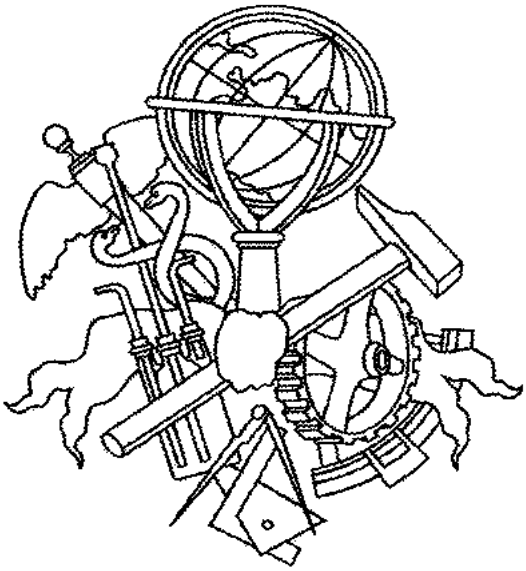
Instituto Superior de Engenharia do Porto



ISEP

Controlador Boundary-Scan

Ricardo Miguel Carvalho Costa

Dissertação realizada no âmbito do

Mestrado em Engenharia Electrónica e Computadores

Orientador: Eng.º André Fidalgo

Resumo

A crescente complexidade dos sistemas electrónicos associada a um desenvolvimento nas tecnologias de encapsulamento levou à miniaturização dos circuitos integrados, provocando dificuldades e limitações no diagnóstico e detecção de falhas, diminuindo drasticamente a aplicabilidade dos equipamentos ICT. Como forma de lidar com este problema surgiu a infra-estrutura *Boundary Scan* descrita na norma IEEE1149.1 "Test Access Port and Boundary-Scan Architecture", aprovada em 1990. Sendo esta solução tecnicamente viável e interessante economicamente para o diagnóstico de defeitos, efectua também outras aplicações. O SVF surgiu do desejo de incutir e fazer com que os fornecedores independentes incluíssem a norma IEEE 1149.1, é desenvolvido num formato ASCII, com o objectivo de enviar sinais, aguardar pela sua resposta, segundo a máscara de dados baseada na norma IEEE1149.1.

Actualmente a incorporação do *Boundary Scan* nos circuitos integrados está em grande expansão e consequentemente usufrui de uma forte implementação no mercado. Neste contexto o objectivo da dissertação é o desenvolvimento de um controlador *boundary scan* que implemente uma interface com o PC e possibilite o controlo e monitorização da aplicação de teste ao PCB. A arquitectura do controlador desenvolvido contém um módulo de Memória de entrada, um Controlador TAP e uma Memória de saída.

A implementação do controlador foi feita através da utilização de uma FPGA, é um dispositivo lógico reconfiguráveis constituído por blocos lógicos e por uma rede de interligações, ambos configuráveis, que permitem ao utilizador implementar as mais variadas funções digitais. A utilização de uma FPGA tem a vantagem de permitir a versatilidade do controlador, facilidade na alteração do seu código e possibilidade de inserir mais controladores dentro da FPGA.

Foi desenvolvido o protocolo de comunicação e sincronização entre os vários módulos, permitindo o controlo e monitorização dos estímulos enviados e recebidos ao PCB, executados automaticamente através do *software* do Controlador TAP e de acordo com a norma IEEE 1149.1. A solução proposta foi validada por simulação utilizando o simulador da Xilinx. Foram analisados todos os sinais que constituem o controlador e verificado o correcto funcionamento de todos os seus módulos. Esta solução executa todas as sequências pretendidas e necessárias (envio de estímulos) à realização dos testes ao PCB. Recebe e armazena os dados obtidos, enviando-os posteriormente para a memória de saída.

A execução do trabalho permitiu concluir que os projectos de componentes electrónicos tenderão a ser descritos num nível de abstracção mais elevado, recorrendo cada vez mais ao uso de linguagens de hardware, no qual o VHDL é uma excelente ferramenta de programação. O controlador desenvolvido será uma ferramenta bastante útil e versátil para o teste de PCBs e outras funcionalidades disponibilizadas pelas infra-estruturas BS.

Abstract

The ever-growing complexity of electronic systems associated to the development of encapsulation technologies has led to the miniaturisation of integrated circuits, causing difficulties and strains in the diagnosis and detection of flaws, hence drastically diminishing the applicability of ICT equipment. As a means of dealing with such difficulty, the infrastructure *Boundary Scan* was created, as ascribed in the IEEE 1149.1, "Test Access Port and Boundary-Scan Architecture", introduced in 1990. As it presented itself as a technically viable and economically interesting solution for the diagnosis of flaws, it is also capable of other applications. The SVF was a direct effect of the resolve to make independent vendors include the IEEE1149.1, and it is developed in ASCII format with the purpose of sending signals and waiting for answers, according to the data mask of the abovementioned standard.

Presently, the inclusion of the *Boundary Scan* in integrated circuits is highly expanded and, subsequently, is having the benefit of strong market implementation. In such context, the main goal of the present thesis is the enhancement of a *Boundary Scan* controller which implements an interface with the computer, thus allowing control and monitoring of test application to the PCB. The software architecture contains an Input Memory module, a TAP Controller and an Output Memory.

Controller implementation has been carried out through an FPGA, a logic reconfigurable device made of logic blocks and a net of reconfigurable interconnects, thus allowing users to implement a vast array of digital functions. The relevance of an FPGA has the advantage of allowing versatility in the controller, easy to modify your code and able to inside more controllers within of an FPGA.

A protocol for communication and synchronisation of different modules has been developed, allowing control and monitoring of values to and from the PCB, automatically executed through the TAP Controller software, according to IEEE1149.1. The solution presented was validated by simulation using Xilinx's simulator. All signals in the controller were thoroughly verified, as it was the accurate use of all its modules. This solution carries out all the required sequences (value sending), mandatory in PCB testing. It collects and stores data, ultimately sending it to the output memory.

The implementation of this assignment has certified the conclusion that electronic component projects tend to be depicted in a higher level of abstraction, by means of hardware description languages, in which VHDL has been proved an outstanding programming tool. The controller will prove to be a very useful and versatile tool for PCB testing and to other applications made available by BS infrastructures.

Agradecimentos

Foram várias as pessoas que, durante a elaboração desta dissertação, contribuíram com a sua ajuda quer na orientação, quer no apoio.

A minha primeira referência vai para o meu orientador, o Professor Eng.º André Fidalgo, a quem gostaria de apresentar os mais sinceros agradecimentos pelo seu empenho e auxilio na resolução dos problemas surgidos. O seu apoio e motivação foram bastante gratificantes desta longa aprendizagem.

Uma palavra de apreço também aos meus colegas do ISEP, com os quais muito convivi e aprendi.

Por último, com um carinho muito especial, aos meus pais, pela confiança e encorajamento e à minha namorada pelo seu constante incentivo, ajuda e paciência, a quem devo muito tempo.

A todos, o meu sincero obrigado.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

AOI	*Automatic Optic inspection*
ATE	*Automatic Test Equipment*
ASIC	*Application Specific Integrated Circuits*
BGA	*Ball Grid Array*
BP	*Bypass*
BS	*Boundary Scan*
BST	*Boundary Scan Test*
CCD	*Charge-Coupled Device*
CPLD	*Complex Programmable Logic Devices*
DUT	*Device Under Test*
EEPROM	*Electrically Erasable Programmable Read-Only Memory*
FPGA	*Field Programmable Gate Array*
IC	*Integrated Circuit*
ICT	*In Circuit Test*
IEEE	*Institute of Electrical and Electronics Engineers*
I/O	*Input/Output*
IR	*Instruction Register*
ISP	*In-System Programming*
JTAG	*Joint Test Action Group*
LASER	*Light Amplification by Stimulated Emission of Radiation*
MUX	*Multiplexer*
PCB	*Printed Circuit Board*

SMT	*Surface Mount Technology*
SVF	*Serial Vector Format*
TAP	*Test Access Port*
TCK	*Test Clock*
TDI	*Test Data Input*
TDO	T*est Data Output*
TMS	*Test Mode Select*
TRST	*Test Reset*
VHDL	VHSIC *Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuits*

1. Introdução

O aumento da complexidade dos circuitos integrados, nomeadamente os dispositivos lógicos programáveis FPGA (*Field Programmable Gate Arrays*), associado a uma expansão considerável das tecnologias de fabricação nas últimas duas décadas levou ao aparecimento de componentes com maior capacidade e complexidade e à sua inerente flexibilidade. Esta evolução foi acompanhada com novas técnicas de encapsulamento e do seu processo de fabrico, reduzindo componentes internos para escalas micrométricas. Este processo originou um aumento da probabilidade de ocorrência de defeitos, levando à procura de novos métodos sem acesso físico à placa que permitissem a detecção e diagnóstico de faltas e que fossem capazes de assegurar a sua fiabilidade a longo prazo.

O surgimento da norma IEEE 1149.1, em 1990, provocou um aumento de componentes com esta tecnologia, levando a um decréscimo na utilização dos testes ICT (*In Circuit Test)* e consequentemente uma rápida afirmação e expansão, mas também, originando várias evoluções da mesma.

A norma permite efectuar testes estruturais em placas de circuito impresso digitais; os integrados que estão de acordo com a norma têm a capacidade de isolar a lógica interna dos pinos (células *Boundary Scan*), proporcionando uma infra-estrutura de teste residente nos próprios circuitos integrados, que funciona de forma semelhante a uma matriz de agulhas electrónica, tornando possível a aplicação de vectores de teste que permitem a detecção de faltas no circuito.

O trabalho descrito nesta dissertação tem como objectivo desenvolver um controlador em VHDL, que permite o controlo dos sinais que são injectados nos pinos reservados à realização do teste *Boundary Scan*, de acordo com a norma de teste IEEE 1149.1.

Os módulos VHDL foram desenvolvidos com o intuito de realizar uma rápida análise dos dispositivos digitais de modo a permitir uma rápida prototipagem.

É abordado o formato SVF, este permite descrever sequências ordenadas de testes baseados na norma.

O *hardware* a utilizar consiste numa placa de desenvolvimento baseada numa FPGA, tipo Spartan-3E da Xilinx.

2. Metodologias de Teste (*PCB*)

A evolução dos tempos leva-nos à necessidade de criar equipamentos mais pequenos, mas também mais complexos.

Essa necessidade levou a um grande avanço na tecnologia dos semicondutores, à redução das dimensões físicas dos circuitos integrados (ICs), a velocidades mais rápidas e a complexas arquitecturas dentro de um único encapsulamento.

O aumento do número de pinos dos ICs e a falta de espaço físico nas placas electrónicas criou um problema complexo para os projectistas de placas electrónicas e para quem tem a tarefa de criar formas de as testar.

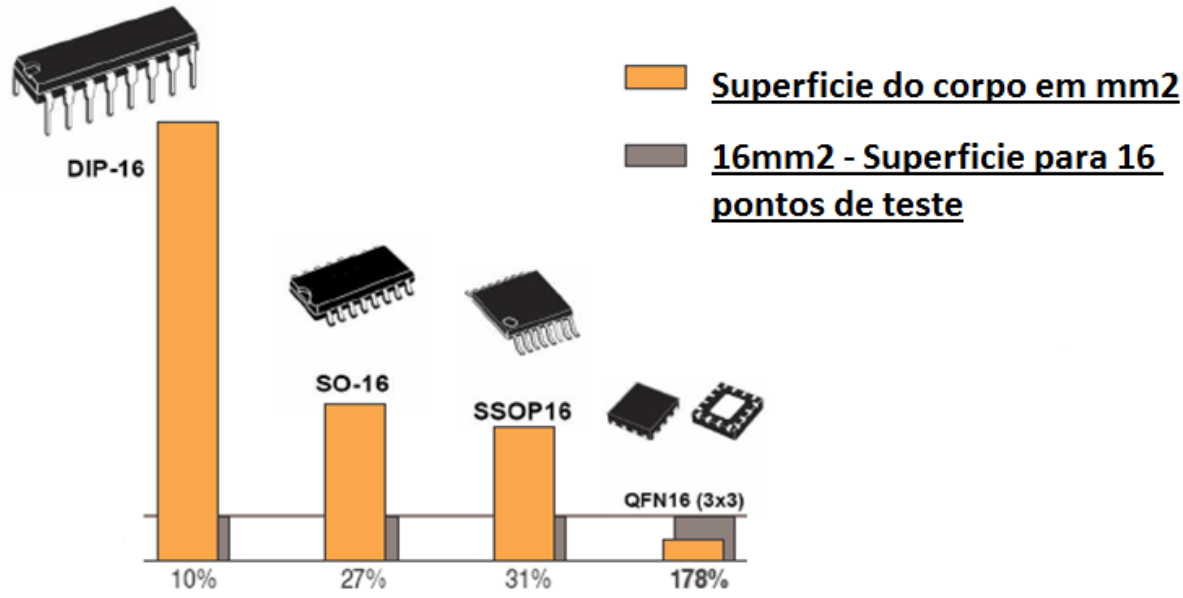


Figura 1 – Área do Componente vs pontos de referência

Na figura verificamos a evolução da tecnologia dos semicondutores dos ICs e consequente diminuição do seu encapsulamento. Conclui-se que para testar na totalidade um IC (inserindo pontos de teste no PCB para todos os pinos) com o *package* DIP-16 somente necessitaríamos de 10% de área quando comparada com a do IC. O mesmo tipo de teste para os novos package QFN-16 a área necessária aumenta para 178%. Concluímos que a área necessária para testes no PCB é aproximadamente o dobro quando comparada com a dos ICs. [Goepel09]

Devido a esta evolução, em 1985 um grupo de empresas europeias de sistemas electrónicos, formaram o "Joint European Test Action Group" (JETAG), decidiram

juntar-se para resolver estes novos desafios, no sentido de resolver a falta de espaço físico na placa e na redução de falhas diagnosticadas.

O método escolhido pelo grupo foi de aceder aos pinos dos ICs por meio de um "*serial shift register*" interno através dos limites dos dispositivos, *boundary scan register*.

Em 1988 com a entrada neste grupo da América do Norte formaram o *"Joint Test Access Group*" (JTAG), em 1990, o IEEE refinou o conceito e criou a norma 1149.1, conhecida como, *IEEE Test Access Port and Boundary Scan Architecture.*

O *boundary-scan* não impede a evolução tecnológica dos ICs, como também não limita o espaço físico dos PCB.

2.1. Inspecção Visual Manual

O ser humano, é sem dúvida o mais flexível e inteligente de todos os sistemas na inspecção de placas electrónicas. Este possui habilidade de relembrar vários detalhes dos critérios de inspecção; perceber detalhes de cor e geometria e interpretar novas e imprevistas circunstâncias que são difíceis de serem alcançadas por qualquer tecnologia de *software* e hardware actualmente existentes.

Figura 2 – Detecção de falhas através de inspecção humana

No entanto, o elemento humano, num processo de inspecção, contribui para uma má qualidade no controlo de detecção de erros de inspecção (devido à fadiga) e para a não uniformidade dos tempos de controlo.

Os erros de inspecção são de várias categorias:

- Erros técnicos (falta de capacidade para o cargo, falta de experiencia)
- Erros por inadvertência (distracção, descuido)
- Erros conscientes (fraude)

Para minimizar os erros de inspecção, é necessário um programa de formação e consciencialização dos operadores e a utilização de especificações de manuseamento (padrões com as definições de critérios de bom/mal ou aceitável/não-aceitável).

As ferramentas que auxiliam na inspecção visual manual variam de um simples microscópio até microscópios com monitor que giram com um determinado ângulo, sendo que, para uma boa inspecção-geral é necessário um aumento de três a dez vezes. Também é imprescindível uma boa iluminação. Imagens fotográficas são especialmente úteis no controle de placas de circuito impresso.

2.2. Inspecção Óptica Automática (AOI)

Esta técnica, utiliza um sistema de recolha de imagem ligado a um computador, que as grava para posteriormente executar alguns tipos de análises matemáticas (algoritmos), a fim de fazer uma avaliação com critérios pré-definidos.

Estas máquinas possuem tecnologias para executarem inspecções na forma 2-D (duas dimensões) ou 3-D (três dimensões).

Nos sistemas de inspecção 2-D, o processo começa com a captura da imagem através de uma câmara CCD. Em seguida, o processador irá transformá-la numa matriz de elementos de imagem ou pixéis; para cada pixel é atribuído um valor analógico, dependendo da luminosidade. Este é então convertido num valor digital para posteriormente ser analisado.

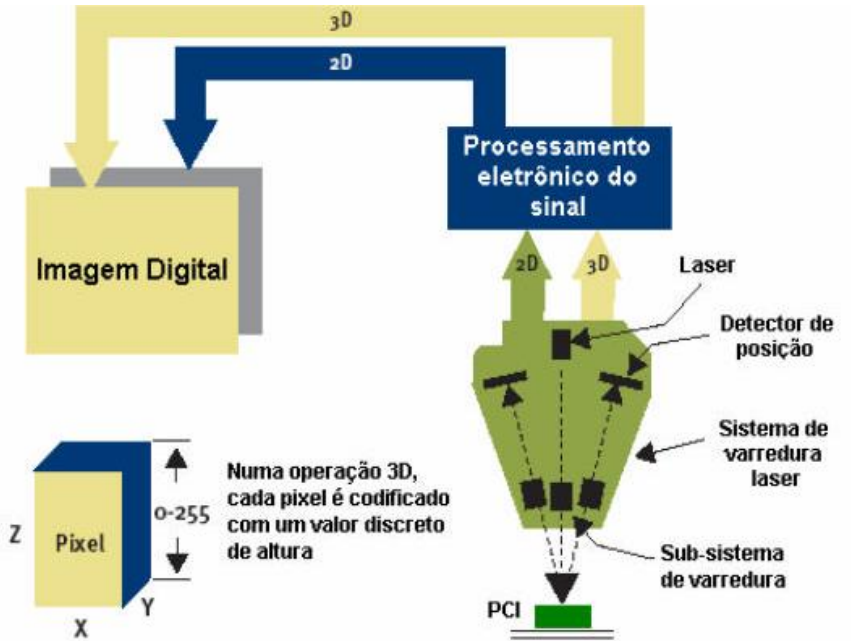


Figura 3 – Principio de funcionamento AOI tridimensional

Os sistemas de inspecção 3D, usam um sistema, geralmente de varredura a laser, para criar a imagem da placa de circuito impresso em forma tridimensional. A imagem 3D é baseada na altura da superfície da placa e dos componentes. Esta tecnologia não foi bem aceite no mercado, devido ao seu elevado custo e qualidade, comparando-a com a tecnologia AOI com câmara CCD.

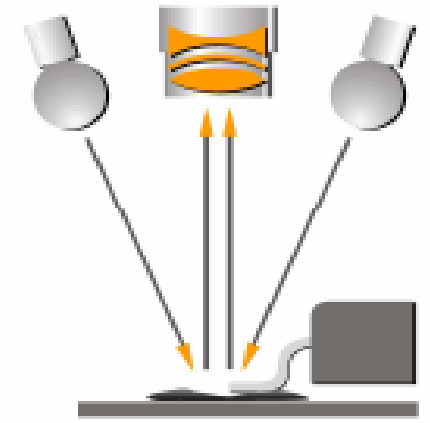


Figura 4 – Principio de funcionamento AOI [Doro 04]

O feixe de luz inserido na parte superior do pino de componente é espelhado para a câmara (aparecendo na imagem uma zona clara), na junção entre a solda com o pino do componente forma-se um ângulo, o feixe de luz ao inserir nesse ângulo não é

reflectido para a câmara (aparece na imagem uma zona escura, predefinida por *software* através de um PCB padrão), caso nessa zona a luz seja reflectida, indica que existe uma falha: solda fria ou inexistência de solda.

A máquina faz uso de uma placa de circuito impresso de referência, também chamada de "*golden board"*, que é gravada na memória. Desta forma, cada ponto da placa de teste é comparado com a placa padrão, pixel a pixel, para determinar se existe algum defeito possível.

A principal característica destas máquinas é o seu alto rendimento, que permite executar a inspecção completa de uma placa à mesma velocidade da linha de produção, permitindo guardar os resultados em memória para posterior reparação. [mckenzie03] [ledden03]

2.3. Inspecção Raio-X

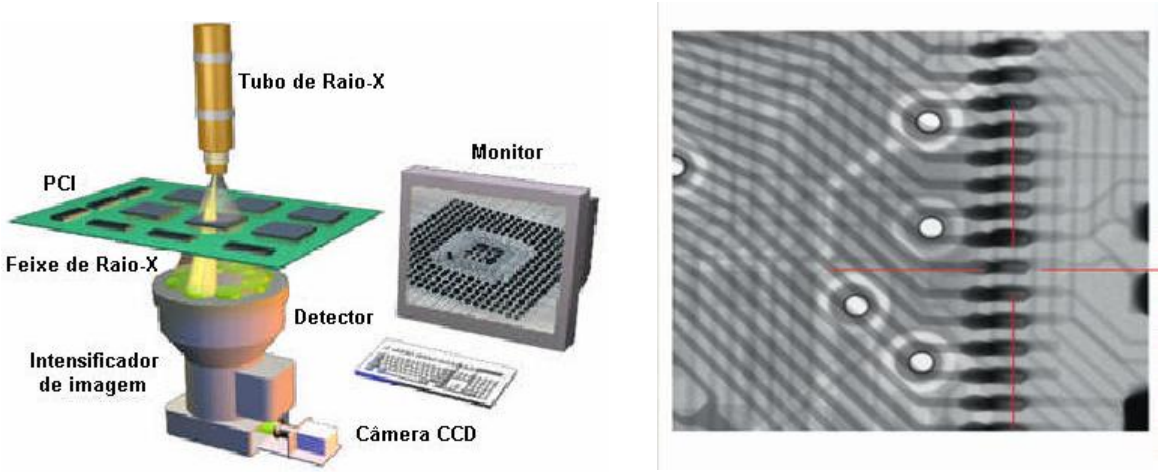


Figura 5 - Funcionamento das máquinas de inspecção por raio-X

O uso industrial do raio-X para testes não destrutivos, é baseado no princípio de absorção da penetração da radiação. Durante a inspecção, a radiação emitida da fonte passa através da placa de circuito impresso, sendo parcialmente absorvida pelo detector (materiais de diferentes espessuras e densidades irão absorver radiação em diferentes quantidades).

O detector, por sua vez, direcciona a imagem através de um espelho para uma câmara de vídeo onde a imagem é digitalizada e enviada ao processador de imagem, a fim de mostrar, ampliar e analisar a imagem. Estas máquinas possuem sistemas de inspecção automática e manual com tecnologia de visualização de imagem na forma 2-D e 3-D.

Nos sistemas de inspecção manual, a avaliação dos defeitos é feita pelo operador através da observação da imagem. Já nos sistemas automáticos, a imagem é

examinada pixel a pixel pelo computador que utiliza algoritmos apropriados para verificar se a quantidade e a localização dos materiais estão de acordo com os valores pré-determinados; sendo que esta análise é registada em ficheiros, onde é indicada a localização dos seus defeitos. [mckenzie03] [ledden 03]

Dada a necessidade de cada vez mais as placas utilizarem circuitos integrados com encapsulamentos *fine pitchs* (distancia entre pinos muito reduzida, geralmente variam entre 0.2 e 0.5mm) e BGAs (*Ball Grid arrays)*, estas originaram um aumento progressivo da inspecção por raio-X. Ver exemplo na figura 6.

A) BGA B) J-Lead

Figura 6 – BGA (*Ball Grid array*) e J-lead

Os pontos de solda dos componentes *fine pitchs* com terminais *J-lead* são muito difíceis de serem inspeccionados, por sua vez, os pontos de solda dos componentes BGAs são impossíveis de serem inspeccionadas visualmente na sua totalidade.

Através do sistema de raio-X, é possível inspeccionar ambos os tipos de soldadura, bem como inspeccionar "*voids"* (buracos dentro de pontos de solda), que nenhum outro método é capaz de detectar.

Este sistema apresenta algumas desvantagens; não detecta a polaridade dos componentes e revela-se um sistema muito caro.

2.4. Equipamentos de teste ICT

Os primeiros ICs eram dispositivos relativamente volumosos testados por equipamentos facilmente disponíveis numa bancada de laboratório (multímetros, geradores de sinais, ...). Com a sua evolução, os ICs tornaram-se mais pequenos e complexos, rapidamente estes métodos ficaram obsuletos (lentos e dispendioso financeiramente).

Surgindo assim o aparecimento dos primeiros ATE (*Automatic Test Equipment*), eram aplicados a testes funcionais e paramétricos de módulos electrónicos.

Dentro dos ATE, o FATE (*Digital Functional Automatic Test*) tornou-se a solução mais popular para um teste digital; era possível ao projectista criar um modelo abstracto de um circuito e, em seguida, aplicar os sinais de teste à entrada (vectores). O circuito respondia aos estímulos e, posteriormente, dependendo da sua resposta, determinava a qualidade das placas.

O responsável pelo desenvolvimento do programa de teste, estava sempre dependente do responsável pela elaboração da placa; o que nem sempre era uma parceria pacífica. Mas com a contínua evolução dos ICs e das complexidades dos novos projectista das placas, estes modelos tornavam-se extremamente complexos, muito difíceis de elaborar, levando ao aumento do custo final do projecto. [Doro 04]

Foi desta necessidade, de resolver as limitações dos simuladores de lógica, que surgiu o ICT (*In circuit test*) método ainda muito utilizado nas indústrias electrónicas.

Existem dois principais métodos:

- Adaptador de agulhas
- *Flying prober*

2.4.1. Adaptador de agulhas

Figura 7 – Funcionamento do teste ICT (I)

Este, consiste num suporte com agulhas, distribuídas de acordo com os pontos de prova previamente desenhados nas placas electrónicos.

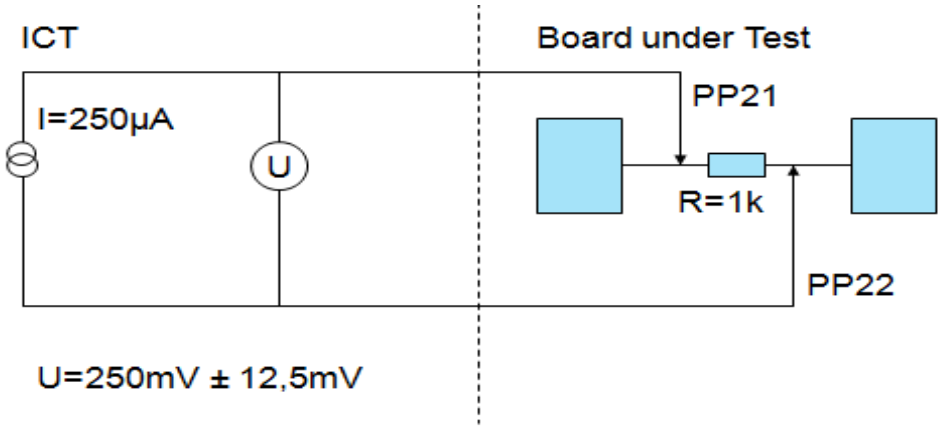


Figura 8 – Funcionamento do teste ICT (II)

A figura descreve o processo de controlo ICT, neste exemplo testa-se uma resistência. A máquina através de uma fonte de corrente interna coloca na agulha PP21 (250µA), a resistência tem um calor de 1kΩ±5%, pela lei de Ohm U=R*I, o valor esperado na agulha PP22 é de 250mV, devido á tolerância do componente ±12.5mV.

O processo é semelhante para outros componentes, nos condensadores é medido o seu tempo de carga, nos ICs são medidos os seus diodos internos. [SPEA09]

2.4.2. Flying prober

Outra forma de teste de circuito é o *flying prober* (ponta de prova flutuante). Neste, o processo de teste eléctrico é executado por cabeças de testes, tipicamente quatro ou oito, que se deslocam em alta velocidade através da placa de circuito impresso. Assim, cada ponta de prova eléctrica, localizada em cada cabeça de teste, faz o contacto e analisa sequencialmente os terminais dos componentes.

São várias as fontes de ruídos electrónicos que podem estar presentes nestes testes: cargas capacitavas parasitas, resistências dos fios e contactos.

Alguns componentes no circuito são usados em paralelo com outros, dificultando ou até mesmo impossibilitando o teste do componente individualmente (ex: resistências de baixo valor em paralelo com condensadores). O principal problema encontrado nestes sistemas reside no acesso aos pontos de testes.

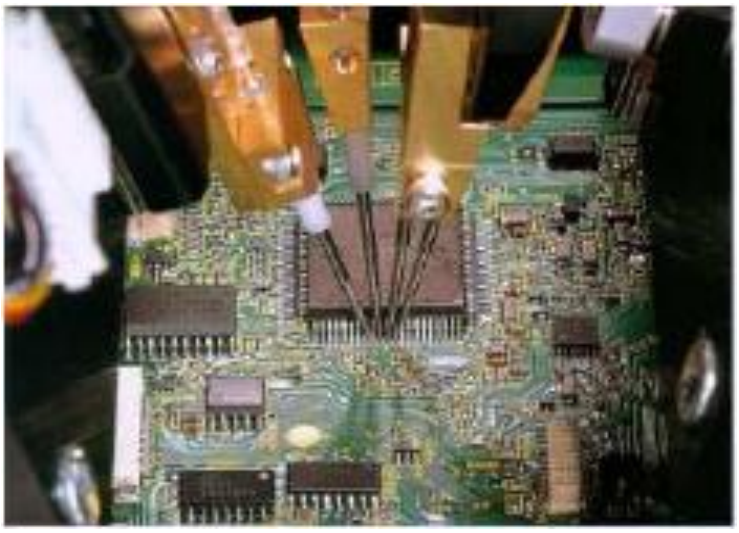


Figura 9 – Funcionamento do teste *Flying prober*

Com os componentes cada vez mais pequenos *(package)* e mais complexos, o acesso aos pontos de testes diminui significativamente, impedindo o teste completo do circuito.

Muitas vezes, uma placa protótipo é construída apressadamente num pequeno laboratório de montagem com controlo de qualidade inferior em comparação com uma produção em série. Um protótipo tem mais defeitos de montagem que uma unidade de produção. [SPEA09]

3. Introdução ao *Boundary Scan*

A norma IEEE 1149.1-1990, foi produto do trabalho realizado por um grupo de companhias interessadas em resolver basicamente o problema do acesso físico aos pinos dos circuitos integrados colocados sobre as camadas dos circuitos impressos PCB.

Com o aumento da complexidade dos circuitos impressos e a diminuição do tamanho dos componentes (devido às constantes melhorias nas tecnologias de fabricação dos circuitos integrados e dos circuitos impressos (mais camadas)), verificou-se uma maior dificuldade na detecção de falhas. Tudo isto fez com que os métodos tradicionais de detecção de falhas ficassem obsoletos (como as pontas de prova, etc.). Por estes motivos eram necessários métodos mais baratos e fiáveis para realizar provas de conectividade/detecção de falhas entre os circuitos integrados montados.

Em 1980, o grupo JTAG iniciou um processo de investigação no sentido de encontrar uma nova especificação no diagnóstico de PCBs, explorando os pinos dos circuitos integrados (*boundary-scan testing*, BST). Este modo de teste passou a norma em 1990, com o nome "IEEE Std. 1149.1-1990 Test Access Port and Boundary-Scan Architecture".

Esta norma, é conhecida pela abreviatura de "BST" e também como "JTAG", nome do grupo criador.

A grande vantagem do *Boundary-Scan*, é a capacidade para controlar e observar, mediante *software*, os valores lógicos ('0', '1') nos pinos dos ICs através de uma interface simples com apenas quatro sinais (TMS, TDI, TDO, TCK), opcionalmente cinco (TRST).

Com esta arquitectura, é possível realizar um diagnóstico ao estado das pistas do circuito impresso; falta de ligação (pistas interrompidas), curto-circuitos, soldas frias.

O teste de blocos analógicos constitui uma das áreas em que o BST encontra maiores limitações, constatamos que o domínio analógico está bastante menos desenvolvido, existem vários estudos para permitir aceder a sinais analógicos através da infraestrutura BST, a revisão da norma IEEE 1149.4 descreve como abordar estas dificuldades.

3.1. História do Boundary Scan

Com o problema da falta de espaço para a inclusão de pontos de teste nos circuitos impressos, um grupo conhecido como JTAG foi criado em 1980. Tinham como objectivo resolver os problemas enfrentados pelos fabricantes de aparelhos electrónicos com os métodos de teste existentes até à data e que permitissem a realização desses testes com outras tecnologias, desde que não interferissem com o aumento do acesso físico dos PCBs.

O objectivo do *boundary scan*, foi complementar as técnicas de teste de todos os fabricantes, que incluíssem um teste funcional, para fornecer um padrão que permitisse o ensaio digital em circuitos analógicos e de sinais mistos.

A norma para o *boundary scan* foi planeada e aprovada pelo Instituto Electric Electronics Engineers nos E.U.A. como IEEE 1149.1.

A primeira edição da norma, foi em 1990. A intenção era facilitar o teste das ligações entre os ICs montados em PCBs, módulos híbridos e outros substratos.

Dado que a maioria dos problemas com circuitos electrónicos ocorrem nas ligações entre circuitos integrados ou entre camadas de ligação do PCB, a estratégia da norma IEEE 1149.1 verificaria e resolveria a maioria dos problemas.

Em 1993, foi realizada uma revisão do *Boundary scan*. A norma IEEE 1149.1 foi reeditada e continha muitos esclarecimentos, melhorias e correcções.

Em 1994, houve uma nova revisão da norma IEEE 1149.1. Introduziu-se a linguagem de descrição *Boundary Scan* "BSDL", fez com que o teste *boundary scan* pudesse ser escrito numa linguagem comum, melhorando a forma de desenvolver os testes e o seu código reutilizado, poupando assim tempo de desenvolvimento.

Em 1995, foi aprovado e publicado a extensão 1149.5 (módulo de test standart). Também neste ano, foi aprovado e publicado a extensão 1149.4 (sinais do barramento de teste mistos "*mixed*").

A revisão da norma IEEE 1149.1 realizada em 2001, descreve quatro instruções de teste obrigatórias: *Bypass, Sample, Preload, Extest*.

A norma IEEE 1149.1, define uma série de instruções opcionais, instruções que não precisam de ser implementadas, mas que têm uma operação definida (*Intest, Idcode,* …). [Ramos] [Goepel09] [Kharagpur]

A norma IEEE 1149.6, aprovado em 2003, foi criada para resolver problemas de teste associados aos sinais DC acoplados a AC. Esta nova norma descreve brevemente estes problemas e resume como os abordar.

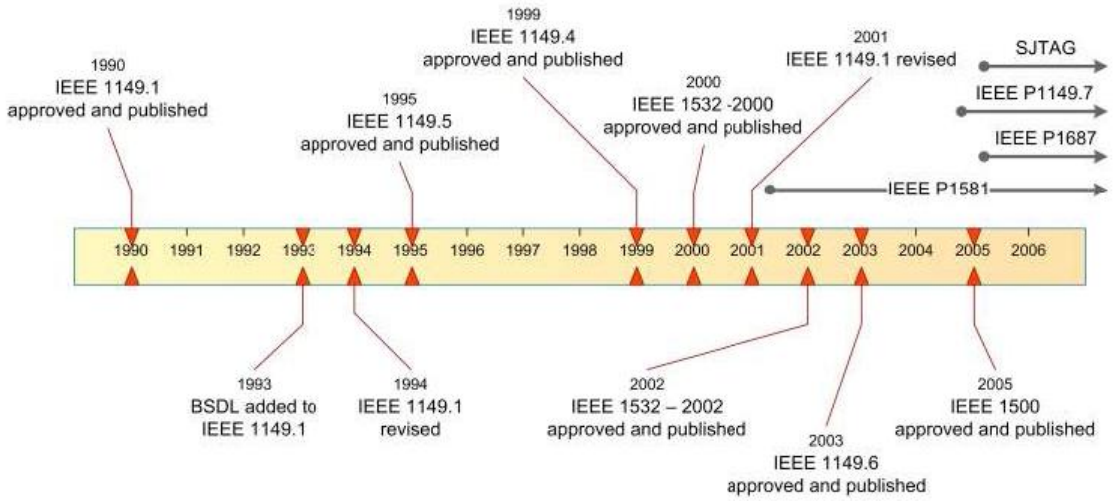


Figura 10 - Historiograma do *Boundary Scan*

A norma IEEE 1500 foi desenvolvida com influência do *Boundary Scan*, as duas normas possuem objectivos semelhantes, porem para diferentes níveis de integração. A norma IEEE 1500 estabelece uma linguagem de teste de núcleos CTL (*Core Test Language*), tem uma estrutura similar que visa estabelecer uma estratégia de teste estruturada para ICs contendo múltiplos núcleos.

A extensão da norma IEEE 1149.7, servirá para ajudar o desenvolvimento e depuração de *software*. Trás benefícios adicionais, tais como:

- A possibilidade de controlar o consumo de *debug* de forma normalizada, uma vez que oferece quatro modos com níveis de consumo diferentes.
- A possibilidade de acesso a um dispositivo específico quando se usa mais de um dispositivo encadeado.
- O suporte a uma tecnologia estrela, além da tradicional serial.
- Operação possível usando somente dois pinos além da tradicional com quatro.

A nova norma também pretende auxiliar o uso de módulos com múltiplos *chips* (ou seja, múltiplos *dies* encapsulados como um único IC) e *stacked-dies* (*dies* sobrepostos um em cima dos outros dentro do encapsulamento). [Asset09]

3.2. Aplicações do *boundary scan*

JTAG ou *Boundary Scan,* é uma infra-estrutura de teste versátil para usar em diversas aplicações, a mais óbvias para o *boundary scan* estão dentro do ambiente de produção. Aqui as placas podem ser testadas e os problemas, que poderiam não ser detectados devido à falta de acesso de teste, podem sê-lo agora adequadamente e atempadamente. A tecnologia *boundary scan*, está a ser combinada com outras tecnologias para fornecer novas aplicações (ex. programação ISP).

Além de ser utilizado em testes de produção o *Boundary scan*, IEEE1149.1, também pode ser usado numa variedade de cenários de teste, incluindo desenvolvimento de produto.

Um dos principais custos para o desenvolvimento de programas de teste é o custo com *software*, isto é, mais real para *boundary scan* onde o *hardware* é limitado a uma simples caixa (controlador). Isso significa, que todas as reduções feitas no tempo do desenvolvimento de *software* podem reduzir significativamente os custos. O seu código pode ser reutilizado para novos testes, portanto, o custo pode ser dividido ao longo destes projectos.

3.3. Motivação para o uso do *Boundary scan*

As novas tecnologias de desenvolvimento e produção dos produtos, tais como telemóveis e câmaras digitais, têm um ciclo de vida muito curto e o tempo de colocação no mercado, tem criado novas tendências tecnológicas. Estas tendências incluem o aumento da complexidade do dispositivo. Os componentes têm cada vez mais um encapsulamento pequeno e fino, usam a tecnologia de montagem à superfície (SMT), contribuindo para um aumento do número de pinos por ICs e uma menor área do PCB, como ilustrada na figura seguinte.

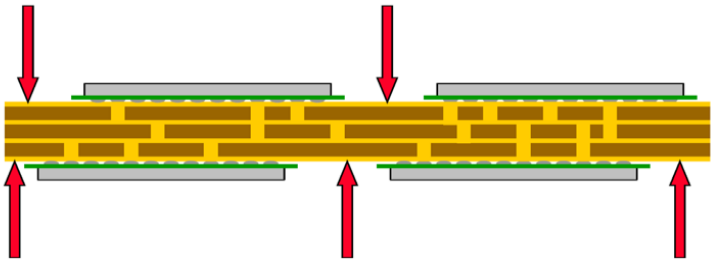


Figura 11 – Tecnologia SMT e *Multi-Layer*

As novas placas incluem componentes que são montados à superfície e em ambos os lados do PCB e com múltiplas camadas. As vias de passagem entre camadas e vias de ligação entre componentes são mais finas e inacessíveis.

A tecnologia *Boundary Scan* é a única solução de custo eficaz que pode solucionar estes problemas. Nos últimos anos, o número de dispositivos que incluem *Boundary Scan* têm crescido bastante. A maioria dos fabricantes estão a incorporar esta tecnologia teste nos ICs que colocam no mercado.

3.4. Célula *Boundary Scan*

O funcionamento da arquitectura BST está ilustrado na figura 12. Como podemos visualizar, esta consiste em associar a cada pino de I/O uma célula *Boundary Scan*.

