursos específicos do dispositivo: LUTs, *flip-flops*, BRAMS e outros. A saída do programa MAP é armazenada no formato *Native Circuit Description* (NCD) o qual contém informações precisas sobre os atrasos do circuito, mas não informações sobre os atrasos de propagação (pois o *layout* ainda não foi processado).

4. Posicionamento e roteamento: O posicionamento e o roteamento são realizados pelo programa PAR, este é o passo mais importante e mais demorado da implementação. Ele define como os recursos do dispositivo estão localizados e interligados dentro de um FPGA. O posicionamento é ainda mais importante do que o roteamento, pois um posicionamento impróprio tornaria impossível um bom roteamento. A saída do programa PAR é também armazenada no formato NCD. A fim de proporcionar a possibilidade para os desenvolvedores em FPGA de refinar o posicionamento, o programa PAR possui diversas opções que permitem elaborar restrições de tempo, e se pelo menos uma das restrições não puder ser cumprida o PAR retornará um erro.

5. Geração do arquivo de configuração: BitGen é a ferramenta responsável por gerar o fluxo de *bits* para configuração do dispositivo Xilinx. Depois que o projeto é totalmente roteado, é possível configurar o dispositivo usando o arquivo gerado pelo BitGen. BitGen tem como entrada uma descrição do circuito totalmente roteado (NCD) e produz um arquivo contendo o fluxo de *bits* de configuração (BIT) como saída. Este arquivo binário contém as informações de configuração do arquivo NCD responsável por definir a lógica interna e as interconexões do dispositivo FPGA, juntamente com informações específicas do dispositivo alvo. Os dados binários no arquivo BIT são então transferidos para as células de memória do dispositivo FPGA, ou utilizados para criar um arquivo de PROM.

É possível observar na figura 14 o fluxo de projeto de um FPGA. Em particular, cada uma das etapas associadas à elaboração do arquivo de configuração do dispositivo.

Observa-se também na figura 14, as ferramentas utilizadas em cada uma das etapas do processo como um todo e o momento em que são gerados o arquivo intermediário para validação funcional do sistema e o arquivo de testes utilizado no dispositivo de *hardware*.

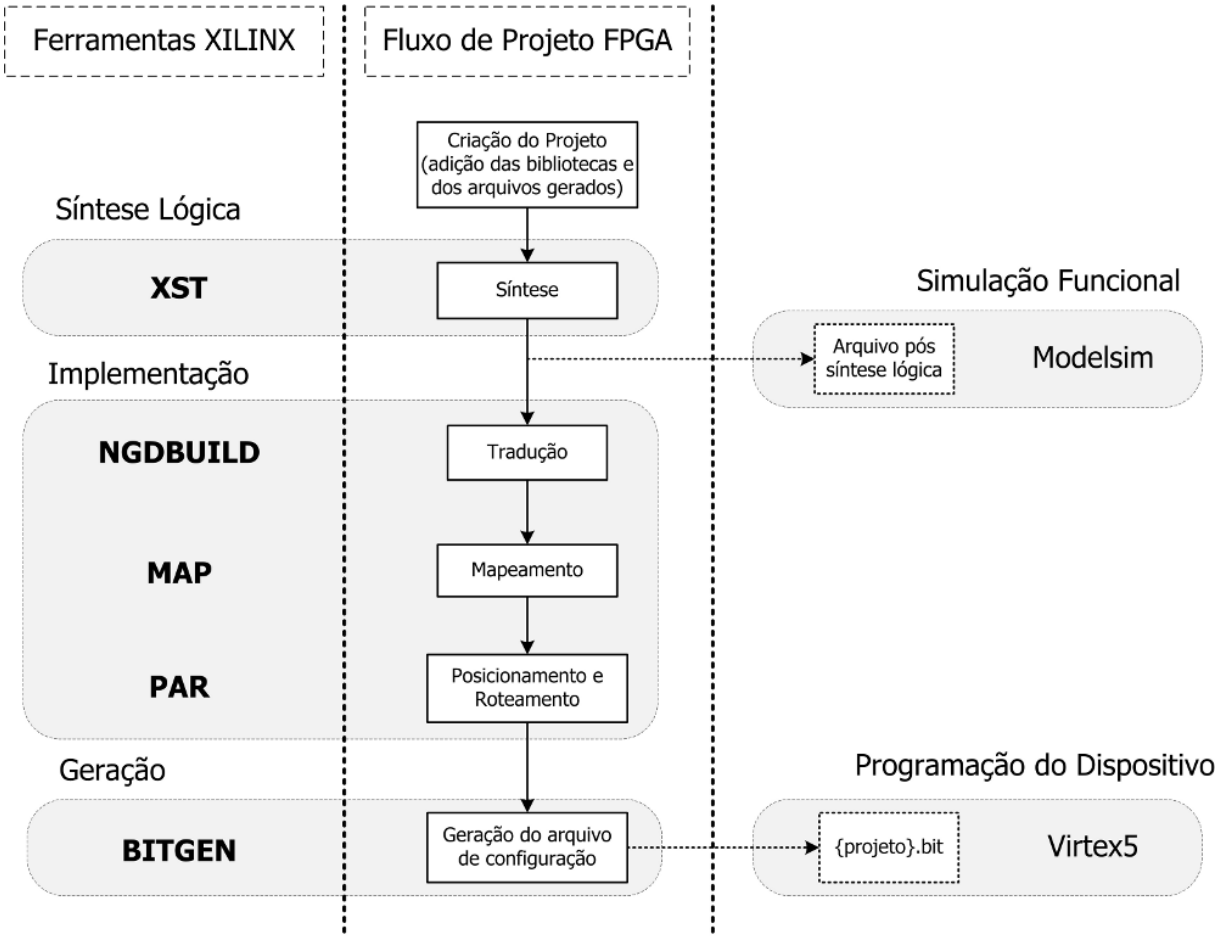


Figura 14: Fluxo de projeto FPGA apresentando as ferramentas Xilinx utilizadas em cada etapa do processo juntamente com os arquivos gerados para validação e avaliação da plataforma.

Para dar início ao processo de síntese são adicionados ao projeto os arquivos gerados nas etapas anteriores (as bibliotecas, descritas na seção 3.3, o arquivo *netlist* instrumentado, seção 3.4.2 e os arquivos de constantes, criados na seção 3.4.4) juntamente com o arquivo (ou arquivos) que descreve o circuito original utilizado na etapa de síntese do ASIC (seção 3.2). Note que este último será utilizado como referência na validação do sistema.

Durante o processo de síntese, a *netlist* (descrita em Verilog) é interpretada pela ferramenta (XST) e todas as instâncias são convertidas em estruturas conhecidas pelo sintetizador. A ferramenta ao realizar a conversão dessas instâncias em estruturas conhecidas busca por construções contendo os mesmos nomes utilizados na alocação das

instâncias, os quais somente existirão nas bibliotecas descritas anteriormente (em VHDL) e previamente adicionadas ao projeto. Dessa forma, ao encontrar a instância de um célula a ferramenta busca pela estrutura descrita na biblioteca (normal ou com capacidade de injetar falhas) e a substitui por uma estrutura análoga que descreve o comportamento desejado (com base no modelo de falhas de atraso que se pretende implementar) para uma determinada célula. Mesmo durante a substituição das instâncias pelas novas estruturas todas as interconexões entre as células são conservadas, garantindo que a estrutura projetada e as células adotadas durante a fase de síntese ASIC sejam mantidas.

O processo de síntese produz um arquivo chamado de modelo de simulação pós-síntese (*post-synthesis simulation model*). Esse arquivo é resultado da conversão do arquivo binário padrão de saída da síntese (NGC) em um modelo de simulação (um arquivo VHDL ou Verilog estrutural baseado na biblioteca de simulação UNISIM). Esse modelo de simulação, pode ser usado para verificar se a funcionalidade do sistema está correta após a síntese, pois possibilita executar uma simulação pós-síntese no simulador. De posse do arquivo de simulação é realizada a etapa de validação do sistema, descrita em mais detalhes no capítulo 4.

Se o processo de validação transcorrer de acordo com o esperado é possível dar continuidade ao fluxo de projeto do FPGA visando a geração do arquivo de configuração do dispositivo. Nesse momento, é iniciada a elaboração do arquivo que irá envolver todos os arquivo descritos anteriormente para realizar a avaliação da plataforma em *hardware* e a geração dos múltiplos casos de teste, os quais são baseados nos testes realizados durante a etapa de validação do sistema. É realizado então, o fluxo de projeto através da sequência de passos da implementação até a geração do arquivo de configuração do dispositivo FPGA, elaborando um arquivo de configuração para cada um dos casos de teste formulado.

4 Validação

O processo de validação descrito em detalhes nesta seção visa demonstrar que a metodologia proposta está de acordo com as especificações apresentadas no capítulo 3. Em mais detalhes, a validação do sistema visa certificar que os atrasos extraídos do arquivo SDF, os quais representam os atrasos específicos das células para um determinado projeto no contexto de uma biblioteca de 65nm, estão corretamente quantizados e transpostos na estrutura de *hardware* reconfigurável.

A validação do sistema tem origem na síntese do ASIC, onde o projeto escolhido para validação é sintetizado utilizando a biblioteca tecnológica de 65nm. O resultado dessa etapa é a *netlist* e o arquivo de atrasos. Em seguida, o processo de instrumentação o qual altera a *netlist* original de modo a incluir os valores de temporização do sistema e a capacidade de injeção de falhas é iniciado. A seguir, o processo avança para a etapa de síntese no FPGA onde é gerado o arquivo de simulação pós síntese lógica (*post-synthesis simulation model*), o qual faz uso das bibliotecas elaboradas e da *netlist* instrumentada para gerar o arquivo de simulação funcional. Este arquivo contém todas as interconexões do sistema incluindo os atrasos agregados das células e os componentes responsáveis por acionar uma falha durante o funcionamento do sistema, porém, descritos somente por meio de elementos disponíveis no FPGA. Com isso é possível simular este arquivo e verificar funcionalmente se o circuito realiza aquilo para o qual foi projetado.

Com o objetivo de prover a controlabilidade necessária para as células com capacidade de injeção de falhas, um módulo capaz de controlar os experimentos de injeção de falhas baseado em linguagem de descrição de *hardware* foi desenvolvido. O controlador de injeção tem a função de designar qual, ou quais, células estarão ativas durante determinada campanha de injeção e controlar a duração dessas falhas. O controlador de injeção possui ainda duas arquiteturas diferentes para prover a injeção de falhas em dois modelos distintos: (1) modelo de falhas de atraso em portas lógicas (*gate delay fault*) e (2) modelo de falhas de atraso em caminhos (*path delay fault*).

Após a elaboração da arquitetura de controle de injeção de falhas é necessário desenvolver a infraestrutura responsável por estimular os circuitos provendo uma carga de trabalho para o sistema durante a injeção das falhas. Com esta finalidade são desenvolvidas as bancadas de teste (*testbenchs*), as quais utilizam um conjunto de entrada conhecido de modo a obter um conjunto de saída também conhecido para realizar a comparação do sistema sobre falhas com o sistema de referência, identificando que a alteração da temporização do circuito foi suficiente para causar uma falha na saída do sistema.

