UNIVERSIDADE FEDERAL DO RIO GRANDE
CENTRO DE CIÊNCIAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO
CURSO DE MESTRADO EM ENGENHARIA DE COMPUTAÇÃO

Dissertação de Mestrado

PROJETO DE UM GERADOR DE CIRCUITOS PARA VALIDAÇÃO DE PORTAS LÓGICAS SEQUENCIAIS

Helder Henrique Avelar

Rio Grande, 2015

UNIVERSIDADE FEDERAL DO RIO GRANDE
CENTRO DE CIÊNCIAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO
CURSO DE MESTRADO EM ENGENHARIA DE COMPUTAÇÃO

Dissertação de Mestrado

PROJETO DE UM GERADOR DE CIRCUITOS PARA VALIDAÇÃO DE PORTAS LÓGICAS SEQUENCIAIS

Helder Henrique Avelar

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal do Rio Grande, como requisito para a obtenção do grau de Mestre em Engenharia de Computação

Orientador: Prof. Dr. Paulo Francisco Butzen

Rio Grande, 2015

UNIVERSIDADE FEDERAL DO RIO GRANDE

Centro de Ciências Computacionais

Programa da Pós-Graduação em Computação

Curso de Mestrado em Engenharia de Computação

ATA DE SESSÃO DE DEFESA DE DISSERTAÇÃO DE MESTRADO

Ata No. ____/2015

Na data de 29 de maio de 2015, às 14 horas, ocorreu a Sessão de Defesa de Dissertação de Mestrado de Helder Henrique Avelar, que apresentou a dissertação intitulada Projeto de um Gerador de Circuitos para Validação de Portas Lógicas Sequenciais, realizada sob a orientação do Prof. Dr. Paulo Francisco Butzen. A banca examinadora foi constituída pelos Profs. Dr. Tiago Roberto Balen (UFRGS), Dra. Cristina Meinhardt (FURG) e Dr. Vagner Santos da Rosa (FURG), sob a presidência do orientador. Após a apresentação do trabalho, a banca arguiu o candidato e, a seguir, deliberou pela

(X) aprovação da Dissertação

( ) aprovação da Dissertação, sugerindo modificações no texto

( ) reprovação da Dissertação

Rio Grande, 29 de maio de 2015

Prof. Dr. Tiago Roberto Balen

Profa. Dra. Cristina Meinhardt

Prof. Dr. Vagner Santos da Rosa

Prof. Dr. Paulo Francisco Butzen
Orientador

Ficha catalográfica

A948c Avelar, Helder Henrique.
Projeto de um gerador de circuitos para validação de portas Lógicas sequenciais / Helder Henrique Avelar. – 2016.
65 f.

Dissertação (mestrado) – Universidade Federal do Rio Grande – FURG, Programa de Pós-graduação em Engenharia da Computação, Rio Grande/RS, 2016.
Orientador: Dr. Paulo Francisco Butzen.

1. Circuitos sequenciais 2. Validação lógica 3. Flip-flops 4. Autoteste integrado 5. Fluxo de células padrão I. Butzen, Paulo Francisco II. Título.

CDU 004:621.3

Catalogação na Fonte: Bibliotecário Me. João Paulo Borges da Silveira CRB 10/2130

RESUMO

AVELAR, Helder Henrique. PROJETO DE UM GERADOR DE CIRCUITOS PARA VALIDAÇÃO DE PORTAS LÓGICAS SEQUENCIAIS. 2015. 65 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal do Rio Grande, Rio Grande.

Latches e *flip-flops* são componentes de fundamental importância para o projeto de circuitos integrados. A maior parte dos circuitos integrados atuais são projetados por meio do fluxo de células padrão. Essa metodologia utiliza como componentes básicos portas lógicas previamente projetadas e caracterizadas de uma biblioteca de células padrão. *Latches* e *Flip-Flops* estão presentes nesta biblioteca. Com a constante diminuição na dimensão dos transistores, novas bibliotecas são necessárias a cada novo nodo tecnológico. As portas lógicas, incluindo os elementos sequenciais, precisam ser re-projetados e re-validados. Considerando os altos custos inerentes à validação em silício de circuitos sequenciais, surge a necessidade do desenvolvimento de técnicas que tornem esse processo mais simples e barato. Esse trabalho propõe um método para geração automática de circuitos de autoteste para células sequenciais em bibliotecas de células, por meio do desenvolvimento de vetores de teste que possam ser aplicados utilizando máquinas de estados finitos.

Palavras-chave: Circuitos sequenciais, validação lógica, *flip-flops,* autoteste integrado, fluxo de células padrão.

ABSTRACT

AVELAR, Helder Henrique. CIRCUIT GENERATOR FOR SEQUECIAL LOGIC GATES VALIDATION. 2015. 65 p. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal do Rio Grande, Rio Grande.

Latches and flip-flops are components with fundamental importance to the design of integrated circuits. Most of modern integrated circuits are designed through the standard cell design flow. This methodology uses, as basic components, pre-designed and pre-characterized logic gates that are in a standard cell library. Latches and flip-flops are present in this library. With the constant transistor scaling, new libraries are requested for each new technology node. The logic gates, including the sequential elements, have to be re-designed and re-validated. Considering the inherent high costs of sequential circuit on-silicon validation, it is necessary to develop techniques that make it simpler and cheaper. This work proposes a method for automatic generation of a built-in self-test for sequential cells in cell libraries, using test vectors that can be implemented in finite states machines.

Keywords: Sequencial circuits, logic validation, flip-flops, built-in self-test, standard cell flow.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ASIC : *Application Specific Integrated Circuit* (CI para aplicação específica)

BIST : *Built-in Self-Test* (autoteste integrado)

CI : Circuito Integrado

CMOS : *Complementary MOS*

DRC : *Design Rules Check* (Verificação das regras de projeto)

DUT : *Design Under Test* (Circuito Sob Teste)

EDA : Eletronic Design Automation (Automação de projetos de eletrônica)

FPGA : *Field-Programmable Gate Array*

FSM : *Finite States Machine* (máquina de estados finitos)

FURG : Universidade Federal do Rio Grande

HDL : *Hardware Description Language* (linguagem de descrição de *hardware*)

LVS : Layout Versus Schematic (comparação lógica entre o leiaute e o esquemático do circuito)

MOS : *Metal-Oxide-Semiconductor*

PDK : *Process Development Kit*

RTL : *Register-Transfer-Level* (a nível de registradores)

SPICE : *Simulation Program with Integrated Circuit Emphasis* (programa de simulações com ênfase em CIs)

UFRGS : Universidade Federal do Rio Grande do Sul

SUMÁRIO

1. INTRODUÇÃO

A microeletrônica evoluiu de tal forma que a humanidade se tornou dependente de dispositivos semicondutores em diversos âmbitos da sociedade. Em termos de pesquisa, aplicação e utilização, a maioria das áreas do conhecimento depende dos sistemas integrados. Como exemplo, pode-se citar a medicina, os transportes e equipamentos de uso geral.

A grande disseminação dos sistemas integrados está condicionada à miniaturização e redução de custos de fabricação dos dispositivos semicondutores. Esse comportamento foi enunciado pela Lei de Moore (MOORE, 1965) e serviu como parâmetro de desenvolvimento para a indústria de semicondutores. Com a redução nas dimensões dos transistores, a complexidade dos sistemas aumentou e atualmente a maioria dos circuitos integrados (CIs) é desenvolvida utilizando um fluxo automatizado de projeto. Esse fluxo faz uso de ferramentas computacionais para tratar os desafios relacionados com a síntese e validação dos sistemas integrados modernos, ao mesmo tempo em que acelera o tempo de projeto dos mesmos.

A maioria dos circuitos digitais integrados atuais são desenvolvidos como circuitos síncronos. Dessa forma, a computação é realizada através da lógica combinacional e o estado do sistema fica armazenado na lógica sequencial. Na lógica combinacional a saída depende somente da combinação das entradas, enquanto a lógica sequencial é caracterizada por algum tipo de realimentação que faz com que a saída dependa das entradas e da saída anterior, conforme ilustrado na Figura 1. Cada tipo de lógica, combinacional e sequencial, possui diferentes características que devem ser observadas e, consequentemente, diferentes formas de serem avaliadas.

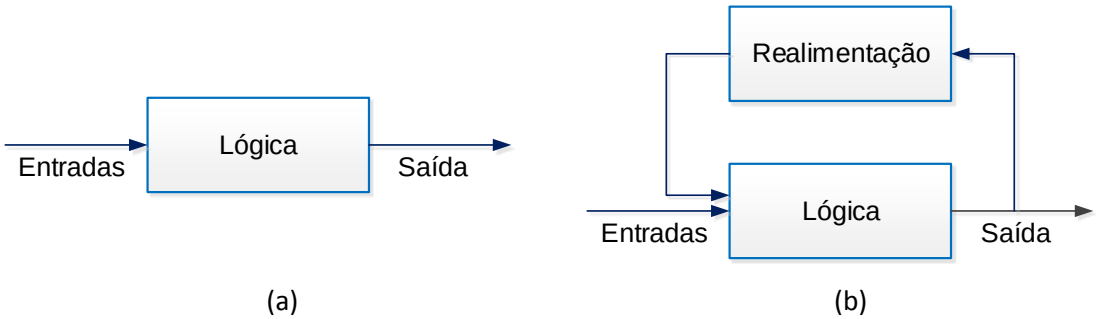


Figura 1. (a) Lógica combinacional; (b) Lógica sequencial

A redução do tamanho dos transistores trouxe muitas vantagens, como redução do custo por transistor, aumento do desempenho e redução de capacitâncias parasitas (BUTZEN, 2012). Porém, com dispositivos cada vez menores e mais complexos, a susceptibilidade a falhas aumenta continuamente (BALEN e LUBASZEWSKI, 2014).

Este fato aumenta a importância da validação e do teste no processo de concepção de CIs. Com o aumento da complexidade dos CIs, surge a necessidade de testes cada vez mais robustos e complexos, e consequentemente mais caros, que devem realizados em todas as etapas do projeto, a fim de detectar, de forma prematura, potenciais falhas nos circuitos (BALEN e LUBASZEWSKI, 2014). Além disso, com as rápidas mudanças de tecnologia, é necessária uma adaptação rápida dos sistemas de teste, de forma a reduzir o tempo para o mercado (do inglês, *time-to-market*). Duas estratégias que vêm sendo empregadas atualmente na solução desses problemas: o autoteste integrado (do inglês, *built-in self test* - BIST) e a geração automática de padrões de teste.

Esse projeto irá se concentrar na geração de circuitos para validação e *debug* em silício de portas lógicas sequenciais. A sequência deste primeiro capítulo localiza o trabalho no escopo dos sistemas digitais, discute a motivação para a execução do mesmo juntamente com o estado de arte e apresenta a proposta do trabalho detalhada. Por fim, a organização dos capítulos subsequentes é apresentada.

1.1. *Background*

A maioria dos CIs atuais é fabricada utilizando a tecnologia CMOS. Apesar de usarem a mesma tecnologia de fabricação, equipamentos eletrônicos podem ser desenvolvidos utilizando diferentes componentes integrados (WESTE e HARRIS, 2011). A escolha da tecnologia utilizada é crucial para o sucesso na indústria de eletrônicos. Essa escolha envolve uma avaliação do custo, do desempenho e do tempo até o produto ser lançado no mercado (BUTZEN, 2012). Os principais produtos associados à tecnologia CMOS são microprocessadores, lógicas programáveis (do inglês, *Field Programmable Gate Array* -FPGAs) e circuitos integrados de aplicação específica (do inglês, *Application Specific Integrated Ciruits* - ASICs).

A Figura 2 mostra a comparação entre essas três abordagens em termos de desempenho e facilidade para implementação. Microprocessadores se mostram bastante flexíveis, já que podem ser programados e adaptados para diferentes aplicações por meio de atualizações de um *software* que utilize seu conjunto de instruções. Seu custo reduzido permite uma fácil aquisição e uso em diversos campos. FPGAs são dispositivos que podem ter sua lógica e interconexões programáveis. Como sua programação está em um nível de abstração menor que a dos processadores, esses dispositivos podem fornecer melhores desempenhos em termos de velocidade e dissipação de potência. Porém, são mais difíceis de ser aplicados ao campo e possuem um custo maior. Por fim, existem os ASICs, CIs desenvolvidos para uma aplicação específica. Seu projeto é muito mais custoso que os demais, porém oferecem o melhor desempenho para determinada aplicação. ASICs são usados apenas quando o volume fabricado é grande, já que dessa forma dilui os custos do projeto.

Existem duas formas principais para o desenvolvimento de ASICs: totalmente customizados e baseados em bibliotecas de células padrão. Sistemas totalmente customizados (do inglês, *full custom*) são projetados no nível de disposição das camadas que irão formar os componentes do sistema. Esse tipo de projeto depende das regras de projeto associadas à tecnologia utilizada. Em outras palavras, o projetista se torna responsável pelo desenho de cada uma das máscaras envolvidas no processo, que deverão usadas na fabricação do circuito.

Como os projetos atuais possuem uma complexidade muito grande, com milhões de transistores, é praticamente impossível realizar projetos totalmente customizados. Por esse motivo, foi desenvolvido o fluxo de projeto baseado em biblioteca de células padrão (do inglês, *standard cell flow*). Nesse fluxo, portas lógicas previamente projetadas e caracterizadas são utilizadas como unidades básicas por ferramentas de síntese para a geração do circuito final. A biblioteca de células padrão é basicamente composta por um conjunto de portas lógicas combinacionais e sequenciais. Estas portas lógicas que compõem a biblioteca de células são normalmente projetadas de forma customizada, para que tenham o melhor comprometimento em termos de área utilizada, consumo de potência e velocidade de operação. O fluxo utiliza como entrada a descrição do circuito juntamente com a biblioteca de células padrão e as restrições do projeto. Os algoritmos de síntese escolhem o melhor conjunto de portas lógicas que pode ser utilizado para implementar o circuito e realizam o posicionamento e conexão dos sinais, gerando automaticamente o circuito a ser fabricado. (WESTE e HARRIS, 2011) (RABAEY, CHANDRAKASAN e NICOLIC, 2003).

