INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Preparação do teste de circuitos digitais CMOS

Alexandre Miguel Fernandes Valério

Dissertação para obtenção do Grau de Mestre em

Mestrado em Engenharia Electrotécnica e de Computadores

Júri

Presidente: Prof. José António Beltran Gerald

Orientador: Prof. Marcelino Bicho dos Santos

Vogais: Prof. Carlos Francisco Beltran Tavares de Almeida

Setembro de 2007

Agradecimentos

Para começar, quero agradecer ao Professor Marcelino Bicho dos Santos, que sempre que lhe foi possível fez tudo ao seu alcance para orientar o meu trabalho, esclarecer as minhas dúvidas com o seu conhecimento e enorme experiência, e finalmente por dar-me ânimo para realizar o trabalho.

À minha família, em especial aos meus pais e irmãos, por todo o apoio e força que me deram, não só durante este trabalho mas também durante o curso.

Também quero agradecer todo o auxílio que o Eng. Abílio Parreira me deu, que partilhou conhecimentos essenciais para a execução deste trabalho.

Ao INESC, que disponibilizou um local de trabalho e os programas comerciais necessários ao trabalho.

Por fim, mas não com menor importância, quero agradecer a todos os colegas e amigos com quem convivi diariamente na duração do curso e trabalho: André, Pedro, Tiago, Diogo Antão, Paulo M., Paulo R., J. Barata, Miguel, Nelson, Dave, Mika, Samuel, J. Leal, Fernando, Diogo C., D. Cordeiro, I. Leitão, João C. e Michael. Sem estas amizades os últimos anos teriam sido bem menos divertidos e mais complicados.

Resumo

Neste trabalho de investigação estuda-se um novo método para a preparação do teste de circuitos digitais CMOS, de modo a obter um teste optimizado tanto na detecção de faltas como no número de vectores de teste.

São utilizados dois modelos de faltas para detecção em tensão: linha fixa a zero/um (*Line Stuck-At* ou LSA), curto-circuito (*Bridging*) e um modelo de faltas em corrente (IDDQ).

A nova metodologia é proposta recorrendo a ferramentas comerciais e à *Programming Language Interface* (PLI).

Com o método de teste proposto é possível obter um número de vectores de teste em tensão em média 36,71% inferior, e uma cobertura de faltas *bridging* em média 8,65% superior para os circuitos de referência testados, em comparação com o método de teste utilizado actualmente na empresa AMI Semiconductor, em parceria com a qual o trabalho de investigação teve lugar.

Palavras-chave

Método de teste

Modelos de faltas

PLI

Simulação de faltas

Redução da lista de faltas

ATPG

Abstract

In this investigation work a new test flow for CMOS digital circuits test preparation is studied. With the proposed method, it is possible to obtain an optimized test in both fault detection and total number of test vectors.

Two voltage fault models are used: Line Stuck-At (LSA), Bridging, and one current fault model: IDDQ.

The new methodology is prepared using commercial tools and the Programming Language Interface (PLI).

With the new test flow, it is possible to obtain, in average, a reduction of 36.71% in the number of voltage test vectors, and a bridging fault detection 8.65% superior, for the benchmark circuits that are tested, when compared with the test flow presently used by AMI Semiconductor, a company that was closely involved in the proposal and advising of this work.

Keywords

Test flow
Fault models
PLI
Fault simulation
Fault list reduction
ATPG

Índice

Agradecimentos ... iii
Resumo ... iv
Palavras-chave ... iv
Abstract ... v
Keywords ... v
Índice ... vi
Lista de tabelas ... viii
Lista de figuras ... ix
Lista de abreviaturas ... x
1. Introdução ... 1
2. Técnicas de Teste ... 3
2.1. Modelos de faltas ... 4
2.1.1. Modelo Line stuck-at ... 5
2.1.2. Modelo *Bridging* ... 6
2.1.3. Modelo IDDQ ... 7
2.2. Geração automática de vectores de teste – ATPG ... 8
2.3. Conclusões ... 8
3. Síntese de Circuitos Digitais ... 9
3.1. Síntese lógica com scan ... 9
3.2. Exemplo de síntese lógica ... 11
3.3. Conclusão ... 14
4. Metodologias de Preparação do Teste ... 15
4.1. Método de teste actual ... 15
4.2. Metodologia proposta ... 17
4.2.1. ATPG de faltas LSA, simulação de faltas IDDQ com vectores de teste LSA e ATPG para as faltas IDDQ restantes ... 19
4.2.2. Geração aleatória e automática de faltas bridging ... 20
4.2.3. Simulação de faltas bridging com vectores de teste LSA ... 22
4.2.4. Simulação de faltas bridging com vectores de teste IDDQ ... 23
4.2.5. ATPG de faltas bridging ... 24
4.3. Programming Language Interface – PLI ... 24
4.4. Conclusão ... 24
5. Implementação das Metodologias ... 25
5.1. Software utilizado ... 25
5.2. Metodologia proposta ... 26
5.2.1. ATPG de faltas LSA, simulação de faltas IDDQ com vectores de teste LSA e ATPG para as faltas IDDQ restantes ... 26
5.2.2. Geração aleatória e automática de faltas bridging ... 32
5.2.3. Simulação de faltas bridging com vectores de teste LSA ... 43
5.2.4. Simulação de faltas bridging com vectores de teste IDDQ ... 45
5.2.5. ATPG de faltas bridging ... 46
5.3. Método de teste actual ... 48
5.4. Conclusões ... 51
6. Resultados ... 53
6.1. Exemplo da preparação do teste do circuito b18 ... 53
6.2. Resultados obtidos ... 56

7. Conclusão e Trabalhos Futuros ... 63
Referências bibliográficas ... 65
Anexos ... 67
Synopsys®.init ... 68
Cadence®.init ... 69
.synopsys_dc.setup ... 70
Circuito b18 ... 71
dv_b18.tcl ... 71
1s2i.txt ... 71
2s2b.txt ... 72
3b.txt ... 73
4s5x2b.txt ... 73
5s2b.txt ... 74
b18_scan.v (alterações após síntese no Design Vision®) ... 75
b18_tb.v (localização da inserção das chamadas de sistema PLI) ... 75

Lista de tabelas

Tabela 5. 1– *Software* utilizado 25
Tabela 5. 2 – Critérios do Tetramax® para a detecção de faltas *bridging* 44

Tabela 6. 1 - Número de faltas LSA e *Bridging* de cada circuito 56
Tabela 6. 2 – Detecção de faltas parcial com o novo método de teste 57
Tabela 6. 3 - Detecção de faltas parcial com o método de teste utilizado pela AMIS 58
Tabela 6. 4 – Resultados totais com o novo método de teste 58
Tabela 6. 5 - Resultados totais com o método de teste da AMIS 59
Tabela 6. 6 – Número total de vectores de teste em tensão e corrente para os dois métodos de teste estudados 60
Tabela 6. 7 – Tempo de processamento do CPU com o Novo método de teste 60
Tabela 6. 8 – Tempo de processamento do CPU com o método de teste da AMIS 61

Lista de figuras

Figura 2. 1 – Exemplo das várias camadas de um circuito integrado 3
Figura 2. 2 – Porta lógica AND com falta stuck-at-0 na saída.. 5
Figura 2. 3 – Exemplo de um nó que se pode controlar e observar 6
Figura 2. 4 – Modelos de faltas *bridging* ba0 e ba1 6
Figura 2. 5 – a) Inversor com transístor defeituoso b)Valores de IDDQ para um transístor bom e para um defeituoso ………………………………………………………………..7

Figura 3. 1 – Ideia geral da inserção de *scan* 10
Figura 3. 2 – Método "Multiplexed Flip-Flops" 10

Figura 4. 1 – Método de teste utilizado pela AMIS 16
Figura 4. 2 – Método de teste proposto.. 18

Lista de abreviaturas

AMIS – empresa AMI Semiconductor

ATE – equipamento de teste automático (Automatic Test Equipment)

ATF – método de teste da AMIS

ATPG – geração automática de vectores de teste (Automatic Test Pattern Generation)

ATPG-BRI – ATPG para faltas BRI

BRI – *Bridging* – curto-circuito

I2B – *IDDQ to BRI* – simulação de faltas BRI com vectores IDDQ

IDDQ – *Idd Quiescence* – corrente de repouso na alimentação (Idd)

LSA – *Line Stuck-At*

NTF – novo método de teste

PLI – *Programming Language Interface*

S2B – *Stuck-at to BRI* – simulação de faltas BRI com vectores LSA e ATPG para as faltas BRI restantes

S2I – *Stuck-at to IDDQ* – simulação de faltas IDDQ com vectores LSA e ATPG para as faltas IDDQ restantes

STIL – *Standard Test Interface Language*

VER_RAND – geração aleatória de faltas do tipo BRI

1. Introdução

Este trabalho enquadra-se numa colaboração com a empresa AMI Semiconductor. O modelo de faltas tradicionalmente utilizado para preparar o teste de circuitos integrados digitais (*Line Stuck-At*) não assegura os níveis de detecção de defeitos exigidos com as tecnologias actuais [1][2][3][4].

O objectivo é propor uma metodologia de preparação do teste, optimizada, e que deverá ser utilizada pela AMIS. Pretende-se maximizar a cobertura de falta, minimizar o número de vectores de teste e utilizar ferramentas comerciais a fim de permitir o processamento de circuitos complexos. Para este efeito, utilizam-se vários modelos de faltas, o que faz com que haja melhor precisão e melhor robustez [5], e várias ferramentas comerciais de uma maneira original, de forma a obter um número reduzido de vectores de teste com uma detecção de faltas optimizado.

O desenvolvimento tecnológico que permite o fabrico de circuitos integrados (CIs) com resoluções litográficas crescentes conduziu à possibilidade de fabricação de CIs digitais muito complexos.

A exigência de qualidade actual levou a que a indústria recorra à geração do teste para detecção múltipla de faltas (*N-detection*) LSA, o que leva a um número elevado de vectores de teste. Além da geração de vectores de teste para faltas LSA, também é gerado o teste para faltas IDDQ de modo a que se obtenha um número reduzido (na ordem de poucas dezenas) de vectores de teste IDDQ, uma vez que este tipo de teste é bastante dispendioso (em termos do tempo necessário para efectuar o teste).

Como o método de teste descrito acima é pouco eficiente no que toca ao elevado número de vectores de teste obtido, e como os modelos de faltas utilizados (LSA e IDDQ) já não são suficientes para representar correctamente o tipo de faltas existentes em circuitos [3][4], principalmente devido à miniaturização dos circuitos com uso de tecnologia cada vez com maior resolução litográfica, que leva ao aparecimento de uma maior quantidade de faltas do tipo curto-circuito (*bridging*) [6][7], é então necessário desenvolver um novo método de teste adaptado à tecnologia actual que seja mais eficiente que o que é feito neste momento.

O novo método de teste aqui estudado utiliza dois modelos de faltas distintos para o teste em tensão: *Line Stuck-At* (LSA), *bridging* (BRI) e um modelo para o teste em corrente: corrente de repouso (IDDQ). Além de se utilizar mais um modelo de faltas (BRI), antes de fazer a geração automática de vectores de teste para um tipo A de faltas, faz-se a simulação de faltas deste tipo com vectores de teste de um tipo de faltas B gerados anteriormente, de modo a detectar e eliminar da lista de faltas as faltas do tipo A que são detectadas por vectores de teste do tipo B, evitando a geração de novos vectores de teste desnecessariamente.

Em comparação com o que é feito actualmente na empresa AMIS, o método de teste proposto tem as seguintes vantagens:

- menor número de vectores de teste em tensão
- maior cobertura de faltas *bridging*

As desvantagens são as seguintes:

- além do gerador de vectores de teste (Tetramax®), necessita de um simulador de faltas em corrente (Verilog-XL®)

- enquanto que no método de teste da AMIS são necessários 4 passos, executados todos no mesmo programa (Tetramax®), com o método de teste proposto são necessários 6 passos, executados em dois programas distintos: no Tetramax® e no Verilog-XL®.

De início estuda-se o procedimento actual de preparação do teste na AMIS e as respectivas ferramentas para seguidamente se proceder à optimização do mesmo.

A fim de atingir este objectivo utiliza-se *software* comercial como a ferramenta de síntese de circuitos Design Vision®, o gerador de vectores de teste e simulador de faltas Tetramax®, o simulador de faltas Verilog-XL® e PLI (*Programming Language Interface*).

Esta dissertação divide-se nos seguintes capítulos:

- No Capítulo 2, Técnicas de Teste, apresentam-se alguns conceitos relativos ao fabrico de circuitos integrados, às técnicas de teste utilizadas neste trabalho como modelos de faltas e à geração automática de vectores de teste;

- No Capítulo 3, Síntese de Circuitos Digitais, faz-se a descrição do fluxo de projecto de circuitos digitais. Os circuitos de referência são disponibilizados no nível de transferência entre registos (RTL) em VHDL, e a preparação do teste é feita no nível lógico com a descrição do circuito em Verilog;

- No Capítulo 4, Metodologias de Preparação do Teste, apresenta-se o método actual e o novo método de teste, e explicam-se os vários passos que os compõem, bem como ferramentas que adicionam funcionalidades a simuladores de faltas já existentes;

- No Capítulo 5, Implementação das Metodologias, apresentam-se as ferramentas comerciais utilizadas neste trabalho, os passos de execução do novo método de teste e do método de teste da AMIS, ao nível de programas utilizados e dos comandos necessários para executar as metodologias consideradas neste trabalho;

- No Capítulo 6, Resultados, apresenta-se um caso prático da aplicação do método de teste proposto e do método de teste utilizado pela AMIS para um circuito, e de seguida apresentam-se os resultados para todos os circuitos utilizados, bem como a análise dos resultados obtidos;

- No Capítulo 7, Conclusões e Trabalhos Futuros, apresentam-se as conclusões finais e considerações sobre o que se pode fazer no futuro para melhorar o método de teste proposto neste trabalho;

2. Técnicas de Teste

Neste capítulo apresenta-se a teoria relativa ao fabrico de circuitos integrados, técnicas de teste de circuitos baseadas em modelos de faltas, técnicas de geração e simulação de vectores de teste, e finalmente faz-se uma breve apresentação das ferramentas comerciais utilizadas para gerar vectores de teste.

Os elementos da tabela periódica denominados como semicondutores foram identificados como os materiais mais indicados para o fabrico de transístores: dispositivos mais pequenos, com menor exigência em termos de energia e mais baratos que os tubos de vácuo ou válvulas. Presentemente, os transístores são utilizados como componentes básicos na construção de portas lógicas.

Actualmente, mono-cristais de silício são utilizados como principal substrato para circuitos integrados, embora algumas composições como arseniato de gálio são utilizados em aplicações especiais como *LED*s, lasers, células solares e circuitos integrados de alta velocidade.

Os circuitos integrados são fabricados por um processo de camadas que inclui estes processos chave:

- fotolitografia (*imaging*)
- deposição (*deposition)*
- corrosão (*etching*)

Estes processos fundamentais são complementados por passos de dopagem de silício, limpeza e planificação.

Como substrato normalmente utilizam-se bolachas (*wafers*) de silício. Fotolitografia é então utilizada para marcar as diferentes áreas (*layers*) do substrato que devem ser dopadas ou onde devem ser colocadas pistas de polisilício, isoladores (SiO_2) ou de metal (Al, Cu). Cada circuito integrado pode ser composto por várias camadas, sendo cada uma definida através deste processo. Umas camadas definem onde os vários dopantes são difundidos no substrato (chamadas de camadas de difusão), outras definem onde iões adicionais são implantados (camadas de implante), outras definem os condutores (camadas de polisilício ou de metal), e outras definem as ligações entre as camadas condutoras (camadas de contacto). Todos os componentes são construídos através de uma combinação específica destas camadas.

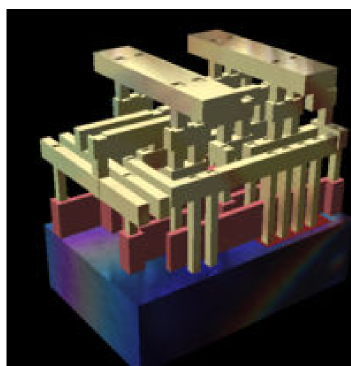

Figura 2. 1 – Exemplo das várias camadas de um circuito integrado

Assim que o processo de fabrico é terminado, os circuitos integrados são sujeitos a uma variedade de testes eléctricos para determinar se estão a funcionar correctamente. A proporção de circuitos na *wafer* que funcionam correctamente é o rendimento do processo de fabrico e é denominada de *yield*.

Os primeiros testes são feitos antes de dividir a *wafer*, colocando pequenas sondas nos nós do circuito, aplicando vectores de teste na entrada e analisando as saídas. O equipamento que faz os testes marca os circuitos defeituosos que depois já não são encapsulados. Estes testes são cobrados pelo tempo que levam a testar, sendo o preço calculado na ordem de cêntimos por segundo. Testes de corrente IDDQ são dos mais caros. Os circuitos normalmente são desenvolvidos tendo em conta a facilidade com que é possível fazer o teste, de modo a acelerar o teste e diminuir custos.

Normalmente considera-se que um defeito que ocorra no processo de fabrico do circuito pode fazer com que certas partes do circuito funcionem incorrectamente. Esta alteração ao comportamento do circuito é modelada recorrendo a modelos de faltas. Por exemplo, se a saída de uma porta lógica apresenta sempre o valor lógico "0", mesmo quando deveria ser "1", então considera-se que há uma falta na saída dessa porta lógica.

2.1. Modelos de faltas

Um defeito pode originar alterações ao funcionamento lógico do circuito de várias formas: uma saída ou entrada de uma porta lógica pode apresentar sempre o valor lógico "0" ou "1", um curto-circuito entre duas pistas de metal fisicamente próximas, ou os transístores que formam a porta lógica podem ter um defeito de fabrico levando ao funcionamento incorrecto dessa porta lógica, entre outros casos.

Para cada um destes problemas há um ou mais modelos de faltas que permitem representar o funcionamento do circuito em questão. É com base nestes modelos de faltas que se faz a geração de vectores de teste que detectem as faltas delineadas pelo modelo de faltas em questão e consequente simulação de faltas de modo a verificar quais as faltas que são detectadas.

Os modelos de faltas estão vocacionados para o tipo de defeitos que conseguem detectar. Por outras palavras, um modelo que detecte uma percentagem elevada de um tipo de faltas pode ser um bom modelo apenas para um tipo de defeitos, enquanto que para outro tipo de defeitos esse modelo já pode não ser suficiente.

Muitas vezes os vectores de teste para um tipo de faltas também conseguem detectar uma percentagem elevada de faltas de outro tipo, mas essa possibilidade depende fortemente da funcionalidade do circuito e deve ser simulado de modo a verificar até que ponto é que de facto acontece.

De seguida apresentam-se os dois modelos de faltas utilizados neste trabalho para o teste em tensão e o modelo usado para o teste em corrente.

