UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA

Especificação de um Auditor de Testabilidade de Projetos de CI's Digitais Baseados em Células

Autora: Bernadete Aparecida de Lima Oliveira

Membros da banca:

Presidente: Prof. Dr. Carlos Ignácio Zamitti Mammana
Membro: Prof. Dr. Mário Lúcio Côrtes
Membro: Dr. José Roberto Amazonas
Suplente: Prof. Dr. Furio Damiani

Dissertação submetida como requisito parcial para a obtenção do título de mestre em Engenharia Elétrica.

UNICAMP - 9 de Setembro de 1991.

Sumário

Esta dissertação trata a especificação de um sistema de auditoria de testabilidade de projetos de CI's digitais baseados em células. Situa a utilização de um sistema como esse no ciclo de projeto, descreve metodologias de projeto para testabilidade, particularmente os métodos de projeto com "scan", e as regras de testabilidade associadas que esse auditor deve verificar. Descreve características de ferramentas de apoio ao projeto de CI's, com enfoque especial às que são dirigidas à síntese com testabilidade ou à verificação de técnicas de projeto para testabilidade. Aproveitando as facilidades de implementação proporcionadas pelas características dos sistemas especialistas, é especificado um sistema baseado em verificação de regras constantes de uma base de conhecimento. É descrito o protótipo implementado e são comentados resultados de processamento de casos práticos. Considerando os resultados obtidos com o protótipo e as perspectivas do ambiente de projeto de CI's digitais são fornecidas conclusões sobre a validade de sistemas de verificação como o sistema especificado.

Agradecimentos

Esta dissertação é resultado de esforço, dedicação e inspiração, e de um grande conjunto de boas e indispensáveis contribuições. Entre os elementos indispensáveis desse conjunto estão: algumas sementes instaladas na massa encefálica desde a infância, graças à orientação recebida de meus pais; a possibilidade de realizar esse trabalho, utilizando recursos do CPqD - TELEBRÁS, com o incentivo de H. Malavazi; a dedicação e a paciência de Paulo Henrique de Oliveira que, na qualidade de cônjuge, deu cobertura às minhas ausências, necessárias para que esse trabalho pudesse se realizar.

Para ser exaustivo e explícito, quanto às boas contribuições, o agradecimento seria demasiado longo, mas é indispensável agradecer à boa vontade de M. L. Côrtes, ao empenho de C. I. Z. Mammana, às muitas contribuições recebidas de colegas de trabalho entre os quais o Túlio pelas discussões de idéias, o André e o Kerr por esclarecimentos de dúvidas quanto ao PROLOG, o Jucá, a Júlia e o Pisani por fornecimento de dados, o Mendonça, o Narcizo e o Hélio Pedrini pelas dicas sobre o latex, e ainda, às realimentações favoráveis de muitos bons amigos de qualquer pedaço da vida.

Muito obrigada.

Para

Luís Renato e Pedro Augusto

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Cada vez são maiores os problemas associados com teste de CI's digitais, devido ao grande aumento em complexidade atingido por esses dispositivos, permitido pelo avanço tecnológico da área.

Muitas técnicas e metodologias de projeto foram desenvolvidas no decorrer dos últimos vinte anos, para assegurar a qualidade dos projetos e para garantir o seu teste. Esse desenvolvimento gerou um extenso conhecimento sobre projeto para testabilidade, que pode ser traduzido em regras.

A influência do custo de teste nos custos dos CI's e a significativa contribuição para diminuir esse custo, possível devida à aplicação de técnicas de projeto para testabilidade, justificam o investimento nesse segmento do ciclo de projeto. No entanto, para que se obtenha os efeitos vantajosos das técnicas de projeto para testabilidade, o projeto deve atender a regras bem definidas. O atendimento às regras deve ser verificado de forma efetiva, o que pode ser atingido com o uso de uma ferramenta de auditoria.

O auditor especificado nesta dissertação se destina a verificar projetos de CI's digitais baseados em células, quanto a regras de projeto estruturado, contidas em uma base de conhecimento. Esse sistema reune características de sistema especialista e características modulares, que permitem atualizações e extensões sem interferência direta sobre regras implementadas. Com base em protótipo implementado utilizando a linguagem e o interpretador PROLOG, é possível verificar a validade de uso dessa ferramenta em casos práticos de projeto de CI's digitais.

Esta dissertação se compõe dos elementos necessários à especificação de um

sistema auditor de testabilidade. As metodologias de projeto para testabilidade são objeto do capítulo 2, com enfoque especial sobre projeto estruturado ou técnicas de projeto com "scan" e as regras de testabilidade associadas.

O capítulo 3 discorre sobre ferramentas de apoio ao projeto de CI's testáveis, passando por caracteríticas relevantes de projeto de CI's digitais, classificação das ferramentas de apoio aos projetos e mencionando exemplos de ferramentas constantes da literatura, que estão relacionadas com projeto de CI's testáveis.

O capítulo 4 descreve a especificação do auditor propriamente dita, o protótipo implementado e os resultados obtidos com a aplicação do auditor a casos práticos.

As conclusões relativas ao desenvolvimento de ferramentas de análise como essa são decorrentes de fatores práticos tais como necessidade de competitividade e de diminuição do ciclo de desenvolvimento de projeto, sendo objeto do capítulo 5.

Para complementar lacunas de definições de termos da área de Inteligência Artificial é apresentado um glossário no apêndice A; um estudo de casos devido a problemas de teste de CI's é assunto do apêndice B; a bibliografia consultada aparece no apêndice C e a listagem do protótipo implementado em PROLOG mais os casos práticos aos quais o auditor foi aplicado são apresentados no Apêndice D.

Capítulo 2

Projeto para Testabilidade

2.1 Introdução

O grande avanço alcançado pela tecnologia de circuitos integrados (CI's) permite a realização de milhares de dispositivos e interconexões numa única pastilha de silício, através de um conjunto comum de estágios de fabricação [Agrawal88]. Tais componentes atingem complexidades na faixa de 10^5 a 10^6 transistores [Kowalski85]. Esse aumento exponencial [Muehldorf81] observado no número de circuitos em um CI traz inúmeras vantagens, tais como melhoria em performance, novas aplicações de sistema com redução nos custos e maior confiabilidade. Entretanto, acarreta também dificuldades relativamente ao seu desenvolvimento.

O ciclo de desenvolvimento de um CI pode ser dividido em três grandes fases: **concepção, fabricação e teste**, sendo que elas compreendem várias sub fases. A **concepção** engloba a especificação, o projeto funcional, lógico e elétrico, validação por simulação funcional, lógica e elétrica e o projeto físico. A **fabricação** envolve diferentes etapas de processo para os dois grandes casos: 1º caso - circuitos dedicados ("full custom") ou semidedicados ("semi custom") do tipo "standard-cells", e 2º caso - semidedicados implementados em "gate arrays"; e a fase de **teste** consiste em: depuração de protótipo (teste funcional e estrutural), teste de caracterização e teste de produção.

Uma das maiores dificuldades associadas ao ciclo de desenvolvimento de CI's complexos está na fase de **teste**, particularmente na validação do projeto desses dispositivos, atividade imprescindível, conhecida como depuração de protótipo

[Côrtes91] e cujo peso econômico é muito alto, especialmente se não forem tomados cuidados importantes durante o projeto.

A necessidade de se obter circuitos mais econômicamente testáveis conduziu ao desenvolvimento de técnicas e/ou metodologias conhecidas como Técnicas de Projeto para Testabilidade, ou "Design For Testability - DFT", já traduzido como Projeto Visando a Testabilidade - PVT [Wagner88] [1].

Há uma grande variedade de técnicas e soluções propostas na literatura ao longo dos últimos 20 anos em termos de Projeto para Testabilidade. Comparações[2] entre soluções de DFT já eram apresentadas por [Wiliams73]. A maior parte das técnicas ou soluções de DFT se baseiam em conceitos básicos (seção 2.2). Dependendo de sua aplicação, essas técnicas são classificadas como: Técnicas Específicas ou AD HOC (ver seção 2.3), Técnicas Estruturadas (ver seção 2.4) ou Técnicas de Circuitos Auto-testáveis (ver seção 2.5). Além disso, podem assumir características específicas associadas aos diferentes estilos e/ou estruturas utilizadas (ver item 2 da subseção 2.5.3).

A escolha e a aplicação de Técnicas de Projeto para Testabilidade deve considerar as compensações dadas por economia de recursos para geração e realização dos testes, relativamente ao custo de implementação, representado pelo aumento de hardware e por diminuição em performance. Mas, de modo geral, pode-se aplicar regras para projeto estruturado, como aquelas enunciadas na seção 2.6.

Este capítulo apresenta conceitos e Técnicas de Projeto para Testabilidade, detalhando especialmente as metodologias estruturadas, cuja aplicação pode ser verificada automaticamente, o que contribui para viabilizar o projeto de CI's complexos com ganhos em qualidade e confiabilidade.

2.2 Conceitos básicos relacionados com testabilidade de CI's

2.2.1 O Problema de Teste de CI's

1. **Definições**

Falha ("fault") em um circuito, de acordo com o IEEE Standard Dictio-

[1] Adotada a sigla DFT nesta dissertação.

[2] Nesse artigo eram comparados os efeitos de se adotar um pino por flip flop contra a sua interligação em registradores de deslocamento.

nary [Graf84] corresponde à geração de resultados incorretos para uma dada sequência de entrada. Portanto, supõe que os resultados esperados para uma dada entrada não foram observados na saída do circuito. Segundo [Muehldorf81], falha é uma condição de um item[3] que pode causar **erro** ("failure", traduzido como erro em [Côrtes91]). Referindo-se a circuitos integrados, falhas podem ser curtos, abertos e condições similares.

Erro ("failure") é a ocorrência da impossibilidade de um item realizar sua função, de acordo com sua especificação [Muehldorf81].

Defeito ("defect") é um problema físico, que provoca um erro e pode ou não manifestar-se como falha. O comportamento elétrico resultante de um defeito físico no circuito é conhecido como **Modo de Falha**. A anomalia ocorrida no processo de fabricação causadora do defeito físico no circuito é conhecida como **Mecanismo de Falha**.

Pode-se classificar as falhas em [Breuer76] :

- **Falhas Lógicas**: mudam a função lógica do circuito, causando uma resposta incorreta no estado estável do circuito; CURTO e ABERTO podem muitas vezes ser modelados como falhas lógicas; em algumas tecnologias, como CMOS, uma falha de ABERTO pode provocar operação imprevisível do circuito e portanto, não pode ser modelada como falha permanente[4].
- **Falhas Paramétricas**: alteram a magnitude de um parâmetro do circuito, causando uma mudança em algum fator tal como velocidade, corrente ou voltagem. É o caso de falhas de atraso que ocorrem quando o atraso de propagação para uma dada porta lógica excede o pior caso especificado. Essa classe de falhas é de menor interesse nesta dissertação.

2. **Modelamento de falha**

Modelo de Falha é uma abstração a nível lógico da falha física real no circuito, utilizada para permitir a sua detecção e localização. Por exemplo, a presença de uma partícula sobre a máscara no momento da metalização (mecanismo de falha), provoca um comportamento elétrico de curto entre linhas de metal (modo de falha), que apresenta comportamento equivalente a uma outra função no nível lógico, a de "wired-and" (modelo de falha), conforme ilustrado na figura 2.1, extraída de [McCluskey91].

Um modelo historicamente considerado adequado e mais simples para o nível de portas lógicas é o modelo de falha de "stuck-at". É um modelo de falha no nível lógico, restrito a falhas permanentes.

[3] Item é o termo genérico que pode ser associado a um terminal de uma porta lógica, um subcircuito, um circuito completo, uma placa, etc [Muehldorf81].

[4] Falha permanente é a que está sempre presente; não muda durante o teste.

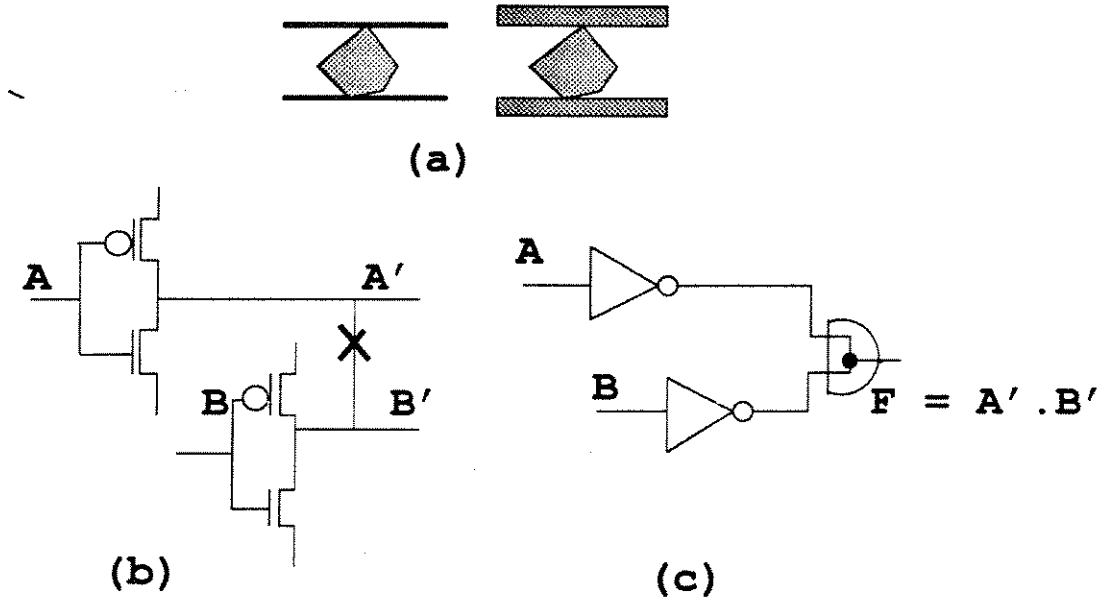

Figura 2.1: Mecanismo de falha (a); Modo de falha (b) e Modelo de falha - "wired-and" (c)

A maior parte das falhas podem ser modeladas pelo modelo de **Falha de "stuck-at"** o qual assume que uma entrada ou saída de uma dada porta está presa em nível zero ou um, independente dos estímulos a ela aplicados.

Há certos tipos de falha que não são corretamente modeladas pelo modelo de "stuck-at", como por exemplo:

- falhas de curto entre linhas, originalmente denominadas **"bridging faults"**. Esse tipo de falhas pode ocasionar um comportamento sequencial para um circuito combinacional.
- falhas em memórias RAM conhecidas como **"pattern sensitive fault"**, cuja detecção é realizada com o uso de teste funcional (ver item 3 desta seção). Essa falha é motivo de preocupação para circuitos de alta densidade [McCluskey86] e se caracteriza por apresentar resultado errado de leitura e escrita de um registro, dependendo do conteúdo de um outro registro adjacente [Breuer76]. O efeito dessa falha é dependente da sequência de sinais de entrada aplicados ao circuito.
- falhas em circuitos MOS: do tipo **"stuck-open"**, em que os transistores se comportam como abertos entre alguns de seus terminais; e do tipo **"stuck-on"**, ou seja o funcionamento do transistor MOS fica

alterado para permanentemente fechado. Essa falha pode criar um caminho estático de VDD para VSS e gerar sinais lógicos de saída fora da faixa de sinais válidos. Sua detecção é feita pela monitoração da corrente da fonte de alimentação, a qual apresenta valores excessivos na presença da falha [Côrtes91].

- falhas de **"cross point"** especificamente associadas a PLA's[5] (Arranjos Lógicos Programáveis). Falhas de "crosspoint" são falhas em cruzamentos entre linhas da PLA do tipo: ausência de contato onde deveria haver ou presença indevida de contato. Para essas falhas são usados modelos de falha funcional específicos.

Modelo de falha de "stuck-at" simples, originalmente "Single stuck-at fault model", é uma particularização do modelo de falhas de "stuck-at"; supõe que uma máquina boa não terá nenhuma falha, ao passo que uma com falha terá uma única das falhas de "stuck-at" possíveis.

Cobertura de falhas de "stuck at" simples para um dado padrão de teste é dada pela razão entre o número de falhas detectadas pelo padrão e o número de falhas consideradas.

Para um circuito com K nós (linhas de interconexão), restrito à classe de falha de "single stuck-at", há um número de falhas simples a considerar igual a 2K (duas possibilidades de falha para cada nó). Se é considerada a possibilidade de falha simultânea há $(3^K - 1)$ falhas múltiplas a considerar. Se um circuito contém K interconexões, qualquer uma delas pode estar boa, presa em zero ("stuck-at" 0) ou presa em 1 ("stuck-at" 1); então, todas as possíveis combinações de estado do circuito seriam 3^K, sendo que uma delas corresponde ao estado livre de falhas.

O modelo de falhas de "single stuck-at" é amplamente utilizado devido à grande diferença no número de falhas a considerar e também porque os conjuntos de teste para falhas simples são relativamente bons para detecção de falhas múltiplas, embora não sejam bons para sua localização. A localização de falha pode ser efetivada usando dicionários de falha, rotinas de diagnóstico ou análise de causa-efeito [Breuer86].

3. **A tarefa de teste**

Define-se **teste** de um dispositivo como o procedimento que examina a habilidade de um item realizar as suas especificações [Muehldorf81]. Esse procedimento consiste da aplicação de estímulos aos seus pinos de entrada e a observação de respostas **esperadas** nos seus pinos de saída. Isso implica na comparação dos sinais observados nas saídas com valores

[5] PLA's são implementadas por dois planos lógicos (AND - OR) e têm um dispositivo (um diodo ou um transistor) em cada "crosspoint" dos planos. A conexão de cada um desses dispositivos é definida pela programação para realizar a lógica desejada.

especificados, para a aplicação das respectivas entradas ou com resultados obtidos de circuitos sem falha.

Pode-se classificar o teste de um dispositivo em três grandes tipos: teste de validação, teste de produção e teste de manutenção.

O **teste de validação** é aquele que comprova que o protótipo ou o modelo[6] de um dado circuito realiza as funções especificadas para o projeto, dentro de características definidas. Tem por objetivo auxiliar a detecção e localização de possíveis erros de projeto.

O **teste de produção** é o que detecta componentes falhos em consequência de características do processo de produção.

O **teste de manutenção** se aplica à identificação de componente falho ou não num sistema em operação.

O grau de dificuldade para a execução de qualquer desses três tipos de teste depende das características do projeto. Uma das características que permite assegurar a qualidade do projeto é a testabilidade. **Testabilidade** é a propriedade associada ao projeto de um circuito, que traduz a sua capacidade de ser testado, ou seja, a facilidade de se gerar padrões para o seu teste.

Projeto para testabilidade refere-se à incorporação de facilidades de teste nos circuitos, tornando-os mais controláveis e / ou observáveis, ou ainda, promovendo o seu auto-teste.

Os projetos podem prever testes para aplicação:

- **"On line"**, também chamados "concurrent testing" ou testes implícitos. São testes aplicados durante o funcionamento normal do circuito. Podem ser implementados nos circuitos utilizando redundância de informação (adotando codificação), redundância de circuitos (utilizando duplicação) ou redundânca de tempo (por meio de repetição e comparação);
- **"Off line"**, ou testes explícitos, que são testes utilizados com o circuito fora de operação normal. São implementados através de reconfiguração funcional da arquitetura dos circuitos.

A tarefa de teste pode ser dividida em: geração de padrão de teste e aplicação dos testes [Graf84]. De modo geral pode-se executar testes num dispositivo de duas maneiras:

- utilizar um equipamento de teste externo, automático ou não, sendo que a automação é imprescindível para circuitos grandes; ou,
- implementar os testes no próprio circuito, de modo que ele mesmo os execute.

[6] A validação aplicada a modelos é feita por simulação.

Vetor de teste é o padrão que aplicado a uma porta sem falha tem uma saída diferente da saída de uma porta com falha. Portanto, está vinculado a falhas abrangidas por um modelo. Define-se **padrão completo** com relação a um conjunto de falhas como sendo o conjunto de vetores de teste que detecta todas as falhas possíveis de serem detectadas, isto é, **detectáveis** [Breuer76].

O problema de geração de padrões de teste pode ser definido como a determinação do conjunto de vetores de teste para se atingir um padrão completo. Normalmente, geram-se testes que garantem uma determinada cobertura, isto é que uma determinada percentagem de falhas modeladas seja detectada. Para tal, são usados métodos de ATPG (Geração Automática de Padrões de Teste).

Os testes podem ser funcionais ou estruturais. **Teste funcional** é o que examina a habilidade de um item operar de acordo com as suas especificações funcionais [Muehldorf81]; no caso de circuitos integrados digitais o teste funcional verifica a sua capacidade de realizar as funções lógicas especificadas segundo os respectivos requisitos de performance. O teste funcional pode ser exaustivo ou então por amostragem. **Teste estrutural** é um teste baseado na estrutura física ou no leiaute do circuito. Ele pode ser probabilístico ou determinístico [Agrawal88].

Pela própria característica digital, para que o teste de um circuito combinacional seja **exaustivo** todas as suas entradas primárias têm que ser testadas para 1 e para 0, em todas as combinações possíveis.

Para um circuito puramente combinacional[7] com N entradas, são requeridos no mínimo 2^N padrões. Se este circuito com N entradas for acrescentado de um circuito sequencial[8] que tem M variáveis de estado, como ilustrado na figura 2.2, o número mínimo de vetores necessário para o teste exaustivo da parte combinacional é igual a 2^{N+M}, pois equivale a incluir mais essas M variáveis como entradas para a lógica combinacional.

Pode-se avaliar o tempo necessário para aplicar esses vetores para um circuito, considerando-se taxa de aplicação da ordem de microsegundo/vetor e um circuito não muito complexo. Supondo um somador de 32 bits, portanto com 64 pinos de entrada, o número de vetores seria de 2^{64}, correspondendo a um tempo de teste de mais de 5000 séculos. Outro exemplo é o caso de um microprocessador cujo número de vetores para teste foi estimado em maior que 10^{32} [Jacob84]. Considerada a mesma taxa de aplicação, o tempo de teste seria da ordem de 10^{20} anos. Ao que se con-

[7] Circuito puramente combinacional é aquele que contém só nós combinacionais. Nó combinacional é uma entrada primária ou é nó de saída de uma célula combinacional. Célula combinacional é aquela cujas saídas dependem somente do estado atual de suas entradas.

[8] Circuito sequencial é aquele cujo estado presente das saídas depende das entradas e também dos estados prévios do circuito propriamente dito [Hicks87].

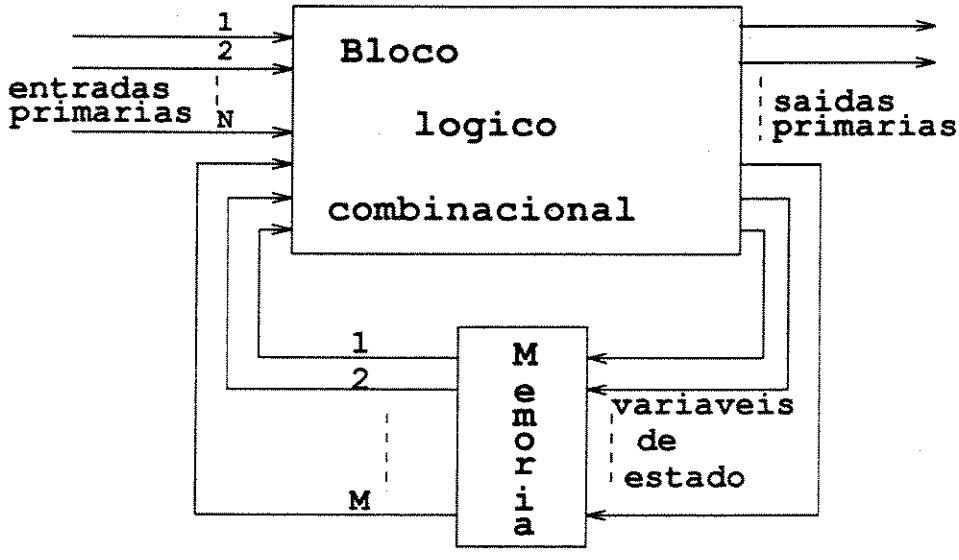

Figura 2.2: Requisito de teste exaustivo para circuito sequencial

clui a inviabilidade de se basear a testabilidade de circuitos integrados em teste exaustivo.

Devido ao grande número de funções possíveis de se implementar num único dispositivo e ao relativamente baixo número de pinos de entrada e saída, a aplicação de teste funcional para validação de projeto depende da adição de circuitos que facilitem o teste e proporcionem aumento de cobertura. Teste totalmente estruturado, por outro lado, pode aumentar o tamanho do dispositivo, provocando influências no "yield" (ver item 5 desta sequência), e aumentando o tempo de teste do CI [Kuban84].

4. **Simulação de Falha**

Simulação de Falhas é uma tarefa realizada para determinar a cobertura de falha de um dado padrão. É a melhor medida da efetividade do teste e está relacionada com um modelo de falhas.

O modelo mais amplamente adotado para simulação de falha em CI's é o de "single stuck-at". Neste caso o processo de simulação de falha consiste em aplicar todo o padrão de teste a uma máquina boa e a cada uma das possíveis máquinas contendo uma única falha de"stuck-at". A simulação de falhas gera a lista de falhas detectadas pela aplicação de cada vetor de teste.

Os métodos desenvolvidos para simulação de falhas são classificados em: paralelos, dedutivos e concorrentes. **Paralelos** quando a simulação é feita simultâneamente para várias instâncias do circuito, só uma delas sem

falha. **Dedutivos** têm como princípio a simulação do circuito correto e a dedução do comportamento do circuito com falha. **Concorrentes** quando somente são simulados os elementos em falha que não concordam com o elemento equivalente no circuito livre de falhas em termos de níveis de entrada e/ou saída. Os concorrentes são os mais efetivos [Fujiwara85].

Outra maneira de avaliar a cobertura de falhas de um padrão evitando-se a simulação de falhas é a avaliação estatística, conhecida por **STAFAN** ("STAtistical Fault ANalysis") [Jain85]. Essa avaliação é realizada através do cômputo do nível de atividade e da avaliação da controlabilidade e da observabilidade de cada nó do circuito durante a simulação do circuito sem falha. Desse modo são obtidas de forma aproximada a cobertura de falhas e a probabilidade de detecção de cada falha.

O tempo de processamento para simulação do circuito sem falha, incluindo essas avaliações estatísticas cresce linearmente com o número de nós do circuito, enquanto que foi observado que o tempo para geração automática de vetores de teste e simulação de falha é aproximadamente proporcional ao número de portas lógicas à 3a potência. Para circuitos grandes os métodos de simulação de falha consomem muito tempo, sendo muito custosos.

5. **Custo de teste**

A tarefa de teste tem grande peso econômico no ciclo de desenvolvimento de projeto de CI's. Os custos de teste combinam os custos de gerar padrões automaticamente e os custos de aplicar esses padrões. Envolvem tempo de processamento computacional de geração de padrões e/ou custo de homem-hora para geração manual. A aplicação dos testes está relacionada com o custo do equipamento de teste e com o tempo requerido para aplicar os testes. A eficiência dos testes tem grande influência nos custos de produção e manutenção dos sistemas, uma vez que é muito mais caro reparar uma placa falha do que descartar um chip falho, e é muito mais caro reparar um sistema falho do que reparar uma placa de circuito eletrônico. É sabido que esses custos crescem na proporção de dez vezes entre esses níveis subsequentes de empacotamento [Williams82,Graf84].

Outra característica muito importante para a composição do custo dos CI's é o "yield" (Y) [9], o qual juntamente com a cobertura de falhas (FC) do teste resulta o equacionamento do nível de defeito[10] dos componentes (DL), conforme a seguinte expressão [Baker] :

[9] "Yield" é a relação entre o número de componentes bons e o número de componentes produzidos. É uma medida da qualidade do processo.

[10] Nível de defeito é a porcentagem dos itens que apresentam defeito no campo, embora aprovados pelos testes de produção. Mede a qualidade do produto. [Williams81]

$$DL = 1 - Y^{(1-FC)}$$

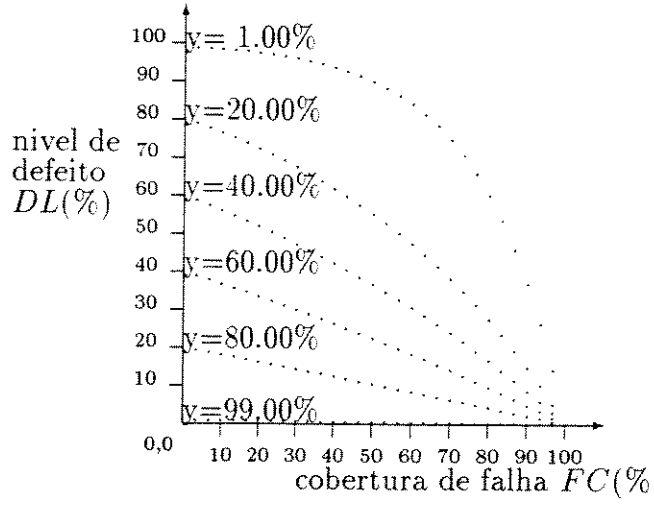

Figura 2.3: Nível de defeito em função de "Yield" e da Cobertura de falhas

Graficamente, tem-se uma família de curvas ilustrada na figura 2.3, na qual se pode observar que os efeitos produzidos por pequena variação da FC são pouco significativos para valores de Y muito altos ou muito baixos. Já para valores típicos de Y, isto é, por volta de 50% e considerando um interesse em produzir com $DL < 100ppm$ é requerida uma FC de praticamente 100%, o que ressalta a importância da qualidade do teste na qualidade do produto comercializado. Há situações em que 1 ou 2% de redução na cobertura de falha pode resultar um custoso aumento no nível de defeito [DAC90]. O "Yield" depende de processos físico-químicos sobre os quais a tecnologia tem controle apenas parcial, sendo muito mais difícil melhorar o "yield" do que aumentar a cobertura do teste.

A opção por uso de técnicas de DFT tem influência indireta na cobertura de falhas dos testes de validação do projeto do CI e também sobre o nível de defeito do CI. Os custos de teste crescem exponencialmente com o aumento de complexidade, em comparação com a **característica quase linear de custos de testes para circuitos que incorporam técnicas de projeto para testabilidade** [Hicks87]. Os custos envolvidos no processo de teste em geral, especialmente para a validação de projetos, considerando as características de geração e aplicação de testes, justificam a aplicação de técnicas de DFT, como solução indireta de tornar viável o projeto de CI's complexos.

2.2.2 Projeto e a testabilidade

As razões para se adotar técnicas ou metodologias de projeto para testabilidade (DFT) são a redução na complexidade da geração de teste e para fazer os CI's total ou parcialmente testáveis, diminuindo os custos da aplicação dos testes.

São conceitos-chave em técnicas de DFT:

- **Controlabilidade**: é a habilidade de produzir um valor de sinal interno específico, aplicando sinais às entradas do circuito.
- **Observabilidade**: é a habilidade de observar um estado de um sinal interno a partir de saídas do circuito.

Esses conceitos estão associados com a verificação do teste de cada porta lógica, cada entrada e cada saída, sendo essas características necessárias para cada uma delas, mesmo dentro de um circuito muito grande.

As técnicas de DFT podem ser classificadas nas categorias:

- técnicas específicas (AD HOC),
- técnicas estruturadas, semi "built-in" ou "scan design" e
- técnicas de circuitos auto-testáveis, ou técnicas de testes "on-line" totalmente implementados no circuito ("built in").

A adequação de qualquer uma delas a um projeto depende das características do CI, do número de portas (tamanho), do estágio de desenvolvimento do seu projeto e da filosofia de projeto e teste. Observa-se que quando o teste não é pensado durante o projeto ele pode se tornar muito caro ou até obrigar o abandono do produto devido às dificuldades de teste em quantidades de produção.

O projeto realizado com técnicas de DFT do tipo AD HOC ou técnicas estruturadas é também denotado por **"facilmente testável"**, e deve resultar em geração de teste e simulação de falha a custos aceitáveis. A principal diferença entre técnicas AD HOC e técnicas estruturadas é provavelmente o custo da implementação e o retorno do investimento deste custo extra. As técnicas AD HOC são aquelas aplicáveis a problemas particulares. Já as técnicas estruturadas são usadas como metodologia de projeto, para resolver problemas gerais.

A tarefa de gerar teste e simular falha não é tão simples e direta, quando se usa AD HOC, como quando se usa técnicas estruturadas [Williams82].

A terceira categoria de técnica de DFT é resultado de uma mistura de idéias de AD HOC e de métodos estruturados [Baker], gerada devido à impossibilidade de se atingir níveis aceitáveis de testabilidade para circuitos mais complexos, com a simples adoção de técnicas estruturadas. Essas técnicas eliminam o problema de gerar ou analisar padrões de teste.

2.3 Técnicas Específicas

São técnicas não generalizáveis, também conhecidas como AD HOC[11], dos seguintes tipos:

1. **Particionamento**

   Consiste em dividir o circuito em subcircuitos, por meio de lógica de desconexão ou usando multiplexações, adicionando linhas de desabilitação e controle de módulos. Exemplo clássico: desliga-se a lógica que gera um dado relógio, seja um oscilador, e permite-se a entrada de um relógio controlável externamente [Williams82]. Vide figura 2.4.

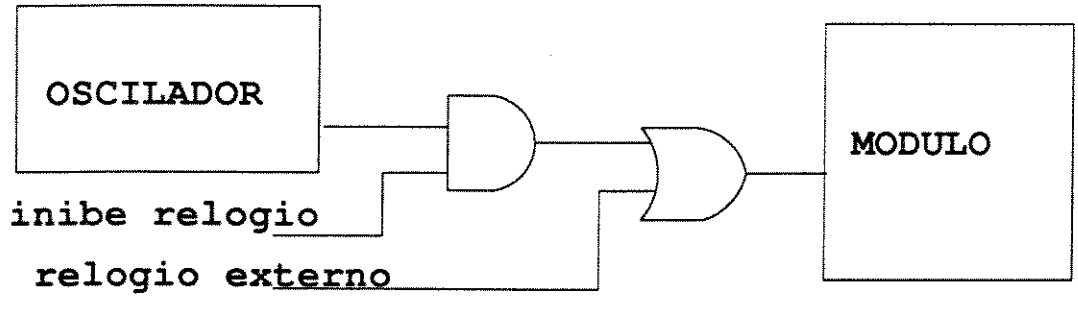

Figura 2.4: Exemplo de Particionamento: Desligamento de relógio interno

Também são exemplos desse tipo de técnica:

- acrescentar circuitos que permitem quebrar os "loops" de realimentação.
- implementar lógica para segmentar circuitos, como por exemplo circuitos sequenciais profundos (contadores e cadeias divisoras) em partes mais facilmente testáveis.

[11] AD HOC tem origem no latim e significa "para este caso".

Considerando-se que a tarefa de gerar padrão de teste e simular falha é proporcional à N^3, onde N = número de portas lógicas, adota-se essa solução a qual se baseia no princípio de "dividir para conquistar". Isto é, divide-se o circuito em blocos para que esses sejam testados individualmente; a soma dos custos de teste dos blocos é bem menor que o custo do processo de teste do dispositivo tratado como um único bloco.

Os problemas mais evidentes relacionados com o particionamento são:

- encontrar a melhor divisão ou pelo menos uma divisão satisfatória;
- realizar acesso físico a cada subcircuito a um custo viável;
- automatizar a inserção de modificação no circuito original por ser uma alteração não estruturada, isto é, particular.

Redes complexas de lógica aleatória requerem particionamento automático. A partição, com o objetivo de permitir controlar e observar circuitos diretamente pode levar a um significativo aumento no número de entradas e saídas primárias extras, além de módulos extras para realizar o desligamento da lógica propriamente dito para realizar o teste.

2. **Pontos de Teste**

Podem ser adicionados pinos de entrada para enriquecer a controlabilidade e de saída para aumentar a observabilidade de pontos internos do CI, particularmente para pontos de " fan in"[12] e de " fan out"[13], evitando-se "wired logic"[14]. Há sempre um custo associado com essa adição de pinos de teste e às vezes é impossível adicioná-los por causa de limitações de pinos de encapsulamento. Em alguns casos um único pino pode ser usado como entrada e saída, dependendo de condições intrínsecas do circuito, ou da maneira como é desligada a lógica.

Os pontos podem ser escolhidos com base em experiência ou com uso de ferramentas software, como os Analisadores de Testabilidade (ver próximo capítulo) que indicam pontos do circuito com caracterísitcas de baixa observabilidade ou controlabilidade.

São sugestões para uso de pontos de teste:

- Usar pinos (pelo menos um) para indicar modos de funcionamento do circuito: um modo para operação normal e outro(s) para teste. No modo teste, pontos do circuito podem ser observados, devido a multiplexação com saídas funcionais.

[12] "Fan in" é usado aqui para denotar entrada de porta lógica.

[13] "Fan out" aqui tem o sentido de sinal de saída de uma porta que se destina a várias entradas de outras portas lógicas.

[14] "Wired logic" é uma implementação lógica que gera o efeito de uma porta lógica através da conexão de duas saídas de outras portas lógicas.

- Quando é necessário controlar N pontos internos ao circuito, pode-se utilizar as saídas de um decodificador interno, que recebe as combinações necessárias de entradas primárias em um modo teste.
- Usar pinos de teste para realizar CLEAR ou PRESET de elementos de memória, por exemplo, possibilitando colocar máquinas sequenciais em estado conhecido, com muito poucos vetores. Isso é denominado Previsibilidade ("Predictability"). É uma maneira de se **inicializar** circuitos. Outra solução para inicializar máquinas sequenciais é acrescentar circuitos de "pull up" ou "pull down" aos seus elementos de memória para torná-las auto inicializáveis.

3. **Arquitetura orientada ao barramento**

   Esta técnica é utilizada principalmente no projeto de microcomputadores. Permite acesso aos barramentos que estão envolvidos com os vários módulos; é uma forma de particionamento, onde as saídas de um módulo em particular podem tornar-se visíveis enquanto as saídas dos outros, que compartilham o mesmo barramento, são colocadas em alta impedância. Um obstáculo à aplicação desse tipo de técnica vem a ser a detecção de falha no barramento propriamente dito. Normalmente se localiza falha em função de erro de informação de voltagem. Isolar uma falha de barramento requer medidas de corrente que são muito mais difíceis de fazer [Williams82].

2.4 Técnicas Estruturadas

O fato de ATPG não ter se mostrado uma ferramenta razoávelmente efetiva para lógica sequencial levou a esquemas estruturados de projeto para testabilidade [Baker], muito conhecidos como "Scan Design Methods". São conjuntos de regras de concepção de aplicação geral, para reduzir a complexidade de geração e verificação de teste de circuitos envolvendo circuitos sequenciais. São baseados em "scan-path" ou LSSD (Disciplina publicada em 1977 no Design Automation Conference) [Eichelberger77], que obrigam o projeto a ser realizado de modo síncrono, com cada um dos elementos de memória baseados em elementos biestáveis (flip flops, FF's) do tipo D ou pares de "latches" mestre-escravo.

Há uma variedade de técnicas de "scan" as quais têm em comum a característica de serializar o dado de teste. A maneira usada para serializar o dado de teste depende da estratégia de relógio do sistema. A mudança de modo normal para modo teste pode ser controlada por um sinal de relógio exclusivo

de teste ou por um sinal de modo teste dado por um nível. Dois enfoques são usados para converter dados paralelos em seriais, um deles usa registradores de deslocamento, sendo conhecido como "scan-path" e o outro usa multiplexadores.

A aplicação de "Scan Design" apresenta como custo o acréscimo de circuito lógico que é usado somente para teste, embora permitam acesso a nós internos do circuito sem requerer um pino para acesso a cada nó; requerem de 1 a 4 pinos para o circuito todo. Devido às interconexões há aumento de área e influência na performance do circuito. A performance do circuito também sofre penalidades com a inserção da lógica específica para teste, devido a aumento de atraso de propagação nos "latches" e flip flops do caminho, bem como devido às interconexões adicionais. O tempo de teste é aumentado pela necessidade de injetar os vetores de teste e enviar as respostas às saídas serialmente. Porém o circuito modificado requer conjuntos de teste bem menores que o circuito original. A maioria das técnicas de projeto estruturado consideram que se os "latches" podem ser controlados para qualquer valor específico e se eles podem ser observados com uma operação direta, então a geração de teste e possivelmente a simulação de falha ficam reduzidas aos circuitos lógicos combinacionais [Williams73,Agrawal88,Hicks87].

2.4.1 "Scan Path" para projetos com Flip Flops

A idéia de "Scan-path" é realizar um caminho que permita controlar e observar as células sequenciais de um circuito. A primeira solução publicada com o objetivo de realizar o "scan-path" propunha a inserção de um multiplexador na entrada de dados de todo flip flop do circuito como se vê na figura 2.5 (b), permitindo conectar todos eles em registradores de deslocamento. Essa solução aumenta o atraso no caminho e em consequência foram projetadas células de flip flop com entrada multiplexada que não acrescentam atraso equivalente ao acréscimo de um nível lógico, denominadas **MDflip flop**. Com a aplicação de um sinal de controle aos elementos de memória, eles são chaveados para operação em modo normal ou modo teste. Em modo normal a entrada normal de dados é selecionada no "latch" ou Flip Flop quando o relógio atua. Em modo "scan" há uma entrada de "scan" secundária que é selecionada para ser amostrada com o relógio. Cada uma dessas entradas secundárias de "scan" são ligadas à saída Q de outro FF para formar uma cadeia de registradores de deslocamento. Isso pode ser usado para controlar e observar todos os nós sequenciais internos do circuito.

A arquitetura proposta por essa idéia é a da figura 2.5, que permite o teste da

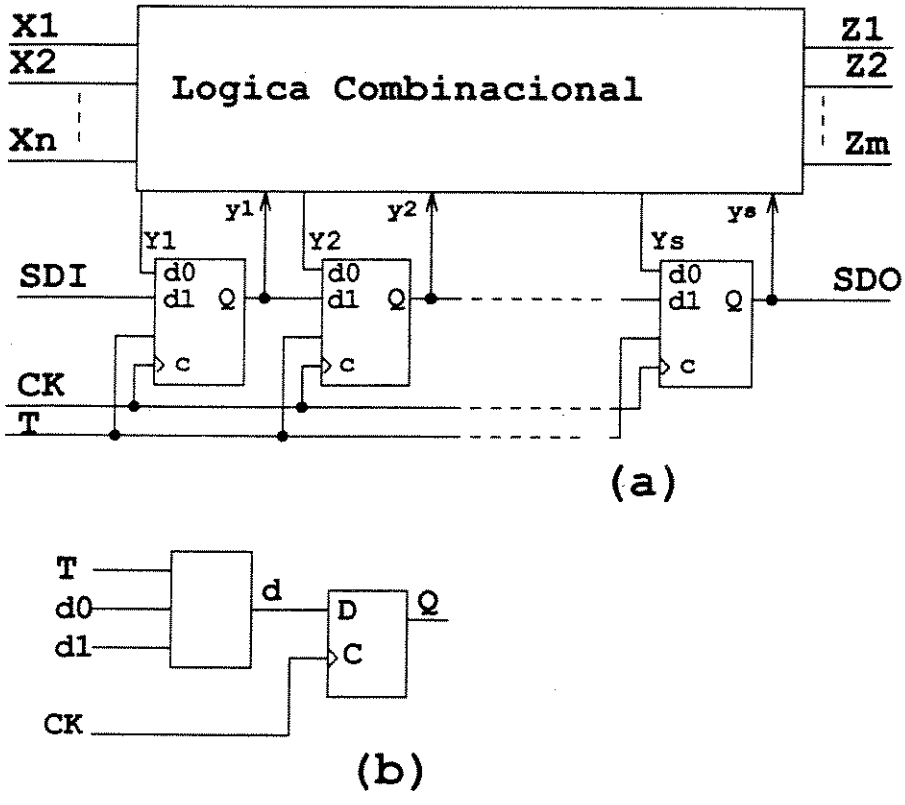

Figura 2.5: Arquitetura de "Scan path" para projetos com flip flop (a); Primeira solução (b)

lógica combinacional da seguinte forma:

- associa-se o nível 1 à entrada de controle dos dados T;
- aplica-se os valores de teste correspondentes às entradas X_i;
- executa-se a entrada serial ("shift in") dos dados y_i nos flip flops através da entrada SDI (de "scan data in") primária de "scan" durante s subidas de relógio, onde s é o comprimento do "scan";
- troca-se o nível de T para 0 e após um tempo para estabilização da lógica combinacional pode-se observar os valores das saídas Z_k ;
- aplica-se o sinal de relógio CK permitindo apenas uma variação ativa nos flip flops;
- troca-se o nível de T para 1 e
- executa-se a saída serial ("shift out") dos dados amostrados pelos flip flops (Y_i), através da saída primária SDO (de "scan data out"), os quais

podem então ser comparados com valores correspondentes às saídas da lógica combinacional. Ao mesmo tempo faz-se a entrada serial do novo vetor de teste.

Os flip flops são testados serialmente inserindo-se, através de SDI, sequências de 1's e 0's, que garantem o seu funcionamento em todas as situações de transição e verificando-se os valores na saída SDO.