Durante a etapa de síntese ASIC algumas informações importantes para a validação são obtidas, a mais importante adquirida durante a etapa de instrumentação é a obtenção da célula de maior atraso do projeto. Essa informação é importante ao definir a resolução máxima que a plataforma consegue proporcionar para determinado projeto, e verificar se o atraso agregado em cada uma das células do sistema esta sendo realizada de acordo com os parâmetros obtidos durante a fase de síntese ASIC. Os quais devem possuir uma representação análoga durante o funcionamento do sistema. Na figura 15 é possível observar um trecho do arquivo SDF gerado a partir da síntese ASIC para uma biblioteca de 65nm, o trecho salienta especificamente uma das células empregadas no projeto ressaltando o tipo de célula (uma XOR com três entradas), o nome de sua instância (g39178) e o maior atraso para borda de subida (144ps) e descida (125ps).

Com os arquivos resultantes da síntese do ASIC é executado o processo de instrumentação da *netlist*. Uma das primeiras etapas que compõem o processo de instrumentação é a escolha das célula que possuirão a capacidade de injetar falhas no sistema. Neste momento, é realizada a etapa de extração das células com o objetivo de individualizar o conjunto de células que se deseja analisar. A plataforma fornece duas alternativas já incorporadas ao desenvolvimento da ferramenta que exploram a extração de célula com base em arquivos gerados durante a síntese do ASIC. Uma das alternativas explora a extração de células pertencentes ao caminho crítico e outra explora a extração de células através de critérios específicos (como tipo de célula ou *fanout* de saída). A figura 16 ilustra a etapa de extração de células através de um fluxograma evidenciando a exigência de optar pela extração de células pertencentes ao caminho crítico ou por células que possuem determinadas características relevantes sobre a perspectiva de falhas de atraso no circuito.

A extração das células com base no caminho crítico se da através do arquivo gerado durante a síntese lógicas do circuito. Este arquivo descreve os caminhos críticos do circuito através das células que compõem este caminho de dados. Os caminhos contidos no arquivo são obrigatoriamente delimitados por palavras chave as quais estabelecem o início e fim

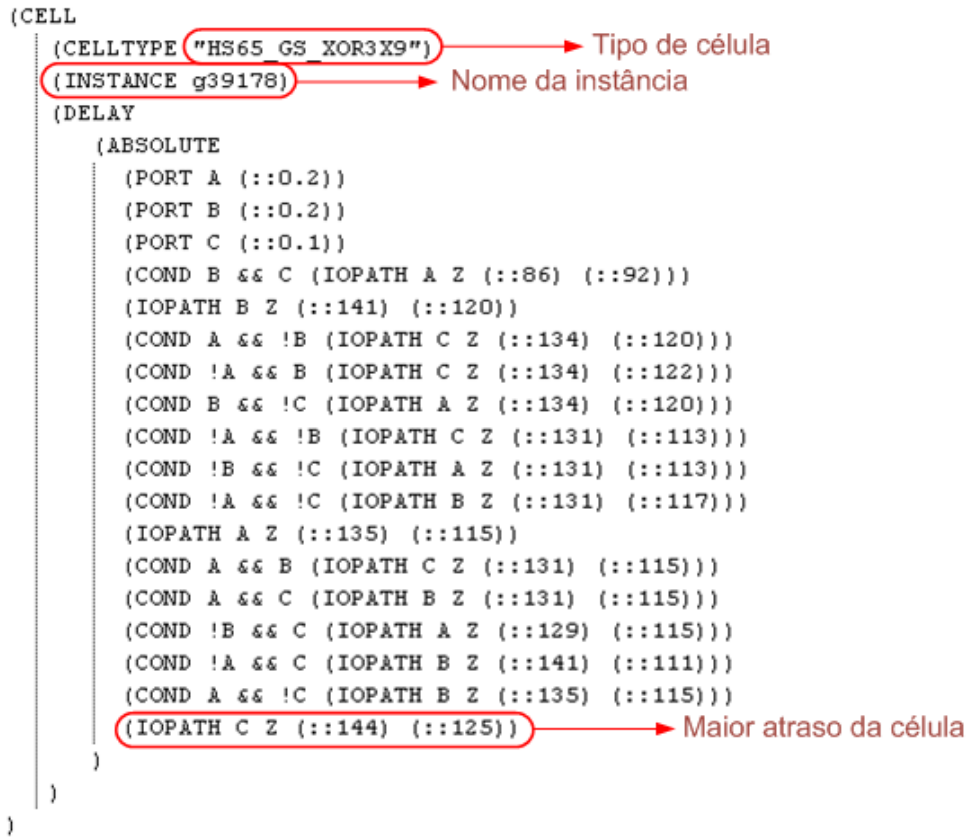


Figura 15: Trecho de um arquivo SDF gerado a partir da síntese lógica com uma biblioteca de 65nm.

da descrição de um caminho. Entre estas palavras chave estão contidas as células que pertencem a um determinado caminho. Realiza-se então leitura das instâncias das células, descartando outras informações presentes no arquivo. A sequência de fixação dos limites dos caminhos e extração das células se repete até que todos os caminhos presentes no arquivo sejam processados. Quando o último caminho é processado um arquivo com todas as células extraídas é gravado.

Para a extração das células através do tipo de lógica realizada utiliza-se o arquivo SDF o qual descreve todas as células presentes no circuito. Após a leitura deste arquivo busca-se por uma célula que corresponde a sequência de caracteres empregados, em seguida avança-se uma linha de onde a igualdade foi encontrada, essa linha contém o nome da instância atribuído a esta célula. Dessa maneira ao encontrar uma equivalência nos caracteres que formam o tipo de célula é possível extrair também o nome da instância a qual esta célula representa.

O mesmo se aplica para a extração de células pelo *fanout*, entretanto, neste caso o tipo de célula lógica implementada é desprezada, sendo levada em consideração apenas a carga de *fanout* suportada pela célula instanciada. Chegando ao fim do arquivo e com a

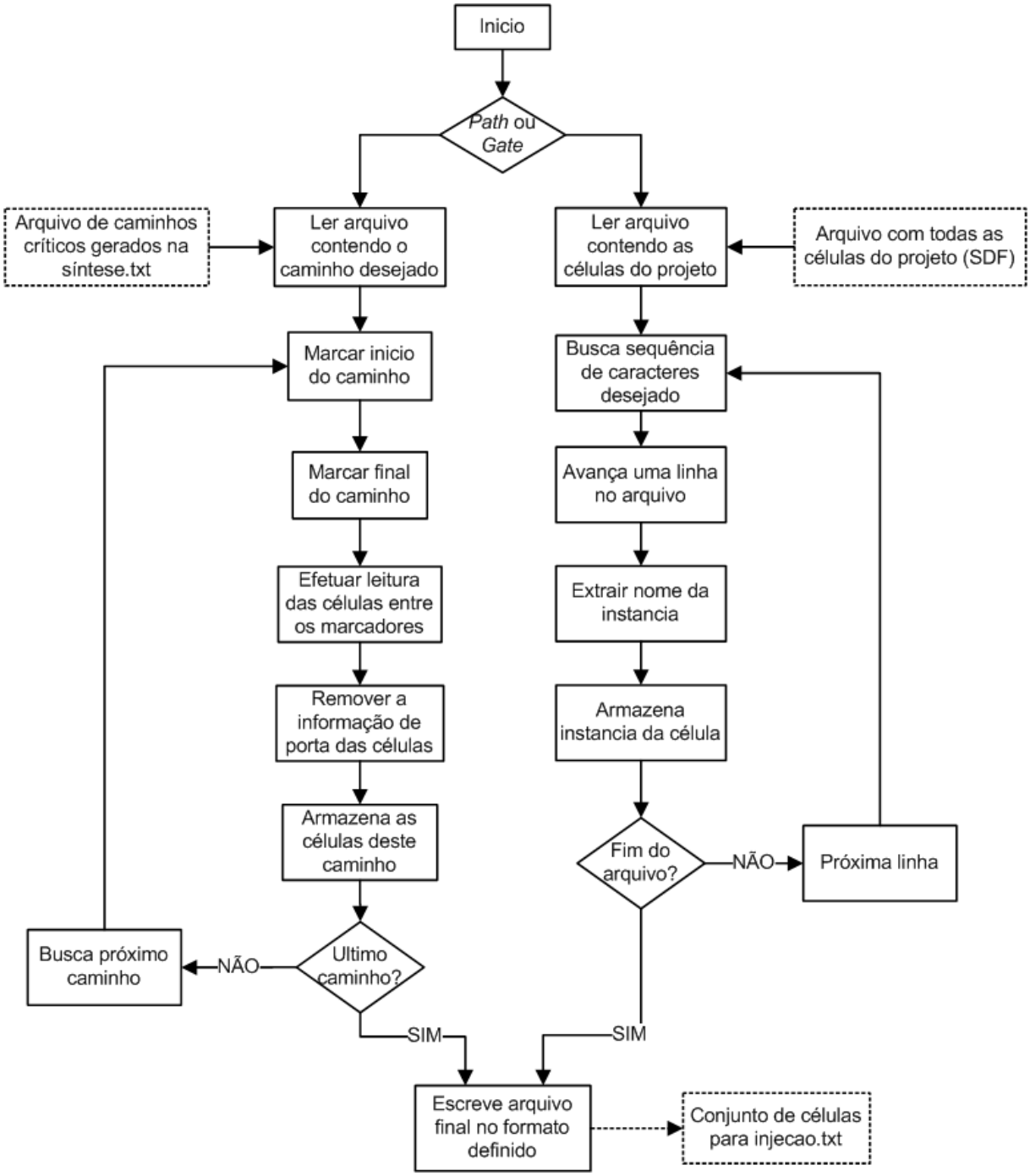


Figura 16: Fluxograma da extração de células.

extração de todas as instâncias de célula com o *fanout* desejado esta informação e gravada em um arquivo para ser utilizado nas etapas seguintes. É importante destacar que tanto a escolha de extração de células por caminho crítico quanto por tipo de célula definidas no início do processo de extração geram o mesmo formato de arquivo de saída, o qual contém o nome das instâncias das células encontradas com base nos parâmetros utilizado e o percentual de modificação desejado para cada uma das bordas de transição. Da mesma forma, qualquer algoritmo que realize a extração de células com base em um determinado

critério de interesse pode ser utilizado bastando apenas que a gravação das células em arquivo siga o formato definido na etapa de extração de células.

Com a seleção das células o processo de instrumentação da *netlist*, através da extração dos atrasos e da geração das constantes, é iniciado. Com isso todos os arquivos necessários para a realização da síntese no FPGA com o objetivo de gerar o arquivo de simulação pós síntese lógica utilizado na validação funcional do sistema é gerado. A figura 17 representa a estrutura empregada durante a validação do sistema no simulador (*Modelsim*) e uma uma sequência numérica que relaciona os sinais do simulador com pontos específicos na estrutura adotada. A estrutura produzida é resultado da síntese lógica executada sobre um codificador *Reed Solomon*, o qual é descrito em detalhes no capitulo 5.