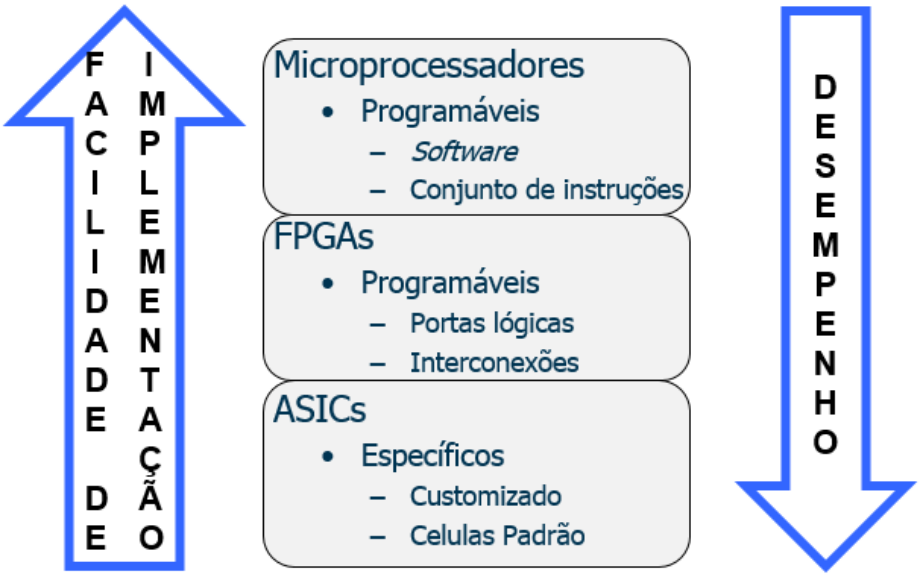


Figura 2. Comparação entre tecnologias para desenvolvimento de CIs

Por se tratar de um processo complexo, cujas etapas são quase sempre dependentes do resultado da etapa anterior, é necessário realizar testes em diversos momentos no projeto de CIs. Além disso, a dificuldade existente no manuseio desses elementos nanométricos torna o teste um dos procedimentos mais importantes e complexos do projeto de um CI, exigindo grande esforço

tecnológico e altos custos. Entretanto, a detecção tardia de falhas pode gerar custos ainda maiores. Estima-se que a reparação de uma falha se torna dez vezes mais custosa a cada nível de produção em que é detectada, conforme ilustrado na Figura 3. Portanto, sempre se busca essa detecção nas etapas iniciais do fluxo de projeto de CIs (WILLIANS e PARKER, 1982).

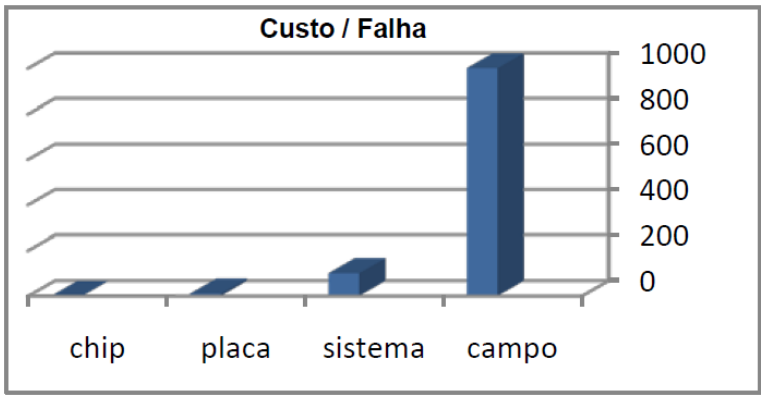


Figura 3. Estimativa do crescimento do custo por falha em sistemas eletrônicos *(BALEN e LUBASZEWSKI, 2014)*.

Esse trabalho irá se concentrar nos circuitos que são projetados utilizando o fluxo de projeto baseado em bibliotecas células padrão, pois é o mais difundido e utilizado atualmente (BAVARESCO, 2008). Mais precisamente, o trabalho focará na verificação funcional das portas lógicas sequenciais de uma biblioteca de células. A garantia de que não há falhas na biblioteca de células utilizada no *design* de um circuito é muito importante, pois se alguma porta lógica da biblioteca de células possuir alguma falha e for utilizada no circuito a ser projetado, este automaticamente estará susceptível a esta falha. Sendo assim, além do projeto da biblioteca de células propriamente dito, sua validação é de extrema importância para as etapas subsequentes do projeto de um CI.

Cada porta lógica de uma biblioteca de células deve ser verificada individualmente, de forma que se garanta seu correto funcionamento (RIBAS, SUN, 2011). Considerando que uma biblioteca de células pode ter mais de uma centena portas diferentes, a implementação manual desses processos de validação pode tornar os custos e esforços de engenharia necessários para a realização de testes altos demais. Além disso, as estratégias de testes podem não explorar todas as células da biblioteca ou não cobrir todas as possíveis condições de funcionamento das mesmas.

1.2. Motivação

O projeto de uma nova biblioteca de células é um processo meticuloso. Geralmente, esse processo é desenvolvido em *foundries* (indústrias destinadas à fabricação de circuitos integrados). Engenheiros especializados no desenvolvimento de bibliotecas de células padrão fazem o projeto e a escolha do conjunto de portas lógicas que farão parte da biblioteca, criando *kits* de desenvolvimento (*Process Development Kits*, PDKs) que são utilizados pelos projetistas de CIs. Além disso, uma mesma tecnologia de fabricação pode ter diferentes versões de bibliotecas de células. Estas diferentes versões são projetadas para dar aos projetistas uma maior flexibilidade em função das necessidades particulares, em termos de atrasos, tensão de alimentação, consumo de potência, etc, de cada projeto.

Dada a necessidade de que as portas lógicas da biblioteca sejam entregues com garantias de funcionamento ao projetista, cada célula da biblioteca deve ser validada logicamente e caracterizada eletricamente. Isso é feito em diversas etapas. Assim que é finalizado o leiaute da célula desenvolvida, técnicas como DRC e LVS verificam o correto funcionamento dessa célula. As portas lógicas desenvolvidas também devem ser testadas no fluxo de células padrão, demonstrando que são compatíveis com sua aplicação. Para isso, são aplicadas a circuitos conhecidos, denominados *benchmarks*. Apesar de serem aplicadas ao fluxo, apenas simulações desses circuitos são realizadas, não sendo levados à fabricação. Porém, como todas essas verificações são baseadas em modelos, o uso de simulações é insuficiente para assegurar o correto funcionamento das portas lógicas. Estas devem também ser fabricadas e testadas em silício (DATTA, SEBASTINE, *et al.*, 2004).

Em geral, os testes em silício para validação e caracterização das células se baseiam na fabricação de alguns dos circuitos *benchmarks* usados nas simulações ou de estruturas específicas, como osciladores em anel, redes de atrasos, contadores, entre outras (BAVARESCO, 2008). O projeto desses circuitos geralmente é feito de forma totalmente customizada, com processos cuidadosos para verificar de forma robusta cada possibilidade de funcionamento da porta lógica.

Benchmarks, como *ISCAS'89, ITC'99* e *IWLS'05*, são circuitos referência que envolvem uma série de operações com aplicações e arquiteturas que emulam o funcionamento de circuitos e blocos de sistemas comerciais. Possuem circuitos puramente combinacionais, máquinas de estados (compostas por lógica combinacional e sequencial), blocos aritméticos, entre outros. Essa característica permite que sejam usados para avaliar determinada biblioteca em comparação a outras e identificar falhas em seu funcionamento (LIN, SHOU e TSAI, 1999). O principal problema desse tipo de abordagem é que não se garante o teste de todas as portas lógicas da biblioteca, já que não necessariamente serão todas usadas nos circuitos. E mesmo que todas as células sejam utilizadas, não se garante que todas as suas combinações de entradas e transições sejam testadas (BAVARESCO, 2008). Isso é especialmente preocupante quando se considera circuitos sequencias, que possuem diversas características específicas de estados estáveis e transições esperadas e não esperadas.

O teste baseado em estruturas totalmente customizadas pode garantir maior eficiência à validação, mas aumenta o tempo de projeto da biblioteca. Além disso, existem tecnologias que geram automaticamente bibliotecas de células, o que inviabilizaria esse tipo de abordagem.

Observa-se que os métodos usados atualmente para validação de bibliotecas de células ainda podem ser aprimorados. Um método ideal para a validação de uma nova biblioteca deve no mínimo garantir que todas as células da biblioteca sejam testadas e que haja uma cobertura completa de funcionalidade das células sob teste. (BAVARESCO, 2008) apresenta uma técnica com essas características para validação em silício de portas lógicas combinacionais. É importante a automatização de um gerador de estruturas de validação de portas lógicas sequenciais, reduzindo o tempo associado ao desenvolvimento de estruturas de validação.

1.3. Trabalhos Relacionados

Diversos livros e trabalhos na literatura abordam o conceito de teste, validação e autoteste em circuitos combinacionais e sequenciais, focando em

diferentes pontos. Nessa seção serão apresentados os principais trabalhos relacionados. Alguns dos termos apresentados nessa seção serão explicados no Capítulo 2. (RABAEY, CHANDRAKASAN e NICOLIC, 2003) dedica toda uma seção a projetos voltados para testabilidade e autoteste. (BREUER, ABRAMOVICI e FRIEDMAN, 1990) e (JHA e KUNDU, 1990) são livros completos sobre testes de CIs, com capítulos descrevendo diferentes abordagens de autoteste. Ainda assim, técnicas para teste de CIs continuam em constante desenvolvimento, pois os custos associados a falhas de projeto podem se tornar muito altos.

Em (ALIOTO, CONSOLI e PALUMBO, 2011) e (ALIOTO, CONSOLI e PALUMBO, 2011a) são discutidos diversos fatores para o desenvolvimento de *flip-flops*, considerando a razão entre atrasos, consumo e área para esses dispositivos. O trabalho mostra a grande variedade de topologias que esses dispositivos podem ter, para diferentes aplicações, o que torna ainda mais evidente a necessidade de técnicas para testes desses diferentes dispositivos.

Em termos de lógica sequencial, a maior parte dos trabalhos se concentra na validação temporal de *latches* e *flip-flops* e poucos mostram métodos para validação lógica desses circuitos. (DATTA, SEBASTINE, *et al.*, 2004) propõe uma técnica para realizar medidas precisas de atrasos em CIs complexos. (NEDOVIC, WALKER e OKLOBDZIJA, 2004) e (JAIN, VEGGETTI, *et al.*, 2011) apresentam métodos para verificar características elétricas de circuitos sequenciais, se concentrando principalmente em análises temporais.

(CHAMPAC, ZENTENO e GARCÍA, 2005) se concentra no estudo de falhas resistivas em elementos sequenciais. O trabalho mostra a importância da prévia inicialização do dispositivo sequencial antes de sua análise. Falhas resistivas geralmente surgem da variabilidade na fabricação de CIs e causam atrasos que podem afetar o desempenho do circuito como um todo. Outros trabalhos se concentram na avaliação da variabilidade dos processos de fabricação. (BHUSHAN, KETCHEN, *et al.*, 2006), por exemplo, se utiliza de osciladores em anel, para verificar a variabilidade dos parâmetros dos circuitos sequenciais.

Os trabalhos que mais se assemelham a esse consideram a validação lógica e o *debug* funcional de CIs, visto que esse é um dos primeiros passos para se garantir que portas lógicas de uma biblioteca de células tenham sido

desenvolvidas corretamente. Além disso, a validação lógica e o *debug* funcional das portas lógicas podem não ser cobertos pelas abordagens que avaliam as características temporais.

Técnicas de geração automática de sequências de teste são muito utilizadas nessas aplicações. (NIERMANN, ROY, *et al.*, 1990) desenvolve algoritmos para otimizar testes gerados automaticamente para circuitos sequenciais. O problema dessa abordagem é que nem sempre garante a validação exaustiva do circuito, que é muito importante no caso do teste de uma biblioteca de células.

(BAVARESCO, 2008) desenvolve um método para teste de células combinacionais em bibliotecas de células. Seu trabalho propõe um método de geração automática de um circuito para validação de bibliotecas de células. Na arquitetura proposta em seu trabalho, o circuito de testes é composto por blocos que garantem validação lógica de todas as portas lógicas combinacionais de uma biblioteca e também é possível verificar suas características elétricas. A autora não inclui análises de células sequenciais. Além disso, é sabido que, devido à realimentação existente nas portas lógicas sequenciais, testes desenvolvidos para portas lógicas combinacionais não geram alta cobertura de falhas nos dispositivos sequenciais (CHAMPAC, ZENTENO e GARCÍA, 2005).

Uma alternativa para validação de células sequenciais por meio de autoteste é apresentada por (RIBAS, SUN, *et al.*, 2011). O autor utiliza a teoria de osciladores em anel para montar uma estrutura com *latches* ou *flip-flops* idênticos ligados em anel de forma que todas as excitações necessárias para a validação lógica e caracterização elétrica sejam realizadas de forma cíclica. Algumas portas lógicas combinacionais foram adicionadas ao projeto de forma a garantir o fluxo de dados e alcançar todas as transições esperadas. Os circuitos para a validação foram desenvolvidos manualmente, de forma a buscar a oscilação entre os estados estáveis do dispositivo. Além de ser um trabalho custoso em termos de tempo, nem sempre é possível garantir a validação de todas as transições e estados.

Outra abordagem é usada por (MAKAR e MCCLUSKEY, 1995), e visa a validação funcional de *latches*. O conceito de tabelas de fluxo é fundamental para o trabalho, pois é uma abstração da tabela verdade especialmente útil para a análise de circuitos sequenciais sensíveis a nível. Cada célula da tabela de

fluxo representa um estado estável do dispositivo e transições podem ser feitas para estados adjacentes. Dessa forma, ao percorrer a tabela de fluxo, todos os estados estáveis do *latch* podem ser verificados, mas não há verificação de todas as suas transições. Além disso, o método não é aplicável a *flip-flops*. Em um trabalho posterior (MAKAR e MCCLUSKEY, 1997), os autores apresentam um método para teste de *flip-flops* por meio do teste elétrico de corrente quiescente (Iddq). O trabalho parte de técnicas para geração automática de padrões de teste, que também não garante a validação exaustiva dos estados e transições do dispositivo sob teste.

Por fim, (VERMEULEN e GOEL, 2002) apresenta uma técnica de projeto para facilitar o *debug* em silício de CIs. Nesse trabalho, o autor também discute características necessárias para um bom sistema de *debug*. Essas características serão mostradas em maiores detalhes no referencial teórico desse trabalho.

1.4. Proposta do Trabalho

Esse trabalho tem como objetivo o desenvolvimento de um sistema de *debug* em silício para validação lógica das portas lógicas sequenciais de uma biblioteca de células. Esse sistema deverá gerar padrões de teste para garantir a completa validação lógica em silício de qualquer porta lógica sequencial cujo comportamento seja conhecido. Esses padrões serão automaticamente implementados em HDL, de forma que seja possível realizar a validação lógica em silício.

Para isso, será proposto um método para geração automática de circuitos de autoteste para validação dessas portas lógicas. O primeiro passo do trabalho é a aquisição dos estados estáveis e das transições esperadas e não esperadas do dispositivo. O segundo passo será um método para geração de uma sequência de vetores de teste que cubra todos os estados estáveis e transições da porta lógica. Por fim, se buscará uma implementação em linguagem de descrição de *hardware* (HDL) do processo desenvolvido, para a implementação em silício.

