

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

ESCOLA DE ENGENHARIA

Curso de Pós-Graduação em Engenharia Elétrica - CPGEE

**TESTE DE SISTEMAS INTEGRADOS UTILIZANDO
CONTROLADORES ESPECÍFICOS**

LEANDRO JOSÉ CASSOL

Dissertação para obtenção do título de Mestre em Engenharia

Porto Alegre

2002

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

ESCOLA DE ENGENHARIA

Curso de Pós-Graduação em Engenharia Elétrica - CPGEE

**TESTES DE SISTEMAS INTEGRADOS UTILIZANDO
CONTROLADORES ESPECÍFICOS**

LEANDRO JOSÉ CASSOL

Engenheiro Eletricista

Dissertação apresentada ao Curso de Pós-Graduação em Engenharia Elétrica - CPGEE, como parte dos requisitos para a obtenção do título de Mestre em Engenharia.

Área de concentração: Instrumentação Eletro-Eletrônica.
Desenvolvida no Laboratório de Prototipação e Testes do Departamento de Engenharia Elétrica da Universidade Federal do Rio Grande do Sul.

Porto Alegre

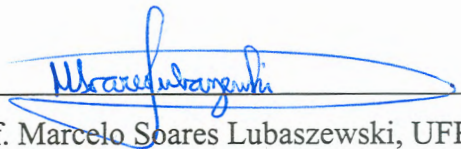
2002

TESTE DE SISTEMAS INTEGRADOS UTILIZANDO CONTROLADORES ESPECÍFICOS

LEANDRO JOSÉ CASSOL

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____



Prof. Marcelo Soares Lubaszewski, UFRGS

Dr. pelo Institut National Polytechnique de Grenoble

Banca Examinadora:

Prof. Marius Strum, USP

Dr. pela Universidade de São Paulo – USP – 1983

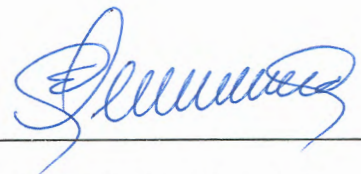
Prof. Luigi Carro, UFRGS

Dr. pela Universidade Federal do Rio Grande do Sul – UFRGS – 1996

Prof. Renato Perez Ribas, UFRGS

Dr. pelo Institut National Polytechnique de Grenoble – INPG – 1998

Coordenador do CPGEE: _____



Prof. Dr. Carlos Eduardo Pereira

Porto Alegre, março de 2002.

*A minha família, em especial a minha mãe
Maria Helena, minhas irmãs Nabel e Rosângela e
minha filha Rafaela pelo apoio, carinho, incenti-
vo e amor.*

AGRADECIMENTOS

É uma tarefa que me agrada e que, no entanto, considero ingrata por ter que agradecer em um curto espaço de tempo a tantas pessoas que de uma maneira ou outra me ajudaram em um percurso longo de êxitos, alguns fracassos e muito trabalho.

Começo por agradecer à CAPES pela provisão da bolsa de mestrado, ao DELET pelas condições de trabalho e a todos os Professores do Curso de Pós-Graduação em Engenharia Elétrica – CPGEE que se dispuseram a passar um pouco dos seus conhecimentos. Em especial ao Professor Dr. Marcelo Lubaszewski a quem desejo expressar a minha profunda gratidão pela paciência, orientação e motivação destes anos. De igual modo agradeço ao Professor Dr. Luigi Carro e a Érika Cota, pela disponibilidade e ajuda.

Aos Professores que fazem parte da banca examinadora por terem aceito o convite e por último desejo manifestar meus agradecimentos a todos os meus colegas pelo ótimo convívio, amizade e ajuda, em especial aos amigos Fabiano Toson, Erik Schüller, Osvaldo Betat, Adriane Parraga, Maria da Glória Flores (Dodóia), Rafael Zeilmann e Ronaldo Husemann.

SUMÁRIO

LISTA DE FIGURAS.....	viii
LISTA DE TABELAS	x
LISTA DE ABREVIATURAS E SÍMBOLOS.....	xi
RESUMO.....	xii
ABSTRACT.....	xiii
1 INTRODUÇÃO	1
1.1 MOTIVAÇÃO	1
1.2 OBJETIVOS DO TRABALHO.....	3
1.3 ORGANIZAÇÃO DA DISSERTAÇÃO.....	4
2 TESTES EM CIRCUITOS INTEGRADOS.....	5
2.1 SISTEMAS INTEGRADOS EM UM ÚNICO CHIP.....	5
2.2 NÚCLEOS DE HARDWARE	6
2.3 EXIGÊNCIAS DO TESTE EM SISTEMAS INTEGRADOS	8
2.4 ARQUITETURA DE TESTE CONCEITUAL.....	11
2.5 MECANISMO DE ACESSO AO TESTE	12
2.6 PADRÃO IEEE P1500.....	13
2.6.1 Linguagem de Descrição do Teste de Núcleos de Hardware.....	14
2.6.2 Arquitetura Escalonável	15
2.7 PADRÃO IEEE STD. 1149.1 (<i>BOUNDARY-SCAN</i>).....	18
3 CONTROLADORES DE TESTE.....	21
3.1 CUSTOS E PRECISÃO DE EQUIPAMENTOS AUTOMÁTICOS EXTERNOS DE TESTE (ATES).	22
3.2 CONTROLADORES INTERNOS	24
3.2.1 Microprocessadores Reutilizados para Produzir Controladores de Teste	24
3.2.2 Microprocessadores Dedicados ao Controle de Teste.....	32
4 O CONTROLADOR DE TESTE MET	37
4.1 LINGUAGEM STIL	38

4.2	PROPOSTA DA ARQUITETURA MET	40
4.2.1	Instruções do Microcontrolador	40
4.3	COMPONENTES PRINCIPAIS DA ARQUITETURA MET	43
4.3.1	Memória de Programa	44
4.3.2	Memória de Dados	45
4.3.3	Contador de Eventos	46
4.3.4	Comparador	47
4.3.5	Controle	47
4.4	EXECUÇÃO DO TESTE	49
4.5	EXEMPLO DE PROGRAMA DE TESTE <i>SCAN</i> PARA UM NÚCLEO BASEADO EM JTAG	50
4.6	RESULTADOS ENTRE SÍNTESES	52
5	ESTUDO DE CASOS.....	54
5.1	CONTROLANDO TESTE EXTERNO.....	55
5.2	CONTROLANDO TESTE <i>SCAN</i>	58
5.3	CONTROLE DO TESTE COM VETORES COMPACTADOS.....	65
5.3.1	Codificação Golomb e Compactação	66
5.3.2	Descompactação	68
5.4	TESTE <i>SCAN</i> PARALELO DE NÚCLEOS.	74
5.5	IMPACTO DO TAMANHO DA MEMÓRIA NO TESTE	75
5.6	CONTROLANDO TESTE BIST NO 8051	82
5.7	WRAPPERS PARAMETRIZADOS	87
5.8	OPERAÇÕES DE SALTOS NOS ENDEREÇADORES DAS MEMÓRIAS	90
6	CONCLUSÕES.....	92
	REFERÊNCIAS BIBLIOGRÁFICAS	94
	ANEXOS.....	100

LISTA DE FIGURAS

Figura 2.1 - Sistema integrado que utiliza hierarquia de blocos funcionais e Lógica Definida pelo Usuário (LDU) [ZOR 97].....	8
Figura 2.2 - Comparação entre os testes de uma placa e de um sistema integrado [ZOR 97]. ..	9
Figura 2.3 - Arquitetura conceitual para o teste de sistemas integrados [ZOR 98].....	12
Figura 2.4 - Sistema integrado com núcleos utilizando o padrão P1500 [ADH 99].	16
Figura 2.5 - Arquitetura conceitual do <i>wrapper</i> segundo IEEE P1500 [MAR 00b].	17
Figura 2.6 - Arquitetura IEEE std. 1149.1.....	19
Figura 2.7 - Diagrama de estados do controlador de TAP.	20
Figura 3.1 - Esquema de teste proposto em [PAP 99].....	25
Figura 3.2 - Diagrama de blocos da transferência de dados de teste do testador para o <i>chip</i> . .	29
Figura 3.3 - Palavra de substituição.	31
Figura 3.4 - Diagrama de blocos da arquitetura de teste proposto por [JAS 99].....	31
Figura 3.5 - Arquitetura BIST utilizando controlador interno [DRE 98].	33
Figura 3.6 - Diagrama de blocos do seqüenciador.	33
Figura 3.7 - Exemplo de estrutura de teste em forma de anel.	35
Figura 4.1 - Exemplo de programa STIL.	39
Figura 4.2 - Segundo vetor do bloco <i>Pattern</i>	40
Figura 4.3 - Arquitetura MET [COT 01].....	43
Figura 4.4 - Palavra da memória de dados.	45
Figura 4.5 - Memória de programa e de dados para o exemplo de código STIL.	46
Figura 4.6 - Diagrama de estados do bloco de controle do MET.....	48
Figura 4.7 - Memória de programa e de dados com valores para um teste <i>scan</i>	51
Figura 5.1 - Sistema com controle de teste interno.	54
Figura 5.2 - Diagrama de estados do bloco de controle com instrução <i>at-speed</i>	56
Figura 5.3 - Simulação para um teste externo <i>at-speed</i>	58
Figura 5.4 - Esquema utilizado para geração do sinal de TCK.	59
Figura 5.5 - Organização dos dados de teste scan nas memórias do MET.....	60
Figura 5.6 - Organização dos dados para um teste <i>scan</i> utilizando três memórias.	62
Figura 5.7 - Diagrama de estado.....	64
Figura 5.8 - Simulação de um teste <i>scan</i> utilizando 3 memórias.	65
Figura 5.9 - T_{diff} e o vetor codificado T_E	68
Figura 5.10 - Diagrama em blocos do decodificador usado para descompactação.	69
Figura 5.11 - Diagrama de estados de um decodificador Golomb com $m = 4$	69
Figura 5.12 - Arquitetura de descompactação baseada em CSR.	70
Figura 5.13 - Arquitetura de descompactação baseada no uso de cadeia scan.....	70
Figura 5.14 - Simulação com sinal de espera no processo de teste.	72
Figura 5.15 - Arquitetura para o teste de vários núcleos a partir de uma memória Scan com vetores compactados.	73
Figura 5.16 - Carga paralela de várias cadeias com o uso de um módulo de memória.....	74
Figura 5.17 - Sistema utilizado como exemplo.	76
Figura 5.18 - Organização dos dados na memória para o teste do circuito s838.	78
Figura 5.19 - Variação do tempo de teste em função da largura de palavra da memória.	79

Figura 5.20 - Variação na quantidade de conexões em função da largura da memória.	79
Figura 5.21 - Planejamento de tempo de teste atendendo restrições de projeto.	82
Figura 5.22 - (a) diagrama em blocos da descrição BIST 8051. (b) diagrama em blocos da descrição BIST 8051 com o controlador MET.	83
Figura 5.23 - Diagrama de estados do bloco de controle com passo variável.	87
Figura 5.24 - Arquitetura do registrador de instruções do <i>wrapper</i>	88
Figura 5.25 - Formato das instruções de salto.	90
Figura 5.26 - Diagrama de estados do bloco de controle com instruções de saltos.	90

LISTA DE TABELAS

Tabela 2.1 - Exemplos de núcleos de hardware classificados por funcionalidade [GUP97].	7
Tabela 3.1 - Parâmetros para o cálculo de custos de ATEs.....	23
Tabela 3.2 - Rendimento <i>versus</i> precisão do testador.	23
Tabela 4.1 - Codificação das instruções.	42
Tabela 4.2 - Primeira seqüência de definição dos sinais.	50
Tabela 4.3 - Segunda seqüência de definição dos sinais.	51
Tabela 4.4 - Comparativo de performance e área entre circuitos.....	52
Tabela 5.1 - Dados sobre os circuitos <i>benchmarks</i> s298 e s344.....	56
Tabela 5.2 - Comparativo da quantidade de bits de memória para teste <i>scan</i>	63
Tabela 5.3 - Exemplo de codificação Golomb para $m = 4$	67
Tabela 5.4 - Comparação entre freqüência interna e freqüência externa para descompactação Golomb.	73
Tabela 5.5 - Dados referentes ao processo de teste dos <i>benchmarks</i> do sistema exemplo da figura 5.16.	76
Tabela 5.6 - Ciclos e hardware extra para uma conexão exclusiva com um ATE.	77
Tabela 5.7 - Tempo de teste para cada bloco do microcontrolador 8051.....	85
Tabela 5.8 - Codificação das instruções.	86
Tabela 5.9 - Comparação entre modelos de configurações de <i>wrappers</i> [LAZ 02].....	89
Tabela 5.10 - Comparativo entre diversas implementações do MET.....	91

LISTA DE ABREVIATURAS E SÍMBOLOS

ASIC	<i>Application Specific Integrated Circuit</i>
ATE	<i>Automatic Test Equipment</i>
BIST	<i>Built-In Self-Test</i>
CI	<i>Circuito Integrado</i>
CPU	<i>Central Processing Unit</i>
CTL	<i>Core Test Language</i>
DFT	<i>Design for Testability</i>
DMA	<i>Direct Memory Access</i>
DRAM	<i>Dynamic Random Access Memory</i>
DSP	<i>Digital Signal Processing</i>
FPGA	<i>Field Programmable Gate Array</i>
FSM	<i>Finite State Machine</i>
IEEE	<i>Institute of Electrical and Electronic Engineers</i>
JTAG	<i>Joint Test Action Group</i>
LDU	<i>Lógica Definida pelo usuário</i>
LFSR	<i>Linear Feedback Shift Register</i>
MET	<i>Microprocessor for Embedded Test</i>
MISR	<i>Multiple Input Shift Register</i>
MPEG	<i>Motion Pictures Experts Group</i>
PCB	<i>Printed Circuit Board</i>
RAM	<i>Random Access Memory</i>
RISC	<i>Reduced Instruction Set Computer</i>
ROM	<i>Ready Only Memory</i>
SOC	<i>System on Chip</i>
SRAM	<i>Static Random Access Memory</i>
TAP	<i>Test Access Port</i>
TDI	<i>Test Data Input</i>
TDO	<i>Test Data Output</i>
TMS	<i>Test Mode Select</i>
UART	<i>Universal Asynchronous Receiver/Transmitter</i>
ULA	<i>Unidade Lógica e Aritmética</i>
VHDL	<i>Very high speed integrated circuits (VHSIC) Hardware Description Language</i>
VSI	<i>Virtual Socket Interface</i>
WRI	<i>Wrapper Register Instruction</i>

RESUMO

O presente trabalho tem como objetivo a avaliação do controle interno do teste em sistemas baseados em núcleos de hardware. No intuito de analisar os problemas e as exigências do teste em SOCs, alguns sistemas são aqui criados utilizando-se a descrição VHDL de um controlador de teste, alguns circuitos *benchmarks* e uma descrição de um microcontrolador 8051 auto-testável. Problemas referentes ao controle de diferentes estratégias de teste (externo, scan, BIST, etc) são abordados e formas de resolver estes problemas são descritas. Também abordam-se problemas referentes ao teste em nível de sistema, como por exemplo, requisitos de memória e conexões. Mudanças são sugeridas e implementadas no controlador de teste, a fim de melhorar seu desempenho e flexibilizar seu uso em diversas circunstâncias distintas em termos de requisitos de estratégias de teste.

ABSTRACT

This work aims at evaluating the internal test control in core-based systems. In order to analyze problems and requirements of testing core-based systems, some systems are herein built making use of a VHDL description of a test controller, of some benchmark circuits and of a description of a self-testing 8051 microcontroller. Problems related to controlling different test strategies (external testing, scan, BIST, etc) are covered and ways of solving those problems are described. Problems related to the system level testing, such as memory and connection requirements, are also discussed. Changes are proposed and implemented into the test controller, in order to enhance its performance and make its use more flexible to face many different situations in terms of required test strategies.

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

O uso de sistemas eletrônicos tornou-se imprescindível para a sustentação e evolução da sociedade moderna. A contínua busca pela modernização tem levado ao desenvolvimento de sistemas que se tornam cada dia menores e mais complexos. A competitividade entre os fabricantes de semicondutores, a necessidade tecnológica de alta velocidade, a diversidade de aplicações e o anseio da sociedade por novos produtos tem conduzido os fabricantes a buscarem formas alternativas de diminuir o tempo entre o projeto e a disponibilização dos seus produtos para os consumidores.

Os avanços em termos de aumento da capacidade de integração de *chips* levaram os fabricantes de circuitos integrados a solucionar parte de seus problemas através do reuso de módulos previamente projetados. Nos dias de hoje, diferentes tipos de módulos podem ser agrupados para juntos formarem um sistema completo em um único chip. Estes sistemas são chamados de SOCs (do inglês, *System-on-Chip*) e os módulos que os compõem recebem o nome de núcleos de hardware. Estes núcleos não são vendidos como ASICs manufaturados em uma pastilha de silício, mas sim, como uma descrição lógica sintetizável. A existência destes componentes permite ao projetista desenvolver um sistema complexo preocupando-se apenas com as questões de comunicação entre seus módulos, sem considerar detalhes de implementação de cada núcleo, assim, diminuindo consideravelmente o tempo de projeto do sistema.

Ao mesmo tempo em que os SOCs trouxeram um impressionante ganho na concepção de novos sistemas, também trouxeram um conjunto de novos problemas relativos ao teste. Este novo desafio inclui o teste de núcleos com diferentes funcionalidades, núcleos oriundos de diferentes fabricantes, restrições de acesso ao conteúdo dos núcleos por motivo de propriedade intelectual, acessibilidade limitada aos núcleos por intermédio das entradas e saídas primárias do sistema, controle do procedimento de teste de todo o sistema, etc.

Para resolver estes problemas são necessárias novas arquiteturas capazes de administrar o teste de mais de 100 milhões de transistores, ao mesmo tempo em que permitam atingir uma alta cobertura dos defeitos que podem se produzir no sistema. No intuito de facilitar a geração do teste, um grupo de trabalho IEEE está desenvolvendo um padrão chamado de P1500. Este grupo de trabalho tem focado seus esforços nos elementos que geram os estímulos e que comparam respostas de teste, no mecanismo que transporta os dados de teste entre fontes, núcleos e receptores, e principalmente na definição de uma lógica envoltória padrão para realizar o interfaceamento entre o núcleo e os mecanismos de teste. O padrão propõe, ainda, a definição de uma linguagem de descrição de teste para facilitar a comunicação entre os provedores dos núcleos de hardware e seus usuários.

Dois problemas importantes relativos ao teste de SOCs são: o acesso aos núcleos de hardware e a combinação da capacidade de teste e diagnóstico dos diferentes núcleos que integram o sistema. Além disso, as exigências de teste devem ser atendidas dada uma quantidade limitada de recursos (geradores de padrões, controladores de auto-teste, pinos extra no chip, área extra em silício, tempo de teste, etc). O uso dos tradicionais equipamentos automáticos de teste pode representar outra importante limitação, devido à largura dos mecanismos de acesso aos núcleos (TAM, do inglês, *Test Access Mechanism*) e da frequência de relógio utilizada no teste. Normalmente, para TAMs com larguras grandes são utilizadas cadeias de varredura (*scan*), as quais limitam a velocidade e também impõem restrições adicionais quanto ao número de vetores de teste.

A solução clássica empregada para suprir a limitação dos recursos de teste é o reuso de hardware. Por exemplo, testar mais de um núcleo de hardware utilizando o mesmo gerador de padrões de teste, ou o mesmo barramento, são estratégias que foram propostas para reduzir área em silício. Controladores de teste também devem ser considerados por ocasião da definição da estratégia de teste do sistema. Normalmente, esta tarefa é executada por um ATE. Porém, o uso destes equipamentos pode encontrar algumas restrições para o controle de teste em SOCs.

A associação internacional de fabricantes de semicondutores Roadmap (ITRS) [ROA 01] prevê que o custo de testadores de alta velocidade vai exceder a casa dos 20 milhões de dólares em 2010. Outro dado importante que é fornecido por essa mesma associação é que a velocidade dos circuitos integrados possui um incremento de 30% ao ano, enquanto a precisão dos testadores tem aumentado a uma taxa de somente 12% ao ano [ROA

01]. Por isso, modos alternativos que exigem menos dos testadores e mantenham a alta qualidade do teste devem ser buscados.

Alguns autores têm sugerido o uso de processadores e memórias disponíveis no próprio sistema para serem utilizados como testadores e elementos armazenadores de dados de teste. Em um ambiente SOC, esta é uma solução interessante, uma vez que, freqüentemente, estes circuitos possuem um ou mais processadores, além de núcleos de memória em abundância.

O uso de um processador dedicado para o teste é outra solução proposta para controladores integrados. Estes processadores podem ser muito eficientes em termos de área e desempenho, já que são dedicados e sintetizados com a mesma tecnologia do sistema sob teste. Assim, o controlador de teste é outro núcleo dentro do SOC e o reuso de hardware é conseguido através da utilização de memórias disponíveis no sistema para armazenar os padrões de teste.

Critérios para a tomada de decisão quanto ao uso ou não de controladores integrados ainda não foram objetivamente explorados na literatura. Alguns trabalhos sugerem a substituição quase total dos equipamentos externos de teste por um controlador interno. Outros trabalhos sugerem uma interação entre processadores internos e equipamentos externos de menor capacidade.

1.2 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é realizar um estudo sobre controladores de teste, mostrando sua complexidade e diferentes possibilidades de uso no contexto de sistemas construídos a partir de núcleos de hardware baseados em diferentes estratégias de teste (BIST, cadeias de varredura, etc). O enfoque principal é o uso de um controlador embutido e dedicado para o teste visando o padrão IEEE P1500.

A avaliação das diferentes estratégias de testes é realizada através da implementação na linguagem de descrição de hardware VHDL do controlador de teste descrito em [COT 01]. Mudanças são propostas e implementadas, sempre que possível, no intuito de melhorar o desempenho do controlador. Os resultados de síntese, simulação e exemplos de teste são apresentados e avaliados neste trabalho.

1.3 ORGANIZAÇÃO DA DISSERTAÇÃO

No próximo capítulo, busca-se fazer uma breve discussão sobre técnicas de teste voltadas para sistemas integrados em um único *chip*. Neste capítulo são apresentadas algumas definições sobre SOCs, definição e tipos de núcleos de hardware, sistemática de teste. A proposta de padrão IEEE P1500, para o teste de SOCs, e o norma IEEE std. 1149.1, conhecida como *boundary-scan*, também são enfocados neste capítulo. No capítulo 3 apresenta-se alguns dados sobre custo de equipamentos externos de teste, e busca-se fazer uma revisão dos principais trabalhos que apresentam controladores internos como sistemas gerenciadores de teste. A arquitetura de um controlador específico para o teste é apresentada no capítulo 4. Estudos de alguns tipos de teste e sugestões para a melhoria de desempenho do controlador de teste são apresentados no capítulo 5. Finalmente o capítulo 6 aborda as principais conclusões que puderam ser retiradas do trabalho realizado e apresenta perspectivas futuras de desenvolvimento.

2 TESTES EM CIRCUITOS INTEGRADOS

2.1 SISTEMAS INTEGRADOS EM UM ÚNICO CHIP

Com o avanço nos métodos de projeto e manufatura de circuitos integrados (CIs), sistemas, que anteriormente eram construídos sobre placas de circuito impresso e que continham múltiplos CIs, agora podem ser integrados em um único substrato. Estes sistemas são chamados de SOCs (do inglês, *System-on-Chip*) e são formados por uma grande diversidade de sub-circuitos, chamados núcleos de hardware ou, do inglês, *cores*. Cada um destes sub-circuitos pode ser visto como um componente distinto, com projeto e verificação lógica independentes, mas que são passíveis de serem utilizados em conjunto com outros núcleos, formando assim, um circuito maior e mais complexo chamado de sistema integrado ou, do inglês, *core-based system*. A diferença entre um SOC e um sistema montado sobre uma placa de circuito impresso (PCB, do inglês, *Printed Circuit Board*), é que no primeiro todos os núcleos e a lógica responsável pela união entre os núcleos foram sintetizados em um único substrato, recebendo, ao final, um único encapsulamento. Em uma PCB, cada um dos núcleos é um circuito integrado distinto que foi produzido e encapsulado separadamente, mas montado sobre uma placa discreta que contém a lógica de comunicação entre os circuitos.

Sistemas integrados são tipicamente heterogêneos, uma vez que eles contêm tecnologias mistas, como circuitos digitais, memórias e circuitos analógicos [RAJ 97]. Tipicamente, núcleos de hardware apresentam funções que incluem CPUs e DSPs, interfaces seriais, módulos para interconexão padrão como PCI, USB e IEEE 1394, funções de computação gráfica como MPEG e JPEG, e memórias [MAR 99]. Os sistemas embarcados, que são caracterizados pela diversidade de funções, e os sistemas de maior complexidade e/ou que possuem um tempo de projeto restrito são as aplicações típicas dos SOCs.

Estes sistemas, por estarem integrados em um único substrato e por serem fabricados utilizando uma mesma tecnologia oferecem vantagens como: alto desempenho, baixo consumo de potência, menor tempo de projeto e grande volume de produção, quando comparados com sistemas tradicionais formados por múltiplos *chips* [MAR 99].

2.2 NÚCLEOS DE HARDWARE

Um núcleo de hardware é um bloco funcional pré-verificado e pré-projetado, contendo pelo menos 5000 portas lógicas [ZOR 97], que pode ser utilizado na construção de sistemas maiores e mais complexos. A denominação original é, do inglês, *core*. Os núcleos de hardware também são chamados de componentes virtuais ou bibliotecas de hardware e são classificados como: *soft*, *firm* ou *hard* [ZOR 97].

1. Um *soft core* é um código RTL sintetizável ou uma *netlist* genérica de parte de uma lógica.
2. Um *firm core* é também um código RTL sintetizável ou uma *netlist* genérica, porém, possui a sua estrutura e/ou topologia otimizadas para desempenho ou área. Um *firm core* não vem roteado e pode ser otimizado durante o posicionamento com o resto do sistema. Algumas vezes, esta otimização ocorre durante o mapeamento para algumas tecnologias.
3. Um *hard core* existe em forma de leiaute implementado para uma determinada tecnologia. Ele é otimizado para um ou mais parâmetros chave, tais como velocidade, área ou potência. Um *hard core* está disponível como uma *netlist* com posicionamento e roteamento definidos ou como um arquivo de leiaute físico (em geral em formato GDSII) ou ainda como uma combinação dos dois.

Os três tipos de núcleos de hardware oferecem vantagens e desvantagens. *Soft cores* deixam uma boa parte da implementação para o projetista do sistema, mas são flexíveis e independentes do processo de fabricação. *Hard cores* são otimizados para um desempenho desejado, porém não apresentam maiores flexibilidades de projeto. *Firm cores* oferecem um compromisso entre os dois, sendo mais flexíveis que *hard cores* e menos genéricos que os *soft cores*.

Estes componentes são frequentemente produtos de tecnologias, software e *know-how* sujeitos a patentes e direitos autorais. Em outras palavras, um núcleo de hardware representa uma propriedade intelectual que o projetista do núcleo concede ao usuário deste bloco,

que é o projetista do sistema final. Por isso, questões como proteção da propriedade intelectual devem ser consideradas nas ações que utilizam esta nova tecnologia.

O número de blocos funcionais disponíveis como bibliotecas de hardware tem crescido continuamente. A Tabela 2.1 mostra alguns exemplos destes blocos existentes no mercado.

Tabela 2.1 - Exemplos de núcleos de hardware classificados por funcionalidade [GUP97].

Categoria	Exemplos de produtos
Processadores:	
RISC	LSI Logic CW4K, ARM 7TDMI, PPC
CISC	680x0, x86
DSP	TI TMS320C54X, DSP Group Pine
BIST	MenBIST-IC, PLL BIST, ADC BIST
Criptografia	PkuP, DES
Controladores	USB, PCI, UART
Multimídia	Compressão JPEG; Decodificadores MPEG, DAC
Redes	ATM SAR, Ethernet
Memórias	Memórias especializadas; alta velocidade, de baixa potência

A diversidade de funcionalidade das bibliotecas virtuais tem aumentado com o surgimento de empresas de projeto especializadas e, a medida em que aumenta a disponibilidade destes componentes, as vantagens de sua aplicação se tornam mais evidentes. Novas tecnologias são incorporadas nos mais diversos produtos e o tempo entre projeto e produto final fica cada vez menor devido à possibilidade de reuso. Entretanto, existem ainda diversas questões não resolvidas neste meio [GUP 97]: métodos de projeto para construção de sistemas em um único circuito integrado, problemas na verificação e teste dos novos sistemas complexos, além das questões de licenciamento e proteção da propriedade intelectual dos blocos funcionais.

A Figura 2.1 apresenta um exemplo genérico de um sistema integrado que utiliza diversos núcleos de hardware, alguns deles hierárquicos. O sistema contém, ainda, uma Lógica Definida pelo Usuário (LDU) que acrescenta determinada funcionalidade ao conjunto. Esta lógica é definida pelo projetista responsável pela integração dos diversos componentes em um sistema maior.

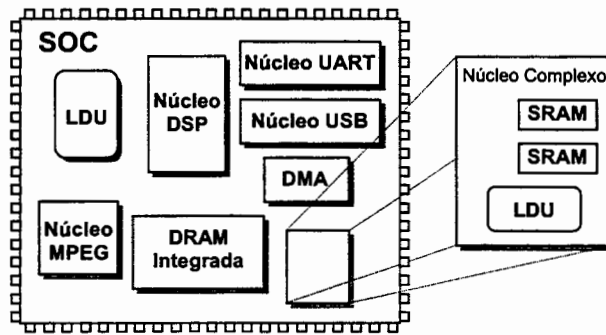


Figura 2.1 - Sistema integrado que utiliza hierarquia de blocos funcionais e Lógica Definida pelo Usuário (LDU) [ZOR 97].

2.3 EXIGÊNCIAS DO TESTE EM SISTEMAS INTEGRADOS

Sistemas integrados em um único chip e sistemas montados sobre placas de circuito impresso são conceitualmente análogos. Porém, as dificuldades de teste relacionadas com sistemas baseados em núcleos de hardware são bastante diferentes daquela relacionada ao teste de ASICs ou dos tradicionais circuitos integrados montados sobre placas. No caso convencional, a fabricação e o teste do circuito integrado ocorre antes da montagem da placa. Um projeto baseado em núcleos de hardware, por outro lado, tem apenas uma etapa de fabricação e teste de todo o sistema [GUP 97]. Neste caso, blocos individuais e sua lógica periférica definida pelo usuário são, primeiro, totalmente projetados e após integrados ao sistema. Somente após a etapa de integração, os passos de produção e teste do sistema final são executados.

Outra diferença importante entre as abordagens de teste para sistemas tradicionais e para sistemas baseados em bibliotecas de hardware é a questão do acesso à periferia do componente [ZOR 98], isto é, o acesso às entradas e saídas primárias dos blocos pré-projetados. Para sistemas montados em uma placa, o acesso físico direto à periferia (pinos) do circuito integrado está tipicamente disponível para ser utilizado durante o teste de produção. Já para os núcleos de hardware, que estão, em geral, profundamente embutidos em um sistema integrado (uma vez que todo o sistema está contido em um único substrato), o acesso direto às suas periferias não está disponível originalmente. Assim, um mecanismo de acesso eletrônico é necessário. Este mecanismo necessita de uma lógica adicional em torno do bloco que pode conter, por exemplo, uma parte que permita o chaveamento entre modos de operação (normal e teste), outra que faz a ligação do bloco com o gerador e o receptor de dados de teste e uma terceira que permite o isolamento deste bloco em relação ao resto do sistema [ZOR 97].

Um dos maiores desafios relacionados à produção dos sistemas integrados é a união e coordenação das capacidades de teste e diagnóstico de todas as partes em um circuito integrado [ZOR 97]. Comparado a um sistema montado em uma placa, os requisitos de teste do sistema integrado são muitos complexos. O teste da placa consiste, simplesmente, no teste das conexões entre circuitos integrados e dos pinos de acesso, pois supõe-se que cada CI tenha sido previamente testado e esteja livre de falhas. O teste de um sistema integrado, por outro lado, é um único teste composto, que envolve a verificação dos núcleos de hardware e do sistema como um todo de forma integrada. A figura 2.2 mostra as diferenças entre estes dois tipos de teste [ZOR 97]. O teste de um sistema integrado é, na verdade, o teste individual de cada núcleo de hardware, o teste da lógica definida pelo usuário (LDU) e o teste de suas conexões. O teste de cada núcleo e LDU, por sua vez, pode envolver componentes periféricos. Certas restrições periféricas (por exemplo, modo de segurança, modo de baixa potência, modo de passagem) são requisitos comuns. Estas restrições necessitam de modos de acesso e isolamento específicos.