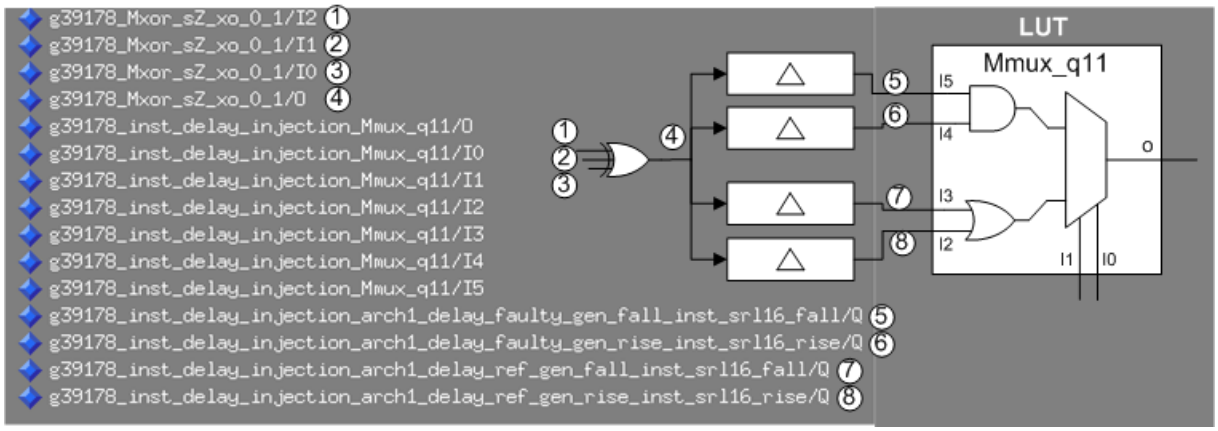


Figura 17: Representação gráfica da arquitetura utilizada durante a validação funcional do sistema.

Os sinais um, dois e três representam as entradas de uma porta lógica do tipo XOR. O sinal número quatro representa a saída desta porta XOR conectada com as entradas dos registradores de deslocamento, onde os dois registradores de deslocamento superiores correspondem aos atrasos do sistema em um funcionamento na presença de falhas (sinal cinco representa a borda de descida e sinal seis representa a borda de subida) e os dois registradores de deslocamento inferiores correspondem aos atrasos do sistema em um funcionamento normal (sinal sete representa a borda de descida e sinal oito representa a borda de subida). Durante a síntese no FPGA o algoritmo empregado pela ferramenta XST julgou eficiente o uso de uma *Look-Up Table* (LUT) de seis entradas para representar a lógica que determina a largura do pulso de saída e a lógica que alterna o funcionamento do bloco entre a saída normal e a saída sobre influência da falha injetada.

Na figura 18 é possível observar a simulação do sistema sem a presença de falhas, mais precisamente o funcionamento de uma célula específica (g39178, um XOR de três entradas, a mesma representada através de seu SDF na figura 15). Ainda na figura 18

é possível observar que a transição em um dos sinais na entrada da XOR (sinais um, dois e três da figura 17) são percebidos na saída da célula (sinal quatro na figura 17) de forma instantânea, porém, ao empregar a utilização dos registradores de deslocamento a transição do sinal somente é percebida nove ciclos de *clock* depois. O mesmo acontece para uma variação na borda de descida do sinal, porém, com uma duração menor de oito ciclos de *clock*, sendo que ambos os atrasos são representações quantizadas dos atrasos originais extraídos do arquivo SDF.

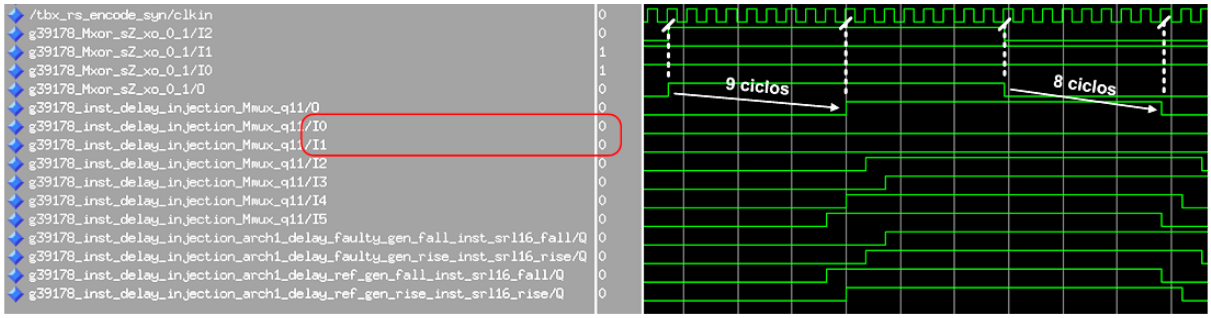


Figura 18: Sistema em funcionamento agregando atrasos diferentes para bordas de subida e descida.

Dessa forma, um paralelo entre o atraso original do SDF e os valores quantizados do sistema é estabelecido através da equação 4.1, substituindo os valores para o projeto em análise. O maior valor de atraso encontrado para o projeto em questão foi de 244ps. Com isso este valor foi dividido pelos níveis de quantização utilizados, neste caso 16, pois é utilizado um registrador de deslocamento de 16 *bits*. Assim, a resolução obtida para este sistema é de 15.25ps, o que possibilita calcular os valores quantizados das bordas de subida e descida utilizados como parâmetros de entrada das células durante a elaboração do arquivo de simulação funcional. Para isso, é empregada a equação 4.2, a qual descreve a relação entre o valor da borda de subida, extraído do arquivo SDF (figura 15) obtido durante a síntese ASIC do projeto, e a resolução encontrada para o sistema. O mesmo desenvolvimento é realizado para a borda de descida e descrito na equação 4.3.

$$resolução = \frac{atraso\_máximo}{níveis\_quantização} \tag{4.1}$$

$$\begin{aligned} resolução &= 244ps/16 \\ resolução &= 15.25ps \end{aligned}$$

$$borda\_subida_Q = \frac{borda\_subida}{resolução} \tag{4.2}$$

$$borda\_subida_Q = 144ps/15.25ps$$
$$borda\_subida_Q = arred(9.442)$$
$$borda\_subida_Q = 9$$

$$borda\_descida_Q = \frac{borda\_descida}{resolução} \tag{4.3}$$

$$borda\_descida_Q = 125ps/15.25ps$$
$$borda\_descida_Q = arred(8.196)$$
$$borda\_descida_Q = 8$$

Durante a injeção de falhas no sistema, o multiplexador, representado na figura 17, altera a saída do sistema do funcionamento normal para o conjunto de registradores de deslocamento com os valores de transição alterados com base em uma porcentagem definida anteriormente. Dessa forma os valores de transição das bordas de subida e descida são modificados emulando uma falha no sistema. A adição de 10% na borda de subida (conforme equação 4.4) e 40% na borda de descida (conforme equação 4.5) foi incluída na mesma porta lógica utilizada anteriormente (g39178). É possível observar na figura 19 que a transição do sinal do nível lógico zero para o nível lógico um efetivamente sofre uma alteração da transição de oito para dez ciclos de *clock*. O mesmo acontece com a adição de 40% na borda de transição de descida onde o sinal passa a atrasar onze ciclos de *clock* ao invés dos oito anteriores.

$$borda\_subida_Q = \frac{borda\_subida + prc_{subida}}{resolução} \tag{4.4}$$

$$borda\_subida_Q = (144ps + 10\%)/15.25ps$$
$$borda\_subida_Q = arred(10.386)$$
$$borda\_subida_Q = 10$$

$$borda\_descida_Q = \frac{borda\_descida + prc_{descida}}{resolução} \tag{4.5}$$

$$borda\_descida_Q = (125ps + 40\%)/15.25ps$$

$$borda\_descida_Q = arred(11.475)$$

$$borda\_descida_Q = 11$$

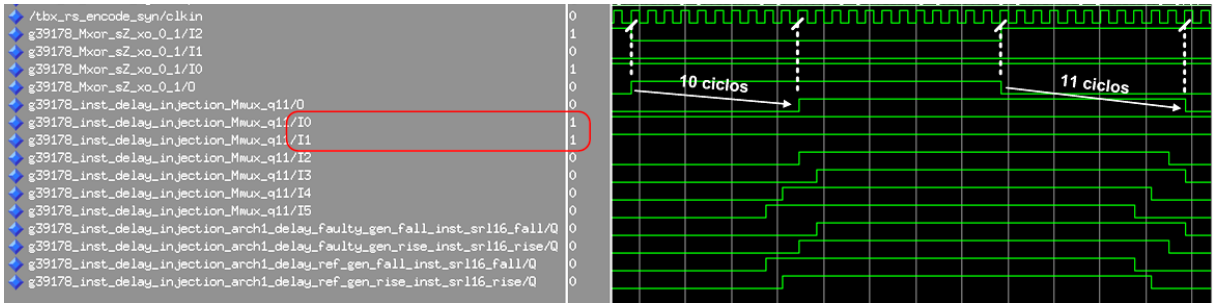


Figura 19: Sistema em funcionamento na presença de falhas agregando atrasos diferentes para bordas de subida e descida.

Durante a etapa de validação é realizado o desenvolvimento do controlador de injeção, o qual visa facilitar a conexão das portas de controle do circuito em teste (já inseridas na entidade de maior hierarquia do circuito final durante a elaboração da *netlist* instrumentada, seção 3.4.2). O controlador tem a capacidade de alterar os parâmetros fundamentais da simulação como quantidade de ciclos de *clock* antes da injeção de falha, duração da injeção, duração do sinal que reinicia o sistema para uma nova campanha e geração do sinal quantizado para os registradores de dados. Além dos atributos citados, o controlador de injeção tem ainda a função de monitorar a quantidade de falhas injetadas e sinalizar o término de uma campanha. Assim, durante a elaboração dos testes somente é necessário instanciar a entidade de controle no mesmo nível hierárquico do circuito sob teste, conectar os sinais adequados e definir os parâmetros desejados para o teste. Conforme anteriormente mencionado o controlador de injeção possibilita dois modelos de injeção de falhas: (1) modelo de falhas de atraso em portas lógicas (*gate delay fault*) e (2) modelo de falhas de atraso em caminhos (*path delay fault*). Essa característica torna essencial a definição do controlador que será utilizado, pois a forma como a falha é injetada depende do modelo de falhas alvo.

No modelo de falhas de atraso em portas lógicas (*gate delay fault*), apenas uma das células (do conjunto de células com capacidade de injeção) é ativada em uma determinada campanha de injeção. Após a injeção de falha na célula, o sinal de controle de injeção é deslocado através da cadeia de registradores para ativar a injeção de falha na célula seguinte. Esse procedimento ocorre até que todas as células com capacidade de injeção instrumentadas em uma determinada *netlist* tenham sido selecionadas. Na figura 20 é

possível observar múltiplas injeções de falhas ocorrendo em sequência, e a cada nova injeção uma nova célula individual do conjunto é selecionada.

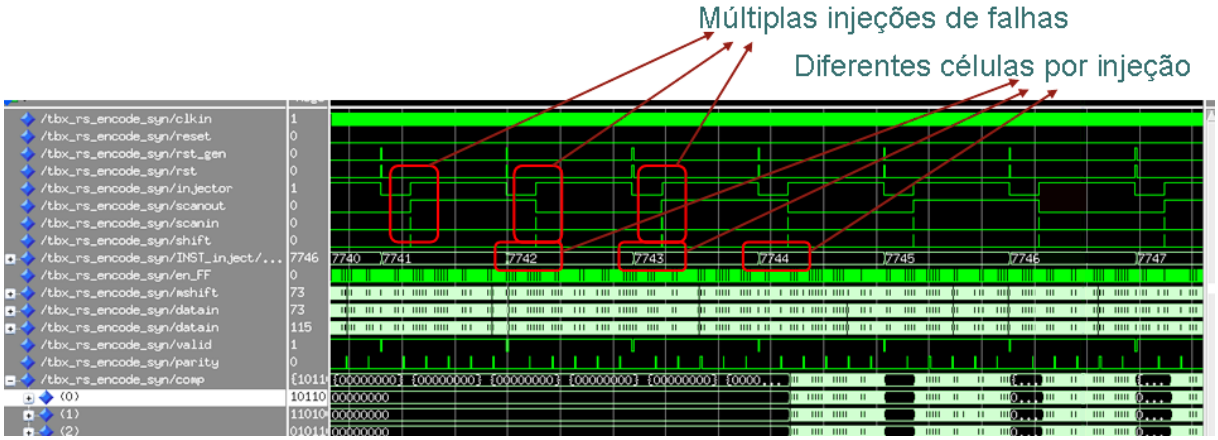


Figura 20: Múltiplas injeções de falhas com diferentes células selecionadas a cada injeção.

