UNIVERSIDADE DO VALE DO RIO DOS SINOS - UNISINOS
UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA
NÍVEL MESTRADO PROFISSIONAL

GUILHERME FERREIRA LOPES

MODELO FUNCIONAL DE MEMÓRIA NAND FLASH COM INJEÇÃO DE FALHAS CARACTERIZADAS

SÃO LEOPOLDO
2018

Guilherme Ferreira Lopes

Modelo funcional de memória NAND Flash com injeção de falhas caracterizadas

Dissertação apresentada como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica, pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade do Vale do Rio dos Sinos - UNISINOS.

Orientador: Prof. Dra. Margrit Krug

Coorientador: Prof. Ms. Lucio Rene Prade

São Leopoldo

2018

L864m Lopes, Guilherme Ferreira.
Modelo funcional de memória NAND Flash com injeção de falhas caracterizadas / Guilherme Ferreira Lopes. – 2018.
81 f. : il. color. ; 30 cm.

Dissertação (mestrado) – Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Engenharia Elétrica, São Leopoldo, 2018.
"Orientador: Prof. Dra. Margrit Krug ; Coorientador: Prof. Ms. Lucio Rene Prade."

1. Modelo funcional. 2. Memória NAND Flash. 3. Inserção de falhas. 4. Algoritmo de teste. 5. Interferência. I. Título.

CDU 621.3

Dados Internacionais de Catalogação na Publicação (CIP)
(Bibliotecária: Bruna Sant'Anna – CRB 10/2360)

Guilherme Ferreira Lopes

Modelo funcional de memória NAND Flash com injeção de falhas caracterizadas

Dissertação apresentada como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica, pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade do Vale do Rio dos Sinos - UNISINOS.

Aprovado em 28 de Maio de 2018.

BANCA EXAMINADORA:

Prof. Dr. Márcio Rosa da Silva – Unisinos
Avaliador

Prof. Dr. Rodrigo Marques de Figueiredo – Unisinos
Avaliador

Prof. Dr. Tiago Roberto Balen, – UFRGS
Avaliador Externo

Prof. Dra. Margrit Krug (Orientador)
Prof. Ms. Lucio Rene Prade (Coorientador)

Visto e permitida a impressão
São Leopoldo

Prof. Dr. Eduardo Luis Rhod
Coordenador PPG em Engenharia Elétrica

AGRADECIMENTOS

Gostaria de agradecer Igor Tedeschi Franco, bolsista do projeto e principal programador do modelo, aos meus orientadores Margrit Reni Krug e Lucio Rene Prade pelas correções e orientações fornecidas e para Júlia Gregol, revisora e apoiadora do projeto.

RESUMO

A memória NAND Flash lidera o mercado de memórias não voláteis por prover soluções para aplicações móveis, juntando alta densidade de armazenamento em uma área de silício muito pequena e consumindo pouca energia (RICHTER, 2014). Devido à mecanismos específicos para a realização de operações na memória, elas se tornam suscetíveis à falhas funcionais de interferências, assim aumentando a importância do teste(HOU; LI, 2014). Esta dissertação apresenta o projeto de um modelo funcional de memória NAND Flash com inserção de falhas caracterizadas em 2 etapas, a primeira etapa ocorreu utilizando a ferramenta Logisim^{TM}, projetada para desenvolver e simular circuitos lógicos de forma que possam ser apresentados visualmente, a segunda etapa consistiu no desenvolvimento também de forma modular e escalar em linguagem de descrição de hardware (VHDL). As 2 ferramentas possuem a implementação de um circuito de injeção de falhas, capaz de simular e aplicar falhas funcionais de interferência e *stuck-at* na memória desenvolvida. Com base no modelo comercial de memórias NAND Flash, o trabalho visa desenvolver os circuitos presentes na memória, respeitando a organização dos sinais e a organização das células em páginas e blocos, sendo uma característica específica para memórias NAND Flash. Após o desenvolvimento do modelo funcional, ocorreu a primeira etapa de verificação e validação da memória, composta pela varredura de endereços, criação e comparação dos valores esperados com valores de saída e utilização de algoritmos de teste para a validação final, finalizando o projeto com a verificação e validação de cada falha injetada para que assim tenha-se um modelo funcional de uma memória NAND Flash capaz de inserir uma determinada falha na posição exata da matriz de memória. Após a modelagem realizou-se simulações para avaliar aplicabilidade do projeto desenvolvido e os resultados mostram o atingimento de 100% de cobertura das falhas desenvolvidas, chegando ao objetivo de criar um modelo funcional para possibilitar a inserção de falhas foi atingido.

Palavras-chaves: Modelo funcional. Memória NAND Flash. Inserção de Falhas. Logisim^{TM}. VHDL. Algoritmo de Teste. Interferência. *Stuck-at*.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AF	*Address Decoder Fault* (Falha no decodificador de endereços)
ATE	*Automatic Test Equipment* (Equipamento automatico de teste)
BIST	*Built In Self Test* (Auto Teste Interno)
CFs	*State Coupling Fault* (Falha de acoplamento)
SAF	*Stuck at Fault* (Falha de circuito preso)
SLC	*Single Level Cell* (Célula com único nível)
SOF	*Stuck Open Fault* (Falha de circuito aberto)
SSD	*Solid State Drive* (Disco de estado sólido)
TF	*Transition Fault* (Falha de transição)

SUMÁRIO

1 INTRODUÇÃO

Dispositivos de Memórias Flash tornaram-se uma parte dominante no mercado de semicondutores, devido à suas características de armazenamento, que levaram o seu uso em dispositivos móveis, dentre eles os smartphones, os quais receberam destaque mundial nas vendas nos últimos anos. O mercado de memória não volátil tornou-se uma peça chave no mercado de semicondutores, pelo fato da memória possibilitar o armazenamento de informações em dispositivos eletrônicos após o seu desligamento. A memória NAND Flash lidera o mercado de não volátil por prover soluções para aplicações móveis, juntando alta densidade de armazenamento em uma área de silício muito pequena e consumindo pouca energia (RICHTER, 2014).

Segundo Hou e Li (2014), a capacidade de detecção de falhas é essencial em dispositivos semicondutores, especialmente em memórias que receberam um crescimento exponencial em densidade nos anos anteriores, portanto, aplicações como *Solid State Drive* (SSD), armazenamento em dispositivos móveis e automotivos, requerem dispositivos altamente confiáveis e testados para que sejam utilizados nos devidos sistemas, fazendo com que o teste de memórias Flash se torne fundamental para a indústria de semicondutores.

As memórias Flash podem ser divididas em dois tipos: NAND Flash e NOR Flash. Algumas características presentes na memória NAND Flash, fazem com que o processo de teste se torne mais complexo em relação ao NOR Flash, por exemplo, memórias NAND possuem alta densidade e podem ser acessadas somente de forma sequencial, o tempo das operações de programação e leitura é muito maior do que o apagamento, o número de entradas e saídas (I/O) é normalmente 8 ou 16 bits causando baixa banda de acesso para o teste e a organização interna da memória para programação e leitura é realizada em páginas, que normalmente giram em torno de 2k bits. Portanto, para testar memórias NAND Flash de alta densidade requer um alto consumo de tempo e diferentes tipos de implementações de algoritmos de teste, para que realizem a maior cobertura de falhas possível (MICHELONI; CRIPPA; MARELLI, 2010).

Neste projeto foi desenvolvido um modelo funcional de memória NAND Flash com inserção de falhas caracterizadas, sendo composto pela emulação do comportamento funcional da memória, um circuito para inserção de falhas no endereço desejado e a verificação e validação de cada circuito desenvolvido. Através da inserção de falhas em endereços da memória previamente definidos , o principal objetivo da dissertação é disponibilizar um modelo que possibilite o estudo, desenvolvimento e otimização de algoritmos de testes baseados em memórias NAND Flash, aplicados diretamente no modelo desenvolvido, para o qual pretende-se alcançar um modelo flexível para qualquer densidade de memória requerida, um modelo que possibilite a inserção de novas operações e funcionalidades, para a adição de novos modos de falhas e por fim para a utilização do projeto em projetos futuros que necessitem de uma memória NAND Flash validada.

1.1 Justificativa

Através das características funcionais da memória NAND Flash, que implicam em falhas de interferência entre células e falhas comuns entre memórias, juntamente com a importância de um modelo que possibilite o estudo, desenvolvimento e aplicação de algoritmos relacionas à memória, sendo este um modelo de memória NAND Flash capaz de inserir tipos de falhas implementadas no endereço desejado, respeitando sempre as características de funcionalidade da memória.

1.2 Definição do tema

Em ordem de desenvolver algoritmos de teste eficientes e soluções de gerenciamento de dados para memórias NAND Flash, torna-se necessário o entendimento consistente e aprofundado das características funcionais desta memória, juntamente com os tipos de falhas possíveis e seus comportamentos, para isto é fundamental o desenvolvimento de um modelo funcional de memória NAND Flash com inserção de falhas validado, capaz de inserir uma falha na posição conhecida, para que assim possibilite o desenvolvimento do teste para memórias NAND Flash.

1.3 Delimitação do trabalho

De acordo com Yun et al. (2012), falhas em memórias NAND Flash são classificadas em 2 tipos, falhas internas que ocorrem e aparecem através da execução de algumas funções, sendo falhas transientes que alteram o valor da célula alvo ou células vizinhas e falhas permanentes compostas pelo tipo de *Stuck-at*. O segundo tipo são falhas externas que se caracterizam por falhas de energia, ocorrendo no bloco da memória responsável por gerar energia para realizar as operações da memória. A etapa de injeção de falhas será desenvolvida considerando as falhas do tipo internas pois o modelo é composto somente pela parte funcional da memória NAND Flash, não levando em consideração o comportamento elétrico do dispositivo.

Falhas de endereçamento não foram consideradas no escopo de implementação, pois o objetivo do trabalho é o modelamento funcional da memória com injeção de falhas nas células de armazenamento, considerando sempre que o circuito de endereçamento não apresenta falhas no circuito.

A implementação do modelo funcional de memória NAND Flash foi iniciado utilizando a ferramenta Logisim^{TM}, designada para projetar e simular circuitos lógicos de forma visual e didática, tendo o objetivo de apresentar didaticamente o comportamento interno da memória e o comportamento da memória após cada falha ser inserida. Devido a limitações da ferramenta Logisim^{TM}com o aumento da complexidade do circuito, o desenvolvimento do modelo com inserção de falhas ocorreu também em linguagem de descrição de hardware (VHDL).

O modelo de memória NAND Flash modelada foi do tipo *Single Level Cell*, ou seja, para cada célula de memória modelada, foi considerado apenas 2 níveis lógicos de dados (0 ou 1). (RICHTER, 2014).

A interface de comunicação da memória baseou-se em memórias do tipo assíncronas, em que o clock está presente somente no controlador NAND Flash, responsável por realizar a comunicação com a memória, devido ao fato de memórias com interface do tipo assíncronas serem mais comuns no mercado mundial, segundo Semiconductor et al. (2011).

Para a validação do modelo e das falhas desenvolvidas foram utilizados algoritmos encontrados na literatura e aplicados comercialmente para memória NAND Flash, como o algoritmo March-FT apresentado pelo autor Yeh et al. (2002) e o algoritmo Bridging Faults & Disturbances apresentado pelo autor Carlo et al. (2010).

1.4 Objetivos

Neste tópico apresenta-se o objetivo geral e os objetivos específicos que devem ser atingidos para o sucesso do trabalho.

1.4.1 Objetivo Geral

O objetivo geral do trabalho é o desenvolvimento de um modelo funcional de memória NAND Flash com capacidade de inserção de falhas caracterizadas em endereços específicos. Disponibilizando assim, uma solução para desenvolvimento e estudo de algoritmos de teste e otimizações para memórias NAND Flash.

1.4.2 Objetivos Específicos

Para alcançar o objetivo geral deste projeto, os seguintes objetivos específicos foram necessários

a) Definir as características da memória NAND Flash e os principais tipos de falhas apresentadas e conhecidas na bibliografia.

b) Desenvolver modelo funcional da memória Nand Flash em Logisim^{TM}e VHDL, observando todos os requisitos comportamentais.

c) Simular e validar comportamento funcional da memória.

d) Desenvolver o circuito para inserção de falhas caracterizadas.

e) Simular e validar comportamento do modelo com cada falha inserida.

Para o atingimento dos objetivo listados, iniciou-se pela revisão bibliográfica dos tópicos relacionados a memória NAND Flash, descrevendo suas características de funcionamento e suas falhas, presentes no próximo capítulo 2 de Revisão Bibliográfica.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo são apresentadas a estrutura e características da memória NAND Flash, as etapas de teste e a importância do teste para a qualidade e custo do produto, os tipos de falhas mais comuns entre memórias NAND Flash, as ferramentas utilizadas no desenvolvimento e por fim os trabalhos relacionados a esta dissertação.

Segundo Bez et al. (2003), o grande crescimento na demanda de Memórias Flash nas últimas décadas é devido ao aumento significativo da produção de smartphones e outros dispositivos móveis, como tablets, câmeras digitais, mp3 players, smartphones, entre outros. Na busca por memórias do tipo não voláteis, as memórias Flash, com suas características flexíveis em termos de densidade e um baixo custo de produção, atendem os requisitos do mercado, que exigem um alto rendimento no armazenamento de aplicações, alta densidade e acesso rápido à memória para execução de códigos.

2.1 Memória Flash

Segundo Aritome (2000), os principais modelos comerciais de memórias Flash encontradas são, NOR Flash com versatilidade no endereçamento de dados e NAND Flash optimizada para o mercado de armazenamento de dados.

A memória do tipo NOR Flash se tornou a primeira memória a ser utilizada em telefones celulares, pelo fato de atender aos requisitos de execução rápida de códigos. Na medida em que as funcionalidades dos telefones celulares aumentaram, a demanda por memórias cada vez maiores em densidade e menores em tamanho também seguiu crescendo. *Multi level* NOR Flash foi introduzida com o armazenamento de 2 bits por célula, reduzindo o custo por bit e aumentando a densidade, no entanto, ainda possuía uma grande limitação na densidade máxima, fato responsável pelo surgimento da Memória NAND Flash. (ARITOME, 2000).

A memória NAND Flash foi desenvolvida com o objetivo de substituir os HDs (*hard drives*), devido à sua característica de armazenamento em blocos, similar aos dispositivos magnéticos. A NAND Flash possui uma arquitetura otimizada, baixo custo e tamanho do *die* reduzido, tendo as suas primeiras aplicações no armazenamento de fotos e vídeos. Com o aumento contínuo na demanda das aplicações para dispositivos móveis, a NAND Flash tornou-se a memória não volátil mais utilizada em telefones celulares. (RICHTER, 2014).

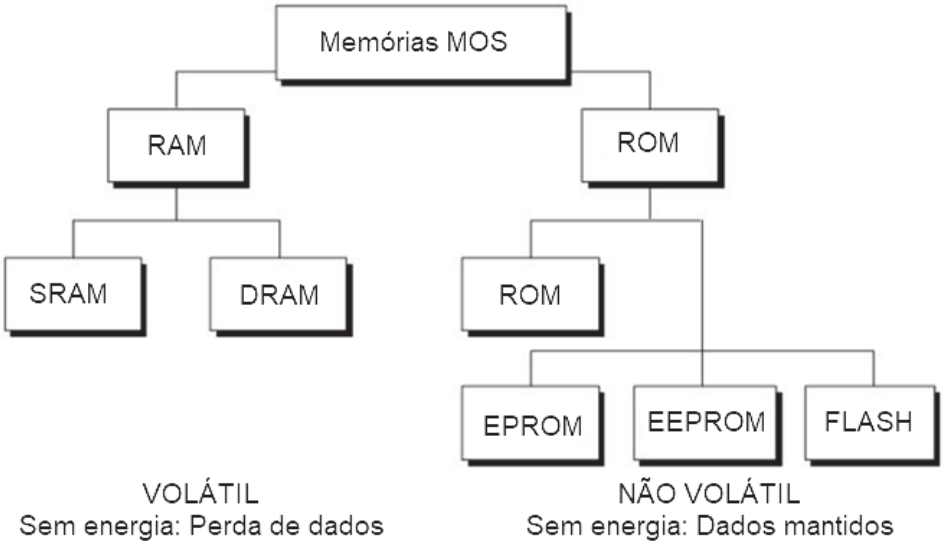
Os desafios desta tecnologia são gerados quando produtores de memória Flash aumentam a densidade da memória e diminuem o tamanho do *die* a cada ano, fazendo com que as quantidades de elétrons armazenados dentro das células diminuam cada vez mais, dificultando o processamento destes elétrons nos níveis de tensão que diferenciam se a célula possui nível

lógico ’0’ ou ’1’, diminuindo a confiabilidade das diversas funções da memória, desafio presente em memórias não voláteis. (RICHTER, 2014).

2.1.1 Memória Volátil e não volátil

Em termos de tecnologia, as memórias são divididas em dois diferentes segmentos, ambas baseadas na tecnologia MOS, em que Memórias MOS são divididas em RAM e ROM, sendo que dentro das memórias do tipo ROM encontra-se o tipo de memória FLASH não volátil e do outro lado memórias do tipo SRAM e DRAM com característica volátil, conforme mostra a Figura 1. (BEZ et al., 2003).

Figura 1 – Memória Volátil e Não Volátil.



Fonte: Adaptado de BEZ et al. (2003, p.2).

Memórias voláteis possuem como característica a perda dos dados contidos nela quando a energia fornecida é cessada, tendo como exemplos SRAM e DRAM, possuindo como característica principal a rapidez na leitura e programação. (BEZ et al., 2003).

De acordo com Bez et al. (2003), memórias não voláteis possuem a capacidade de armazenar os dados contidos na memória mesmo quando não existe energia de alimentação, tendo como exemplos FLASH, *Erasable Programmable Read-Only Memory* (EPROM) e EEPROM, que são capazes de ler e escrever numa velocidade bem considerável e ainda realizar o armazenamento dos dados em grande quantidade.

Devido às características das memórias não voláteis, foi possível o desenvolvimento de aplicações na área automobilística, da comunicação e aplicações para o consumo em geral, em que era necessário o armazenamento de informações. Os tipos de famílias de memórias não voláteis podem ser comparadas pela flexibilidade e custo, sendo a palavra flexibilidade a

possibilidade de programar e apagar muitas vezes nos diversos tipos de divisões na memória (blocos, páginas, células). (RICHTER, 2014).

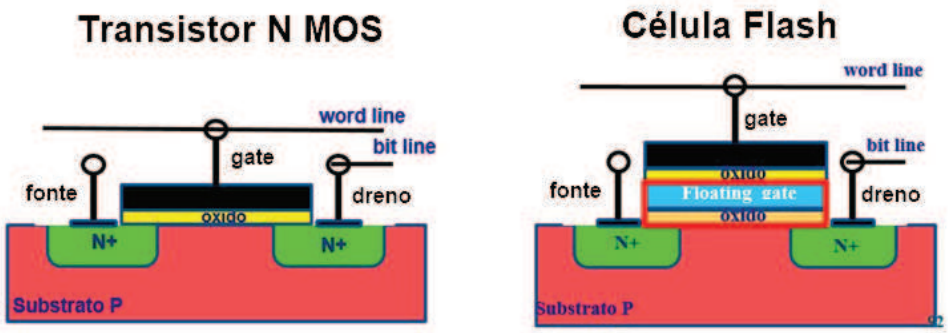
Segundo Richter (2014), a complexidade das aplicações aumentou com o passar do tempo e juntamente com o decréscimo das dimensões físicas das memórias, dando início ao desenvolvimento da tecnologia *floating gate* (FG), presente nas memórias Flash do tipo NOR e NAND e detalhado na próxima seção 2.1.2.

2.1.2 Estrutura da célula (*Floating Gate*)

Uma célula Flash básica, consiste em um transistor N MOS no qual é adicionado um *floating gate* (FG) (Figura 2), sendo um *gate* totalmente envolto e isolado por duas camadas de óxido, dando a possibilidade ao FG de agir como um armazenador de elétrons na célula, no qual qualquer carga injetada no FG tende a manter-se armazenada. (BEZ et al., 2003).

O funcionamento da célula com FG é simplificado devido a possibilidade de determinar o potencial armazenado dentro dela. Uma diferença de potencial entre o *source* e *drain* cria um campo elétrico entre os dois terminais, por sua vez, este campo elétrico faz com que o material (Substrato P) se torne condutivo, possibilitando a passagem de elétrons do *source* para o *drain*. O campo elétrico gerado pela alta tensão no terminal *gate* é utilizado para atrair os elétrons através do óxido, ficando armazenados no FG. (BEZ et al., 2003).

Figura 2 – Arquitetura da Célula Flash.



Fonte: Adaptado de Bez et al. (2003) p.491

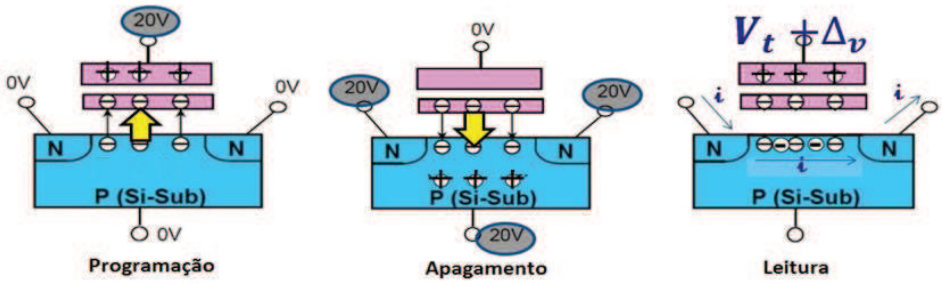
Através do FG é possível caracterizar os três tipos de operações básicas realizadas no transistor (Figura 3), são elas: Programação, Leitura e Apagamento.

A operação de Programação ou Escrita é controlada pela tensão aplicada no *gate*. Quando a tensão atinge um determinado valor, como por exemplo 20V, os elétrons contidos no substrato P seguem a atração gerada pelo *gate*. O tempo de programação depende da potência aplicada no *gate* e a espessura da camada do óxido inferior, uma célula que ocorreu a operação de programação recebe o valor lógico '0'. (RICHTER, 2014).

A operação de Apagamento é reversa da operação de programação. Aplica-se tensão positiva no *source* ou no *drain*, formando um túnel de corrente no substrato, assim fazendo com que os elétrons armazenados no FG sejam atraídos para fora. Uma célula que ocorreu a operação de apagamento recebe o valor lógico '1'. (RICHTER, 2014).

A operação de Leitura é realizada pela medição da tensão contida no FG do transistor. A forma mais simples deste procedimento é realizando a medição da corrente que passa pela célula, dada uma determinada tensão no terminal *gate*. (RICHTER, 2014).

Figura 3 – Operações Básicas da Memória Flash.



Fonte: Adaptado de Bez et al. (2003) p.491

Através da organização das células de armazenamento, que diferencia-se as memórias NAND e NOR Flash, apresentando vantagens e desvantagens para determinadas aplicações que são detalhadas na seção 2.1.3 de comparação entre elas.