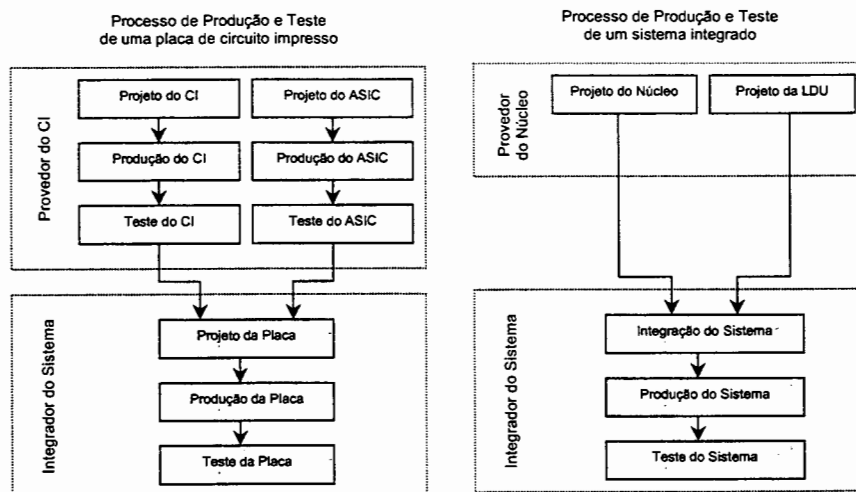


Figura 2.2 - Comparação entre os testes de uma placa e de um sistema integrado [ZOR 97].

Um projetista de núcleos de hardware deve preparar um esquema de teste interno que seja adequadamente descrito, portátil e passível de ser facilmente integrado e utilizável, de forma automática se possível, para interagir com o teste do restante do sistema.

Além de problemas de integração e interdependência, o teste composto de sistemas integrados requer um seqüenciamento adequado das tarefas de teste. Este escalonamento é necessário para atender determinados requisitos do sistema, como tempo de teste, potência dissipada, acréscimo de área, entre outros. O escalonamento do teste é necessário, ainda, para

executar o teste interno dos núcleos e a comunicação entre eles, em uma ordem tal que não influencie a inicialização e o conteúdo final dos blocos funcionais individualmente.

Sabendo que o teste do sistema integrado é composto pelo teste dos núcleos e do sistema, pode-se classificar os requisitos de teste do sistema integrado em três níveis distintos:

1. No primeiro nível estão os requisitos para a verificação dos núcleos que compõem o sistema. As soluções devem ser geradas pelos provedores dos componentes virtuais. O teste interno deve ser pensado pelo projetista do núcleo e transmitido para o integrador de alguma forma (vetores, sinais de controle necessários, etc.). Como o nível de preparação para o teste interno de um núcleo varia de um caso para outro, o teste individual dos núcleos deve ser adequadamente descrito e transportado para o processo de integração do sistema. A alternativa para estas descrições é o uso de uma linguagem padrão de teste, como é o caso da linguagem STIL [SOC 99].
2. No segundo nível, composto basicamente pelas interconexões entre os diversos núcleos, há um único requisito a ser atendido, porém de extrema importância para o desempenho e qualidade do teste final do sistema. Este requisito é o acesso a cada núcleo no momento de teste. Um núcleo, por exemplo, pode ser acessível apenas através das entradas e saídas de outro(s) componente(s). Neste caso, a inserção de vetores de teste externos e a verificação das respostas de teste deste núcleo podem não ser possíveis. No caso do acesso disponível pela funcionalidade normal do sistema não ser suficiente, é necessária a implementação de novos meios de acesso para o teste. Uma vez que, em geral, é o integrador quem define esta camada, ele deve também decidir sobre a melhor implementação dos caminhos de teste.
3. No terceiro nível estão os requisitos para o sistema final, que também devem ser resolvidos pelo integrador do sistema. Estes requisitos são: teste da lógica definida pelo usuário, teste do sistema como um todo, seqüenciamento do teste, e o controlador de teste.

Dentre as técnicas existentes para teste dos componentes virtuais tem-se observado uma forte tendência para o uso de estruturas BIST como parte importante do teste dos núcleos de um sistema integrado. Estruturas BIST são eficientes, pois podem ser embutidas nos núcleos permitindo a manutenção da propriedade intelectual associada ao bloco funcional;

possuem uma arquitetura que permite a automatização de partes do projeto; apresentam alta cobertura de falhas; são técnicas bastante sedimentadas, tendo sido aplicadas a uma grande variedade de circuitos; e por fim, facilitam a definição do teste do sistema como um todo [WUN 98]. Espera-se que o esquema BIST, voltado para núcleos de hardware, garanta a proteção da propriedade intelectual, o baixo consumo no dispositivo, a integração simplificada em um sistema com BIST, a compatibilidade com o reuso de projeto e a capacidade de diagnóstico.

Embora um esquema BIST satisfaça a maioria dos requisitos de teste e torne possível a implementação de uma variedade de metodologias de projeto visando o teste, ele pode resultar em um acréscimo de área considerável e, muitas vezes, na repetição inútil de estruturas de teste como os geradores de vetores, ou analisadores de respostas, por exemplo. Técnicas baseadas em cadeias de varredura também podem atender os requisitos de teste de um núcleo de hardware, porém estas podem apresentar problemas devido ao tempo de teste necessário e à necessidade de geração externa de padrões de teste [WUN 98]. Por este motivo, técnicas que reduzam o acréscimo de área e o tempo de teste são de grande interesse. Estes objetivos podem ser alcançados com o uso de controladores de teste mais elaborados e projetados especialmente para serem utilizados em sistemas integrados constituídos por núcleos de hardware.

O conceito de projeto visando o teste (do inglês, *DFT – Design For Test*) visa uma metodologia onde o teste e a verificação do sistema são considerados desde as primeiras especificações do projeto. Nesta metodologia, junto com a lógica do sistema são inseridas estruturas que facilitam a verificação do circuito em diversos níveis, desde a verificação funcional até o teste de manutenção. Estas estruturas podem ser tanto pontos de acesso a modos internos a partir de suas entradas e saídas primárias, como subcircuitos capazes de realizar, independente do testador, o teste do sistema como um todo.

2.4 ARQUITETURA DE TESTE CONCEITUAL

Baseada em alguns requisitos de teste, uma arquitetura conceitual e genérica para o teste foi introduzida em [ZOR 98]. Esta arquitetura, mostrada na figura 2.3, consiste em quatro elementos estruturais básicos:

- Uma fonte (dentro ou fora do *chip*) que gera estímulos de teste;
- Um receptor (dentro ou fora do *chip*) que avalia as respostas de teste;

- Mecanismos de acesso de teste que transportam os estímulos da fonte ao núcleo e deste ao receptor;
- Uma lógica de acesso periférica acoplada à periferia de um bloco interno ao sistema. Sua função é conectar os terminais deste bloco ao restante do circuito e aos mecanismos de acesso de teste.

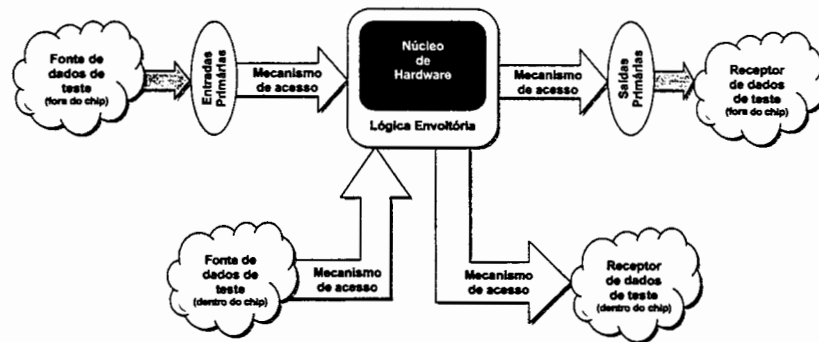


Figura 2.3 - Arquitetura conceitual para o teste de sistemas integrados [ZOR 98].

Os elementos que compõem a arquitetura conceitual proposta por [ZOR 98] podem ser implementados de várias formas. As estruturas fonte e receptora de dados de teste podem ser implementadas externamente ao sistema, através de um equipamento de teste externo; dentro do sistema, através de estruturas BIST ou um controlador de teste embutido; ou como uma combinação entre o meio externo e os componentes internos.

2.5 MECANISMO DE ACESSO AO TESTE

O mecanismo de acesso de teste cuida do transporte das informações de teste dentro do sistema, tanto da fonte de estímulos até o núcleo quanto deste até o receptor. Embora, para um núcleo, o mesmo tipo de mecanismo de acesso seja usado para transporte tanto de estímulos quanto de respostas de teste, isto não é estritamente necessário e várias combinações podem coexistir [ZOR 98]. O projeto de um mecanismo de acesso de teste procura o equilíbrio entre a capacidade de transporte do mecanismo e o custo de aplicação de teste que ele gera. A capacidade de transporte, por sua vez, é limitada pela capacidade da fonte e do receptor e pela área que pode ser utilizada para este fim [ZOR 98].

Segundo [ZOR 98], a implementação de um mecanismo de acesso tem as seguintes opções:

- Um mecanismo pode reutilizar barramentos e caminhos já existentes para o transporte das informações de teste ou ser formado por um caminho exclusivo;
- O mecanismo pode passar através de outros módulos dentro do sistema (utilizando ou não suas funções) ou passar em torno de outros núcleos, através de alguma lógica envoltória;
- Pode existir um mecanismo de acesso independente para cada núcleo ou pode-se ter o compartilhamento de caminhos entre diversos núcleos;
- O mecanismo de acesso pode ser apenas um transportador de sinais ou pode conter algumas funções para o controle do acesso.

[ZOR 98] apresenta, ainda, algumas abordagens já adotadas para implementação de mecanismos de acesso.

2.6 PADRÃO IEEE P1500

Em setembro de 1995, o *IEEE Test Technology Technical Committee (TTTC)* iniciou um Comitê de Atividade Técnica (do inglês, TAC, *Technical Activity Committee*) no campo de teste e DFT para sistemas integrados baseados em bibliotecas de hardware. Este Comitê evoluiu, mais tarde, para um Grupo de Trabalho de Padronização, aprovado em junho de 1997 como P1500, com o objetivo de desenvolver um padrão para o teste de blocos funcionais embutidos (núcleos de hardware).

A necessidade de padronização surgiu devido à existência de duas partes no processo de desenvolvimento e teste destas estruturas: de um lado, os provedores de núcleos de hardware e, de outro lado, o grupo usuário destes núcleos, projetistas ou integradores de sistemas. O grupo IEEE P1500 procura definir um padrão que facilite a interoperabilidade de blocos funcionais de diferentes origens a fim de melhorar a eficiência de ambos, provedores e usuários.

O grupo P1500 conta com a participação de diversas companhias de vários segmentos da indústria de semicondutores, incluindo-se ainda, o grupo de Trabalho em Teste relacionado à Manufatura da VSI Alliance [ALL 00].

Atualmente, o padrão ainda está em fase de elaboração. A versão atual da proposta de padronização considera núcleos como memórias e circuitos com lógica digital integrados ao sistema, mas vistos como blocos distintos dentro do mesmo, ou seja, não se pode mo-

dificar a lógica interna do componente durante a síntese. Uma versão preliminar do padrão foi finalizada em maio de 2000 (P1500/D0.2), alguns *drafts* foram apresentados em 2001 e uma versão final do padrão está prevista para 2002. Algumas definições do padrão têm sido divulgadas por meio de artigos em conferências recentes. Uma compilação destas informações é apresentada na seção seguinte.

O IEEE focaliza a padronização de partes do sistema relativas à interação entre o provedor e o usuário dos núcleos de hardware. Esta padronização envolve a transferência do conhecimento sobre o teste interno do núcleo, e o acesso de teste aos núcleos embutidos no sistema (bloco de acesso junto ao núcleo) [MAR 00a]. Tarefas como: métodos de teste ou DFT internos aos núcleos, integração do teste do sistema como um todo, otimização do teste do sistema, estímulos de teste e TAMs, não são cobertas pelo padrão P1500 [ZOR 2000]. Segundo [MAR 99], estas tarefas estão nas mãos dos provedores e dos usuários dos núcleos, e não estão sujeitas a padronização devido à grande diversidade de requisitos.

Os dois principais elementos do padrão P1500 são:

1. Uma linguagem, chamada *Core Test Language* (CTL), que servirá para transferência do conhecimento relativo ao teste interno do núcleo. CTL baseia-se na linguagem STIL (Padrão IEEE 1450.0) [ALL 2000] ou *Standard Test Interface Language*, que está sendo estendida para acomodar construções específicas do teste de núcleos de hardware.
2. Uma arquitetura escalonável (parametrizável) para teste de núcleos. Em relação ao acesso de teste aos núcleos embutidos no sistema, foi decidido padronizar-se apenas o bloco de acesso ao núcleo ou sua lógica envoltória (denominada *Wrapper*) e sua interface para um ou mais mecanismos de acesso de teste (ou, do inglês, TAM, *Test Access Mechanisms*) do sistema.

2.6.1 Linguagem de Descrição do Teste de Núcleos de Hardware

A linguagem de Teste de Núcleos de hardware (do inglês, CTL, *Core Test Language*) definida no P1500 é uma linguagem capaz de capturar e expressar informações relativas ao teste de núcleos de hardware reutilizáveis [KAP 99]. Seu objetivo é complementar e co-existir com modos (orientados a estrutura) de representação de teste e atributos de testabilidade de uma entidade existente. Em particular, CTL procura prover um modelo de informação universal, explícito e conciso para controles específicos de núcleos de hardware, de forma a configurar um núcleo para seu próprio teste ou de suas funções anexas em um sistema. A-

lém disso, ela procura, ainda, prover as informações sobre requisitos e restrições para a implementação das interfaces entre o núcleo e o sistema no qual ele está inserido [ZOR 00].

CTL está sendo definida utilizando-se a sintaxe da Linguagem Padrão para Interface de Teste (do inglês, STIL, *Standard Test Language Interface*), definida pelo padrão IEEE 1450.0 [SOC 99]. STIL foi desenvolvida para ser uma linguagem comum, padronizada, para transferência de informações de testes digitais entre o ambiente de geração (domínio de ferramentas EDA) e o ambiente de aplicação (domínio dos equipamentos de teste). Então, STIL é capaz de descrever os dados relativos ao teste, tais como, estímulos de teste e formas de onda, para circuitos integrados. O P1500 utiliza, tanto quanto possível, a linguagem STIL original, mas dois elementos foram acrescentados a ela. Foram necessárias novas **palavras-chave** para descrever aspectos específicos de núcleos de hardware e suas lógicas envoltórias. Além disso, um **modelo de informação de teste** está sendo criado, para permitir a definição das diversas configurações do núcleo, necessárias para torná-la parcial ou completamente compatível com o P1500. CTL descreve três tipos de informações:

1. **Aspectos dos dados de teste:** trata das características básicas, tais como, tipo de dado, razão, estabilidade dos valores, e informações relativas à metodologia de teste.
2. **Diferentes configurações do núcleo:** informações sobre o modo de teste e sobre conectividade como atributos e protocolos.
3. **Instruções para integração do sistema:** informações sobre conectividade.

A partir de um código CTL o projetista tem acesso a informações suficientes da periferia do núcleo que permitam a instalação correta da lógica envoltória, o mapeamento dos terminais do núcleo para os terminais do *wrapper*, o reuso dos dados de teste do núcleo, o teste da lógica definida pelo usuário (LDU) e das conexões externas ao núcleo. Em [MAR 99] pode-se encontrar, com mais detalhes, as funções e os conteúdos da linguagem CTL definidas até este momento.

2.6.2 Arquitetura Escalonável

O P1500 tenta definir uma interface uniforme e flexível entre o núcleo de hardware e seu ambiente de forma a facilitar sua integração. Um conjunto básico de requisitos de teste de núcleos de hardware foi estabelecido. Eles podem ser vistos como modos de operação que a arquitetura padrão deve suportar. São eles [ADH 99]:

- **Modo Normal** → Este modo permite que o sistema integrado e seus blocos embutidos funcionem em seu modo normal de operação.
- **Modo de Teste Interno** → Este modo habilita a aplicação de vetores pré-definidos em um núcleo do sistema.
- **Modo de Teste de Interconexão** → Este modo de teste habilita a lógica envoltória do núcleo de hardware a ser utilizada para o teste das conexões do sistema e da lógica entre blocos funcionais.
- **Modo de Isolamento** → Este modo permite que o núcleo de hardware seja isolado a fim de facilitar o teste de outros blocos ou da LDU no sistema integrado.

A figura 2.4 mostra um diagrama da arquitetura de teste proposta. Existem três componentes chave nesta arquitetura: a lógica envoltória do núcleo de hardware, o mecanismo de controle do teste e o mecanismo de acesso ao núcleo. Estes componentes são descritos detalhadamente em [AHD 99] e [IEE 15].

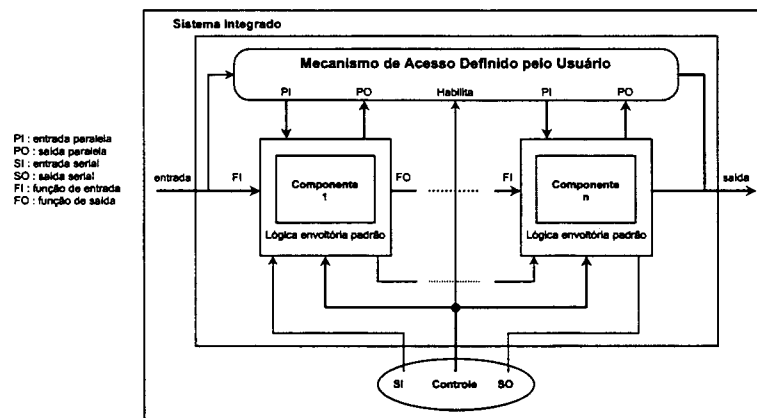


Figura 2.4 - Sistema integrado com núcleos utilizando o padrão P1500 [ADH 99].

A lógica envoltória do núcleo de hardware, também chamada de *Wrapper*, provê mecanismos para o acesso de dados de teste e o isolamento de um núcleo do resto do sistema. Esta lógica permite o controle das entradas e saídas através do mecanismo de acesso, de forma que os testes internos do núcleo possam ser reaplicados dentro do sistema. Esta lógica deve ser também capaz de prover o isolamento do bloco durante seu teste ou durante o teste de estruturas próximas, evitando dano ao sistema. O circuito envoltório pode ser visto como separado logicamente do bloco funcional e pode vir com o componente ou ser acrescentado du-

rante a síntese do sistema. O padrão P1500 define apenas o comportamento funcional da lógica envoltória e não sua forma de implementação.

O modo de operação da lógica envoltória é determinado pela carga serial das instruções no Registrador de Instruções do *Wrapper*. O Grupo de Trabalho do P1500 pretende definir três categorias de instruções:

1. **Instruções obrigatórias:** para que um *wrapper* seja compatível com o P1500, ele deve implementar este conjunto de instruções que resultam em um comportamento específico e pré-definido.
2. **Instruções Opcionais:** se implementadas, deve resultar em um comportamento pré-definido para que o *wrapper* seja compatível com o padrão P1500.
3. **Instruções definidas pelo usuário:** sem restrições ou regras.

O conteúdo de um *wrapper* é: um Registrador de Instruções do *Wrapper* (WRI) que serve para controlar a operação da lógica envoltória; múltiplas células capazes de prover controlabilidade e observabilidade nos terminais dos núcleos; um registrador de *bypass* de um bit que serve como passagem para o mecanismo de acesso de teste serial; e sinais de conexão para selecionar os vários modos de operação do *wrapper*. Observe que a lógica envoltória é muito semelhante a uma máquina da TAP definida pelo padrão IEEE std. 1149.1 [IEE 90] e resumida na seção 2.7. A principal diferença é que o *wrapper* permite um maior paralelismo, a fim de aumentar a dinâmica do teste provendo um maior controle e observabilidade durante o processo de teste. A figura 2.5 mostra a arquitetura conceitual do *wrapper* proposta pelo padrão IEEE P1500 [MAR 00b].

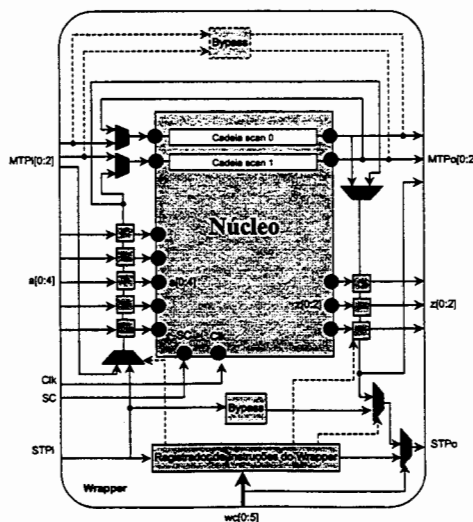


Figura 2.5 - Arquitetura conceitual do *wrapper* segundo IEEE P1500 [MAR 00b].

2.7 PADRÃO IEEE STD. 1149.1 (*BOUNDARY-SCAN*)

Nos meados dos anos 80, um grupo de engenheiros de teste de várias companhias européias de sistemas eletrônicos reuniu-se para examinar o problema crescente na dificuldade do teste de pontos de solda e das conexões entre CIs que eram montado sobre de placas de circuitos impressos. Conforme a tecnologia evoluía as placas de circuitos impressos eram construídas com trilhas mais finas e com quantidades maiores de camadas tornando o teste das interconexões praticamente impossível. Este grupo foi chamado de Joint European Test Action Group (JETAG).

A solução proposta por este grupo foi baseada no conceito de um caminho serial de registradores de deslocamentos colocado na periferia da lógica central que forma o CI, daí o conseqüente nome de *Boundary-scan*. Mais tarde este grupo uniu-se a engenheiros de testes de companhias da América do Norte e o grupo passou a ser chamado de Joint Test Action Group (JTAG). Esta organização converteu a idéia do *boundary-scan* em um padrão internacional chamado de IEEE std. 1149.1 [IEE 90].

Apesar do enfoque principal do padrão IEEE 1149.1 ser o teste de interconexões, o padrão também permite o uso das suas estruturas para testar a lógica central dos componentes. Na era SOC, o padrão *boundary-scan* passou a ser menos utilizado para o teste de interconexões e mais explorado para o teste da lógica central dos núcleos, provendo um importante mecanismo de acesso para a inserção de dados de teste do testador nos núcleos de hardware e destes de volta ao testador.

O padrão IEEE std. 1149.1 consiste basicamente na arquitetura da figura 2.6 e está dividido nos seguintes elementos:

- Quatro pinos de teste dedicados – *Test Data Input (TDI)*, *Test Mode Select (TMS)*, *Test Clock (TCK)*, *Test Data Output (TDO)*. Estes pinos são coletivamente chamados de TAP (do inglês, *Test Port Access*).
- Células *boundary-scan* colocadas em todos os pinos de entradas e saídas, e unidas entre si para juntas formarem um registrador serial *boundary-scan*. O pino de TDI está na entrada desse registrador e o pino de TDO na saída.
- Uma máquina de estados com 16 estados (figura 2.7), para controlar a TAP através dos sinais de TCK e TMS.
- Um registrador de instruções.

- Um registrador de *Bypass* de 1-bit.

As células *boundary-scan* (CB) podem ser configuradas em modo de entrada paralela, saída paralela e modo deslocamento. Uma operação de carga paralela é chamada de operação de “CAPTURA”. Esta operação faz com que os valores que foram introduzidos sejam carregados pelas células de entrada e os valores que passaram pela lógica central do dispositivo sejam colocados nas células de saída. Uma operação de descarga em paralelo é chamada de operação de “ATUALIZAÇÃO”. Esta operação faz com que os valores que estão nas células de entrada serem enviados para a lógica central. Isto causa a substituição de valores dos pinos de entrada do componente e dos valores presentes nas células *boundary-scan* de saída que são passados através dos pinos de saídas do componente, substituindo os valores de saídas gerados pela lógica central.

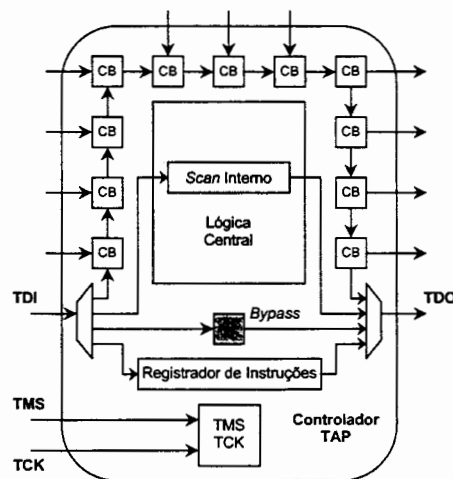


Figura 2.6 - Arquitetura IEEE std. 1149.1.

Para cada célula *boundary-scan* também é proporcionado um estado de registrador de deslocamento, onde todas as células são conectadas em série formando um caminho serial na periferia do CI. Os sinais são introduzidos a partir do pino de entrada TDI e são coletados a partir do pino de saída TDO.

No padrão IEEE std. 1149.1, todos os sinais são inseridos e retirados serialmente através dos pinos de TDI e TDO respectivamente. O controle de estados é feito através dos sinais de TMS e TCK. TCK é um sinal de relógio dedicado que serve para o deslocamento serial dos sinais nas células *boundary-scan*.

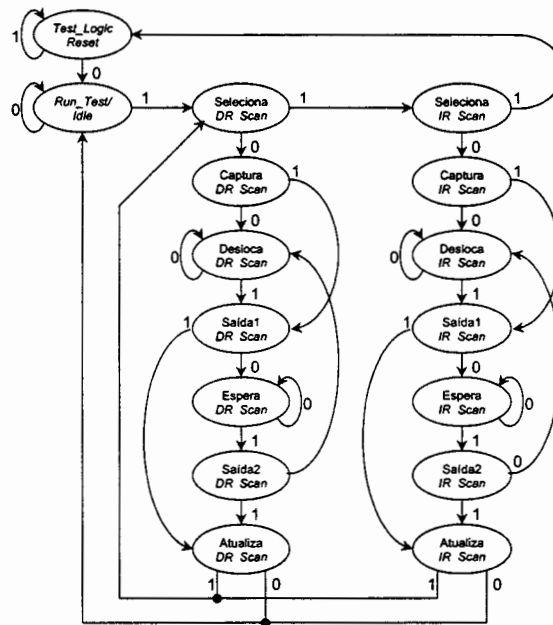


Figura 2.7 - Diagrama de estados do controlador de TAP.

O controlador de TAP (figura 2.7) inicializa no estado de *Test_Logic_Reset*. Enquanto TMS permanece em '1', o estado permanece inalterado. Quando TMS baixa, causa a transição para o estado inativo *Run_Test*. Normalmente deseja-se chegar ao estado *Seleciona IR_Scan* e assim ficar prontos para carregar e executar uma nova instrução. Uma seqüência adicional em TMS alcançará este estado. A partir deste estado, pode-se mover pelos estados de *Captura IR_Scan*, *Desloca IR_Scan* e *Atualiza IR_Scan* conforme desejado. Uma instrução que esteja no registrador de instruções pode ser carregada diretamente a partir do estado de *Captura IR_Scan*. Também pode-se carregar a instrução no registrador de instruções pelo deslocada serialmente através TDI no estado de *Desloca IR_Scan*. No estado *Atualiza IR_Scan*, a instrução é habilitada para tornar-se a instrução atual preparando o sistema para o teste, como por exemplo, selecionar uma cadeia *scan* interna.

A partir do estado *Seleciona DR_Scan* os dados de teste são manipulados. No estado *Desloca DR_Scan* os dados são deslocados serialmente através do pino de TDI, e no estado de *Atualiza DR_Scan* o tipo de teste que foi selecionado pela instrução é executado, como por exemplo, a transferência dos dados das cadeias para o lógica central do núcleo.

Note que não existe nenhum sinal de *Reset* no controlador de TAP. Se houver uma necessidade para reinicializar o controle, isto pode ser feito mantendo TMS alto e ativando o sinal de TCK por no máximo cinco pulsos de relógio, ou seja, de qualquer estado, com no máximo cinco pulsos de relógio e TMS = '1' pode-se chegar no estado *Test_Logic_Reset*.

3 CONTROLADORES DE TESTE

À medida que aumenta a complexidade dos sistemas integrados e das estruturas de testes, surge a necessidade do uso de controladores de teste mais sofisticados e que atendam rapidamente a contínua evolução tecnológica.

Um controlador de teste é o módulo responsável pelo gerenciamento do teste como um todo, enviando sinais de controle, estímulos de teste e analisando respostas para cada componente do sistema (núcleos de hardware, LDU e interconexões), de acordo com a sequência definida e através dos caminhos de acesso e mecanismos de teste disponíveis.

O gerenciamento e envio dos sinais de teste pode ser realizado por equipamentos tradicionais chamados de ATEs (do inglês, *Automatic Test Equipments*), os quais realizam o controle do teste externamente, através das entradas e saídas primárias do circuito integrado, e de pinos extra, que são utilizados somente para o teste. Outra forma de fazer o controle é através de um módulo interno ao SOC. Fazendo uso da dinâmica re-configurável e escalonável da lógica envoltória e da flexibilidade das estruturas propostas pelo padrão IEEE P1500, dispositivos do tipo microprocessadores/microcontroladores podem ser transformadas em controladores de teste, transferindo, assim, a funcionalidade do controle para dentro do SOC [PAP99], [ABH99], [RAJ99], [COT 01].

O uso de um controlador interno tem, entre outras, a vantagem de ser dedicado, além de poder reutilizar módulos como, por exemplo, memórias existentes no sistema. Além disso, o controlador interno possui a vantagem de poder trabalhar na mesma frequência do SOC, facilitando a execução de testes *at-speed* e contribuindo para a diminuir o tempo global de teste do sistema integrado. Outra característica importante no uso de um controlador interno, em relação aos ATEs, é a redução do número de pinos de entrada e saída do CI.

A transferência da funcionalidade do controle do teste para dentro do sistema é economicamente uma solução muito atrativa porque equipamentos de teste externos capazes de manipular uma grande quantidade de dados de teste e operar com altas velocidades possuem custos elevados. O acréscimo de pinos no encapsulamento para realizar a comunicação

entre o controlador de teste externo e os núcleos que constituem o sistema é outro problema a se considerar.

Além do custo elevado para a inserção de pinos e caminhos extra no sistema integrado, existe o fato de que, em alguns sistemas, pode ser inviável a inserção de todos os pinos necessários para o teste, devido à limitação física do encapsulamento. O uso dos mecanismos de acesso ao teste pode reduzir o número de pinos extra para o teste, uma vez que os sinais podem ser roteados através de outros módulos até o núcleo que se deseja testar. Porém, o tempo total de teste do sistema pode aumentar significativamente e até mesmo tornar-se impraticável. Isto ocorre porque é necessário obedecer a uma seqüência de teste, já que o teste de um ou mais núcleos que utilizam um caminho específico deve ser concluído para que os caminhos de acesso possam ser reutilizados para testar outros elementos do sistema.

3.1 CUSTOS E PRECISÃO DE EQUIPAMENTOS AUTOMÁTICOS EXTERNOS DE TESTE (ATES).

Os custos dos ATEs são tradicionalmente medidos utilizando-se uma simples aproximação de “custo por pino digital”. Embora este método seja conveniente, ele é incompleto, porque ignora os custos básicos do sistema associados com a infra-estrutura de equipamentos e instrumentos centrais como também ignora o escalonamento benéfico que ocorre com o incremento da quantidade de pinos. Por essa razão, foi sugerido o uso da equação 1 para o cálculo dos custos de ATEs de modo que os valores estejam mais próximo do real [ROA 01].

$$\text{Custo do Testador} = b + \sum (m * x)_n \quad (1)$$

Na equação um (1), b representa o custo base de um sistema de teste com zero pinos, m representa o incremento do custo por pino, e x é o número de pinos. Note que b considera a capacidade, desempenho, e características do sistema básico, enquanto m depende da característica e capacidade analógica dos pinos. A tabela 3.1 mostra, como exemplo, alguns dados utilizados pelos fabricantes de semicondutores para o cálculo dos custos de equipamentos automáticos de testes [ROA 01].