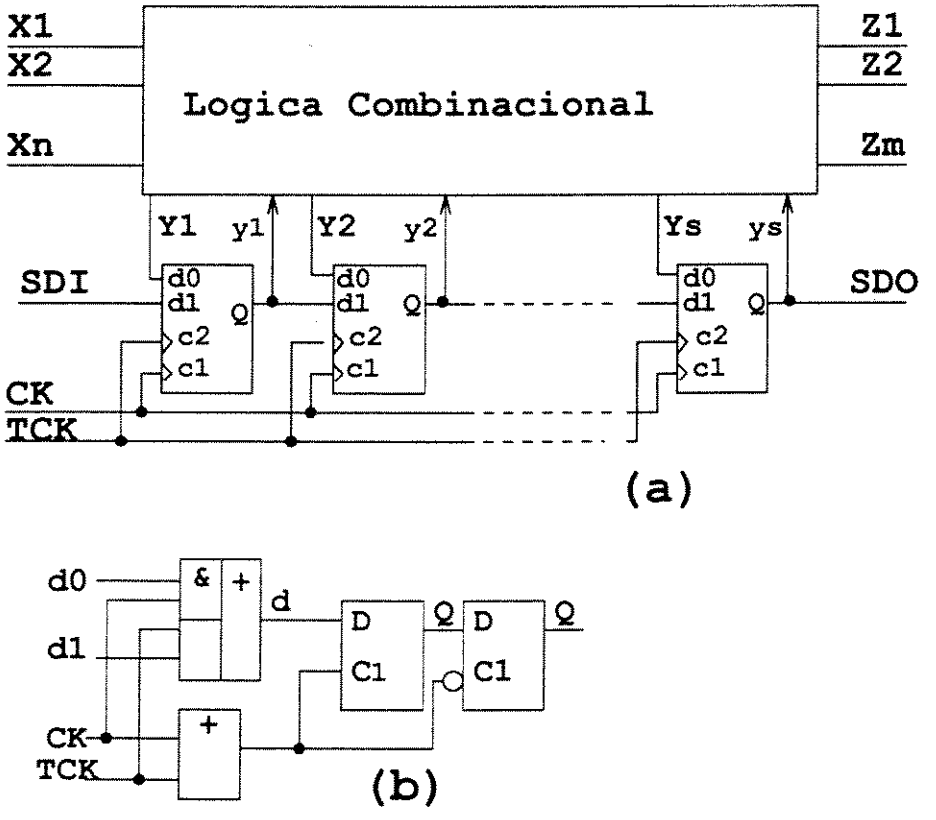

Figura 2.6: Arquitetura de "Scan path" com "two port flip flops" (a); "Two port flip flop" (b)

Outra maneira de realizar o "scan path" através dos flip flops é conhecida como arquitetura de "two port flip flop", apresentada na figura 2.6. Nesse caso, os flip flops (figura2.6 (b)) têm duas entradas de controle e o dado amostrado depende da entrada de controle pulsada. O procedimento de teste da lógica combinacional apresenta como diferença relativamente ao descrito para a arquitetura anterior o seguinte: O "scan in" e o "scan out" são acionados pelo sinal TCK, e o sinal de controle CK amostra os dados de saída da lógica combinacional em teste.

2.4.2 "Scan" para projetos com "Latches" de duas fases

Esta técnica originalmente denominada **Level Sensitive Scan Design (LSSD)**, tem como característica prover a capacidade de realizar "scan path" e produzir um sistema em que a resposta de estado estável para as variações de entrada permitidas é independente de atrasos de circuito e de interconexões. Para tanto impõe o uso de "hazard-free latches"[15] e uma arquitetura de circuito como a da figura 2.7.

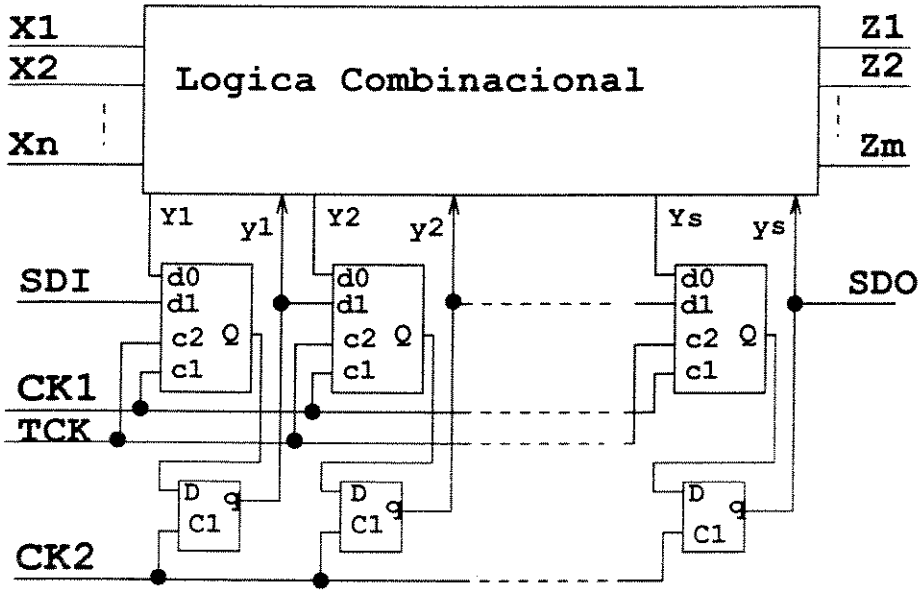

Figura 2.7: Arquitetura de "Scan path" para projetos com "latches" de duas fases

Para essa arquitetura os testes são aplicados à lógica combinacional da seguinte forma:

- os vetores de teste y_i são injetados serialmente através de SDI, aplicando-se alternadamente pulsos aos sinais de relógio de teste TCK e de sistema $CK2$;
- são aplicados os valores de X_i correspondentes e após tempo suficiente para propagação dos sinais através da lógica combinacional. observa-se os valores das saídas Z_k;
- um pulso de relógio de sistema $CK1$ aplicado permite amostrar as saídas da lógica combinacional nos respectivos "latches";

[15] "Hazard-free latches" são aqueles cujos sinais de saída não dependem dos atrasos de seus elementos e interconexões.

- os valores amostrados são serializados para fora do CI com a aplicação de pulsos alternados de $CK2$ e TCK, para que sejam verificados contra valores esperados.

No projeto de uma unidade de translação de endereço de uma memória virtual [Aylor86], essa técnica foi considerada mais econômica do que o uso de "scan-path", em termos de área total requerida e com melhor "performance", para o mesmo grau de observabilidade e controlabilidade.

2.4.3 Outras Técnicas de Projeto com "Scan"

1. **Lógica de "scan / set"**

   Similar ao LSSD e ao "scan-path" uma vez que são usados registradores de deslocamento para carregar e descarregar dados. Entretanto, estes registradores não são parte do caminho de dados do sistema e todos os "latches" do sistema não são necessáriamente controláveis e observáveis através dos registradores.

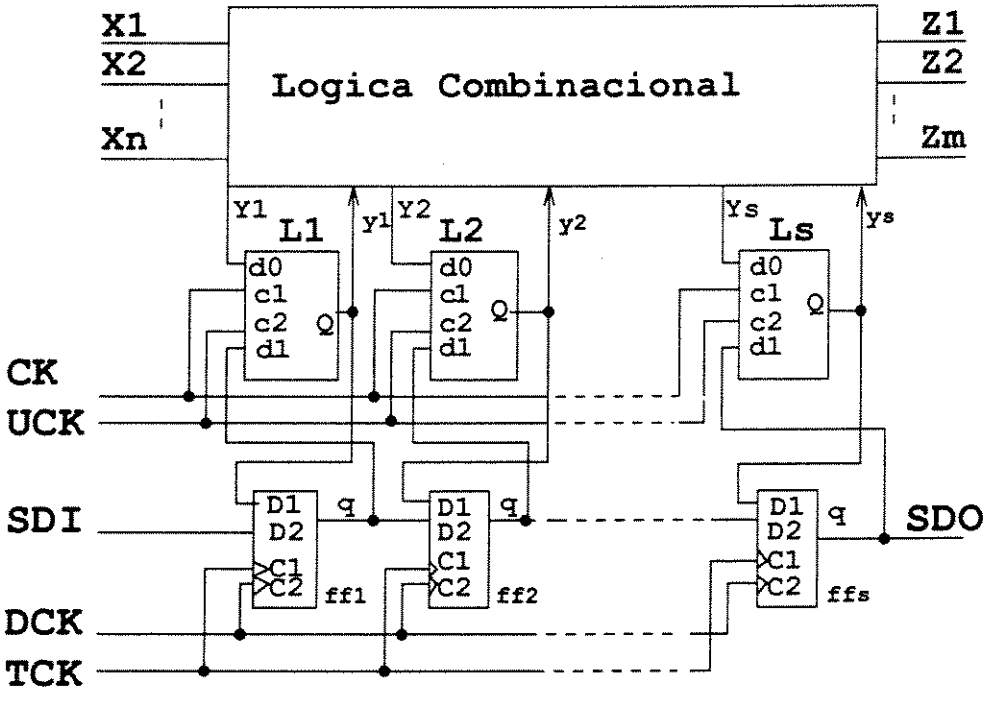

Figura 2.8: Arquitetura de "scan/set"

Pontos do circuito podem ser serialmente controlados e/ou amostrados

por flip flops de um registrador de deslocamento como na arquitetura da figura 2.8. Os dados de teste são injetados serialmente no registrador de deslocamento através de SDI pulsando TCK. Os dados de teste são transferidos aos "latches" do sistema em paralelo, aplicando-se um pulso de UCK. Para observar os dados de saída Y_i da lógica combinacional eles são amostrados nos "latches" L_1 a L_s com CK; as saídas Q dos "latches" são carregadas em paralelo nos flip flops do registrador de deslocamento, com um pulso de DCK. A seguir são enviados para fora do CI por SDO com acionamento através de pulsos em TCK.

Esta técnica não requer que todos os "latches" do sistema sejam carregados e deslocados para a saída, o que é uma vantagem pois oferece maior flexibilidade. Isso implica em que a geração de testes e a simulação de falhas não é necessariamente reduzida a circuitos totalmente combinacionais, mas de qualquer maneira reduz essa tarefa. Outra vantagem é que a função de"scan" pode ocorrer durante a operação normal do sistema. Portanto, pode-se obter valores instantâneos de sinais internos,de nós sequenciais ou não, sem qualquer degradação na sua performance. Porém, apresenta um "overhead" equivalente a dois "latches" por "latch" do sistema para o estágio correspondente de registrador de deslocamento, portanto maior do que o "overhead" associado com a técnica LSSD.

2. **Estruturas de "Scan" com Multiplexação**

   Através dessa técnica são observáveis dados paralelos provenientes de saídas de "latches" ou de outros nós do CI, através de serialização realizada com o uso de multiplexação.

   Essa arquitetura, apresentada na figura 2.9, permite exteriorizar os nós conectados ao multiplex. de acordo com a habilitação dada por $SADR$, através da saída SDO. Para aumentar a velocidade da observação vários (K) pinos de saída do CI devem ser usados. Para evitar um "overhead" muito grande em pinos para teste, podem ser usadas multiplexações com pinos funcionais.

3. **"Scan" de acesso aleatório**

   Apresenta o mesmo objetivo de "Scan-path" e LSSD, que é a completa controlabilidade e observabilidade de todos os "latches" internos. Difere dessas duas técnicas por não utilizar registradores de deslocamento. Usa um esquema de endereçamento que permite que cada "latch" seja exclusivamente selecionado, para que possa ser controlado ou observado. O mecanismo para endereçamento é muito similar ao de uma memória de acesso aleatório. Uma arquitetura para esta técnica é mostrada na figura 2.10. Utiliza uma entrada de "scan" ("scan data in" = SDI) que só é amostrada pelo relógio de teste TCK quando o endereçamento o habi-

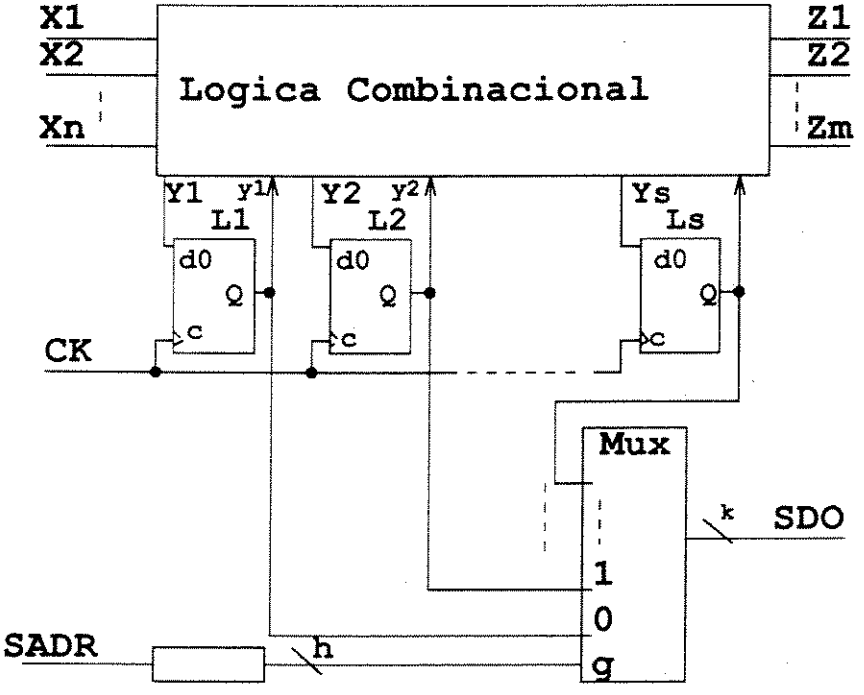

Figura 2.9: Arquitetura de "scan" com multiplexação

lita, através de S_i. A observação do "latch" é feita pela saída de "scan" (SDO). CK é o relógio do sistema e Y_i é o dado do sistema.

Básicamente, a configuração de sistema dessa técnica inclui um decodificador de endereços, os elementos de memória endereçáveis, além da máquina sequencial com relógios de sistema e de teste. Outra configuração proposta por esta técnica é um "latch" de set/reset endereçável e o mecanismo de observação também é pela saída de "scan". Ainda, qualquer ponto no circuito combinacional pode ser observado com a adição de uma porta lógica e de um endereço por ponto de observação [Williams82]. Esta técnica apresenta um "overhead" de 3 a 4 portas por elemento de memória e um "overhead" da ordem de 6 pinos caso se use "scan" serial para o contador de endereços.

4. "Scan Path" de Entrada / Saída

A estrutura de "scan-path" de entrada e saída é mostrada na figura 2.11. Os "latches" do sistema são implementados em projeto tipo LSSD para formar um "scan-path" para finalidade de teste. Um par de "latches" de "scan-path" é introduzido para cada pino de entrada e/ou saída. Estes "latches" de entrada/saída são configurados como um outro "scan-path" do tipo LSSD, chamado "scan-path" externo ou anel. O procedimento de

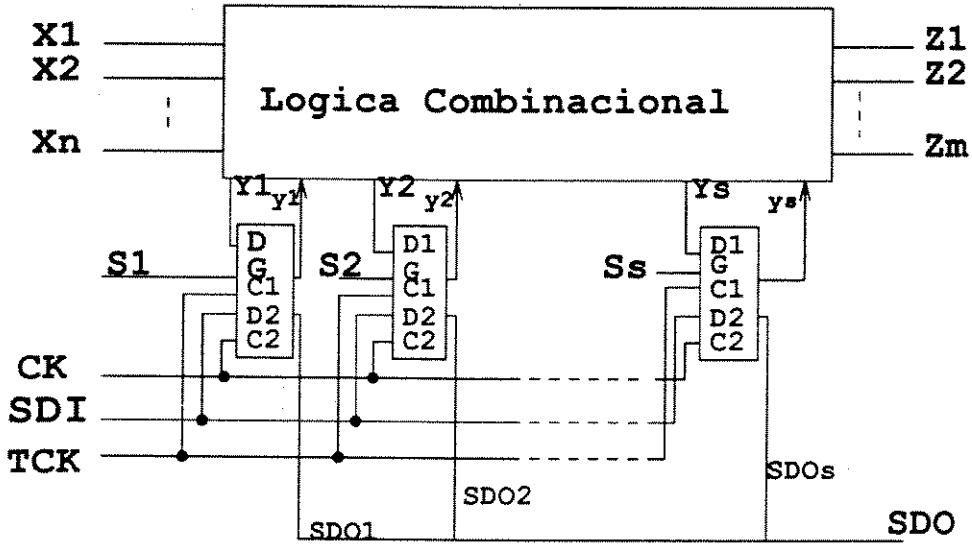

Figura 2.10: Arquitetura de "scan" com acesso aleatório

teste é similar ao descrito para LSSD.

A presença do "scan" externo permite fazer o teste do chip através de um número pequeno de pinos de prova, já demonstrado para projeto de CI de 256 pinos, o qual pode ser testado com 7 pinos de controle e dois de alimentação (potência e terra), conforme mencionado em [McCluskey86]. Essa estrutura permite configurar o anel externo como um oscilador em anel. Isso é feito conectando-se o último "latch" do "scan" externo à entrada do primeiro deles. Esta estrutura também pode ser facilmente configurada para auto-teste.

Esta técnica aperfeiçoa a testabilidade do CI, reduzindo os requisitos para o equipamento de teste. É uma extensão da técnica de projeto com "scan" para a qual foi desenvolvido um padrão IEEE pelo grupo JTAG ("Joint Test Action Group"), denominado "boundary-scan", para placas e circuitos integrados [McCluskey86]. O "boundary-scan" ou "scan" de entrada / saída também é aplicável a grandes subcircuitos tais como microprocessadores e PLA's. Às entradas e saídas desses subcircuitos é associado um registrador de deslocamento para facilitar o seu teste.

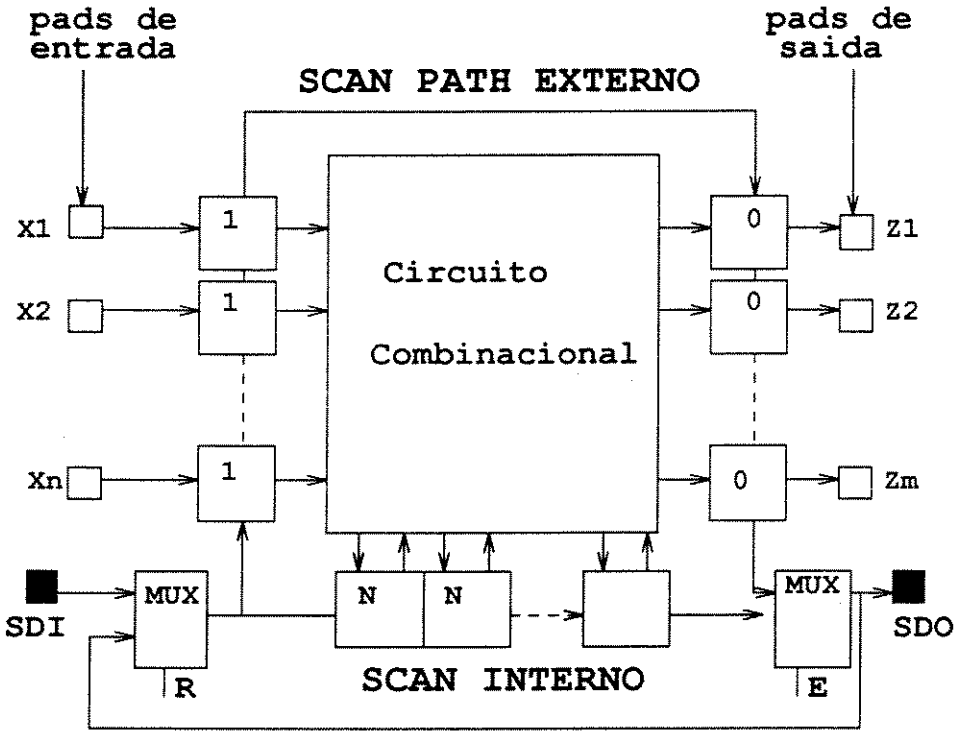

Figura 2.11: Arquitetura de "scan-path" de entrada/saída

2.5 Circuitos Auto-testáveis

Os circuitos mais complexos e facilmente testáveis são capazes de gerar uma parte ou todos os estímulos necessários para o teste e de efetuar auto-avaliação das saídas resultantes de teste. Eles são conhecidos como circuitos auto-testáveis pois contêm toda a lógica necessária para a geração de seus estímulos, armazenamento ou operação das respostas e sua avaliação. Cabe ao testador externo a inicialização do procedimento de teste pela geração de um sinal, espera pela execução do procedimento e verificação do resultado que é do tipo **passa/falha**, expressa por um pino de saída [Wagner88]. Trata-se, portanto de compensar um pequeno "overhead" em área de Silício por geração e verificação de teste no próprio circuito, conhecida como BIST[16], evitando-se a necessidade de um custoso testador automático[17] .

No enfoque BIST, para fazer os sinais internos mais controláveis e observáveis é feito um particionamento do CI de modo a permitir testar exaustivamente

[16] Originalmente built-in self test (BIST).
[17] Os testadores automáticos atingem preços na faixa de 2 milhões de dólares.

ou aleatoriamente cada bloco .

Há vários fatores que afetam as decisões quanto ao uso de circuitos auto-testáveis: a qualidade de detecção de falha desejada, o tempo de aplicação de teste aceitável, o "overhead" permitido de área de silício devido a hardware adicionado e a degradação em performance suportada. A inclusão de circuitos para o auto-teste é chamada "built-in self test" e pode ser encarada como um conjunto de três tarefas: a estratégia de geração de entradas de teste, a de avaliação das respostas e a implementação dos mecanismos.

2.5.1 Geração de estímulos

Em circuitos auto-testáveis os estímulos são gerados internamente aos próprios circuitos, podendo ser calculados, obtidos de tabelas ou gerados por circuitos especificamente projetados para isso. Há dois enfoques de auto-teste: um para teste **pseudo - aleatório**, usa padrões gerados aleatoriamente e o outro para teste **exaustivo**, usa todas as combinações possíveis, não requerendo assim modelo de falha e/ou simulação de falha. O enfoque de teste pseudo - aleatório requer simulação de falha para determinar a cobertura de falha do padrão.

Os padrões de teste pseudo - aleatórios podem ser gerados por um programa se é disponível no CI um processador de instruções. Caso contrário, eles podem ser gerados por meio de um circuito simples chamado registrador de deslocamento de realimentação linear (ALFSR de "autonomous linear feedback shift register"). Esse circuito é implementado por um registrador de deslocamento sem entradas externas a ele e com as realimentações realizadas através de portas ou-exclusivo, como o exemplo da figura 2.12.

Esse ALFSR está associado a um polinômio, chamado de função geratriz, que para o exemplo da figura 2.12 é:

$$f(x) = x^4 + x^3 + 1$$

O circuito ALFSR gera uma sequência de $2^n - 1$ estados diferentes, onde n é o número de estágios do registrador.Há um estado proibido (0000 para o exemplo), no qual o circuito permanece se isso não for devidamente evitado[18]. A função gerada correspondente ao ALFSR de máximo comprimento em número

[18] O circuito pode ser alterado para permitir o estado proibido. Neste caso o número máximo de estados passa a ser 2^n.

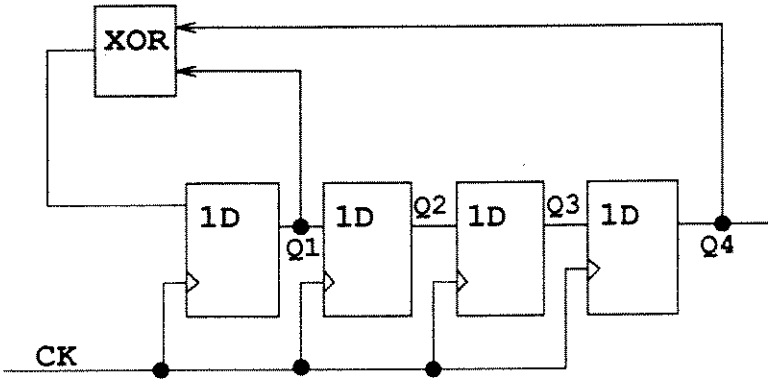

Figura 2.12: ALFSR - Circuito para geração de estímulos pseudo-aleatórios

de estados é chamada polinômio primitivo. Esses polinômios podem ser encontrados em [Peterson72].

Estímulos exaustivos ou pseudo-exaustivos podem ser gerados por um contador, porém isso é mais econômico utilizando um ALFSR e permitindo que a sequência passe pelo estado proibido, através de modificação no circuito.

Os estados gerados pelos ALFSR podem ser aplicados a entradas de blocos de circuitos integrados, diminuindo o tempo de inserção de estímulos de teste ao circuito completo.

2.5.2 Análise de respostas

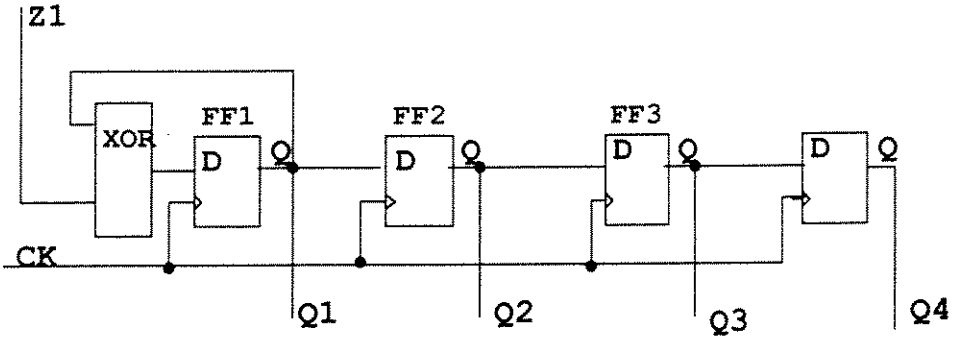

Figura 2.13: Arquitetura de análise de assinatura

Para análise dos vetores de saída por comparação com resultados esperados é usada a técnica de compactação ou compressão dos resultados, realizada por

analisadores de assinatura.

Análise de assinatura (originalmente Signature Analysis) é uma técnica de compressão de dados, baseada em cálculos de código de redundância cíclica (CRC) [Wagner88]. A componente principal dessa técnica ilustrada pela figura 2.13 é um registrador de deslocamento de realimentação linear (Linear Feedback Shift Register - LFSR), definido de modo análogo ao ALFSR usado para geração de estímulos para teste.

A análise de assinatura para um nó Z1 é feita inicializando-se o LFSR e verificando-se os níveis lógicos remanescentes nas saídas dos flip flops, após um número fixo de pulsos de relógio. A assinatura é composta dos níveis lógicos de Q1, Q2, Q3 e Q4, resultante da compressão do trem de pulsos amostrado no nó Z1. A assinatura não é o valor que o LFSR conta durante esses relógios, pois o sinal em teste não é necessariamente uma sequência de 1's. Qualquer mudança no comportamento do nó em teste deverá produzir uma assinatura diferente, indicando que há falha. Há no entanto, possibilidades de ocorrer bits errados no trem de pulsos e mesmo assim produzir uma assinatura correta.

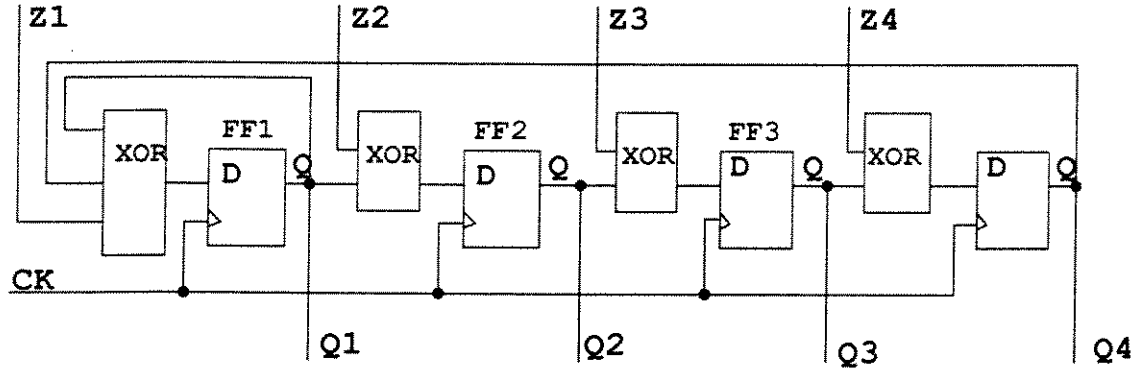

Figura 2.14: Arquitetura de análise de assinatura de múltiplas entradas

Para análise de múltiplas saídas de circuito há duas propostas de solução. Uma delas faz multiplexação das saídas e executa a análise serialmente. A outra propõe um analisador de assinatura paralelo,também conhecido como MISR (Multiple Input Signature Register). A arquitetura de um analisador de assinatura paralelo de 4 estágios é ilustrado na figura 2.14. Em geral os MISR são mais rápidos que os analisadores que serializam os pontos em teste, embora requeiram mais circuitos adicionais.

O objetivo de se utilizar análise de assinatura é projetar um circuito que possa avaliar a si mesmo. O "overhead" em área associado com essa técnica pode ser reduzido se for possível utilizar registradores existentes no próprio circuito a ser testado. Essa técnica tem a desvantagem de exigir avaliação da cobertura de falhas através de simulação de falhas e se a cobertura for insuficiente, não há orientação prática suficiente para alterar os vetores de teste para incrementá-la [Wagner88].

2.5.3 Estruturas baseadas em BIST

Há vários esquemas para uso de BIST em CI's, alguns supõem a adição de circuitos (LFSR), outros aproveitam estruturas existentes no circuito reconfigurando-as como geradores de padrão ou analisadores de assinatura para fins de teste, e outros ainda fazem uso de esquema de verificação concorrente do próprio circuito para verificação "off-line".

Além disso, são considerados BIST de acordo com a classificação deste documento, técnicas conhecidas como métodos microprogramados, métodos orgânicos e a técnica lançada recentemente com o nome de "cross check", pela sua característica de embutir hardware e prover circuito para análise de assinatura.

1. Métodos baseados em BILBO

   São métodos que se baseiam na reconfiguração de circuitos de "scan-path" e que usam o módulo "Built In Logic Block Observer" (BILBO), divulgado no 1979 International Test Conference [Konemann79]. O registro BILBO é o módulo BIST mais comumente utilizado, tendo atributos de LSSD e "Scan-Path", de análise de assinatura e de geração de padrões pseudoaleatórios. Aplica-se a circuitos que podem ser particionados para teste em módulos independentes, permitindo isolar blocos puramente combinacionais. A cada módulo é associado um registro de entrada e um de saída, caso ele não tenha no projeto original, ou é utilizado um BILBO reconfigurável como o da figura 2.15 .

   Esse BILBO é tal que seus registros são reconfigurados para que atuem como:

   - registros de leitura: com B1 = 1 e B2 =1, Zi's são entradas diretas para os FF's.
   - registrador de deslocamento: com B1 = 0 e B2 = 0, é possível realizar "scanin" e "scanout".

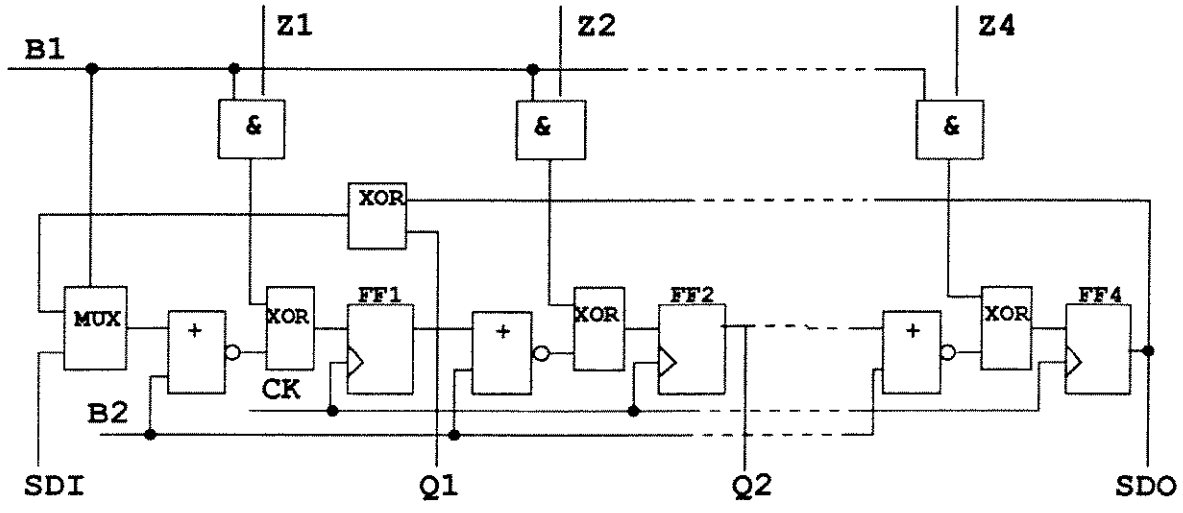

Figura 2.15: Arquitetura de "BILBO"

- analisador de assinatura de entradas múltiplas: com B1 = 1 e B2 = 0. Nesse modo, o BILBO também pode ser usado para gerar sequências pseudo-aleatórias, bastando para isso manter fixas as entradas lógicas Zi's.
- reset no registrador: com B1 = 0 e B2 = 1.

Quando são disponíveis no circuito um controlador inteligente e um sistema de barramento é possível criar esquemas eficientes de teste utilizando o conceito de BILBO, os quais são também chamados de métodos microprogramados . A Motorola usou esse método com o microcomputador Mc6804p2 [Baker].

2. Métodos orgânicos

São métodos estritamente adequados a uma dada estrutura regular, como por exemplo RAM's, ROM's ou PLA's, que devido ao tamanho ou à necessidade de uma sequência de teste especial não são compatíveis com as técnicas propostas, necessitam de estratégia de DFT específica. Estes métodos podem ser muito efetivos, mas seguramente implicarão em grande "overhead" em área [Baker].

Para essas estruturas regulares os métodos estruturados não funcionam [Baker], devido a:

- Necessidade de transformar cada elemento de memória em um FF de caminho de "scan" acarretar um "overhead" muito grande.

- Necessidade de precisão dos modelos de falhas.
- Necessidade de aplicar longos testes para detectar sensibilidade a padrão para RAM's e ser requerido teste exaustivo para ROM's, o que requer tempo de teste muito grande. O acesso por "scan" aumentaria ainda mais o tempo de teste. Deve ser previsto acesso paralelo às entradas dessas estruturas.
- Diminuição de performance devido aos atrasos introduzidos pelas lógicas do "scan", o que pode ser inaceitável para essas estruturas.

3. Técnica de "cross check"

   Para projetos baseados em matrizes de células, do tipo semi dedicados ou para ASIC's (circuitos integrados de aplicação específica), foi proposta a técnica de **"cross-check"** [Swan89]. Essa técnica adiciona uma espécie de cama de pregos ao circuito original, permitindo uma observabilidade quase ilimitada do circuito, e dispondo de um bloco para compactação dos valores lidos do CI, através de um MISR. Não tem entretanto, efeito algum sobre a sua controlabilidade. Através de software são exploradas as facilidades do hardware para acelerar a geração de padrão e simulação de falhas sobre um amplo conjunto de modelos de falha. A facilidade de acesso também favorece a depuração de protótipo e o diagnóstico do CI.

2.6 Regras de Testabilidade

Regras de testabilidade são diretrizes ou restrições de projeto que apresentam como objetivo a execução e verificação de projetos quanto a técnicas ou metodologias de projeto para testabilidade. Elas são utilizadas para assegurar a correta aplicação de uma metodologia de DFT, contribuindo para evitar repetições de fases do ciclo de desenvolvimento de CI's. As técnicas de projeto estruturado apresentam características que facilitam a geração de regras, embora possam também ser geradas regras em função dos outros tipos de técnicas de DFT.

2.6.1 Projeto estruturado

As regras geradas a partir das disciplinas de projeto estruturado são função das imposições para obter projetos tais que seja possível generalizar a geração de padrões de teste. Os circuitos devem ser síncronos, embora seja possível aplicar métodos de "scan" a projetos com mais de um relógio de sistema ou até

a circuitos assíncronos. As regras para projeto estruturado tem sido utilizadas por vários autores e se baseiam nas regras definidas em [Eichelberger77]. O enunciado dessas regras difere algumas vezes devido a alguma simplificação ou particularização.

As regras definidas em [Eichelberger77] tem por objetivo o projeto de sub sistemas lógicos sensitivos a nível (LSSD) e verificáveis através de "scan". São definidas para utilização de um "latch" específico que inclui lógica para realizar um registrador de deslocamento e podem ser enunciadas como:

1. Os elementos de memória do circuito devem ser implementados por "latches" que não apresentem problemas de "hazard"[19], e cujos dados armazenados não mudam se o relógio está inativo.

2. Pode haver dois ou mais relógios de sistema mas não deve haver superposição entre eles. Cada "latch" é acionado por um relógio, sendo que:

   Um "latch" X pode alimentar a entrada de dados de um outro "latch" Y se e somente se o sinal de relógio que amostra o dado no "latch" Y não amostra o dado para o "latch" X.

   Um "latch" X pode chavear um sinal de relógio C1 para produzir um relógio chaveado C1G que é entrada de relógio de um outro "latch" Y se e somente se o relógio associado ao "latch" X não é produzido a partir de C1.

   Esta regra visa evitar problemas de corrida ("race"[20]) entre sinais de relógio e dado.

3. Há um conjunto de entradas primárias de relógio a partir das quais as entradas de relógio dos "latches" são controladas quer seja através de uma árvore de distribuição de relógios, ou através de lógica chaveada por "latches" e/ou entradas primárias comuns (que não são de relógio), tal que se assegure que:

   Todas as entradas de relógio de todos os "latches" possam estar inativas, quando todas as entradas primárias de relógio estão inativas.

   Seja possível colocar um sinal de relógio no estado ativo, colocando-se o pino correspondente no estado ativo.

   Nenhum relógio seja chaveado ("ANDed") com outro sinal de relógio ou com o seu complemento.

[19] "Hazard" é a geração de um pulso de sinal indesejado na saída de uma lógica combinacional, que ocorre devido a chaveamentos de sinais de suas entradas.

[20] "Race" é a situação em que mais do que um nó interno é instável, podendo provocar um estado final errado para o circuito, devido a caracterísitcas de atraso de células e interconexões do caminho.

4. Entradas primárias de relógio não podem alimentar as entradas de dados dos "latches", nem diretamente nem através de lógica combinacional; somente podem alimentar entradas de relógios de "latches" ou saídas primárias.

Segundo [Eichelberger77], as regras acima produzem um projeto sensitivo a nível e para simplificar os testes, minimizar pinos de entrada e saída e permitir controlar e observar o conteúdo dos "latches" recomenda as seguintes regras adicionais:

1. Todos os "latches" e flip flops devem pertencer a um registrador de deslocamento, com uma entrada e uma saída primária e sinais de relógio disponíveis nos pinos.
2. Uma entrada primária do CI deve impor uma condição de "scan" associada a um nível, tal que:

   Cada "latch" ou saída de "scan" é alimentado por um único "latch" ou por uma entrada primária.

   Todos os relógios exceto o de "scan" são mantidos inativos na situação de "scan".

   Qualquer relógio para "scan" pode ser ativado e desativado a partir de suas respectivas entradas primárias.

Várias ferramentas foram propostas para verificar esse tipo de regras [Godoy77], [Horstmann84], [Camurati88], [Cosgrove88], [Adhem87]. Algumas delas verificam ou inserem BIST, além de regras para LSSD. Em [Agrawal84] essas regras são simplificadas e referidas ao uso de flip flops master slave do tipo D. [Bidjan-Irani91] trabalha com algumas dessas regras.

2.6.2 Outras recomendações

É possível basear-se nas recomendações geradas a partir das Técnicas AD HOC ou das Técnicas de Circuitos Auto-testáveis e gerar regras para verificação. É provável que a automatização de tarefas de projeto que obedeçam essas regras não sejam tão facilmente generalizáveis, mas a sua verificação pode oferecer bons resultados. Por exemplo, considerando-se as técnicas AD HOC é possível verificar a existência de conflitos entre sinais para acesso a pontos importantes de "fan in" ou de "fan out" de um dado circuito. Ou então verificar a existência

de "wired logic". Ou ainda, o acesso direto a estruturas do tipo RAM's, ROM's e PLA's, por entradas e saídas paralelas multiplexadas dentro do CI.

Exemplificando, o Test Assistant [VTI] insere automaticamente multiplexações a blocos de circuito, definidos pelo projetista. Essa ferramenta de auxílio à síntese para testabilidade focaliza esse último tipo de regra AD HOC mencionado.

Já as estruturas que implementam Técnicas de Circuitos Auto-testáveis podem ser verificadas quanto a padrões previstos para os BILBO' s (geradores de estímulos ou analisadores de assinatura), através de análise da descrição topológica dos circuitos.

Ferramentas que verificam esse tipo de metodologia (BIST) ou que as inserem nos circuitos são descritas em [Kim88], [Fung86], [Abadir89], [Kalinowski88], [Camurati88].

Há ferramentas que se preocupam com metodologias específicas para um estilo de projeto (Metodologias Orgânicas). É o caso de PLAESS [Breuer85] e PLATA [Yousuf88], duas ferramentas que permitem ao projetista avaliar várias metodologias de projeto de PLA'S e seus reflexos sobre área e performance.

Outras características de ferramentas de auditoria ou inserção de metodologias de DFT são mencionadas na seção 3.3.

2.7 Auditoria de Testabilidade e Automatização

Os métodos de DFT atacam a raiz do problema que é a controlabilidade e a observabilidade de nós dos CI's, enquanto que a geração de teste se preocupa com o sintoma que é a complexidade do teste. Ambos adquirem um certo peso no ciclo de desenvolvimento de circuitos integrados devido aos custos relacionados com o teste. A economia do teste para ser otimizada precisa ser considerada de modo completo [Baker]. Assim também a tarefa de concepção com objetivo de projetar para ser testável precisa ser pensada de modo global, desde o início do desenvolvimento, principalmente para circuitos complexos em que a necessidade de automatizar as tarefas é indispensável.

Há problemas insolúveis em automação de teste e projeto como é o caso de grandes circuitos sequenciais assíncronos. Várias companhias (IBM, Fujitsu, Sperry-Univac e Nippon Electric Co.) já reconheceram como resultado de

suas pesquisas, que projeto não estruturado pode conduzir a problemas de teste inaceitáveis [Williams82].

A adoção de projeto estruturado oferece como benefícios:

- A simplificação de geração de teste, reduzindo-a aos circuitos lógicos combinacionais.
- A simplificação do teste propriamente dito.
- A capacidade de monitorar dinâmicamente todos os elementos de memória, eliminando a necessidade de pontos de teste especiais e simplificando a depuração.
- A simplificação do desenvolvimento de ferramentas para projeto, verificação e simulação.
- A redução de problemas devidos a modificações de projeto.

Quando se usa técnicas de DFT, verificadores de regra de algum tipo são também quase essenciais. O problema se divide em dois: verificar regras e formular regras sensíveis, que assegurem um projeto testável e que ao mesmo tempo sejam praticáveis. No nível estrutural é fácil formular um conjunto de regras tais como aquelas adequadas a LSSD ou para "scan-path", mas é difícil fazer os projetos obedecerem essas regras.

O que é observado da prática é que regras de DFT que especificam o comportamento de um "scan-path" ou um barramento de dados e não sua estrutura, são mais aceitas e abrangem maior quantidade de circuitos. Além disso, um esquema de DFT no nível comportamental pode ser verificado mais cedo.

A verificação de regras de projeto para testabilidade no nível estrutural tem características repetitivas que indicam a automatização como solução. A identificação dos elementos que satisfazem regras de projeto para DFT, uma vez definidas, é uma tarefa que pode ser realizada com maior confiabilidade por uma ferramenta. Essa atividade no contexto das ferramentas existentes para a fase de concepção de circuitos integrados é de pequeno investimento, podendo no entanto representar significativa economia no ciclo de desenvolvimento.

Capítulo 3

Ferramentas de Apoio ao Projeto de CI's Testáveis

3.1 Introdução

Com o advento da era de circuitos integrados VLSI (Very Large Scale Integration) o uso de ferramentas de auxílio ao projeto (CAD) tornou-se indispensável. As ferramentas de CAD não só são necessárias para projeto e verificação de circuitos integrados, como também a disponibilidade dessas ferramentas deve possibilitar condições vantajosas para competir no mercado de ASIC's (circuitos integrados de aplicação específica) da próxima década [Sangiovanni87].

Os eixos da pesquisa atual em ferramentas de CAD são: **a síntese de circuitos** (leiaute, síntese lógica e arquitetural), **a verificação de CI's** (incluindo verificação de regras elétricas (ERC), verificação de regras de leiaute (DRC), simulação e técnicas de verificação formal) **e as ferramentas de gerenciamento de sistemas de projeto** (para versões e alternativas num ambiente de projeto distribuído, gerenciamento de dependência de dados e interfaces eficientes e flexíveis para novas ferramentas).