Apesar do projeto não focar diretamente na caracterização elétrica, espera-se que seja possível observar falhas temporais, tornando possível a verificação das características elétricas dos elementos sequencias avaliados.

1.5. Organização do trabalho

No próximo capítulo serão discutidos os principais conceitos relacionados à pesquisa realizada e o estado da arte da validação de circuitos sequenciais. No Capítulo 3, o procedimento proposto por esse trabalho é apresentado, acompanhado de um estudo de caso ilustrando o processo. No Capítulo 4 são expostos os resultados obtidos para diferentes dispositivos aplicados ao procedimento proposto. Por fim, conclusões e trabalhos futuros são apresentados no Capítulo 5.

2. REFERENCIAL TEÓRICO

Nessa seção serão descritos os principais conhecimentos necessários para uma melhor compreensão do trabalho proposto. Neste sentido, o primeiro aspecto a ser abordado é o projeto de circuitos integrados. Como este trabalho tem a lógica sequencial como principal objeto de estudo, seus principais representantes, *latches* e *flip-flops*, serão apresentados em maiores detalhes. Da posse dos elementos chave do trabalho, serão discutidos alguns métodos de testabilidade de circuitos. Por fim, serão brevemente apresentadas linguagens de descrição de *hardware*, nas quais o circuito gerado será descrito.

2.1. Projeto de ASICs

Muitos fatores são levados em consideração na escolha do estilo de projeto de ASICs. Entre eles, destacam-se a área do *chip*, consumo de potência, desempenho desejado e volume de produção (RABAEY, CHANDRAKASAN e NICOLIC, 2003). A complexidade dos circuitos atuais tornou a metodologia baseada em biblioteca de células padrão a mais utilizada. Porém, o método totalmente customizado ainda é muito utilizado no projeto dessas células padrão (WESTE e HARRIS, 2011).

2.1.1. Fluxo Totalmente Customizado (*Full Custom*)

O método mais antigo e tradicional para o projeto de CIs é o desenho customizado de máscaras, em que o projetista desenha manualmente cada uma das máscaras do projeto de forma geométrica, levando em consideração as diversas regras de projeto que precisam ser aplicadas durante esse processo.

O uso desse método geralmente leva a circuitos menores e mais eficientes, já que cada um de seus componentes é desenvolvido

especificamente para aquela aplicação. Em contrapartida, quando comparado a métodos automatizados, o tempo de projeto se torna muito maior. A maior parte do trabalho não pode ser reutilizada em outros projetos e uma habilidade muito maior é exigida do projetista (WESTE e HARRIS, 2011). Por consequência, o custo do projeto também é elevado.

Com o aumento da complexidade dos projetos, esse método foi se tornando cada vez mais pontual, sendo usado somente em frações de projetos, que poderiam posteriormente ser ligadas às demais. Atualmente, os circuitos integrados são tão complexos que o uso de técnicas totalmente customizadas se restringem a aplicações muito específicas, quando se deseja desempenho máximo de alguma região de um circuito.

2.1.2. Fluxo de Células Padrão (*Standard Cell Flow*)

Devido à grande complexidade dos CIs atuais, na maioria dos casos é praticamente impossível o desenvolvimento de circuitos totalmente customizados. Por esse motivo, o fluxo baseado em biblioteca de células padrão se tornou o mais difundido atualmente para o desenvolvimento de ASICs.

O fluxo de projeto baseado em biblioteca de células padrão, ilustrado na Figura 4, parte de uma descrição comportamental do circuito em linguagem de descrição de *hardware* (HDL). Por meio de uma síntese de alto nível, o código HDL comportamental é convertido em uma descrição HDL a nível de registradores (RTL). Essa síntese é independente da tecnologia, ocorrendo apenas em nível lógico.

A síntese lógica, segunda transição na Figura 4, produz uma descrição HDL estrutural do circuito a partir da descrição RTL do circuito e das informações das portas lógicas que estão disponíveis na biblioteca de células padrão. Esta descrição HDL estrutural contém todas as portas lógicas utilizadas para implementar o circuito bem como suas conexões. Neste ponto, já é possível avaliar o comportamento do circuito com as informações das portas lógicas da biblioteca de células. Os algoritmos de síntese lógica também recebem como entrada as restrições que o circuito deve atender. Baseado nestas restrições,

que normalmente são restrições temporais, é definido o conjunto de portas lógicas que irá implementar o circuito.

A síntese física, terceira e última transição da Figura 4, gera o leiaute do circuito a partir da descrição estrutural fornecida pela síntese lógica. Ela é responsável pela organização do circuito, posicionando as portas lógicas e roteando seus sinais (KAHNG, LIENIG, *et al.*, 2011).

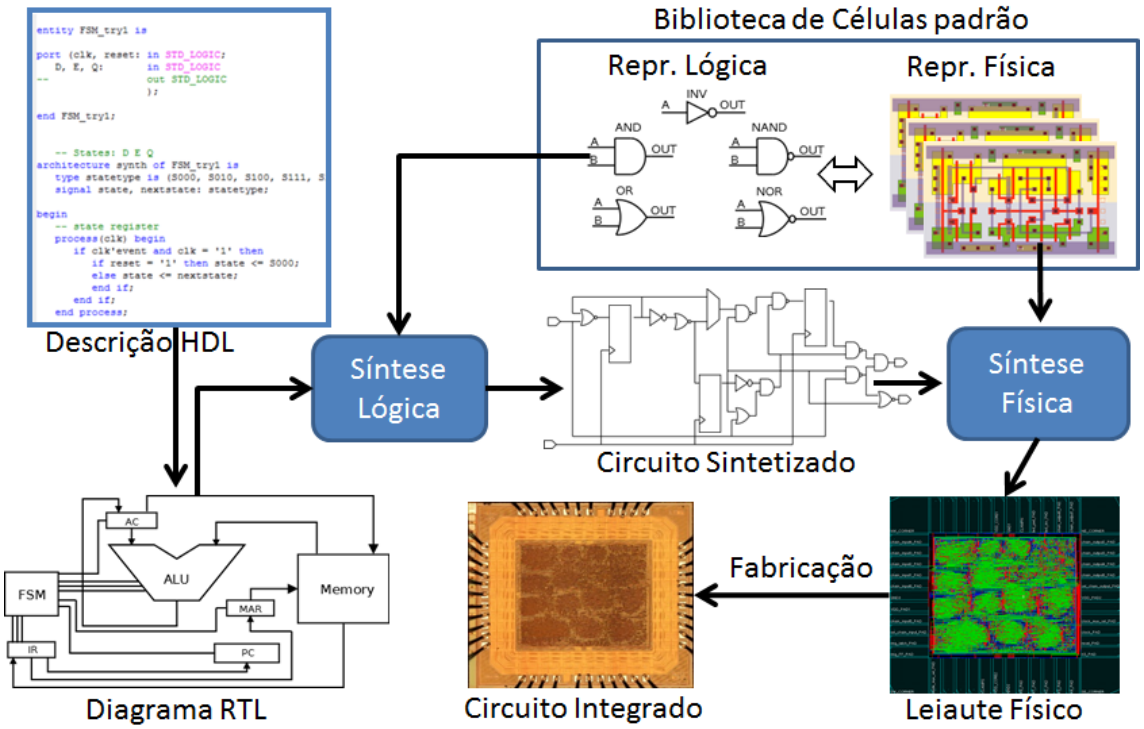


Figura 4. Fluxo de Projeto Automatizado

A metodologia baseada em biblioteca de células padrão se baseia no uso de uma biblioteca de células, com portas lógicas combinacionais e sequenciais previamente projetadas e caracterizadas, que possa ser usada para o desenvolvimento automatizado do circuito desejado. O *chip* é dividido em bandas de altura fixa e as portas lógicas são distribuídas de forma a otimizar uma série de métricas como comprimento dos fios, congestionamento, desempenho, entre outros (KAHNG, LIENIG, *et al.*, 2011).

Devido ao uso de células genéricas previamente projetadas, o fluxo de células padrão produz circuitos com desempenho inferior aos desenvolvidos de forma totalmente customizada. Contudo, ganho em produtividade é significativo, dado que os algoritmos de síntese reutilizam de forma automática as portas lógicas previamente projetadas em diversas partes do projeto. Ferramentas de

EDA (do inglês, *Eletronic Design Automation*) são utilizadas para escolha e distribuição automática de portas lógicas no *chip*. Essa distribuição pode ser feita de forma a otimizar parâmetros como atraso, área ou consumo de potência. (WESTE e HARRIS, 2011).

A partir do discutido anteriormente, fica claro que a qualidade de um projeto baseado em bibliotecas de células padrão depende de três fatores: da ferramenta de síntese, das ferramentas de posicionamento e roteamento e da escolha da biblioteca correta (SCOTT e KEUTZER, 1994).

2.1.3. Biblioteca de Células

Bibliotecas de células costumam possuir uma grande variedade de portas lógicas simples, além de alguns agrupamentos complexos que possam ser comumente usados. Algumas estruturas comuns em bibliotecas são inversores, portas *OR, NOR, AND, NAND, XOR, XNOR, latches, flip-flops, buffers*, portas lógicas compostas, como *AOI* e *OAI*, memórias, multiplexadores, somadores, comparadores, entre outros (BAVARESCO, 2008). Além disso, a biblioteca pode agrupar implementações variadas de uma mesma função lógica. A fim de simplificar o processo de posicionamento das células no *chip*, o leiaute de todas as células de uma biblioteca deve possuir a mesma altura, mesmo que a largura possa variar.

A escolha dos componentes da biblioteca utilizada é determinante para o CI em desenvolvimento. A quantidade e a qualidade das portas lógicas do projeto podem determinar o custo e a área do *chip* e também seu consumo de potência, atrasos e frequência de operação. Notavelmente, a qualidade de uma biblioteca não é um conceito absoluto, já que para cada aplicação pode-se considerar diferentes objetivos de desempenho. Muitos desenvolvedores se concentram somente no desenvolvimento desses agrupamentos de portas lógicas, que pode ser feito de forma automática ou manual (BAVARESCO, 2008). A realização de testes e validações é imprescindível para garantir a correta funcionalidade dessas portas lógicas.

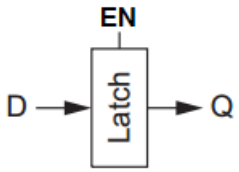
2.2. *Latches e Flip-Flops*

Enquanto em células combinacionais o nível lógico de saída depende somente da combinação de suas entradas naquele exato momento, em células sequenciais as saídas dependem das entradas e também do estado anterior do dispositivo. Fica evidente que uma das principais características das portas lógicas sequenciais é a capacidade de memorizar dados (TOCCI, WIDMER e MOSS, 2010). Esse trabalho irá se concentrar nas células sequenciais, representadas principalmente por *latches* e *flip-flops*. O propósito desses elementos, também conhecidos como registradores ou multivibradores biestáveis, não é só o armazenamento de dados, mas também o controle de sequências, o que nos permite chamá-los de “elementos sequenciais”.

Circuitos compostos por *latches* ou *flip-flops* podem ser tanto estáticos, quanto dinâmicos. Elementos estáticos mantêm seu estado de saída por tempo indeterminado, enquanto estiverem energizados. Por esse motivo, são usados quando o registrador não é atualizado por grandes períodos de tempo. Elementos dinâmicos mantêm o dado guardado por pequenos períodos, da ordem de milissegundos. Baseados na carga de capacitâncias associadas a transistores MOS, eles mantêm sua carga somente enquanto esses capacitores estão carregados. Esses circuitos costumam ser mais simples, o que resulta em maior performance e menor consumo. Porém, para que o estado se mantenha por um período maior, é necessário que esses capacitores sejam recarregados periodicamente, o que pode demandar estruturas de recarga que acabam aumentando a complexidade do projeto (RABAEY, CHANDRAKASAN e NICOLIC, 2003).

Como existem divergências entre as nomenclaturas existentes na literatura, serão apresentadas as definições utilizadas nesse texto. *Latches* são elementos sequenciais sensíveis a nível, ou seja, suas saídas dependem do estado anterior e da combinação das entradas. *Flip-flops* são sensíveis a borda, isto é, suas saídas dependem do estado anterior e da combinação das entradas no momento de uma borda de subida ou descida do sinal de controle (VAHID, 2008). Como esses elementos podem ser implementados de diversas formas, em geral utiliza-se um bloco para representa-los. Também existem diferentes tipos de *latches* e *flip-flops*, sendo o tipo *D* o mais utilizado.

O *latch* tipo *D* possui duas entradas, *ENABLE (EN)* e *D*. O funcionamento de um *latch* tipo *D* sensível a nível lógico alto é ilustrado pela tabela verdade da Figura 5. Quando a entrada *EN* está em nível lógico alto, a entrada *D* é copiada para a saída *Q*, e o *latch* é dito transparente. Quando a entrada *EN* está em nível lógico baixo, a saída *Q* não muda, independente de mudanças na entrada *D*. Nesse momento, se diz que o *latch* é opaco, pois a saída não é afetada pela entrada.



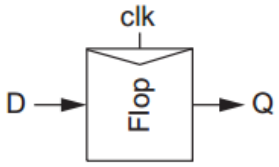
(a)

EN	D	SAÍDA (Q)
0	X	Não muda
1	0	0
1	1	1

(b)

Figura 5: (a) Representação *Latch* tipo D (WESTE e HARRIS, 2011); (b) Tabela verdade do *Latch* tipo D

O *flip-flop* tipo *D* se diferencia do *latch* tipo *D* pela adição de um detector de borda. Sua entrada só é propagada para a saída na borda do sinal de controle (*clk*). Na Figura 6 é mostrado o comportamento de um *flip-flop* tipo *D* sensível à borda de subida. Utilizando um *clock* síncrono em sua entrada, é possível saber todos os momentos em que pode ocorrer uma transição de saída, tornando o dispositivo totalmente síncrono (VAHID, 2008).



(a)

clk	*D*	SAÍDA (*Q*)
↑	0	0
↑	1	1

(b)

Figura 6: *Flip-Flop* tipo D: (a) Representação *(WESTE e HARRIS, 2011)*; (b) Tabela verdade.

O funcionamento de *latches* e *flip-flops* tipo *D* ainda pode ser alterado por meio da adição de entradas *set* ou *reset*, que podem ser síncronas (mudam na próxima borda do sinal de controle) ou assíncronas (mudam imediatamente). Essas entradas são prioritárias e causam transições de saída para “1” ou para “0”, respectivamente, quando acionadas (RIBAS, SUN, *et al.*, 2011).

2.3. Falha, Erro e Defeito

Falhas podem ser consideradas como operações diferentes das especificadas para um sistema, podendo ser originadas de erros de projeto, defeitos de fabricação ou interferência externa. Falhas podem ser permanentes ou transientes. Quando são permanentes, uma vez que o componente falha, ele nunca volta a funcionar corretamente. Já as transientes são aquelas que possuem duração limitada. Tais falhas também podem ser intermitentes, ocorrendo repetidamente por curtos intervalos de tempo (ZIMPECK, BUTZEN e MEINHARDT, 2014).