Tabela 3.1 - Parâmetros para o cálculo de custos de ATEs.

Segmentos de Testadores	<i>b</i>	<i>m</i>	<i>x</i>	Custo Total
	Custo Base	Incremento do Custo por Pino	Quantidade de Pinos	
	KUS\$	US\$		US\$
ASIC/MPU de alto desempenho	250 - 400	2.700 - 6.000	512	1.632.400 - 3.472.000
Sistemas Mistos	250 - 350	3.000 - 18.000	128 - 192	634.000 - 3.806.000
Testador DFT	100 - 350	150 - 650	512 - 2500	176.800 - 1.975.000
Microcontroladores / ASIC	200 - 350	1.200 - 2.500	256 - 1024	507.200 - 2.910.000
Memória	200+	800 - 1.000	1024	1.019.200 - 1.224.000
RF	200+	≈50.000	32	≈1.800.000

O incremento na velocidade de microprocessadores e pinos de E/S (entrada e saída) exige o aumento da precisão para uma resolução adequada dos tempos dos sinais medidos. Enquanto a velocidade fora do chip tem um incremento de 30% ao ano, a precisão dos testadores tem crescido a uma taxa de 12% ao ano. Se essa tendência atual continuar, erros do testador irão se aproximar do ciclo de tempo de dispositivos rápidos. Como mostra a tabela 3.2 [ROA 01] em 2001 às perdas de rendimento devido à imprecisão dos testadores tornaram-se um problema quando metodologias tradicionais de testes funcionais são utilizadas.

Tabela 3.2 - Rendimento *versus* precisão do testador.

Anos		2001	2002	2003	2004	2005	2006	2007
Frequência do barramento do chip-placa – alto desempenho	MHz	1700	1870	2057	2262	2488	2737	3011
Período do dispositivo	ps	588	535	486	442	402	365	332
Precisão global do ATE (OTA)	ps	200	176	155	136	120	106	93
Exigência de precisão global do dispositivo (objetivo de 5%)	ps	29	27	24	22	20	18	17



- Soluções de fabricação são conhecidas

- Soluções de fabricação não são conhecidas

Este grave potencial de perda de rendimento deve ser reduzido pela introdução de métodos alternativos para testes funcionais *at-speed*. Novas metodologias devem ser buscadas

para aliviar o risco da perda de rendimento em projetos, devido a um testador impreciso. Devem também fornecer um benefício adicional na redução da complexidade da interface entre o testador e o dispositivo sob teste, buscar a redução nas exigências das capacidades do equipamento e o aumento na ênfase de projetos baseados em DFT, incrementando o uso do teste paralelo e reduzindo o tempo de teste. Isto deve resultar na redução total do impacto no custo do equipamento de teste e, por consequência, no custo total do teste. O uso de testadores internos se insere perfeitamente neste desafio de fornecer uma alternativa de melhorar a relação da precisão necessária nos testadores externos e diminuir o impacto do custo dos ATEs e no teste como um todo.

3.2 CONTROLADORES INTERNOS

Alguns trabalhos de pesquisa sobre o uso de controladores internos de teste têm sido apresentados nos últimos anos. Nestes trabalhos encontra-se uma grande variedade de implementações (processadores dedicados, reutilizados, etc.) e propostas de teste (BIST, cadeias de varredura *scan*, IDDq, etc.), que transferem parcial ou totalmente a funcionalidade do testador para dentro do *chip*. Nas seções seguintes deste capítulo, os principais trabalhos relacionados com controladores de teste internos serão apresentados.

3.2.1 Microprocessadores Reutilizados para Produzir Controladores de Teste

É fato conhecido que grande parte dos projetos de sistemas integrados possuem algum tipo de microprocessador (ou algum componente semelhante) embutido. Conseqüentemente, uma metodologia baseada no reuso de processadores para teste pode ser utilizada para a maioria dos sistemas baseados em núcleos de hardware. Em [PAP 99] é sugerido um esquema de teste baseado no uso de um microprocessador contido em um SOC para testar os demais núcleos do sistema. Neste caso, o tamanho do controlador externo de teste seria muito pequeno, uma vez que o microprocessador executaria a maioria das funções de controle do teste.

Os principais componentes exigidos pela estratégia sugerida seriam um microprocessador e memórias para serem utilizadas durante o teste. Um caminho deve estar disponível entre os pinos de entrada e saída primários do sistema, e a memória de teste requerida, de forma que os dados de teste possam ser carregados a partir de um equipamento externo. O

teste do microprocessador e da memória não é discutido em [PAP 99], assume-se apenas que estes componentes foram previamente testados.

O conceito proposto em [PAP 99] é ilustrado na figura 3.1. Os dados de teste compactados que estão armazenados em um equipamento externo de teste, são carregados na memória de teste através de uma técnica de DMA (do inglês, *Direct Memory Access*). Após a carga dos dados, o microprocessador testa os núcleos. Os núcleos de hardware podem ser agrupados para o teste, sendo que núcleos pertencentes ao mesmo agrupamento não podem ser testados simultaneamente, enquanto que núcleos que fazem parte de agrupamentos diferentes podem ser testados em paralelo. Núcleos que pertencem ao mesmo agrupamento também podem ser diretamente conectados uns aos outros, enquanto, núcleos de diferentes grupos não podem ser diretamente conectados. A vantagem de fazer agrupamentos está em exercer um maior paralelismo do teste.

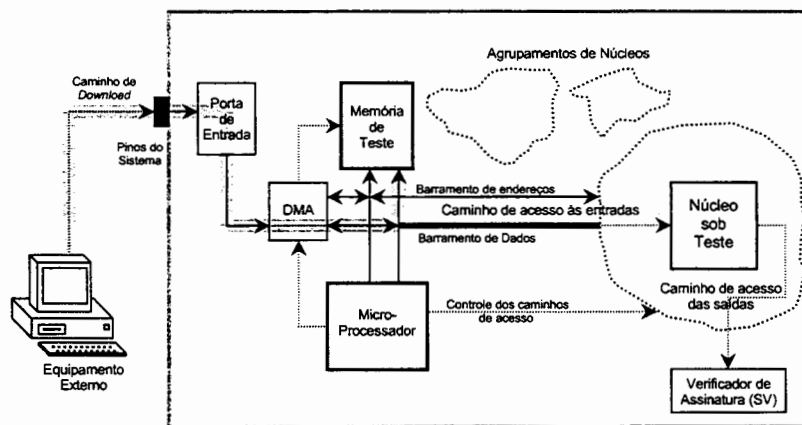


Figura 3.1 - Esquema de teste proposto em [PAP 99].

Para testar um núcleo de um agrupamento, é necessário prover um caminho de acesso do microprocessador até as entradas do núcleo e também, um caminho de acesso das saídas do núcleo até um bloco chamado de Verificador de Assinatura (SV, do inglês, *Signature Verifier*). Este bloco é um registrador de assinatura de múltiplas entradas, chamado de MISR (do inglês, *Multiple Input Signature Register*), capaz de compactar das respostas fornecidas pelo núcleo em uma assinatura que será analisada pelo microprocessador.

O microprocessador habilita os caminhos de acesso com a ajuda de um controlador de teste simplificado e envia os dados para o núcleo. Depois de aplicar os padrões de teste no núcleo, as saídas são levadas até o bloco verificador de assinaturas.

Na figura 3.1 observa-se a existência de três caminhos diferentes: um caminho de carga de dados (*download*), o caminho de acesso às entradas, e o caminho de acesso às saídas. O caminho de acesso às entradas inclui o caminho que faz a conexão da memória de teste com o microprocessador e deste com o núcleo.

Considerando que o tamanho da memória de teste é limitado deve-se assegurar que a carga dos dados de teste não exceda a capacidade da memória de teste. Isto é feito através do “empacotamento” dos vetores de teste, ou seja, uma certa quantidade de vetores é agrupada formando um pacote. A memória é carregada com estes pacotes e o tamanho de um pacote deve ser sempre menor que a capacidade da memória. Novas cargas de pacotes somente poderão ser feitas quando existir espaço na memória. O microprocessador deve esperar a carga total do pacote antes de começar a leitura deste.

O microprocessador deve executar as seguintes funções:

1. Antes de ter acesso a cada pacote, o microprocessador deve conferir se o DMA já carregou todo o pacote exigido no teste;
2. Controlar a carga do DMA, para que este não carregue um novo pacote sobre algum outro pacote que esteja sendo usado em um teste. Isto é feito através de interrupções;
3. Enquanto estiver testando um núcleo, o microprocessador deve configurar os caminhos, a fim de que os corretos acessos de entrada e saída dos núcleos sob teste sejam habilitados. Deve configurar ainda, o bloco verificador de assinaturas para compactar/verificar as saídas como exigido;
4. No caso de testes pseudo-aleatórios o microprocessador pode gerar os padrões de testes por si só. Neste caso, um controlador BIST não é necessário para testar qualquer núcleo com estrutura BIST, como, por exemplo, núcleos com testes pseudo-aleatórios. Esta seria uma forma de um software BIST e em alguns casos pode executar um teste mais rapidamente que um teste pseudo-aleatório externo.

Algumas informações não estão claras neste trabalho: o tempo de teste necessário e a definição da seqüência de teste. Além disto, o exemplo apresentado é bastante simples e o tempo de teste pode ser proibitivo para sistemas mais complexos. Outro problema refere-se a verificação do microprocessador e da memória utilizados no teste. O primeiro é, em geral, uma das partes mais complexas de um sistema (pode ser, inclusive, um núcleo hierárquico, com memória interna, unidade lógica e aritmética, máquina de controle, etc.), cujo teste deve

ser uma das tarefas mais complicadas. Desta forma, o problema principal do teste não é resolvido.

Em [RAJ 99] é proposta uma metodologia de teste para sistemas baseados em núcleos de hardware que contenham um núcleo microprocessador. Neste esquema, primeiro é testado o núcleo microprocessador. Em seguida, este é utilizado para testar memórias e outros núcleos do sistema. A metodologia proposta em [RAJ 99] pode ser dividida em:

1. Teste do microprocessador assegurando o seu correto funcionamento;
2. Uso do microprocessador para testar memórias;
3. Teste de outros núcleos;

Para o teste do microprocessador é utilizado um Registrador de Controle de teste (TCR, do inglês, *Test Control Register*), para prover os códigos operacionais das instruções do microprocessador durante o modo de teste. Também é utilizado um LFSR (do inglês, *Linear Feedback Shift Register*) e um MISR para gerar dados aleatoriamente e compactar as respostas, respectivamente. O controle deste esquema de teste é implementado através de um controlador TAP definido pelo padrão IEEE 1149.1 [IEE 90].

Para o teste das memórias, [RAJ 99] propõe o uso de um programa em uma linguagem de baixo nível (*assembly*) para gerar os padrões de teste de um algoritmo *March* [LAL 97]. O programa de teste deve ser convertido em um código binário que é armazenado em um equipamento externo. Durante o teste, este código passa diretamente do testador para o microprocessador que executa as operações e testa a memória. A saída final do microprocessador para o testador é um sinal indicando se a memória está livre de falhas ou possui algum defeito. Este conceito é utilizado para testar os outros núcleos do sistema. Um programa gerado a partir de uma linguagem de baixo nível é convertido em um código binário e armazenado em um testador, o microprocessador utiliza este código para testar os núcleos.

Este trabalho apresenta uma visão geral do processo de controle interno do teste através de alguns exemplos específicos. O controlador de teste é utilizado para o envio dos dados de teste, porém o autor não faz referência do controle ou do mecanismo de acesso utilizado para o teste de um sistema completo.

Usualmente, o teste funcional de sistemas complexos requer grandes quantidades de padrões de testes. Entretanto, espaço em memória nem sempre está disponível no sistema. Os vetores de teste podem ser de dois tipos: determinísticos ou pseudo-aleatórios. Padrões determinísticos são seqüências inteiras e específicas de estímulos gerados, normalmente, por simulação e que visam maior cobertura de falhas com menores quantidades de valores. Os padrões pseudo-aleatórios são valores gerados a partir de um polinômio. Para padrões de testes determinísticos, todos os vetores utilizados no teste devem ser armazenados em alguma memória e para padrões de testes pseudo-aleatórios, é suficiente armazenar o polinômio ou as sementes utilizadas para gerar os vetores.

A desvantagem dos padrões pseudo-aleatórios em relação aos determinísticos está na cobertura de falhas. Padrões pseudo-aleatórios têm, em geral, menor cobertura de falhas ou precisam de um maior número de vetores para a mesma cobertura oferecida por padrões determinísticos. Em geral, não se consegue gerar todos os vetores de teste necessários a partir de um, ou um conjunto, de polinômios. Outro fato importante é que os polinômios geram uma grande quantidade de padrões desnecessários, aumentando significativamente o tempo de teste em muitos sistemas complexos.

Em [HEL 96] é proposto um modelo que utiliza um modo misto para a geração de padrões de testes para circuitos com cadeias *scan*. Um processador embutido é utilizado para gerar padrões de testes pseudo-aleatórios, diminuindo a necessidade de grandes espaços em memória. Em complemento, serão armazenados somente os padrões determinísticos necessários para cobrir as falhas que os vetores gerados pelo processador não consigam detectar.

Em modos mistos existe um forte compromisso entre o tempo de execução de um teste gerado por padrões pseudo-aleatórios e o espaço em memória ocupado pelos padrões determinísticos. Por isso, é importante explorar a flexibilidade do uso de software para gerar os padrões de teste em busca de um melhor desempenho. Por exemplo, um pequeno melhoramento no algoritmo que gera vetores pseudo-aleatórios, que conduz a um incremento na cobertura de falhas de 99,2% para 99,6%, reduz o número de falhas não detectadas e demanda uma área em memória que é a metade do espaço necessário para o armazenamento de padrões determinísticos que levariam à mesma cobertura [HEL 96].

O uso de um processador embutido permite essa flexibilidade, já que implementações em software podem ser capazes de fornecer padrões de teste com alta cobertura de falhas, diminuindo a quantidade extra de vetores determinísticos necessários. Outra vantagem

deste método é a insignificante quantidade de hardware extra para a execução do teste, uma vez que o hardware já está disponível no sistema.

O trabalho apresentado em [HEL 96] enfoca o uso de um modo misto para a geração de vetores, enfatizando a vantagem de menores exigências de memória. Porém, a exemplo do trabalho apresentado em [RAJ 99], este trabalho também não apresenta uma descrição do teste em nível de sistema.

O tempo de teste e o armazenamento de dados são grandes preocupações no teste de SOCs. Em sistemas integrados, se existe somente a possibilidade do uso de padrões determinísticos, os padrões para todos os núcleos do sistema devem ser armazenados no testador e transferidos para o *chip* durante o teste, como mostrado na figura 3.2. Isto pode causar uma série de problemas devido às limitações do ATE discutidos no início deste capítulo. Testadores são limitados em velocidade, capacidade de canal e memória. O tempo necessário para o teste de um SOC depende de quantos dados de teste precisam ser transferidos para o *chip* e quão rápido estes dados podem ser transferidos. Este tempo depende diretamente da velocidade e da capacidade do canal de dados do testador, bem como da organização das cadeias *scan* dentro do *chip* e dos caminhos de acesso a cada núcleo interno.

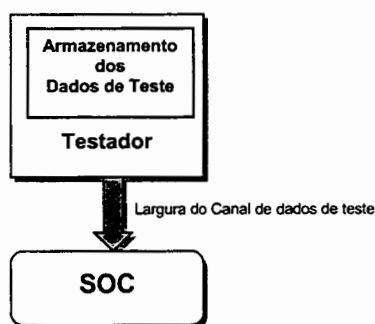


Figura 3.2 - Diagrama de blocos da transferência de dados de teste do testador para o *chip*.

[JAS 99] propõe um modelo onde um testador é utilizado para carregar um programa de teste e dados compactados de teste na memória de um microprocessador embutido. O processador executa o programa que realiza a descompactação dos dados de teste e os aplica nas cadeias *scan* dos núcleos de hardware do SOC. Este modelo reduz a quantidade de memória necessária para o armazenamento dos dados de teste no testador e também reduz o tempo total de teste. O tempo de teste é computado como sendo o tempo de transferência dos

dados para o SOC, descompactação e aplicação nas cadeias *scan*. O tempo de teste é reduzido pela menor quantidade de valores que são transferidos para o processador. Como a frequência do SOC tende a ser superior a frequência do testador, os dados podem ser descompactados e aplicados nas cadeias *scan* dos núcleos com uma velocidade superior em relação a um testador externo aplicando diretamente os dados de teste.

A vantagem no uso de vetores de testes determinísticos para testar núcleos com cadeias *scan* está em se conseguir uma maior cobertura de falhas com um menor tempo global de teste. Isto acontece porque o número de vetores a serem aplicados aos circuitos é, em geral, menor que os vetores exigidos por um método pseudo-aleatório.

O modelo de compactação/descompactação proposto por [JAS 99] está baseado na geração do próximo vetor de teste a partir de outro previamente armazenado, como é o caso de vetores diferença. Neste esquema, cada vetor de teste deve ser dividido em blocos de tamanho fixo, sendo que o tamanho do bloco de vetores de teste depende do tamanho da palavra do processador. O próximo vetor de teste é então construído a partir de um vetor prévio, substituindo os blocos nos quais eles diferem [JAS 99]. Por causa da relação estrutural entre falhas de um circuito, os vetores de teste possuem uma grande correlação entre si. Deste modo, os vetores podem ser ordenados de forma que dois vetores de teste sucessivos diferem em um número de blocos relativamente pequeno. Conseqüentemente, a quantidade de memória necessária para armazenar estas diferenças será muito menor que a quantidade exigida para armazenar o vetor de teste inteiro.

Estas diferenças são representadas por “palavras de substituição”, são pedaços codificados da informação que dizem para o processador como construir o próximo vetor a partir do anterior. Cada palavra de substituição tem três campos, como mostra a figura 3.3: um campo de 1-bit chamado de último *flag*, um campo de $\log_2 N$ bits chamado de número do bloco (onde N é o número de blocos nos quais o vetor de teste é dividido) e um campo de b bits chamado de novo bloco padrão. O número do bloco contém o endereço do bloco que será substituído com o novo padrão contido no campo de novos blocos padrões. Se dois vetores de teste sucessivos diferem em x blocos, então esta informação é representada como uma seqüência de x palavras de substituição, onde o campo último *flag* da x -th palavra de substituição (a última palavra da seqüência), tem seu último *flag* colocado em ‘1’. Todas as outras palavras de substituição tem seus *flags* colocados em ‘0’. O processador lê estas palavras e então substitui os blocos apropriados pelos novos blocos padrão. Quando o bit do campo *flag*

for habilitado, sabe-se que a formação do novo vetor de teste foi completada. Então, o vetor de teste é deslocado pelas cadeias *scan* aplicando o teste no(s) núcleo(s).

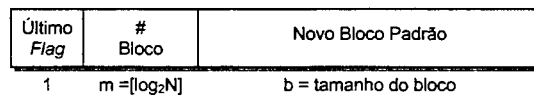


Figura 3.3 - Palavra de substituição.

O diagrama em blocos da figura 3.4 mostra a arquitetura proposta por [JAS 99]. Um processador é utilizado para carregar múltiplas cadeias *scan* nos núcleos do *chip*. A memória embutida possui as instruções que o processador deve executar e os dados com os quais o processador opera. Na fase de inicialização, antes do teste começar, o testador carrega o programa de teste na memória do *chip*. Então um conjunto de posições de memória é reservado para o testador carregar continuamente os dados codificados (palavras de substituição). Enquanto o processador estiver rodando o programa, ele gera os vetores de teste e os aplica nas cadeias *scan*. Paralelamente, o testador carrega continuamente a memória com as novas palavras de substituição.

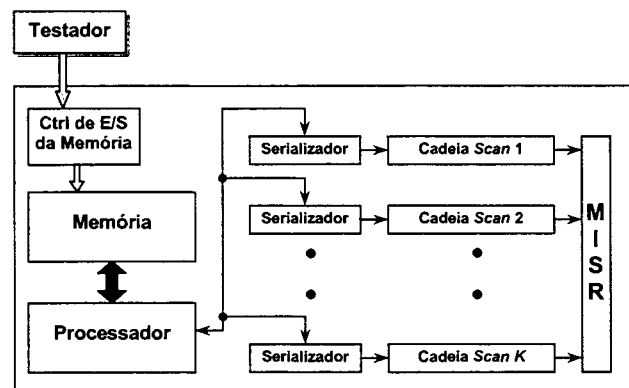


Figura 3.4 - Diagrama de blocos da arquitetura de teste proposto por [JAS 99].

Em geral, SOCs já possuem um controlador de entrada e saída da memória para fazer o interfaceamento com o mundo externo, durante o modo normal de operação. Este controlador pode ser utilizado pelo testador para carregar os dados na memória durante o teste.

Tipicamente este tipo de controlador é capaz de ser manipulado a uma frequência menor de relógio para facilitar o interfaceamento com sistemas externos. Deste modo, o testador pode facilmente trabalhar com uma frequência menor de relógio, enquanto o processador opera na frequência nominal de relógio do SOC, diminuindo o tempo de teste.

O mecanismo utilizado para aplicar os vetores de teste nas cadeias *scan* é controlado por um serializador. O processador envia cada bloco para o serializador apropriado em um determinado tempo e o serializador realiza o deslocamento serial dos blocos para as cadeias *scan*. Este serializador é um registrador controlado por uma máquina de estados e é capaz de deslocar um bit por pulso de relógio na cadeia. Quando todo vetor de teste estiver inserido na cadeia *scan*, o relógio do sistema é aplicado e então a resposta é deslocada para um MISR para ser compactada enquanto novos vetores de testes são introduzidos nas cadeias.

3.2.2 Microprocessadores Dedicados ao Controle de Teste

Em [DRE 98] um modelo de microprocessador dedicado é utilizado para implementar uma estratégia BIST de alto desempenho para testar memórias DRAMs embutidas. Uma estrutura do tipo microprocessador busca e executa uma série de instruções que formam padrões a serem aplicados na memória sob teste. Devido à flexibilidade de alguns elementos chave, uma grande variedade de padrões de testes pode ser programada para prover uma alta cobertura de falhas para DRAMs embutidas. A arquitetura do modelo proposto em [DRE 98] pode ser vista na figura 3.5.

O multiplexador da figura 3.5 serve para isolar todas as entradas das memórias DRAM embutidas e selecionar as entradas normais do sistema ou as entradas de teste utilizadas pela lógica BIST. Nas entradas do multiplexador e nos pinos de saída existem células *boundary-scan* (registradores). O seqüenciador é um microprocessador das instruções de teste. Ele executa cada instrução de teste e comunica as operações requeridas para os seguintes elementos: gerador de endereços, gerador de dados e controlador de endereços. O bloco gerador de relógio fornece todos os controles de relógios necessários para a inicialização do auto-teste, retirada de dados dos registradores (*scan-out*), e controle da DRAM. A comparação e compactação dos resultados do teste são executados por um elemento chamado “LER/COMPARAR” que fica dentro do bloco RAL (do inglês, *Redundancy Allocation Logic*). Este bloco determina se a memória DRAM embutida está perfeita ou foi reparada com auxílio de elementos redundantes da matriz. É também o bloco RAL que determina o uso ó-

timo dos elementos de ordem redundantes para o máximo rendimento alcançável e informa a localização dos fusíveis que devem ser queimados.

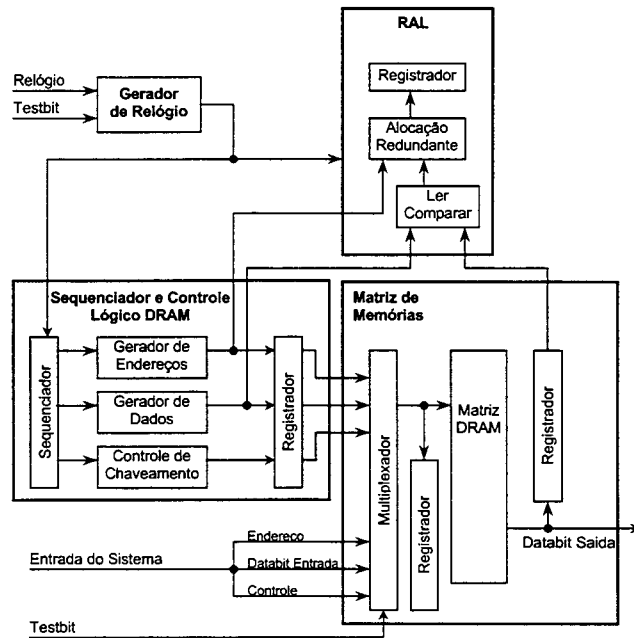


Figura 3.5 - Arquitetura BIST utilizando controlador interno [DRE 98].

Um microprocessador embutido foi utilizado para alcançar uma maior flexibilidade através do uso de padrões de teste programáveis, sem a necessidade de mudar o projeto. Um diagrama de blocos deste microprocessador, que é chamado de seqüenciador, pode ser visto na figura 3.6. O projeto deste microprocessador consiste de três componentes principais: um contador de instruções, a memória de instruções e um controlador de saltos.

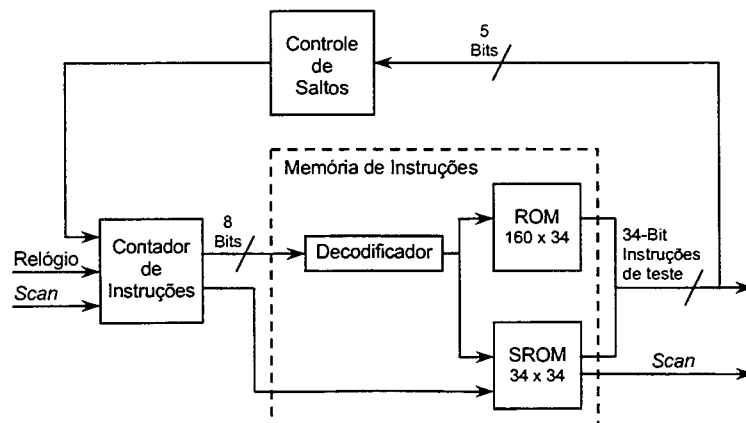


Figura 3.6 - Diagrama de blocos do seqüenciador.

O contador de instruções aponta para a seqüência de execução de um teste. O endereço apontado por ele representa, na memória de instruções, a localização da próxima instrução. Um contador de 8-bits é utilizado permitindo que sejam armazenados na memória até 256 instruções de teste.

A memória de instruções é composta por dois blocos de memorização (figura 3.6). O primeiro é uma memória do tipo ROM e é utilizado para armazenar as primeiras 192 instruções de teste. Nesta memória são armazenados as instruções relativas a padrões de teste tradicionalmente utilizados para cobrir falhas em DRAMs, além de qualquer outro tipo especial de padrão para teste de alta eficiência. Um padrão de teste possui em média 3 à 17 instruções de teste, dependendo da aplicação. O segundo bloco utilizado é uma memória do tipo SROM (do inglês, *Scannable Read-Only Memory*) que possui registradores endereçáveis com uma porta *scan*. Nesta memória é possível armazenar o restante das 64 instruções de teste. Alternativamente, ou em combinação, as instruções ou grupos de instruções na ROM podem ser mudadas de saltos para a SROM.

O controlador de saltos representa o circuito final do seqüenciador (microprocessador). Este circuito opera com cinco bits das instruções de teste para controlar a seqüência do contador. Um dos bits é utilizado para identificar que um “salto” é solicitado e os outros quatro bits permitem a codificação de 16 saltos condicionais diferentes. Os saltos condicionais incluem condições de saltos normais como *carry* do contador de endereços das linhas, *carry* do contador de endereços das colunas, etc. Se um salto não é codificado na instrução de teste, o controlador de saltos faz com que o contador de instruções seja incrementado e aponte para a próxima instrução.

As instruções são codificadas com 34 bits. Os 13 primeiros bits são utilizados para controlar a seqüência de operações a ser executada na DRAM embutida sob teste (oito bits para o contador de instruções e cinco para o controlador de salto). Os outros 21 bits são divididos da seguinte forma: sete bits para controle do gerador de endereços; seis bits para controle do gerador de dados; seis bits para o circuito de controle de chaveamento; um bit para instruções que não ocorrem operações e um bit para indicar que o teste foi completado.

Este é um exemplo de uso de um controlador embutido específico para o teste BIST de matrizes de memórias DRAMs. Em nível de SOC, este controlador poderia ter suas funções incrementadas e incorporar o teste de outros núcleos do sistema. Porém, é importante salientar que o uso de um controlador interno para o teste de apenas alguns núcleos conside-

rados críticos (que exigem maiores velocidades, maiores quantidades de vetores, etc) podem trazer um grande benefício ao teste em termos de tempo total de teste e custo.

Finalmente, [BEN 00] propõe o uso de um modelo distribuído para o controlador de teste, considerando testes baseados em cadeias de varredura *scan* (completas ou parciais) e núcleos com estruturas BIST autônomas. Neste trabalho, cada núcleo tem uma interface lógica para fazer a conexão entre os dados de teste e o controle dos barramentos. Núcleos em um mesmo nível de hierarquia são conectados no mesmo barramento de teste formando uma topologia tipo anel, conforme mostrado na figura 3.7.

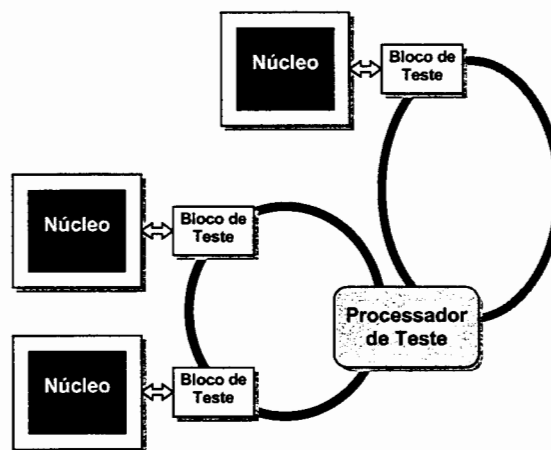


Figura 3.7 - Exemplo de estrutura de teste em forma de anel.

Os blocos de teste representam a lógica inserida na periferia do bloco funcional que é utilizada para controlar seus procedimentos de teste através de uma interface padrão. O gerenciamento do barramento de teste é feito por um processador de teste dedicado e as informações são transferidas entre os núcleos utilizando um protocolo auto-testável baseado na passagem de bastões (*tokens*). A interface entre os núcleos e o barramento de teste implementa um conjunto de comandos de teste que são emitidos pelo processador associado. O programa de teste para o sistema total é carregado a partir de um ATE, por meio de uma estrutura do tipo TAP (no caso de testes que utilizam cadeias *scan*) ou a partir de controladores BIST quando deseja-se executar um teste em núcleos com BIST.

Não foi claramente definido pelos autores, se a lógica envoltória necessária para cada núcleo é em parte ou no todo a lógica envoltória definida pelo padrão P1500 explicado

na seção 2.6. Também não foi tratado neste trabalho o aproveitamento das informações providas pelos desenvolvedores dos núcleos de hardware (por exemplo, como é tratado um núcleo que já venha com uma lógica envoltória padronizada).

4 O CONTROLADOR DE TESTE MET

A arquitetura MET (do inglês, *Microprocessor for Embedded Test*) proposta em [COT 01], foi elaborada pensando-se no desenvolvimento de um controlador interno, cujo objetivo seria único e exclusivamente o gerenciamento do teste em sistemas que utilizam núcleos de hardware com três estratégias de teste diferentes: o teste externo, *boundary-scan* e o auto-teste.