O desenvolvimento de técnicas de inteligência artificial influenciou significativamente o desenvolvimento de ferramentas CAD para síntese e verificação de CI's e em particular, deu impulso aos sistemas de apoio ao projeto de CI's testáveis.

As ferramentas de apoio ao projeto de circuitos são associadas às características

relevantes de projeto de CI's nos vários níveis de refinamento, pelos quais eles são descritos (ver subseção 3.2.1). As classificações associadas às ferramentas de apoio aos projetos são objeto da subseção 3.2.2.

Um sistema de síntese completo deveria gerar máscaras de leiaute a partir das especificações de nível comportamental, algorítmica ou funcional; da descrição de uma tecnologia escolhida e da descrição das funções de restrições e de custos. A idéia de Compilador de Silício é suportar todo esse processo. Muito ainda está por ser feito em termos de síntese de circuitos [Sangiovanni87], embora já existam importantes contribuições resultantes da pesquisa sobre compiladores de silício, dentre as quais o desenvolvimento de linguagens de projeto procedural e o uso de sistemas baseados no conhecimento, descritos na subseção 3.2.3.

Para projeto de CI's testáveis algumas ferramentas já desenvolvidas são de síntese, outras são de verificação (ver seção 3.3). As ferramentas de síntese com testabilidade caracterizam-se por incluir ou modificar os projetos em função de metodologias de testabilidade, as quais geram um objeto: um projeto ou uma modificação de projeto. As ferramentas de verificação geram relatórios, e podem ser analisadores, isto é, baseiam -se em descrições de projeto para estimar valores de controlabilidade e observabilidade, ou podem ser auditores para verificar essas descrições relativamente à metodologias de DFT.

Há um conjunto razoável de verificadores ou auditores de testabilidade implementados segundo diversas técnicas. Os avanços mais recentes em auditores de testabilidade tiveram grande impulso devido ao desenvolvimento de Sistemas Especialistas.

O ambiente para projetos de CI's tem sofrido grandes modificações. O reflexo dessas mudanças e as consequências do uso de Sistemas Especialistas são comentados na seção 3.4.

3.2 Conceitos de ferramentas de apoio ao projeto

3.2.1 Características de Projeto Digital

O projeto de CI's digitais é uma tarefa executada sobre três eixos, através de muitos níveis de refinamento, indo da mais abstrata arquitetura ao mais detalhado leiaute, representável pelo diagrama Y de Gajski-Kuhn [Gajski83].

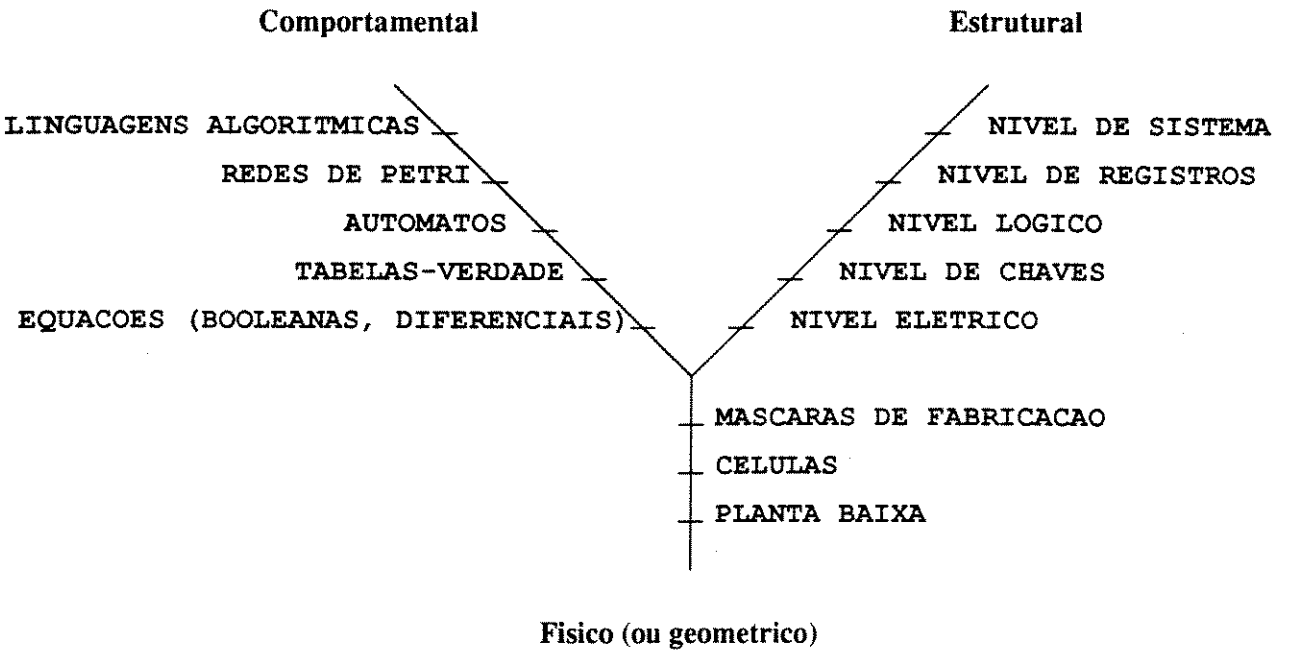

Figura 3.1: Diagrama Y: eixos e níveis de projeto

ilustrado pela figura 3.1 .

Algumas tarefas de projeto produzem mudanças de eixo relativamente ao diagrama Y e outras se processam sobre o mesmo eixo, envolvendo diferentes níveis ou um mesmo nível de descrição. O centro do Y é associado ao sistema físico gerado. Tarefas de transformação de um nível mais alto de especificação de projeto para um nível mais baixo são tarefas de síntese; podem envolver diferentes eixos ou diferentes níveis de abstração, como ilustram as figuras 3.2 e 3.3.

As tarefas de análise realizam verificações ou simulações para descrições de diferentes níveis ou eixos de projeto, ou para descrições de um nível/ eixo contra especificações de regras ou objetivos.

"A síntese funcional transforma uma descrição comportamental do circuito, dada sob a forma de algoritmo, em uma representação estrutural ao nível de transferência de registradores ou portas lógicas, ou em uma representação mista (estrutural e comportamental) ao nível de transferência de registradores e máquinas de estado" [Wagner88].

Síntese lógica gera como produto uma representação do circuito ao nível lógico. Quando um sistema de síntese parte de uma descrição do circuito em níveis altos, mais comumente do nível lógico, e gera uma descrição física do circuito, esse sistema é designado como compilador de Silício. A representação de compilador de Silício no diagrama Y aparece na figura 3.4. Os compiladores de Silício comerciais se baseiam em biblioteca de células, em que os componentes elementares são células parametrizáveis.

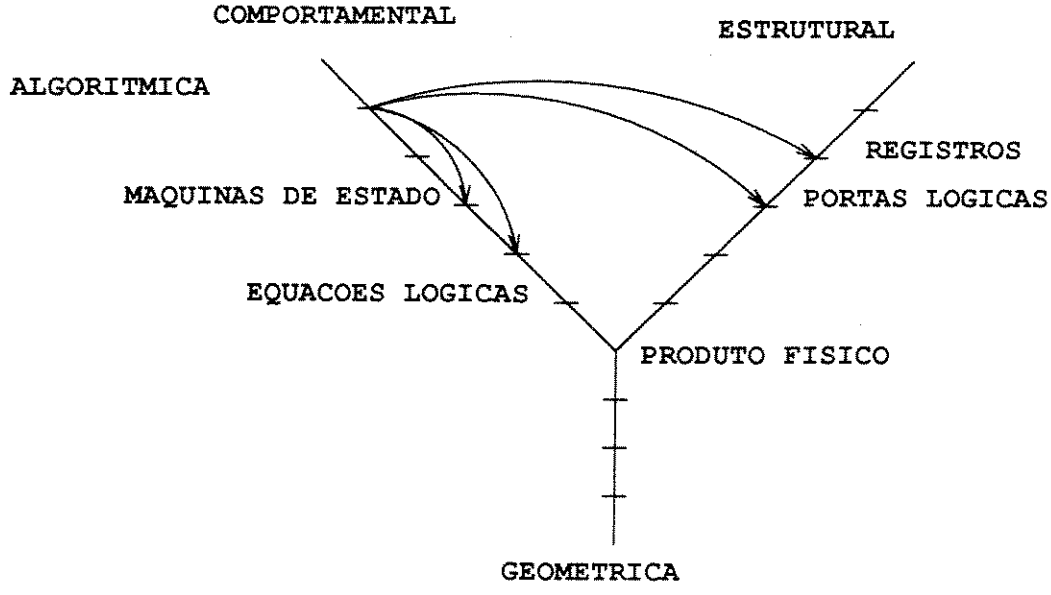

Figura 3.2: Diagrama Y: síntese funcional

Para viabilizar a realização dos projetos são utilizados: sistemas CAD ("computer-aided design"), softwares que envolvem descrições mais precisas das tarefas: ou sistemas CAE ("computer-aided engineering"), sistemas de projeto em estações de trabalho, com grande ênfase em análise de circuitos.

As ferramentas de apoio aos projetos lidam com quatro tipos de características de projeto eletrônico digital [Rubin87]:

- hierarquia estrutural, que é a característica segundo a qual um objeto pode ser visto como parte de outro.
- abstração, através da qual é possível utilizar elementos essenciais para cada perspectiva ("view") de projeto, abandonando-se detalhes de outras perspectivas.
- conectividade, que é dada pela coleção de caminhos através do circuito.

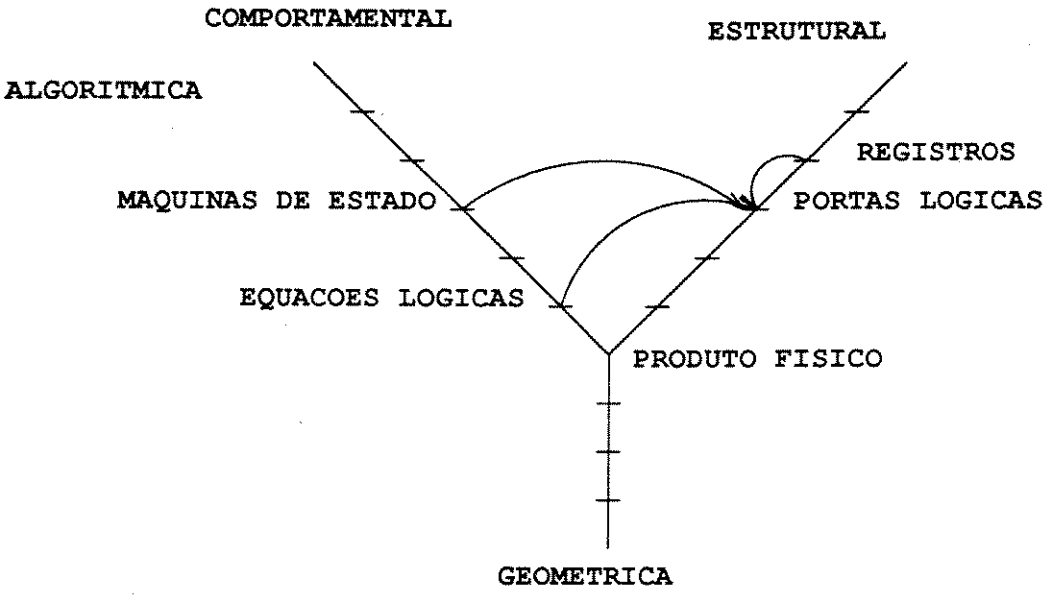

Figura 3.3: Diagrama Y: síntese lógica

conhecida como a topologia do circuito.

- dimensionalidade espacial, que é a natureza geométrica particular significativa de uma perspectiva. Por exemplo, o projeto físico VLSI consiste de camadas bidimensionais colocadas uma sobre a outra.

Com a característica de hierarquia estrutural são associados os conceitos de: instância (uso de uma célula num dado nível da hierarquia); célula-folha (uma célula da hierarquia que não tem descendentes, ou a primitiva da hierarquia); a célula-raiz (o mais alto nível da hierarquia); e o conceito de hierarquia de profundidade 1, associado a uma descrição achatada ou totalmente instanciada.

A organização hierárquica de projetos tem sido preferida por ser mais auto-documentada, mais fácil de entender, mais eficiente quando se tem blocos de utilização repetida e por ser efetiva e apropriada a metodologias de projeto "top down" e "bottom up".

3.2.2 Classificação das Ferramentas de Apoio ao Projeto

As ferramentas de CAD são resultantes da variedade e complexidade das tarefas envolvidas no projeto de circuitos e de suas características de repetição e

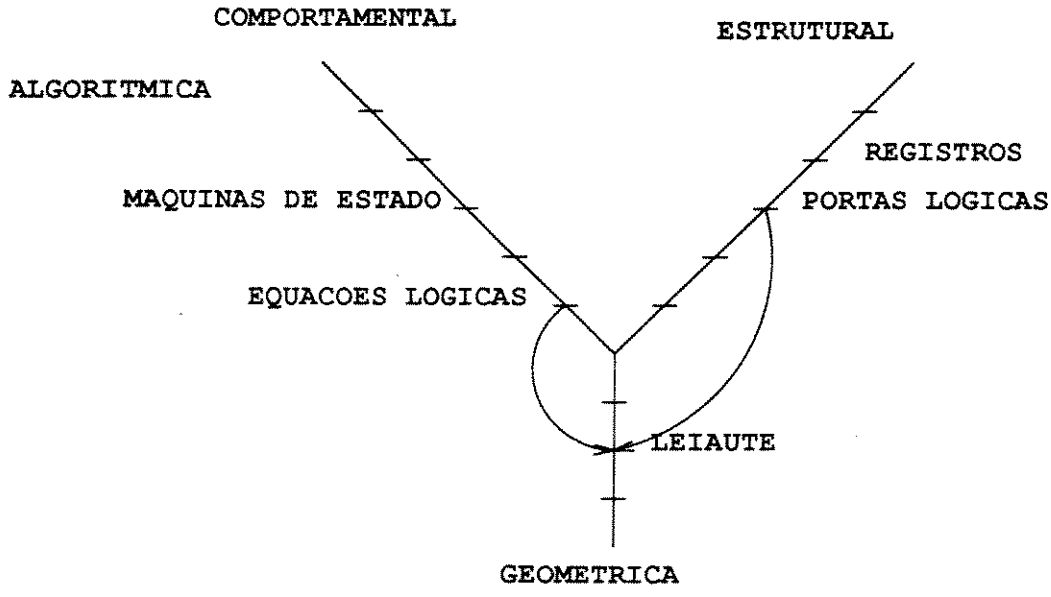

Figura 3.4: Diagrama Y: uma representação de Compiladores de Silício

regularidade, com objetivos de realizar CI's que executem funções especificadas de forma otimizada.

Estão associadas com linguagens de descrição de "hardware" e podem ser específicas para uma dada tarefa ou podem abranger um conjunto de tarefas sob uma mesma interface. Classificam-se em:

1. **ferramentas de síntese**, que realizam conversões entre diferentes níveis de refinamento de projeto e geram objetos de projeto (uma descrição lógica de circuito, um leiaute e outros). Partem da premissa que um circuito é composto de células e suas interconexões.

   Algumas ferramentas manipulam o conteúdo de células, trabalham no chamado nível local, isto é dentro das células de um CI. É o caso de leiaute de células, o qual é tipicamente restrito a padrões regulares. Outras ferramentas trabalham fora das células e obedecem características de projeto não tão regulares.

   As ferramentas para síntese de circuitos integrados são muitas, podendo aplicar-se a vários níveis de projeto ou a alguns deles. As ferramentas de síntese apresentam básicamente dois elementos fundamentais:

   - a característica de implementarem hardware a partir de uma linguagem de descrição hardware, ou uma ou algumas de suas sub-tarefas:

- e a realização de otimização, enriquecendo o projeto.

Há duas escolas de síntese de circuitos integrados, quando se trata da maneira como são enfocados os elementos fundamentais dessa atividade: uma delas faz síntese lógica e otimização em dois passos separados, e a outra faz ambas em uma única ferramenta. A primeira faz otimização em nível de portas lógicas e a segunda faz otimização em níveis mais altos de abstração. Os que propõem síntese integrada com a otimização, argumentam que o resultado final de tais ferramentas é muito mais eficiente (25% menos portas lógicas) que aqueles obtidos com ferramentas que fazem essas tarefas separadamente [Harding89].

As ferramentas de síntese que abrangem um número maior de tarefas são os compiladores de Silício, cujo objetivo é gerar máscaras de leiaute a partir de descrições de alto nível. Mas, devido à variedade e complexidade de tarefas de projeto, ainda não foi possível substituir todas as ferramentas de apoio ao projeto por um compilador de Silício único.

Compiladores de Silício modernos são capazes de traduzir descrições estruturais em leiaute de CI's, usando muitos dos mesmos passos que os compiladores tradicionais. Há duas classes mais comuns de Compiladores de Silício:

(a) geradores de módulo, que geram geometrias de leiaute a partir de uma descrição estrutural e/ou comportamental ao nível lógico do bloco (PLA's, memórias) a ser implementado.

(b) geradores de leiaute a partir da execução de programas de linguagem geométrica, chamada de projeto procedural.

Muitos compiladores de Silício são especializados em produzir um tipo de projeto, e nesse caso é possível usar linguagens de descrição de circuito de muito alto nível, podendo ser utilizado por não programadores.

2. **ferramentas de análise**, que podem ser dinâmicas ou estáticas, ajudam a verificar a correção do circuito existente e apresentam relatórios como resultado. A maioria das ferramentas de análise usam a conectividade como característica primordial dos circuitos.

   As ferramentas de análise dinâmica verificam o circuito funcionando ao longo do tempo. É o caso de simuladores lógicos ou elétricos.

   As ferramentas de análise estática examinam o circuito em função da entrada de dados e não se alteram com o tempo, como por exemplo a geração de padrões de teste. Pode-se classificar as análises estáticas em dois tipos:

   (a) verificadores de regras (geométricas ou elétricas, por exemplo): o circuito é analisado contra restrições colocadas pelo ambiente de projeto.

(b) verificadores de restrições colocadas pelo projetista para que o comportamento especificado concorde com o real. Dentro desse tipo se encaixam verificadores funcionais, que comparam descrições simbólicas da funcionalidade do circuito com o comportamento derivado das partes individuais do circuito.

É sabido que a análise computacional de CI's VLSI podem consumir muito tempo devido ao tamanho dos CI's. Para reduzir esse tempo é conveniente realizar análises enquanto o projeto vai sendo desenvolvido. Verificações realizadas incrementalmente podem economizar grandes quantidades de tempo [Rubin87].

A maioria das tarefas de CAD/CAE são caracterizadas por terem amplos espaços de busca, por isso elas têm resistido a enfoques algorítmicos tradicionais. O enfoque de sistemas especialistas baseados no conhecimento promete novas soluções [Birmingham86].

Por exemplo, para aumentar a flexibilidade dos compiladores de Silício de modo que eles mesmos selecionem automaticamente a arquitetura correta, a partir da especificação de entrada, tem sido usados Sistemas Especialistas.

Os Sistemas Especialistas usados são coleções de regras de alto nível que interagem para produzir resultados inteligentes. Em geral, as regras utilizadas por esses sistemas são regras do tipo **SE ... ENTÃO ....** que indicam uma ação para uma pré-condição.

O Sistema Especialista sempre verifica a lista de regras e executa as aplicáveis. Há dois tipos de regras: de controle e de projeto propriamente dito. Regras de controle dirigem a tarefa global e as de projeto são mais específicas e geram o resultado final.

Alguns Sistemas Especialistas realizam tarefas particulares. Eles tomam decisões e tem a habilidade de explicar como chegaram a essa solução. Há Sistemas Especialistas só para funções de controle, dirigindo outras ferramentas especialistas ou não.

A adoção de soluções através de sistemas CAD / CAE obedece a tendência já definida por [Birmingham86] como: o compromisso entre qualidade de solução, generalidade de enfoque e grau de intervenção humana, contra velocidade da solução. A velocidade de solução ganha espaço devido a justificar-se econômicamente, provocando em geral, menor qualidade de solução, inserção de restrições que particularizam as soluções, e aumento de intervenção humana.

3.2.3 Características de Sistemas Especialistas (S.E.)

A área de Sistemas Especialistas investiga métodos e técnicas para construir sistemas homem-máquina com habilidade ("expertise") de resolver problemas. Essa habilidade consiste de conhecimento sobre uma área particular ou **domínio**, entendimento dos problemas do domínio e capacidade para resolver alguns problemas [Hayes83].

O conhecimento pode ser classificado em dois tipos:

- Público: consiste de definições, fatos e teorias publicadas em livros e revistas do domínio de estudo.
- Privado: é o conhecimento do especialista humano, que não foi publicado e que consiste de regras de bom senso, que é também chamado heurística.

A heurística habilita o especialista humano a fazer opções quando necessárias, reconhecer enfoques promissores para resolução de problemas e lidar com dados incorretos (cheios de erros) e incompletos.

A tarefa principal do sistema especialista é elucidar e reproduzir tal conhecimento. Sistemas Especialistas são preferencialmente utilizáveis relativamente a técnicas de raciocínio formal ou sistemas tradicionais nos seguintes casos:

- para a classe de problemas que tem difíceis soluções algorítmicas.
- para reduzir custos de reprodução do conhecimento humano e permitir a sua aplicação, tornando público o conhecimento de especialistas humanos.

A área de Sistemas Especialistas é uma área de Inteligência Artificial que envolve paradigmas, ferramentas e estratégias de desenvolvimento de sistemas. Ela difere dos sistemas de processamento de dados tradicionais e de sistemas desenvolvidos em outras áreas de Inteligência Artificial.

Os sistemas de Inteligência Artificial se distinguem dos sistemas tradicionais por incluirem representação simbólica, inferência simbólica e busca heurística (ver definições no Apêndice A).

Os Sistemas Especialistas, denominados originalmente "Knowledge-based expert systems" (KBES), são os sistemas de Inteligência Artificial que apresentam as seguintes características:

- possibilitam aprendizado de novos fatos.
- tratam de forma independente os dados e o mecanismo de inferência.
- realizam tarefas difíceis com altos níveis de performance.
- enfatizam estratégias para resolver problemas de domínio específico.
- usam auto-conhecimento para raciocinar sobre seus próprios processos de inferência e provêm explicações ou justificativas para as conclusões a que chegaram.
- resolvem problemas das seguintes categorias: interpretação, predição, diagnóstico, depuração, projeto, planejamento, monitoração, reparo, instrução ou controle.
- não obedecem a uma ordem tão bem definida como no caso dos sistemas algorítmicos.

Uma das disciplinas de Sistemas Especialistas combina elementos metodológicos, tecnológicos e científicos, e é conhecida como Engenharia do Conhecimento. Um princípio dessa disciplina é que o conhecimento especialista raramente é adaptável a algum processo algorítmico rigoroso.

O conhecimento em um dado domínio consiste de descrições, relações e procedimentos a ele associados. É classificado como conhecimento de domínio específico, quando o conhecimento é diretamente relacionado com uma tarefa, por exemplo a implementação de projeto; quando o conhecimento é de um tipo mais geral é classificado como independente de domínio("domain-independent") [Kowalski85].

As descrições numa base de conhecimento que identificam e diferenciam objetos e classes são sentenças ou fatos em alguma linguagem. Uma base de conhecimento inclui regras e procedimentos para aplicar e interpretar descrições em aplicações específicas; também contém descrições conhecidas como relações, as quais expressam dependências e associações entre itens da base de conhecimento. A validade de uma regra da base de conhecimento é decorrente da frequência com que ela prediz resultados corretos. Os procedimentos por outro lado, especificam operações a serem realizadas quando os sistemas tentam raciocinar e resolver um problema. Estão associados com a efetividade de aplicação do conhecimento.

Os Sistemas Especialistas completos são compostos dos elementos que aparecem na figura 3.5, cujas funções são descritas a seguir:

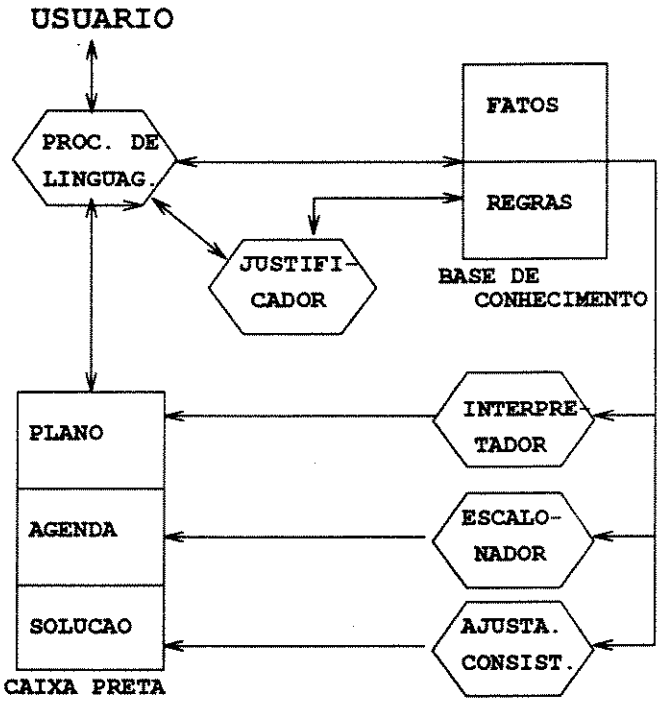

Figura 3.5: Sistema Especialista Ideal

- Processador de linguagem: executa comunicações orientadas ao problema, entre o usuário e o S.E.. A linguagem utilizada, textual ou gráfica, é interpretada para o S.E. e no sentido inverso a informação gerada pelo sistema é formatada para ser transferida ao usuário.
- Caixa preta [Hayes83] ou memória de trabalho [Kowalski85]: armazena resultados intermediários manipulados pelo S.E.. Nos sistemas ideais essa caixa preta é explícita e contém três tipos de elementos: de plano, relativos à forma de ataque ao problema; de agenda, contendo as ações potenciais que aguardam execução; e de solução, que representam as hipóteses candidatas e decisões geradas pelo sistema e as dependências relativas a essas decisões.
- Base de conhecimento [Hayes83] ou memória de regras [Kowalski85], compreendendo fatos, heurística e regras para resolver problemas. As regras da base de conhecimento têm interpretação procedural, mas os fatos têm papel passivo.
- Interpretador: aplica as regras, executando o item da agenda escolhido.
- Escalonador: controla a ordem de processamento das regras, mantém

controle da agenda e determina a sequência de ações pendentes a serem executadas.

- Ajustador de consistência: trata conclusões prévias quando novos dados ou conhecimento alteram as bases de suporte dessas conclusões.
- Justificador: racionaliza e explica o comportamento do sistema, como o sistema chegou a uma conclusão.

Um Sistema Especialista deve construir sua solução seletivamente e eficientemente a partir do espaço de alternativas; encontrar o dado útil o mais cedo possível, sugerir caminhos promissores e ajudar a evitar esforços de baixo retorno por eliminação de caminhos sem saída.

Para atingirem sucesso nos seus objetivos os sistemas especialistas têm como idéia básica encontrar respostas suficientemente boas com os recursos disponíveis e de forma eficiente. Por isso utilizam conhecimento de fatos, de heurísitca e de crendices.

A construção de S.E. está baseada num procedimento conhecido como **Aquisição de conhecimento** que consiste de extração do conhecimento de um especialista e da sua codificação em forma de programa. É uma técnica de desenvolvimento evolucionária e incremental que emergiu como metodologia dominante na área de S.E. [Hayes83].

O procedimento de Aquisição do conhecimento caracteriza-se por um conjunto de estágios, quais sejam:

(a) identificação das características do problema (definição do problema, conceitos relacionados e desenvolvimento de relações entre esses conceitos),

(b) conceitualização ou representação do conhecimento (fluxo de informações necessário para resolver problemas no domínio dado),

(c) formalização ou organização do conhecimento (estruturar a organização do conhecimento dentro das estruturas permitidas na linguagem escolhida),

(d) implementação de um protótipo executável e testável, e

(e) teste da implementação (para validação das regras, avaliação da performance e revisão do sistema).

Esses estágios não são totalmente independentes. Formalização e implementação

por exemplo, são bem relacionados, e falhas na geração de regras podem conduzir a reformulações. A fase de teste pode envolver reformular regras e processos de controle, reprojetar estruturas de conhecimento, redescobrir novos conceitos e redefinir o escopo e os objetivos do sistema.

Na fase de teste, ou no decorrer do desenvolvimento do S.E. pode ser necessário repensar as categorizações e representações inicialmente selecionadas para a base de conhecimento, em função da dificuldade de manejo atingido devido a suas características de tamanho e forma. Remodelamentos arquiteturais e reorganizações de conhecimento também são necessárias quando as capacidades adicionais desejadas para o S.E., excedem a capacidade do sistema existente.

A escolha de uma ferramenta para construir um S.E. é um ponto muito importante pois se reflete diretamente na facilidade do processo de desenvolvimento, no tempo de desenvolvimento e na eficiência do produto. A escolha da ferramenta envolve predominantemente o casamento das características do problema com as facilidades das ferramentas. A ferramenta escolhida deve ser capaz de lidar com a generalidade necessária para resolver o problema e deve passar por teste de um protótipo o mais cedo possível.

3.3 Ferramentas e o Projeto para Testabilidade

3.3.1 Sistemas de Síntese e a Testabilidade

Quanto à testabilidade, nem todas as ferramentas de síntese realizam tarefas que incorporam ou facilitam a sua implementação. Sendo uma preocupação que se deve ter desde o início do projeto, a incorporação de testabilidade terá influência na síntese e na otimização do projeto em questão. É uma consideração a ser feita hierarquicamente [DAC90], desde os mais altos níveis de projeto. Um dos maiores problemas de projetar em altos níveis de abstração é saber se o projeto vai funcionar quando implementado em nível estrutural. Algumas respostas podem ser obtidas, para uma implementação em particular no nível de porta, por simulação comportamental ou no nível de registros, mas outras somente através da síntese.

Em geral, pode-se considerar que existe um certo número de realimentações entre as várias etapas de síntese, como A, B e C, que aparecem na figura 3.6, sendo que os sistemas que implementam a síntese e a testabilidade de forma integrada têm melhores condições de realizar o projeto eficientemente.

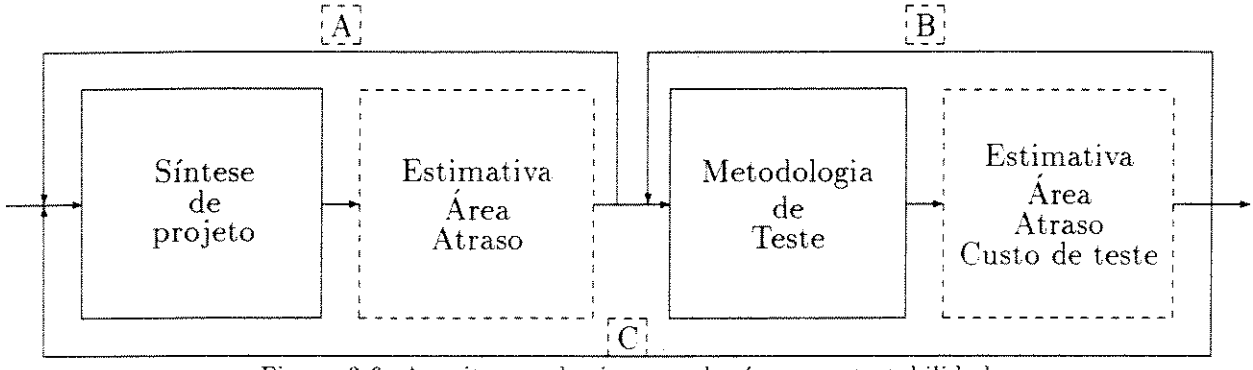

Figura 3.6: Arquitetura de sistemas de síntese + testabilidade

O projeto resultante do processo de síntese que inclui testabilidade, estará não só validado para suas especificações funcionais como também já terá resolvido nele o problema de teste associado. Isso é argumentado por autores de sistemas como o CATREE (Computer Aided TREE) [Gebotys88], que implementa testabilidade como parte da solução do projeto VLSI, e cuja metodologia de projeto é executada em estágios como mostra a figura 3.6. O CATREE implementa testabilidade avaliando custo de teste, área de Silício e atrasos e usando essas informações como realimentação para a síntese do projeto.

Para a tarefa de síntese, incluindo testabilidade, o CATREE utiliza uma estrutura de dados em árvore e algoritmos de árvore binária, que admite representação hierárquica e explora diferentes metodologias para cada circuito. Trata alocação de operadores na árvore, unidades funcionais, registros, interconexões e multiplexadores. Assume que unidades funcionais do circuito em projeto incluem "scan-path" e BIST. Para avaliações de custo de teste inclui medidas de cobertura de falha e tempo de teste para cada técnica utilizada. O processo de síntese sofre realimentação se as restrições de custo, área e atraso com o teste implementado não são satisfeitos.

O TITUS, "Testability Implementation and Test generation Using Scan" [Agrawal84] é um CAD que faz verificação algorítmica de um conjunto de regras para LSSD, implementa "scan" e gera testes, minimizando "hardware".

Entre os sistemas CAD disponíveis para projeto de CI's, existe preocupação quanto à testabilidade, mas a maioria deles não incorpora nem verifica testa-

bilidade. Alguns como o CAD da VTI dispõe de módulos especiais para tratar alguma metodologia de testabilidade. No caso [VTI], é disponível o Test Assistant que permite facilitar a observabilidade e controlabilidade de módulos do circuito, através de inserção automática de multiplexações.

Existem sistemas mais sofisticados como o GENESYL Designer System [Goering88], para projeto de ASIC's de alta performance e alta integração, baseado em compilador de Silício e que é ativado por arquiteturas, que inclui ferramentas que permitem realizar avaliações e realimentações rápidas, promovendo com isso o uso de DFT. No entanto, o enfoque das avaliações do GENESYL é baseado nos resultados do seu gerador de padrões de teste, e não em síntese ou verificação usando metodologia DFT.

A síntese de circuitos e particularmente, a inserção de características de projeto testável ou a verificação de regras de projeto para testabilidade, teve um grande impulso com o desenvolvimento de sistemas na área de inteligência artificial, que apresenta vantagens e que levou à ampliação do espaço de soluções para esses problemas.

Um sistema de síntese de projeto desenvolvido com Sistema Especialista baseado no conhecimento, o "Design Automation Assistant" (D A A) [Kowalski85], contém 7 tipos diferentes de regras, 3 de domínio específico (regras que estendem o projeto parcial, regras de controle de contexto e regras de manutenção de base de dados independente da tecnologia) e 4 do tipo independente de domínio (regras que removem elementos de memória de trabalho desnecessários, transformam a descrição de entrada, simulam operações de lista e conjunto, e simulam chamadas de procedimentos ou cálculos).

O D A A sintetiza uma representação independente da tecnologia, de memórias, registros, operadores, "data-paths" e uma decrição algorítmica de um sistema VLSI. Para isso, decompõe o problema em três representações de estruturas de dados e quatro tarefas de controle. Tem por restrições ou objetivos: projetos síncronos testáveis, a otimização de uso partilhado de hardware e a minimização da conectividade.

Outro sistema de síntese baseado no conhecimento é o ADAM synthesis system [Camurati88]. Combina interface de linguagem natural, base de dados para representação do projeto e uma coleção de bases de conhecimento. Contém um planejador baseado no conhecimento, que supervisiona procedimentos especializados e sistemas para síntese,análise e leiaute de CI's dedicados, baseado em regras. O conhecimento sobre o processo de projeto é representado como um conjunto de grafos interpenetrantes similar a redes semânticas (ver Glossário

- Apêndice A).

O ADFT "Automatic Design For Testability" é um sistema de projeto para testabilidade para o compilador de Silício SILC. Usa BIST e técnicas de "scan-path" para projetos VLSI. Segue uma abordagem "testável por construção" para sintetizar blocos lógicos. Usa um especialista de testabilidade, o TEST-PERT, que gerencia o conhecimento de testabilidade durante o processo de síntese, indica elementos do projeto mais difíceis de testar e assegura as interconexões necessárias a um barramento de teste, o OCTEBUS [Fung86].

Com o objetivo específico de projeto de CI's VLSI auto testáveis, foi desenvolvido pela Univeridade de Rochester o CAST ("Computer-Aided Self-Test") [Kalinowski88], que admite como entrada um grafo de circuito no nível de registros e um conjunto de requisitos de teste. O CAST incrementa o projeto de circuitos com características que o fazem auto-testável, minimizando o "hardware" necessário para tal. Determina no grafo onde os módulos BIST devem ser alocados, considerando dois critérios: o de mínimo "overhead" ou o de teste exaustivo. Liga todos os segmentos auto-testáveis para formar "scan-path" e provê sinais de controle de teste aos segmentos, com mínimo custo de conexão. Produz o arranjo de todos os testes e a distribuição das linhas de controle de teste aos módulos BIST.

O TDES ("Testable Design Expert System") [Abadir85], que aceita entrada de descrições estruturais de circuito no nível de registros e insere estruturas de testabilidade, usa uma estrutura que incorpora aspectos estrutural, comportamental, quantitativo e qualitativo das técnicas DFT conhecidas. Usa uma estrutura hierárquica de projeto baseada em grafos, faz particionamento e insere estruturas "hardware" necessárias segundo a sua base de conhecimento para enriquecer a testabilidade do circuito

Trabalhos também foram realizados para investigar inserção automática de circuitos BIST usando VHDL [Kim88]. Enfoques algorítmicos e baseados em regra foram usados objetivando a inserção de BILBO's, tais como geradores de sequência pseudo-aleatória ou analisadores de assinatura.

Algumas ferramentas para projeto de CI's testáveis são mais específicas, devido às próprias características e tendências dos projetos de CI's digitais. Por exemplo, a substituição de circuitos de lógica aleatória por PLA's, aumenta a importância de teste de PLA. Apesar das muitas vantagens oferecidas pelo uso de PLA's, tem se apresentado novos problemas de teste. Métodos tradicionais de teste, como o Algoritmo-D e outros [Côrtes91] e teste aleatório não tem sido efetivos para PLA's porque elas tem alto "fan-in", "fan-out", redundância e

requerem modelos de falha específicos.

Um grande número de métodos de projeto para testabilidade, de geração de teste e de teste foram desenvolvidos para PLA's, com diferentes propriedades, graus de testabilidade e "overhead". Para ajudar os projetistas na escolha de um método de testabilidade adequado para PLA's existem sistemas tais como o PLAESS [Breuer85] e o PLATA [Yousuf88], que foram desenvolvidos com essa finalidade específica. Ambos são sistemas baseados no conhecimento, que permitem avaliar o uso de alternativas de projeto para PLA's com diferentes técnicas DFT específicas para esse estilo de projeto.

O objetivo das ferramentas em geral é compartilhar o máximo dos recursos e obter a melhor testabilidade do circuito, considerando cobertura de teste, custo de geração de padrão e restrições de projeto.

3.3.2 Sistemas de Análise e a Testabilidade

As ferramentas que não incorporam testabilidade mas verificam a implementação do projeto quanto a características de testabilidade podem ser analisadores de testabilidade ou auditores de testabilidade.

1. **Analisadores de Testabilidade**

   Os analisadores de testabilidade foram desenvolvidos com o objetivo de promover soluções para o problema de teste de CI's. identificando pontos do projeto difíceis de observar ou controlar através de medidas de testabilidade, que não dependem de ATPG ou simulação de falhas.

   Duas dessas ferramentas são o SCOAP [McCluskey86] e o CAMELOT ou HI-TAP. Ambas utilizam métodos de cálculo para associar valores de controlabilidade e observabilidade aos nós do circuito. com o objetivo de prever o custo de gerar padrões.

   O SCOAP (Sandia Controlability / Observability Analysis Program) usa um método que se baseia na topologia do circuito, sem referir-se a um conjunto específico de vetores de teste ou método de geração de teste. Ele calcula 6 funções para caracterizar as propriedades de controlabilidade e observabilidade do circuito: a controlabilidade combinacional e sequencial para 0 e para 1 (CC0 e CC1, CS0 e CS1). e a observabilidade combinacional e sequencial para 0 e para 1 (CO e SO).

   Os valores de controlabilidade combinacional (CC) representam o mínimo número de nós que devem ser posicionados para produzir um nível 0 ou

1 no nó em questão.

Os valores de observabilidade combinacional representam ambos: o número de células-padrão combinacionais entre o nó e uma saída primária, e o número mínimo de assinalamento de nós combinacionais requerido para propagar o valor do nó até a saída primária.

Os valores de observabilidade sequencial representam: o número de células-padrão sequenciais entre o nó e uma saída primária e o número mínimo de células-padrão sequenciais que devem ser controladas para propagar o valor do nó até a saída primária.

Os valores combinacionais dão uma estimativa do custo de geração de teste no sentido espacial, isto é a proporção do circuito que deve ser instantâneamente controlada para testar o nó.

Os valores sequenciais são análogos, porém se referem só aos nós sequenciais do circuito. Então esses valores dão uma estimativa de custo da geração de teste no sentido temporal. As dificuldades de teste podem ser isoladas para características temporal e espacial, identificando onde é preciso aumentar a testabilidade.

Os nós de um circuito são mais difíceis de testar, quanto mais altos os valores de controlabilidade / observabilidade calculados pelo SCOAP. O SCOAP é bom para detectar dificuldades de inicialização, mas em geral é aplicado muito tarde no projeto para ser efetivo.

O HI-TAP ou CAMELOT (Computer Aided Measure for Logic Testability) associa aos nós valores de controlabilidade que variam entre 0 e 1. O máximo valor representa um nó tal como uma entrada primária, para o qual é facil associar um nível lógico 1 ou 0. O outro extremo é um nó que é um ponto flutuante, incontrolável. A observabilidade é análogamente definida, sendo o valor máximo associado às saídas primárias.

O método de cálculos do CAMELOT associa valores de controlabilidade aos nós internos a partir das entradas, usando um fator de tranferência de controlabilidade. Para uma porta esse fator é muito próximo ou muito relacionado com o número de zeros e uns da saída de sua tabela da verdade.

Os valores de Observabilidade são calculados para os nós do circuito a partir das saídas primárias, usando um fator de transferência de observabilidade [Baker]. O CAMELOT faz uso de valores de controlabilidade para determinar observabilidade de nós. Isso é feito para considerar a necessidade de colocar valores nos nós internos para sensitizar um caminho até a saída [McCluskey86].

O CAMELOT é considerado menos elaborado que o SCOAP, apresentando maiores dificuldades para distinguir entre controlabilidade para 0

ou para 1 e ignorando os conceitos de testabilidade combinacional e sequencial.

Há dúvidas quanto à validade do uso dos analisadores existentes, devido à sua imprecisão e ao fato de não garantirem a geração automática de teste; mas há aplicações consideradas plausíveis, como por exemplo para projeto de "gate arrays" pequenos, ou placas de circuito eletrônico, onde a testabilidade pode ser um problema mas o uso de técnicas estruturadas de projeto seja impraticável.

2. **Auditores de Testabilidade**

As ferramentas que, exclusivamente, verificam os projetos quanto a regras e metodologias específicas de projeto para testabilidade são os auditores de testabilidade. Entre eles encontram-se ferramentas que:

(a) utilizam técnica similar à simulação lógica. No caso da referência [Godoy77], foi adaptado um simulador lógico, adotando o seu procedimento de "tracing", e substituindo as rotinas de cálculo de respostas a estímulos por modelos comportamentais que permitem a verificação de regras. Usa procedimentos algorítmicos e provê modelos comportamentais para portas lógicas primitivas (AND, OR, NAND e NOR) e para "latches". A rotina de escalonamento da simulação ativa os modelos comportamentais que resultam respostas que variam algoritmicamente, para verificação das regras.

A maioria das regras implementadas se aplicam a configurações e controle de caminhos do circuito, como o "scan" entre "latches" e sinais de relógio. Segundo [Agrawal84] esta ferramenta não pode ser usada eficientemente para estruturas de circuito descritas no nível de porta MOS.

(b) usam técnicas de inteligência artificial ou sistemas baseados em regras, usam linguagem PROLOG como os sistemas referidos por [Horstmann84], [Cosgrove88] e [Bidjan-Irani91], sendo que este último trata descrições de circuito em qualquer linguagem, no nível de registros e em níveis mais baixos. A linguagem e o interpretador Prolog para regras baseadas em LSSD é defendida por [Cabodi86].

(c) muitas ferramentas verificam o projeto quanto a regras das disciplinas de projeto com "scan" definidas em [Eichelberger77] como o sistema da referência [Adhem87]; geram teste e os compactam e avaliam falhas, e identificam falhas de stuck-at não testáveis como o TESTSCAN [gat]; outras usam base de dados contendo informações sobre técnicas de projeto BIST [Jones85].

(d) algumas das ferramentas desenvolvidas para resolver o problema de projetos de CI's são de amplo espectro de aplicação, como o "NCR Design Advisor" [Richardson88], que é composto de vários módulos de auditoria, os quais interagem com o projetista propondo soluções quanto a

análises de "timing", de distribuição de relógios, de velocidade, de interface e de testabilidade. A base de conhecimento do Módulo de Testabilidade contém conhecimento sobre controlabilidade de entradas assíncronas ("preset" e "reset").