2.1.3 Comparação da tecnologia NAND e NOR

Segundo Aritome (2000) as tecnologias NOR e NAND Flash dominam o mercado de dispositivos de memórias não voláteis, a qual anteriormente era dominada pelo tipo EPROM e Eletricaly EPROM. A memória NOR Flash foi introduzia pela Intel em 1988 revolucionando as memórias não voláteis, no ano seguinte em 1989, a Toshiba introduziu a arquitetura NAND Flash, pela necessidade de um baixo custo por bit, alto desempenho e tamanho menor por célula. (ARITOME, 2000).

A memória NAND Flash possui arquitetura de alta densidade de células, leitura e programação rápida, além de ter o tamanho da célula em torno da metade do tamanho de uma célula NOR, características que fizeram com que sua popularidade aumentasse drasticamente. (ARITOME, 2000).

A memória do tipo NOR Flash é efetiva em armazenamento com baixa capacidade e entrega uma alta velocidade na leitura, tendo como característica a função *Execute In Place*, ou seja, ela possibilita as aplicações rodarem diretamente na memória, não necessitando de leituras na memória RAM. A Tabela 1 mostra um resumo das vantagens e desvantagens de cada arquitetura. (ARITOME, 2000).

Tabela 1 – Vantagens e Desvantagens da Memória NAND e NOR

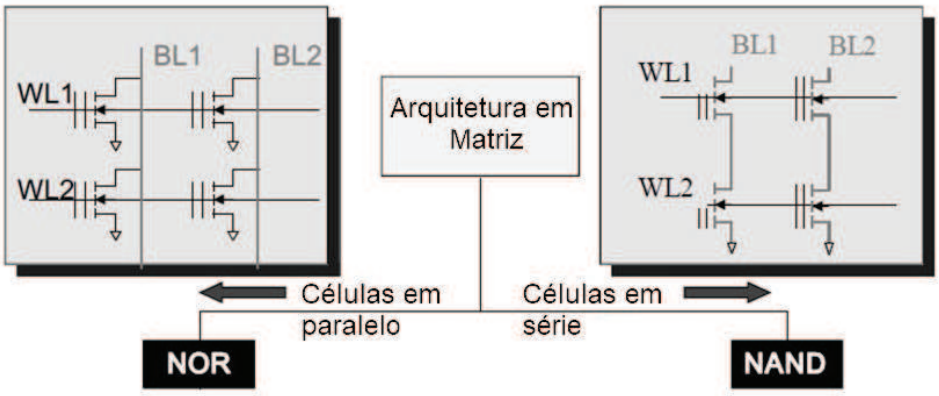
Tipo	Vantagens	Desvantagens	Aplicações
NAND Flash	Programação rápida	Acesso randômico lento	Voz, dados, gravação de vídeo
	Apagamento rápido	Dificuldade em programar Bytes	Qualquer sequência de dados longo
NOR Flash	Acesso randômico	Programação lenta	Substituir a EEPROM
	Possibilidade de gravação em Bytes	Apagamento lento	

Fonte: Adaptado de Micheloni, Crippa e Marelli (2010) p.4

Conforme Micheloni, Crippa e Marelli (2010) e apresentado pela Figura 4, NOR Flash apresenta uma matriz com acesso paralelo, o que possibilita o acesso a cada célula através de um contato independente, sendo este acesso direto às células a razão pela qual a NOR Flash possui vantagem no acesso randômico a memória.

A memória NAND Flash apresenta uma arquitetura em serial, em que cada bloco de células está acessível por dois transistores seletores, possibilitando assim, uma diminuição significativa no tamanho da célula em comparação com a NOR, sendo as duas arquiteturas apresentadas na Figura 4 .(MICHELONI; CRIPPA; MARELLI, 2010).

Figura 4 – Organização da Célula NAND Flash.



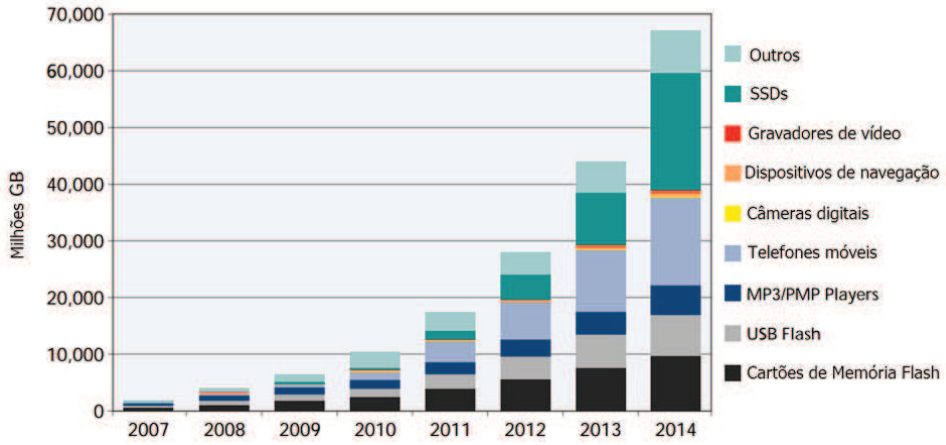
Fonte: Adaptado de Micheloni, Crippa e Marelli (2010, p. 2)

Nesta seção foram apresentadas as principais características que diferenciam as duas tecnologias de memórias Flash, explicando a razão pela qual a memória NAND Flash é a mais utilizada para armazenamento de dados e consequentemente a alta demanda de mercado para dispositivos móveis.

2.2 Mercado de memória NAND Flash

Sistemas e dispositivos móveis utilizaram NOR Flash como memória não volátil, devido às vantagens da alta densidade para a época, baixo custo e alto desempenho. Como o desenvolvimento da tecnologia continuou crescendo em termos de baixo consumo de energia, maior densidade em menor espaço, a NAND Flash vem se tornando líder em demanda para armazenamento em diversas aplicações, substituindo muitas vezes o disco rígido quando trata-se de armazenamento de 8GB ou 16GB como pode ser visto na Figura 5, em que apresenta a demanda em vários seguimentos tecnológicos aumentar entre 2007 e 2014.(Micron Technology, 2006).

Figura 5 – Demanda de memória NAND Flash no Mercado



Fonte: Adaptado de Micron Technology (2006, p. 1).

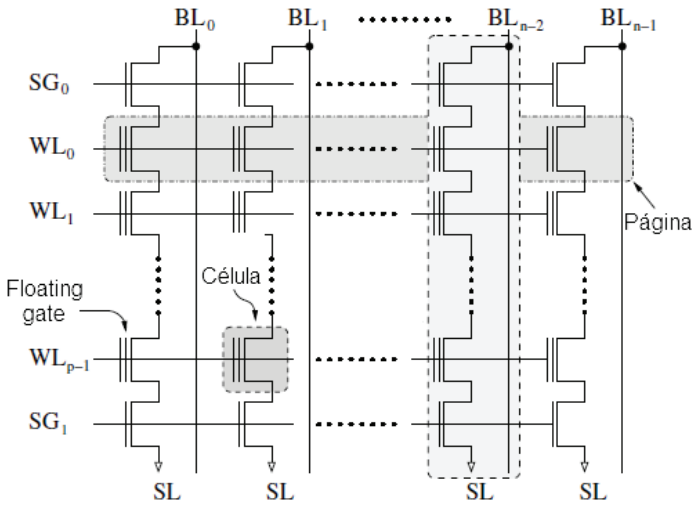
A Figura 5 apresenta o crescimento na demanda de densidade e memória NAND Flash entre 2007 e 2014, sendo o crescimento para telefones móveis e SSDs os grandes responsáveis pelo aumento, devido a sua estrutura e características explicadas na próxima seção 2.2.1.

2.2.1 Estrutura e Características da NAND Flash

De acordo com Micheloni, Crippa e Marelli (2010), para distinguir uma NOR Flash de uma NAND é necessária uma simples análise de como as células são organizadas. Ambas são em forma de matrizes, mas com acesso e layout diferentes. A estrutura da NAND Flash, como mostra a Figura 6, é construída em linhas de células (WL - *wordlines*) que formam as páginas, e um conjunto de *wordlines* ou páginas formam blocos de memória.

A matriz é composta por células ligadas em série e conectadas através do *bitline* (BL) e *source line* (SL), formando grupos de 32 ou 64 células que quando selecionadas através do gate *wordline* (WL), a sua informação é enviada para fora da matriz. Todos os terminais *gate* são conectados através dos *wordlines* (WLs) formando as páginas do bloco. (MICHELONI; CRIPPA; MARELLI, 2010).

Figura 6 – Arquitetura de um bloco da Memória NAND Flash



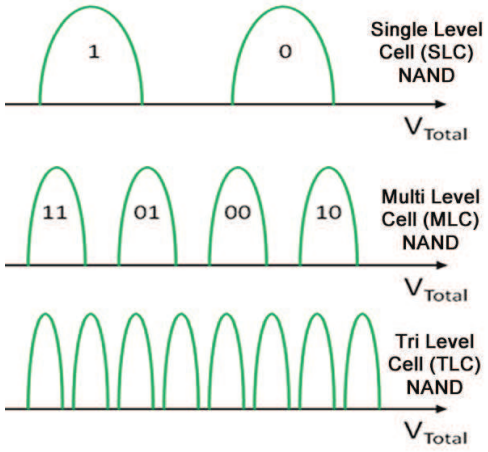
Fonte: Adaptado de Micheloni, Crippa e Marelli. (2010, p. 21).

As memórias NAND são divididas em níveis de armazenamento, como pode ser visto na Figura 7, o *Single Level Cell* (SLC) consiste no armazenamento de 1 bit de informação por célula (1 = Nível alto, 0 = Nível baixo), uma memória *Multi Level Cell* (MLC) armazena dois níveis de informação em cada célula e *Triple Level Cell* (TLC) armazena três níveis de informação. Comparando as estruturas SLC e MLS, percebe-se que nesta última as memórias terão uma velocidade de transferência menor juntamente com alto consumo de energia comparado com a SLC, pelo fato da necessidade de algoritmos extras para trabalhar com os diferentes níveis lógicos e a diminuição de confiabilidade gerada pela pequena diferença de tensão entre os níveis lógicos. Como mencionado anteriormente, SLC possui melhor confiabilidade sobre os dados obtidos devido ao fato de não obter vários níveis de reconhecimento de dados, tendo um espaçamento entre os níveis lógicos maior em relação a MLC e TLC. (RICHTER, 2014).

Segundo Micheloni, Crippa e Marelli (2010), a informação contida na memória NAND Flash é organizada como mostra a Figura 8, sendo dividida em páginas e blocos. Página é a menor porção de endereçamento para a função de leitura e programação, bloco é a menor unidade de apagamento possível. O número de páginas contidas em um bloco é normalmente múltiplo de 16, composta pela área principal de armazenamento e a área de reposição, usada para algoritmos ECC, reconhecimento de blocos inválidos e ponteiros do sistema.

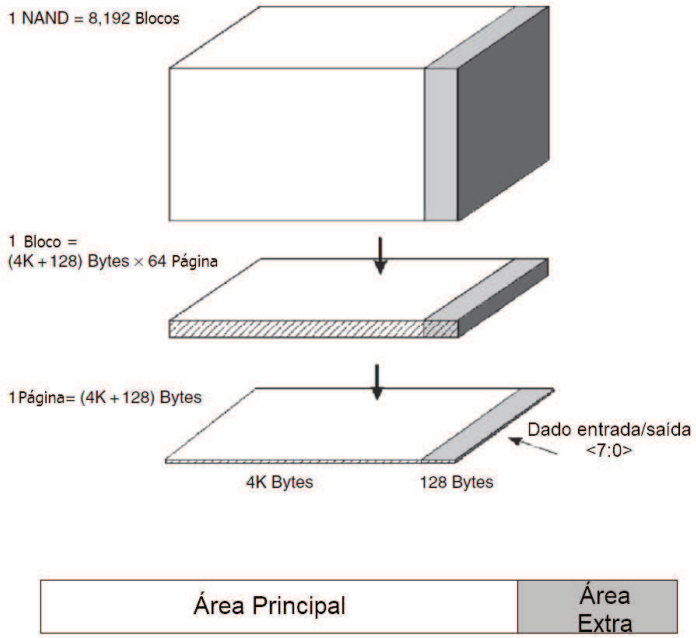
Operações enviadas para a memória (Programação, Leitura e Apagamento) devem ser compostas pelo endereço na memória onde desejamos operar, sendo composto pela linha de endereço que identifica o bloco e a página, além da coluna de endereço que identifica os *bytes* dentro da página. (MICHELONI; CRIPPA; MARELLI, 2010).

Figura 7 – Níveis de armazenamento, SLC, MLC, TLC.



Fonte: Adaptado de Micheloni, Crippa e Marelli. (2010, p. 6).

Figura 8 – Estrutura da memória NAND Flash



Fonte: Adaptado de Micheloni, Crippa e Marelli. (2010, p. 27).

2.2.2 Interface da memória

Os dispositivos de memória NAND Flash realizam a comunicação com dispositivos externos através dos pinos de comunicação presentes na Figura 9, que são descritos a seguir (MICHELONI; CRIPPA; MARELLI, 2010):

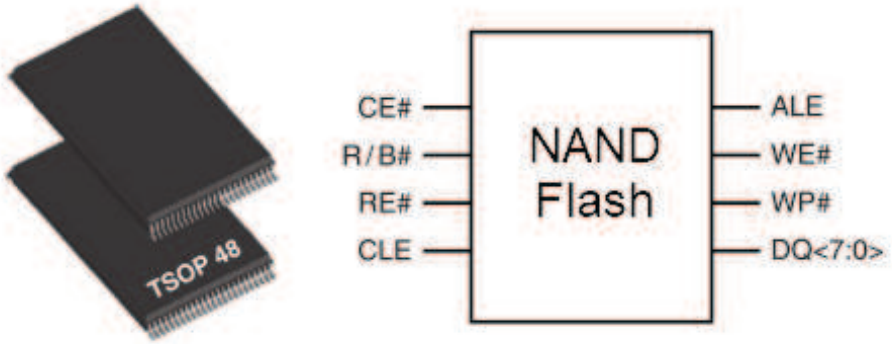
- **CE#:** é o sinal para habilitar a memória (*Chip Enable*). Sinal de entrada que é '1' quando o dispositivo está em espera, em caso contrário recebe sinal '0';
- **R/B#:** é o sinal de pronto/ocupado (*Ready/Busy*). Sinal de saída utilizado para indicar o status da memória, quando está em '0' o dispositivo está com uma operação em andamento;
- **RE#:** é o sinal para habilitar a operação de leitura (*Read Enable*). Sinal de entrada utilizado para habilitar os dados para a saída;
- **CLE#:** é o sinal de chaveamento dos comandos (*Command Latch Enable*). Sinal de entrada utilizado pelo *host* para indicar que o barramento de dados é utilizado para inserir o comando na memória;
- **ALE#:** é o sinal de chaveamento dos endereços (*Address Latch Enable*). Sinal de entrada utilizado pelo *host* para indicar que o barramento de dados é utilizado para inserção do endereço da memória;
- **WE#:** é o sinal de habilitação da escrita (*Write Enable*). Sinal de entrada que controla o chaveamento da entrada dos dados para escrita. Dados, comando e endereço são chaveados na borda de subida do sinal WE#;
- **WP#:** é o sinal para proteger a escrita (*Write Protect*). Sinal de entrada utilizado para desabilitar as operações de programação e apagamento;
- **DQ 7:0#:** São os 8 pinos de entrada e saída (*I/Os)* do barramento de dados, utilizado para a programação e leitura dos dados, comandos e endereços.

A interface da memória NAND Flash é conectada em muitos casos diretamente com um controlador de memória, sendo este responsável pela complexidade das operações da memória e outras funções detalhadas na seção 2.2.3 do controlador NAND Flash.

2.2.3 Controlador NAND Flash

Conforme Harari (2012), memórias NAND Flash são mais difíceis de trabalhar em comparação com a NOR, devido a sua complexidade no acesso da interface e nos diferentes tipos de interfaces existentes, como exemplo temos *host side interface* (Comunicação do controlador com o host) e *flash device interface* (Comunicação do controlador com a memória). Outra dificuldade apresentada é a existência de leituras de dados incorretas, esta falta de confiança na

Figura 9 – Interface da memória NAND Flash



Fonte: Adaptado de Micheloni, Crippa e Marelli. (2010, p. 27).

leitura da NAND Flash obriga a existência de algoritmos de prevenção de falha ou detecção de falhas implementados dentro do controlador.

Devido à enorme complexidade encontrada nas memórias NAND Flash, fabricantes implementam no componente controlador de memória algoritmos que realizam as operações básicas, com a maior confiabilidade e maior rapidez possível, juntamente com outros algoritmos que realizam o mapeamento dos endereços físicos para lógicos. Outras arquiteturas que não contenham o controlador realizam a complexidade no *host*, fazendo com que as tarefas gerais do *host* concorram com as tarefas proporcionadas pela memória, acarretando em um sistema mais lento de um modo geral. Portanto, a implementação e aplicação do controlador proporciona diversas vantagens em comparação à outras arquiteturas. (HARARI, 2012).

A alta complexidade da memória executada pelo controlador da memória, faz com que o teste para memórias NAND Flash também se torne mais complexo e necessário para a qualidade e o custo do produto, detalhes que são explicados na próxima seção de teste para memórias NAND Flash (2.3).

2.3 Introdução ao teste de memória

Para o melhor entendimento do teste em memórias NAND Flash, é necessário primeiramente a definição dos 3 termos básicos utilizados na literatura, os quais são: o defeito, a falha e o erro.

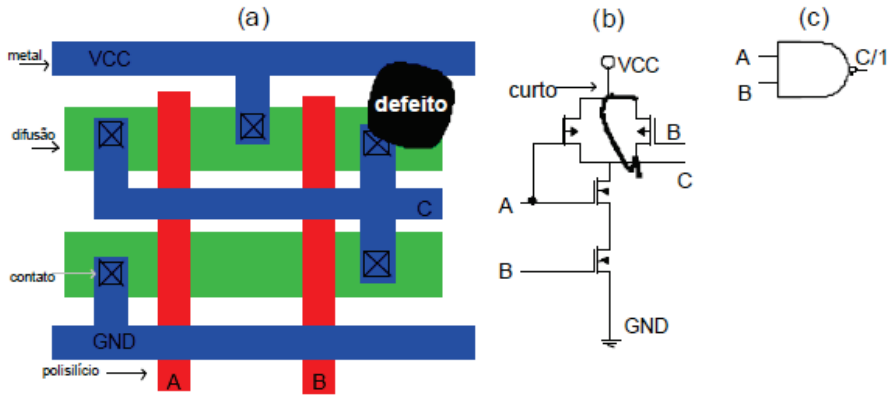
2.3.1 Defeito, Falha e Erro

Um defeito está relacionado a uma imperfeição física presente no circuito. A Figura 10(a) ilustra um defeito de fabricação, sendo uma área de metal não desejada que provoca um curto entre duas áreas de metal, modificando o comportamento do circuito (BUSHNELL; AGRAWAL, 2013).

Uma falha é uma abstração de um defeito. O defeito ilustrado em Figura 10(a) pode ser modelado como um curto em nível de transistor (Figura 10(b)), ou um SA (*stuck-at*) 1 no nível de portas lógicas como pode ser observado na Figura 10(c).(BUSHNELL; AGRAWAL, 2013).

Um erro é uma manifestação de um defeito, isto é, a sua propagação para as saídas primárias do sistema através da geração de uma resposta incorreta. Por exemplo, na Figura 10(c) existe um erro quando as portas de entrada A e B forem iguais ao nível lógico 1, pois o valor da saída é 1 quando deveria ser 0. Por outro lado, quando A e B forem iguais a 0, não há erro uma vez que o valor esperado é igual ao valor gerado.(BUSHNELL; AGRAWAL, 2013).

Figura 10 – Definição Defeito e Falha



Fonte: Adaptado de Amory (2003). pag.8

Ao definir os 3 termos básicos de defeito, falha e erro, pode-se caracterizar as etapas de teste e a sua devida importância para o mercado de semicondutores, conforme explicados na seção de teste em memórias NAND Flash, apresentada a seguir.

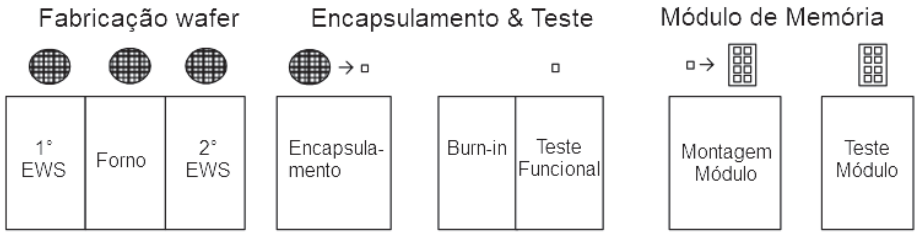
2.3.2 Teste em memória NAND Flash

Segundo Micheloni, Crippa e Marelli (2010), a cada novo nó tecnológico o custo e os problemas com o *yield* em memórias NAND Flash aumentam, fazendo com que o teste se torne uma etapa chave para diminuir os problemas com o *yield* e manter um custo competitivo no mercado.

O conceito do fluxo de teste em memórias NAND Flash tem o objetivo de obter a melhor cobertura de falhas para evita-las nas aplicações finais da memória, aumentando a produtividade, reduzindo o tempo de teste e o custo. O fluxo de teste é dividido como mostra a Figura 11, em *Front End* (Fabricação do wafer) *Back End* (Encapsulamento & Teste) e alguns componentes necessitam de teste em módulo. (MICHELONI; CRIPPA; MARELLI, 2010).

O teste realizado no *Front End* é dividido em duas etapas (*Electrical Wafer Stress*, EWS), com uma etapa de aquecimento em forno presente entre elas para o estresse de retenção de dados. Esta etapa é usada na detecção de falhas de manufatura, testando a funcionalidade das

Figura 11 – Fluxo de teste básico de NAND Flash



Fonte: Adaptado de Micheloni, Crippa e Marelli. (2010, p. 429).

células, aplicando testes de stress e finalizando com escrita na memória antes de inserir na etapa de aquecimento em forno. Em caso de falhas na matriz de dados, as falhas de curtos entre colunas são reparadas através de colunas redundantes e no caso de curtos entre *wordlines*, o bloco será tratado como *Bad Block*, ou seja, o bloco é identificado falho para que não seja utilizado posteriormente. (MICHELONI; CRIPPA; MARELLI, 2010).

Após a primeira etapa de teste, é realizado a fase de aquecimento dos *wafers* (*Bake*), sendo utilizada para forçar falhas de retenção na memória, com energia de ativação acima de 0.6eV e temperaturas entre 150ºC e 250ºC. (MICHELONI; CRIPPA; MARELLI, 2010).