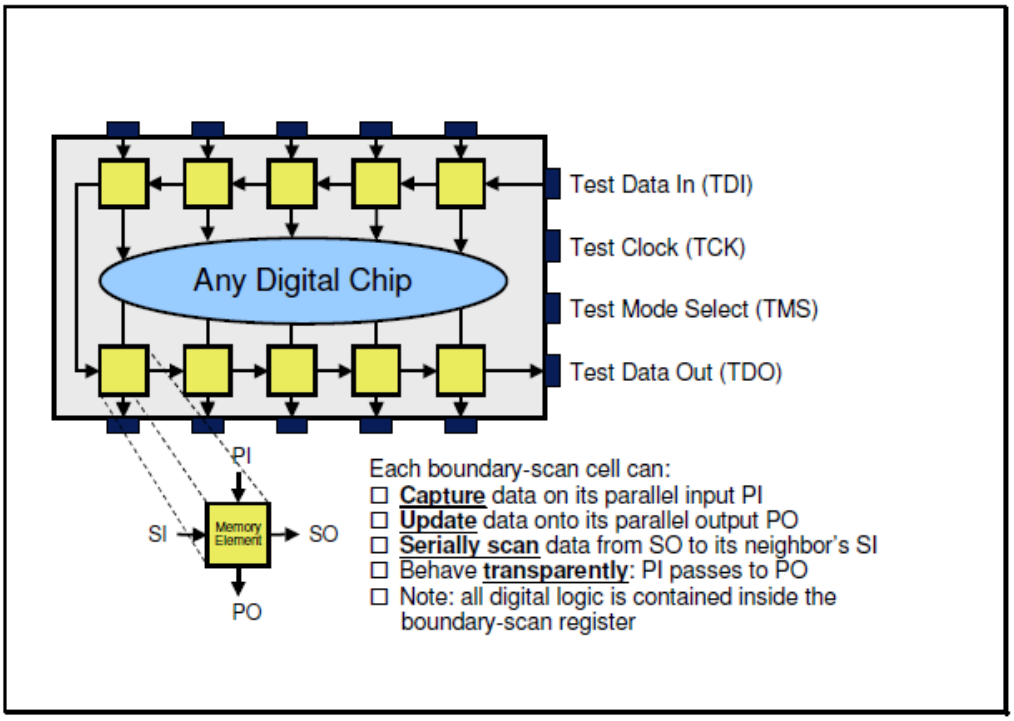


Figura 12 – Arquitectura do componente com infra-estrutura *Boundary-Scan*

As células BS têm como objectivo separar a lógica interna dos pinos do integrado, permitindo assim o acesso directo às entradas e saídas; ler e escrever nas mesmas independentemente da lógica interna e dos valores por esta imposta.

As Células *Boundary-scan*, podem ter duas funcionalidades: são utilizadas para testar a funcionalidade de um circuito integrado, teste interno *Intest*. Usando a Células *Boundary-scan* testamos a interligação entre dois circuitos integrados, designados de teste externo *Extest,* este é principal modo de aplicação, verifica se o circuito está em aberto ou se existe curtos entre os circuitos integrados, ou no próprio circuito impresso. [Asset00]

O *Intest* permite testar a operacionalidade de integrados depois de montados no PCB, permite também testar módulos que seriam de outra forma impossíveis de testar separadamente. Identifica defeitos, tais como; falta de componente ou posição do componente invertido (polaridade).

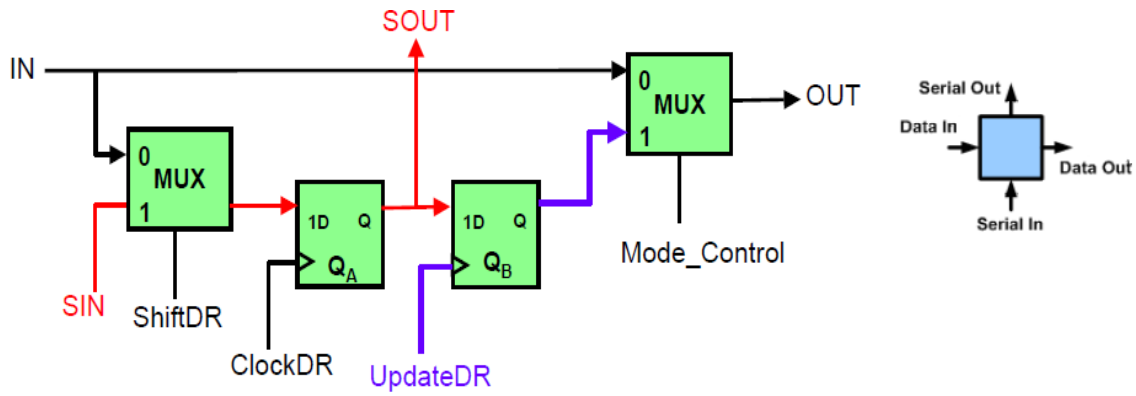


Figura 13 – Arquitectura interna da célula BST

Essencialmente, as Células *Boundary-scan* podem ser pensadas como "agulhas virtuais" comparando-as ao teste ICT.

A célula *Boundary scan* contém elementos de memória para capturar dados do circuito, pode capturar transições nos pinos AC (IEEE1149.6), carrega dados no circuito fazendo o deslocamento série desses dados para a próxima célula do IC.

As figuras seguintes, mostram os princípios básicos de uma célula, com os quatro modos de operação: Transparência, Controlabilidade, Observabilidade, Deslocamento.

A funcionalidade das células tem por objectivo proporcionar um controle total dos sinais presentes nos pinos do IC e a ela associados, de acordo com os seguintes requisitos:

- Transparência: sempre que não se pretendam efectuar operações de teste, a entrada paralela da célula BST deve estar ligada á saída paralela.

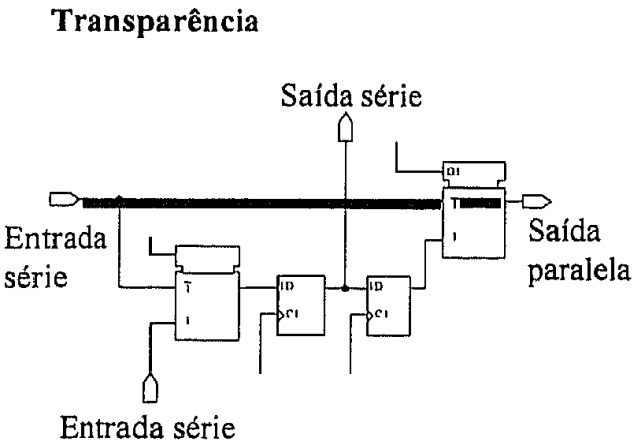


Figura 14 – Modo de operação da célula BST, Transparência

- Deslocamento (estímulos/respostas): deve ser possível promover uma operação de deslocamento série ao longo da cadeia formada pelo conjunto das células BST.

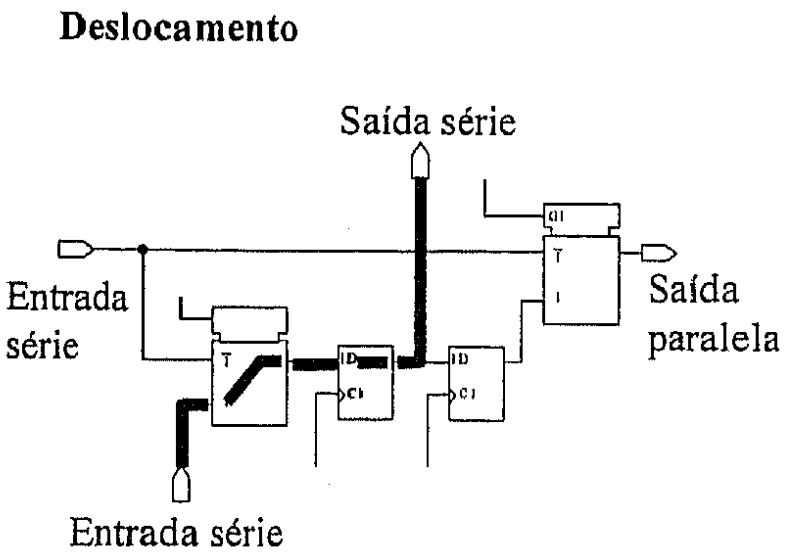


Figura 15 – Modo de operação da célula BST, Deslocamento

- Observabilidade: deve ser possível capturar o valor presente na entrada paralela da célula BST, com o objectivo de o deslocar para o exterior.

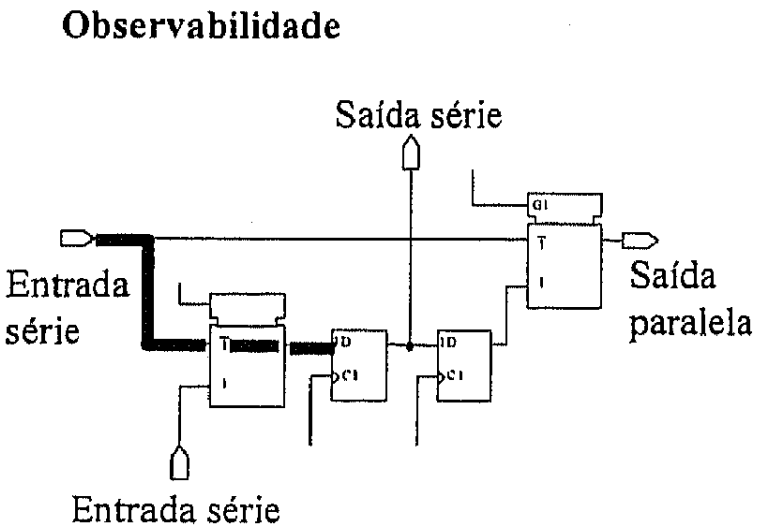


Figura 16 – Modo de operação da célula BST, Observabilidade

- Controlabilidade: deve ser possível forçar um valor na saída paralela, previamente inserido na célula através de uma operação de deslocamento, independentemente do valor presente na entrada paralela.

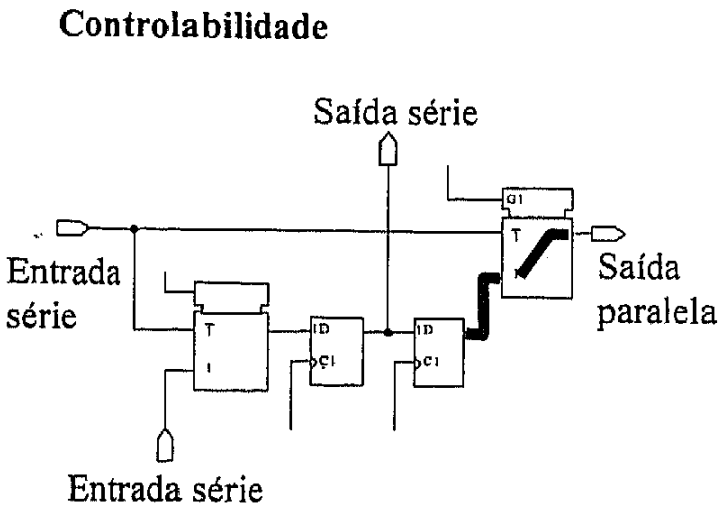


Figura 17 – Modo de operação da célula BST, Controlabilidade

Estas funcionalidade permitem o desacoplamento completo entra o exterior e a lógica interna do IC, possibilita que o teste das células do IC seja efectuada de forma cíclica, com modos de operações definidos:

- Deslocar para o interior da cadeia BST os valores lógicos que pretendemos aplicar às ligações (deslocamento)
- Aplicar às ligações os valores presentes no interior das células BST associadas a pinos de saída (controlabilidade). Aplicar também os valores presentes nos canais de teste associados aos nós da célula do IC a que exista acesso paralelo.
- Capturar os valores presentes nas ligações para o interior das células BST associadas aos pinos de entrada (observabilidade). Capturar também os valores presentes nos canais de teste associados aos nós do acesso paralelo.
- Deslocar para o exterior da cadeia BST os valores capturados (deslocamento). [Ferreira92]

3.5. Infra-estrutura BST num Circuito Integrado

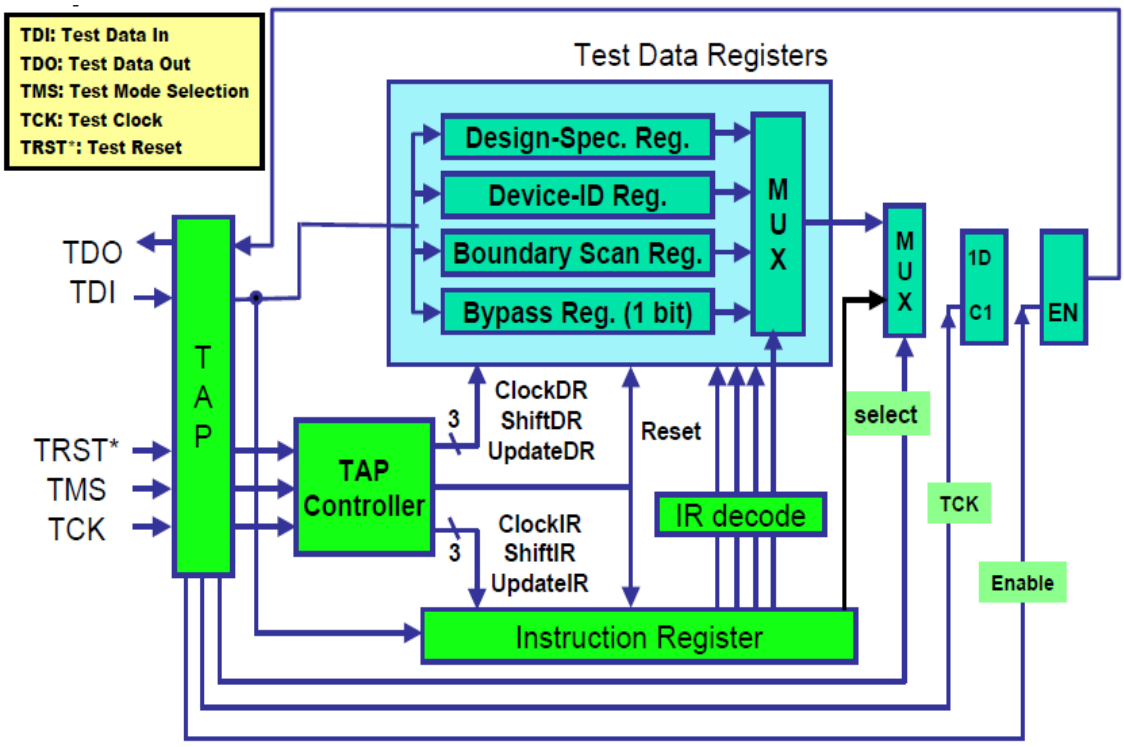


Figura 18 – Infra-estrurura *Boundary Scan*

A Figura mostra:

- Os dados do TDI e TDO passam por dois *multiplexers*, um N:1 "*Data Multiplexer*" (Data mux) e um 2:1 "*Data/Instruction Multiplexer*" (Data/instr mux)
- O *multiplexer* 2:1, é controlado pelo controlador TAP; este decide se os dados à saída vêm do registo de instrução ou de um dos quatro registos de dados.
- O multiplexer N:1, é controlado pelo registo de instruções; coloca na saída um dos quatro registos de dados (*BST Register, User Data Register, Identification Register, BP Register*), para posteriormente serem ou não colocados no registo TDO pelo multiplexer 2:1. [Texas07]

Arquitectura *Boundary Scan*:

- Test Access Port (TAP)
- Controlador TAP
- Registos de Instrução
- Registos de dados de teste

Bypass register – Este registo tem a dimensão de um bit, os dados do TDI passam através deste registo e são enviados para o TDO com um único atraso de ciclo de relógio.

Boundary-scan Register - Permite a detecção de problemas nas placas, como curtos, circuito abertos, …. Também permite acesso para os pinos de entrada e saída dos componentes.

Design Specific Test data Register - Este registo permite aceder às características de teste do componente, tais como BIST e *internal scan paths,* sendo opcional.

Device identification - Permite a identificação dos dispositivos na placa.

O *User Data Register* (opcional), fornece a possibilidade de os utilizadores integrarem os seus próprios registos de teste.

Os registos de instrução e de dados devem ser paralelos e terem entradas e saídas comuns. A escolha entre os registos de instrução e dados é feita através do controlador TAP.

A norma IEEE 1149.1, obriga a que os componentes que utilizem BST incorporado tenham mais 4 pinos: TDI, TDO, TMS e TCK. Pode ainda conter um sinal adicional: TRST (*Test Reset*).

TCK (*Test Clock*)

- Relógio dedicado, independente do relógio do sistema;
- A frequência do relógio deve ser suportada pelos componentes que compõem o sistema de teste.
- Os chamados "*Stored-state devices*" constituídos por *flip-flop* e *latches*, devem memorizar o valor quando o relógio estiver em zero;
- O driver de relógio deve suportar a carga;

TDI (*Test Data Input*)

- Os sinais são amostrados no impulso de subida do relógio;
- Recomendação de pull-up, pois o driver não pode ficar a flutuar, mas sim deve manter lógica '1'.

TDO (*Test Data Output*)

- Os sinais são amostrados no impulso de descida do relógio;
- Deve estar inactivo quando nenhum dado estiver a ser lido, para permitir conexões paralelas ao nível da placa.

TMS (*Test Mode Select*)

- Os sinais são amostrados no impulso de subida do relógio;
- Recomendação de pull-up, pois o driver não pode ficar a flutuar, mas deve manter lógica '1';
- O driver deve suportar a carga.

TRST (*Test Reset*)

- Inicialização assíncrona do controlador TAP;
- Activo a '0';
- Um pull-up é recomendado;

TMS deve estar a alto quando o sinal do TRST mudar de '0' para '1'.

Estes sinais encontram-se ligados internamente com circuitos lógicos que permitem executar os testes pretendidos.

O acesso e controlo da infra-estrutura BS de um componente é efectuado aplicando sequências de bits no pino TMS.

Cada acesso (teste) efectuado realiza-se segundo a sequência dos seguintes passos:

1. O controlador TAP coloca o *multiplexer* 2:1 de modo a que os dados na saída sejam provenientes do registo de instruções.
2. Ao introduzir a instrução pretendida, o registo de instrução decide de que modo deve operar o *multiplexer* N:1, ligando à sua saída um dos quatro registos de dados anteriormente referidos.
3. O controlador TAP, em conjunto com o *multiplexer* 2:1, coloca na saída o registo de dados.
4. A partir deste ponto, podem ser introduzidos os vectores (dados) de teste e analisadas as respostas.

Registos de dados

O *multiplexer* N:1 (*Data mux*) representado na figura, permite a escolha de um dos quatro registos de dados, *BST Register, User Data Register, Identification Register, Bypass.*

Esta escolha é feita de acordo com os dados do registo de instruções.

3.6. Controlador TAP

O controlador TAP é uma máquina de estados finita (16 estados) que responde por variações nos sinais de TCK e TMS.

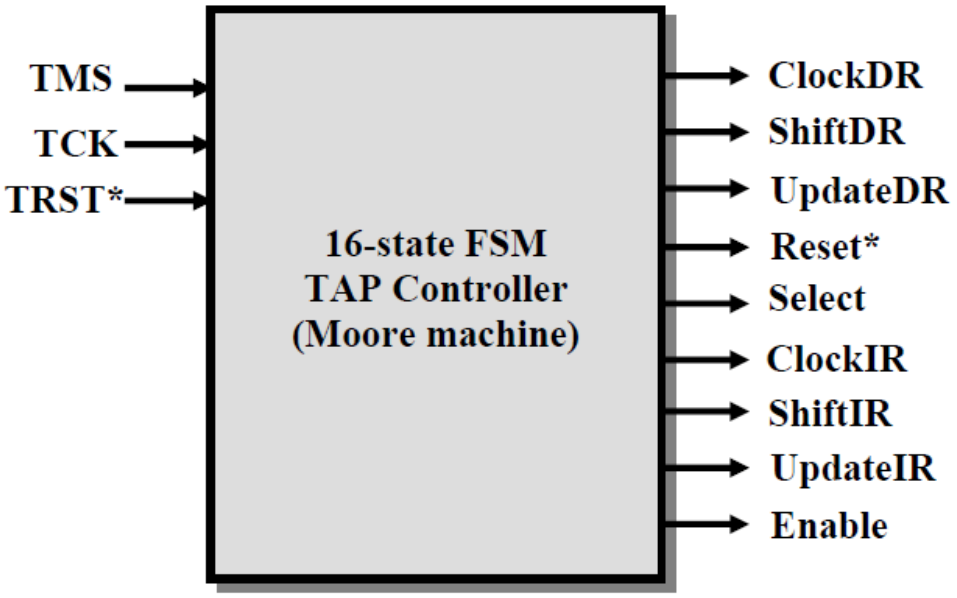


Figura 19 – Controlador TAP

Esta máquina controla os estados do circuito de teste. A transição de estado ocorre na transição ascendente do TCK, a saída do controlador e os valores variam na transição descendente do TCK.

A linha do TMS gera vários sinais de controlo interno, incluindo o sinal de controlo para o *multiplexer* de selecção entre registos de instruções ou registo de dados.

Como anteriormente visto, na arquitectura de um dispositivo com BST existe um módulo designado por controlador do TAP, como se pode ver na Figura n.º 14. Este módulo é uma máquina de estados que controla o *multiplexer* 2:1 (*Data/instruction multiplexer*) e gera os sinais de controlo que definem quais os registos de dados a usar.

Cada uma das operações da máquina de estados pode ser executada, tanto no registo de dados, como no registo de instruções; este facto leva a que, no diagrama de estados representado na Figura nº 15, se vejam dois ramos idênticos

3.6.1. Diagrama de estados do controlador TAP

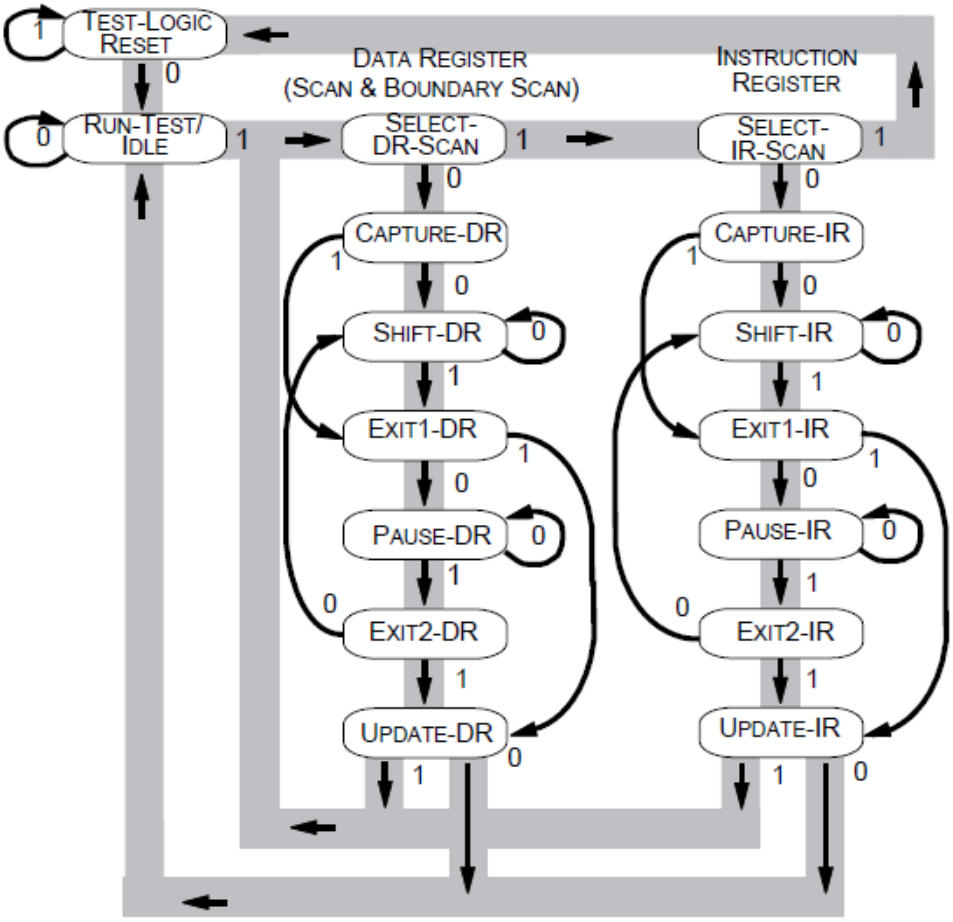


Figura 20 – Diagrama de estados Controlador TAP

A máquina de estados é controlada a partir da entrada TMS (*Test Mode Select*) que indicam que operação deve executar:

· Capturar os dados das portas I/O para registo BST.

· Introduzir dados no registo BST.

· Aplicar dados às portas I/O do registo BST.

Estes três estados, considerados principais, encontram-se relacionados, na Figura n.º 15, com os estados *Capture* (DR e IR), *Shift* (DR e IR) e *Update* (DR e IR) respectivamente.

O ramo da esquerda é relativo ao registo de dados, e o da direita é relativo ao registo de instruções.

Os dezasseis estados estão divididos em três partes:

A primeira parte contém os estados *Reset* e *Idle*, a segunda e terceira partes controlam as operações dos dados e dos registos de instruções, dado que a única diferença entre a segunda e a terceira partes estão nos registos que tratam, o procedimento é o mesmo.

1. ***Test-Logic-Reset*:** Neste estado, o circuito *BS* está desactivado e o sistema está na sua função normal. Sempre que um sinal de *Reset* é aplicado ao circuito *BS*, também volta a este estado. Independentemente do estado que o controlador TAP está, ele vai voltar a este estado se existirem cinco 1's consecutivos, aplicados através de TMS ao controlador TAP.
2. ***Run-Test/Idle:*** Este estado é o que permite correr as instruções opcionais como a RUNBIST.
3. ***Select (DR ou IR):*** Permite seleccionar o tipo de registo que se vai colocar entre a entrada TDI e a saída TDO, se um registo de dados (DR) ou o registo de instrução (IR)
4. ***Capture-DR:*** Neste estado, os dados podem ser carregados em paralelo com os dados de registos, seleccionados pela instrução actual.
5. ***Shift-DR:*** Neste estado, os dados de teste são deslocados em série através dos registos de dados, seleccionados pela instrução actual. O controlador TAP pode ficar neste estado, enquanto o TMS for igual a '0'. Para cada ciclo de relógio, um bit de dados é deslocado através de um vector de dados para o TDI (TDO).
6. ***Exit1-DR:*** Todos os carregamentos paralelos do estado *"Capture-DR"* ou transferidos a partir do estado *"Shift-DR",* esses dados são mantidos no registo de dados do mesmo.
7. ***Pause-DR:*** A sua função é a de esperar algumas operações externas.
8. ***Exit2-DR:*** Este estado representa o fim da operação *"Pause-DR"* e permite que o controlador TAP volte ao estado *"Shift-DR,* para esperar mais dados para o deslocamento.
9. ***Update-DR:*** Os dados de teste, armazenado no primeiro estado das células *BS,* é carregado para a segunda fase neste estado. [Kharagpur]

3.7. Sequência de operações do controlador TAP

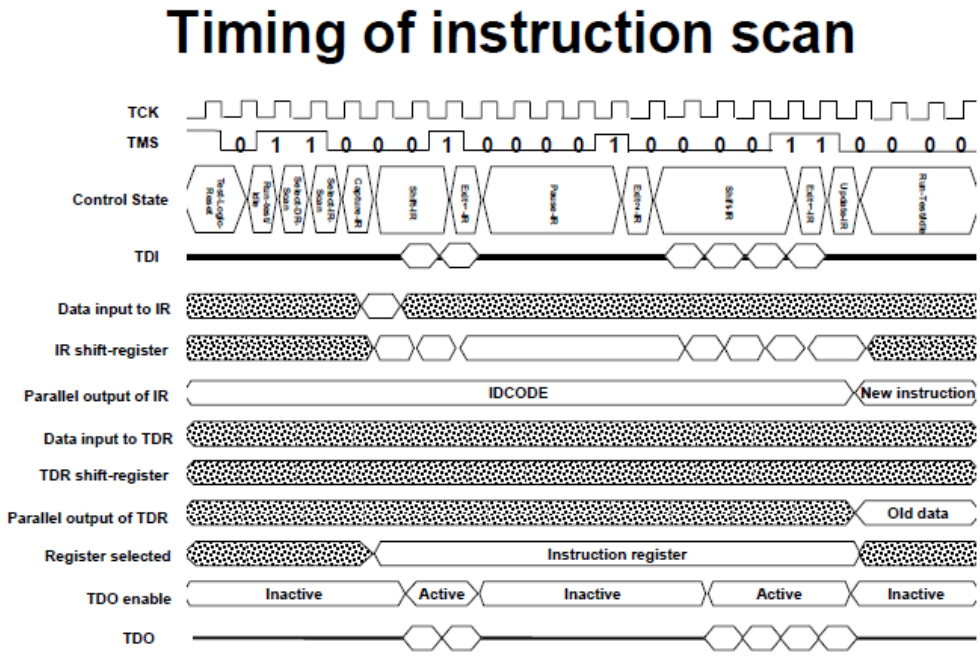


Figura 21 – Procedimento para definir modo de teste no Registo de Instruções

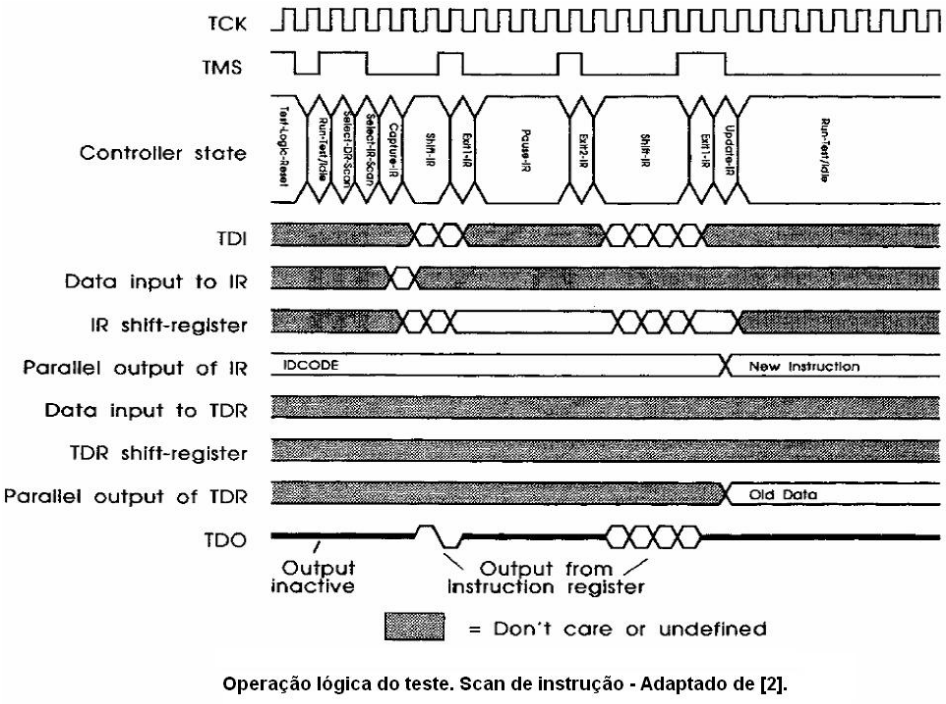


Figura 22 – Procedimento de envio de dados no Registo de dados

Exemplo de procedimento de BST num IC, pode ser efectuado através da seguinte sequência de operações:

- Colocação do controlador TAP no estado "*Shift-IR"*
- Deslocação da instrução que selecciona o registo/modo de funcionamento que pretendemos
- Colocação do controlador TAP no estado "*Shift-DR"*
- Acesso ao registo de dados e deslocamento de bits para o mesmo registo.

Exemplo da sequência no modo *External Test,* esta faz-se através da execução cíclica do conjunto de operações a seguir mencionadas:

- Deslocamento, para o interior da cadeia BST, dos valores lógicos a aplicar nas saídas paralelas das células (só no estado "S*hiftDR"*)
- Aplicação, nas saídas paralelas, dos valores deslocados para as células BST (estado "*UpdateDR"*)
- Captura dos valores lógicos presentes nas entradas paralelas da célula BST (estado "*CaptureDR"*)
- O deslocamento dos vectores capturados nas células BST do exterior, em simultâneo com o deslocamento do novo vector a aplicar.

3.8. Registos de Dados definidos pela norma 1149.1

BST Register- A função deste registo é aplicar vectores de teste e recolher (analisar) as suas respostas, passando em série por todas as células BST e captando ou aplicando os valores pretendidos.

Bypass Register- Tem como função encurtar o registo BST, permitindo passar por uma célula BST, sem captar ou aplicar valores.

Identification Register- Este registo é opcional. A sua função é fornecer ao utilizador a identificação do dispositivo.

User Data Register- Estes registos são opcionais. Fornecem a possibilidade aos utilizadores de integrarem os seus próprios registos de teste. [Mentor06]

3.9. Registo de Instruções

A norma IEEE 1149.1 define um conjunto de instruções obrigatórias e opcionais que permitem a selecção do registo de dados a ser usado.

De acordo com a norma existem três instruções obrigatórias: *EXTEST, SAMPLE/PRELOAD, e BYPASS.*

3.9.1. EXTEST

A instrução *EXTEST* selecciona o registo BS e impõe o modo de controlabilidade em cada célula BS, desacoplando a lógica interna dos pinos. A configuração para cada célula BS tanto permite também a realização de testes à lógica interna do IC, já que as células associadas aos pinos de entrada podem ser usadas para aplicar os vectores de teste e as que estão associadas aos pinos de saída usadas para capturar as respostas. [Ferreira99]

O código da instrução *EXTEST* está pré-definido, e corresponde a colocar em '0' todos os bits do registo de instrução.

Com este comando, o *Boundary Scan Register* é ligado entre o TDI e o TDO.

Exemplo de um *External Test* a um circuito electrónico

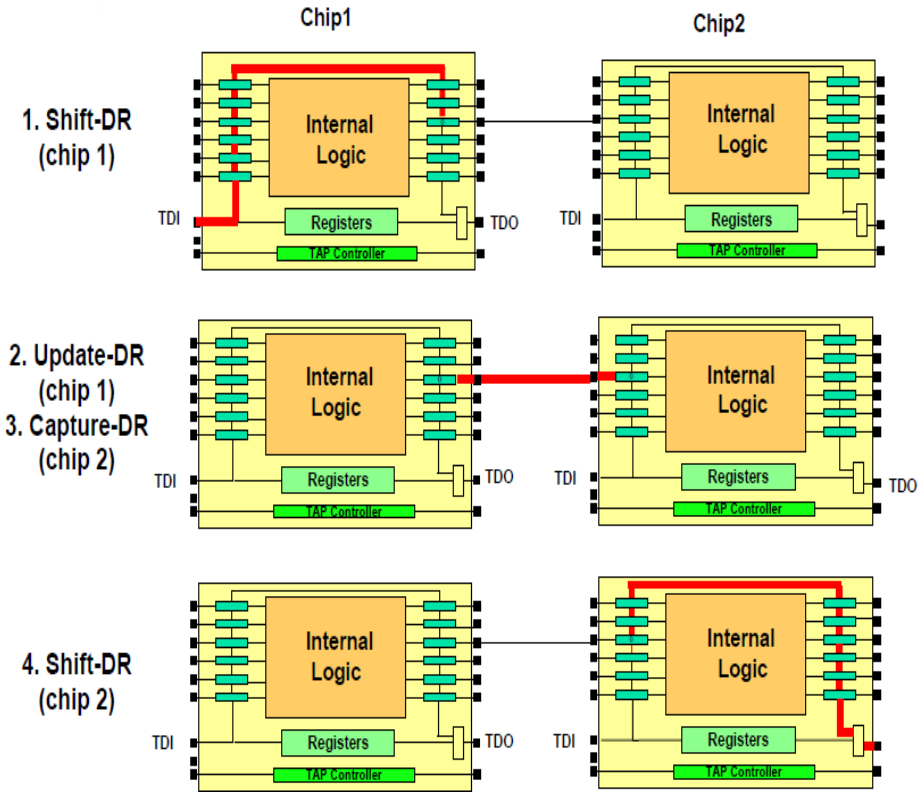


Figura 23 – *External Test*

Diagrama de blocos/instruções internas

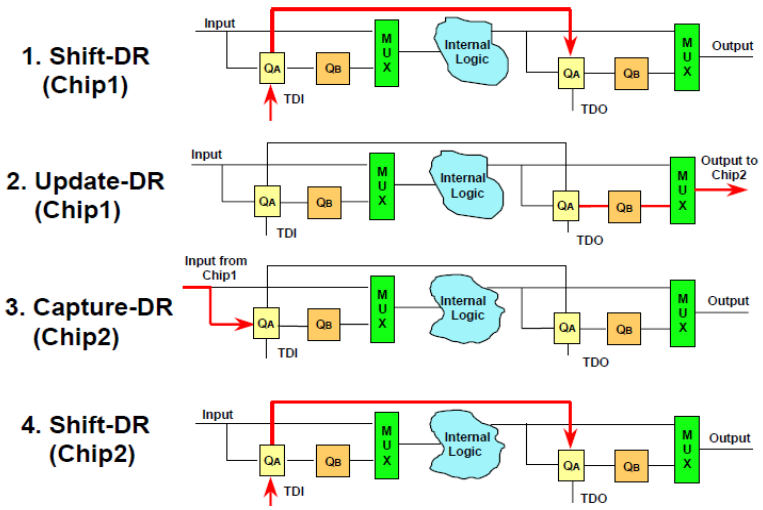


Figura 24 – Sequência interna do *External Test*

Permite capturar os dados presentes nos pinos de entrada do IC (estado *CaptureDR*) no registo *Boundary Scan* para posterior comprovação (estado *ShiftDR*).

Também permite aplicar os dados presentes no registo auxiliar (estado *UpdateDR*) aos terminais de saída.

No estado *ShiftDR* carregam-se novos dados que se aplicaram aos terminais de saída no próximo estado *UpdateDR*. Permite assim controlar outros circuitos do circuito, como as interligações da placa de circuito impresso.

3.9.2. SAMPLE/PRELOAD

A instrução *SAMPLE / PRELOAD* também selecciona o registo BS, mas agora com as células BS em modo transparente. O principal objectivo desta instrução consiste em permitir que um primeiro vector seja deslocado para o interior dos registos BS, antes de se promover o desacoplamento entre a lógica interna e os pinos (enquanto os andares de retenção não forem carregados com o primeiro vector a aplicar, poderá ser necessário impedir que o seu conteúdo inicial surja nas saídas paralelas das células). Esta instrução não tem código pré-definido. [Ferreira99]

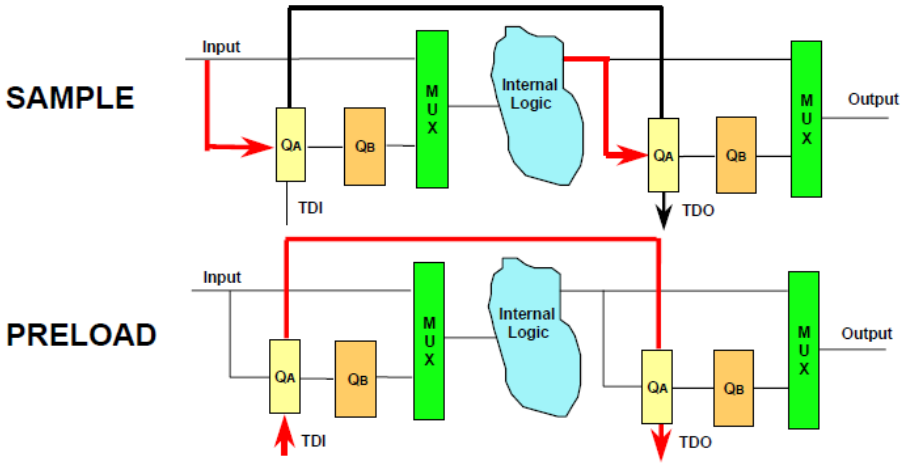


Figura 25 – Sample/Preload

Esta instrução não executa a separação dos pinos da lógica interna, pelo que o circuito poderá continuar a funcionar em modo normal. Permite recolher os dados no terminal de entrada, no estado *CaptureDR* e no registo *Boundary Scan* para posterior comprovação no estado *ShiftDR*. Também permite introduzir novos dados no registo auxiliar, no estado *UpdateDR* antes de executar nova instrução. Não deve afectar o comportamento normal do circuito porque o *"*Modo*"* deve valer '0'.

Esta instrução facilita a verificação dos outros circuitos, bem como interligações do circuito impresso.