Um defeito é definido como um desvio da especificação do circuito que ocorre fisicamente, podendo ser decorrente do projeto, da fabricação ou da utilização. A abstração do defeito é chamada de falha, indicando que tipo de funcionalidades podem ser afetadas por aquele desvio do que era esperado. Dependendo das funcionalidades afetadas, uma falha poderá levar a um erro, isto é, uma alteração indesejada no estado do sistema. Define-se que um sistema estará no estado errôneo quando uma falha se manifesta externamente ao circuito, alterando suas saídas ou margens de especificação (BALEN e LUBASZEWSKI, 2014). A Figura 7 ilustra a relação entre essas três condições.

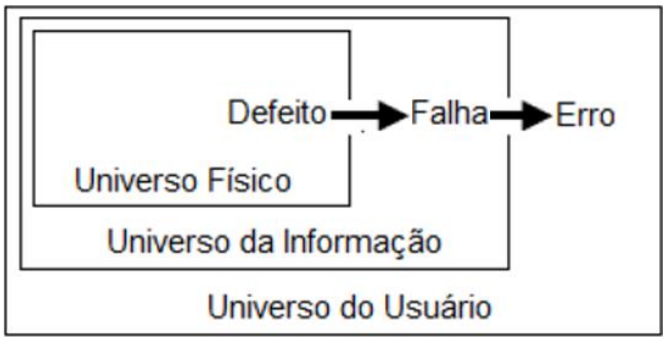


Figura 7. Comparação entre falhas, erros e defeitos.

Normalmente, a falha é a causa raiz do efeito anômalo que poderá ser observado. Portanto, a modelagem das falhas é o passo inicial de teste de circuitos integrados. Nesse trabalho serão discutidas apenas as falhas permanentes. Falhas transientes são descritas em maiores detalhes em (BALEN e LUBASZEWSKI, 2014). Existem dois principais tipos de falhas permanentes que ocorrem em transistores MOS, as falhas *stuck-on* e as falhas *stuck-open*. As

falhas *stuck-on* se caracterizam por fazer com que o transistor conduza permanentemente, independente da tensão aplicada ao seu terminal de controle. Ao contrário das falhas *stuck-on*, as falhas *stuck-open* se caracterizam por fazer com que o transistor opere sempre como uma chave aberta, independe da tensão aplicada ao terminal de controle.

Quando a análise é feita em nível de portas lógicas, um nível de abstração acima do nível de transistores, é mais conveniente analisar o modelo de falhas do tipo *stuck-at*. Estas falhas podem ser abstraídas a partir das falhas do tipo *stuck-on* ou *stuck-open* e também de falhas nas interconexões. Falhas do tipo *stuck-at* se caracterizam por determinados nós do circuito estarem sempre em nível lógico baixo, denominadas *stuck-at-0*, ou em nível lógico alto, denominadas *stuck-at-1*.

2.4. Teste de Circuitos Integrados

Testes e verificações de CIs são divididos em três grandes grupos: verificação de funcionalidade ou validação lógica, *debug* em silício e teste de fabricação. A validação lógica se preocupa com o funcionamento do dispositivo desenvolvido, ou seja, verifica se ele atua conforme esperado, e é feita antes mesmo do leiaute. O *debug* em silício é feito após a primeira leva de equipamentos fabricados, verificando se ele realmente funciona como esperado e buscando falhas que possam tê-lo levado a um mal funcionamento. Os testes de fabricação verificam se os transistores, portas, e demais elementos utilizados na fabricação estão funcionando corretamente (WESTE e HARRIS, 2011). Diversos equipamentos utilizam também o teste em utilização, que verifica falhas mesmo depois de o equipamento estar em utilização. Atualmente, muitos dispositivos se utilizam de técnicas de autoteste integrado. Essa abordagem será utilizada nesse trabalho e, portanto, será discutida em maiores detalhes nessa seção, junto à validação lógica e ao *debug* em silício, que estão diretamente relacionados ao trabalho.

2.4.1. Autoteste e Autochecagem (*Self-testing* e *Self-checking*)

Diferentes estruturas de teste podem ser usadas diretamente na construção do *chip* para verificar pontos que dificilmente podem ser verificados externamente, como o desempenho de transistores ou resistências de contato. (WESTE e HARRIS, 2011). De forma a reduzir os custos inerentes ao teste de CIs e também torná-los mais robustos, é desejável, e às vezes necessário, o uso de circuitos que, fabricados junto ao CI, testem e forneçam informações pertinentes a seu funcionamento. Denominados *built-in self-testing circuits* (BIST), em inglês, esses circuitos de autoteste integrado geralmente analisam o circuito por meio de vetores de testes, realizando uma varredura de erros em sua operação, sem a necessidade de um testador externo (RABAEY, CHANDRAKASAN e NICOLIC, 2003).

Esses circuitos não analisam o circuito em tempo real, mas podem ser usados para verificar seu funcionamento em diferentes estágios. Antes da fabricação, pode-se fazer uma verificação por meio de simulações. Após a fabricação, pode-se obter informações das partes a serem verificadas. No produto final, o circuito BIST pode fornecer informações sobre envelhecimento do dispositivo, podendo realizar testes na inicialização ou em momentos ociosos de seu funcionamento.

Alguns sistemas exigem confiabilidade muito grande quando comparados aos dispositivos de uso geral. Como exemplo, pode-se citar sistemas médicos, controle de aviões, controle de plantas nucleares e computadores para missões espaciais. Em qualquer um desses sistemas, uma falha pode ser fatal, seja de fabricação, de uso ou de envelhecimento. Portanto, é necessário aumentar a robustez dos circuitos envolvidos nessas operações e realizar testes em tempo real, verificando seu funcionamento e apontando erros assim que ocorrerem, de forma a evitar a contaminação dos dados (JHA e KUNDU, 1990). Circuitos com essa função são chamados de *self-checking circuits*.

Existem diversas formas de se implementar um circuito *self-checking*. Em suas versões mais simples, utilizam-se somente algoritmos, enquanto nas mais complexas se utiliza redundância. De forma geral, pode-se dizer que consistem de uma célula funcional e uma célula verificadora, denominada

checker, conforme a Figura 8. As entradas e saídas do circuito funcional são codificadas de forma que possam ser decodificadas pelo circuito verificador. Durante uma operação sem falhas, as entradas e saídas produzirão um código que poderá ser decodificado pelo verificador e o circuito poderá funcionar normalmente. Se um código que não possa ser decodificado pelo verificador for gerado, este irá indicar a existência de uma falha no circuito. Como o espaço amostral de códigos é limitado, geralmente não há preocupações com falhas no verificador, pois este provavelmente irá indicar uma falha da mesma forma. O uso dessa abordagem se reduz a nichos de mercado que exijam altíssima confiabilidade, uma vez que o processo pode envolver grandes aumento na área e na potência dissipada pelo projeto e reduções em seu desempenho.

Por definição, um circuito é *self-checking* para determinado grupo de falhas *F*, se para toda falha de *F* o circuito produzir um código na entrada que não pode ser decodificável com o código gerado na saída (JHA e KUNDU, 1990).

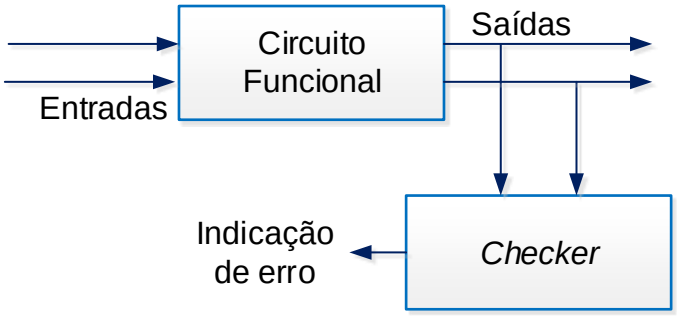


Figura 8. Estrutura de um circuito *self-checking (JHA e KUNDU, 1990)*.

2.4.2. Validação Lógica

Validação lógica de um dispositivo é um passo muito importante em seu desenvolvimento. Como o comportamento esperado de um dispositivo é conhecido, com base em tabelas verdade, a validação consiste em uma comparação entre o funcionamento esperado e o funcionamento real do dispositivo, o qual pode ser obtido por meio de uma simulação elétrica. Para um dispositivo combinacional, essa comparação pode ser feita observando somente seus estados estáveis, ou seja, verificando se a saída é correta para cada combinação possível de entradas (RIBAS, BAVARESCO, *et al.*, 2011).

Dispositivos sequenciais dependem não só de seu estado atual, como também de seu estado anterior. Nesses elementos, os estados estáveis podem considerar diferentes saídas para uma mesma combinação de entradas. Por esse motivo, para sua validação, também é necessário levar em consideração as transições do dispositivo (RIBAS, CALLEGARO, *et al.*, 2011). Essas transições podem ser divididas em esperadas, não esperadas e não possíveis.

As transições esperadas compreendem as alterações na saída que devem surgir de uma alteração em somente uma das entradas do dispositivo, considerando a configuração atual de entradas e saídas. A Figura 9(a) ilustra uma transição esperada. Neste caso a entrada D de um *Latch* é “0” e o valor armazenado no Latch é “1” (Q = “1”). Quando o sinal de controle E muda para o nível alto, a sua saída Q assume o valor “0” da entrada, como esperado.

Contrárias às esperadas, as transições não esperadas compreendem as alterações na saída que não devem surgir de uma alteração em uma das entradas do dispositivo, considerando a configuração atual de entradas e saídas. Na Figura 9(b) a entrada D é transmitida para a saída Q enquanto a habilitação E está em nível baixo, portanto, se trata de uma transição não esperada de um *Latch D.*

Por fim, as transições não possíveis são aquelas que não podem ocorrer no dispositivo, por gerar uma saída desconhecida ou por partirem de um estado estável que nunca pode ter sido alcançado (AVELAR, BUTZEN e ROSA, 2014). A Figura 9(c) se inicia com um estado não estável para o *Latch D*, portanto não é possível de ocorrer. Outro caso que pode gerar transições não possíveis ocorre quando o elemento possui entradas mutuamente exclusivas, como *set* e *reset.* Nesses casos, qualquer transição que coloque ambas em nível alto é considerada não possível, a não ser que uma das entradas seja considerada prioritária.

(a) (b) (c)

Figura 9. Formas de onda para um *Latch* tipo *D*: (a) Transição esperada; (b) Transição não esperada; (c) Transição não possível

2.4.3. *Debug* em silício

Sistemas de *debug* podem ser usados tanto para análise de falhas de projeto, quanto para análise de variações na fabricação. Conforme definido por (VERMEULEN e GOEL, 2002), são 3 as capacidades necessárias para um bom sistema de *debug*:

a) Acesso aos pinos funcionais do *chip;*
b) Acesso a sinais e memórias internos do chip;
c) Execução controlada do *chip*.

O acesso aos pinos funcionais é necessário para que se possa enviar estímulos relevantes para excitar seu funcionamento. As respostas a esses estímulos deverão ser comparadas com dados previamente conhecidos de seu comportamento. O acesso à memória, ou seja, ao estado de dispositivo, facilita a detecção do local da falha. A execução controlada do dispositivo permite isolar a falha para análise do motivo de sua ocorrência. Um sistema desse tipo permite detectar falhas de funcionamento e identificar possíveis erros de *design* em uma análise do circuito em silício.

2.5. Linguagens de Descrição de Hardware

Linguagens de descrição de *hardware* (HDL, *Hardware Description Langua*ge) são conjuntos de ferramentas textuais utilizadas para a descrição de *hardwares* complexos de forma eficiente. O poder das HDLs vem justamente da possibilidade de abstração simples de circuitos que teriam esquemáticos bastante complexos (WESTE e HARRIS, 2011).

Além disso, o uso dessas linguagens tem duas vantagens principais: simulação e síntese. Simulações permitem a comprovação do funcionamento do *hardware* implementado, sem a necessidade de fabricação ou descrição SPICE de seu esquemático. Existem ferramentas específicas para bancos de testes para códigos em HDL. A síntese lógica tem justamente a função de converter a descrição textual em um *netlist* que represente o esquemático do projeto, ou

seja, um conjunto de portas lógicas que possa ser implementado na prática. A ferramenta de síntese ainda pode escolher as portas lógicas utilizadas de forma a otimizar diferentes parâmetros do circuito, como área, atrasos ou consumo de potência.

As HDLs mais conhecidas e utilizadas são Verilog e VHDL. Uma vez que se conhece uma delas, aprender a outra se torna simples, pois o modo de aplicação de ambas é similar. Existem longas discussões sobre qual das duas é melhor, mas grande parte dos projetistas de CIs precisa conhecer ambas, devido ao uso de código legado ou de blocos com propriedade intelectual (WESTE e HARRIS, 2011).

3. PROCEDIMENTO PROPOSTO

Nesse capítulo serão apresentados em detalhes os procedimentos utilizados para alcançar o objetivo do trabalho. Um estudo de caso, utilizando um *latch* tipo *D*, será desenvolvido no decorrer da explicação do procedimento, de forma a exemplificar seu funcionamento. A escolha do *latch* tipo *D* se deu por ser um dispositivo simples que apresenta todas as características necessárias para o uso total das capacidades propostas.

Com o objetivo de automatizar a validação lógica de uma célula sequencial e a geração do circuito para o debug em silício, o procedimento proposto foi dividido em diferentes módulos, conforme mostrado na Figura 10. Primeiramente, é imprescindível para a obtenção dos vetores de testes de um elemento sequencial o conhecimento de sua descrição comportamental. A partir da descrição comportamental pode-se conhecer todos os demais dados necessários para o desenvolvimento de seus vetores de teste. Isso permite o desenvolvimento de um algoritmo que analisa o comportamento da porta lógica e gera informações que possam ser utilizadas para o teste. Basicamente estas informações consistem nos estados estáveis e transições possíveis, esperadas e não esperadas. Esses dados permitem o entendimento da porta sequencial como uma máquina de estados finitos (FSM). Conhecida a FSM que determina o funcionamento da célula sequencial, é necessário um procedimento para aquisição de uma sequência de vetores de teste que percorra todos os estados estáveis e transições do dispositivo. Esse processo irá levar em consideração o uso de algoritmos de análise de grafos dirigidos, passando por todos os seus nodos e transições. Definida a sequência de teste, poderá ser feita uma implementação HDL da solução obtida. Cada uma das etapas mencionadas será detalhada nos tópicos a seguir.