No modelo de falhas de atraso em caminhos (*path delay fault*), as células estão dispostas em uma sequência lógica combinacional, possuindo um atraso muito próximo a frequência de *clock* do sistema. Durante a injeção de falhas, o maior número de combinações possíveis de células pertencentes a esse caminho são utilizadas para avaliar a robustez do circuito frente à variações nas mesmas. Na figura 21 é possível observar o controlador de injeção aplicando uma variação no sinal *scanin* e, em paralelo, realizando a ativação do sinal de deslocamento (*shift*) responsável por conduzir esses valores presentes no *scanin* através da cadeia de registradores (descrita na seção 3.3).

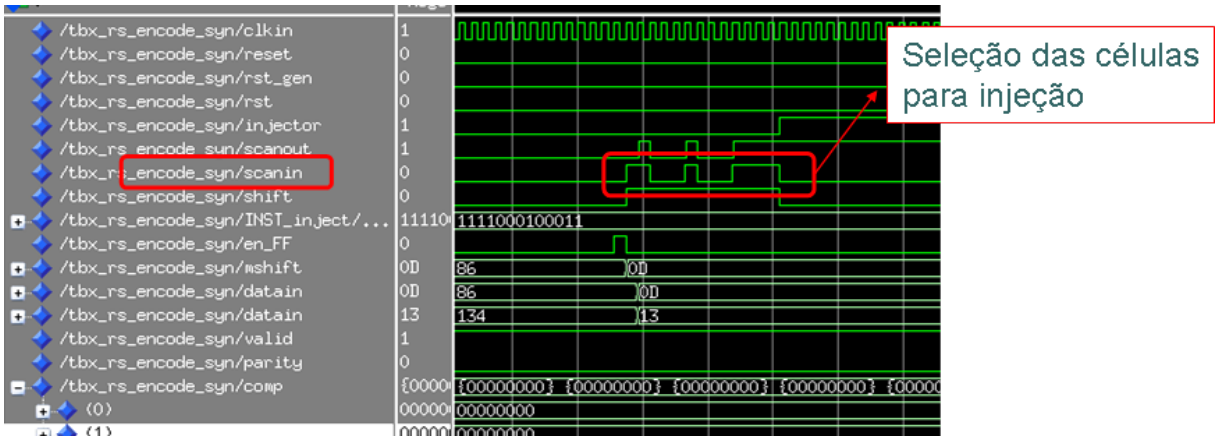


Figura 21: Seleção das células para injeção através da cadeia de registradores.

Na figura 22 observa-se os valores presentes na cadeia de registradores através da sequência de uns e zeros, representando um conjunto específico de células pertencentes a um caminho que terão seus valores modificados no momento em que o sinal de injeção

estiver ativo no sistema. Na figura 23 é possível ver o instante em que a injeção de falha é aplicada ao sistema e alguns ciclos de *clock* depois o momento em que o resultado da injeção de falha se manifesta na saída do sistema. É importante notar que a manifestação da falha é percebida com base na comparação *bit* a *bit* da saída de referência com a saída do sistema instrumentado.

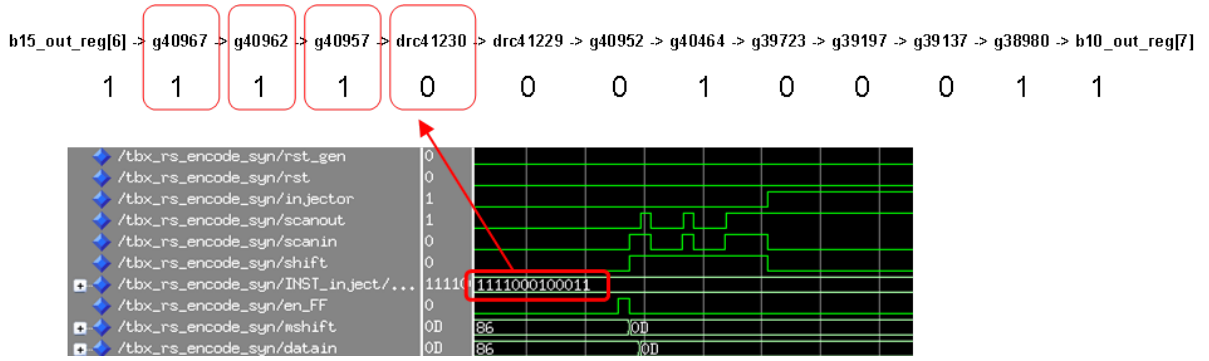


Figura 22: Escolha de cada uma das células pertencentes a um caminho para injeção.

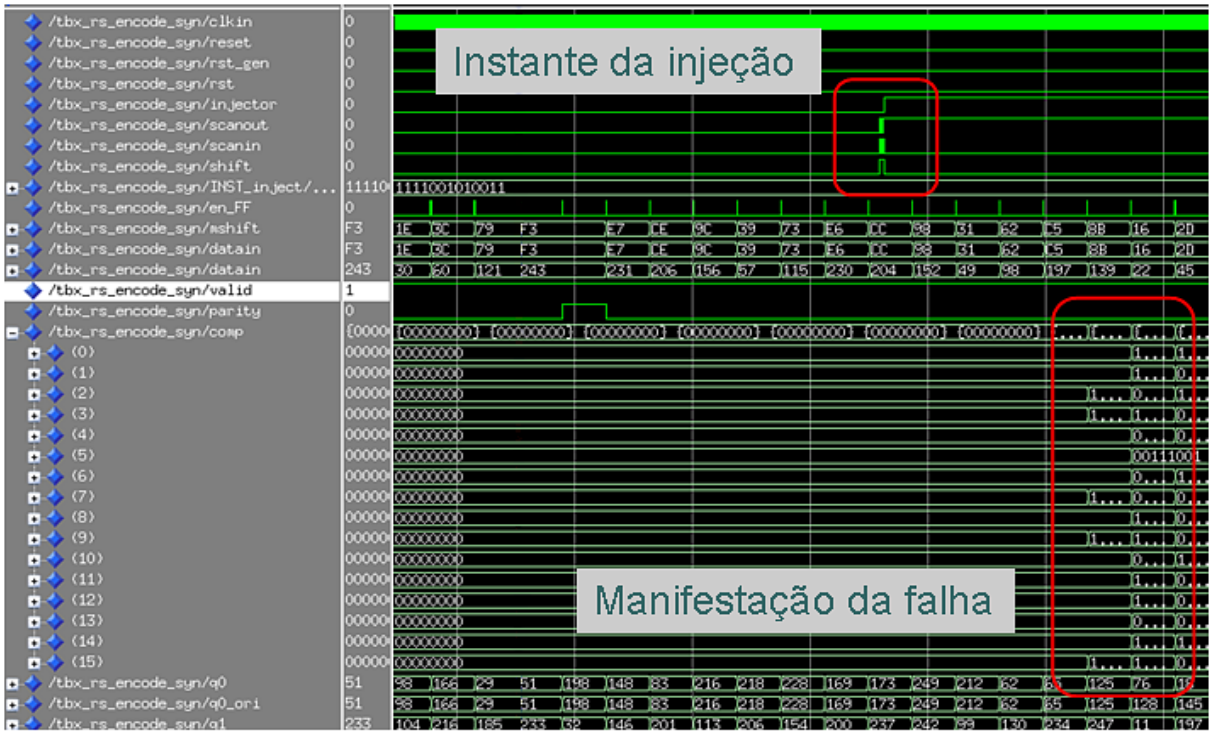


Figura 23: Instante da injeção com posterior manifestação da falha na saída do sistema.

Visto que os resultados das simulações confirmam o que foi proposto, fornecendo respostas consistentes e obedecendo as especificações pré-determinadas no capítulo 3, pode-se dar por concluída a validação da metodologia proposta e iniciar a etapa de avaliação da plataforma.

5 Avaliação

Nesta seção a etapa de avaliação da metodologia de injeção de falhas de atraso, sob a perspectiva de desempenho e funcionalidade durante as principais etapas de elaboração do circuito instrumentado é apresentada, fazendo uma análise dos *overheads* de tempo, área e performance da metodologia proposta. Durante a descrição dos estudos de caso serão enfatizados alguns aspectos como a quantidade de portas lógicas nos CIs utilizados, frequência máxima obtida, tempo médio de instrumentação do circuito, célula de maior atraso do projeto, entre outros atributos relevantes.

Nesta dissertação foram definidos dois estudos de caso para análise de desempenho da plataforma de injeção de falhas proposta, o codificador *Reed Solomon* (RS) fundamental em qualquer sistema de comunicação onde transmissão de informações com precisão é imprescindível, e o microcontrolador 8051 amplamente utilizado na indústria de sistemas de controle, telecomunicação, automação e robótica.

Os sistemas de correção de erros são amplamente utilizados para reduzir a probabilidade de erro e, por conseguinte, aumentar a precisão da transmissão. O codificador RS é um dos mais amplamente utilizado na correção de erros em blocos, e é capaz de detectar e corrigir vários erros, dando especial atenção a erros em rajada. Adotado nesta dissertação por ser computacionalmente eficiente e possuir uma arquitetura bastante uniforme no nível de transferência entre registradores, o codificador RS é composto basicamente de blocos de lógica combinacional responsáveis por realizar a adição ou subtração dos elementos, seguida de um registrador de dados com a função de multiplicar os coeficientes do polinômio sobre os elementos obtidos. Essa regularidade na descrição de sua arquitetura permite uma avaliação ampla do comportamento do circuito instrumentado e do resultado das saídas.

O microcontrolador 8051 utilizado como segundo estudo de caso é descrito em linguagem VHDL, totalmente sintetizável e modelando com bastante proximidade a implementação original da Intel (tendo 100% de instruções compatíveis). Essa compatibilidade

permite que o circuito seja alvo de compiladores 8051 C já existentes, podendo-se utilizar todo o conjunto de ferramentas previamente desenvolvido para o 8051 da Intel. Este estudo de caso de arquitetura heterogêneo nos permitirá realizar comparações significativas entre um sistema mais complexo (8051) e uma arquitetura regular (RS) avaliando o desempenho de ambos os circuitos sob falhas de atraso em portas lógicas (*gate delay fault*) e falhas de atraso em caminhos (*path delay fault*). O funcionamento e os parâmetros utilizados em cada um dos sistemas durante a avaliação da plataforma serão apresentados a seguir.

5.1 Estudo de Caso 1: Codificador *Reed Solomon*

O código RS é um código de correção de erros que consiste em uma técnica de adicionar informação redundante a um sinal transmitido, para que o receptor possa detectar e corrigir erros que possam vir a ocorrer durante a transmissão. Existem várias categorias de códigos corretores, o *Reed Solomon* se encaixa nos códigos de blocos, ou seja, a mensagem a ser transmitida é dividida em vários blocos separados de dados. Em cada bloco se tem a informação de paridade que junto com a mensagem forma a palavra de código. Um código RS pode ser classificado como RS(n,k), representando a palavra de código, composta pela mensagem original (k) mais os símbolos de paridade (n-k). A capacidade de correção de erros está diretamente relacionada com a metade do valor dos *bits* usados na paridade [37].