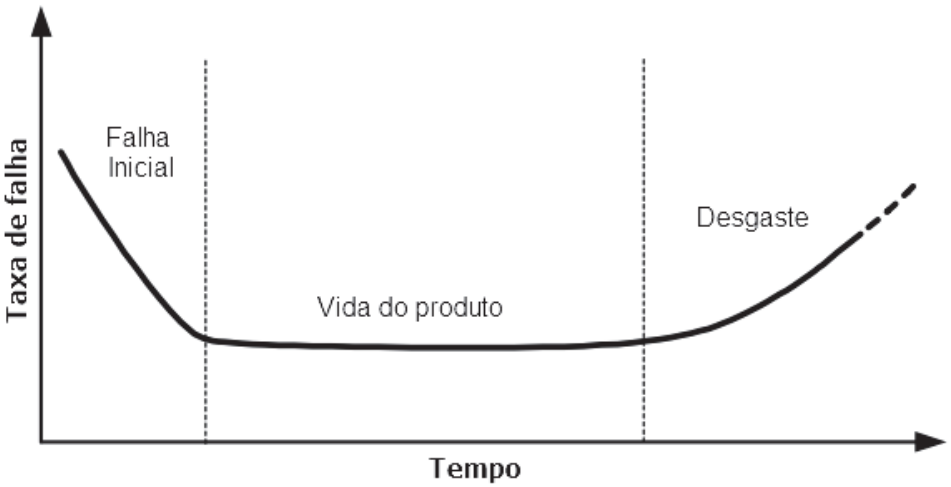
Em seguida, realiza-se a segunda etapa de teste (EWS2), que tem como principal objetivo destacar as células com falhas de retenção após o aquecimento, realizando a leitura dos dados escritos na primeira etapa de teste e também alguns testes de modo usuário (MICHELONI; CRIPPA; MARELLI, 2010).

Realizado o encapsulamento do *die*, então inicia-se a etapa de *Burn-In* para a introdução e detecção de defeitos de mortalidade infantil e consequentemente a melhora na taxa de falhas em campo, conforme pode-se observar na Figura 12, conhecida como curva da banheira, a taxa de falha em relação ao tempo decresce exponencialmente na primeira etapa, sendo caracterizada pela mortalidade infantil dos componentes.

Conforme Micheloni, Crippa e Marelli (2010), uma boa estratégia de teste de *Burn-in* para memórias NAND Flash é fundamental, pois deve-se evitar o desgaste excessivo das células da memória, portanto um controle de tensão, temperatura e um controle do número de leituras e escritas realizadas durante o teste é vital para o desgaste de cada célula, devido a característica de leituras e escritas limitadas presente em memórias NAND Flash.

O teste funcional presente na etapa de encapsulamento da Figura 11, segundo Micheloni, Crippa e Marelli (2010), é realizado através de algoritmos de testes capazes de detectar um alto grau de falhas funcionais presentes na memória, algoritmos normalmente desenvolvidos pelos próprios fabricantes da memória, pois são estes que conhecem o comportamento interno do seu dispositivo, facilitando o desenvolvimento de testes específicos e otimizados.

Figura 12 – Curva da Banheira



Fonte: Adaptado de Micheloni, Crippa e Marelli. (2010, p. 429).

O mercado de memórias NAND Flash cresceu mundialmente e ao mesmo tempo houve crescimento em termos de densidade das memórias, fazendo com que o teste receba um alto custo devido ao aumento da densidade, enquanto que o preço de venda das memórias decresceram, forçando os fabricantes a introduzirem o conceito de *Design for Testability* (DFT) nos projetos a serem desenvolvidos. (MICHELONI; CRIPPA; MARELLI, 2010).

2.3.3 Otimização do Teste

O termo *Desing for Testability* (DFT) refere-se a como o circuito é implementado ou modificado para que o teste seja simplificado e otimizado, chegando aos conceitos de Controlabilidade e Observabilidade. Estes por sua vez, são implementados na etapa inicial de design do circuito, ou seja, a Controlabilidade é a habilidade de aplicar no circuito padrões de entrada com o intuito de colocar um valor lógico em uma posição desejada do circuito. Após a inserção de um valor lógico conhecido para excitação de uma falha, a habilidade de observar a resposta da falha em um nó interno do circuito é conhecida como observabilidade. (LALA, 2009).

Os principais objetivos da implementação do DFT em memórias NAND Flash são o aumento da produtividade e competitividade, aumento do paralelismo no teste, aumento do *yield* e maximização da cobertura de teste para evitar falhas em campo, realizando operações como avaliação da funcionalidade da matriz da memória, avaliação das funcionalidades do produto final, otimização dos padrões de estresse, identificação e investigação de falhas. (LALA, 2009).

Devido a tarefa de testar todas as funcionalidades de um chip, por ser muito complexa e necessitar de uma grande quantidade de tempo, o conceito de *Built-in Self-test* (BIST) também é implementado com o intuito de aumentar a controlabilidade e observabilidade. (LALA, 2009).

No teste convencional, a geração de padrões de teste é realizada de forma externa,

utilizando ferramentas de geração de teste. A comparação das respostas do circuito com as respostas esperadas é realizada utilizando máquinas com alto custo chamadas de *automatic test equipment* (ATE). Devido a esta complexidade apresentada no teste convencional, o conceito de BIST é bastante aceito nos desenvolvimentos de circuitos, em que o modelo de teste é implementado dentro do próprio circuito, ou por um circuito extra encapsulado junto ao circuito principal, eliminando ou diminuindo a utilização de máquinas ATE. (LALA, 2009).

Uma configuração básica de BIST pode ser observada na Figura 13, em que o gerador de padrões de teste tem o objetivo de forçar o aparecimento de falhas no circuito sobre teste, sendo realizada análise da resposta de saída do circuito, verificando a consistência com o resultado esperado.(LALA, 2009).

Figura 13 – Configuração básica de um BIST

Fonte: Adaptado de Lala (2010, p. 71).

Em memórias NAND Flash, o teste é realizado utilizando as operações básicas descritas no tópico 2.1.2 de programação, leitura e apagamento, sendo estas operações complexas controladas em geral pelo dispositivo Controlador, citado no tópico 2.2.3, sendo uma solução que possibilita a flexibilidade ao teste, ou seja, o programa de teste pode controlar todos os sinais do circuito de forma flexível, sendo rotinas de BIST implementadas dentro do controlador (MICHELONI; CRIPPA; MARELLI, 2010).

Para o desenvolvimento de um algoritmo de teste eficiente para um determinado dispositivo, é necessário o conhecimento prévio dos possíveis tipos de falhas e suas características, detalhes estes descrito na seção 2.4 de falhas em memórias NAND Flash.

2.4 Falhas em memórias NAND Flash

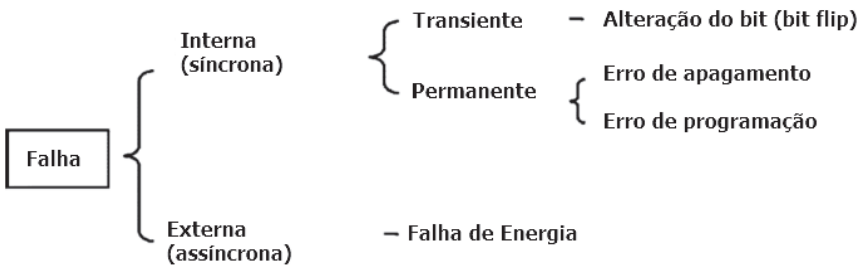
Memórias não voláteis, diferentes de memórias RAM, armazenam os dados através de operações que consomem um alto tempo para serem realizadas, quando comparadas com as operações das memórias RAM, tendo como principal característica a implementação de um circuito de alta tensão para realizar as operações de apagamento e escrita, fazendo com que, devido a este circuito de alta tensão presente na arquitetura da memória não volátil, falhas por interferência nas células apareçam devido ao alto estresse gerado. (HOU; LI, 2014).

O autor Yun et al. (2012), apresenta um resumo dos modelos de falhas presentes nas memórias NAND Flash e seus efeitos durante as operações da memória (Figura 14). Segundo ele, os tipos de falhas em memórias podem ser classificados em 2 categorias. A primeira categoria de

falhas chamada de falhas externas, não estão associadas às operações da memória, ou seja, são falhas assíncronas, como por exemplo falhas de energia que podem ocorrer a qualquer momento do funcionamento, transformando a memória para um estado indefinido, podendo ocorrer quando durante uma operação de apagamento em um bloco, ocorrer uma falha de energia na memória, o bloco que deveria conter os dados apagados (0xFF) pode conter valores diferentes do esperado, fazendo com que a próxima operação de escrita obtenha problemas de confiabilidade e dados incorretos.

A segunda categoria de falhas na memória apresentado na Figura 14, segundo (YUN et al., 2012) é chamada de falhas internas, sendo estas sensíveis às operações realizadas, portanto síncronas ao sistema. Uma operação de apagamento pode reportar uma falha quando um ou mais bits estão presos (*stuck at*) no valor 0, não podendo serem apagados (0xFF), ou uma operação de programação que em alguns bits estão presos no valor 1 e não podem serem programados. Estes 2 exemplos de falhas são do tipo permanentes, ou seja, o bloco ou a página com a falha deve ser substituída por blocos ou páginas reservas na memória, não afetando o funcionamento geral do sistema. O tipo de falha transiente acontece quando ocorre a alteração do valor (*bit flip*) da célula alvo ou de uma célula vizinha após a operação da memória ser realizada, outros autores denominam como falhas por interferência (MOHAMMAD; SALUJA, 2008).

Figura 14 – Classificação de falhas em memórias NAND Flash



Fonte: Adaptado de Yun et al. (2012) p.1.

Conforme Mohammad e Saluja (2008), as falhas permanentes consistem em falhas que são comuns para todos os tipos de memórias como DRAMs e SRAMS, sendo as falhas de *stuck at faust* (SAF), *stuck open fault* (SOF), *Transition Faults* (TF) e *adderss decoder fault* (AF).

A existência de falhas de SAF ou SOF significa que o valor lógico de uma célula de memória fique fixa em valor lógico ’0’ ou ’1’, desconsiderando qualquer função do circuito. Falhas de TF são células que alteram o seu estado lógico de ’0’ para ’1’ ou de ’1’ para ’0’. Falhas de AF são falhas presentes no circuito de endereços da memória, chamado *address decoder*. Este tipo de falha faz com que certos endereços da memória não sejam acessados ou através de um endereço, múltiplas células são acessadas.(GADGE; KARMORE, 2014).

Outro tipo de falha segundo Mohammad e Saluja (2008), é específica para memórias

não voláteis, não presentes em outros tipos de memórias voláteis, estas falhas são conhecidas como falhas por interferência (*disturb faults*). Segundo a classificação elas podem ser falhas permanentes, pois estão relacionadas a um defeito físico na camada de isolação entre células da memória ou a falha pode ser induzida pelo estresse elétrico durante as operações básicas da memória, sendo do tipo transiente e podendo se tornar permanente conforme a quantidade de estresse aplicado.

Falhas por interferência em memórias não voláteis estão ligadas diretamente com a estrutura da memória NAND Flash explicadas no tópico 2.2.1, em que a organização da matriz de memória é em *bitline* que acessa as páginas e *wordline* que acessa os blocos da memória NAND, sendo as falhas de programação descritas pelo autor Mohammad e Saluja (2008) e demonstradas na Figura 15, são elas:

- **Interferência do apagamento no *wordline* (WED):** ocorre quando em uma *wordline* compartilhada, uma célula sendo programada (C11), causa em uma outra célula (C10) o seu apagamento.

- **Interferência da programação no *wordline* (WPD):** ocorre quando em uma *wordline* compartilhada, uma célula sendo programada (C11) causa em uma outra célula (C12) a sua programação.

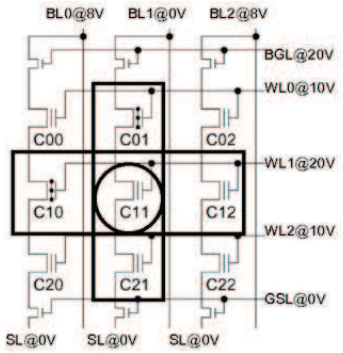
- **Interferência do apagamento no *bitline* (BED):** acontece quando em um *bitline* compartilhado, uma célula sendo programada (C11) causa em uma outra célula (C01) o seu apagamento.

- **Interferência da programação no *bitline* (BPD):** Ocorre quando em uma *bitline* compartilhada, uma célula sendo programada (C11) causa em uma outra célula (C21) a sua programação.

- **Interferências de programação na leitura (RPD):** É caracterizada pela operação de leitura conforme demonstra a Figura 16, em que a célula alvo é ativada pela tensão definida no *Control Gate* e as outras células do mesmo *bitline* são transformadas em um transistor passivo; operações de leituras realizadas em uma mesma célula é capaz de produzir transições nos dados da célula alvo ou em outras células no mesmo *bitline*, sendo no caso RPD o valor da célula será alterado para valor lógico '0' ou programada.

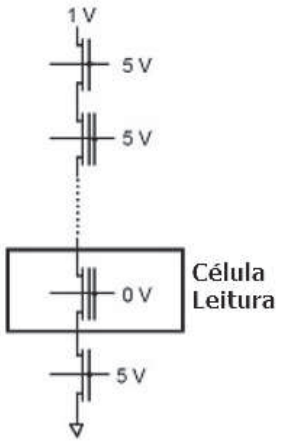
- **Interferências de apagamento na leitura (RED):** Ocorre do mesmo modo que a falha do tipo RPD, mas o valor da célula que possui a falha é alterada para o valor lógico '1' ou apagada.

Figura 15 – Interferência por apagamento em memória NAND Flash



Fonte: Adaptado de Carlo et al. (2010). pag.2

Figura 16 – Operação de Leitura em Memórias NAND Flash



Fonte: Adaptado de Carlo et al. (2010). pag.2

Os tipos de falhas de interferências por leitura são classificadas segundo Carlo et al. (2010) em 4 grupos: Interferência por leitura endereçada (Apagamento), em que uma célula programada é lida e seu conteúdo apagado; interferência por leitura endereçada (Programação), em que a célula apagada é lida e após é realizado programação indesejada; Interferência por leitura não endereçada (Apagamento) em que a célula é lida e outra célula com endereço diferente é apagada, e; Interferência por leitura não endereçada (Programação) em que a célula é lida e outra célula com endereço diferente com seu conteúdo apagado recebe uma programação indesejada.

Outro modelo de falhas por interferência é chamado de *Over Erased* (OED), ou seja, quando blocos da memórias são apagados, todos os elétrons presos no *floating gate* (FG) são simultaneamente removidos e durante esta operação que pode ocorrer a falha por interferência *Over Erased*. Esta falha ocorre quando algumas células são apagadas mais rapidamente que

outras, fazendo com que elétrons fiquem presos no FG de algumas células, fazendo com que estas células sejam mais difíceis de serem programadas.(CARLO et al., 2010).

Conforme Carlo et al. (2010), o oposto da falha de OED é o *Over Program Disturbance* (OPD) que ocorre quando uma programação de dados em uma página acontece, algumas células são programadas mais rápidas que outras, fazendo com que um excesso de carga negativa fique no FG, dificultando o seu apagamento.

Ao apresentarmos os tipos de falhas presentes em memórias NAND Flash, é essencial abordar as diferentes técnicas para injetar as falhas no modelo desenvolvido, sendo este um dos objetivos do trabalho final, injetando uma falha específica em um determinado endereço da memória.

2.4.1 Injeção de falhas

Um método comparativo entre diferentes técnicas e abordagens de injeção de falhas é descrito por Arlat e Crouzet (2003), apresentando uma série de propriedades das quais definem o nível de efetividade das técnicas e limitações, que são caracterizadas a seguir:

- **Alcançabilidade:** Define a propriedade de gerar falhas em locais específicos ou muitas vezes inacessíveis para certos tipos de técnicas;
- **Controlabilidade:** Define a propriedade relacionada a espaço e tempo. O tempo significa o número de ciclos do relógio que devem ser contados até o momento da aplicação da falha e o espaço significa a localização para injetar a falha;
- **Repetibilidade:** Define a propriedade que trata da repetição exata dos experimentos. Dessa forma, tal propriedade depende da controlabilidade sobre espaço e tempo;
- **Reprodutibilidade:** Define a propriedade de reproduzir os resultados obtidos em experimentos anteriores;
- **Não intrusividade:** Define a propriedade de minimização de qualquer impacto no comportamento do sistema;
- **Medição do tempo:** Define a propriedade relacionada à aquisição de informações de tempo, como contagem de ciclos de relógio, com associação aos eventos observados no experimento. Em alguns casos, um processador de referência é utilizado para geração de resultados;
- **Eficácia:** Define a propriedade de redução do número de experimentos não significantes, que não produzem efeito no sistema alvo, produzindo erros de medida. Normalmente a abordagem para redução desta ineficácia é o aumento da amplitude, tempo ou duração dos estímulos, ou interferência física aplicada ao hardware em teste.

Os estudos relacionados ao desenvolvimento de metodologias e ferramentas para injeção de falhas apresentam três tipos de injeção de falhas, são elas: injeção de falhas por hardware, injeção de falha por software e injeção de falha por simulação (HSUEH; TSAI; IYER, 1997).

Técnicas de injeção de falhas baseadas em hardware podem ser classificadas em duas categorias segundo (HSUEH; TSAI; IYER, 1997): injeção de falhas com contato, no qual o injetor requer contato físico com o hardware em teste, explorando modificações nas condições de corrente e tensão do sistema alvo e injeção de falhas sem contato, em equipamentos externos utilizados não exigem contato com o circuito ou sistema alvo em teste. Esta última utiliza-se de fenômenos físicos para geração de erros no sistema alvo como, por exemplo, aceleradores de partículas com utilização de íons pesados. Tais técnicas permitem o acesso a partes do sistema que outros métodos de injeção de falhas não conseguem acessar, impossibilitando a análise completa da solução de tolerância.

Embora a técnica de injeção de falhas baseada em hardware se aproxime de situações reais, a mesma necessita de equipamentos dedicados para execução de experimentos e, em consequência disso, o custo do projeto tende a aumentar. Em adição, aspectos como controlabilidade e reprodutibilidade representam problemas evidentes nos experimentos. O uso de tal técnica é recomendado para fases finais do processo de desenvolvimento, sendo utilizadas outras técnicas nas demais etapas do projeto (GEISSLER, 2014).

Na busca por redução de custo no desenvolvimento de projetos, técnicas de injeção de falhas baseadas em software surgem como uma solução atrativa em virtude da não necessidade de um hardware customizado. Esta técnica permite que em alto nível se possa acessar hardware e software de forma a reproduzir falhas enfrentadas em situações reais em ambientes com radiação, mas em contrapartida possui limitação na característica de Alcançabilidade, pois não consegue injetar falhas em locais do circuito que não são acessíveis por software (HSUEH; TSAI; IYER, 1997).

Outra técnica de injeção de falhas encontrada na literatura é baseada em simulação ou emulação de hardware (ANTONI; LEVEUGLE; FEHÉR, 2001), (CIVERA et al., 2001). Por meio de descrição de hardware (*Hardware Description Language)*, é possível construir a estrutura e descrever o comportamento de circuitos lógicos em alto nível. A linguagem VHDL é um exemplo bem difundido e utilizado em projetos de ASICs (*Application Specific Integrated Circuit*), sendo aplicada em análise de mecanismos de tolerância a falhas devido a sua alta controlabilidade e estrutura bem organizada. Esta descrição de hardware é realizada com a definição de componentes que se comunicam entre si por meio de sinais, podendo simbolizar barramentos e pinos, entre outros elementos. Assim, cada componente define suas características e funções no sistema. Outro fator relevante é a hierarquia entre os componentes, que é definida em tempo de programação.

A alta controlabilidade e facilidade de análise nos experimentos com uso de injeção de falhas em modelos de VHDL é fator motivador para utilização de simuladores no desen-

volvimento de técnicas de tolerância. Sem acesso a este modelo, a injeção de falhas com uso de simulação pode ficar restrita a pequenas etapas de projeto nas quais se tem controle sob a descrição de hardware, ficando para o restante a recomendação de outras técnicas de injeção de falhas (GEISSLER, 2014).

2.5 Logisim

O Logisim™é uma ferramenta educacional para a criação e a simulação digital de circuitos lógicos. Com uma interface simples e com ferramentas para simulação, possibilita a fácil aprendizagem dos conceitos básicos relacionados à circuitos lógicos, juntamente com a capacidade de construir projetos maiores a partir da reutilização de subcircuitos menores, realizando as conexões entre eles através de linhas que são desenhadas no próprio software (TENDELOO; VANGHELUWE, 2013).

O Logisim™pode ser usado para uma variedade de propósitos, incluindo: um módulo para o ensino de ciência da computação em geral, cursos de organização de computadores, e até mesmo semestres inteiros em cursos mais avançados de arquiteturas de computadores (BURCH, 2011).

A Figura 17 representa a janela principal do Logisim™. Pode-se dizer que o Logisim™está dividido em 3 partes: um painel explorador, no qual o usuário pode encontrar todas os componentes possíveis de serem utilizados; uma tabela de atributos, que permite ao usuário personalizar o componente que está selecionado; e por fim uma tela, onde o usuário pode incorporar todas os componentes e desenhar/criar o circuito digital que pretende. Por cima destas partes, encontramos a barra de menus e a barra de ferramentas (COUTINHO, 2014).

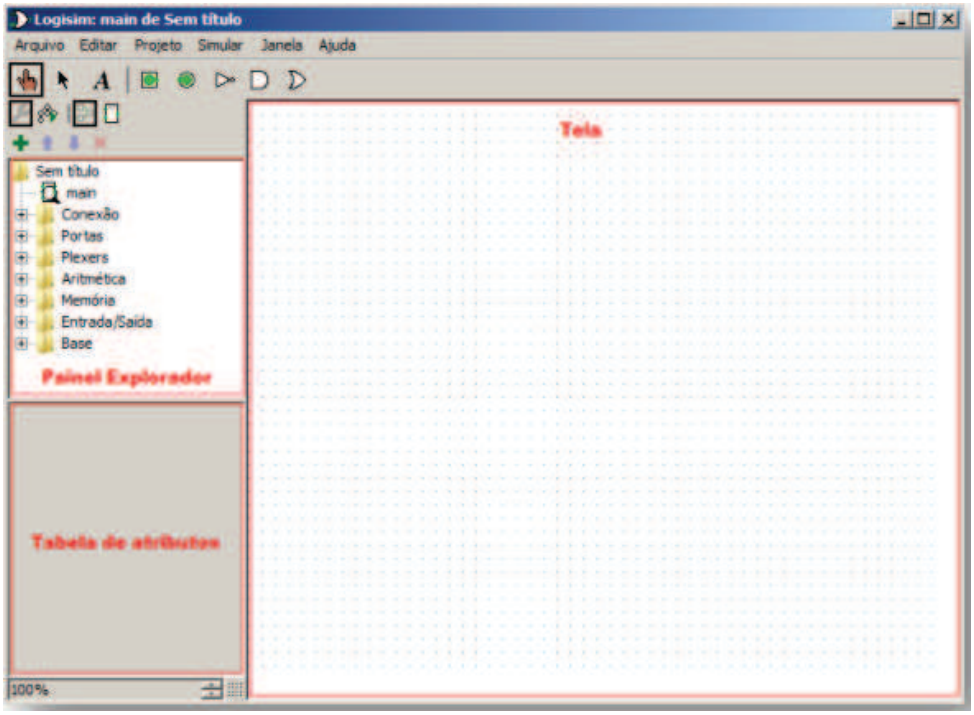
Através do desenvolvimento utilizando o Logisim™e devido as suas características explicadas anteriormente, foi possível obter um modelo funcional com inserção de falhas com maior controlabilidade e observabilidade do sistema, podendo verificar após o desenvolvimento cada funcionalidade dos circuitos internos da memória, de como é organizada a estrutura e o comportamento das falhas inseridas no sistema, características fundamentais para a escolha da utilização desta ferramenta.