(e) utilizam grafos para mapear o circuito combinacional como o AUDIT [Brglez88], componente do OASIS ("Open Architecture Silicon Implementation System") [Brglez89], escrito em "C", para UNIX. O objetivo do AUDIT é verificar adequação e conformidade a regras de projeto com "scan".

(f) Sistemas que utilizam linguagem PROLOG como o ESTA ("Expert Sistem for DFT rule verification") [Camurati88], e que utiliza o interpretador PROLOG como suporte computacional. O ESTA verifica circuitos quanto a LSSD e a BILBO, explorando a hierarquia e utilizando-se de regras e estruturas ("frames") na sua base de conhecimento. Para verificação de LSSD classifica os circuitos em combinacionais e sequenciais e identifica explicitamente entradas primárias de relógio, entradas e saídas de "scan". Para verificação de BILBO's representa os registradores de deslocamento por um nome, especifica sua lista de entradas e saídas de registros, sinais de controle, entradas e saídas de "scan" e seu relógio.

3.4 Projeto para Testabilidade: perspectivas do ambiente

Grandes mudanças estão ocorrendo em termos de sistemas automáticos para síntese de circuitos integrados ou até mesmo de placa, como o aparecimento de ambientes de projeto e conjuntos de ferramentas que integram hardware, software, projeto mecânico e fabricação, como noticiado por [eet90].

São plataformas ("framework") software que devem implementar um ambiente de projeto concorrente ou paralelo, com troca de dados entre aplicações altamente interativa; construídas sobre um sofisticado sistema de gerência de base de dados e feitas para atender o ciclo de projeto completo, acomodando modelos de dados de "netlist", projeto físico, software, texto e 3D(para CAD mecânico).

Ocorre, em contrapartida, que quando a faixa de ferramentas é muito ampla, não necessariamente é acompanhada da mais alta performance. Além disso é muito difícil ser completo, já que o número de refinamentos e capacitações especiais para projeto é muito amplo. Projeto incremental e projeto em alto

nível ainda são muito falhos.

Há, portanto, espaços a serem preenchidos mesmo dentro dos ambientes mais sofisticados de CAE para síntese, e nos mais diversos níveis, porém deve-se ter em mente que há uma grande preocupação com a utilização de uma linguagem padrão (VHDL) [Shadad] e com a integração com ferramentas disponíveis do ambiente de projeto.

Essa preocupação decorre do fato de se objetivar o uso eficiente de ferramentas desenvolvidas para preencher os espaços mencionados. de modo a contribuir para diminuir o ciclo de desenvolvimento dos projetos. A competitividade é influenciada pela duração do ciclo de desenvolvimento o qual é dependente da identificação e solução de problemas da forma mais eficiente possível.

O aprimoramento no projeto de CI's digitais é decorrente dos dois tipos de ferramenta de apoio: de síntese e de análise. Sabe-se que é possível economizar tempo de desenvolvimento com o uso de ferramentas de análise, conforme defendido por [Rubin87]. Resta identificar e resolver problemas chave do ciclo de projeto.

No caso de ferramentas de projeto e auditoria de testabilidade, por exemplo o uso de Sistemas Especialistas é interessante pela própria característica de evolução gradual dos S.E., que se baseiam em experimentação para atingir alta performance. A heurística associada às tarefas de projeto e verificação pode ser incrementada aos poucos e evoluir de sistemas simples a sistemas mais completos.

Os S.E. apresentam como importante produto a codificação do conhecimento e também têm a característica potencial do auto-conhecimento. Por meio dessa característica, considerada a mais inovativa e significativa relativamente aos programas convencionais, é esperado que se consiga reconstrução adequada de linhas racionais de argumentos, a partir de princípios fundamentais do domínio. O auto-conhecimento deve prover bases para automodificação incluindo correção de regras, reorganização da base de conhecimento e reconfiguração de sistema, o que é muito interessante para resolver problemas, em particular problemas de projeto.

Capítulo 4

SMAuT : um auditor de testabilidade

4.1 Introdução

4.1.1 Razões da automatização da auditoria de testabilidade

Durante o desenvolvimento de projeto de CI's digitais, verifica-se que a auditoria de testabilidade realizada por um engenheiro de testabilidade, nem sempre é muito eficaz (Ver Apêndice B : Estudo de Casos de Violações de Regras de Projeto para Testabilidade), pois há dificuldades em controlar vários projetos em desenvolvimento paralelamente; há necessidade de um vasto conhecimento das técnicas existentes e a verificação dos problemas de testabilidade nos projetos é muitas vezes um gargalo para o ciclo de desenvolvimento.

A solução normalmente adotada para as falhas no projeto para testabilidade é apelar para o desenvolvimento de padrões funcionais, o que só é possível com a ajuda do projetista, exigindo bastante tempo, podendo gerar padrões muito longos, altamente comprometidos com o funcionamento de muitas áreas do chip e exigindo ainda, conhecimento do equipamento de teste a ser utilizado.

Como alternativa de solução é razoável propor a automatização do controle da implementação de metodologias de testabilidade aos projetos, para verificação da consistência dos projetos com relação à testabilidade. Um sistema para auditoria de testabilidade deve ser realizado de modo gradativo, permitindo expandir a abrangência das verificações sem interferir no seu funciona-

mento. Optou-se pelo desenvolvimento de um sistema modular que se denominou **SMAuT**, Sistema Modular de Auditoria de Testabilidade; modular pela sua arquitetura, bem como pela característica de aumentar o conhecimento e a abrangência do sistema, independentemente das verificações já implementadas.

4.1.2 Finalidade do auditor

A finalidade do auditor de testabilidade aqui especificado é controlar a implementação de metodologias de testabilidade, durante o desenvolvimento de projetos de CI's digitais, garantindo a satisfação de regras compatíveis com metodologias estruturadas de projeto, conhecidas como "Scan Design", e permitindo verificar a consistência dos testes previstos no projeto.

Este auditor destina-se à verificação de projetos de CI's digitais baseados em células ou também chamados semidedicados. Pode-se aplicar este auditor também para CI's dedicados, descritos no seu nível de portas lógicas, pois as verificações são feitas no nível lógico, não sendo relevantes as demais características. Por outro lado, este auditor não se preocupa com a estrutura interna de blocos grandes e regulares de circuito, supondo que já tenham sido adotadas técnicas de DFT adequadas para o seu projeto e que elas apresentem um modo de teste definido e verificado.

O auditor de testabilidade aqui especificado não tem características de inserção de estruturas de testabilidade no projeto e não verifica problemas detectáveis por simulação lógica.

4.1.3 Aplicação

A aplicação do auditor de testabilidade deve ocorrer desde o início do projeto lógico de cada bloco do circuito em desenvolvimento, antes mesmo de sua validação por simulação lógica. A aplicação do auditor paralelamente ao desenvolvimento deve contribuir para a minimização do ciclo de desenvolvimento. Se ele só for aplicado no final do projeto pode obrigar modificações, que consequentemente implicarão em ressimulações e portanto em maior duração de ciclo de desenvolvimento. O auditor de testabilidade também deve ser aplicado sobre a descrição lógica final do circuito completo, até mesmo aquela obtida a partir de extração do leiaute, para garantir consistência de interligações do circuito, que satisfazem as regras de testabilidade verificadas.

4.1.4 Organização desta especificação

Esta especificação do **Sistema Modular de Auditoria de Testabilidade - SMAuT**, contém uma descrição das características gerais do sistema planejado na seção 4.2, seguida da descrição geral do sistema protótipo na subseção 4.3.1 e da descrição das funções implementadas no protótipo do SMAuT na subseção 4.3.3. São descritos os casos práticos submetidos ao protótipo e as consequentes considerações relativas às avaliações dos resultados obtidos na subseção 4.3.4. Ao final são propostas tarefas complementares ao protótipo, que podem ser exploradas e incorporadas ao sistema na seção 4.4.

4.2 Especificação de Características do SMAuT

4.2.1 Funções especificadas para o SMAuT

A tarefa do sistema de auditoria de testabilidade de projetos de CI's digitais aqui especificado consiste em analisar o circuito descrito em termos de suas interconexões, segundo as regras de testabilidade embutidas na base de conhecimento, apresentadas na subseção 4.2.5, gerando como saídas: a identificação das possíveis violações de regras de projeto e as estruturas válidas de testes implementadas no circuito em questão.

O sistema de auditoria especificado é um sistema de análise estática, que aplica regras a descrições topológicas de circuitos. As descrições dos circuitos podem ser hierárquicas ou não, atingindo o nível de portas lógicas como o mais baixo nível de refinamento das descrições. Isto implica que estruturas tais como memórias ou PLA's são descritas como um módulo do circuito ou um subcircuito fechado com entradas e saídas definidas. Esses grandes blocos de circuito devem ter suas entradas e saídas analisadas relativamente aos pinos do circuito integrado, para garantir a controlabilidade de suas entradas e a observabilidade de suas saídas, sem conflitos entre níveis lógicos necessários aos sinais para seu controle e observação.

A arquitetura do Sistema Modular de Auditoria de Testabilidade - SMAuT, aqui especificado é ilustrada na figura 4.1. Para acesso ao sistema são previstos dois tipos de interação:

- Operação normal do auditor por usuário comum, através de entrada da

descrição do circuito para realização da auditoria de testabilidade, que é feita por interação com o Módulo de Interface com o usuário.

- Manutenção do sistema para acréscimo de regras à base de conhecimento ou ao próprio mecanismo de inferência[1] do sistema. Esse tipo de operação deve ser independente da operação normal do sistema e das verificações já implementadas, como um desenvolvimento paralelo e pode eventualmente acrescentar interações com o usuário. Nesse caso o Módulo de Interface com o usuário também sofrerá atualização paralela e independente analogamente à atualização da base de conhecimento e do mecanismo de inferência.

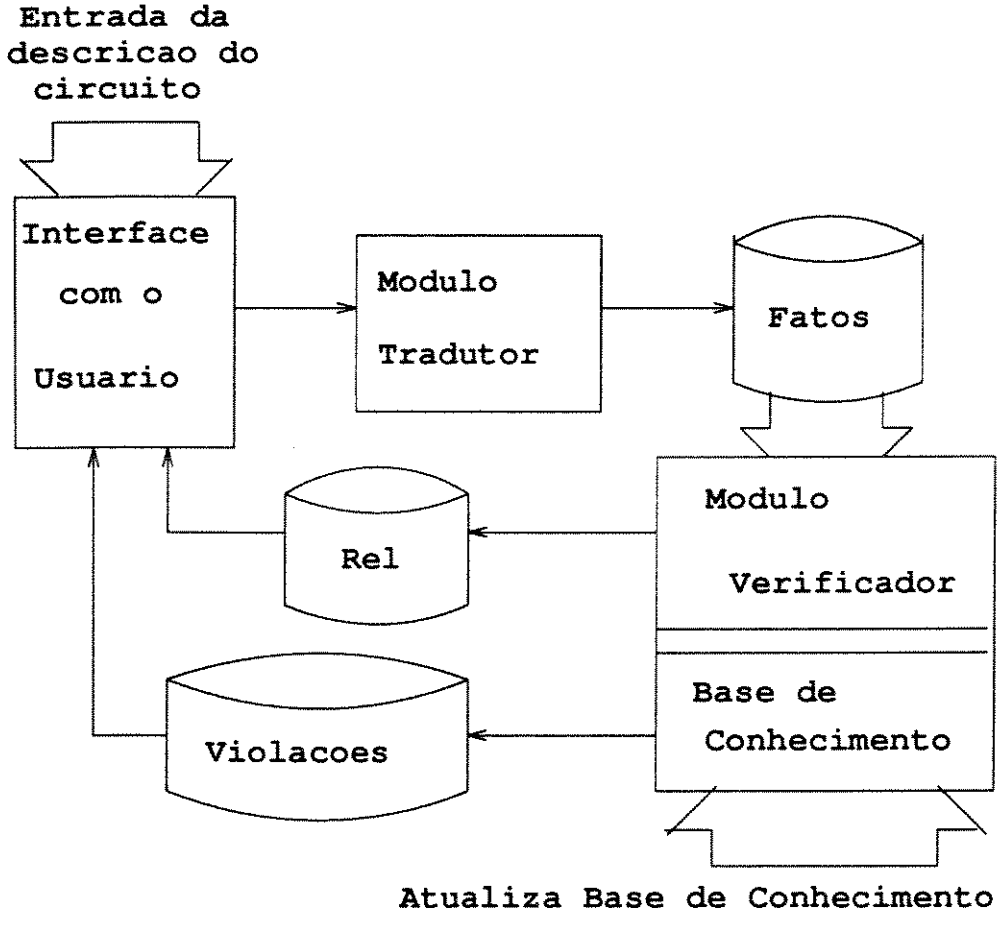

Figura 4.1: Arquitetura do SMAuT

[1] Ver definições no Apêndice A

4.2.2 Descrição da entrada

A entrada de dados do sistema é a descrição do circuito no nível de portas lógicas e blocos de circuito, dada pelo "netlist", normalmente utilizado para simulação lógica, acrescentado de algumas informações adicionais, e consistindo de:

1. uma descrição hierárquica do circuito, do tipo estrutural, consistente como as descrições usadas para simulação lógica, isto é, cada circuito é descrito pela interconexão de células e módulos e todas as conexões estão associadas com origem (saída) e destinos (entradas) de células instanciadas[2]. Os módulos ou subcircuitos grandes tais como memórias, PLA's e outros, fazem parte da descrição do circuito como caixas pretas, com entradas e saídas definidas.

2. identificação de pinos de modo teste, nomes e níveis respectivos; identificação de pino especial para modo teste de grandes blocos discriminando pinos envolvidos, uma vez que o modo teste também pode ser identificado por uma combinação não esperada de pinos em operação normal.

3. identificação de barramentos e sinais tri-state.

4. identificação opcional de pinos de relógio. Se os pinos de relógio são identificados podem ser verificadas regras associadas com relógios relativamente a esses pinos; caso contrário esses tipos de regras são verificadas a partir dos sinais das entradas de relógio das células sequenciais.

5. lista de pinos de entrada, de saída e de pinos bidirecionais para cada nível hierárquico.

6. especificações de restrições particulares do projeto em questão, por exemplo, pinos de controle de diferentes modos de operação normal do circuito.

4.2.3 Descrição da saída

A saída do sistema consta de mensagens quanto ao status do circuito corrente, as quais podem ser apresentadas interativamente ao usuário e também compor um arquivo de saída, bem como da discriminação dos arquivos gerados pelo sistema, quais sejam:

[2]Uma instância de uma célula é a associação unívoca por um nome a um dado tipo de célula lógica da biblioteca de células utilizada para o projeto.

- **Violações** é o arquivo de violações de regras de testabilidade, onde comparecem as falhas detectadas no circuito quanto às regras mencionadas na subseção 4.2.5.

- **Rel** é o arquivo de testes identificados no circuito e que satisfazem as regras de testabilidade embutidas no mecanismo de inferência do Módulo Verificador. Nesse arquivo são listados:

  - os "scans" implementados, isto é, a lista das instâncias dos elementos de memória nele contidos; lista das instâncias das células combinacionais (multiplex e buffers) que viabilizam a sua observabilidade: os pinos de entrada e saída associados e suas respectivas funções ("scan in", "scan out", sinais de controle de reset ou preset e de habilitadores de entradas de multiplex ou células tri-state)
  - lógica combinacional para a qual é necessária a geração de padrões de teste, associadas a suas entradas e saídas, e às condições e aos "scans" pelos quais esses nós de entrada da lógica combinacional podem ser controladas e os de saída podem ser observados.
  - casos de habilitação e desabilitação de módulos grandes do circuito (através de barramentos) que se verificaram implementadas segundo as regras de testabilidade embutidas na base de conhecimento e os respectivos pinos de entrada e saída afetados.

4.2.4 Descrição da Arquitetura do Sistema

O **SMAuT** é dividido em tres módulos básicos de programa e permite verificar um projeto de CI digital descrito hierarquicamente em linguagem de descrição hardware, com base em regras de projeto para testabilidade, contidas em sua base de conhecimento. A descrição do circuito é traduzida para um conjunto de fatos[3] e estruturas ("frames") que podem ser verificadas pelo Módulo Verificador.

A verificação da testabilidade deve ser hierárquica, permitindo que cada subcircuito seja verificado uma única vez e não uma vez por instância que ele ocorre no circuito total, para que se consiga um uso eficiente de recursos e tempo de máquina. O SMAuT deve poder aceitar definições de exceção do projetista durante a execução do programa, ao encontrar alguma violação de regra no projeto em questão. O mecanismo de inferência do SMAuT não requer simulação lógica, como no caso do sistema da referência [Godoy77], pois

[3] Ver definições no Apêndice A.

pode identificar a violação de regras de projeto, analisando a estrutura do circuito, mais que a sua função, como o sistema referido por [Agrawal84]. Por simplificar a análise, melhores características de velocidade são esperadas para o sistema aqui especificado .

Os módulos básicos de programa do SMAuT são:

1. **O Módulo de Interface com o Usuário** que estabelece a comunicação do usuário do auditor de testabilidade com o SMAuT propriamente dito. Solicita informações de entrada : arquivo de entrada do circuito, tipo de regras a serem verificadas, por exemplo identificando se o circuito prevê circuitos com relógios duplos e não sobrepostos ou não. Fornece informações relativas ao fim da execução do programa, ou à sua interrupção, bem como aos arquivos de saída gerados. Quando ocorrer interrupção na operação do sistema devido a infração de regra que admita exceção, esse módulo deve obter do usuário uma senha para prosseguir o procedimnto de auditoria, considerando a exceção, bem como para permitir que essa exceção seja considerada num próximo processamento do mesmo circuito.

   Esse módulo gera o **Arquivos de Fatos do Circuito** que contém todos os fatos relativos ao circuito em análise, em formato compatível para tratamento pelo Módulo Verificador de Regras de projeto para Testabilidade.

   O Módulo de Interface tem ainda a função de ativar um sub sistema de explicação contido no mecanismo do Módulo Verificador para justificar qualquer violação de falha detectada. Essa explicação está relacionada ao papel do Justificador da descrição do Sistema Especialista Ideal(ver figura 3.5).

2. **O Módulo Tradutor** do circuito é o que faz a preparação dos dados do arquivo de descrição hardware de entrada. Trata-se de um conversor de formato da descrição disponível do circuito para uma descrição compatível com a linguagem conhecida pelo Módulo Verificador. São funções desse módulo:

   - gerar os fatos, isto é, a associação de cada elemento (instância) da descrição hardware do circuito a uma cláusula que o identifica como célula conhecida ou não, em linguagem compreensível pelo Módulo Verificador. O sistema supõe um conjunto de **células conhecidas** e considera tudo o que não estiver nesse conjunto como **desconhecido**. Isso será usado pelo Módulo Verificador.
   - traduzir a hierarquia para uma estrutura representativa dos módulos e suas interconexões, que será visitada segundo uma estratégia de

controle independente de domínio. Para implementação utilizando a linguagem e o interpretador PROLOG [Bowen86], é possível ter acesso individual às descrições hardware traduzidas para cada bloco. O Módulo Verificador então terá como aplicar as regras a cada um deles de cada vez, e realizar a sua sequência de ações com economia de ações de busca.

3. **Módulo Verificador** de regras de projeto para testabilidade. É aquele que estabelece uma sequência de ações, conhecida como mecanismo de inferência, para análise dos fatos contidos no arquivo de fatos do circuito, considerada a base de conhecimento de testabilidade associada.

   A verificação de regras de projeto para testabilidade é feita sobre o circuito descrito topologicamente, mantendo-se a sua característica hierárquica. Para tanto, deve executar dois níveis de ação: um deles é o de controle de visita aos nós da estrutura associada ao circuito e o outro é o de sequência das regras de testabilidade aplicada aos nós.

   A visita à estrutura associada ao circuito, gerada pelo Módulo Tradutor é ilustrada pela figura 4.2. O circuito é associado à estrutura de uma árvore onde o nó A é a raiz, ou seja o mais alto nível hierárquico de descrição do circuito. Os nós que não têm descendentes são nós-folha e os demais são nós comuns.

   A estratégia de controle que rege as visitas à estrutura é dada por:

   - Localização de um nó-folha da estrutura, a partir da raiz.
   - Aplicação de conhecimento de DFT para o nó-folha identificado.
   - Abstração da descrição da interface do nó-folha com o restante do circuito, identificando estruturas de teste definidas e verificadas pela aplicação do conhecimento de DFT, tais como entradas e saídas virtuais de "scan", além de níveis associados a cada "scan" identificado, e a lógica combinacional extraída a partir da descrição desse nó.
   - Verificação: **se** todos os nós-folha descendentes de um nó comum encontrado no caminho de volta da visita já foram analisados e todas as suas interfaces são disponíveis, **então** são aplicados conhecimentos de DFT ao nó comum descrito a partir das interfaces.**Senão** seguir o procedimento a partir do terceiro item desta estratégia, para o nó-folha descendente ainda não analisado.
   - **Se** todos os nós da estrutura já foram visitados e o nó raiz foi analisado **então** o processo de aplicação de conhecimento de DFT está concluído e portanto o circuito completo sofreu a auditoria de testabilidade desejada.

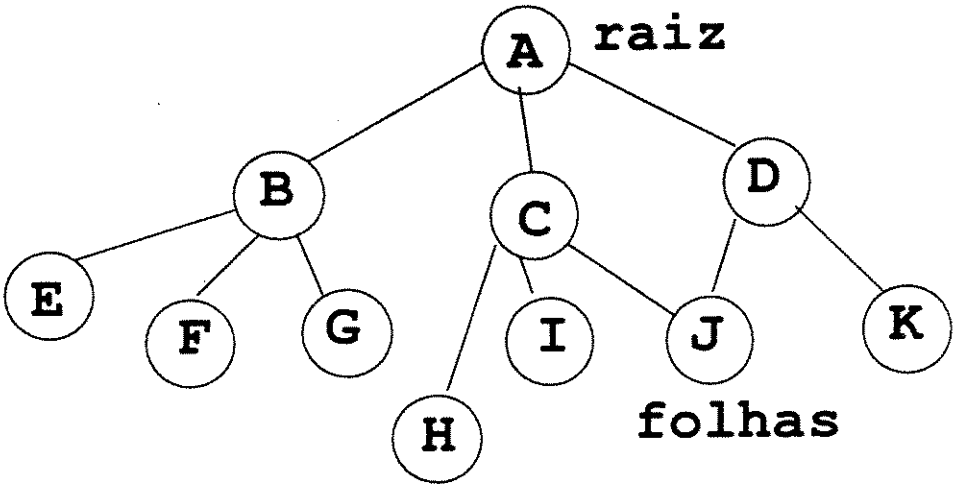

Figura 4.2: Estrutura associada à descrição hierárquica

Essa visita pode ser feita como um procedimento de busca do tipo "depth-first"(ver definição no Apêndice A) ou outro mais eficiente.

Esse módulo gera dois tipos de arquivos acessíveis ao usuário, discriminados na subseção 4.2.3. O Módulo Verificador está fortemente ligado à chamada Base de Conhecimento, uma vez que o seu mecanismo de inferência verifica regras por ela suportadas.

Para realizar com eficiência a tarefa de verificação das regras esse módulo deve considerar mecanismos de raciocínio do tipo "reasoning -backward chaining" e "reasoning-forward chaining"(definições no Apêndice A). Em função da complexidade das deduções causada pelo tamanho ou número de interconexões de um dado circuito é possível que ocorram contradições entre fatos gerados na verificação. Em vista disso, o sistema de auditoria deve contar com um mecanismo de resolução de contradições, bem como se apoiar num subsistema de manutenção da verdade.

Para verificar descrições hierárquicas em PROLOG deve associar cláusulas do tipo "frame" como em [Camurati88]. Como facilidade para chegar a conclusões mais rapidamente poderá utilizar de procedimentos "demon" e assumir valores "default" para campos (ver definições no Apêndice A) relacionados com os "frames" usados para descrever os circuitos. Pode-se aproveitar descrições contidas em [Camurati88] que explicita esse tipo de implementação.

4.2.5 Conhecimento de DFT considerado

A **Base de Conhecimento** consiste das informações necessárias para a execução da análise do circuito com vistas à sua testabilidade. Essas informações podem ser classificadas em:

- Conhecimento de células dos projetos a serem verificados, em termos de tipo, entradas, saídas e polaridade de nível de ativação de dadas funções como por exemplo: reset ativo baixo associado com um código N para um campo específico de descrição da célula flip flop na base de conhecimento.

- Conhecimento para inferência de resultados intermediários ou finais pretendidos por este auditor. São regras que aplicadas ao arquivo de fatos associado ao circuito durante a sequência de ações do verificador produzem fatos intermediários ou finais. Funciona como se o conhecimento para o circuito em questão fosse atualizado à medida em que condições da sequência de ações do verificador vão sendo satisfeitas. A própria sequência de ações compõe o mecanismo de inferência específico do SMAuT e está dentro desta classe de conhecimento da base.

- Conhecimento especialista de Projeto para Testabilidade, adequado às diversas estruturas e / ou metodologias de projeto. No conjunto de conhecimentos especializados de Projeto para Testabilidade aparecem regras específicas para cada metodologia. Trata-se de um espectro muito amplo de conhecimento, cuja programação pode ser implementada gradualmente. Como primeira fase implementou-se a verificação das regras de testabilidade apresentadas em seguida, normalmente satisfeitas por projeto para sistemas com "Scan" usando células de relógio único e/ou de duas fases de relógio sem superposição. Posteriormente poderão ser adicionadas regras relativas à verificação de acesso a grandes blocos do circuito (RAM's, PLA's e outros) para verificação de sua controlabilidade e observabilidade, ou ainda, relativas à metodologia de BIST, com a verificação de BILBO's por exemplo.

De modo geral e considerando-se o projeto de um CI qualquer como um todo, sugere-se a adoção de "scan-path" para a parte sequencial, com sinais de relógio controláveis, isolando-se a parte puramente combinacional. Associado a essa recomendação há um conjunto específico de regras de projeto mais comumente utilizado, de considerável flexibilidade para projeto e que devem resultar num projeto com implementação de "scan" apropriada, sem problemas de corrida entre sinais ("hazard and race free"), definidos em 2.4.2 e 1 da subseção 2.6.1).

De forma simplificada são apresentadas a seguir as regras adotadas na implementação do protótipo do SMAUT:

- REGRA 1: Estabilidade dos sinais

  Todos os elementos de memória devem ser implementados em flip flops mestre-escravo do tipo D.

- REGRA 2: Sinal de modo teste

  Um (ou mais do que um) pino de entrada primária deve ser alocado para especificar os modos ("scan" ou normal) de operação do circuito.

- REGRA 3: Controlabilidade dos sinais de relógio

  As entradas de relógio dos flip flops devem ser controláveis a partir de entradas primárias e não de lógica ligada às saídas de dados de flip flops ou de outras entradas primárias.

  Além disso, as entradas de relógio de um flip flop não devem estar ligadas às saídas de dados de outro flip flop, nem diretamente, nem através de lógica combinacional. E ainda, nenhum sinal de relógio está ligado à entrada de dados de células sequenciais, nem diretamente, nem através de lógica combinacional.

- REGRA 4: Limitação de entradas assíncronas

  Os flip flops não devem ter mais do que uma entrada assíncrona ("clear" ou "preset"), a qual deve ser controlável por entrada primária.

- REGRA 5: Serialização de elementos sequenciais

  Todos os flip flops são conectados em "scan" (registrador de deslocamento). Preferencialmente, todas as mudanças de sinais ocorrem com uma mesma fase de relógio em modo teste.

- REGRA 6: Controlabilidade dos "scans"

  Cada "scan" tem uma entrada primária e uma saída primária no modo scan. Para isso, podem ser multiplexadas entradas e saídas primárias.

- REGRA 7: Garantia de verificação de flip flops

  No modo "scan" as saídas de cada flip flop assim como a saída primária de "scan" são função da saída do flip flop precedente ou da entrada primária do "scan".

A satisfação da REGRA 3 e da 4 objetiva evitar circuitos assíncronos.

Além dessas regras outras podem ser geradas, como as abaixo definidas:

- REGRA 8 Controlabilidade e Observabilidade de blocos combinacionais

  As entradas e saídas de cada bloco combinacional devem ser acessíveis via entradas/saídas primárias ou ligadas a um mesmo "scan".

- REGRA 9: Coerência entre sinais de controle de teste

  As condições de controlabilidade e observabilidade de nós conectados a barramentos de entradas e saídas de grandes blocos de circuito são exclusivas e consistentes, isto é sem a ocorrência de conflitos de níveis lógicos para os sinais envolvidos. Preferencialmente o acesso a esses blocos (memórias, RAM's, ROM's, PLA's) é feito via entradas/saídas primárias, diretamente ou através de multiplexação.

4.3 Protótipo Implementado

4.3.1 Descrição Geral

Foi implementado um protótipo para o SMAuT para processamento em estações de trabalho, que utiliza o Interpretador PROLOG fornecido pela Universidade de Edinburgh, Inglaterra, disponível nesse tipo de equipamento.

Relativamente à arquitetura do SMAuT ilustrada pela figura 4.1 o protótipo implementa parte do que foi especificado para o Módulo Verificador e para o Módulo de Interface com o Usuário. Implementa as regras definidas em 4.2.5 para projetos de CI's digitais semidedicados baseados em células amostradas por um único relógio. Gera os relatórios **Rel** e **Violações**, mas não admite ainda exceções como as mencionadas no item 1 da sub seção 4.2.4, e realiza o processamento até o final a partir de poucas interações iniciais com o usuário.

O Módulo Tradutor tem sua especificação de saída definida pela entrada aceita pelo Módulo Verificador. A entrada do Módulo Tradutor dependerá[4] da ferramenta normalmente usada para projeto, a ser adotada para integração com o SMAuT. Para implementação dess[illegible] módulo devem ser adotadas as linguagens C e PROLOG. As razões para us[illegible]ssas linguagens se devem a:

- o Módulo Tradutor poderá utilizar programação realizada em linguagem C, para procedimentos que não exijam abstrações, para aumentar a velocidade do processamento. A linguagem PROLOG deverá ser usada para

[4] A função prevista para o Tradutor foi realizada manualmente para o protótipo implementado.

funções que exijam abstração como por exemplo a associação da hierarquia do circuito com uma estrutura ("frame").

- o PROLOG se mostra adequado para a realização do Módulo Verificador de Regras de Projeto para Testabilidade por permitir que todas as informações necessárias à análise sejam declaradas como fatos, sobre os quais há mecanismos de inferência e de prova definidos, colocando á disposição uma máquina de inferência "built-in", isto é, do próprio interpretador.
- Além disso, de acordo com suas características o PROLOG permite que se implemente um conjunto de regras de testabilidade ao qual podem ser adicionados novos conjuntos com uma certa independência e facilidade.
- O algoritmo de unificação embutido no interpretador PROLOG provê suporte computacional para a verificação e para estratégias de controle independente de domínio, conforme argumentado por [Camurati88].

4.3.2 Classificação adotada

O Módulo Verificador implementado não prevê a entrada de descrições hierárquicas de circuito. Admite fatos associados ao circuito que atendam a seguinte classificação de células com as respectivas características de entrada e saída conforme a posição:

```
cel (TIPO, INSTANCIA, SAIDA):-""
comb (TIPO, INSTANCIA, SAIDA, -, -);"" %entrada, selecao
seq (NP, TIPO, INSTANCIA, SAIDA, -, -, -, -, -);"" %da, db"c-c~k, rst, sa

comb (TIPO, INST, Q, E, S):-""
(nor (INST, Q, E), TIPO = nor, S = []);""
(nand (INST, Q, E), TIPO = nand, S = []);""
(or (INST, Q, E), TIPO = or, S = []);""
(and (INST, Q, E), TIPO = and, S = []);""
(exor (INST, Q, E), TIPO = exor, S = []);""
(exnor (INST, Q, E), TIPO = exnor, S = []);""
(nbuf (INST, Q, ELE), TIPO = nbuf, E = [ELE], S = []);""
(bufl (INST, Q, ELE), TIPO = bufl, E = [ELE], S = []);""
(obufl (INST, Q, E), TIPO = obufl, S = []);""
(tribuf (NP, INST, Q, E, S), TIPO = tribuf);""
(iobuf (NP, INST, Y, Q, E, S), TIPO = iobuf);""
(ibufl (INST, Q, E), TIPO = ibufl, S = []);""
(mx (INST, Q, E, S), TIPO = mx);""
(bloco (INST, Q, E, S), TIPO = bloco).""

buf (INST, Q, E):-""
ibufl (INST, Q, E);""
bufl (INST, Q, E);""
((iobuf (NP, INST, Q, -, E, NEN);""
 tribuf (NP, INST, Q, E, NEN)), ""
 piv (-, NEN, -)).""
mx (I, Q, E, S):- ""
 (mx21 (I, Q, E, S);""
mx41 (I, Q, E, S);""
mx81 (I, Q, E, S)).""

seq (NP, TIPO, I, Q, DA, DB, CK, RST, SA):-""
 (TIPO = latch, latch (NP, I, Q, -, DA, EN), CK = 1, SA = vdd, DB = vdd);""
 (TIPO = ffd, ffd (I, Q, -, DA, CK), NP = vdd, RST = vdd, SA = vdd, DB = vdd);""
 (TIPO = ffdr, ffdr (NP, I, Q, -, DA, CK, RST), SA = vdd, DB = vdd);""
 (TIPO = mxffd, mxffd (NP, I, Q, -, DA, DB, SA, CK, RST)).""

mxffd (NP, I, Q, -, DA, DB, SA, CK, RST):-""
mxffd1 (NP, I, Q, -, DA, DB, SA, CK, RST);""
```

```
bitn (NP, I, Q, -, -, -, -, SA, DB, DA, CK, RST)."
```

Conforme cláusulas PROLOG explicitadas acima adotou-se as seguintes convenções:

- cel = célula; comb = célula combinacional; seq = célula sequencial; nor, nand, and, or, exor, exnor são portas combinacionais conhecidas por esses respectivos nomes; nbuf, bufl, obufl, tribuf, iobuf, ibufl são respectivamente associados a buffer inversor, não-inversor, de saída, tri-state, bidirecional, de entrada; mx para células multiplex, piv para pinos; latch, ffd, ffdr, mxffd, bitn são células sequenciais onde ffdr é o ffd(flip flop tipo D) com reset, mxffd é o ffd com entrada multiplexada e bitn é uma célula que além de um ffd apresenta um multiplex para implementação de contadores e um multiplex para facilitar o teste; bloco = bloco combinacional composto.
- NP = n ou p, para designar a polaridade em que o sinal de "reset" ou "preset" ou de habilitação de uma dada célula é ativo. Portanto, considera os dois tipos de polaridade no mesmo bloco ou circuito sob auditoria para qualquer tipo de célula.
- INST ou I = número ou alfanumérico univocamente associado a cada instância.
- Q e E são saídas, e entradas respectivamente, sendo que para muitas das células E corresponde a uma lista de entradas.
- S e SA são sinais de seleção. S pode ser uma lista (caso de células multiplex). SA é o sinal de seleção de uma ou outra entrada de células de flip flop com entrada multiplexada codificado como **mxffd**.
- NEN são os habilitadores de saída de buffers tri-state ou buffers bidirecionais.
- CK e RST são as entradas de sinais de relógio e "reset" ou "preset" respectivamente. O Tradutor deve associar o sinal de "reset" ou o de "preset" para uma posição comum no fato gerado.

Para exemplificar é apresentada a seguir a descrição associada ao circuito INVI, cujo diagrama lógico aparece no Apêndice D (ver figura D.3).

```
%%%%%%%%%%%%%%%%%%%%% INVI %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ffd(1,bciva,_,bciv,ra)
```

```
ff1(2,bcivaa,pen,bciva,ra).
mxffd(n,3,iciv,niciv,nre,bcivaa,ntest,ra,vdd).

nand(4,nen,[bciva,pen]).
nand(5,pnen,[niciv,nen]).
nand(6,nres,[nen"c-c"divt]).
nand(7,nitres,[bctl1,itres]).
nand(8,nicin,[bctl2,icin]).
nand(9"c-c"divt,[nitres,nicin]).
and(10,nre,[pnen,nres]).

pi([ntest,bciv,ra,bctl1,itres,bctl2,icin]).

po([iciv]).

modo'scan([[ntest,0]]).

clock([ra]).
```

Nessa descrição simples observa-se os números associados às instâncias, o sinal de seleção para a célula do tipo mxffd é ntest e a entrada de reset está ligada a vdd; mxffd tem entrada de reset ativo no nível baixo como indica o primeiro campo do fato **mxffd(n,...)**. pi(..), po(..), modo-scan(..) são fatos associados respectivamente aos pinos de entrada, de saída e à lista de sinais associados aos modos de teste das estruturas de teste implementadas no circuito.

A classificação de células adotadas permite aplicação bastante ampla do Verificador aos projetos de CI's digitais pois:

- não se restringe a núcleos de circuito, pois envolve células de periferia e provê ao protótipo a característica de controle total das condições dos testes implementados, de pino de entrada a pinos de saída. Para um circuito qualquer sob auditoria, como o ilustrado[5] pela figura 4.3 e descrito em um único nível hierárquico é possível verificar as interconexões a partir dos pinos de entrada, até os pinos de saída, como U4 ou bidirecionais como U3. Outro ponto positivo da classificação adotada pelo protótipo é a flexibilidade de particionamento em blocos obtida, admitindo qualquer desses tipos de célula em um subcircuito.
- é genérica quanto ao número de entradas de portas básicas do tipo NAND, NOR, AND, OR e MX (multiplex); neste caso (MX) a generalidade se refere a entradas de dado e de seleção, naturalmente.

4.3.3 Descrição da implementação das regras

Para realização das verificações o protótipo obedece uma ordem de ações que começa por uma classificação de sinais de relógio, "reset" e de seleção associados com as células sequenciais. Referindo-se às regras definidas em 4.2.5

[5]Não são explícitas na figura todas as entradas e saídas para os blocos U1 e U5.

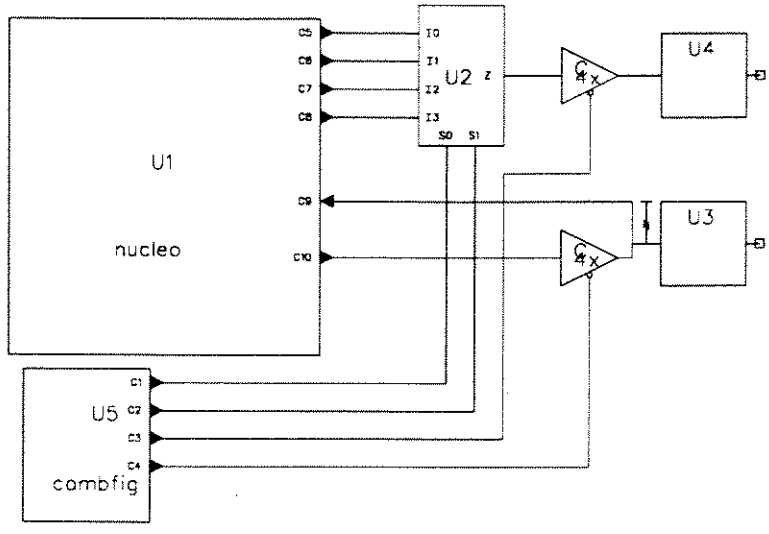

Figura 4.3: Flexibilidade derivada da classificação adotada

são descritas a seguir as ações implementadas para a verificação de cada uma delas:

1. REGRA 1: Estabilidade dos sinais

   Todos os elementos de memória devem ser implementados em flip flops mestre-escravo do tipo D.

   Para a verificação da ocorrência de flip flops que não satisfaçam essa regra, isto é que não sejam mestre escravo do tipo D (FFMS-D), dois tipos de verificações são necessárias.

   A primeira é que as células de flip flop utilizadas sejam desse tipo, o que é verificável na tradução da descrição hardware do circuito, por comparação contra um conjunto de células aceitas como conhecidas pelo Módulo Tradutor, ainda não implementado.

   A segunda se refere à verificação de configurações de portas combinacionais como elementos sequenciais. O Módulo Verificador implementa a detecção de uma configuração particular de "flip flop set reset" através do predicado[6] **claffsr** a seguir, que detecta pares de portas combinacionais quaisquer tais que a saída de uma é entrada da outra e vice-versa.

```
%detetar FFSR
cla'ffsr:-
```

[6]Predicado é a implementação em Prolog de uma verificação ou classificação que pode gerar um fato novo.

```
comb(',I,Q,L,[]),
(comb(',I1,Q1,L1,[]),
 I \== I1,
 not((ffsr(LI,',',',')),
 membro(I,LI))),
membro(Q1,L),
membro(Q,L1)),
asserta(ffsr([I,I1],Q,Q1,L,L1)),
%tab(2),write(ffsr([I,I1],Q,Q1,L,L1)),nl,
fail.
cla'ffsr
```

Esse predicado é apresentado na listagem do programa totalmente independente da sequência global listada para o **smaut**, apenas para ilustrar essa possibilidade. Os predicados são modulares e são tratados independentemente da ordem em que aparecem na codificação. São executados quando acionados por uma chamada, isto é, quando aparecem na sequência de programa em execução. O **claffsr** em particular poderia ser aplicado a módulos de circuito descritos por células combinacionais simples ou a descrições em que blocos combinacionais já tenham sido agregados, verificando a possibilidade de realimentações entre blocos combinacionais quaisquer. Isso quer dizer poderia ser acionado em mais de uma vez na sequência do **smaut**.

Por outro enfoque, para ser mais abrangente, foi implementada uma sequência de ações para determinar a existência de realimentações entre portas combinacionais, as quais podem resultar em comportamento sequencial. A busca de realimentações realizada pelo protótipo é ilustrada na figura 4.4 pelas interconexões A, B e C entre as portas combinacionais U1, U2, U3 e U4.

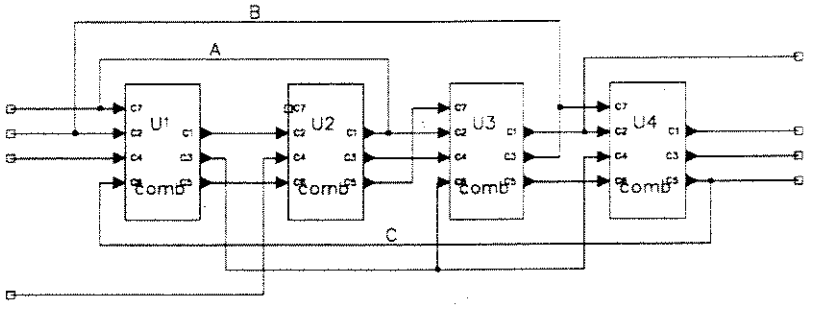

Figura 4.4: Detecção de realimentações entre portas combinacionais

Para atingir esse objetivo foi adotada a seguinte sequência de ações, que está implementada nos predicados **agregaportas**, **agregablocos** e pelos predicados chamados por eles:

- Associação de função de "fanout" aos sinais de saída de células combinacionais: a cada sinal Q de saída de porta combinacional é gerado um fato, no(Q, I, D), onde I é a instância que originou o sinal e D é o número de entradas às quais o sinal Q está conectado.
- Ligação entre portas: primeiramente são detectadas ligações entre duas portas combinacionais simples quaisquer (todas as combinacionais, exceto do tipo multiplex e do tipo bloco), conforme ilustrado na figura 4.5. Cada par de portas interligadas passa a ser associado a um fato que as define como um bloco combinacional; a seguir é feita a pesquisa que verifica se a saída (reali) de qualquer desses blocos é membro da própria entrada. Nesse caso, é verificado o "fanout" desse sinal (reali), potencialmente uma realimentação. Se ele apresenta "fanout" é gerado um novo fato que o define como realimentação, senão é verificado o fanout do sinal que interconectou as duas portas geradoras do bloco (Q1). Se Q1 tem "fanout" então Q1 é a realimentação entre essas portas.

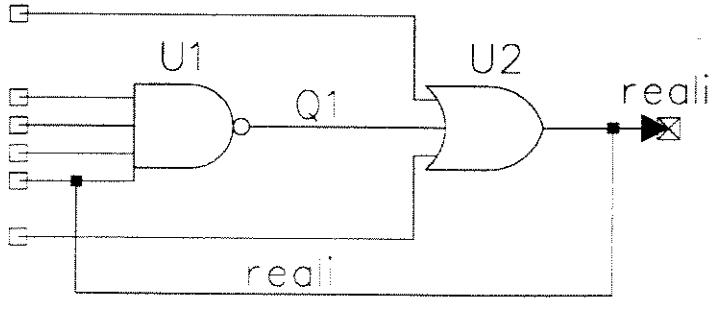

Figura 4.5: Portas Interligadas e detecção de realimentação

- Detecção de portas que não fazem parte de nenhum bloco(soltas), implementada pelo predicado **predncontido** usando um predicado do próprio PROLOG (setof[7](...)), que coloca em uma lista elementos que satisfazem uma dada condição.
- Detecção de ligação de portas soltas com blocos e detecção de realimentação analogamente à detecção do item 1 da sub seção 4.3.3.
- Ligação entre blocos combinacionais representada pela conexão X na figura 4.6: São pesquisados os sinais de interconexão entre blocos combinacionais, gerando um novo bloco. A seguir são pesquisadas as

[7]Esse predicado do Prolog(setof(..)) foi usado em vários dos predicados implementados no protótipo

saídas do novo bloco com relação às próprias entradas. Deve-se ressaltar que X e reali podem ser vários. A mesma regra de necessidade de "fanout" mencionada no item 1 da sub seção 4.3.3 é exigida.