2.1.1. Modelo Line stuck-at

Faltas do tipo *line stuck-at*, também denominadas faltas *stuck-at* ou LSA, são faltas em que uma entrada ou saída de uma porta lógica tem um valor lógico fixo, independentemente do valor que realmente deve ter.

Como foi descrito acima, é o caso em que, por exemplo, a saída de uma porta lógica tem sempre o valor lógico "0", mesmo quando deveria ser "1".

Faltas deste tipo são representadas através de dois modelos:

- stuck-at-0, que representa sinais que têm sempre o valor lógico "0", independentemente dos outros sinais que normalmente controlam o nó em questão.

- stuck-at-1, que representa sinais que têm sempre o valor lógico "1", independentemente dos outros sinais que normalmente controlam o nó em questão.

A nível físico, este tipo de faltas corresponde a ter um nó ligado à massa ou à alimentação do circuito, resultando numa falta stuck-at-0 ou stuck-at-1, respectivamente. Outros defeitos que podem ser modelados por este modelo de faltas são os defeitos que interrompem totalmente o caminho de *pull-up* ou *pull-down* dos nós lógicos.

A Fig.2.2 mostra um exemplo do modelo stuck-at-0. Para o modelo stuck-at-1 é a situação complementar.

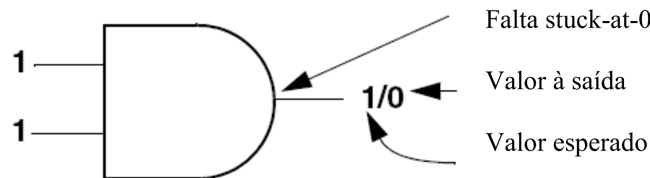

Figura 2. 2 – Porta lógica AND com falta stuck-at-0 na saída

De modo a que uma falta LSA possa ser detectada é necessário que o nó do circuito (uma entrada ou uma saída) possa ser controlado e observado.

Um nó pode ser controlado se colocando um conjunto de valores lógicos nas entradas primárias do circuito é possível forçar um valor lógico nesse nó. Uma entrada primária é uma entrada que pode ser directamente controlada pelo ambiente de teste.

Um nó pode ser observado se é possível observar mudanças nas saídas primárias conforme o valor lógico desse nó é alterado. Uma saída primária é uma saída que pode ser directamente observada pelo ambiente de teste.

A Fig.2.3 mostra um exemplo em que um nó é passível de ser controlado e observado.

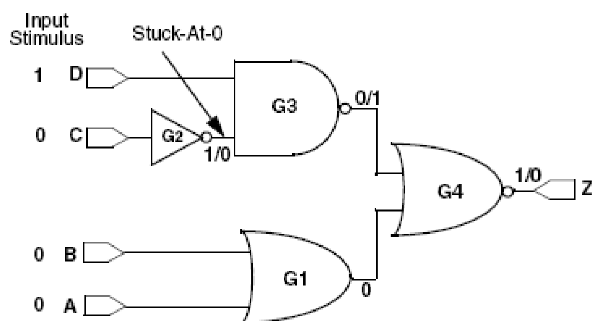

Figura 2. 3 – Exemplo de um nó que se pode controlar e observar

Na Fig.2.3, representa-se um circuito em que a saída do inversor G2 tem um falta do tipo stuck-at-0, forçando esse nó a ter o valor “0” permanentemente.

É possível controlar este nó porque colocando um valor lógico numa entrada primária, C neste caso, é possível forçar um valor lógico nesse nó.

Este nó também é observável, porque neste exemplo a diferença verificada no nó também é observável na saída do circuito Z. Em vez da saída de G3 ser “0”, é “1” e G4 em vez de ter “1” à saída, tem “0”. Como a saída de G4 é uma saída primária, é possível observar que há uma diferença no funcionamento do circuito (num circuito sem falta deveria ter o valor “1” em vez de “0”) e assim a falta stuck-at-0 no nó de saída de G2 é detectável.

2.1.2. Modelo *Bridging*

Este modelo de faltas descreve o comportamento de faltas em que um nó do circuito influencia o valor lógico de outro nó do circuito, devido a uma ligação resistiva entre os dois. Este tipo de faltas normalmente ocorre entre nós fisicamente muito próximos entre si no *layout* do circuito.

Quando uma falta *bridging* é activada há um nó agressor que vai forçar o valor lógico de outro nó, chamado nó vítima.

A Fig.2.4 ilustra duas possíveis situações com este modelo de faltas. Faltas *bridging* têm dois tipos, ba0 quando o nó agressor está a “0” e ba1 quando o nó agressor está a “1”.

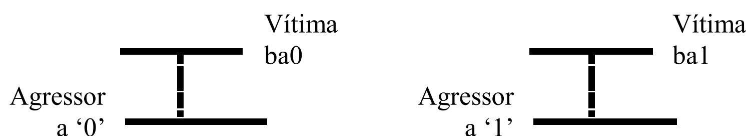

Figura 2. 4 – Modelos de faltas *bridging* ba0 e ba1

Uma falta ba0 é considerada detectável se a falta stuck-at-0 associada ao nó vítima é detectada ao mesmo tempo que o nó agressor tem o valor “0”. Considera-se que o nó agressor não tem faltas associadas.

De modo semelhante, uma falta ba1 é considerada detectável se a falta stuck-at-1 associada ao nó vítima é detectada ao mesmo tempo que o nó agressor tem o valor “1”, sem faltas associadas. Mais uma vez, considera-se que o nó agressor não tem faltas associadas

Assim, o modelo de faltas do tipo *bridging* pode ser encarado como o modelo LSA condicionado a valores no nó agressor. É para maximizar a probabilidade de satisfazer as condições necessárias à detecção de faltas *bridging* que a AMIS recorre actualmente à detecção múltipla de faltas LSA.

2.1.3. Modelo IDDQ

Este modelo de faltas permite detectar defeitos de fabrico que podem levar ao mau funcionamento imediato ou a um tempo de vida e fiabilidade limitados do circuito. É um facto que circuitos CMOS com corrente de fuga (*leakage current*) elevada são de facto defeituosos.

Os testes para este tipo de faltas são baseados na corrente durante um intervalo de tempo, e não na tensão num determinado instante, como as faltas LSA. A corrente de repouso (IDDQ) do dispositivo que está a ser testado é monitorizada enquanto se excitam as entradas do dispositivo e caso se verifique que a corrente de repouso se mantém elevada, durante uma situação estável, significa que há um defeito. Numa situação ideal, as correntes de fuga devem ser desprezáveis embora nas tecnologias actuais, devido à condução sub-limiar, já sejam consideráveis e dificultem a definição de limites para a rejeição de dispositivos. Para resolver este problema utiliza-se detecção IDDQ probabilística [5].

As Fig.2.5 a) e b) mostram o caso em que um transístor de uma porta lógico inversora está defeituoso e qual o efeito deste defeito nos valores da corrente de repouso.

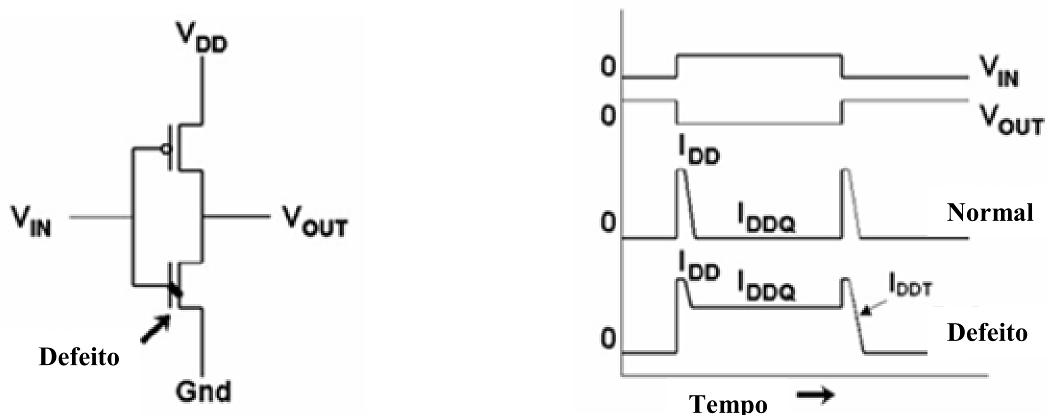

Figura 2. 5 – a) Inversor com transístor defeituoso b)Valores de IDDQ para um transístor bom e para um defeituoso

Com a tecnologia actual, e como os testes IDDQ requerem que se aguarde que termine o efeito do transitório decorrente da mudança de vectores, testes com base neste modelo de faltas são bastante mais dispendiosos que os testes em tensão. Normalmente um conjunto de 10 a 20 vectores de teste é utilizado para teste de um circuito, com um nível de detecção de faltas aceitável [8].

2.2. Geração automática de vectores de teste – ATPG

De modo a preparar o teste de circuitos integrados utiliza-se *software* próprio para o efeito, como o Tetramax® da Synopsys®.

Tendo como base a estrutura lógica do circuito e um modelo de faltas seleccionado é possível gerar vectores de teste automaticamente através do que se denomina ATPG – *Automatic Test Pattern Generation*.

O que a ATPG faz é gerar automaticamente vectores de teste que detectem o maior número possível de faltas cujo modelo se está a utilizar. Estes vectores de teste são utilizados mais tarde em equipamento próprio de teste de circuitos (ATE – *Automatic Test Equipment*) para testar os circuitos recém-fabricados.

O Tetramax® disponibiliza três modos de ATPG:

- *Basic scan*, um modo combinatório eficiente para descrições *full-scan*
- *Fast-sequential*, para suporte limitado de descrições *partial-scan*
- *Full-sequential*, para obter a máxima cobertura de faltas para descrições *partial-scan*

De notar que a versão do Tetramax® utilizada não permite a ATPG para faltas *bridging* no modo *Full-sequential* (não sendo atingida a cobertura máxima de faltas) e a simulação de faltas BRI não permite a simulação de vectores de teste *Full-sequential* (mais faltas *bridging* podiam ser consideradas detectadas por vectores LSA) [8].

Embora não venha referenciado no manual de utilizador [8], a simulação de faltas IDDQ utilizando o método *Full-sequential* também não é permitido, embora os vectores de teste LSA obtidos através de ATPG *Full-sequential* sejam simulados através do método PROOFS: um simulador de faltas para circuitos sequenciais rápido e eficiente em termos de memória, não havendo assim as limitações que existem na simulação de faltas *bridging*. A ATPG IDDQ já permite o modo *Full-sequential*.

2.3. Conclusões

A fim de permitir reduzir a complexidade do problema da preparação do teste dos circuitos digitais, recorre-se a modelos de faltas que representam um funcionamento diferente do correcto em níveis de abstracção superiores.

O modelo LSA é o modelo tradicionalmente usado na preparação do teste, mas as novas tecnologias submicrométricas e os crescentes níveis de exigência impõem a utilização de modelos de faltas mais próximos do nível físico.

3. Síntese de Circuitos Digitais

Neste capítulo descreve-se o procedimento utilizado para fazer a síntese dos circuitos de teste utilizados neste trabalho.

3.1. Síntese lógica com scan

Para que o Tetramax® reconheça correctamente todos os nós e portas lógicas do circuito em análise é necessário que este esteja descrito de modo estrutural e não comportamental, ou seja, todos os componentes do circuitos têm de ser definidos como blocos funcionais e com ligações entre si, em vez de se utilizarem expressões baseadas nos comandos "assign" ou "trans" ("assign c = a+b;" para definir uma porta OR de 2 entradas, por exemplo). Expressões do tipo "assign a=b;" não causam problemas.

Como os circuitos de referência utilizados neste trabalho são disponibilizados no nível de transferência entre registos (RTL) na linguagem de descrição de circuitos VHDL, e como o simulador Verilog-XL® necessita de uma descrição do circuito em Verilog, é então necessário obter uma descrição estrutural em Verilog destes circuitos. Para tal, utiliza-se o programa Design Vision® e as bibliotecas de tecnologia da AMS® de modo a fazer a síntese do circuito com uma biblioteca de tecnologia e assim obter uma descrição estrutural do circuito.

As bibliotecas de tecnologia são descrições de portas lógicas e outros módulos funcionais (multiplexers, contadores, etc.). Estas descrições definem atributos como número de entradas e saídas, tempos de propagação, área que ocupa e funcionamento para uma determinada tecnologia. A tecnologia normalmente tem o nome da largura dos transístores utilizados nessa tecnologia: c18 para 0.18µm e c35 para 0.35µm, por exemplo.

Como os circuitos utilizados são para referência, ou seja, são apenas utilizados para fazer testes, não têm como objectivo fazer uma funcionalidade útil. Tendo isto em conta, quando se faz a síntese do circuito no Design Vision® optou-se por evitar ao máximo possível as optimizações de área e de portas lógicas, de modo que para um dado circuito se teste o maior número de faltas possível e que as portas lógicas visíveis ao Tetramax® sejam as mesmas que o simulador Verilog-XL® consegue observar.

Para preparar o teste de circuitos sequenciais no Tetramax® é necessário fornecer a informação relativa ao relógio e cadeias de *scan* existentes no circuito em questão. O Design Vision® permite guardar esta informação automaticamente num ficheiro de formato STIL com extensão ".spf", que pode ser lido pelo Tetramax® mais tarde na altura do DRC (*Design Rule Checking*).

Para todos os circuitos testados utiliza-se como relógio de sistema um relógio com forma de onda quadrada, com *duty cycle* de 50% e com período de 100 unidades de tempo. A unidade de tempo é definida na biblioteca de tecnologia utilizada para sintetizar o circuito, que no caso da utilizada neste trabalho, c35, está definida como 1 ns (tempo de referência) e 1 ps (tempo de precisão).

De forma a utilizar o formato de base de dados antigo, o que permite utilizar comandos do programa que de outro modo não são permitidos, como o "insert_scan" que adiciona automaticamente cadeias de *scan* (*scan chains*) ao circuito sequencial, é necessário iniciar o Design Vision® com a opção "–db_mode".

Para fazer a ATPG de faltas para circuitos sequenciais de modo a obter a melhor cobertura de faltas, é necessário fazer a inserção de cadeias de *scan*, porque com circuitos sequenciais não é possível controlar e observar todos os nós do circuito. A inserção de cadeia de *scan* consiste em inserir registos em cada porta lógica do circuito sequencial, o que permite controlar e observar as entradas e saídas dessas portas lógicas, respectivamente, aumentando o nível de controlo e de observação das faltas do circuito e por consequência a cobertura de faltas do circuito

Há vários métodos de inserção de cadeias de *scan*, e o utilizado neste trabalho é o "Multiplexed Flip-Flops" (predefinição do Design Vision®), que utiliza como registos flip-flops controlados por multiplexers, que numa posição permite que o circuito funcione normalmente e noutra posição faz com que as portas lógicas utilizam os valor presentes nos registos. As Fig.3.1 e 3.2 representam o conceito de inserção de *scan*: a primeira representa a ideia geral e a segunda representa o método de inserção utilizado.

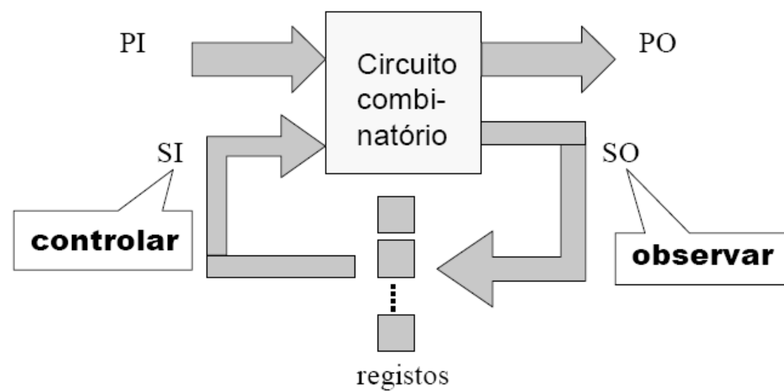

Figura 3. 1 – Ideia geral da inserção de *scan*

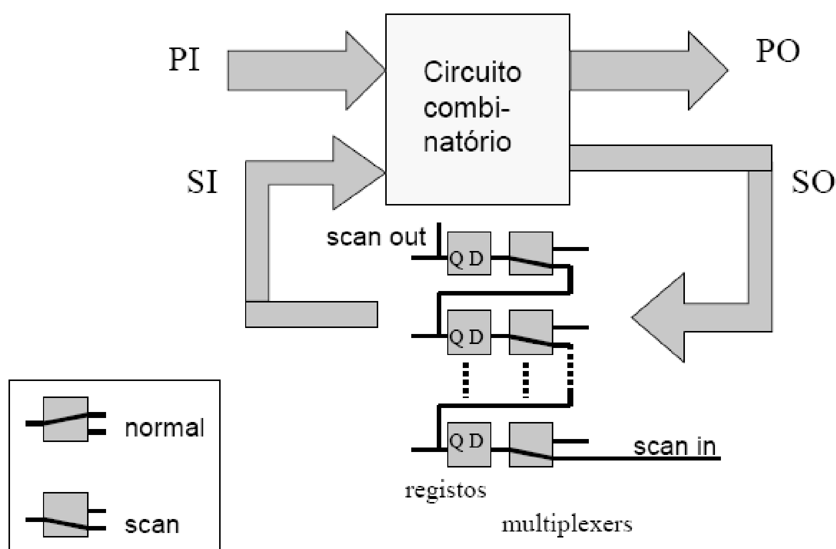

Figura 3. 2 – Método "Multiplexed Flip-Flops"

A nível de descrição do circuito, são utilizados modelos de flip-flops próprios para a cadeia de *scan*, normalmente com mais alguns pinos de entrada e/ou saída do que os modelos de portas lógicas de um circuito sem cadeia de *scan*, pinos estes que permitem mudar o modo de funcionamento de normal para modo de *scan* e vice-versa.

Uma vantagem da inserção de cadeia de *scan* num circuito sequencial, além do aumento da cobertura de faltas, é que no modo funcionamento normal a cadeia de *scan* é praticamente invisível, no sentido de que o circuito tem a mesma funcionalidade caso tenha ou não cadeia de *scan*, embora que a inserção da cadeia de *scan* aumenta a área de circuito, possivelmente influencia o caminho crítico do circuito, e faz com que seja necessário aplicar uma frequência de relógio mais elevada à cadeia de *scan*, comparativamente à frequência do relógio de sistema. O modo de *scan* do circuito só é utilizado quando se testa o circuito para detecção de faltas.

Para fazer a síntese do circuito com as considerações indicadas acima, utiliza-se um ficheiro com a descrição dos comandos a utilizar. Este ficheiro é chamado de *script*, e para o Design Vision® o poder reconhecer é necessário que tenha a extensão ".tcl".

O uso de um *script* facilita a compilação e síntese de um circuito, pois basta apenas indicar os ficheiros com a descrição do circuito, qual a entrada que é o relógio e o nome dos ficheiros de saída (descrição do circuito em Verilog, base de dados, descrição do relógio e cadeias de *scan*) e o resto é predefinido e automático.