A codificação RS é amplamente utilizada em sistemas de armazenamento em massa para corrigir os erros em rajada associados a defeitos das mídias. Grande parte dos códigos de barras bidimensionais, como *PDF-417*, *MaxiCode*, *Datamatrix*, *QR Code*, fazem uso da correção de erros com o *Reed Solomon* para permitir uma leitura correta, mesmo se uma parte do código de barras estiver danificado.

O codificador RS utilizado como estudo de caso é o RS(38,22) [38], que possui 22 *bytes* de dados e 16 *bytes* de paridade. Os 22 *bytes* referentes aos dados de entrada são introduzidos serialmente no bloco a cada novo ciclo de *clock* sempre que o sinal indicador de dados válidos (*Data Valid*) estiver ativo. A figura 24 mostra que no momento em que todos os dados de entrada (*Data IN*) são entregues ao bloco é gerado um sinal de paridade (*Parity*) sinalizando que os dados presentes na saída (*Data OUT*) formam efetivamente a paridade restante do bloco.

Após verificado o funcionamento lógico do bloco o processo de instrumentação do

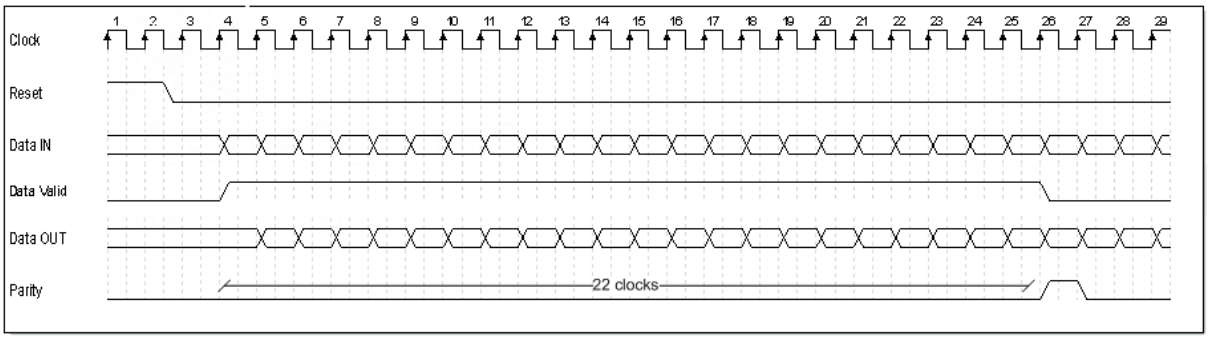


Figura 24: Diagrama de tempo do codificador *Reed Solomon*.

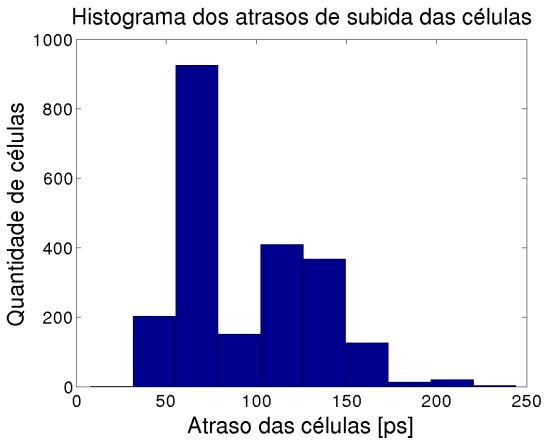
circuito acompanhando a sequência de etapas descritas no capítulo 3 é iniciado. Esse processo começa através da síntese do circuito que, por sua vez, utiliza o circuito descrito nos parágrafos anteriores, adiciona os sinais de *clock* e *reset* no arquivo de *constraints* estabelecendo a frequência máxima de operação desejada para o circuito, a fim de garantir que a ferramenta de síntese possa direcionar a escolha das células de modo a atingir a frequência de operação pretendida. Caso a ferramenta não consiga atingir a frequência determinada, a mesma deve ser reduzida, pois neste caso não pretende-se alterar a concepção já estabelecida para o circuito. Neste contexto, a frequência de operação alcançada para o circuito foi de 550MHz empregando um total de 2220 células em sua construção. Na tabela 3 a quantidade de pinos de entrada (PE) e saída (PS), bem como um resumo das principais características do circuito resultante do processo de síntese lógica (como quantidade de células combinacionais, células sequenciais e frequência máxima atingida), são mostrados.

Codificador *Reed Solomon*				
# Comb.	# FF	# PE	# PS	$f_{max}[MHz]$
2092	128	139	128	550

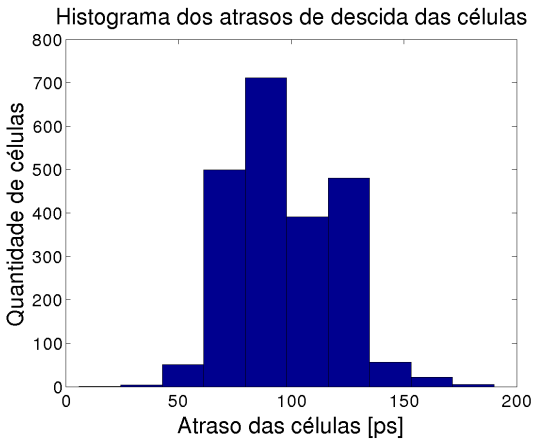
Tabela 3: Características da síntese lógica do circuito codificador *Reed Solomon*.

As figuras 25(a) e 25(b) mostram a distribuição dos atrasos de subida e descida, respectivamente, obtidos durante o processo de extração de atrasos descrito na seção 3.4.3. Em ambas as figuras é possível observar que a distribuição dos atrasos ocorre de forma distante do valor máximo de transição, ou seja, grande parte das células do projeto possuem atrasos muito menores que a célula de maior atraso. Para a borda de transição de subida o valor máximo do atraso entre todas as transições de subida é de 244ps e para a de descida 190ps. É possível notar na figura 25(a) uma quantidade reduzida de células próximas a 244ps, sendo grande parte dos atrasos concentrados entre 50 e 80ps. Com isso um número elevado de células é capaz de sofrer uma variação ampla dos valores de

transição dentro dos limites fornecidos de um sistema quantizado. Da mesma forma, para a distribuição apresentada na figura 25(b) os atrasos de descida estão com uma distribuição regular entre 60 e 140ps o que também fornece uma condição bastantes favorável para a alteração dos valores de transição de descida, tanto para adicionar como para reduzir os atrasos.



(a) Histograma dos atrasos de subida.



(b) Histograma dos atrasos de descida.

Figura 25: Histogramas das células obtidos para ambas as bordas de transição durante o processo de extração dos atrasos (seção 3.4.3).

Finalizado o processo de síntese é iniciado o processo de instrumentação do circuito através da sua *netlist* e do arquivo contendo a descrição dos atrasos do sistema (SDF). O arquivo da *netlist* mapeada do circuito *Reed Solomon* contêm $170Kbytes$ e o arquivo de atrasos contêm $995Kbytes$. Esses valores serão relevantes durante a análise do tempo consumido para instrumentar a *netlist*. A primeira etapa da instrumentação consiste em selecionar um conjunto restrito de células que será alvo da injeção de falhas no sistema.

Na tabela 4 é possível observar o desempenho da metodologia na instrumentação dos três aspectos distintos de seleção de células para injeção das falhas de atraso integrados à plataforma. Na segunda linha da tabela observa-se a extração de células por caminho crítico, a qual obteve 31 células sendo que este valor abrange 1.40% do total de células contidas no projeto (2220). Para realizar a instrumentação, do momento da extração das células do caminho crítico até o momento da geração da *netlist* alterada de acordo com as células escolhidas e seus arquivos de constantes, a ferramenta consumiu 800ms. Para a análise sobre *fanout* da célula, contida na terceira linha da tabela, a ferramenta extraiu 549 células lógicas atingindo 24.73% do total de células do projeto. Para executar o processo de instrumentação destas 549 células na *netlist* do circuito a plataforma consumiu 1.93 segundos. Na última linha da tabela está relacionada a extração de células por tipo

de função lógica executada, a célula selecionada neste caso foi uma NAND com duas entradas. O que corresponde a 45.68% do conjunto total de células contidas no circuito. Para realizar a instrumentação das 1014 células selecionadas a ferramenta consumiu 3.45 segundos.

Instrumentação	Tipo	# Células	% Células	Tempo [s]
Caminho crítico	–	31	1.40	0.8
Fanout da célula	X4	549	24.73	1.93
Função lógica	NAND2	1014	45.68	3.45

Tabela 4: Características da instrumentação do circuito codificador *Reed Solomon*.

5.2 Estudo de Caso 2: Microcontrolador 8051

O segundo estudo de caso adotado nesta dissertação é o microcontrolador 8051 de 8 *bits*. O 8051 foi projetado originalmente em 1980 pela Intel e ganhou grande popularidade desde a sua introdução. Suas centenas de derivados, fabricados por várias empresas diferentes incluem diversos periféricos embarcados, como conversores analógico-digitais, moduladores de largura de pulso e diversas interfaces de barramento, tudo isso custando apenas alguns dólares por CI. O Intel 8051 possui arquitetura *Harvard* capaz de endereçar 64K para programas e 64K para a memória de dados. O modelo é composto de cinco blocos principais detalhados a seguir [5]:

- Decodificador de instruções: O decodificador é responsável por representar as instruções não uniformes do 8051 em códigos enumerados. Este modelo é descrito como um fluxo de dados de execução de um bloco de lógica combinacional.
- Unidade lógica aritmética: Este bloco descreve o modelo de uma ALU 8051 que realiza as operações lógicas e aritméticas específicas do microcontrolador. Este modelo é descrito comportamentalmente como um bloco de lógica combinacional.
- Modelo de RAM: Este modelo descreve a memória RAM de 128 *bytes*, específica do 8051 com capacidade de endereçamento por *bits*. Este modelo é descrito comportamentalmente como um bloco lógico sequencial.
- Modelo de ROM: Modelo de até $64Kbytes$ de uma memória ROM, específica do 8051. Este modelo é gerado automaticamente em linguagem de descrição de *hardware* com base no código C utilizado.

- Bloco de controle: Modelo de um microcontrolador 8051 completo. Este modelo combina estruturalmente todos os sinais dos blocos acima mencionados, controlando o avanço dos estados do microcontrolador.

A figura 26 mostra o diagrama de blocos do circuito. No centro do diagrama é possível ver o bloco de controle responsável por coordenar o funcionamento do sistema entre todos os outros blocos, e na extremidade inferior do diagrama a interface do microcontrolador com o mundo externo através das portas de entrada e saída e dos sinais de controle da memória externa. É possível notar também que o modelo utilizado possui ainda um bloco de depuração descrito comportamentalmente como um bloco lógico sequencial com a finalidade de produzir um registro de cada instrução executada durante a simulação do sistema, o que auxilia no desenvolvimento e verificação dos programas elaborados.