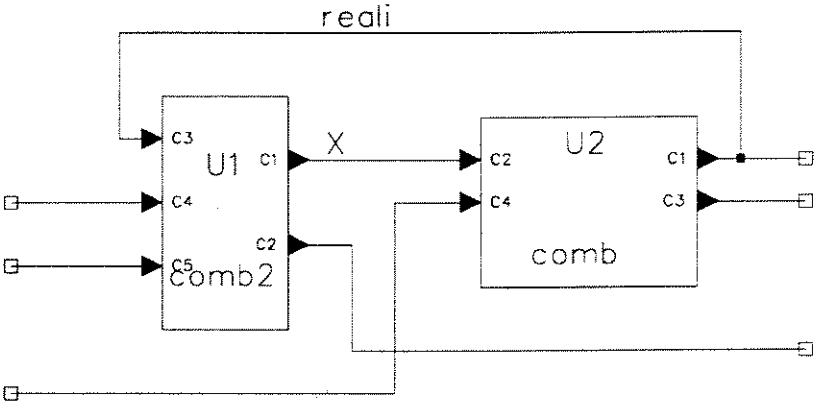

Figura 4.6: Blocos interligados e detecção de realimentação

2. REGRA 2: Sinal de modo teste

Um (ou mais do que um) pino de entrada primária deve ser alocado para especificar os modos ("scan" ou normal) de operação do circuito.

Aos sinais da lista que definem o modo "scan", implementada como dado de entrada para o Verificador, é aplicada uma pesquisa para identificar células sequenciais cujas entradas assíncronas ("reset" ou "preset") ou entrada de seleção(SA) de dado podem ser por eles inibidas. A figura 4.7 ilustra essa pesquisa. Para um sinal qualquer da lista de modo "scan" verifica-se o caminho através de U1, U2, ... que leva a SA de U3 ou CDN de U4 ou S0 de U5.

Para um sinal qualquer (sin-modo) da lista de modo "scan" está associado um nível (niv-sin), que é dado de entrada. É feita a busca de todas as portas combinacionais simples que tem esse sinal como entrada. A cada saída desses combinacionais é associado um novo nível de saída, dependente da função lógica do combinacional ser inversora ou não e do nível do sinal inicial. Na figura 4.7 o sinal de entrada passa pelo buffer U1 e pela AND U2 e portanto mantém a mesma polaridade de atividade definida para o sinal de entrada(constante da lista de modo-scan = dado

de entrada). Essa pesquisa portanto utiliza raciocínio do tipo "reasoning forward"(definição no Apêndice A).

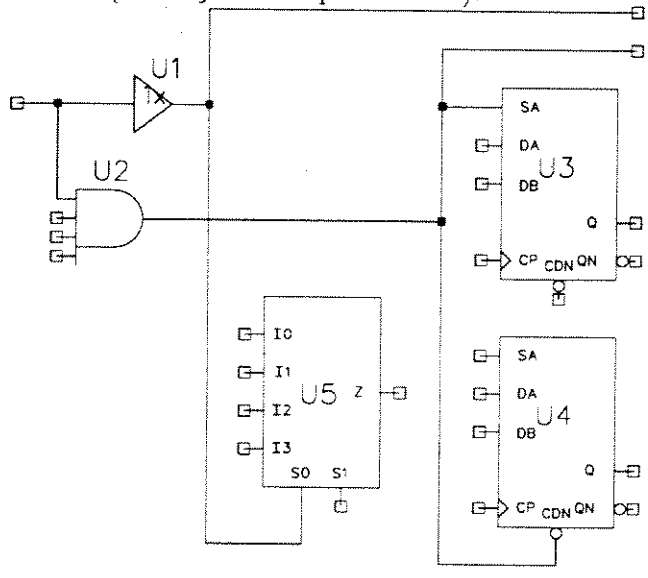

Figura 4.7: Cone associado com um pino de entrada - modo "scan"

A pesquisa é feita também para identificar destinos (entradas de "reset", "preset" ou SA) em células sequenciais. Quando encontrado um destino em uma célula sequencial N é verificada a possibilidade de inibi-lo comparando o nível propagado até N contra o nível de ativação associado com a instância da célula N. Essa pesquisa verifica a propagação de sinais da lista sin-modo por todo o circuito. A codificação associada a essa regra compõe-se do predicado **cla-cone** e dos predicados chamados por ele, que aparecem no código do protótipo no Apêndice D.

Essa regra é completamente verificada em conjunto com outros predicados(**clancel**, **claselsc** e os chamados por estes) que detectam células sequenciais fora de qualquer "scan", ou que detectam todos os sinais que controlam um mesmo "scan" e o compõem em uma lista de células.

Como exemplo da aplicação dessa regra pode ser vista a saída fornecida pelo SMAuT para o circuito CONFRA, diagrama lógico na figura D.4, conforme consta também no relatório REL do Apêndice D, qual seja:

```
 Modo scan dado pelos sinais e respectivos niveis
listados a seguir:

  [ntest,0] com [ntest,0]
```

```
rst:[nrese,1] com [ntest,0]
```

3. REGRA 3: Controlabilidade dos sinais de relógio

   As entradas de relógio dos flip flops devem ser controláveis a partir de entradas primárias e não de lógica ligada às saídas de dados de flip flops ou de outras entradas primárias.

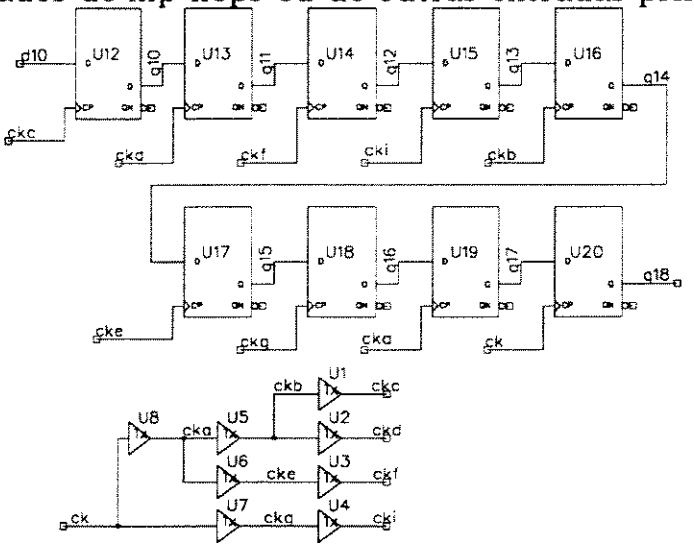

Figura 4.8: Distribuição de relógio - CLKTREE

De acordo com esta regra, no caminho entre pino e entrada de relógio de qualquer célula sequencial só pode haver buffers (inversores ou não). São ilustrados nas figuras 4.8 e 4.9 as condições aceitas para um sinal de relógio associado com uma célula sequencial relativamente à sua entrada de relógio. No caso da figura 4.8 o relógio das células U12 a U20 ou é pino de entrada(ck) oué bufferizado(cka, ckb, ckc, ckd, cke, ckf, ckg, cki). No caso da figura 4.9 representa-se uma árvore de distribuição de relógios análoga à anterior, porém com o uso de buffers inversores. Para cada sinal de relógio são contados os números de buffers inversores entre o pino e o relógio distribuído e é emitido um alerta ao usuário quanto à existência de relógios invertidos num mesmo "scan". O código associado com esta regra compõe-se dos predicados **claclk**, **peclk** e dos predicados chamados por eles que aparecem na lista de programa protótipo do Apêndice D.

Dos relatórios de saída associados com os dois circuitos exemplos CLKTREE e CLKNBUF apresentados no Apêndice D, as conclusões devidas à aplicação dessa pesquisa ao circuito CLKTREE aparecem a seguir:

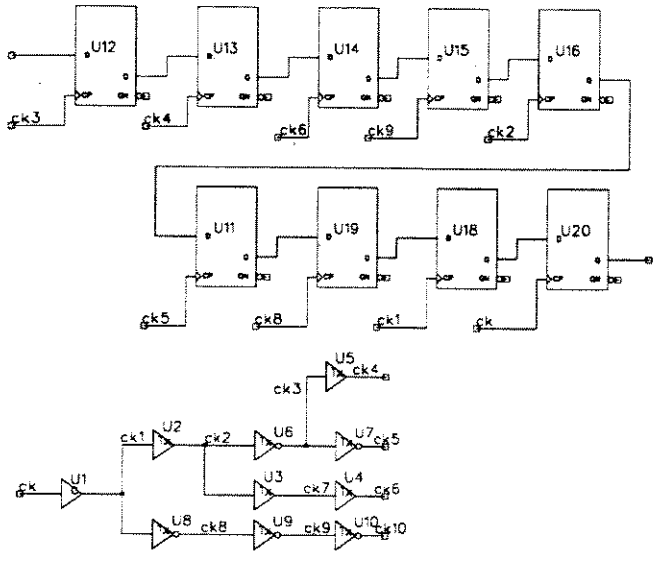

Figura 4.9: Distribuição de relógio com inversores - CLKNBUF

```
###          Arquivo de falhas
###
###   CI analisado:    clktree
###
###   Lista de sinais de clock verificada:
      SCAN(0) :   [ck"c-c"ka"c-c"kb"c-c"kc,
                   ckd"c-c"ke"c-c"kf"c-c"kg,
                   ckl]

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal cka, numero de buffers: 1
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal ckb, numero de buffers: 2
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal ckc, numero de buffers: 3
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal ckd, numero de buffers: 3
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal cke, numero de buffers: 2
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal ckf, numero de buffers: 3
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal ckg, numero de buffers: 1
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck
```

```
Para o sinal cki, numero de buffers: 2
e para o sinal ck, numero de buffers:0
```

Verifica-se também se as entradas de dados das células sequenciais dependem de qualquer sinal de relógio. Se houver uma lista de pinos de relógio a verificação é feita para todos os elementos da lista e sinais dele derivados; se essa lista não for dada como entrada as verificações são feitas relativamente aos sinais que são ligados às entradas de relógio de todas as células sequenciais e aos sinais que os geraram. Ver resultados obtidos para o circuito EXEMPLO na subseção 4.4. O texto Prolog correspondente à verificação das entradas de dados de células sequenciais é o dos predicados **veri-dado-nclk**, **veri-dado-clka** e dos predicados chamados por estes(ver lista de programa no Apêndice D).

4. REGRA 4: Limitação de entradas assíncronas

   Os flip flops não devem ter mais do que uma entrada assíncrona ("clear" ou "preset"), a qual deve ser controlável por entrada primária.

   A tradução da descrição do circuito para o arquivo de fatos deve reconhecer somente células sequenciais com uma única entrada assíncrona, assim como deve reconhecer células FFMS-D. A verificação da controlabilidade das entradas assíncronas é feita pelo Módulo Verificador através de:

   (a) busca, como ilustrada pela figura 4.10, da origem desses sinais a qual deve ser saída de alguma porta combinacional simples (Q1, Q2, Q3, Q4 e Q5), que tem pelo menos uma entrada diretamente controlável (A1, A2, A3, A4 e A5) a partir de pinos de entrada. Por "diretamente controlável" entende-se que só há buffers no caminho entre pino de entrada e o sinal usado para controle assíncrono.

   (b) comprovação da existência de um nível que inibe essa entrada assíncrona no modo "scan", obtido do predicado descrito para a REGRA 2 no item 2 da subseção 4.3.3. As cláusulas (ou predicados) PROLOG implementadas para essa busca são **clanpi**, **pin** e aquelas utilizadas por estas(ver lista de programa no Apêndice D).

5. REGRA 5: Serialização de elementos sequenciais

   Todos os flip flops são conectados em "scan" (registrador de deslocamento). Preferencialmente, todas as mudanças de sinais ocorrem com uma mesma fase de relógio em modo teste.

   É feita a pesquisa de interligação entre células sequenciais, que satisfazem da PRIMEIRA à QUARTA regras, conforme ilustrado pela figura 4.11. Para que uma conexão entre duas células sequenciais U4 e U11, por exemplo, seja aceita devem ser satisfeitas as condições de relógio, de entradas assíncronas e inibição por um sinal de entrada para sinais de seleção de entrada de dados, verificadas por essas regras. É gerado um

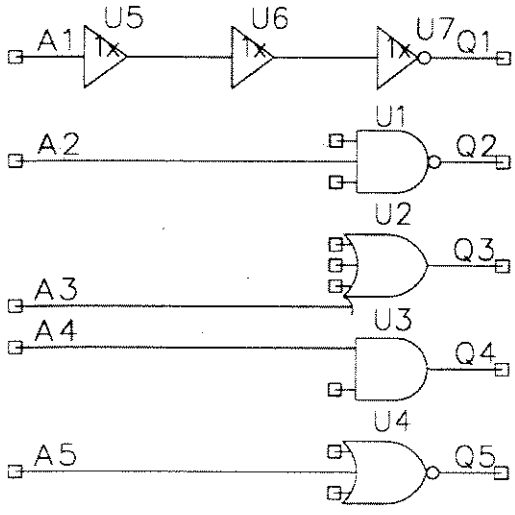

Figura 4.10: Controlabilidade de entradas assíncronas de sequenciais

fato denominado **caminho**[8] associado às células sequenciais que formam um registrador de deslocamento, partindo de: pinos de saída originados em células sequenciais (QSAI), ou de qualquer ligação entre duas células sequenciais. Os predicados **claqsai**, **clacaminho** e os predicados chamados por estes foram usados para implementar essa regra(Ver lista de programa no Apêndice D).

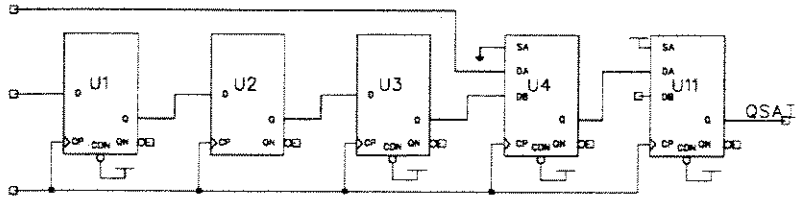

Figura 4.11: Caminho formado por células sequenciais

Para as células sequenciais que não pertencem a qualquer **caminho** são pesquisadas conexões com células multiplex, como ilustrado na figura 4.12. Se for detectada uma conexão como o sinal QFA entre U1 - U3, tal que QM está associada a uma saída, é observado em relatório de saída que essa célula não está ligada a nenhum **caminho**, embora seja observável.

[8] No texto que segue quando a palavra caminho for associada ao fato ela aparece em negrito

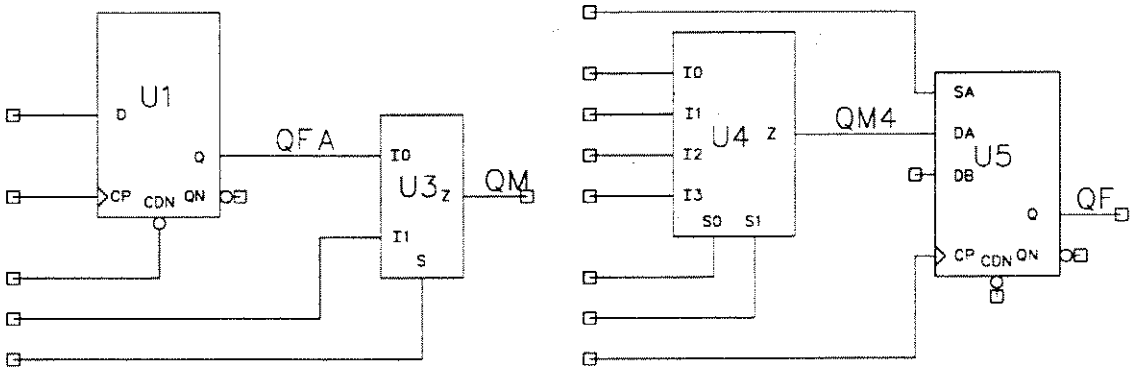

Figura 4.12: Células sequenciais fora de **caminho**

Se é detectada uma conexão como QM4 entre U4 e U5, então é sugerida em relatório de saída, uma modificação de projeto, com adoção de uma célula sequencial com entrada de dados multiplexada para que ela seja incluída em algum **caminho** em modo "scan".

Para os **caminhos** gerados são pesquisadas as conexões dos últimos elementos sequenciais de cada **caminho** com uma célula combinacional do tipo: multiplex, buffer tri-state, buffer de saída ou buffer bidirecional, como U12, U13 e U14 da figura 4.13. A instância associada a essa célula é acrescentada ao **caminho** desde que suas entradas de seleção ou de habilitação sejam controláveis. É verificado também se as saídas dessas células combinacionais (multiplex U12 e buffer tri-state C) são associadas com pinos de saída em condições controláveis.

6. REGRA 6: Controlabilidade dos "scans"

   Cada "scan" tem uma entrada primária e uma saída primária no modo scan. Para isso, podem ser usadas ou compartilhadas entradas e saídas primárias.

   Para cada **caminho** gerado são verificadas as possibilidades ilustradas nas figuras 4.14 e 4.15 de se fazer respectivamente:

   (a) "scanin" através de uma entrada de dados da primeira célula do **caminho**, que é uma célula sequencial. Essa verificação é feita pelo predicado **clascanin**;

   (b) bem como de se fazer "scanout" através da saída de dados da última célula de cada **caminho** detectado. Essa última célula de cada **caminho** podera ser combinacional ou sequencial. Se combinacional, sua saída

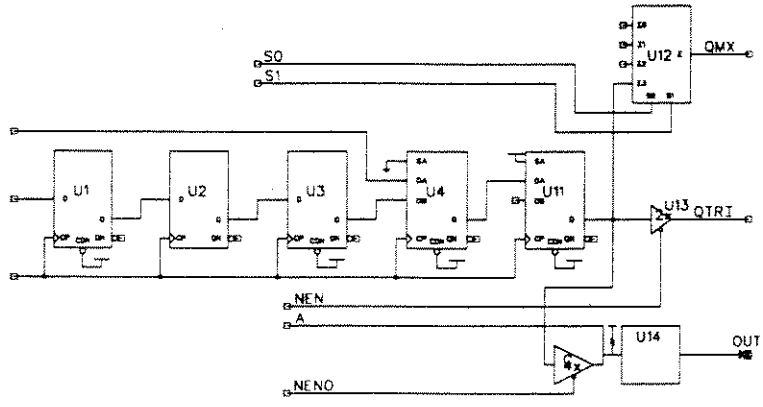

Figura 4.13: Caminho conectado a célula combinacional

deve estar relacionada com uma lista de pinos de saída ou de pinos bidirecionais. Se sequencial, a saída da última célula do **caminho** pode ainda admitir um buffer tri-state ou um multiplex antes do pino de saída ou bidirecional. A implemetação dessa verificação é feita pelo predicado **clascanout**.

7. REGRA 7: Garantia de verificação de flip flops

   No modo "scan" as saídas de cada flip flop assim como a saída primária de "scan" são função da saída do flip flop precedente ou da entrada primária do "scan".

   Para verificar esta regra são geradas três listas e tipos de verificação:

   (a) todos os sinais de relógio de cada **caminho** são listados e comparados para ver se existe número diferente de buffers inversores ou não entre eles ou se há pinos de relógio diferentes associados com esse **caminho**. É considerado aceitável um projeto que inclua diferente número de buffers não-inversores entre dois sinais de relógio de um mesmo "scan", sendo emitido um aviso ao projetista o qual deverá se ocupar das questões de temporização para evitar problemas de corrida entre os sinais de dado e de relógio, como ilustrado na figura 4.16. D1 pode mudar antes que U2 consiga amostrá-lo, devido ao atraso em U3 em condições críticas de atraso.

   (b) todos os sinais de "reset" ou "preset" são listados e os níveis associados à inibição de cada um deles são relacionados numa nova lista onde os conflitos podem ser verificados.

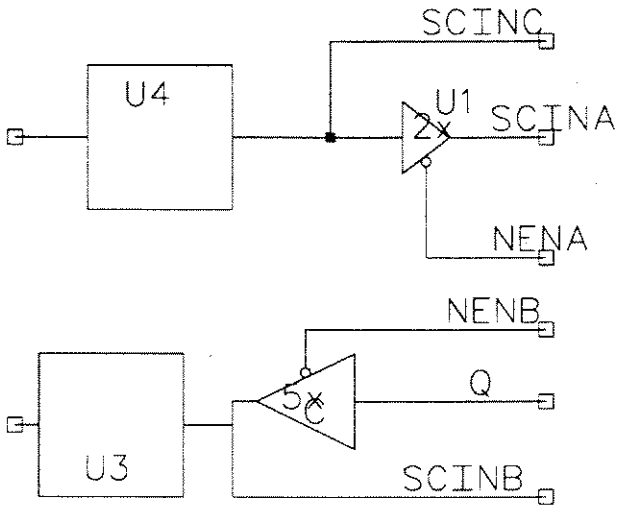

Figura 4.14: Possibilidades de "scanin": SCINA, SCINB e SCINC

(c) todos os sinais de seleção de entrada de dados e sinais de habilitação de buffers tri-state de cada **caminho** são listados com níveis necessários para realizar o "scan" associado.

A lista de sinais envolvidos e níveis necessários para satisfazer a REGRA 7 é associada a fatos relacionados ao "scan"(um número é atribuído a cada "scan" e ele é descrito pelo fato **scan**). Esses fatos gerados por vários predicados chamados por **claselsc** constam das listas de estruturas de teste implementadas que aparecem no relatório **Rel**.

8. REGRA 8: Controlabilidade e Observabilidade de blocos combinacionais

   As entradas e saídas de cada bloco combinacional devem ser acessíveis via entradas/saídas primárias ou ligadas a um mesmo "scan".

   A REGRA 1 teve como consequência a geração de blocos de circuito combinacionais presentes no circuito sob auditoria, com lista de sinais de entrada e de saída. Não incluiu células multiplex e não elimina saídas cujos destinos são internos aos blocos, a não ser que o "fanout" seja igual a um.

   A lógica combinacional interconectada resultou num único bloco pela aplicação de um predicado **agrega-bl-entco** que gera blocos a partir de dois blocos que tenham entradas comuns (pelo menos uma). Isso poderá ser visto para o exemplo da figura D.4, no respectivo relatório Rel no apêndice D.

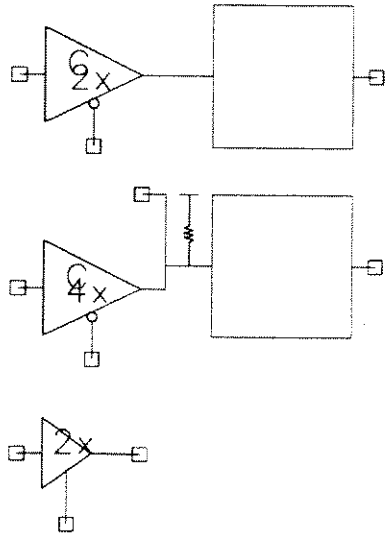

Figura 4.15: Possibilidades de "scanout": buffers tri-state e buffers de saída

9. REGRA 9: Coerência entre sinais de controle de teste

As condições de controlabilidade e observabilidade de nós conectados a barramentos de entradas e saídas de grandes blocos de circuito são exclusivas e consistentes, isto é sem a ocorrência de conflitos de níveis lógicos para os sinais envolvidos.

A implementação desta regra é muito simples considerada uma descrição de circuito em um único nível hierárquico. Trata-se de verificar as condições de controlabilidade das entradas, analogamente à verificação de "scanin", e de verificar as condições de observabilidade das saídas, como realizado para os "scanout". Os conflitos entre níveis lógicos associados a sinais de

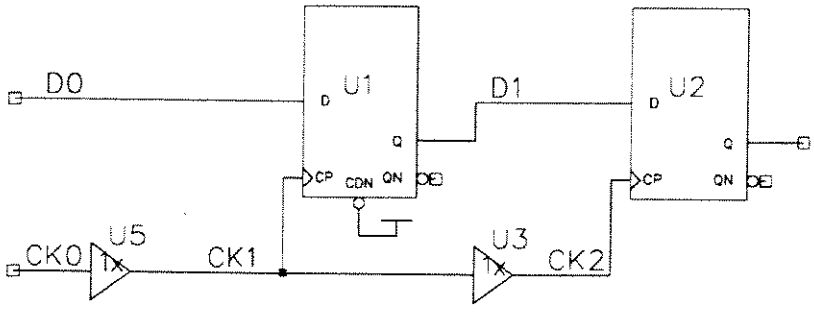

Figura 4.16: Verificação de diferentes atrasos para diferentes relógios de um "scan"

controle para que essa controlabilidade e observabilidade sejam satisfeitas, são verificados de forma análoga às verificações realizadas para sinais de controle associados aos "scans".

Outras regras podem facilmente ser acrescentadas ao SMAuT sem interferir nas regras já implementadas, como por exemplo a detecção de "wired logic". Trata-se de uma simples verificação de curto entre duas saídas de um combinacional qualquer, ilustrada pela figura 4.17 pelo sinal QW, conectado em curto como saída de U1 e U2 e que não seja declarado como TRI-state. A programação desse predicado ficaria assim:

```
%"begin-center"

wlogi:-""
comb (-, INSTANCIA1, SAIDA1, -, -), ""
comb (-, INSTANCIA2, SAIDA2, -, -), ""
membro (X, SAIDA1), ""
membro (X, SAIDA2), ""
asserta (wired-logic (X, INSTANCIA1, INSTANCIA2), ""
fail.""
wlogi.""

%"end-center"
```

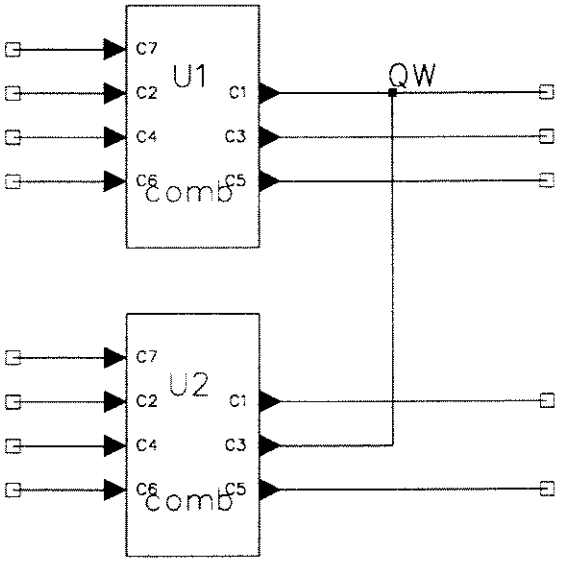

Figura 4.17: Detecção de "wired-logic"

4.3.4 Casos práticos verificados com o protótipo

São apresentados no Apêndice D os diagramas lógicos, os dados de entrada utilizados e os respectivos relatórios de saída para os circuitos MAQSUP,

CONV, CONTMOD7K, INVI, CONFRA, CTRMEM e COMBRX, submetidos ao SMAuT.

O circuito MAQSUP é o mesmo discutido no estudo de casos do apêndice B. O protótipo identifica diferentes sinais de relógio associados ao mesmo "scan" e os apresenta como ERRO.

Dos outros casos práticos analisados pelo auditor verifica-se nos relatórios a detecção dos "scans", múltiplos ou não em cada um deles, os níveis associados aos sinais de controle necessários para que os "scans" funcionem e a lógica combinacional interligada em blocos.

As realimentações detectadas no caso de COMBRX ocorrem devido a alta densidade de interconexões. A figura 4.18 ilustra a situação prática detectada. O bloco I é resultado da agregação das células 64 e 11. A saída **nqz** de I é entrada do bloco I1 que tem **nmxr** como saída e **nmxr** é entrada do bloco I. Portanto, I e I1 podem ser agregados como um único bloco combinacional, mas **nqz** é detectada como uma realimentação entre I e I1, o que não corresponde à interpretação correta pois existe na verdade um circuito em malha aberta. Seria realimentação se **g11ny** que é função de **nmxr** fosse entrada do bloco I1 (Ver observações a respeito no capítulo 5).

4.4 Considerações Finais

Os resultados obtidos com o protótipo são observáveis através dos casos práticos apresentados no Apêndice D. Para melhor ilustrar todos os casos de células, níveis e polaridades considerados pelo protótipo indicamos o circuito EXEMPLO da figura 4.19 (sua descrição de entrada e seus relatórios de saída que aparecem no Apêndice D permitem verificar, por exemplo a dependência de uma entrada de dados com relação a um sinal de relógio qualquer).

Da aplicação do protótipo aos vários circuitos observa-se também como resultado as listas de estruturas de teste implementadas e consideradas válidas pelo Verificador. Essas listas incluem as instâncias das células por "scan" na ordem em que estão ligadas e o número de elementos sequenciais interligados (Ver Apêndice D).

O protótipo está implementado por aproximadamente 1500 linhas de programa em PROLOG. Para que ele seja consultado, isto é, seja lido pelo interpretador PROLOG e se torne disponível para interpretação são necessários menos de

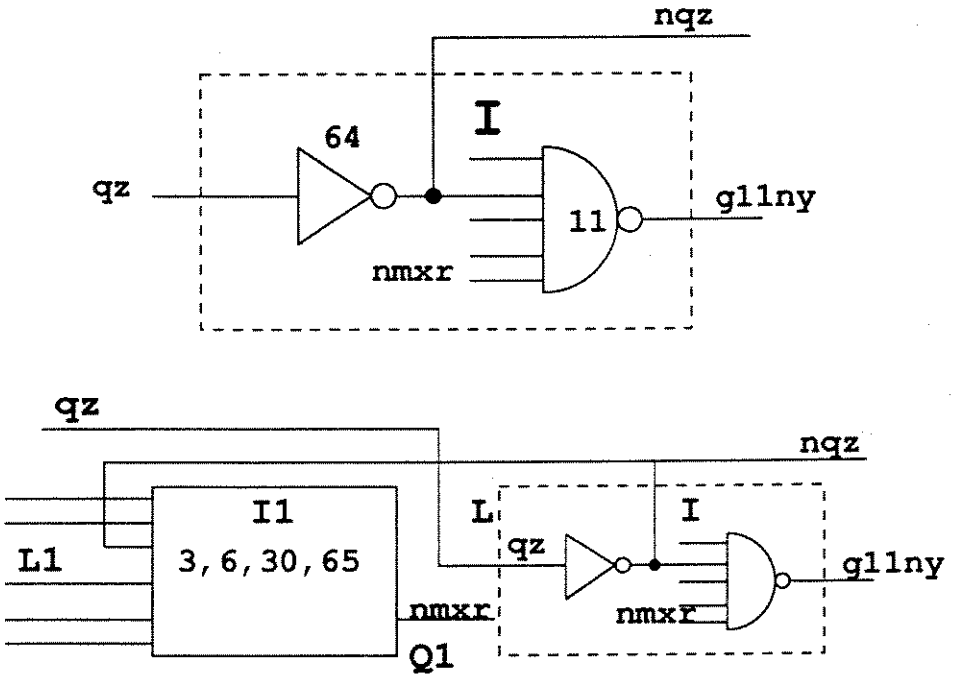

Figura 4.18: Paralelismo entre portas de diferentes blocos

15 segundos de tempo de processamento em estação SUN3.

Quanto à performance não foi observada demora significativa para o processamento de qualquer dos casos, nem tampouco problemas de espaço mesmo no caso do circuito COMBRX que é altamente denso, considerada a utilização de estação de trabalho SUN3. A estatística fornecida pelo PROLOG após a aplicação do SMAuT ao COMBRX é a seguinte:

- quanto à área utilizada para átomos[9]: 346 Kbytes em uso no total de 501 Kbytes. Para o CONTMOD7K essa área é de 258 Kbytes em uso.
- área ocupada por estruturas de dados construídas na execução do programa: 63 Kbytes em uso no total de 1050 Kbytes.
- área para informações de controle e variáveis necessárias na execução do programa: 732 bytes em uso no total de 130 Kbytes.
- área para referências das variáveis: 32 bytes em uso no total de 65 Kbytes.
- tempo de processamento de 170.07 segundos; no caso do circuito CONTMOD7K esse tempo é de 30.59 segundos.

[9] Ver Apêndice A.

Para o circuito COMBRX foi detectado o problema de geração de blocos combinacionais repetidos, devido às características da linguagem PROLOG, que pode ser resolvido facilmente por um programa em linguagem C. Pode-se ver com facilidade que o mesmo bloco aparece como saída repetidas vezes, como resultado da agregação programada em Prolog. Com o uso de um programa em C pode ser feita a eliminação dessas repetições e pode-se prosseguir em Prolog.

Como em todo sistema baseado no conhecimento, é necessária a reavaliação de objetivos, da arquitetura, da linguagem utilizada e de todas as suas características para se dar continuidade ao seu desenvolvimento. O tratamento de descrições hierárquicas por exemplo, se considerado viável pode ser feito traduzindo-as inicialmente para uma descrição num único nível e tratando a verificação da descrição achatada (módulos expandidos substituídos por internos até que só haja células básicas). Porém, é prevista a perda em eficiência se considerarmos a possibilidade de criar fatos associados com a verificação de subcircuitos e aproveitá-los quando outros circuitos de mesmo tipo são instanciados. O sistema descrito em [Camurati88] usa frames descritos em forma de cláusulas PROLOG para as descrições de módulos hierárquicos e prevê o uso de módulos hierárquicos definidos em diferentes frames.

Em termos de abrangência das regras e da classificação adotada, o SMAuT está preparado para atender uma ampla faixa de projetos, dado que as descrições hardware de circuitos podem ser traduzidas para as células consideradas, abandonando-se características como tamanho de buffer, definições de biblioteca de células e outras, devido ao fato de que as verificações são estáticas, não levando em conta características temporais.

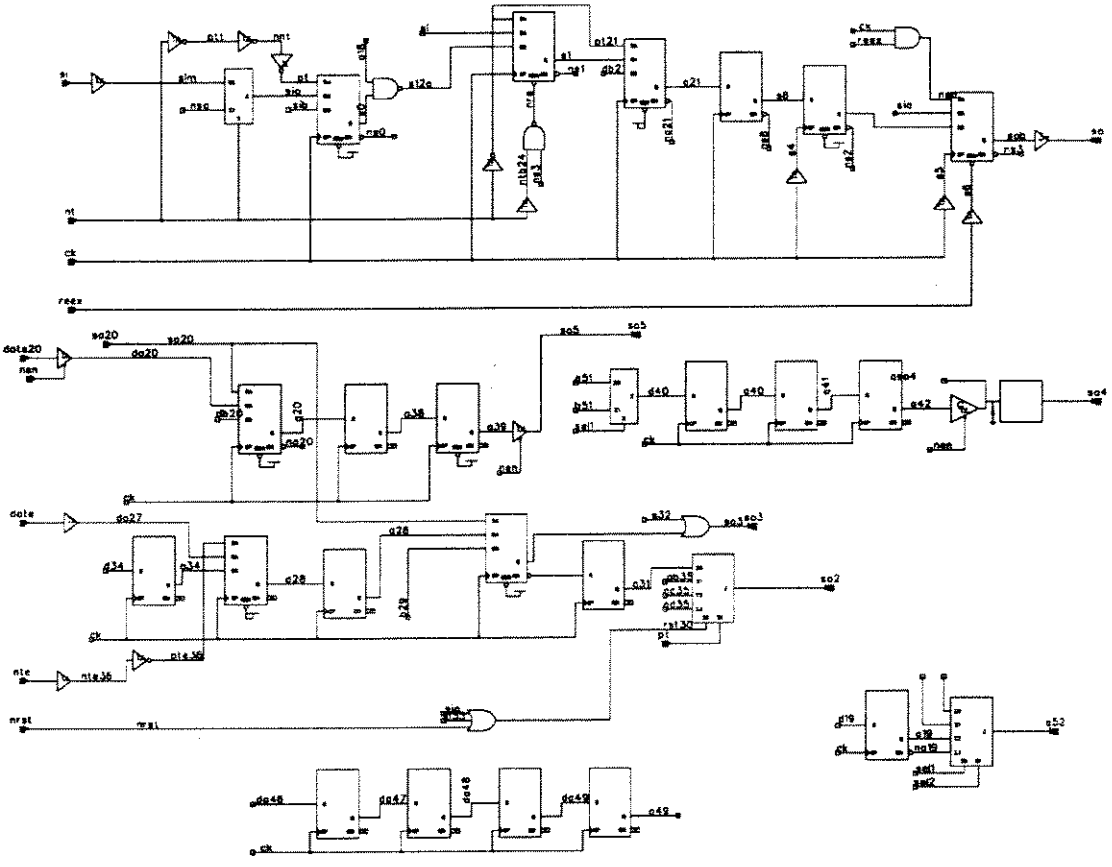

Figura 4.19: Diagrama Lógico - EXEMPLO

Capítulo 5

Conclusão

O aumento na complexidade dos projetos causa impacto no tempo necessário para projeto, indo de poucas semanas para projetos simples a milhares de homens-hora para projetos mais complexos. Segundo [Niessen83], o desenvolvimento de metodologias de projeto hierárquico é necessário para que os projetos sejam limitados pela tecnologia, em vez de serem limitados pela capacidade de projeto.

Os objetivos de uma metodologia de projeto VLSI incluem: considerar todo o ciclo de projeto, controlar a complexidade tal que haja confiança na correção dos resultados, utilizar eficientemente as possibilidades tecnológicas, resultar aumento considerável de produtividade de projeto e permitir a criação de ferramentas eficientes.

Com a opção de sistemas especialistas, que permitem uma evolução dos sistemas por ampliação de bases de conhecimento, é promissora a idéia de desenvolvimento de ferramentas de análise, de síntese e de gerenciamento de projetos de CI's, utilizando aquele enfoque.

A inclusão de auditoria de testabilidade na metodologia de desenvolvimento de CI's não envolve muito tempo para execução, dependendo sómente da tradução da descrição do circuito e de sua adequação ao conjunto de células previsto pelo auditor, contra um significativo ganho em custo de projeto e mesmo em tempo de desenvolvimento total.

O uso de um auditor de testabilidade é justificado ainda, devido aos seguintes fatos:

- as ferramentas disponíveis para projeto, não realizam o ciclo completo de concepção de CI's, e o acréscimo em tempo devido ao uso de uma ferramenta de análise como essa é muito baixo relativamente ao que pode ser economizado na duração total do ciclo.
- a metodologia de projeto mais comumente adotada, que mescla enfoque "top-down" e "bottom-up", permite dividir o projeto de um CI entre vários projetistas. A auditoria de testabilidade é um meio de se realizar uma equiparação da qualidade de cada projeto em termos da capacidade de teste do CI, o que assume importância expressiva, especialmente para grandes sistemas integrados.
- a tarefa de auditoria é executada de modo mais confiável, auto documentada, e pode apresentar como resultado os testes implementados, claros e bem definidos, para auxiliar a atividade subsequente de geração automática de padrões de teste.
- a possibilidade de aplicar o auditor aos circuitos repetidas vezes durante a concepção e em diferentes níveis hierárquicos de descrição dos circuitos.

A aplicação de um auditor como o prototipado, em diferentes etapas de projeto, permite assegurar o atendimento a regras de DFT a circuitos descritos de forma não-hierárquica e consequentemente, contribui para a eficiência do ciclo de desenvolvimento de CI's e para estender os limites da capacidade de projeto. Melhores resultados são possíveis, inclusive utilizando resultados de programas em C no meio da sequência de ações necessária, alternativa possível para o PROLOG[Bowen86]. Além disso, a implementação do módulo Tradutor integrado ao ambiente de desenvolvimento, bem como a possibilidade de tratar descrições hierárquicas como o sistema descrito por [Camurati88] tornam a adoção desse auditor mais viável no ciclo de desenvolvimento de circuitos integrados.

Por outro lado o sistema auditor prototipado tem funções não previstas pela maioria das ferramentas disponíveis para apoio aos projetos. Embora as ferramentas existentes sejam cada vez mais sofisticadas e aperfeiçoadas para o projeto de determinados tipos de estruturas, incluindo técnicas de testabilidade, o sistema prototipado preenche um espaço para o projeto como um todo, que é o controle de todas as situações de teste implementadas.

A experiência obtida com a aplicação do protótipo a circuitos práticos gera otimismo quanto à concretização dos efeitos pretendidos. A classificação de células considerada proporciona flexibilidade para verificação de circuitos. As regras implementadas produziram resultados corretos para a maior parte dos casos de circuitos testados. Somente foi detectada a interpretação errada

de realimentação para o caso do COMBRX que tem alta densidade de conexões(problema apontado na subseção 4.3.4). Considerando que as regras são validadas pela frequência com que produzem resultados corretos, as regras implementadas se mostraram válidas. A linguagem e o interpretador PROLOG utilizados se mostraram adequados à implementação. É possível implementar um predicado em PROLOG para evitar essa interpretação errônea: tal predicado deveria pesquisar se as realimentações detectadas são resultantes de lógica que recebe nas suas entradas sinais do bloco que está sendo agregado; se isso não acontece configura-se malha aberta e portanto deve-se invalidar o fato **reali(X)** associado ao sinal X.

O auditor especificado inclui tratamento de descrições hierárquicas de circuitos. Para dar continuidade ao sistema protótipo esse é um passo muito importante a ser desenvolvido. Outro ponto não abordado pelo protótipo é o Módulo Tradutor, que pode ser implementado a partir de uma linguagem de descrição de hardware. No CPqD o CAD disponível ou potencialmente disponível oferece três alternativas de linguagem de descrição para entrada do Tradutor: o formato de descrição de hardware da linguagem HILO (GenRad). o formato de saída de "netlist" do software usado pela VLSI Technology Incorporated-VTI, ou uma linguagem padrão como EDIF ou VHDL. A última alternativa é a mais promissora, para os dois casos possíveis: ferramenta integrada ao CAD em uso ou ferramenta independente.

As regras implementadas pelo protótipo são associadas aos projetos que utilizem células sequenciais de relógio único. Em termos de Módulo Verificador a evolução pode envolver o acréscimo de regras para:

- Verificar sistemas que utilizam mais de um relógio e sem superposição, quanto a regras de projeto estruturado.
- Verificar lógica para segmentar circuitos sequenciais de muitos elementos sequenciais encadeados (contadores e cadeias divisoras), em partes mais facilmente testáveis (técnica AD HOC).
- Verificar acessos de teste a pontos de "fan in" e de "fan out", de modo não conflitante com outros modos de teste implementados, eventualmente concorrentes (técnica AD HOC).
- Detectar sub circuitos mais difíceis de testar, como por exemplo uma PLA embutida num circuito com dificuldades de acesso.

Como ampliação de suas funções o auditor de testabilidade especificado poderá ser utilizado para avaliar diferentes técnicas ou metodologias de DFT que

se tornarem disponíveis na base de conhecimento, objetivando realizar comparações entre várias soluções possíveis. Esse tipo de avaliação é possível, mas de acordo com a literatura depende de variáveis geradas por outras ferramentas, tais como:

- de um ATPG, o comprimento de padrão para avaliar custo de teste,
- de um simulador de falhas, a cobertura de falhas,
- de um gerador de leiaute, o "overhead" de área,
- e de um simulador lógico, os atrasos.

Para viabilizar essas avaliações é necessário estabelecer interfaces com todas essas ferramentas ou integrar o SMAuT ao CAD utilizado. No caso de ser possível integrar o auditor ao sistema de projeto (entrada de esquemático, simulador e demais ferramentas), a entrada de dados fica embutida no sistema, sendo transparente para o usuário do auditor, que só informaria ao SMAuT dados do tipo dos items 2 e 6 da subseção 4.2.2.

Além da saída mencionada na subseção 4.2.3, a saída gráfica e totalmente integrada ao software utilizado para o projeto, na forma de "back annotation", permitiria identificar mais rapidamente a falha e proporcionaria facilidade de modificação mais imediata. Após a modificação do circuito, a integração com a ferramenta utilizada para o projeto permitiria o processamento ágil do auditor.

O uso de auditoria de testabilidade, especialmente de forma integrada ao software utilizado para desenvolvimento de projetos de circuitos integrados, contribui para tornar o projeto robusto, com baixíssimo aumento de tempo nesse ciclo total de desenvolvimento.

Apêndice A

GLOSSÁRIO

São muitos os termos usados em inteligência artificial e para descrever o sistema auditor de testabilidade é usado um pequeno subconjunto deles, cujas definições extraídas de [Winston] são listadas abaixo:

- ÁTOMO [Clocksin84] : são os nomes de constantes utilizadas em PROLOG. Há dois tipos de átomos: os que são feitos de letras e dígitos e os que são feitos de sinais.
- FATO : expressa uma verdade sobre uma relação.
- REGRA: expressa uma relação entre fatos.
- INFERÊNCIA: o processo de atingir conclusões através de raciocínio lógico.
- MECANISMO DE INFERÊNCIA: a parte de raciocínio de um sistema baseado no conhecimento que aplica um conjunto de regras a um conjunto de fatos dados e deriva conclusões lógicas.
- SISTEMA DE MANUTENÇÃO DA VERDADE : um sistema que quando confrontado com dados adicionais ou uma modificação de dados, decide quais condições, regras ou suposições devem mudar para manter consistência.