De notar que mesmo utilizando um *script* continua a ser necessário rever as mensagens que o programa desenvolve para verificar se é necessário fazer alguma correcção aos comandos utilizados ou mesmo à descrição do circuito.

A partir da próxima versão do Design Vision® (2007.03) muitos comandos poderão deixar de funcionar, passando a ter nova denominação, e assim o *script* apresentado abaixo pode deixar de funcionar correctamente.

A Synopsys® disponibiliza uma ferramenta que faz a conversão do formato antigo para o novo: "db2xg".

3.2. Exemplo de síntese lógica

Um exemplo genérico de *script* e a explicação de cada comando são feitos de seguida.

- Começa-se por fazer a análise do modelo do circuito em VHDL. Esta análise faz uma compilação do modelo do circuito e verifica se o código de descrição do circuito é sintetizável.

analyze -library WORK -format vhdl {*circuito*.vhd}

- Após a análise do modelo, faz-se a elaboração da descrição do circuito, fazendo uma pré-síntese do modelo analisado e sobretudo identificando os registos que vão ser necessários.

elaborate *top_module* -architecture BEHAVIOR -library WORK

- Conforme indicado pela AMS, deve-se definir o seguinte comando nesta altura:

set_fix_multiple_port_nets -all

Este comando faz com que problemas relacionados com *nets* ligadas a várias entradas ou saídas de portas lógicas sejam resolvidos automaticamente pelo Design Vision®.

- De seguida definem-se as condições de funcionamento do circuito: indica-se qual o pino do relógio, o período e o *duty cycle*, e quais as condições de funcionamento do circuito (indústria, militar, melhor comportamento, pior comportamento), que no caso deste trabalho são o pior comportamento, para indústria. Nestas condições de funcionamento, considera-se que a temperatura tem o valor de 85ºC, que a tensão tem o valor de 3V (o processo C35 é um processo 3,3V) e que se trata de um processo lento.

create_clock -name "clock" -period 100 -waveform { 0 50 } { *CLOCK* }
set_operating_conditions -library c35_CORELIB WORST-IND

Com alguns circuitos é necessário incluir os seguintes comandos neste ponto, pois sem eles o Design Vision® faz optimizações que mais tarde dão problemas com a ATPG e simulação de faltas. Estes comandos evitam que registos constantes sejam substituídos por curto-circuitos à massa ou à alimentação, consoante o caso.

set compile_seqmap_propagate_constants false
printvar compile_seqmap_propagate_constants

- Com isto, faz-se a compilação do circuito. Esta compilação é dependente da tecnologia que é definida quando se inicia o programa, e o que faz é compor o mapeamento e optimização da área do circuito com base nas portas lógicas e outros módulos funcionais disponíveis pela tecnologia. Deste modo fica-se com uma descrição estrutural do circuito que pode ser utilizada correctamente pelo Tetramax®.

De modo a evitar ao máximo a existência de "assign" e "trans" e ao mesmo tempo evitando optimizações de lógica (por exemplo, 3 XOR passarem a ser representados por 1 XOR maior, mantendo a funcionalidade mas perdendo a informação de ligações internas, o que se quer evitar para não perder pontos de observação e controlo de faltas), escolhe-se o esforço de mapeamento médio (mapeamento *low* é considerado obsoleto), esforço de optimização de área nulo e apenas correcção de regras de desenho do circuito.

uplevel #0 compile -map_effort medium -area_effort none -only_design_rule

- De seguida faz-se a inserção da cadeia de *scan*, adicionando multiplexers aos flip-flops do circuito, o que vai permitir que se controle e observa as entradas e saídas desses flip-flops, aumentando assim a cobertura de faltas que se vai obter neste circuito. Primeiro pede-se uma previsão da cadeia de *scan*, insere-se automaticamente a cadeia de *scan* e depois verifica-se se não há problemas com as células que compõem a cadeia.

preview_scan
insert_scan
check_scan

- De modo a que se possa voltar a trabalho com este circuito neste programa sem ter de repetir os passos anteriores, guarda-se toda a informação até agora reunida pelo programa. Mais tarde, basta carregar o ficheiro gerado por este comando para se começar a trabalhar a partir deste ponto.

write -hierarchy -format db -output *circuito*_scan.db

- Com a cadeia de *scan* inserida, faz-se uma estimativa da cobertura de faltas:

estimate_test_coverage -sample 100

A opção "-sample N" serve para indicar a percentagem N de faltas que se deve ter em conta quando se faz a estimativa. Esta estimativa é feita com base em dados estatísticos, o que permite ter uma ideia sobre os resultados que se vão obter no Tetramax® mais tarde. Esta ATPG estatística utiliza os mesmos algoritmos determinísticos que a ATPG standard para calcular a cobertura de faltas, mas pode ser mais rápida pois não guarda os padrões de teste, e pode processar apenas uma certa percentagem de faltas para fazer a previsão da cobertura de faltas.

- Faz-se a correcção dos nomes das entradas, saídas e ligações internas do circuito com as regras predefinidas do programa para a linguagem de descrição Verilog, e depois grava-se a descrição estrutural do circuito para ficheiro com o formato Verilog.

change_names -hierarchy -rules verilog –verbose

No fim do *output* deste comando deve-se verificar se existe algum *warning* relacionado com "assign" ou "trans". Caso haja, é necessário voltar atrás e fazer a compilação com outras opções de modo a evitar a existência destes comandos, pois em certos casos podem causar erros na ATPG ou em simulações de faltas mais tarde.

write -hierarchy -format verilog -output *circuito*_scan.v

- Finalmente, guarda-se a informação das entradas, saídas e pinos de *enable* da cadeia de *scan*, e a descrição do comportamento do relógio no formato STIL para ficheiro. Este ficheiro é necessário para o DRC do circuito no Tetramax®.

check_scan
write_test_protocol -format stil -out *circuito*_scan.spf

É necessário fazer o "check_scan" antes de escrever o ficheiro STIL, senão o comando "write_test_protocol" dá erro.

Para utilizar este *script* genérico para um circuito, basta mudar o "*circuito*" para o nome do circuito em teste (b04, por exemplo), o "*top_module*" para o nome do módulo principal do circuito (B04, por exemplo) e o nome do pino do relógio de sistema de "*CLOCK*" para o nome utilizado no circuito (CLOCK, por exemplo).

A título informativo, caso se queira ver a informação detalhada da cadeia de *scan* basta executar este comando no fim do *script*:

report_test -scan_path

3.3. Conclusão

Os requisitos de testabilidade (observabilidade e controlabilidade) têm que estar presentes desde a fase inicial da síntese lógica. Estes requisitos são normalmente satisfeitos se todos os registos do circuito forem substituídos por registos com *scan* ligados em cadeia.

As ferramentas de síntese comerciais, como o Design Vision®, permitem inserir automaticamente a cadeia de *scan*.

4. Metodologias de Preparação do Teste

Neste capítulo descreve-se o método de teste actualmente em uso na AMIS, e o método proposto neste trabalho. Explicam-se os vários passos que compõem estes métodos, bem como ferramentas que adicionam funcionalidades a simuladores comerciais já existentes.

4.1. Método de teste actual

A AMI Semiconductor, tal como outras empresas da área, utiliza um método de teste, ou método de geração de vectores de teste, que consiste na detecção múltipla de faltas (*N-detection*) LSA, na ATPG IDDQ com poucos vectores de teste como resultado (com a tecnologia actual um conjunto de 10 a 20 *patterns* de teste é o intervalo de valores comum), e não se faz ATPG nem simulação de faltas para mais nenhum modelo de faltas. De modo obter um número reduzido de vectores de teste (um pattern pode ser composto por um ou mais vectores de teste), faz-se a simulação de faltas IDDQ com vectores de teste LSA antes de fazer a ATPG para faltas IDDQ.

Considera-se neste trabalho como método de teste utilizado pela AMIS o seguinte procedimento:

- ATPG LSA com detecção múltipla de faltas (cinco vezes cada falta)
- Simulação de faltas BRI com os vectores de teste LSA gerados no passo anterior
- ATPG LSA sem detecção múltipla de faltas, simulação de faltas IDDQ com vectores de teste LSA (com detecção simples) e ATPG para as faltas IDDQ não detectadas por vectores de teste LSA
- Simulação de faltas BRI com os vectores de teste IDDQ gerados no passo anterior a este, e simulação de faltas BRI com os vectores de teste LSA gerados no passo anterior, de modo a determinar o nível de detecção de faltas BRI.

As simulações feitas para testar o nível de detecção de faltas BRI são feitas apenas a título de comparação com o novo método de teste, num caso real as simulações de faltas BRI não são necessárias.

De notar também que para realizar testes com esta metodologia utiliza-se uma lista de faltas *bridging* gerada de modo aleatório, uma vez que as ferramentas utilizadas neste trabalho não permitem extrair informação relacionada com capacidades parasitas do *layout* e, consequentemente, obter uma lista de faltas *bridging* real. Num caso real, a lista de faltas *bridging* utilizada é uma lista de faltas reais.

A Figura 4.1 representa os passos indicados acima.

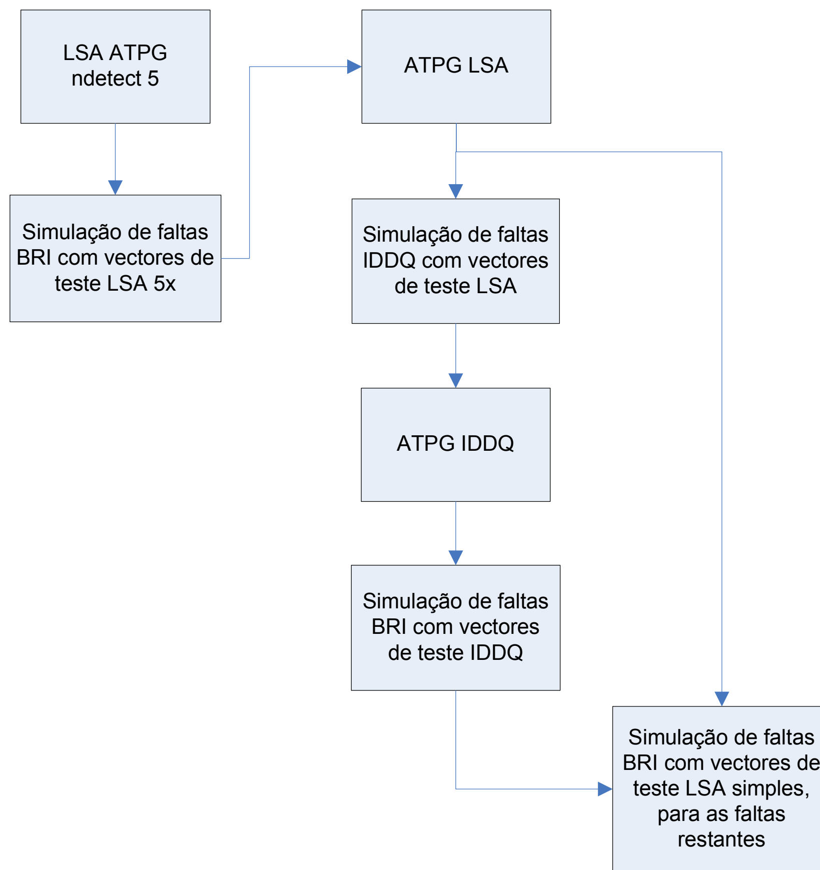

Figura 4. 1 – Método de teste utilizado pela AMIS

4.2. *Metodologia proposta*

De modo a criar um novo método de teste optimizado escolheram-se dois modelos de faltas em tensão: LSA e BRI, e um modelo de faltas em corrente: IDDQ. Este conjunto de modelos de faltas, de acordo com a AMIS, permite detectar a maioria dos defeitos de fabrico que podem ocorrer com as tecnologias actuais.

De forma a criar um conjunto de vectores de teste reduzido é necessário testar quais as faltas de um certo tipo que são detectadas por vectores de teste gerados anteriormente para outros tipos de faltas, fazer ATPG com compactamente (*merge*) de vectores de teste, e usar a ordem com que se utiliza os modelos de faltas para reduzir a lista de faltas (por exemplo, fazendo ATPG LSA, simulação de faltas IDDQ com vectores de teste LSA e ATPG IDDQ faz com que se obtenha um número reduzidos de faltas IDDQ).

Depois de estudar o método de teste da AMIS, decidiu-se que o novo método de teste deve primeiro utilizar o modelo LSA, depois o IDDQ e finalmente o BRI, fazendo simulação de faltas no modo descrito acima.

A Figura 4.2 apresenta a ordem da geração de vectores e de simulação de faltas considerada para estudo.

O novo método de teste tem uma série de passos principais, que consistem no seguinte:

- ATPG de faltas LSA, simulação de faltas IDDQ com vectores de teste LSA e ATPG para as faltas IDDQ restantes, no Tetramax®. Utiliza-se “S2I” (*Stuck-at to Iddq*) para denominar este passo.
- Geração aleatória e automática de faltas *bridging* no Verilog-XL®, já que as ferramentas utilizadas neste trabalho não permitem obter uma lista de faltas BRI real. Utiliza-se “VER_RAND” (*Verilog Random*) para denominar este passo.
- Simulação de faltas *bridging* com vectores de teste LSA, no Tetramax®. Utiliza-se “S2B” (*Stuck-at to Bridging*) para denominar este passo.
- Simulação de faltas *bridging* com vectores de teste IDDQ, no Verilog-XL®. Utiliza-se “I2B” (*Iddq to Bridging*) para denominar este passo.
- ATPG de faltas *bridging*, no Tetramax®. Utiliza-se “ATPG-BRI” para denominar este passo.

De notar que o passo VER_RAND num caso real não é necessário, pois a lista de faltas BRI que se utiliza é uma lista de faltas real. No caso deste trabalho, como as ferramentas utilizadas não permitem extrair as capacidades parasitas do *layout*, não é possível obter uma lista de faltas BRI real.

A descrição de cada um destes passos e das opções tomadas é feita de seguida.

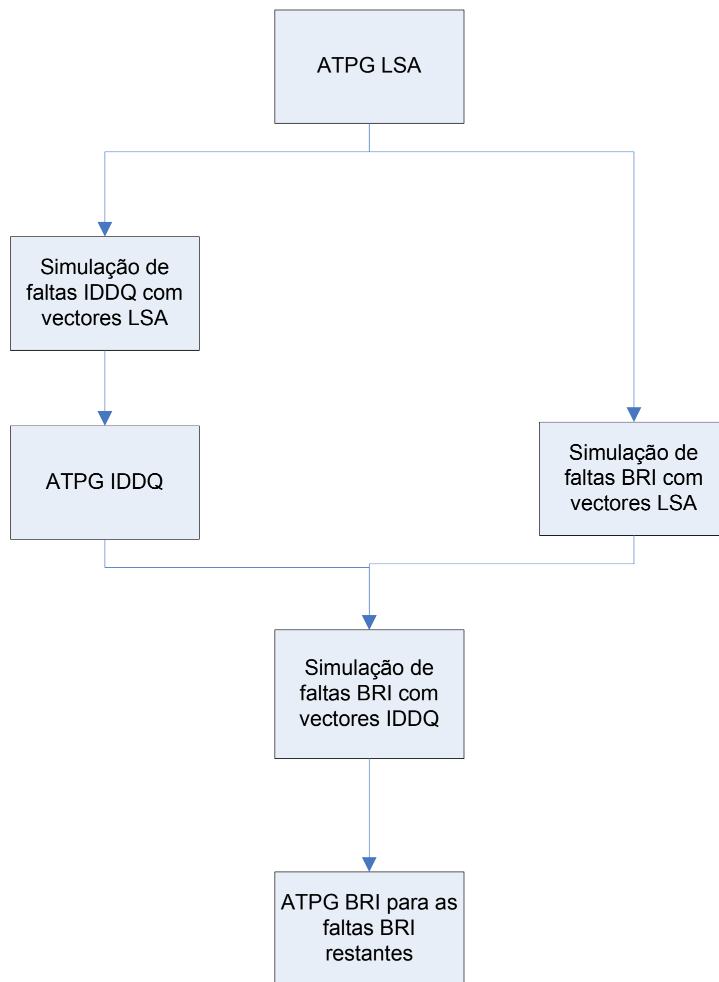

Figura 4. 2 – Método de teste proposto

4.2.1. ATPG de faltas LSA, simulação de faltas IDDQ com vectores de teste LSA e ATPG para as faltas IDDQ restantes

Nesta parte do método de teste utiliza-se o Tetramax® para fazer a ATPG para faltas LSA, simulação de faltas IDDQ com vectores de teste LSA e ATPG para as faltas IDDQ restantes.

Todas as faltas LSA do circuito são adicionadas à lista de faltas do programa e de seguida é feita a ATPG. Todos os vectores de teste são compactados (*merge*) o máximo possível, de modo a minimizar o número de vectores de teste obtidos. O Tetramax® começa por fazer a detecção de faltas através do modo *Basic scan*, e caso seja necessário passa para métodos de ATPG mais avançados como o *Fast-sequential* e *Full-sequential*, necessários para obter a cobertura de faltas mais elevada possível para um circuito sequencial.

Os vectores de teste obtidos são guardados para ficheiro, de modo a poderem ser utilizados mais tarde tanto para simulação de faltas como para testes com equipamento de teste automático (ATE). Os ficheiros são guardados no formato binário do Tetramax®, para quando for necessário fazer simulação de faltas com este vectores, e como *testbench* em Verilog no formato *tables*, que é uma *testbench* definida como um máquina de estados constituído por vários ficheiros. O Tetramax® não permite a leitura de vectores de teste a partir de *testbench* em Verilog, por isso utiliza-se o formato binário do programa para esse propósito.

De seguida, faz-se a simulação de faltas IDDQ, para todas as faltas IDDQ do circuito, com os vectores de teste LSA gerados anteriormente. Deste modo as faltas IDDQ detectadas pelos vectores de teste LSA são retiradas da lista de faltas do programa.

Finalmente, faz-se a ATPG para as faltas IDDQ restantes e guardam-se os vectores de teste para faltas IDDQ em ficheiro. Os formatos utilizados são os seguintes: formato binário do Tetramax®, o de *testbench* em Verilog, no formato *tables*, pelas mesmas razões pelas quais se guardam os vectores LSA neste formato, e também no formato *single file*, que vai ser necessário para o passo I2B. De notar que com qualquer das duas *testbench* é possível fazer a simulação de faltas em I2B, mas a em *single file* é mais fácil de analisar e de modificar como um todo.

Neste caso, a ATPG para faltas LSA e IDDQ é feita com o modo *Full-sequential* (que inclui os modos *Basic scan* e *Fast-sequential*), e na simulação de faltas IDDQ embora não se possa utilizar o modo *Full-sequential*, utiliza-se o PROOFS para se poder fazer a simulação de faltas com todos os vectores LSA, e assim reconhecer o maior número de faltas IDDQ possíveis de detectar com vectores LSA.

No Capítulo 5 faz-se a descrição detalhada do *software* necessário para concretizar esta metodologia.