2.6 VHDL

Com a complexidade dos circuitos integrados modernos, as linguagens de descrição de hardware (Hardware Description Language - HDL) se tornaram extremamente importantes como ferramentas no desenvolvimento e prototipagem de projetos eletrônicos. Elas têm como principal utilidade descrever o comportamento e a estrutura de um hardware. Dentre elas as mais conhecidas e utilizadas são Verilog e VHDL (MAITY; MAITY, 2007).

Em 1985, a empresa Xilinx introduziu o FPGA, sendo um circuito integrado específico

Figura 17 – Janela principal do Logisim^{TM}



Fonte: Adaptado de Carlo et al. (2010). pag.2

para construção de sistemas digitais e possuindo diversos blocos de entradas e saídas e blocos funcionais interconectados, cabendo ao usuário a programação e as especificações de cada bloco. Os FPGAs possuem em torno de 2,5 milhões de portas lógicas, portanto o desenvolvimento de projetos em forma de diagramas esquemáticos se tornou inviável, fazendo com que projetistas adotem os projetos baseados em HDL. A utilização desta linguagem realiza o aumento de produtividade na construção do projeto, abstraindo os níveis mais baixos de programação do projetista, evitando trabalhar no uso de portas lógicas ou nível lógico booleano (D'AMORE, 2012).

D'Amore (2012) afirma que VHDL (VHSIC Hardware Description Language) é a descrição de hardware mais utilizada como programação de FPGAs na automação de projetos de circuitos eletrônicos. Seu desenvolvimento foi motivado pela necessidade de um padrão para o intercâmbio de informações entre fornecedores de equipamentos do Departamento de Defesa dos Estados Unidos, substituindo manuais complexos.

A linguagem VHDL pode ser utilizada no modelo estrutural ou comportamental nas descrições de hardware, onde normalmente emprega-se uma mistura de modelos para formar um projeto complexo, possibilitando também o uso de diversos níveis de abstração no mesmo projeto (D'AMORE, 2012).

Através do capítulo 2 Referencial Bibliográfico, foi possível introduzir as características funcionais da memória NAND Flash, juntamente com as suas falhas, as técnicas de injeção das

mesmas e a importância do teste para a confiabilidade do produto final, formando uma base de conhecimento para os próximos capítulos da dissertação, em que trabalhos relacionados a este projeto estão descritos no capítulo 3 e o fluxograma de desenvolvimento do projeto detalhado no capítulo 4.3.

3 ESTADO DA ARTE

Neste capítulo são apresentados os trabalhos relacionados ao projeto proposto, apresentando as suas características e a maneira de como contribuíram para o desenvolvimento deste trabalho. Wu, Huang e Wu (1999) desenvolveu o mecanismo de simulação de falhas para memória chamado RAMSES (*Random Access Memory Simulator*), que consiste em um simulador de memória RAM juntamente com descrições de falhas que podem ser aplicadas ao simulador seguindo os padrões do simulador. Segundo o autor, a arquitetura do RAMSES é flexível e expansível devido ao fato de que, ao inserir um novo modo de falha, é necessário somente descrever o novo modo de falha no sistema.

Após o desenvolvimento do RAMSES, foi realizado o desenvolvimento do mecanismo de simulação de falhas para memórias flash chamado RAMSES-FT (CHENG et al., 2002), em que é considerado falhas de interferência entre células, característica presente em memórias Flash, ambos os simuladores proporcionam uma ferramenta para aplicação de algoritmos de teste e o diagnóstico de falhas para diversos modelos de memórias, vindo ao encontro do objetivo geral deste trabalho de proporcionar um modelo que possibilite o estudo e desenvolvimento de algoritmos de teste para memórias NAND Flash, mas que demonstre didaticamente o comportamento interno das falhas aplicadas através do software Logisim^{TM}e que possa ser adaptado e utilizado futuramente em diversos outros projetos que necessitem de uma memória NAND, através do desenvolvimento em VHDL.

Um resumo dos modelos de falhas para memórias NAND Flash foi apresentado por Yun et al. (2012), juntamente com a descrição detalhada dos efeitos das falhas durante as operações da memória, o objetivo do autor é apresentar um modelo de falhas que proporcione o desenvolvimento de ações corretivas e o desenvolvimento de softwares de gerenciamento da memória, localizados no controlador de memória NAND Flash visando a construção de algoritmos, sendo este modelo de falhas utilizado como referência deste trabalho (Seção 2.4).

Com o objetivo de desenvolver uma ferramenta para validação de mecanismos de detecção de falhas para auto teste (BIT – do inglês, *Built-in Test*), Xu e Xu (2012) propõe uma técnica de injeção de falhas baseada no emulador QEMU. A técnica baseia-se na injeção de falhas na memória RAM, a qual é emulada pela máquina, com modificações no software desta. Uma evolução deste trabalho é caracterizada por um injetor de falhas chamado BitVaSim, que cobre injeção de falhas em mais elementos de memória da máquina, foi proposto por Li, Xu e Wan (2013). O objetivo do trabalho foi gerar exceções no processador, através de alterações do código fonte da máquina, para verificar o comportamento do sistema operacional em execução.

Uma metodologia de injeção de falhas baseada em emulação de processadores é apresentado pelo autor Geissler (2014), tendo o principal objetivo de disponibilizar uma ferramenta

para o desenvolvimento de mecanismos de tolerância de falhas para processadores, para isto o uso de técnicas de injeção de falhas é válido, segundo o autor, uma abordagem normalmente empregadas para injeção de falhas é a utilização durante a emulação em FPGA, sendo alterações no código do programa ou um módulo de hardware que age como sabotadores no circuito alvo, mesma técnica que será abordada neste trabalho para a injeção de falhas na memória NAND Flash, sendo um módulo de injeção de falhas capaz de sabotar qualquer endereço da memória, sendo explicado em detalhes no tópico da metodologia.

Na literatura existem trabalhos que realizaram a emulação do comportamento de memórias não voláteis e técnicas para estudar, aplicar e simular falhas presentes em memórias. O trabalho de Prodromakis e Antonakopoulos (2015) apresenta uma plataforma desenvolvida em FPGA capaz de simular memórias não voláteis, tendo o foco em memórias NAND Flash MLC. O seu principal objetivo é desenvolver e avaliar qualquer algoritmo relacionado à memória, pois oferece uma emulação em tempo real das condições aplicadas pelo usuário do emulador, a arquitetura da memória desenvolvida segue os padrões estabelecidos pela ONFI (*Open Nand Flash Interface*), organização que padroniza a interface de comunicação da memória, mesma interface que será seguida por este trabalho.

O autor Ribeiro (2015), apresenta uma proposta que busca validar, funcionalmente, o conceito de ponteiro absoluto ao longo do mapeamento dos endereços em memórias voláteis síncronas, tendo como segundo objetivo do modelo proposto, promover uma interface para a criação de testes de caráter intrusivo, bem como o de propor a funcionalidade de testes sob demanda. Logo, o autor através de *scripts* identifica e trata a existência de erros nos processos de leitura e escrita em blocos de memória, validando o modelo proposto através desta metodologia de validação.

Recentemente Zhang et al. (2017) desenvolveu uma plataforma em FPGA para teste e investigação de falhas para memórias NAND Flash MLC, com foco em falhas na escrita dos bits mais significativos (MSB) e menos significativos(LSB), falhas na alocação de páginas e falhas de leitura que ocorrem em memórias MLC. Para coletar um grande número de dados para análise, o autor desenvolveu a plataforma de teste, que possibilita uma análise das características dos erros apresentadas pelas memórias, apresentando, por fim, os novos modos de falha e explicando as suas características.

Considerando os poucos trabalhos recentes relacionados a simuladores de falhas apresentados anteriormente, este projeto vem para contribuir para o teste de memórias através do principal objetivo de disponibilizar um modelo base para o estudo e desenvolvimento de algoritmos de teste para memórias NAND Flash (SLC). O projeto visa contribuir academicamente através do Logisim^{TM}, que apresenta visualmente o comportamento funcional da memória e as falhas implementadas e por fim disponibilizar um modelo funcional da memória com inserção de falhas em VHDL, que possibilita o estudo e desenvolvimento de algoritmos de testes, que futuramente poderá estar presente em projetos da universidade que necessitem de uma memória

NAND Flash validada.

Os trabalhos mencionados neste capítulo, contribuíram para o desenvolvimento da metodologia detalhada no próximo capítulo 4 de Materiais e Métodos, em que é definido o modelo funcional desenvolvido e o fluxo de desenvolvimento seguido no projeto.

4 MATERIAIS E MÉTODOS

Após os esclarecimentos iniciais acerca da importância do teste para circuitos de memória, juntamente com a importância de obter o maior controle sobre as funções da memória, apresenta-se as etapas de desenvolvimento do modelo proposto. Este capítulo irá apresentar e caracterizar as etapas de desenvolvimento do modelo funcional da memória NAND Flash com inserção de falhas caracterizadas. Para tanto algumas etapas descritas a seguir foram necessárias:

4.1 Pesquisa do tema

Esta etapa especifica a parte teórica para o qual propõe-se o modelo, sendo delimitado pela compreensão e conceitualização da arquitetura e comportamento da memória NAND Flash, bem como da importância do teste e os tipos de falhas presentes.

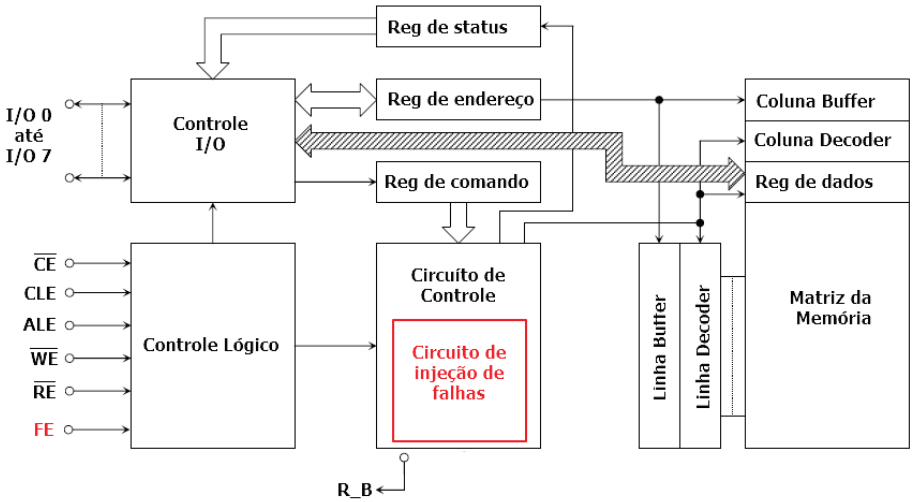
Através do capítulo da Revisão Bibliográfica escrito anteriormente pode-se obter uma visão geral sobre a introdução a memória NAND Flash e suas aplicabilidades, a organização interna da memória em páginas e blocos, a caracterização da etapa de teste da memória, os tipos de falhas presentes e técnicas de injeção de falhas, sendo este um objetivo do modelo de memória proposto, injeção de falhas no modelo para o fácil desenvolvimento de algoritmos de teste, capazes de detectar as falhas caracterizadas no tópico de falhas em memórias.

4.2 Definição do modelo funcional com inserção de falhas

Com base na arquitetura de uma memória NAND Flash comercial, presente no *datasheet* de Electronics (2010), foi possível definir a arquitetura geral do projeto, que leva primeiramente em consideração as delimitações do projeto especificadas no tópico 1.3, sendo estas: o modelamento de uma memória do tipo SLC (*Single Level Cell)*, o modelamento funcional da memória desconsiderando o comportamento elétrico para a realização de programação, leitura e apagamento, retirando da arquitetura circuitos de geração de alta tensão e amplificadores de sinais responsáveis por tais operações, utilizando também a interface detalhada na Seção 2.2.2 e por fim acrescentando ao projeto o circuito de inserção de falhas e o pino de saída FE (*Fail Enable*), chegando a arquitetura do modelo presente na Figura 18 e detalhado nos parágrafos a seguir.

De forma visual, a Figura 18 apresenta da esquerda para direita, a interface com entradas e saídas da memória, caracterizados no tópico anterior de interface de memórias NAND Flash (Seção 2.2.2), adicionando ao projeto, o sinal em vermelho de habilitação do circuito de injeção de falhas. (FE - *Fail Enable*), responsável pela ativação do circuito de inserção de falhas quando se deseja aplicar falhas na matriz da memória.

Figura 18 – Arquitetura do Modelo



Fonte: Adaptado de Electronics (2010)

O módulo de controle dos sinais de dados (Controle I/O) realiza o controle dos 8 pinos do barramento de dados que são utilizados para a inserção do endereço da memória, do comando enviado e dos dados que serão salvos na memória, encaminhando os dados para seus respectivos registradores.

O módulo de controle lógico é responsável pelo controle dos sinais de chaveamento dos endereços (ALE) e de comandos (CLE), juntamente com o controle da habilitação da escrita (WE) ou leitura (RE) e a habilitação do módulo de injeção de falha (FE), enviando o sinal ao circuito de controle da memória.

O circuito de controle da memória é responsável pelo recebimento do registrador que contém o comando enviado, o recebimento do sinal lógico que realizará o chaveamento do endereço e comando, por fim gerenciar os dados recebidos para o controle da memória enviando-os ao registrador de dados e aos decodificadores de endereço.

Dentro do circuito de controle da memória, foi desenvolvido um circuito específico para inserção de falhas, que é responsável pelo controle do tipo de falhas que será aplicada a memória, juntamente com a emissão dos sinais do endereço alvo da falha e o dado que será alterado, simulando o comportamento funcional da falha no modelo de memória implementado, utilizando portanto a técnica de injeção de falhas por simulação (ANTONI; LEVEUGLE; FEHÉR, 2001), sendo caracterizada por um módulo interno do circuito, capaz de injetar falhas na memória no endereço que se desejar.

O módulo da matriz da memória e as células de memória presentes dentro da matriz, foram desenvolvidos seguindo as características explicadas na Seção da Estrutura e Característica da memória NAND Flash (2.2.1), em uma organização das células em páginas e blocos, caracterizando as páginas como a menor porção de memória para leitura e escrita, e os blocos como a

menor porção de memória para apagamento. O circuito das células de memória foi desenvolvido de forma expansível, em que através da implementação utilizando o software Logisim^{TM}foi possível replicar os circuitos de células, páginas e blocos desenvolvidos e também através da implementação em VHDL foi possível alterar as variáveis responsáveis por configurar o tamanho da matriz da memória, sendo o número de células, o número de colunas por página, o número de páginas e a quantidade de blocos, conforme pode ser visto na Figura 19 e o desenvolvimento da matriz será detalhado na Seção 5.1.1.

Figura 19 – Definição da densidade da memória

```
1  ----------------------------------------------------------
2  -- Código Principal - Guilherme Lopes & Igor Franco
3  ----------------------------------------------------------
4  library IEEE;
5  use IEEE.STD_LOGIC_1164.ALL;
6  USE ieee.numeric_std.ALL;
7  use ieee.std_logic_unsigned.all;
8  use work.common_pkg.all;
9
10 entity Main is
11     generic(
12             num_cell : integer := 8;
13             num_col : integer := 3;
14             num_page : integer := 4;
15             num_block : integer := 4;
16             num_reads : integer := 2
17     );
```

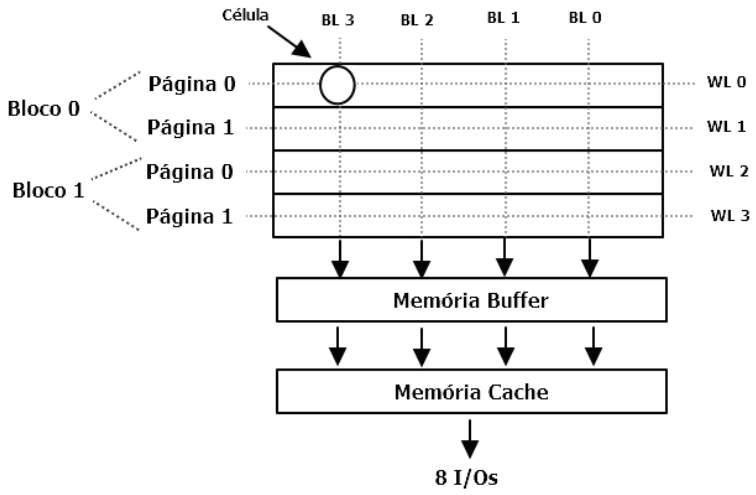
Fonte: Elaborado pelo Autor

Segundo a Seção de falhas em memórias NAND Flash (2.4), pode-se classificar os tipos de falhas em 2 tipos, o primeiro são falhas internas que ocorrem e aparecem através da execução de algumas operações da memória chamadas falhas transientes que alteram o valor da célula alvo e falhas permanentes compostas pelo tipo de *Stuck at*, o segundo são falhas externas, que se caracterizam por falhas de energia que ocorre no circuito da memória responsável pela geração de energia para realizar as operações da memória.

Como a arquitetura do modelo considera somente o comportamento funcional e não leva em consideração o comportamento elétrico do circuito, faz com que o desenvolvimento das falhas seja desenvolvido considerando apenas as falhas do tipo internas, ou seja, falhas externas que envolvem os circuitos de energia da memória são desconsideras.

Os tipos de falhas internas que foram abordadas por este trabalho e seus funcionamentos detalhadas na Seção do Circuito de Inserção de Falhas (5.2), envolvem apenas as falhas que afetam somente as células da memória, sendo falhas do tipo *stuck-at* 0 e 1, e falhas por interferência através do *bitline* que liga as células entre páginas e blocos, interferência através do *wordline* que liga as células presentes na mesma página e falhas de interferência pela repetição da operação de leitura na mesma posição da memória. Conforme pode ser visto na Figura 20, um exemplo reduzido da organização da matriz da memória em que são interligados por *bitline* e *wordline* e que no cruzamento destas duas ligações está presente uma célula da memória.

Figura 20 – Exemplo de organização da matriz da memória



Fonte: Elaborado pelo Autor

Após a definição das características do modelo funcional, com base no modelo apresentado pelo *datasheet* de Electronics (2010) e adaptado com um circuito de inserção de falhas, é necessário a definição do fluxo de desenvolvimento que será seguido, detalhado e apresentado pela próxima seção 4.3.

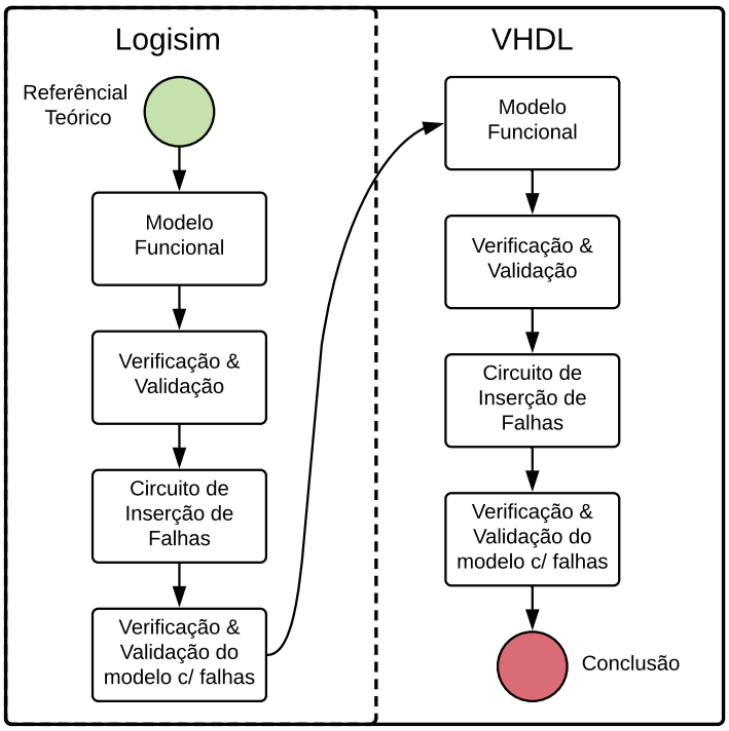
4.3 Fluxo de desenvolvimento

O fluxo de desenvolvimento do projeto é composto inicialmente pela pesquisa e revisão do referencial teórico da memória NAND Flash, seguido por 2 etapas principais de desenvolvimento, sendo a primeira o desenvolvimento do modelo de memória NAND Flash com inserção de falha através do software Logisim^{TM}, introduzido na seção do software Logisim^{TM}no Referencial Teórico (2.5) e a segunda etapa sendo o desenvolvimento em VHDL, conforme detalhado na Figura 21.

Dentro de cada etapa da Figura 21, é possível observar a implementação do modelo funcional da memória que será descrito na seção do modelo funcional (5.1) do capítulo de desenvolvimento (5), a verificação do funcionamento de cada função da memória, seguido da validação do modelo final, que serão descritos no capítulo de Validação e Resultados (6.1), após a validação do modelo é acrescentado ao projeto o circuito de inserção de falhas, capaz de injetar uma falha específica na posição desejada da memória, descrito na Seção do circuito de falhas (5.2) do capítulo de desenvolvimento e por fim a verificação e validação do circuito final com cada falha implementada, descrito no mesmo capítulo de Validação e Resultados.