- BASE DE CONHECIMENTO: a porção de um sistema especialista ("expert system") que contém o conhecimento factual e inferido a ser processado. Estes dados são distintos do controle de programa ou mecanismo de inferência.
- SISTEMA BASEADO NO CONHECIMENTO: um sistema de computador que consiste de conhecimento sobre uma área específica de habilidade mais que de algoritmos. Do inglês KBES, Knowledge-Based Expert System.
- ENGENHARIA DO CONHECIMENTO: construir sistema baseado em conhecimento por aquisição e representação de conhecimento de peritos humanos.
- ENCADEAMENTO DE RACIOCÍNIO PARA TRÁS ("Reasoning-backward chaining"): dedução de uma conclusão voltando através de um conjunto de regras e fatos para determinar se aquele conjunto teria levado à conclusão. Também chamado "goal-driven reasoning".
- ENCADEAMENTO DE RACIOCÍNIO PRA FRENTE ("Reasoning-forward chaining"): raciocinar para frente numa sequência lógica, usando fatos e regras para chegar a uma conclusão. Também chamado "data driven reasoning".
- CONTRADIÇÃO: dois fatos que não podem ser simultâneamente verdade.
- RESOLUÇÃO DE CONTRADIÇÃO: resolver uma contradição mudando a verdade de um dos dois fatos que causaram a contradição, encontrando uma razão para desacreditar no fato prèviamente considerado verdadeiro.
- REPRESENTAÇÀO : conjunto de convenções sintática e semântica que permite descrever uma classe de coisas.
- DESCRIÇÀO : faz uso de convenções da representação para descrever alguma coisa.
- SINTAXE DE UMA REPRESENTAÇÃO : especifica os símbolos que podem ser usados e os modos em que estes símbolos podem ser arranjados. Trata-se da relação de vocabulário e estrutura com algum mundo.
- SEMÂNTICA DE UMA REPRESENTAÇÃO : especifica como o significado está englobado nos símbolos e nos arranjos de símbolo permitidos pela sintaxe.
- REDES SEMÂNTICAS : denotam objetos e descrevem a relação entre eles. Consistem de nós e enlaces ("links") e alguns significados de nós e enlaces relativos a objetos, relações, ações e eventos.

- ESTRUTURA ("FRAME") : coleção de nós e espaços de redes semânticas que juntas descrevem um objeto, ato ou evento estereotipado.
- CAMPO ("SLOT") de um nó : corresponde a uma ligação ("link") diferenciada por um nome.
- "DEFAULT" : valor provávelmente verdadeiro assumido em lugar de um fato. Trata-se de um valor provável, mas não certo.
- "DEMON" : são procedimentos armazenados em "slots" tal que eles possam ser invocados automáticamente. "IF- NEEDED DEMON" computa valor quando ele é necessário.
- "BUSCA HEURÍSTICA": são métodos de busca que usam informações de domínio específico para guiar o processo de busca e reduzir o tempo de computação. Pode não garantir encontrar a solução [Hayes].
- "DEPTH-FIRST" : procedimento de busca em que dado que um caminho é tão bom quanto um outro qualquer, um nó é escolhido e outras alternativas no mesmo nível são ignoradas completamente, enquanto há esperança de atingir o destino, usando essa escolha. É um dos procedimentos mais simples de busca, não encontra o melhor caminho e nem é mais eficiente que procedimentos mais informatizados.
- ABSTRAÇÃO : sumarização de descrições complicadas por reconhecimento de padrões.

Apêndice B

ESTUDO DE CASOS

Apresentamos alguns casos detectados em projeto de circuitos integrados digitais que resultaram em problemas para teste, devido a violações de regras ou conflitos de teste implementados e que poderiam ser evitados com a utilização de um sistema de auditoria.

B.1 Sub circuitos do TB15

Circuito projetado usando "standard-cells", com aproximadamente 11000 transistores ou 3000 gates equivalentes, incluindo memória de 64 X 9 bits, tendo sido implementado dois modos funcionais e um modo teste. O modo teste inclui a implementação de 7 "scans", a multiplexação para observação de sinais importantes do circuito, como por exemplo os sinais gerados internamente para enable da memória, a aceleração de contadores por relógio externo, além do acesso às entradas de dados, endereços e enables da memória. O circuito também permite testar o funcionamento da memória através de teste "on-line" implementado com o uso de geração e detecção de paridade do dado escrito e lido na memória interna do TB15.

B.1.1 Exemplos de violações de regras de projeto

- No circuito da figura B.1 observa-se como violações de regras de projeto para testabilidade:

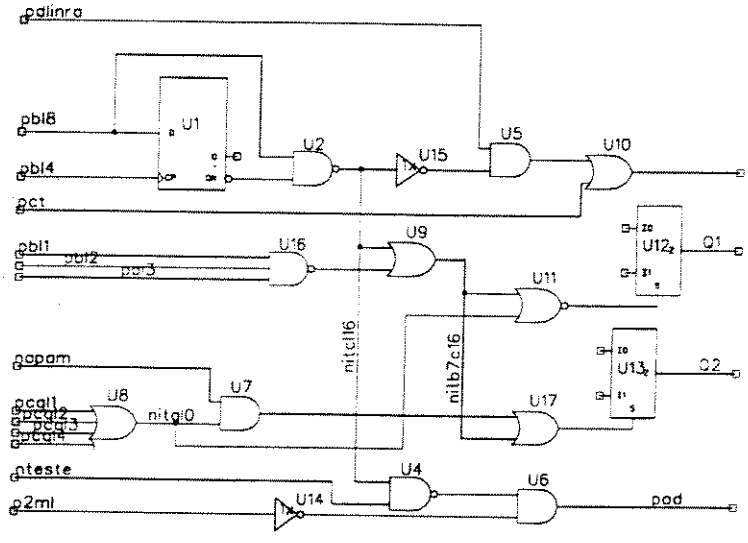

Figura B.1: Diagrama Lógico - Casos de Violações de Regras para Testabilidade

1. O flip flop U1 não está ligado a nenhum "scan".
2. A entrada de relógio PBL4 é ativada por uma saída de um elemento de memória, não sendo, portanto, controlável diretamente por pino de entrada nem mesmo em modo teste.
3. NITCL16 depende de 3 "scans" para ser controlado e / ou observado.
4. POD é um relógio que pode ser aceito por negociação do sistema auditor com o projetista, pois é possível controlá-lo por pino de entrada (P2ML) em modo teste, embora não seja a melhor alternativa de implementação.
5. NITQL0 é gerado a partir de saídas de elementos de memória de um contador não inicializável, ressetado por NRESIN. Este por sua vez não é controlável em modo teste, sendo que em modo funcional (NTESTE = 1), depende de PSQL=1, PAPAM=0 e NAPAMA=0. Estes sinais dependem de entrada serial de dados por dois "scans" implementados no TB15, para serem controlados e para serem observados também dependem de dois "scans" (um dos "scans" usado para observação é o mesmo que é usado para controle o que complica o teste).

6. A maneira adotada para fazer nós do circuito passarem por vários estados foi usando o modo funcional o que resultou em padrão muito longo.

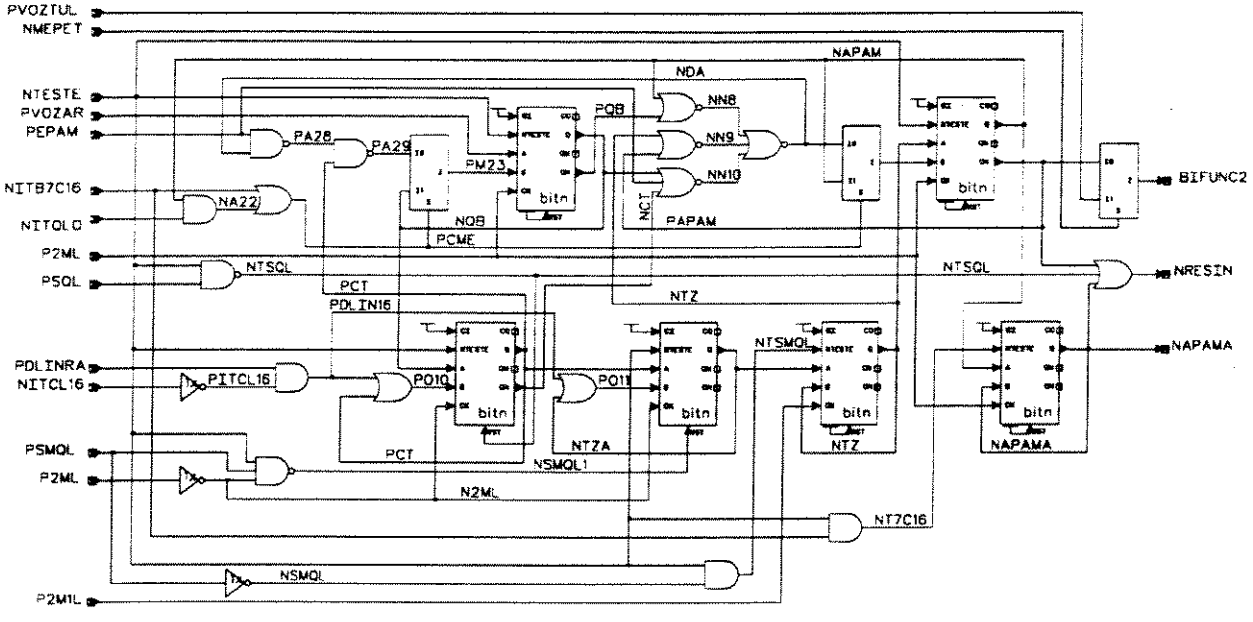

Figura B.2: Diagrama Lógico de Scan com Clocks invertidos: MAQSUP

- A presença de relógios diferentes (invertidos) no mesmo "scan", em modo teste teve como consequência a perda de controlabilidade e observabilidade de nós do circuito. Isso ocorre sempre que houver um número maior do que 2 relógios invertidos ativando elementos de memória conectados sequencialmente no mesmo scan, como na figura B.2, referente ao subcircuito MAQSUP (As células sequenciais de MAQSUP são células BITN cujo diagrama lógico é apresentado na figura B.3). O caminho de teste implementado("scan") tem como entrada scin = PVOZAR e saída scout = NAPAMA, incluindo as células bitn que geram (NQB, PQB), (PCT, NCT), (PCTA, -), (NTZ,-), (PAPAM, NAPAM) e (NAPAMA, -), respectivamente.

  A observabilidade se dá por geração de uma situação diferente da que estava armazenada no FF e pelo seu deslocamento até um pino. Com o uso dessa sequência de relógios invertidos além de ser mais difícil contro-

lar os nós Q das células bitn para valores necessários, a observação fica prejudicada pois 2 delas armazenam sempre o mesmo dado dentro de um ciclo completo.

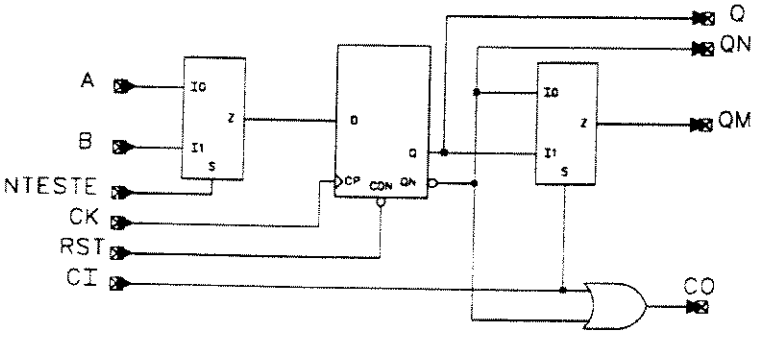

Figura B.3: Célula BITN utilizada em MAQSUP

B.1.2 Exemplo de conflito entre planos de teste

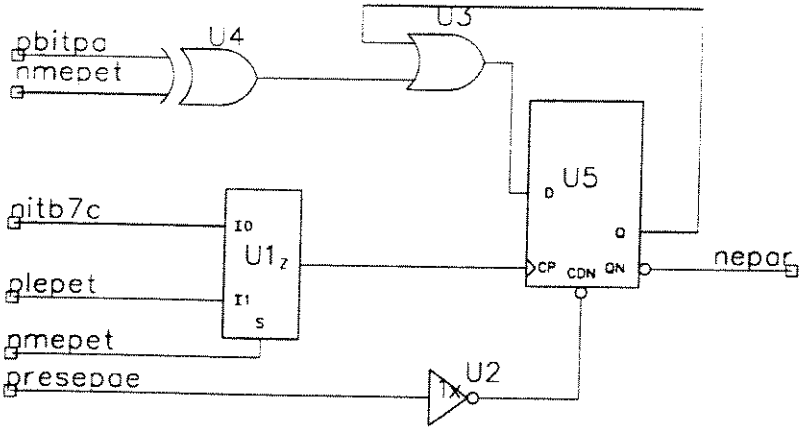

Figura B.4: Diagrama Lógico - Conflitos de Planos de Teste

- O circuito da figura B.4 gera o erro NEPAR que se refere ao erro de paridade de amostras lidas da memória. O FF não faz parte de nenhum

"scan", sendo que essa decisão foi tomada tendo em vista que o NEPAR é pino de saída. Porém, o reset é feito através do pino PRESEPAE, que também é usado no teste da memória. Em modo teste (NTESTE = 0), PRESEPAE = R / W (read/write) da memória e PPCMINE = ME (memory enable). Para resolver esse conflito e gerar teste para esses componentes (OR, MU, FF) é necessário:

1. Escrever e ler em posições determinadas da memória em modo teste, incluindo dados com PFEPARE = 1, isto é, fixando erro de paridade.
2. Ressetar o erro de paridade através de PRESEPAE.
3. Em modo funcional, ler dado da memória que tenha sido escrito com PFEPARE = 1, o que implica em controlar o endereçamento convenientemente através de entrada serial por um "scan", antes de mudar do modo teste para este modo.
4. Ressetar a paridade com PRESEPAE.
5. Ler novamente em modo funcional, mas agora um dado que tenha sido escrito com PFEPARE = 0, controlando convenientemente o seu endereçamento.
6. Repetir o procedimento acima a partir do item 2 para o outro modo funcional normal que é definido por NMEPET, que é o sinal de controle do MU para que esse mux seja testado, bem como a entrada de relógio do FF.

B.2 Sub circuitos do TB34

O projeto deste circuito envolveu o uso de 700 flip flops, sendo que parte deles (há 6 feixes de 84 flip flops) seriam facilmente observados se tivesse sido implementada uma metodologia de projeto com "scan". O circuito inclui várias máquinas de estado e o teste previsto para todo o circuito se baseou em padrão funcional feito pelos projetistas. A única característica de projeto específica para testabilidade inserida no circuito foi a inicialização de todos os flip flops através de pino, o que não resultou suficiente; a depuração do componente exigiu bastante tempo com o uso maciço de micromanipuladora. Entre outros problemas foi detectada a existência de flip flops com preset e reset simultâneos, não apontada como problema na simulação lógica, mas que apresentou características diferentes quando observado com microscópio e micromanipuladora. Esse tipo de implementação seria recusado pelo auditor.

B.3 Sub circuitos do TB14

Houve preocupação com testabilidade neste projeto, o que levou a inclusão de algumas técnicas de projeto especiais tais como a colocação de todos os flip flops em "scans", a possibilidade de testar exaustivamente as PLA's existentes e o uso de geradores pseudo aleatórios. Mesmo assim, esse circuito faz uso de relógios que não são diretamente controláveis por pino em operação normal, isto é são resultado de combinação de sinais internos. Esse tipo de implementação também seria apontado pelo sistema auditor como violação de regra de projeto para testabilidade.

B.4 Conclusões

O fato de conhecer regras básicas para o projeto de circuitos testáveis por "scan" não leva necessáriamente a projetos testáveis. Muitas vezes podem ser infringidas regras de projeto que complicam o teste, fazendo-o altamente dependente do (s) seu (s) modo (s) funcional (is), tornando o padrão muito longo ou extremamente interdependente de funcionamento de várias partes do circuito.

A implementação de um ou mais modos de teste no projeto de um circuito integrado pode levar a uma variedade de planos de teste e a visão completa deles pode ser bastante dificultada. A ajuda de um sistema automatizado pode chegar a ser indispensável à medida que muitos blocos de circuito precisem de modos de teste independentes.

A maneira de obter êxito no projeto com relação ao teste, independente do tipo de estrutura ou estilo de circuito usado, e do número de projetistas envolvidos pode ser a adoção de automatização através de um sistema auditor.

Apêndice C

BIBLIOGRAFIA

Abadir85 M. Abadir e M.A. Breuer, *"A knowledge based system for designing testable VLSI chips"* IEEE design and test of computers, agosto 1985.

Abadir89 M.S. Abadir, *"TIGER: Testability Insertion Guidance Expert System"*, 1989 IEEE International Conference on CAD.

Adhem87 S.M.I.Adhem, H.T.Mouftah e J.I.Glasgow, *"M1-based expert system LSSD rules checker for VLSI circuits"* IEEE Montech'87 - Compint'87.

Agrawal84 V.D. Agrawal, S.K. Jain e D.M. Singer, *"A CAD system for design for testability"* VLSI Design, outubro 1984.

Agrawal88 V.D. Agrawal e S.C. Seth - *"Tutorial: Test generation for VLSI chips"*. IEEE Computer Society Press - 1988.

Aylor86 J.H.Aylor e B.W.Johnson, *"Structured design for testability in semicustom VLSI"* IEEE Micro, fevereiro / 1986.

Baker *"Computer Aided Tools for VLSI system design"* editado por G.Russel, chapter 5: K.Baker, e G. Russel, *"Testing VLSI circuits"* .

Bidjan-Irani91 M. Bidjan-Irani, *"A rule-based design-for-testability rule checker"*. IEEE Design and Test of Computers, março de 1991.

Birmingham86 W. Birmingham, R. Jobbani e J. Kim, *"KBES and their application"* 23rd DAC, 1986.

Breuer76 M.A. Breuer e A. D. Friedman, Eds.Rochville, *"Diagnosis and Reliable design of digital Systems"* Computer Science Press, 1976.

Breuer85 M.A. Breuer e X. Zhu, *"A knowledge based system for selecting a test methodology for a PLA"* 22nd Design Automation Conference, 1985.

Breuer86 M.A.Breuer - *"A methodology for the design of testable VLSI chips"*. January 1986.

Brglez88 F.Brglez, D. Brian, J. Calloun e R. Lisanke, *"A modular Scan-based Testability System"* International Conference on Computer Design, 1988.

Brglez89 F. Brglez, D. Brian, J. Calloun, G. Kedem e R. Lisanke, *"Automated synthesis for testability"* IEEE Transactions on Industrial for Electronics, maio, 1989.

Cabodi86 G. Cabodi, P. Camurati e P. Prinetto, *"The use of PROLOG Specification and Verification of easily testable designs"* Proc. 16th Int'l Symposium on Fault-Tolerant Computing, 1986.

Camurati88 P. Camurati, P. Gianoglio e P. Prinetto, *"ESTA : An expert system for DFT rule verification"* IEEE Transactions on CAD, novembro 1988.

Clocksin84 W.F.Clocksin e C.S. Mellish, *"Programming in PROLOG"* .

Côrtes91 M.L.Côrtes e J.Mendonça F.N., *"Introdução ao teste de CI's digitais"*, V Escola Brasileira Argentina de Informática, 1991.

Cosgrove88 S.J. Cosgrove, N.Burgess e G.Musgrave *"TEXAS : Testing EXpert AdviSor"* IEEE Colloquium on Design for Testability, março 1988.

DAC90 Painel: Estratégias de teste para os 1990 ' s.

eet90 Electronic Engineering Times, february - 1990 - Mentor details 8.0.

Bowen86 D.L.Bowen, l.Byrd, p.W.H.Chung, F.C.N.Pereira, L.M.Pereira, R.Rae, D.H.D.Warren, Manual do Usuário - *"Edinburgh PROLOG"*, Universidade de Edinburgh, Instituto de Aplicações de Inteligência Artificial, 1986.

Eichelberger77 E.B. Eichelberger e T.W. Williams, *"A logic design structure for LSI testability"* 14th Design Automation Conference, 1977.

Fung86 H.S.Fung e S. Hirschhorn, *"An automatic DFT system for the Silc Silicon Compiler"* IEEE Design and Test of Computers, fevereiro 1986.

Fujiwara85 H. Fujiwara, *"Logic Testing and Design for Testability"*, Cambridge, MIT Press 1985.

Gajski83 Daniel D. Gajski e Robert H. Kuhn, " *New VLSI tools"* . Computer - dezembro1983.

gat *"Testscan"* Gateway Design Automation Corp., Sun Microsystems/ Analysis Applications.

Gebotys88 C.H. Gebotys e M.I. Elmasry. *"VLSI design synthesis with testability"* 25th Design Automation Conference, 1988.

Godoy77 H.C. Godoy, C.B. Franklin e P. Bottorff, *"Automatic checking of logic design structures for compliance with testability ground rules"* 14th Design Automation Conference, 1977.

Goering88 R. Goering, *"Logic-synthesis tools forge link between behavior and structure"* Computer Design, junho/1988.

Graf84 M.C. Graf, *"Testing - A major concern for VLSI"* Solid State Technology, janeiro/1984.

Harding89 B. Harding, *"Logic design synthesis forces rethinking of design methods"* , Computer Design, dezembro/1989.

Hayes83 F. Hayes-Roth, D.A.Waterman e D.B. Lenat, *"Building Expert Systems"*, Addison-Wesley Publishing Company, Inc..

Hicks87 P.J.Hicks, *"Semi-custom IC Design and VLSI"*. Capítulo 13 - *"Design for Testability"* 1987.

Horstmann84 P.W. Horstmann, *"A knowledge-based system for using design for testability rules"* Fault-Tolerant Computing Systems Conference, 1984, pp278-284.

Jacob84 G.W. Jacob, *"Designing for Testability"*, Electrical Communication, volume 58, N o 4, 1984.

Jain85 S.K. Jain e V.D. Agrawal, *"Statistical Fault Analysis"* IEEE Design and Test, fevereiro/ 1985.

Jones85 M.A.Jones e K.Baker, *"An intelligent knowledge-based system tool for high level BIST design"* . Proceedings of International Test Conference, 1985.

Kalinowski88 J. Kalinowski e A. Albicki, *"Computer-Aided Design of Self-Testable VLSI circuits"*, COMPEURO 1988.

Kim88 K.Kim, J.G.Tront e D.S.Ha, *"Automatic insertion of BIST Hardware Using VHDL"* 25th ACM/IEEE Design Automation Conference, 1988.

Konemann79 B.Konemann, J. Muncha e G. Zwiehoff, *"BILBO Techniques"* Proc. IEEE International Test Conference, 1979.

Kowalski85 T.J. Kowalski, *"An artificial intelligence approach to VLSI design"* Kluwer Academic Publishers.1985.

Kuban84 J.R. Kuban e J.E. Salick, *"Testing approaches in the MC68020"* VLSI Design, novembro/ 1984.

McCluskey86 E.J.McCluskey, *"Logic Design Principles with emphasis on testable semi-custom circuits"* , Capítulo 10 - *"Design for testability"* .

McCluskey91 E.J.McCluskey, *"Why we need design for testability"*, 1991 International Solid State Circuits Conference.

Muehldorf81 E.I. Muehldorf. *"LSI logic testing - An overview"* IEEE Transactions on Computers, janeiro/ 1981.

Niessen83 C. Niessen, *"Hierarchical design methodologies and tools for VLSI chips"* Proceedings of IEEE, volume 71, N o 1, janeiro/1983.

Peterson72 W.W. Peterson e E.J. Weldon, *"Error correcting codes"*, 2 nd ed., The Colonial Press, Inc., 1972.

Richardson88 S.Richardson, R. Steele e D. Ellsworth, *"AI for ASIC's pinpoints potential problems"* Electronic System Design Magazine, junho/ 1988.

Rubin87 S.M.Rubin - Addison Wesley Publishing Company, 1987 - Computer Aids for VLSI Design.

Sangiovanni87 A. Sangiovanni- Vincentelli e P. Antognetti, *"Síntese de circuitos VLSI"*, Martinus Nijhoff Publishers 1987.

Shadad M.Shadad, *"An overview of VHDL Language Technology"* Proc. 23rd DAC, .

Swan89 G.Swan, Y. Trivedi e D. Wharton, *"Cross-check - A practical solution for ASIC testability"*, 1989 International Test Conference.

VTI - VLSI Technology, Inc. - *"VGT users guide"* e *"ASIC Design Course"* .

Wagner88 F.R. Wagner, I.J. Porto, R.F. Weber, e T.S. Weber, *"Métodos de validação de sistemas digitais"* Capítulo 8 - *"Projeto visando a testabilidade"* . VI Escola de Computação - Campinas-SP.1988

Williams73 M.J.Y. Williams e J.B. Angell, *"Enhancing testability of large scale integrated circuits via test points and additional logic"* IEEE transactions on computers, janeiro/ 1973.

Williams81 T.W. Williams e N.C. Brown, *"Defect-level as a function of fault coverage"* IEEE Transactions on Computers, dezembro/1981.

Williams82 T.W. Williams e K.P. Parker, *"Design for testability: a survey"*, IEEE Transactions on Computers, vol.C31, N o 1, janeiro/1982, páginas de 2 a 15.

Winston P.Winston, *"Artificial intelligence"* 001535win2951.

Yousuf88 N. Yousuf e K.H. Chang, "*An expert system for incorporating design for testability in Programmable Logic Arrays*" IEEE Southeastcon, 1988, p. 6 -10.

Apêndice D

SMAuT: Lista de Programa e Casos Práticos

Neste apêndice são apresentados a listagem de programa protótipo implementado em PROLOG e casos de circuitos aos quais esse protótipo foi aplicado. A ilustração dos casos é dada pelos diagramas lógicos dos circuitos e listagens de descrição dos circuitos seguida dos respectivos relatórios do tipo REL e FALHA emitidos. conforme referidos a seguir:

- MAQSUP: o diagrama lógico é apresentado no Apêndice B. figura B.2. O "netlist" de entrada e os relatórios Violações e Rel aparecem em seguida.
- Para os circuitos CONTMOD7K (figura D.2), INVI (figura D.3) e CONFRA (figura 4.18) não são apresentados os relatórios de Violações por que não foram detectadas violações pelo SMAuT.
- Seguem os diagramas lógicos de CONV (figura D.1), CTRMEM (figura D.4) e COMBRX (figuraD.5). Estes últimos com seus relatórios Violações e Rel e "netlists".
- São apresentados os "netlists" e relatórios Violações e Rel dos circuitos CLKTREE (Ver diagrama lógico: figura 4.8) e CLKNBUF (Ver diagrama lógico: figura 4.9).