4.2.2. Geração aleatória e automática de faltas bridging

Neste passo faz-se a geração aleatória de faltas BRI com o Verilog-XL®, uma vez que o *software* utilizado neste trabalho não permite obter uma lista de faltas BRI real. Na metodologia proposta à AMIS este passo é substituído pela leitura de uma lista de faltas BRI obtida com base nas capacidades parasitas do *layout*.

O simulador Verilog-XL® só por si não faz o que é necessário para gerar as faltas BRI aleatoriamente ou a detecção de faltas no passo I2B. De modo a adicionar estas funcionalidades utiliza-se a PLI e funções em C próprias para o efeito.

Neste passo do trabalho apenas é necessário inicializar as estruturas necessárias ao programa, que é um passo comum a todos os modos de funcionamento do simulador, fazer a geração aleatória de faltas BRI e escrever a lista de faltas para ficheiro.

O número de faltas BRI depende directamente do número de faltas LSA do circuito. Na chamada de sistema $iddq_init indica-se o número de faltas LSA e o número por que se deve multiplicar o número de faltas LSA, no segundo e terceiro argumento respectivamente. Seguidamente o programa gera faltas BRI aleatoriamente até se obter o número pedido de faltas, ou o número máximo possível de faltas caso o número pedido seja superior a este último.

O número máximo de faltas BRI depende do número de nós do circuito, e é dado calculando o número de combinações de N, 2 a 2, e multiplicando este valor por 2 (uma falta pode do tipo ba0 ou ba1). Sendo N o número de nós, o número máximo de faltas de um circuito pode ser obtido através da equação (1).

$$({}^{N}C_{2}) \times 2 \qquad (1)$$

De notar que a falta BRI entre os nós genéricos “a” e “b” só é considerada caso a falta entre “b” e “a” não exista. Com isto em mente é possível deduzir a equação (1).

Faltas com nós denominados SYNOPSYS_UNCONNECTED_*número* (nós ligados a lado nenhum) também não são consideradas para faltas *bridging*, uma vez que no Tetramax® não é possível o seu processamento.

Neste passo do teste é necessário alterar a *testbench single file* com vectores de teste IDDQ e o ficheiro de descrição de circuito de modo a funcionarem correctamente com o Verilog-XL®.

No “event IDDQ” da testbench é necessário adicionar fora do bloco “if” a chamada de sistema $iddq_bri que executa a função que detecta quais as faltas detectadas no momento em que a função é chamada, ou seja, sempre que se faz uma medição IDDQ.

Dentro do bloco “initial” também é necessário colocar a chamada de sistema $iddq_init logo antes do bloco “if” com as definições IDDQ do Tetramax®. Esta chamada de sistema vai chamar a função que inicializa as estruturas necessárias ao programa.

Finalmente, antes da chamada $finish da *testbench* coloca-se a 3ª e última chamada de sistema PLI: $iddq_close. Esta função escreve para ficheiro a lista de faltas BRI que não foram detectadas e limpa as estruturas utilizadas.

De notar que a *testbench* em verilog gerada pelo Tetramax® já contém código específico para simulação de faltas IDDQ, mas como estas instruções são específicas para o programa Synopsys® PowerFault e o programa não é utilizado neste trabalho, decidiu-se ignorá-las e colocar instruções PLI feitas para o efeito.

No início do ficheiro com a descrição do circuito é necessário adicionar a definição "timescale", que está no início da *testbench*.

No Capítulo 5 faz-se a descrição detalhada dos procedimentos necessários para concretizar este passo.

No Capítulo 6 é apresentado um exemplo da utilização das chamadas de sistema PLI para o caso do circuito b18 e em Anexo apresentam-se as modificações feitas na *testbench* e no ficheiro de descrição do circuito em questão.

4.2.3. Simulação de faltas bridging com vectores de teste LSA

Nesta parte do método de teste faz-se a simulação de faltas BRI com os vectores de teste LSA gerados em S2I. Desta maneira é possível eliminar da lista de faltas BRI as que são detectadas por vectores de teste LSA e assim evita-se gerar vectores de teste que detectem faltas BRI já detectadas por vectores de outro tipo de faltas.

Utilizando o Tetramax® novamente, começa-se por ler o ficheiro com a lista de faltas *bridging*, gerado no passo VER_RAND.

Após a leitura da lista de faltas, é possível começar a fazer a simulação de faltas BRI com vectores de teste LSA. Para tal usa-se o ficheiro no formato binário do Tetramax® guardado no passo S2I. Neste caso, a simulação de faltas BRI não utiliza o modo *Full-sequential*, por limitação do Tetramax®, o que faz com que *patterns* (conjuntos de um ou mais vectores de teste) LSA obtidos através do modo ATPG *Full-sequential* não sejam utilizados na simulação de faltas, e assim a detecção de faltas BRI com vectores de teste LSA não é tão boa como poderia ser caso a simulação *Full-sequential* fosse permitida.

Após a simulação de faltas, a lista das faltas *bridging* restantes é guardada para ficheiro no formato do Tetramax® apresentado de seguida:

Tipo de falta	Código de estado de detecção	Localização do nó vítima	Localização do nó agressor
ba0	NC	nodeA	nodeB
ba1	NC	nodeA	nodeB
ba0	NC	nodeB	nodeA
ba1	NC	nodeB	nodeA

Este ficheiro pode ser lido pelo simulador Verilog-XL® no passo I2B, graças a funções em C criadas para o efeito.

No Capítulo 5 faz-se a descrição detalhada dos procedimentos necessários para concretizar esta metodologia.

4.2.4. Simulação de faltas bridging com vectores de teste IDDQ

Nesta secção descreve-se a utilização dos vectores de teste para faltas IDDQ obtidos com o Tetramax® para fazer a simulação do circuito no Verilog-XL® com verificação de detecção de faltas do tipo *bridging*.

Enquanto se faz a simulação do circuito faz-se a análise dos valores lógicos dos nós presentes na lista de faltas do tipo *bridging*, removendo da lista as faltas detectadas pelos vectores de teste IDDQ. As faltam podem ser adicionadas tanto por leitura do ficheiro de faltas no formato Tetramax® anteriormente referido como por inserção aleatória de faltas.

Esta parte é feita no Verilog-XL® e não no Tetramax® porque com o primeiro faz-se a simulação em corrente, enquanto que o Tetramax® faz a simulação em tensão, e o que se pretende é fazer a simulação em corrente dos vectores de teste IDDQ.

Neste passo do método de teste utiliza-se novamente o simulador de faltas Verilog-XL®, com as funcionalidades adicionadas através de PLI.

Relativamente à parte VER_RAND, a única alteração necessária na *testbench* é alterar o modo de utilização, e é explicada em detalhe no Capítulo 5.2.2.

Neste modo de utilização, sempre que há o evento "iddq_capture" (momento em que se faz uma medição IDDQ) a chamada de sistema $iddq_bri é executada, fazendo a detecção e eliminação de faltas *bridging* como é explicado anteriormente. De notar que os últimos vectores de teste normalmente não usam o evento "iddq_capture". Estes vectores são necessários para que as faltas possam ser observáveis nas saídas do circuito.

Após acabar a simulação para todos os vectores de teste e de verificar todas as faltas para testar se foram detectadas ou não, a lista de faltas *bridging* resultante é escrita para ficheiro, no formato de comandos de inserção de faltas *bridging* do Tetramax®.

No Capítulo 5 faz-se a descrição detalhada dos procedimentos necessários para concretizar esta metodologia.

4.2.5. ATPG de faltas bridging

Neste passo final do novo método de teste, utiliza-se o Tetramax® para fazer a ATPG para as faltas BRI que não são detectadas pelos vectores de teste LSA nem pelos vectores IDDQ.

Neste caso, a ATPG para faltas BRI não é feita com o modo *Full-sequential* por limitação do Tetramax®. Assim, só são utilizados os modos *Basic scan* e *Fast-sequential*, o que faz com que a detecção de faltas BRI não seja tão boa como poderia ser caso o modo ATPG *Full-sequential* fosse permitido.

Após a ATPG guardam-se os vectores de teste obtidos para ficheiro, tanto no formato binário como no formato de *testbench* em Verilog *tables*, tal como se faz para os vectores de teste LSA no passo S2I e pelas mesmas razões.

No Capítulo 5 faz-se a descrição detalhada dos procedimentos necessários para concretizar esta metodologia.

4.3. Programming Language Interface – PLI

A PLI é uma interface de acordo com a norma IEEE 1364, que permite adicionar funcionalidades a um simulador de circuitos. Esta interface é aceite por vários simuladores como o Verilog-XL® da Cadence® ou o NCVerilog®, entre outros.

A PLI disponibiliza funções que permitem aceder à estrutura do circuito e a dados relativos ao seu estado durante a simulação. Com base nestas funções disponíveis através da PLI, é possível adicionar vasta funcionalidade extra ao simulador através de funções e procedimentos descritos na linguagem de programação C. Estas funções ou procedimentos são chamados através de chamadas de sistema definidas no ficheiro "veriuser.c", depois de compiladas de forma própria para o simulador. Para utilizar as chamadas de sistema, normalmente colocam-se na *testbench* do circuito a simular e utilizam-se como qualquer primitiva da linguagem Verilog.

4.4. Conclusão

No método actual de teste, em prática na AMIS, os requisitos de qualidade (DL, *Defect Level*) só são atingidos se o teste for preparado tendo em vista a detecção de cada LSA múltiplas vezes. Este procedimento conduz a um elevado número de vectores de teste.

A metodologia proposta visa a detecção de faltas do tipo *bridging* e optimiza os vectores gerados, em cada passo, por ter em consideração os vectores gerados em passos anteriores.

5. Implementação das Metodologias

Neste capítulo apresentam-se as ferramentas comerciais utilizadas, os passos de execução do novo método de teste e do método de teste da AMIS. Apresentam-se os programas utilizados e os comandos necessários para executar as metodologias consideradas neste trabalho.

O método de teste da AMIS é apresentado após o novo método de teste uma vez que é por esta ordem que são feitos os testes para cada circuito, cujos resultados são apresentados no Capítulo 6.

5.1. Software utilizado

Neste trabalho utiliza-se *software* comercial próprio para geração do teste de circuitos, e *software* gratuito como apoio ao *software* comercial. O sistema operativo e programas utilizados são indicados na Tabela 5.1.

Tabela 5. 1– *Software* utilizado

Nome	**Tipo de *software***	**Versão**
CentOS	Sistema Operativo	4.4 Final
Design Vision®	Síntese de circuitos	Y-2006.06
Tetramax®	ATPG e simulação de faltas	Y-2006.06
PLI Wizard	Gerador de Makefiles para compilar funções PLI	05.50.004-p
Verilog-XL®	Simulador verilog da Cadence®	05.50.003-p
glibc	Bibliotecas utilizadas na compilação de código C	2.3.4
gcc	Compilador de código C	3.4.6
AMS®	Bibliotecas de tecnologia c35 3.3V	3.7

O sistema operativo utilizado não utiliza as versões de *software* mais recentes mas sim as que permitam um funcionamento estável do sistema como um todo, o que permite fazer a compilação das funções PLI para o Verilog-XL®. Isto porque para fazer a compilação é necessário criar uma biblioteca dinâmica, a que o Verilog-XL® acede para identificar as funcionalidades adicionadas pela PLI. O principal problema é que a versão das bibliotecas *glibc*, nas versões mais recentes, não permite compilar bibliotecas dinâmicas.

Esta versão do CentOS tem a versão *glibc* indicada na Tabela 5.1, e viabiliza a compilação.

Para os ficheiros de inicialização "synopsys.init" e "cadence.init", e para o ficheiro de configuração ".synopsys_dc.setup", considera-se que os programas estão instalados nas seguintes directorias:

Synopsys®: /soft/synopsys/

Cadence®: /soft/cadence/

AMS®: /soft/ams/3.7/

O conteúdo destes ficheiros é apresentado em anexo.

Os dois primeiros ficheiros são utilizados para adicionar a localização dos programas da Synopsys® e da Cadence® à *PATH* do sistema operativo, o que permite chamar a aplicação na consola de comandos sem ter de escrever o caminho para a aplicação e para que quando o programa está em funcionamento saiba onde estão os componentes que necessita.

Para utilizar estes dois ficheiros, basta escrever na consola de comandos o seguinte:

source synopsys.init
source cadence.init

O último ficheiro indicado, o ".synopsys_dc.setup", é lido pelo Design Vision® automaticamente e indica ao programa onde estão as bibliotecas da tecnologia que se vai utilizar para sintetizar o circuito, além de opções de síntese do circuito como a denominação de certos tipos de nós do circuito, por exemplo. No conjunto de ferramentas da AMS® para cada tecnologia vem um ficheiro de exemplo com estas opções.

Este ficheiro deve estar na directoria de onde se executa o Design Vision® para ser detectado automaticamente.

5.2. Metodologia proposta

Nesta parte apresentam-se os passos de execução do método de teste proposto. A descrição detalhada de cada um destes passos é feita de seguida.

5.2.1. ATPG de faltas LSA, simulação de faltas IDDQ com vectores de teste LSA e ATPG para as faltas IDDQ restantes

Nesta parte do método de teste utiliza-se o Tetramax® para fazer a ATPG para faltas LSA, simulação de faltas IDDQ com vectores de teste LSA e ATPG para as faltas IDDQ restantes.

De modo a realizar o procedimento descrito em 4.2.1, utiliza-se um ficheiro de texto denominado *script*, que contém os comandos próprios do Tetramax®. Este ficheiro não necessita de extensão especial, como no caso do Design Vision®.

Um exemplo genérico deste *script* e a explicação de cada comando são feitos de seguida.

- Começa-se por definir o nome do ficheiro para o qual se deve escrever o *output* completo da sessão do Tetramax®. É útil guardar o *output* para se poder analisar posteriormente os resultados das coberturas de faltas, entre outros.

set messages log *circuito*_s2i.log -replace

- Definem-se as opções de leitura das descrições da tecnologia e do circuito de modo a ler correctamente toda a informação. Vectores não expandidos são adicionados bit a bit e vistos como escalares pelo Tetramax®.

set netlist -sequential_modeling -pin_assign -scalar_net

Assim definições do tipo "wire/input/output [3:0] C" passam a ser tratadas como C[3], C[2], C[1] e C[0], sendo deste modo reconhecidos da mesma forma no Tetramax® e no Verilog-XL®.

- Lêem-se as bibliotecas da tecnologia, para que o programa reconheça as portas lógicas utilizadas na descrição do circuito.

read netlist /soft/ams/3.7/verilog/c35b3/c35_CORELIB.v -library -delete
read netlist /soft/ams/3.7/verilog/c35b3/c35_UDP.v -library

De notar que agora as bibliotecas estão numa pasta diferente das que se usam no Design Vision®, antes eram as da pasta "/soft/ams/3.7/synopsys/c35_3.3V". Para os circuitos testados estas duas bibliotecas são suficientes.

A opção "-library" serve para indicar ao programa que os ficheiros são bibliotecas de tecnologia.

- Lê-se o ficheiro com a descrição do circuito. De notar que neste caso já não é preciso adicionar a opção "-library" ao comando.

read netlist *circuito*_scan.v

É importante ter em mente que o Tetramax® considera por omissão como módulo principal o último módulo a ser lido. É por isso que se lêem primeiro as bibliotecas da tecnologia e só depois é que se lê a descrição do circuito. No menu "Netlist" do ambiente gráfico do Tetramax® é possível alterar esta opção.

- Definem-se as opções para o *build* do circuito de modo que portas lógicas não utilizadas não sejam eliminadas, e que não haja perdas de pinos do circuito. Assim é possível analisar o maior número de faltas para um dado circuito.

set build -nodelete_unused -merge notied_gates_with_pin_loss

- Seleccionam-se opções relacionadas com a aprendizagem do programa sobre as informações do circuito, sem limite de tempo para aprendizagem.

set learning -disable_time_limit

- Faz-se o *build* do circuito. O Tetramax® identifica as partes do circuito que vão fazer parte do processo de ATPG, remove a hierarquia e coloca-as numa imagem em memória que o programa usa.

run build_model *top_module*

- Definem-se regras de descrição do circuito com base na informação do relógio e cadeias de *scan* presentes no ficheiro "*circuito*_scan.spf" (formato STIL), e adicionam-se as opções que apresentam as primitivas das cadeiras de *scan* sejam apresentadas ("trace") e que permitam que sets e resets instáveis não sejam considerados como violações das regras de descrição ("allow_unstable_set_resets "), aumentando assim a cobertura de faltas.

set drc *circuito*_scan.spf -trace -allow_unstable_set_resets

- Faz-se a verificação de regras de descrição do circuito (DRC).

run drc

- Usa-se o seguinte comando para mostrar um sumário do tipo de variáveis que foram eliminadas durante a optimização do circuito. É útil para ver se houve algum problema e portas lógicas foram optimizadas indevidamente.

report summaries library_cells optimizations primitives

De notar que apenas nós vistos pelo programa como primitivas é que se podem definir como local de falta de um determinado modelo. Caso contrário o programa não permite adicionar faltas nesse nós.

- Declaram-se as opções para a ATPG.

set atpg -capture_cycles 4 -full_seq_atpg -verbose
set atpg -merge high -full_seq_merge high

Utiliza-se o valor "4" para o número de ciclos de captura (*capture cycles*), como é aconselhado em [8]. Este valor pode ser definido de "0" e "10", em que com "0" não faz *Full-Sequential* ATPG, mesmo que se use a opção " full_seq_atpg ", e "10" faz *Full-Sequential* ATPG com o maior detalhe que o Tetramax® permite.

É boa prática que se façam algumas experiências com este valor: começa-se por testar com "4" e depois ir aumentando até que a cobertura se aproxime do desejado. Claro que se com o número de ciclos igual a "10" não se obtém a cobertura desejada então é porque não é possível obtê-la. De notar que quanto maior o número de ciclos maior é o tempo de ATPG e maior a exigência do programa em relação a recursos do sistema.

O segundo comando faz com que os *patterns* gerados pelo programa sejam compactados o máximo possível de modo a obter o menor número possível de vectores de teste.

- Removem-se todas as faltas existentes na lista de faltas, define-se o modelo de faltas como LSA e adicionam-se todas as faltas do tipo LSA à lista de faltas.

remove faults -all
set faults -model stuck
add faults -all

- Executa-se o processo de ATPG com as opções definidas e com as faltas adicionadas anteriormente. A opção "auto_compression" indica ao programa para fazer a ATPG de modo optimizado, fazendo um equilíbrio entre o esforço de redução de *patterns* e a utilização do processador.

run atpg -auto_compression

- Para o caso de haver *warnings* do tipo "N20", é sugerido pelo Tetramax® fazer a simulação do circuito com os vectores gerados por ATPG.