3.9.3. BYPASS

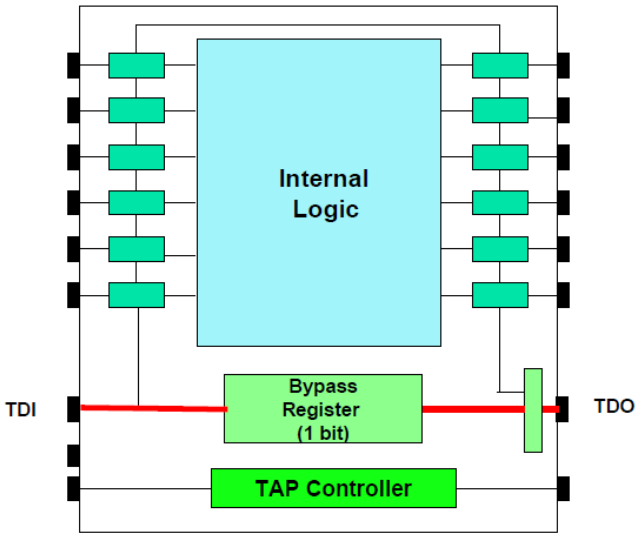


Figura 26 - Bypass

A instrução *BYPASS* selecciona o registo com o mesmo nome (*Bypass Register*), garantindo um percurso com comprimento mínimo entre TDI e TDO (vantajoso no caso

dos componentes que não desempenhem qualquer função na operação de teste pretendida). As linhas do TDI e TDO são ligadas passando por um único *flip-flop* (passa do TDI para o TDO com um único atraso relógio). O código da instrução *BYPASS* está pré-definido, corresponde a colocar em '1' todos os bits do registo de instrução e é automaticamente carregado quando se reinicializam os componentes. Neste modo de funcionamento todas as células BS estarão em modo transparente, o que significa que a infra-estrutura de teste fica inactiva quando se reinicializam os componentes. [Ferreira99]

A norma IEEE 1149.1 define ainda um conjunto de instruções opcionais, tais como:

INTEST, IDCODE, HIGHZ, CLAMP e *RUNBIST*.

3.9.4. *INTEST*

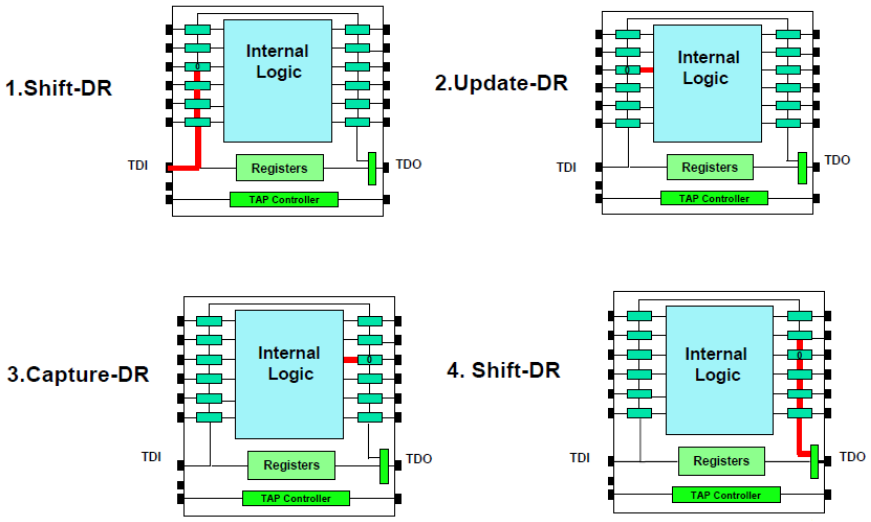


Figura 27 – Intest

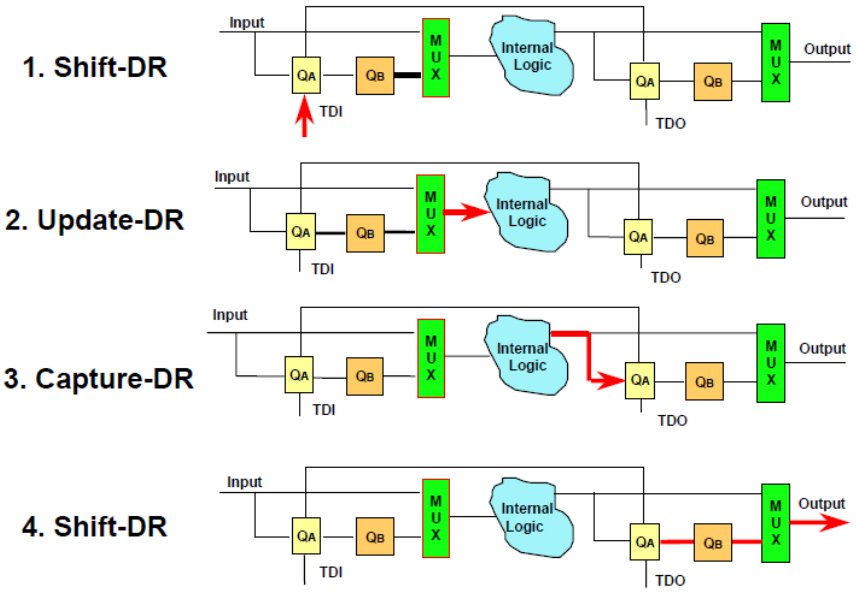


Figura 28 – Sequência interna do *Internal Test*

Com este comando o registo do *Boundary Scan* é ligado entre o TDI e os sinais de TDO, os sinais internos do núcleo do componente são recolhidos pelas células BSR no estado *CaptureDR.*

O conteúdo do registo BSR é deslocado através da linha de TDO no estado *UpdateDR.* Depois da passagem pelo núcleo do componente, os dados são recolhidos no estado *CaptureDR* e enviados para o TDO no estado *UpdateDR,* como podemos constatar na figura.

3.9.5. IDCODE

Esta instrução selecciona o "*Identification Register*", possibilita a leitura de um código de 32 bits, dando a conhecer ao utilizador o ID do dispositivo (código do fabricante, número da peça e código de revisão).

3.9.6. HIGHZ

Selecciona o registo "*bypass*", o componente fica em "*tri-state*" nas três saídas de estado.

3.9.7. CLAMP

Selecciona o registo "*bypass*", o componente define os *"outputs"* num determinado estado lógico.

3.9.8. RUNBIST

Esta instrução é opcional, mas cada vez mais utilizada devido à crescente importância de estruturas de auto-teste internas e é essencialmente o resultado de um registo de auto-teste entre TDI e TDO. [Mentor06]

Consiste em aplicar um número pré-definido de impulsos no relógio (TCK), mantendo a linha TMS a '0'. Esta operação destina-se a permitir a execução da instrução de auto_teste.

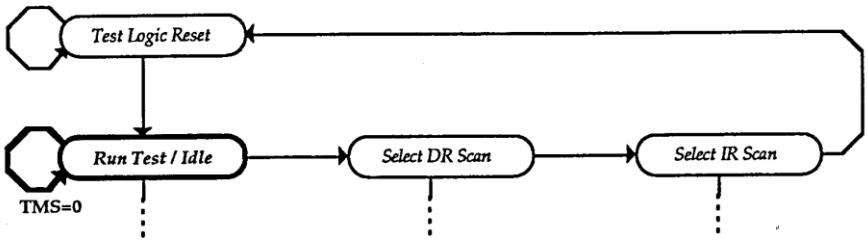


Figura 29 - Diagrama de sequências definido para auto-teste

No final do ciclo de auto-teste, obtém o resultado *Pass / Fail*.

Nenhuma das instruções opcionais tem código pré-definido. A definição das instruções opcionais a suportar é naturalmente da responsabilidade do projectista.

3.10. Tipo/formas de Controladores de teste em placas

Existem três formas de interligar os módulos TAP, estas observam-se nas seguintes figuras.

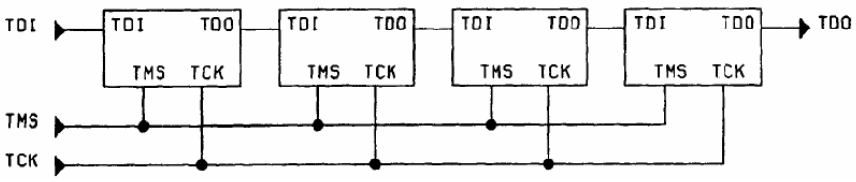


Figura 30 - Controlador BS com estrutura série

A figura mostra uma cadeia de módulos ligados em série, este tem uma única entrada TDI e as saídas TDO dos módulos passam de uns para os outros, todos estão ligados às mesmas entradas TMS e TCK, assim todas as máquinas de estados encontram-se no mesmo estado ao mesmo tempo.

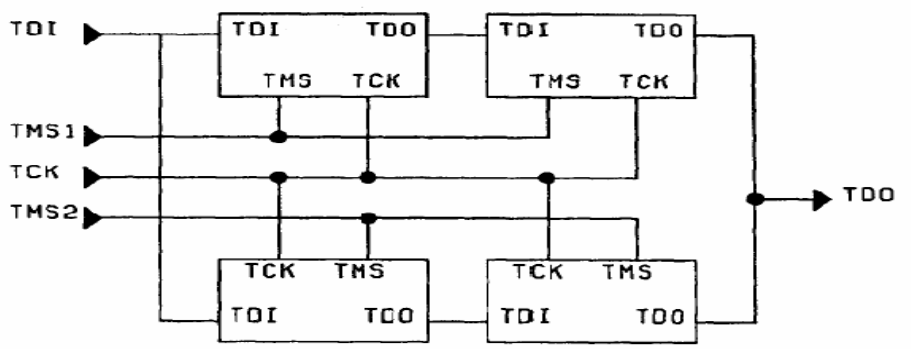


Figura 31 - Controlador BS com estrutura paralela-série

Nesta figura mostra uma conexão paralelo-série, neste caso os módulos em regime de estados misto, pois partilham as entradas TMS e TCK, o TDO é interligado e a máquina de estados estão em diferentes estados.

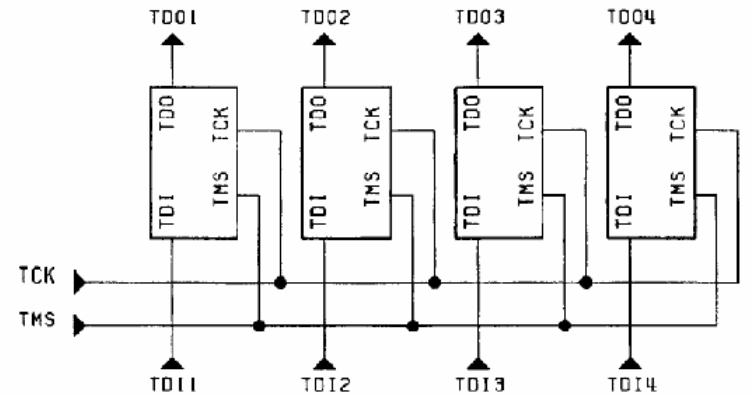


Figura 32 – Conexões com múltiplos módulos com TMS e TCK comuns

Nesta figura os controladores partilham os sinais TMS e TCK, indica que os módulos estão a operar no mesmo estado e as saídas TDO são separadas e activadas em simultâneo. [IEEE01]

3.11. Detecção de falhas

Este diagrama mostra três ensaios consecutivos aplicados às redes de 1-4. O primeiro teste é na vertical “1110”, o segundo é “0101”, e a terceira é “1001”.

Podemos verificar os dados na horizontal, isto é, a sequência de “101” aplicada à rede um e assim sucessivamente.

Inicialmente *Kautz* mostrou que, para obter condição suficiente para detectar qualquer curto-circuito, interrupções no PCB ou soldas frias, os códigos na horizontal devem ser únicos para cada rede. Isto significa que o número total de bits em cada código (o número de testes) é dado pela fórmula log2 (N), onde N é o número de redes. [Asset00]

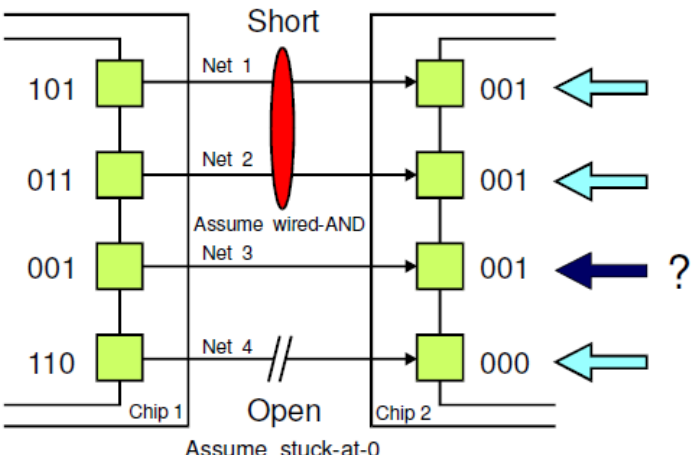


Figura 33 – Simulação de falhas

Na Figura acima, visualizamos que os códigos de sinais na horizontal, são construídos a partir dos três ensaios diferentes na vertical.

Na resposta captada, os códigos de envio e recepção são diferentes nas redes 1 e 2, devido ao curto-circuito existente entre ambas.

Com quatro redes; log2 (N) é 2 e a cada rede pode ser atribuído um código único, dois bits.

O que iria acontecer é que no código horizontal numa ou mais redes os valores lógicos na vertical não se iriam alternar de ‘0’ para ‘1’ ou vice-versa, iríamos ter “11” ou “00”. Não teríamos a possibilidade de retirar conclusões.

Então, significa que o número total de bits em cada um dos códigos, para satisfazer a condição de somente termos um único código variável, a fórmula terá que ser log2 (N +2), isto resulta num código de três bits para as quatro redes representadas na Figura.

Os códigos de resposta das redes 1 e 2, são diferentes dos respectivos códigos injectados à entrada, mas ambos têm o mesmo código (001).

A partir desta informação, deduzimos que há uma falha com um curto-circuito entre as redes 1 e 2 e o curto-circuito é do tipo "*AND*".

Neste caso especifico, o nosso diagnóstico pode não ser totalmente correcto porque a rede 3 tem o mesmo código e não está em curto; são situações que podem aparecer e o programador necessita estar atento.

Para anularmos ou diminuirmos este problema é essencial a inserção de mais uma linha de códigos, basicamente um quarto teste.

Na Figura 26, podemos ver a nova configuração entre sinais enviados e recebidos.

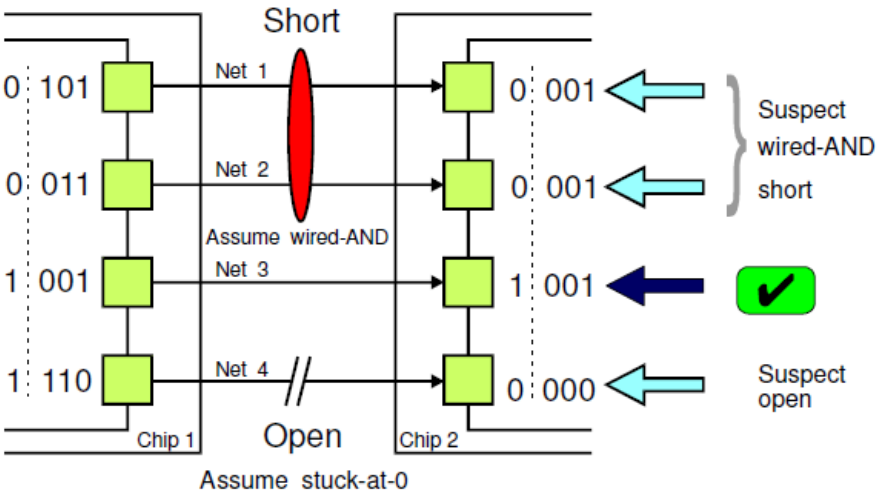


Figura 34 – Simulação de falhas

Na rede 4, verificamos que o circuito está em aberto, logo a entrada do "receptor" vai ficar no ar e vai ter um estado indefinido. Aqui o resultado é "000", mas poderia ser outras combinações. Para eliminar este problema, aconselha-se sempre que possível a utilização de resistências de *pull-up*.

3.12. Outras aplicações

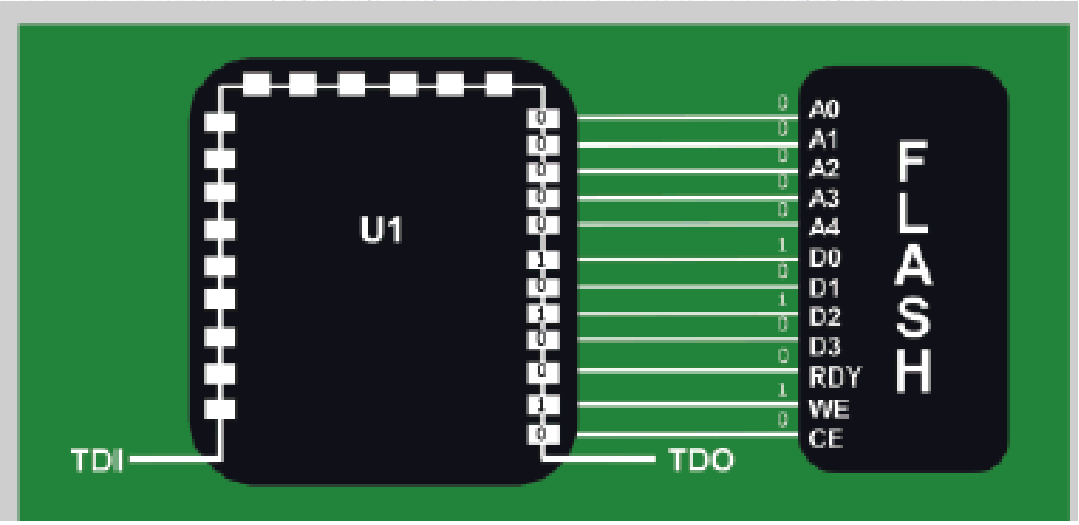


Figura 35 – *In-System Programming*

A utilização da tecnologia *boundary-scan,* permite aos utilizadores criarem aplicações de programação usando o método ISP (*In-system Programming*). Permite a programação de quase todos os tipos de CPLDs e memórias flash na placa após a montagem no PCB, independentemente do tamanho ou tipo de corpo.

Este sistema de programação, economiza custo e aumenta o rendimento, reduzindo o manuseamento das placas e respectivos componentes, simplificando também o processo de controlo (fluxo) da própria linha de montagem.

In System Programming

O uso do ISP para a programação ajuda na fase final do processo de fabricação de um produto, onde o componente já está soldado na placa de circuito impresso, este método pode ser usado também na fase de desenvolvimento, como uma forma rápida de gravar o programa sem a necessidade de retirar o IC do protótipo.

Os ICs tem de possuir um *bootloader* gravado de fábrica e este, está localizado na memória de programa. Este *bootloader* possui rotinas de baixo nível para auxiliar o processo de gravação do IC (apagar, programar, etc.).

Existem três maneiras de fazer o IC entrar no modo ISP:

1. *Status bit* e vector de *boot* (condição padrão em *power-up* inicial);
2. Detecção de um *break reset*;
3. Impulsos no pino de *reset* após *power-up* (activação por *hardware*).

Exemplo de introdução do programa:

1. Inicia o VDD e RST do IC a ser gravado em ‘0’;
2. Gera-se impulsos no VDD e RST. Neste momento o IC estará no modo ISP, pronto para receber o programa (HEX);
3. Após a gravação do programa, reinicia-se o IC, fazendo com que o programa gravado corra;

3.13. BSDL (*Boundary Scan Description Language*)

O BSDL (Boundary Scan Description Language) foi aprovado com a designação de norma IEEE 1149.1b; a norma original do *Boundary Scan* foi a IEEE 1149.1a. Compatível com a linguagem VHDL, esta reduz muito o esforço de incorporar *Boundary Scan* a um componente e, portanto, é bastante útil quando um projectista precisa de implementar uma infra-estrutura *Boundary Scan* de acordo com requisitos específicos. Para o controlador TAP e o registo *Bypass*, o projectista não necessita de os descrever pois podem ser gerados automaticamente. O projectista, só tem de descrever as especificações relativas ao seu próprio projecto, tais como o comprimento do registo BS, definição das instruções BS, elaboração do descodificador para as suas próprias instruções, atribuição dos pinos I/O. Muitas ferramentas de CAD já implementam todo o processo BS e, portanto, não é necessário para o projectista escrever um arquivo BSDL: as ferramentas podem gerar automaticamente o circuito BS necessário para verificar qualquer projecto quando as I/O do projecto são especificados. [Agilent10] [Rodriguez07]

Qualquer fabricante de um dispositivo compatível com JTAG deve fornecer um arquivo BSDL do mesmo. O arquivo BSDL contém informações sobre a função de cada um dos pinos do dispositivo usados como I/O, alimentação e massa. O arquivo BSDL inclui:

1. ***Entity Declaration*:** A declaração da *entity* é construida em VHDL e é usada para identificar o nome do dispositivo descrito pelo arquivo BSDL.
2. ***Generic Parameter***: O *Generic parameter* especifica qual é o *package* descrito pelo arquivo BSDL.
3. ***Logical Port Description***: Descrição de todos os pinos num dispositivo; indica se o pino é uma entrada (*in bit*;), saida (*out bit*;), bidirecional (*inout bit*;) ou indisponível para boundary scan (*linkage bit*;).
4. ***Package Pin Mapping***: O *Package Pin Mapping* mostra como os pinos internos do dispositivo são ligados aos pinos externos no package do dispositivo.
5. ***Use statements***: Usa declarações de chamadas aos *packages* VHDL que contêm atributos; tipos, constantes, …, que são referenciados no Arquivo BSDL.
6. ***Scan Port Identification***: Identificação dos pinos JTAG: TDI, TDO, TMS, TCK e TRST (se utilizado).

7. ***TAP description***: Fornece informações adicionais sobre a lógica do dispositivo JTAG; o comprimento do registo de instrução, *IDCODE*, …. Estas características são específicas de cada dispositivo.
8. ***Boundary Register description***: Fornece a estrutura das células *Boundary Scan* no dispositivo. Cada pino num dispositivo pode ter até três células *Boundary Scan*, cada célula consiste de um *register* e uma *latch*.

Arquitectura do núcleo do componente

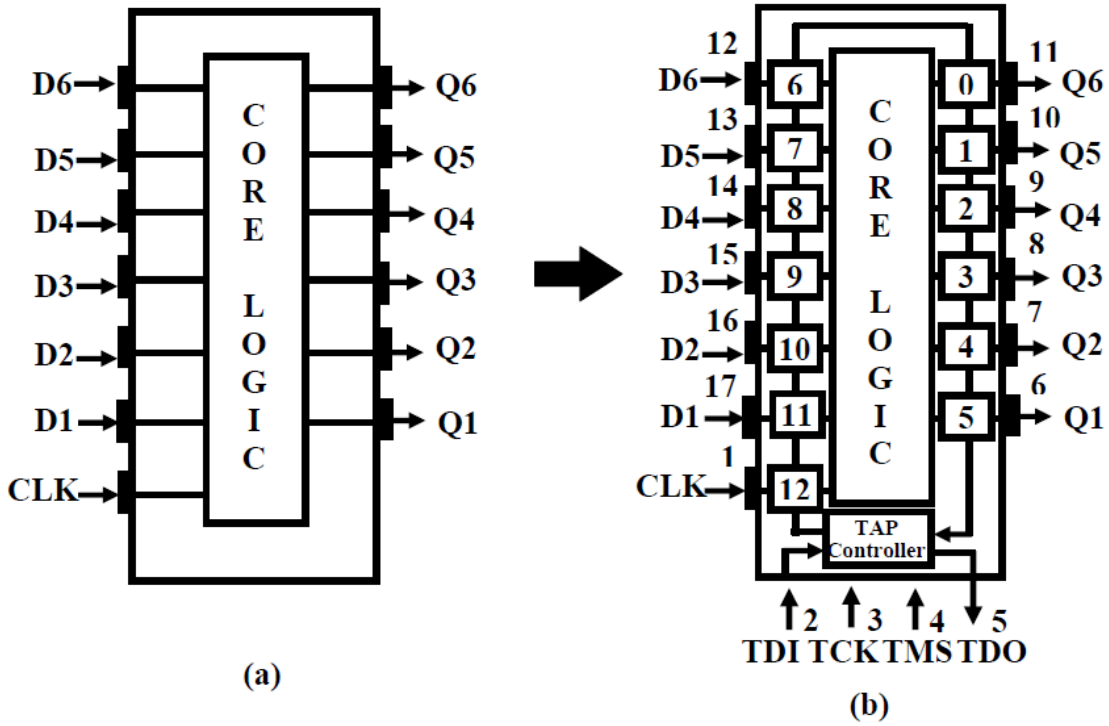


Figura 36 - Exemplo para ilustrar a lógica do núcleo (b) após a inserção do BS

A arquitectura BS aumenta a complexidade do componente; há necessidade de introduzir mais pinos, consequentemente aumenta o tamanho do corpo e aumenta o seu custo.

3.14. Exemplo *Boundary Scan*

Na tabela abaixo descrita, iremos visualizar o número de portas lógicas adicionais necessárias para Implementar uma infra-estrutura BST num determinado circuito. Esta pode fornecer uma estimativa geral do tamanho necessário para testar qualquer circuito implementando a norma IEEE 1149.1. O exemplo usa um componente com um encapsulamento de 40 pinos.

Logic Element	**Gate Equivalent**
Variable Size	
Boundary-scan Register (40 cells)	680 Approx
Fixed Sizes	
TAP controller	131
Instruction Register (2 bits)	28
Bypass Register	9
Miscellaneous Logic	20 Approx
Total	**868 Approx**

Tabela 2 – Portas lógicas necessárias à implementação do BS

Para a implementação do *Boundary Scan*, a aplicação de teste requer 868 portas lógicas adicionais para um teste interno ao componente e às suas ligações externas. Um componente com um encapsulamento com 40 pinos pode conter até 10000 portas lógicas. Conclui-se que a implementação do *Boundary Scan* utiliza 8% das portas lógicas do componente. Observa-se a existência de um tamanho fixo (dependente sempre do tipo de teste estipulados) e um tamanho variável (depende do numero de células BS do IC e das ligações ao PCB). [Kharagpur]

3.15. Vantagens e desvantagens do Boundary Scan

A opção pela utilização do *Boundary Scan* geralmente envolve factores económicos (diminuição de custos). A análise, tem de ser bastante cuidada pelos projectistas, que podem hesitar usar o *Boundary Scan* devido ao aumento de custo dos componentes. A análise tem de ser global, a todos os níveis de montagem e de teste, durante o ciclo de vida do produto no processo produtivo. Os benefícios compensam geralmente os prejuízos.

Vantagens

Os benefícios proporcionados pela infra-estrutura *Boundary Scan* são:

- Redução de custos na criação do *software* de teste
- Redução do tempo de teste
- Redução do tempo de criação de teste, consequentemente uma entrada mais rápida do produto no mercado
- O teste é simples e mais barato comparando com outras tecnologias
- Compatibilidade com interfaces de outros testes
- Diagnóstico rápido e eficaz de problemas que possam surgir numa linha de montagem

Ao fornecer o acesso a todas as entradas e saídas, a necessidade de pontos de teste físicos na placa é eliminada ou bastante reduzida, resultando numa poupança significativa com *layouts* mais simples (dispositivos de teste menos dispendiosos), o tempo de teste é reduzido, a reutilização do *software* e a possível utilização do interface para novos projectos.

Desvantagens

- Os problemas decorrentes da utilização do *Boundary scan* surgem na utilização extra de mais "material" silício devido à introdução do *Boundary scan* no núcleo do componente
- Aumento de pinos no componente
- Esforço adicional no design do *Layout*
- Aumento do consumo de energia

O *Boundary-Scan* é uma infra-estrutura de teste amplamente praticada que torna possível acelerar o desenvolvimento e a melhoria da qualidade dos produtos electrónicos.

A indústria, ao confiar na norma IEEE 1149.1, torna os testes relativamente mais rápidos, fáceis, baratos e altamente eficazes. Para muitos PCB há poucas alternativas devido ao acesso físico limitado dos circuitos.

3.16. Comparação entre ICT teste e Boundary Scan

	In Circuit Test		Boundary Scan
	Adaptador de agulhas	**Flying Probe**	
Velocidade	**+**	**--**	**O**
Componentes analógicos	**+**	**++**	**- (IEEE 1149.4)**
Componentes digitais	**O**	**-**	**+**
Ligações	**- (mecânica)**	**- (mecânica)**	**+ (via teste bus)**
Flexibilidade	**--**	**+**	**++**
Custo por projecto	**-- (novo adaptador)**	**O (criar teste)**	**o (criar teste)**
Custo do equipamento	**-**	**O**	**++**

Tabela 3 – Comparação entre ICT teste e Boundary Scan

Legenda:

-- Não funcional - Pouco Funcional + Bom ++ Muito Bom O Ideal

Comparando várias tecnologias de teste por estímulos (sinais injectados), analisando os vários parâmetros na tabela descritos, conclui-se que existe uma grande diferença entre eles, a opção de escolha requer uma ampla avaliação para o objectivo pretendido. Avaliando genericamente a tabela verificamos que o teste mais flexível, rápido e com custos mais interessantes é o *Boundary Scan*. [Goepel09]

Defeito/Procedimento de teste	Percentagem do valor total	AOI	AXI	ICT	FP	FT	BS
Defeito do componente electrónico							
Mecânico	5	5				K.A.	
Eléctrico	5			5		K.A.	4
Falta de componentes electrónicos	5	5		5	5	K.A.	5
Componentes electrónicos errados	5	5		5	5	K.A.	4
Posição errada	3	3		1	1	K.A.	
Componentes invertidos	1	1		1	1	K.A.	
Defeito de solda	5	5		5	5	K.A.	4
Defeito do forno de soldar							
Mau contacto/soldadura	10	10	2				
Solda fria	20	15	5	10	10	K.A.	15
Curto	20	15	15	20	20	K.A.	15
Posicionamento do componente	5	5		5	5	K.A.	
BGA	8	1	8	1	1	K.A.	8
BGA solda fria	3					K.A.	3
SMT soldadura por onda	5	5		2	2	K.A.	2
Percentagem total em %	100	80	30	60	60	K.A.	60

Tabela 4 - Comparação entre as técnicas mais utilizadas na indústria

A tabela mostra a detecção de defeitos em PCBs, defeitos provocados no processo e montagem e defeitos dos componentes (fabricante), concluímos que na detecção de defeitos o AOI é a solução mais eficaz, tendo no entanto a limitação de não controlar BGA. O *Boundary Scan* tem a limitação de não verificar componentes passivos numa placa, a sua posição e se está invertido. Conclui-se pela tabela que será interessante uma solução conjunta com o AOI e *Boundary Scan*. [Goepel09]

4. Exemplo de um simulador *Boundary Scan*

Exemplo baseado num Simulador Boundary Scan disponível na Internet. [PLD]

Neste exemplo, é possível estudar os princípios fundamentais de teste *Boundary Scan*. Podemos verificar os conceitos mais importantes; controlador TAP (*Test Access Port*), a instrução (modo de teste) e os registos de dados e instruções.

No final, podemos verificar as análises do diagnóstico da placa. Neste exemplo não é necessário qualquer tipo de programação em VHDL.

Este simulador é um objecto importante para assimilar os conhecimentos acima descritos no módulo *Boundary Scan*.

Sequência a utilizar

Inicialmente, precisamos de nos deslocar até ao registo de instruções no diagrama de estados do controlador TAP. De seguida, vamos para o estado *ShiftIR*. Neste estado, carregamos os valores encontrados para que o PCB realize os testes que pretendemos para cada componente (*extest, intest, sample, bypass, clamp, highz*).

O registo de instruções, é o rectângulo de cor amarela situado em cada componente que mostra os dados carregados actualmente; o rectângulo vermelho significa os dados que serão actualizados

Quando posicionado no estado *ShiftIR*, todas as combinações do registo de instruções do primeiro componente são carregadas, usando os sinais TMS e TCK. Em seguida, inserimos todas as combinações do registo de instruções do segundo componente e assim sucessivamente.

Existe uma caixa “TAP MODE” que, quando alternamos os sinais TCK e TMS, possibilita a visualização do deslocamento entre estados.

O Registo de Dados (DR)

No menu *Comand Mode*, vamos escolher o modo de teste *IDCODE* e definir o código *ID*; para outros componentes com a instrução *BYPASS,* colocamos '1' nos dados. Depois de escolher as outras opções de teste, vamos seleccionar a combinação de entrada que desejamos.

De seguida pressionamos o botão *ScanIR* e posteriormente *ScanDR*. Depois as informações de diagnóstico (resposta) vão aparecer. Podemos também escolher o modo "animado", onde visualizamos os dados a passar sequencialmente entre os pinos do componente.

Após esta sequência, podemos comparar os dados que foram carregados no *ScanDR* com os dados que foram lidos à saída dos componentes.

***Boundary Scan* estudo das instruções**

Neste simulador, podemos verificar as diferenças entre os modos: *EXTEST, INTEST, BYPASS*, *CLAMP* e *HIGHZ*.

Escolhemos os comandos *EXTEST* e *INTEST* para melhor assimilar o conceito *Boundary scan*.

Para este teste, temos de seleccionar os modos de teste pretendidos, verificar os sinais de controlo de todos os componentes e simular os vectores de dados escolhidos. Carregamos no botão *ScanIR*, posteriormente em *ScanDR* e observamos a informação à saída de todos os componentes.

Diagnóstico das ligações

Existe a possibilidade de inserir falhas no sistema (*Menu Mode*, *modo Random*, *menu Diagnostics*). Esta função irá introduzir uma falha aleatória numa das ligações do *PCB*, utilizando o modo de teste *EXTEST*. O utilizador tem a possibilidade de definir a falha e onde a introduzir no PCB.

Exemplo

Vamos utilizar uma placa padrão que contem três componentes

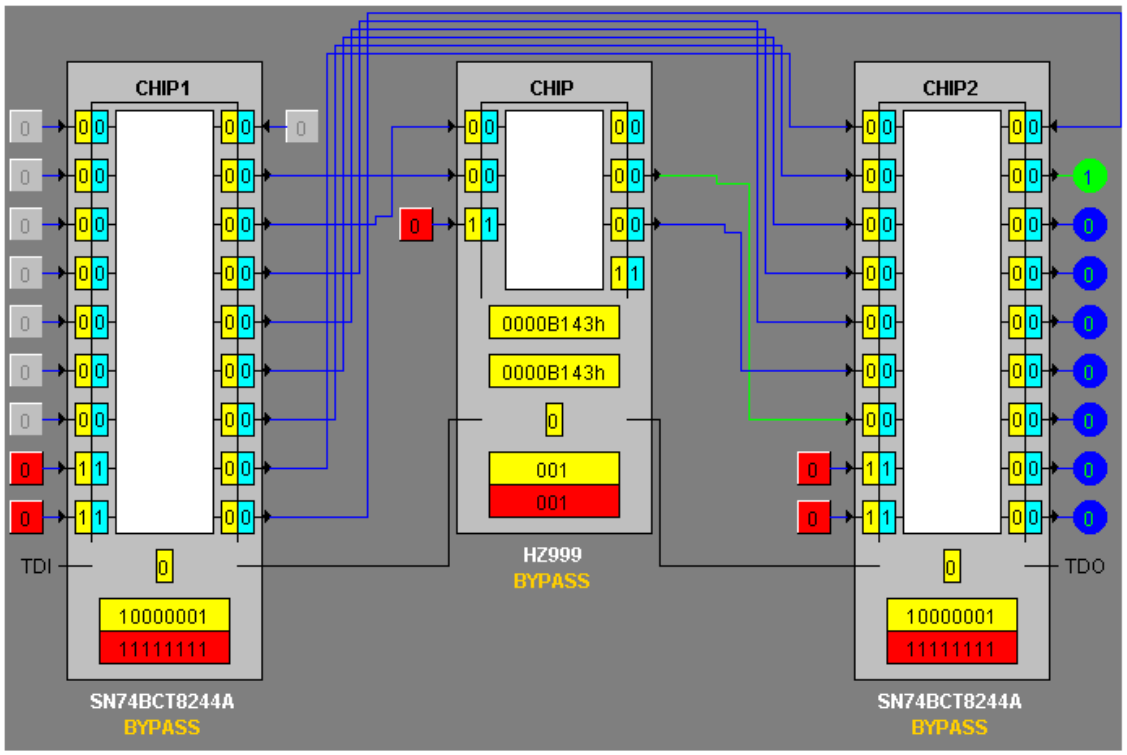


Figura 37 – Exemplo controlador *Boundary Scan*

O Registo de Instrução (*IR*)

Suponhamos que nos encontramos no estado *ShiftIR* e precisamos de carregar alguns valores para o *IR* para cada componente da placa, próximo passo?

Temos que definir o número de bits para todos os *IRs*: para os COMPONENTE1 e COMPONENTE2 os tamanhos são iguais de 8 bits, para o COMPONENTE tem um IR de 3 bits.

Carregamos nos registos do COMPONENTE1 e COMPONENTE2 com oito 0's e do COMPONENTE com três 1's.

No simulador, a direcção do fluxo de dados através dos componentes é da esquerda para a direita. Isso significa que o primeiro *IR* a preencher é o COMPONENTE2, depois o *IR* do COMPONENTE e por último, o registo do COMPONENTE1.

O Registo de Dados (DR)

Principais instruções do *Boundary Scan*: *EXTEST, INTEST, BYPASS, PRELOAD, CLAMP e HIGHZ*.

Vamos escolher as instruções *BYPASS* e *EXTEST* e estudá-las.

Bypass

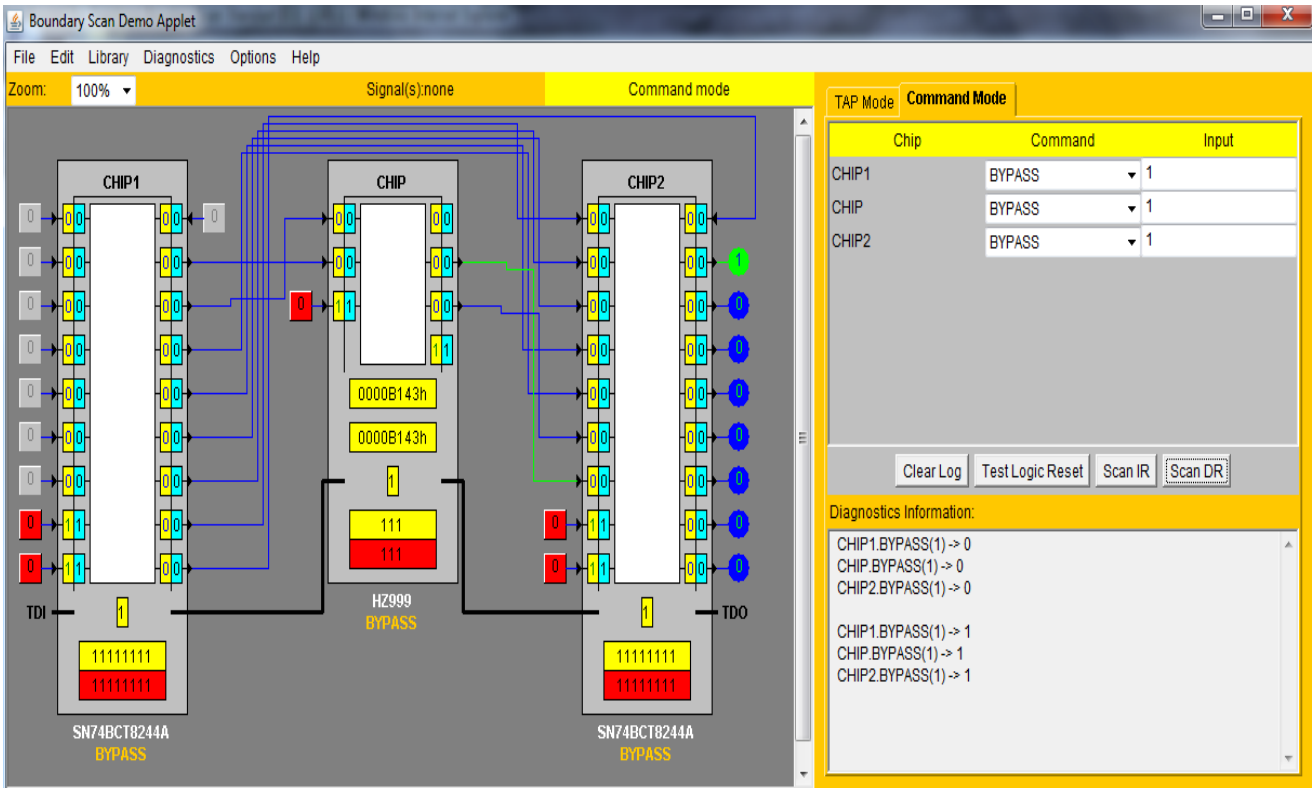


Figura 38 – Exemplo controlador Boundary Scan I

No modo *BYPASS*, nenhuma das ligações são testadas; os dados somente entram no registo de um bit e saem novamente, inserindo somente um atraso de relógio.