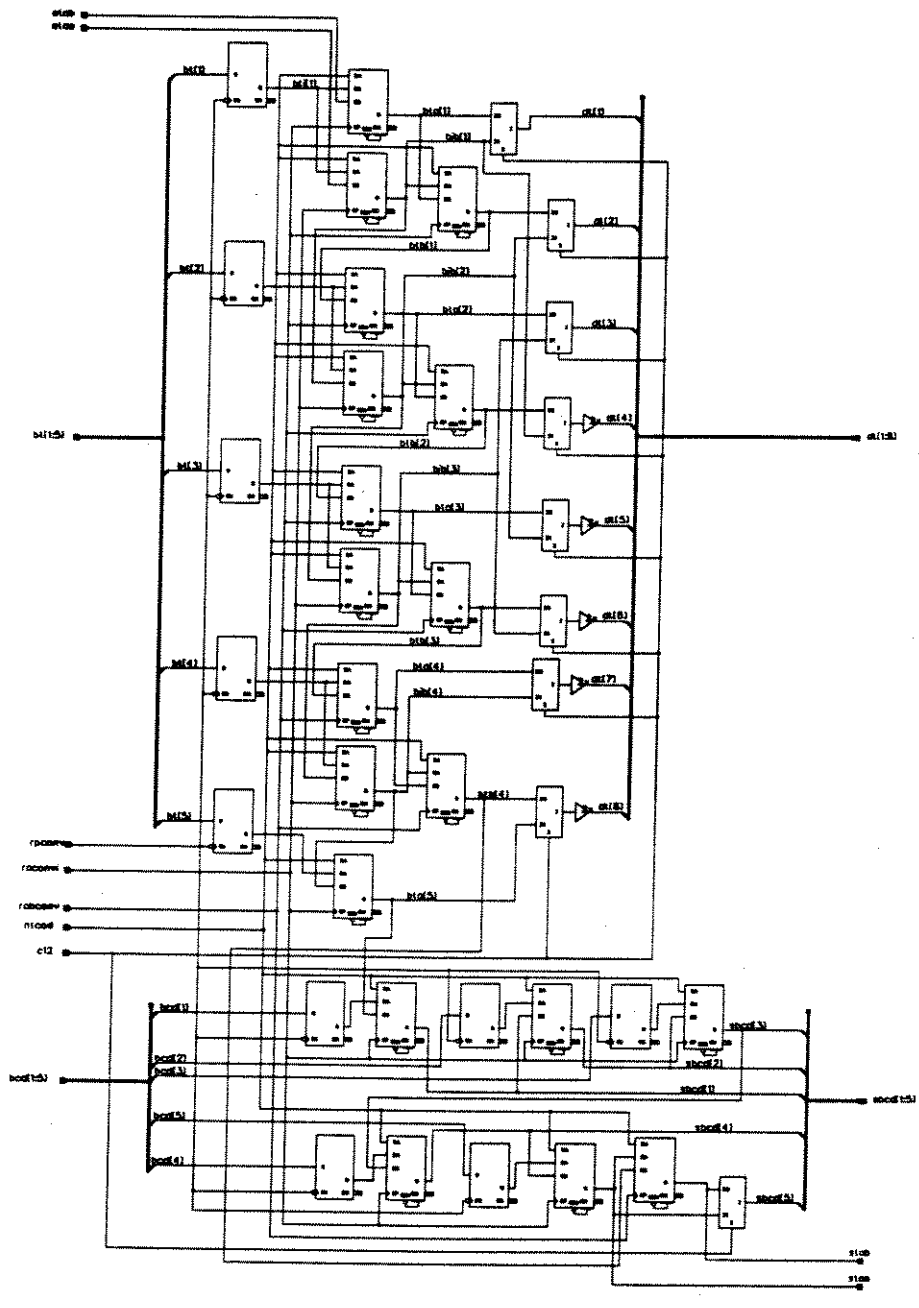

Figura D.1: Diagrama Lógico: CONV

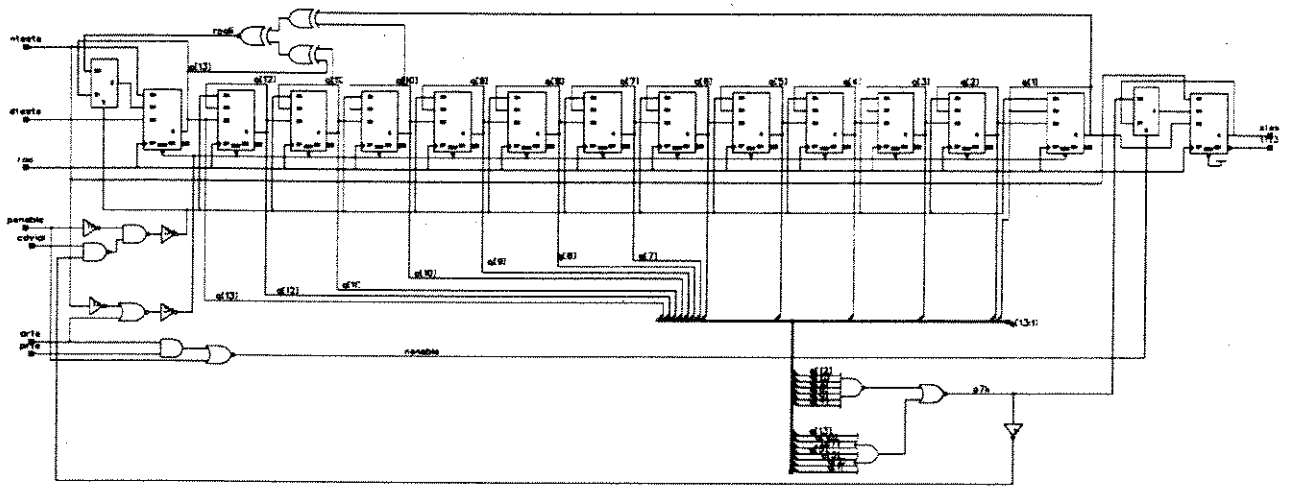

Figura D.2: Diagrama Lógico: CONTMOD7K

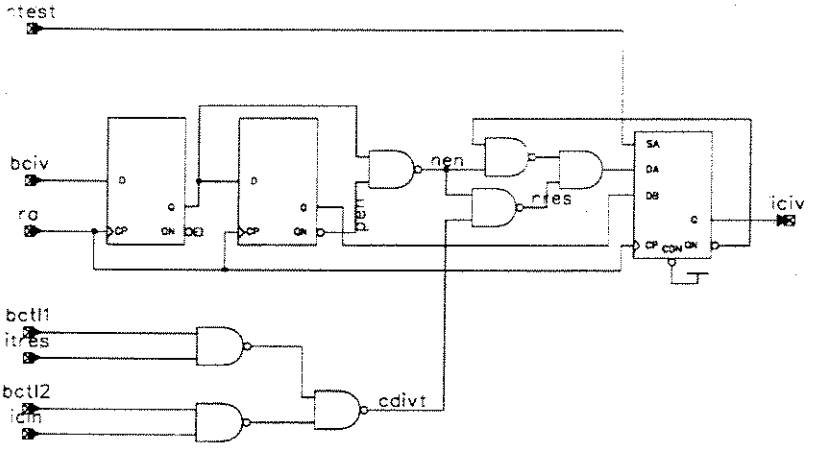

Figura D.3: Diagrama Lógico: INVI

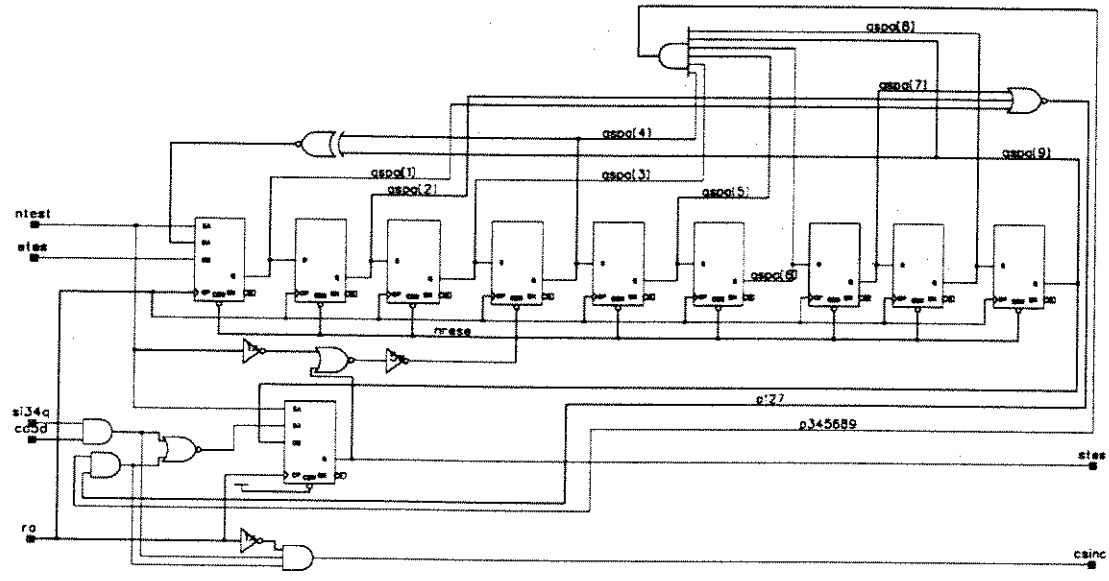

Figura D.4: CONFRA - Caso prático processado

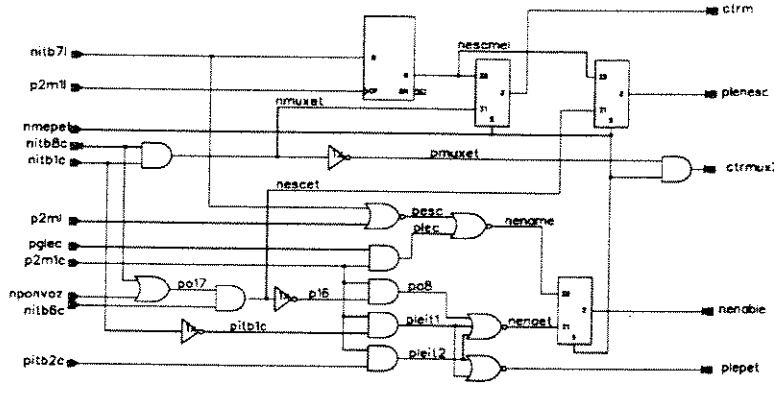

Figura D.5: Diagrama Lógico: CTRMEM

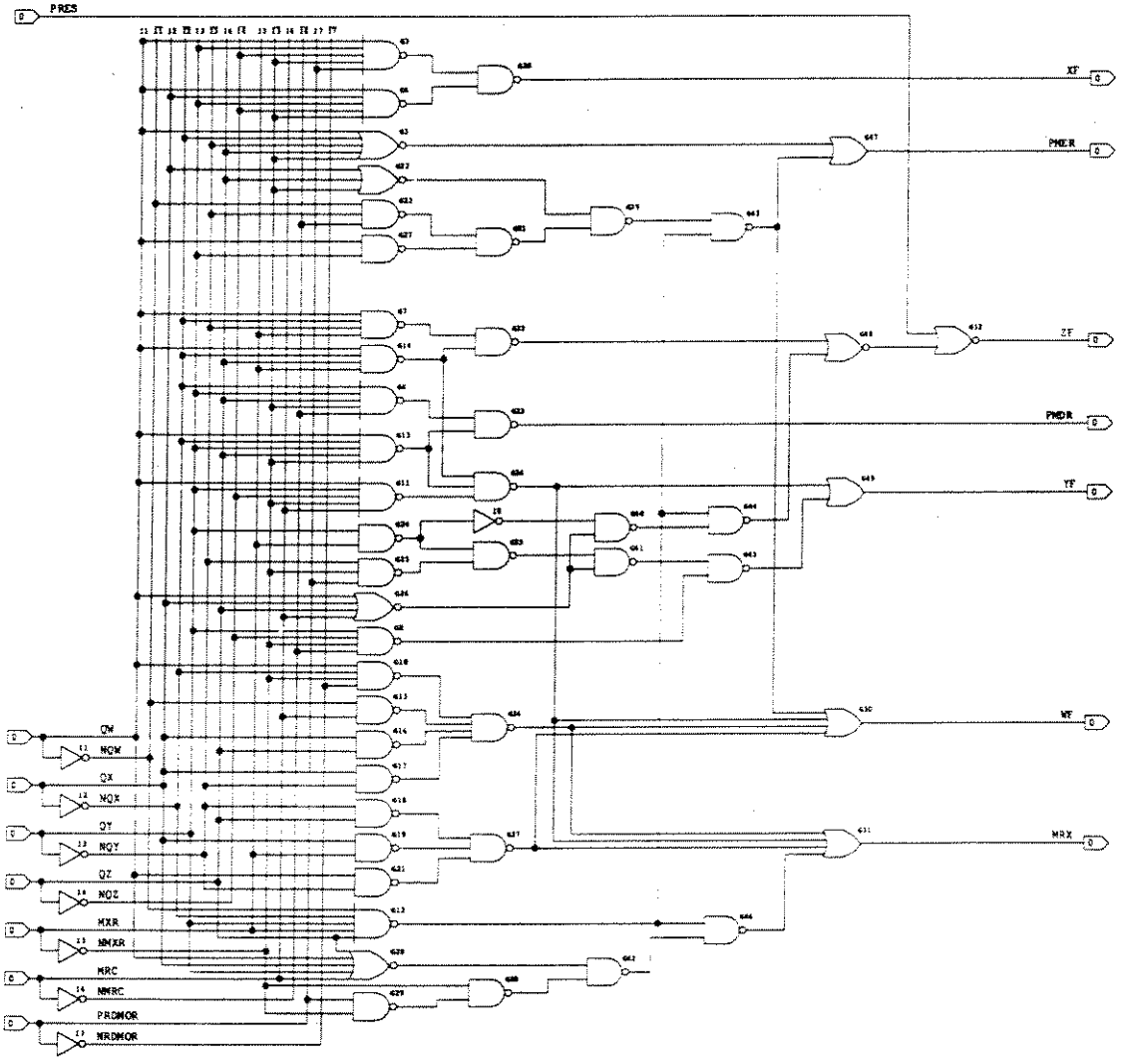

Figura D.5: Diagrama Lógico: COMBRX

```
%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%
Programa - prot^otipo
%%%%%%%%%%%%%%%%%%%%%%
  S M A U T
%%%%%%%%%%%%%%%%%%%%%%

smaut :-
tab(2),
       write('— Interface: dar nome de arquivo de entrada —-'),
       nl,
ini,
tell(user),
tab(2),
       write('— Tradutor: gera arquivo de entrada para o prolog'),
       nl,
tra,
tab(2),
       write('—- Classificador de interconexoes ————————'),
       nl,
cla,
tab(2),write('—- Verificador de regras de testabilidade '),
       nl,
veri,
agregacomb,
tab(2),write('—- Gerador de arquivo de saida ———-'),
       nl,
gera,
told.

%%%%%%%%%%%%%%%%%%%%
  I N T E R F A C E
%%%%%%%%%%%%%%%%%%%%

ini :-
tab(2),write('— Entre nome do CIRCUITO a ser verificado:'),
read(CI),
asserta(ci(CI)),
tab(2),write('- Escolha o tipo de regras a verificar:'),nl,
tab(2),write('- Clock unico ? Resposta: s.(sim) ou n.(nao):'),
       nl,
read(CLU),
asserta(clu(CLU)),
tab(2),write('- Clocks non-overlapping ?
                        Resposta: s.(sim) ou n.(nao):'),
       nl,
read(CKNO),
asserta(ckno(CKNO)),!,
(CKNO == n ->
 (reconsult(def),
  header);
(tab(2),write('— Regras ainda nao implementadas —'),nl)).


%%%%%%%%%%%%%%%%%%%%%%
  T R A D U T O R
%%%%%%%%%%%%%%%%%%%%%%

tra :-
% 'net'pro',
ci(CI),
reconsult(CI).

%%%%%%%%%%%%%%%%%%%%%%
CLA
%%%%%%%%%%%%%%%%%%%%%%

cla :-
tell(user),nl,
tab(2),write('— clacik %piclki,npiclki—'),nl,
clacik,nl,
l(piclki),
l(npiclki),
tab(2),write('— clarst %pirsti,npirsti—'),nl,
clarst,nl,
l(pirsti),
l(npi),
tab(2),write('— clasa %pisamux,npisamux —'),nl,
clasa,nl,
l(pisamux),
l(npi),
tab(2),write('— cla'cone %pini,modo —'),nl,
cla'cone,
l(pini),
tab(2),write('— clanpi %pini,modo —'),nl,
clanpi,nl,
l(pini),
tab(2),write('— claeloff %ffq=ffa -> elo —'),nl,
claeloff1,nl,
claeloff0,nl,
tab(2),write('— claqsai %ffq = pov -> qsai —'),nl,
claqsai,nl,
l(qsai),
tab(2),write('— clacaminho %acha'caminho de pov(LIGA) ->
                                caminho([N—M]) —'),nl,
clacaminho,nl,
tab(2),write('— clacaminhoa %acha caminho sem pov ->
                                caminho([N—M]) —'),nl,
clacaminhoa,nl,
tab(2),write('— claelocomb - elo com MX, TRIBUF ou
                                IOBUF ———-'),nl,
claelocomb,nl,
l(scout),
tab(2),write('— cancel %celulas fora de caminhos(PROC15) ->
                                ncel —'),nl,
cancel,nl,
tab(2),write('— claelomxq %mx'A =ffq ou piv e mx'Q = ffa ->
                                elo—'),nl,
claelomxq,nl,
tab(2),write('— cla'elo'mxa %ffq=mx'A -> elo —'),nl,
cla'elo'mxa,nl,
tab(2),write('— clascania %estrutura de scan e SCIN—-'),nl,
clascania,nl,nl,
l(modo),nl,
l(fscania),
tab(2),write('— clascanout % agrega SCOUT —-'),nl,
clascanout,nl,nl,
l(fscanout),
tab(2),write('— claselsc % agrega SELs e NENs —-'),nl,
claselsc,nl,
listing(scan),nl,
tab(2),write('— pe'clk % pesquisa clocks de scans'),nl.

%%%%%%%%%%%%%%%%%%%%
  C L A C L K
%%%%%%%%%%%%%%%%%%%%

%clock = pino(ou bufferizado) ?
clacik :-
seq(NP,',N,',',"c-C"P,','),
abolish(conta,2),
abolish(conta'in,2).
asserta(conta(0,1)),
asserta(conta'in(0,1)),
(piclk(CP,R) -> % P -> Q;R.
 (conta(I,'),
  conta'in(IN,'),
  asserta(piclki(N"c-C"P,R,I,IN)));
 asserta(npiclki(NP,N"c-C"P))),
tell(user),
tab(2),write([N"c-C"P,R,I,IN]),nl,
       fail.
clacik.

%%%%%%%%%%%%%%%%%%%%
piclk(CK,R):-
piclki('"c-C"K,R,I,IN) ->
(conta(IPI,'),
 conta'in(INI,'),
 incr(conta'in(INI,IN)),
 incr(conta(IPI,I)),
 true);
((((ibuf1(N"c-C"K,E),%N = instancia do input - buffer
   conta(I,'),
   incr(conta(I,1)),
   R = E);
  (pi(L),
   membro(CK,L),
   R = CK);
  (buf1(N"c-C"K,E),
   conta(I,'),
   incr(conta(I,1)),
   piclk(E,R)),
  (nbuf(N"c-C"K,E),
   conta'in(IN,'),
   incr(conta'in(IN,1)),
   piclk(E,R)))).

%%%%%%%%%%%%%%%%%%%%
%reset = piv(pino ou bufferizado) ?
clarst :-
seq(NP,',N,',',',"c-C"DN,'),
(pirst(NP"c-C"DN,R)-> % P->Q;R.
 asserta(pirsti(NP,N"c-C"DN,R));
 asserta(npi(NP,N"c-C"DN))).
       write([N"c-C"DN,R]),nl,
fail.
clarst.

%%%%%%%%%%%%%%%%%%%%
pirst(NP"c-C"K,R):-
pirsti(NP,'"c-C"K,R) -> %NP = polaridade ativa do p/reset
true;
piv('"c-C"K,R).

%%%%%%%%%%%%%%%%%%%%
piv(N,X,E) :- %X = sinal
(pivi(NP,N,X,E) ->
 true; %E = entrada de buffer
 (ibuf1(N,X,E); %N = instancia do input - buffer
  (pi(L),
   membro(X,L),
   E = X);
  (buf1('X,X,XE),
   piv(N,XE,E))).
```

```
  asserta(pivi(NP,N,X,E))).

%%%%%%%%%%%%%%%%%%%%
%SA de seq = piv ?
clasa :-
mxffd(NP,N,',',',',SA,',').
(piv(',SA,E) -> % P->Q;R.
(asserta(pisamux(N,SA,E)),
 write([N,SA,E]),nl,
 fail);
 asserta(npi(NP,N,SA))),
tab(2),write([N,SA]),nl,
fail.
clasa.

%%%%%%%%%%%%%%%%%%%%
 CLA-CONE
%%%%%%%%%%%%%%%%%%%%

cla'cone:-
modo'scan(LSC),
membro([SIN,NIV],LSC),
nl,write([SIN,NIV]),nl,
cone'comb(SIN,NIV,SIN,NIV),
nl,write([SIN,NIV]),nl,
fail.
cla'cone.

cone1(Q,SIN):-
(buf(',Q,SIN);
 (and(',Q,L),
  membro(SIN,L))).
cone2(NQ,SIN):-
(nbuf(',NQ,SIN);
 (nand(',NQ,NL),
  membro(SIN,NL))).
cone3(OQ,SIN):-
(buf(',OQ,SIN);
 (or(',OQ,OL),
  membro(SIN,OL))).

cone4(NOQ,SIN):-
(nbuf(',NOQ,SIN);
 (nor(',NOQ,NOL),
  membro(SIN,NOL))).


%%%%%%%%%%%%%%%%%%%%
cone'comb(SINI,NIVI,SIN,NIV):-
((igual(NIV,0)) ->
(write(NIV),nl,
 (setof(Q,Q'cone1(Q,SIN),LQ),
write('LQ'),tab(2),write(LQ),nl,
%%%length(LQ,LLQ),
  membro(SAI,LQ),
  write('NIV0bufand'),tab(2),write(NIV),nl,
  NSAI = 0,
  NP = p,
  (cone'seqmx(NP,SAI,NSAI,SINI,NIVI),!;
   (cone'comb(SINI,NIVI,SAI,NSAI),
    write([SINI,NIVI,SAI,NSAI]),nl)),
write('******************'),write('SAI'),tab(2),write(SAI),nl,
');
(setof(NQ,NQ'cone2(NQ,SIN),LQ2),
write('LQ2'),tab(2),write(LQ2),nl,
 membro(SAI,LQ2),
 write('NIV0nbufnand'),tab(2),write(NIV),nl,
 NSAI = 1,
 NP = n,
 (cone'seqmx(NP,SAI,NSAI,SINI,NIVI),!;
  (cone'comb(SINI,NIVI,SAI,NSAI),
   write([SINI,NIVI,SAI,NSAI]),nl)),
write('******************'),write('SAI'),tab(2),write(SAI),nl,
');
 (write(NIV),nl,
  (setof(OQ,OQ'cone3(OQ,SIN),LQ3),
write('LQ3'),tab(2),write(LQ3),nl,
   membro(SAI,LQ3),
   write('NIV1bufor'),tab(2),write(NIV),nl,
   NSAI = 1,
   NP = n,
   (cone'seqmx(NP,SAI,NSAI,SINI,NIVI),!;
    (cone'comb(SINI,NIVI,SAI,NSAI),
     write([SINI,NIVI,SAI,NSAI]),nl)),
write('******************'),write('SAI'),tab(2),write(SAI),nl,
');
  (setof(NOQ,NOQ'cone4(NOQ,SIN),LQ4),
write('LQ4'),tab(2),write(LQ4),nl,
      membro(SAI,LQ4),
   write('NIV1nbufnor'),tab(2),write(NIV),nl,
   NSAI = 0,
   NP = p,
  (cone'seqmx(NP,SAI,NSAI,SINI,NIVI),!; %.!
   (cone'comb(SINI,NIVI,SAI,NSAI),
    write([SINI,NIVI,SAI,NSAI]),nl)),
write('******************'),write('SAI'),tab(2),write(SAI),nl,
!)))).

%%%%%%%%%%%%%%%%%%%%
cone'seqmx(NP,SAI,NSAI,SINI,NIVI):-
 ((seq(NP,',I,',',',SAI,'),!;
   seq(',',I,',',',',SAI),!;
   (mx(',',',SEL),
    membro(SAI,SEL))),
  (pini(SAI,SINI,NSAI,NIVI) ->
   (cone'comb(SINI,NIVI,SAI,NSAI),
%%%REPETE MUITAS VEZES
    write([SINI,NIVI,SAI,NSAI]),nl),
   (asserta(pini(SAI,SINI,NSAI,NIVI)),
    write(pini(SAI,SINI,NSAI,NIVI)),
    write(SAI)
   "c-c"one'comb(SINI,NIVI,SAI,NSAI)
),nl)).


%%%%%%%%%%%%%%%%%%%%
  CLANPI
%%%%%%%%%%%%%%%%%%%%

%(reset ou select) = inibido por modo?
%modo dado por pini(sinais,niveis correspondentes) == >>
clanpi :-
(npi(NP,N,Q),
 pin(NP,N,Q)),
fail.
clanpi.

%%%%%%%%%%%%%%%%%%%%
   PIN
%%%%%%%%%%%%%%%%%%%%

pin(NP,N,Q):-
barra(NP,NPB),
pini(Q,',NPB,') ->
true;
 (((nbuf(KCO,Q,A),
    not(piclki(',A,',',')),
    (piv(KBU,A,E);
     (buf1(',A,AI),
      piv(',AI,E))));
   (nand(KCO,Q,L),
    (membro(A,L),
    not(piclki(',A,',',')),
     (piv(KBU,A,E);
      (buf1(',A,AI),
       piv(',AI,E))))));
  asserta(pini(Q,E,1,0)));
 (or(KCO,Q,L),
  (membro(A,L),
   not(piclki(',A,',',')),
   (piv(KBU,A,E);
    (buf1(',A,AI),
     piv(',AI,E)))),
 asserta(pini(Q,E,1,1)));
 (and(KCO,Q,L),
  (membro(A,L),
   not(piclki(',A,',',')),
   (piv(KBU,A,E);
    (buf1(',A,AI),
     piv(',AI,E)))),
 asserta(pini(Q,E,0,0)));
 (nor(KCO,Q,L),
  (membro(A,L),
   not(piclki(',A,',',')),
   (piv(KBU,A,E);
    (buf1(',A,AI),
     piv(',AI,E)))),
  asserta(pini(Q,E,0,1)));
tab(2),write([N,Q,KCO,A,E,KBU]),nl,!.
pin(',',').

%%%%%%%%%%%%%%%%%%%%
%saida de ff = entrada de outro ?
claeloff1 :-
(seq(',',N,Q,',"c-C"K,RST,SA),
 ffq(N"c-C"K,RST,SA),
 seq(NP,',M,',Q,"c-C"KM,RSTM,SAM),
 M "== N,
 ffa1(NP"c-C"KM,RSTM,M,SAM)),
asserta(elo([N,M])),
tab(2),write([N,M]),nl,
fail.
claeloff1.

claeloff0:-
(seq(',',N,Q,',"c-C"K,RST,SA),
 ffq(N"c-C"K,RST,SA),
 seq(NP,',M,',Q"c-C"KM,RSTM,SAM),
 M "== N,
 ffa0(NP"c-C"KM,RSTM,M,SAM)),
asserta(elo([N,M])),
tab(2),write([N,M]),nl,
fail.
claeloff0.

barra(NP,NPB):-
```

```
((NP == vdd,
  NPB = vss);
 (NP == n,
  NPB = 1);
 (NP == p,
  NPB = 0);
 (NP == 1,
  NPB = 0);
 (NP == 0,
  NPB = 1)).

%%%%%%%%%%%%%%%%%%%%
%ffa = ffda ; ffdb
ffa1(NP"c-C"K,RST,M,SA):-
barra(NP,NPB),
piclk(CK,'),
(igual(SA,vdd);
 pini(SA,',1,');
 (pisamux(M,SA,E),
  asserta(pini(SA,E,1,1)))),
(igual(RST,vdd);
 pini(RST,',NPB,');
%%%%desativa preset ou reset ativo baixo ou alto
 pirsti(NP,',RST,')),!.

ffa0(NP"c-C"K,RST,M,SA):-
barra(NP,NPB),
piclk(CK,'),
(igual(SA,vss);
 pini(SA,',0,');
 (pisamux(M,SA,E),
  asserta(pini(SA,E,0,0)))),
(igual(RST,vdd);
 pini(RST,',NPB,');
%%%%desativa preset ou reset ativo baixo ou alto
 pirsti(NP,',RST,')),!.

ffq(N"c-C"K,RST,SA):-
 piclk(CK,'),
 (igual(RST,vdd);
   pirsti(',',RST,');
   pini(RST,',1,')),
  (igual(SA,vdd);
   igual(SA,vss);
   pisamux(N,SA,');
   pini(SA,',',')),!.
igual(X,X).

%%%%%%%%%%%%%%%%%%%%
%entrada de mux = (saida de FF ) ?
cla'elo'mxa :-
((mx(N,Q,E,S),
  pov(',Q,'),
  not(todos(S))),
% == todos os membros de S sao piv
 membro(A,E),
 ((seq(',',M,A,',',"c-C"K,RST,SA),
   ncel(',M),
   ffq(M"c-C"K,RST,SA)))),
(asserta(nelo([M,N])),
 write([N,A,M]),nl),
fail.
cla'elo'mxa.

todos(S):-
membro(SEL,S),
(not(piv(',SEL,'))) ->
write(SEL),nl,
true;
fail.

%%%%%%%%%%%%%%%%%%%%
entrada de mux = (piv ou ffq) e saida de mux = ffa ?
claelomxq :-
((mx(N,Q,',S),
  not(todos(S))),
% == todos os membros de S sao piv
 ((seq(NP,',MA,',Q,',"c-C"K,RST,SA),
   (M "== N ->
    ffa1(NP"c-C"K,RST,M,SA);
    fail));
  (seq(NP,',MA,',',Q,"c-C"K,RST,SA),
   (M "== N ->
    ffa0(NP"c-C"K,RST,M,SA);
    fail)))),
asserta(melo([N,MA])),
       write([N,MA]),
tab(2),write(melo([N,MA])),tab(2),
write('pode ser substituido por um MXFFD'),nl,
fail.
claelomxq.

%%%%%%%%%%%%%%%%%%%%
  CLAQSAI
%%%%%%%%%%%%%%%%%%%%

%destino = pov ?
claqsai :-
seq(',',N,Q,',',',',');
pov(',Q,X),
asserta(qsai(N,Q,X)),
tab(2),write([N,Q,X]),nl,
fail.
claqsai.

%%%%%%%%%%%%%%%%%%%%
   CLACAMINHO
%%%%%%%%%%%%%%%%%%%%

%acha caminho de pov.
clacaminho :-
qsai(M1,',',),
liga([',M1]),
fail.
clacaminho.

%%%%%%%%%%%%%%%%%%%%
%liga elos de pov.
liga([N1—M1]) :-
elo([N1—M1]),
(elo([X—Y]),
 last(N1,Y)),!,
append([X—Y],M1,[X—W]),
asserta(elo([X—W])),
retract(elo([N1—M1])),
retract(elo([X—Y])),
liga([X—W]),tab(2),write([X,W]),nl,
length([X—W],LE),
asserta(caminho([X—W],LE)),
tab(2),write(caminho([X—W],LE)),nl,
retract(elo([X—W])),
fail.
liga(').

%acha caminho sem pov.
clacaminhoa :-
elo([N1—M1]),
nliga([N1—M1]),
fail.
clacaminhoa.

%liga elos sem pov.
nliga([N1—M1]) :-
(elo([X—Y]),
 last(N1,Y)),!,
(acerta(([X—Y]),M1,([X—W])),
 nliga([X—W]),
 pliga([X—W]),
length([X—W],LE),
 asserta(caminho([X—W],LE)),
 retract(elo([X—W])),
 write(caminho([X—W],LE)),nl),
fail.
nliga(').

pliga([N1—M1]) :-
elo([X—Y]),
%%%%% membro(X,M1),!,
last(X,M1),!,
(acerta(([N1—M1]),Y,([N1—W])),
%%%?????pliga([N1—M1]),
length([N1—W],LE),
 asserta(caminho([N1—W],LE)),
 retract(elo([N1—W])),
 write(caminho([N1—W],LE)),nl),
fail.
pliga(').

acerta(([X—Y]),M1,([X—W])) :-
append([X—Y],M1,[X—W]),
asserta(elo([X—W])),
retract(elo([—M1])),
retract(elo([X—Y])),
tab(2),write(elo([X—W])),nl.

%%%%%%%%%%%%%%%%%%%%
%elo com MX. TRIBUF. IOBUF
claelocomb :-
caminho([N—M],LE),
 last(X,M),
 seq(',',X,Q,',',',',');
 ((mx(I,QI,L,'),
   membro(Q,L),
   pov(',QI,')),
  (((tribuf(NP,I,QI,Q,S),
     pov(',QI,')),
    iobuf(NP,I,QI,',Q,S)),
   (piv(',S,');
    pini(S,',0,')))),
append([N—M],[I],[N—W]),
asserta(scout(NP,I,QI)),
tab(2),write(I),nl,
tab(2),write(scout(NP,I,QI)),nl,
asserta(caminho([N—W],LE)),
retract(caminho([N—M],LE)),
tab(2),write(caminho([N—W],LE)),nl.
```

```
fail.
claelocomb.


%%%%%%%%%%%%%%%%%%%%%
  CLASCANIN
%%%%%%%%%%%%%%%%%%%%%

%verifica e gera falha: acha scanin e scanout de caminho
%       ou fscanin e fscanout
clascanin :-
abolish(conta,2),
asserta(conta(0,')),
caminho([N—M],LE),
conta(I,'),
incr(conta(I,1)),
seq(',',N,',A,B"c-C"K,RST,SA),
asserta(scan(I,LE,[CK],[RST],[SA],[N—M],',')),
cla'sc'scin(I,N,A,B),
fail.
clascanin.

incr(conta(I,INC)):-
X is I + INC,
abolish(conta,2),
asserta(conta(X,INC)).
incr(conta'in(I,INCI)):-
X is I + INCI,
abolish(conta'in,2),
asserta(conta'in(X,INCI)).

%%%%%%%%%%%%%%%%%%%%%
%fscin
cla'sc'scin(I,N,A,B):-
scan(I,L,[CK],[RST],[SA],[N—M],SCIN,'),
(scin(I,KBU,A,E),
 (retract(scan(I,L,[CK],[RST],[SA],[N—M],SCIN,')),
  SCIN = [E,KBU,A],
  asserta(scan(I,L,[CK],[RST],[SA],[N—M],SCIN,'))),
 tab(2),write(scan(I,L,[CK],[RST],[SA],[N—M],SCIN,')),nl,!);
(B "== vdd,
 scin(I,KBU,B,BE),
 (retract(scan(I,L,[CK],[RST],[SA],[N—M],SCIN,')),
  SCIN = [BE,KBU,B],
  asserta(scan(I,L,[CK],[RST],[SA],[N—M],SCIN,'))),
 tab(2),write(scan(I,L,[CK],[RST],[SA],[N—M],SCIN,')),nl,!);
(asserta(modo(I,[])),
 asserta(fscanin(I,N,A,B)),
!).

scin(I,KBU,A,E):-
(piv(KBU,A,E),
 asserta(modo(I,[])),!;
(tribuf(NP,KBU,A,E,NEN),
 piv(',E,'),
 barra(NP,NPB),
 asserta(modo(I,[NEN,NP]))),!;
(iobuf(NP,KBU,E,A,',NEN),
 piv(',E,'),
 asserta(modo(I,[NEN,NPB]))),!.

%%%%%%%%%%%%%%%%%%%%%
  CLASCANOUT
%%%%%%%%%%%%%%%%%%%%%

%fscout
clascanout:-
scan(I,',',',[',—M],',SCOUT),
last(X,M),
tab(2),write(X),nl,
(seq(',',X,Q,',',',',',
 mx(X,Q,',');
 iobuf(NP,X,Q,',',');
 tribuf(NP,X,Q,',')),
cla'sc'scout(I,SCOUT,X,Q),
fail.
clascanout.

%%%%%%%%%%%%%%%%%%%%%
cla'sc'scout(I,SCOUT,X,Q):-
pov(KBU,Q,QS) ->
(retract(scan(I,L,[CK],[RST],[SA],[N—M],SCIN,SCOUT)),
 SCOUT = [Q,QS],
 (nonvar(KBU) ->
  append([N—M],[KBU],KS);
  KS = [N—M]),
 asserta(scan(I,L,[CK],[RST],[SA],KS,SCIN,SCOUT)),
 write(scan(I,L,[CK],[RST],[SA],KS,SCIN,SCOUT)),nl),!;
(SCOUT = [X,Q],
asserta(fscanout(I,X,Q)),
tell(user),
retract(scan(I,L,[CK],[RST],[SA],[N—M],SCIN,SCOUT)),
asserta(scan(I,L,[CK],[RST],[SA],[N—M],SCIN,SCOUT)),
tab(2),
write(scan(I,L,[CK],[RST],[SA],[N—M],SCIN,SCOUT)),nl),!.

%%%%%%%%%%%%%%%%%%%%%
  CLASELSC
%%%%%%%%%%%%%%%%%%%%%

%acrescenta CK's, RST's e SA's
claselsc:-
scan(I,L,',',',[N—M],',),
length([N—M],LE),
abolish(conta'in,2),
mesmo(I,L,LE),
fail.
claselsc.

%%%%%%%%%%%%%%%%%%%%%
mesmo(I,L,LE):-
asserta(conta'in(2,1)),
repeat.
conta'in(IN,'),
incr(conta'in(IN,1)),
ascan(IN,I),
IN =:= L,!,
LE > L ->
(repeat.
IX is L + 1,
incr(conta'in(IX,1)),!,
ascani(IX,I),
IX =:= LE,!).

%%%%%%%%%%%%%%%%%%%%%
%agrega CK's, RST's e SELECT's
ascan(IN,I):-
scan(I,L"c-C",R,S,[N—M],SCIN,SCOUT),
nth1(IN,[N—M],IM),
seq(',',IM,',',"c-C"K,RST,SA),
append(C,[CK]"c-C"KF),
sort(CKF"c-C"KFI),
append(R,[RST],RSTF),
sort(RSTF,RSTFI),
append(S,[SA],SAF),
sort(SAF,SAFI),
retract(scan(I,',',',',',',')),
asserta(scan(I,L"c-C"KFI,RSTFI,SAFI,[N—M],SCIN,SCOUT)),!.
ascan(',').

%%%%%%%%%%%%%%%%%%%%%
%acrescenta SEL's de MX's, enables de TRIBUF's e IOBUF's
ascani(IN,I):-
scan(I,L"c-C",R,S,[N—M],SCIN,SCOUT),
nth1(IN,[N—M],IM),!,
(mx(IM,',',SA);
 ((iobuf(NP,IM,',',',SI);
   tribuf(NP,IM,',',SI)),
  SA = [SI],
  asserta(nivel(I,IM,SI,NP)),
  write(nivel(I,IM,SI,NP)),nl)),
append(S,SA,SAF),
sort(SAF,SAFI),
retract(scan(I,',',',',',',')),
asserta(scan(I,L"c-C",R,SAFI,[N—M],SCIN,SCOUT)),!.
%tab(2),
%write(scan(I,L"c-C",R,SAFI,[N—M],SCIN,SCOUT))
%%%,!.
ascani(',').

%%%%%%%%%%%%%%%%%%%%%
  CLANCEL
%%%%%%%%%%%%%%%%%%%%%

%acha celulas fora de caminhos.
clancel'b(CEL,I) :-
(((caminho(W,')), %;
    %elo(W)),
  membro(I,W)}->
 fail;
 (asserta(ncel(CEL,I)),
  write(ncel(CEL,I)))).
clancel'b(',').

clancel :-
%cel(TIPO,INST,')
seq(',TIPO,INST,',',',',',')
clancel'b(TIPO,INST),
fail.
clancel.

%%%%%%%%%%%%%%%%%%%%%
  VERI
%%%%%%%%%%%%%%%%%%%%%

veri:-
pe'clk,nl,
tab(2),write('— pe'rst % pesquisa reseta de scans'),nl,
pe'rst,nl,
l(modo),nl,
tab(2),write('— pe'sa % pesquisa selecao de scans'),nl,
pe'sa,
l(modo),nl,
l(nivel),nl,
veri'dado'nclk,
l(falha'dado'nclk),
```

```
l(falha'dado'nclkx).

%%%%%%%%%%%%%%%%%%%%%%%%%%
  VERI-DADO-NCLK
%%%%%%%%%%%%%%%%%%%%%%%%%%

%verifica entrada de dado de seq nao depende de clock
veri'dado'nclk:-
veri'dado'nclka,
seq(',',N,',DA,DB"c-C"K,',SA),
veri'dado(N,DA"c-C"K),
(DB "== vdd,
 veri'dado(N,DB"c-C"K)),
(SA "== vdd,
 veri'dado(N,SA"c-C"K)),
fail.
veri'dado'nclk.

%%%%%%%%%%%%%%%%%%%%%%%%%%
  VERI-DADO-NCLKA
%%%%%%%%%%%%%%%%%%%%%%%%%%

% para verificar entrada de dados quanto a qualquer sinal de clock
veri'dado'nclka:-
clock(LCLK),
membro(XC,LCLK),
abolish(conta,2),
abolish(conta'in,2),
asserta(conta(0,1)),
asserta(conta'in(0,1)),
(piclk(XC,R) -> % P -> Q;R.
 (conta(I,'),
  conta'in(IN,'),
  asserta(piclki(N,XC,R,I,IN))),
 asserta(npiclki(NP,N,XC))),
tell(user),
tab(2),write([N,XC,R,I,IN]),nl,
fail.
veri'dado'nclka.

%%%%%%%%%%%%%%%%%%%%%%%%%%
veri'dado(N,DA"c-C"K):-
(comb(',',DA,LA,SELA), %%% LA = lista de entrada
 (veri'ck(CK,DA,LA,SELA,N); %%% SELA = lista de selecao
  veri'ckx(DA,LA,SELA,N),!)).
veri'dado(',',').

%%%Entrada ou Selecao de comb que gera DA depende de CK
veri'ck(CK,DA,LA,SELA,N):-
(CK == LA:
  membro(CK,LA);
  CK == SELA: %%% CK depende de selecao do comb
  membro(CK,SELA)), %%%
 asserta(falha'dado'nclk([N,DA"c-C"K])),!.

%%%Entrada ou Selecao de comb que gera DA depende de algum CK
veri'ckx(DA,LA,SELA,N):-
((membro(X,LA);
   membro(X,SELA)),
  piclki(',X,X))),
asserta(falha'dado'nclkx([N,DA,X,X])),!.

%%%%%%%%%%%%%%%%%%%%%%%%%%
  PECLK
%%%%%%%%%%%%%%%%%%%%%%%%%%

% pesquisa CLK's dos scan's
pe'clk:-
scan(I,',[C—K],',',',',');
length([C—K],L),
L =:= 1,
file(CIFAL),
tell(falhas),nl,nl,nl,
tab(2),write(' ###  Lista de sinais de clock verificada:'),nl,
tab(10),write('SCAN('),write(I),write(') : '),tab(2),write([C—K]),nl,
tell(user),
(abolish(conta,2),
 peclk(L,[C—K])),
fail.
pe'clk.

peclk(L,[C—K]):-
asserta(conta(2,1)),%%%%tell(user),nl,write(conta(2,1)),nl,
repeat,
conta(I,'),
incr(conta(I,1)),
pes(I,[C—K]),
I = L,!.
peclk(',').

pes(I,[C—K]):-

nth1(I,[C—K],IC),!,
(peclki(IC"c-C") ->
 (write(IC),nl);
 (file(CIFAL),
  tell(falhas),nl,
  tab(2),
  write(' ###### FALHA = DIFERENTES PINOS DE CLOCK PARA:'),
   tab(2),write(IC),tab(2),write(C),nl,tell(user))),
pes(',').

peclki(C,IC):-
 piclki(C"c-C",R,I,IN),
 piclki(',IC,R,I1,IN1),
 tell(falhas),nl,
((IN =:= IN1,
  I =:= I1,
  tab(2),write(' ### clocks OK :'),
         tab(2),write(C),tab(2),write(IC),nl);
 (fase(IN,IN1),
  (write(' ### mesma fase e possivel distorcao de clock:'),
nl,
   tab(2),write('Pino associado aos sinais de clock:'),
write(R),nl,
   nl,tab(2),write('Para o sinal '),write(C),
         write(', numero de buffers: '),write(I),
   nl,tab(2),write('e para o sinal '),write(IC)
         write(', numero de buffers:'),write(I1),nl);
  (write(' ### ERRO ###: diferentes fases de clock no caminho'),
nl,
   tab(2),write('Pino associado aos sinais de clock:'),
write(R),nl,
   tab(2),write('Para o sinal '),write(C),tab(2),
         write(', numero de buffers inversores: '),write(IN),
tab(2),
   nl,tab(2),write(' e para o sinal '),write(IC),
         write(', numero de buffers inversores:'),
write(IN1),nl))),
tell(user).

fase(IN,IN1):-
%%%%tell(user),nl,write(IN),tab(2),write(IN1),nl,
IQ is IN / 2, %%%%nl,write(IQ),tab(2),
IQB = IN // 2, %%%%write(IQB),tab(2),
IQ1 = IN1 / 2, %%%%write(IQ1),tab(2),
IQB1 = IN1 // 2, %%%%write(IQB1),tab(2),
(IQ =:= IQB, %
 IQ1 =:= IQB1); %numero par de inversoes
(IQ  =\= IQB, %%
 IQ1 =\= IQB1). %%numero impar de inversoes


%%%%%%%%%%%%%%%%%%%%%%%%%%
%pesquisa RST's dos scan's
pe'rst:-
scan(IS,',',RST,',',',',');
delete(RST,vdd,[R—ST]),
pri'rst(IS,R,NR),
length([R—ST],L),
L > 1,
abolish(conta,2),
outro'rst(IS,L,[R—ST],NR),
fail.
pe'rst.

pri'rst(IS,R,NR):-
modo(IS,LS),
((pini(R,AR,NR,NAR),
  append(LS,[rst(R,AR,NR,NAR)],LNS),
  sort(LNS,LN),
  write(LN),nl,!);
 (pirsti(NP,',R,RE),
  append(LS,[rst(NP,R,RE)],LNS),
  sort(LNS,LN),
  NR = 1,
  write(LN),nl,!)),
retract(modo(IS,LS)),
asserta(modo(IS,LN)),
write(modo(IS,LN)),nl,!.
pri'rst(',',').

outro'rst(IS,L,[R—ST],NR):-
asserta(conta(2,1)),
repeat,
(conta(I,'),
 incr(conta(I,1)),
 nth1(I,[R—ST],IC),!,
 ((pe(IS,NR,IC)) ->
  (write(IC),tab(2),write(R),nl);
  (write('falha de reset:'),
   tab(2),write(IS),tab(2),write(IC),nl))),
I = L,!.
outro'rst(',',').

pe(IS,NR,IR):-
modo(IS,LN),
((pini(IR,ARI,NIR,NARI),
  tab(2),write(ARI),nl,
  NR == NIR,
  append(LN,[rst(IR,ARI,NIR,NARI)],LNIS),
  sort(LNIS,LNI),
  write(LNI),nl);
 (pirsti(NP,',IR,IRE),
  append(LN,[rst(NP,IR,IRE)],LNIS),
```

```
  sort(LNIS,LNI),
  write(LNI),nl)),
retract(modo(IS,LN)),
asserta(modo(IS,LNI)),
write(modo(IS,LNI)),nl.


%%%%%%%%%%%%%%%%%%%%%%%%%%
%pesquisa SA's dos scan's
pesa:-
scan(IS,',',SA,',',),
delete(SA,vdd,[S—A]),
pesa(IS,S),
length([S—A],L),
L > 1,
abolish(conta,2),
outro'sa(IS,L,[S—A]),
fail.
pesa.

pesa(IS,R):-
modo(IS,LS),
(((pini(R,AR,NR,NAR),
   append(LS,[[R,AR,NR,NAR]],LNI),
   sort(LNI,LN),
   write(LN),nl,!);
  (pisamux(',R,E), %%%%%%
   append(LS,[[R,E]],LNI),
   sort(LNI,LN),
   write(LN),nl,!);
  (piv(',R,RA),
   append(LS,[[R,RA]],LNI),
   sort(LNI,LN),
   write(LN),nl),!),
 retract(modo(IS,LS)),
 asserta(modo(IS,LN)),
 write(modo(IS,LN)),nl),!;
write('falha de selecao'),tab(3),write(IS),tab(2),
write(R),nl.
pesa(',').

outro'sa(IS,L,[R—ST]):-
asserta(conta(2,1)),
repeat,
(conta(I,IC),
 incr(conta(I,1)),
 nth1(I,[R—ST],IC),!,
 ((pesa(IS,IC)) ->
  (write(IC),tab(2),write(R),nl);
  (write('falha de selecao:'),
   tab(2),write(IS),tab(2),write(IC),nl)),
 write(I),tab(5),write(L),nl),
I = L,!.
outro'sa(',',').

pesa(IS,IR):-
modo(IS,LN),
       ((pini(IR,ARI,NIR,NARI),
  append(LN,[[IR,ARI,NIR,NARI]],LNIS),
  sort(LNIS,LNI),
  write(LNI),nl);
 (pisamux(',IR,E),
  append(LN,[[IR,E]],LNIS),
  sort(LNIS,LNI),
  write(LNI),nl);
 (piv(',IR,IRA),
  append(LN,[[IR,IRA]],LNIS),
  sort(LNIS,LNI),
  write(LNI),nl)),
retract(modo(IS,LN)),
asserta(modo(IS,LNI)),
write(modo(IS,LNI)),nl,!.


%%%%%%%%%%%
%detetar FFSR
cia'ffsr:-
comb(',I,Q,L,[]),
(comb(',I1,Q1,L1,[]),
 I \== I1,
 not((ffsr(LI,',',',',',),
 membro(I,LI))),
memtro(Q1,L),
membro(Q,L1)),
asserta(ffsr([I,I1],Q,Q1,L,L1)),
%tab(2),write(ffsr([I,I1],Q,Q1,L,L1)),nl,
fail.
cia'ffsr.

%%%%%%%%%
concatenate([A],[B],[A,B]).
concatenate([A],[B],[A,B]):- concatenate([],[B],[B]).

%%subseq(Seq, Subseq, Complement)
%%If Subseq is a subsequence of Seq then
%%Complement is the other elements of Seq
subseq([],[],[])
```

```
subseq([Head—Tail],Sbsq,[Head—Cmpl]) :-
       subseq(Tail,Sbsq"c-C"mpl).

subseq([Head—Tail],[Head—Sbsq]"c-C"mpl) :-
       subseq(Tail,Sbsq"c-C"mpl).

%subseq0(Sequence, Subsequence)
%%Is Subsequence a sub-sequence of or
%the same sequen%ce as Sequence?
subseq0(List,List).

subseq0(List,Rest) :-
       subseq1(List,Rest).

%subseq1(Sequence, Subsequence)
% Is Subsequence a subsequence of
% Sequence?
subseq1([_1—Tail],Rest) :-
       subseq0(Tail,Rest).

subseq1([Head—Tail],[Head—Rest]) :-
       subseq1(Tail,Rest).

%%%Enxuga listas de entrada e saida para geracao de bloco %
comum(X,L,Q1):- %%ligacoes unicas
(membro(X,Q1),
 membro(X,L),
 no'(X,',F),
 F < 3).

com(X,L,Q1):- %%% ha fanout
membro(X,Q1),
membro(X,L),
no'(X,',F),
(F >= 3).

ent'comum(X,L,L1):-
membro(X,L1),
membro(X,L),
write('X'),tab(2),write(X),nl.

enxuga1(L,Q1,Q1C,LC):- %%%%enxuga ligacoes unicas
setof(X,X^comum(X,L,Q1)"c-C"),
write('C'),tab(2),write(C),nl,
(subseq(Q1"c-C",Q1C),
 subseq(L"c-C",LC)),!.

enxuga2(Q1C,LC,Q1B,LB"c-C"F):- %%%%enxuga entradas,
                               %%%%ha fanout
 setof(Y,Y^com(Y,LC,Q1C)"c-C"F),
 write('CF'),tab(2),write(CF),nl,
 (Q1B = Q1C,
  subseq(LC"c-C"F,LB)),!.

%%%%%%%%%%%%%%%%%%%%%%
acomb(I,I1,Q,Q1B,L1,LB):-
append(I,I1,IB),
sort(IB,IBC),
tab(2),write('IBC'),tab(2),write(IBC),nl,
((nonvar(Q1B),
  append(Q,Q1B,QB),!),
  QB = Q,!),
sort(QB,QBC),
tab(2),write('QBC'),tab(2),write(QBC),nl,
((nonvar(LB),
  append(LB,L1,LBC),!),
  LBC = L1,!),
sort(LBC,LCC),
tab(2),write('LCC'),tab(2),write(LCC),nl,
asserta(comb(bloco,IBC,QBC,LCC,[])),
write(comb(bloco,IBC,QBC,LCC,[])),nl,
nl,!.

%%seleciona blocos e portas combinacionais
%%que nao sao subseq de outros
contido(I1):-
comb(TIPO,I1,',',',),
(comb(bloco,I,',',',),
(TIPO == bloco->
subseq1(I,I1),
subseq1(I,[I1]))).

predcontido(IUN):-
setof(I1,I1^contido(I1),IU),
nl,
write("Instancias combinacionais do CI verificado
considas em blocos: "),
nl,tab(5),write(IU),nl,nl,nl,nl.
predncontido(IU,IUN).
predcontido(').

ncontido(I,IU):-
comb(',I,',',',),
not(membro(I,IU)).

%%%%%%%%%%%%%%%%%%%%%%
 PREDNCONTIDO
%%%%%%%%%%%%%%%%%%%%%%
```

```
predncontido(IU,IUN):-
setof(I,I^nconstido(I,IU),IUN),
nl,
write("'Instancias combinacionais do CI verificado e
nao contidas em outras :'),
nl,tab(5),write(IUN),nl,nl,nl,nl.
predncontido(',').

%%%%%%%%%%%%%%%%%%%%%
  AGREGACOMB
%%%%%%%%%%%%%%%%%%%%%

agregacomb:-
tell(user),
nl,
write("'Intera"c~c^"^ao enviada ao arquivo saida...
aguarde...'),
        nl,
tell(saida),nl,
write('-- agregacomb --'),nl,
write('-- clano:
gera fato no'(nome,origem,[destino]) --'),
nl,
clano.
%gera fato no'(nome,oriem,[destino])
write('-- agregaportas:
 gera blocos a partir de portas simples -'),nl,
agregaportas.
%agrega portas comb simples
l(reali),
write('-- predcontido(IUN):
IUN = lista de blocos nao contidos --'),nl,
predcontido(IUN),
% IUN = lista de blocos comb nao contidos
write('-- agrega'porta'bl(IUN):
 agrega portas de IUN
        a blocos de IUN -'),nl,
agrega'porta'bl(IUN),
% agrega portas comb a blocos comb
write('-- agregablocos:
gera blocos de blocos --'),nl,
agregablocos.
%agrega blocos combinacionais
%%%%%%nl,nl,l(comb),
write('-- prlimpa:
elimina subsequencias combinacionais --'),nl,
prlimpa.
%%%%%%l(comb),
l(reali),
predcontido(IUN1),
write('-- agrega'bl'entco(IUN1):
 agrega blocos de entradas comuns --'),nl,
agrega'bl'entco(IUN1),
write('-- predcontido(IUN2):
 IUN2 = lista de blocos nao contidos --'),nl,
prlimpa,
predcontido(IUN2),
tell(user),
nl,write(' -- Lista de blocos combinacionais detetados:
 '),nl,
write(IUN2),nl,
write(' -- Deseja continuar a agregacao dos combinacionais:
        s.(sim) ou n. (nao)'),
tab(2),
read(CONTINUA),
((CONTINUA == n -> true;
(tell(saida),
 write('-- agrega'bl'entco(IUN2):
agrega blocos de entradas comuns --'),
 agrega'bl'entco(IUN2),
 prlimpa,
 write('-- predcontido(IUN3):
IUN3 = lista de blocos nao contidos --'),nl,
 predcontido(IUN3))),
statistics,
told.


%%%%%%%%%%%%%%%%%%%%%
%geracao de clausula origem destino COMB
clano:-
tab(2),write('con0'),nl,
con0,
tab(2),write('con1'),nl,
con1,
tab(2),write('con2'),nl,
con2,
fail.
clano.

con0:-
comb(',I,Q,',',),
asserta(no'(Q,I,[])),
fail.
con0.

con1 :-
        no'(Q,I,D),
((setof(INST,INST^fan(Q,INST),FAN),
  length(FAN,L),
  write(L),nl,
  L1 is L + 1);
  (L1 is 1)),
        retract(no'(Q,I,D)),
        asserta(no'(Q,I,L1)),
        fail.
con1.

fan(Q,INST):-
comb(',INST,',E,S),
 (membro(Q,E);
  membro(Q,S)).

con2:-
no'(Q,I,D),
((po(L),
  membro(Q,L));
 (seq(',',',',',Q,',');
  seq(',',',',Q,',',');
  seq(',',',',Q,',',');
  seq(',',',',',Q,',');
  seq(',',',',',',Q,',');
  seq(',',',',',',',Q,'))),
retract(no'(Q,I,D)),
DI is D + 2,
asserta(no'(Q,I,DI)),
fail.
con2.

%%%%%%%%%%%%%%%%%%%%%
  AGREGAPORTAS
%%%%%%%%%%%%%%%%%%%%%

%gera bloco combinacional de duas portas
agregaportas:-
comb(TIPO,I,Q,L,[]),
(comb(TIPO1,I1,Q1,L1,[]),
 TIPO \== bloco,
 TIPO1 \== bloco,
 I \== I1,
 %%not(comb(bloco,[I1,I],',',[])),
not((comb(bloco,IBL,',',[]),
     (membro(I,IBL);
      membro(I1,IBL)))),
((membro(Q,L1),
  agrega(I,I1,Q,Q1,L,L1)))),
 fail.
agregaportas.

agrega(I,I1,Q,Q1,L,L1):-
delete(L1,Q,LB),
no'(Q,',F),
(F < 3 ->
% (fan < 3) => %so uma ligacao
 (QBS = [Q1],!);
 (%LB = L1.
  concatenate([Q],[Q1],QBS),!)),
sort(QBS,QB),
concatenate([I],[I1],IBS),
sort(IBS,IB),
append(LB,L,LBS),
sort(LBS,LBC),
asserta(comb(bloco,IB,QB,LBC,[])),
write(comb(bloco,IB,QB,LBC,[])),nl,
detrealia(Q,QB,LBC),!.

%%%%%%%%%%%%%%%%%%%%%
%detetar realimentacoes entre blocos gerados por agregaportas
detrealia(Q1,QB,LBC):-
membro(X,QB),
X \== Q1,
not(reali(X)),
(membro(X,LBC),
 no'(X,',F),
 (F < 3 ->
   (asserta(reali(Q1)),
    write(reali(Q1)),nl);
   (asserta(reali(Q1)),
    write(reali(Q1)),tab(2),
    asserta(reali(X)),
    write(reali(X)),nl))),
fail.
detrealia(',',',').

%
agrega'porta'bl(IUN):-
comb(TIPO1,I1,Q1,L1,[]),
TIPO1 \== bloco,
membro(I1,IUN),
(comb(TIPO,I,Q,L,[]),
 TIPO == bloco,
not((comb(bloco,IBL,',',[]),
     (subseq1([I,IBL);
      subseq1([IBL,I);
      membro(I1,IBL)))),
((membro(Q1,L),
```

```
    a1(L,Q1,Q,I,I1,L1)),
   (membro(X,Q),
    membro(X,L1),
    a2(L1,X,Q,Q1,I,I1,L))))).
fail.
agrega'porta'bl('').

a1(L,Q1,Q,I,I1,L1):-
delete(L,Q1,LB),
no'(Q1,',F),
(F < 3 ->
% (fan < 3) => %so uma ligacao
 (QBS = Q),
 (append(Q,[Q1],QBS))),
append(LB,L1,LBIS),
sort(QBS,QBL),
append(I,[I1],IBIS),nl,write(IBIS),tab(3),
sort(IBIS,IBI),write(IBI),tab(2),
sort(LBIS,LBI),write(LBI),nl,
comb(bloco,I,Q,L,[]),
write(comb(bloco,I,Q,L,[])),nl,
retract(comb(bloco,I,Q,L,[])),
asserta(comb(bloco,IBI,QBL,LBI,[])),
write(comb(bloco,IBI,QBL,LBI,[])),nl,
detrealib(Q1,Q,L1),!.

a2(L1,X,Q,Q1,I,I1,L):-
delete(L1,X,L1B),
no'(X,',F),
(F < 3 ->
 (delete(Q,X,Q1BSI),
  append(Q1BSI,[Q1],Q1BS));
 (append(Q,[Q1],Q1BS))),
append(L1B,L,L1BS),
sort(Q1BS,Q1B),nl,write(Q1B),tab(3),
append(I,[I1],IBS),write(IBS),tab(3),
sort(IBS,IB),write(IB),tab(3),
sort(L1BS,LBC),write(LBC),nl,
comb(bloco,I,Q,L,[]),
write(comb(bloco,I,Q,L,[])),nl,
retract(comb(bloco,I,Q,L,[])),
asserta(comb(bloco,IB,Q1B,LBC,[])),
write(comb(bloco,IB,Q1B,LBC,[])),nl,
detrealix(X,Q1,L),!.

detrealib(Q1,Q,L1):-
membro(Y,Q),
Y "== Q1,
(membro(Y,L1),nl,write(Q),tab(2),write(L1),tab(2),
 not(reali(Y)),
 no'(Y,',F),
 (F < 3->
  (not(reali(Q1)),
   no'(Q1,',F1),
   F1 >= 3,
   asserta(reali(Q1)),
   write(reali(Q1)),tab(3),write(Y),nl);
 (asserta(reali(Y)),
  write(reali(Y)),nl))),
fail.
detrealib(',',').

detrealix(X,Q1,L):-
membro(Q1,L),
(not(reali(Q1)),
 no'(Q1,',F),
 (F < 3->
  (not(reali(X)),
   asserta(reali(X)),
   write(reali(X)));
 (asserta(reali(Q1)),
  write(reali(Q1)),nl))),
fail.
detrealix(',',').

enxuga'comb:-
comb(bloco,I,Q,L,[]),
(enxuga1(L,Q,Q1,L1),
(Q1 = Q,
L1 = L)),
(enxuga2(Q1,L1,Q12,L12"c-C"F),
(Q12 = Q1,
L12 = L1)),
retract(comb(bloco,I,Q,L,[])),
asserta(comb(bloco,I,Q12,L12,[])),
fail.
enxuga'comb.

%%%%%%%%%%%%%%%%%%%%%%
 AGREGABLOCOS
%%%%%%%%%%%%%%%%%%%%%%

%gera comb(bloco,... de comb(blocos
agregablocos:-
comb(TIPO,I,Q,L,[]),
TIPO == bloco,
(comb(TIPO,I1,Q1,L1,[]),
 TIPO == bloco,
 I "== I1,
 (membro(X,Q1),
  membro(X,L)),
  nl,write('X'), tab(2), write(X),nl,