Para tal, definem-se as opções da simulação, neste caso para apresentar toda a informação que se obtém da simulação do circuito bom, e depois faz-se a simulação do circuito sequencial de modo a verificar se todos os vectores de teste são válidos.

set simulation -verbose
run simulation -sequential

- Escrevem-se os vectores de teste num formato binário próprio do Tetramax®, para mais tarde se voltar a carregar os *patterns* LSA aqui gerados. Os *patterns* também se guardam num ficheiro do tipo *testbench* em Verilog *tables*. No ficheiro com extensão ".pis" gerado é possível obter o número exacto de vectores de teste gerados, uma vez que cada *pattern* pode ser composto por um ou mais vectores de teste, especialmente em circuitos sequenciais.

write patterns *circuito*_stuck_pat.bin -replace -internal -format binary
write patterns *circuito*_patterns_stuck -replace -internal -format verilog_tables -serial

- Removem-se todas as faltas existentes na lista de faltas, para se poder mudar o modelo de faltas, define-se o novo modelo de faltas como IDDQ e adicionam-se todas as faltas IDDQ existentes.

remove faults -all
set faults -model iddq
add faults -all

- Definem-se os vectores de teste LSA como vectores a serem utilizados para a simulação de faltas.

set patterns external *circuito*_stuck_pat.bin

- É aconselhado no manual de utilizador [8] fazer primeiro uma simulação para o circuito sem faltas, pois se houver erros para um circuito sem faltas nem vale a pena simular o circuito com faltas. De seguida, faz-se a simulação de faltas IDDQ com vectores de teste LSA, eliminando da lista de faltas as faltas IDDQ detectadas por vectores de teste LSA.

run simulation -sequential
run fault_sim

- Passa-se a utilizar o conjunto interno de *patterns* e faz-se a ATPG para as faltas IDDQ que restaram da simulação de faltas.

set patterns internal
run atpg -auto_compression

- Mais uma vez, para o caso de haver *warnings* do tipo "N20", é boa prática fazer a simulação do circuito com os vectores gerados por ATPG. O primeiro comando serve para que toda a informação gerada ao fazer a simulação seja apresentada no ecrã.

set simulation -verbose
run simulation

- Finalmente, escrevem-se os padrões de teste para um ficheiro binário no formato do Tetramax®, para um ficheiro com uma *testbench* em Verilog e para um conjunto de ficheiros que também são uma *testbench* em Verilog mas que neste caso apenas se utilizam para saber o número total de vectores de teste IDDQ.

write patterns *circuito*_iddq_pat.bin -replace -internal -format binary
write patterns *circuito*_tb.v -replace -internal -format verilog_single_file -parallel 1
write patterns *circuito*_patterns_iddq -replace -internal -format verilog_tables -serial

No último comando indicado utiliza-se a opção "-serial" e no penúltimo "-parallel 1". Com a opção "-serial", a *testbench* é gerada de modo a que a aplicação da cadeia de *scan* seja feita em série quando se prepara a simulação para um determinado vector de teste, e é feita a simulação dos *shifts* da cadeia de *scan*.

Já com a opção "-parallel 1", a aplicação cadeia de *scan* é em paralelo e os *shifts* da cadeia de *scan* não são simulados, diminuindo assim o tempo necessário para fazer a simulação do circuito. O "1" da opção "-parallel 1" faz com que o último bit de cada pattern seja aplicado em série na simulação. De modo geral, são os N últimos bits de cada pattern que são aplicados em série na simulação, sendo neste caso N=1 de modo a diminuir ao máximo o tempo necessário à simulação do circuito, pois quanto maior o N mais bits são aplicados em série e mais tempo leva a simulação pelas mesmas razões que tornam a simulação com a opção "-serial" mais lenta que a "-parallel N".

A *testbench* com a opção "-parallel 1", *circuito*_tb.v, é a que é utilizada com o Verilog-XL®, pois permite uma simulação mais rápida.

A opção "-replace", que é utilizada em vários comandos, serve para indicar que se deve substituir o ficheiro caso este já exista.

5.2.2. Geração aleatória e automática de faltas bridging

Neste passo faz-se a geração aleatória de faltas BRI com o Verilog-XL®, uma vez que o *software* utilizado neste trabalho não permite obter uma lista de faltas BRI real.

A descrição das funções e procedimentos que adicionam funcionalidades ao simulador Verilog-XL® através da PLI é feita de seguida.

Descrição das funções e procedimentos em C

Para adicionar as funcionalidades desejadas ao simulador Verilog-XL® utilizam-se 4 ficheiros ("veriuser.c", "monitor.h", "monitor_user.c" e "monitor.c") e o programa Pli Wizard, que constrói a *Makefile* que vai compilar o código para o simulador final.

De seguida descrevem-se os ficheiros e funções que conferem novas funcionalidades ao simulador verilog.

veriuser.c

Neste ficheiro, que vem com as ferramentas PLI da Cadence®, definem-se funções e procedimentos que adicionam novas funcionalidades ao simulador Verilog-XL®.

No caso deste trabalho, adicionam-se as chamadas de sistema $iddq_init, $iddq_bri e $iddq_close, associadas às funções em C mon_call(), mon_misc() e write_faults_file(), respectivamente. Estas funções e as variáveis globais utilizadas são descritas nos ficheiros monitor.c, monitor_user.c e monitor.h.

O simulador obtido após compilação reconhece estas chamadas de sistema (usertask) e executa a função a que está associada sempre que é chamada.

Neste ficheiro também é possível definir uma *string* identificadora da versão deste ficheiro, que aparece no início da execução do simulador.

No caso deste trabalho, indica-se a principal utilidade do simulador e a versão de desenvolvimento:

"Simulação de faltas BRIDGING com vectores de teste IDDQ vX.Y "

monitor.h

Neste ficheiro declaram-se as estruturas que são utilizadas, os ficheiros a incluir na compilação (sejam bibliotecas C ou da PLI), variáveis globais utilizadas no programa e os cabeçalhos das funções e procedimentos descritos no ficheiro "monitor_user.c".

De seguida faz-se a descrição do conteúdo deste ficheiro.

A estrutura "lista" é uma lista do tipo ligada, e é utilizada para representar a lista de faltas associadas a um determinado nó. Contém a informação do nome (name) e tipo de faltas (type): ba0 ou ba1.

Cada nó vai ter nesta lista os nós vítima de faltas *bridging* e o tipo de falta (ba0 ou ba1) correspondente.

```
typedef struct lista{
  char *name;
  char type[4];
  struct lista *next;
  }fault_list;
```

A estrutura t_mon_node guarda a informação sobre cada nó observado, possuindo a informação do "handle" associado a este nó (param), que é utilizado para aceder ao objecto correspondente ao nó durante a simulação. O nome desse nó também é guardado (name), tal como o seu valor lógico actual (value) e a lista de faltas correspondente a esse nó (faults), utilizando a estrutura "lista" descrita acima.

```
typedef struct t_mon_node{
  handle param;
  char name[MAXCHAR];
  short int value;
  fault_list *faults;
}s_mon_node, *p_mon_node;
```

É representada como uma tabela, em que cada elemento é uma estrutura do tipo t_mon_node.

O conjunto destas duas estruturas representa o seguinte: um nó do tipo p_mon_node é o nó agressor, e os nós vítima de faltas BRI com este nó agressor estão guardados na lista de faltas deste nó.

Nestas duas estruturas o nome que é guardado é o nome completo do nó (fullname), que além de ter o nome do nó tem a hierarquia associada a esse nó. Deste modo é possível distinguir *wires* ou vectores com o mesmo nome em módulos diferentes do circuito, por exemplo.

Utilizam-se as seguintes variáveis globais, que servem para guardar e aceder à informação dos nós que estão a ser observados:

p_mon_node mon_array; →estrutura com a informação dos nós e faltas
char *mon_instance_p; →necessária para o funcionamento de funções PLI

As seguintes variáveis globais também são utilizadas:

int mon_num_params; →número de nós do circuito
double kapa; →número de faltas LSA
int limite; →número máximo de faltas BRI
int nro_faltas_inicial; →número de faltas inicial
int nro_faltas_final; →número de faltas final
int nro_faltas; →número de faltas num dado instante
int qwerty; →número de faltas duplicadas
double kapa2; →número por que se vai multiplicar o número de faltas LSA para obter o número de faltas BRI que se devem gerar

monitor_user.c

Neste ficheiro descrevem-se as funções e procedimentos que tratam da leitura do ficheiro de faltas indicado na chamada $iddq_init, que preenchem as estruturas que guardam a informação dos nós a observar e respectivas faltas associadas, que testam se uma determinada falta é detectada ou não e caso seja retira-a da lista de faltas correspondente, e que escrevem para ficheiro a lista de faltas resultante da simulação, com o mesmo formato que se considera para o ficheiro de entrada.

De seguida faz-se a descrição de cada função/procedimento:

p_mon_node realloc_info()

Esta função reduz o espaço reservado para a estrutura "mon_array" para o tamanho que é realmente necessário e retorna a "mon_array" actualizada.

int enter(fault_list *first_ptr, char *name, char *type, char *state)

Função utilizada para inserir um elemento na lista de faltas. Esse novo elemento é inserido no fim da lista. Caso a falta já exista na lista de faltas, não é inserida.

Retorna 0 caso não insira ou 1 caso insira a falta.

p_mon_node insert_info(char *name, char *fault_name, char *fault_type, char *state, int flag)

Função que insere em "mon_array" um nó que deve ser observado durante a simulação.

O argumento "flag" é utilizado para distinguir inserção de nós *bridging* vítima (flag a 1) dos nós agressor (flag a 0).

Sempre que se adiciona uma falta à lista de faltas, o número total de faltas é incrementado e guardado em "nro_faltas". Caso já exista uma lista de faltas, utiliza-se a função enter() descrita acima.

Depois de inserir o nó, retorna a "mon_array" actualizada.

int duplicado(char *a, char *b, char *type)

Função utilizada para testar se a falta BRI (b,a) já existe quando se tenta inserir (a,b).

Caso já exista retorna 1, caso contrário devolve 0.

Com esta função é possível evitar ter duas faltas sobre dois nós, sendo uma do tipo (a,b) e a outra (b,a).

O tipo da falta (ba0 ou ba1) também é tido em conta para este teste.

void remover(fault_list *first_ptr, char *name, char *type)

Procedimento que remove a falta cujo nome e tipo é passado por argumento, da lista de faltas indicada. É utilizado para remover um nó da lista de faltas quando essa falta é detectada. Sempre que um nó é removido, o número de faltas total (nro_faltas) é actualizado.

void update_faults(char *nome, int logic)

Procedimento que verifica se um dado nó com um dado valor lógico permite a detecção de alguma falta *bridging*. O teste é feito da seguinte forma: para cada nó (agressor), verifica-se para todas as faltas presentes na lista de faltas correspondente (nós vítima) se o valor lógico dos nós permite detectar a falta *bridging*, ou seja, se o valor lógico de um nó vítima for o contrário do valor do nó agressor, e o tipo de falta for o correcto, a falta é detectável. Por exemplo:

- update_faults() recebe como argumento o nó "A" com o valor lógico "0"
- a lista de faltas do nó "A" é percorrida e para cada nó vítima presente na lista é feito o seguinte teste:
    - verifica-se qual o valor lógico do nó vítima
    - se o valor lógico do nó vítima for "1", e o tipo de falta entre "A" e o nó vítima for do tipo "ba0", então a falta é detectável e o nó é eliminado da lista de faltas

Caso uma falta seja detectável, o procedimento remover() é chamado para a remover da lista de faltas.

void read_file(char *filename)

Procedimento que recebe o nome do ficheiro com a lista de faltas no formato que o Tetramax® utiliza para representar faltas, abre-o e lê linha a linha a informação das faltas e dos nós a observar. Caso o ficheiro não possa ser lido, é pedido o nome de um ficheiro de faltas que possa ser lido até que um seja fornecido pelo utilizador.

Para isto, chama as funções duplicado(), insert_info(), realloc_info() de modo a preencher a tabela "mon_array".

Apenas se podem inserir faltas que não tenha já a simétrica inserida, o que é garantido com o auxílio da função duplicado().

Este procedimento é apenas utilizado para efeitos de teste. O procedimento read_list_tmax() faz o mesmo papel que esta função, mas de modo mais completo.

void write_faults_file()

Procedimento que escreve a lista de faltas resultante após a simulação do circuito, e liberta a memória reservada para as estruturas. Este é o procedimento que é chamado quando se utiliza a chamada de sistema $iddq_close.

O formato deste ficheiro é constituído por comandos próprios do Tetramax® que servem para adicionar uma falta *bridging* entre dois nós, indicando os nós, o tipo de falta e qual é o nó agressor, que neste caso é sempre o segundo nó indicado no comando. Assim evitam-se problemas com a possível diferença entre o nome dos nós no simulador Verilog-XL® e no Tetramax®.

Como exemplo, a inserção de uma falta entre o nó vítima n103 e o nó agressor G11, do tipo ba1, é feito da seguinte forma:

```
add faults -bridge_location n103 G11 -bridge 1 -aggressor_node Second
```

Esta informação é escrita para o ficheiro fault_list_*#faltasLSA*.txt. Caso o ficheiro exista, o seu conteúdo é eliminado. Caso não exista, é criado um novo com este nome.

Este ficheiro mais tarde é lido como um ficheiro de comandos para o Tetramax®.

Antes de se escrever a lista de faltas restantes para ficheiro é necessário fazer a conversão entre o formato de nós do Verilog-XL® e do Tetramax®.

Para identificar cada nó o simulador Verilog-XL® utiliza a seguinte forma:

testbench.top_module.module.port

que é diferente da usada no Tetramax®

top_module/module/port

Esta conversão é feita automaticamente antes de escrever para ficheiro.

void read_lista_faltas(char *ficheiro)

Procedimento que permite usar a lista de faltas gerada aleatoriamente pela função *random_faults*, de forma a fazer a simulação das mesmas faltas BRI com vectores IDDQ diferentes.

void read_list_tmax(char *nome)

Procedimento que permite a leitura de listas de faltas guardadas pelo Tetramax®, tendo cada falta o seguinte formato:

Tipo de falta BRI   Código de detecção no Tetramax®   Nó vítima   Nó agressor

Devido a optimizações do Tetramax® que não se conseguiram evitar, algumas faltas dizem respeito a nós relacionados a *buffers* que não existem no circuito inicialmente. Estas faltas são ignoradas e não são tidas em conta para a simulação de faltas, uma vez que no circuito estes *buffers* não existem e no Verilog-XL® não é possível aceder à informação relativa a estes nós.

A informação sobre o número de nós que foram adicionados para observação durante a simulação, o número de faltas inicial e o número de faltas final são apresentados no ecrã.

Toda a informação que é impressa para o ecrã também é guardada em ficheiro. Este ficheiro tem o seguinte formato:

*circuito*_i2b.txt

ou

*circuito*_ver_rand.txt

quer seja para fazer a simulação de faltas ou gerar faltas *bridging* aleatoriamente.

Sempre que é reservado espaço na memória para variáveis ou estruturas testa-se se a operação foi bem sucedida ou não, utilizando asserções. Caso haja algum erro deste tipo, é impresso para o ecrã o ficheiro, a posição do ficheiro, o teste que deu erro, e de seguida o programa termina, já que não é possível continuar com o funcionamento correcto deste.

monitor.c

Neste ficheiro estão descritos as funções/procedimentos principais, sendo chamadas funções/procedimentos descritos no ficheiro monitor_user.c conforme necessário.

Estas funções/procedimentos são utilizados para definir as novas chamas de sistema $iddq_init, $iddq_bri e $iddq_close em veriuser.c.

De seguida faz-se a descrição dos procedimentos presentes neste ficheiro.

int is_unconn(char *name)

Função que verifica pelo nome de um nó se é um nó que está ligado ou não a alguma primitiva ou módulo, procurando na *string* "name" a letra "S" e se a encontrar verifica se as letras seguintes formam a palavra SYNOPSYS_UNCONN, que é parte do nome que o Design Vision® dá a nós que estão desligados: SYNOPSYS_UNCONNECTED_*Número*.

Caso seja um nó desligado devolve 1, senão devolve 0.

Esta função é utilizada no procedimento random_faults(), para evitar erros no Tetramax®.

void random_faults()

Este procedimento é utilizado para gerar a inserção aleatória de faltas sobre todos os nós de todos os módulos do circuito. É importante referir que neste trabalho apenas se trabalham com nós do tipo "simulated_net", ou seja, em vez de trabalhar com o nó saída de um módulo, o nó de entrada de outro módulo e o nó correspondente ao *wire* que liga estes dois nós trabalha-se apenas com o nó de simulação que corresponde a estes três nós. Um nó de simulação é, neste contexto, um nó que só existe durante a simulação no Verilog-XL®, estando associado a nós reais. Deste modo existem menos nós a observar e testar do que no caso em que se usa todos os nós reais, e o resultado é o mesmo.

Este procedimento começa por adicionar todos os nós do circuito à estrutura "mon_array", calcula o número máximo de faltas com base na equação (2), sendo Kapa o valor pelo qual se deve multiplicar o número de faltas LSA.

$$\text{Limite} = \#\text{faltasLSA} \times \text{Kapa} \qquad (2)$$

De notar que é necessário testar o valor deste *limite*, pois como não é permitido a inserção de faltas simétricas, com o auxílio da função duplicado(), o número máximo de faltas possível corresponde ao valor dado pela equação (1). O número de faltas LSA e o Kapa são passados por argumento na chamada de sistema $iddq_init.

Depois de calcular o número de faltas a inserir, são gerados dois números aleatoriamente, impedindo que sejam iguais. Cada um destes números corresponde a um nó de “mon_array”, sendo o primeiro nó o nó agressor e o segundo o nó vítima. De seguida insere-se uma falta entre esses dois nós. O tipo de falta (ba0 ou ba1) também é escolhido aleatoriamente.

Mais uma vez, não é permitida a inserção de faltas simétricas, situação identificada pela função duplicado() que obriga a nova geração.

Depois de gerar o número pedido de faltas, a lista de faltas gerada é guardada num ficheiro, num formato semelhante ao utilizado na estrutura de dados “mon_array”. Este ficheiro pode ser lido mais tarde, com o auxílio da função read_lista_faltas().

int mon_consume(p_vc_record change_data)

Esta função corresponde à rotina PLI *consumer*, estando associada ao *trigger* adicionado a cada nó através da função PLI acc_vcl_add(), sendo assim chamada quando há mudança de valor lógico do nó a que está associada.

Quando é chamada, actualiza o valor lógico do nó associado a esse *trigger*. De notar que cada nó que está a ser observado tem um trigger independente.

void mon_call()

Procedimento que corresponde à rotina PLI *calltf*, e é o procedimento associado à chamada de sistema $iddq_init.

Neste procedimento reserva-se espaço para a estrutura “mon_array”, sendo reservada memória suficiente para 100000 nós. Este número foi escolhido pois é o que permite o bom funcionamento da parte PLI do método com todos os circuitos testados.