Extest

Sequência sem falhas

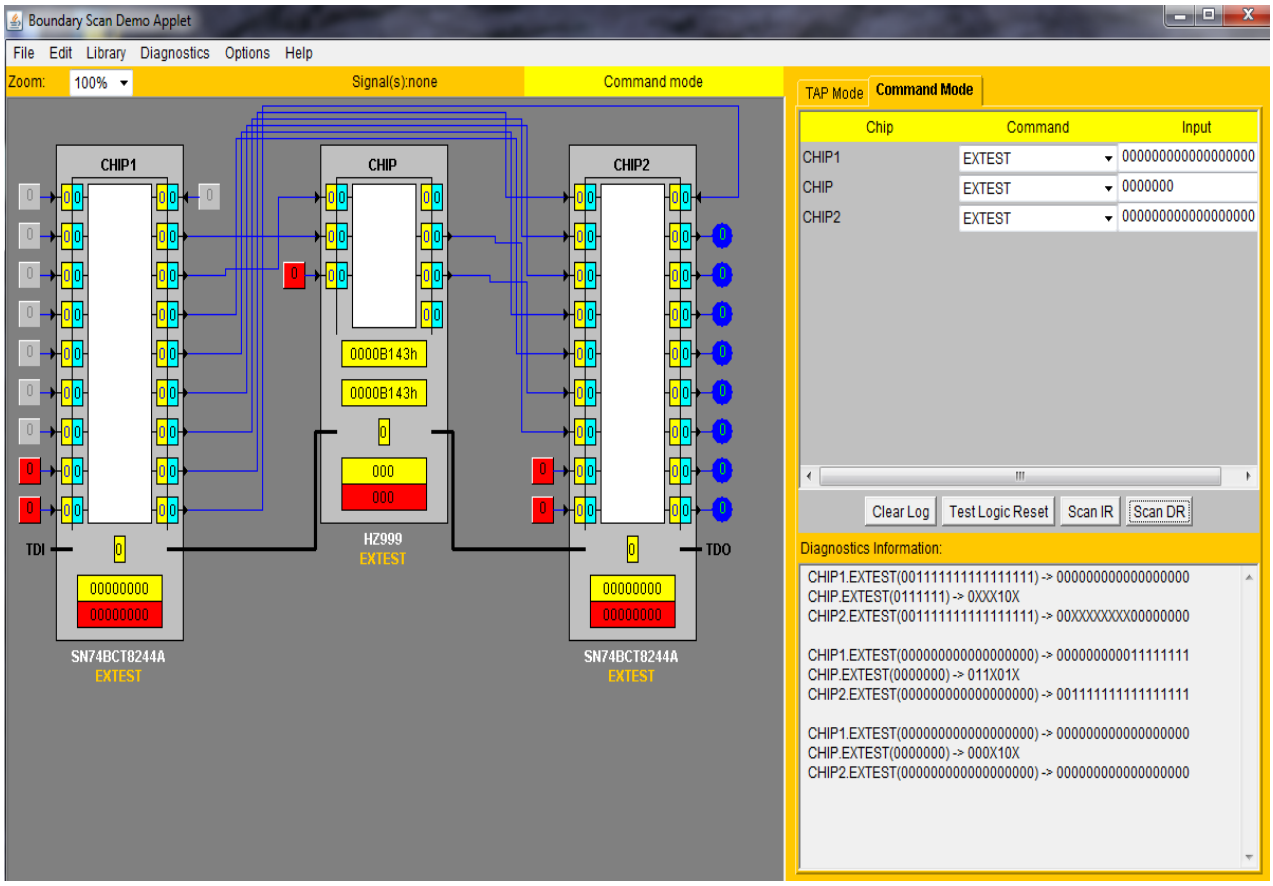


Figura 39 – Exemplo controlador *Boundary Scan* II

Fazer o diagnóstico das ligações

Vamos inserir uma falha no circuito, *Menu Diagnostics→ Insert fault→ Random Fault*. Podemos escolher os tipos de defeito que pretendemos aplicar ao circuito; circuito aberto ou curto-circuito.

De seguida, colocamos no modo *EXTEST* e seleccionamos o tamanho e o valor dos vectores de teste. Os sinais de controlo de todos os componentes devem estar a 0.

A mesma sequência, mas com falhas

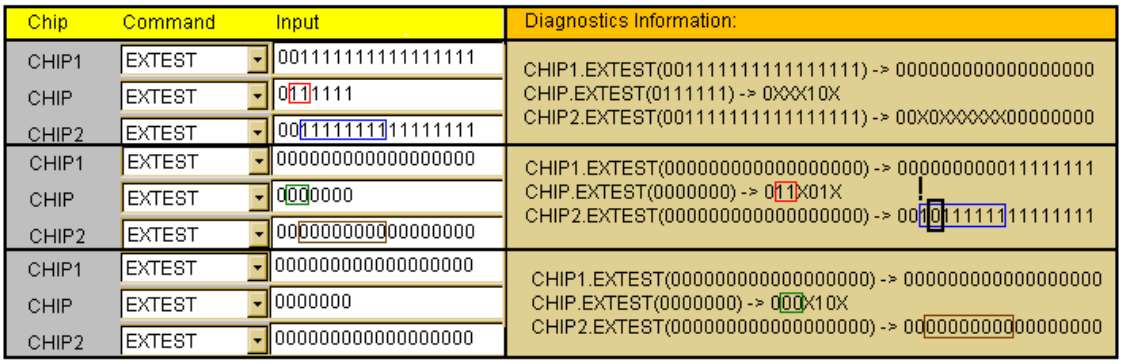


Figura 40 – Exemplo controlador *Boundary Scan* III

Vamos analisar as informações de diagnóstico das últimas duas simulações.

Devemos comparar a informação de diagnóstico (os valores à saída dos componentes) da segunda simulação com os valores de entrada da etapa anterior.

A mesma observação também deve ser aplicada para o terceiro que recebeu informações de diagnóstico. Deve-se comparar com os valores de entrada da segunda simulação.

Vamos começar com a saída COMPONENTE (são definidos 2 bits no rectângulo vermelho). Como podemos ver, há um 0 antes destes dois bits, faz de controlo. Então deveremos estar sempre atentos a este bit, estes bits são os únicos que estão ligados ao componente1.

Os dois bits de entrada da primeira simulação são os mesmos que foram lidos na saída COMPONENTE, isso significa que não há erro.

Na saída do COMPONENTE2, como foi feito anteriormente, não prestamos atenção aos primeiros 2 bits de controlo. Ao invés disso, olhamos para a entrada correspondente à simulação anterior; temos na entrada oito 1's e ao compará-los com a saída do COMPONENTE2, verificamos que a saída de dados neste componente não é a mesma que inserimos à entrada. A diferença está no segundo bit; esperado é '1', o obtido é '0'. Descobrimos assim a falha inserida aleatoriamente pelo programa.

Depois de descoberto, clicamos no fio correspondente; vamos encontrar o nome da conexão e o applet confirma se a nossa resposta é verdadeira.

5. *Serial Vector Format*

O Serial Vector Format normalmente designado como SVF, foi desenvolvido conjuntamente pelaTexas Instruments e pela Teradyne em 1991. O SVF permite descrever sequências ordenadas de testes baseados na norma IEEE 1149.1, é desenvolvido num formato ASCII. O objectivo é o de enviar sinais de teste "estímulos", aguardar pela sua resposta, segundo a máscara de dados baseada na norma IEEE 1149.1.

A necessidade do SVF surgiu do desejo de conseguir fazer com que os fornecedores independentes incluíssem a norma IEEE 1149.1. Estes componentes deveriam ser testados através de uma ampla selecção de *software* de simulação e de equipamentos de testes.

O SVF é uma linguagem que pretende normalizar o modo como os testes baseados na norma IEEE 1149.1 são descritos, permitindo a reutilização dos vectores de teste e, em última análise, a utilização destes em diferentes ferramentas de teste. Torna-se fácil incorporar o programa de teste estrutural no programa a executar pelo controlador BS. Considerar-se o formato SVF como modelo no que se refere ao teste estrutural através da infraestrutura BST, dada a sua utilização generalizada. [Asset99]

O programa de *Boundary-scan* é conduzido pela sequência de sinais que são inseridos no controlador interno dos componentes TAP. O comportamento do componente é determinado exclusivamente pelos estados dos pinos TAP.

As ferramentas *Boundary-scan,* devem manter o conhecimento das sequências necessárias para exercer determinadas operações dentro do dispositivo quando este está a fazer a sequência série nos pinos do componente (*serial scan path*).

Os constróis SVF da norma IEEE 1149.1, utiliza comandos que fazem a transição do controlador TAP a partir de um estado para outro.

Ao invés de descrever o estado explícito em cada ciclo TCK, o SVF descreve as transacções realizadas entre estados estáveis.

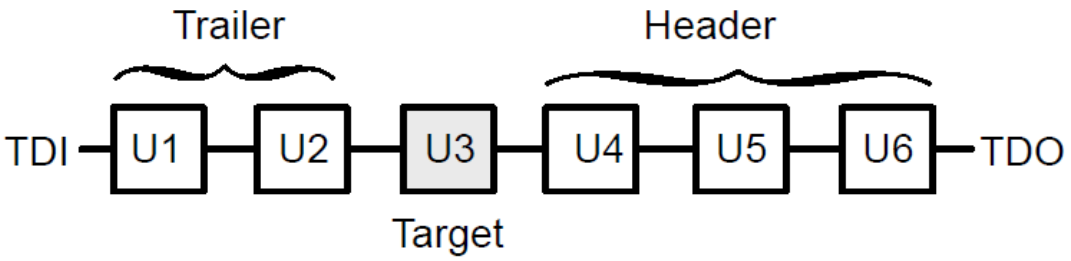
Os estados *Capture, Update, Pause*, …, não são explicitamente representados. Os dados são inseridos na entrada "*data in*", as respostas (dados) surgem no "*data out*" e comparadas com a máscara. De seguida, são todos agrupados numa forma fácil de compreensão.

Existe um comando para apoio que determina os estados TAP sempre que necessário. O SVF também suporta operações de combinações série e paralelo. Isso permite ao SVF adaptar-se a ambientes ATE, em que alguns estímulos/respostas são tratadas através da I/O paralela e os sinais série são enviados através de constróis da norma IEEE 1149.1.

O SVF também suporta o conceito de *"scan offsets".* Faz com que os deslocamentos sejam aplicados num teste a um componente ou à sua lógica incorporada no seu núcleo.

Quando em modo *BYPASS*, os dispositivos introduzem três bits no registo de dados antes e dois registos de dados depois do dispositivo de destino.

O SVF permite um "*Header"* e um "trailer" para ser definido, uma vez que mantém o registo de instruções e registos de dados dos dispositivos que não é o destino desejado no estado *BYPASS*.



Se o mesmo teste for direccionado a outro equipamento a jusante, seria simplesmente necessário alterar o "*Header"* e o "trailer". [Cammon]

Serial Vector Format

Uma especificação de testes, no formato SVF, configura-se como um ficheiro de texto, no formato ASCII, que contém um conjunto de comandos e parâmetros SVF.

Algumas considerações sobre o formato SVF

· Uma instrução é composta por um comando SVF e pelos respectivos parâmetros

· Cada linha de código não pode ultrapassar os 256 caracteres, embora uma instrução possa ser dividida por mais do que uma linha

· Uma instrução tem de terminar com um ponto e vírgula

· O formato SVF aceita comentários de linha usando "! " ou "//".

5.1. Exemplo de um ficheiro SVF

```
!Begin Test Program
TRST OFF;                                   !Disable Test Reset line
ENDIR IDLE;                                 !End IR scans in IDLE
ENDDR IDLE;                                 !End DR scans in IDLE
HIR  8  TDI (00);                           !8-bit IR header
HDR 16  TDI (FFFF) TDO (FFFF) MASK (FFFF);!16-bit DR header
TIR 16  TDI (0000);                         !16-bit IR trailer
TDR  8  TDI (12);                           !16-bit DR trailer
SIR  8  TDI (41);                           !8-bit IR scan
SDR 32  TDI (ABCD1234) TDO (11112222);      !32-bit DR scan
STATE   DRPAUSE;                            !Go to stable state DRPAUSE
RUNTEST 100 TCK ENDSTATE IRPAUSE;           !RUNBIST for 100 TCKs
!End Test Program
```

5.2. Principais comandos *SVF*.

· **ENDDR, ENDIR:**

Função: Especifica o estado final (por omissão) após uma operação de leitura no registo de dados e instrução.

Parâmetros: Tem um parâmetro que indica o estado estável da norma 1149.1 onde se quer que a operação de leitura termine. Entende-se por estado estável, um dos seguintes: *IRPAUSE, DRPAUSE, RESET e IDLE*.

Sintaxe: ENDDR [estado estável], ENDIR [Estado estável].

Informação extra: Uma vez especificado, o efeito do comando mantém-se até ser novamente invocado. Por defeito o estado inicial é *IDLE*.

· **FREQUENCY:**

Função: Especifica a frequência máxima do relógio de teste.

Parâmetros: Tem um parâmetro que indica a frequência em Hz (este parâmetro é opcional).

Sintaxe: FREQUENCY [frequência]

Informação extra: O valor da frequência do relógio é um número real. A frequência por omissão depende da implementação do controlador e, se for introduzida uma frequência ilegal, deve avisar o utilizador.

Exemplo

```
SIR 8 TDI(F3) TDO(01) MASK(03);    ! Set up BIST, full speed
FREQUENCY 90E3 HZ;                 ! Decrease to 90 kHz
RUNTEST 100000 TCK;                ! Execute BIST
FREQUENCY 1E5 HZ;                  ! Increase to 100 kHz
RUNTEST 300000 TCK 1 SEC           ! Error! 300000 TCK at 100 kHz is
   MAXIMUM 2 SEC;                  !    3 SEC, but MAXIMUM is 2 SEC
FREQUENCY;                         ! Return to full speed
```

· **HDR, HIR:**

Função: Especifica um cabeçalho padrão para ser usado no início de cada sequência de operações de leitura do registo de dados ou instrução.

Parâmetros: Tem cinco parâmetros opcionais: comprimento, TDI, TDO, MASK, SMASK, em que ordem não importa.

O parâmetro comprimento indica o tamanho do vector de teste. O TDI, especifica o valor do vector de teste a ser aplicado no início de cada teste expresso em hexadecimal. O TDO, especifica o valor da resposta ao teste anterior expresso em hexadecimal. O MASK, específica a máscara a ser usada na comparação do parâmetro TDO com a resposta obtida '1' para "*care*" e '0' para "*don't care*". O SMASK, específica a máscara a ser usada na comparação do parâmetro TDI com a resposta obtida '1' para "*care*" e '0' para "*don't care*".

Sintaxe: HDR [comprimento] TDI([HEX]) TDO(([HEX]) MASK(([HEX]) SMASK(([HEX]),

HIR [comprimento] TDI(([HEX]) TDO(([HEX]) MASK(([HEX]) SMASK(([HEX]).

Informação extra: Este comando é opcional e por omissão o cabeçalho não existe, mas uma vez especificado mantém-se até ser definido um novo cabeçalho ou o comprimento ser definido como '0', o que implica o desaparecimento do cabeçalho.

O valor dos parâmetros TDI, TDO, MASK, SMASK, não pode representar valores com comprimento superior ao parâmetro comprimento.

Exemplo

```
HDR 32 TDI(00000010) TDO(81818181) MASK(FFFFFFFF) SMASK(0);
HIR 16 TDI(ABCD);
...
HDR 0;    ! Removes the previous DR scan header.
```

· **PIO:**

Função: Especifica um padrão de testes paralelos.

Parâmetros: Vector de teste. "H" representa um '1', "L" representa um '0', "Z" representa alta impedância, "U" significa que deve ser detectado um '1', "D" significa que deve ser detectado um '0' e o "X" significa que não importa o que detecta.

Sintaxe: PIO (vector de teste).

Informação extra: Necessita que o comando *PIOMAP* seja efectuado primeiro.

Exemplo

PIO (HLUDXZHHLL);

· **PIOMAP:**

Função: Realiza o mapeamento dos nomes lógicos e direcção de I/O dos pinos do teste paralelo.

Parâmetros: Este comando pode ter dois parâmetros por cada pino de teste paralelo. O primeiro, define a direcção I/O (IN, OUT) e o segundo, o nome lógico.

Sintaxe: PIOMAP ([direcção] [nome] [direcção] [nome] ... [direcção] [nome]),

PIOMAP(IN STROBE OUT PSEN OUT P1 OUT P2 IN P3).

Informação extra: Este comando é obrigatório se for usado o comando PIO.

Exemplo

```
!PIOMAP must be placed before PIO statement
PIOMAP (IN  STROBE
        IN  ALE
        OUT DISABLE
        OUT ENABLE
        OUT CLEAR
        IN  SET);

PIO (HLUDXZ);

!Vector is:
!  STROBE  <- H
!  ALE     <- L
!  DISABLE <- U
!  ENABLE  <- D
!  CLEAR   <- X
!  SET     <- Z
```

· **RUNTEST:**

Função: Obriga a máquina de estados TAP a ficar num determinado estado por um período de tempo ou número de ciclos de relógio.

Parâmetros: O primeiro parâmetro é opcional e define o estado em que o controlador TAP deve ficar. O segundo parâmetro, também opcional, é o número de ciclos de relógio que deve esperar num determinado estado. O terceiro parâmetro, especifica o relógio a ser usado, TCK (relógio de teste) ou STK (relógio do sistema a testar). Existem também dois comandos opcionais que controlam o tempo mínimo e máximo para a execução do comando *RUNTEST*. Finalmente existe um parâmetro também ele opcional que define o estado final do comando *RUNTEST*.

Sintaxe: RUNTEST [estado] [número de ciclos] [tipo de relógio] [tempo mínimo] SEC MAXIMUM [tempo máximo] SEC ENDSTATE [estado final].

Exemplo

```
! Run in Run-Test/Idle for 1000 TCKs, then go to Pause-DR.
RUNTEST 1000 TCK ENDSTATE DRPAUSE;
! Go back to Run-Test/Idle for 20 SCKs, then go to Pause-DR.
RUNTEST 20 SCK;
! Run in Run-Test/Idle for 1000000 TCKs or at least one second,
! then go to Pause-DR.
RUNTEST 1000000 TCK 1 SEC;
! Run in Run-Test/Idle for at least one millisecond and at most
! 50 milliseconds, then remain in Run-Test/Idle.
RUNTEST 10.0E-3 SEC MAXIMUM 50.0E-3 SEC ENDSTATE IDLE;
! Run in Pause-DR for at least 50 ms, then go to Run-Test/Idle.
RUNTEST DRPAUSE 50E-3 SEC ENDSTATE IDLE;
! Run in Pause-DR for at least one second, then go to Run-Test/Idle.
RUNTEST 1 SEC;
! Run in Run-Test/Idle for at least 10 ms, then remain in
! Run-Test/Idle.
RUNTEST IDLE 1E-2 SEC;
```

· **SDR, SIR:**

Função: Introduz um vector de teste ou uma instrução e retorna o resultado do teste anterior.

Parâmetros: Tem cinco parâmetros opcionais; *comprimento*, TDI, TDO, MASK, SMASK, em que ordem não importa.

O parâmetro comprimento, indica o tamanho do vector de teste. O TDI, especifica o valor do vector de teste a ser aplicado no inicio de cada teste expresso em hexadecimal. O TDO, especifica o valor da resposta ao teste anterior expresso em hexadecimal. O MASK, especifica a máscara a ser usada na comparação do parâmetro TDO com a resposta obtida ‘1’ para “*care*” e ‘0’ para “*don’t care*”. O *SMASK* especifica a máscara a ser usada na comparação do parâmetro *TDI* com a resposta obtida ‘1’ para “*care*” e ‘0’ para “*don’t care*”.

Sintaxe: SDR [comprimento] TDI([HEX]) TDO(([HEX]) MASK(([HEX]) SMASK(([HEX]),

SIR [comprimento] TDI(([HEX]) TDO(([HEX]) MASK(([HEX]) SMASK(([HEX]).

Informação extra: Se existir um cabeçalho, este é introduzido no início do vector, assim como, se existir um "finalizador" é introduzido no fim do vector.

O valor dos parâmetros TDI, TDO, MASK, SMASK, não pode representar valores de comprimento superior ao parâmetro comprimento.

Exemplo

SDR 24 TDI(000010) TDO(818181) MASK(FFFFFF) SMASK(0);

SIR 16 TDI(ABCD);

· **STATE:**

Função: Obriga o controlador TAP a ficar num determinado estado ou a percorrer um conjunto de estados.

Parâmetros: O primeiro parâmetro é o caminho que se pretende seguir até ao estado final. O segundo parâmetro é o estado estável final. Entende-se por estado estável um dos seguintes: *IRPAUSE, DRPAUSE, RESET e IDLE.*

Sintaxe: STATE [estado1 estado2 estado3 ... estadoN] [estado estável]

Exemplo

!Force bus to DRPAUSE from current state it is in

STATE DRPAUSE;

!Dictate explicit path bus will take moving from

! DRPAUSE to IRPAUSE

STATE DREXIT2 DRUPDATE DRSELECT IRSELECT IRCAPTURE IREXIT1 IRPAUSE;

· **TDR, TIR:**

Função: Especifica um "finalizador" padrão para ser usado ao fim de cada sequência de operações de leitura do registo de dados ou instrução.

Parâmetros: Tem cinco parâmetros opcionais; comprimento, TDI, TDO, MASK, SMASK, em que ordem não importa.

O parâmetro comprimento, indica o tamanho do vector de teste. O TDI, especifica o valor do vector de teste a ser aplicado no inicio de cada teste expresso em hexadecimal. O TDO, especifica o valor da resposta ao teste anterior expresso em hexadecimal. O MASK, especifica a máscara a ser usada na comparação do parâmetro TDO com a resposta obtida '1' para "*care*" e '0' para "*don't care*". O SMASK especifica a máscara a ser usada na comparação do parâmetro TDI com a resposta obtida '1' para "*care*" e '0' para "*don't care*".

Sintaxe: TDR [número real] TDI([HEX]) TDO(([HEX]) MASK(([HEX]) SMASK(([HEX]),

TDI [número real] TDI(([HEX]) TDO(([HEX]) MASK(([HEX]) SMASK(([HEX]).

Informação extra: Este comando é opcional e, por omissão, o "finalizador" não existe, mas uma vez especificado mantém-se até ser definido um novo cabeçalho ou o comprimento ser colocado a '0' que implica o desaparecimento do cabeçalho**.** O valor dos parâmetros *TDI, TDO, MASK, SMASK* não pode representar valores com comprimentos superiores ao parâmetro *comprimento*.

Sintaxe: TDR [comprimento] TDI(([HEX]) TDO(([HEX]) MASK(([HEX]) SMASK(([HEX]),

TIR [comprimento] TDI(([HEX]) TDO(([HEX]) MASK(([HEX]) SMASK(([HEX])

Exemplo

TDR 32 TDI(00000010) TDO(81818181) MASK(FFFFFFFF) SMASK(0);

TIR 16 TDI(ABCD);

...

TDR 0; ! Removes the previous DR scan trailer.

TRST (*Test Reset*) - Executa um reset (opcional), se o dispositivo tiver o pino de reset na porta TAP.

Parâmetros:

ON.....Activo (Lógica 0)

OFF.....Inactivo (Lógica 1)

Z.....Alta Impedância

ABSENT.....Não está presente

Exemplo

TRST ON;

TRST OFF;

Os parâmetros dependem do comando a que estão associados, sendo que alguns são obrigatórios e outros opcionais. [Asset99]

Alguns parâmetros referem-se a estados do controlador TAP da norma IEEE 1149.1.

Estados do controlador TAP da norma IEEE 1149.1	Estado TAP do formato SVF
Test-Logic-Reset	RESET
Run-Test/Idle	IDLE
Select-DR-Scan	DRSELECT
Capture-DR	DRCAPTURE
Shift-DR	DRSHIFT
Exit1-DR	DREXIT1
Pause-DR	DRPAUSE
Exit2-DR	DREXIT2
Update-DR	DRUPDATE
Select-IR-Scan	IRSELECT
Capture-IR	IRCAPTURE
Shift-IR	IRSHIFT
Exit1-IR	IREXIT1
Pause-IR	IRPAUSE
Exit2-IR	IREXIT2
Update-IR	IRUPDATE

Tabela 5 - Correspondência entre os estados SVF e os respectivos estados da norma IEEE 1149.1.

6. FPGA

6.1. Introdução

Com o crescente avanço da tecnologia de implementação dos circuitos integrados (ICs) é possível desenvolver dispositivos com 10^7 a 10^8 transístores.

O ritmo do avanço da tecnologia de fabrico tem-se mantido exponencial nas últimas décadas, conforme a *Lei de Moore*. Esta lei é dividida com a de *Gordon*, que em 1965, observou que a densidade de componentes em circuitos integrados dobrava a intervalos regulares, deduzindo que este comportamento perduraria por muito tempo. O intervalo medido por *Moore* para que a densidade média dos circuitos integrados dobrasse foi de 18 meses, esta taxa tem vindo a diminuir, sendo de actualmente de 24 meses. Tudo isto significa que com o mesmo tamanho se conseguem sistemas mais complexos numa pequena área de silício, o que possibilita a implementação de módulos reconfiguráveis que são dados pelos transístores, devido à sua rápida transição do seu estado. [wiki10]

Uma FPGA (*Field Programmable Gate Array*) é um dispositivo lógico programável constituído por blocos lógicos e por uma rede de interligações, ambos configuráveis, que permitem ao utilizador implementar as mais variadas funções digitais.

Desde a sua introdução no mercado, em meados dos anos 80, estes dispositivos têm sido utilizados para os mais diversos fins e têm apresentado uma densidade e complexidade crescentes, permitindo deste modo o aumento da sua aplicabilidade.

Estes circuitos integrados configuráveis podem ser personalizados como diferentes ASICs (Circuitos Integrados de Aplicação Específica). Esta tecnologia permite com que o projecto, teste e correcção de circuitos integrados dedicados com um baixo custo, representem uma conformidade entre custo *versus* versatilidade.

O mercado actual dos semicondutores não é só caracterizado pela elevada complexidade dos sistemas, mas também pelo curto *time-to-market* (tempo até que um novo produto chegue ao mercado), alto desempenho e baixo consumo de potência. Nos últimos anos, este componente faz parte dos mais diversos produtos electrónicos, desde telemóveis, electrónica de consumo, consolas de jogos e sistemas avançados de telecomunicações e satélites, entre outros.

Existem actualmente vários fabricantes que apresentam diferentes tipos de FPGAs. Na minha dissertação, utilizo uma FPGA do tipo Spartan-3E da Xilinx, bem como o *software* disponibilizado pela mesma, ISE Versão 10.

6.2. História

Em 1985 uma empresa Americana, a Xilinx Inc, apresentou um novo modelo de *chip*, capaz de ser reprogramado de acordo com as aplicações do utilizador, designado de FPGA.

O objectivo que inspirou o desenvolvimento desta nova arquitectura foi o de alcançar a densidade lógica e a flexibilidade das matrizes de portas lógicas (MPGAs – *Mask Programmable Gate Arrays*) mantendo as melhores características da lógica programável tradicional. Um dispositivo lógico cuja funcionalidade fosse programável e facilmente modificável pelo utilizador, que proporcionasse uma redução nos tempos de desenvolvimento e que contornasse os riscos associados a soluções específicas do tipo ASIC. Esta nova arquitectura, denominada *Logic Cell Array* (LCA), constituiu o protótipo do que desde então se convencionou designar por FPGA (*Field Programmable Gate Array).*

Actualmente a evolução da tecnologia levou a ter mais de 20 milhões de portas lógicas disponíveis, permitindo a implementação de soluções do tipo SoC (*System-on-a-Chip),* o IC pode conter funções digitais, analógicas, de sinais mistos e muitas vezes de frequências de radio; tudo numa única FPGA.

6.3. Estrutura

O FPGA é composto, basicamente, por:

- Blocos lógicos: circuitos idênticos, constituídos pelo agrupamento de *flipflops*, permitem a criação de elementos lógicos funcionais;
- Blocos de entrada e saída: permitem o interface dos sinais entre os blocos lógicos e a saída do circuito;
- Matriz de interligações: são chaves e caminhos usados para interligar os blocos.

Este processo, conhecido como roteamento, é bastante complexo e definido pelo *software* usado na programação do IC.

Os blocos de entrada e saída (*IOB – In/Out Blocks*) da FPGA são responsáveis pelo interface entre as saídas provenientes das combinações de CLBs.

Uma FPGA contém uma enorme quantidade de blocos lógicos. Através de interligações entre eles, é possível a implantação de lógicas bastante complexas. A estrutura típica interna de um bloco lógico configurável de um FPGA consiste em *flip-flops*, um determinado número de multiplexers e uma estrutura de função combinatória, para implementar as funções lógicas.

Optimizado para lógica multi-nível e construção de circuitos complexos, como máquinas de estados.

Uma FPGA é programada com o uso de chaves electrónicas programáveis. As propriedades destas são: tamanho, resistência de contacto e capacitivos, definem os compromissos de desempenho da arquitectura interna do FPGA.

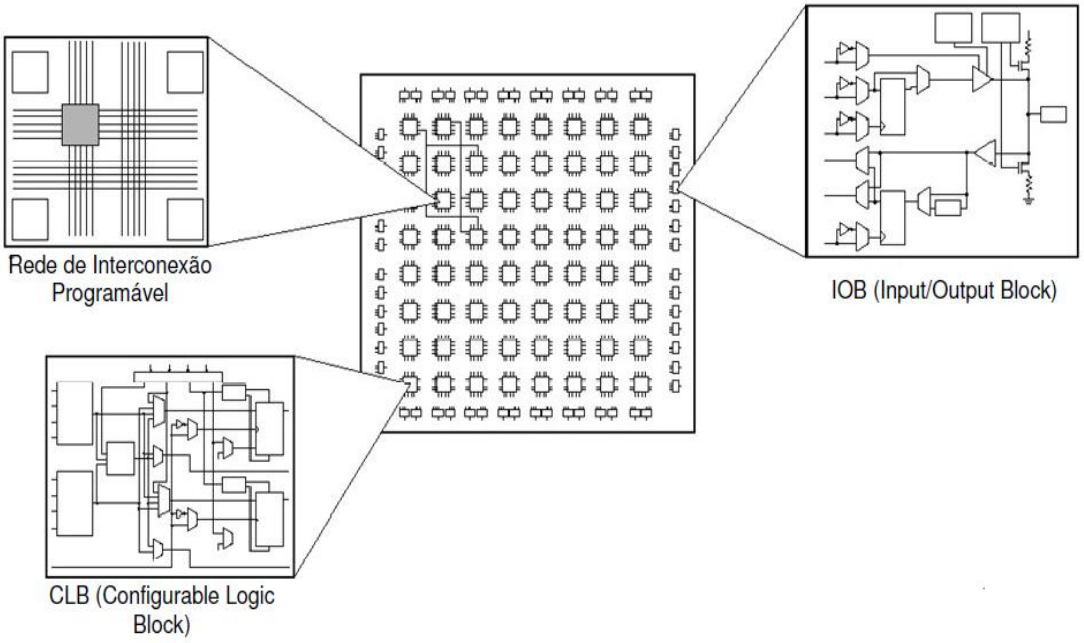


Figura 41 – Exemplo de uma FPGA

O desenvolvimento de módulos ou dispositivos nos componentes FPGA é feito através de linguagens de descrição de *hardware*, do inglês "*Hardware Description Language*" ou simplesmente, HDL.

À primeira vista, semelhantes à linguagem de computador tradicionais as linguagens HDL são um conjunto de instruções que permitem aos programadores descrever a lógica desejada para o *hardware*, de forma mais intuitiva e rápida.

Quando concluído, o código passa por um processo de compilação, sendo então implantando no componente.

As linguagens mais conhecidas são VHDL e Verilog. Existem inúmeras aplicações para as FPGA e, por serem bastante usadas, alguns fabricantes desenvolveram modelos especializados para determinadas funções, como para redes de comunicação ou processamento de sinais.

Vários modelos de FPGA, possuem microprocessadores embebidos para que seja possível executar programas em paralelo com a lógica do FPGA. Mesmo em componentes FPGA de menor custo, onde não há a presença de microprocessadores, é possível implantar um através de programação de lógica física.

Vários fabricantes possuem versões já prontas dos seus microprocessadores, que podem ser implementadas caso seja necessário.

Os microprocessadores "criados" com lógica no FPGA não possuem uma eficiência comparável à de microprocessadores fabricados directamente no componente (energia, frequência de relógio e I/Os).

Existem no mercado inúmeros projectos de desenvolvimento de circuitos FPGA, tornando-os cada vez mais específicos e rápidos.

6.4. Programmable Logic Devices

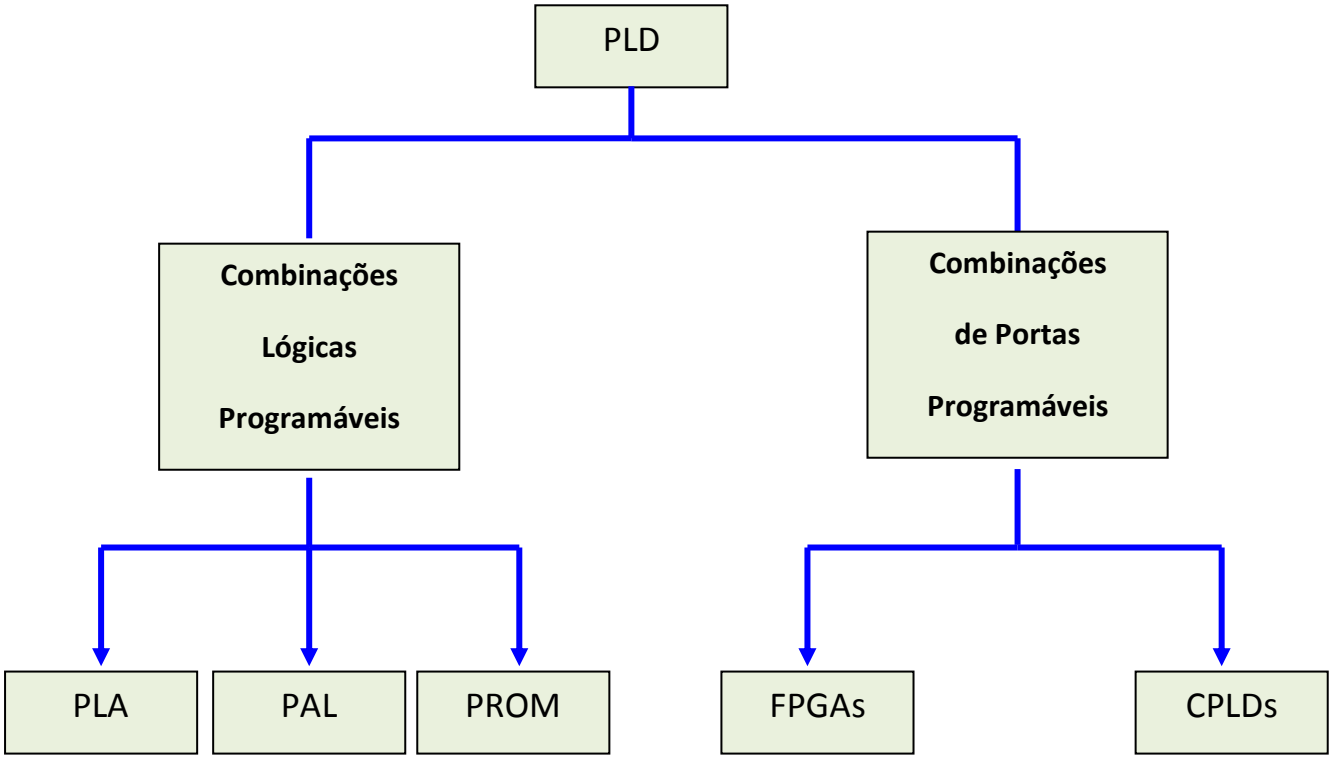


Figura 42 – Família Dispositivos Lógicos Programáveis (I)

Combinações Lógicas Programável

PROM - As *PROMs,* são dispositivos onde as combinações *"AND"* são pré definidas em fábrica e somente as combinações *"OR"* são programáveis, portanto capazes de gerar qualquer função lógica das suas variáveis de entrada, pelo facto de poderem gerar cada um dos possíveis produtos *"AND"* de tais variáveis.

PAL – *Programmable Array Logic* - O PAL tem as portas *"AND"* programáveis, enquanto as portas *"OR"* são pré definidas em fábrica.

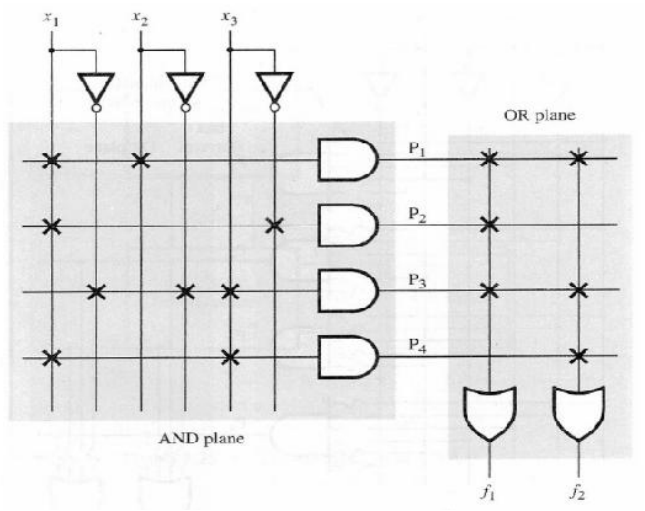


Figura 43- Combinações Lógicas Programável (II)

PLA – *Programmable Logic Arrays* - Um PLA possui tanto a matriz de portas *"AND"* quanto a matriz de portas *"OR"* programáveis, combinando as características de uma PROM e de um PAL.

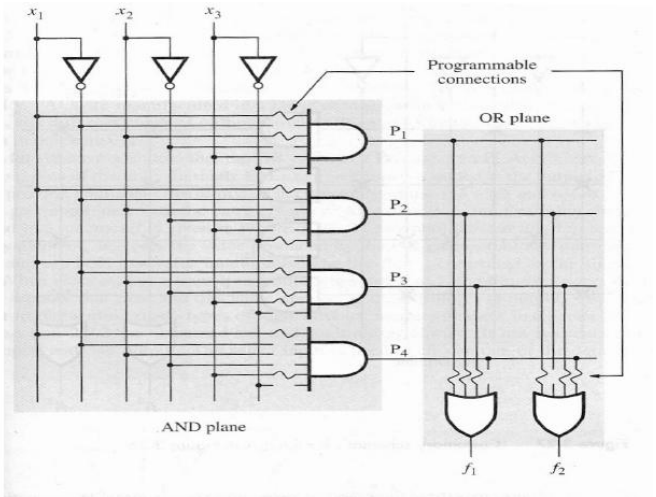


Figura 44- Combinações Lógicas Programável (III)

Combinações de Portas Programáveis

CPLD – *Complex* PLD - Os CPLDs podem ser vistos como dispositivos que utilizam na sua estrutura vários PLDs (PLA ou PAL).

FPGA – As *Field Programable Gate* Arrays representam uma classe de dispositivos de lógica programável. Diferentes dos dispositivos programáveis de menor densidade como os CPLDs, os FPGAs apresentam uma grande densidade interna de blocos lógicos configuráveis (CLBs) que podem ser interligados através de uma rede interna de roteamento de sinais. Tecnologias mais recentes de FPGAs permitem uma operação denominada reconfiguração dinâmica ou em tempo de execução. [Espinosa09]

Esta operação é caracterizada pela habilidade de substituição de contextos de *hardware* ou subsistemas digitais para rapidamente modificar (durante a operação) as funcionalidades dos componentes em operação e as suas interligações. Consideram-se pontos fundamentais neste tipo de operação, as facilidades oferecidas pela arquitectura para reconfiguração e a velocidade como ocorre tal operação. Apresentar-se bastante poderoso, implementações práticas baseadas em algoritmos de reconfiguração dinâmica são de difícil execução e muitas vezes restringem-se a uma faixa de aplicação. Considerando que diversos termos utilizados são recentes, é necessário um esclarecimento da terminologia usada.