A primeira etapa de desenvolvimento utilizando a ferramenta Logisim^{TM}foi realizada com o principal objetivo de poder caracterizar e apresentar visualmente o comportamento

Figura 21 – Fluxo de desenvolvimento



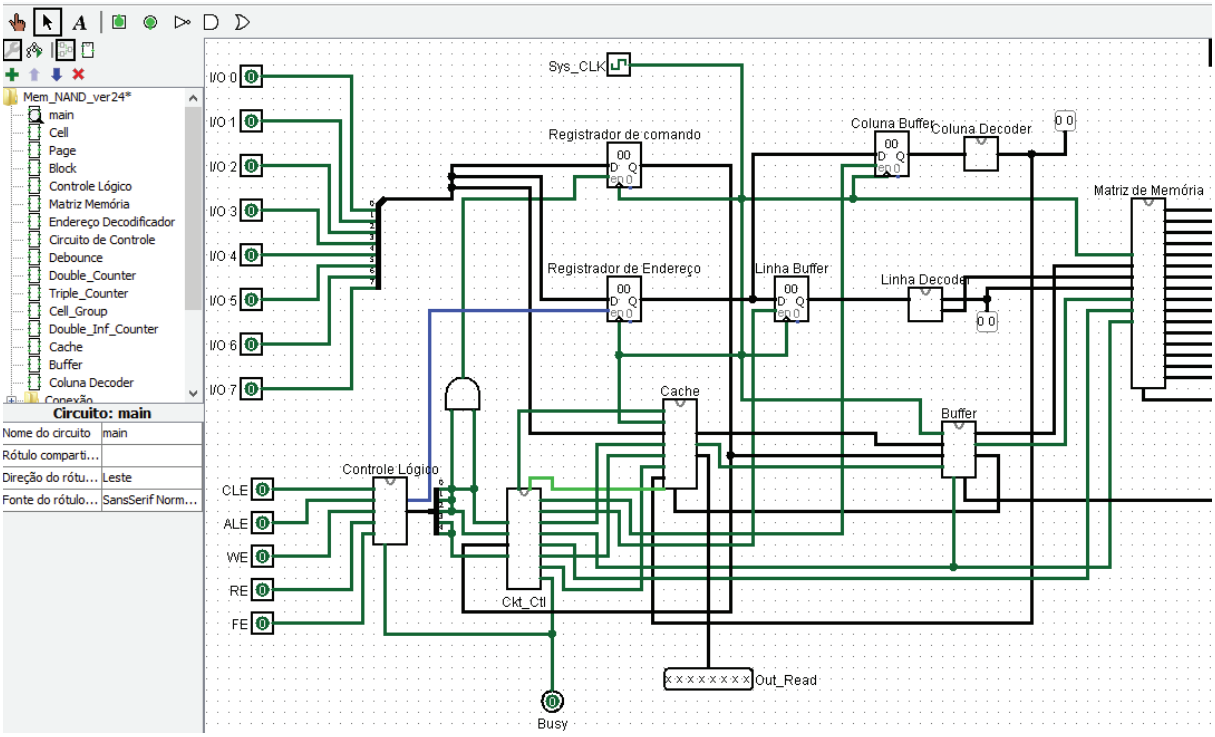
Fonte: Elaborado pelo Autor

funcional da memória, juntamente com a caracterização das falhas aplicadas ao projeto, pois o Logisim^{TM}por se tratar de uma ferramenta educacional conforme Tendeloo e Vangheluwe (2013), consegue demonstrar o comportamento interno de cada circuito desenvolvido junto com as interconexões entre eles, se tornando uma ferramenta bastante manual para a implementação, mas com um grande poder didático que poderá ser utilizado na disseminação do conhecimento em memórias NAND Flash dentro da universidade.

Pode-se observar na Figura 22 o circuito final do modelo da memória NAND Flash em Logisim^{TM}, em que ocorre o desenvolvimento funcional de cada circuito interno da memória e as interconexão entre eles com base na arquitetura do modelo proposto, mas a medida em que o projeto foi sendo desenvolvido, a ferramenta que possui a principal finalidade acadêmica se mostrou incapaz de suportar um circuito complexo, apresentando travamentos do software a medida em que o número de circuitos e interligações ia aumentando, apresentando também erros durante a inicialização do programa, em que fios ligados a matriz da memória inicializavam em um estado inválido, impossibilitando o uso da memória e fazendo com que necessitasse reinicializar o programa diversas vezes para que o erro não ocorra.

Devido aos problemas encontrados com a ferramenta descritos no parágrafo anterior, não foi possível desenvolver os modos de falhas por interferência, pois estes exigem uma comunicação entre células, páginas e blocos para que a falha se manifeste, fazendo com que um número maior de interligações fosse implementado e novos circuitos adicionados ao projeto,

Figura 22 – Circuito em Logisim^{TM}



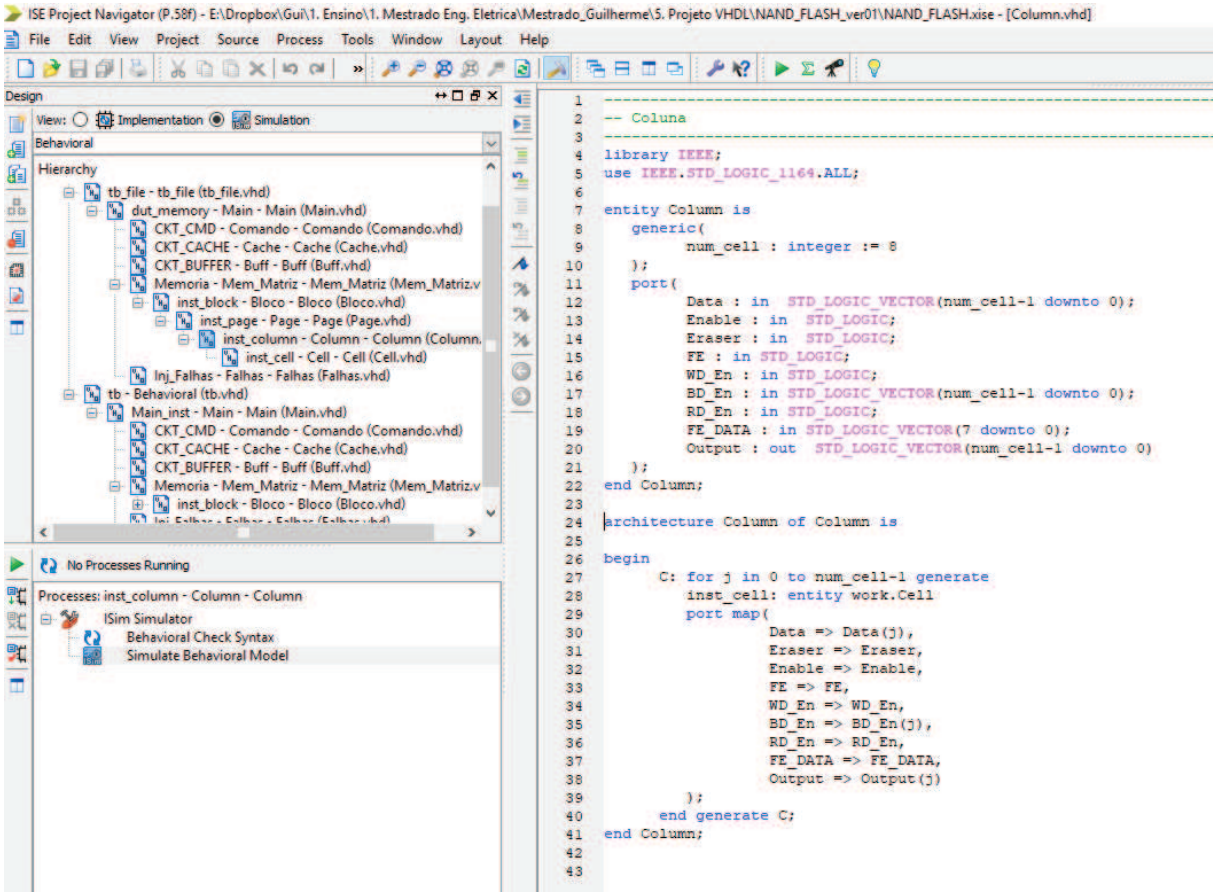
Fonte: Elaborado pelo Autor

portanto ao final do projeto em Logisim^{TM}foi possível implementar e validar o modelo funcional da memória, juntamente com o circuito de injeção de falha, implementado somente 2 tipos de falhas *Stuck-at* 1 e *Stuck-at* 0 e excluindo as falhas de interferência do projeto. Para que o projeto obtivesse uma relevância de aplicação na indústria e ao mesmo tempo fosse possível implementar todas as falhas de interferência propostas nos objetivos, iniciou-se o desenvolvimento da memória funcional com inserção de falhas em VHDL.

Através do desenvolvimento do projeto em VHDL em nível comportamental foi possível a implementação do modelo funcional da memória, adicionando ao projeto o circuito injetor de falha e a implementação das falhas de *Stuck-at* e interferência, utilizando o método modular, ou seja, o circuito funcional da célula por exemplo, é criado e utilizado 8 células para compor cada *Byte ou coluna* de uma página, e assim utilizar o circuito da página para compor um bloco e assim por diante, conforme pode ser visto na Figura 23 de exemplo de uma coluna em que é realizado um *loop* utilizando o circuito de uma célula.

As etapas de verificação e validação em Logisim^{TM}e VHDL, detalhadas no Capítulo 6 de Validação e Resultados, foram realizadas de forma distintas, a primeira etapa de verificação e validação foi realizada após a conclusão do modelo funcional da memória, tendo a certeza de que o circuito final obedece as funcionalidades desenvolvidas e após isto, foi adicionado ao projeto o circuito de inserção de falhas para que não houvesse qualquer interferência durante a validação do circuito de falhas ao modelo funcional da memória, ao final do desenvolvimento

Figura 23 – Byte de Memória em VHDL



Fonte: Elaborado pelo Autor

portanto realizou-se a verificação e validação do circuito de inserção de falha e separadamente para cada tipo de falha implementado, sem que ocorra também uma interferência de uma falha sobre a outra, a Tabela 2 apresenta um resumo dos itens desenvolvidos para cada etapa de projeto em Logisim^{TM}e VHDL, os itens desenvolvidos são classificados como 'OK' e os não desenvolvidos como '-', mostrando portanto que as falhas de interferência estão presentes somente no desenvolvimento da memória funcional com inserção de falhas em VHDL.

Tabela 2 – Comparação entre projetos

Etapas	**Logisim^{TM}**	**VHDL**
Modelo Funcional da NAND	OK	OK
Validação do modelo	OK	OK
Circuito de Inserção de falha	OK	OK
Falha Stuck-at 1	OK	OK
Falha Stuck-at 0	OK	OK
Falhas por interferência	-	OK
Validação do circuito de inserção de falhas	OK	OK

Fonte: Desenvolvido pelo autor

Ao definir-se o fluxo de desenvolvimento do projeto neste capítulo, em que através da

implementação em Logisim^{TM}foi possível validar a organização da memória e os circuitos necessários para seu funcionamento, partindo para o desenvolvimento em VHDL, que proporcionou o desenvolvimento de todas as falhas desejadas e por fim validadas, apresenta-se no próximo capítulo 5 de Desenvolvimento o detalhamento das funções de cada circuito implementado, juntamente com as conexões entre eles e o comportamento das falhas aplicadas em cada etapa do projeto.

5 DESENVOLVIMENTO

Através deste capítulo são descritos o desenvolvimento dos circuitos que compõem o modelo da memória com inserção de falhas, para isto foram implementados o modelo funcional da memória com seus respectivos circuitos e o circuito de inserção de falhas com o detalhamento de cada implementada.

5.1 Modelo funcional da memória

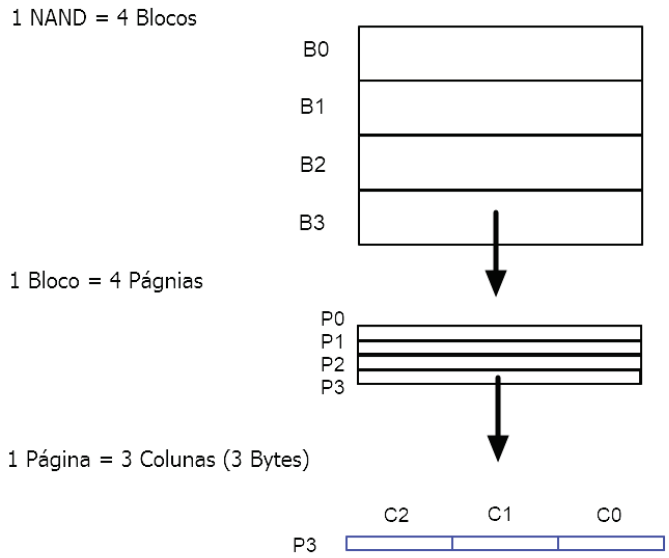
A etapa de desenvolvimento da memória NAND seguiu o método de projeto chamado *Botton Up*, ou seja, a implementação se inicia pela menor porção de memória, sendo uma única célula, verificando o seu funcionamento, até ao circuito final em que ao longo do desenvolvimento foi acrescentado funcionalidades e maior complexidade ao projeto. As 2 maneiras de implementações em Logisim^{TM}e VHDL foram também realizadas de forma modular, em que é desenvolvido um módulo do circuito que pode ser replicado e reutilizado da maneira necessária, tendo como exemplo uma célula com as suas funcionalidades sendo replicada para formar um *Byte* de células e consequentemente uma página, exemplo detalhado na próxima seção 5.1.1 do circuito da matriz da memória.

5.1.1 Circuito da Matriz da memória

Seguindo o planejamento da implementação explicado anteriormente, o desenvolvimento iniciou-se através do circuito da matriz da memória, onde são armazenadas os bits de informações, composta por 4 blocos, que contém 4 páginas cada e cada página possui 3 colunas de 8 bits, sendo um total de 24 bits por página, multiplicando por 4 páginas tem-se 96 bits por bloco, juntando com os 4 blocos implementados, temos uma matriz da memória de 384 bits ou 48 *Bytes* de memória, como pode ser visto na Figura 24.

Conforme explicado na Seção de caracterização da memória NAND Flash (2.2.1) do referencial teórico, o modelo desenvolvido segue a divisão em página como a menor porção de endereçamento para a função de leitura e programação e o bloco como a menor unidade de apagamento possível, portanto a implementação do pino de entrada e saída de dados do circuito da matriz da memória segue o tamanho exato em bits de uma página, sendo 24 bits de dados que são diretamente enviados à página selecionada ou enviado para fora da matriz em caso de leitura, pinos seletores para o endereçamento são as entradas para selecionar qual bloco que ocorrerá a operação (*Select Block*) e para qual página também (*Select Page)*, juntamente com um sinal de liberação que depende exatamente de qual operação está sendo realizada, programação (*En Write)*, apagamento (*En Erase)* ou leitura (*En Read)* e o último pino implementado para a matriz da memória é o pino de *Clock*, utilizado somente no projeto em Logisim^{TM}, devido ao fato da

Figura 24 – Matriz da memória



Fonte: Elaborado pelo Autor

célula de armazenamento em Logisim^{TM}ser um circuito *flip-flop*, ou seja, para que haja uma troca de estados nos *flip-flops* é necessário um *clock* que funciona em máxima frequência para simular uma troca de estado instantânea, segue a descrição dos pinos na Tabela 3.

Tabela 3 – Conexões da Matriz da Memória

Pino	**Entrada/Saída**	**No. de Bits**	**Descrição**
Dado	Entrada/Saída	24	Entrada para programação e saída para leitura dos dados
Select Block	Entrada	2	Seleciona o bloco
Select Page	Entrada	2	Seleciona a página
En Write	Entrada	1	Sinal para liberar a escrita na página
En Erase	Entrada	1	Sinal para liberar o apagamento no bloco
En Read	Entrada	1	Sinal para liberar a leitura na página
Clock (Logisim)	Entrada	1	Clock para atualizar o estado dos flip flops no logisim

Fonte: Desenvolvido pelo autor

A matriz da memória possui como a menor porção de armazenamento a célula de memória, que por sua vez possui algumas características de funcionamento específicas para cada operação enviada ao modelo, sendo detalhadas na próxima seção 5.1.2 do circuito da célula de memória.

5.1.2 Célula de memória

A célula de memória representa funcionalmente um único bit, que em caso de célula programada, o valor lógico armazenado é ’0’ e em caso de célula apagada o valor lógico se torna ’1’.

Funcionalmente a célula NAND Flash possui a característica de se tornar apagada (’1’) somente com o recebimento do comando de apagamento, ou seja, em caso da célula receber pelo pino de dado uma programação com o valor lógico ’1’, ela ignora o valor e mantém o seu estado atual na célula, portanto o valor da célula é alterado pelo recebimento de dados somente quando o valor do pino de dado receber ’0’ de programação.

O projeto em Logisim^{TM}possui como implementação da funcionalidade da célula de armazenamento o circuito *flip flop*, que possui como requisito de funcionamento a entrada do sinal de *clock* para atualização do valor registrado no *flip flop*, conforme explicado anteriormente, para o funcionamento da célula são necessários os pinos de *Dado* para entrada e saída do bit, pino de *Enable* que habilita o funcionamento da célula, o pino de *Erase* para a célula saber que está sendo apagada.

Tabela 4 – Conexões da Célula da Memória

Pino	Entrada/Saída	No. de Bits	Descrição
Dado	Entrada/Saída	1	Entrada para programação e saída para leitura dos dados
Enable	Entrada	1	Habilita a célula a receber o dado ou enviar
Erase	Entrada	1	Sinal ativado para apagamento da célula
Clock (Logisim)	Entrada	1	Clock para atualizar o estado dos flip flops no logisim

Fonte: Desenvolvido pelo autor

De acordo com a organização da memória explicado na seção da estrutura da memória NAND Flash (2.1.2), a página é composta por *Bytes* de memória que são selecionados através do endereçamento em coluna e o endereçamento em linha realiza o acesso ao bloco desejado e a página desejada, desenvolvendo portanto o circuito de endereçamento detalhado na próxima seção 5.1.3 do endereçamento da memória.

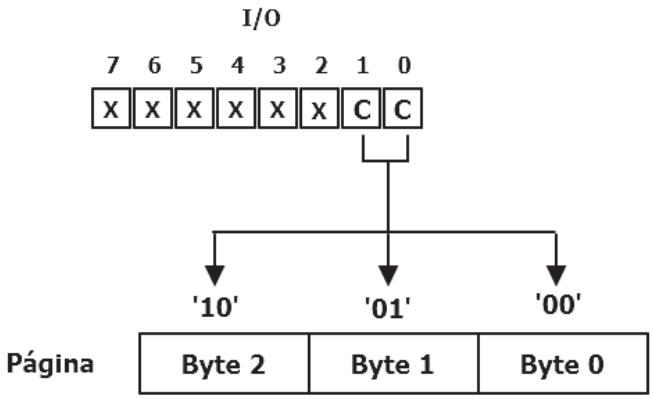
5.1.3 Endereçamento da memória

Para o acesso a matriz da memória foi realizado o circuito de endereçamento, sendo o circuito responsável por ativar o endereço alvo da operação que é enviado através dos 8 pinos de I/O da memória NAND.

Cada operação enviada para a memória, deve obedecer a regra de endereçamento descrita no datasheet da NAND, para as operações de leitura e programação o primeiro endereço recebido pela matriz é a coluna, que seleciona a partir de qual dos *Bytes* é iniciado a operação, sendo desenvolvidas 3 colunas por página, ou seja, cada Byte da página é acessado através da coluna, conforme Figura 25.

Após o envio do endereço da coluna é realizado o envio do endereço da linha da matriz que contém o endereço do bloco e página alvo, no caso da operação de apagamento, é enviado somente o endereço em linha pois o dado que é utilizado para apagamento é somente o endereço do bloco. A linha desenvolvida no projeto é conforme o número de blocos e páginas da matriz,

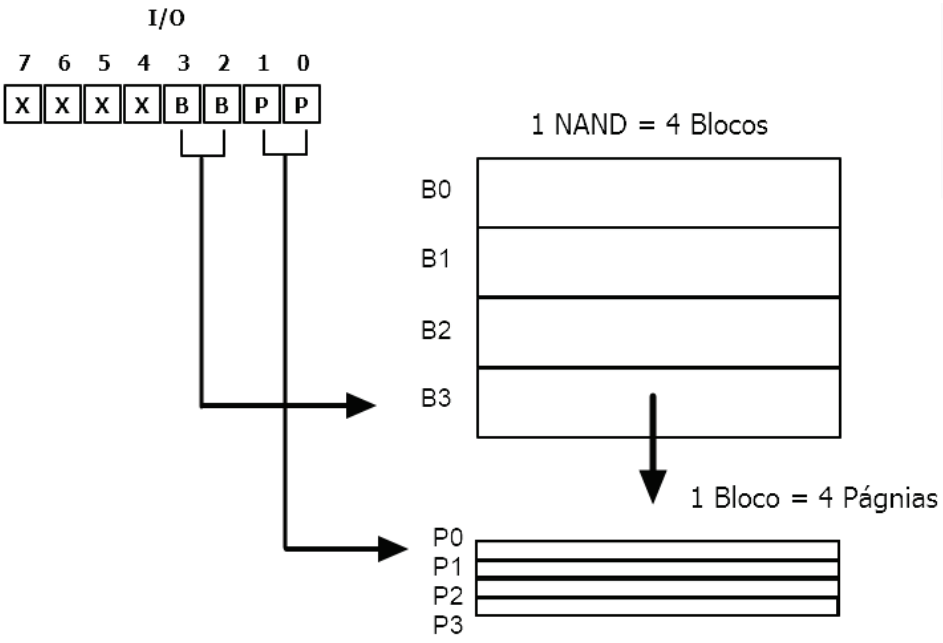
Figura 25 – Endereçamento em Coluna



Fonte: Elaborado pelo Autor

sendo composta por 4 bits de endereçamento, 2 primeiros bits para acesso as 4 páginas no bloco e os outros 2 bits para acesso aos 4 blocos implementados, conforme mostra a Figura 26.

Figura 26 – Endereçamento em Linha



Fonte: Elaborado pelo Autor

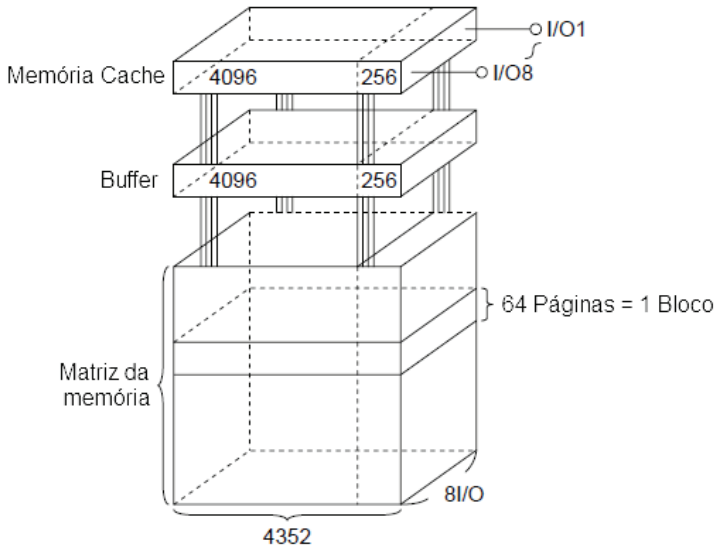
Ao finalizar o desenvolvimento da matriz da memória e o endereçamento, obedecendo sempre as características funcionais, realizou-se o desenvolvimento dos 2 circuitos de memórias auxiliares à matriz da memória, sendo o circuito de *cache* e *buffer* de dados, detalhados na seção 5.1.4.

5.1.4 Circuito de Cache e Buffer de dados

A entrada e saída dos dados da matriz da memória é realizado através dos 8 pinos de I/Os externos da NAND Flash e para isto a memória possui 2 circuitos internos de armazenamento dos dados chamados Cache de dados e Buffer de dados, conforme mostra a Figura 27.

O circuito de *buffer* é o circuito de armazenamento interno mais próximo a matriz da memória, tendo o espaço de armazenamento igual ao de uma página da matriz, em caso de uma operação de Leitura a matriz da memória envia o conteúdo da página selecionada para o *buffer* que envia os dados ao circuito de cache, ou o caminho oposto em caso de uma operação de programação. As finalidades do circuito de *buffer* são caracterizadas de forma a otimizar as operações realizadas na memória, sendo por exemplo a funcionalidade de em caso de leitura, os dados da página subsequente do endereço enviado é pré carregado no *buffer* para que possa otimizar o tempo da próxima leitura, em caso de programação os dados são movidos do *cache* para o *buffer* e assim liberando o cache para receber os próximos dados de programação enquanto os dados do *buffer* são transferidos para a memória. Estas 2 funcionalidades do circuito de *buffer* são consideradas como trabalhos futuros de aprimoramento das funcionalidades da memória NAND, pois desvia do objetivo geral do projeto de proporcionar as 3 operações básicas para a aplicação do teste na memória, sem levar em consideração qualquer otimização de tempo.