Para adicionar os nós a observar a “mon_array”, há vários modos de funcionamento disponíveis, que podem ser seleccionados quando se executa a chamada de sistema $iddq_init. Esta chamada de sistema necessita dos seguintes argumentos:

$iddq_init(“*lista de faltas*”, *#faltasLSA*, *Kapa*, *modo de funcionamento*);

De notar que é necessário usar sempre o número de argumentos indicado acima, mesmo que não seja necessário usar a lista de faltas indicada. De outro modo os argumentos não são correctamente reconhecidos.

Para seleccionar o modo de funcionamento basta indicar um número de 1 a 4, com as seguintes funcionalidades:

1- Faz a simulação de faltas com a lista de faltas no formato Tetramax®, de modo semelhante ao modo de funcionamento 3, a única diferença é que o modo como procura os nós do circuito é feito de outra maneira, menos eficiente. Apenas se mantém por uma questão de referência.

2- Gera a lista de faltas aleatória para o Tetramax®, e fecha o simulador

3- Faz simulação de faltas BRI com a lista de faltas do Tetramax®

4- Faz simulação de faltas com a lista de faltas gerada pela função random_faults()

De seguida, é adicionada a "mon_array" a informação relativa ao valor lógico inicial e ao *handle* para o objecto correspondente a um determinado nó.

Finalmente, é adicionado um *trigger* a esse nó, através da função PLI acc_vcl_add, que faz com que sempre que haja uma alteração no valor lógico deste nó a rotina PLI *consumer* seja invocada: mon_consume().

void mon_misc()

Procedimento que verifica qual é o valor lógico de cada nó e quais as faltas que são detectadas com os valores lógicos de cada nó. Corresponde à chamada de sistema $iddq_bri.

De modo a gerar a *Makefile* que é usada na compilação deste código utiliza-se o PLI Wizard®.

Na linha de comandos basta escrever "pliwiz" para iniciar programa.

Os seguintes passos são necessários:

- escolher o nome e directoria para a sessão de trabalho
- indicar a localização do ficheiro veriuser.c modificado
- indicar os ficheiros com o código C que contêm as funções a compilar (o ficheiro .h não é necessário neste passo, basta estar na mesma directoria dos ficheiro .c)
- escolher como compilador o GCC e escolher a opção de parâmetros avançados de compilação
- escolher como parâmetros –Wall, -pedantic, -g e –ansi de modo a que todos os *warnings* sejam apresentados e seja possível fazer *debug* ao código
- gerar a *Makefile*

Depois de gerar a *Makefile*, basta escrever "make" na linha de comandos para compilar o código e assim adicionar as funcionalidades descritas acima ao simulador Verilog-XL®.

Para esta parte da preparação do teste de um circuito, a opção que interessa é a 2: gerar a lista de faltas para o Tetramax® e fechar o simulador logo após ter terminado. A chamada de sistema $iddq_init tem o seguinte aspecto:

$iddq_init("lista de faltas", #faltasLSA, Kapa, **2**);

Na linha de comandos, a forma com que se chama o Verilog-XL® é a seguinte:

verilog *cir*.v *cir_testbench*.v –v *libraries*.v –l *circuito*_ver_rand.txt

Caso a parte da PLI ainda não tenha sido compilada, é necessário fazer "make" na linha de comandos antes de chamar o simulador.

Caso se queira executar o Verilog-XL® com interface gráfica, o SimVision®, é necessário adicionar "+gui" quando se invoca o simulador.

Quando for necessário alterar o modo de funcionamento do programa não é necessário recompilar o programa, basta mudar o número do modo de funcionamento na *testbench* na chamada de sistema $iddq_init.

5.2.3. Simulação de faltas bridging com vectores de teste LSA

Nesta parte do método de teste faz-se a simulação de faltas BRI com os vectores de teste LSA gerados em S2I. Desta maneira é possível eliminar da lista de faltas BRI as que são detectadas por vectores de teste LSA e assim evita-se gerar vectores de teste que detectem faltas BRI já detectadas por vectores de outro tipo de faltas.

Para executar o procedimento descrito em 4.2.3, utiliza-se um *script*. Um exemplo genérico e a explicação dos comandos são apresentados de seguida.

- Começa-se por definir o nome do ficheiro para o qual se deve escrever o *output* completo da sessão do Tetramax®.

set messages log *circuito*_s2b.log -replace

- Definem-se as opções de leitura das descrições da tecnologia e do circuito de modo a ler correctamente toda a informação, lêem-se as bibliotecas da tecnologia, e lê-se o ficheiro com a descrição do circuito.

set netlist -sequential_modeling -pin_assign -scalar_net
read netlist /soft/ams/3.7/verilog/c35b3/c35_CORELIB.v -library -delete
read netlist /soft/ams/3.7/verilog/c35b3/c35_UDP.v -library
read netlist *circuito*_scan.v

- Definem-se as opções para o *build* do circuito, seleccionam-se opções relacionadas com a aprendizagem do programa sobre as informações do circuito, faz-se o *build* do circuito, definem-se regras de descrição do circuito com base na informação do relógio e cadeias de *scan* presentes no ficheiro "*circuito*_scan.spf", adicionam-se as opções que apresentam as primitivas das cadeiras de *scan* sejam apresentadas ("trace") e que permitam que *sets* e *resets* instáveis não sejam considerados como violações das regras de descrição ("allow_unstable_set_resets "), e faz-se a verificação de regras de descrição do circuito (DRC).

set build -nodelete_unused -merge notied_gates_with_pin_loss
set learning -disable_time_limit -verbose
run build_model *top_model*
set drc *circuito*_scan.spf -trace -allow_unstable_set_resets
run drc

- Usa-se o seguinte comando para mostrar um sumário que apresenta o tipo de variáveis que foram eliminadas durante a optimização do circuito.

report report summaries library_cells optimizations primitives

- Define-se o modelo de faltas como BRI, indica-se ao programa para aceitar faltas *bridging* pinos de *input* do circuito e adicionam-se todas as faltas do tipo BRI do ficheiro gerado em VER_RAND à lista de faltas.

set faults -model bridging
set faults -bridge_inputs
source *circuito*_tmax_list_#*faltas_LSA*.txt

- Define-se os vectores de teste LSA como vectores de teste para a simulação de faltas.

set patterns external *circuito*_stuck_pat.bin

- Faz-se primeiro uma simulação para o circuito sem faltas. De seguida, faz-se a simulação de faltas BRI com vectores de teste LSA, eliminando da lista de faltas as faltas BRI detectadas por vectores de teste LSA. A opção "strong_bridge" no comando de simulação de faltas faz com que a detecção de faltas seja optimizada, ou seja, a falta é assinalada como detectável quando os critérios para detecção de faltas *bridging* completamente optimizada são respeitados. Estes critérios são apresentados na Tabela 5.2.

Os comandos utilizados são os seguintes:

run simulation
run fault_sim -strong_bridge

Tabela 5. 2 – Critérios do Tetramax® para a detecção de faltas *bridging*

Porta lógica condutora	ba0		ba1	
	Condutora de vítima	**Condutora de agressor**	**Condutora de vítima**	**Condutora de agressor**
AND		Maximiza entradas da condutora com ‘0’	Minimiza entradas da condutora com ‘0’	
NAND	Maximiza entradas da condutora com ‘0’			Maximiza entradas da condutora com ‘0’
OR	Minimiza entradas da condutora com ‘1’			Maximize driver condutora com ‘1’
NOR		Maximiza entradas da condutora com ‘1’	Minimiza entradas da condutora com ‘1’	

- Finalmente, escrevem-se as faltas *bridging* que não foram detectadas na simulação de faltas (que não foram observados: –class NO (*Not Observed*)) num ficheiro que mais tarde é utilizado na parte I2B.

write faults *circuito*_tmax_faults.txt -replace -class NO

5.2.4. Simulação de faltas bridging com vectores de teste IDDQ

Nesta parte utilizam-se os vectores de teste para faltas IDDQ obtidos com o Tetramax® para fazer a simulação do circuito no Verilog-XL® com verificação de detecção de faltas do tipo *bridging*.

A descrição do procedimento indicado em 4.2.4 é feita de seguida.

Relativamente à parte VER_RAND, a única alteração necessária na *testbench* é alterar o modo de utilização de 2 para 3, considerando que se colocou desde de início no primeiro argumento o nome do ficheiro com as faltas BRI resultantes de S2B:

VER_RAND: $iddq_init("lista de faltas", #faltasLSA, Kapa, **2**);
I2B: $iddq_init("lista de faltas", #faltasLSA, Kapa, **3**);

A lista de faltas *bridging* resultante é escrita para ficheiro, no formato de comandos de inserção de faltas *bridging* do Tetramax®. Um exemplo geral é o seguinte:

add faults -bridge_location *NóA NóB* -bridge *0/1(ba0/ba1)* -aggressor_node Second

Para efectuar a simulação de faltas basta invocar o Verilog-XL® de modo semelhante ao que se faz no passo VER_RAND, a única diferença é o nome do ficheiro de *log* da simulação:

verilog *circuito*.v *circuito_testbench*.v –v *libraries* –l *circuito*_i2b.txt

5.2.5. ATPG de faltas bridging

Neste passo final do novo método de teste utiliza-se o Tetramax® para fazer a ATPG para as faltas BRI que não são detectadas pelos vectores de teste LSA nem pelos vectores IDDQ.

Utiliza-se um *script* para executar o procedimento descrito em 4.2.5. Um exemplo genérico e explicação dos comandos são apresentados de seguida.

- Começa-se por definir o nome do ficheiro para o qual se deve escrever o *output* completo da sessão do Tetramax®.

set messages log *circuito*_b.log -replace

- Definem-se as opções de leitura das descrições da tecnologia e do circuito de modo a ler correctamente toda a informação, lêem-se as bibliotecas da tecnologia, e lê-se o ficheiro com a descrição do circuito.

set netlist -sequential_modeling -pin_assign -scalar_net
read netlist /soft/ams/3.7/verilog/c35b3/c35_CORELIB.v -library -delete
read netlist /soft/ams/3.7/verilog/c35b3/c35_UDP.v -library
read netlist *circuito*_scan.v

- Definem-se as opções para o *build* do circuito. Seleccionam-se opções relacionadas com a aprendizagem do programa sobre as informações do circuito, faz-se o *build* do circuito, definem-se regras de descrição do circuito com base na informação do relógio e cadeias de *scan* presentes no ficheiro "*circuito*_scan.spf", adicionam-se as opções que apresentam as primitivas das cadeiras de *scan* sejam apresentadas ("trace") e que permitam que *sets* e *resets* instáveis não sejam considerados como violações das regras de descrição ("allow_unstable_set_resets "), e faz-se a verificação de regras (DRC).

set build -nodelete_unused -merge notied_gates_with_pin_loss
set learning -disable_time_limit -verbose
run build_model *top_model*
set drc *circuito*_scan.spf -trace -allow_unstable_set_resets
run drc

- Usa-se o seguinte comando para mostrar um sumário que apresenta o tipo de variáveis que foram eliminadas durante a optimização do circuito.

report report summaries library_cells optimizations primitives

- Define-se o modelo de faltas como BRI, indica-se ao programa para aceitar faltas *bridging* nos pinos de *input* do circuito e adicionam-se todas as faltas do tipo BRI do ficheiro gerado em I2B à lista de faltas.

set faults -model bridging
set faults -bridge_inputs
source *circuito*_fault_list_*#faltas_LSA*.txt

- Definem-se as opções para fazer a ATPG de faltas BRI, de modo semelhante ao que se faz na parte S2I para a ATPG de faltas LSA e de faltas IDDQ.

set atpg -optimize_bridge_strengths -capture_cycles 4
set atpg -merge high -full_seq_merge high
set patterns internal
run atpg -auto_compression
write patterns *circuito*_bridging_pat.bin -replace -internal -format binary
write patterns *circuito*_patterns_bri -replace -internal -format verilog_tables -serial

5.3. Método de teste actual

De modo a executar o procedimento apresentado no Capítulo 4.1, utilizam-se dois *scripts* e o simulador Verilog-XL® no modo de utilização 3. Como este método é utilizado após o novo método de teste, para sua avaliação, não é necessário fazer o passo referido em 4.1 que consiste na ATPG para faltas LSA, seguido da simulação de faltas IDDQ com vectores de teste LSA e ATPG para as faltas IDDQ restantes.

Começa-se por utilizar um *script* para fazer a ATPG para faltas LSA com detecção múltipla de cada faltas (cinco vezes cada falta), e de seguida faz-se a simulação de faltas BRI com os vectores de teste LSA gerados. Esta simulação de faltas não se faz num caso real. Neste caso faz-se de modo a poder fazer a comparação do nível de detecção de faltas BRI por vectores LSA entre este método e o novo método de teste.

De seguida apresenta-se um *script* genérico que executa este procedimento:

- Começa-se por definir o nome do ficheiro para o qual se deve escrever o *output* completo da sessão do Tetramax®.

set messages log *circuito*_s5x2b.log –replace

- Definem-se as opções de leitura das descrições da tecnologia e do circuito de modo a ler correctamente toda a informação. Lêem-se as bibliotecas da tecnologia. Lê-se o ficheiro com a descrição do circuito.

set netlist -sequential_modeling -pin_assign -scalar_net
read netlist /soft/ams/3.7/verilog/c35b3/c35_CORELIB.v -library -delete
read netlist /soft/ams/3.7/verilog/c35b3/c35_UDP.v -library
read netlist *circuito*_scan.v

- Definem-se as opções para o *build* do circuito. Seleccionam-se opções relacionadas com a aprendizagem do programa. Faz-se o *build* do circuito.

set build -nodelete_unused -merge notied_gates_with_pin_loss
set learning -disable_time_limit
run build_model *top_module*

- Definem-se regras de descrição do circuito com base na informação do relógio e cadeias de *scan* presentes no ficheiro "*circuito*_scan.spf" (formato STIL), e adicionam-se as opções que apresentam as primitivas das cadeiras de *scan* sejam apresentadas ("trace") e que permitam que sets e resets instáveis não sejam considerados como violações das regras de descrição ("allow_unstable_set_resets "). Faz-se a verificação de regras de descrição do circuito (DRC).

set drc *circuito*_scan.spf -trace -allow_unstable_set_resets
run drc

- Imprime-se um sumário do tipo de variáveis que foram optimizadas.

report summaries library_cells optimizations primitives

- Declaram-se as opções para a ATPG. Define-se o modelo de faltas como LSA e adicionam-se todas as faltas do tipo LSA à lista de faltas. Executa-se a ATPG. A opção "-ndetect 5" permite fazer ATPG com detecção múltipla de cada falta (cinco vezes cada falta).

set atpg -capture_cycles 4 -full_seq_atpg -verbose
set atpg -merge high -full_seq_merge high
set faults -model stuck
add faults -all
run atpg -auto_compression -ndetect 5

- Faz-se a simulação do circuito com os vectores gerados por ATPG, por causa de erros do tipo "N20".

set simulation -verbose
run simulation -sequential

- Escreve-se os vectores de teste para ficheiro.

write patterns *circuito*_stuck_pat_ami.bin -replace -internal -format binary
write patterns *circuito*_pat_stuck_ami -replace -internal -format verilog_tables –serial

- Removem-se todas as faltas presentes na lista de faltas. Define-se o modelo de faltas como BRI, indica-se ao programa para aceitar faltas *bridging* pinos de *input* do circuito e adicionam-se todas as faltas do tipo BRI do ficheiro gerado em VER_RAND à lista de faltas.

remove faults -all
set faults -model bridging
set faults -bridge_inputs
source *circuito*_tmax_list_*#faltasLSA*.txt

- Definem-se os vectores de teste LSA como vectores de teste para a simulação de faltas. Executa-se primeiro uma simulação do circuito sem faltas. Faz-se a simulação de faltas BRI com vectores de teste LSA, eliminando da lista de faltas as faltas BRI detectadas por vectores de teste LSA. A opção "strong_bridge" no comando de simulação de faltas faz com que a detecção de faltas seja optimizada. Os critérios de detecção são apresentados na Tabela 5.1.

set patterns external *circuito*_stuck_pat_ami.bin -sensitive
run simulation
run fault_sim -strong_bridge

- Finalmente, escrevem-se as faltas *bridging* que não foram detectadas na simulação de faltas (que não foram observados: –class NO (*Not Observed*)) para ficheiro.

write faults *circuito*_bri_faults.txt -replace -class NO

Para utilizar este *script* basta mudar *circuito* para o nome do ficheiro, mudar *top_module* para o nome do módulo principal do circuito, e mudar *#faltasLSA* para o número de faltas LSA do circuito.

Como é referido acima, o segundo passo de execução não é necessário fazê-lo novamente, uma vez que é exactamente igual ao passo S2I do novo método de teste e, como este método de teste da AMIS é utilizado após o novo método de teste, os resultados necessários para esta parte já estão disponíveis.

Para o passo seguinte, simulação de faltas BRI com vectores de teste IDDQ, utiliza-se o simulador Verilog-XL® com funcionalidades adicionadas através da PLI, da mesma forma que se faz em 5.1.4:

- utiliza-se o modo de utilização 3, e a lista de faltas BRI obtida no fim do primeiro passo deste método de teste. A chamada de sistema $iddq_init tem o seguinte aspecto:

$iddq_init("lista de faltas BRI", #faltasLSA, Kapa, 3);

- guarda-se o *output* do programa para um ficheiro com o formato

"*circuito*_i2b_ami.txt"

- guarda-se a lista de faltas BRI restante num ficheiro com o formato

"*circuito*_fault_list_#faltaLSA.txt"

De resto, é em tudo igual ao passo I2B do novo método de teste.

Finalmente, utiliza-se um *script* para executar o último passo no Tetramax®, simulação de faltas com os vectores de teste LSA obtido através de ATPG com detecção de faltas simples. Este *script* é semelhante ao utilizado no passo S2B do novo método de teste, só que em vez de se utilizar a lista de faltas BRI completa utiliza-se a lista de faltas resultante da simulação de faltas BRI com vectores de teste IDDQ feita no passo anterior a este.

As diferenças deste script para o utilizado no passo S2B do novo método de teste são as seguintes:

set messages log *circuito*_s2b_ami.log -replace

source b18_fault_list_386966_ami.txt

write faults b18_bri_final.txt -replace -class NO

5.4. Conclusões

As ferramentas comerciais disponíveis permitem realizar a maioria dos passos propostos na metodologia descrita no Capítulo 4.

O PLI permitiu adicionar ao simulador Verilog-XL® da Cadence® a capacidade de avaliar a detecção, por IDDQ, de faltas do tipo *bridging*, concretizando um passo fundamental da metodologia proposta para a optimização dos vectores gerados.

6. Resultados

Para testar o método de teste desenvolvido neste trabalho utilizam-se 10 circuitos de referência, disponibilizados pelo Politecnico di Torino.

De modo a comparar o novo método de teste e o método de teste da AMIS, fazem-se testes com os dois métodos para todos os circuitos utilizados.

6.1. Exemplo da preparação do teste do circuito b18

De modo geral, para cada circuito, começa-se por fazer a síntese do circuito no Design Vision® com base num *script* e analisam-se os resultados para ver se há erros.