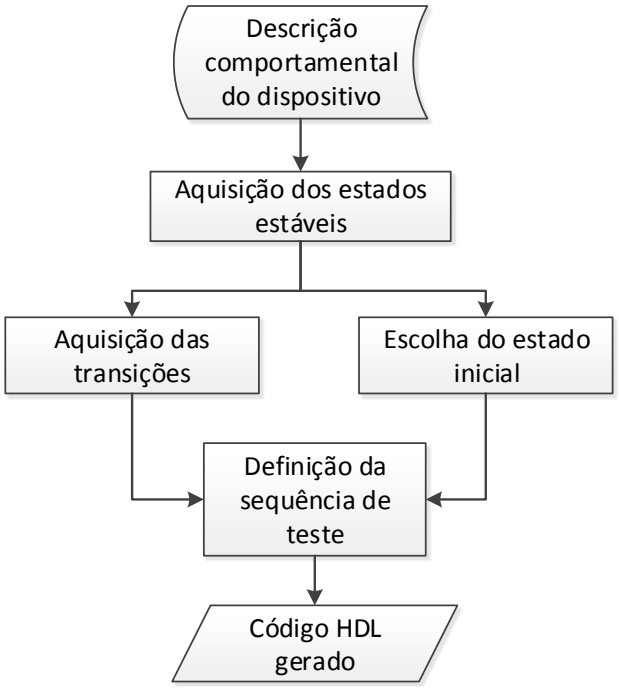


Figura 10. Procedimento Proposto

Descrição Comportamental do Dispositivo a ser Verificado

A entrada do processo para a aquisição dos estados estáveis de um circuito sequencial consiste em uma descrição do comportamento do dispositivo sob teste. Essa análise comportamental se refere às suas respostas a diferentes entradas e indica como o componente deverá operar se estiver funcionando corretamente. Esta descrição comportamental é o parâmetro existente para se gerar um testador que valide corretamente qualquer dispositivo com aquele comportamento, independente da tecnologia utilizada em sua fabricação ou da arquitetura interna do mesmo.

A descrição comportamental do *latch* tipo *D* (LatD) é mostrada no pseudocódigo da Figura 11. A entrada *D* é atribuída à saída *Q* quando *E* está habilitado e caso contrário, mantém a saída com o valor já armazenado Q_0. Por fim, a saída da descrição comportamental deverá sempre retornar a saída Q do dispositivo ao final da iteração.

```
LatD (E, D, Q0)
    Se E = 1:
        Q ← D
    Senão:
        Q ← Q0
    Retorna Q
```

Figura 11. Descrição comportamental do LatD

Aquisição dos Estados Estáveis

Conhecido o comportamento do dispositivo, pode-se obter todos os seus estados estáveis. Estado estável é qualquer combinação de entradas e saídas que o dispositivo é capaz de manter em regime permanente. A Figura 12 apresenta o procedimento para aquisição dos estados estáveis.

```
verificaEstadosEstaveis (n)
    Cria uma sequência de todas as combinações binárias de "n" bits
    Para cada combinação:
        Usar seus bits como parâmetros de entrada na descrição comportamental do dispositivo,
tanto para estado atual quanto para o estado anterior
        Se a saída for igual à entrada:
            O estado é estável
            O estado é guardado na lista de estados estáveis
        Senão:
            O estado não é estável
```

Figura 12. Pseudocódigo para obtenção dos estados estáveis

O procedimento descrito no pseudocódigo da Figura 12 depende do número de entradas e saídas do dispositivo, representado por "n". Dessa forma, se pode testar todas as combinações de entradas e saídas possíveis. Se, ao se aplicar uma combinação de entradas e saídas ao dispositivo, considerando o estado atual igual ao anterior, a saída não mudar, o estado do dispositivo pode ser considerado estável, já que não mudou espontaneamente. Caso contrário, o estado é descartado por não ser estável.

Para a exemplificação desse pseudocódigo, a verificação dos estados estáveis do *latch* tipo *D* será realizada. Como os *latch D* possui duas entradas e uma saída, todas as combinações binárias de 3 *bits* serão utilizadas como parâmetros (E, D, Q_0) na descrição comportamental do LatD (Figura 11) e sua saída Q será comparada com Q_0.

Por exemplo, para verificar se o estado E = 0, D = 1, Q_0 = 1 é estável, teria que ser executado *LatD (0, 1, 1)* e depois verificado se Q = Q_0. Nesse caso, Q será igual a Q_0, portanto, é um dos estados estáveis. Se fosse verificado o estado E = 1, D = 1 e Q_0 = 0, se observaria um Q_0 = 0 e Q = 1, o que indica que o estado não é estável. Realizando esse procedimento para todas as combinações possíveis de E, D e Q_0, se obteve lista de estados estáveis da Tabela 1.

Tabela 1. Estados estáveis do LatD

E	D	Q_0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	1	1

Aquisição das Transições

Procedimento similar ao da Figura 12 é repetido para obtenção das transições do dispositivo, representada no pseudocódigo da Figura 13. Neste procedimento, todos os estados estáveis do dispositivo já são conhecidos e comparados em pares para verificar se é possível realizar uma transição de um para o outro. Para fins de validação, o conceito de transição entre estados de um dispositivo sequencial parte do pressuposto que só uma de suas entradas mude de cada vez. No pseudocódigo da Figura 13 que são comparados apenas os n-1 primeiros *bits* do estado, pois último *bit* representa o valor da saída. Quando se compara dois estados estáveis, o primeiro passo é verificar se somente uma de suas entradas é diferente. Se houver mais de uma, a transição pode ser descartada.

Também é importante observar que, como circuitos sequenciais dependem não só de suas entradas, como também do seu estado anterior, a comparação de 2 estados deve sempre ser feita em duas vias, já que o fato de a transição existir em uma direção não garante que ela exista na outra.

As variáveis *estadoAnterior* e *estadoAtual* no pseudocódigo da Figura 13 representam os dois estados estáveis que estão sendo comparados. Se, ao se aplicar as entradas do *estadoAtual* em um dispositivo com as saídas do *estadoAnterior*, a nova saída for igual à do *estadoAtual*, então a transição é possível de ocorrer e deve ser guardada em uma lista de transições possíveis. Caso a saída seja diferente, a transição deve ser descartada, pois não é esperada do dispositivo.

As transições ainda podem ser divididas entre transições que geram uma transição de saída e transições que não mudam a saída. Diferentes tipos de falhas podem ser avaliados em diferentes tipos de transições. Se ressalta, por

fim, que o pseudocódigo desenvolvido considera dispositivos com apenas uma saída, podendo ser adaptado para dispositivos com mais saídas.

A lista de transições obtida a partir do pseudocódigo da Figura 13 pode ser representada por um grafo dirigido. Desta forma, a lista de transições será convertida em uma matriz de transições, utilizando-se o conceito de matriz de adjacências. Cada linha da matriz deverá representar um estado de origem e cada coluna deverá representar um estado destino. Utilizando-se um sistema de pesos, se define peso nulo quando a transição da linha para a coluna é possível e peso infinito quando a transição não é possível. O pseudocódigo da Figura 14 ilustra a geração dessa matriz de adjacências. O valor “m” representa o número de estados estáveis do dispositivo sob teste. Cada linha e cada coluna da matriz deverá representar um estado estável.

```
verificaTransição (estadoAnterior, estadoAtual)
    Usar as entradas do estadoAtual como estado atual e o estadoAnterior como estado anterior na
descrição comportamental do dispositivo
    Se a saída for igual à do estadoAtual:
            É uma transição possível
            Enviar para lista de transições
            Se a saída do estadoAtual for diferente da do estadoAnterior:
                    Enviar para a lista de transições de saída esperadas
            Senão:
                    Enviar para a lista de transições de saída não esperadas
            Senão:
                    Não é uma transição possível
```

Figura 13. Aquisição das transições possíveis

```
geraMatrizDeTransições (estadosEstaveis, transições)
    Gerar uma matriz quadrada de tamanho “m” com todos os valores infinitos
    Para cada transição esperada:
            Buscar a linha da matriz com o estado origem
            Buscar a coluna da matriz com o estado destino
            Substituir o valor infinito por zero na posição origem x destino
```

Figura 14. Geração da matriz de adjacências representando as transições.

Seguindo com o estudo de caso desenvolvido, dessa vez será exemplificada a aquisição das transições possíveis. O primeiro pré-requisito a ser observado é a avaliação dos *n-1* primeiros *bits* de cada estado, em que *n* é

o número total de *bits*. Isso se deve ao fato de o último *bit* não representar uma entrada propriamente dita, mas sim o estado atual do elemento sequencial.

Assim, a comparação entre os estados [0, 0, 1] e [0, 0, 0] resultará na não existência de transição possível, já que não há *bits* diferentes entre os *n-1* (3 – 1 = 2) primeiros *bits*. O mesmo ocorre entre [0, 0, 1] e [1, 1, 1], pois existem 2 *bits* diferentes. Quando forem comparados os estados [0, 0, 0] e [0, 1, 1], existirá exatamente 1 *bit* diferente entre os 2 primeiros. Neste caso, a transição entre os dois estados será avaliada conforme a descrição comportamental.

Para a chamada da descrição comportamental, o estado de origem irá contribuir com o parâmetro Q_0 (Q_0 = 0, neste exemplo) e o estado de destino com os parâmetros E e D (E = 0, D = 1, neste exemplo). Sendo assim, deve-se executar a seguinte chamada: *LatD (0, 1, 0).* O resultado oriundo da execução do processo comportamental será comparado com a saída Q do estado de destino (Q = 1, neste exemplo). Se observamos o comportamento da função LatD, verifica-se que a mesma irá retornar Q = 0, diferente do esperado. Isso caracteriza a impossibilidade de haver uma transição entre os dois estados. Analogamente, ao se realizar a análise para a comparação entre [0, 1, 0] e [1, 1, 1], se obtém a saída Q esperada, portanto, essa é uma das transições esperadas do dispositivo.

Após a verificação de todas as possíveis combinações de estados, 2 a 2, se obtém as 12 transições listadas na Tabela *2*. Essas transições são representadas por uma matriz de adjacências ilustrada na Tabela 3. Nas duas tabelas os valores são apresentados na seguinte sequência: Entradas *E* e *D* e saída *Q*. Essa matriz é mostrada com um sistema de pesos para transição. Peso infinito (∞) indica que a transição nunca irá ocorrer, peso 0 indica que a transição pode ocorrer.

Tabela 2. Lista de transições para o LatD.

[0, 0, 0] para [0, 1, 0]
[0, 0, 0] para [1, 0, 0]
[0, 0, 1] para [0, 1, 1]
[0, 0, 1] para [1, 0, 0]
[0, 1, 0] para [0, 0, 0]
[0, 1, 0] para [1, 1, 1]
[0, 1, 1] para [0, 0, 1]
[0, 1, 1] para [1, 1, 1]
[1, 0, 0] para [0, 0, 0]
[1, 0, 0] para [1, 1, 1]
[1, 1, 1] para [0, 1, 1]
[1, 1, 1] para [1, 0, 0]

Tabela 3. Matriz de adjacências do LatD (linhas para colunas)

EDQ	000	001	010	011	100	111
000	∞	∞	0	∞	0	∞
001	∞	∞	0	∞	0	∞
010	0	∞	∞	∞	∞	0
011	∞	0	∞	∞	∞	0
100	0	∞	∞	∞	∞	0
111	∞	∞	∞	0	0	∞

Escolha do Estado Inicial

Outro passo importante para o testador é definir um estado inicial que deverá ser utilizado para a geração da sequência de vetores de testes a serem aplicados ao dispositivo sequencial. Quando se liga um dispositivo sequencial, não se sabe quais valores poderão existir em sua saída, portanto é necessário um passo extra que o leve a um estado conhecido. Para se garantir um estado inicial estável, pode-se partir das entradas prioritárias do dispositivo. Entradas prioritárias podem ser exemplificadas como as entradas assíncronas de *set* ou *reset*. Essas entradas mudarão a saída independente do estado anterior, podendo levar a um estado estável. Da mesma forma, se houver uma entrada de habilitação, quando esta é colocada no nível de habilitação, a entrada de

dados será transmitida para saída. No caso de dispositivos dependentes de bordas de *clock*, como *flip-flops*, é necessária uma borda que o leve a um estado conhecido. Neste trabalho, o método de escolha do estado inicial é realizado de forma manual.

Seguindo o estudo de caso do *Latch* tipo *D*, a análise do seu comportamento permite observar que a saída será igual à entrada sempre que a entrada de habilitação (E) estiver ativa. Dessa forma, tanto o estado [1, 0, 0] quanto o estado [1, 1, 1] podem ser utilizados como estado inicial.

Definição da Sequência de Teste

Com todos os dados da modelagem do dispositivo obtidos, pode-se finalmente partir para o desenvolvimento do testador. Para validar completamente um dispositivo, é necessário que todas as suas transições sejam verificadas. A partir do momento que todas essas transições são conhecidas, deve-se buscar um caminho de transições que passe por todas. Isso é feito pelo pseudocódigo da Figura 15, que define um método guloso para percorrer todas as transições do dispositivo. O algoritmo guloso se caracteriza por sempre realizar a escolha que parecer melhor no momento, baseado em sua vizinhança imediata, sem levar em consideração o sistema como um todo. Para reduzir a repetição de caminhos, cada vez que se passa por uma transição, seu peso é incrementado em 1 ponto. Dessa forma, também se pode garantir que não há transições não verificadas quando todas as transições tiverem peso maior que zero. Cada uma das transições realizadas é guardada em forma de uma sequência de estados do testador, que será usada para a geração de uma descrição do *hardware* proposto em HDL.

```
sequênciaDeTesteGulosa (estadoInicial, matrizDeTransições)
    Definir estado inicial como estado atual
    Enquanto houver transição com peso igual a zero (0) na matriz:
            A partir do estado atual, realizar a transição com menor peso
            Incrementar em 1 (uma unidade) o peso da transição
            Substituir o estado atual pelo estado de destino da transição
            Guardar a transição na sequência de teste
```

Figura 15. Algoritmo guloso para aquisição de uma sequência de validação.

A definição da sequência de teste é função do estado inicial escolhido e da matriz de transições mostrada na Tabela 3. O primeiro passo dessa etapa é definir o estado inicial como estado atual. A partir disso, sempre se busca o estado com menor peso na linha do estado atual, na matriz de transições. Se houver mais de uma transição com mesmo peso, a primeira encontrada será a realizada. Sempre que uma transição é realizada, seu peso é incrementado em 1 ponto para evitar que o teste entre em *loop* infinito. Com esse procedimento, será obtida a sequência de teste mostrada na Tabela 4 para o LatD, com estado inicial [1, 0, 0]. Essa sequência também pode ser ilustrada como uma máquina de estados, conforme a Figura 16.