· **Reconfiguração total**: É a forma de configuração onde o dispositivo reconfigurável é inteiramente alterado.

· **Reconfiguração parcial**: É a forma de configuração que permite somente uma zona do sistema digital seja reconfigurada. Uma reconfiguração parcial pode ser não-disruptiva, quando as porções do sistema que não estão sendo reconfiguradas permanecem completamente funcionais durante o ciclo de reconfiguração; ou disruptiva, quando a reconfiguração parcial afecta outras partes do sistema, tipicamente necessitando de uma paragem no sistema inteiro. Reconfiguração parcial não-disruptiva é frequentemente abreviada para reconfiguração parcial apenas.

· **Reconfiguração dinâmica**: Também chamada de *run-time reconfiguration* (RTR), *on-the-fly reconfiguration* ou *in-circuit reconfiguration*. Todas essas expressões podem ser traduzidas também como reconfiguração em tempo de execução. Reconfiguração dinâmica é outra forma de expressar a reconfiguração parcial não-disruptiva. O termo implica não haver necessidade de reiniciar o circuito ou remover elementos durante a reconfiguração.

· **Chaves de interconexão** (***SwitchMatrix***): É a capacidade de um dispositivo ou sistema de ser configurado em tempo de execução, durante a operação do sistema, em função de um conjunto de arquivos de configuração pré-carregados numa memória de controlo interno do dispositivo. Pode estar associado a procedimentos de reconfiguração parcial ou total.

6.5. Dispositivos Lógicos Programáveis

Os componentes da lógica programável são dispositivos que possuem lógica interna, centenas ou milhares de portas lógicas são chamados de dispositivos lógicos programáveis.

As combinações de portas programáveis, são estruturas mais genéricas e versáteis comparando-as com as baseadas na estrutura tradicional *"AND-OR"* das combinações lógicas programáveis.

A principal vantagem deste tipo de circuito é a possibilidade de reprogramação de um circuito as vezes que forem necessárias; ao contrário das combinações lógicas programáveis que só podem ser programadas uma vez, ou seja, definida a sua função lógica ela não poderá ser mudada.

As opções de *ASICs* actualmente disponíveis no mercado são:

a) "*Gate Arrays";*

b) *"Standard Cells";*

c) *"Full Custom";*

d) *PLDs* (*Programmable Logic Devices*);

Os *"Gate Arrays"* são matrizes de transístores, sem função definida separados por canais. Através de ligações feitas nestes canais conseguem-se ligar os transístores, para executarem uma função lógica específica. Levando à implementação de células básicas que serão utilizadas em projectos de sistemas digitais mais complexos. O ASIC é projectado como uma rede destas células interligadas. Esta tecnologia, pode ser usada sem problemas por projectistas de sistemas, não entrando em detalhes mais específicos dos blocos.

Os *"Standard Cells"* (Células Padronizadas) são ICs projectados com base em células de bibliotecas pré-definidas. Cada célula pode ser interligada para a implementação do sistema digital pretendido.

A vantagem deste tipo de circuito é o tempo muito reduzido do projecto. Em contrapartida, o projectista só pode utilizar blocos de *hardware* existentes em ficheiros do fabricante, sem modificar as suas características internas, o que não permite a optimização das mesmas.

Este problema é resolvido com a utilização dos ICs *"Full Custom".* Este permite ao projectista personalizar o *"layout"* e as interligações do IC. A principal desvantagem destes ICs é o seu alto custo de projecto porque o projectista participa praticamente em todo o ciclo juntamente com o fabricante.

Os PLDs são ICs que possuem uma matriz de interligações com portas *"AND"* e *"OR"* que podem ser programadas para realizar as mais diversas funções lógicas.

A principal vantagem dos PLDs é o tempo bastante reduzido gasto durante o projecto do *IC*, diminuindo assim os custos de desenvolvimento.

Existem *PLDs* que podem ser reprogramados possibilitando a sua utilização em diferentes projectos.

Os *PLD* (*Programmable Logic Devices*) dividem-se em:

1. *Field-Programmable Logic Devices* (**FPLDs**): os FPLDs mais usados comercialmente são os *Field-Programmable Gate Arrays* (FPGAs)
2. *Simple Programmable Logic Devices (**SPLDs**):* possuem funções lógicas mais simples, como funções *AND* e *OR*.
3. *Complex Programmable Logic Devices (**CPLDs**):* possuem vários blocos de *SPLDs* num único componente;
4. *High Capacity Programmable Logic Devices (**HCPLDs**):* são dispositivos lógicos programáveis de alta capacidade que oferecem mais de 600 portas disponíveis.

6.6. Desempenho

FPGA vs Microprocessadores

O ganho de velocidade com o uso de FPGAs advém pelo facto do *hardware* programado ser personalizado para um algoritmo em particular. Desta forma, o FPGA pode ser configurado para conter exactamente e somente as operações necessárias para a elaboração do algoritmo. Os microprocessadores de utilização geral precisam de incorporar todas as possíveis operações que os algoritmos poderão requerer para todas as estruturas de dados suportadas. Os FPGAs podem operar com o formato de dados, o número de unidades aritméticas e sua interligação definida unicamente pelo algoritmo. Além disso, podem ser processados diversos dados num único ciclo de relógio e optimizado o acesso à memória seguindo o comportamento do algoritmo.

Em processadores, a cache torna mais eficiente o uso da memória, mas somente permite um controle indirecto por parte do programador, que tem de adequar o seu programa ao funcionamento do *hardware*.

Apesar do potencial ganho de velocidade na execução de algoritmos com FPGAs, algumas classes de problemas dificilmente terão sucesso quando implementados com esta plataforma. Nos casos em que são poucas as oportunidades de paralelismo, quando utiliza operações com ponto flutuante, podem ter melhor eficiência quando executadas em CPUs.

Por este motivo, a utilização conjunta de CPUs e FPGAs pode significar um bom compromisso para a execução de aplicações que precisam de alto desempenho.

FPGA vs ASIC

De forma a contornar o problema de desempenho, faz-se a utilização de circuitos integrados de aplicação específica (ASICs). Este dispositivo, por ser projectado para apenas uma tarefa específica, apresenta um desempenho bastante superior aos microprocessadores. Entretanto, apresentam um alto custo de implementação, o que pode tornar o produto final inviável do ponto de vista económico.

A utilização de ASICs justifica-se a partir da produção de um grande volume de unidades, de forma a diminuir o custo por unidade deste componente. Podemos colocar os microprocessadores e os ASICs em extremos opostos considerando flexibilidade e desempenho: Os primeiros, apresentam grande flexibilidade de aplicações, baixo custo mas perdem em eficiência, o segundo, apresenta alto nível de eficiência. Contudo, com alto custo de implementação e restringe-se a uma aplicação específica.

Funcionalmente, os FPGAs estão colocados no meio destas características. Estes dispositivos têm a característica de agregar a flexibilidade dos processadores de uso geral com a personalização do *hardware* que é oferecida nos ASICs. Adicionalmente, apresenta um custo mais acessível que os ASICs, especialmente, a cada nova geração do componente.

. Microprocessador
. Uso geral
. Bom desempenho
(media das aplicações)

. ASIC
. Uso específico
. Excelente desempenho

Figura 45 – características dos dispositivos para implementação de sistemas digitais

6.7. Etapas da Implementação de *Hardware* em FPGA.

Primeira Etapa: Especifica a funcionalidade das tarefas da aplicação e quais serão os recursos (da arquitetura do FPGA) reservados para a sua execução. Estas tarefas são descritas numa linguagem funcional (neste caso VHDL – *Very High Speed Hardware Description Language*).

Segunda Etapa: Síntese do código. Tem como resultado uma rede de configuração que indica genericamente quais os elementos básicos que devem ser usados.

Terceira Etapa: *place and route.*

Tem como resultado um arquivo que indica quais os elementos reais de uma FPGA e específica quais serão usados na construção de um circuito. O arquivo também indica qual o caminho a ser seguido pelos sinais, de entrada e saída, pode ser chamado de plano de configuração.

Em seguida, é estimado o cálculo da área necessária no FPGA e tempo de execução destas tarefas. Estes cálculos vão determinar uma ordem de carregamento das tarefas na arquitectura reconfiguráveis, é mapeada num programa de controlo na etapa de síntese do sistema. É gerado um modelo de mapeamento das características de temporização das tarefas que compõe a aplicação, a interdependência de resultados entre as tarefas e das características da arquitectura.

A sobrecarga temporal de reconfiguração (tempo de reconfiguração) também é mapeada neste modelo. Além disso, todos os arquivos binários para configuração inicial e parcial do FPGA são gerados nesta etapa.

Quarta Etapa: configuração da FPGA.

Finalmente, os arquivos de configuração podem ser programados numa plataforma de testes. Esta plataforma é baseada num FPGA que permite reconfiguração parcial e dinâmica, pois apresenta em seu projecto um FPGA. A etapa de gerar um modelo de mapeamento é realizada através de uma ferramenta específica construída para este fim. A etapa de geração de arquivos de configuração é realizada através do uso de ferramentas específicas do fabricante do FPGA.

Tarefa	Area (CLBs)	Tempo (Ck Cycles)	Dependência de resultados
A	45	30	-
B	45	25	-
C	45	20	-
D	35	35	-
E	35	10	-
F	20	10	C
G	30	15	E, F

Tabela 6 – Característica de aplicação de teste

A tabela mostra uma aplicação de teste composta por tarefas individuais nomeadas de A até G. Temos o tempo de execução para cada tarefa da aplicação incluindo o tempo necessário para troca de tarefas no FPGA. [Guilhermino]

7. VHDL

7.1. História

A linguagem VHDL foi originalmente desenvolvida pelo comando do Departamento de Defesa dos Estados Unidos (DARPA) na década de 1980, pensado para documentar o comportamento de ASICs que compunham os equipamentos adquiridos pelas Forças Armadas.

A linguagem VHDL foi desenvolvida para substituir os manuais que descreviam o funcionamento dos ASICs; era a metodologia utilizada no projecto de circuitos. Os manuais resumiam-se em esquemas (circuitos) muito complexos, com pouca portabilidade, de difícil compreensão e eram extremamente dependentes da ferramenta utilizada por quem os produzia.

O projecto VHSIC era de alta prioridade militar com dezenas de fornecedores envolvidos, cada um destes agia desenvolvendo partes de projectos ou componentes que viriam a interligar-se noutros sistemas, tornando-os mais complexos. Estes optaram por desenvolver uma linguagem que servisse como base para troca de informações sobre estes componentes e projectos; uma linguagem que, independente do formato original do circuito, pudesse servir como uma documentação eficiente do circuito, possibilitando aos mais diferentes fornecedores e projectistas entender o funcionamento das outras partes. Esta facilita o desenvolvimento de um circuito e possibilita criar algoritmos sem a necessidade de especificar as ligações entre componentes.

O VHDL encaixou nesse pressuposto, podendo ser utilizado para tarefas de documentação, descrição, síntese, simulação, teste, verificação formal e ainda compilação de *software*, em alguns casos.

Após o sucesso inicial do uso do VHDL, este foi distribuído para o domínio público e foi normalizado pelo Institute of Electrical and Electronic Engineers, designada por IEEE1076 em 1987. Esta abertura aumentou a sua utilização, o que levou a uma revisão e a um novo padrão mais actualizado, lançado em 1993. Pequenas alterações foram feitas em 2000 e 2002. Em Setembro de 2008 foi aprovado pelo "REVCOM" a mais recente versão, IEEE1076-2008.

O VHDL expandiu-se tornando-se rapidamente numa das mais importantes linguagens de programação electrónica, dado ser independente da tecnologia do fabricante ou *software* com uma vasta gama de capacidades descritivas. A linguagem VHDL, é utilizada para descrever a parte do *hardware* dos sistemas computacionais em diversos níveis de abstracção: algorítmico, transferência entre registos, nível lógico com atraso unitário ou nível lógico com atraso detalhado. A linguagem permite a

exploração num nível mais alto de abstracção, havendo diferentes alternativas de implementação.

O facto da linguagem VHDL ser normalizada torna as ferramentas de *softwares* compatíveis entre si. Assim a escolha do fabricante pode ser adiada para uma fase posterior do projecto. A reutilização de projectos anteriores torna-se possível de uma forma mais simples com o uso de bibliotecas que podem ser partilhadas por diversos programadores, têm ainda a vantagem de representar um meio importante para a troca de informação de *hardware* entre organizações

A síntese do VHDL, para o nível físico, é efectuada automaticamente por ferramentas de *software* através da inserção de restrições de área e temporização, evitando a ocorrência de erros no nível físico. [Wiki11]

7.2. Descrição

A maioria dos circuitos digitais são baseados em portas lógicas e *flip-flops,* projectados a partir de equações Booleanas. Várias técnicas foram desenvolvidas para optimizar este procedimento incluindo a minimização de equações para um uso mais racional de portas lógicas e *flip-flops*.

Pela técnica Booleana deve-se obter uma expressão para cada entrada dos *flip-flops* e dos blocos lógicos, isto torna esta técnica impraticável para projectos maiores que contenham muitos componentes, devido ao grande número de equações.

Os projectos desenvolvidos por esquemas aumentaram as capacidades da técnica Booleana por permitir o uso de portas lógicas, *flip-flops* e sub-circuitos. Estes sub-circuitos por sua vez podem ser compostos por portas lógicas e outros sub-circuitos e assim sucessivamente podendo formar vários níveis hierárquicos. Com esta técnica obtêm-se uma visão mais clara da interligação entre os vários blocos. Nestas técnicas, o sistema é normalmente especificado como uma interligação de elementos e não na forma de comportamento do sistema.

A maior dificuldade dos métodos tradicionais na elaboração de projectos é a conversão manual da descrição deste em equações Booleanas. Esta dificuldade é eliminada com o uso de linguagens de descrição de *hardware* HDL (*Hardware Description Languages*). A partir de uma tabela de *verdade* ou da descrição de uma máquina de estado podemos implementar um circuito usando HDL.

Dentro da gama das HDLs, as mais populares são; VHDL e Verilog.

7.3. Fluxo de projeto definido em VHDL

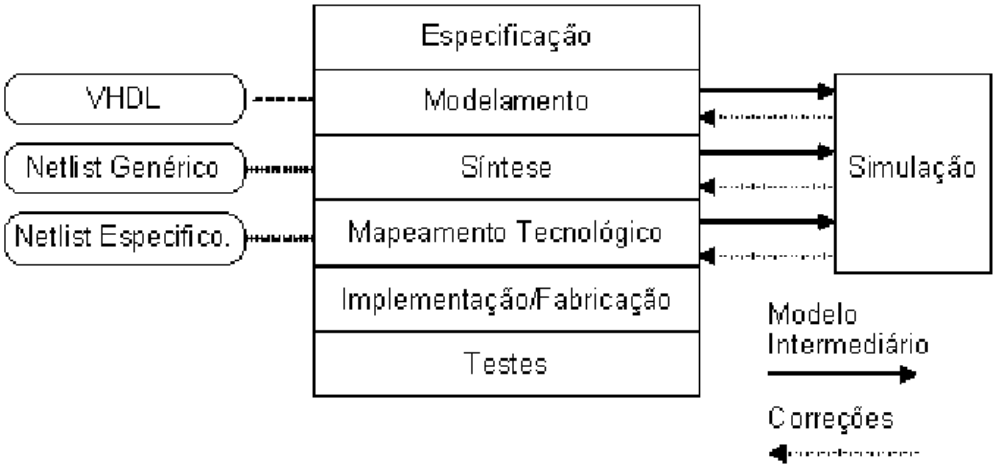


Figura 46 – Fluxo de projecto em VHDL

Especificação de Requisitos – Durante a etapa de Especificação de Requisitos, o programador faz um estudo de levantamento de todos os requisitos e características do sistema que definem o seu funcionamento. Esta fase é muito importante, pois permite o correcto entendimento do funcionamento do sistema, o que evita a ocorrência de erros futuros.

Modelamento - O projecto é iniciado na etapa de modelamento, onde com base nas especificações da etapa inicial, o programador escreverá os modelos que representam o circuito.

Síntese – A Síntese de Alto Nível está para o *hardware* assim como a compilação está para o *software*. Na síntese, o modelo descrito será convertido para estruturas de dados representando as ligações, blocos, componentes e portas lógicas. Esta etapa é automática, dependente da ferramenta de *software* utilizada. Durante a síntese são pré-avaliados os requisitos do sistema a fim de indicar se o circuito irá atendê-los adequadamente. Após a síntese ainda não está definido o circuito a ser implementado, a especificação intermediária resultante é ainda bastante genérica e pode ser direccionada para uma de muitas possibilidades de tecnologias de implementação.

Mapeamento Tecnológico – O Mapeamento Tecnológico permite que o circuito seja definido dentro da tecnologia em que será implementado, pouco influindo nesse processo o programador.

Implementação – Na etapa de implementação/fabricação são criados os primeiros protótipos, avaliadas as condições finais, detalhes de produção entre outros detalhes de implementação final.

Testes – Em seguida à fabricação, os circuitos são testados para que possam ser entregues ao utilizador com garantia de isenção de falhas. [Pessoa08]

7.4. Estrutura

Com a linguagem VHDL é possível descrever um sistema digital através de um arquivo de texto, possibilitando, deste modo, a implementação do mesmo através de diferentes ferramentas de desenvolvimento.

Os sistemas necessitam de uma interface com o exterior em VHDL. Esta interface designa-se por *entity* e é fundamental para qualquer sistema. A parte interna responsável pelas transformações dos dados é chamada de corpo ou *architecture*.

Qualquer sistema, independente da sua complexidade, necessita de uma interface *entity* e de um corpo *architecture*. Por vezes, alguns deles necessitam de funcionalidades adicionais, conhecidas como *package*.

Entity

A *entity* é a parte principal de qualquer projecto que descreve a interface do sistema. O que é descrito na *entity* fica automaticamente visível a outras unidades associadas a ela. O nome do sistema também é o próprio nome da *entity*, esta deverá ser a primeira etapa ao iniciar-se um projecto em VHDL.

A *entity* é composta por duas partes: *parameters* e *connections*.

- *Parameters*- diz respeito aos parâmetros vistos do exterior, tais como, a largura do barramento, frequência de operação e também os declarados como *generics*.
- *Connections*- menciona onde acontece a transferência de informações para dentro e fora do sistema, são declarados por *ports*.

Architecture

A *entity* de um sistema é tão importante que a *architecture* é especificada na forma de *architecture of entity*. Um sistema pode ser descrito em termos de funcionalidade. Este indica qual o comportamento do componente de *hardware* que é a parte onde guarda o algoritmo interno do circuito.

Podem existir muitas *architecture* numa mesma entidade, mas apenas uma delas pode estar activa, de cada vez.

Package

Aquando necessário utilizar algo não definido nas bibliotecas do VHDL padrão, faz-se uso do *package*. A única limitação é que o package deve ser previamente

definido antes do início da *entity* e efectuado por meio de duas declarações: *library* e *use*.

Dos vários *packages* existentes, o mais conhecido e usado é o *STD_LOGIC_1164* da IEEE, que contém a maioria dos comandos usados em VHDL.

O uso deste *package* é descrito por: *library IEEE; use IEEE.std_logic_1164.all*

7.5. Sinais

Os sinais são importantes nos sistemas electrónicos transmitindo dados internamente ou externamente ao sistema, assumindo assim um papel muito importante em VHDL. Os externos são apresentados na *entity* e os internos na *architecture*.

Os sinais podem ser transmitidos em série (uma vez) ou paralelo (barramento chamado vector). Estes são chamados de *std_logic* e *std_logic_vector*, respectivamente.

Num sinal *std_logic_vector*, a ordem dos bits é importante se o bit '7' for o mais significativo e o bit '0' o menos significativo, em VHDL será representado por *std_logic_vector (7downto0)*.

Os sinais externos são apresentados na *entity* pela designação *port*; os *ports* são a interligação entre a *entity* e o ambiente.

Cada sinal tem nome e tipo únicos. A direcção do sinal deve estar definida e ser de entrada (*input*), saída (*output*) ou bidirecional (*inout*).

7.6. Descrição de comportamento (*PROCESS*)

A funcionalidade de um sistema corresponde a uma lista de operações a serem executadas para obter um determinado resultado.

O *Process,* é o modo formal de efectuar uma lista sequencial dessas operações tendo um formato estruturado.

Existem algumas regras para realizar uma descrição *process*. Deve-se especificar um processo que é realizado pelo comando *process* que fica posicionado entre o nome e os dois pontos. No final, temos de o fechar com um *end of process*. A declaração *begin* é usada para dar início às operações sequenciais.

Ao contrário de linguagens de programação convencionais, os processos descritos em VHDL não terminam após a execução do último comando. O *process* é novamente executado desde o primeiro comando, aguardando a ocorrência das condições de entrada e executando posteriormente as tarefas definidas.

7.7. Elementos utilizados na elaboração do programa

Escalares

Escalar é o oposto ao *array*, é um único valor

- *character / bit / boolean / real / integer / physical_unit*
- std_logic (IEEE)

As variáveis e sinais no *package* IEEE_std_logic_1164, podem assumir nove valores:

0, **1**, **U** (não inicializado), **X** (desconhecido), **Z** (*tri-state*), **W** (fraco), **H** (alto), **L** (baixo), **-** (*don't care*).

Character

O VHDL não é "*case sensitive*", excepto para caracteres.

- Valor entre aspas simples: 'a', 'x', '0', '1', …
- Declaração explícita: *character* ('1'), neste caso o '1' também pode ser um '*bit*'.
- *String*: tipo que designa um conjunto de caracteres.

Exemplo: “Ricardo”.

Bit

· Assume valores ‘0’ e ‘1’

· Declaração explícita: bit (‘1’), neste caso o ‘1’ também pode ser ‘*character*’.

· *Bit* não tem relação com o tipo *boolean*.

· *Bit_vector*: tipo que designa um conjunto de bits.

Exemplo: “001100” ou x”00FF”.

Boolean

· Assume valores *true* e *false*.

· Útil apenas para descrições abstractas, onde um sinal só pode assumir dois valores

Real

· Utilizado durante o desenvolvimento da especificação

· Sempre com ponto decimal

· Exemplos: -1.0 / +2.35 / 37.0 / -1.5E+23

Inteiros

· Exemplos: +1 / 1232 / -1234

· Não é possível realizar conversões sobre inteiros

· Permitem operações lógicas e aritméticas: *signed, natural, unsigned, bit_vector*

Physical

· Representam uma medida: tensão, tempo, ….

· Tipos pré-definidos: fs, ps, ns, um, ms, sec, min, hr

Arrays

· Colecção de elementos com o mesmo tipo:

· Type word is array (31 downto 0) of bit;

· Type memory is array (address) of word;

· Type transform is array (1 to 4, 1 to 4) of real;

· Type register_bank is array (byte range 0 to 132) of integer;

· Array sem definição de tamanho

· Type vector is array (integer range <>) of real;

· Arrays pré definidos:

· Type string is array (positive range <>) of character;

· Type bit_vector is array (natural range <>) of bit;

Preenchimento de um array: posicional ou por nome

Type a is array (1 to 4) of character;

Posicional*:* ('f', 'o', 'o', 'd')

Por *nome:* (1 => 'f', 3 => 'o', 4 => 'd', 2 => 'o')

Valores *default:* ('f', 4 => 'd', others => 'o')

Constantes

· Nome dado a um valor fixo

· Consiste de um nome, tipo e de um valor (opcional, com possibilidade de declaração posterior)

· Sintaxe: constant identificador : tipo [:=expressão];

· Correcto: *constant gnd: real := 0.0;*

· Incorrecto *gnd := 4.5;*

Variáveis

· Utilizada em processos sem temporização, atribuição imediata.

· Sintaxe: variable identificador (es) : tipo [restrição] [:=expressão];

Exemplo:

variable índice : integer range 1 to 50 := 50;

variable ciclo_de_máquina : time range 10 ns to 50 ns := 10ns;

variable memória : bit_vector (0 to 7)

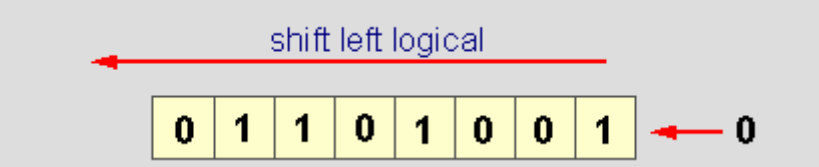
variable x, y : integer;

Deslocamento

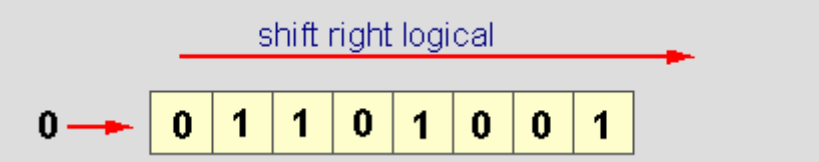
As operações de deslocamento, são restritas a *arrays* cujos elementos devem ser *bit* ou *boolean*. Estas operações exigem dois operandos; um o *array* e o outro do tipo *integer,* que determina o numero de posições a serem deslocadas. Se o valor do segundo operando for negativo, o sentido do deslocamento fica invertido.

As operações de deslocamento são:

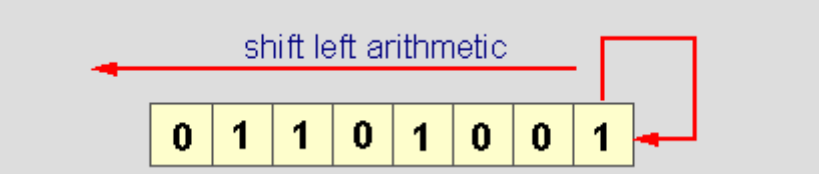
Sll: *shift left logical* (deslocamento lógico a esquerda),



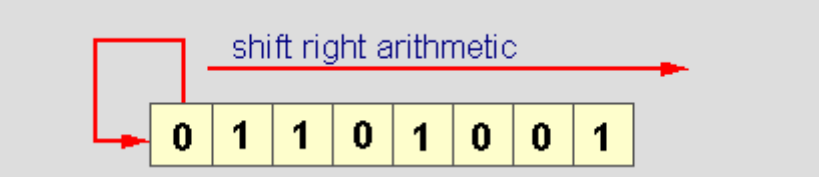
Srl: *shift right logical* (deslocamento lógico a direita),



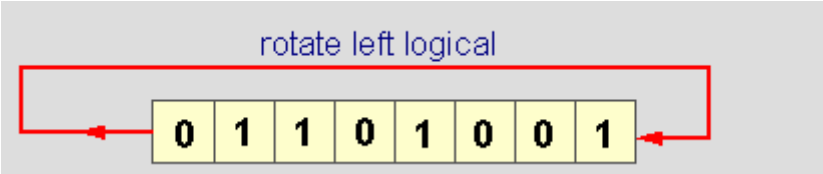
Sla: *shift left arithmetic* (deslocamento aritmético a esquerda),



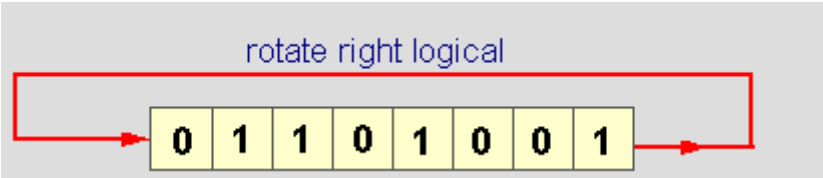
Rla: *shift right arithmetic* (deslocamento aritmético a direita),



Rol: *rotate left logical* (rotação lógica a esquerda),



Ror: *rotate right logical* (rotação lógica a direita).



Sinais

- Comunicação entre módulos.
- Temporizados.
- Podem ser declarados na entity, architecture ou no package.
- Não podem ser declarados em processos, podendo ser utilizados no interior destes.
- Sintaxe: signal identificador (es) : tipo [restrição] [:=expressão];

Exemplo:

signal cont : integer range 50 downto 1;

signal ground : bit := ´0´;

signal bus : bit_vector;

Expressões

Expressões são fórmulas que realizam operações sobre objectos do mesmo tipo

Operações lógicas: *and, or, nand, nor, xor, not*

Operadores Numéricos

As operações de adição (+), subtracção (-), multiplicação (*), divisão (/), módulo

(*mod*), valor absoluto (*abs*), resto (*rem*) e potência (**) são aplicados aos tipos *integer* e *real*.

Os operandos devem ser do mesmo tipo. O tipo *time,* exige que os operandos sejam iguais nas operações de adição e subtracção, mas um operando *time* pode ser multiplicado ou dividido por um operador *integer* ou *real*.

Comparações

As comparações entre dois objectos podem ser igual (=), diferente (/=), menor (<), menor ou igual (<=), maior (>) e maior ou igual (>=). Os objectos comparados devem ser do mesmo tipo e podem ser *boolean, bit, character, integer, real, time, string* ou *std_logic_vector*. O resultado é sempre *boolean* (*true* ou *false*).

Concatenação

Concatenação, é uma forma conveniente de se unirem dois ou mais vectores, criando um novo vector cujo tamanho é a soma dos vectores dos operandos. Os vectores devem ser do mesmo tipo e podem unir-se também a um bit e a um vector.

Atribuição de Sinais

Os resultados das operações em VHDL podem ser atribuídos às saídas. Isto é feito pelo símbolo (<=) em que o valor da expressão a direita é atribuído à expressão da esquerda. Para auxiliar na memorização desta operação, basta observar que a seta formada, indica o fluxo da informação.

Exemplos:

```
x <= y <= z;
a <= b or c;
k <= '1';
m <= "0101";
n <= m & k;
```

Um projeto fica incompleto se não for verificado, o VHDL tem a funcionalidade de auxiliar à detecção de erros antes do início de simulação. Exemplo: Interligar um barramento de 4 bits num barramento de 8 bits.

7.8. TEST BENCH

Uma das formas de testar um projecto em VHDL é pelo uso do *test bench*. É um ambiente onde o projecto é verificado através da aplicação de sinais ou estímulos e da visualização das respostas.

O *test bench* substitui o ambiente de projecto, faz com que o comportamento do projecto possa ser observado e analisado. Consiste num gerador de sinais que podem ser gerados internamente ou provenientes de um arquivo. Isto é possível em VHDL, porque a chamada de componentes e os processos são comandos concorrentes.

Há muita flexibilidade na criação de um *test bench*. Os estímulos são um conjunto de sinais declarados internamente na arquitectura da *test bench* e passada para os *ports,* que são definidos como formas de onda num ou mais processos comportamentais.

Os projectos simples podem ser facilmente simulados; projectos mais complexos podem exigir mais tempo para a simulação. [UFI]

7.9. Metodologia de Projecto de Sistemas Digitais utilizando FPGAs

Todos os projectos criados no Xilinx-ISE são iniciados através de:

a) Definição do directório de trabalho

b) Definição do nome do projecto

c) Definição do tipo de arquivo que descreve o nível mais alto de hierarquia de projecto

d) Escolha da FPGA, na qual será implementado o sistema

e) Criação de arquivos novos de projecto ou adição de arquivos já existentes

Se for escolhida a criação de um novo arquivo, as opções mais utilizadas são:

a) *Schematic*: esta opção cria um arquivo gráfico para desenhar o projecto através de blocos de biblioteca.

b) *Test Bench Waveform*: permite a criação de um arquivo com as formas de onda dos sinais de entrada/saída do sistema, para posterior simulação do mesmo.

c) *VHDL Module*: permite a criação de arquivos do tipo VHDL.

A ferramenta Xilinx-ISE, facilita a criação do arquivo VHDL através da possibilidade de gerar um arquivo modelo.

7.10. VHDL vs Linguagens Convencionais

Vantagens

- Projecto independente da tecnologia;
- Ferramentas de CAD compatíveis entre si;
- Facilidade na actualização dos projectos;
- Redução do tempo de projecto e custo;
- Eliminação de erros de baixo nível;
- Reduz "*time-to-market*".

Desvantagens

- O *Hardware* gerado é menos optimizado;
- Redução do controlo e observação do projecto;
- Falta de pessoal treinado para programar esta linguagem;

7.11. Síntese

Actualmente, os projectos de electrónica serão, cada vez mais, descritos a um nível de abstracção elevado, recorrendo para isso ao uso de linguagens de *hardware*.

Ao contrário de outras linguagens de descrição de *hardware*, que geralmente são de propriedade de determinados fabricantes, o VHDL é público, o que garante a sua independência face às políticas de cada companhia. A própria definição da linguagem envolveu a participação dos utilizadores, levando a que o VHDL se tornasse numa linguagem cada vez mais usada.

A aceitação do VHDL, como uma norma para a descrição de circuitos electrónicos por parte do IEEE, trouxe enormes vantagens para a comunidade de CAD.

Do ponto de vista dos utilizadores é necessário aprender uma única linguagem para "todas as fases" do projecto (simulações, síntese, …). Permite de uma forma normalizada, a partilha de informação entre diferentes grupos de projecto, usando diferentes níveis de abstracção.

Os fabricantes dos dispositivos podem distribuí-los sem fornecer qualquer tipo de informação confidencial sobre a sua construção, sendo possível a documentação de sistemas digitais de uma forma tecnologicamente independente.

8. Desenvolvimento do controlador *Boundary scan*

8.1. Objectivo

O objectivo desta dissertação de mestrado é o de projectar uma solução que facilite o controlo de qualidade dos PCBs e, com isso, melhorar a eficiência do seu processo de fabrico e competitividade geral das empresas.

De uma forma mais específica, pretendeu-se desenvolver um controlador BST que permita o acesso e controlo de infra-estruturas do tipo *Boundary Scan*, tendo como finalidade detectar falhas estruturais nos ICs e verificar a funcionalidade das ligações num PCB.

As potencialidades proporcionadas pelo controlador são asseguradas por um conjunto de instruções que permitem a realização de testes através da infra-estrutura BST. O controlador será capaz de ler o conteúdo de dados enviados pela FIFO_IN, interpretá-los e decidir qual o procedimento a efectuar; deslocamento entre *State*, definir modo de teste pretendido *ShiftIR* ou introduzir uma sequência de dados *ShiftDR*.

A execução das sequências de operações definidas conduz à configuração do controlador TAP nos modos de testes pretendidos (*Extest, Bypass*, …). A configuração é realizada através de estímulos introduzidos nas linhas TDI, TMS e requer o TCK activo. O TCK está sempre sincronizado com a linha de relógio. Após a definição do modo de teste serão enviados dados através da linha TDI. Na linha TDO serão capturados os dados obtidos através do teste ao PCB. De seguida, serão armazenados em blocos de oito bits e posteriormente, enviados para a FIFO_OUT, onde serão armazenados por ordem de chegada.

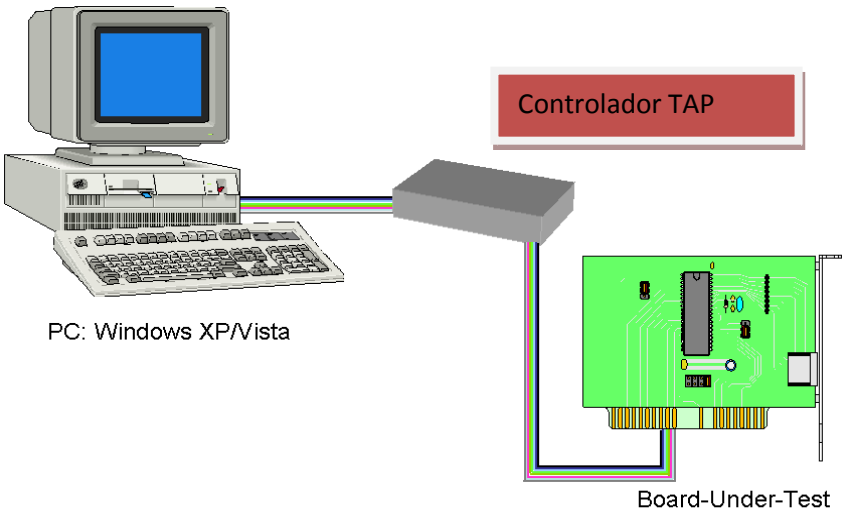


Figura 47 – Interface JTAG

A estrutura do controlador BST requer a existência de três blocos:

I. **FIFO_IN** - Simula a recepção de instruções ou dados enviados pelo *software* do PC
II. **FIFO_OUT** - Armazena os dados obtidos do teste ao PCB, armazena-os e, quando cheia, simula o envio para o PC
III. **Controlador TAP -** Controla o envio de estímulos necessários para testar o PCB; fá-lo através das linhas TMS, TDI e TCK. Recebe e armazena os dados através da linha TDO.

8.2. Organigrama do controlador BST

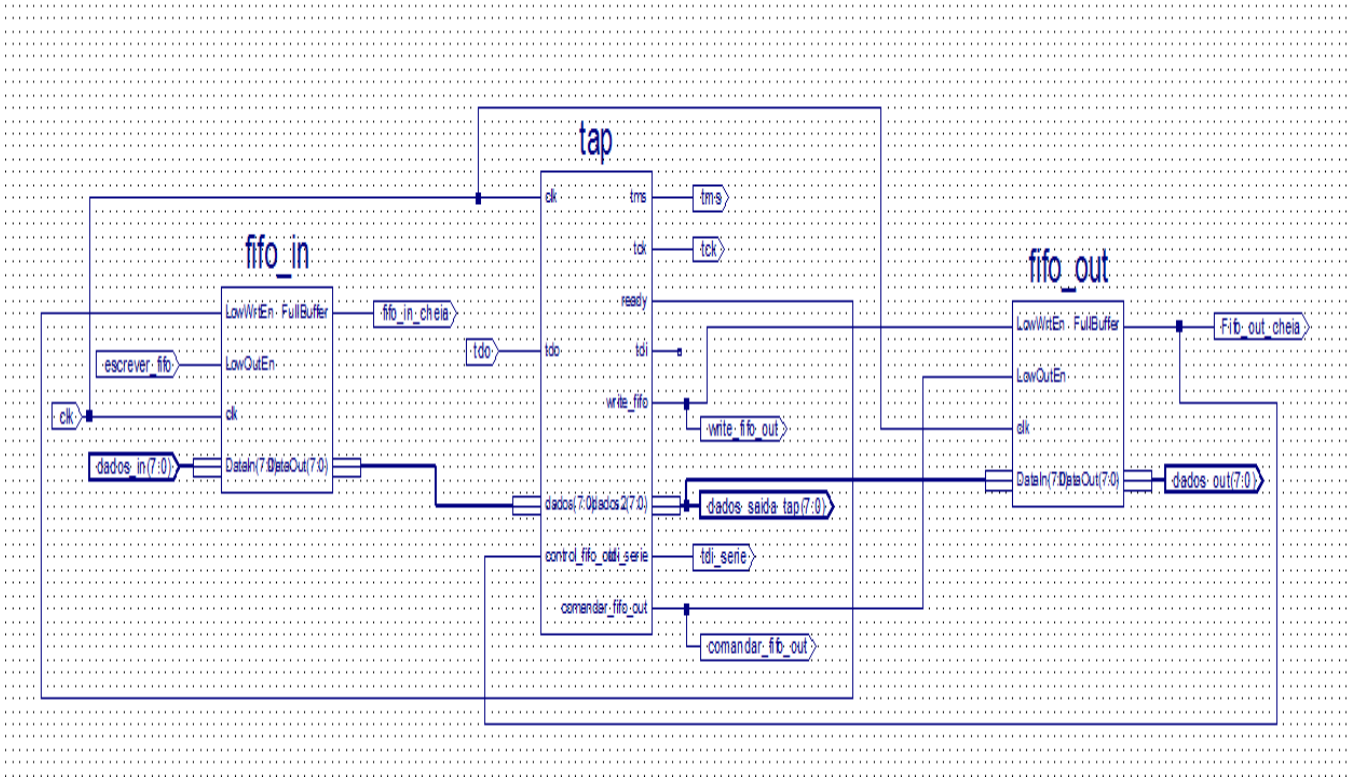


Figura 48 - Organigrama do controlador

8.3. Descrição do controlador BST

O controlador foi projectado de acordo com a infra-estrutura *Boundary Scan* e o seu funcionamento depende dos dados recebidos da FIFO_IN.

A FIFO é carregada uma vez em todo o processo. Esta operação de escrita é controlada pelo pino *LowWriteEnable*, sendo o utilizador o responsável por esta operação. O utilizador terá de enviar toda a informação das instruções de teste pretendidas e os dados a enviar ao PCB. A FIFO tem uma capacidade finita definida pelo programador e o utilizador não a pode alterar. Quando essa capacidade for excedida, o pino FIFO_CHEIA muda de estado lógico e fica associado a uma mensagem de erro.