De seguida, utiliza-se primeiro o novo método de teste e posteriormente executa-se o método de preparação do teste da AMIS.

Começa-se por executar o passo S2I no Tetramax®, utilizando o *script* "1s2i.txt" com a informação do circuito em questão.

Depois faz-se a geração aleatória e automática de faltas *bridging*, executando o passo VER_RAND com o simulador Verilog-XL® no modo 2 de funcionamento.

Executa-se o passo S2B no Tetramax®, utilizando o *script* "2s2b.txt".

Depois, executa-se o passo I2B fazendo a simulação de faltas *bridging* com vectores IDDQ no Verilog-XL® no modo de funcionamento 3.

Finalmente, executa-se o passo B, fazendo a ATPG para as faltas *bridging* restantes utilizando o *script* "3b.txt". Isto conclui a parte do teste com o novo método de teste.

Para fazer o teste do circuito com o método de teste da AMIS, executa-se o passo de ATPG com detecção múltipla de faltas LSA e simulação de faltas BRI com vectores de detecção múltipla LSA no Tetramax®, utilizando o *script* "4s5x2b.txt".

Depois faz-se a simulação de faltas *bridging* com vectores IDDQ no Verilog-XL®, utilizando o modo 3 de funcionamento e aproveitando os vectores IDDQ gerados no passo S2I do novo método de teste.

Finalmente, faz-se a simulação de faltas BRI com vectores LSA, gerados com detecção simples de faltas, executando o *script* "5s2b.txt" no Tetramax®.

O *script* "4s5x2b.txt" é bastante semelhante ao *script* utilizado no passo S2B, a diferença é que se faz ATPG para faltas LSA com detecção múltipla (5 vezes cada falta) antes de se fazer a simulação de faltas BRI com vectores de teste LSA. O comando que faz a ATPG LSA com detecção múltipla é o seguinte:

```
run atpg -auto_compression -ndetect 5
```

A opção “-ndetect 5” é a que indica ao Tetramax® que é para fazer a ATPG com detecção múltipla de faltas (cinco vezes cada falta).

A simulação no Verilog-XL® é feita do mesmo modo que a simulação no passo I2B do novo método de teste, só a lista de faltas é que é diferente, sendo utilizada a obtida com o *script* “4s5x2b.txt”.

O *script* “5s2b.txt” é basicamente igual ao utilizado no passo S2B, só a lista de faltas BRI utilizada (utiliza-se a resultante da simulação Verilog-XL® referido no parágrafo anterior a este) e o nome do ficheiro de saída é que são diferentes.

De seguida faz-se a descrição dos passos feitos na consola de comandos para o caso do circuito b18. Os *scripts* utilizados para este circuito são apresentados em anexo.

Considerando que o caminho dos programas utilizados já foi adicionado à *PATH* do sistema operativo e que o código PLI já foi compilado para o Verilog-XL®, os passos para executar os dois métodos de teste são os seguintes:

>design_vision –db_mode (o *script* é carregado dentro do programa)

>tmax –shell 1s2i.txt

-> Acrescentar no ficheiro b18_tb.v as chamadas de sistema $iddq_init, $iddq_bri e $iddq_close, copiar das primeiras linhas deste ficheiro a linha com a definição da “timescale” para a primeira linha do ficheiro b18_scan.v. Estas alterações são colocadas a negrito nos ficheiros apresentados em anexo. O $iddq_init tem os seguintes argumentos:

$iddq_init("b18_tmax_faults.txt", 386966, 3.7, 2);

O valor “3.7” foi escolhido como Kapa para calcular o número de faltas BRI a gerar, segundo a equação (2), pois após alguns testes foi com este valor que para circuitos relativamente grandes (como o b15) é obtido um número de vectores BRI semelhante ao número de vectores LSA.

O primeiro argumento de $iddq_init é o nome do ficheiro de faltas BRI resultante da execução do *script* 2s2b.txt. Neste ponto o ficheiro ainda não existe, mas inclui-se já para facilitar as alterações nesta chamada de sistema no passo I2B.

>verilog b18_scan.v b18_tb.v –v /soft/ams/3.7/verilog/c35b3/c35_CORELIB.v /soft/ams/3.7/verilog/c35b3/c35_UDP.v –l b18_ver_rand.txt

>mv fault_list_386966.00.txt b18_tmax_list_386966.txt

Faz-se esta cópia de modo a que se possa repetir o teste a partir deste ponto do método de teste, em vez de recomeçar do início. Sempre que se faz uma cópia deste género aplica-se o mesmo raciocínio.

>tmax –shell 2s2b.txt

-> Mudar na chamada de sistema $iddq_init do ficheiro b18_tb.v o modo de funcionamento de “2” para “3”:

$iddq_init("b18_tmax_faults.txt", 386966, 3.7, **3**);

>verilog b18_scan.v b18_tb.v
-v /soft/ams/3.7/verilog/c35b3/c35_CORELIB.v /soft/ams/3.7/verilog/c35b3/c35_UDP.v –l b18_i2b.txt
>mv fault_list_386966.00.txt b18_fault_list_386966.00.txt
>tmax –shell 3b.txt

Para preparar o teste do circuito b18 utilizando o método de teste da AMIS os seguintes passos são necessários:
>tmax –shell 4s5x2b.txt

->Mudar o nome do ficheiro de faltas *bridging* para o ficheiro gerado no fim do *script* 4s5x2b.txt:

$iddq_init(**"b18_bri_faults.txt"**, 386966, 3.7, 3);

>verilog b18_scan.v b18_tb.v –v /soft/ams/3.7/verilog/c35b3/c35_CORELIB.v /soft/ams/3.7/verilog/c35b3/c35_UDP.v –l b18_i2b_ami.txt
>mv fault_list_386966.00.txt b18_fault_list_386966_ami.txt
>tmax –shell 5s2b.txt

Deste modo prepara-se o teste do circuito b18 utilizando os dois métodos de teste considerados neste trabalho.

Sempre que se utiliza o Verilog-XL® para simular os circuitos sintetizados com o Design Vision®, ao ser lida a descrição do circuito aparece uma série de *warnings* desta categoria:

“Warning! Too few module port connections [Verilog-TFNPC]”

Este *warning* deve-se ao facto da Synopsys® definir as células (principalmente flip-flops) com mais pinos do que os que são necessários (por exemplo Q e Q negado, e só se utilizar Q).

Este *warning* é só um aviso, e não influencia negativamente a simulação de faltas.

Caso se queira desactivar este *warning*, é necessário adicionar a seguinte opção quando se executa o simulador:

verilog b18_scan.v b18_tb.v –v /soft/ams/3.7/verilog/c35b3/c35_CORELIB.v /soft/ams/3.7/verilog/c35b3/c35_UDP.v –l b18_i2b.txt **+nowarnTFNPC**

em que TFNPC é o tipo de *warning* que se quer desactivar.

Além de este *warning*, há outro que está relacionado com o ficheiro c35_UDP.v:

"Warning! primitive table : the entry is redundant -
line-number = 449 entry = rx?:?:x [Verilog-PTER]
"c35_UDP.v", 439: table"

Isto deve-se a um excesso de pessimismo na descrição do comportamento dos módulos da tecnologia c35 da AMS. Este é apenas um aviso, e não influencia a simulação de faltas negativamente. Para evitar que este *warning* apareça, basta adicionar "+nowarnPTER" quando se invoca o Verilog-XL®.

6.2. Resultados obtidos

Apresentam-se neste capítulo os resultados obtidos para todos os circuitos testados. As percentagens apresentadas dizem respeito a faltas detectáveis pelo Tetramax®.

Para facilitar a leitura deste capítulo, referem-se como *vectores X* os vectores gerados para detecção de faltas do tipo X.

Inicialmente apresentam-se os circuitos estudados. Em seguida apresentam-se alguns resultados intermédios que permitem efectuar um estudo comparativo da eficácia de cada passo no fluxo de preparação do teste para ambos os métodos.

A Tabela 6.1 apresenta o número de faltas LSA (valor obtido com o Tetramax®) e o número de faltas *bridging* considerado para cada circuito, sendo este último obtido multiplicando o número de faltas LSA desse circuito por "3,7".

Tabela 6. 1 - Número de faltas LSA e *Bridging* de cada circuito

Circuito	# faltas LSA	# faltas *bridging*
b04	3994	14778
b05	10044	37163
b07	2046	7570
b11	3352	12402
b12	9198	6006
b14	42598	157613
b15	38712	143234
b17	121178	448359
b18	386966	1431774
b22	158854	587760

No caso do circuito b12 não foi possível gerar o número de faltas *bridging* pedido (34032 faltas) porque não existem nós distintos suficientes para gerar o número de faltas *bridging* necessário, sendo apenas possível gerar com os 78 nós distintos existentes no circuito ($^{78}C_2$) x 2 = 6006 faltas. Isto deve-se principalmente ao facto de se usar apenas *simulated nets* como nós passíveis de ter faltas associadas, que existem em menor número que todos os nós (pinos de saída, entrada e *wires*) do circuito. Este facto acontece em todos os circuitos testados, mas

só neste caso é que influencia o número máximo de faltas *bridging* de modo a que não se possa gerar o número necessário.

A Tabela 6.2 apresenta os resultados intermédios de detecção de faltas obtidos com o novo método de teste, e a Tabela 6.3 apresenta os resultados intermédios obtidos com o método de teste da AMIS.

Na coluna "IDDQ<-LSA" das Tabelas 6.2 e 6.3 apresenta-se a percentagem de faltas IDDQ detectadas por vectores de teste LSA. Este raciocínio aplica-se às restantes colunas das Tabelas 6.2 e 6.3. A coluna "BRI<-IDDQ" da Tabela 6.2 apresenta a percentagem de BRI não detectadas pelos vectores LSA (coluna "BRI<-LSA") e que são detectados pelos vectores IDDQ. De igual forma, a coluna "BRI<-IDDQ" da Tabela 6.3 apresenta a percentagem de BRI não detectadas pelos vectores LSA com detecção múltipla (coluna "BRI<-LSA5x") e que são detectados pelos vectores IDDQ, e a coluna BRI<-LSA apresenta a percentagem de BRI não detectadas pelos vectores LSA com detecção múltipla nem pelos vectores IDDQ (coluna "BRI<-LSA5x" e "BRI<-IDDQ") e que são detectados pelos vectores LSA com detecção simples.

Nas Tabelas 6.2 e 6.3 pode-se observar que a detecção média de faltas IDDQ através de simulação de faltas com vectores de teste LSA é de 96,60% para os dois métodos de teste, sendo possível concluir que a grande maioria de faltas IDDQ de um circuito são detectadas por vectores LSA. Deste modo apenas é necessário fazer a ATPG para um número reduzido de faltas IDDQ, sendo gerado um número reduzido de vectores de teste IDDQ, o que é muito importante pois com a tecnologia actual o teste IDDQ é bastante dispendioso em termos de tempo.

A razão de se ter a mesma percentagem média de detecção com os dois métodos de teste deve-se ao facto de que nos dois métodos o teste é feito exactamente da mesma forma.

Tabela 6. 2 – Detecção de faltas parcial com o novo método de teste

Circuito	IDDQ <- LSA	BRI <- LSA	BRI <- IDDQ
b04	94,61%	63,15%	65,60%
b05	96,33%	85,20%	73,60%
b07	93,40%	76,12%	54,59%
b11	97,55%	75,70%	56,96%
b12	95,63%	85,95%	58,53%
b14	98,48%	73,46%	61,33%
b15	97,34%	74,18%	71,10%
b17	96,05%	72,50%	66,67%
b18	97,87%	73,69%	74,67%
b22	98,76%	68,73%	66,11%
Média	96,60%	74,87%	64,92%

Nas Tabelas 6.2 e 6.3 observa-se que a percentagem média de detecção de faltas BRI com vectores LSA para detecção múltipla, é de cerca de 82,90%. Assim, obtém-se uma cobertura de faltas 8% superior à obtida com os vectores que asseguram a detecção simples de faltas LSA. Isto deve-se ao facto de que, fazendo ATPG com detecção múltipla de faltas, obtém-se um número de vectores de teste LSA superior ao obtido com ATPG LSA sem detecção múltipla. Como há mais vectores de teste LSA, mais faltas BRI são detectadas.

Tabela 6. 3 - Detecção de faltas parcial com o método de teste utilizado pela AMIS

Circuito	IDDQ <- LSA	BRI <- LSA5x	BRI <- IDDQ	BRI <- LSA
b04	94,61%	68,64%	64,78%	7,86%
b05	96,33%	94,09%	74,13%	21,20%
b07	93,40%	91,28%	66,82%	8,62%
b11	97,55%	92,35%	66,71%	7,79%
b12	95,63%	84,54%	57,48%	6,37%
b14	98,48%	84,14%	64,28%	5,95%
b15	97,34%	78,83%	70,16%	1,72%
b17	96,05%	77,62%	65,87%	2,57%
b18	97,87%	79,66%	73,98%	3,86%
b22	98,76%	77,82%	66,54%	4,44%
Média	96,60%	82,90%	67,08%	7,04%

Tendo em conta os resultados das Tabelas 6.2 e 6.3, também se pode observar que uma grande percentagem de faltas BRI é detectada com vectores de teste LSA e IDDQ: em média 91,18% com detecção simples de LSA (novo método de teste) e 94,77% com detecção múltipla de faltas LSA (método de teste da AMIS). Contudo, ainda existe uma percentagem significativa de faltas BRI que ainda não foram detectadas, o que justifica a inclusão da ATPG para faltas BRI no novo método de teste de modo a aumentar a detecção de faltas BRI.

No que diz respeito à detecção parcial "BRI<-IDDQ", não se podem comparar os dois métodos de teste pois o número de faltas BRI inicial é diferente num método e no outro.

Na Tabela 6.3 é possível observar que, das faltas BRI que não foram detectadas pelas simulações de faltas com vectores LSA e IDDQ, são detectadas em média mais 7,04%, com os vectores LSA gerados para detecção simples de faltas LSA. O facto de os vectores LSA detectarem faltas BRI não detectadas pelos vectores LSA com detecção múltipla ocorre devido à conjugação frutuita das condições necessárias à activação dos vectores BRI e das atribuições aleatórias que ocorrem no final do processo de ATPG.

As Tabelas 6.4 e 6.5 apresentam a cobertura total de faltas e número total de vectores de teste obtidos com o novo método e com o método de teste da AMIS, respectivamente.

Tabela 6. 4 – Resultados totais com o novo método de teste

	Número total de vectores de teste			Cobertura de faltas total			
Circuito	LSA	IDDQ	BRI	LSA	IDDQ	BRI	
b04	51	9	121	89,09%	100,00%	90,22%	
b05	195	9	91	99,82%	100,00%	99,83%	
b07	63	18	41	100,00%	100,00%	99,95%	
b11	108	11	57	100,00%	100,00%	100,00%	
b12	175	12	31	100,00%	100,00%	99,20%	
b14	829	32	1804	99,14%	100,00%	99,43%	
b15	540	18	530	99,88%	100,00%	99,37%	
b17	549	16	1778	99,90%	100,00%	99,03%	
b18	1224	22	4205	99,75%	100,00%	99,22%	
b22	1241	11	5180	99,32%	100,00%	98,96%	
				98,69%	100,00%	98,52%	Média

Nas Tabelas 6.4 e 6.5 é possível observar que em média o número de faltas LSA e IDDQ detectadas tanto por um método como por outro é o mesmo, cerca de 98,69% para faltas LSA e 100% para faltas IDDQ.

No caso das faltas *bridging*, com o novo método de teste detecta-se em média 98,52% e com o método da AMIS 89,87%. Assim pode-se concluir que com o novo método de teste atinge-se uma maior cobertura de teste para faltas *bridging*, porque com o novo método de teste faz-se ATPG para faltas *bridging*, detectando assim uma percentagem das faltas *bridging* não detectadas por vectores de teste de outros modelos de faltas, o que não acontece no caso do método de teste da AMIS.

Tabela 6. 5 - Resultados totais com o método de teste da AMIS

	Número total de vectores de teste			Cobertura de faltas total				
Circuito	LSA 5x	LSA	IDDQ	LSA 5x	LSA	IDDQ	BRI	
b04	132	51	9	89,09%	89,09%	100,00%	81,28%	
b05	195	195	9	99,82%	99,82%	100,00%	98,05%	
b07	208	63	18	100,00%	100,00%	100,00%	94,68%	
b11	406	108	11	100,00%	100,00%	100,00%	96,47%	
b12	551	175	12	100,00%	100,00%	100,00%	94,37%	
b14	3605	829	32	99,15%	99,14%	100,00%	90,42%	
b15	2340	540	18	99,90%	99,88%	100,00%	85,42%	
b17	2205	549	16	99,91%	99,90%	100,00%	85,68%	
b18	4917	1224	22	99,74%	99,75%	100,00%	86,41%	
b22	5357	1241	11	99,31%	99,32%	100,00%	85,96%	
				98,69%	98,69%	100,00%	89,87%	Média

Na Tabela 6.6, o circuito b14 tem um número de vectores de teste em corrente (IDDQ) que parece ser superior ao que é normalmente utilizado, mas na realidade apenas 18 *patterns* IDDQ foram gerados para este circuito, o que está dentro do que é normalmente utilizado: 10 a 20 *patterns* de teste IDDQ. A razão de 18 *patterns* serem compostos por 32 vectores de teste deve-se ao facto de que em circuitos sequenciais com cadeias de *scan* são necessários vectores de teste para inicializar estas cadeias, outros vectores de teste para fazer a detecção de faltas e ainda outros para tornar as faltas observáveis nas saídas do circuito.

Observando a Tabela 6.6, chega-se à conclusão de que em todos os casos o número total de vectores de teste em tensão obtido com o novo método de teste é inferior ao obtido com o método de teste da AMIS, existindo casos em que é 2,6 (b07, b15) ou 3 (b11, b12) vezes inferior.

Tabela 6. 6 – Número total de vectores de teste em tensão e corrente para os dois métodos de teste estudados

	Novo método de teste (NTF)		Método de teste AMIS (ATF)		
Circuito	Número de vectores de teste em tensão (LSA + BRI)	Número de vectores de teste em corrente (IDDQ)	Número de vectores de teste em tensão (LSA5x + LSA1x)	Número de vectores de teste em corrente (IDDQ)	Relação entre o número de vectores em tensão (ATF/NTF)
b04	172	9	183	9	1,06
b05	286	9	390	9	1,36
b07	104	18	271	18	2,61
b11	165	11	514	11	3,12
b12	206	12	726	12	3,52
b14	2633	32	4434	32	1,68
b15	1070	18	2880	18	2,69
b17	2327	16	2754	16	1,18
b18	5429	22	6141	22	1,13
b22	6421	11	6598	11	1,03

No que diz respeito ao tempo de processamento, necessário para ATPG e simulação de faltas, de cada um dos passos de cada método de teste, nas Tabelas 6.7 e 6.8 é possível observar que globalmente os tempos requeridos por ambos os métodos são da mesma ordem de grandeza. De notar que os valores são apresentados em segundos, tal como o Tetramax® e o Verilog-XL® o indicam, e que estes valores não têm em conta o tempo necessário para ler a lista de faltas BRI nos passos "Simulação de faltas BRI com vectores LSA", "Simulação de faltas BRI com vectores IDDQ" e "ATPG BRI" do novo método de teste, leitura esta que adiciona um quantidade considerável de tempo ao tempo de preparação do teste do circuito em questão.