Com a finalidade de esclarecer a nomenclatura de testes adotada a partir de agora até o final neste trabalho é importante reconsiderarmos alguns conceitos sob o ponto de vista da hierarquia. Considerando-se o teste externo, que consiste, originalmente, na inserção de vetores nas entradas primárias de um circuito que deseja-se testar e na comparação das respostas fornecidas pelas suas saídas primárias. Estes valores são comparados com valores previamente conhecidos e que atestam a integridade do circuito. Quando observado externamente ao circuito integrado, o teste externo sendo executado em um ou vários núcleos que compõem um sistema maior, pode ser visto como um procedimento de auto-teste, uma vez que os elementos geradores e/ou avaliadores do teste se encontram inseridos no próprio contexto do sistema.

Neste trabalho considera-se como teste externo aquele tipo de teste onde vetores são injetados nas entradas primárias de um núcleo e posteriormente faz-se a comparação dos valores fornecidos pelo núcleo com valores previamente conhecidos. Logo, o termo “externo” é referente ao núcleo de hardware e não ao *chip*. Em termos de auto-teste, consideram-se núcleos de hardware que possuem a propriedade de se testarem autonomamente, a partir de um único estímulo externo. Todas as estruturas responsáveis pela geração de dados de teste (sinais de configuração, padrões de testes, etc), bem como as estruturas que fazem a análise das respostas (assinaturas), se encontram incorporadas dentro do circuito que compõe o núcleo de hardware. O estímulo externo é um sinal do tipo habilita o auto-teste. Uma vez ativado este sinal, o núcleo entra em modo teste, permanecendo neste estado até o término do auto-teste, momento no qual o núcleo retorna um sinal indicando se o mesmo está livre de falhas ou não.

Testes do tipo *boundary-scan* são aqueles regidos pela norma IEEE std 1149.1 conforme comentado na seção 2.7.

A arquitetura MET realiza a execução de um programa de teste que é gerado a partir de um código CTL/STIL. Este programa de teste está relacionado com o teste do sistema inteiro e deve ser construído levando-se em conta os diversos códigos CTL que acompanham todos os núcleos de hardware que devem ser testados, juntamente com a tradução das exigências de teste dos núcleos locais e dados para o contexto global. Assim, as instruções a serem executadas pelo microcontrolador correspondem ao tipo de teste descrito pelos códigos CTL. Uma vez que esta linguagem não está completamente definida e muito menos disponível, a linguagem STIL foi utilizada para definir as instruções de teste do MET, já que a linguagem CTL será fortemente baseada em STIL.

4.1 LINGUAGEM STIL

A linguagem STIL (do inglês, *Standard Test Interface Language*) foi desenvolvida, inicialmente, pelos fornecedores de equipamentos de teste, fabricantes de circuitos integrados e fornecedores de ferramentas de CAD, para facilitar a transferência dos dados entre o circuito e o equipamento de teste.

Um programa STIL define as formas de onda dos sinais a serem enviados como estímulos ao circuito sob teste, contendo os tempos de envio e os valores envolvidos em cada sinal. O programa define, ainda, uma seqüência de vetores de teste que serão aplicados e os valores esperados como resposta do teste. Dentro do contexto do MET, a linguagem STIL foi utilizada como um formato intermediário para um subseqüente processamento destas informações pelo microcontrolador.

O programa de teste da figura 4.1 é um exemplo de um programa STIL. Este exemplo descreve parte do teste de um barramento de 8 bits (*transceiver*) [COT 01]. O bloco *SignalGroups* do código STIL mostrado na figura 4.1 é utilizado para indicar se os sinais são de entrada ou saída, e também quantos bits possuem. O bloco *WaveFormTable* traz informações sobre as restrições de tempo dos vetores de teste e os possíveis valores à serem aplicados em um sinal. Cada caracter (0, 1, H, L e Z) associado a um sinal representa um possível vetor de teste a ser aplicado por este sinal. Para o sinal OEM, por exemplo, um vetor de teste chamado de "0" representa a seqüência de valores lógicos '1' em 0ns, '0' em 200ns e novamente '1' em 300ns, como pode ser visto na figura ao lado da seqüência definida para o sinal OEM.

O período define o tempo de duração de cada padrão de teste (500ns para este exemplo) e a seqüência dos vetores de teste a serem aplicados por todos os sinais é determinada pelo bloco *Pattern*. Neste bloco, os valores são mapeados para um sinal chamado de *ALL*, definido no bloco *SignalGroups*. Deste modo como mostra a figura 4.2, o segundo padrão de teste indica que os sinais DIR e OEM recebem os vetores de teste descritos pelo caracter “0” associado a estes sinais, o primeiro bit de ABUS recebe o vetor de teste definido pelo caracter “1”, enquanto os outros 7 bits recebem o vetor definido pelo caracter “0” deste sinal. Por outro lado, o primeiro bit de BBUS, deve ser comparado com os valores definidos no vetor associado H, enquanto os outros 7 bits são comparados com os valores descritos pelo vetor L.

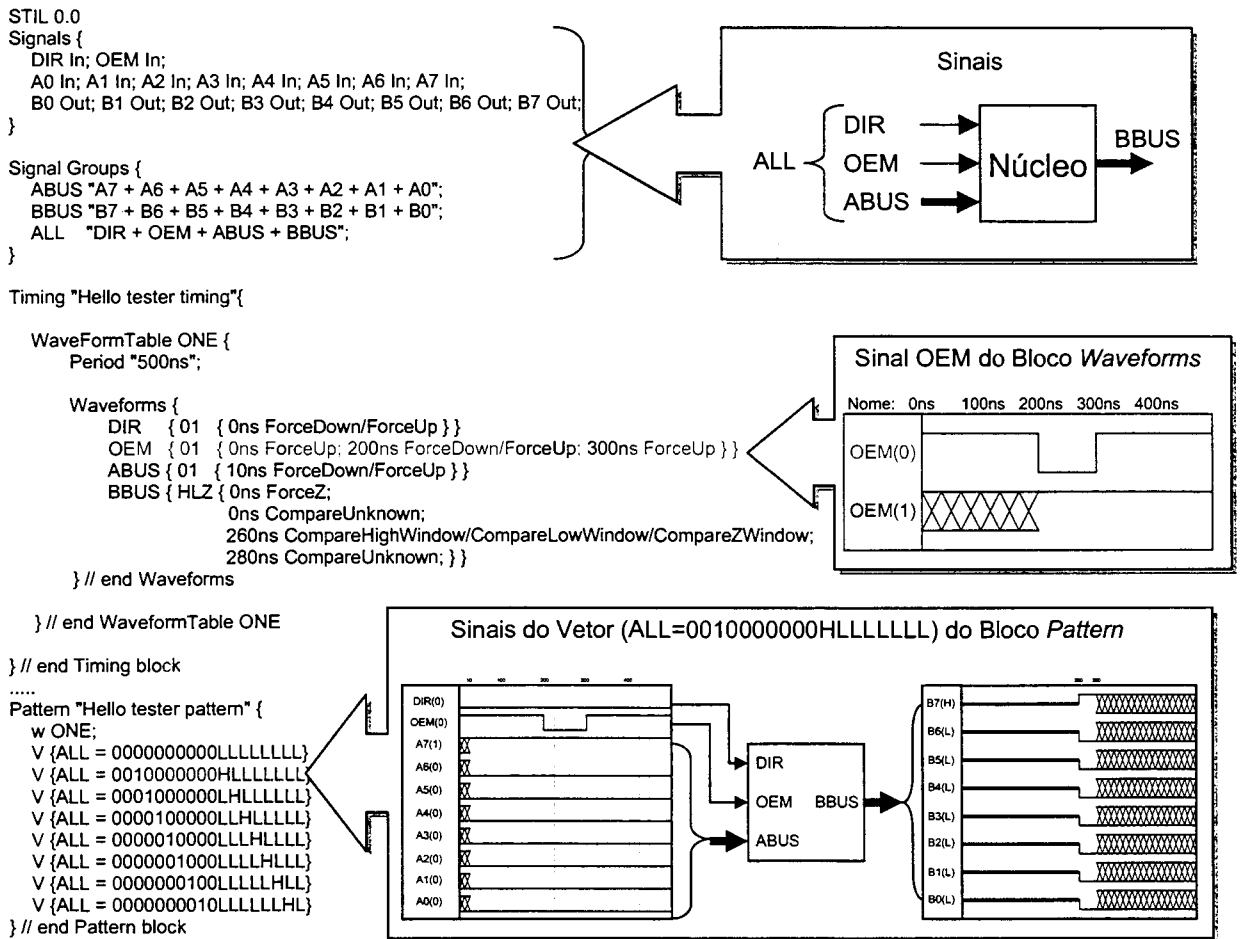


Figura 4.1 - Exemplo de programa STIL.

$V \{ALL = 001000000HLLLLLLLL\}$

Figura 4.2 - Segundo vetor do bloco *Pattern*.

Informações sobre testes através de cadeias de varredura são determinadas de forma semelhante por este tipo de linguagem. Porém, algumas informações extra sobre o tamanho das cadeias e o relógio de teste devem ser incluídas.

4.2 PROPOSTA DA ARQUITETURA MET

A arquitetura proposta para o MET surgiu a partir da análise de exemplos de programas STIL. Com base nestes exemplos, foram definidas as instruções necessárias para a execução dos testes.

Existe um nível de compilação acima da arquitetura MET que é responsável por gerar os comandos para o microcontrolador, ou seja, o compilador recebe um programa de teste descrito em STIL, faz o cruzamento de informações entre as formas de ondas e os vetores de tempo e gera duas tabelas que são enviadas para o microcontrolador. Estas duas tabelas contêm informações sobre tempos de execução, instruções e padrões de testes, e são armazenadas em duas memórias, as quais são chamadas de memória de programa e memória de dados.

A memória de programa armazena os valores referentes às instruções e tempos de teste, sendo estes ordenados pelo tempo de execução dentro de cada vetor de teste. A memória de dados contém os padrões de teste (que são os valores a serem aplicados nas entradas do circuito) e valores para comparação com as respostas fornecidas pelo núcleo. Nesta memória, também são armazenados os valores utilizados em um teste do tipo *boundary-scan*, além de dois bits específicos para o auto-teste, dos quais, um dos bits é utilizado para habilitação e o outro é utilizado para verificação do resultado.

4.2.1 Instruções do Microcontrolador

Com base nos exemplos de programas STIL e também nas informações sobre as estratégias de teste que o MET deve executar, 10 (dez) instruções foram definidas para serem executadas pelo microcontrolador de teste. A partir de um tempo inicial (tempo zero) de teste, as instruções são realizadas em seqüência. Algumas instruções devem ser executadas em momentos específicos (dentro do período do vetor), como mostrado no exemplo STIL.

1. **EXCITA** < *tempo* > (E) \Rightarrow indica que um padrão de teste deve ser enviado ao núcleo em um determinado tempo de teste. Esta instrução tem como parâmetro o tempo e o valor dos estímulos a serem enviados para o núcleo. Na memória de programa existirá o código referente à instrução EXCITA e o tempo de teste correspondente para a mesma. Os estímulos que devem ser enviados ao núcleo estão armazenados na memória de dados. Esta deve possuir uma largura de palavra suficientemente grande para armazenar todos estímulos de entrada do circuito sob teste, bem como os valores para comparação. As estruturas das memórias serão apresentadas detalhadamente mais adiante.
2. **COMPARA** < *tempo* > (C) \Rightarrow indica que um valor fornecido por um núcleo deve ser comparado com um valor esperado que está armazenado na memória de dados. Quando o microcontrolador estiver executando uma instrução deste tipo, na memória de programa existirá o código referente a esta instrução e o tempo de teste correspondente. A memória de dados contém os valores esperados para os bits de saída do núcleo naquele instante de tempo, sempre obedecendo a uma determinada ordem que foi previamente definida pelo compilador que gerou o programa de teste. O resultado da comparação entre os valores fornecidos pelo núcleo e os valores armazenados na memória de dados indica a presença de falhas ou o seu funcionamento correto.
3. **EXCITA&COMPARA** < *tempo* > (EC) \Rightarrow com esta instrução é possível enviar um vetor de teste ao mesmo tempo em que se faz a comparação com uma resposta (de um teste anterior) fornecida pelo núcleo. Esta instrução favorece testes *at-speed*, onde deseja-se que o teste se realize na frequência de operação do circuito sob teste.
4. **FIM_VETOR** (FV) \Rightarrow indica que um vetor de teste foi concluído. Quando esta instrução é executada, a contagem de tempo de teste é zerada e o microcontrolador aguardará o tempo de envio do próximo vetor.
5. **SC_TMS** < *tamanho_da_cadeia* > (SS) \Rightarrow instrução *Scan* que indica que um sinal de TMS deve ser enviado à TAP de um núcleo que implementa o padrão JTAG. Esta instrução tem como parâmetro o número de deslocamentos dentro da cadeia *scan* (tamanho da cadeia), e os valores de TMS. Os valores referentes ao

tamanho da cadeia são armazenados na memória de programa e os valores de TMS são armazenados na memória de dados.

6. **SC_TMTI** < *tamanho_da_cadeia* > (SI) \Rightarrow instrução *Scan* que indica que os valores devem ser enviados para os sinais de TMS e TDI de um núcleo baseado em JTAG. Também nesta, o número de deslocamentos deve vir acompanhando a instrução.
7. **SC_TMTO** < *tamanho_da_cadeia* > (SO) \Rightarrow instrução *Scan* que define os valores para TMS e o valor esperado como resposta do núcleo. Além disso, ela executa a comparação dos valores lidos da saída TDO do núcleo com o valor de resposta esperado.
8. **SC_TCK** (SK) \Rightarrow instrução *Scan* que define a frequência do relógio para a cadeia *Scan*. Este sinal foi previsto, porém, não foi implementado na prática.
9. **FIM_TESTE** (FT) \Rightarrow indica o término do teste de um núcleo, ou seja, a máquina de estados do controle do microcontrolador fica em um estado de espera, aguardando um novo teste.
10. **FIXA_CONTADOR** < *valor* > \Rightarrow define o passo para o relógio do microcontrolador. Este sinal serve para representar os valores de tempo com um número menor de bits, como será explicado na seção 4.3.1.

Para a codificação das instruções do microcontrolador foram utilizados quatro bits, conforme mostrado na tabela 4.1. As instruções FV, C, E e EC são utilizadas para as aplicações de testes descritos no bloco *Patterns* de uma programa STIL e as instruções SS, SI, SO e SK são utilizadas por núcleos de hardware baseados no padrão JTAG.

Tabela 4.1 - Codificação das instruções.

Instruções	Código
FIM_TESTE (FT)	0000
FIM_VETOR (FV)	0001
COMPARA (C)	0010
EXCITA (E)	0011
EXCITA&COMPARA (EC)	0100
SC_TMS (SS)	1000
SC_TMTI (SI)	1001
SC_TMTO (SO)	1010
SC_TCK (SK)	1011

Como mencionado anteriormente, assume-se que existe um código STIL/CTL gerado por alguma ferramenta de planejamento de teste (ordem de teste, mecanismo de acesso, etc.) descrevendo o teste do sistema. Assim, este código deve ser traduzido por um programa baseado nas instruções definidas anteriormente. Uma ferramenta capaz de fazer esta tarefa está, atualmente, em fase de desenvolvimento na UFRGS. Durante este trabalho é considerado que um programa STIL, como o mostrado na figura 4.1, está disponível para ser traduzido e executado pelo microcontrolador de teste.

4.3 COMPONENTES PRINCIPAIS DA ARQUITETURA MET

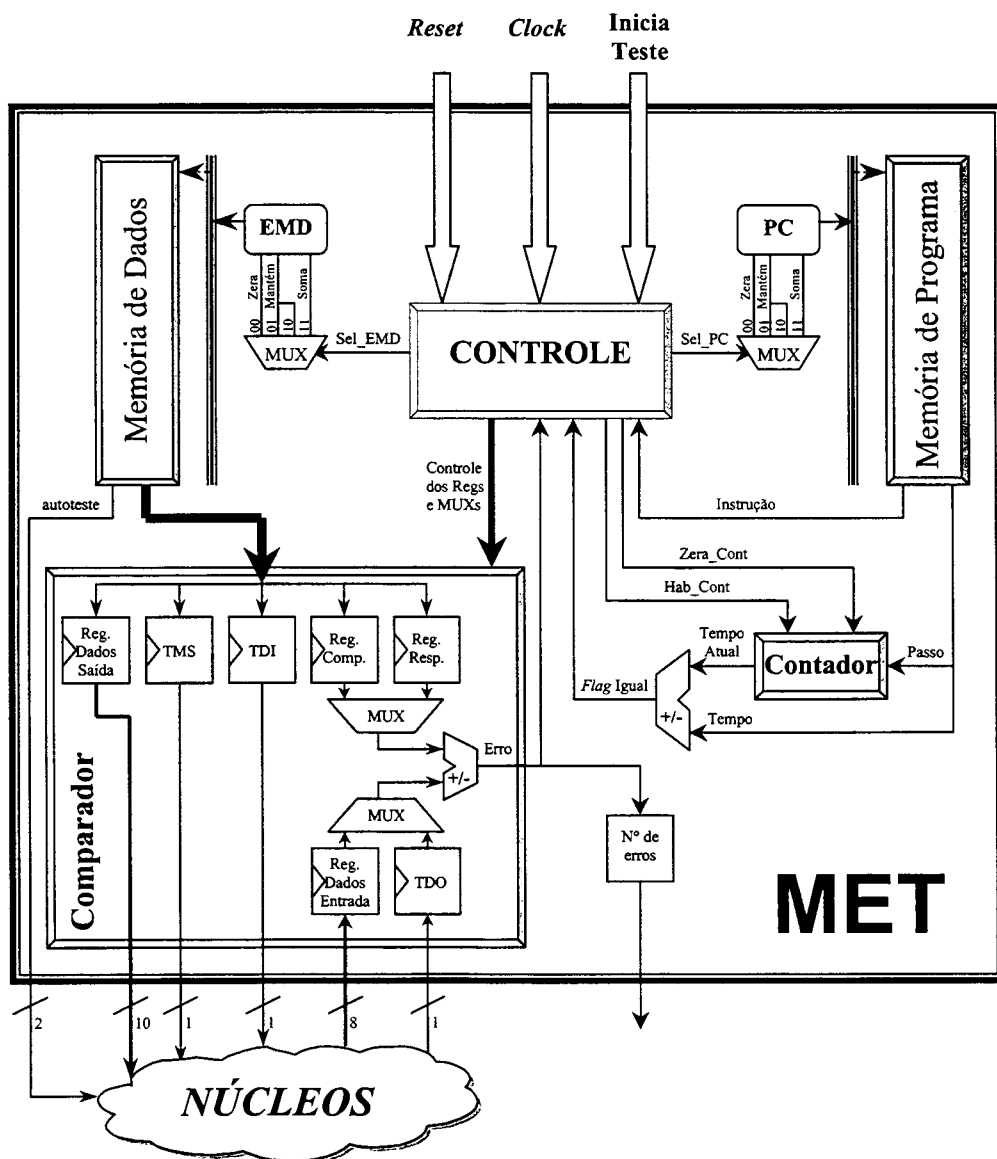


Figura 4.3 - Arquitetura MET [COT 01].

A implementação mostrada na figura 4.3 apresenta os três blocos principais que compõem o MET. Estes blocos são: controle, contador de eventos e comparador. Todos os blocos foram descritos em VHDL e são instanciados na descrição do controlador de teste, juntamente com as memórias, alguns registradores e multiplexadores. Nesta instância, também existe um comparador, que verifica se, em um determinado passo do contador de eventos, existe alguma instrução a ser executada. Em outras palavras, este comparador compara o valor do contador de eventos com o tempo de teste lido da memória de programa. Também são utilizados dois contadores (tanto na memória de programa como na memória de dados), responsáveis pelo apontamento da próxima palavra de memória a ser lida, passando para a palavra seguinte somente depois de verificar que o tempo do contador de eventos é igual ao valor lido anteriormente da memória de programa.

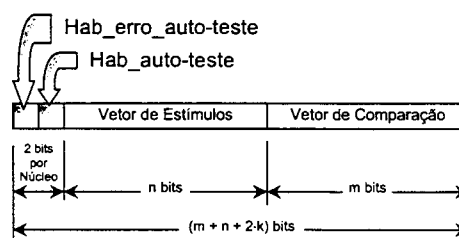
4.3.1 Memória de Programa

Como comentado anteriormente, a partir de um programa CTL/STIL um compilador gera os dados que serão armazenados na memória de programa, indicando a instrução e o tempo de teste em que a mesma deve ser executada. As instruções são ordenadas na memória de acordo com o tempo, sendo que o tempo é fornecido em passos do relógio. Um contador chamado de PC é responsável pelo apontamento do próximo endereço da memória de dados a ser lido. A informação de tempo está representada em passos múltiplos do relógio do microcontrolador, de modo que o valor que indica o tempo pode ser representado com um número menor de bits. Esse valor de passo é gerado pelo compilador que analisa os tempos de todos os testes do sistema e encontra o valor mais apropriado como base de tempo.

Para instruções *Scan*, armazena-se na memória a instrução seguida do número de deslocamentos que devem ser realizados nos registradores que fazem parte da infra-estrutura de um teste JTAG, ao invés do tempo de ocorrência da instrução. Esse procedimento foi adotado, porque em testes baseado no padrão *boundary-scan* sempre carrega-se os vetores de teste de forma serial. Assim, a informação sobre o tamanho da cadeia *scan* é o que realmente interessa. Para as instruções *scan*, o MET realiza um deslocamento a cada ciclo de relógio, independente do valor que foi definido para o passo. O tamanho de palavra da memória de programa foi definido como 12 bits, sendo que 4 bits são utilizados pela instrução e os outros bits são utilizados para armazenar o tempo de teste ou o número de deslocamentos, para o caso de instruções *Scan*.

4.3.2 Memória de Dados

Os padrões de teste coletados pelo compilador são armazenados na memória de dados. Nestes padrões encontram-se os valores utilizados para excitar o núcleo sob teste e também os valores utilizados para a comparação com as respostas fornecidas. Na memória de dados existem dois bits, por núcleo de hardware, que se destinam exclusivamente a habilitar o auto-teste e indicar que o microcontrolador deve verificar o resultado obtido. O bit que habilita o auto-teste deve ser fixado no nível lógico '1' e permanecerá assim até o término do auto-teste. A figura 4.4 mostra como está dividida a palavra da memória de dados do MET.



A largura da palavra da memória de dados é igual a $(m + n + 2 \cdot k)$ bits, onde:
 n = tamanho do vetor de estímulos
 m = tamanho do vetor de comparação
 k = número de núcleos com auto-teste

Figura 4.4 - Palavra da memória de dados.

O contador responsável pelo incremento dos endereços da memória de dados é chamado de EMD. Os apontadores de endereçamento das duas memórias (dados e programa) são controlados pelo bloco de controle do microcontrolador.

Para um teste externo, a largura da memória de dados deve ser suficientemente grande para armazenar todos os estímulos e respostas do teste. Isto porque cada bit da memória está associado a um sinal do núcleo que se deseja testar. Por exemplo, para testar um núcleo que possui 10 entradas e 8 saídas, a largura de memória deve ter 18 bits destinados ao teste externo, 10 bits serão utilizados para armazenar o vetor de estímulos do núcleo e os outros 8 bits serão utilizados para armazenar o vetor de comparação.

No caso de uma instrução SC_TMS, toda a largura da memória, com exceção dos bits reservados para o auto-teste, é utilizada para armazenar os valores enviados ao sinal de TMS do controlador de TAP do núcleo. Para as instruções SC_TMTI e SC_TMTO, a largura da memória deve ser dividida em duas partes, sendo que uma parte armazena os dados para o

signal de TMS da TAP, e a outra, os dados a serem enviados ao pino de TDI do núcleo, caso a instrução seja SC_TMTI. Se a instrução for SC_TMTO, os valores armazenados são os valores que serão utilizados para a comparação dos sinais vindos de pino de TDO do núcleo. A figura 4.5 mostra como ficariam as memórias de programa e dados do MET para o exemplo de programa STIL apresentado na figura 4.1.

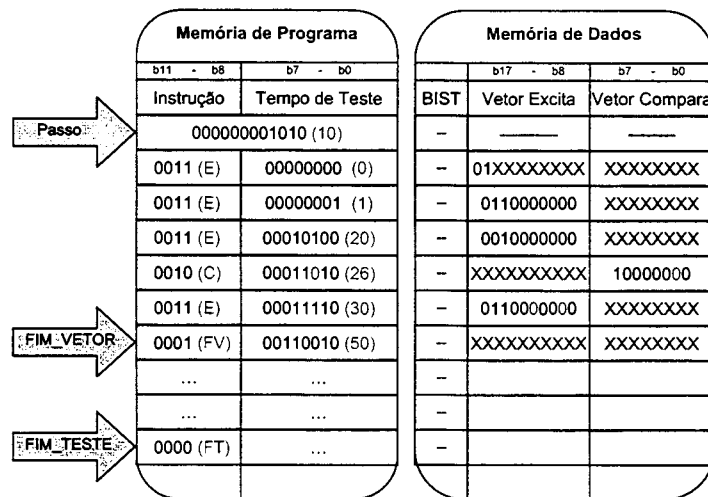


Figura 4.5 - Memória de programa e de dados para o exemplo de código STIL.

O cálculo do passo para este exemplo foi realizado sem levar em consideração o relógio do interpretador. Os tempos de testes exigidos pelas formas de ondas encontradas no programa STIL são de 0ns, 10ns, 200ns, 260ns e 300ns, logo o valor que representa o passo é de 10 ciclos de relógio, pois este é o máximo divisor comum entre todos os tempos envolvidos. A figura 4.5 não mostra toda a memória para o exemplo do teste, mas sim, apenas o segundo vetor de teste. Na figura da memória de dados, da esquerda para a direita, tem-se os valores para os sinais DIR (bit17), OEM (bit16), ABUS (bit8 à bit15), BBUS (bit0 à bit7).

4.3.3 Contador de Eventos

O contador de eventos é o bloco responsável pelo controle do tempo de execução das instruções do microcontrolador, ou seja, é este bloco quem controla o instante de tempo para execução de uma determinada ação. O compilador deve fornecer o valor que servirá de passo para o contador de eventos (base de tempo). Este valor deve ser um múltiplo da frequência de funcionamento do microcontrolador, e sempre será armazenado no endereço 0 (zero) da memória de programa. A partir da contagem efetuada pelo contador de eventos, é

determinado o instante de tempo para o envio de um padrão de teste para o núcleo ou para a leitura de uma resposta.

4.3.4 Comparador

Neste bloco encontram-se os registradores que fazem parte da interface com o núcleo. Estes registradores armazenam os valores que serão enviados ao circuito que se deseja testar, no caso de uma instrução de envio de estímulos, e os valores a serem utilizados na comparação com as respostas vindas dos circuitos sob teste, quando a instrução for de comparação. Além dos registradores, este bloco também contém um subtrator, que é o elemento responsável pela comparação entre o valor desejado e o valor obtido como resposta do núcleo.

Os registradores utilizados para o teste externo (bloco comparador da figura 4.3) são chamados de Reg_Dados_Saída e Reg_Comp que recebem os valores da memória de dados para fazer o envio dos dados de teste e comparação, respectivamente, e o Reg_Dados_Entrada que recebe os valores vindos dos núcleos para comparação com o valor armazenado em Reg_Comp. Estes registradores, como no caso da memória de dados, devem possuir tamanhos de acordo com os núcleos que se deseja testar. Os registradores utilizados para testes em núcleos baseados em JTAG são TMS, TDI, TDO, além de um registrador que recebe o valor esperado para a resposta do núcleo (Reg_Resp). O valor deste registrador é utilizado como comparação com o valor do registrador TDO. Dois multiplexadores são utilizados para fazer a seleção dos registros de comparação dependendo do teste ser externo ou baseado em JTAG.

4.3.5 Controle

Este bloco é composto de uma máquina de estados finitos (FSM, do inglês, *Finite State Machine*), a qual pode ser observada na figura 4.6. A máquina de controle gera os sinais responsáveis pela carga dos registradores que compõem o bloco comparador, realiza a seleção dos multiplexadores, zera o contador de eventos, e habilita a contagem deste mesmo contador.

A máquina de estados do bloco de controle sai do estado 0 (zero) para o estado 1 quando o sinal de INÍCIO muda para o nível lógico '1'. No estado 1, ocorre a gravação do passo do contador de eventos, sendo que este sempre está situado na posição zero da memória de programa. O laço no estado 3 fica ativado até que o valor da contagem do contador de eventos seja igual ao tempo de teste lido na memória de programa. No caso de uma instrução de FIM_TESTE, a máquina vai para o estado 4, e após volta para o estado 0 (zero). Quando a

instrução *FIM_VETOR*, a máquina de estados vai para o estado 5 e deste retorna para o estado 3. Caso nenhuma instrução *Scan* seja solicitada, a máquina retorna para o estado 2.

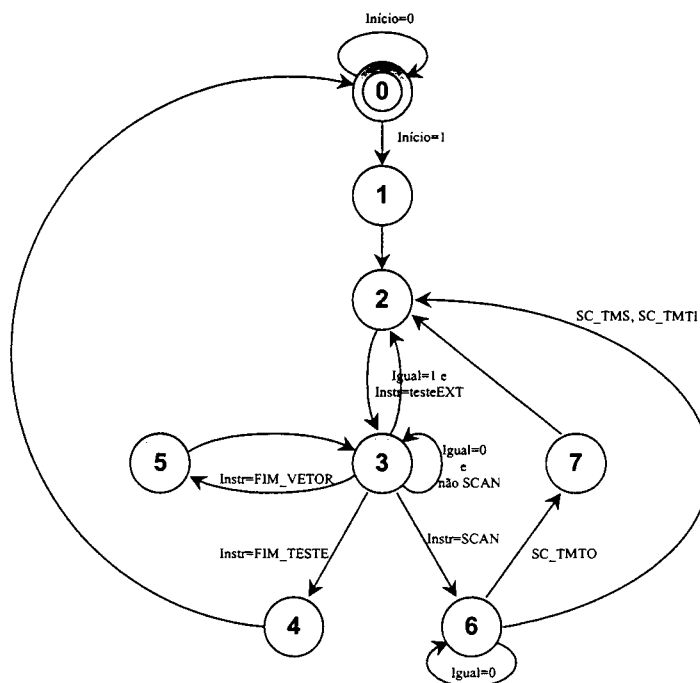


Figura 4.6 - Diagrama de estados do bloco de controle do MET.

Os estados 6 e 7 destinam-se às instruções *Scan*. Quando o microcontrolador reconhece uma instrução *Scan*, o laço que testa o resultado da comparação do valor do contador de eventos com o valor lido na memória de programa não ocorre no estado 3, mas sim, no estado 6, onde são gerados os sinais para a operação de deslocamento dos registradores envolvidos no teste *boundary-scan*. O estado 7 é utilizado para execução da instrução *SC_TMTO*, servindo para habilitar a comparação e selecionar os valores vindos do registrador TDO e do registrador que armazena a resposta esperada, como entradas do comparador. Neste estado também ocorre o incremento do contador de erros, caso os valores comparados sejam diferentes.

O compilador deve indicar o número de deslocamentos que devem ser feitos nos bits dos registradores utilizados em testes *SCAN*. Na implementação do MET, considera-se que o valor colocado na memória de programa, que será utilizado na comparação com a contagem do contador de eventos para indicar a saída do laço (estado 6), é uma unidade menor que o número de deslocamentos que devem ser realizados. Esse procedimento foi utilizado para diminuir o número de estados da máquina de controle, pois, o resultado da comparação é

testado no mesmo estado em que se habilita o deslocamento dos registradores envolvidos na instrução.

É importante salientar que o passo do contador para um teste com *boundary-scan* deve ser 1 (um), sendo este valor automaticamente assumido pelo microcontrolador MET quando este reconhece uma instrução *Scan*. Devido à limitação no tamanho da palavra da memória de dados e do elemento de temporização, não pode ocorrer o teste de um núcleo baseado em JTAG paralelamente à execução de um teste externo de um outro núcleo. Porém, o teste de um núcleo com BIST autônomo pode ocorrer paralelamente ao teste externo ou ao teste *Scan* de um outro núcleo.

Nos estados 1, 2 e 5 o PC (apontador da memória de programa) é incrementado. O contador EMD (apontador da memória de dados) é incrementado no estado 2.