O tamanho máximo de dados a enviar pelo *software* do PC é limitado a um conjunto de oito bits de cada vez, tamanho esse definido pelo barramento de ligação à FIFO. A capacidade de armazenamento definida para as FIFOs foi determinada pelas sequências de simulações a realizar pelo controlador; as FIFOs têm 21 posições, cada uma com oito bits. Depois de carregados todos os dados para a FIFO, o utilizador não interfere mais no processo de teste. É o controlador TAP, através do pino READY, que comanda o processo de leitura da FIFO e também todo o restante procedimento.

O controlador TAP será descrito em VHDL e implementado através de vários processos:

- Processo “Menu” - Descodifica os dados recebidos da FIFO_IN, que podem ser instruções ou dados a enviar ao PCB.
- Processo “STATE” – Efectua o deslocamento entre estados
- Processo “SDR” – Realiza a introdução de dados na linha TDI, quando posicionado nos registos de instrução ou dados (*ShiftIR* ou *ShiftDR*)
- Processo “TDI” - Atribui estimulos ao PCB através das linhas TDI, TMS, TCK
- Processo “TDO” - Recebe os dados obtidos do PCB através da linha TDO e envia-os para a FIFO_OUT

Estes processos representam blocos de lógica combinatória, que definem o deslocamento entre estados e o envio/leitura de sequências de dados ao PCB. A vantagem de ter múltiplos processos deve-se ao facto das instruções serem mais curtas; somente contêm os sinais de entrada necessários a esse processo, facilitando o controlo do mesmo.

O controlador foi projectado para percorrer o diagrama de estado da infra-estrutura BS, não sendo possível garantir um controlo total. Por exemplo, ocorrências de erros de comunicação que possam afectar o comportamento do controlador.

Atribuiu-se um estado inicial ao controlador definido pela norma (*TestLogicReset)*. Também a arquitectura e todos os parâmetros (portas, variáveis, sinais, …) foram especificados no início do *software*.

As sequências dos estímulos para o deslocamento entre estados foram definidas através de tabelas (*arrays*).

Proxima posição		Test logic-reset	Runtest -idle	SelectDR -scan	Captur e-DR	Shift-DR	Exit1-DR	Pause -DR	Exit2-DR	Updat e-DR	SelectIR-scan	Captur e-IR	Shift-IR	Exit1 -IR	Pause-IR	Exit2-IR	Update-IR
Posição actual		1111	1101	1100	1011	1001	1000	1010	0110	0111	1110	0000	0001	0010	0011	0100	0101
Test logic-reset	1111	-	0	10	010	0010	1010	01010	101010	11010	110	0110	00110	10110	010110	1010110	110110

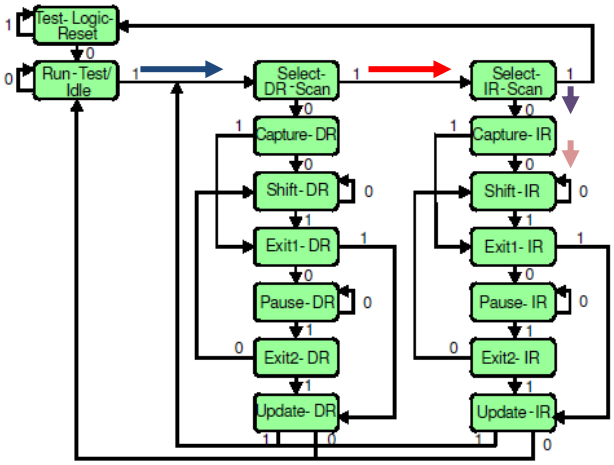


Figura 49 - **Exemplo:** Deslocamento entre o estado *TestLogicReset* e *ShiftIR*

A tabela mostra o percurso na tabela de estados e a sequência de estímulos necessários ao deslocamento entre o estado *TestLogicReset* e *ShiftIR,* dados através da saída TMS, com o sinal TCK activo.

Constant table_1: **test_logic_reset** := ("--------","-------0","------10","-----010","----0010","----1010","---01010","--101010","---11010","-----110","----0110","**---00110**","---10110","--010110","11010110","--110110");

Tabela 7- Transcrição da tabela para linguagem VHDL

Os *arrays* indicam quais os estímulos necessários para cada mudança de estados. O cálculo do deslocamento entre estados no controlador TAP é descrito numa estrutura tipo *case*; o estado seguinte está sempre dependente do anterior e é sempre realizado através do percurso mais curto. O deslocamento é efectuado por sequências de estímulos na linha TMS com a linha TCK a '0'.

O controlador somente executa uma instrução de cada vez; deslocamento entre estados ou envio/recepção de dados através das linhas TDI e TDO.

A validação do modelo desenvolvido será efectuada por ciclos sucessivos de aplicação de estímulos e captura de respostas. Para a simulação de teste pretendida, o controlador terá de fazer múltiplas passagens entre os estados *ShiftIR* (definir modos de teste) e *ShiftDR* (enviar dados de teste ao PCB) correspondentes ao diagrama de estados do controlador TAP.

8.4. Funcionalidade de FIFOs no controlador BST

8.4.1. FIFO_IN

Este bloco simulará o *software* do computador, enviará instruções de teste ou dados para testar o PCB. Terá o objectivo de armazenar, por ordem de entrada, dados enviados através de um barramento com oito bits. Terá uma extensão definida pelo utilizador que dependerá do modo de teste a realizar e dos ICs e das ligações do PCB a controlar.

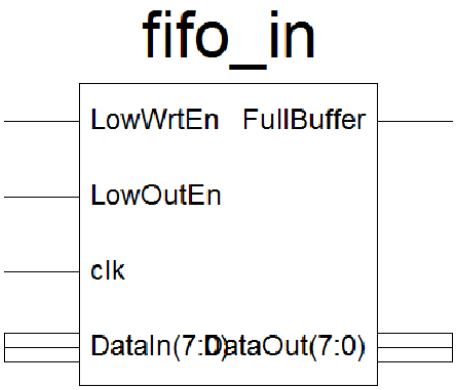


Figura 50 - Esquema de pinos da *FIFO_IN*

A FIFO é constituída por:

Dois sinais de controlo.

- *LowWrtEn* – este sinal será controlado pelo *software* do computador. Define juntamente com o sinal *LowOutEn* a tarefa a realizar, podendo ser de escrita ’1’ ou leitura ‘0’. Inicialmente está a ‘1’ e efectua o carregamento de todos os dados necessários para o teste escolhido. Depois coloca-o a ‘0’.
- *LowOutEn* – este sinal será dirigido pelo controlador TAP; juntamente com o sinal *LowWrtEn* define o processo a realizar pela memória (escrita ou leitura). Inicialmente esta a ‘0’ e quando pretendemos receber dados da memória, colocamo-lo a ‘1’.

Dois sinais de entrada.

- CLK – O relógio está sincronizado com os restantes blocos
- *DadosIn* – Barramento de oito bits; a sua sequência de valores controla toda a infra-estrutura BST, define o modo de teste a realizar, a quantidade de estímulos enviados ao PCB e o valor lógico dos mesmos.

Dois sinais de saída:

- *FullBuffer* – Indica a ocorrência de um erro e anuncia que a memória está cheia. Indica que o programador cometeu um erro na definição do tamanho da memória ou dos dados enviados.
- *DataOut* – Barramento de oito bits, terá a mesma sequência dos dados de entrada.

Descrição de funcionamento

LowWrtEn	***LowOutEn***	**Operação**
0	0	Escreve e lê ao mesmo tempo
1	0	Escrita na FIFO
0	1	Leitura da FIFO

A capacidade de armazenamento da memória é definida através de um *array*. Quando o valor é excedido, o pino *FullBuffer* muda de estado lógico, passando a '1'. Este terá ligação com o programa de *software* do computador e estará associado a uma mensagem de erro.

O pino *LowWrtEn* é controlado pelo *software* do computador; por omissão está a '1' definindo assim uma acção de escrita. Quando os dados estão todos armazenados passa a '0'.

O pino *LowOutEn* é gerido pelo controlador TAP (sinal *Ready)*; por omissão está a '0', passando a '1' quando pretendemos receber instruções ou dados da memória através do barramento *DataOut*.

O código e respectiva descrição estão disponíveis no Anexo1.

8.4.2. FIFO_OUT

Este bloco efectua o processo inverso da FIFO_IN e tem como objectivo armazenar os dados enviados pelo controlador TAP. Estes são o resultado dos testes efectuados ao PCB. O tamanho da FIFO_OUT é igual ao da FIFO_IN, sendo definida pelo programador.

O armazenamento na memória será efectuado por ordem de entrada. Quando cheia, activa o pino *FullBuffer* e indicará ao programa de *software* que nos próximos ciclos de relógio colocará os dados no barramento de saída.

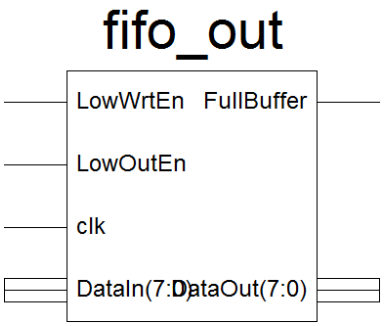


Figura 51 - Esquema de pinos da *FIFO_OUT*

A FIFO é constituída por:

Dois sinais de controlo

- *LowWrtEn e LowOutEn* – estes sinais são controlados pelo controlador TAP, definem se a tarefa a realizar é de escrita, leitura. Quando cheia coloca os dados sequencialmente na sua saída.

Dois sinais de entrada

- CLK – O relógio está sincronizado com os restantes blocos
- *DadosIn* – Barramento de oito bits. Este receberá os dados obtidos pelos testes efectuados ao PCB.

Dois sinais de saída:

- *FullBuffer* – Indica que a memória está cheia e anuncia ao programa de *software* que, nos próximos ciclos de relógio, colocará os dados no barramento de saída.
- *DataOut* – Barramento de oito bits e têm a mesma sequência dos dados de entrada armazenados.

Descrição de funcionamento

LowWrtEn	***LowOutEn***	**Operação**
0	0	Escreve e lê ao mesmo tempo
1	0	Escrita na FIFO
0	1	Leitura da FIFO

O código VHDL e respectiva descrição estão disponíveis no Anexo 4.

8.5. Controlador TAP

Este bloco controla os estímulos nas linhas TMS, TCK e TDI. Estes sinais, são os responsáveis pelo controlo da execução de teste pretendido a realizar no DUT.

Este bloco efectua também o controlo das FIFOs:

- Controla o processo de leitura de dados da FIFO_IN
- Controla o armazenamento dos dados recebidos no TDO na FIFO_OUT e quando cheia, coloca todos os dados armazenados no barramento de saída.

Com este procedimento visualizamos e analisamos os dados obtidos. Estes seriam enviados para um *software* no computador para analisar e determinar o estado da placa.

O módulo TAP contém:

Três sinais e um barramento de entrada

- DadosIn – Barramento de oito bits ligado ao barramento da memória de entrada, os dados recebidos podem ser instruções de teste ou dados a enviar para ao PCB
- TDO – Recebe os dados obtidos através do teste ao PCB
- CLK – Relógio sincronizado com os restantes blocos
- Control_Fifo_Out – Indica quando a memória de saída está cheia

Seis sinais e um barramento de Saída

- Ready – Responsável pelo controlo da memória de entrada juntamente com a linha *LowWrtEn*, dependendo do seu estado lógico activa o processo de escrita ou de leitura
- TMS - Sinal que define o modo de funcionamento do controlador
- TCK - Dependente do sinal de relógio, é activado para executar os testes pretendidos ou o envio de dados ao PCB

- DataOut – Barramento de oito bits, envia os dados obtidos no TDO para a memória de saída
- TDI_série – Linha responsável pelo envio de estímulos de teste ao PCB
- Write_Fifo_Out e Comamdar_Fifo_Out – Estas linhas são responsáveis pelo controlo de funcionamento da FIFO_OUT

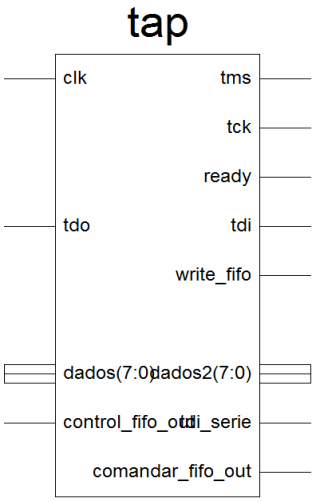


Figura 52 - Esquema de pinos do Controlador TAP

8.5.1. Descrição de programação do controlador TAP

Um controlador *Boundary Scan* possui na sua lógica interna de funcionamento dezasseis estados. Para implementar esta lógica foi necessário construir uma tabela de estados de quatro bits (2^4=16). Transpondo-a para o *software*, foi necessário criar dezasseis *arrays* cada um com dezasseis posições e cada posição composta por oito bits. Ver no Anexo 2.

As instruções utilizadas no controlador são simplificações das instruções propostas na linguagem SVF.

Definimos a posição inicial do controlador no diagrama de estados (*TestLogicReset*). Inicializamos as variáveis e respectivas especificações "entrada/saída" e tipo, na função *entity* e *arquitecture* do ficheiro VHDL.

O *software* inicia-se no processo MENU; este é o responsável por definir o próximo procedimento. Caso a sequência de dados seja:

- "11111111" – Instrução STATE, indica a necessidade de transitar de estado e encaminha para o processo STATE.
- "11111110" – Instrução SDR, encaminha para o processo SDR que é responsável pelo envio de dados através das linhas TDI, TCK e TMS. A próxima sequência

indicará a quantidade de dados a enviar ao PCB. As restantes sequências contêm os dados a enviar. Este procedimento ocorre quando posicionado no estado *SifhtIR ou ShiftDR.* Quando todo o processo for finalizado, regressa ao processo MENU para nova operação de teste.

- "Outras" – O controlador não efectua qualquer tarefa; fica bloqueado e precisa de ser reiniciado.

O processo STATE tem dois processos auxiliares:

- O "processo_proximo_estado" verifica a posição do estado actual e compara-o com o próximo. Este indica a posição do vector na tabela de estados e o número de ciclos TCK necessários para o deslocamento até ao estado pretendido.
- O "processo_tms" é responsável pela colocação de dados no pino TMS que são recolhidos da tabela de estados. Ver Anexo 2.

Uma vantagem do *software* surge na transição de estados; não é necessário inicializar o diagrama de estados (*TestLogicReset*), deslocamo-nos pelo trajecto mais rápido, ganhando em eficácia de processamento.

Quando um dos processos está activo STATE ou SDR, o outro tem obrigatoriamente estar inactivo.

O processo SDR é activo quando o processo STATE está inactivo e o barramento DADOS_OUT da FIFO_IN tem a sequência "11111111".

O processo SDR tem dois processos auxiliares:

- O "Processo_TDI" lê o contador (n.º de bits a enviar) de seguida, coloca os dados no pino TDI até o contador estar a '0'.
- O "Processo_TDO" verifica as entradas do pino TDO e insere os dados recebidos num vector para serem enviados para a FIFO_OUT.

Quando o "Processo_TDO" termina, envia os dados para a FIFO_OUT usando o processo "processo_fifo_out" e este faz o controlo dos pinos de escrita da FIFO. Quando a FIFO está cheia activa o pino de leitura, fazendo com que esta coloque os dados armazenados na sua saída. Este procedimento serve para testar o nosso *software* e simula os dados que seriam enviados para o *software* do computador.

O processo READY controla a FIFO_IN, quando activo esta coloca os dados na sua saída para posteriormente serem lidos e processados pelo controlador TAP.

O processo "processo_controlo_operação" é responsável pelo controle do processo STATE e SDR, quando um está activo o outro está inactivo e vice-versa

8.6. Sequências de teste

Iremos determinar a sequência de testes a realizar e consequentemente o número de estímulos necessários para testar o circuito. Para simulação, usamos um componente com um encapsulamento de 18 pinos.

Tamanho variável:
- Dados enviados: 54

Tamanho Fixo:
- Controlador TAP: 19
- Registo de instruções: 16
- Registo de Bypass: 1
- Perdas: 2

Total estímulos: 92

Tabela 8 - Tabela de estímulos enviados

Para a implementação da infra-estrutura *Boundary Scan* no circuito, a aplicação requer 92 estímulos para efectuar um teste às suas ligações externas. O total de estímulos necessários à realização do teste é dado por quatro tipos de estímulo diferentes; dados enviados, controlador TAP, registo de instruções e perdas.

Os dados enviados - Designam a quantidade de dados enviados ao PCB através da linha TDI. Têm três sequências diferentes, cada uma com dezoito bits. A primeira alterna entre 1's e 0's, a segunda alterna entre 0's e 1's, e a terceira tudo a 0's.

O Controlador TAP - Indica os estímulos necessários aos deslocamentos entre estados em todo o processo. São introduzidos na linha TMS.

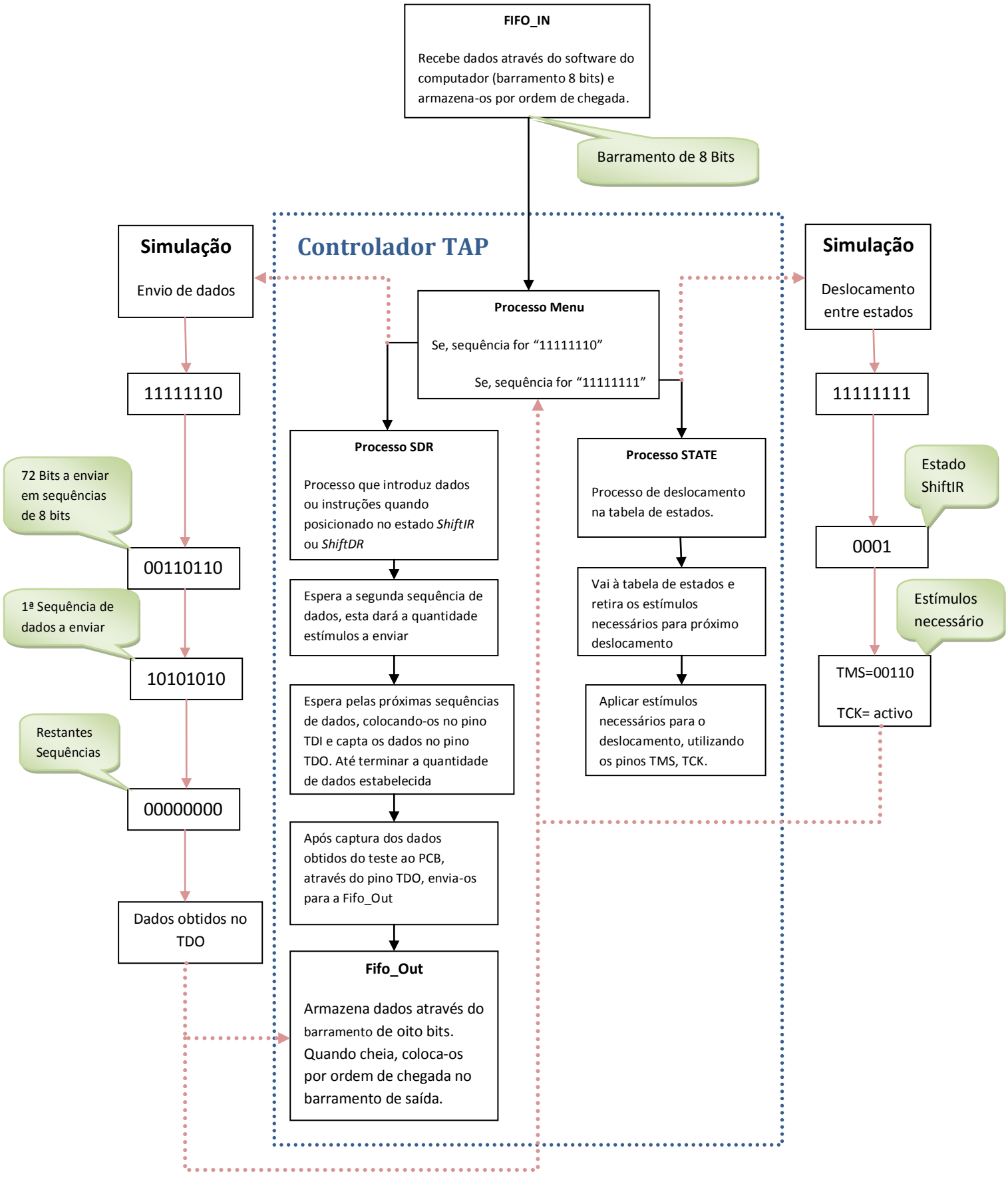
1.	*TestLogicReset -> ShiftIR*	"00110"	5 estimulos
2.	*ShiftIR -> ShiftDR*	"00111"	5 estimulos
3.	*ShiftDR -> ShiftIR*	"001111"	6 estimulos
4.	*ShiftIR -> RunTest/Idle*	"011"	3 estimulos

O registo de instruções - Determina os dados enviados quando posicionado no estado *shiftIR* (*extest, intest, bypass, …*), são responsáveis por definir o modo de teste a efectuar.

1.	*EXTEST*	"00000000"	8 estimulos
2.	*BYPASS*	"11111111"	8 estimulos

Analisando a tabela, verificamos que na totalidade de estímulos enviados há uma taxa de perdas de aproximadamente 2%. Para executar o teste ao PCB são necessários 54 bits, divididos em três sequências diferentes, cada uma com 18 bits. Para o envio total, o controlador tem de enviar 7 sequências com 8 bits cada, o que totaliza 56 bits. Como apenas necessitamos 54 bits, dois dos bits serão desperdiçados.

Procedimento de programação



O controlador inicialmente posicionar-se-á no estado *TestLogicReset*, vamos deslocar-lo até ao estado *ShiftIR*, para esse deslocamento inserirmos estímulos nos pinos TMS e activamos o TCK.

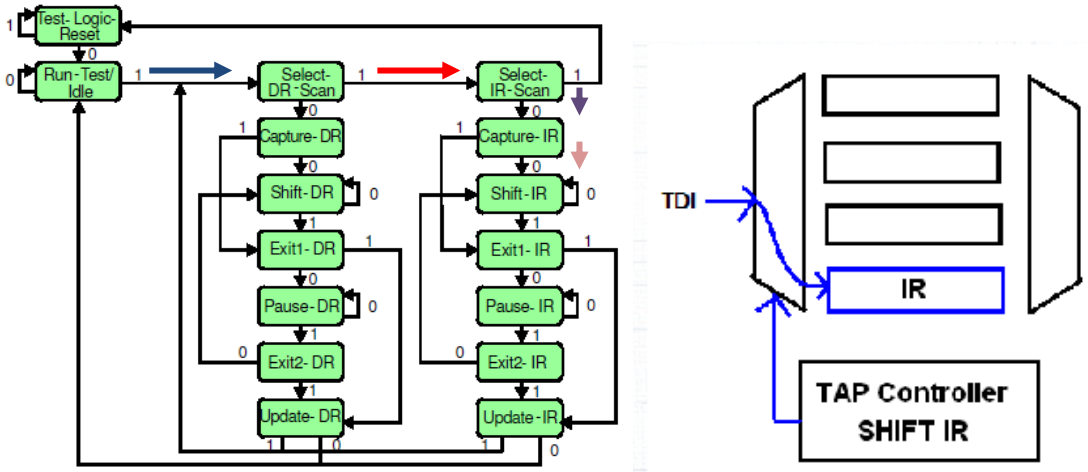


Figura 53 - Transição entre estado *TestLogicReset* e *ShiftIR*

O procedimento seguinte é enviarmos uma sequência de oito bits a '0'. Esta sequência é responsável por colocar o componente a testar em modo *Extest.* É necessário mencionar que o tamanho do registo de instruções varia entre componentes, o componente simulado utiliza um registo de instruções de 8 bits.

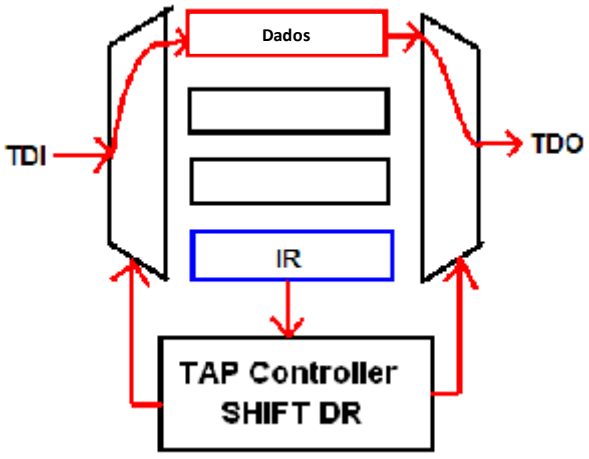


Figura 54 - Carregar registo de Instruções

Definido o modo de teste, deslocamo-nos até ao *ShiftDR* no diagrama de estados, inserindo estímulos nos pinos *TMS* e activamos o *TCK*.

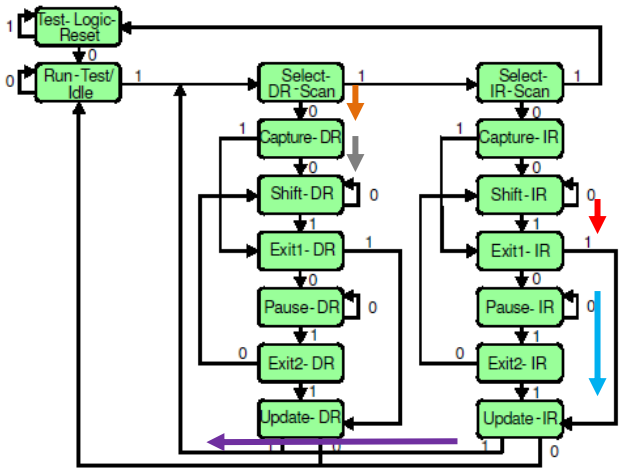


Figura 55 - Transição entre estado *ShiftIR* e *ShiftDR*

Este estado é responsável pelo deslocamento dos dados através das células BS anexadas aos pinos do componente, aplicando estímulos no *TDI*, activamos o *TCK* e mantemos o *TMS* a '0'.

Neste estado enviamos três sequências com dezoito bits cada, estas auxiliarão na detecção de falhas entre as ligações ao PCB, de acordo com a explicação dada na secção 3.11.

- A primeira inicia-se alternando 1's e 0's
- Na segunda inserimos a mesma sequência mas de forma inversa
- Na terceira sequência enviamos tudo a 0's, preparação para novo teste.

Terminado o envio das sequências de dados, deslocamos novamente para o estado *ShiftIR* e escolhemos uma nova instrução de teste, modo *BYPASS*.

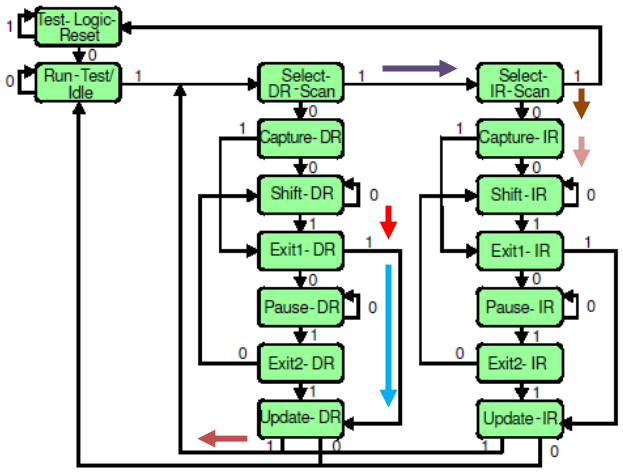


Figura 56 - Transição entre estado *ShiftDR* e *ShiftIR*

Para finalizar a simulação colocaremos o programa no estado *RunTest/Idle,* colocará o controlador preparado para novo teste. Esta operação destina-se a permitir a execução de instruções de auto-teste (e.g. *Runbist*).

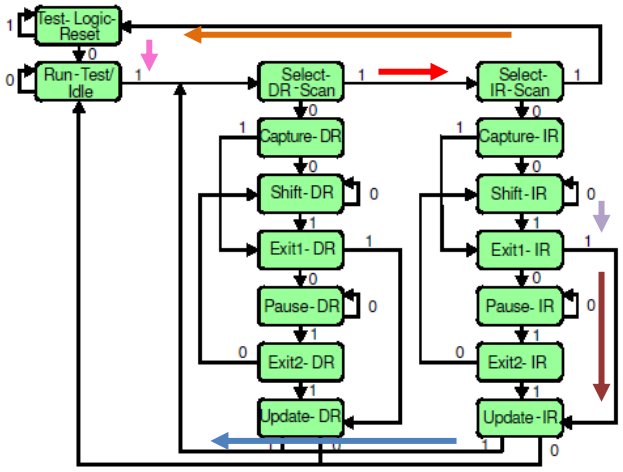


Figura 57 - Transição entre estado *ShiftIR* e *TestLogicReset*

Tabela descritiva

	Estado	Dados	Operação
1.	*Shift IR*	00000000	*EXTEST*
2.	*Shift DR*	101010101010101010	
3.	*Shift DR*	010101010101010101	
4.	*Shift DR*	000000000000000000	
5.	*Shift IR*	11111111	*BYPASS*
6.	*RunTest/Idle*	110	

Para visualização detalhada da sequência realizada pelo *software*, ver Anexo 5.

8.6.1. Inserir dados na FIFO_OUT

Iniciamos a simulação carregando os dados para a memória FIFO-IN, necessitamos de colocar a entrada do ESCREVER_FIFO um ’1’, de seguida os dados serão inseridos no barramento DADOS_IN (oito bits).

Como anteriormente referido o tamanho da memória FIFO_IN é definido no seu *software* pelo programador. Quando cheia passa a saída FIFO_IN_CHEIA para o estado lógico ‘1’. Alertando para a ocorrência de um erro.

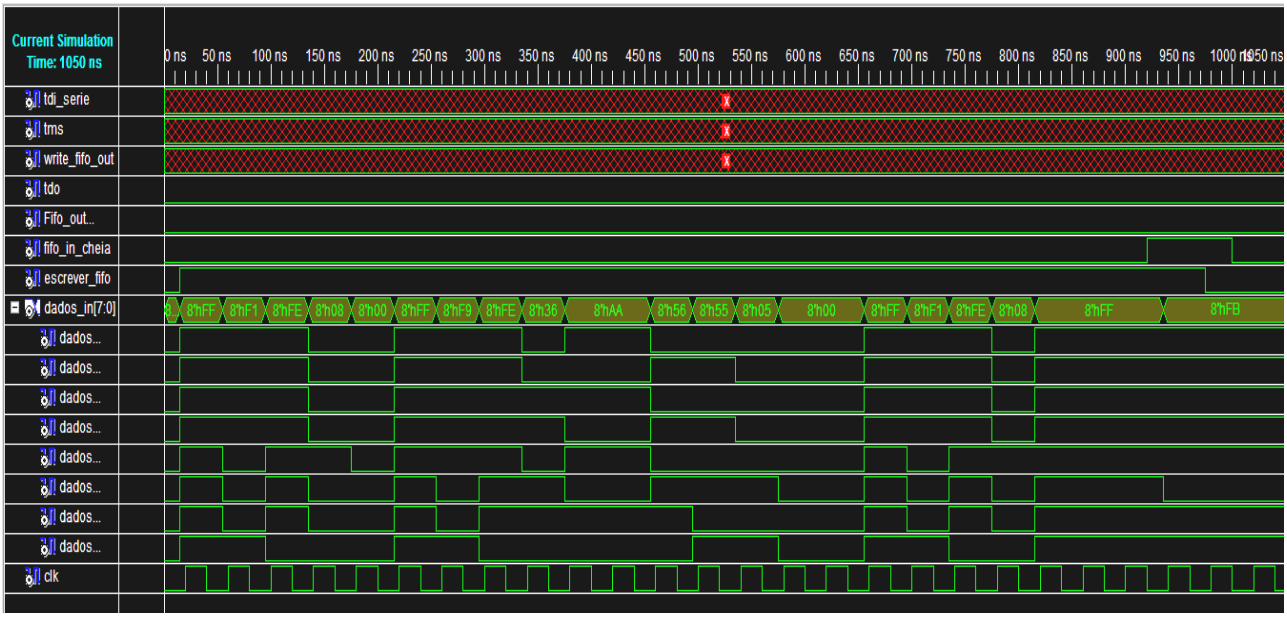


Figura 58 – Escrever dados no módulo FIFO_IN

A figura mostra a sequência de dados enviados à memória através do barramento Dados_IN, podemos visualizar o funcionamento dos sinais que controlam a memória, escrever_fifo e Ready.

Para análise do *software* introduzimos uma sequência a mais do que a definida no tamanho da memória, verificamos que activou o sinal Fifo_In_Cheia. Constatou-se assim que o *software* funciona correctamente.

Esta sequência está dependente do tipo de teste que pretendemos realizar ao PCB e do número de IC que queremos testar e das ligações que existem na placa.

Após carregar a memória, a leitura é efectuada através do comando Ready, este inicialmente está sempre a ‘0’ quando carregamos a memória. Quando o sinal passa a ’1’ e com o pino ESCREVER_FIFO a ‘0’, a FIFO_IN coloca os dados no barramento de saída DATA_OUT para ser lido posteriormente pelo controlador TAP.

O controlador TAP lê os dados obtidos e analisa a sequência recebida.

1.1. Caso tudo 1's indica que é a próxima operação será de deslocamento entre estados, o processo STATE permite o deslocamento até á posição desejada através do diagrama de estados.

1.1.1. O controlador inicializa-se sempre no estado *TestLogicReset* conforme infra-estrutura BST.

1.1.2. Quando pretendemo-nos deslocar para outro estado, o *software* analisa a tabela de estados construída e determina o percurso mais curto, tornando-o mais eficiente.

1.2. Caso a sequência seja "11111110" indica que a próxima operação é de envio de dados de teste ao PCB.

1.2.1. O controlador fica à espera pela segunda sequência, esta indicará a quantidade de bits que serão enviados ao PCB

1.2.2. As próximas sequências são os dados a enviar ao PCB

8.6.2. Deslocamento para o estado *ShiftIR*

Esta sequência de simulação, consiste no deslocamento até ao estado *Shift IR*. Para isso necessitamos de uma sequência de estímulos na saída TMS de "01100", com o sinal TCK activo.

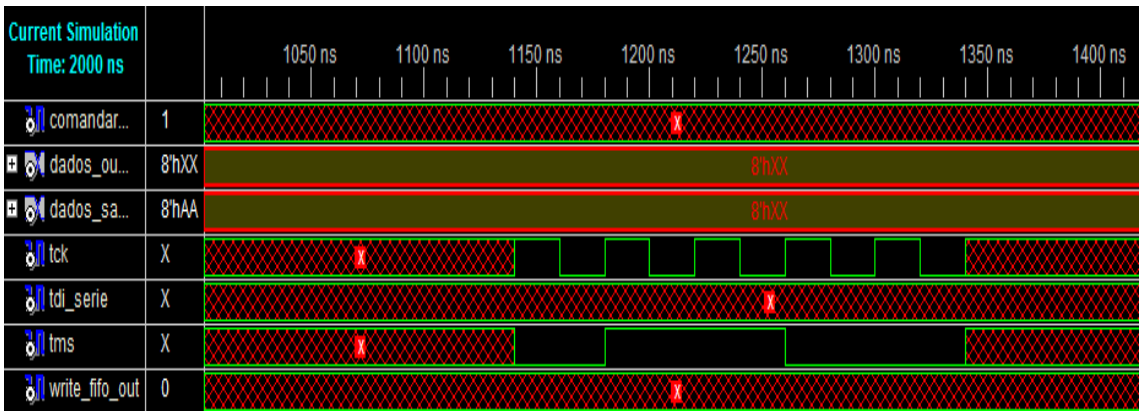


Figura 59 – Deslocamento *ShiftIR*

8.6.3. Colocar em *ExternalTest*

Situados no estado *ShiftIR* inserimos uma sequência de oito 0's, esta determinará que o componente funcionará em modo *External Test*.

Para uma sequência correcta temos os seguintes requisitos; sinal TCK activo, sinal TMS a '0' e na saída TDI_SERIE são colocados os oito bits em série.

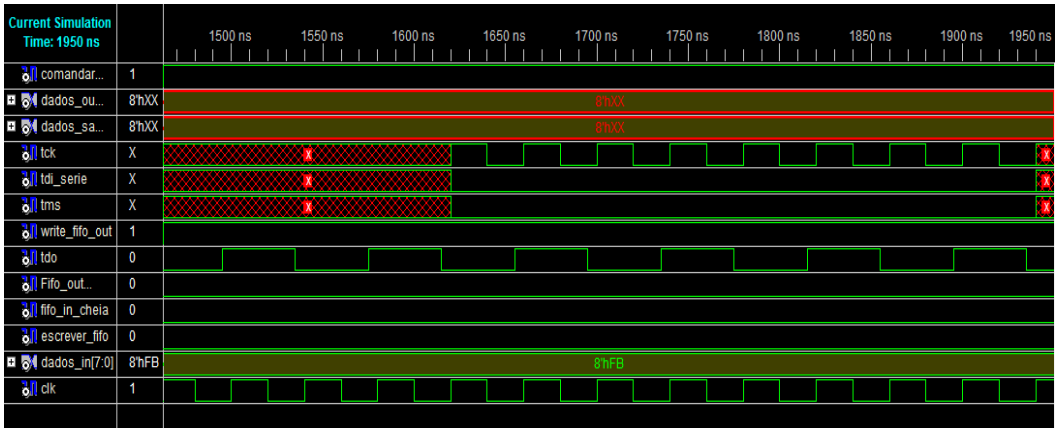


Figura 60 – Colocar componente em *External Test*

8.6.4. Deslocar para o estado *ShiftDR*

Após definir o modo de funcionamento de componente, efectuamos o deslocamento para o estado *ShiftDR,* neste estado inserimos os dados que irão percorrer todos os pinos do componente e respectivas ligações.

Para efectuar o deslocamento necessitamos de enviar a seguinte sequência de bits; TMS com "11100" que é calculada automaticamente pelo controlador e com o TCK activo.

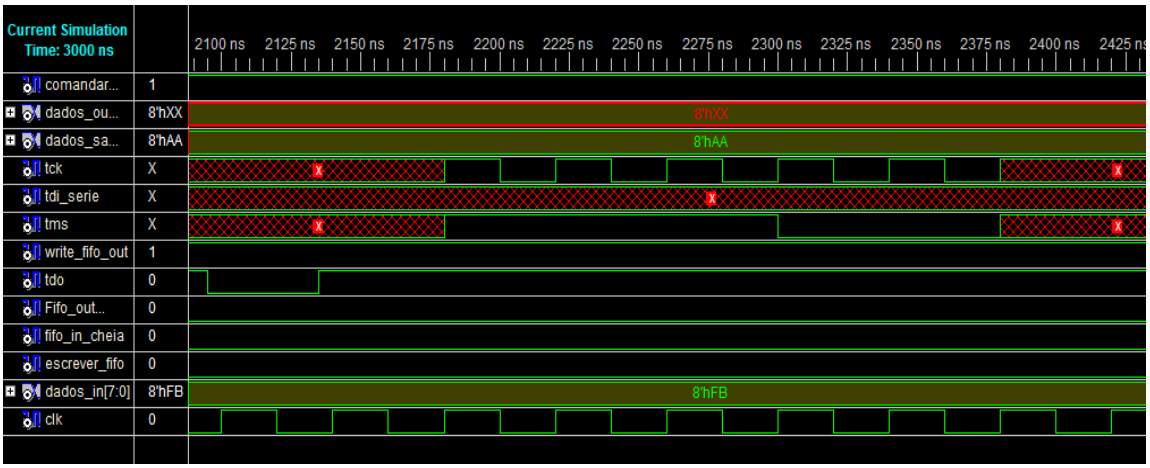


Figura 61 – Deslocamento para estado *ShiftDR*

Quando posicionados no estado *ShiftDR*, introduzimos os dados que vão percorrer todo o circuito e posteriormente recebidos no pino TDO. Após o seu armazenamento far-se-á uma comparação e análise. Se os dados forem coincidentes entre a saída TDI_SERIE e a entrada TDO, concluiremos que a placa está em perfeitas condições, senão teremos que analisar caso a caso.

Os dados são introduzidos em três etapas, cada uma com dezoito bits:

- A primeira consiste em alternar os sinais entre 0's e 1's
- A segunda é o inverso alternar sinais entre 1's e 0's
- A terceira somente com 0's

Os dados enviados nas duas primeiras etapas têm como finalidade auxiliar na verificação e detecção de falhas quer no PCB ou nos ICs que a compõem; curtos, soldas frias, filetes interrompidos ou falhas entre camadas.