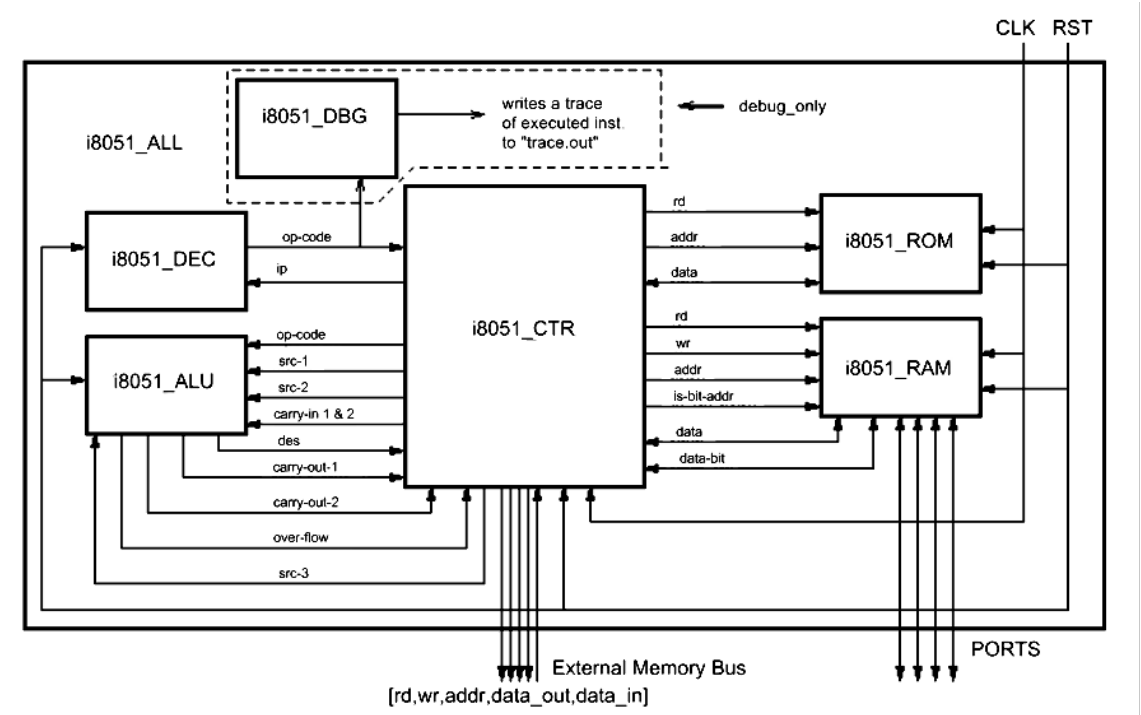


Figura 26: Diagrama de blocos do circuito microcontrolador 8051 [5].

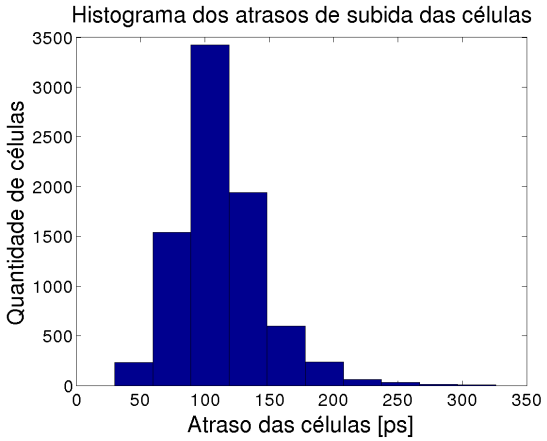
Seguindo o mesmo procedimento adotado no estudo de caso 1, o processo através da síntese ASIC é iniciado utilizando o circuito em VHDL do microcontrolador 8051. Com os sinais de *clock* e *reset* descritos no arquivo de *constraints* é estabelecido a frequência máxima de operação desejada. A frequência de operação alcançada para o circuito foi de 160MHz utilizando um total de 8084 células em sua concepção. A tabela 5 apresenta a quantidade de pinos de entrada e saída juntamente com o resumo das principais características do circuito resultantes do processo de síntese lógica. A frequência alcançada

para este circuito é menor devido a quantidade elevada de elementos sequenciais utilizados pelo algorítimo de síntese para representar o caminho de dados deste circuito. Para o primeiro estudo de caso são 128 elemento sequenciais com 139 pinos de entrada e para este estudo de caso são 1357 para 42 pinos de entrada, um valor dez vezes maior em termos de elementos sequenciais, isso se reflete então em caminhos de dados mais longos e consequentemente uma frequência de operação mais baixa.

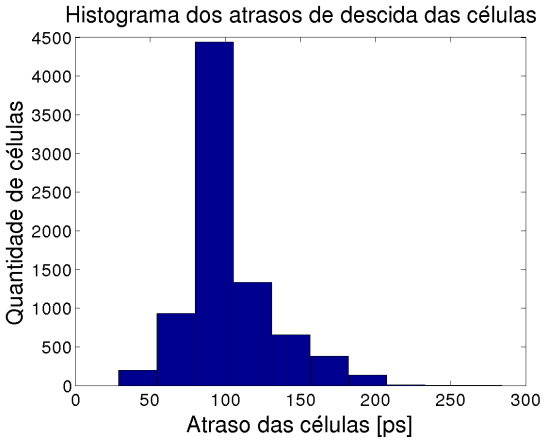
Microcontrolador 8051				
# Comb.	# FF	# PE	# PS	$f_{max}[MHz]$
6727	1357	42	58	160

Tabela 5: Características do síntese lógica do circuito microcontrolador 8051.

Nas figuras 27(a) e 27(b) é possível observar a distribuição dos atrasos de subida e descida, respectivamente, obtidos durante o processo de extração de atrasos da seção 3.4.3. Em ambas as figuras é possível notar que a distribuição dos atrasos está concentrada em um intervalo bastante estreito, ou seja, uma grande número de células com atrasos muito próximos. Na figura 27(a) os atrasos estão concentrados no intervalo de 80 e 130ps, sendo que o maior valor obtido para uma transição de subida deste circuito é de 326ps. Da mesma forma, para a figura 27(b) os atrasos estão concentrados no intervalo de 70 e 120ps, sendo que o maior valor obtido para uma transição de descida é de 284ps. Para ambas as bordas de transição há uma concentração de células bastante afastada tanto do valor máximo quanto do valor mínimo, novamente contribuindo para a utilização de um número bastante grande de células do circuito com um intervalo amplo de alterações percentuais dos valores de atraso.



(a) Histograma atrasos de subida.



(b) Histograma atrasos de descida.

Figura 27: Histogramas das células obtidos para ambas as bordas de transição durante o processo de extração dos atrasos (seção 3.4.3).

Com o processo de síntese concluído, o processo de instrumentação do circuito é iniciado utilizando a *netlist* e o arquivo de descrição dos atrasos do sistema (SDF). O arquivo *netlist* mapeado do circuito microcontrolador 8051 contêm $1.476 Kbytes$ e o arquivo de atrasos contêm $5.022 Kbytes$, estes valores como visto anteriormente, são relevantes ao traçar um paralelo entre a quantidade de células existentes no circuito e a velocidade de instrumentação da ferramenta. A seguir, seleciona-se um conjunto de células que será alvo da injeção de falhas no sistema.

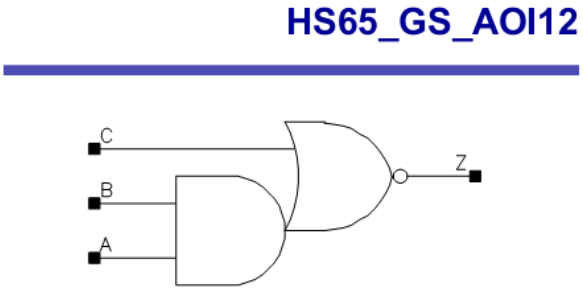


Figura 28: Célula lógica complexa AOI12.

Na tabela 6 é possível observar o desempenho da metodologia para os três critérios distintos de seleção de células para injeção das falhas de atraso integrados à plataforma. Na segunda linha da tabela observa-se a extração de células por caminho crítico, a qual obteve 55 células sendo que este valor abrange 0.68% do total de células contidas no projeto (8084). Para realizar o processo de instrumentação, com a geração da *netlist* alterada e os arquivos de constantes, a ferramenta consumiu 3.86 segundos. Para a análise sobre *fanout* da célula, contida na terceira linha da tabela, a ferramenta extraiu 627 células lógicas atingindo 7.76% do total de células do projeto. Para executar o processo de instrumentação destas 627 células na *netlist* do circuito a plataforma consumiu 9.42 segundos. Na última linha da tabela é apresentada a extração de células por tipo de função lógica executada, a célula selecionada neste caso foi uma AOI12, uma célula lógica complexa composta por uma AND de duas entradas conectada a uma OR também de duas entradas seguida de um inversor (conforme figura 28). Esta célula complexa corresponde a 10.34% do conjunto total de células contidas no circuito. Para realizar a instrumentação das 836 células selecionadas a ferramenta consumiu 10.65 segundos.

Instrumentação	Tipo	# Células	% Células	Tempo [s]
Caminho crítico	–	55	0.68	3.86
Fanout da célula	X7	627	7.76	9.42
Função lógica	AOI12	836	10.34	10.65

Tabela 6: Características da instrumentação do circuito microcontrolador 8051.

5.3 Análise do *Overhead* de Área

É possível estimar com certa precisão a quantidade de recursos utilizados no FPGA com base na quantidade de células instanciadas no projeto em ASIC, na quantidade de células instrumentadas e no tipo de arquitetura utilizada na instrumentação. Neste caso, deve-se levar em consideração que a plataforma proposta está sendo avaliada sobre um dispositivo Virtex 5 da Xilinx que possui em sua arquitetura LUTs de 6 entradas, o que se ajusta perfeitamente às células descritas nas bibliotecas VHDL, que possuem até 6 entradas em suas descrições.

A tabela 7 apresenta a quantidade de recursos de LUTs e FFs utilizados para cada uma das células em um projeto ASIC. Uma célula normal, ou seja, sem capacidade de injetar falhas no sistema consome 2 LUTs, uma para descrever seu comportamento lógico e outra para compor o registrador de deslocamento (SRL16). Se as bibliotecas descritas em VHDL possuíssem mais de 6 entradas em suas descrições seria necessário a utilização de mais uma LUT para representar seu comportamento lógico, ou seja, cada célula do projeto ASIC precisaria de 3 LUTs na arquitetura FPGA.

Célula	Borda	LUT 6 entradas		FFs	Total [LUTs]
		Lógica	SRL16		
Normal	Única	1	1	-	2
	Subida/Descida	2	2	-	4
Injeção de falhas	Única	1	2	1	3
	Subida/Descida	2	4	1	6

Tabela 7: Estimativa do uso de recursos de LUT e FF do dispositivo FPGA para cada uma das células empregadas na síntese ASIC.

Dependendo da arquitetura utilizada, com um único registrador de deslocamento para ambas as bordas de transição, ou com um registrador de deslocamento para cada borda de transição, a quantidade de recursos consumidos difere. Para uma célula sem injeção de falha e com arquitetura com valores diferentes para cada borda de transição o FPGA consome 2 LUTs como registradores de deslocamento (uma para cada transição), e 2 LUTs de lógica, uma para representar a função lógica da célula e outra para realizar a transição das saídas dos registradores de deslocamento no momento correto.

Para uma célula instrumentada com capacidade de injetar falhas no sistema, e com a arquitetura com bordas de transição diferentes para subida e descida é onde ocorre o maior consumo de recursos do FPGA. Neste caso, a descrição lógica da célula ocupa 1 LUT e cada um das bordas de transição (normal e sobre falha) ocupa uma LUT para compor

os registradores de deslocamento, somando um total de 4 LUTs. Além disso, mais uma LUT responsável por realizar a transição das saídas dos registradores de deslocamento no momento correto e multiplexar a saída de acordo com o momento da injeção de falha é utilizada.