4.4 EXECUÇÃO DO TESTE

Quando o microcontrolador MET recebe o sinal de início de teste, ele lê o endereço 0 (zero) da memória de programa e carrega o passo que será utilizado durante o teste. Após, é incrementado o apontador da memória de programa (PC) e a primeira instrução e o primeiro tempo de teste são lidos. Neste momento, o contador de eventos é inicializado e efetua-se a comparação da contagem do mesmo com o tempo de teste. Quando estes sinais forem iguais, a instrução é verificada. Se a instrução for *EXCITA* ou *EXCITA&COMPARA*, um padrão é colocado nas entradas primárias do núcleo, ou seja, o registrador de saída da interface é carregado com o valor lido da memória de dados e incrementa o seu apontador de endereços (EMD). Se a instrução for *COMPARA* ou *EXCITA&COMPARA*, o registrador de interface é carregado com o valor lido nas saídas primárias do núcleo, um outro registrador é carregado com os valores esperados, que estão armazenados na memória de dados, e os conteúdos destes dois registradores são comparados. No caso destes valores serem diferentes, incrementa-se o conteúdo do registrador que armazena o número de erros. Para a instrução *EXCITA&COMPARA*, os padrões são enviados da interface para o núcleo ao mesmo tempo em que as respostas são recebidas do núcleo e comparadas com os valores armazenados. Assim, testes do tipo *at-speed* podem ser realizados. Se a instrução for *FIM_VETOR*, o contador de eventos é zerado, PC é incrementado e o microcontrolador aguarda até que a contagem do contador de eventos chegue ao primeiro tempo de teste envolvido no próximo vetor. Se a instrução for *FIM_TESTE*, o contador é zerado e o microcontrolador fica aguardando por um sinal que inicie outro teste.

Quando uma instrução `SC_TMS` for solicitada, o registrador TMS é carregado paralelamente com o valor lido da memória de dados e repassa este valor, serialmente, para o núcleo, sendo seu valor deslocado para a direita uma posição a cada ciclo de relógio. O número de deslocamentos está descrito ao lado da instrução. O mesmo ocorre para a instrução `SC_TMTI`, quando o registrador TDI é carregado e tem seu valor deslocado e enviado serialmente para o núcleo. No caso da instrução ser `SC_TMTO`, o registrador TDO, diferentemente dos demais registradores *Scan* da arquitetura, é carregado serialmente, mas a sua leitura é paralela. Para um núcleo que possua uma cadeia *boundary-scan* muito grande, o compilador gera várias instruções *Scan* de forma que os deslocamentos gerados por estas instruções preencham (`SC_TMTI`) ou esvaziem (`SC_TMTO`) toda a cadeia.

4.5 EXEMPLO DE PROGRAMA DE TESTE *SCAN* PARA UM NÚCLEO BASEADO EM JTAG

Um exemplo de um programa de teste é a seqüência de sinais para o teste de um circuito com uma cadeia *scan* de 12 bits. Os valores de TMS determinam o funcionamento da máquina de controle do *boundary-scan* (TAP), de acordo com o padrão IEEE std. 1149.1 e com as cadeias de varredura (*scan*) implementadas internamente no circuito de teste. Cada seqüência de valores de TMS corresponde à execução de determinada operação, definida na TAP, como a carga da instrução, que seleciona a cadeia de 12 bits, e os deslocamentos necessários para se inserir ou retirar valores da cadeia por TDI e TDO, respectivamente. Nas tabelas 4.2 e 4.3 mostra-se como ficariam os sinais para a TAP. Nestas tabelas, consideram-se:

? : bits que dependem da implementação da TAP no circuito. Os três bits na tabela representam o código da instrução que seleciona a cadeia *scan* de 12 bits.

Z : Alta impedância.

X : valor *don't care*.

Tabela 4.2 - Primeira seqüência de definição dos sinais.

Sinais	Reset	Captura IR_Scan	Desloca IR_Scan (carrega instrução)	Captura DR_Scan
TMS	11111	01100	001	1100
TDI	XXXXX	XXXXX	???	XXXX
TDO	ZZZZZ	ZZZZZ	ZZZ	ZZZZ

Tabela 4.3 - Segunda seqüência de definição dos sinais.

Sinais	Desloca DR_Scan (entra vetor de teste)	Atualiza DR_Scan	Captura DR_Scan	Desloca e verifica valor	Reset
TMS	000000000001	1	100	0000000000001	1111
TDI	110101110110	X	XXX	XXXXXXXXXXXX	XXXX
TDO	ZZZZZZZZZZZZ	Z	ZZZ	110110110101	XXXX

A figura 4.7 mostra como ficariam as memórias do microcontrolador MET, para a execução de um teste do tipo *stuck-at-1* para o circuito.

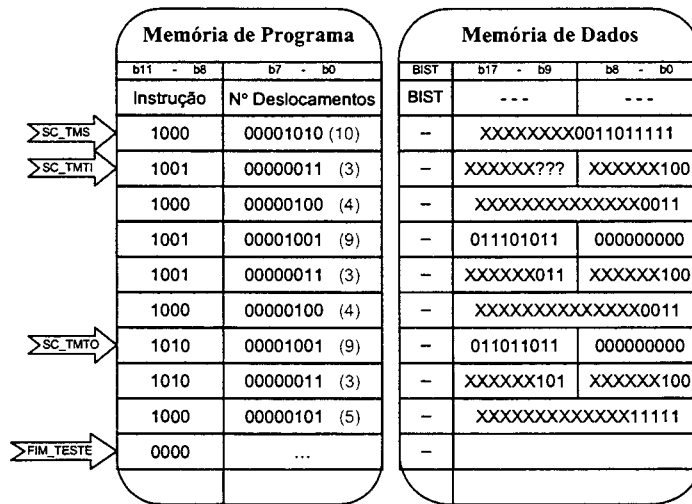


Figura 4.7 - Memória de programa e de dados com valores para um teste scan.

Na primeira posição da memória de dados estão os valores de *Reset* (1111) e de *Captura IR_Scan* (01100). Na segunda posição, estão os sinais de TMS e o valor da instrução (???) que será armazenado no registrador de instruções da TAP. O valor de *Captura DR_Scan* (1100) está na posição três e os valores a serem enviados para o sinal TDI (110101110110) e TMS (000000000001) do circuito estão nas posições quatro e cinco respectivamente. Uma vez que a cadeia *scan* é maior que o número de bits da memória de programa destinados aos sinais de TDI, os valores devem ser divididos, sendo que a instrução SC_TMTI deve ser solicitada duas vezes (até que a cadeia esteja completa). *Atualiza DR_Scan* (1), *Captura DR_Scan* (100) estão colocados na posição seis. Os valores utilizados pela instrução SC_TMTO para a comparação com os valores deslocados do sinal de TDO do circuito estão nas posições sete e oito. Observa-se novamente a divisão deste valor nas duas posições de memória. Na última posição estão os valores para um novo *Reset*.

4.6 RESULTADOS ENTRE SÍNTESES

A tabela 4.4 mostra um comparativo de resultados entre a síntese VHDL do microcontrolador MET com as sínteses, também em VHDL, de outros microprocessadores/microcontroladores e de um filtro LMS. A descrição do 8051 utilizada é uma versão reduzida (com menos instruções), porém compatível com a versão clássica deste microcontrolador. A versão com auto-teste do mesmo microcontrolador foi apresentada em [COT 99]. RISCO – 32 bits e RISCO – 16 bits são processadores RISC de 32 bits e 16 bits respectivamente, e FentoJAVA é um interpretador *byte-code* JAVA descrito em [ITO 00]. O filtro LMS é composto por multiplicadores e 32 coeficientes adaptativos de 8 bits que são armazenados em memória. Todos os circuitos foram simulados utilizando o software MAX+PLUS II da Altera® e, a fim de obter-se dados comparativos mais coerentes, estes foram simulados com um único dispositivo (EPF10K100GC503-3). Os processadores utilizados na comparação possuem diferentes tamanhos de palavra, frequências de operações e área, como mostrado na tabela 4.4.

Tabela 4.4 - Comparativo de performance e área entre circuitos.

Microcontrolador	Nº de Células Lógicas	% do Dispositivo Usado	Frequência (MHz)
8051 – 8 bits	724	14%	3,19
8051_BIST – 8 bits	2450	49%	2,12
RISCO – 16 bits	797	15%	8,5
RISCO – 32 bits	1159	23%	4,85
FentoJAVA	1491	29%	4,42
Filtro LMS – 8 bits	182	3%	12,40
MET	257	5%	34,24

Observa-se que a área ocupada pelo MET possui um incremento de, aproximadamente, 42% da área utilizada pelo filtro LMS, que é o dispositivo que ocupa a menor área no comparativo. Analogamente, o MET é cerca de 5,75 vezes menor em área que o maior processador, ou seja, o MET ocupa, aproximadamente, 17,4% da área ocupada pelo processador FentoJAVA. Fazendo uma análise em nível de sistema, um exemplo seria um sistema formado por um processador escalar e um filtro [FRA 00], como é o caso de um SOC composto por

um processador 8051 e um filtro LMS que, juntos, somariam 906 células lógicas. O microcontrolador MET ocuparia uma área de 28% em relação ao sistema.

Por outro lado, o desempenho em frequência de operação do microcontrolador MET é, certamente, muito maior que os processadores de uso geral aqui mostrados. Isto significa que o MET pode executar várias instruções enquanto um núcleo do tipo microcontrolador espera por uma excitação, por exemplo. Isto mostra a vantagem de se definir um processador dedicado para o controle do teste, ao invés de se reutilizar o processador disponível na aplicação.

5 ESTUDO DE CASOS

A figura 5.1 apresenta um exemplo conceitual de teste em um SOC utilizando um controlador embutido do tipo MET. O sistema desta figura é formado por duas memórias RAM, uma memória ROM, um microcontrolador 8051, um algoritmo de MPEG e uma UART. A UART é testada através de cadeias *scan*, o bloco MPEG deve ser testado através de teste externo, o microcontrolador e as memórias possuem auto-teste.

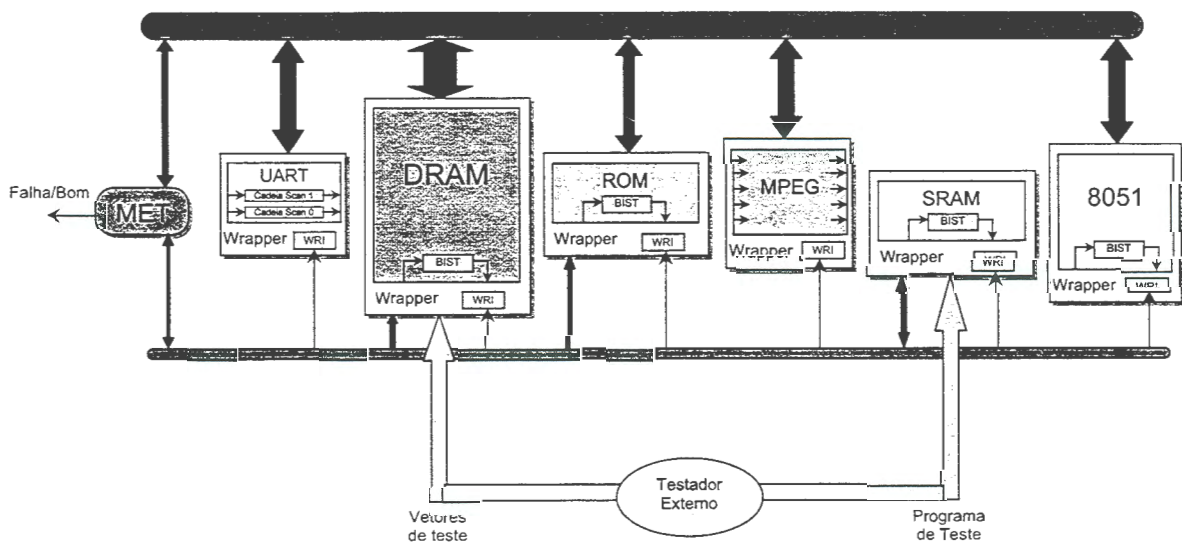


Figura 5.1 - Sistema com controle de teste interno.

Uma vez em modo teste o gerenciamento do processo de teste passa para o testador. A primeira etapa do processo de teste é o auto-teste do controlador que não foi abordado neste trabalho. O programa e os vetores utilizados no auto-teste do controlador podem estar armazenados na memória ROM. Em seguida o controlador configura os *wrappers* das memórias do sistema para que estas sejam testadas. O controlador executa o teste das memórias (preferencialmente as memórias devem apresentar algum tipo de estrutura BIST). Se outros núcleos possuírem estruturas de auto-teste, estas também podem ser disparadas conjuntamente com o teste das memórias, respeitando-se o consumo de potência.

Depois que o teste das memórias tiver terminado, o controlador configura o sistema através dos *wrappers* (e da interface de saída no caso de múltiplos barramentos) e as memórias do sistema serão utilizadas pelo controlador de teste para realizar o teste dos outros núcleos. Os dados para estas re-configurações encontram-se em uma memória ROM. Os *wrappers* podem funcionar como DMA permitindo a leitura e escrita dos dados de teste nas memórias. O programa de teste que foi gerado pelo compilador, a partir de um planejamento do teste e da linguagem STIL/CTL, é carregado nas memórias. O controlador executa o programa configurando os caminhos, os *wrappers*, enviando estímulos de teste e avaliando as respostas. Caso o espaço disponível em memória não seja suficiente para o armazenamento de todos os dados do teste, um novo programa pode ser carregado à medida que o programa de teste for terminando e espaços nas memórias sejam liberados. Instruções de saltos nos endereçamentos das memórias permitem que novos programas e estímulos de teste sejam carregados ao mesmo tempo em que o teste está sendo executado. Outra possibilidade para diminuir a exigência de memória e manter a qualidade do teste é o uso de dados compactados.

Este exemplo mostra a variedade de exigências de testes que um sistema pode exigir de um testador interno. Pensando-se nestas possibilidades, um estudo de casos foi realizado com a finalidade de avaliar o comportamento do controlador de teste dedicado MET. Este estudo, que será apresentado nas seções seguintes deste capítulo, permitiu a implementação de novas funcionalidades no controlador melhorando o seu desempenho. Este estudo também faz uma avaliação do impacto de memórias e conexões quando o processo do teste é controlado de forma interna.

5.1 CONTROLANDO TESTE EXTERNO

Para o estudo do comportamento do controle de teste diante de um teste externo foram adotados circuitos *benchmarks ISCAS'85* [ISC 85] e *ISCAS'89* [ISC 89] como núcleos. Os *benchmarks ISCAS'85* são formados por um conjunto de dez circuitos combinacionais e os *benchmarks ISCAS'89*, por 31 circuitos seqüenciais digitais. Estes circuitos são freqüentemente utilizados por muitos pesquisadores como base para comparar resultados na área de geração, simulação de teste e diagnósticos de falhas.

A opção por este tipo de circuito, neste trabalho, se deu pelo fato destes serem fornecidos com suas respectivas descrições em VHDL [VHD 01] e também com seus vetores de teste para falhas do tipo *stuck-at* [HIT 01].

Os circuitos *benchmarks* utilizados para avaliação do teste externo foram os circuitos seqüenciais denominados de s298 e s344. As descrições VHDL do controlador MET [COT 01] e dos *benchmarks* foram unidas formando sistemas. A tabela 5.1 informa alguns dados sobre os circuitos utilizados.

Tabela 5.1 - Dados sobre os circuitos *benchmarks* s298 e s344.

<i>Benchmark</i>	Nº de Entradas	Nº de Saídas	Nº de flip-flipos	Nº de portas lógicas	Nº de vetores
s298	3	6	14	119	322
s344	9	11	15	160	127

O teste destes *benchmarks* deve ser realizado com a inserção de estímulos nos pinos de entrada e após um ciclo de relógio deve-se fazer a comparação com os resultados fornecidos pelos circuitos. Para este teste, a instrução EXCITA&COMPARA foi utilizada. O uso desta instrução se deve à necessidade de comparação de uma resposta ao mesmo tempo em que se envia um estímulo para a entrada do circuito.

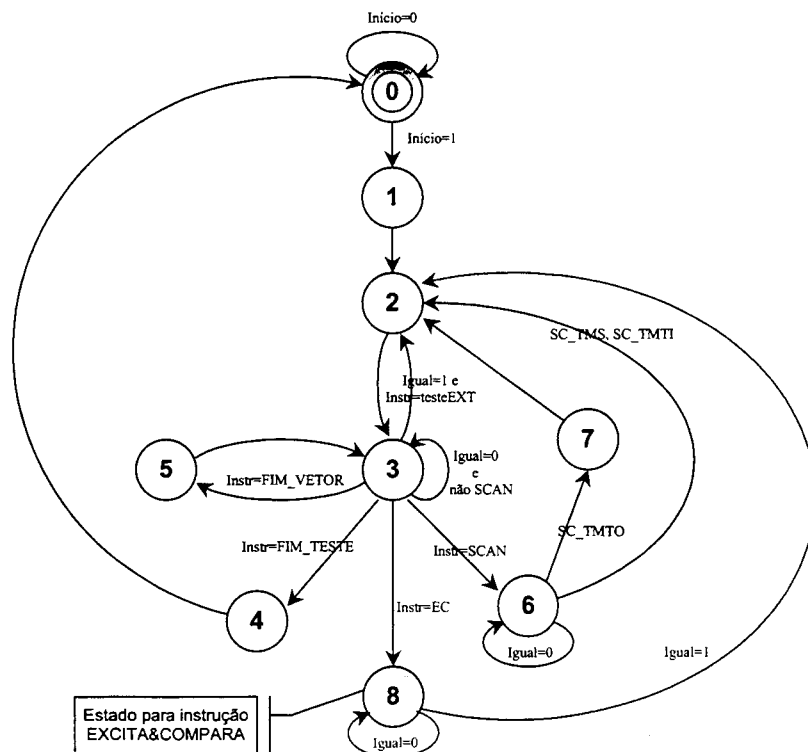


Figura 5.2 - Diagrama de estados do bloco de controle com instrução *at-speed*.

O controlador MET descrito em [COT 01] necessita no mínimo dois ciclos de relógio para realizar a instrução EXCITA&COMPARA. Isto ocorre porque uma instrução é executada no estado 3 da máquina de estados do bloco de controle do controlador (figura 5.2), e o incremento dos apontadores das memórias de programa e de dados acontecem no estado 2. Em outras palavras, a máquina de estados executa uma instrução de EXCITA&COMPARA no estado 3 enviando os dados da memória de dados para as entradas do núcleo e para comparação, dirige-se ao estado 2, incrementa os apontadores das memórias, carrega a nova instrução de EXCITA&COMPARA e os dados de teste, retornando ao estado 3 para executar a instrução.

Devido à necessidade de dois estados para a execução de uma nova instrução e por motivo de sincronização entre o controlador de teste e o núcleo (circuito *benchmark*) é necessário que o MET tenha uma frequência de relógio que seja o dobro da frequência do *benchmark*. Uma vez que o núcleo está trabalhando com uma frequência de relógio menor que a frequência do controlador ocorre um aumento no tempo de teste.

Para resolver este problema, o controlador de teste MET foi remodelado. A principal alteração ocorreu na máquina de controle onde foi inserido um novo estado para executar uma operação na mesma frequência de relógio do núcleo. Toda vez que uma instrução de EXCITA&COMPARA for solicitada, a máquina de controle vai para o estado 8 (figura 5.2) permanecendo neste estado enviando estímulos e comparando respostas até o final do tempo de teste. O tempo de teste indica o número de vezes que a instrução EXCITA&COMPARA deve ser executada. Este valor é inserido na memória de programa conjuntamente com a instrução. Quando o controlador verificar que o número de vezes foi atingido, a máquina de estados retorna para o estado 2 onde faz o incremento do apontador da memória de dados e de programa carregando a nova instrução de teste.

Para o circuito *benchmark* s298, por exemplo, que contém 322 vetores de teste, a instrução EXCITA&COMPARA deve vir acompanhada de um valor que indique que a instrução deve ser executada 322 vezes. O valor que indica o número de vezes que uma instrução EXCITA&COMPARA deve ser executada é calculado em função do passo. Isto é, se o passo for um valor P e o número de vetores for um valor N , então o valor que deve ser armazenado é dado pela divisão de N por P .

Na figura 5.3 mostra-se uma simulação do sistema composto pelo controlador de teste mais o *benchmark* s298. O passo utilizado nesta simulação foi dois, e a quantidade de vetores utilizados foi somente de dez vetores por motivo de visualização. Observe na figura

5.3 a seqüência de teste do estado 1 até o estado 8, onde a instrução EXCITA&COMPARA (0100) é executada, retornando-se para o estado 2, onde incrementa-se os endereçadores das memórias e carrega-se a nova instrução que é executada no estado 3. A nova instrução é de FIM DE TESTE (0000), o que faz com que a máquina de estados vá para o estado 4 e deste para o estado 0.

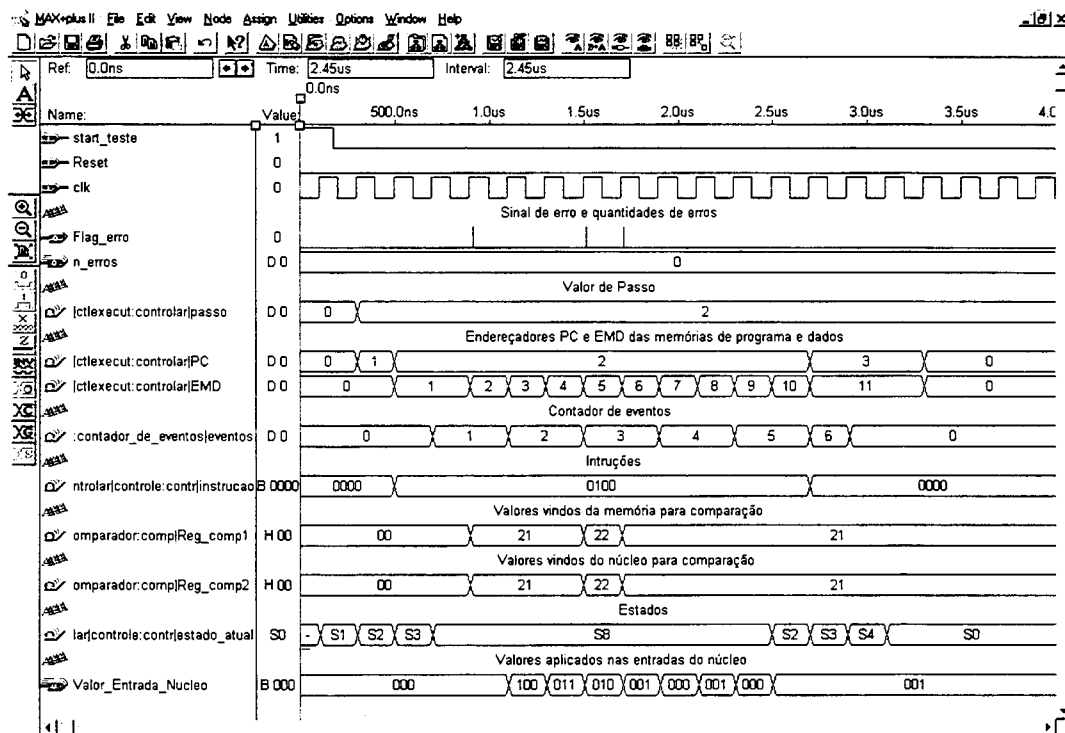


Figura 5.3 - Simulação para um teste externo *at-speed*.

5.2 CONTROLANDO TESTE SCAN

O uso de cadeias *scan* é uma alternativa para aumentar a qualidade do teste. Esta técnica aumenta a cobertura de falhas em relação ao teste externo, ao mesmo tempo em que não necessita de grandes modificações para a inserção de elementos geradores de estímulos e analisadores de assinaturas, demandando, portanto, um acréscimo em área normalmente inferior ao exigido pelas estruturas BIST.

Como descrito no capítulo 4, em sistemas integrados em um único chip, o padrão *boundary-scan* tem sido muito utilizado para o teste da lógica central dos núcleos como, por exemplo, carregar cadeias *scan* internas.

Para o estudo dos mecanismos necessários para um controlador dedicado de teste realizar o controle de testes *scan*, foram utilizados o controlador descrito em [COT 01] e al-

guns *benchmarks* ISCAS. Uma vez que os circuitos *benchmarks* não possuem nenhuma estrutura *boundary-scan*, um controlador TAP teve que ser implementado em VHDL. A implementação da TAP segue o padrão comentado na seção 2.7.

Como explicado na seção 4.2, o controlador MET prevê o uso de instruções *scan*. A primeira dificuldade encontrada para o controle de um teste *scan* foi à ausência de um sinal de TCK. Este sinal é muito importante porque é ele quem vai ser utilizado para o controle da TAP e o deslocamento serial dos vetores de teste nas cadeias *scan*. Como deseja-se que o processo de teste seja o mais independente possível do equipamento externo, considera-se necessário que o controlador de teste seja capaz de gerar este sinal TCK.

Por motivos de sincronização entre o controlador de teste e os núcleos, o sinal TCK não pode ser diretamente conectado ao sinal de relógio de sistema. Como exemplo, pode-se citar a inserção de uma seqüência em TMS fornecida pela instrução SC_TMS, seguida da inserção de valores em TDI pela instrução SC_TMTI. Durante a transição de instruções o MET necessita de três ciclos extras de relógio: um ciclo no estado 2 para o incremento do apontador de programa, um ciclo no estado 3 para o incremento da memória de dados e outro para retornar ao estado 6 e executar a instrução SC_TMTI. Se o sinal de TCK estiver ativo durante este tempo, pode ocorrer uma transição na máquina de estados da TAP fazendo com que seja perdido o sincronismo dos sinais.

Este sinal pode ser facilmente gerado a partir do relógio do sistema e um sinal de controle como mostrado na figura 5.4. O controlador gera o sinal de controle a partir da instrução *scan* e do número de deslocamentos que acompanham esta instrução. Por exemplo, para um sinal de TMS com cinco (5) deslocamentos (SC_TMS < 5 >), o controlador irá enviar para a TAP os valores de TMS e juntamente com estes valores também envia 5 pulsos de TCK.



Figura 5.4 - Esquema utilizado para geração do sinal de TCK.

O estudo também mostrou que a organização das instruções, dos dados e a maneira como os dados são enviados aos núcleos no modelo proposto por [COT 01] demanda uma grande quantidade de espaço em memória.

Para exemplificar o controle da execução de um teste *scan* o circuito *benchmark* s38584.1 foi utilizado como núcleo de hardware. O *benchmark* s38584.1 possui:

- 38 entradas;
- 304 saídas;
- 1426 flip-flops tipo D;
- 110 valores de estímulos;
- 110 valores de respostas.

Este circuito possui uma única cadeia *scan* formada pelos 1426 flip-flops. O tamanho do vetor de estímulo é formado pelo tamanho da cadeia, mais os estímulos de entradas primárias (1464 valores). O tamanho do vetor resposta é dado pelo número de saídas e pelo tamanho da cadeia (1730 valores). A figura 5.5 mostra como ficam organizados os dados e as instruções para o teste *scan* do circuito s38584.1 nas memórias de dados e de programa. Neste exemplo, a largura da memória de programa é de 16 bits e da memória de dados é de 32 bits.

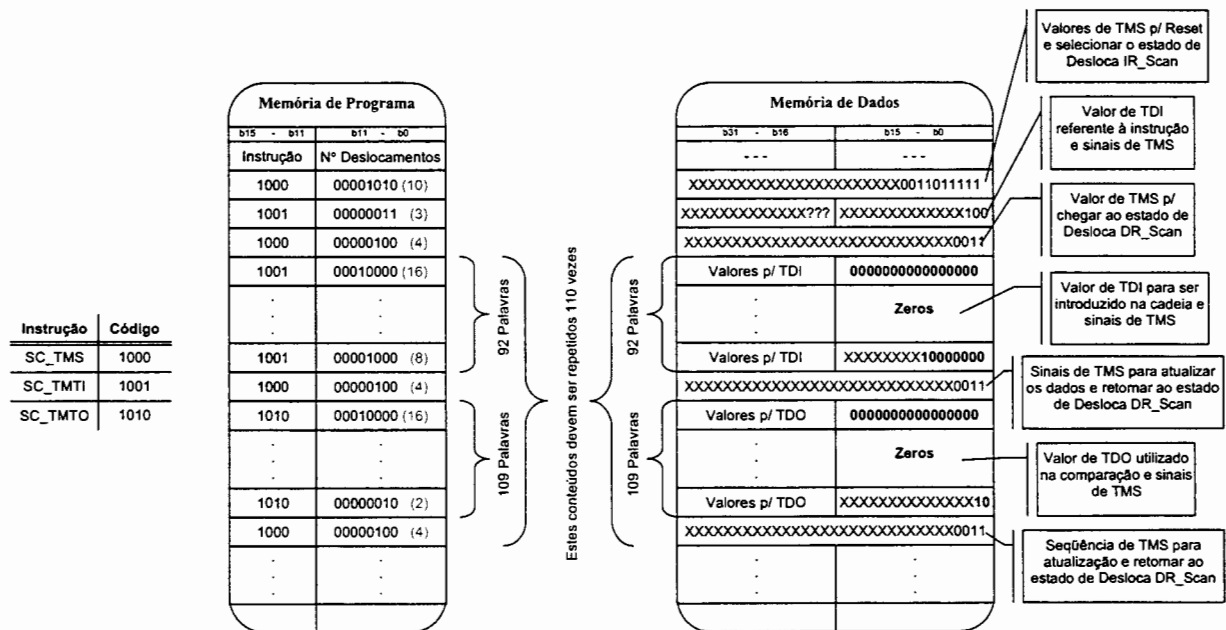


Figura 5.5 - Organização dos dados de teste scan nas memórias do MET.

Em uma instrução SC_TMS, toda a largura da palavra é utilizada para armazenar os valores de TMS. Para as instruções SC_TMTI e SC_TMTO as palavras da memória são

divididas em duas partes, sendo que uma parte é utilizada para armazenar os valores para TDI ou TDO e a outra parte é utilizada para os valores TMS.

Devido à estrutura da arquitetura, cada instrução que está na memória de programa tem relacionada uma posição na memória de dados onde são armazenados os estímulos. Quando for necessária uma seqüência grande de valores, esta seqüência deve ser dividida de tal forma que caiba nas posições de memória, e a instrução deve ser repetida tantas vezes quantas forem necessárias para que toda a seqüência seja deslocada na cadeia. Na figura 5.5 (memória de dados) podemos observar uma quantidade grande de “zeros” armazenados nas posições destinadas aos valores de TMS. Isto ocorre porque durante a inserção serial dos valores na cadeia *scan* o valor de TMS deve permanecer em “zero”, sendo que este muda para “um” somente quando o último valor for introduzido na cadeia. Este valor lógico “um” fará a máquina de estados da TAP sair do estado de deslocamento de dados. Em seguida, uma seqüência de valores é aplicada em TMS para que: os valores que estão na cadeia sejam inseridos na lógica central do núcleo, os valores fornecidos como respostas sejam capturados pela cadeia e após deslocados para fora da cadeia para análise. O processo se repete fazendo com que a máquina de estados da TAP retorne ao estado de deslocamento de dados para inserção dos novos valores de teste, e assim sucessivamente.

Fazendo uma análise do conteúdo das memórias, observam-se valores redundantes que são repetidos muitas vezes durante o processo de teste. Como espaço em memória é sempre algo que deve ser considerado, pois pode ser necessária grande quantidade de vetores para conseguir-se uma alta cobertura de falhas, propõe-se alterações nas memórias para reduzir a quantidade de informações para a execução do teste *scan*

O modelo proposto para esta nova arquitetura é formado por três memórias. Uma memória é utilizada para armazenar as instruções e o número de deslocamentos, outra é utilizada para armazenar instruções da TAP e os valores de TMS e uma terceira memória onde são armazenados os dados a serem deslocados na cadeia ou utilizados como respostas do teste.

Esta terceira memória é chamada de memória Scan e tem como principal característica o deslocamento serial e contínuo de seus valores. A informação sobre a quantidade de deslocamentos é indicada com a instrução. Por exemplo, em uma instrução do tipo *SC_TMTI < 500 > 500* valores são deslocados.