A terceira etapa colocará o circuito pronto para futuros testes.

8.6.5 Inserir 1ª sequência de dados ao PCB

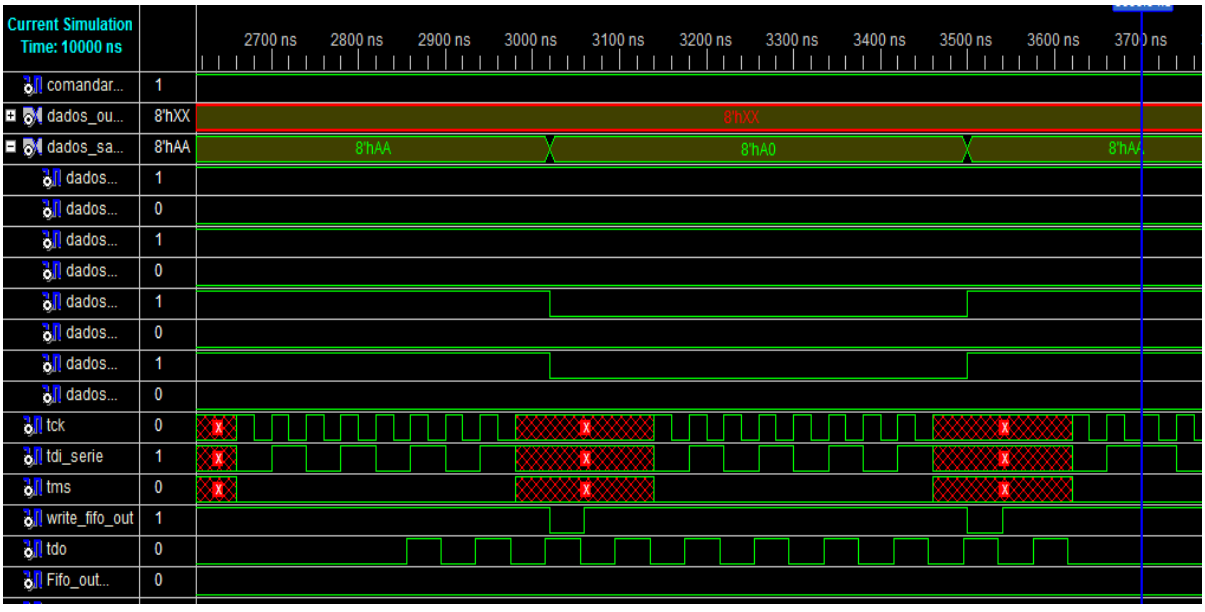


Figura 62 – Primeira série de Dados

Na figura podemos visualizar o envio da primeira sequência de dezoito bits alternando-os entre 0's e 1's na transição positiva do TCK. Após finalizar a primeira sequência seguidamente inicializar-se-á a segunda com uma sequência inversa, alterna entre 1's e 0´s.

Os dados são colocados em série na saída TDI_SERIE e são lidos na entrada TDO do controlador. São guardados num vector que posteriormente enviá-los-á para a memória FIFO_OUT, podemos visualizar os dados recebidos na variável *"Dados_saida"* e analisar se o *software* está a funciona correctamente.

A saída "*Write_Fifo_Out"* é responsável pelo controlo do processo de *escrita* na memória FIFO_OUT, quando está a '0' a memória armazena os dados da saída "*Dados_Saida".*

Os dados na linha TDO foram manualmente introduzidos, simulam o resultado obtido a partir do PCB. No futuro serve para analisar os resultados do teste de componentes electrónicos.

8.6.6 Inserir 2ª sequência de dados ao PCB

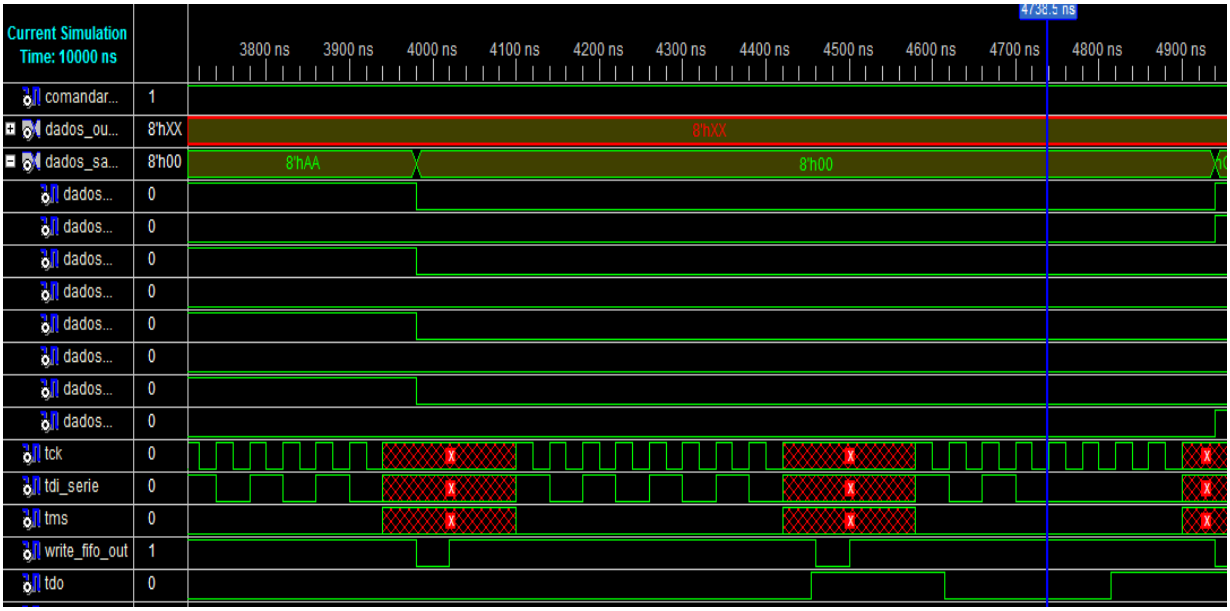


Figura 63 – Segunda série de Dados

Na figura visualizamos a segunda sequência de dezoito bits, novamente o TCK tem de estar activo e o TMS a '0'.

8.6.7 Inserir 3ª sequência de dados ao PCB

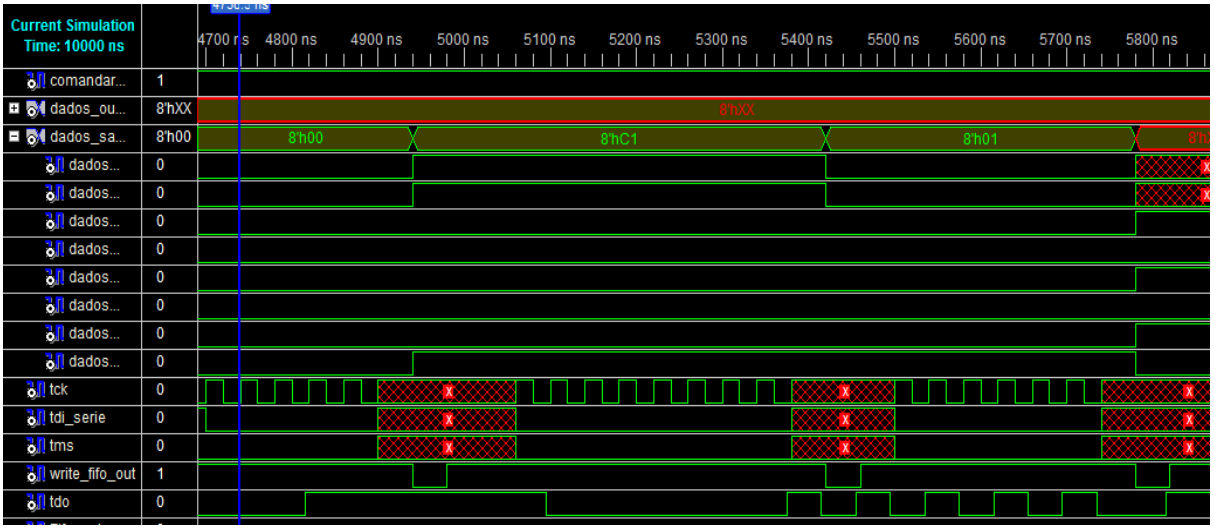


Figura 64 – Terceira série de Dados

Terceira sequência de dezoito bits de 0's tem como finalidade de deixar as entradas dos componentes todas a 0's para futuros testes.

Depois de concluída toda a simulação do modo *ExternalTest*, vamos gerar um teste *BYPASS*. Temos a necessidade de deslocar-nos novamente ao estado *ShiftIR* e introduzir nova sequência com oito bits de 1's.

Para este deslocamento até ao estado *ShiftIR*, introduzirmos a sequência "111100" este é o trajecto mais curto calculado pelo controlador.

8.6.7 Deslocamento para o estado *ShiftIR*

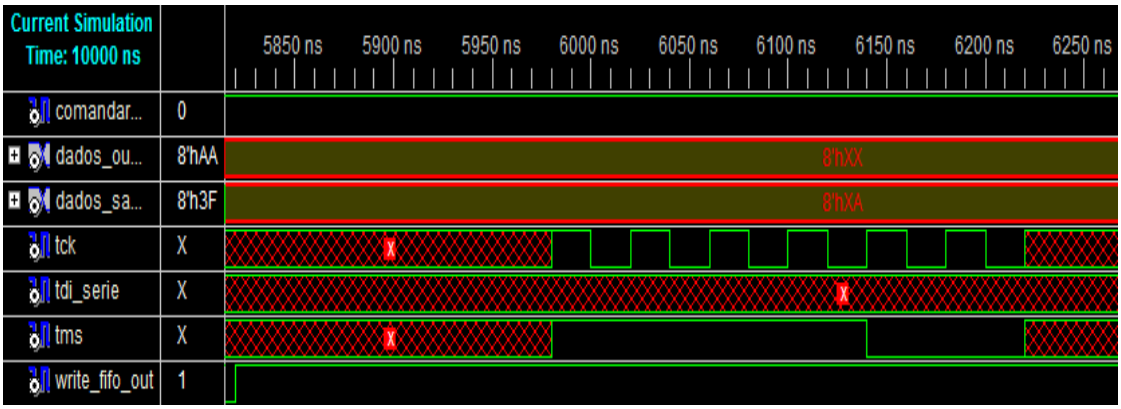


Figura 65 – Deslocamento para estado *ShiftIR*

Para nos deslocarmos temos o TCK activo e a referida sequência de bits no TMS. Quando posicionados no estado *ShiftIR* inserimos oito bits de 1's referentes ao modo *BYPASS*.

8.6.8 Colocar em *Bypass*

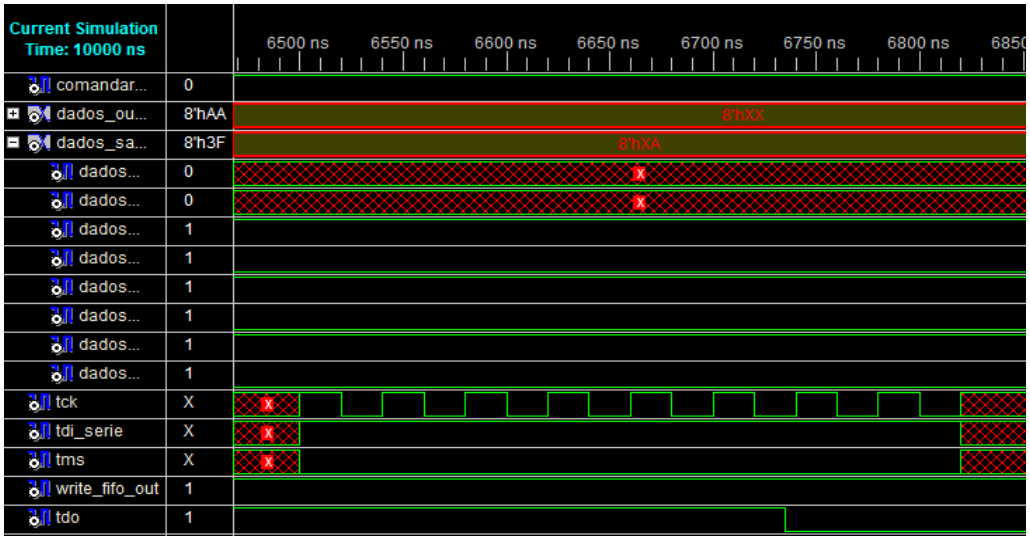


Figura 66 – Sequência de Dados de oito 1's

Repetimos novamente a sequência de envio de dados através da saída TDI_SERIE, que serão lidos posteriormente na entrada TDO, guardados num vector e enviados para a memória FIFO_OUT, para posterior análise.

Após efectuado o teste *Bypass*, vamos colocar o controlador TAP na posição inicial, fazendo o deslocamento através do diagrama de estados até á posição *RunTest/Idle*, ficará assim preparado para um auto-teste.

8.6.9 Deslocamento para o estado *RunTest/Idle*

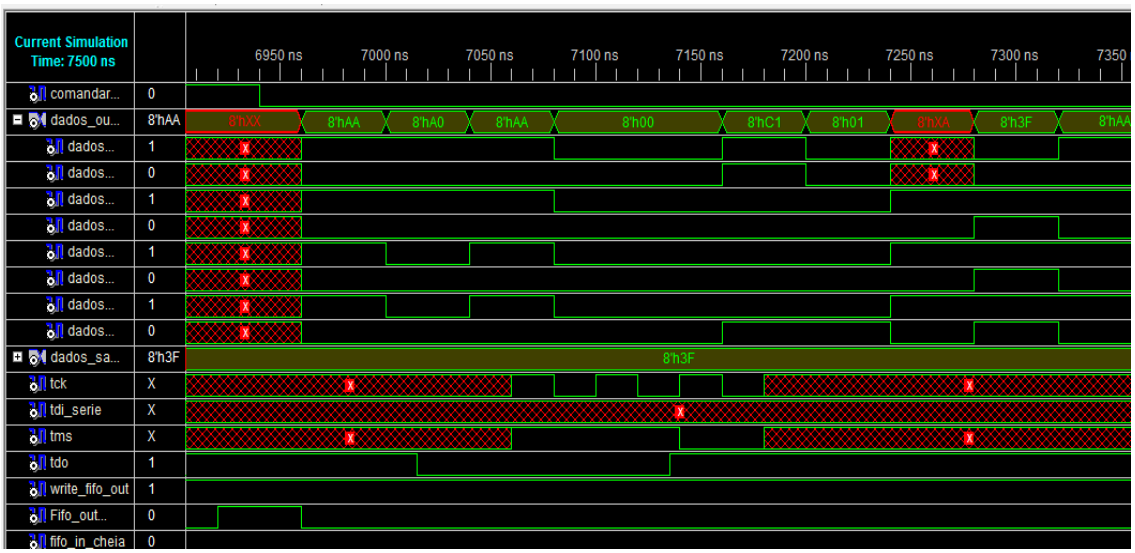


Figura 67 – Deslocamento para estado *RunTest/Idle* e DADOS_OUT

Para efectuar o deslocamento até ao estado *RunTest/Idle* é necessária uma sequência "011" na saída TMS, o TCK activo.

A figura mostra também o funcionamento da FIFO_OUT, esta memória foi definida para armazenar os dados recebidos do controlador TAP, sinais esses recebidos na linha TDO. Os dados TDO são simulados, representam os futuros dados a receber do PCB a controlar.

Quando a memória está cheia é activada a saída FIFO_OUT_CHEIA, o controlador acciona a sequência de leitura da memória, temos então todos os dados anteriormente armazenados no barramento de saída da memória, DADOS_OUT.

Podemos verificar que a saída da memória FIFO_OUT_CHEIA foi activada, indicando que está cheia. O controlador coloca a entrada COMANDAR_FIFO_OUT a '0', a memória inicia o ciclo de colocar os dados armazenados na saída DADOS_OUT, todos pela ordem de entrada.

Este procedimento serve de simulação, os dados que serão enviados para o *software* do computador.

8.7 Simulação Completa

Concluída a explicação detalhada de cada processo, visualizamos a simulação na sua totalidade.

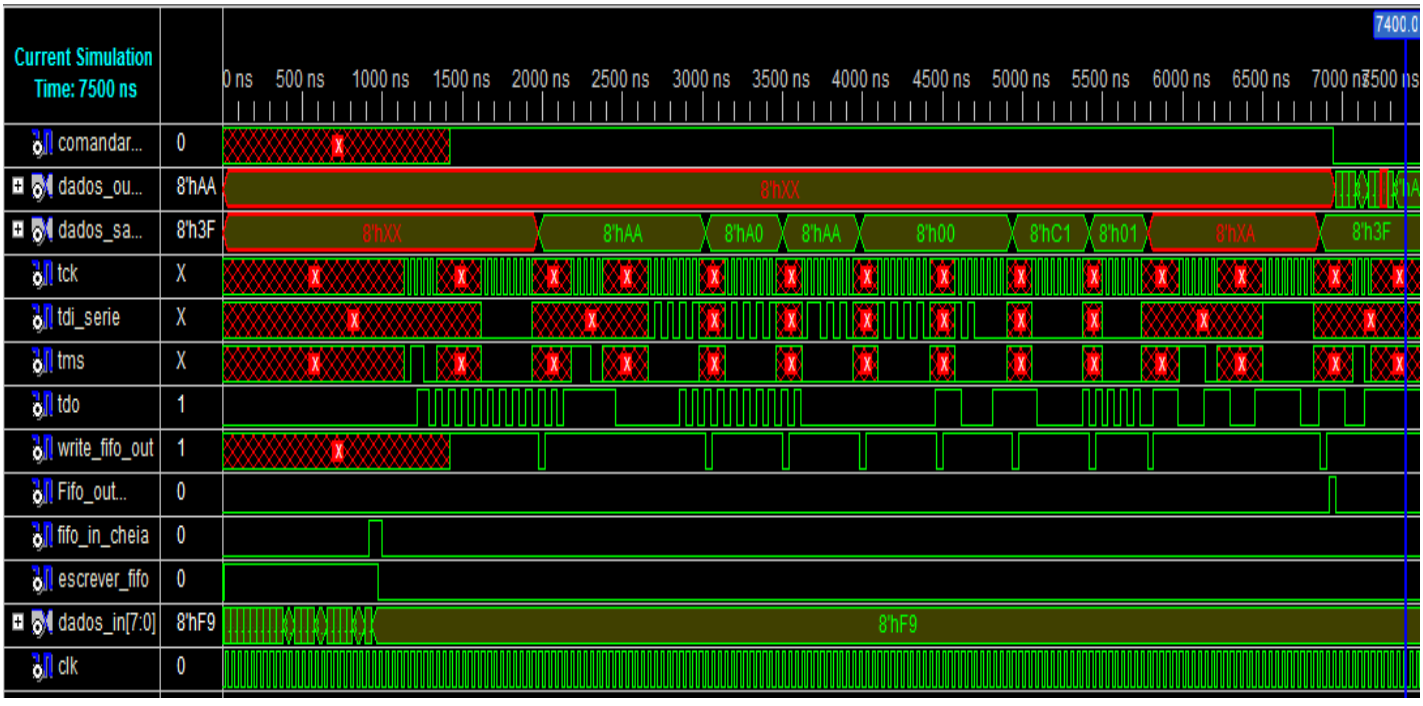


Figura 68 - Todas as sequências de teste

Na figura acima visualizamos a sequência total dos testes efectuados e dos dados enviados ao PCB. Constatamos que o *software* é eficaz, envia todas as sequências de estímulos necessárias à realização do controlo do PCB. Constatamos também que este não é eficaz, porque existem alguns ciclos de relógio desaproveitados, que tornam o *software* mais lento.

Este desaproveitamento é devido:

- Na transição entre processos, que não consegui sincronizar, especificamente entre o processo de envio e recepção de dados ao PCB, nessa transição são desperdiçados dois ciclos de relógio.
- Na sequência de dados a enviar ao PCB. A primeira sequência recebida da memória indica a quantidade de dados a enviar ao PCB e tendo de esperar pelas restantes sequência de dados, estas contêm os dados a enviar ao PCB. Nesta espera é desaproveitado um ciclo de relógio.
- No deslocamento entre estados. Após finalizar o processo de envio de dados, esperamos a próxima etapa, esta será de deslocamento no diagrama de estados. Para este procedimento temos de receber a sequência “11111111”, define a operação de deslocamento, esperamos então pela segunda sequência de dados, esta definirá o próximo estado. Neste procedimento são desaproveitados três ciclos de relógio.

Descrição dos programas em VHDL de cada módulo estão disponíveis nos Anexos 1,3 e 4.

9. Conclusões

O projecto proposto e desenvolvido nesta dissertação teve por objectivo desenvolver um controlador BST descrito em linguagem VHDL com a finalidade de realizar o controlo de PCBs e ICs que as constituem, baseado numa infra-estrutura *Boundary Scan*.

A execução do trabalho permitiu concluir que as vantagens da utilização do método de teste *Boundary Scan* superam largamente as suas desvantagens. A rapidez de criação e execução dos testes, *layouts* mais simples, o seu rápido diagnóstico de teste contribuem para uma rápida entrada do produto no mercado, superando a sua principal desvantagem, o aumento de preço dos componentes.

Com a crescente disponibilidade de dispositivos compatíveis com a norma IEEE 1149.1, maior è o potencial para testar placas electrónicas e para o desenvolvimento de aplicações *In-System Programming* (ISP) para programar memórias FLASH.

Comparativamente com outras metodologias de teste (*ICT, Flyprobe e Raio X*) o *Boundary Scan* revela-se uma alternativa mais barata e aquela que obtém genericamente melhores resultados, como observamos anteriormente na secção 3.14. Para garantir uma cobertura de falhas de 100% seria necessário conciliar todos estes testes, tornando-se inviável devido ao seu elevado custo e tempo de controlo.

No futuro os projectos de componentes electrónicos tenderão a ser descritos num nível de abstracção mais elevado, recorrendo por isso cada vez mais ao uso de linguagens de modelização de *hardware*.

As FPGAs são dispositivos reconfiguráveis de alto desempenho, permite fazer modificações futuras de modo mais rápido e eficiente, oferece flexibilidade, personalização das I/O e a possibilidade de programação ISP através dos pinos *Boundary Scan*.

Ao contrário de outras linguagens de descrição de *hardware*, que em geral são proprietárias de determinados fabricantes, o VHDL é público, o que garante a sua independência face às políticas de cada fabricante. A própria definição da linguagem envolveu a participação dos utilizadores, o que garante ao VHDL venha a ser cada vez mais usado na prática.

No trabalho descrito nesta tese a aprendizagem da linguagem VHDL foi uma grande mais-valia, porque somente foi necessário aprender uma única linguagem para todas as fases do projecto (simulações, síntese, …). Assim foi possível reduzir o tempo de desenvolvimento final do projecto.

Durante o desenvolvimento do *software* para o controlador, foram sendo testados separadamente todos os módulos envolvidos, criando-se simulações para todos os processos separadamente, eliminando assim praticamente todos os problemas ainda durante o seu desenvolvimento.

O controlador é dividido em três blocos distintos que interagem entre eles; FIFO_IN (memória de entrada), Controlador TAP (controla modo de teste e envio e recepção dos dados ao PCB) e FIFO_OUT (memória de saída).

A validação do modelo desenvolvido foi das partes mais importante do desenvolvimento do projecto, nesta fase foram analisados todos os sinais que o constituem e verificado o correcto funcionamento de todos os módulos.

Concluímos que o *software* funciona correctamente, de acordo com os requesitos exigidos. Executa o envio de todas as sequências de estímulos pretendidos e necessários à realização do controlo ao PCB. Armazena os dados obtidos através dos testes de controlo realizados, enviando-os posteriormente para a memória de saída.

Constatamos a necessidade do controlador ser optimizado, existem vários ciclos de relógio desaproveitados, tornando-o assim mais lento, como observado pormenorizadamente na secção 8.6.3.

10. Perspectivas de trabalho futuro

Dentro do âmbito desta dissertação e do seu objectivo inicial, alguns pontos merecem continuidade de investigação, no sentido de melhorar e aprofundar as soluções encontradas, bem como incorporar novos elementos de teste que possam aumentar as funcionalidades do controlador bem como o seu desempenho e facilidade de utilização.

Futuramente deve ser elaborado um *software* que faça a conexão do controlador a um sistema de gestão de testes, este deverá utilizar a linguagem SVF, e comunicaria com o controlador utilizando um porto normalizado da FPGA onde este fosse implementado (e.g. Ethernet, USB, RS232).

Desenvolver uma interface programável que automatize todo o procedimento de configuração de testes, acrescentando um suporte de arquivos SVF que permita enviar instruções de programação como também funções de teste aos dispositivos. Estes arquivos serão convertidos em ficheiros de teste utilizados pelo *software* do controlador BST.

O *software* deverá as seguintes valências

- Mostrar o estado que se encontra o controlador do TAP.
- Mostrar ao utilizador os sinais dos quatro pinos do controlador TAP.
- Conter a visualização e análise dos dados obtidos e listagem de defeitos.

Optimizar o código de forma a permitir a síntese em diversas FPGAs.

Inserir o *software* desenvolvido numa carta BST a fim de testar o controlador proposto com equipamento real, em cenários operacionais.

Num contexto produtivo, futuramente deve ser implementada uma ligação á internet para que o técnico responsável pela reparação possa, na sua bancada de trabalho e pelo número do PCB, ter acesso a todos os defeitos detalhados para posterior reparação. Com este método será possível implementar este controlador numa linha de produção.

11. Referências

[Agilent10] Agilent Technologies BSDL verification service, http://www.agilent.com/see/bsdl_service, 2010

[Ashenden90] Peter J. Ashenden, The VHDL Cookbook First Edition July, 1990

[Asset00] Boundary-Scan Tutorial, http://www.asset-intertech.com, 2010

[Asset99] ASSET InterTech, Inc, Serial Vector Format Specification, 1999

[Asset09] http://www.asset-intertech.com/connect/2009q1/1149dot7.htm, 2010

[Cammon] Brendan Bridgford e Justin Cammon, SVF and XSVF File Formats for Xilinx Devices, 2010

[Doro 04] Marinovic Doro, Qualidade em empresas montadoras de placas de circuito impresso, 2004

[Espinosa09] Ruben Dario Cardenas Espinosa, Curso FPGA (Programacion de arreglos de compuertas), 2009

[Favela] Eduardo Favela, Benefits of VLSI Boundary-Scan Testing, Texas Tech University

[Ferreira92] José Martins Ferreira, Introduction to Design for Test Techniques, FEUP, 1992

[Flores] Paulo Flores, Aplicação da Linguagem VHDL, INESC

[Goepel09] Goepel electronic GmbH, Boundary Scan Controller, 2009

[Goepel10] http://www.goepel.com/eng/bsc/, 2010

[Guilhermino] Manoel E. de Lima, Paulo S. Nascimento, Abel Guilhermino, FPGAS dinamicamente reconfiguraveis

[Gutierrez] MELO, P. R. ; RIOS, E. D. ; Gutierrez R., Placas de Circuito Impresso, http://www.bndes.gov.br/conhecimento/publicacoes, 2010

[IEEE01] Test Technology, IEEE Standard Test Access Port and Boundary-Scan Architecture, 2001

[Kharagpur] Kharagpur, Testing of Embedded System, Version 2 EE IIT

[Ledden03] LEDDEN, PCB Test Strategies, PennWell Corporation, 2003.

[Mckenzie03] J. MCKENZIE, Test and Inspection, PennWell Corporation, 2003.

[Mentor06] Boundary Scan Process, Guide Mentor Graphics Corporation, 2006

[Monteiro] Pedro Monteiro, Uma biblioteca VHDL para controladores BST, Universidade Porto, 2008

[Moraes] Prof. Fernando Moraes, Linguagem de Descrição de *Hardware* VHDL

[Ojeda 07] Luis Jacobo Álvarez de Ojeda, Verificación de sistemas digitales, 2007

[Perry04] Douglas L. Perry, VHDL: Programming by Example, Fourth Edition, McGraw-Hill, 2004

[Pessoa08] João Pessoa, Desenvolvimento e implementação em FPGA, 2008

[PLD] BS example, http://www.pld.ttu.ee/applets/bs/bs.html, 2010

[Ramos] Caio Ramos, Boundary Scan IEEE 1149.1

[Rodriguez07] Guillermo Güichal Gastón Rodriguez, Emtech, 2007
http://www.emtech.com.ar

[SCIS08] Standards Committee of the IEEE Computer Society, 2008

[SPEA09] http://www.spea.com/, 2009

[Texas97] Texas Instruments, IEEE 1149.1 Boundary Scan, 1997

[UFI] Tutorial VHDL, Universidade Federal de ITAJUBÁ

[Velho] Vitor C. F. Gomes, Andrea S. Charão, Haroldo F. C. Velho, Field Programmable Gate Array – FPGA - Mini-curso de Computação Híbrida Reconfigurável

[Wiki10] Wikipédia VHDL - http://pt.wikipedia.org/wiki/FPGA, 2010

[Wiki11] Wikipédia VHDL - http://pt.wikipedia.org/wiki/VHDL, 2010

[Xilinx10] Xilinx ISE 10 Tutorial, 2010

[Zeroplus] JTAG Measurement and Analysis, http://www.zeroplus.com.tw, 2010

12. Anexo 1

O *Software* desenvolvido para o armazenamento dos dados de entrada FIFO_IN, desenvolvido através de linguagem VHDL e simulado no *software* disponibilizado pela Xilinx versão ISE 10.

A descrição do programa em VHDL tem sempre a mesma sequência em todos os módulos; Fifo_IN, Fifo_Out, TAP.

Para descrever circuitos em VHDL devemos definir quais são as portas de entrada e saída e qual será o seu comportamento. Nesse contexto surgem as entidades "*entity* " e arquitecturas do circuito "*architecture"*.

A entidade é um espaço no código que descreve os sinais de entrada e de saída, definindo assim, o modo de comunicação do componente com o mundo exterior.

A organização de uma *architecture* é dada por:

Arquitectura: indica qual o comportamento do componente de hardware, parte que guarda o algoritmo interno do circuito.

Declarações:

As variáveis a utilizar devem ser declarados e definidas ("*CONSTANT"*, "*VARIABLE"* e "*SINAL")* para que possam ser identificados os seus possíveis valores e quais as operações que podem ser executadas.

Comandos:

begin

Blocos, atribuições a sinais, chamadas a subprogramas, processos, ….

end

No programa somente utilizei processos, um processo è; uma porção de código delimitada pelas palavras ***Process*** e ***End Process*****,** que contém comandos sequenciais. Todos os processos em VHDL são executados concorrentemente num *timestep*.

Código FIFO_IN

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
Definição das portas de entradas e saídas
entity fifo_in is
   Port ( DataIn : in  STD_LOGIC_VECTOR (7 downto 0);
          LowWrtEn : in  STD_LOGIC;
          LowOutEn : in  STD_LOGIC;
          clk : in  STD_LOGIC;
          FullBuffer : out  STD_LOGIC;
          DataOut : out  STD_LOGIC_VECTOR (7 downto 0));
end fifo_in;
architecture Behavioral of fifo_in is
Declaração do tamanho da memória, esta simulação tem dezanove posições e vai
armazenar vectores com oito bits.
TYPE FIFO_Buff IS ARRAY(0 TO 18) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL FIFO_Array    :        FIFO_Buff;
Definição de ponteiros de leitura e escrita
SIGNAL Write_ptr     :        INTEGER RANGE 0 TO 18 := 0;
SIGNAL Read_ptr              :        INTEGER RANGE 0 TO 18 := 0;
SIGNAL TempFull              :        STD_LOGIC;
SIGNAL TempEmpty   :        STD_LOGIC;
SIGNAL Last_wrt              :        STD_LOGIC;
SIGNAL Last_rd               :        STD_LOGIC;
BEGIN
```

- Verifica se a memória está cheia

- Define o " condição de *FullBuffer* "

- Quando os dois *"Write_ptr"* e *"Read_ptr"* são iguais, indica que a última acção foi uma gravação.

```
PROCESS(clk, LowOutEn)
BEGIN
        IF (clk'event AND clk='0') THEN
                IF(LowOutEn='0') THEN
                        TempFull <= '0';
                ELSE
                        IF((Write_ptr = Read_ptr) AND (Last_wrt='1')) THEN
                                TempFull <= '1';
                        ELSIF ((Last_rd='1') AND (TempFull='1')) THEN
                                TempFull <= '0';
                        ELSIF(Last_wrt='1' AND Last_rd='1') THEN
                                TempFull <= '0';
                        ELSE
                                TempFull <= '0';
                        END IF;
                END IF;
        END IF;
END PROCESS;
```

Processo de Escrita:

- Escreve quando a transacção de relógio é ascendente

- Escrever enquanto a memória não está completa

- Incremento de Ex: 0 1 2 3 4 5, …, 0 1 2 3 4 5, ….

```
PROCESS(clk,LowWrtEn)
        BEGIN
        IF(clk'event AND clk='1') THEN
        IF(Last_rd='1') THEN
                                Last_wrt <= '0';
        END IF;
```

```
            IF((LowWrtEn = '0') AND (TempFull = '0')) THEN
                FIFO_Array(Write_ptr) <= DataIn;
                    Last_wrt <= '1';
                IF (Write_ptr < 18)THEN
                    Write_ptr <= Write_ptr + 1;
                    ELSE
                  Write_ptr <= 0;
                END IF;
        END IF;
        END IF;
        END PROCESS;
```

- Verifica se memória está vazia *FIFO_EMPTY*

- Define *EmptyBuffer*

Se os dois apontadores são iguais indica que a última acção foi de leitura, de seguida define *TempEmpty* a 1.

- Se existe uma acção de escrita, a saída *TempEmpty* é '0'.

```
        PROCESS(clk, LowWrtEn)
        BEGIN
            IF(clk'EVENT AND clk='0') THEN
                IF(LowWrtEn='0') THEN
                    TempEmpty <= '0';
                ELSE
                    IF((Write_ptr=Read_ptr) AND (Last_rd='1')) THEN
                                TempEmpty <= '1';
                    ELSIF((Last_wrt= '1')and (TempEmpty = '1')) THEN
                        TempEmpty <= '0';                                --****
                    ELSIF(Last_wrt='1' AND Last_rd='1') THEN
                        TempEmpty <= '1';---**********
                    ELSE
                        TempEmpty <= '1';
                    END IF;
```

```
                END IF;
        END IF;
    END PROCESS;
```

Processo de leitura

- Lê quando a memória não está vazia

- A memória está vazia quando *Write_ptr = Read_ptr*

- Incremento de EX: 0 1 2 3 4 5, …, 0 1 2 3 4 5, ….

```
    PROCESS(clk, LowOutEn)
    BEGIN
        IF(clk'event AND clk='1')THEN
        IF(Last_wrt='1') THEN
                Last_rd <= '0';
        END IF;
        IF(LowOutEn='0' AND TempEmpty='0')THEN
                DataOut <= FIFO_Array(Read_ptr);
                Last_rd <= '1';
                IF (Read_ptr < 18)THEN
                        Read_ptr <= Read_ptr + 1;
                ELSE
                        Read_ptr <= 0;
                END IF;
        END IF;
        END IF;
    END PROCESS;
FullBuffer <= TempFull;
End Behavioral;
```

13. Anexo 2

Tabela 9 - Tabela de estados

Proxima posição		Test logic-reset	Runtest-idle	SelectDR-scan	Capture-DR	Shift-DR	Exit1-DR	Pause-DR	Exit2-DR	Update-DR	SelectIR-scan	Capture-IR	Shift-IR	Exit1-IR	Pause-IR	Exit2-IR	Update-IR
Posição actual		1111	1101	1100	1011	1001	1000	1010	0110	0111	1110	0000	0001	0010	0011	0100	0101
Test logic-reset	1111	-	0	10	010	0010	1010	01010	101010	11010	110	0110	00110	10110	010110	1010110	110110
Runtest-idle	1101	111	-	011	01	001	101	0101	10101	1101	11	011	0011	1011	01011	101011	11011
SelectDR-scan	1100	11	1	-	0	00	10	010	1010	110	1	01	001	101	0101	10101	1101
Capture-DR	1011	11111	01	0	-	0	1	01	101	11	1111	01111	001111	101111	0101111	10101111	1101111
Shift-DR	1001	11111	001	00	0111	-	1	01	101	11	1111	01111	001111	101111	0101111	10101111	1101111
Exit1-DR	1000	1111	101	10	011	0011	-	0	10	1	111	0111	00111	10111	010111	1010111	110111
Pause-DR	1010	11111	0101	010	01111	00111	10111	-	1	11	1111	01111	001111	101111	0101111	10101111	1101111
Exit2-DR	0110	1111	10101	1010	0111	0011	1011	010	-	1	111	0111	00111	10111	010111	1010111	110111
Update-DR	0111	1111	0	1	01	001	101	0101	10101	-	11	011	0011	1011	01011	101011	11011
SelectIR-scan	1110	1	11	1	01101	00101	10101	010101	1010101	110101	-	0	00	10	010	1010	110
Capture-IR	0000	11111	011	01	01111	00111	10111	010111	1010111	110111	1111	-	0	1	01	101	11
Shift-IR	0001	11111	011	001	0111	00111	10111	010111	1010111	110111	1111	01111	-	1	01	101	11
Exit1-IR	0010	1111	1011	101	011	0011	1011	01011	101011	11011	111	0111	010	-	0	10	1
Pause-IR	0011	11111	01011	0101	0111	00111	10111	010111	1010111	110111	1111	01111	01	101	-	1	11
Exit2-IR	0100	1111	101011	10101	011	0011	1011	01011	101011	11011	111	0111	0	10	010	-	1
Update-IR	0101	1111	0	1	01	001	101	0101	10101	1101	11	011	0011	1011	01011	01011	-

14. Anexo 3

Programa do controlador TAP desenvolvido através de linguagem VHDL e simulado pelo *software* disponibilizado pela Xilinx versão 10.

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

Definição das portas de entradas e saídas
entity tap is
   Port ( clk : in  STD_LOGIC;
          tms : out  STD_LOGIC;
          tck : out  STD_LOGIC;
          dados : in  STD_LOGIC_VECTOR (7 downto 0);
          dados2 : out  STD_LOGIC_VECTOR (7 downto 0);
          ready : out  STD_LOGIC;
          tdi : out  STD_LOGIC;
            tdi_serie: out std_logic;
          write_fifo : out  STD_LOGIC;
           control_fifo_out: in std_logic;
            comandar_fifo_out  : out  STD_LOGIC;
          tdo : in  STD_LOGIC);
end tap;

architecture Behavioral of tap is
```

Formação da tabela com 16 estados, cada estado pode deslocar-se entre os outros 15 estados ou continuar no seu próprio estado.

Começamos por definir o tamanho do *array*, cada posição tem um tamanho de oito bits.