Tabela 4. Sequência de teste para o LatD

[1, 0, 0] → [0, 0, 0] → [0, 1, 0] → [0, 0, 0] → [1, 0, 0] → [1, 1, 1] → [0, 1, 1] → [0, 0, 1] → [0, 1, 1] → [1, 1, 1] → [1, 0, 0] → [0, 0, 0] → [0, 1, 0] → [1, 1, 1] → [0, 1, 1] → [0, 0, 1] → [1, 0, 0]

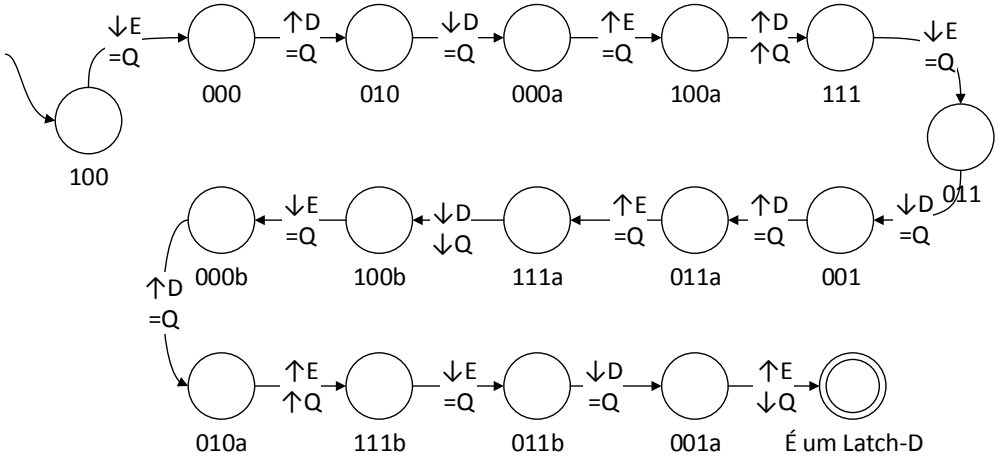


Figura 16. Sequência de teste para o LatD

Geração do Código HDL Sintetizável

A geração do código HDL sintetizável parte das características de uma FSM típica em HDL. No trabalho desenvolvido, a geração é feita tanto para a linguagem VHDL quanto para o Verilog. Com a estrutura básica montada, se pode adicionar as características e a sequência de testes referentes ao dispositivo sob teste, como a mostrada na Figura 16. Por fim, o testador em HDL deverá funcionar conforme as características mostradas no pseudocódigo da Figura 17.

O testador deve enviar dados às entradas do dispositivo sob teste e ler suas saídas, verificando se o dispositivo cumpre com suas características. Se uma de suas premissas não for cumprida, o dispositivo deverá ser descartado e seu projeto deverá ser revisado.

```
hdlTester (dispositivo)
    Enviar entradas para o estado inicial do teste
    Enquanto o estado não indicar teste completo ou erro:
            Ler os dados da saída do dispositivo
            Se os dados da saída são os esperados:
                    Mudar para o próximo estado
                    Enviar dados do próximo estado para a entrada do dispositivo
            Senão:
                    Mudar para estado de erro
    Teste completo
```

Figura 17. Algoritmo de funcionamento do testador em HDL.

Com a descrição HDL do circuito testador pronta, a mesma pode ser utilizada no fluxo de projeto de circuitos integrados baseado em biblioteca de células padrão para a geração do circuito testador propriamente dito.

Os códigos gerados para o *Latch* tipo *D* foram gerados e estão apresentados no Apêndice B, em VHDL e Verilog. Sua síntese lógica foi feita com sucesso e as informações adquiridas estão apresentadas, junto aos demais resultados, na Tabela 6 e na Tabela 8 do Capítulo 4.

Para ilustrar o correto funcionamento do procedimento, a simulação lógica da descrição HDL foi realizada usando como elemento sob teste um *Latch D* de uma biblioteca de células comercial e também a descrição deste mesmo *Latch D* alterada com o intuito de representar uma falha. A Figura 18 mostra as formas de onda do HDL gerado operando com uma célula com comportamento correto enquanto a Figura 19 corresponde à operação com a descrição que contém uma falha no comportamento. Neste sentido, a Figura 18 apresenta a correta operação por todos os estados, enquanto a Figura 19 a simulação para no estado em que ocorre a falha.

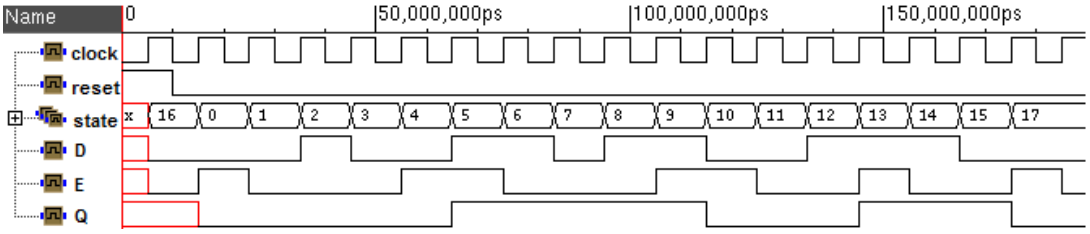


Figura 18. Simulação realizada para o LatD.

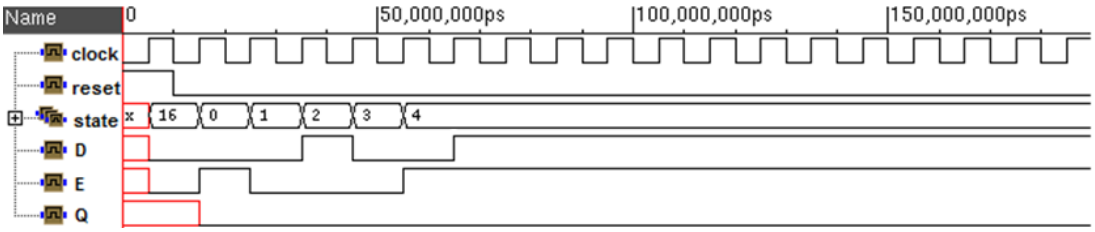


Figura 19. Simulação realizada para o LatD com falha.

4. RESULTADOS

Nessa seção, serão apresentados os resultados obtidos por meio do procedimento descrito para diferentes células sequenciais. Primeiramente, serão analisados os resultados do procedimento de aquisição dos dados de cada elemento verificado e da geração das descrições do *hardware* em VHDL e Verilog. Depois, os dados obtidos da síntese no fluxo de células padrão.

O algoritmo para geração dos HDLs foi escrito em linguagem Python, que é uma linguagem interpretada, própria para *scripts*. Sua implementação simplificada, com alto nível de abstração, se mostrou apropriada para o desenvolvimento de *scripts* para diferentes funções.

Os algoritmos desenvolvidos têm a característica de gerar um circuito para completa validação lógica de qualquer elemento sequencial cujo comportamento é conhecido. Para tal, são mapeados todos os estados estáveis e transições do dispositivo e um caminho de teste que passe por todos os estados e transições possíveis é definido. A síntese lógica no fluxo de células padrão foi utilizada para obter dados sobre características de área dos circuitos desenvolvidos. Os testes foram repetidos para oito dispositivos diferentes, nomeados conforme a Tabela *5*. A descrição comportamental de cada um desses dispositivos está apresentada no Apêndice C.

Tabela 5. Dispositivos utilizados para teste.

LatD	*Latch* tipo D sensível a nível alto
LatDS	*Latch* tipo D sensível a nível alto com *Set*
LatDR	*Latch* tipo D sensível a nível alto com *Reset*
LatDSR	*Latch* tipo D sensível a nível alto com *Set* e *Reset*
FFD	*Flip-Flop* tipo D sensível a borda de subida de *clock*
FFDS	*Flip-Flop* tipo D sensível a borda de subida de *clock* com *Set* assíncrono
FFDR	*Flip-Flop* tipo D sensível a borda de subida de *clock* com *Reset* assíncrono
FFDSR	*Flip-Flop* tipo D sensível a borda de subida de *clock* com *Set* e *Reset* assíncronos

4.1. Dados para geração do código HDL

As descrições dos 8 circuitos apresentados na Tabela *5* foram utilizadas como entradas no procedimento proposto. Nesta seção, serão apresentados os dados que qualificam os elementos em termos de estados estáveis e transições possíveis. Também é apresentada a quantidade de transições necessárias para a validação do dispositivo. Este conjunto de transições foi obtido conforme o método proposto no Capítulo 3. Todo esse conjunto de informações está apresentado nesta mesma ordem na Tabela 6.

Tabela 6. Dados obtidos pelo algoritmo.

Dispositivo	**Estados estáveis**	**Transições possíveis**	**Transições para teste completo**	**Eficiência**
LatD	6	12	15	0,80
LatDS	10	30	90	0,33
LatDR	10	30	121	0,25
LatDSR	14	48	71	0,68
FFD	8	16	31	0,52
FFDS	12	36	127	0,28
FFDR	12	36	146	0,25
FFDSR	16	56	84	0,67

Os estados estáveis e transições representam dados intrínsecos do dispositivo sob teste, que só dependem da descrição comportamental apresentada. É importante notar que esses valores representam cada tipo de

dispositivo, independentemente de sua arquitetura. Ou seja, qualquer LatD terá 6 estados estáveis e 12 transições possíveis, podendo ser representado por uma máquina de estados finitos, como a mostrada na Figura 20. A escolha do nome dos estados foi tal que representassem as entradas e saídas do LatD, na sequência *D*, *E* e *Q*.

As transições para teste completo definem características do testador, que são usadas para a geração da descrição em HDL. Esses dados dependem do caminho de teste gerado. Os resultados obtidos demonstram que o algoritmo guloso desenvolvido é eficiente para alguns casos, como o LatD, o LatDSR e o FFDSR, mas ineficiente em outros, como o LatDR e o FFDR, que utilizam muitas repetições de transições para garantir que todas as transições são avaliadas. Isso evidencia a dificuldade do algoritmo guloso de alcançar alguns estados.

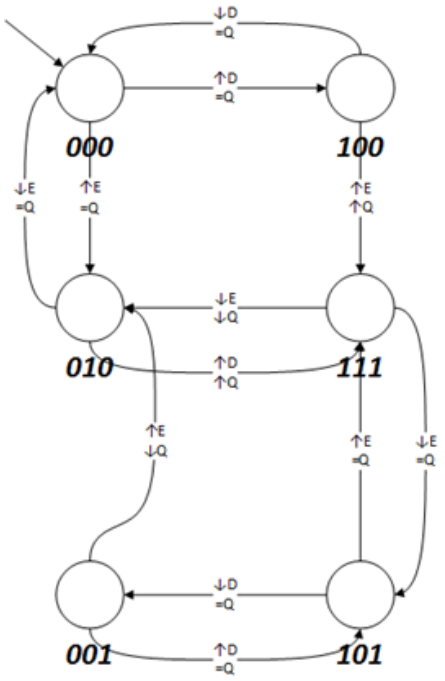


Figura 20. FSM representando o LatD.

A dificuldade de alcançar alguns estados ocorre principalmente devido ao fato de as transições não serem simétricas, ou seja, não ocorrerem em duas vias. Esse comportamento explica por que os piores desempenhos ocorreram nos elementos com apenas *set* ou apenas *reset*. A quantidade de transições que ocorrem em apenas uma via aumenta com a presença dessas entradas. Além disso, alguns estados são alcançados apenas pela transição de *set* ou *reset*, mas podem levar a diferentes estados. Isso faz com que a mesma transição

necessite ser percorrida mais vezes, o que é dificultado pelo aumento em seu peso. Em contrapartida, os elementos com *set* e *reset* mostraram desempenho mais aceitável, pois as transições de uma das entradas assíncronas compensam as transições da outra, facilitando o caminho de teste.

Se observou também que o estado inicial escolhido pode influenciar em parâmetros do testador, como número de estados necessários para o teste. O número de estados do testador irá interferir diretamente na área que ocupará no circuito em sua fabricação e no tempo necessário para o teste, sendo desejável que seja o menor possível.

4.2. Síntese do Circuito no Fluxo *Standard Cell*

De posse dos HDLs gerados pelo procedimento, realizou-se a síntese dos circuitos no fluxo comercial. Esse experimento é importante, pois gera uma série de informações a respeito do *hardware* desenvolvido, como área, número de células utilizadas, número de *flip-flops*, entre outras informações.

O primeiro passo é a realização da síntese lógica do projeto. A síntese lógica irá converter a descrição HDL do circuito gerado pelo procedimento para um conjunto de portas lógicas que possa ser implementado fisicamente, por meio de uma síntese física. Após os resultados desse processo, o *hardware* pode ser enviado para fabricação, de forma que essa validação possa ser feita em silício.

A síntese no fluxo de células padrão foi feita utilizando-se uma ferramenta comercial e duas bibliotecas de células comerciais. Inicialmente, foi utilizada uma biblioteca de células na tecnologia de 180nm. Posteriormente, com a disponibilidade do envio para fabricação, outra síntese foi feita, na tecnologia de 600nm. A Tabela 7 mostra os dados analisados: Número de *flip-flops*, número total de células e área total do testador. Todos esses dados são proporcionais ao número de estados necessários para o teste completo. Nesse circuito, serão necessários tantos *flip-flops* quantos *bits* para representação de todos os estados da sequência de teste. A todos os modelos de teste devem ainda ser adicionados mais 2 estados: Estado inicial (SInit) e estado de teste completo (Scompleto).

Um ponto interessante a ser observado é a diferença no número de portas lógicas utilizadas para implementar o mesmo circuito em cada uma das tecnologias avaliadas. Essa diferença se deve ao fato das bibliotecas de células padrão de cada tecnologia possuírem diferentes conjuntos de portas lógicas. Muito provavelmente a ferramenta de síntese utilizou uma maior quantidade de portas mais complexas na tecnologia de 600nm enquanto que na tecnologia de 180nm foram utilizadas portas lógicas mais simples em maior quantidade.

Em uma biblioteca de células padrão, normalmente a mesma função lógica é implementada com diferentes *drive strengths*. O conceito de *drive strength* pode ser simplificado pela capacidade que a porta lógica possui de conduzir corrente. Normalmente, quanto maior o *drive strength* da porta lógica, maior a área que a porta lógica ocupa. Na Tabela 8 são apresentadas as áreas limites (com menor e maior *drive strength*, respectivamente) das células sequenciais analisadas neste trabalho. A tecnologia em questão é a de 180nm.

Tabela 7. Dados obtidos da síntese no fluxo de células padrão.

Dispositivo	**#FFs no testador**	**180 nm**		**600 nm**	
		Portas lógicas	**Área (μm²)**	**Portas lógicas**	**Área (μm²)**
DLat	5	57	1014	39	31784
DLatS	7	287	4227	196	140180
DLatR	7	297	4392	227	165779
DLatSR	7	204	3098	191	146097
DFF	5	50	925	52	40761
DFFS	8	230	3591	196	147526
DFFR	8	276	4168	227	170582
DFFSR	7	219	3253	193	137329
Total	54	1620	24668	1321	980038

De posse da informação da área da célula sequencial e da área do circuito testador sintetizado para validar a mesma, é possível avaliar o custo em termos de área que a solução proposta apresenta. Neste sentido, a última coluna da Tabela 8 apresenta a razão entre o circuito testador e a área da célula sob teste, levando em consideração a área do menor e do maior *drive strength* de cada dispositivo sequencial da biblioteca. Para cada tipo de elemento, a área do

testador é sempre a mesma, independente do *drive strength* do dispositivo, já que o teste depende somente das entradas e saídas do circuito sob teste.