Figura 27 – Circuito de Cache e Buffer



Fonte: Adaptado de Electronics (2010)

O circuito de memória Cache em caso de programação é responsável pelo armazenamento dos dados da página antes de serem enviados para o *buffer*, armazenando de forma que a cada pulso de programação (WE), os 8 bits de dados contidos nos pinos de *I/O* são armazenados na Cache e posteriormente enviados ao circuito de *buffer*, em caso de leitura, a memória cache recebe do buffer o conteúdo lido na página e a cada pulso de leitura (RE), envia para os 8 pinos de

I/O 8 bits de dados de cada vez, fazendo portanto o papel de uma memória cache com densidade de uma página.

Ao desenvolver todo o armazenamento dos dados através da matriz da memória e os 2 circuitos auxiliares de *cache* e *buffer*, realizou-se a aplicação do circuito de controle que realiza o gerenciamento dos sinais que realizam cada operação de leitura, apagamento ou programação, sendo detalhado na seção 5.1.5 do circuito de controle.

5.1.5 Circuito de controle

O circuito de controle é responsável diretamente pelo gerenciamento dos comandos enviados à memória, pois é ele que recebe o comando e verifica se é um comando válido conforme o *datasheet* da memória NAND, enviando para a matriz da memória sinais de liberação (*Enable)* do acesso ao endereço, enviando também ao circuito de cache a liberação dos dados que são enviados para dentro da matriz em caso de programação ou a liberação dos dados para os pinos de I/Os em caso de leitura, em caso de apagamento envia para a matriz o sinal de liberação de apagamento do bloco desejado. Para a realização de cada operação é necessário obedecer uma lógica combinacional dos sinais de controle e seguir a tabela de comandos de acordo com o datasheet Electronics (2010).

A Tabela 5 apresenta os 3 comandos implementados no modelo da memória NAND, escrita, leitura e apagamento, de forma que a operação desejada necessita de 2 ciclos de comando, sendo o primeiro ciclo para dar início ao comando e o segundo ciclo para informar à memória que os endereços e dados já foram enviados, encerrando o comando e liberando a memória para realizar internamente a operação enviada.

Tabela 5 – Tabela de comandos

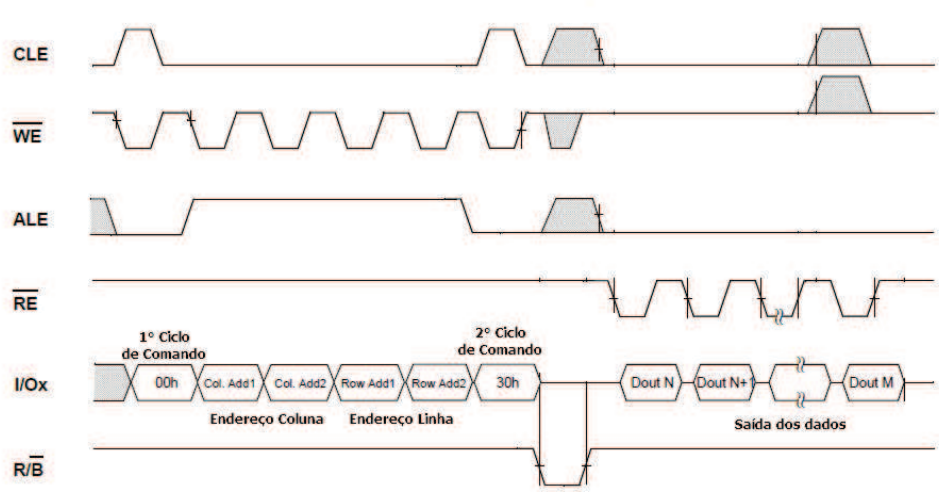
Operação	**Primeiro Ciclo**	**Segundo Ciclo**	**Descrição**
Leitura	00h	30h	Leitura em página
Escrita	80h	10h	Escrita em página
Apagamento	60h	D0h	Apagamento em bloco

Fonte: Desenvolvido pelo autor

A operação de leitura ilustrada na Figura 28 se inicia pelo comando 00h, com a utilização do sinal CLE para envio de comandos e um pulso do sinal WE, após é enviado o endereço coluna e linha juntamente com a utilização do sinal ALE para endereçamento e pulsos do sinal WE para enviar a informação. O endereço no modelo desenvolvido é realizado somente em 2 pulsos, sendo coluna seguido da linha. Para finalizar o comando de Leitura é necessário o comando 30h, sinalizando para a memória que pode carregar os dados da página para o circuito de cache, neste período o sinal de saída da memória R/B envia 1 para a saída, sinalizando que a memória está ocupada em processamento, sendo implementado um tempo fixo de 40us fornecido pelo *datasheet* como tempo médio de processamento da NAND para leitura. Após a finalização do

processamento é utilizado pulsos do sinal RE para enviar cada byte para os 8 *I/Os*, realizando assim a leitura da página.

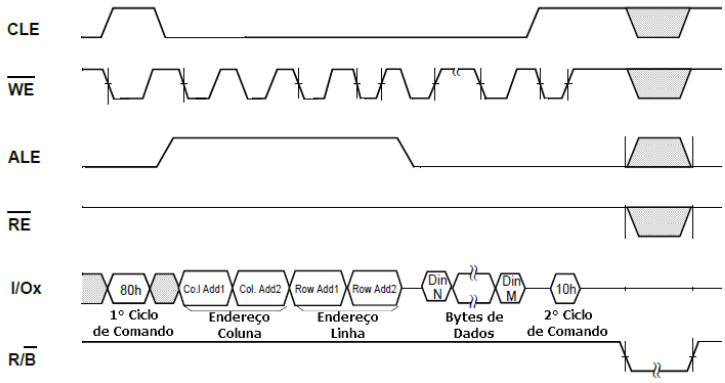
Figura 28 – Comando de Leitura



Fonte: Adaptado de Electronics (2010)

A operação de programação ilustrada na Figura 29 se inicia com o comando 80h, utilizando os mesmos sinais de CLE e WE da leitura, após é enviado o endereço de coluna e linha com o sinal ALE habilitado e em seguida é enviado os dados para a escrita, sendo que a cada pulso do sinal WE, os 8 bits nos pinos de *I/Os* vão sendo carregados no circuito de cache. Ao finalizar o envio dos dados para o cache é necessário o envio do segundo comando 10h para finalizar a programação e fazer com que a memória inicie o processamento de enviar os dados do cache para a matriz da memória, sendo necessários 250us de processamento, obtido pela média de tempo informado pelo *datasheet*.

Figura 29 – Comando de Programação

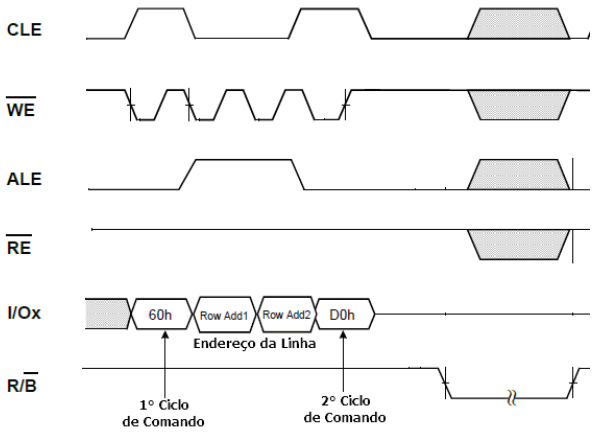


Fonte: Adaptado de Electronics (2010)

A operação de apagamento ilustrada na Figura 30 inicia com o comando 60h, o endereçamento do apagamento é composto somente pelo envio da linha, pois nela está contido o endereço

do bloco que se deseja apagar. Para finalizar a operação de apagamento é necessário o envio do comando D0h que sinaliza para a memória iniciar o processamento de apagamento do bloco, sendo necessários 2 ms para realização da operação conforme informado pelo *datasheet*.

Figura 30 – Comando de Apagamento



Fonte: Adaptado de Electronics (2010)

Para realizar as 3 operações básicas da memória o circuito de controle possui os seguintes pinos descritos na Tabela 6, em que o comando requerido é recebido através do pino de entrada *Comando* de 8 bits, recebidos diretamente dos 8 pinos de *I/Os* da memória, o sinais de entrada *CLE* é responsável pela liberação do recebimento do comando dos pinos de I/Os, os pulsos *RE* e *WE* são recebidos diretamente dos pinos externos da memória e são responsáveis pelos pulsos de liberação da leitura na matriz e da escrita respectivamente, um sinal de *clock* é criado no circuito de controle para simular o tempo de processamento da memória após o envio do segundo ciclo de comando, os sinais de saída são composto pelo sinal de *Busy* que é interligado diretamente ao pino de saída externo da memória (*R/B*) e pelos 3 pinos de liberação para cada operação que são enviados a matriz da memória, sendo o pino *EN ERASE* para liberação do apagamento, o pino *EN READ* para liberação da leitura e o pino *EN SEND* para a liberação da escrita.

Tabela 6 – Conexões do Circuito de Controle

Pino	Entrada/Saída	No. de Bits	Descrição
Comando	Entrada	8	Recebimento do comando enviado
CLE	Entrada	1	Sinal para liberar o recebimento do comando
RE	Entrada	1	Pulso para liberar a leitura
WE	Entrada	1	Pulso para liberar a escrita
CLK	Entrada	1	Clock utilizado para o tempo de processamento (*Busy*)
Busy	Saída	1	Sinal para indicar o tempo de processamento da memória
EN ERASE	Saída	1	Sinal para liberar o apagamento
EN READ	Saída	1	Sinal para liberar a leitura
EN SEND	Saída	1	Sinal para liberar a escrita

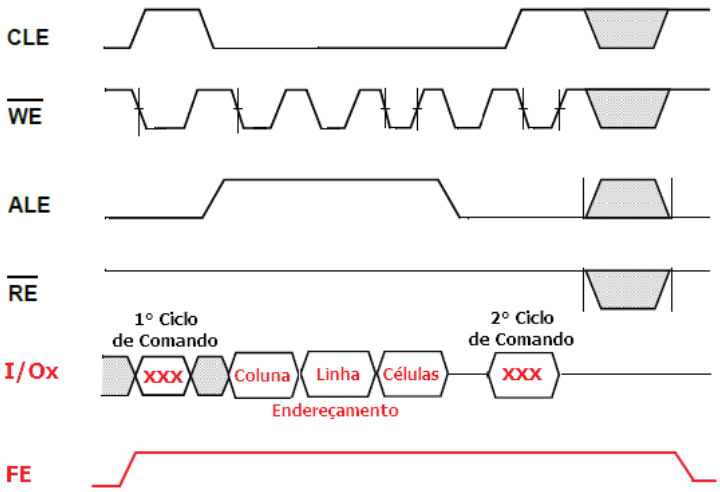
Fonte: Desenvolvido pelo autor

Através da finalização do desenvolvimento do modelo funcional da memória, capaz de realizar as 3 operações básicas na matriz da memória, foi possível verificar e validar o seu funcionamento descrito no próximo capítulo 6 de verificação e validação. Seguindo o fluxo de desenvolvimento após a verificação e validação do modelo funcional da memória foi inserido o circuito de inserção de falhas detalhado na seção 5.2, capaz de inserir uma determinada falha na posição desejada.

5.2 Circuito de inserção de falhas

O desenvolvimento do circuito de inserção de falhas iniciou-se através da adição do pino externo da memória chamado FE (*Fail Enable*), conforme apresentado na Figura 18, responsável por habilitar o modelo funcional da memória a receber as 3 informações para se inserir uma falha na memória, a primeira é o comando referente a falha que se deseja inserir, o endereço específico da memória e por fim o comando de encerramento da operação, para que então as operações de inserção da falha desejada sejam executadas, conforme mostra a Figura 31 e explicado em detalhes nas Seções a seguir.

Figura 31 – Comando de inserção de falha



Fonte: Elaborado pelo autor

O comando de inserção de falhas obedece o mesmo formato de um comando de programação da memória NAND, em que são utilizados os pinos CLE para liberar o primeiro ciclo de comando e o segundo ciclo de comando de encerramento, o pino ALE para liberar o envio dos 3 endereços da falha e através do pino WE que é realizado os pulsos para passar para o próximo estado do sistema. Portanto, primeiramente é necessário a definição dos comandos para inserir a falha desejada, conforme pode-se observar na Tabela 7 em hexadecimal, para cada falha implementada possui o respectivo comando para o primeiro ciclo de comando e após o envio

do endereço é necessário o envio do segundo ciclo de comando para encerramento da operação, sendo implementado o mesmo comando (81x) para todos os casos.

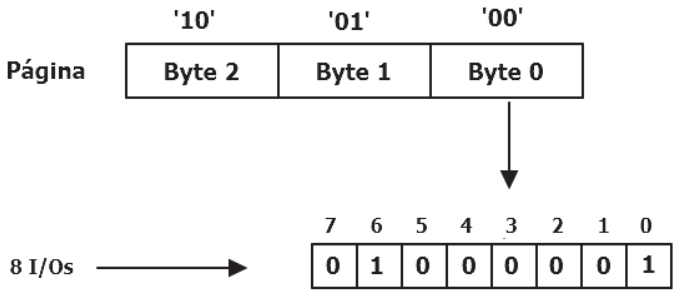
Tabela 7 – Comandos para habilitar falha

Falha	1º Ciclo	2º Ciclo
Stuck at 0	00x	81x
Stuck at 1	01x	81x
WPD	02x	81x
BPD	03x	81x
WED	04x	81x
BED	05x	81x
RPD	06x	81x
RED	07x	81x
Clear	FFx	81x

Fonte: Desenvolvido pelo autor

O endereçamento da falha consiste basicamente em 3 etapas, as 2 primeiras são as mesmas já detalhadas no tópico de endereçamento da memória NAND (5.1.3), em que é enviado o endereço da coluna em que se deseja inserir a falha, após é enviado o endereço em linha que define em qual bloco e página e por fim a terceira etapa é aplicada especificamente para o circuito de inserção de falha, ou seja, esta etapa de endereçamento é utilizada somente quando o sinal FE está habilitado, que consiste no envio da localização exata da célula que se deseja aplicar a falha. Através do endereçamento da célula é possível habilitar até no máximo 8 células a receberem a falha, a medida em que a implementação consiste em verificar o endereço enviado através dos 8 pinos de *I/Os* em qual deles possui o valor lógico '1' para ativar a falha ou valor lógico '0' para não aplicar nenhuma falha, conforme apresenta a Figura 32 de exemplo, em que após ser selecionado a coluna, as células 0 e 6 são aplicadas a falha desejada.

Figura 32 – Endereçamento da célula com falha



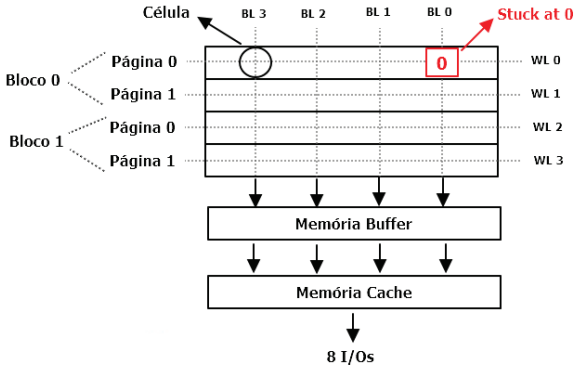
Elaborado pelo autor

Finalizando o modo de endereçamento da falha e os comandos referentes a cada uma, são descritos na seção 5.2.1 o desenvolvimento e o comportamento de cada falha aplicada ao projeto, demonstrando através de figuras o comportamento específico dentro da matriz da memória.

5.2.1 Tipos de Falhas Desenvolvidas

Após ser descrito o procedimento para inserir um tipo de falha na matriz da memória apresenta-se o detalhamento funcional para cada falha desenvolvida, iniciando-se pela falha do tipo *stuck-at* 0 que afeta a célula da memória de modo que o seu valor lógico fique fixado em '0' (programada), ou seja, o comportamento funcional da célula desconsidera qualquer comando que seja enviado e sempre envia para a saída o valor de uma célula programada, conforme pode ser visto na Figura 33 em vermelho.

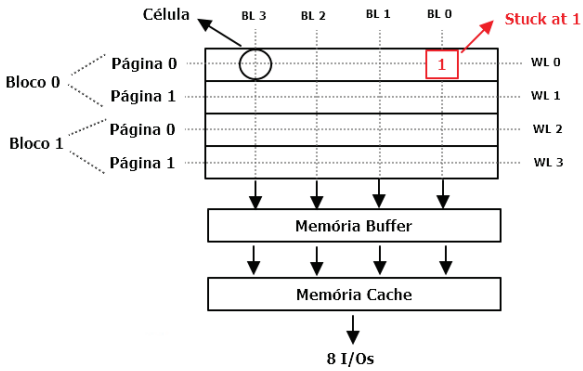
Figura 33 – Falha de *Stuck-at* 0



Elaborado pelo autor

A falha de *stuck-at* 1 afeta a célula da memória de modo que o valor lógico da célula fique fixada em '1' (apagada). Para a realização do modelamento da falha do tipo *stuck at* 1, o circuito de injeção de falha tem o controle para fixar o valor lógico na célula, sem que possa ser alterado por qualquer outra operação realizada na memória após a injeção da falha, comportamento observado na Figura 34.

Figura 34 – Falha de *Stuck-at* 1



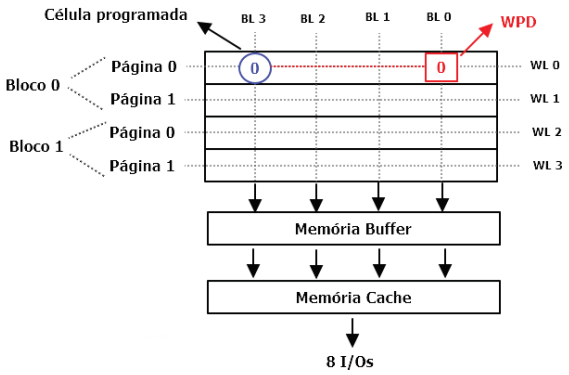
Fonte: Elaborado pelo autor

Além das falhas de *stuck-at*, foram desenvolvidas no projeto falhas de interferências entre células, característica presente em memórias NAND Flash e introduzida na Seção de

Falhas em memórias NAND Flash (2.4) do referencial teórico. É importante ressaltar que as falhas de interferência só se manifestarão nos casos da utilização das células interligadas pelos mesmos *bitlines* ou *wordlines*, ou seja, a célula que possui a falha é totalmente funcional, não apresentando erros a menos que se utilize suas células vizinhas.

O desenvolvimento iniciou-se pela falha de interferência *Wordline Program Disturbance (WPD)* ou Interferência de programação no *wordline*, fazendo com que a célula que possui a falha seja programada (valor lógico '0') se houver uma programação em qualquer outra célula no mesmo *worline*, ou seja, uma programação em qualquer outra célula na mesma página, como pode ser visto na Figura 35. A implementação da falha ocorre no circuito da Página da matriz da memória, pois há uma comunicação entre os bytes da página para saber se o *wordline* foi ativado ou não, habilitando uma *flag* que sinaliza a escrita no *wordline* e assim libera a falha.

Figura 35 – Falha de WPD



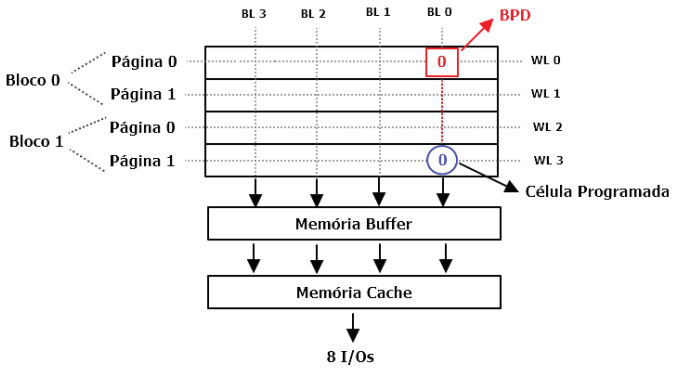
Fonte: Elaborado pelo autor

Outra falha de interferência por programação é o *Bitline Program Disturbance (BPD)* ou Interferência de Programação no *Bitline*, fazendo com que a célula que possui a falha seja programada (valor lógico '0') se houver uma programação em qualquer outra célula no mesmo *bitline*, isto significa que o *bitline*, conforme pode ser visto na Figura 36 de exemplo, é uma ligação entre as páginas e até mesmo entre blocos, sendo utilizado para o envio de cada bit da página para o *Buffer* da memória, no modelo implementado da memória estão presentes 24 *bitlines*, pois cada página possui 3 colunas de 8bits, dando um total de 24 colunas.

Tem-se também a falha *Wordline Erase Disturbance (WED)* ou Interferência de Apagamento no *Wordline*, alterando somente o comportamento da célula que possui a falha, tendo o seu valor lógico alterado para '1' (apagado) quando uma ou mais células são programadas no mesmo *wordline*, portanto o erro só será visível se a célula com falha esteja anteriormente programada ('0') e apagada após a falha ser ativada, conforme pode se observar na Figura 37.

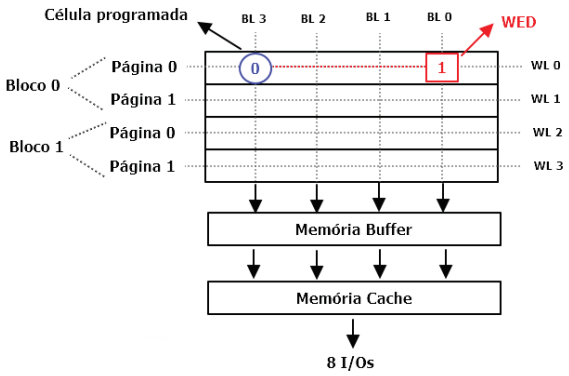
Outra falha implementada de apagamento da célula com falha é a de Interferência pelo *Bitline* ou *Bitline Erase Disturbance (BED)*, sendo uma falha na célula que está previamente

Figura 36 – Falha de BPD



Fonte: Elaborado pelo autor

Figura 37 – Falha de WED

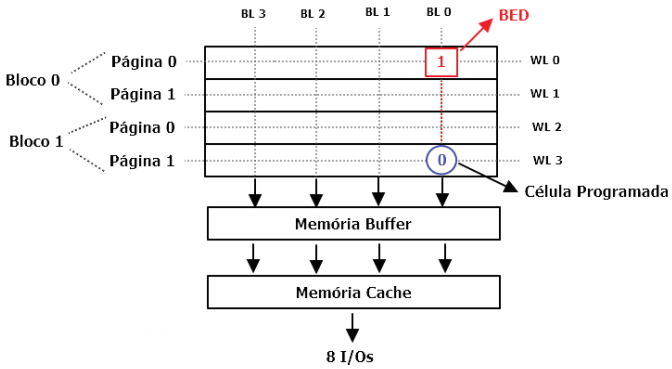


Fonte: Elaborado pelo autor

programada (’0’) que acontece quando outra célula do mesmo *bitline* é programada, fazendo com que aquela com falha seja apagada (’1’), conforme observa-se na Figura 38.