----------state----------

```
type test_logic_reset is array (15 downto 0) of std_logic_vector (7 downto 0);
type runtest_idle is array (15 downto 0) of std_logic_vector (7 downto 0);
type selectdr_scan is array (15 downto 0) of std_logic_vector (7 downto 0);
type capturedr is array (15 downto 0) of std_logic_vector (7 downto 0);
type shiftdr is array (15 downto 0) of std_logic_vector (7 downto 0);
type exit1dr is array (15 downto 0) of std_logic_vector (7 downto 0);
type pausedr is array (15 downto 0) of std_logic_vector (7 downto 0);
type exit2dr is array (15 downto 0) of std_logic_vector (7 downto 0);
type updatedr is array (15 downto 0) of std_logic_vector (7 downto 0);
type selectir_scan is array (15 downto 0) of std_logic_vector (7 downto 0);
type captureir is array (15 downto 0) of std_logic_vector (7 downto 0);
type shiftir is array (15 downto 0) of std_logic_vector (7 downto 0);
type exit1ir is array (15 downto 0) of std_logic_vector (7 downto 0);
type pauseir is array (15 downto 0) of std_logic_vector (7 downto 0);
type exit2ir is array (15 downto 0) of std_logic_vector (7 downto 0);
type updateir is array (15 downto 0) of std_logic_vector (7 downto 0);
```

Preenchimento das tabelas acima definidas com o valor necessário para o deslocamento entre estados.

```
constant table_1:test_logic_reset := ("--------","-------0","------10","-----010","----0010","----1010","---01010","--101010","---11010","-----110","----0110","---00110","---10110","--010110","11010110","--110110");
constant table_2:runtest_idle := ("-----111","--------","-----011","------01","-----001","-----101","----0101","---10101","----1101","------11","-----011","----0011","----1011","---01011","--101011","---11011");
constant table_3:selectdr_scan := ("------11","-------1","--------","-------0","------00","------10","-----010","----1010","-----110","-------1","------01","-------001","-----101","----0101","---10101","----1101");
constant table_4:capturedr := ("---11111","------01","-------0","--------","-------0","-------1","------01","-----101","------11","----1111","---01111","--001111","--101111","-0101111","10101111","-1101111");
constant table_5:shiftdr := ("---11111","-----001","------00","----0111","--------","-------1","------01","-----101","------11","----1111","---01111","--001111","--101111","-0101111","10101111","-1101111");
constant table_6:exit1dr:= ("----1111","-----101","------10","-----011","----0011","--------","-------0","------10","-------1","-----111","----0111","---00111","---10111","--010111","-1010111","--110111");
constant table_7:pausedr:= ("---11111","----0101","-----010","---01111","---00111","---10111","--------","-------1","------11","----1111","---01111","--001111","--101111","-0101111","10101111","-1101111");
constant table_8:exit2dr:= ("----1111","---10101","----1010","----0111","----0011","----1011","-----010","--------","-------1","-----111","----0111","---00111","---10111","--010111","-1010111","--110111");
constant table_9:updatedr:= ("----1111","-------0","-------1","------01","-----001","-----101","----0101","---10101","--------","------11","-----011","----0011","----1011","---01011","--101011","---11011");
```

constant table_10:selectir_scan := ("-------1","------11","-------1","---01101","---00101","---10101","--010101","-1010101","--110101","--------","--------0","------00","------10","-----010","----1010","-----110");

constant table_11:captureir := ("---11111","-----011","------01","---01111","---00111","---10111","--010111","-1010111","--110111","----1111","--------","-------0","-------1","------01","-----101","------11");

constant table_12:shiftir := ("-----111","----0011","-----001","----0111","---00111","---10111","--010111","-1010111","--110111","----1111","---01111","--------","-------1","------01","-----101","------11");

constant table_13:exit1ir:= ("----1111","----1011","-----101","-----011","----0011","----1011","---01011","--101011","---11011","-----111","----0111","-----010","--------","-------0","------10","-------1");

constant table_14:pauseir:= ("---11111","---01011","----0101","----0111","---00111","---10111","--010111","-1010111","--110111","----1111","---01111","------01","-----101","--------","-------1","------11");

constant table_15:exit2ir:= ("----1111","--101011","---10101","-----011","----0011","----1011","---01011","--101011","---11011","-----111","----0111","-------0","------10","-----010","--------","-------1");

constant table_16:updateir:= ("----1111","-------0","-------1","------01","-----001","-----101","----0101","---10101","----1101","------11","-----011","---0011","----1011","---01011","---01011","--------");

Definição das variáveis globais utilizadas no programa; *std_logic_vector, integer, std_logic*.

signal count1: std_logic_vector (4 downto 0);

```
signal prox_estado: std_logic_vector (3 downto 0);
signal count2: std_logic_vector (7 downto 0);
signal actual_estado: std_logic_vector (3 downto 0) := "1111";
signal memoria,prox_posicao,variavel,variavel2: std_logic_vector (7 downto 0);
signal n_tabela,escrita_fifo,ready_status,ready_sdr,fim_state,passo,passos,k,w,tms1, main,sdr_ativo,counter_sdr,status_ativo, sdr_completo,
status_completo: integer:=0;
signal tck_sdr, tck_status,counter,start_tms: integer;
signal sdr,state: std_logic :='0' ;
signal status,start,receive: std_logic;
-----------------------------------------------------------------------------------------------
```

Begin

Iniciação da lógica do programa, começamos com no processo “Menu”, este verifica a entrada “dados” (correspondente á saída *Data_out* da FIFO_IN), existe duas possíveis sequências:

- Se corresponder a uma sequência de “11111111” e o estado actual for; *TestLogic Reset*, *Shift DR* ou *Shift IR* então vai para o processo “SDR”. Este realizará o procedimento de colocação dos dados recebidos na porta *Tdi_Serie.* É responsável também pelo tratamento dos dados da entrada TDO, bem como o controlo da escrita na memória FIFO_OUT. Para isso necessita de dois processos auxiliares; “processo_tdi”, “processo_tdo”.

- Se corresponder a uma sequência “11111110”, o controlador passará para o processo “STATE”, faz o deslocamento entre estados, necessita de dois processos auxiliares; “processo_tms”, “processo_proximo_estado”.

```
-------------------------processo MENU----------------------------------------
process (status_completo,sdr_completo,sdr,state,clk)
begin
        if (sdr ='0' and state ='0') then
                        if (dados = "11111111") then
                                case actual_estado is
                                        when "0001"   => sdr_ativo <=1;
                                        when "1001"   => sdr_ativo <=1;
                                        when "1110"   =>  sdr_ativo <=1;
                                        when others =>  sdr_ativo <=2;
                                end case;end if;end if;
        if (state ='0' and sdr ='0' ) then
                        if (dados = "11111110") then
                        status_ativo <=1;       sdr_ativo <=0;
                        else status_ativo <=2;end if;  end if;
```

```
if (status_completo =1 ) then status_ativo <= 0; end if;
if (sdr_completo =1 ) then sdr_ativo <= 0;end if;
end process;
```

O processo “SDR” é responsável pelo envio dos dados Série e recepção dos dados recebidos na porta *TDO* e escrita da Fifo_Out.

O processo começa por ler o barramento “Dados”, esse indica o número de dados a enviar (contador), de seguida espera até receber a segunda sequência de dados, estes serão enviados posteriormente para a porta Tdi_Serie do controlador.

Enquanto envia os dados está a ler simultaneamente a entrada TDO, quando termina a recepção de uma série de oito bits são enviados para a memória do Fifo_Out, onde são armazenados pela ordem de chegada.

O procedimento repete-se sucessivamente até o contador atingir o ‘0’.

```
----------------------------------processo SDR---------------------------------
process_sdr: process (clk,sdr,w,counter_sdr,variavel,count1,variavel2)
begin
if ( clk'event and clk = '1') then
case sdr is
        when '1' =>
                if (passo=0) then ready_sdr <= 1;counter <=1; end if;--teste1, verificar se entro na subrotina
                if ( counter_sdr =1)then  counter <=2; passo <=1;end if;
```

```
            if ( passo=1)then ready_sdr <= 0; passo <=2;escrita_fifo <=0;end if;
            if (passo = 2)then ready_sdr <= 1;start <= '1';end if;
            if (passo = 2 and start = '1')then receive<= '1';end if;
            if (w = 2 and count1> "0000")then escrita_fifo <=1;dados2 <= variavel;passo <=1;receive <= '0';start <= '0';end if;
            if (w = 2 and count1= "0000")then ready_sdr <= 0;passo <=3;escrita_fifo  <=1;dados2 <= variavel;receive <= '0';start <= '0';end if;
            if (passo =3 )then start <= '1';ready_sdr <= 1;passo <=5;escrita_fifo <=0;end if;
            if (passo =5 )then receive<= '1';end if;
            if (w = 1 )then escrita_fifo <=1;dados2 <= variavel2;passo <=4;ready_sdr     <= 0;start <= '0';receive <= '0';end if;
            if (passo = 4  )then sdr_completo<=1;escrita_fifo <=0;end if;
        when '0' => sdr_completo<=0; passo <=0;counter <=0;
        when others =>sdr_completo<=2; end case;
end if;
end process;
```

Processo responsável pelo envio dos dados série para a porta de saída *Tdi_Serie*, para esse procedimento a saída *TMS* tem de estar a '0' e o *TCK* activo.

```
process_tdi: process (clk,counter,start,passo)
variable i: integer :=0;
variable y: integer :=0;
```

```
begin
if (counter =1) then count1 <= dados (7 downto 3);count2 <= dados (7 downto 0);counter_sdr<=1;
else counter_sdr<=0; end if;
if (clk'event and clk = '1') then
        if ( start = '1' and count1> "0000" and w=0) then     tck_sdr <= 1; tms1 <= 1;
                if (i<=7) then tdi<=dados(i); tdi_serie<=dados(i);i := i+1;count2 <= count2-1;
                else tdi<='-';tdi_serie<='-';i:= 0;count1 <= count1-1;tck_sdr <= 0; tms1 <= 2;end if;end if;
                if ( start = '1' and count1= "0000" and w=0)then
                if (count2>= "0000") then variavel2(i)<= tdo;tdi<=dados(i);tdi_serie<=dados(i);i := i+1;
                count2 <= count2-1;tck_sdr <= 1; tms1 <= 1;
                else i:=0;end if; end if;end if;
if ( start = '1' and count1= "0000" and count2="0000")then tdi<='-';tdi_serie<='-';tck_sdr <= 0; tms1 <= 2;i:= 0;end if;
if (passo=4) then i:= 0;tck_sdr <= 0; tms1 <= 0;counter_sdr<=0;end if;
end process;
```

Processo responsável pela recepção dos dados na entrada TDO, armazena os dados numa variável com tamanho de oito bits.

```
process_tdo: process (clk, passo,count1, receive, variavel2)
variable y: integer :=0;
begin
```

```
if ( clk'event and clk = '0') then
                    if (receive = '1' and count1>= "0000" and w=0  ) then
                            if ( y<=7) then variavel(y)<= tdo;y := y+1;
                            elsif ( y>7) then y:= 0;w<= 2;end if; end if;
if ( passo=1 or passo=5 )then y:= 0;w<= 0;end if;
if ( count2= "0000"  ) then  w<=1;y:= 0;end if;          end if;
if ( passo=0) then y:= 0;w <= 0;end if;
end process;
-----------------------fim processo SDR---------------------------------
```

Processo responsável pela colocação da saída TCK activa, funciona como auxiliar dos processos " STATE" e "SDR", o TCK está sempre dependente da frequência do CLK.

```
-----------------------processo tck ------------------------------------
processo_tck: process (clk, tck_sdr, tck_status)
begin
if (tck_sdr = 1 or tck_status = 1) then tck <=clk;
elsif (tck_sdr = 0 or tck_status = 0 ) then tck <= '-'; end if;
end process;
-----------------------fim processo tck---------------------------------
```

Processo responsável pelo controle dos dados de escrita e leitura na memória “FIFO_IN”; se na saída está ‘0’,fazemos uma operação de escrita, caso seja ‘1’ o controlador está pronto a receber dados da memória.

```
----------processo ready ----------
processo_ready: process (ready_sdr, ready_status)
begin
if (ready_sdr = 1 or ready_status = 1) then ready <= '1';
elsif (ready_sdr = 0 or ready_status = 0 ) then ready <= '0'; end if;
end process;
----------fim processo ready----------
```

Neste processo o controlador administra os sinais de escrita e leitura da memória:

- Quando o sinal Control_Fifo_Out fica activo indica que a memória está cheia, então o controlador dá ordem para iniciar a colocação na saída *Dados_Out* os dados armazenados, para posterior análise. Esses dados são os recebidos da placa através da porta TDO, serão analisados e comparados com os enviados pela porta TDI.
- Caso o sinal Control_Fifo_Out não está activo, sabemos que podemos armazenar mais informação, essa tarefa é efectuada através das saídas write_fifo e comandar_fifo_out.

```
----------processo fifo_out ----------
processo_fifo_out: process (escrita_fifo,sdr_completo,sdr)
variable y: integer :=10;
begin
```

```
if (sdr='1' and control_fifo_out ='0') then comandar_fifo_out<='1';end if;
if (control_fifo_out ='1') then --y:=9;
        if (y>0) then comandar_fifo_out<='0'; y:=y-1;write_fifo<= '1';
        else comandar_fifo_out<='1'; --write_fifo<= '1';
        end if;end if;
if (escrita_fifo =1 and control_fifo_out ='0') then write_fifo<= '0'; comandar_fifo_out<='1';
elsif (escrita_fifo =0 and control_fifo_out ='0')  then write_fifo<= '1'; comandar_fifo_out<='1'; end if;
end process;
```

Após recepção da primeira sequência de dados enviada pela FIFO_IN, através do processo "MENU" passamos para o modo "STATE" ou "SDR", este processo é auxiliar e assegura que quando estamos nestes modos e recebemos uma sequência de dados coincidentes com a determinada para o seu procedimento, mantêm-se no modo actual e trata essa informação como dados.

Exemplo:

1.ª Sequencia "11111111", passamos para o modo "STATE".

2.ª Sequencia "11111111", trata como dados, neste exemplo deslocar-se-á na tabela de estados até ao "*TestLogicReset*".

```
----------processo escolher status ----------
processo_variavel: process (status_ativo, sdr_ativo)
begin
```

```
if (status_ativo = 1) then state <= '1';sdr <= '0';
 else state <= '0'; end if;
if (sdr_ativo = 1) then sdr <= '1';state <= '0';
else sdr <= '0';end if;
end process;
-----------------------fim processo ready---------------------------------
```

Este processo é responsável pelo deslocamento entre estados, utilizei dois processos auxiliares; “Processo_TMS” e “Proximo_estado”.

Separei o “processo_TMS” porque tem de estar activo, quer no modo “STATE” quer no “SDR”.

No modo “STATE” a saída TMS tem a sequência definida pela tabela de estados e no modo “SDR” está sempre o zero.

O processo “Proximo_estado” recebe o valor correspondente ao próximo estado, verifica o estado actual na tabela e desloca o apontador até ao vector correspondente que indica os estímulos a inserir na saída TMS e activa a saída TCK com o mesmo número de impulsos.

```
--**************************processo STATE**************************
processo_state: process (clk,state,fim_state)
begin
case state is
                when '1' =>
                        if ( clk'event and clk = '1') then
                                                if ( passos=0) then ready_status <= 1;passos <=1; end if;
                                                if ( passos=1 ) then ready_status <= 0;passos <=2; prox_estado<= dados (3 downto 0);end if;
```

```
                                        if ( passos=2 ) then start_tms<=1;ready_status <= 1;end if;
                                        if ( fim_state= 1 ) then ready_status <= 0;actual_estado
<=prox_estado;start_tms<=0;status_completo <=1;end if; end if;
                when '0' => passos <=0;status_completo<=0;
                when others =>status_completo<=2;end case;
end process;
-------------------------------------------------------------------------------------------------
```

Este processo é auxiliar do "STATE", trata da informação recebida pelo processo "Proximo_estado" e envia-o para a saída TMS e activa a saída TCK.

Quando o controlador está em modo "SDR", este processo coloca o TMS a zero.

```
processo_tms: process (clk,sdr,tms1,n_tabela,memoria)
variable z: integer :=0;
begin
if (sdr ='1') then
        if (tms1 = 1)  then tms<= '0';
        elsif (tms1 = 2) then tms<= '-';end if;end if;
if (clk'event and clk = '1' and start_tms =1) then
                        if (n_tabela>z) then
                        tms<= memoria(z); z:=z+1;tck_status <= 1;
```

```
                    elsif (n_tabela=z) then tms<= memoria(z); z:=z+1;tck_status <= 0;
                    elsif (n_tabela<z) then
                    z:=0;fim_state <= 1;tms<= '-';tck_status <= 0;
                    end if; end if;
if ( fim_state = 1) then  tms<= '-';tck_status <= 0;z:=0; end if;
if (passos = 0) then fim_state <= 0; end if;
end process;
--**************************************************************************--
```

Este processo determinada a próxima sequência para o deslocamento entre o estado actual e o próximo estado, através da pesquisa dos vectores na tabela, indica também o número de ciclos do TCK necessários para o deslocamento.

```
--*********************************processo proximo estado************************--
processo_proximo_estado: process (prox_estado,actual_estado)
begin
--dadoss1 <=prox_estado;--somente para verificar resultado do vector
case prox_estado is

 when "1111" =>
        case actual_estado is
```

```
        when "1111" => memoria <=table_1 (15); n_tabela<=0;when "1101" => memoria <=table_2 (15); n_tabela<=3;when "1100" =>
                        memoria <=table_3 (15); n_tabela<=2;when "1011" => memoria <=table_4 (15); n_tabela<=5;
        when "1001" => memoria <=table_5 (15); n_tabela<=5;when "1000" => memoria <=table_6 (15);n_tabela<=4; when "1010" =>
                        memoria <=table_7 (15); n_tabela<=5;when "0110" => memoria <=table_8 (15); n_tabela<=4;
        when "0111" => memoria <=table_9 (15); n_tabela<=4;when "1110" => memoria <=table_10 (15);n_tabela<=1;  when "0000" =>
                        memoria <=table_11 (15); n_tabela<=5;when "0001" => memoria <=table_12 (15); n_tabela<=5;
        when "0010" => memoria <=table_13 (15);          n_tabela<=4;when "0011" => memoria <=table_14 (15);  n_tabela<=5;when "0100"
                        => memoria <=table_15 (15); n_tabela<=4; when "0101" => memoria <=table_16 (15);n_tabela<=4;
        when others=> end case;
when "1101" =>
        case actual_estado is
        when "1111" => memoria <=table_2 (14); n_tabela<=1;when "1101" => memoria <=table_2 (14); n_tabela<=0;when "1100" =>
                        memoria <=table_3 (14); n_tabela<=1;when "1011" => memoria <=table_4 (14); n_tabela<=2;
        when "1001" => memoria <=table_5 (14); n_tabela<=3;when "1000" => memoria <=table_6 (14);n_tabela<=3; when "1010" =>
                        memoria <=table_7 (14); n_tabela<=4;when "0110" => memoria <=table_8 (14); n_tabela<=5;
        when "0111" => memoria <=table_9 (14); n_tabela<=1;when "1110" => memoria <=table_10 (14);n_tabela<=2;  when "0000" =>
                        memoria <=table_11 (14); n_tabela<=3;when "0001" => memoria <=table_12 (14); n_tabela<=4;
        when "0010" => memoria <=table_13 (14);          n_tabela<=4;when "0011" => memoria <=table_14 (14);  n_tabela<=5;when "0100"
                        => memoria <=table_15 (14); n_tabela<=6; when "0101" => memoria <=table_16 (14);n_tabela<=1;
```

```
            when others=> end case;
when "1100" =>
        case actual_estado is
          when "1111" => memoria <=table_1 (13); n_tabela<=2;when "1101" => memoria <=table_2 (13); n_tabela<=3;when "1100" =>
                         memoria <=table_3 (13); n_tabela<=0;when "1011" => memoria <=table_4 (13); n_tabela<=1;
          when "1001" => memoria <=table_5 (13); n_tabela<=2;when "1000" => memoria <=table_6 (13);n_tabela<=2; when "1010" =>
                         memoria <=table_7 (13); n_tabela<=3;when "0110" => memoria <=table_8 (13); n_tabela<=4;
          when "0111" => memoria <=table_9 (13); n_tabela<=1;when "1110" => memoria <=table_10 (13);n_tabela<=1;  when "0000" =>
                         memoria <=table_11 (13); n_tabela<=2;when "0001" => memoria <=table_12 (13); n_tabela<=3;
          when "0010" => memoria <=table_13 (13);          n_tabela<=4;when "0011" => memoria <=table_14 (13);  n_tabela<=5;when "0100"
                         => memoria <=table_15 (13); n_tabela<=6; when "0101" => memoria <=table_16 (13);n_tabela<=1;
                         when others=> end case;
when "1011" =>
        case actual_estado is
          when "1111" => memoria <=table_1 (12); n_tabela<=3;when "1101" => memoria <=table_2 (12); n_tabela<=2;when "1100" =>
                         memoria <=table_3 (12); n_tabela<=1;when "1011" => memoria <=table_4 (12); n_tabela<=0;
          when "1001" => memoria <=table_5 (12); n_tabela<=4;when "1000" => memoria <=table_6 (12);n_tabela<=3; when "1010" =>
                         memoria <=table_7 (12); n_tabela<=5;when "0110" => memoria <=table_8 (12); n_tabela<=4;
```

```
        when "0111" => memoria <=table_9 (12); n_tabela<=2;when "1110" => memoria <=table_10 (12);n_tabela<=5;  when "0000" =>
                        memoria <=table_11 (12); n_tabela<=5;when "0001" => memoria <=table_12 (12); n_tabela<=4;
        when "0010" => memoria <=table_13 (12);          n_tabela<=3;when "0011" => memoria <=table_14 (12);  n_tabela<=4;when "0100"
                        => memoria <=table_15 (12); n_tabela<=3; when "0101" => memoria <=table_16 (12);n_tabela<=2;
        when others=> end case;
when "1001" =>
        case actual_estado is
        when "1111" => memoria <=table_1 (11); n_tabela<=4;when "1101" => memoria <=table_2 (11); n_tabela<=3;when "1100" =>
                        memoria <=table_3 (11); n_tabela<=2;when "1011" => memoria <=table_4 (11); n_tabela<=1;
        when "1001" => memoria <=table_5 (11); n_tabela<=0;when "1000" => memoria <=table_6 (11);n_tabela<=4; when "1010" =>
                        memoria <=table_7 (11); n_tabela<=5;when "0110" => memoria <=table_8 (11); n_tabela<=4;
        when "0111" => memoria <=table_9 (11); n_tabela<=3;when "1110" => memoria <=table_10 (11);n_tabela<=5;  when "0000" =>
                        memoria <=table_11 (11); n_tabela<=5;when "0001" => memoria <=table_12 (11); n_tabela<=5;
        when "0010" => memoria <=table_13 (11);          n_tabela<=4;when "0011" => memoria <=table_14 (11);  n_tabela<=5;when "0100"
                        => memoria <=table_15 (11); n_tabela<=4; when "0101" => memoria <=table_16 (11);n_tabela<=3;
        when others=> end case;
when "1000" =>
        case actual_estado is
```

```
        when "1111" => memoria <=table_1 (10); n_tabela<=4;when "1101" => memoria <=table_2 (10); n_tabela<=3;when "1100" =>
                        memoria <=table_3 (10); n_tabela<=2;when "1011" => memoria <=table_4 (10); n_tabela<=1;
        when "1001" => memoria <=table_5 (10); n_tabela<=1;when "1000" => memoria <=table_6 (10);n_tabela<=0; when "1010" =>
                        memoria <=table_7 (10); n_tabela<=5;when "0110" => memoria <=table_8 (10); n_tabela<=4;
        when "0111" => memoria <=table_9 (10); n_tabela<=3;when "1110" => memoria <=table_10 (10);n_tabela<=5;  when "0000" =>
                        memoria <=table_11 (10); n_tabela<=5;when "0001" => memoria <=table_12 (10); n_tabela<=5;
        when "0010" => memoria <=table_13 (10);          n_tabela<=4;when "0011" => memoria <=table_14 (10);  n_tabela<=5;when "0100"
                        => memoria <=table_15 (10); n_tabela<=4; when "0101" => memoria <=table_16 (10);n_tabela<=3;
        when others=> end case;
when "1010" =>
        case actual_estado is
        when "1111" => memoria <=table_1 (9); n_tabela<=5;when "1101" => memoria <=table_2 (9); n_tabela<=4;when "1100" => memoria
                        <=table_3 (9); n_tabela<=3;when "1011" => memoria <=table_4 (9); n_tabela<=2;
        when "1001" => memoria <=table_5 (9); n_tabela<=2;when "1000" => memoria <=table_6 (9);n_tabela<=1; when "1010" => memoria
                        <=table_7 (9); n_tabela<=0;when "0110" => memoria <=table_8 (9); n_tabela<=3;
        when "0111" => memoria <=table_9 (9);    n_tabela<=4;when "1110" => memoria <=table_10 (9);n_tabela<=6;  when "0000" =>
                        memoria <=table_11 (9); n_tabela<=6;when "0001" => memoria <=table_12 (9); n_tabela<=6;
        when "0010" => memoria <=table_13 (9); n_tabela<=5;when "0011" => memoria <=table_14 (9);  n_tabela<=6;when "0100" =>
                        memoria <=table_15 (9); n_tabela<=5; when "0101" => memoria <=table_16 (9);n_tabela<=4;
```

```
        when others=> end case;
         when "0110" =>
       case actual_estado is
        when "1111" => memoria <=table_1 (8); n_tabela<=6;when "1101" => memoria <=table_2 (8); n_tabela<=5;when "1100" => memoria
                       <=table_3 (8); n_tabela<=4;when "1011" => memoria <=table_4 (8); n_tabela<=3;
        when "1001" => memoria <=table_5 (8); n_tabela<=3;when "1000" => memoria <=table_6 (8);n_tabela<=2; when "1010" => memoria
                       <=table_7 (8); n_tabela<=1;when "0110" => memoria <=table_8 (8); n_tabela<=0;
        when "0111" => memoria <=table_9 (8);    n_tabela<=5;when "1110" => memoria <=table_10 (8);n_tabela<=7;  when "0000" =>
                       memoria <=table_11 (8); n_tabela<=7;when "0001" => memoria <=table_12 (8); n_tabela<=7;
        when "0010" => memoria <=table_13 (8); n_tabela<=6;when "0011" => memoria <=table_14 (8);  n_tabela<=7;when "0100" =>
                       memoria <=table_15 (8); n_tabela<=6; when "0101" => memoria <=table_16 (8);n_tabela<=5;
        when others=> end case;
when "0111" =>
       case actual_estado is
        when "1111" => memoria <=table_1 (7); n_tabela<=5;when "1101" => memoria <=table_2 (7); n_tabela<=4;when "1100" => memoria
                       <=table_3 (7); n_tabela<=3;when "1011" => memoria <=table_4 (7); n_tabela<=2;
        when "1001" => memoria <=table_5 (7); n_tabela<=2;when "1000" => memoria <=table_6 (7);n_tabela<=1; when "1010" => memoria
                       <=table_7 (7); n_tabela<=2;when "0110" => memoria <=table_8 (7); n_tabela<=1;
```

```
        when "0111" => memoria <=table_9 (7);    n_tabela<=0;when "1110" => memoria <=table_10 (7);n_tabela<=6;  when "0000" =>
                        memoria <=table_11 (7); n_tabela<=6;when "0001" => memoria <=table_12 (7); n_tabela<=6;
        when "0010" => memoria <=table_13 (7); n_tabela<=6;when "0011" => memoria <=table_14 (7);  n_tabela<=6;when "0100" =>
                        memoria <=table_15 (7); n_tabela<=5; when "0101" => memoria <=table_16 (7);n_tabela<=4;
        when others=> end case;
when "1110" =>
        case actual_estado is
        when "1111" => memoria <=table_1 (6); n_tabela<=3;when "1101" => memoria <=table_2 (6); n_tabela<=2;when "1100" => memoria
                        <=table_3 (6); n_tabela<=1;when "1011" => memoria <=table_4 (6); n_tabela<=4;
        when "1001" => memoria <=table_5 (6); n_tabela<=4;when "1000" => memoria <=table_6 (6);n_tabela<=3; when "1010" => memoria
                        <=table_7 (6); n_tabela<=4;when "0110" => memoria <=table_8 (6); n_tabela<=3;
        when "0111" => memoria <=table_9 (6);    n_tabela<=2;when "1110" => memoria <=table_10 (6);n_tabela<=0;  when "0000" =>
                        memoria <=table_11 (6); n_tabela<=4;when "0001" => memoria <=table_12 (6); n_tabela<=4;
        when "0010" => memoria <=table_13 (6); n_tabela<=3;when "0011" => memoria <=table_14 (6);  n_tabela<=4;when "0100" =>
                        memoria <=table_15 (6); n_tabela<=3; when "0101" => memoria <=table_16 (6);n_tabela<=2;
        when others=> end case;
when "0000" =>
        case actual_estado is
```

```
        when "1111" => memoria <=table_1 (5); n_tabela<=4;when "1101" => memoria <=table_2 (5); n_tabela<=3;when "1100" => memoria
                        <=table_3 (5); n_tabela<=2;when "1011" => memoria <=table_4 (5); n_tabela<=5;
        when "1001" => memoria <=table_5 (5); n_tabela<=5;when "1000" => memoria <=table_6 (5);n_tabela<=4; when "1010" => memoria
                        <=table_7 (5); n_tabela<=5;when "0110" => memoria <=table_8 (5); n_tabela<=4;
        when "0111" => memoria <=table_9 (5);   n_tabela<=2;when "1110" => memoria <=table_10 (5);n_tabela<=1;  when "0000" =>
                        memoria <=table_11 (5); n_tabela<=0;when "0001" => memoria <=table_12 (5); n_tabela<=5;
        when "0010" => memoria <=table_13 (5); n_tabela<=4;when "0011" => memoria <=table_14 (5);  n_tabela<=5;when "0100" =>
                        memoria <=table_15 (5); n_tabela<=4; when "0101" => memoria <=table_16 (5);n_tabela<=3;
        when others=> end case;
when "0001" =>
        case actual_estado is
        when "1111" => memoria <=table_1 (4); n_tabela<=5;when "1101" => memoria <=table_2 (4); n_tabela<=4;when "1100" => memoria
                        <=table_3 (4); n_tabela<=3;when "1011" => memoria <=table_4 (4); n_tabela<=6;
        when "1001" => memoria <=table_5 (4); n_tabela<=6;when "1000" => memoria <=table_6 (4);n_tabela<=5; when "1010" => memoria
                        <=table_7 (4); n_tabela<=6;when "0110" => memoria <=table_8 (4); n_tabela<=5;
        when "0111" => memoria <=table_9 (4);   n_tabela<=4;when "1110" => memoria <=table_10 (4);n_tabela<=2;  when "0000" =>
                        memoria <=table_11 (4); n_tabela<=1;when "0001" => memoria <=table_12 (4); n_tabela<=0;
        when "0010" => memoria <=table_13 (4); n_tabela<=3;when "0011" => memoria <=table_14 (4);  n_tabela<=2;when "0100" =>
                        memoria <=table_15 (4); n_tabela<=1; when "0101" => memoria <=table_16 (4);n_tabela<=4;
```

```
            when others=> end case;
when "0010" =>
        case actual_estado is
          when "1111" => memoria <=table_1 (3); n_tabela<=5;when "1101" => memoria <=table_2 (3); n_tabela<=4;when "1100" => memoria
                        <=table_3 (3); n_tabela<=3;when "1011" => memoria <=table_4 (3); n_tabela<=6;
          when "1001" => memoria <=table_5 (3); n_tabela<=6;when "1000" => memoria <=table_6 (3);n_tabela<=5; when "1010" => memoria
                        <=table_7 (3); n_tabela<=6;when "0110" => memoria <=table_8 (3); n_tabela<=5;
          when "0111" => memoria <=table_9 (3);    n_tabela<=4;when "1110" => memoria <=table_10 (3);n_tabela<=2;  when "0000" =>
                        memoria <=table_11 (3); n_tabela<=1;when "0001" => memoria <=table_12 (3); n_tabela<=1;
          when "0010" => memoria <=table_13 (3); n_tabela<=0;when "0011" => memoria <=table_14 (3);  n_tabela<=3;when "0100" =>
                        memoria <=table_15 (3); n_tabela<=2; when "0101" => memoria <=table_16 (3);n_tabela<=4;
          when others=> end case;
when "0011" =>
        case actual_estado is
          when "1111" => memoria <=table_1 (2); n_tabela<=6;when "1101" => memoria <=table_2 (2); n_tabela<=5;when "1100" => memoria
                        <=table_3 (2); n_tabela<=4;when "1011" => memoria <=table_4 (2); n_tabela<=7;
          when "1001" => memoria <=table_5 (2); n_tabela<=7;when "1000" => memoria <=table_6 (2);n_tabela<=6; when "1010" => memoria
                        <=table_7 (2); n_tabela<=7;when "0110" => memoria <=table_8 (2); n_tabela<=6;
```

```
        when "0111" => memoria <=table_9 (2);    n_tabela<=5;when "1110" => memoria <=table_10 (2);n_tabela<=3;  when "0000" =>
                        memoria <=table_11 (2); n_tabela<=2;when "0001" => memoria <=table_12 (2); n_tabela<=2;
        when "0010" => memoria <=table_13 (2); n_tabela<=1;when "0011" => memoria <=table_14 (2);  n_tabela<=0;when "0100" =>
                        memoria <=table_15 (2); n_tabela<=3; when "0101" => memoria <=table_16 (2);n_tabela<=5;
        when others=> end case;
when "0100" =>
        case actual_estado is
        when "1111" => memoria <=table_1 (1); n_tabela<=7;when "1101" => memoria <=table_2 (1); n_tabela<=6;when "1100" => memoria
                        <=table_3 (1); n_tabela<=5;when "1011" => memoria <=table_4 (1); n_tabela<=8;
        when "1001" => memoria <=table_5 (1); n_tabela<=8;when "1000" => memoria <=table_6 (1);n_tabela<=7; when "1010" => memoria
                        <=table_7 (1); n_tabela<=8;when "0110" => memoria <=table_8 (1); n_tabela<=7;
        when "0111" => memoria <=table_9 (1);    n_tabela<=6;when "1110" => memoria <=table_10 (1);n_tabela<=4;  when "0000" =>
                        memoria <=table_11 (1); n_tabela<=3;when "0001" => memoria <=table_12 (1); n_tabela<=3;
        when "0010" => memoria <=table_13 (1); n_tabela<=2;when "0011" => memoria <=table_14 (1);  n_tabela<=1;when "0100" =>
                        memoria <=table_15 (1); n_tabela<=0; when "0101" => memoria <=table_16 (1);n_tabela<=5;
        when others=> end case;
when "0101" =>
        case actual_estado is
```

```
        when "1111" => memoria <=table_1 (0); n_tabela<=6;when "1101" => memoria <=table_2 (0); n_tabela<=5;when "1100" => memoria
                        <=table_3 (0); n_tabela<=4;when "1011" => memoria <=table_4 (0); n_tabela<=7;
        when "1001" => memoria <=table_5 (0); n_tabela<=7;when "1000" => memoria <=table_6 (0);n_tabela<=6; when "1010" => memoria
                        <=table_7 (0); n_tabela<=7;when "0110" => memoria <=table_8 (0); n_tabela<=6;
        when "0111" => memoria <=table_9 (0);    n_tabela<=5;when "1110" => memoria <=table_10 (0);n_tabela<=3;  when "0000" =>
                        memoria <=table_11 (0); n_tabela<=2;when "0001" => memoria <=table_12 (0); n_tabela<=2;
        when "0010" => memoria <=table_13 (0); n_tabela<=1;when "0011" => memoria <=table_14 (0);  n_tabela<=2;when "0100" =>
                        memoria <=table_15 (0); n_tabela<=1; when "0101" => memoria <=table_16 (0);n_tabela<=0;
        when others=> end case;
when others=> end case;
end process;
 ----------------------------------fim processo state--------------------------------
end behavioral;
```

15. Anexo 4

O programa da FIFO_ OUT é igual ao utilizada na FiFO_IN, a sua função no módulo Boundary_Scan é a oposta, tem como objectivo receber e armazenar todos os dados enviados pelo controlador TAP, armazena-los e quando cheia coloca-los á saída através do barramento *Data_Out*.

Programa em VHDL da FIFO_OUT

```
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity fifo_out is
   Port ( DataIn : in  STD_LOGIC_VECTOR (7 downto 0);
          LowWrtEn : in  STD_LOGIC;
          LowOutEn : in  STD_LOGIC;
          clk : in  STD_LOGIC;
          FullBuffer : out  STD_LOGIC;
          DataOut : out  STD_LOGIC_VECTOR (7 downto 0));
end fifo_out;
architecture Behavioral of fifo_out is
TYPE FIFO_Buff IS ARRAY(0 TO 9) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL FIFO_Array    :       FIFO_Buff;
SIGNAL Write_ptr     :       INTEGER RANGE 0 TO 9 := 0;
SIGNAL Read_ptr              :       INTEGER RANGE 0 TO 9 := 0;
SIGNAL n             :       INTEGER  := 0;
SIGNAL TempFull              :       STD_LOGIC;
SIGNAL TempEmpty     :       STD_LOGIC;
SIGNAL Last_wrt              :       STD_LOGIC;
SIGNAL Last_rd               :       STD_LOGIC;
```

```
BEGIN
PROCESS(clk, LowOutEn)
BEGIN
	IF (clk'event AND clk='0') THEN
		IF(LowOutEn='0') THEN
			TempFull <= '0';
		ELSE
			IF((Write_ptr = Read_ptr) AND (Last_wrt='1')) THEN
				TempFull <= '1';
			ELSIF ((Last_rd='1') AND (TempFull='1')) THEN
				TempFull <= '0';
			ELSIF(Last_wrt='1' AND Last_rd='1') THEN
				TempFull <= '0';
			ELSE
				TempFull <= '0';
			END IF;
		END IF;
	END IF;
END PROCESS;

PROCESS(clk,LowWrtEn)
	BEGIN
	IF(clk'event AND clk='1') THEN
	IF(Last_rd='1') THEN
				Last_wrt <= '0';
	END IF;
		IF((LowWrtEn = '0') AND (TempFull = '0')) THEN
			FIFO_Array(Write_ptr) <= DataIn;
				Last_wrt <= '1';
			IF (Write_ptr < 9)THEN
				Write_ptr <= Write_ptr + 1;
				ELSE
```

```
                    Write_ptr <= 0;
                END IF;
        END IF;
        END IF;
        END PROCESS;

        PROCESS(clk, LowWrtEn)
        BEGIN
                IF(clk'EVENT AND clk='0') THEN
                        IF(LowWrtEn='0') THEN
                                TempEmpty <= '0';
                        ELSE
                                IF((Write_ptr=Read_ptr) AND (Last_rd='1')) THEN
                                        TempEmpty <= '1';
                                ELSIF((Last_wrt= '1')and (TempEmpty = '1')) THEN
                                        TempEmpty <= '1';                       --****
                                ELSIF(Last_wrt='1' AND Last_rd='1') THEN
                                        TempEmpty <= '1';---**********
                                ELSE
                                        TempEmpty <= '0';
                                END IF;
                        END IF;
                END IF;
        END PROCESS;
        PROCESS(clk, LowOutEn)
        BEGIN
                IF(clk'event AND clk='0')THEN
                IF(Last_wrt='1') THEN
                                Last_rd <= '0';
                END IF;
```

```
                IF(LowOutEn='0' AND TempEmpty='0')THEN
                                DataOut <= FIFO_Array(Read_ptr);
                                Last_rd <= '1';
                                if (n<9) then
                                IF (Read_ptr < 9)THEN
                                        Read_ptr <= Read_ptr + 1;
                                ELSE
                                        Read_ptr <= 0;
                                END IF;n<=n+1;end if;
                END IF;
                END IF;
        END PROCESS;
FullBuffer <= TempFull;
end Behavioral;
```

16. Anexo 5

Tabela 10 - Tabela com a sequência de simulação

Passo	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Operação	**State**	**Shift IR**	**Dados**	**Contador**	**Dados-1**	**State**	**Shift DR**
	1	-	1	0	0	1	-
	1	-	1	0	0	1	-
	1	-	1	0	0	1	-
	1	0	1	0	0	1	0
	1	0	1	1	0	1	0
	1	1	1	0	0	1	1
	1	1	1	0	0	1	1
	1	0	0	0	0	1	1
	Ir para estado shift-IR		**Colocar em Extest**			**Ir para estado shift-DR**	
Passo	**8**	**9**	**10**	**11**	**12**		
Operação	**Dados**	**Contador**	**Dados-1**	**Dados-2**	**Dados-3**		
	1	0	1	1	0		
	1	0	0	0	0		
	1	1	1	1	0		
	1	1	0	0	0		
	1	0	1	1	0		
	1	1	0	0	0		
	1	1	1	1	1		
	0	0	0	0	0		
	Introduzir 1ª sequência de 18 bits, total 54 bits						
Passo			**13**	**14**	**15**		
Operação			**Dados-1**	**Dados-2**	**Dados-3**		
			0	0	0		
			1	1	0		
			0	0	0		
			1	1	0		
			0	0	0		
			1	1	0		
			0	0	0		
			1	1	1		
		Introduzir 2ª sequência de 18 bits					
Passo			**16**	**17**	**18**		
Operação			**Dados-1**	**Dados-2**	**Dados-3**		
			0	0	0		
			0	0	0		
			0	0	0		
			0	0	0		
			0	0	0		
			0	0	0		
			0	0	0		
			0	0	0		
		Introduzir 3ª sequência de 18 bits					
Passo	**19**	**20**	**21**	**22**	**23**	**24**	**25**
Operação	**State**	**Shift IR**	**Dados**	**Contador**	**Dados-1**	**State**	**Test Logic Reset**
	1	-	1	0	1	1	-
	1	-	1	0	1	1	-
	1	0	1	0	1	1	-
	1	0	1	0	1	1	-
	1	1	1	1	1	1	-
	1	1	1	0	1	1	0
	1	1	1	0	1	1	1
	1	1	0	0	1	1	1
	Ir para estado shift-IR		**Colocar em Bypass**			**Ir para estado *RunTest/Idle***	