Tabela 8. Área de portas lógicas compatíveis com as testadas, com diferentes drive strengths, pertencentes à biblioteca de 180 nm utilizada e sua razão com a área do testador.

Porta lógica	Área das portas sequenciais (μm²)	Razão Testador / porta sequencial
DLat	36,9 – 55,3	27,5 – 18,3
DLatS	43,0 – 64,6	98,3 – 65,4
DLatR	36,9 – 64,6	119,0 – 68,0
DLatSR	46,1 – 70,7	67,2 – 43,8
DFF	52,3 – 83,0	17,7 – 11,1
DFFS	58,4 – 86,1	61,5 – 41,7
DFFR	61,5 – 92,2	67,8 – 45,2
DFFSR	67,6 – 98,4	48,1 – 33,1

Se comparado com a quantidade de células do circuito gerado pela síntese lógica, os números apresentados são mais animadores. Isso se deve pelo fato das células combinacionais terem normalmente uma área menor do que a célula sequencial sob teste. Contudo, a necessidade de um circuito com uma área 100 vezes maior do que a área da célula sob teste pode vir a se tornar inconveniente, já que podem existir várias células sequenciais diferentes em uma biblioteca, exigindo uma área muito grande para o teste. Este número torna mais evidente a necessidade de um algoritmo mais otimizado para a geração da sequência de transições que irá ser utilizada pelo testador.

Por outro lado, mesmo com essa razão alta, a área reservada para o testador fica em 4392 μm², o que ainda seria menor que a ducentésima parte de um *chip* de apenas 1 mm², ou seja, menos de 0,5% do *chip* será usado pelo testador no caso de maior área. Considerando-se que o teste seja feito para todos os 10 dispositivos estudados em um único testador, o percentual para um *chip* de 1 mm² seria de cerca de 2,5% de sua área total. O tamanho do testador se torna bem mais relevante quando se considera uma biblioteca de 600 nm. Como mostrado na tabela, nesse caso a área sobe para 0,98 mm², cerca de 40 vezes maior que na tecnologia de 180 nm.

Como o *script* desenvolvido pode gerar o testador tanto em Verilog quanto VHDL, também pôde ser avaliada a diferença entre as sínteses realizadas na mesma tecnologia e com a mesma biblioteca para o código gerado em cada uma das linguagens. Os resultados obtidos estão apresentados na Tabela 9. Pode-se observar que apesar da biblioteca de células ser a mesma, e o circuito de entrada descrever o mesmo circuito, existem diferenças entre as soluções. Isso se deve a forma como os algoritmos de síntese armazenam os circuitos na sua estrutura de dados interna. De todo modo, os resultados apresentam grande similaridade, com áreas ligeiramente menores para os códigos em VHDL na maioria dos casos, apesar de sua maior quantidade de portas lógicas.

Tabela 9. Comparação da área e do número de portas lógicas para códigos em Verilog e VHDL

Dispositivo	**Verilog**		**VHDL**	
	Portas lógicas	**Área (μm²)**	**Portas lógicas**	**Área (μm²)**
LatD	49	962	45	864
LatDS	197	3221	210	3258
LatDR	210	3432	225	3394
LatDSR	160	2747	171	2666
FFD	64	1215	66	1107
FFDS	198	3228	209	3237
FFDR	200	3309	197	3003
FFDSR	219	3538	223	3440

4.3. Síntese Física e Fabricação

Por fim, foi realizada uma síntese física do projeto, na tecnologia de 600 nm, para que fosse enviado para a fabricação em um projeto conjunto entre UFRGS, FURG e CEITEC S.A.. Para evitar a necessidade de se utilizar muitos pinos, um módulo superior foi utilizado, ligando todos os testadores desenvolvidos a um multiplexador e aos dispositivos sob teste. Dessa forma, o número de pinos utilizados corresponde ao número de *bits* de saída do maior circuito verificador, mais 3 pinos para controle do multiplexador. Nesse caso, foram 8 *bits*, correspondentes à quantidade necessária para o testador do FFDR. Na Figura 21 é mostrado o leiaute completo do projeto, junto a outros circuitos

que foram sintetizados em paralelo no mesmo *chip*. O *chip* enviado para a fabricação possui uma área de 3 mm², sendo que a área circulada na figura representa a parte que foi utilizada para o projeto. Além disso, 12 pinos do *chip* foram utilizados, sendo 1 pino de reset, que reinicia o circuito de *debug*, 3 pinos de entrada para o seletor do multiplexador e 8 pinos de saída que mostram o estado de parada. No momento da publicação desse trabalho, o projeto fabricado ainda não havia retornado para verificações.

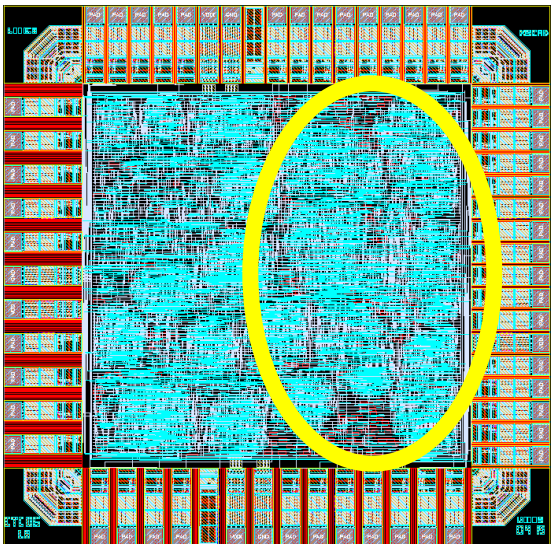


Figura 21. Leiaute do chip enviado para fabricação com destaque para a parte referente a esse projeto.

4.4. Considerações Finais

O sistema desenvolvido tem a capacidade de detectar diversos tipos de falhas. Qualquer falha de interconexão aberta ou em curto, curto para a fonte ou para a terra, *stuck-open* ou *stuck-on* pode ser verificada com o circuito, desde que seja detectável. A primeira situação que torna a falha observável também pode ser observada. Porém, a característica sequencial do circuito não permite que o teste continue após a primeira ocorrência, dificultando analisar outras situações em que a falha possa ser observada, o que reduz a capacidade de isolar o local exato e o motivo da ocorrência da falha.

Também é importante observar que o circuito desenvolvido se concentra na validação lógica dos elementos sob teste. Porém, a mesma estrutura pode ser usada para outros tipos de validação a serem estudadas no futuro. Por exemplo, ao se analisar o dispositivo em diferentes frequências, pode-se fazer uma análise de desempenho.

5. CONCLUSÕES E TRABALHOS FUTUROS

A escolha e o projeto de *latches* e *flip-flops* é de fundamental importância para o projeto de circuitos integrados. Além disso, a dimensão dos transistores diminui continuamente, conforme a lei de Moore (MOORE, 1965), fazendo muitas vezes necessário reprojetar os *flip-flops* a cada novo nodo tecnológico. Como mesmo alterações nas dimensões dos transistores podem influenciar o funcionamento do dispositivo (ALIOTO, CONSOLI e PALUMBO, 2011), a validação do elemento a cada mudança é muito importante. Com os altos custos associados ao teste e *debug* de circuitos sequenciais, surge o desenvolvimento de técnicas que tornem esse processo mais simples e barato para o projetista.

Esse trabalho propôs um método de autoteste para células sequenciais em bibliotecas de células. Esse é um passo fundamental para a garantia de que novas bibliotecas de células funcionem perfeitamente. Os circuitos obtidos atingem a completa validação lógica de portas lógicas sequenciais. Além disso, as condições para um bom *debug* propostas por (VERMEULEN e GOEL, 2002) foram atendidas. O acesso aos pinos funcionais do *chip* e aos sinais e memórias internos do chip são claros devido à ligação direta ao testador e a execução pode ser controla, já que se trata de uma máquina de estados.

A solução proposta apresenta um elevado custo em termos de área. O algoritmo utilizado no projeto foi um algoritmo guloso que percorre a matriz de adjacências da máquina de estados que testa o dispositivo. Essa estrutura se mostrou eficiente para o teste do LatD e do FFD e até dos elementos com *Set* e *Reset*. Porém, foi bastante ineficiente para encontrar uma sequência que validasse todas as transições de *latches* e *flip-flops* somente com *Set* ou *Reset*. Isso aumentou consideravelmente a área do testador que realiza toda a verificação. Isso se deve ao fato de que a área depende diretamente da quantidade de estados necessária para completar o teste. Um importante

trabalho futuro é a implementação de um algoritmo mais eficiente para a aquisição da sequência de teste.

É necessário observar também que para o início do teste de um dispositivo sequencial é necessário o conhecimento de suas condições iniciais. O único jeito de realizar isso é forçando uma condição inicial conhecida. No caso do LatD, basta existir um nível alto na entrada de habilitação que a saída será igual à entrada D, mas isso pode mudar para diferentes tipos de *latches*. No caso dos *flip-flops*, é necessária uma transição de *clock* que transmita a entrada para a saída. Quando o elemento possui entradas *Set* ou *Reset* assíncronas, essas entradas também podem ser usadas para definição da condição inicial. Também se observou que a escolha da condição inicial pode provocar um aumento ou diminuição na quantidade de estados necessários para realização do teste completo. No trabalho realizado, as condições iniciais foram obtidas manualmente por observação. Um estudo futuro pode gerar uma automatização da geração de estados iniciais. Como existem diferentes possibilidades, também pode ser feita uma seleção dos estados iniciais ideais de cada elemento para que o teste seja ótimo.

Os testadores gerados possuem capacidade para testar apenas um dispositivo, porém, uma mesma biblioteca pode ter diferentes implementações de um dispositivo com mesmo comportamento. Portanto, também deve ser verificado o uso de paralelismo ou multiplexadores para testar diversos dispositivos do mesmo tipo com apenas uma máquina de estados em silício.

REFERÊNCIAS

ALIOTO, M.; CONSOLI, E.; PALUMBO, G. Analysis and Comparison in the Energy-Delay-Area Domain of Nanometer CMOS Flip-Flops: Part I - Methodology and Design Strategies. **IEEE TRANSACITIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS**, v. 19, n. 5, p. 725-736, Maio 2011.

ALIOTO, M.; CONSOLI, E.; PALUMBO, G. Analysis and Comparison in the Energy-Delay-Area Domain of Nanometer CMOS Flip-Flops: Part II - Results and Figures of Merit. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 19, n. 5, p. 737-750, Maio 2011.

AVELAR, H. H.; BUTZEN, P. F.; ROSA, V. S. Validating Logical Functionality of Sequential Circuits Using Finite State Machines. **SIM**, Alegrete, 9 Maio 2014.

BALEN, T. R.; LUBASZEWSKI, M. Teste e Projeto Visando a Testabilidade de Circuitos e Sistemas Integrados. In: _____ **Tópicos em Micro e Nano Eletrônica**. 1ª. ed. Ijuí - RS: [s.n.], 2014. p. 219-264.

BAVARESCO, S. **On-Silicon Testbenh for Validation of Soft Logic Cells Libraries**. Porto Alegre. 2008.

BHUSHAN, M. et al. Ring oscillator based technique for measuring variability statistics. **IEEE International Conference on Microelectronic Test Structures**, 2006. 87-92.

BREUER, M.; ABRAMOVICI, M.; FRIEDMAN, A. D. **Digital systems testing and testable design**. New York, USA: AT&T Bell Laboratories and WH Freeman, 1990.

BUTZEN, P. F. **Aging Aware Design Techniques and CMOS Gate Degradation Estimative**. UFRGS. Porto Alegre, p. 79. 2012.

CHAMPAC, V. H.; ZENTENO, A.; GARCÍA, J. L. Testing of Resistive Opens in CMOS Latches and Flip-flops. **Proceedings of the European Test Symposium**, 2005.

DATTA, R. et al. On-chip Delay Measurement for Silicon Debug. **Proceedings of GLSVLSI**, 2004. 145-148.

JAIN, A. et al. An On-Chip Flip-Flop Characterization Circuit. In: _____ **Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation**. [S.l.]: Springer, 2011. p. 41-50.

JHA, N. K.; KUNDU, S. **Testing and Raliable Design of CMOS Circuits**. 1ª. ed. Norwell - USA: Kluwer Academic Publishers, 1990.

LIN, R. B.; SHOU, S. H.; TSAI, C. M. Benchmark Circuits Improve the Quality of a Standard Cell Library. **Asia and South Pacific Design Automation Conference**, Hong Kong, 1999. 173-176.

MAKAR, S. R.; MCCLUSKEY, E. J. Checking Experiments To Test Latches. **VLSI Test Symposium**, 30 April 1995. 196-201.

MAKAR, S.; MCCLUSKEY, E. Iddq Test Pattern Generation for Scan Chain Latches and Flip-Flops. **IEEE International Workshop on IDDQ Testing**, 1997. 2-6.

MOORE, G. E. Cramming more components onto integrated circuits, p. 114-117, 1965.

NEDOVIC, N.; WALKER, W. W.; OKLOBDZIJA, V. G. A test circuit for measurement of clocked storage element characteristics. **Solid-State Circuits, IEEE Journal**, v. 8, n. 39, p. 1294-1304, 2004.

NIERMANN, T. M. et al. Test Compaction for Sequential Circuits. **IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN**, v. II, n. 2, p. 260-267, set. 1990. ISSN 0278-0070/92.

RABAEY, J. M.; CHANDRAKASAN, A.; NICOLIC, B. **Digital Integrated Circuits:** A Design Perspective. New Jersey: Pearson, 2003.

RIBAS, R. et al. Ring Oscillators for Functional and Delay Test. **SBCCI'11**, João Pessoa, 30 Agosto 2011. 67-72.

RIBAS, R. P. et al. Circuit Design for Testing Standard Cell Libraries. **WCAS**, João Pessoa, 30 Agosto 2011.

RIBAS, R. P. et al. Contributions to the evaluations of ensembles of combinational logic gates. **Microelectronics Journal**, n. 42, p. 371-381, 2011.

SCOTT, K.; KEUTZER, K. Improving cell libraries for synthesis. **Proceedings of the IEEE**, Maio 1994. 128-131.

TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. **Sistemas Digitais:** Princípios e Aplicações. [S.l.]: Pearson, 2010.

VAHID, F. **Sistemas Digitais**. Porto Alegre: Bookman, 2008.