As 2 falhas de interferência por Leituras chamadas *Read Erase Disturbance (RED)* e *Read Program Disturbance (RPD)* são distintas das outras falhas por interferência já descritas anteriormente, de modo que a falha não se caracterize pelas ligações de *wordline* ou *bitline* na matriz da memória, mas sim pelo número de ocorrências da operação de leitura em uma determinada página. Para fins de utilização da falha e o aparecimento do erro, foi implementado um contador de leituras por página e um limite de 2 leituras, ou seja, se ocorrer 2 operações de leituras na mesma página, não consecutivas, as células que já receberem as falhas de RED ou RPD irão manifestar o erro. O limite de leituras se trata de uma variável de fácil alteração definindo o limite que o usuário deseja simular. A diferença entre os 2 modos de falhas de interferências é basicamente o valor de saída da célula que possui a falha, no caso da falha RED a célula será apagada (valor lógico ’1’), conforme visto na Figura 39(a) em que em azul a célula selecionada para a leitura e em vermelho o resultado da célula com falha, no caso da falha RPD

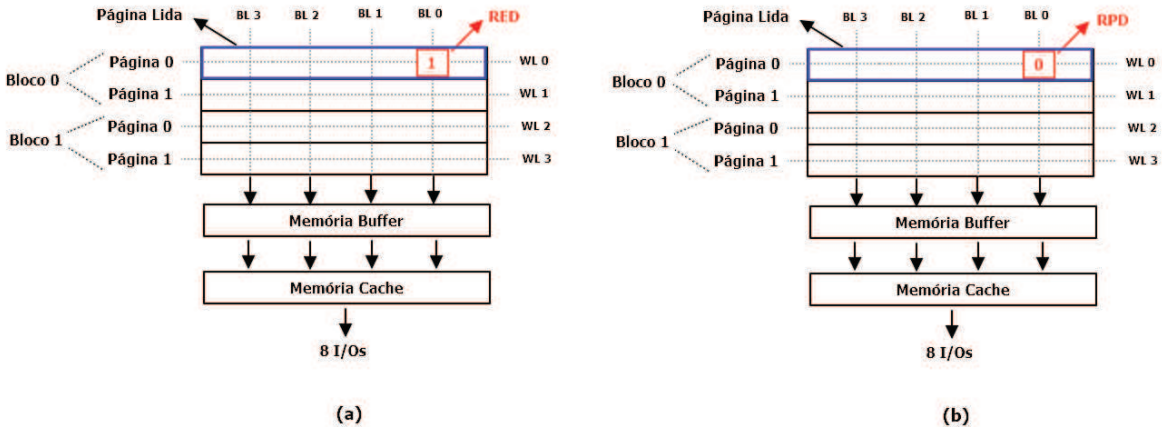
Figura 38 – Falha de BED



Fonte: Elaborado pelo autor

a célula resultará em um valor lógico ’0’ de programada conforme apresenta a Figura 39(b).

Figura 39 – Falha de RED



Fonte: Elaborado pelo autor

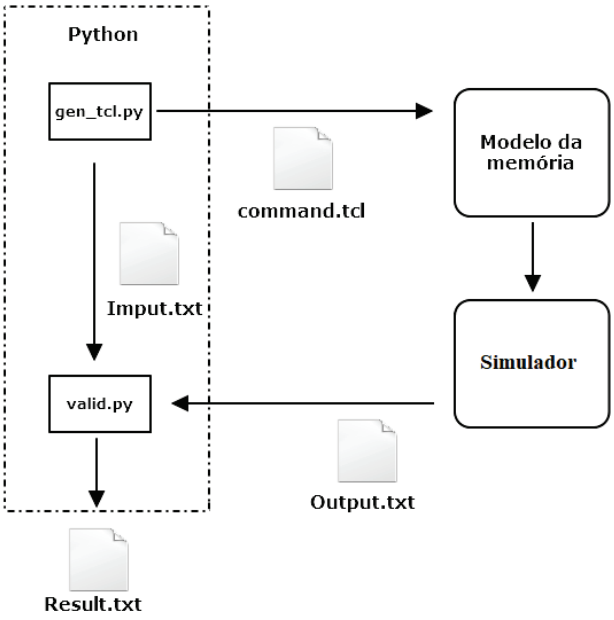
Após o desenvolvimento do circuito de inserção de falhas, juntamente com as falhas aplicadas ao modelo, realizou-se a verificação e validação sobre cada falha separadamente, utilizando algoritmos de testes para memória NAND Flash apresentados por outros autores, sendo este processo detalhado no capítulo 6 de verificação e validação.

6 VERIFICAÇÃO E VALIDAÇÃO

Neste capítulo serão detalhadas a etapa de verificação e a validação do projeto, sendo realizadas em duas etapas, a primeira sendo a verificação e validação do modelo funcional da memória e a segunda realizada após o desenvolvimento do circuito de inserção de falhas, verificando e validando cada falha separadamente, conforme detalhado nas seções a seguir.

Para a realização da etapa de verificação e validação do modelo funcional, foi necessário desenvolver uma automação em linguagem python, apresentada pela Figura 40, utilizada para: a geração dos dados de entrada à memória, a geração automatizada dos sinais de liberação (WE, RE, ALE, CLE) para cada operação desejada, a geração dos resultados esperados e por fim a comparação dos resultados obtidos da memória com o arquivo com os resultados esperados do sistema, para que assim o sistema possa ser verificado e validado de forma eficiente e otimizada.

Figura 40 – Automatização em linguagem Python



Fonte: Elaborado pelo autor

A linguagem *Python* foi utilizada para a otimização devido ao conhecimento prévio do autor e bolsista do projeto, realizando a geração de arquivos texto para as operações citadas acima, conforme observa-se na Figura 40, em que através do arquivo em Python *gen_tcl.py* são gerados 2 arquivos, o primeiro chamado *command.tcl* é enviado para o modelo da memória contendo todas operações desejadas e o segundo arquivo gerado *Imput.txt*, contém os dados esperados de saída do modelo, que são enviados para o programa em Python *valid.py*, como saída do modelo da memória temos o arquivo *Output.txt* contendo os dados armazenados na

memória a cada operação, ao final o programa *valid.py* realiza a comparação linha a linha dos arquivos *Imput.txt* e *Output.txt*, gerando o arquivo final chamado *Result.txt* que contém em cada dado comparado a informação de 'OK' para dados iguais e 'Fail' em caso de dados divergentes.

6.1 Verificação e validação do modelo funcional

Conforme Ribeiro (2015), a técnica de varredura sequencial dos endereços é utilizada para percorrer a matriz de armazenamento composta por n colunas e m linhas, percorrendo todas as posições sequencialmente atribuindo para cada endereço um valor de referência previamente mapeado, técnica utilizada para verificar o funcionamento das operações da memória e o endereçamento.

Após o desenvolvimento do método de verificação e validação em *Python*, iniciou-se a verificação funcional das 3 operações básicas desenvolvidas (programação, leitura e apagamento), efetuadas sequencialmente no endereço da memória, conforme apresenta a Figura 41 em hexadecimal, que representa o arquivo de saída gerado da memória, em que é realizado a escrita de valores crescentes para cada endereço de forma sequencial, verificando o funcionamento da função de escrita e o endereçamento, onde 'f' significa as células apagadas e cada coluna representa o conteúdo de cada bloco e de cada página, sendo a primeira coluna da Figura o Bloco 0 (B0) e Página 0 (P0) e a última coluna (Read) os 8 bits de saída em caso de leitura.

Figura 41 – Verificação da escrita e endereçamento

	B0.P0	B0.P1	B0.P2	B0.P3	B1.P0	B1.P1	B1.P2	B1.P3	B2.P0	B2.P1	B2.P2	B2.P3	B3.P0	B3.P1	B3.P2	B3.P3	Read
2	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000	xx
3	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	xx
4	020100	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	xx
5	020100	050403	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	xx
6	020100	050403	080706	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	xx
7	020100	050403	080706	0b0a09	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	xx
8	020100	050403	080706	0b0a09	0e0d0c	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	xx
9	020100	050403	080706	0b0a09	0e0d0c	11100f	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	xx
10	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	xx
11	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	xx
12	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	xx
13	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	xx
14	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	ffffff	ffffff	ffffff	ffffff	ffffff	xx
15	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	ffffff	ffffff	ffffff	ffffff	xx
16	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	ffffff	ffffff	ffffff	xx
17	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	ffffff	ffffff	xx
18	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	ffffff	xx
19	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	xx

Fonte: Elaborado pelo autor

A verificação da operação de leitura foi realizada através da leitura sequencial de cada endereço previamente escrito, quando ocorre uma operação de leitura, os 8 bits da coluna de uma página são enviados diretamente para os 8 pinos de I/Os, registrando no arquivo de saída (*Output.txt*, conforme observa-se na última coluna (Read) da Figura 42 em hexadecimal, em que inicia-se pelo dado lido '00' e encerra-se no último valor do último endereço lido '2f'.

A última verificação de funcionamento foi sobre a operação de apagamento, em que o comportamento da operação é o apagamento total de um bloco, seguindo de forma sequencial para os próximos blocos até que a memória seja totalmente apagada, conforme observa-se na Figura 43 os blocos previamente escritos apagando-se ou 'ffffff' em hexadecimal.

Figura 42 – Verificação da Leitura

	B0.P0	B0.P1	B0.P2	B0.P3	B1.P0	B1.P1	B1.P2	B1.P3	B2.P0	B2.P1	B2.P2	B2.P3	B3.P0	B3.P1	B3.P2	B3.P3	Read
2	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	00
3	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	01
4	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	02
5	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	03
6	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	04
7	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	05
8	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	06
9	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	07
10	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	08
11	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	09
12	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	0a
13	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	0b
14	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	0c
15	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	0d
16	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	0e
17	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	0f
18	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	10
19	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	11
20	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	12
21	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	13
22	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	14
23	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	15

Fonte: Elaborado pelo autor

Figura 43 – Verificação da Apagamento

	B0.P0	B0.P1	B0.P2	B0.P3	B1.P0	B1.P1	B1.P2	B1.P3	B2.P0	B2.P1	B2.P2	B2.P3	B3.P0	B3.P1	B3.P2	B3.P3	Read
2	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	2e
3	020100	050403	080706	0b0a09	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	2f
4	ffffff	ffffff	ffffff	ffffff	0e0d0c	11100f	141312	171615	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	xx
5	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	1a1918	1d1c1b	201f1e	232221	262524	292827	2c2b2a	2f2e2d	xx
6	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	262524	292827	2c2b2a	2f2e2d	xx
7	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	ffffff	xx

Fonte: Elaborado pelo autor

Realizada a verificação das funcionalidades desenvolvidas no modelo funcional e comparadas com os valores esperados, aplicou-se ao modelo a validação através de algoritmo de teste March-FT ((YEH et al., 2002)) que é capaz de verificar falhas funcionais no projeto e assim validar o modelo funcional sem que ocorra nenhuma falha. O algoritmo March-FT é utilizado por diversos autores para detecção de falhas em memórias NAND Flash (Wu, Huang e Wu (1999), Cheng et al. (2002) e Yeh et al. (2002)), conforme observa-se o algoritmo na Figura 45, em que *'f'*, *'r'* e *'p'* significam apagamento, leitura e programação respectivamente, '1' e '0' significam todos os valores lógicos '1' ou '0' respectivamente, e $\Uparrow$, $\Downarrow$ e $\Updownarrow$ significa o endereçamento ascendente, descendente e ambos respectivamente.

Figura 44 – Algoritmo de teste March-FT

$$\{(f); \Uparrow (r\mathbf{1}, p\mathbf{0}, r\mathbf{0}); \Updownarrow (r\mathbf{0}); (f); \Downarrow (r\mathbf{1}, p\mathbf{0}, r\mathbf{0}); \Updownarrow (r\mathbf{0}); \}$$

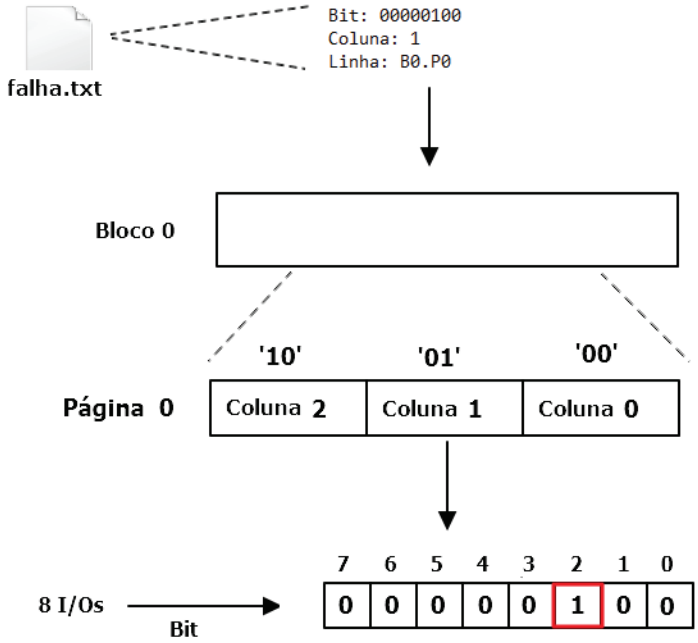
Fonte: Yeh et al. (2002)

Após a aplicação do algoritmo de teste ao modelo funcional, foi realizada através da automação da validação em Python a comparação dos arquivos com os dados esperados e os dados obtidos da matriz da memória, resultando em nenhuma falha encontrada e tendo portanto a memória funcional validada para poder ser desenvolvido o circuito de inserção de falhas.

6.2 Validação do circuito de inserção de falhas

Ao ser desenvolvido o circuito de inserção de falhas detalhado na Seção 5.2, para cada falha implementada e aplicada ao modelo funcional realizou-se a validação de cada falha separadamente para que não haja interferência entre elas, sendo primeiro necessário o desenvolvimento em Python de um gerador randômico de 16 endereços para a aplicação das falhas, para que não haja qualquer preferência por um endereço específico, chamado de *falha.txt*, conforme pode ser visto na Figura 45 em que a linha *Bit* representa qual dos 8 bits receberá a falha, a *Coluna* representa em qual das 3 colunas de uma página que receberá a falha e por último a *Linha* representa para qual bloco (B0) e para qual página (P0) a falha irá.

Figura 45 – Arquivo de endereçamento de falhas

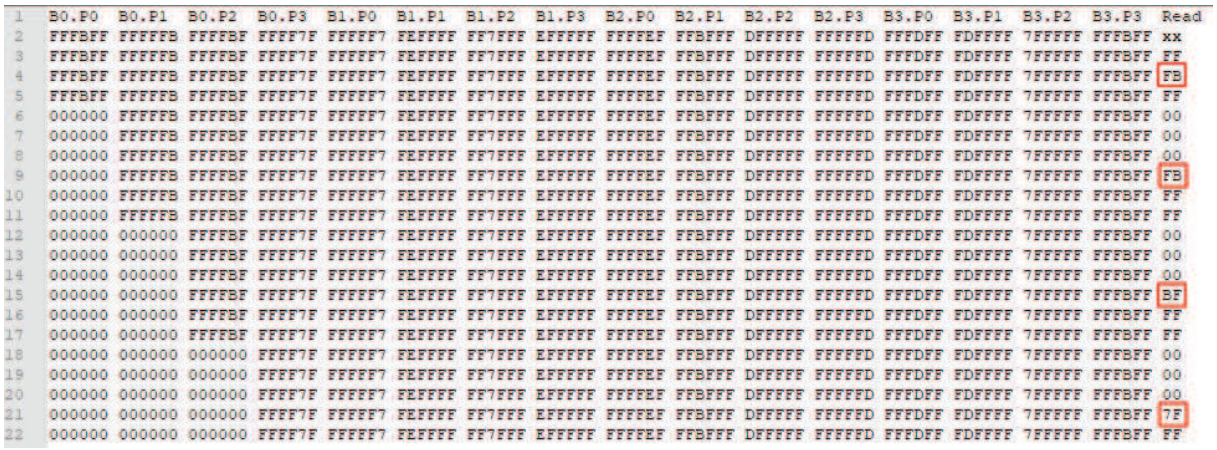


Fonte: Elaborado pelo autor

A validação iniciou-se pelas falhas de *stuck-at* 0 e *stuck-at* 1 utilizando o mesmo algoritmo de teste March-FT detalhado na Seção anterior, em que são inseridas 16 falhas em posições aleatórias na matriz da memória e a medida em que é executado o algoritmo de teste, ao erro da falha *stuck-at* 0 se manifesta já no primeiro comando de apagamento da memória, conforme pode ser visto na Figura 46, em que o comportamento da falha de fixar o valor da célula em valor lógico '0' aparece no momento que esta célula não é apagada e detectada pelo algoritmo, realizando o comando de '*r1*', em que se espera todos os valores '1', podendo ser detectadas as 16 falhas inseridas no sistema.

Por outro lado, a falha de *Stuck-at* 1, que faz com que a célula fique sempre apagada, se manifeste no momento que se realiza as operações de '*p0*' e detectadas quando se realiza as operações de '*r0*', em que se espera o valor lógico '0' de todas as células, conforme observa-se na

Figura 46 – Arquivo de falhas *Stuck-at* 0



	B0.P0	B0.P1	B0.P2	B0.P3	B1.P0	B1.P1	B1.P2	B1.P3	B2.P0	B2.P1	B2.P2	B2.P3	B3.P0	B3.P1	B3.P2	B3.P3	Read
2	FFFBFF	FFFFFB	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	xx
3	FFFBFF	FFFFFB	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	FF
4	FFFBFF	FFFFFB	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	FB
5	FFFBFF	FFFFFB	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	FF
6	000000	FFFFFB	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	00
7	000000	FFFFFB	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	00
8	000000	FFFFFB	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	00
9	000000	FFFFFB	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	FB
10	000000	FFFFFB	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	FF
11	000000	FFFFFB	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	FF
12	000000	000000	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	00
13	000000	000000	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	00
14	000000	000000	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	00
15	000000	000000	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	BF
16	000000	000000	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	FF
17	000000	000000	FFFFBF	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	FF
18	000000	000000	000000	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	00
19	000000	000000	000000	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	00
20	000000	000000	000000	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	00
21	000000	000000	000000	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	7F
22	000000	000000	000000	FFFF7F	FFFFF7	FEFFFF	FF7FFF	EFFFFF	FFFFEF	FFBFFF	DFFFFF	FFFFFD	FFFDFF	FDFFFF	7FFFFF	FFFBFF	FF

Fonte: Elaborado pelo autor

Figura 47, onde é possível observar o comportamento da falha na matriz da memória e portanto as 16 falhas inseridas de *stuck-at* 1 são todas detectadas.

Figura 47 – Arquivo de falhas *Stuck-at* 1

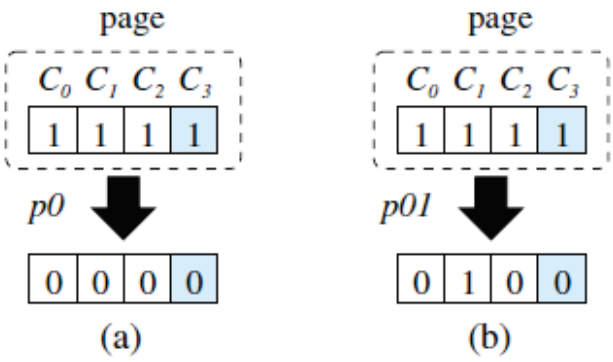
	B0.P0	B0.P1	B0.P2	B0.P3	B1.P0	B1.P1	B1.P2	B1.P3	B2.P0	B2.P1	B2.P2	B2.P3	B3.P0	B3.P1	B3.P2	B3.P3	Read
2	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FF
3	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FF
4	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FF
5	000400	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	00
6	000400	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	04
7	000400	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	00
8	000400	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FF
9	000400	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FF
10	000400	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FF
11	000400	020000	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	00
12	000400	020000	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	00
13	000400	020000	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	02
14	000400	020000	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FF
15	000400	020000	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FF
16	000400	020000	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FF
17	000400	020000	000800	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	00
18	000400	020000	000800	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	08
19	000400	020000	000800	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	00
20	000400	020000	000800	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FF
21	000400	020000	000800	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FF
22	000400	020000	000800	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FF

Fonte: Elaborado pelo autor

Para a validação das falhas por interferência, Hou e Li (2014) realizou um estudo da cobertura de falhas do algoritmo March-FT em memórias NAND Flash e concluiu que para falhas de interferência, o algoritmo possui 0% de cobertura de falha para o tipo de falha WPD, em que a célula com falha é programada se outra célula do mesmo *wordline* é programada, pois conforme mostra a Figura 48 (a), se uma página com 4 células, *C0*, *C1*,*C2*, *C3*, todas inicialmente apagadas, a célula *C3* possui a falha de WPD, quando é aplicado uma operação de '*p0*' do algoritmo March-FT, o valor da página se torna todos '0', fazendo com que as células que não possuem falha tenham o mesmo valor que a célula *C3*, mascarando o erro e tornando impossível de ser detectada.

Para evitar que ocorram falhas mascaradas, foi utilizado para a validação das falhas de

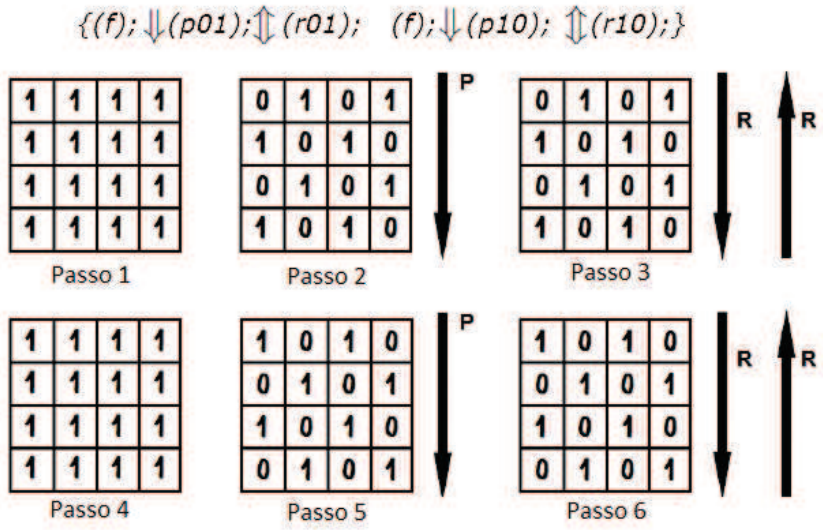
Figura 48 – Exemplo de página com 4 células que demonstra falha por WPD



Fonte: Adaptado de Hou e Li (2014)

interferência o algoritmo BF&D (Bridging Faults & Disturbances) para NAND Flash, apresentado pelo autor Carlo et al. (2010) e demonstrado pela Figura 49, em que é dividido em 6 passos, com os passos 2 e 5 de programação intercalada entre ’0’ e ’1’ sendo o maior diferencial em relação ao March-FT, pois dependendo da localização da falha ela pode se manifestar com a programação do passo 2 ou com a programação do passo 5, como pode ser visto na Figura 48 (b) de exemplo em que o valor esperado seria ’1’ e pode ser detectado.