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	130	64,000	1%	
Number used as Flip Flops	130			
Number of Slice LUTs	671	64,000	1%	
Number used as logic	671	64,000	1%	
Number using O6 output only	671			
Number of occupied Slices	391	16,000	2%	
Number of LUT Flip Flop pairs used	673			
Number with an unused Flip Flop	543	673	80%	
Number with an unused LUT	2	673	1%	
Number of fully used LUT-FF pairs	128	673	19%	
Number of unique control sets	1			
Number of slice register sites lost to control set restrictions	2	64,000	1%	
Number of bonded IOBs	267	680	39%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Average Fanout of Non-Clock Nets	5.14			

(a) Circuito Original

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	169	64,000	1%	
Number used as Flip Flops	169			
Number of Slice LUTs	4,309	64,000	6%	
Number used as logic	2,095	64,000	3%	
Number using O6 output only	2,095			
Number used as Memory	2,214	19,840	11%	
Number used as Shift Register	2,214			
Number using O6 output only	2,214			
Number of occupied Slices	2,549	16,000	15%	
Number of LUT Flip Flop pairs used	4,340			
Number with an unused Flip Flop	4,171	4,340	96%	
Number with an unused LUT	31	4,340	1%	
Number of fully used LUT-FF pairs	138	4,340	3%	
Number of unique control sets	3			
Number of slice register sites lost to control set restrictions	5	64,000	1%	
Number of bonded IOBs	272	680	40%	

(b) Circuito Instrumentado

Figura 29: Resultado da síntese em FPGA do circuito *Reed Solomon* em um dispositivo Virtex 5.

Na figura 29 é possível observar o resultado da síntese de ambos os circuitos *Reed Solomon* (original e instrumentado) em um dispositivo Virtex 5 da Xilinx. Na figura 29(b) a instrumentação do circuito é referente às células pertencentes ao caminho crítico extraído do circuito original com arquitetura utilizando valores diferentes de transição para subida e descida.

A tabela 8 apresenta um comparativo que relaciona a quantidade de recursos de LUTs e FFs utilizados na concepção do circuito original e a quantidade de recursos adicionais empregados na construção do circuito instrumentado. É possível observar que o circuito instrumentado tem um incremento de 1.3 vezes na quantidade de FFs consumidos e 6.4 vezes na quantidade de LUTs consumidas em sua implementação.

Na figura 30 é possível observar o resultado da síntese de ambos os circuitos microcontrolador 8051 (original e instrumentado) em um dispositivo Virtex 5 da Xilinx. Na figura 30(b) a instrumentação do circuito é referente as células pertencentes ao caminho crítico extraído do circuito original com arquitetura utilizando valores diferentes de transição para subida e descida.

Recursos	Original	Instrumentado	Incremento
FFs	130	169	1.30x
LUTs	671	4309	6.42x

Tabela 8: Utilização dos recursos de LUT e FF para o circuito *Reed Solomon* original e instrumentado.

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	1,356	64,000	2%	
Number used as Flip Flops	1,355			
Number used as Latch-thrus	1			
Number of Slice LUTs	2,246	64,000	3%	
Number used as logic	2,244	64,000	3%	
Number using O6 output only	2,216			
Number using O5 output only	7			
Number using O5 and O6	21			
Number used as exclusive route-thru	2			
Number of route-thrus	9			
Number using O6 output only	9			
Number of occupied Slices	1,135	16,000	7%	
Number of LUT Flip Flop pairs used	3,152			
Number with an unused Flip Flop	1,796	3,152	56%	
Number with an unused LUT	906	3,152	28%	
Number of fully used LUT-FF pairs	450	3,152	14%	
Number of unique control sets	367			
Number of slice register sites lost to control set restrictions	717	64,000	1%	
Number of bonded IOBs	100	680	14%	

(a) Circuito Original

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	1,413	64,000	2%	
Number used as Flip Flops	1,413			
Number of Slice LUTs	13,610	64,000	21%	
Number used as logic	6,715	64,000	10%	
Number using O6 output only	6,715			
Number used as Memory	6,895	19,840	34%	
Number used as Shift Register	6,895			
Number using O6 output only	6,895			
Number of occupied Slices	4,007	16,000	25%	
Number of LUT Flip Flop pairs used	13,619			
Number with an unused Flip Flop	12,206	13,619	89%	
Number with an unused LUT	9	13,619	1%	
Number of fully used LUT-FF pairs	1,404	13,619	10%	
Number of unique control sets	4			
Number of slice register sites lost to control set restrictions	4	64,000	1%	
Number of bonded IOBs	105	680	15%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Average Fanout of Non-Clock Nets	2.14			

(b) Circuito Instrumentado

Figura 30: Resultado da síntese em FPGA do circuito microcontrolador 8051 em um dispositivo Virtex 5.

A tabela 9 apresenta um comparativo que relaciona a quantidade de recursos de LUTs e FFs utilizados na concepção do circuito original e a quantidade de recursos adicionais empregados na construção do circuito instrumentado. Observa-se que o circuito do microcontrolador 8051 por possuir uma quantidade maior de células em sua concepção, durante a instrumentação não representa uma valor expressivo (4%) na adição de FFs no resultado final da síntese. Na tabela 9 é possível ver ainda que o circuito instrumentado consome 6 vezes mais LUTs que a implementação original, um valor ligeiramente menor que para o circuito *Reed Solomon*.

Falhas de atraso podem ser abordadas adotando diferentes estratégias. Nos testes em que o circuito é executado na velocidade projetada através de um dispositivo externo específico raramente é um sistema economicamente viável. Engenheiros de teste podem desenvolver padrões de teste que têm como alvo os dois modelos de falhas relacionados ao

Recursos	Original	Instrumentado	Incremento
FFs	1.356	1.413	1.04x
LUTs	2.246	13.610	6.02x

Tabela 9: Utilização dos recursos de LUT e FF para o circuito microcontrolador 8051 original e instrumentado.

tempo mais aceitos: falhas de transição e falhas de atraso em caminho. Esta combinação oferece uma ampla cobertura de teste e localização de defeitos distribuídos que impedem um dispositivo de operar corretamente em sua velocidade nominal. Porém ainda possuem a desvantagem de serem gerados em um computador, consumindo muito tempo para produzir os padrões de teste. Em muitos casos não são levados em consideração as quedas de tensão ao longo da trilha de roteamento e o acoplamento entre trilhas adjacentes.

Metodologias de auto teste estruturais, como BIST [39][40] têm sido considerados como uma alternativa, mas apresentam desvantagens como a exigência de modificação do circuito original (mesmo que utilizando somente a periferia do circuito). Essas metodologias muitas vezes acabam por adicionar uma área excessiva de prototipação e apresentam um decréscimo de desempenho, além disso, diversos problemas relacionados ao *clock* do sistema podem surgir devido a introdução das estruturas de BIST. Diferente da metodologia apresentada neste trabalho, a qual pode ser integrada ao fluxo do projeto em diversas etapas diferentes da elaboração do dispositivo avaliando os pontos mais propensos a falhas, porém sem a necessidade de agregar o *hardware* adicional ao projeto final.

6 Conclusões

Esta dissertação de mestrado, que teve como principais objetivos a especificação, implementação, validação e avaliação de uma metodologia de injeção de falhas de atraso em ASIC, foi motivada sobre o contexto atual associado ao desenvolvimento de CIs e aos problemas relacionados à evolução da tecnologia em termos de densidade e proximidade das interconexões, onde pequenas variações no processo de litografia podem ocasionar defeitos no circuito final. Considerando que alguns dos defeitos resultantes da variação do processo de produção não manifestam-se a partir de uma alteração lógica no comportamento do circuito final, e sim através da alteração na temporização do circuito. Em vista disso, foi elaborada uma metodologia que empregasse a estrutura lógica do sistema desenvolvido no contexto ASIC juntamente com seus atrasos, de modo a introduzir essas variações do processo de produção na emulação a partir do uso de FPGA e observar o comportamento do sistema na presença dessas variações.

Com o intuito de conservar a estrutura lógica e consequentemente os respectivos atrasos do circuito gerados durante a síntese ASIC, foi desenvolvido um conjunto de bibliotecas responsáveis por transpor o comportamento temporal do sistema para um contexto de *hardware* reconfigurável. O desenvolvimento da biblioteca ocorreu em duas etapas distintas, uma para incluir os atrasos extraídos do arquivo SDF e outra para adicionar a capacidade de injeção de falhas. O desenvolvimento de ambas as bibliotecas em linguagem de descrição de *hardware* apresenta uma forma flexível para adicionar as diversas células presentes em uma biblioteca tecnológica. Apesar da descrição das bibliotecas ocorrerem apenas uma vez a cada salto tecnológico (considerando que haverá uma adição ou remoção de funções lógicas a cada evolução) a flexibilidade da descrição em *hardware* permite a alteração da arquitetura utilizada para modelar as falhas de atraso sem a necessidade de iterar sobre cada uma das células da biblioteca.

A arquitetura de injeção de falhas apresentada nessa dissertação possui duas alternativas de implementação, uma com um único atraso para ambas as bordas de transição e outra com um valor singular para cada uma das bordas de transição, sendo que a escolha

por uma das arquiteturas disponíveis representa um compromisso direto entre a área de prototipação disponível e o nível de detalhamento desejado nos ensaios. A fim de minimizar a área e optimizar o desempenho de um projeto baseado em *hardware* reconfigurável, é importante assegurar que o mesmo esta efetivamente utilizando os recursos disponíveis no *slice* do FPGA. Durante a elaboração da arquitetura de injeção de falhas buscou-se impor, por meio da descrição da entidade responsável por representar os atrasos do circuito, a utilização de recursos específicos do FPGA. Ao impor a utilização dos recursos do *slice* através de uma descrição específica sugerida nas diretrizes de implementação da Xilinx, foi possível garantir a mínima utilização de recursos necessários durante a implementação da arquitetura de injeção de falhas.

Todo os estágios descritos durante a concepção da proposta, capítulo 3, (com exceção do elaboração de bibliotecas, seção 3.3) desenvolvida ao longo dessa dissertação foram elaborados em uma linguagem de alto nível (TCL). A utilização dessa linguagem de alto nível permitiu que houvesse uma transcrição, quase que direta da concepção idealizada para a implementação dos algoritmos, fornecendo um código de fácil compreensão o que facilita também a manutenção do código a longo prazo. Ao iniciar o desenvolvimento não houve necessidade de configuração do ambiente pois a linguagem TCL já vem integrada em grande parte das distribuições Linux, e por ser uma linguagem interpretada não há necessidade de compilação dos arquivos. Ao finalizar a etapa de instrumentação (seção 3.4) foi possível verificar que o desempenho da ferramenta descrita em TCL seria bastante satisfatório, e ao realizar a etapa de avaliação confirmou-se que a performance da ferramenta é realmente expressiva na instrumentação de CIs orientados à injeção de falhas de atraso.

A metodologia empregada possibilita a injeção de falhas de atraso em dois modelos, falhas de atraso em portas lógicas (*gate delay fault*) e falhas de atraso em caminhos (*path delay fault*). Para isso, são extraídas as células do projeto original utilizando determinadas características ao conjunto de células que se deseja analisar. Outros modelos de falhas podem ser utilizados, bastando para isso que se modifique o método de seleção das células e se verifique que a arquitetura utilizada é adequada ao novo modelo implementado.