Tabela 6. 7 – Tempo de processamento do CPU com o Novo método de teste

	Novo método de teste				
Circuito \ Tempo de processamento do CPU(s)	ATPG LSA, simulação de faltas IDDQ e ATPG IDDQ	Simulação de faltas BRI com vectores LSA	Simulação de faltas BRI com vectores IDDQ	ATPG BRI	Total
b04	1146,65	0,21	1,20	1,15	1149,21
b05	48,52	0,22	1,00	1,00	50,74
b07	581,29	0,16	0,70	0,21	582,36
b11	2,71	0,14	0,80	0,22	3,87
b12	7,71	0,30	0,50	0,31	8,82
b14	2156,67	1,60	35,70	14,79	2208,76
b15	94,19	1,87	25,50	9,13	130,69
b17	301,66	6,20	1412,20	116,42	1836,48
b18	5477,95	24,36	7484,80	458,25	13445,36
b22	7087,95	20,35	2977,20	823,76	10909,26

Tabela 6. 8 – Tempo de processamento do CPU com o método de teste da AMIS

	Método de teste da AMIS		
Circuito \ Tempo de processamento do CPU(s)	ATPG LSA com detecção múltipla de faltas (5x)	ATPG LSA, simulação de faltas IDDQ e ATPG IDDQ	Total
b04	1094,25	1146,65	2240,90
b05	46,14	48,52	94,66
b07	0,40	581,29	581,69
b11	0,56	2,71	3,27
b12	1,05	7,71	8,76
b14	2153,09	2156,67	4309,76
b15	94,72	94,19	188,91
b17	314,06	301,66	615,72
b18	4634,57	5477,95	10112,52
b22	6542,30	7087,95	13630,25

Tendo em conta estes dados, de modo geral o novo método de teste necessita de mais tempo a completar a preparação do teste para cada circuito em comparação com o método de teste da AMIS, mas como com o novo método de teste o teste para cada circuito é optimizado em termos de vectores de teste e da cobertura de faltas atingida, justifica-se que se gaste mais tempo na preparação do teste.

7. Conclusão e Trabalhos Futuros

A análise dos resultados apresentados no capítulo anterior permite concluir que a metodologia de teste proposta é claramente mais eficiente do que o método baseado na detecção múltipla de faltas LSA em prática actualmente na AMIS.

Embora o nível de detecção de faltas LSA e IDDQ seja semelhante com os dois métodos, com o novo método de teste o nível de detecção de faltas *bridging* é bastante melhor. Em média são detectadas 8,65% mais faltas do que no método de teste da AMIS. Este facto deve-se à ATPG de faltas *bridging* realizado no novo método de teste.

Quanto ao número total de vectores de teste em tensão, com o novo método de teste obtém-se um número de vectores inferior ao obtido com o método de teste da AMIS, sendo em alguns casos 2,6 (b07, b15) ou 3 (b11, b12) vezes inferior. O número de vectores de teste é inferior uma vez que com o novo método de teste não é feita ATPG para LSA com detecção múltipla de faltas, apenas detecção simples, o que reduz bastante o número de vectores de teste. Contudo, com os vectores de teste gerados na ATPG para faltas BRI o número de vectores de teste em tensão é na maioria dos casos apenas ligeiramente inferior aos obtidos com o método de teste de AMIS.

No que diz respeito ao tempo de processamento necessário para concluir a preparação do teste de um circuito, chega-se à conclusão que o novo método de teste necessita de um tempo de processamento mais elevado que o necessário para o método da AMIS, sendo este facto devido principalmente ao modo de leitura da lista de faltas BRI para os programas comerciais. Contudo, como a preparação do teste para cada circuito apenas se faz uma vez, e como o teste de cada circuito integrado é optimizado em termos de vectores de teste e de cobertura de faltas, conclui-se que o tempo extra necessário à preparação do teste com o novo método de teste se justifica.

É então possível afirmar que o objectivo do trabalho foi atingido, tendo sido obtido um método de preparação do teste praticável com circuitos complexos pois é baseado em ferramentas comerciais, optimizado tanto em número total de vectores de teste gerados como na detecção de faltas com vários modelos, sendo assim adaptado às necessidades das tecnologias actuais.

Como trabalho futuro, há algumas áreas que podem ser investigadas com o objectivo de melhorar o novo método de teste:

- utilizar como lista de faltas bridging uma lista de faltas real, obtida com base em informação de capacidades parasitas do circuito;

- melhorar o modo como se introduz faltas *bridging* no Tetramax®, sem utilizar um comando de adição de falta por cada falta;

- estudar o porquê de em alguns circuitos se ter um número bastante inferior de vectores de teste em tensão em comparação com o obtido com o método de teste da AMIS e noutros circuitos se ter apenas um número ligeiramente inferior;

- devido a limitações relativas à simulação de faltas e à ATPG para faltas do tipo *bridging* e IDDQ na versão do Tetramax® utilizado, é possível que em versões futuras se obtenham melhores resultados;

- estudar a eficiência de fazer um método de teste em que se faz ATPG para LSA, depois para *bridging* e finalmente para IDDQ, fazendo sempre a simulação de faltas de um modelo de faltas com os vectores de teste de outro modelo de modo a eliminar faltas detectadas por esses vectores;

- adicionar mais modelos de faltas ao método de teste, como o modelo de transição (transição lenta de um valor lógico para outro num dado nó) e o modelo atrasos de caminho (que testa o caminho crítico do circuito, à velocidade máximo do circuito, de modo a verificar se houve defeitos de fabrico, por exemplo);

Referências bibliográficas

[1] Kohei Miyase, Kenta Terashima, Seiji Kajihara, Xiaoqing Wen, Sudhakar M. Reddy, "On Improving Defect Coverage of Stuck-at Fault Tests," ats, pp. 216-223, 14th Asian Test Symposium (ATS'05), 2005

[2] Arumi, Rodriguez-Montanes, Figueras, Eichenberger, Hora, Kruseman, Lousberg, Majhi, "Diagnosis of Bridging Defects Based on Current Signatures at Low Power Supply Voltages," vts, pp. 145-150, 25th IEEE VLSI Test Symmposium (VTS'07), 2007

[3] Peter C. Maxwell, Robert C. Aitken, Vic Johansen, Inshen Chiang, "The Effectiveness of IDDQ, Functional and Scan Tests: How Many Fault Coverages Do We Need?", International Test Conference, IEEE, Paper 6.3, pp.168 a 177, 1992.

[4] Vijay R. Sar-Dessai, D. M. H. Walker, "Resistive Bridge Fault Modeling, Simulation and Test Generation", International Test Conference, 1999.

[5] David B. Lavo, "A Comprehensive Look at VLSI Fault Diagnosis", 19 de Julho de 2002.

[6] R. Rodriguez-Montanes, E. M. J. G. Bruls, J. Figueras, "Bridging Defect Resistance Measurements in a CMOS Process," *International Test Conference*, pp.892 a 899, 1992.

[7] E. Isern, J. Figueras, "IDDQ Test and Diagnosis of CMOS Circuits", IEEE Design & Test of Computers, IDDQ Series, pp.60 a 67, 1995.

[8] *Tetramax® ATPG User Guide*, Synopsys®, Pacote de manuais de instalação de ferramentas da Synopsys®, Versão Y-2006.06, Junho 2006.

[9] *PLI 1.0 User Guide and Reference*, Fevereiro 1997.

[10] *VERILOG-HDL PLI Reference Manual*, Open Verilog International, Version 1.0, Novembro 1991.

[11] http://www.asic-world.com/verilog/index.html

[12] Alain Vachoux, "Top-down digital design flow", Microelectronic Systems Lab, STI-IMM-LSM, alain.vachoux@epfl.ch, versão 3.0.2, 6 de Dezembro de 2005.

Anexos

Synopsys.init

```
export SYNOPSYS=/soft/synopsys/syn
export SYNDIR=/soft/synopsys/syn
export TMAXDIR=/soft/synopsys/tmx
export AMS_DIR=/soft/ams/3.7
export TMAX_32BIT=1
export
PATH=$PATH:$SYNDIR/bin:$SYNDIR/linux/bin:$TMAXDIR/bin:$TMAXDIR/linux/bin:$TMAXDIR/linux/
syn/bin
export LM_LICENCE_FILE=/soft/synopsys/scl/admin/licence/centos.dat
export SNPSLMD_LICENSE_FILE=/soft/synopsys/scl/admin/licence/centos.dat
```

Cadence.init

export CDSDIR=/soft/Cadence/ius

export CDS_INST_DIR=/soft/Cadence/ius

export CADDIR=/soft/Cadence

export AMS__DIR=/soft/Cadence/ius

export CDS_Netlisting_Mode Analog

export ASSURAHOME=$CADDIR/Assura

export LANG=C

export LD_LIBRARY_PATH=/soft/Cadence/ius/tools/lib:${LD_LIBRARY_PATH}

export CDS_LIC_FILE=/soft/Cadence/lic.dat

export LM_LICENSE_FILE=/soft/Cadence/lic.dat

export

PATH=$CADDIR/Aptivia/tools.lnx86/bin:$CADDIR/Aptivia/tools.lnx86/acv/bin:$CADDIR/Assura/tools.lnx86/bin:$CADDIR/Assura/tools.lnx86/assura/bin:$CADDIR/Assura/tools.lnx86/dfII/bin:$CADDIR/Assura/tools.lnx86/stream_mgt/bin:$CDSDIR/tools.lnx86/bin:$CDSDIR/tools.lnx86/dfII/bin:$CDSDIR/tools.lnx86/plot/bin:$CADDIR/Ldv/tools.lnx86/bin:$CADDIR/Ldv/tools.lnx86/vtools.lnx86/vfault/bin:$AMS_DIR/artist/bin:/home/nando/Soft/VeriDOS:$CADDIR/Ldv/tools.lnx86/vtools/vfault/bin:$PATH

.synopsys_dc.setup

```
set AMS_DIR       get_unix_variable(AMS_DIR)
set SYNOPSYS    get_unix_variable(SYNOPSYS);
set search_path     {. /soft/ams/synopsys/c35_3.3V /soft/synopsys/syn/libraries/syn
/soft/synopsys/syn/dw/sim_ver}

set target_library  {/soft/ams/3.7/synopsys/c35_3.3V/c35_CORELIB.db
/soft/ams/3.7/synopsys/c35_3.3V/c35_IOLIBV5_4M.db /soft/ams/3.7/synopsys/c35_3.3V/c35_IOLIB_4M.db};
set symbol_library  {/soft/ams/3.7/synopsys/c35_3.3V/c35_CORELIB.sdb
/soft/ams/3.7/synopsys/c35_3.3V/c35_IOLIBV5_4M.sdb
/soft/ams/3.7/synopsys/c35_3.3V/c35_IOLIB_4M.sdb};

set link_library   {* /soft/ams/3.7/synopsys/c35_3.3V/c35_CORELIB.db
/soft/ams/3.7/synopsys/c35_3.3V/c35_IOLIBV5_4M.db /soft/ams/3.7/synopsys/c35_3.3V/c35_CORELIB.sdb
/soft/ams/3.7/synopsys/c35_3.3V/c35_IOLIBV5_4M.sdb /soft/ams/3.7/synopsys/c35_3.3V/c35_IOLIB_4M.db
/soft/ams/3.7/synopsys/c35_3.3V/c35_IOLIB_4M.sdb};

set verilogout_equation      "false";
set verilogout_no_tri          "true"
set write_name_nets_same_as_ports             "true"
set verilogout_single_bit    "false"
set hdlout_internal_busses "true"
set bus_inference_style      "%s[%d]";
set sdfout_no_edge            "true"
```

Circuito b18

dv_b18.tcl

```
analyze -library WORK -format vhdl {b18.vhd}
elaborate B18 -architecture BEHAV -library WORK
set_fix_multiple_port_nets -all
create_clock -name "clock" -period 100 -waveform { 0 50  }  { clock  }
set_operating_conditions -library c35_CORELIB WORST-IND
set compile_seqmap_propagate_constants false
printvar compile_seqmap_propagate_constants
uplevel #0 compile -map_effort medium -area_effort none -only_design_rule
check_test
preview_scan
insert_scan
check_scan
write -hierarchy -format db -output b18_scan.db
estimate_test_coverage -sample 100
change_names -hierarchy -rules verilog -verbose
write -hierarchy -format verilog -output b18_scan.v
check_scan
write_test_protocol -format stil -out b18_scan.spf
```

1s2i.txt

```
set messages log b18_s2i.log -replace

set netlist -sequential_modeling -pin_assign -scalar_net
read netlist c35_CORELIB.v -library -delete
read netlist c35_UDP.v -library
read netlist b18_scan.v
set build -nodelete_unused -merge notied_gates_with_pin_loss
set learning -disable_time_limit
run build_model b18
set drc b18_scan.spf -trace -allow_unstable_set_resets
run drc
report summaries library_cells optimizations primitives
```

```
set atpg -capture_cycles 4 -full_seq_atpg -verbose
set atpg -merge high -full_seq_merge high
remove faults -all
set faults -model stuck
add faults -all
run atpg -auto_compression
set simulation -verbose
run simulation -sequential
write patterns b18_stuck_pat.bin -replace -internal -format binary
write patterns b18_patterns_stuck -replace -internal -format verilog_tables -serial

remove faults -all
set faults -model iddq
add faults -all
set patterns external b18_stuck_pat.bin
run simulation -sequential
run fault_sim
set patterns internal
run atpg -auto_compression
set simulation -verbose
run simulation
write patterns b18_iddq_pat.bin -replace -internal -format binary
write patterns b18_tb.v -replace -internal -format verilog_single_file -parallel 1
write patterns b18_patterns_iddq -replace -internal -format verilog_tables –serial
```

2s2b.txt

```
set messages log b18_s2b.log -replace

set netlist -sequential_modeling -pin_assign -scalar_net
read netlist c35_CORELIB.v -library -delete
read netlist c35_UDP.v -library
read netlist b18_scan.v
set build -nodelete_unused -merge notied_gates_with_pin_loss
set learning -disable_time_limit
run build_model b18
set drc b18_scan.spf -trace -allow_unstable_set_resets
run drc
report summaries library_cells optimizations primitives
```

```
set faults -model bridging
set faults -bridge_inputs
source b18_tmax_list_386966.txt
//read faults list_tmax.txt
set patterns external b18_stuck_pat.bin -sensitive
run simulation
run fault_sim -strong_bridge
write faults b18_tmax_faults.txt -replace -class NO
```

3b.txt

```
set messages log b18_b.log -replace

set netlist -sequential_modeling -pin_assign -scalar_net
read netlist c35_CORELIB.v -library -delete
read netlist c35_UDP.v -library
read netlist b18_scan.v
set build -nodelete_unused -merge notied_gates_with_pin_loss
set learning -disable_time_limit -verbose
run build_model b18
set drc b18_scan.spf -trace -allow_unstable_set_resets
run drc
report summaries library_cells optimizations primitives
set faults -model bridging
set faults -bridge_inputs
source b18_fault_list_386966.00.txt
set atpg -optimize_bridge_strengths -capture_cycles 4
set atpg -merge high -full_seq_merge high
set patterns internal
run atpg -auto_compression
write patterns b18_bridging_pat.bin -replace -internal -format binary
write patterns b18_patterns_bri -replace -internal -format verilog_tables –serial
```

4s5x2b.txt

```
set messages log b18_s5x2b.log -replace

set netlist -sequential_modeling -pin_assign -scalar_net
```

```
read netlist c35_CORELIB.v -library -delete
read netlist c35_UDP.v -library
read netlist b18_scan.v
set build -nodelete_unused -merge notied_gates_with_pin_loss
set learning -disable_time_limit
run build_model b18
set drc b18_scan.spf -trace -allow_unstable_set_resets
run drc
report summaries library_cells optimizations primitives

set faults -model stuck
add faults -all
set atpg -capture_cycles 4 -full_seq_atpg -verbose
set atpg -merge high -full_seq_merge high
run atpg -auto_compression -ndetect 5
set simulation -verbose
run simulation -sequential
write patterns b18_stuck_pat_ami.bin -replace -internal -format binary
write patterns b18_pat_stuck_ami -replace -internal -format verilog_tables -serial

remove faults -all
set faults -model bridging
set faults -bridge_inputs
source b18_tmax_list_386966.txt
set patterns external b18_stuck_pat_ami.bin -sensitive
run simulation
run fault_sim -strong_bridge
write faults b18_bri_faults.txt -replace -class NO
```

5s2b.txt

```
set messages log b18_s2b_ami.log -replace

set netlist -sequential_modeling -pin_assign -scalar_net
read netlist c35_CORELIB.v -library -delete
read netlist c35_UDP.v -library
read netlist b18_scan.v
set build -nodelete_unused -merge notied_gates_with_pin_loss
set learning -disable_time_limit
```

```
run build_model b18
set drc b18_scan.spf -trace -allow_unstable_set_resets
run drc
report summaries library_cells optimizations primitives

remove faults -all
set faults -model bridging
set faults -bridge_inputs
source b18_fault_list_386966_ami.txt
set patterns external b18_stuck_pat.bin -sensitive
run simulation
run fault_sim -strong_bridge
write faults b18_bri_final.txt -replace -class NO
```

b18_scan.v (alterações após síntese no Design Vision®)

No início do ficheiro:

“**`timescale 1 ns / 1 ns**

module b18_DW02_mult_2 (A, B, TC, PRODUCT);
(…)”

Só o que está a negrito é que representa as modificações. Não foram feitas mais alterações no ficheiro.

b18_tb.v (localização da inserção das chamadas de sistema PLI)

Foram feitas as seguintes modificações:

“(…)
always @ IDDQ begin
  `ifdef tmax_iddq
    $ssi_iddq("strobe_try");
    $ssi_iddq("status drivers leaky AAA_tmax_testbench_1_16.leaky");
  `endif
  $iddq_bri;

```
  end
(…)”
```

Dentro do bloco “initial”:

```
    “(…)
     // --- IDDQ PLI initialization
    //     User may activite by using '+define+tmax_iddq' on verilog compile line.
    //     Or by defining `tmax_iddq in this file.
    //
```

$iddq_init("b18_tmax_faults.txt", 386966, 3.7, 3); (A posição é aqui, o conteúdo muda consoante o passo do método de teste)

```
    `ifdef tmax_iddq
(…)”
```

Finalmente, no fim do ficheiro:

```
“(…)
$display("// %t : Simulation of %0d patterns completed with %0d errors\n", $time, pattern+1, nofails);
    $iddq_close;
    if (verbose >=2) $finish(2);
    /* else */ $finish(0);
  end
endmodule”
```