VERMEULEN, B.; GOEL, S. K. Design for Debug: Catching Design Errors in Digital Chips. **IEEE Design & Test of Computers**, May 2002. 37-45.

WESTE, N. H.; HARRIS, N. M. **CMOS VLSI Design:** A Circuits and Systems Perspective. 4th Edition. ed. Boston: Pearson, 2011.

WILLIANS, T.; PARKER, K. Design for Testability - A Survey. **IEEE Transactions on Computers**, v. C-31, n. 1, p. 98-112, Janeiro 1982.

ZIMPECK, A. L.; BUTZEN, P.; MEINHARDT, C. Análise do comportamento de portas lógicas CMOS com falhas Stuck-On em nanotecnologias. **ICCEEg**, v. 1, n. 7, p. 1-10, 2014.

APÊNDICE A – Artigos Publicados Durante o Mestrado

- AVELAR, Helder H.; BUTZEN, Paulo F.; ROSA, Vagner S. **VALIDATING LOGICAL FUNCTIONALITY OF SEQUENTIAL CIRCUITS USING FINITE STATE MACHINES**. **29º Simpósio Sul de Microeletrônica**, Alegrete, Brasil, Maio de 2014.
- AVELAR, Helder H.; BUTZEN, Paulo F.; FRANCO, Denis T. **ANÁLISE DE FALHAS EM FILTROS DE IMPULSO FINITO. 13ª Mostra de Produção Universitária**, Rio Grande, BR, Outubro de 2014.
- AFONSO, Renato M.; AVELAR, Helder H. BUTZEN, Paulo F.; MEINHARDT, Cristina **PROCEDIMENTO PARA CARACTERIZAÇÃO TEMPORAL DE FLIP-FLOPS. 13ª Mostra de Produção Universitária**, Rio Grande, BR, Outubro de 2014.
- AVELAR, Helder H. **CIRCUIT DESIGN FOR SEQUENTIAL LOGIC CELLS VALIDATION. LASCAS 2015 Forum for Young Professionals/PhD/MSc Students.** Montevidéu, Uruguai, Fevereiro de 2015.
- AVELAR, Helder H.; BUTZEN, Paulo F.; FRANCO, Denis T. **ANÁLISE DA ROBUSTEZ A FALHAS DE UM FILTRO FIR. Iberchip 2015.** Montevidéu, Uruguai, Fevereiro de 2015.
- AVELAR, Helder H.; BUTZEN, Paulo F.; RIBAS, Renato P. **AUTOMATIC TEST SEQUENCE GENERATION FOR SEQUENTIAL LOGIC VALIDATION. 30º Simpósio Sul de Microeletrônica.** Santa Maria, Brasil, Maio de 2015.
- AVELAR, Helder H.; BUTZEN, Paulo F.; RIBAS, Renato P. **AUTOMATIC CIRCUIT GENERATION FOR SEQUENTIAL LOGIC DEBUG. ICECS 2016.**

APÊNDICE B – HDLs Gerados para o *Latch* tipo *D*

Verilog:

```
module latd (clock, reset, q, state_out, OUT0, OUT1);
parameter SIZE = 5;
//-------------Input Ports-----------------------------
input     :          clock, reset, q;
//-------------Output Ports----------------------------
output    [SIZE-1:0] state_out;
output    OUT0, OUT1;
//-------------Input ports Data Type-------------------
wire    clock, reset, q;
//-------------Output Ports Data Type------------------
reg     OUT0, OUT1;
//-------------Internal Constants----------------------------
-
parameter S0 = 5'b00000, S1 = 5'b00001, S2 = 5'b00010, S3 = 5'b00011, S4 = 5'b00100, S5 = 5'b00101, S6 = 5'b00110, S7 = 5'b00111, S8 = 5'b01000, S9 = 5'b01001, S10 = 5'b01010, S11 = 5'b01011, S12 = 5'b01100, S13 = 5'b01101, S14 = 5'b01110, S15 = 5'b01111, SInit = 5'b10000, SComplete = 5'b10001;
//-------------Internal Variables----------------------------
reg   [SIZE-1:0]        state       ;// Seq part of the FSM
wire  [SIZE-1:0]        next_state   ;// combo part of FSM
//----------Code startes Here------------------------
assign next_state = fsm_function(state, q);
//----------Function for Combo Logic-----------------
function [SIZE-1:0] fsm_function;
          input  [SIZE-1:0]  state ;
          input    q ;
          case(state)
                    SInit :
                              fsm_function = S0;
                    S0 : if (q == 1'b0) begin
              fsm_function = S1;
end else begin
                fsm_function = S0;
                                        end

                    S1 : if (q == 1'b0) begin
              fsm_function = S2;
end else begin
                fsm_function = S1;
                                        end

                    S2 : if (q == 1'b0) begin
              fsm_function = S3;
end else begin
                fsm_function = S2;
                                        end

                    S3 : if (q == 1'b0) begin
              fsm_function = S4;
end else begin
                fsm_function = S3;
                                        end

                    S4 : if (q == 1'b0) begin
              fsm_function = S5;
end else begin
                fsm_function = S4;
                                        end

                    S5 : if (q == 1'b1) begin
              fsm_function = S6;
end else begin
                fsm_function = S5;
                                        end

                    S6 : if (q == 1'b1) begin
              fsm_function = S7;
end else begin
                fsm_function = S6;
                                        end

                    S7 : if (q == 1'b1) begin
              fsm_function = S8;
end else begin
                fsm_function = S7;
                                        end

                    S8 : if (q == 1'b1) begin
              fsm_function = S9;
end else begin
                fsm_function = S8;
                                        end

                    S9 : if (q == 1'b1) begin
              fsm_function = S10;
end else begin
                fsm_function = S9;
                                        end

                    S10 : if (q == 1'b0) begin
              fsm_function = S11;
end else begin
                fsm_function = S10;
                                        end
```

```
                    S11 : if (q == 1'b0) begin
              fsm_function = S12;
end else begin
                fsm_function = S11;
                                    end

                    S12 : if (q == 1'b0) begin
              fsm_function = S13;
end else begin
                fsm_function = S12;
                                    end

                    S13 : if (q == 1'b1) begin
              fsm_function = S14;
end else begin
                fsm_function = S13;
                                    end

                    S14 : if (q == 1'b1) begin
              fsm_function = S15;
end else begin
                fsm_function = S14;
                                    end
                    S15 : if (q = '1') begin
                fsm_function = SComplete;
end else begin
                fsm_function = S15;
                                    end
          SComplete : fsm_function = SComplete;
          default : fsm_function = SInit;
  endcase
endfunction

assign state_out = state;

//----------Seq Logic-----------------------------
always @ (posedge clock)
begin : FSM_SEQ
  if (reset == 1'b1) begin
    state <=  #1  SInit;
  end else begin
    state <=  #1  next_state;
  end
end

//----------Output Logic-----------------------------
always @ (posedge clock)
begin : OUTPUT_LOGIC
if (reset == 1'b1) begin
          OUT0 <= #1 1'b0;
          OUT1 <= #1 1'b0;
end
else begin
   case(state)
     SInit : begin
   OUT0 <= #1 1'b1;
   OUT1 <= #1 1'b0;
   end
     S0 : begin
   OUT0 <= #1 1'b0;
   OUT1 <= #1 1'b0;
end
     S1 : begin
   OUT0 <= #1 1'b0;
   OUT1 <= #1 1'b1;
end
     S2 : begin
   OUT0 <= #1 1'b0;
   OUT1 <= #1 1'b0;
end
     S3 : begin
   OUT0 <= #1 1'b1;
   OUT1 <= #1 1'b0;
end
     S4 : begin
   OUT0 <= #1 1'b1;
   OUT1 <= #1 1'b1;
end
     S5 : begin
   OUT0 <= #1 1'b0;
   OUT1 <= #1 1'b1;
end
     S6 : begin
   OUT0 <= #1 1'b0;
   OUT1 <= #1 1'b0;
end
     S7 : begin
   OUT0 <= #1 1'b0;
   OUT1 <= #1 1'b1;
end
     S8 : begin
   OUT0 <= #1 1'b1;
   OUT1 <= #1 1'b1;
end
     S9 : begin
   OUT0 <= #1 1'b1;
   OUT1 <= #1 1'b0;
end
     S10 : begin
   OUT0 <= #1 1'b0;
   OUT1 <= #1 1'b0;
end
     S11 : begin
   OUT0 <= #1 1'b0;
   OUT1 <= #1 1'b1;
end
     S12 : begin
   OUT0 <= #1 1'b1;
   OUT1 <= #1 1'b1;
```

```
      end
          S13 : begin
         OUT0 <= #1 1'b0;
         OUT1 <= #1 1'b1;
      end
          S14 : begin
         OUT0 <= #1 1'b0;
         OUT1 <= #1 1'b0;
      end
          S15 : begin
         OUT0 <= #1 1'b1;
         OUT1 <= #1 1'b0;
      end
      default : begin
                OUT0 <=  #1  1'b0;
                OUT1 <=  #1  1'b0;
                end
    endcase
   end
   end // End Of Block OUTPUT_LOGIC

   endmodule // End of Module arbiter
```

VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FSM is

port (clk, reset: in STD_LOGIC;
          Q:         in STD_LOGIC;
          OUT0, OUT1:        out STD_LOGIC
          state_out:          out
STD_LOGIC_VECTOR (4 down to 0)
                              );
end FSM;

architecture behavioral of FSM is
   type statetype is (SInit, S0, S1, S2, S3, S4, S5,
S6,  S7,  S8,  S9,  S10,  S11,  S12,  S13,  S14,
Complete);
          signal state, nextstate: statetype;

begin

          changeState: process(clk) begin
                    if rising_edge(clk) then
                              if reset = '1' then

          state <= SInit;
                    else state <= nextstate;
                              end if;
          end if;
end process;

stateProc: process (state, q)
begin
          OUT0<= '0';
          OUT1<= '0';
          nextstate <= S0;

case state is
          when SInit =>
                    nextstate <= S0;
                    OUT0<='0';
                    OUT1<='1';
          when S0=>
                    if (q = '1') then
                    nextstate <= S1;
                    OUT0<='0';
                    OUT1<='0';
                    else
                    nextstate <= S0;
                    end if;
          when S1=>
                    if (q = '1') then
                    nextstate <= S2;
                    OUT0<='0';
                    OUT1<='1';
                    else
                    nextstate <= S1;
                    end if;
          when S2=>
                    if (q = '1') then
                    nextstate <= S3;
                    OUT0<='1';
                    OUT1<='1';
                    else
                    nextstate <= S2;
                    end if;
          when S3=>
                    if (q = '1') then
                    nextstate <= S4;
                    OUT0<='0';
                    OUT1<='1';
                    else
                    nextstate <= S3;
                    end if;
          when S4=>
                    if (q = '1') then
                    nextstate <= S5;
                    OUT0<='0';
                    OUT1<='0';
                    else
                    nextstate <= S4;
                    end if;
          when S5=>
```

```
            if (q = '1') then
            nextstate <= S6;
            OUT0<='1';
            OUT1<='0';
            else
            nextstate <= S5;
            end if;
when S6=>
            if (q = '0') then
            nextstate <= S7;
            OUT0<='0';
            OUT1<='0';
            else
            nextstate <= S6;
            end if;
when S7=>
            if (q = '0') then
            nextstate <= S8;
            OUT0<='0';
            OUT1<='1';
            else
            nextstate <= S7;
            end if;
when S8=>
            if (q = '0') then
            nextstate <= S9;
            OUT0<='0';
            OUT1<='0';
            else
            nextstate <= S8;
            end if;
when S9=>
            if (q = '0') then
            nextstate <= S10;
            OUT0<='1';
            OUT1<='0';
            else
            nextstate <= S9;
            end if;
when S10=>
            if (q = '0') then
            nextstate <= S11;
            OUT0<='1';
                        OUT1<='1';
                        else
                        nextstate <= S10;
                        end if;
                  when S11=>
                        if (q = '1') then
                        nextstate <= S12;
                        OUT0<='1';
                        OUT1<='0';
                        else
                        nextstate <= S11;
                        end if;
                  when S12=>
                        if (q = '0') then
                        nextstate <= S13;
                        OUT0<='0';
                        OUT1<='0';
                        else
                        nextstate <= S12;
                        end if;
                  when S13=>
                        if (q = '0') then
                        nextstate <= S14;
                        OUT0<='0';
                        OUT1<='1';
                        else
                        nextstate <= S13;
                        end if;
                  when S14=>
                        if (q = '0') then
                  nextstate <= Complete;
                        else
                        nextstate <= S14;
                        end if;
when Complete => nextstate <= Complete;
                  when others =>

            end case;
   end process;

end behavioral;
```

APÊNDICE C – Protótipo das Funções de Descrição Comportamental dos Dispositivos Testados

Latch tipo D

```
LatD (state0, state1):
    if EN == 1:
        Q = D
    else:
        Q = Q0
    return Q
```

Latch tipo D com Set

```
LatDS (state0, state1):
    if S == 0:
        Q = 1
    elif EN == 1:
        Q = D
    else:
        Q = Q0
    return Q
```

Latch tipo D com Reset

```
LatDR (state0, state1):
    if R == 0:
        Q = 0
    elif EN == 1:
        Q = D
    else:
        Q = Q0
    return Q
```

Flip-Flop tipo D

```
FFD (state0, state1):
    if clk == 1 and clk0 == 0:
        Q = D
    else:
        Q = Q0
    return Q
```

Latch tipo D com Set e Reset (com prioridade para Reset)

```
LatDSR (state0, state1):
    if R == 1:
        Q = 0
    elif S == 1:
        Q = 1
    elif EN == 1:
        Q = D
    else:
        Q = Q0
    return Q
```

Latch tipo D com Set e Reset

```
LatDSR0 (state0, state1):
    if S == 0 and R == 0:
        Q = "x"
    if R == 0:
        Q = 0
    elif S == 0:
        Q = 1
    elif EN == 1:
        Q = D
    else:
        Q = Q0
    return Q
```

Flip-Flop tipo D Falling-Edge

```
FFFD (state0, state1):
    if clk == 0 and clk0 == 1:
        Q = D
    else:
        Q = Q0
    return Q
```

Flip-Flop tipo D com Set

```
FFDS (state0, state1):
    if S == 0:
        Q = 1
    elif clk == 1 and clk0 == 0:
        Q = D
    else:
        Q = Q0
    return Q
```

Flip-Flop tipo D com reset

```
FFDR (state0, state1):
    if R == 0:
        Q = 0
    elif clk == 1 and clk0 == 0:
        Q = D
    else:
        Q = Q0
    return Q
```

Flip-Flop tipo D com Set e Reset

```
FFDSR (state0, state1):
    if S == 0 and R == 0:
        Q = "x"
    elif S == 0:
        Q = 1
    elif R == 0:
        Q = 0
    elif clk == 1 and clk0 == 0:
        Q = D
    else:
        Q = Q0
    return Q
```