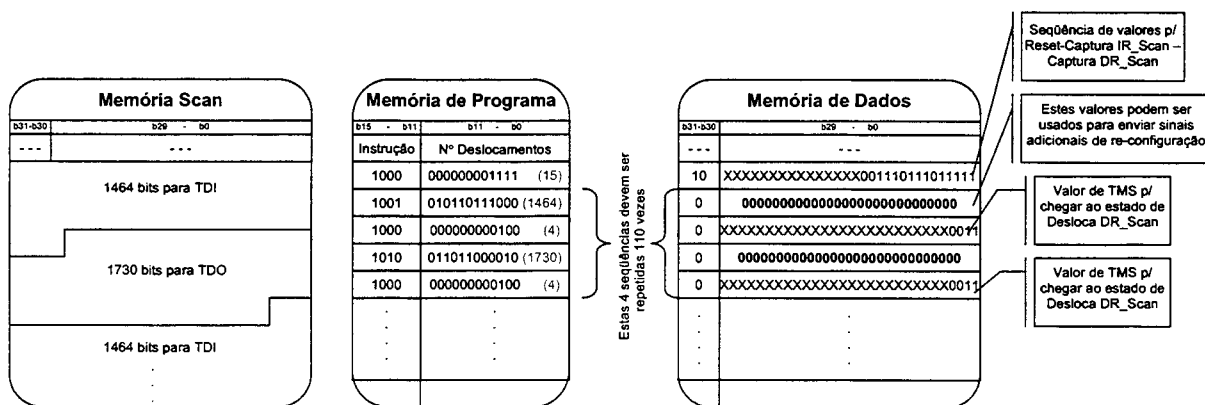


Figura 5.6 - Organização dos dados para um teste scan utilizando três memórias.

A figura 5.6 mostra como ficam as memórias no controlador para o mesmo teste do benchmark s38584.1. A memória de programa tem uma largura de 16 bits e as memórias de dados e Scan têm 32 bits cada uma. Nesta figura pode-se verificar a redução substancial da quantidade de informações armazenadas nas memórias de programa e de dados. Esta redução ocorreu porque uma única instrução é solicitada durante todo o processo de preenchimento ou esvaziamento da cadeia. O controlador gera automaticamente o valor “um” em TMS, quando o último valor for introduzido da cadeia.

Para a redução no tempo de carga de instruções da TAP, a memória de dados foi dividida em duas partes, sendo que uma parte é utilizada para armazenar instruções que vão para a TAP e a outra é utilizada para as seqüências de TMS. A opção por esta divisão vai no sentido de aumentar a dinâmica do teste diminuindo o seu tempo de execução. O padrão IEEE std. 1149.1 prevê a carga serial das suas instruções através de TDI. Porém, algumas pequenas modificações na TAP podem ser realizadas, sem desviar-se da norma, para que a carga destas instruções seja feita em modo paralelo.

Esta forma utilizada na implementação da memória de dados, onde uma parte da palavra é reservada para instruções, foi assim definida pensando-se não somente no uso da TAP, mas principalmente no uso de wrappers. Os wrappers fornecem um maior dinamismo ao teste, provendo acessos paralelos para instruções e dados, permitindo que os caminhos de acesso sejam configurados mais rapidamente e com isso diminuindo o tempo de teste.

Observando a memória de dados da figura 5.6, pode-se verificar que determinadas palavras estão preenchidas com o valor “zero”. Estas posições foram reservadas para geração de sinais de configurações (TAM, wrapper, etc). Estes sinais podem ser muito úteis para o controle dos wrappers onde, por exemplo, este pode ser re-configurado em uma situação de

teste de vários núcleos em paralelo. Uma vez que o teste de um dos núcleos tenha terminado, os caminhos de acesso que este núcleo utilizava podem ser utilizados pelos núcleos que continuam em teste, aumentando o paralelismo no envio dos dados, ou para o teste de um novo núcleo. É por causa desta dinâmica na reconfiguração dos caminhos que a memória de dados foi definida desta maneira.

Como pode-se ver na tabela 5.2, onde foi realizado o teste *scan* de alguns circuitos *benchmarks*, a redução percentual no número de bits para execução de teste pode chegar a mais de 65% com o uso deste método. Esta redução ocorre pela necessidade de uma quantidade menor de palavras para definir um mesmo teste. O número de palavras de memória necessárias para um teste *scan* em [COT 01] é dado pela quantidade de vetores e também pelo tamanho dos vetores, uma vez que este deve ser dividido pela largura da memória disponível. Para o novo modelo proposto a quantidade de palavras das memórias de programa e dados é dada simplesmente pelo número de vetores que devem ser armazenados.

Tabela 5.2 - Comparativo da quantidade de bits de memória para teste *scan*.

Benchmark	Nº de Entradas	Nº de Saídas	Nº de Flip-Flops	Nº de Vetores	Total de bits [COT 01]	Total de bits [novo modelo]	Redução %
s9234.1	36	39	211	210	171.504	72.393	58
s15850.1	77	150	534	188	379.152	139.862	63,12
s13207.1	62	152	638	466	1.073.808	391.954	63,5
s38584.1	38	304	1426	220	1.071.984	372.508	65,25
s38417	28	106	1636	136	701.904	244.712	65,14
s35932	35	320	1728	24	139.536	48.084	65.54

Para a implementação deste modelo, que utiliza três memórias, foram feitas alterações nos blocos de controle e no comparador do modelo proposto em [COT 01]. O tratamento de erros agora não é mais realizado no estado 7 (figura 5.2) mas no estado 6. Neste novo modelo todo o teste *scan* é tratado no estado 6 da máquina de controle (figura 5.7).

As principais modificações ocorreram no bloco comparador que foi totalmente remodelado. Um registrador recebe paralelamente partes dos dados vindos da memória de dados e os envia serialmente para o núcleo (por exemplo, sinais para a TAP). Os bits da memória de dados destinados às instruções (da TAP, do *Wrapper*, etc) são enviados paralelamente

te para os núcleos. Um contador é responsável pelo deslocamento serial dos dados da memória Scan quando uma instrução de SC_TMTI ou SC_TMTO for solicitada.

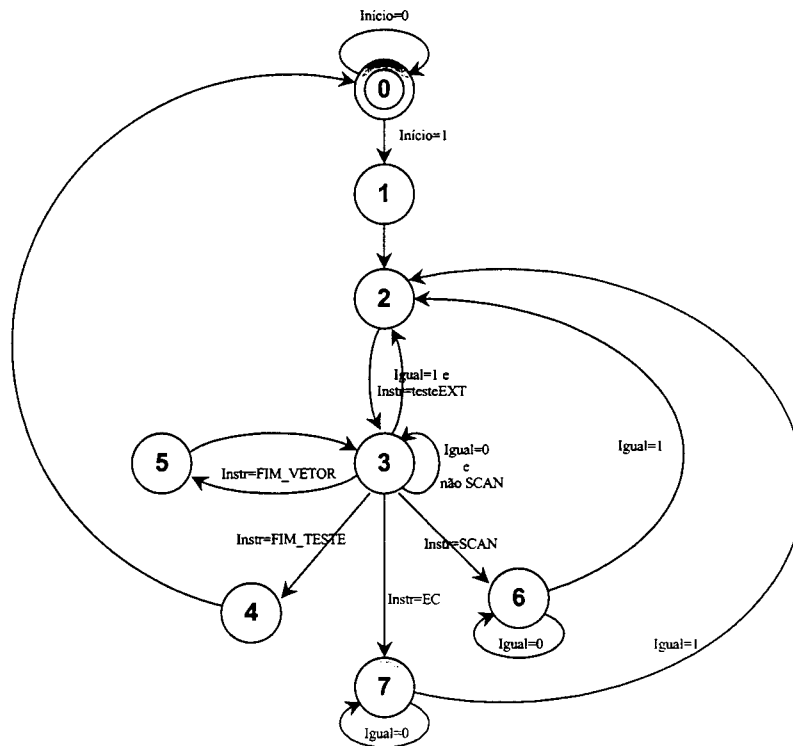


Figura 5.7 - Diagrama de estado

Durante o processo de comparação das respostas fornecidas pelo núcleo, ocorre simultaneamente o deslocamento dos valores que estão na cadeia e dos valores que estão na memória Scan. Estes valores são comparados e a cada vez que um erro é encontrado, o contador de erros é incrementado.

A figura 5.8 mostra uma simulação onde podem ser vistos os diversos sinais que fazem parte do processo de teste. O número de deslocamentos nesta simulação foi somente nove por motivos de visualização. Observe o sinal de TCK diferenciado do sinal do relógio do sistema e o sinal de TMS = '1' no final de uma instrução SC_TMTI ou SC_TMTO. Nesta simulação também podem ser vistos os sinais inseridos na cadeia "011000010", o sinal utilizado como comparação "101001100" e o sinal fornecido pelo núcleo "101011100". Como existe um bit diferente nas seqüências comparadas o microcontrolador incrementa o contador de erros informando que um erro foi encontrado. Modificações no controlador de teste podem

gerenciar o controle de várias memórias Scan no sistema. Assim, cada memória pode, por exemplo, estar associada ao teste de um núcleo.

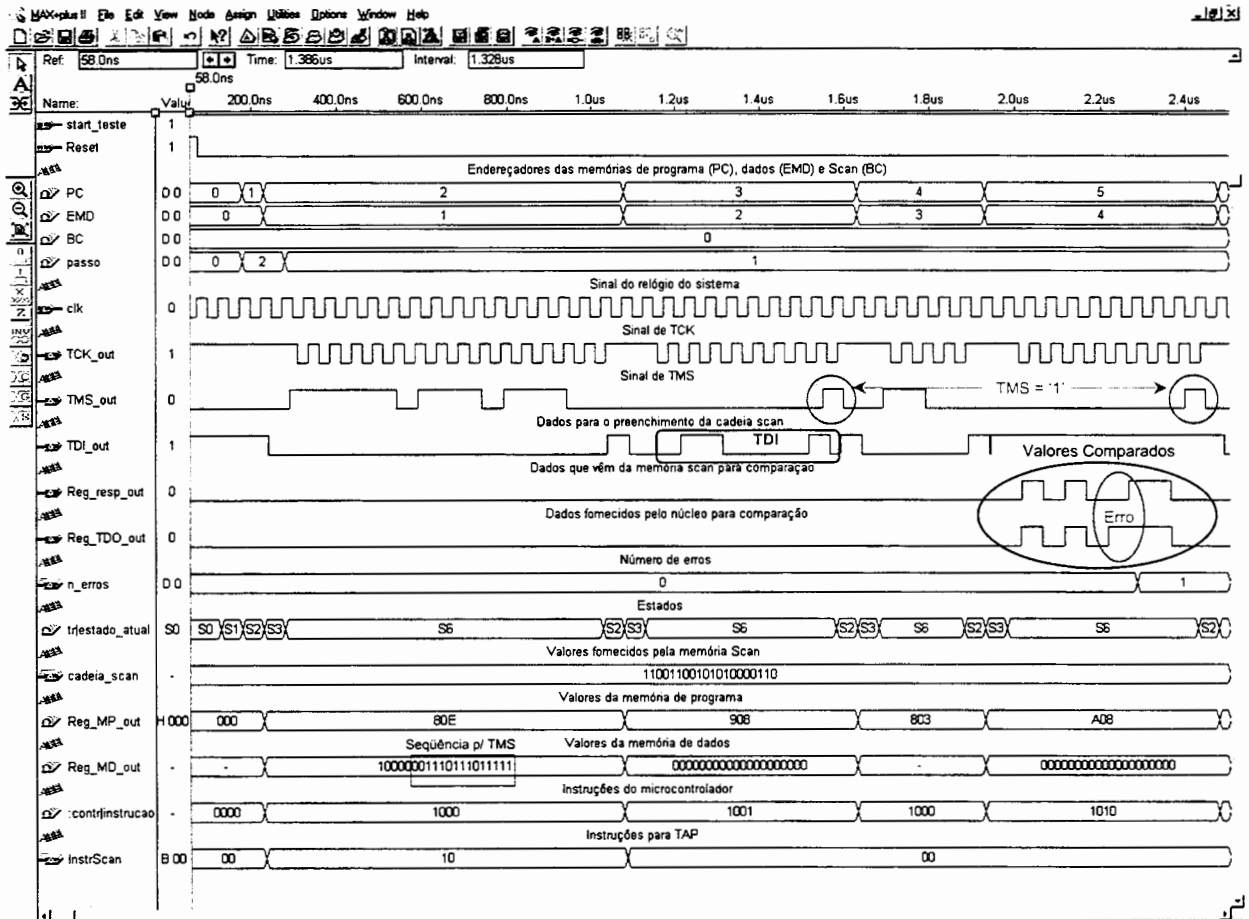


Figura 5.8 - Simulação de um teste *scan* utilizando 3 memórias.

5.3 CONTROLE DO TESTE COM VETORES COMPACTADOS

O volume de dados de teste pode ser muito superior à quantidade de memória disponível no sistema. A compactação tem sido utilizada para diminuir o volume de dados, e muitas pesquisas sobre compactação têm sido apresentadas nos últimos anos [AGA 81], [A-BO 83], [DAN 84], [EDI 92], [DUF 93], [IYE 98], [CHA 01a]. A diminuição no volume de dados reduz a quantidade de memória necessária para o teste e também o tempo de teste. Isto ocorre porque o tempo associado ao teste de um SOC depende da quantidade de dados de teste, do tempo necessário para transferência destes dados de um meio externo até os núcleos e da taxa em que estes dados são transferidos. Como o volume de dados é menor, devido à compactação, o tempo para a transferência destes dados também será menor para uma mesma taxa de transferência. Como a velocidade de operação do SOC tende a ser elevada, a descompactação e aplicação dos dados de teste podem ocorrer rapidamente.

Por motivos de redução de área e também pela grande quantidade de técnicas de compactação utilizadas, considera-se que o controlador de teste não possua a capacidade de executar autonomamente um algoritmo de descompactação.

Devido à diversidade de técnicas de compactação de dados, supõe-se que o hardware que executa a descompactação seja um elemento intrínseco ao núcleo, ou um componente implementado pelo integrador do sistema. Este procedimento foi considerado porque alguns algoritmos apresentam vantagens para alguns núcleos e desvantagens para outros, assim o integrador escolhe o melhor algoritmo de compactação para os núcleos que vão compor o seu sistema. Não cabe ao controlador de teste descompactar os dados, mas sim, ter a capacidade de comunicar-se com os elementos responsáveis pela descompactação, fornecendo os sinais e os dados a este processo.

O estudo aqui realizado utilizou os trabalhos de pesquisa sobre compactação e descompactação de dados de teste para SOC apresentados por [CHA 00], [CHA 01], [CHA 01a]. Estes trabalhos são baseados no algoritmo Golomb [GOL 66] para compactação. A escolha do método ocorreu por este ser um método proposto para SOCs e pela grande quantidade de material encontrado, o que facilitou a sua implementação.

5.3.1 Codificação Golomb e Compactação

O algoritmo Golomb consiste basicamente na substituição de uma seqüência grande de valores zero ('0') por uma seqüência menor que representa uma palavra código. Quanto maior a seqüência de zeros, maior será a taxa de compactação obtida.

O primeiro passo para codificar um conjunto de vetores T_D é gerar um vetor diferença T_{diff} . Se $T_D = \{t_1, t_2, t_3, \dots, t_n\}$ o vetor diferença pode ser determinado por $T_{diff} = \{t_1, t_1 \oplus t_2, t_2 \oplus t_3, \dots, t_{n-1} \oplus t_n\}$, o algoritmo de compactação será aplicado no vetor diferença.

Uma vez que existe uma relação estrutural entre as falhas de um circuito, os vetores de teste possuem uma grande correlação entre si. Quando mais relacionados forem estes vetores, menor será a quantidade de '1s' no vetor diferença. Também, os vetores podem ser ordenados de tal forma que favoreça a correlação no momento em que o vetor diferença é gerado. A quantidade de '1s' está intimamente ligada à taxa de compactação, quanto menor for a quantidade deste valor maior será a taxa de compactação.

O próximo passo é escolher o parâmetro m de codificação Golomb que determina o tamanho do grupo. Uma tabela é formada a partir deste parâmetro para formar as palavras

código que irão substituir as seqüências de valores. O parâmetro m pode determinar uma maior ou menor taxa de compactação e este pode ser determinado experimentalmente ou analiticamente [CHA 01].

Uma vez que o valor de m foi determinado, a quantidade de zeros em uma seqüência de T_{diff} é mapeada para o “tamanho do grupo” (cada grupo correspondendo a um “tamanho”). O conjunto de “tamanhos” $\{0, 1, 2, \dots, m-1\}$ forma o grupo A_1 ; o conjunto $\{m, m+1, m+2, \dots, 2m-1\}$, o grupo A_2 ; e assim por diante. Em geral, o conjunto de “tamanhos” $\{(k-1)m, (k-1)m+1, (k-1)m+2, \dots, km-1\}$ consiste de A_k grupos. Para cada grupo A_k é nomeado um grupo de prefixos de $(k-1)$ valores ‘1’ seguido por ‘0’. Este procedimento recebe a notação $1^{(k-1)}0$. Se m for escolhido como sendo uma potência de 2, por exemplo $m = 2^N$, cada grupo contém 2^N membros e uma seqüência de $\log_2 m$ bits (corpo) identifica cada membro dentro do grupo. Então, o código final é formado por duas partes, o grupo prefixo ($1^{(k-1)}0$) e o corpo (seqüência de $\log_2 m$ bits). Um processo de codificação para $m = 4$ é ilustrado na tabela 5.3.

Tabela 5.3 - Exemplo de codificação Golomb para $m = 4$.

Grupo	Tamanho	Prefixo	Corpo	Palavra Código
A_1	0	0	00	000
	1		01	001
	2		10	010
	3		11	011
A_2	4	10	00	1000
	5		01	1001
	6		10	1010
	7		11	1011
A_3	8	110	00	11000
	9		01	11001
	10		10	11010
	11		11	11011
...

O procedimento de compactação consiste em substituir uma seqüência de zeros (‘0s’) seguida do valor ‘1’ por uma palavra código. Por exemplo, no vetor diferença da figura 5.9, a primeira seqüência encontrada é “0001” que possui três zeros ($l_1 = 3$). Este valor será o tamanho do grupo. Consultando a tabela 5.3, observamos que este tamanho corresponde à palavra código “011”, logo este código irá substituir a seqüência “0001”. Repetindo este procedimento para todas as seqüências encontradas, no final do processo teremos um vetor compactado T_E que representa o vetor T_{diff} . A compactação para o exemplo resulta em um vetor com 32 bits a partir de um vetor de 42 bits.

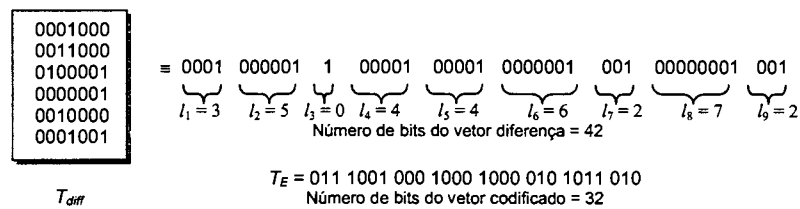


Figura 5.9 - T_{diff} e o vetor codificado T_E .

Um programa em MatLab® foi elaborado para realizar o procedimento de compactação dos vetores de teste. O primeiro passo para gerar um vetor compactado é fazer o reordenamento dos vetores de tal maneira que a melhor correlação seja encontrada. O melhor ordenamento pode ser um problema NP-completo.

A rotina de re-ordenamento primeiro encontra o vetor que contém o menor número de ‘1s’ nomeando este vetor de t_1 . Após, verifica em todos os vetores qual deles tem a melhor correlação com t_1 , nomeando este de t_2 . O processo repete-se agora buscando o vetor que melhor se relaciona com t_2 e assim sucessivamente até que todos os vetores sejam calculados. Caso os vetores de teste possuam *don't cares* estes podem ser manipulados para aumentar a correlação dos vetores. O passo seguinte é a construção do vetor diferença $T_{diff} = \{t_1, t_1 \oplus t_2, t_2 \oplus t_3, \dots, t_{n-1} \oplus t_n\}$, que é feito aplicando-se a operação de “XOR” entre dois vetores consecutivos.

O operador informa o parâmetro m desejado para que o programa construa as palavras código a partir deste parâmetro. Uma vez calculado o vetor diferença e as palavras código, encontram-se as seqüências de ‘0s’ seguidas do valor ‘1’ em t_{diff} . Para cada seqüência, calcula-se o número de zeros e a partir desta informação ocorre a substituição da seqüência pela palavra código.

5.3.2 Descompactação

Para a descompactação um decodificador foi implementado em VHDL e é formado por uma máquina de estados finita (FSM) e um contador de $\log_2 m$ bits. Um digrama em blocos do decodificador utilizado para descompactação pode ser visto na figura 5.10.

O sinal *bit_ent* é a entrada da máquina de estados finita e o sinal *en* serve para indicar que o decodificador pode receber os valores de entrada. O sinal *inc* é utilizado para incrementar o contador, *rs* indica que a contagem terminou. O sinal *saída* é o valor decodificado

e o sinal *valida* indica que a saída é válida. Os sinais *en* e *valida* são utilizados para sincronizar as operações de entrada e saída do decodificador.

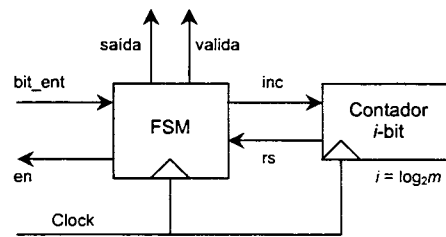


Figura 5.10 - Diagrama em blocos do decodificador usado para descompactação.

O decodificador funciona da seguinte maneira:

- Sempre que a entrada for '1', o contador conta até m . O sinal *en* é negado enquanto o contador estiver contando e após habilitado para que a entrada receba outro bit após o final da contagem. O decodificador gera m '0s' durante esta operação mantendo a saída válida (sinal *valida* ativo).
- Quando a entrada for '0' a FSM inicializa decodificando o corpo da palavra código.

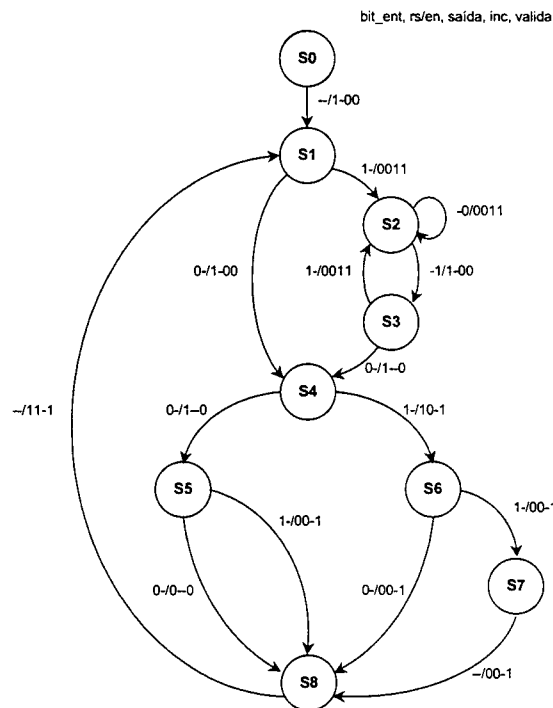


Figura 5.11 - Diagrama de estados de um decodificador Golomb com $m = 4$.

O diagrama de estados correspondente a um decodificador para $m = 4$ é mostrado na figura 5.11. Os estados $S0$ a $S3$ e $S4$ a $S8$ correspondem à decodificação do prefixo e do corpo, respectivamente.

Os valores decodificados correspondem ao vetor diferença. Para obtenção dos valores de teste descompactados um registrador de deslocamento cíclico (CSR, do inglês, *Cyclical Scan Register*) [JAS 98] ou cadeias *scan* internas podem ser utilizadas conforme mostrado nas figuras 5.12 e 5.13, respectivamente.

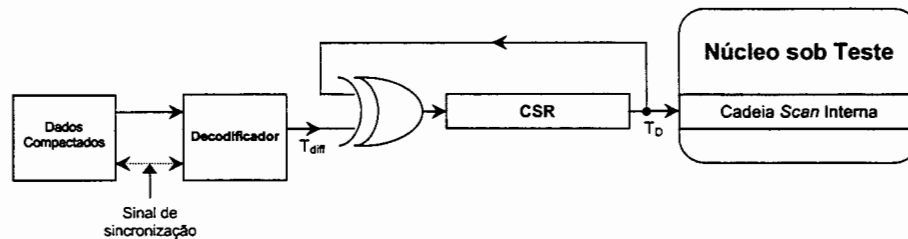


Figura 5.12 - Arquitetura de descompactação baseada em CSR.

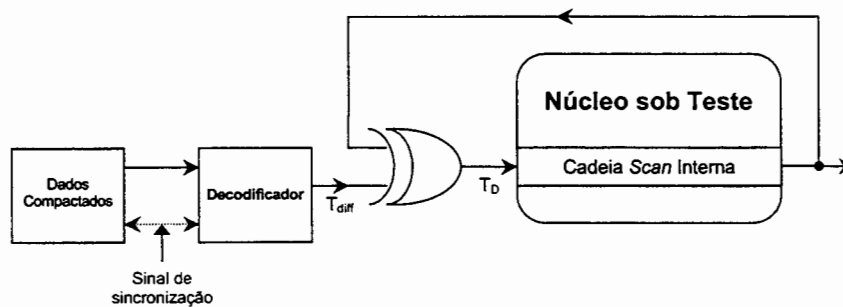


Figura 5.13 - Arquitetura de descompactação baseada no uso de cadeia scan.

No caso de ser utilizada a própria cadeia *scan* para a realimentação, como mostra a figura 5.13, os valores de resposta livres de falha devem ser utilizados no processo de compactação. Um MISR pode ser utilizado para compactar a resposta fornecida pelo núcleo e gerar uma assinatura que será avaliada pelo controlador de teste.

O uso da cadeia *scan* para realizar a realimentação do vetor diferença oferece a vantagem de uma assinatura ser oferecida no final do processo sem a necessidade de um MISR. Uma vez gerado um valor errado por algum elemento da cadeia, este irá se propagar durante todo o teste produzindo valores incorretos quando a última resposta do teste for deslocada para fora da cadeia. A desvantagem no uso das cadeias para realimentação está na ne-

cessidade de mais espaço em memória e também pelo maior tempo de teste. Isto ocorre pela maior quantidade de dados utilizados (estímulos e respostas), ao invés de se utilizar apenas os vetores estímulos e uma assinatura para comparação, como no caso de uma arquitetura de descompactação baseada em CSR.

Em [JAS 98] podem ser encontradas várias maneiras de como formar CSRs a partir de células *boundary-scan*, através de outras cadeias *scan*, ou mesmo com o uso integrado de células *boundary-scan* e cadeias *scan*.

Como explicado anteriormente, a estrutura responsável pela decodificação pode estar incluída no núcleo, ou pode ser implementada pelo integrador do sistema. O controlador de teste seria utilizado para enviar os dados para o decodificador e após comparar os dados vindos dos núcleos. Para o envio dos dados para o decodificador, as instruções *scan* da arquitetura proposta por [COT 01] foram avaliadas.

Observou-se a necessidade de sincronismo entre o controlador e o decodificador. O controlador estudado tem apenas a capacidade de enviar dados para os núcleos em um determinado tempo e após fazer a análise das respostas através de comparação. Não foi previsto que o controlador entre em um estado de espera a partir de um sinal fornecido por um núcleo, ou algum elemento do tipo descompactador.

Este sincronismo pode ser feito através da temporização das instruções. Para que isto seja feito é necessário que o compilador, que gera o programa de teste, conheça perfeitamente o processo de descompactação e os tempos necessários para o decodificador realizar o processamento dos seus sinais. Com o uso da temporização das instruções, se um bit for enviado fora do tempo previsto, todo o processo de teste estará comprometido. Outra desvantagem da temporização é a grande quantidade de instruções exigidas por um processo de descompactação. Ora, se estamos utilizando compactação para reduzir o volume de dados, não é uma boa estratégia o uso de grandes quantidades de instruções somente para saber em que momento um bit deve ser enviado.

Este problema pode ser simplesmente resolvido com a colocação de um sinal que paralise a contagem do contador de eventos e do mecanismo responsável pelo deslocamento/incremento dos dados em uma memória Scan, conforme descrito na seção 5.2. A figura 5.14 mostra uma simulação onde o processo de teste foi momentaneamente paralisado a partir de um estímulo externo.

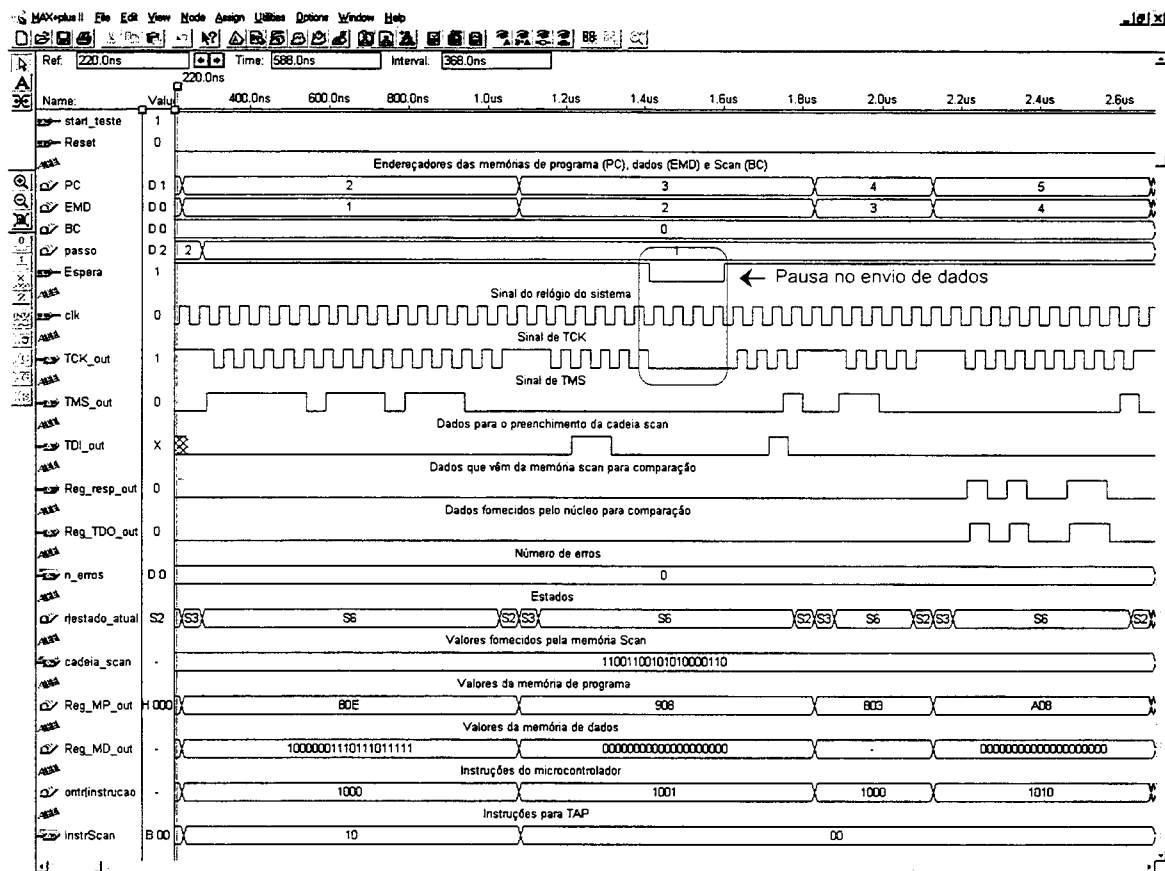


Figura 5.14 - Simulação com sinal de espera no processo de teste.

A tabela 5.4 mostra um comparativo entre vetores compactados utilizando codificação Golomb para $m = 4$ com o uso do controlador MET. Td representa o tamanho do vetor sem compactação e Te o tamanho do vetor compactado. Na codificação Golomb o tempo necessário para a descompactação está ligado ao valor de m . Em geral, a relação entre a frequência interna e a frequência externa, para equiparação nos tempos de aplicação dos vetores de teste, é dada por $f_{ext} = f_{int}/m$ [CHA 01a]. Isto mostra um importante benefício no uso de controladores internos, isto porque, geralmente, a frequência de relógio de um sistema integrado é superior a do testador. Assim, a partir de um volume reduzido de dados e um testador externo lento pode-se realizar o teste do sistema no mesmo tempo que um testador rápido.