Uma das etapas mais importantes do processo de instrumentação é a extração dos atrasos do arquivo SDF gerado durante a síntese ASIC. Essa etapa é essencial pois seleciona os atrasos, gerados com base na biblioteca tecnológica utilizada, de cada célula presente no projeto. Esse arquivo SDF pode ser obtido em diversas etapas no decorrer da elaboração do CI, sempre garantidamente com o mesmo formato utilizado na espe-

cificação de seu conteúdo, alterando entre as etapas somente os valores que descrevem os atrasos das transições, representando os atrasos com maior precisão a medida que o desenvolvimento é conduzido para os estágios finais de produção das máscaras. Dado que o tempo até a produção das máscaras pode ser extenso, a utilização da plataforma desenvolvida nessa dissertação em estágios anteriores do projeto pode ser uma alternativa para detectar precocemente problemas de temporização no circuito.

Após extrair os atrasos do arquivo SDF, esses eram quantizados de acordo com o resolução (e consequentemente a largura do registrador de deslocamento) desejada. No decorrer das simulações foi possível observar com sucesso a transposição desses valores quantizados para os registradores de deslocamento durante o funcionamento do circuito. Por ser gerado automaticamente através da ferramenta de síntese (XST) e por conter apenas os elementos lógicos disponíveis no FPGA, o arquivo usado na simulação funcional garante uma equivalência excelente entre o circuito simulado e o circuito final gravado em FPGA. Assim, ao observar o funcionamento correto do sistema neste nível de abstração, descrito através do arquivo gerado após a síntese lógica, é possível garantir a execução correta a nível de execução do *hardware* reconfigurável.

Para concluir, a metodologia proposta nesta dissertação apresenta um *overhead* de tempo de aproximadamente 1% durante a etapa de *front end* do projeto, sendo capaz de extrair os atrasos e instrumentar uma *netlist* de alguns *Megabytes* em poucos segundos. Em relação ao *overhead* de área, a metodologia emprega cerca de 6 vezes mais componentes combinacionais, e consequentemente para um CI no contexto de um *hardware* reconfigurável que originalmente ocupe 5000 LUTs é necessário uma FPGA de 30000 LUTs para realizar a emulação dos atrasos agregados.

Os resultados obtidos demostram que a área consumida com a metodologia empregada e bastante grande, porém ao contrapor o compromisso entre a área utilizada e a resolução temporal que pode ser obtida através do uso da metodologia proposta, pode-se afirmar que os resultados são expressivos na avaliação da robustez de ASICs complexos frente a falhas de atraso, bem como na cobertura de falhas de metodologias de teste e técnicas de tolerância a falhas. É importante destacar que até o momento da conclusão dessa dissertação, não ha técnica de injeção de falhas de atraso semelhante que utilize atrasos quantizados a partir dos atrasos originais gerados na síntese ASIC na etapa de emulação do circuito em FPGA.

6.1 Trabalhos Futuros

Com a continua redução do tamanhos dos circuitos integrados digitais, a variação dos parâmetros do dispositivo tornou-se uma grande preocupação. Durante a fabricação, variações dos parâmetros do processo são inevitáveis. Assim, os parâmetros de desempenho, como o consumo de energia ou frequência de operação diferem dos valores para os quais foram projetados. Após fabricado, o desempenho do circuito degrada ao longo do tempo devido a vários efeitos de envelhecimento que causam um desvio dos parâmetros originais do dispositivo. O NBTI (*Negative Bias Temperature Instability*) e o HCI (*Hot Carrier Injection*) são considerados como sendo os efeitos de envelhecimento dominantes nas tecnologias atuais. O NBTI degrada as características dos transistores PMOS e seu impacto pode ser modelado por um aumento na magnitude das tensões de limiar. O HCI é responsável por degradar a corrente de dreno de ambos transistores, PMOS e NMOS. A análise de tempo para circuitos digitais complexos só pode ser realizada a nível de portas lógicas devido a razões de complexidade. Mas há uma falta de métodos que possam analisar o impacto de variações do processo e os efeitos do envelhecimento sobre o desempenho do circuito [41].

Como trabalho futuro a plataforma proposta nessa dissertação pode vir a ser uma metodologia alternativa na análise das variações do processo de produção e os efeitos resultantes do envelhecimento em CIs. Para que isto aconteça é necessário que sejam desenvolvidos esforços no sentido de caracterizar, através de simulações em HSPICE, o comportamento dos transistores sobre efeitos do NBTI e do HCI para que se desenvolva um modelo matemático o qual possa ser utilizado para representar de forma efetiva a degradação dos atrasos do sistema com o passar do tempo, e esses valores possam ser transferidos para o contexto da metodologia desenvolvida nessa dissertação.

Referências

[1] V. Agrawal, *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*. Frontiers in Electronic Testing, Springer, 2000.

[2] J. H. Patel, "A tutorial on delay fault testing." Department of Electrical and Computer Engineering, University of Illinois Urbana-Champaign, 2005.

[3] M. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, no. April, pp. 75–82, 1997.

[4] H. Bhatnagar, *Advanced ASIC Chip Synthesis Using Synopsys® Design Compiler® Physical Compiler® and PrimeTime®*. Springer, 2001.

[5] F. Vahid, "Synthesizable vhdl model of 8051." `<http://www.cs.ucr.edu/~dalton/i8051/i8051syn/>`, University of California, Dept. of Computer Science, Riverside, CA, 2001, Acesso em: 15 jan. 2013.

[6] H. Ziade, R. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.

[7] L. Entrena, "Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection," *Computers, IEEE ...*, vol. 61, no. 3, pp. 313–322, 2012.

[8] M. García Valderas, L. Entrena, R. Fernández Cardenal, C. López Ongil, and M. Portela García, "SET Emulation Under a Quantized Delay Model," *Journal of Electronic Testing*, vol. 25, pp. 107–116, Dec. 2008.

[9] A. Avezienis, J. Laprie, and B. Randell, "Fundamental Concepts of Computer System Dependability," *Workshop on Robot Dependability: Technological Challenge of Dependable Robots in Human Environments*, pp. 1–16, 2001.

[10] A. Avizienis and J. Laprie, "Basic concepts and taxonomy of dependable and secure computing," *... Secure Computing, ...*, vol. 1, no. 1, pp. 11–33, 2004.

[11] B. Parhami, "A multi-level view of dependable computing," *Computers & electrical engineering*, vol. 20, no. 4, pp. 347–368, 1994.

[12] B. Parhami, "From defects to failures: a view of dependable computing," *ACM SIGARCH Computer Architecture News*, pp. 157–168, 1988.

[13] C. Stroud, *A Designer's Guide to Built-in Self-Test*. Frontiers in Electronic Testing, Springer, 2002.

[14] A. Krstić and K. Cheng, *Delay fault testing for VLSI circuits*, vol. 14 of *Frontiers in Electronic Testing*. Boston, MA: Springer US, 1998.

[15] J. Clark and D. Pradhan, "Fault injection: A method for validating computer-system dependability," *Computer*, no. June, 1995.

[16] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," in *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*, pp. 340 –347, jun 1989.

[17] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: a methodology and some applications," *Software Engineering, IEEE Transactions on*, vol. 16, pp. 166 –182, feb 1990.

[18] G. Choi and R. Iyer, "Focus: an experimental environment for fault sensitivity analysis," *Computers, IEEE Transactions on*, vol. 41, pp. 1515 –1526, dec 1992.

[19] H. M. Mrio, H. Madeira, F. Moreira, and J. G. Silva, "Rifle: A general purpose pin-level fault injector," in *Proc. First European Dependable Computing Conference (EDCC-1*, pp. 199–216, Springer Verlag, 1994.

[20] J. Carreira, H. Madeira, and J. Silva, "Xception: a technique for the experimental evaluation of dependability in modern computers," *Software Engineering, IEEE Transactions on*, vol. 24, pp. 125 –136, feb 1998.

[21] D. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. Iyer, "Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International*, pp. 91 –100, 2000.

[22] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "Goofi: generic object-oriented fault injection tool," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pp. 83 –88, july 2001.

[23] J. Duraes and H. Madeira, "Emulation of software faults: A field data study and a practical approach," *Software Engineering, IEEE Transactions on*, vol. 32, pp. 849 –867, nov. 2006.

[24] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into vhdl models: the mefisto tool," in *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pp. 66 –75, jun 1994.

[25] V. Sieh, O. Tschache, and F. Balbach, "VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions," *Fault-Tolerant Computing, ...*, pp. 2–6, 1997.

[26] C. Rousselle, M. Pflanz, A. Behling, T. Mohaupt, and H. Vierhaus, "A register-transfer-level fault simulator for permanent and transient faults in embedded processors," in *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, p. 811, 2001.

[27] W. Chao, F. Zhongchuan, C. Hongsong, and C. Gang, "Fsfi: A full system simulator-based fault injection tool," in *Instrumentation, Measurement, Computer, Communication and Control, 2011 First International Conference on*, pp. 326 –329, oct. 2011.

[28] D. D. Andres, J. C. Ruiz, D. Gil, and P. Gil, "Fades: a fault emulation tool for fast dependability assessment," in *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pp. 221 –228, dec. 2006.

[29] H. Zheng, L. Fan, and S. Yue, "Fitvs: A fpga-based emulation tool for high-efficiency hardness evaluation," in *Parallel and Distributed Processing with Applications, 2008. ISPA '08. International Symposium on*, pp. 525 –531, dec. 2008.

[30] M. Jeitler, M. Delvai, and S. Reichor, "Fuse - a hardware accelerated hdl fault injection tool," in *Programmable Logic, 2009. SPL. 5th Southern Conference on*, pp. 89 –94, april 2009.

[31] M. Sharma and J. Patel, "Testing of critical paths for delay faults," *Test Conference, 2001. Proceedings. ...*, pp. 634–641, 2001.

[32] L. Wang, C. Stroud, and N. Touba, *System-on-Chip Test Architectures: Nanometer Design for Testability*. Systems on Silicon, Elsevier Science, 2010.

[33] B. B. Welch, *Practical programming in tcl and tk*. Upper Saddle River (N. J.): Prentice Hall, 1995.

[34] J. Bhasker, *Static timing analysis for nanometer designs a practical approach*. New York: Springer, 2009.

[35] T. Williams, B. Underwood, and M. Mercer, "The interdependence between delay-optimization of synthesized networks and testing," in *Design Automation Conference, 1991. 28th ACM/IEEE*, pp. 87 –92, june 1991.

[36] W. Qiu, X. Lu, J. Wang, Z. Li, D. Walker, and W. Shi, "A statistical fault coverage metric for realistic path delay faults," in *VLSI Test Symposium, 2004. Proceedings. 22nd IEEE*, pp. 37 – 42, april 2004.

[37] A. de Carvalho Neto, "Reed solomon e viterbi." `<http://www.teleco.com.br/tutoriais/tutorialtvdentr1/pagina_2.asp>`, São José dos Campos, SP, 2011, Acesso em: 03 jan. 2013.

[38] R. Pathak, "Reed solomon encoder synthesizable ip core." `<http://opencores.org/project,rsencoder>`, 2009, Acesso em: 22 dez. 2012.

[39] N. Das, P. Roy, H. Rahaman, and P. Dasgupta, "Build-in-Self-Test of FPGA for diagnosis of delay fault," *2011 3rd Asia Symposium on Quality Electronic Design (ASQED)*, pp. 54–61, July 2011.

[40] M. Abramovici and C. Stroud, "BIST-based delay-fault testing in FPGAs," *Journal of electronic testing*, pp. 4–7, 2003.

[41] D. Lorenz, G. Georgakos, and U. Schlichtmann, "Aging analysis of circuit timing considering nbti and hci," in *On-Line Testing Symposium, 2009. IOLTS 2009. 15th IEEE International*, pp. 3–8, 2009.