Figura 49 – Algoritmo de teste BF&D



Fonte: Adaptado de Carlo et al. (2010)

A primeira falha validada através do algoritmo BF&D foi a de WPD, em a matriz da memória é escrita em valores lógicos alternados ’55’ ou ’AA’ em hexadecimal e as 16 falhas inseridas na memória se manifestam quando qualquer programação na mesma página ou *wordline* ocorre e o valor da célula com a falha for originalmente ’1’, se tornando um valor lógico ’0’,

podendo acontecer tanto no passo 2 do algoritmo ou no passo 5 dependendo da localização da falha, portando como pode ser visto na Figura 50 a detecção da falha no arquivo de saída do projeto, com o algoritmo BF&D foi possível detectar as 16 falhas.

Figura 50 – Arquivo de falha de interferência WPD

	B0.P0	B0.P1	B0.P2	B0.P3	B1.P0	B1.P1	B1.P2	B1.P3	B2.P0	B2.P1	B2.P2	B2.P3	B3.P0	B3.P1	B3.P2	B3.P3	Read
2	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
3	555555	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
4	555555	AAA2AA	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
5	555555	AAA2AA	455555	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
6	555555	AAA2AA	455555	AAAAAA	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
7	555555	AAA2AA	455555	AAAAAA	555555	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
8	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
9	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
10	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
11	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
12	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
13	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
14	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
15	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	FFFFFF	FFFFFF	FFFFFF	xx
16	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	AAAAAA	FFFFFF	FFFFFF	xx
17	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	AAAAAA	555555	FFFFFF	xx
18	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	AAAAAA	555555	AAAAAA	xx
19	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	AAAAAA	555555	AAAAAA	55
20	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	AAAAAA	555555	AAAAAA	55
21	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	AAAAAA	555555	AAAAAA	55
22	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	AAAAAA	555555	AAAAAA	AA
23	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	AAAAAA	555555	AAAAAA	A2
24	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	AAAAAA	555555	AAAAAA	AA
25	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	AAAAAA	555555	AAAAAA	55
26	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	AAAAAA	555555	AAAAAA	55
27	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	AAAAAA	555555	AAAAAA	45
28	555555	AAA2AA	455555	AAAAAA	555555	AAAAAA	555555	AAAA8A	555554	AAAAAA	555155	AA8AAA	555555	AAAAAA	555555	AAAAAA	AA

Fonte: Elaborado pelo autor

A falha de interferência WED possui o mesmo modo de acionamento da falha WPD, sendo através de uma programação na mesma página da célula com falha, mas a manifestação da falha para célula apagada (’1’), ocorre somente se o valor anterior a falha for programada (’0’), ou seja, o mesmo modo de acionamento da falha WPD, mas com comportamento inverso, sendo portanto validada e encontrado as 16 falhas inseridas de WED conforme apresenta a Figura 51.

Figura 51 – Arquivo de falha de interferência WED

	B0.P0	B0.P1	B0.P2	B0.P3	B1.P0	B1.P1	B1.P2	B1.P3	B2.P0	B2.P1	B2.P2	B2.P3	B3.P0	B3.P1	B3.P2	B3.P3	Read
2	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
3	555555	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
4	555555	AAAAAA	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
5	555555	AAAAAA	5555D5	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
6	555555	AAAAAA	5555D5	AAAAAA	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
7	555555	AAAAAA	5555D5	AAAAAA	555555	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
8	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
9	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
10	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
11	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
12	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
13	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	FFFFFF	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
14	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	FFFFFF	FFFFFF	FFFFFF	FFFFFF	xx
15	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	FFFFFF	FFFFFF	FFFFFF	xx
16	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	FFFFFF	FFFFFF	xx
17	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	FFFFFF	xx
18	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	AAAAAA	xx
19	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	AAAAAA	55
20	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	AAAAAA	55
21	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	AAAAAA	55
22	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	AAAAAA	AA
23	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	AAAAAA	AA
24	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	AAAAAA	AA
25	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	AAAAAA	D5
26	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	AAAAAA	55
27	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	AAAAAA	55
28	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	AAAAAA	AA
29	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	AAAAAA	AA
30	555555	AAAAAA	5555D5	AAAAAA	555555	AAAAAA	55D555	AAAEAA	555555	AAAAAA	55555D	AAABAA	D55555	AAAAAA	555555	AAAAAA	AA

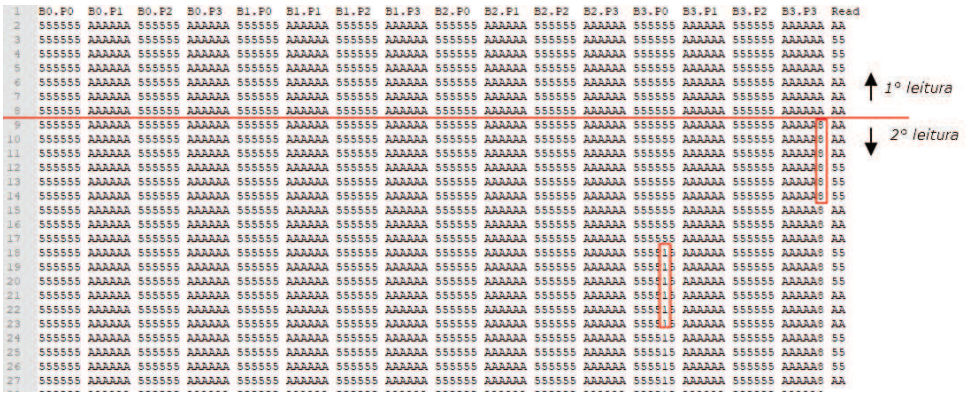
Fonte: Elaborado pelo autor

As falhas de interferência pelo *bitline*, BPD e BED também são manifestadas através da programação alternada do algoritmo BF&D, mas com a característica da programação na mesma coluna da célula que possui a falha, para a falha BPD a célula com falha deve estar previamente

com o valor ’1’ de apagada para que a falha possa acontecer e trocar o valor lógico para ’0’ de programada. Por outro lado, a falha BED deve estar previamente programada ’0’ para que a falha se manifeste para o valor ’1’ de apagada, sendo detectadas através do algoritmo as 16 falhas inseridas para cada um dos tipos.

Conforme detalhado na Seção 5.2.1, as falhas por leituras RPD e RED somente se manifestam após a segunda leitura realizada na página que possui uma célula com falha, portanto, utilizando o algoritmo BF&D no Passo 3, onde são realizadas 2 leituras em endereços ascendentes e descendentes, que as 2 falhas RPD e RED são manifestadas, conforme mostra a Figura 52 da falha RPD, que se manifesta se o valor da célula com falha for previamente apagada ’1’, se tornando um valor programado ’0’ após a segunda leitura, no caso da falha RED, o valor prévio da célula com falha deve ser programada ’0’ para que a falha se manifeste e apague o seu valor ’1’, detectando ao final as 16 falhas inseridas separadamente de RPD e RED.

Figura 52 – Arquivo de falha de interferência RPD



Fonte: Elaborado pelo autor

Utilizando os 2 algoritmos de teste apresentados, foi possível detectar e validar os 2 tipos de falhas *Stuck-at* e as 6 falhas de interferências desenvolvidas no projeto, dando a liberdade ao usuário do projeto, de definir a localização exata da falha, o tipo de falha que se deseja aplicar e de como detecta-las. Para a total simulação das falhas inseridas com um tamanho de memória pré-determinado, é necessário um determinado tempo de simulação do modelo, sendo um dos resultados do projeto apresentado na seção 6.3 de análise da simulação do modelo.

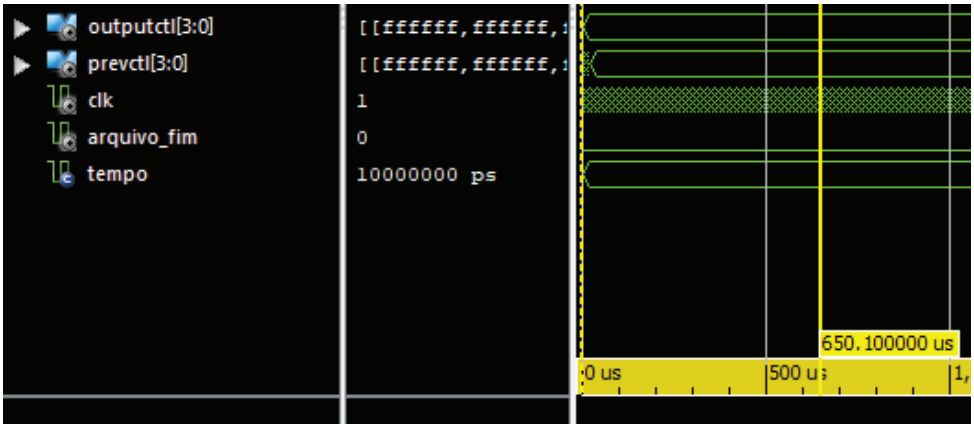
6.3 Tempo de simulação do modelo

Nesta seção são apresentados os dados de tempo de simulação do modelo em VHDL, considerando uma simulação de inserção de 16 falhas de interferência e a aplicação do algoritmo de teste BF&D, para determinadas densidades de memória.

Para a obtenção do tempo de simulação foi utilizado o modelo desenvolvido em VHDL, que através do software de simulação ISim da plataforma ISE da Xilinx, fornece o tempo total

de execução, conforme mostra a Figura 53, em que é selecionado o ponto inicial do modelo e o ponto final que se deseja medir (ex: 650.1us).

Figura 53 – Método de obtenção do tempo de simulação



Fonte: Elaborado pelo autor

O tempo de simulação é constituído pela etapa de inserção das 16 falhas na memória e o tempo da execução de todo o algoritmo de teste BF&D para a detecção das 16 falhas, possuindo o *clock* como parâmetro definido pelo autor de 10us e as operações de leitura, programação e apagamento obedecem os valores informados pelo *datasheet* de 40us, 250us e 2ms respectivamente. Conforme observa-se na Tabela 8, foram realizadas medições de tempo de simulação para diferentes densidades de memória.

Tabela 8 – Tempo de simulação

Densidade da memória	Tempo de teste	Tempo de simulação
24 Bytes	65,4ms	75,4ms
48 Bytes	136,7ms	147,1ms
96 Bytes	273,8ms	284,1ms

Fonte: Elaborado pelo autor

Analisando os dados obtidos através das simulações em VHDL foi possível observar a direta relação entre a densidade da memória com o tempo de teste necessário para verificar todas as falhas, em que ao dobrar a densidade simulada, o tempo de teste tende a dobrar o seu valor. Os valores obtidos levam em consideração apenas a execução do mesmo algoritmo de teste (BF&D) para a detecção das falhas, portanto apresentando um valor de tempo de teste que pode ser utilizado como referência para futuras implementações de algoritmos mais otimizados e com a mesma cobertura de falha, quando comparados ao algoritmo utilizado.

Nesta seção foram demonstrados os resultados em tempo das simulações realizadas para a detecção de falhas na memória desenvolvida, tendo como objetivo a apresentação de um tempo de teste base para futuras utilizações do projeto, em que se deseja desenvolver um algoritmo de teste que detecte as mesmas falhas desenvolvidas e com um tempo de teste menor, vindo ao

encontro do objetivo principal da dissertação de fornecer um modelo de memória NAND Flash com inserção de falhas validadas, que possibilite o estudo e desenvolvimento de algoritmos de teste. Para o atingimento do objetivo principal a seção 6.4 de análise dos resultados apresenta uma visão geral dos resultados obtidos durante todo o desenvolvimento.

6.4 Análise dos resultados

Através da utilização do modelo funcional da memória validado (Seção 6.1) e do circuito de inserção de falhas, inseriu-se na matriz de memória 16 falhas de cada uma das 8 desenvolvidas, para que então, foi utilizado algoritmos de testes como método de validação e detecção das falhas na memória, resultando na Tabela 9, resultando em 100% de detecção e validação dos 8 tipos de falhas desenvolvidos.

Tabela 9 – Resumo da detecção de falhas

Falha	**March-FT**	**BF&D**
Stuck at 0	100%	-
Stuck at 1	100%	-
WPD	-	100%
BPD	-	100%
WED	-	100%
BED	-	100%
RPD	-	100%
RED	-	100%

Fonte: Elaborado pelo autor

Com a utilização do algoritmo de teste March-FT, foi possível a detecção de 100% das falhas funcionais de *stuck-at*, devido a limitação do algoritmo para detecção de falhas de interferência, utilizou-se o teste BF&D para a detecção de 100% das falhas de interferência implementadas. Finalizando o desenvolvimento com uma memória NAND Flash funcional, capaz de inserir 8 tipos diferentes de falhas em qualquer endereço da memória, realizando o procedimento de validação através de algoritmos de testes que detectaram 100% das falhas inseridas e por fim contribuindo para o estudo e desenvolvimento de algoritmos de teste e a caracterização do comportamento interno da memória e suas falhas.

7 CONCLUSÃO

Através das características funcionais da memória NAND Flash, que implicam em falhas de interferência entre células e falhas comuns entre memórias, juntamente com a importância do teste na qualidade e custo de cada produto, é fundamental um modelo que possibilite o desenvolvimento e avaliação de algoritmos relacionas à memória, sendo este um modelo de memória NAND Flash capaz de inserir tipos de falhas implementadas no endereço desejado, respeitando sempre as características de funcionalidade da memória. Desenvolvendo portanto um modelo funcional de memória NAND Flash em Logisim^{TM}e VHDL, contendo um circuito para inserção de falhas caracterizadas, realizando através de algoritmos de teste (March-FT e BF&D) a detecção e validação do modelo da memória e de cada falha desenvolvida para o modelo.

O desenvolvimento do modelo em Logisim^{TM}, possui a sua principal relevância acadêmica, com a implementação de um circuito capaz de inserir falhas validadas, demonstrando primeiramente o entendimento específico do comportamento interno da memória, juntamente com o comportamento sobre cada falha inserida ao circuito, podendo ser aplicado para o ensino da memória NAND Flash em turmas de graduação, para que desde cedo alunos tenham o contato com as características da memória e assim impulsionar a pesquisa no assunto.

Com o desenvolvimento do modelo funcional com inserção de falhas em VHDL, atingiu-se o principal objetivo deste projeto, que era o desenvolvimento de um modelo funcional de memória NAND Flash, que possibilite o ensino e o desenvolvimento de algoritmos de teste aplicados à memória, utilizando o circuito simulado e validado com a falha escolhida no endereço que desejar. Além do objetivo principal, o modelo em VHDL possui as características de ser dinâmico em termos de densidade da memória e em termos da adição de outras funcionalidades, possibilitando a implementação de outras operações presentes no *datasheet* que aproxime mais a memória de um comportamento real, acrescente outros tipos de falhas ao sistema para que algoritmos de teste mais complexos sejam aplicados, o desenvolvimento de um controlador para a NAND, a alteração das células da memória para múltiplos níveis lógicos por célula (MLC) e por fim a implementação do modelo em um dispositivo programável (FPGA) para que possa ser utilizado por outros projetos que necessitem de uma memória que possua falhas ou não.

Portanto, através das duas implementações realizadas neste projeto, destaca-se as contribuições que o modelo é capaz de fornecer, que se inicia pela aplicação acadêmica, demonstrando o comportamento interno da memória e das falhas aplicadas, seguindo para aplicações em projetos de desenvolvimento de algoritmos de teste e por fim contribuindo para projetos que envolvam desde uma memória NAND Flash validada de qualquer densidade, até na continuidade do modelo, através da inserção de novas falhas e novas funcionalidades presentes no *datasheet*, contribuindo diretamente para futuros projetos que podem ser desenvolvidos a partir deste.

REFERÊNCIAS

AMORY, A. Integração e avaliação de técnicas de teste baseado em software no fluxo de projetos de SOCS. 2003. Citado na página 31.

ANTONI, L.; LEVEUGLE, R.; FEHÉR, B. Using run-time reconfiguration for fault injection applications. *Proceedings of the 18th IEEE*, v. 3, p. 1773–1777, 2001. Citado 2 vezes nas páginas 39 e 48.

ARITOME, S. Advanced Flash Memory Technology and Trends for File Storage Application. *IEDM Tech Dig*, p. 763–766, 2000. Citado 2 vezes nas páginas 21 e 24.

ARLAT, J.; CROUZET, Y. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on Computers*, v. 52, n. 9, p. 1115–1133, 2003. ISSN 0018-9340. Citado na página 38.

BEZ, R. et al. Introduction to Flash Memory. v. 91, n. 4, 2003. Citado 4 vezes nas páginas 21, 22, 23 e 24.

BURCH, C. *Logisim a graphical tool for designing and simulating logic circuits.* 2011. Disponível em: <http://www.cburch.com/logisim/pt/>. Citado na página 40.

BUSHNELL, M.; AGRAWAL, V. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. [S.l.]: Springer Publishing Company, Incorporated, 2013. ISBN 1475781423, 9781475781427. Citado 2 vezes nas páginas 30 e 31.

CARLO, S. D. et al. Exploring modeling and testing of NAND flash memories. *2010 East-West Design & Test Symposium (EWDTS)*, p. 47–50, 2010. Citado 5 vezes nas páginas 19, 37, 38, 41 e 74.

CHENG, K. L. et al. RAMSES-FT: A fault simulator for flash memory testing and diagnostics. *Proceedings of the IEEE VLSI Test Symposium*, v. 2002-Janua, p. 281–286, 2002. Citado 2 vezes nas páginas 43 e 71.

CIVERA, P. et al. Exploiting FPGA-based techniques for fault injection campaigns on VLSI circuits. *Proceedings 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, p. 250–258, 2001. ISSN 10636722. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=966777>. Citado na página 39.

COUTINHO, P. O Logisim como ferramenta cognitiva na aprendizagem de circuitos digitais lógicos. 2014. Citado na página 40.

D'AMORE, R. *VHDL - Descrição e Síntese de Circuitos Digitais.* [S.l.: s.n.], 2012. ISBN 8521614527. Citado na página 41.

ELECTRONICS, S. Samsung Datasheet - 1Gb NAND Flash (K9F1G08U0D). Rev. 1.1, 2010. Citado 7 vezes nas páginas 47, 48, 50, 59, 60, 61 e 62.

GADGE, M. P.; KARMORE, S. P. Design of Fault Detection Module for Embedded Ram Memory. v. 5, n. 1, p. 797–799, 2014. Citado na página 35.

GEISSLER, F. d. A. Metodologia de injeção de falhas baseada em emulação de processadores. 2014. Citado 3 vezes nas páginas 39, 40 e 43.

HARARI, E. *Nand Flah Memory Controller Exporting a Nand Interface.* [S.l.]: Saratoga, CA (US), 2012. Citado 2 vezes nas páginas 29 e 30.

HOU, C. S.; LI, J. F. Testing Disturbance Faults in Various NAND Flash Memories. *Journal of Electronic Testing: Theory and Applications (JETTA)*, v. 30, n. 6, p. 643–652, 2014. ISSN 15730727. Citado 5 vezes nas páginas 7, 17, 34, 73 e 74.

HSUEH, M.-C.; TSAI, T. K.; IYER, R. K. Fault injection techniques and tools. *Computer*, v. 30, n. April, p. 75–82, 1997. ISSN 0018-9162. Citado na página 39.

LALA, P. K. *An Introduction to Logic Circuit Testing*. [S.l.]: Morgan \& Claypool Publishers, 2009. 99 p. ISBN 9781598293500. Citado 2 vezes nas páginas 33 e 34.

LI, Y.; XU, P.; WAN, H. A fault injection system based on qemu simulator and designed for bit software testing. Trans Tech Publications, v. 347, p. 580–587, 10 2013. Citado na página 43.

MAITY, N.; MAITY, R. Vhdl and verilog: Unbiased compared and contrasted. v. 2, p. 356–363, 04 2007. Citado na página 40.

MICHELONI, R.; CRIPPA, L.; MARELLI, A. Inside NAND Flash Memories. In: . [S.l.]: Springer Netherlands, 2010. v. 1. ISBN 9788578110796. Citado 9 vezes nas páginas 17, 25, 26, 27, 29, 31, 32, 33 e 34.

Micron Technology. NAND Flash 101: An Introduction to NAND Flash and How to Design it into Your Next Product. *Technical Note*, p. 1–27, 2006. Disponível em: <papers3://publication/uuid/9355CCBD-2000-4573-B4C8-A5E2A83B44E0>. Citado na página 26.

MOHAMMAD, M. G.; SALUJA, K. K. Testing flash memories for tunnel oxide defects. *Proceedings of the IEEE International Frequency Control Symposium and Exposition*, p. 157–162, 2008. Citado 2 vezes nas páginas 35 e 36.

PRODROMAKIS, A.; ANTONAKOPOULOS, T. Non-Volatile Memory Emulator. p. 26504, 2015. Citado na página 44.

RIBEIRO, I. S. Mapeamento em Hardware para conversão dos sinais de controle de uma matriz SDRAM com base em um endereço absoluto. v. 1, 2015. ISSN 1098-6596. Citado 2 vezes nas páginas 44 e 70.

RICHTER, D. *Flash Memories*. Dordrecht: Springer Netherlands, 2014. v. 40. (Springer Series in Advanced Microelectronics, v. 40). ISBN 978-94-007-6081-3. Disponível em: <http://link.springer.com/10.1007/978-94-007-6082-0>. Citado 8 vezes nas páginas 7, 17, 19, 21, 22, 23, 24 e 27.

SEMICONDUCTOR, H. et al. Open NAND Flash Interface Specification. *Discovery*, 2011. Citado na página 19.

TENDELOO, Y. V.; VANGHELUWE, H. Logisim to DEVS translation. p. 13–20, 2013. Citado 2 vezes nas páginas 40 e 51.

WU, C.-F.; HUANG, C.-T.; WU, C.-W. RAMSES: a fast memory fault simulator. *Defect and Fault Tolerance in VLSI Systems, 1999. DFT '99. International Symposium on*, p. 165–173, 1999. ISSN 10636722. Citado 2 vezes nas páginas 43 e 71.

XU, J.; XU, P. The research of memory fault simulation and fault injection method for bit software test. p. 718–722, 2012. Citado na página 43.

YEH, J.-c. et al. Flash Memory Built-In Self-Test Using March-Like Algorithms. p. 2–6, 2002. Citado 2 vezes nas páginas 19 e 71.

YUN, J. H. et al. An Abstract Fault Model for NAND Flash Memory. v. 4, n. 4, p. 86–89, 2012. Citado 4 vezes nas páginas 18, 34, 35 e 43.

ZHANG, M. et al. FPGA-Based Failure Mode Testing and Analysis for MLC NAND Flash Memory. p. 434–439, 2017. Citado na página 44.