A tabela 4.4 (página 52) apresenta um comparativo de frequências entre o controlador MET e alguns outros circuitos, comprovando que o testador pode facilmente trabalhar com dados compactados. A pior relação apresentada pela tabela 4.4 (MET e Filtro LMS) mostra que a frequência do MET é cerca de três vezes superior à frequência de operação do filtro, ou seja, em um sistema formado pelo MET e o Filtro LMS, e um teste utilizando dados com

compactação Golomb com $m = 4$, a relação de equiparação nos tempos de teste apresentadas na tabela 5.4 são facilmente alcançáveis.

Tabela 5.4 - Comparação entre frequência interna e frequência externa para descompactação Golomb.

Tamanho de Td (bits)	Tamanho de Te (bits)	% de Compactação	Ciclos para Descompactação	$f_{\text{interna}} / f_{\text{externa}}$
42	32	23%	64	1,54
3922	661	83%	7020	1,79
34720	16983	51%	66315	1,91
39273	22250	43%	73833	1,88

A metodologia de teste utilizando vetores compactados é semelhante à metodologia apresentada na figura 5.15. Os vetores de teste compactados são armazenados na memória Scan e o controlador de teste gerencia o envio dos dados para as cadeias *scan* dos núcleos.

Uma forma de teste com vários núcleos em paralelo é o uso de várias memórias Scan. [CHA 01a] apresenta um modelo onde dados compactados pelo algoritmo Golomb podem ser interlaçados de forma que várias cadeias *scan* podem ser carregadas simultaneamente.

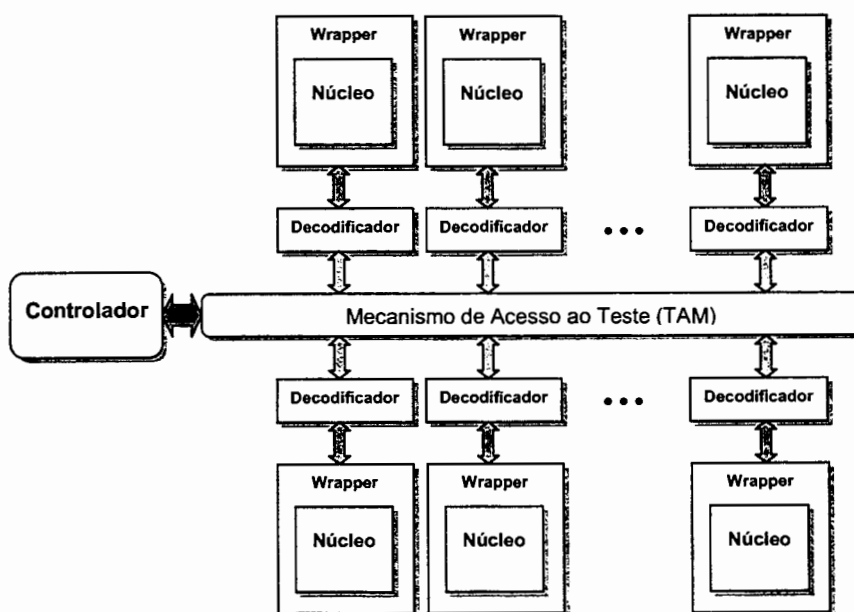


Figura 5.15 - Arquitetura para o teste de vários núcleos a partir de uma memória Scan com vetores compactados.

5.4 TESTE *SCAN* PARALELO DE NÚCLEOS.

Os estudos apresentados até aqui foram realizados considerando-se o teste de um núcleo isolado. Porém o objetivo é explorar o máximo paralelismo a fim de diminuir o tempo total do teste do sistema. Para o teste em paralelo duas possibilidades foram estudadas. A primeira delas utiliza um único módulo de memória de dados para o envio simultâneo destes para as cadeias scan. A figura 5.16 mostra um exemplo onde um módulo de memória com uma largura de 32 bits (uma memória ou várias memórias compostas) é utilizado para um teste em paralelo de três núcleos que possuem diversas cadeias *scan*.

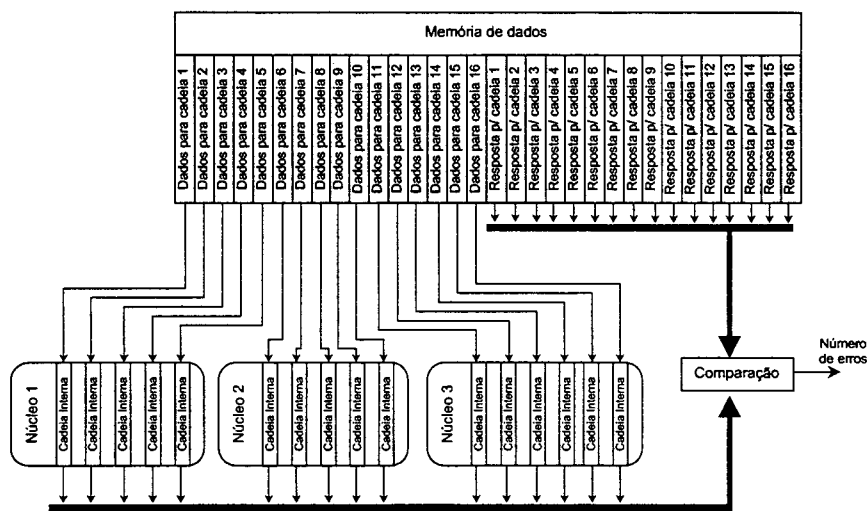


Figura 5.16 - Carga paralela de várias cadeias com o uso de um módulo de memória.

Nesta estratégia, todas as cadeias envolvidas no teste devem possuir o mesmo tamanho. Caso isso não seja possível, valores sem significado para o teste devem ser acrescentados para torná-las do mesmo tamanho. Este procedimento é necessário, pois os dados serão deslocados nas cadeias através do incremento do apontador da memória de dados que é único.

Parte da memória é utilizada para o armazenamento dos dados de envio e parte é utilizada para os dados de comparação como mostrado na figura 5.16. Na inserção dos valores nas cadeias, a instrução *EXCITA&COMPARA* é utilizada. A vantagem do uso desta instrução é que as cadeias podem ser carregadas ao mesmo tempo em que as respostas são analisadas. Uma discrepância grande entre os tamanhos das cadeias pode ser um problema, porque o acréscimo de valores pode tornar os vetores de teste maiores que a quantidade de memória disponível para o seu armazenamento. Caso exista um MISR para realizar a compactação das

respostas fornecidas pelas cadeias, toda a largura da memória pode ser utilizada para realizar o deslocamento dos estímulos de teste. A instrução COMPARA será utilizada para realizar a comparação entre um valor armazenado na memória e pela resposta fornecida pelo MISR.

Sabe-se que, em geral, SOCs apresentam um grande número de memórias. Estas memórias podem ser utilizadas para formar um único módulo conforme apresentado acima, ou podem ser usadas individualmente como memórias Scan. Cada memória Scan deve estar associada a um núcleo. Também, neste método, os tamanhos das cadeias devem ser aproximadamente o mesmo porque o controle do deslocamento de todas as memórias é único. Este método pode ser útil quando o sistema é formado por núcleos que possuam uma única cadeia *scan* e o sistema tem uma alta frequência de operação. Modificações no controlador podem fazer com que ele gerencie as memórias para núcleos com cadeias de diferentes tamanhos.

5.5 IMPACTO DO TAMANHO DA MEMÓRIA NO TESTE

O SOC apresentado na figura 5.17 é um sistema arbitrário que tem por finalidade mostrar o impacto das memórias e das conexões no processo de controle interno do teste. O sistema é composto por sete núcleos formados por *benchmarks ISCAS'85 e ISCAS'89*. As conexões internas que ligam os núcleos foram determinadas aleatoriamente. As informações sobre o número de entradas/saídas, número de cadeias scan internas (linhas tracejadas) e também a quantidade mínima de ciclos de teste para cada núcleo são mostradas na figura 5.17. As informações referentes aos ciclos de teste foram calculadas assumindo acesso exclusivo às cadeias scan e aos pinos de entrada/saída. Os dados utilizados para o cálculo foram obtidos a partir de [HAM 98] e [HIT 01].

O cálculo do número de ciclos de teste de um *benchmark* é realizado segundo a equação 5.1. Nesta equação, P_i representa o número total de vetores envolvidos no teste e L representa o tamanho de cada cadeia. A tabela 5.5 mostra os valores de P_i , L e os tempos de teste para os *benchmarks*, conforme a equação 5.1.

A expressão 5.1 foi calculada assumindo-se conexões exclusivas para cada núcleo. Ao mesmo tempo em que um vetor está sendo carregado na cadeia, a resposta é deslocada para fora. Logo, o tempo total do teste é dado pelo tempo necessário para o preenchimento da cadeia (L) vezes (P_i+1) padrões. Como os vetores também incluem os valores das entradas e saídas primárias, e assumindo-se que o tempo de carga desses valores é de apenas um ciclo

de relógio, deve-se acrescentar à expressão a quantidade de padrões (P_i). Se o núcleo utilizar somente o teste externo, como é o caso dos circuitos combinacionais, a variável L deve ser zero.

$$N_c = (P_i + 1) \cdot L + P_i \tag{5.1}$$

Tabela 5.5 - Dados referentes ao processo de teste dos *benchmarks* do sistema exemplo da figura 5.16.

Benchmark	Nº. de E/S	Nº. de cadeias (C)	Tamanho total da cadeia	Tamanho de cada cadeia (L)	Nº. de vetores (P _i)	Nº. de ciclos
s838	35	1	32	32	75	2507
s5378	84	4	179	45	97	4507
c6288	64	-	-	-	12	12
c1908	58	-	-	-	106	106
s9234	41	4	211	53	105	5723
s15850.1	101	16	534	38	95	3359
s38417	134	32	1636	52	68	3656

Os seguintes experimentos foram realizados utilizando o sistema da figura 5.17:

1. Teste paralelo de todos os núcleos com acesso exclusivo por um ATE.
2. Teste serial dos núcleos com diferentes larguras de memória utilizando um controlador interno.
3. Teste explorando paralelismo com diferentes larguras de memória e utilizando um controlador interno.

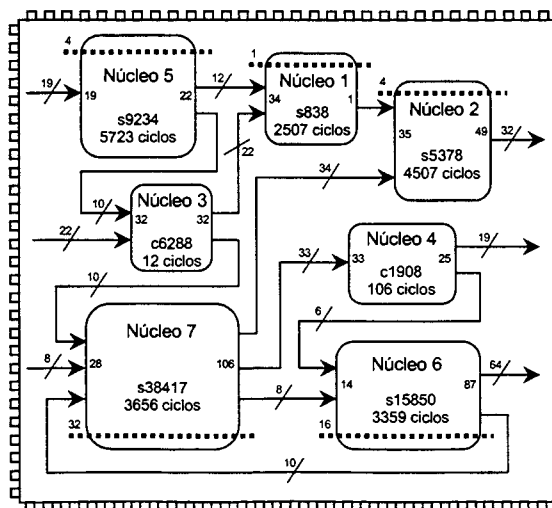


Figura 5.17 - Sistema utilizado como exemplo.

Assumindo que o sistema avaliado não possui memórias e que um ATE está conectado por um acesso exclusivo a todos os núcleos do sistema, os testes de todos os módulos serão realizados em paralelo.

Considerando as entradas/saídas e acessos para as cadeias *scan* internas de cada núcleo, seriam necessários 454 pinos extra no *chip* para realizar a comunicação entre os núcleos e o ATE. Este valor representa o somatório de todos os pinos de E/S e de acesso às cadeias para cada núcleo, descontando-se o uso dos pinos funcionais do *chip*. Além do custo associado a estes pinos extra, comentados na seção 3.1, também seria necessário um *buffer* de grande capacidade para enviar todos os estímulos necessários e analisar as respostas fornecidas. Em contra-partida, seriam necessários apenas 5723 ciclos de relógio para o término do teste total do sistema. Este tempo corresponde ao tempo de teste do núcleo mais lento. A tabela 5.6 mostra os valores de tempo, conexões e pinos extra para o teste descrito. O número de ciclos calculados nesta tabela foi obtido pela expressão 5.1. O número de pinos extras representa a diferença entre os pinos funcionais do *chip* e os pinos que devem ser acrescentados para realizar o teste considerando-se as conexões exclusivas. A quantidade de conexões internas extras representa as ligações que devem ser implementadas entre os pinos de E/S do *chip* e os núcleos.

Este exemplo visa mostrar um dos principais problemas do teste de sistemas formados por núcleos de hardware. No momento em que os circuitos migraram da superfície de uma placa de circuito impresso para o interior de uma pastilha de silício, o acesso direto para o teste destes circuitos foi reduzido, gerando uma série de dificuldades.

Tabela 5.6 - Ciclos e hardware extra para uma conexão exclusiva com um ATE.

Núcleo	Ciclos	Conexões Internas Extra	Pinos Extra
1	2507	37	37
2	4507	60	60
3	12	42	42
4	106	39	39
5	5723	30	30
6	3359	56	56
7	3656	190	190
Total →	5723	454	454

Realizando esta mesma análise para a arquitetura descrita em [COT 01] seria necessária uma memória interna com uma largura de palavra de 631 bits, e conexões desta memória com os módulos. O valor 631 representa o somatório de todos os pinos funcionais e de acesso às cadeias *scan* de todos os núcleos do sistema que agora não precisam mais ser ligados a pinos do *chip*, mas devem possuir uma conexão até a memória do MET. Esta estratégia pode levar a uma grande quantidade de hardware extra, aumentando significativamente a área em silício.

No caso do sistema possuir uma ou mais memórias, estas podem ser utilizadas para armazenar os padrões de teste. É possível realizar o teste serial de cada núcleo ou, caso exista espaço disponível em memória, este espaço pode ser utilizado para armazenar dados de outros núcleos e realizar o teste de alguns módulos com paralelismo. Caso o sistema possua várias memórias, estas podem ser configuradas para formar uma única memória com uma largura de palavra maior. As figuras 5.19 e 5.20 mostram os resultados da análise feita para o teste de módulos com diferentes tamanhos de memórias. Esta análise mostra o teste sem paralelismo e explorando o paralelismo sempre que existe espaço disponível em memória.

Para a análise do impacto, os dados de teste foram arranjados na memória de forma a minimizar o tempo de teste. Um exemplo pode ser visto na figura 5.18, onde é mostrado como foram organizados os dados para o circuito *benchmark* s838 em uma memória com uma largura de palavra de oito bits. Como este *benchmark* possui uma cadeia *scan* de 32 bits, este é o fator determinante no tempo de teste, pois a cadeia não pode ser dividida, ou seja, são necessários no mínimo 32 ciclos para o preenchimento da cadeia. Este circuito possui 34 entradas que foram divididas na memória em dois vetores de 17 bits cada um. Neste exemplo são necessários 5 conexões entre o núcleo e a memória e sobram 3 posições de memória que não são utilizadas para o teste individual do núcleo s838.

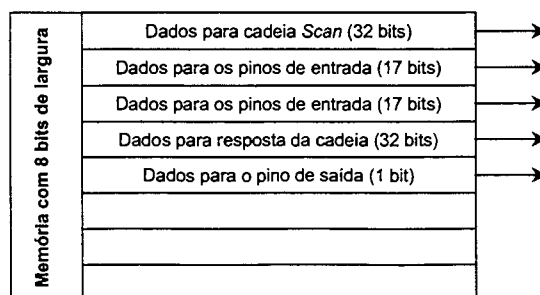


Figura 5.18 - Organização dos dados na memória para o teste do circuito s838.

No caso dos circuitos *benchmark* que possuem muitas cadeias (mais cadeias que o tamanho da palavra), os vetores são ordenados seqüencialmente. Um exemplo é o circuito s15850.1 que possui 16 cadeias, 14 entradas e 87 saídas. Neste circuito os dados de quatro cadeias são colocados na primeira posição da palavra de memória, outros quatro vetores são colocados na posição dois e assim sucessivamente até a colocação de todos os vetores (estímulos e respostas) na memória. Os dados para os pinos de entrada e saída são colocados após os vetores das cadeias.

Assim, os circuitos que possuem mais cadeias que o tamanho da palavra de memória tem o tempo de teste calculado pela expressão 5.2, onde P_i representa o número de vetores, L o tamanho da cadeia, C o número de cadeias, M a largura da palavra de memória e S a quantidade de entradas e saídas. O valor da divisão entre S e M deve ser o maior inteiro.

$$N_c = ((P_i + 1) \cdot L + P_i) \cdot \left(\frac{2 \cdot C}{M} \right) + (P_i + 1) \cdot \left(\frac{S}{M} \right) \quad (5.2)$$

O esquema descrito acima foi utilizado para a análise de todos os circuitos do sistema. Na figura 5.19 mostra-se a variação da quantidade de ciclos necessários para o teste em função do tamanho da palavra de memória, e a figura 5.20 mostra a variação do número de conexões internas. Para o cálculo das conexões foi considerado um barramento ligando a memória com os núcleos.

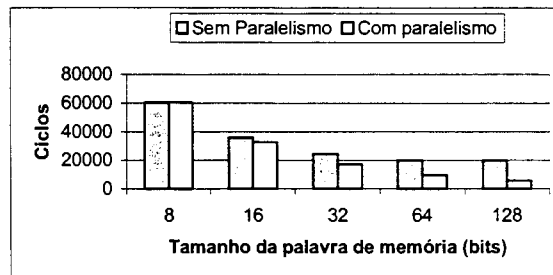


Figura 5.19 - Variação do tempo de teste em função da largura de palavra da memória.

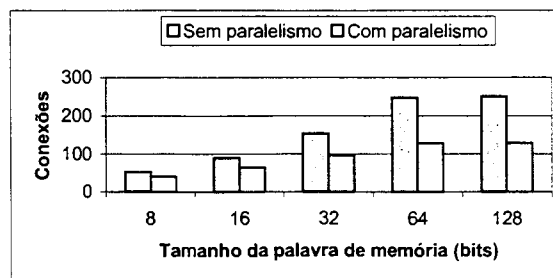


Figura 5.20 - Variação na quantidade de conexões em função da largura da memória.

A largura da memória é muito importante quando um controlador de teste embutido é utilizado. A figura 5.19 mostra que, quanto maior a largura da memória, menores são os tempos de teste. O paralelismo pode ser muito bem explorado de acordo com a largura da palavra e pode contribuir para um menor tempo de teste. Observa-se na figura 5.19 que em uma memória de 64 bits o tempo de teste em paralelo decai praticamente em 50% do tempo de teste sem paralelismo. Dobrando o tamanho da memória para 128 bits o tempo de teste em paralelo fica em torno de 25% de um teste onde o paralelismo não é explorado.

A quantidade de conexões necessárias para o teste está relacionada com o tamanho da memória e a quantidade de entradas de sinais que os núcleos possuem para o teste. Observa-se na figura 5.20 que um teste onde os núcleos são testados de forma individual exige mais conexões do que o teste em paralelo de todos os núcleos. Isto ocorre pela maneira como os dados foram organizados nas memórias. No teste individual, cada núcleo exige uma conexão exclusiva com a memória. Estas conexões sempre são formadas no sentido de se explorar o máximo paralelismo dos estímulos e captação de respostas. Por exemplo, em um teste individual de um núcleo que possui três entradas, três saídas e uma cadeia *scan*, serão utilizados três fios para entrada, três fios para a saída e dois fios para a cadeia (um para inserção de valores e um para verificação das respostas).

Na figura 5.18 mostra-se a memória e as conexões para o teste do circuito s838. Nota-se nesta figura que as três posições da palavra que não foram usadas podem armazenar os dados de teste de um outro núcleo, como por exemplo, o circuito c6288. Este *benchmark* possui 32 entradas e 32 saídas, e tem um tempo de teste de 12 ciclos, que é o tempo necessário para a inserção e comparação dos 12 vetores de teste. No teste deste circuito toda a memória é utilizada fazendo com que oito conexões exclusivas sejam necessárias para o teste individual. Mesmo utilizando-se as oito conexões não será necessário atender o tempo de apenas 12 ciclos, isto porque é necessário o envio de 32 valores mais a comparação de 32 outros valores em um único ciclo de relógio. O tempo de teste, segundo a equação 5.1, é de 2507 ciclos de relógio, tempo este mais que suficiente para testar-se o circuito c6288 utilizando-se apenas as três posições da memória que estavam vazias.

No exemplo, o tempo de teste do circuito c6288 aumenta em relação àquele feito de maneira individual, porém o tempo de teste de dois núcleos que era dado pelo somatório dos tempos individuais agora diminui em virtude do paralelismo. O número de conexões também diminui em função de serem utilizados apenas três conexões ao invés de oito.

A figura 5.20 mostra que a taxa de crescimento do número de conexões nos modos individual e paralelo diminui à medida que a largura da memória aumenta, até que se atinja um limite onde o número de conexões não varia mais. Este limite está relacionado com o núcleo que possui o maior tempo de teste. Se todos os vetores deste núcleo cabem na memória, atende-se seu tempo de teste e ainda resta espaço para a colocação dos vetores de outros núcleos, de forma a poder testá-los em paralelo.

Da análise do gráfico da figura 5.20, obtém-se informações aparentemente contraditórias. Como um teste em paralelo exige menos conexões que um teste sem paralelismo? Tudo gira em torno de como os dados são organizados nas memórias do controlador interno e das exigências do teste (tempo, conexões, memórias, etc.). Neste exemplo buscou-se sempre o menor tempo de teste, independente do número de conexões necessárias. É claro que dados organizados de outras formas têm outro tipo de impacto na quantidade de conexões. Essa é uma decisão que o integrador do sistema deve tomar com base nas estruturas de teste disponíveis (memórias, TAM, etc) e nas suas exigências de teste (tempo, área, etc).

Alguns autores têm sugerido técnicas para gerar um plano de teste [COT 02], [CHA 00a], [IYE 01], [LAR 01]. Em [COT 02] é proposto um modelo de planejamento de teste em um ambiente SOC. O planejamento é gerado por um algoritmo a partir do modelamento do sistema, considerando diferentes mecanismos de acesso utilizados para o teste. Ele gera um plano com máximo paralelismo e insere o menor número de pinos extra a partir da especificação do Engenheiro de Sistema. O plano gerado apresenta o menor tempo de teste dentre as diversas alternativas. Na figura 5.21 mostra-se o planejamento de teste gerado pela ferramenta descrita em [COT 02] para o sistema exemplo da figura 5.17. Este planejamento foi gerado com o incremento extra de 94 pinos no chip, para fazer os acessos aos núcleos. A parte inicial do plano, vista na figura 5.21, mostra que os núcleos 1, 6, 2, 5 e 7 podem ser testados em paralelo. Devido à limitação dos acessos aos núcleos, existem casos onde é preciso terminar o teste de um módulo antes de começar outro. Este é o caso dos núcleos 4 e 3 mostrados na figura 5.21.

Supondo que o plano de teste mostrado na figura 5.21 tenha que ser administrado pelo controlador MET, observa-se que seria necessária uma largura de memória grande para atender o paralelismo e os tempos de teste exigidos. A exigência de uma memória com largura grande deve-se ao fato dos padrões de teste para estes núcleos serem muitos para atender grandes quantidades de cadeias scan e de entradas/saídas. Neste sistema a presença do controlador interno diminui o número de pinos extra no *chip*, considerando que os vetores de teste

seriam buscados pelo controlador no testador externo e aplicados ao núcleo. Em contrapartida, existiria um acréscimo de área devido à presença do MET e de conexões internas e também um incremento no tempo de teste caso a largura da memória não seja atendida.

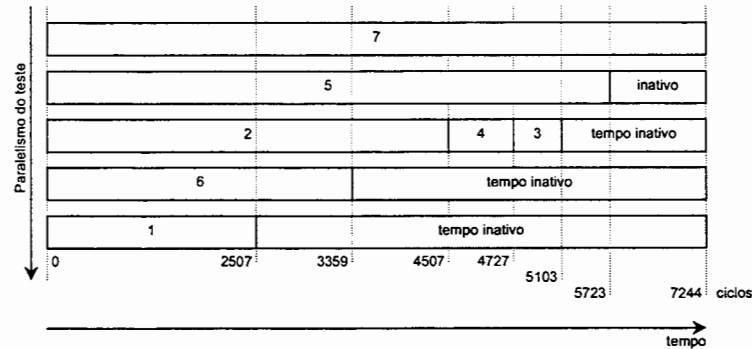


Figura 5.21 - Planejamento de tempo de teste atendendo restrições de projeto.

O planejamento acima especificado é um plano gerado para um sistema que não possui memórias internas e que utiliza equipamento externo para controle do teste. Este planejamento foi impropriamente gerado para o uso de controladores de teste embutidos. Observe que é fundamental, no processo de geração de planejamento de teste, que a ferramenta considere além dos mecanismos de acesso ao teste, a presença do testador como um módulo interno e também a quantidade de memória embutida associada ao processo de teste. Também é importante a existência de uma análise das memórias e conexões, semelhante ao apresentado. Somente assim o integrador do sistema terá uma idéia real dos tempos de teste e do hardware extra (conexões) que ele deve implementar.

5.6 CONTROLANDO TESTE BIST NO 8051

Para exemplificar o controle do teste BIST, utilizou-se o microcontrolador auto-testável 8051 [COT 99] como SOC. Este microcontrolador está dividido nos seguintes blocos: parte operativa, de controle, de validação, geradora de estados, RAM e ROM. O diagrama apresentado na figura 5.22(a) mostra os blocos que compõem o microcontrolador 8051. Para o estudo, o microcontrolador 8051 foi considerado como um SOC sendo que os blocos que compõem o sistema foram avaliados como núcleos de hardware.

Cada bloco que compõe a microcontrolador 8051 auto-testável possui o tipo de auto-teste que é mais adequado à sua função. Na parte de geração de estados, o auto-teste

consiste em uma técnica de verificação de paridade em uma máquina de estados [SHE 94]. Esta técnica é capaz de detectar falhas de transições de estados (SST, do inglês, *Single-State Transitions*) durante o funcionamento normal do sistema, ou seja, uma falha existe se, na transição de estado, o estado destino é indesejado ou incorreto. Este é um teste que não necessita de um controlador, pois trata-se de teste em funcionamento.

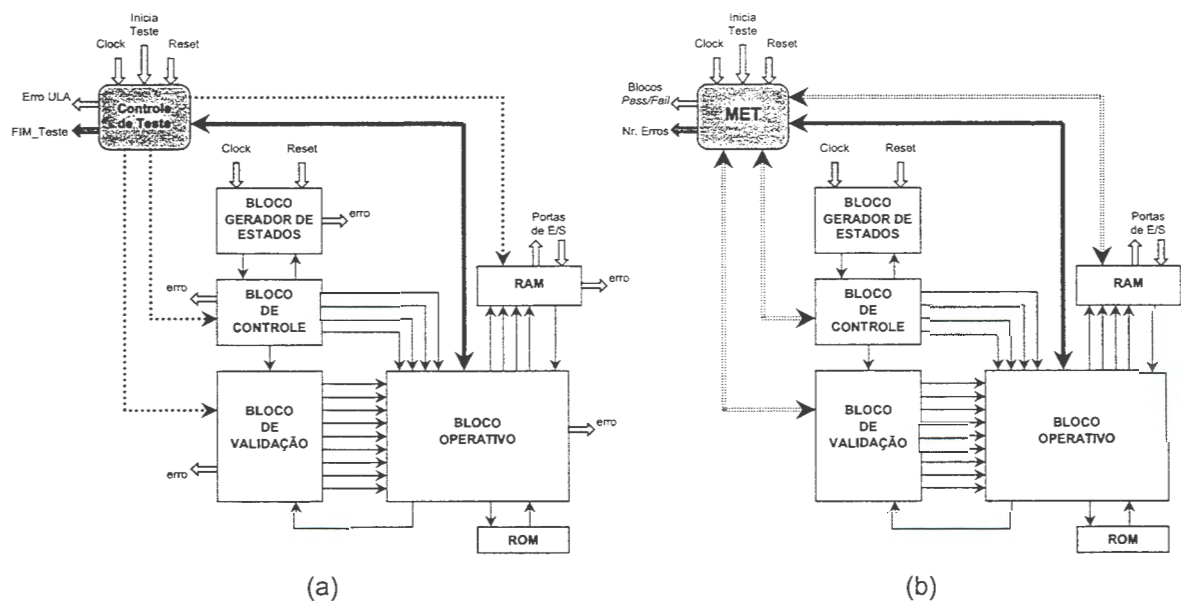


Figura 5.22 - (a) diagrama em blocos da descrição BIST 8051. (b) diagrama em blocos da descrição BIST 8051 com o controlador MET.

Para a memória RAM, o auto-teste é baseado em um algoritmo *March*, que consiste basicamente no preenchimento da memória com o valor '0' (zero), na leitura de cada bit substituindo-se seu valor por '1', finalizando-se com uma re-leitura.

O auto-teste da ULA consiste em uma técnica de BIST circular [STR 98] que utiliza os próprios operadores existentes para geração de vetores de teste pseudo-aleatórios e compactação da resposta.

Os blocos de controle e validação são compostos, basicamente, de estruturas que se comportam como multiplexadores. Estes dois blocos são auto-testados da mesma forma, utilizando a técnica de teste para estruturas de seleção [REN 98] com algumas modificações.

Com a finalidade de avaliar o funcionamento de um controlador específico para o teste, o controlador de teste da descrição BIST original do 8051 foi substituído pelo controlador de teste descrito em [COT 01], conforme apresentado na figura 5.22(b). A função do con-

trolador de teste no 8051 auto-testável é gerar os sinais para habilitar o auto-teste nos blocos de controle, validação, ULA e RAM. Os blocos de controle, validação e a RAM necessitam somente de sinais de habilita, após o término do teste estes blocos informam o status de erro. O bloco operativo, além de um sinal de habilita, necessita da comparação do valor retornado por ele.

Um controlador dedicado ao teste deve habilitar o BIST de todos os blocos, comparar o valor fornecido pela ULA com um valor previamente armazenado em sua memória de dados, verificar o status de erro de cada bloco, informando se algum bloco possui erros, e finalmente indicar o número de erros encontrados.

A primeira análise feita foi em relação ao modo como é feita a habilitação dos sinais de auto-teste dos núcleos. O controlador de teste descrito em [COT 01] prevê dois bits da palavra da memória de dados, por núcleo, reservados para gerar sinais de BIST. Na hipótese de um sistema integrado possuir muitos núcleos com funções BIST, uma grande quantidade da memória de dados pode ser exigida. Levando-se em conta que estes bits são utilizados somente em momentos específicos (durante o auto-teste do núcleo a que se destinam) e que no restante do tempo de teste estes bits não são utilizados, uma grande quantidade de memória será desperdiçada. Em alguns casos, a largura da memória do sistema pode não ser suficientemente grande para atender o teste de todos os núcleos, uma vez que nesta memória também são armazenados os estímulos e respostas para outros tipos de teste.

Para solucionar este problema foram inseridas novas instruções no MET. Uma destas instruções serve para disparar o auto-teste em todos os núcleos ao mesmo tempo, e as outras instruções servem para disparar o auto-teste e verificar o sinal de erro de maneira individual para cada componente virtual. Em termos de hardware extra, essa modificação consiste na inserção de alguns novos registradores e multiplexadores.

Dois modelos de controlador foram criados. Um modelo mantém dois bits reservados por palavra de memória para gerar sinais de habilitação e verificação de status de erro. Neste modelo, a mesma instrução pode ser utilizada para habilitar o BIST e verificar o status. No outro modelo, inseriu-se instruções diferenciadas para habilitação e verificação.

Como comentado no capítulo 4, assumiu-se neste trabalho que núcleos de hardware com funções BIST são aqueles núcleos com capacidade de realizar o teste autonomamente a partir de um simples sinal de habilita e sem a necessidade de qualquer outro estímulo externo. Pode-se imaginar que seria dispensável o uso de outras funções que habilitem o auto-teste

de cada núcleo individualmente, uma vez que poderíamos habilitar as funções BIST de todos os núcleos ao mesmo tempo e ficar aguardando o término do teste. Porém, formas de habilitar individualmente funções BIST são de grande importância, porque algumas estruturas de teste BIST podem dissipar altas quantidades de potência durante o teste. Caso vários destes núcleos, com alta dissipação de potência, tenham suas funções BIST habitadas ao mesmo tempo, existe a possibilidade do teste causar danos irreparáveis ao sistema.

A tabela 5.7 mostra a quantidade de ciclos de relógio necessários para o auto-teste de cada bloco que compõe o microcontrolador 8051 auto-testável. Os valores desta são utilizados para calcular o valor do passo que será usado pelo controlador de teste.

Como explicado na seção 4.3, o valor do passo deve ser o máximo divisor comum entre os tempos de teste. Neste caso, o menor tempo de teste é o valor que se refere ao bloco operativo, ou seja, cinco ciclos de relógio. Logo, este valor será o passo do controlador.

Tabela 5.7 - Tempo de teste para cada bloco do microcontrolador 8051.

Bloco	Ciclos
Operativo (ULA)	5
Controle e Validação	6600
RAM	25905

Em uma situação onde exista um limite físico na largura da memória de programa e que o espaço disponível para armazenar os valores dos tempos de teste seja apenas de oito bits, por exemplo, o maior valor de tempo possível de ser armazenado nesta memória será de 255 passos. Para um passo de apenas cinco ciclos, o máximo tempo de teste, em ciclos de relógio, será de 1275 ($5 \cdot 255$) ciclos, valor este insuficiente para o teste de todos os núcleos (blocos) em paralelo (25905 ciclos). Em uma memória que disponha apenas de oito bits para a temporização, será necessário um passo com 102 ($25905/255$) ciclos para ser realizado o teste de todos blocos do microcontrolador 8051 auto-testável, ao mesmo tempo.

Diante do fato descrito criam-se duas situações: A primeira delas utiliza o menor valor do tempo (cinco ciclos) como o passo do teste. O problema no uso deste passo está em correr-se o risco de algum tempo de teste não poder ser representado no tamanho de palavra disponível. Na outra situação, pode-se utilizar um passo maior e verificar o status de erro algum tempo depois que este foi fornecido. Aqui corre-se o risco de obter-se uma informação

errada como, por exemplo, no caso do bloco operativo do 8051 que fornece um valor que deve ser comparado em um tempo específico, nem antes, nem depois.

Uma solução simples para resolver este problema seria, primeiro, realizar o teste da ULA, que consome somente cinco ciclos, e após reconfigurar o sistema com novo valor de passo e tempos de teste para testar os blocos restantes. Porém, considerando-se que a dissipação de potência durante o teste permita realizar o teste de todos os núcleos em paralelo, que o tempo mínimo de teste não seja apenas de cinco ciclos, mas de milhares de ciclos e que o tempo máximo seja centenas ou até mesmo, milhares de vezes este valor, esta solução parece não ser a melhor possível. Também deve-se levar em conta que a reconfiguração de novos valores de teste causa perdas no tempo de teste encarecendo o custo final deste. Essa reconfiguração consiste em gerar um novo programa pelo compilador e carregar este novo programa nas memórias do controlador de teste.

A solução proposta para este problema é o uso de um passo variável. Agora, o passo não é mais um único valor carregado no principio do teste e que permanece até o final, mas sim, um valor que pode ser atualizado durante o teste.

A alteração necessária para a implementação do passo variável consiste na criação de uma nova instrução chamada de “Valor de Passo”, e também na inserção de um novo laço na máquina de estados do bloco de controle descrito em [COT 01] (figura 5.23). Toda vez que a máquina de controle estiver no estado 3, e uma instrução de Valor de Passo for solicitada, a máquina volta para o estado 1 onde ocorre a gravação do novo valor de passo. O novo Valor do Passo fica armazenado na posição seguinte à instrução de Valor de Passo, na memória de programa.

Tabela 5.8 - Codificação das instruções.

Instruções	Código
FIM_TESTE (FT)	0000
FIM_VETOR (FV)	0001
COMPARA (C)	0010
EXCITA (E)	0011
EXCITA&COMPARA (EC)	0100
EXCITA_NUCLEO1	0101
EXCITA_NUCLEO2	0110
EXCITA_NUCLEO3	0111
EXCITA_TODOS	1100
SC_TMS (SS)	1000
SC_TMTI (SI)	1001
SC_TMTO (SO)	1010
SC_TCK	1011
VALOR_PASSO (VP)	1111

A tabela 5.8 mostra a codificação de quatro novas instruções para habilitar testes do tipo BIST. Três instruções são utilizadas para o controle BIST de cada núcleo e uma é utilizada para o controle simultâneo de três núcleos.

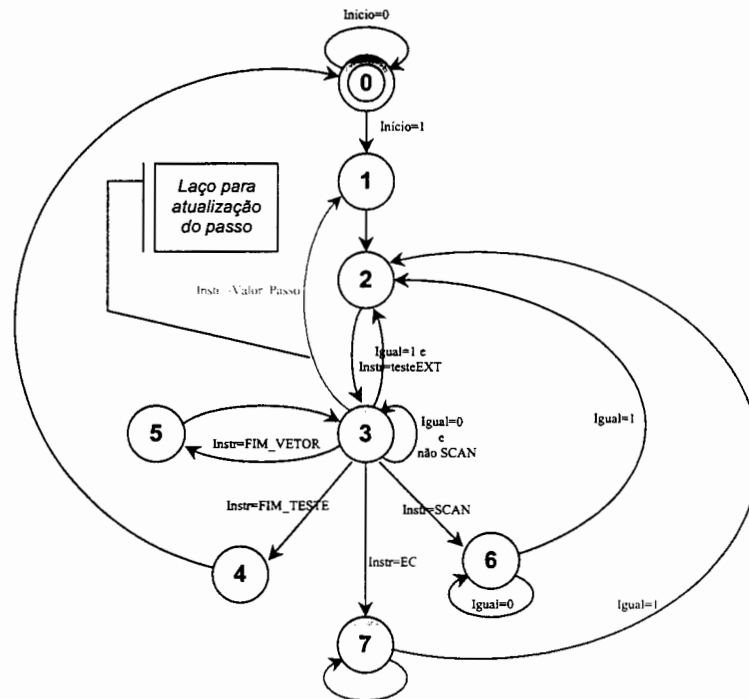


Figura 5.23 - Diagrama de estados do bloco de controle com passo variável.

5.7 WRAPPERS PARAMETRIZADOS

O padrão P1500 não define uma estrutura de controle para os *wrappers*, como no caso da norma IEEE std. 1149.1, onde existe um controlador de TAP que é utilizado para configurar, simultaneamente, a lógica periférica de todos os integrados do sistema através de quatro pinos básicos (TDI, TDO, TMS e TCK). Cabe ao integrador do sistema a melhor decisão de como fazer o controle dos *wrappers*.

[LAZ 02] propõe um modelo para geração de descrições em VHDL de *wrappers* parametrizados. O controle dos *wrappers* gerados deve ser feito de forma individual através da aplicação direta dos sinais. Os sinais utilizados para configurar o *wrapper* são:

- **WRCK**: sinal de relógio do *wrapper*. Trata-se de um sinal dedicado usado pelo registrador de instruções do *wrapper*.
- **WRSTN**: o sinal de inicialização do *wrapper*.
- **WSI**: entrada serial do registrador de instruções.

- **WSO**: saída serial do registrador de instruções.
- **SelecionaWRI**: determina que alguma operação deve ser executada no Registrador de Instruções.
- **CapturaWRI**: faz com que uma instrução seja capturada pela porta paralela.
- **DeslocaWRI**: habilita a entrada serial pela porta serial do *wrapper* e executa uma operação de deslocamento serial no registrador de instruções.
- **AtualizaWRI**: a instrução armazenada no registrador é decodificada e os novos sinais de controle são passados para todos os componentes do *wrapper* que dependem do registrador de instruções.

A figura 5.24 ilustra a arquitetura do Registrador de Instruções do *wrapper* com seus respectivos sinais.

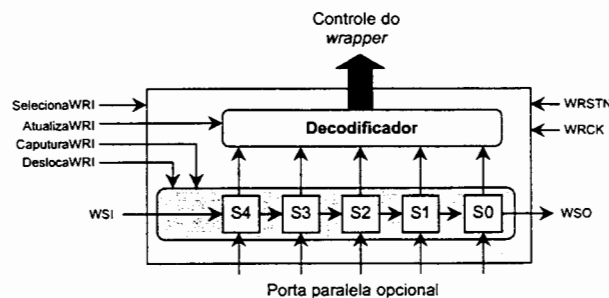


Figura 5.24 - Arquitetura do registrador de instruções do *wrapper*.

[LAZ 02] realiza uma avaliação do controlador descrito em [COT 01] para realizar o envio dos sinais de controle e as instruções dos *wrappers*. O objetivo do controlador de teste em [LAZ 02] é apenas enviar os sinais de controle e as instruções que configuram as operações de todos os *wrappers* simultaneamente. Os vetores de teste devem ser inseridos por um testador externo. O objetivo é fazer uma análise do impacto da área ocupada, do tamanho do barramento e do tempo de configuração para modelos que utilizam acesso exclusivo aos registradores de instruções dos *wrappers*, acessos paralelos por meio de barramentos e acesso serial. [LAZ 02] parametrizou *wrappers* para os núcleos do sistema exemplo mostrado na figura 5.17 e a partir de um planejamento de teste.

As conclusões apresentadas em [LAZ 02] e resumidas na tabela 5.9 demonstram que um modelo com conexões dedicadas tem o melhor desempenho em relação ao tempo de

configuração dos *wrappers*, mas possui uma área em hardware superior aos outros métodos. No modelo serial a área em hardware é menor, porém o tempo de teste é superior (oito vezes em relação ao modelo dedicado). Os outros modelos estudados apresentam valores intermediários aos métodos dedicado e serial. Este estudo é importante, pois apresenta mais uma vez a complexidade do teste em SOCs. Apesar do modelo serial necessitar de oito vezes a quantidade de ciclos em relação ao modelo dedicado, este modo de teste não deve ser desprezado. Esta informação é importante, porém outras informações foram negligenciadas, como por exemplo, quantas vezes o processo de configuração dos *wrappers* será necessário durante o teste, ou a frequência de relógio em que o teste do sistema será executado. Se o controlador de teste for suficientemente rápido a quantidade de configurações for reduzida, o modelo serial pode ser utilizado no teste com o mínimo de tempo. O integrador do sistema deve tomar uma série de decisões importantes que irão influenciar no desempenho e no custo de todo o teste.

Tabela 5.9 - Comparação entre modelos de configurações de *wrappers* [LAZ 02].

Modelo de Sistema	Células Lógicas	Memória Total (bits)	Barramento (bits)	Palavra de Memória (bits)	Flip-flops Adicionais	Ciclos de Relógio
Dedicado	2410	16384	66	52	0	6
1° Paralelo	2402	7424	46	17	0	13
2° Paralelo	2471	7424	23	17	60	21
Serial	2397	5120	4	8	24	54

*Dispositivo EPF10K130EBC600-1 (Altera @ FLEX10KE)

O trabalho de [LAZ 02] enfatiza a liberdade do P1500 em relação ao controle dos *wrappers*. Esta liberdade pode gerar um gama grande de possibilidades de mecanismos de acesso ao teste. Como o enfoque principal desta dissertação é o controle do processo de teste e não o mecanismo de acesso ao teste, o que se tentou fazer foi uma análise do impacto da memória e do acesso para os estímulos de teste (apresentados na seção 5.5), e das diferentes possibilidades de acesso para configurações da lógica envoltória que possibilitem que o mesmo controlador possa se aplicar independente do mecanismo utilizado.

Uma situação de teste inserida nesta seção foi à utilização de barramentos distintos para estímulos de teste e instruções para a lógica envoltória. Para resolver este problema de múltiplos barramentos, sugere-se a colocação de uma interface para comunicação entre os registradores de saída do MET e os caminhos de teste. Esta interface é responsável pela sele-

ção dos caminhos de teste e é composta basicamente por multiplexadores que são controlados por instruções geradas pelo controlador de teste.

5.8 OPERAÇÕES DE SALTOS NOS ENDEREÇADORES DAS MEMÓRIAS

Instruções que permitem a repetição de seqüências (saltos) no programa e/ou nos vetores de teste foram implementadas no controlador MET. Estas instruções também podem ser utilizadas para executar a carga de um novo programa ou de dados de teste ao mesmo tempo em que o teste está sendo executado. As instruções de saltos foram implementadas na memória de programa, na memória de dados e na memória scan do MET.

As alterações para a inserção destas instruções consistem na inserção de três novos estados na máquina de controle (estados 8, 9 e 10). Toda vez que uma instrução de salto for solicitada, o endereçador da memória apontará para a posição indicada pelo valor que encontra-se ao lado da instrução, conforme mostra a figura 5.25.

Instrução	Endereço da memória
-----------	---------------------

Figura 5.25 - Formato das instruções de salto.

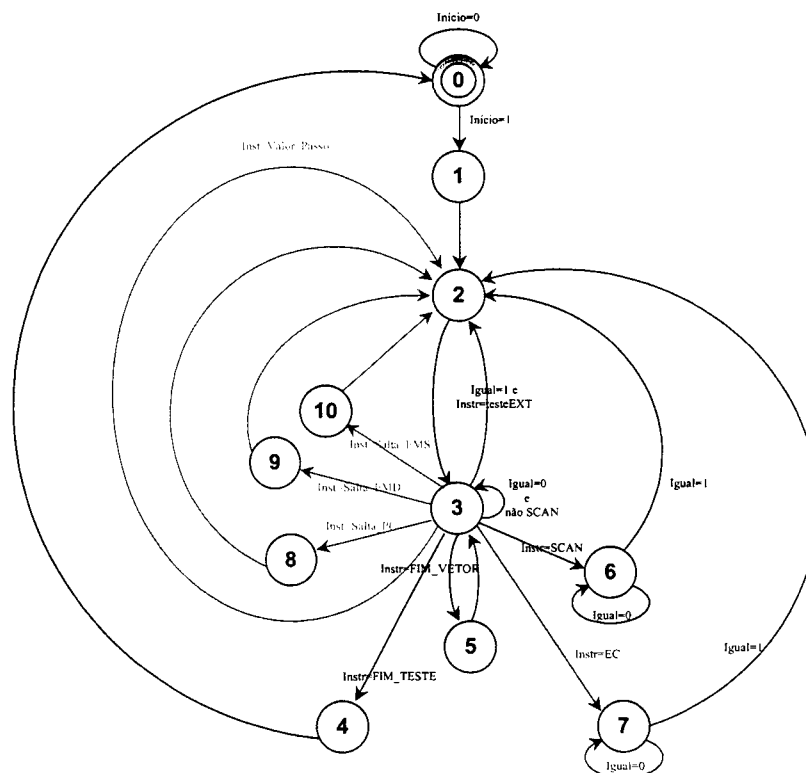


Figura 5.26 - Diagrama de estados do bloco de controle com instruções de saltos.

A figura 5.26 mostra o diagrama de estados para a máquina de controle do MET para todos os modelos implementados e a tabela 5.10 apresenta comparações de área e frequência de operação entre os modelos.

Tabela 5.10 - Comparativo entre diversas implementações do MET.

Versão do MET	Dispositivo (FLEX10K - Altera®)	Nº Células Lógicas	Nº de Flip-flops	Frequência (MHz)
<i>Original</i>	EPF10K40RC208-3	257	158	37,87
<i>At-speed e passo variável</i>	EPF10K40RC208-3	264	159	35,97
<i>At-speed, passo variável e instruções BIST</i>	EPF10K40RC208-3	292	182	33,44
<i>At-speed, passo variável, memória Scan e sinal de espera</i>	EPF10K40RC208-3	304	184	35,58
<i>At-speed, passo variável, memória Scan, sinal de espera e instruções de salto</i>	EPF10K40RC208-3	378	188	22,77

Naturalmente o incremento de funcionalidades no controlador irá causar um acréscimo em área e uma degradação na frequência de operação, conforme podemos verificar na tabela 5.10. Outras funções podem ser pensadas para tornar o controlador mais eficiente neste ou naquele tipo de teste. Porém, um controlador de teste interno não precisa necessariamente possuir todas as funções de teste, isto porque raramente o teste completo de um sistema terá essa exigência. O ideal é que o controlador seja um elemento que possa ser parametrizado pelo integrador do sistema a partir de informações básicas e específicas para aquele sistema que esta sendo integrado. Desta forma, o controlador será menor e mais eficiente.

6 CONCLUSÕES

Alguns dos exemplos apresentados neste trabalho exploram situações que visam enfatizar a complexidade do teste em sistemas baseados em núcleos de hardware, principalmente no item que se refere ao gerenciamento do teste. Os estudos demonstram a viabilidade de uso de um controlador interno e as alterações realizadas buscam fazer com que o controlador ofereça um melhor desempenho do teste e, principalmente, que este dispositivo seja compatível com sistemas que apresentam diferentes estratégias de teste. Também busca-se fazer com que o controlador seja o mais independente possível de equipamentos externos, assim, diminui-se a exigência de pinos extras e testadores caros.

O estudo primeiro se preocupa com os problemas individuais dos núcleos com teste externo, teste com cadeias de varredura, teste com vetores compactados, teste com BIST e termina apresentando alguns aspectos do teste do sistema relativos à necessidade de memórias e conexões, demonstrando que um estudo adequado da forma como os dados são dispostos na memória pode reduzir o tempo de teste, bem como o número de conexões necessárias para execução do teste.

O uso de memórias deve ser realizado da forma mais eficiente possível. Para a otimização do uso das memórias, o teste deve ser executado considerando a reconfigurabilidade dos caminhos de acesso, de forma a utilizar a máxima largura de palavra sempre que possível. A reconfiguração dos caminhos deve ser executada *on-line* durante o teste por meio da TAM e dos *wrappers*. A medida em que os núcleos são retirados ou colocados no sistema, os *wrappers* são re-configurados para melhorar os acessos. Com esta estratégia o número de conexões e o tempo de teste do sistema total tende a ser diminuído.

Também existem casos onde a quantidade de memória é restrita ou não existe a possibilidade de uso de conexões internas para o teste. Nestes, o controlador interno pode ser utilizado para realizar o controle do teste nos núcleos críticos, onde exista a necessidade de melhores desempenhos durante o teste, e deixa-se para um ATE de menor custo, os núcleos de menores exigências diminuindo-se, assim, os requisitos de interface entre o SOC e o testador.

A medida em que vai-se atribuindo mais funcionalidades ao controlador observa-se que a área em hardware necessária para a sua implementação é incrementada, e a sua frequência de operação diminui. O incremento de funcionalidades no controlador tem o intuito de tornar este dispositivo compatível com as diferentes estratégias de teste. Porém, sabe-se que raramente um sistema exigirá todos os tipos de estratégias para o seu teste.

Na era SOC o controlador (externo ou interno), sem dúvida, é um elemento de fundamental importância para o sucesso do teste. No entanto, alguns aspectos devem ser considerados devido à complexidade de decisões que o integrador do sistema deve tomar antes de decidir a maneira como o teste vai ser executado. Um elemento de apoio fundamental para a tomada de decisões pelo integrador, e que pode melhorar sensivelmente o desempenho de controladores internos, é o uso das ferramentas de planejamento do teste. Porém, é muito importante que este planejamento leve em conta os parâmetros do controlador, além de informações básicas sobre o sistema (estratégias de teste, tipos de acesso, etc) e das exigências do integrador (tempo, área extra).

O desenvolvimento de uma ferramenta que gere um controlador embutido a partir de parâmetros básicos é trabalho futuro. O planejamento de teste é proposto a partir do controlador gerado. Reuso de memória, TAM, posição do controlador no sistema e *wrappers* devem ser considerados no plano de teste, assim como informações sobre os vetores e como eles devem ser montados nas memórias também devem ser considerados pela ferramenta. Velocidade de operação do controlador de teste e dos núcleos também merecem especial atenção a fim de obter um melhor desempenho.

Em resumo, este trabalho procurou fazer uma abordagem ampla do teste de sistemas integrados utilizando um controlador específico, com a finalidade de proporcionar um meio alternativo em substituição aos testadores externos. Os resultados aqui apresentados demonstram que este tipo de dispositivo tem grande aplicabilidade, mas muito ainda tem que ser feito do ponto de vista da utilização de controladores em nível de sistema.

REFERÊNCIAS BIBLIOGRÁFICAS

- [ABO 83] M. E. Aboulhamid and E. Cervy. **A Class of Test Generators for Built-In Testing.** *IEEE Transactions on Computers*, Vol. C-32, No 10, pages 957-959, October, 1983.
- [ADH 99] Saman Adham et al. **Preliminary Outline of IEEE P1500 Scalable Architecture for Testing Embedded Cores.** In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 483–488, Dana Point, CA, April, 1999. IEEE Computer Society Press.
- [AGA 81] V. K. Agarwal and E. Cervy. **Store and Generate Built-In Testing Approach.** In *Proceedings of FTCS-11*, pages 35-40, 1981.
- [ALL 00] Alliance, VSI. IEEE P1450 Web Site, <http://grouper.ieee.org/groups/1450/>
- [BEN 00] A. Benso, S. Chiusano, S. D. Carlo, P. Prinetto, F. Ricciato, M. Spadari, and Y. Zorian. **HD2BIST: a Hierarchical Framework for BIST Scheduling, Data Patterns Delivering and Diagnosis in SoCs.** In *International Test Conference (ITC)*, pages 892–901. IEEE Computer Society, October, 2000.
- [CHA 00] A. Chandra and K. Chakrabarty. **Test Data Compression for System-on-a-Chip Using Golomb Codes.** In *Proceedings IEEE VLSI Test Symposium*, pages 113-120, 2000.
- [CHA 00a] K. Chakrabarty. **Test Scheduling for Core-based Systems Using Mixed-integer linear Programming.** In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(10):1163-1174, October, 2000.

- [CHA 01] A. Chandra and K. Chakrabarty. **System-on-a-chip Test Data Compression and Decompression Architectures Based on Golomb Codes**. In *IEEE Trans. CAD*, vol. 20, March, 2001.
- [CHA 01a] A. Chandra and K. Chakrabarty. **Efficient Test Data Compression and Decompression for System-on-a-Chip Using Internal Scan Chains and Golomb Coding**. In *Proceedings Design Automation and Test in Europe DATE*, pages 145-149, Germany, March, 2001.
- [COT 01] Érika Cota, L. Brisolara, L. Carro, A. Susin, M. Lubaszewski. **MET: An Embedded Processor for Test Controlling**. In: *5th IEEE International Workshop on Testing Embedded Core-based Systems (TECS)-Marina Beach Marriott – Marina del Rey, Los Angeles, CA, May, 2001*.
- [COT 02] Érika Cota, L. Carro, A. Orailoglu and M. Lubaszewski. **Test Planning and Design Space Exploration in a Core-based Environment**. *To appear in Design Automation and Test in Europe (DATE 2002)*. Paris, France, March, 2002.
- [COT 99] Érika Cota, M. R. Krug, M. Lubaszewski, L. Carro and A. Susin. **Implementing a Self-testing 8051 Microprocessor**. In *Symposium on Integrated Circuits and Systems Design, 12*, pages 202–205. IEEE Computer Society, October, 1999.
- [DAN 84] R. Dandapani, J. Patel and J. Abraham. **Design of Test Patetern Generators for Built-In Test**. In *Proceedings International Test Conference*, pages 315-319, 1984.
- [DRE 98] J. Dreibelbis, J. Barth, and H. Kalter. **Processor-Based Built-In Self-Test for Embedded DRAM**. *IEEE Journal of Solid-State Circuits*, 33(11):1731–1740, November, 1998.
- [DUF 93] C. Dufaza, C. Chevalier and L. F. C. Lew Yan Voon. **LFSROM: A hardware test Pattern Generator for Detetministic ISCAS 85 Test Sets**. In *Proceedings of Asian Test Symposium*, pages 160-165, 1993.

- [EDI 92] G. Edirisooriya and J. P. Robinson. **Design of Low Cost ROM Based Test Generators**. In *Proceedings of VLSI Test Symposium*, pages 61-66, 1992.
- [FRA 00] D. Franco and L. Carro. **FPGA Based Systems with Linear and Non-linear Signal Processing Capabilities**. In *Euromicro*, pages 260-64, September, 2000.
- [GOL 66] S. W. Golomb. **Run-length Encoding**. In *IEEE Trans. Inf. Theory*, vol. IT-12, pages 399-401, 1966.
- [HAM 98] K. Hamzaoglu and J. H. Patel. **Test Set Compaction Algorithms for Combinational Circuits**. In *Proceedings International Test Conference on CAD*, pages 283-289, 1998.
- [HEL 96] S. Hellebrand, H.-J. Wunderlich, and A. Hertwig. **Mixed-Mode BIST Using Embedded Processors**. In *Proceedings International Test Conference (ITC)*, pages 195-204, Washington, DC, October, 1996. IEEE Computer Society Press.
- [HIT 01] The University of Illinois, <http://www.crhc.uiuc.edu/IGATE/hitec-vectors.html>
- [IEE 15] IEEE P1500 Web Site, <http://grouper.ieee.org/groups/1500/>.
- [IEE 90] IEEE Standard 1149.1 – 1990, *IEEE Standard Test Port Access Port and Boundary Scan architecture*, IEEE Standards Board, 345 East 47th Street, New York, NY 10017-2394, USA, 1990.
- [ISC 85] F. Brglez and H. Fujiwara. **A Neural of 10 Combinational Benchmark Circuits and a Target Translator in Fortran**. Distributed on a tape to participants of the Special Session on ATPG and Fault Simulation, *International Symposium on Circuits and Systems*, June, 1985; partially characterized in F. Brglez, P. Pownall, R. Hum. **Accelerated ATPG and Fault Grading via Testability Analysis**. In *Proceedings IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 695-698, June, 1985.

- [ISC 89] F. Brglez, D. Bryan and K. Kosminski. **Combinational Profiles of Sequential Benchmark Circuits**. In *Proceedings IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1929-1934, May 1989.
- [ITO 00] S. A. Ito, L. Carro, and R. Jacobi. **System Design Based on Single Language and Single-chip Java Asip Microcontroller**. In *Design, Automation & Test in Europe*, pages 703–707, Los Alamitos, March, 2000. IEEE Computer Society.
- [IYE 01] Vikram Iyengar and Krishnendu Chakrabarty. **Precedence-Based, Preemptive, and Power-Constrained Test Scheduling for System-on-a-Chip**. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 368-374, Marina del Rey, CA, May, 2001. IEEE Computer Society Press.
- [IYE 98] Vikram Iyengar, K. Chakrabarty and B. T. Murray. **Built-In Self Testing of Sequential Circuits Using Precomputed Test Sets**. In *Proceedings of VLSI Test Symposium*, pages 418-423, 1998.
- [JAS 98] A. Jas and N.A. Touba. **Test Vector Decompression Via Cyclical Scan Chains and Its Application to Testing Core-Based Designs**. In *proceedings International Test Conference (ITC)*, pages 458-464, 1998.
- [JAS 99] Abhijit Jas and Nur Touba. **Using an Embedded Processor for Efficient Deterministic Testing of Systems-on-a-Chip**. In *Proceedings International Conference on Computer Design (ICCD)*, Austin, TX, October, 1999.
- [KAP 99] Rohit Kapur et al. **P1500-CTL: Towards a Standard Core Test Language**. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 489–490, Dana Point, CA, April, 1999. IEEE Computer Society Press.
- [LAL 97] Lala, P.G. **Digital Circuit Testing and Testability**, Academic Press, 1997.
- [LAR 01] Erik Larsson and Zebo Peng. **Test Scheduling and Scan-Chain Division Under Power Constraint**. In *Proceedings IEEE Asian Test Symposium (ATS)d*, pages 259–264, Kyoto, Japan, November, 2001. IEEE Computer Society Press.

- [LAZ 02] Cristiano Lazzari. **Arquiteturas de Apoio ao P1500 para Teste em SOCs: Wrapper Parametrizável e Controlador de Teste**. RP315, PPGC-UFRGS, Porto Alegre, 2002.
- [MAR 00a] Erik Jan Marinissen, Rohit Kapur and Yervant Zorian. **On Using IEEE P1500 SECT for Test Plug-in-Play**. In *Proceedings International Test Conference (ITC)*, pages 770–777, Atlantic City, NJ, October, 2000. IEEE Computer Society Press.
- [MAR 00b] Erik Jan Marinissen, Sandeep Kumar Goel, and Maurice Lousberg. **Wrapper Design for Embedded Core Test**. In *Proceedings International Test Conference (ITC)*, pages 911–920, Atlantic City, NJ, October, 2000. IEEE Computer Society Press.
- [MAR 99] Erik Jan Marinissen, Yervant Zorian, Rohit Kapur, Tony Taylor, and Lee Whetsel. **Towards a Standard for Embedded Core Test: An Example**. In *Proceedings International Test Conference (ITC)*, pages 616–627, Atlantic City, NJ, September, 1999. IEEE Computer Society Press.
- [PAP 99] C. A. Papachristou, F. Martin, and M. Nourani. **Microprocessor Based Testing for Core-Based System on Chip**. In *ACM/IEEE Design Automation Conference*, pages 586–591. ACM, June, 1999.
- [RAJ 97] Rajesh K. Gupta and Yervant Zorian. **Introducing Core-Based System Design**. In *IEEE Design & Test of Computers*, 14(4):15–25, December, 1997.
- [RAJ 99] Rochit Rajsuman. **Testing a System-on-a-Chip with Embedded Microprocessor**. In *Proceedings International Test Conference (ITC)*, pages 499–508, Atlantic City, NJ, September, 1999. IEEE Computer Society Press.
- [REN 98] M. Renovell. **SRAM-Based FPGAs: A Structural Test Approach**. In *Proceedings XI Brazilian Symposium on Integrated Circuit Design (SBCCI98)*, pages 67-72, Rio de Janeiro, October, 1998.
- [ROA 01] The National Technology Roadmap for Semiconductors, Semiconductor Industry Association, 2001, <http://public.itrs.net/Files/2001ITRS/Home.htm>

- [SHE 94] Meng-Lieh Sheu, Chung Len Lee. **Simplifying Sequential Circuit Test Generation**. In *IEEE Design and Test of Computers*, pages 28-38 (Vol 11, No.3), July/September, 1994.
- [SOC 99] Society, IEEE Computer. **IEEE Standard Test Interface Language (STIL) for digital Test Vector Data – Language Manual – IEEE Std. 1450.0 – 1999**. New York: [s.n.], 1999.
- [STR 98] A. P. Ströle. **Bit Serial Pattern Generation and Response Compaction Using Arithmetic Functions**. In *Proceedings IEEE VLSI Test Symposium*, pages 78-84, Monterey, CA, April, 1998.
- [VHD 01] <ftp://erm1.u-strasbg.fr/pub/vhdl/ISCAS.vhdl/>
- [WUN 98] Hans-JoachimWunderlich. **BIST for Systems-on-a-chip**. *International, the VLSI Journal*, [s.l.], n.26, pages 55-78, 1998.
- [ZOR 00] Yervant Zorian and Erik Jan Marinissen. **System Chip Test: How Will It Impact Your Design?** In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 136–141, Los Angeles, June, 2000. Association for Computing Machinery, Inc.
- [ZOR 97] Yervant Zorian. **Test Requirements for Embedded Core-Based Systems and IEEE P1500**. In *Proceedings International Test Conference (ITC)*, pages 191–199, Washington, DC, November, 1997. IEEE Computer Society Press.
- [ZOR 98] Yervant Zorian, Erik Jan Marinissen, and Sujit Dey. **Testing Embedded-Core Based System Chips**. In *Proceedings International Test Conference (ITC)*, pages 130–143, Washington, DC, October, 1998. IEEE Computer Society Press.

ANEXOS

Os anexos estão em formato eletrônico, gravados no disquete que acompanha esta dissertação. Nestes anexos encontram-se as rotinas em MatLab® para codificação Golomb e também as diversas implementações em VHDL dos controladores de teste utilizados durante os estudos deste trabalho. Os anexos estão divididos em:

- Anexo 1: Controlador com at-speed e passo variável;
- Anexo 2: Controlador com at-speed, passo variável e instruções BIST;
- Anexo 3: Controlador com memória Scan;
- Anexo 4: Controlador com decodificador Golomb;
- Anexo 5: Controlador com todas modificações comentadas;
- Anexo 6: Rotinas em MatLab® para codificação Golomb;
- Anexo 7: Diversas implementações.