  (enxuga1(L,Q1,Q1C,LC);
%%% elimina ligacoes unicas
   (Q1C = Q1,
    LC = L)),
nl,write('Q1C  LC'),write(Q1C),tab(2),write(LC),nl,
write("'enxuga1'),nl,
  (enxuga2(Q1C,LC,Q1B,LB"c-C"F1);
%%% elimina sinais  comuns da entrada
   (Q1B = Q1C,
    LB = LC)),
nl,write('Q1B  LB CF1'),
write(Q1B),tab(2),write(LB),tab(2),write(CF1),nl,
write("'enxuga2'),nl,
  nl,write('Q1B'),tab(2),write(Q1B),tab(2),
  write('LB'),tab(2),write(LB),nl,
  acomb(I,I1,Q,Q1B,L1,LB),
retract(comb(',I,',',[])),
write(I),tab(5),
retract(comb(',I1,',',[])),
write(I1),nl,
!,
detrealib(X,Q,L1)),
fail.
agregablocos.

limpa(I,I1):-
((subseq1(I,I1),
   retract(comb(',I1,',',''))),
   %write(comb(bloco,I,',',''))),
!,
  (subseq1(I1,I),
   retract(comb(',I,',',''))),
   %write(comb(bloco,I1,',',''))),
!).
limpa(',').

prlimpa:-
comb(bloco,I,',',''),
comb(bloco,I1,',',''),
limpa(I,I1),
fail.
prlimpa.

%%%%%%%%%%%%%%%%%%%%
  AGREGA-BL-ENTCO
%%%%%%%%%%%%%%%%%%%%%%

agrega'bl'entco(IUN):-
comb(TIPO,I,Q,L,[]),
TIPO == bloco,
((comb(TIPO1,I1,Q1,L1,[]),
 ((TIPO1 == bloco) ->
  (I "== I1,write("I,I1 : '),IM = I1,QM = Q1,
write(I),tab(2),write(I1),nl;
  ((membro(I1,IUN)),IM = [I1],QM = [Q1],write("I,I1,IM : '),
write(I),tab(2),write(I1),tab(2),write(IM),nl))),
setof(X,X"'ent'comum(X,L,L1),EC),
write("'EC'),tab(2),write(EC),nl,
acomb(I,IM,Q,QM,L1,L),
retract(comb(',I,',',[])),
write(I),tab(5),
retract(comb(TIPO1,I1,Q1,L1,[])),
write(I1),nl),
%prlimpa,
fail.
agrega'bl'entco(').

%%%%%%%%%%%%%%%%%%%%
header:-
ci(CI),
%concatenate([CI],['rel],[CIREL]),
%concatenate([CI],['fal],[CIFAL]),
% asserta(file(CIFAL)),
% asserta(file(CIREL)),
tell(rel),nl,
tab(2),write(' ###                                          ###'),nl,
tab(2),write(' ###          SMAuT                           ###'),nl,
tab(2),write(' ###                                          ###'),nl,
tab(2),write(' ###  Sistema Modular de Auditoria de
 Testabilidade      ###'),nl,
tab(2),write(' ###                                          ###'),nl,
tab(2),write(' ###  Prototipo realizado por: Bernadete A.L.Oliveira  ###'),nl,
tab(2),write(' ###                                          ###'),nl,
tab(2),write(' ###                                          ###'),nl,
tab(2),write(' ###  Circuito verificado:'),tab(2),
tab(2),write(CI),nl,
tab(2),write(' ###                                          ###'),nl,
%file(CIFAL),
tell(falhas),nl,
tab(2),write(' ###          SMAuT                       ### '),nl,
tab(2),write(' ###  Sistema Modular de Auditoria de
                    Testabilidade   ### '),nl,
tab(2),write(' ###  Prototipo realizado por:
            Bernadete A.L.Oliveira ### '),nl,nl,
```

```
tab(2),write(' ###                                         ### '),nl,
tab(2),write(' ###        Arquivo de falhas                 ### '),nl,
tab(2),write(' ###                                         ### '),nl,
ci(CI),tab(2),write(' ###   CI analisado:'),tab(3),
write(CI),nl,
tab(2),write(' ###                                         ### '),
nl,nl,nl.


%%%%%%%%%%%%%%%%%%%%%
  INTERFACE
%%%%%%%%%%%%%%%%%%%%%

%geracao de arquivo de saida
gera :-
tell(rel),
tab(2),write(' ###   Caminhos identificados/verificados:'),nl,
lscan,
lqsai,
tab(2),write(' ###   Logica Combinacional em blocos:'),nl,
%l(comb),
predcontido(IUN),nl,
liste(IUN),
nl,nl,tab(6),write(' ####  Realimenta~c~~oes detetadas: '),
nl,l(reali),nl,nl,
%nl,tab(2),write(' ####        AVISO: vdd n~ao se aplica a pos-
sivel scanin'),
nl,tab(2),write(' ####        Fim de relatorio        #### '),
told,
tell(falhas),nl,nl,
tab(2),write(' ###   Elementos de circuito fora das regras:'),nl,
tab(2),write(' ###   Clocks nao controlaveis: '),nl,
lnpiclki,nl,nl,
lfdclk,
lfdclkx,nl,nl,
tab(2),write(' ### Celulas sequenciais fora de scan:      ### '),
nl,
tab(2),write(' ###   Instancias sequenciais fora de qualquer scan:'),
      nl,
lfalha'fsc,nl,nl,
lnelo,
lmelo,
nl,nl,tab(2),write(' ###   Falhas de scanin: '),nl,
lfscanin,nl,nl,
tab(2),write(' ###   Falhas de scanout: '), nl,
lfscanout,nl,nl,
tab(2),write(' ####        Fim de relatorio        #### '),
told.
gera.

lfdclk:-
falha'dado'nclk([N,DA~c-C~K]),
nonvar(N),
nl,tab(2),write('### ERRO  ###: Clock alimenta entrada de
dados da instancia:'),tab(2),write(N),nl,
tab(20),write('sinal de clock:'),tab(1),write(CK),tab(2),write('
      entrada de dados:'),tab(2),write(DA),nl,nl,
fail.
lfdclk.

lfdclkx:-
falha'dado'nclkx([N,DA,X,XI]),
nonvar(N),
nl,write(' ### Clock alimenta entrada de dados da instancia:'),
      tab(2),write(N),nl,
tab(20),write(' ### sinal de clock(na instancia e pino):'),
      tab(1),write(X),tab(2),write(XI),tab(2),write("'entrada:'),
      tab(2),write(DA),nl,nl,
fail.
lfdclkx.

lnelo:-
nelo([N—']),
nonvar(N),
tab(3),write(' ### Detetado por cla'elo'mxa = instancia '),
      tab(2),write(N),tab(2),
nl,tab(2),
write(' ### observavel mas nao controlavel ###'),nl,nl,
fail.
lnelo.

lmelo:-
melo([N,M]),
nonvar(M),
seq(',TIPO,M,',',',','),
tab(2),
write(' ### Substituir as seguintes instancias por MXFFD:'),
      tab(2),write(TIPO),tab(2),write(M),tab(2),write("'e'),
      tab(2),write('MUX : '),write(N),nl,
fail.
lmelo.

lnpiclki:-
npiclki(',N~c-C~P),
seq(',TIPO,N,',',',','),
%file(CIFAL),
tell(falhas),nl,
tab(2),
'write(' ### Sinal de clock nao controlavel a partir de pino: '),
      tab(2),write(CP),nl,
tab(3),write(' ### Associado com a instancia: '),
      tab(2),write(N),tab(2),write('Tipo: '),write(TIPO),nl,
fail.
lnpiclki.

lfalha'fsc:-
ncel(CEL,I),
tab(6),write("'~"i~nstancia'tipo,numero: '),
      write(CEL),write(','),write(I),nl,
fail.
lfalha'fsc.

lfscanin:-
fscanin(I,N,A,B),
 %file(CIFAL),
 tell(falhas),nl,
 tab(2),write(' ### Detetada falha de scanin por cla'sc'scin:'),
nl,
 tab(6),write('SCAN:'),tab(2),write(I),tab(4),
      write("'~"i~nstancia da celula:'), tab(2),write(N),nl,
 tab(6),write('Possiveis entradas de scan:'),tab(2),write(A),
((B ~== vdd,tab(2),write(B),nl,nl,fail);
fail).
lfscanin.

lqsai:-
qsai(N,Q,X),
nonvar(N),
nl,nl,tab(2),write(' ###   Possivel saida de scan atraves da ins-
tancia '),
write(N),write(': '),tab(2),write(Q),tab(2),write(X),nl,nl,
fail.
lqsai.

lscan:-
scan(I,LE,LCK,LRST,LSA,LINST,'.'),nl,nl,
nonvar(I),
tab(2),write('SCAN : '),tab(2),write(I),nl,
tab(2),write('Numero de instancias sequenciais do scan:'),
      tab(2),write(LE),nl,
tab(2),write('lista de sinais de clock:'),tab(2),write(LCK),nl,
tab(2),write('lista de sinais de reset:'),tab(2),write(LRST),nl,
tab(2),write('lista de sinais de selecao:'),tab(2),write(LSA),nl,
tab(2),write("'~"i~nstancias do scan '),
      write(I),write(':'),write(LINST),nl,nl,
lsci(I),nl,
lscout(I),nl,
tab(2),write('Modo scan dado pelos sinais e respectivos niveis
      listados a seguir:'),nl,
lmodo(I),nl,
tab(2),write('Niveis associados com o scan '),
write(I),write(':'),nl,
lnivel(I),
nl,nl,write(' ###################### '),
      nl,nl,
fail.
lscan.

lsci(I):-
scan(I,',',',',[BE,',B],'),
nonvar(B),
(tab(2),write('SCANIN atraves de  '),
      tab(2),write(B),tab(2),write(BE),nl),!.
lsci(').

lscout(I):-
scan(I,',',',',',[Q,QS]),
nonvar(Q),
(tab(2),write('SCANOUT atraves de : '),
      tab(2),write(Q),tab(2),write(QS),nl),!.
lscout(').

lmodo(I):-
modo(I,LMO),
LMO ~== [],
membro(X,LMO),nl,
((X = [SI,SO,NI,NO],
  tab(2),write('['),write(SI),write(','),write(NI),write(']'),
  write(' com '),
      write('['),write(SO),write(','),write(NO),write(']'),nl);
 (X = rst(SIR,SOR,NIR,NOR),
    tab(2),
      write('rst:'),
      write('['),write(SIR),write(','),write(NIR),write(']'),
  write(' com '),
write('['),write(SOR),write(','),write(NOR),write(']'),nl)),
fail,!.
lmodo(').

lnivel(I):-
nivel(I,IM,SI,NP),
nonvar(SI),nl,
tab(2),write("'~"i~nstancia: '),
write(IM),tab(2),write(SI),tab(2),write(NP),nl,
fail,!.
lnivel(').
```

```
liste(IUN):-
(nonvar(IUN),
 IUN \== []),
write('Blocos combinacionais:'),
membro(I,IUN),
comb(bloco,I,Q,E,'),
nl,write('\'-\"i\'nstancias do bloco:'),write(I),
nl,tab(2),write('saidas do bloco:'),write(Q),
nl,tab(2),write('\'entradas do bloco:'),write(E),nl,
fail.
liste('):

lfscanout:-
fscanout(I,X,Q),
%file('CIFAL),
tell(falhas),nl,
tab(2),
write(' ### Detetada falha de scanout por cia'sc'scout:'),nl,
tab(6),write('SCAN:'),tab(2),write(I),
tab(4),write('\'-\"i\'nstancia da celula:'),
tab(2),write(X),nl,
tab(6),write(' ### Possivel saida de scan:'),
tab(2),write(Q),nl,nl,
fail.
lfscanout.

%%%%%%%%%%%%%%%%%%%
append([],X,X).
append([X—R],S,[X—W]) :- append(R,S,W).

delete([],A,[]) :- !.

delete([Kill—Tail],Kill,Rest) :- !,
        delete(Tail,Kill,Rest).

delete([Head—Tail],Kill,[Head—Rest]) :- !,
        delete(Tail,Kill,Rest).

last(X,[X]).
last(X,[_—Y]) :- last(X,Y).

membro(X,[X—_]).
membro(X,[_—Y]) :- membro(X,Y).

not(X) :- call(X),!,fail.
not(_).

nth1(1,[Head—_1],Head) :- !.

nth1(N,[_1—Tail],Elem) :-
        nonvar(N),
        M is N-1,
        nth1(M,Tail,Elem).

nth1(N,[_1—T],Item) :-
        var(N),
        nth1(M,T,Item),
        N is M+1 .


%%%%%%%%%%%%%%%%%%%
pov(N,X,Q) :-
obut1(N,Q,X);
(iobuf(NP,N,Q,',X,NEN),
piv(',NEN,'));
 tribuf(NP,N,Q,X,NEN),
 piv(',NEN,'),
 (pio(LIO),
  membro(Q,LIO));
(pio(LIO),
membro(X,LIO));
(po(LO),
 membro(X,LO)).



%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%

F I M  D E  P R O G R A M A


%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%



E X E M P L O S :



%%%%%%%%%%%%                maqsup
%%%%CCT AMAQSUP
```

```
%
%(bifunc2,nresin,napama,ntsql,papam,
%% saidas napama = saida p/sca%n de teste
%
%% saidas
%%% ntsql  =% controle para auxilio ao teste
%pvostil,nitql0,nitb7c16,pepam,pdlinra,psmql,
%% entradas
%nmepet,p2ml,p2ml1,nteste,psql,nitcl16,
%pshtest,vdd).
%%%%%% trocados os nomes pvostil por pvostul e pshtest
por pvozar%%

%BIT
%%%%%bitn(n,50,nqb,pqb,',',vdd,nteste,pshtest,
pm23,p2ml,vdd).
bitn(n,50,nqb,pqb,',',vdd,nteste,pvozar,pm23,
p2ml,vdd).
        bitn(n,51,pct,nct,',',vdd,nteste,nqb,po10,n2ml,
ntsql).
        bitn(n,52,ntza,',',',vdd,nteste,pct,po11,n2ml,
nsmql1).
        bitn(n,53,ntz,',',',vdd,ntsmql,ntza,ntz,p2ml1,
vdd).
        bitn(n,54,napam,papam,',',vdd,nteste,ntz,pm24,
p2ml,vdd).
        bitn(n,55,napama,',',',vdd,nt7c16,napam,napama,
p2ml,vdd).
%AA025
and(22,na22,[napam,nitql0]).
        and(23,pdlin16,[pdlinra,pitcl16]).
        and(24,ntsmql,[nteste,nsmql]).
        and(25,nt7c16,[nteste,nitb7c16]).
%NA025
nand(28,pa28,[pepam,nda]).
        nand(29,pa29,[pa28,pct]).
        nand(38,ntsql,[psql,nteste]).
%NA035
nand(45,nsmql1,[psmql,n2ml,nteste]).
%IN015
nbuf(19,pitcl16,nitcl16).
        nbuf(20,nsmql,psmql).
        nbuf(21,n2ml,p2ml).
%OR025
or(8,pcme,[na22,nitb7c16]).
        or(10,po10,[pdlin16,pct]).
        or(11,po11,[pdlin16,ntza]).
%OR035
or(13,nresin,[papam,ntsql,napama]).
%NO035
nor(17,nda,[nn8,nn9,nn10]).
%NO025
nor(18,nn8,[napam,pqb]).
        nor(9,nn9,[ntz,papam]).
%NO035
nor(36,nn10,[pepam,nqb,nct]).
%MUX
mx(31,pm23,[pa29,nqb],[pcme]).
        mx(32,pm24,[nda,napam],[pcme]).
        mx(33,bifunc2,[papam,pvostul],[nmepet]).

%INPUT
pi([pepam,nitb7c16,nitql0,nitcl16,pdlinra,psql,psmql,
pvostul,nmepet,p2ml,p2ml1, nteste,pvozar,vdd]).

po([bifunc2,nresin,napama,ntsql,papam]).
modo'scan([[nteste,0]]).
clock([p2ml,p2ml1]).


  ###
  ###                SMAuT
  ###
  ###   Sistema Modular de Auditoria de Testabilidade
  ###
  ###   Prototipo realizado por: Bernadete A.L.Oliveira
  ###
  ###
  ###   Circuito verificado:    maqsup
  ###
  ###   Caminhos identificados/verificados:


SCAN :  0
Numero de instancias sequenciais do scan:  6
lista de sinais de clock:  [n2ml,p2ml1,p2nil]
lista de sinais de reset:  [nsmql1,ntsql,vdd]
lista de sinais de selecao:  [nt7c16,nteste,ntsmql]
instancias do scan 0:[50,51,52,53,54,55]

SCANIN atraves de :  pvozar  pvozar

SCANOUT atraves de :  napama  '131952

 Modo scan dado pelos sinais e respectivos niveis
listados a seguir:

[nt7c16,0] com [nitb7c16,0]
```

```
[nteste,0] com [nteste,0]

[ntsmql,0] com [nteste,0]

rst:[nsmql1,1] com [nteste,0]

rst:[ntsql,1] com [nteste,0]

Niveis associados com o scan 0:

####


 ###   Possivel saida de scan atraves da instancia 55:
 napama  `387780

 ###   Logica Combinacional em blocos:

Instancias combinacionais do CI verificado contidas em
 blocos:
    [8,9,10,11,13,17,18,19,20,21,22,23,24,25,28,29,30,
38,45]



Instancias combinacionais do CI verificado e  nao
contidas em outras :
    [31,32,33,[8,9,10,11,13,17,18,19,20,21,22,23,24,25,
28,29,30,38,45]]



Blocos combinacionais:
instancias do bloco:[8,9,10,11,13,17,18,19,20,21,22,23,
24,25,28,29,30,38,45]
  saidas do bloco:[n2ml,nda,nresin,nsmql1,nt7c16,ntsmql,
ntsql,pa29,pcme,pdlin16,po10,po11]
  entradas do bloco:[napam,napama,nct,nitb7c16,nitcl16,
nitql0,nqb,nteste,ntz,ntza,p2ml,papam,pct,pdlinra,pepam,
pqb,psmql,psql]

instancias do bloco:[8,9,10,11,13,17,18,19,20,21,22,23,
24,25,28,29,30,38,45]
  saidas do bloco:[n2ml,nda,nresin,nsmql1,nt7c16,ntsmql,
ntsql,pa29,pcme,pdlin16,po10,po11]
  entradas do bloco:[napam,napama,nct,nitb7c16,nitcl16,
nitql0,nqb,nteste,ntz,ntza,p2ml,papam,pct,pdlinra,pepam,
pqb,psmql,psql]

instancias do bloco:[8,9,10,11,13,17,18,19,20,21,22,23,
24,25,28,29,30,38,45]
  saidas do bloco:[n2ml,nda,nresin,nsmql1,nt7c16,ntsmql,
ntsql,pa29,pcme,pdlin16,po10,po11]
  entradas do bloco:[napam,napama,nct,nitb7c16,nitcl16,
nitql0,nqb,nteste,ntz,ntza,p2ml,papam,pct,pdlinra,pepam,
pqb,psmql,psql]

instancias do bloco:[8,9,10,11,13,17,18,19,20,21,22,23,
24,25,28,29,30,38,45]
  saidas do bloco:[n2ml,nda,nresin,nsmql1,nt7c16,ntsmql,
ntsql,pa29,pcme,pdlin16,po10,po11]
  entradas do bloco:[napam,napama,nct,nitb7c16,nitcl16,
nitql0,nqb,nteste,ntz,ntza,p2ml,papam,pct,pdlinra,pepam,
pqb,psmql,psql]

instancias do bloco:[8,9,10,11,13,17,18,19,20,21,22,23,
24,25,28,29,30,38,45]
  saidas do bloco:[n2ml,nda,nresin,nsmql1,nt7c16,ntsmql,
ntsql,pa29,pcme,pdlin16,po10,po11]
  entradas do bloco:[napam,napama,nct,nitb7c16,nitcl16,
nitql0,nqb,nteste,ntz,ntza,p2ml,papam,pct,pdlinra,pepam,
pqb,psmql,psql]

instancias do bloco:[8,9,10,11,13,17,18,19,20,21,22,23,
24,25,28,29,30,38,45]
  saidas do bloco:[n2ml,nda,nresin,nsmql1,nt7c16,ntsmql,
ntsql,pa29,pcme,pdlin16,po10,po11]
  entradas do bloco:[napam,napama,nct,nitb7c16,nitcl16,
nitql0,nqb,nteste,ntz,ntza,p2ml,papam,pct,pdlinra,pepam,
pqb,psmql,psql]

instancias do bloco:[8,9,10,11,13,17,18,19,20,21,22,23,
24,25,28,29,30,38,45]
  saidas do bloco:[n2ml,nda,nresin,nsmql1,nt7c16,ntsmql,
ntsql,pa29,pcme,pdlin16,po10,po11]
  entradas do bloco:[napam,napama,nct,nitb7c16,nitcl16,
nitql0,nqb,nteste,ntz,ntza,p2ml,papam,pct,pdlinra,pepam,
pqb,psmql,psql]

instancias do bloco:[8,9,10,11,13,17,18,19,20,21,22,23,
24,25,28,29,30,38,45]
  saidas do bloco:[n2ml,nda,nresin,nsmql1,nt7c16,ntsmql,
ntsql,pa29,pcme,pdlin16,po10,po11]
```

```
  entradas do bloco:[napam,napama,nct,nitb7c16,nitcl16,
nitql0,nqb,nteste,ntz,ntza,p2ml,papam,pct,pdlinra,pepam,
pqb,psmql,psql]

    ####  Realimenta"c-c""oes detetadas:


  ####        Fim de relatorio



  ###              SMAuT
  ###   Sistema Modular de Auditoria de Testabilidade
  ###   Prototipo realizado por: Bernadete A.L.Oliveira

  ###
  ###          Arquivo de falhas
  ###
  ###   CI analisado:   maqsup
  ###


  ###  Lista de sinais de clock verificada:
       SCAN(0) :   [n2ml,p2ml1,p2ml]

 ###### FALHA = DIFERENTES PINOS DE CLOCK PARA:
p2ml1
n2ml

### ### ERRO : diferentes fases de clock no caminho
 Pino associado aos sinais de clock: p2ml
 Para o sinal p2ml, numero de buffers inversores: 0
  e para o sinal n2ml, numero de buffers inversores:1


  ###   Elementos de circuito fora das regras:
  ###   Clocks nao controlaveis:


  ###   Instancias sequenciais fora de qualquer scan:


  ###   Falhas de scanin:


  ###   Falhas de scanout:


  ####        Fim de relatorio


%%%%%%%%%%%%%%%%%%%%   CONV
%%%% latches %%%%%%%%%
latch(n,1,bti1,`,bt1,rp).
latch(n,2,bti2,`,bt2,rp).
latch(n,3,bti3,`,bt3,rp).
latch(n,4,bti4,`,bt4,rp).
latch(n,5,bti5,`,bt5,rp).
latch(n,6,bcdi1,`,bcd1,rp).
latch(n,7,bcdi2,`,bcd2,rp).
latch(n,8,bcdi3,`,bcd3,rp).
latch(n,9,bcdi4,`,bcd4,rp).
latch(n,10,bcdi5,`,bcd5,rp).

mxffd(n,11,bta1,`,bti1,etab,ntest,rab,vdd)
mxffd(n,12,bib1,`,bti1,etaa,ntest,raa,vdd).
mxffd(n,13,btb1,`,bib1,bta1,ntest,rab,vdd).
mxffd(n,14,bta2,`,bti2,btb1,ntest,rab,vdd).
mxffd(n,15,bib2,`,bti2,bib1,ntest,raa,vdd).
mxffd(n,16,btb2,`,bib2,bta2,ntest,rab,vdd).
mxffd(n,17,bta3,`,bti3,btb2,ntest,rab,vdd).
mxffd(n,18,bib3,`,bti3,bib2,ntest,raa,vdd).
mxffd(n,19,btb3,`,bib3,bta3,ntest,rab,vdd).
mxffd(n,20,bta4,`,bti4,btb3,ntest,rab,vdd).
mxffd(n,21,bib4,`,bti4,bib3,ntest,raa,vdd).
mxffd(n,22,btb4,`,bib4,bta4,ntest,rab,vdd)
mxffd(n,23,bta5,`,bti5,bib4,ntest,raa,vdd).
mxffd(n,24,sbcd1,`,bcdi1,bta5,ntest,raa,vdd)
mxffd(n,25,sbcd2,`,bcdi2,sbcd1,ntest,raa,vdd).
mxffd(n,26,sbcd3,`,bcdi3,sbcd2,ntest,raa,vdd).
mxffd(n,27,sbcd4,`,bcdi4,sbcd3,ntest,raa,vdd).
mxffd(n,28,staa,`,bcdi5,sbcd4,ntest,raa,vdd).
mxffd(n,29,stab,`,staa,btb4,ntest,rab,vdd).

mx(30,sbcd5,[stab,staa],[c12]).
mx(31,dt1,[bta1,bib1],[c12]).
mx(32,dt2,[btb1,bib2],[c12]).
mx(33,dt3,[bta2,bib3],[c12]).
mx(34,dt4,[btb2,bib1],[c12]).
mx(35,dt5,[bta3,bib2],[c12]).
mx(36,dt6,[btb3,bib3],[c12]).
mx(37,dt7,[bta4,bib4],[c12]).
mx(38,dt8,[btb4,bta5],[c12])
```

```
modo'scan([[ntest,0]]).

pi([vdd,vss,etab,etaa,bt1,bt2,bt3,bt4,bt5,rp,raa,rab,
ntest"c-c"12,bcd1,bcd2,bcd3,bcd4,bcd5]).

po([dt1,dt2,dt3,dt4,dt5,dt6,dt7,dt8,sbcd1,sbcd2,sbcd3,
sbcd4,sbcd5,staa,stab]).

clock([raa,rab,rp]).


  ###
  ###                SMAuT
  ###
  ###   Sistema Modular de Auditoria de Testabilidade
  ###
  ###   Prototipo realizado por: Bernadete A.L.Oliveira
  ###
  ###
  ###   Circuito verificado:    conv
  ###
  ###   Caminhos identificados/verificados:


 SCAN :   1
 Numero de instancias sequenciais do scan:  10
 lista de sinais de clock:  [raa]
 lista de sinais de reset:  [vdd]
 lista de sinais de selecao:  [c12,ntest]
 instancias do scan 1:[12,15,18,21,23,24,25,26,27,28,30]

 SCANIN atraves de :   etaa  etaa

 SCANOUT atraves de :   sbcd5  '125731

 Modo scan dado pelos sinais e respectivos niveis
listados a seguir:


 [ntest,0]  com [ntest,0]

 Niveis associados com o scan 1:


 #####


 SCAN :   0
 Numero de instancias sequenciais do scan:  9
 lista de sinais de clock:  [rab]
 lista de sinais de reset:  [vdd]
 lista de sinais de selecao:  [c12,ntest]
 instancias do scan 0:[11,13,14,16,17,19,20,22,29,30]

 SCANIN atraves de :   etab  etab

 SCANOUT atraves de :   sbcd5  '125728

 Modo scan dado pelos sinais e respectivos niveis
listados a seguir:


 [ntest,0]  com [ntest,0]

 Niveis associados com o scan 0:


 #####



  ###   Possivel saida de scan atraves da instancia 29:
 stab  '387678
  ###   Possivel saida de scan atraves da instancia 28:
staa  '387678
  ###   Possivel saida de scan atraves da instancia 27:
sbcd4  '387678
  ###   Possivel saida de scan atraves da instancia 26:
sbcd3  '387678
  ###   Possivel saida de scan atraves da instancia 25:
sbcd2  '387678
  ###   Possivel saida de scan atraves da instancia 24:
sbcd1  '387678

  ###   Logica Combinacional em blocos.


    ####   Realimenta"c-c""oes detetadas:


  ####        Fim de relatorio
```

```
  ###
  ###                SMAuT
  ###   Sistema Modular de Auditoria de Testabilidade
  ###   Prototipo realizado por: Bernadete A.L.Oliveira
  ###
  ###        Arquivo de falhas
  ###
  ###   CI analisado:    conv
  ###
  ###   Elementos de circuito fora das regras:
  ###   Clocks nao controlaveis:

 ### Sinal de clock nao controlavel a partir de pino:1
 ### Associado com a instancia:   10  Tipo: latch

 ### Sinal de clock nao controlavel a partir de pino:1
 ### Associado com a instancia:   9  Tipo: latch

 ### Sinal de clock nao controlavel a partir de pino:1
 ### Associado com a instancia:   8  Tipo: latch

 ### Sinal de clock nao controlavel a partir de pino:1
 ### Associado com a instancia:   7  Tipo: latch

 ### Sinal de clock nao controlavel a partir de pino:1
 ### Associado com a instancia:   6  Tipo: latch

 ### Sinal de clock nao controlavel a partir de pino:1
 ### Associado com a instancia:   5  Tipo: latch

 ### Sinal de clock nao controlavel a partir de pino:1
 ### Associado com a instancia:   4  Tipo: latch

 ### Sinal de clock nao controlavel a partir de pino:1
 ### Associado com a instancia:   3  Tipo: latch

 ### Sinal de clock nao controlavel a partir de pino:1
 ### Associado com a instancia:   2  Tipo: latch

 ### Sinal de clock nao controlavel a partir de pino:1
 ### Associado com a instancia:   1  Tipo: latch

 ### Celulas sequenciais fora de scan:
 ###   Instancias sequenciais fora de qualquer scan:
   instancia'tipo,numero:  latch,10
   instancia'tipo,numero:  latch,9
   instancia'tipo,numero:  latch,8
   instancia'tipo,numero:  latch,7
   instancia'tipo,numero:  latch,6
   instancia'tipo,numero:  latch,5
   instancia'tipo,numero:  latch,4
   instancia'tipo,numero:  latch,3
   instancia'tipo,numero:  latch,2
   instancia'tipo,numero:  latch,1

 ###   Falhas de scanin:


 ###   Falhas de scanout


 ####        Fim de relatorio


%%%%%%%%%%%%%%%%%%   CONTMOD7K
%mxffd(NP,I,Q,'',DA,DB,SA"c-c"K,RST)

mxffd(n,1,q13,'',rpo13,dteste,nteste,raa,reset).
mxffd(n,2,q12,'',q12,q13,penviol,raa,reset).
mxffd(n,3,q11,'',q11,q12,penviol,raa,reset).
mxffd(n,4,q10,'',q10,q11,penviol,raa,reset).
mxffd(n,5,q9,'',q9,q10,penviol,raa,reset).
mxffd(n,6,q8,'',q8,q9,penviol,raa,reset).
mxffd(n,7,q7,'',q7,q8,penviol,raa,reset).
mxffd(n,8,q6,'',q6,q7,penviol,raa,reset).
mxffd(n,9,q5,'',q5,q6,penviol,raa,reset).
mxffd(n,10,q4,'',q4,q5,penviol,raa,reset).
mxffd(n,11,q3,'',q3,q4,penviol,raa,reset).
mxffd(n,12,q2,'',q2,q3,penviol,raa,reset).
mxffd(n,13,q1,'',q1,q2,penviol,raa,reset).
mxffd(n,14,stes,tft3,q0,q1,nteste,raa,reset).

mx(15,rpo13,[rpoli,q13],[penviol]).
mx(16,q0,[p7k,stes],[nenable])

exor(17,po,[q10,q1]).
exor(18,li,[q13,q11]).
exnor(19,rpoli,[po,li]).

nbuf(20,nena,penable).

nand(22,pena,[nena,ncdviol])
nand(21,ncdviol,[cdviol,n7k])

nbuf(23,penviol,pena).
nbuf(24,pteste,nteste).
```

```
nor(25,ndrte,[drte,pteste]).

nbuf(26,reset,ndrte).

and(27,dprte,[drte,prte]).

nor(28,nenable,[dprte,penable]).

nand(29,n1,[q12,q11,q9,q8,q6,q4]).

or(30,n2,[q13,q10,q7,q5,q3,q2,q1]).

nor(31,p7k,[n1,n2]).

nbuf(32,n7k,p7k).

pi([nteste,dteste,raa,penable"c-c"dviol,drte,prte]).

po([stes,tft3]).

clock([raa]).

modo'scan([[nteste,0],[penable,1]]).


  ###
  ###              SMAuT
  ###
  ###   Sistema Modular de Auditoria de Testabilidade
  ###
  ###   Prototipo realizado por: Bernadete A.L.Oliveira
  ###
  ###
  ###   Circuito verificado:   contmod7k
  ###
  ###   Caminhos identificados/verificados:


 SCAN :   0
 Numero de instancias sequenciais do scan:  14
 lista de sinais de clock:  [raa]
 lista de sinais de reset:  [reset]
 lista de sinais de selecao:  [nteste,penviol]
 instancias do scan 0:[1,2,3,4,5,6,7,8,9,10,11,12,13,14]

 SCANIN atraves de :   dteste  dteste

 SCANOUT atraves de :   stes  `130209

  Modo scan dado pelos sinais e respectivos niveis
listados a seguir:

 [nteste,0] com [nteste,0]

 [penviol,0] com [penable,1]

 rst:[reset,1] com [nteste,0]

 Niveis associados com o scan 0:


 _


  ###   Possivel saida de scan atraves da instancia 14:
 stes  `387780

  ###   Logica Combinacional em blocos:

Instancias combinacionais do CI verificado contidas em
blocos:
    [17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32]


Instancias combinacionais do CI verificado e  nao
 contidas em outras :
    [15,16,[17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
32]]


Blocos combinacionais:
instancias do bloco:[17,18,19,20,21,22,23,24,25,26,27,28,
29,30,31,32]
  saidas do bloco:[nenable,p7k,penviol,reset,rpoli]
  entradas do bloco:[cdviol,drte,nteste,penable,prte,q1,
q10,q11,q12,q13,q2,q3,q4,q5,q6,q7,q8,q9]

instancias do bloco:[17,18,19,20,21,22,23,24,25,26,27,28,
29,30,31,32]
  saidas do bloco:[nenable,p7k,penviol,reset,rpoli]
  entradas do bloco:[cdviol,drte,nteste,penable,prte,q1,
q10,q11,q12,q13,q2,q3,q4,q5,q6,q7,q8,q9]

instancias do bloco:[17,18,19,20,21,22,23,24,25,26,27,28,
29,30,31,32]
  saidas do bloco:[nenable,p7k,penviol,reset,rpoli]
  entradas do bloco:[cdviol,drte,nteste,penable,prte,q1,
q10,q11,q12,q13,q2,q3,q4,q5,q6,q7,q8,q9]

instancias do bloco:[17,18,19,20,21,22,23,24,25,26,27,28,
29,30,31,32]
  saidas do bloco:[nenable,p7k,penviol,reset,rpoli]
  entradas do bloco:[cdviol,drte,nteste,penable,prte,q1,
q10,q11,q12,q13,q2,q3,q4,q5,q6,q7,q8,q9]

instancias do bloco:[17,18,19,20,21,22,23,24,25,26,27,28,
29,30,31,32]
  saidas do bloco:[nenable,p7k,penviol,reset,rpoli]
  entradas do bloco:[cdviol,drte,nteste,penable,prte,q1,
q10,q11,q12,q13,q2,q3,q4,q5,q6,q7,q8,q9]

instancias do bloco:[17,18,19,20,21,22,23,24,25,26,27,28
29,30,31,32]
  saidas do bloco:[nenable,p7k,penviol,reset,rpoli]
  entradas do bloco:[cdviol,drte,nteste,penable,prte,q1,
q10,q11,q12,q13,q2,q3,q4,q5,q6,q7,q8,q9]

instancias do bloco:[17,18,19,20,21,22,23,24,25,26,27,28,
29,30,31,32]
  saidas do bloco:[nenable,p7k,penviol,reset,rpoli]
  entradas do bloco:[cdviol,drte,nteste,penable,prte,q1,
q10,q11,q12,q13,q2,q3,q4,q5,q6,q7,q8,q9]


     ####   Realimenta"c-c""oes detetadas:


  ####        Fim de relatorio


%%%%%%%%%%%%%%%%%%%     INVI

ffd(1,bciva,',bciv,ra).
ffd(2,bcivaa,pen,bciva,ra).
mxffd(n,3,iciv,niciv,nre,bcivaa,ntest,ra,vdd).

nand(4,nen,[bciva,pen]).
nand(5,pnen,[niciv,nen])
nand(6,nres,[nen"c-c"divt]).
nand(7,nitres,[bctl1,itres]).
nand(8,nicin,[bctl2,icin])
nand(9"c-c"divt,[nitres,nicin]).
and(10,nre,[pnen,nres])

pi([ntest,bciv,ra,bctl1,itres,bctl2,icin]).

po([iciv]).

modo'scan([[ntest,0]]).

clock([ra]).
   ###   Caminhos identificados/verificados:


 SCAN :   0
 Numero de instancias sequenciais do scan:  3
 lista de sinais de clock:  [ra]
 lista de sinais de reset:  [vdd]
 lista de sinais de selecao:  [ntest,vdd]
 instancias do scan 0:[1,2,3]

 SCANIN atraves de :   bciv  bciv

 SCANOUT atraves de :   iciv  `125515

 Modo scan dado pelos sinais e respectivos niveis
listados a seguir:

 [ntest,0] com [ntest,0]

 Niveis associados com o scan 0:


################


  ###   Possivel saida de scan atraves da instancia 3
 iciv  `387670

  ###   Logica Combinacional em blocos:

Instancias combinacionais do CI verificado contidas em
 blocos:
   [4,5,6,7,8,9,10]
```

```
Instancias combinacionais do CI verificado e  nao
 contidas em outras :
    [[4,5,6,7,8,9,10]]

Blocos combinacionais:
instancias do bloco:[4,5,6,7,8,9,10]
  saidas do bloco:[nen,nre]
  entradas do bloco:[bciva,bctl1,bctl2,icin,itres,niciv,
pen]

      ####   Realimenta"c-c^"oes detetadas:


   ####          Fim de relatorio


%%%% CONFRA  %%%%%

mxffd(n,1,qspa1,'',nreali,etes,ntest,ra,nrese).
ffdr(n,2,qspa2,'',qspa1,ra,nrese).
ffdr(n,3,qspa3,'',qspa2,ra,nrese).
ffdr(n,4,qspa4,'',qspa3,ra,nrese).
ffdr(n,5,qspa5,'',qspa4,ra,nrese).
ffdr(n,6,qspa6,'',qspa5,ra,nrese).
ffdr(n,7,qspa7,'',qspa6,ra,nrese).
ffdr(n,8,qspa8,'',qspa7,ra,nrese).
ffdr(n,9,qspa9,'',qspa8,ra,nrese).

mxffd(n,10,stes,nstes,da10,qspa9,ntest,ra,vdd).


nand(11,nrese,[prese]).
nor(12,prese,[ptest,stes]).
nand(13,ptest,[ntest]).
nor(14,da10,[sicd,pp]).
and(15,sicd,[si34q"c-c^d5d]).
and(16,pp,[p127,p345689]).
nand(17,nra,[ra]).
and(18"c-c^sinc,[nra,sicd,pp]).
exnor(19,nreali,[qspa4,qspa9]).
and(20,p345689,[qspa3,qspa4,qspa5,qspa6,qspa8,qspa9]).
nor(21,p127,[qspa7,qspa2,qspa1]).
pi([ntest,etes,si34q"c-c^d5d,ra,vdd]).
po([stes"c-c^sinc]).


modo'scan([[ntest,0]]).  %%%% OBS: E uma lista de pares

clock([ra]).


   ###
   ###                 SMAuT
   ###
   ###   Sistema Modular de Auditoria de Testabilidade
   ###
   ###   Prototipo realizado por: Bernadete A.L.Oliveira
   ###
   ###
   ###   Circuito verificado:    confra
   ###
   ###   Caminhos identificados/verificados:


 SCAN :  0
 Numero de instancias sequenciais do scan:  10
 lista de sinais de clock:  [ra]
 lista de sinais de reset:  [nrese,vdd]
 lista de sinais de selecao:  [ntest,vdd]
 instancias do scan 0:[1,2,3,4,5,6,7,8,9,10]

 SCANIN atraves de :   etes  etes

 SCANOUT atraves de :   stes  '127221

 Modo scan dado pelos sinais e respectivos niveis
listados a seguir:

 [ntest,0] com [ntest,0]

 ret:[nrese,1] com [ntest,0]

 Niveis associados com o scan 0:


##################


   ###   Possivel saida de scan atraves da instancia 10:
stes  '387780

   ###   Logica Combinacional em blocos:
```

```
Instancias combinacionais do CI verificado contidas em
 blocos:
    [11,12,13,14,15,16,17,18,19,20,21]

Instancias combinacionais do CI verificado e  nao
 contidas em outras :
    [[11,12,13],[14,15,16,17,18,19,20,21]]

Blocos combinacionais:
instancias do bloco:[11,12,13]
  saidas do bloco:[nrese]
  entradas do bloco:[ntest,stes]

instancias do bloco:[14,15,16,17,18,19,20,21]
  saidas do bloco:[csinc,da10,nreali,pp,sicd]
  entradas do bloco:[cd5d,qspa1,qspa2,qspa3,qspa4,qspa5,
qspa6,qspa7,qspa8,qspa9,ra,si34q]

      ####   Realimenta"c-c^"oes detetadas:


   ####          Fim de relatorio

%%%%%%%%%        trmem       %%%%%%%%
%%CCT ACTRMEM
%% saidas
%%(plenesc,nenable,plepet"c-c^trm"c-c^trmux2,
%%nitb1c,pitb2c,nitb6c,nitb8c,nitb7l,
%% entradas
%%nponvoz,nmepet,pglec,p2ml,p2ml1,p2m1c)
%%**
%%AA025
     and(2,plec,[p2m1c,pglec]).
     and(3,pa8,[pi8,p2m1c]).
     and(4,pleit1,[p2m1c,pitb1c]).
     and(5,pleit2,[pitb2c,p2m1c]).
     and(6,nescet,[po17,nitb6c]).
     and(7"c-c^trmux2,[nmepet,pmuxet]). %       **v04,v05
     and(29,nmuxet,[nitb1c,nitb8c]). %;         **v04
%%DF0F5
ffd(26,nescmei,'',nitb7l,p2ml1).  %;
%%IN015
 nand(31,pi8,[nescet])
     nand(41,pitb1c,[nitb1c]).
     nand(34,pmuxet,[nmuxet]).  %;                **v05
%%mux
 mx(10,nenable,[nename,nenaet],[nmepet]).  %
     mx(22,plenesc,[nescmei,nescet],[nmepet]).
     mx(32"c-c^trm,[nescmei,nmuxet],[nmepet]).  %;
%%NO025
nor(20,pesc,[p2ml,nitb7l]).
     nor(30,nename,[pesc,plec]).
     nor(40,plepet,[pleit1,pleit2]).  %;
%%NO035
 nor(50,nenaet,[pleit1,pleit2,pa8]).  %.
%%OR025
 or(21,po17,[nponvoz,nitb8c]).  %.
%%INPUT
pi([nitb1c,pitb2c,nitb6c,nitb8c,nitb7l,nponvoz,nmepet,
pglec,p2ml,
p2ml1,p2m1c]).
%%.


   ###
   ###                 SMAuT
   ###
   ###   Sistema Modular de Auditoria de Testabilidade
   ###
   ###   Prototipo realizado por: Bernadete A.L.Oliveira
   ###
   ###
   ###   Circuito verificado:    ctrmem
   ###
   ###   Caminhos identificados/verificados:
   ###   Logica Combinacional em blocos:

Instancias combinacionais do CI verificado contidas em
blocos:
    [2,3,4,5,6,7,20,21,29,30,31,34,40,41,50]

Instancias combinacionais do CI verificado e  nao
contidas em outras :
    [10,22,32,[2,3,4,5,6,7,20,21,29,30,31,34,40,41,50]]

Blocos combinacionais:
instancias do bloco:[2,3,4,5,6,7,20,21,29,30,31,34,40,41,
50]
  saidas do bloco:[ctrmux2,nenaet,nename,nescet,nmuxet,
pa8,pleit1,pleit2,plepet]
  entradas do bloco:[nitb1c,nitb6c,nitb7l,nitb8c,nmepet,
nponvoz,p2m1c,p2ml,pa8,pglec,pitb2c]

instancias do bloco:[2,3,4,5,6,7,20,21,29,30,31,34,40,41,
50]
  saidas do bloco:[ctrmux2,nenaet,nename,nescet,nmuxet,
```

```
pa8,pleit1,pleit2,plepet]
  entradas do bloco:[nitb1c,nitb6c,nitb7l,nitb8c,nmepet,
nponvoz,p2m1c,p2ml,pa8,pglec,pitb2c]

instancias do bloco:[2,3,4,5,6,7,20,21,29,30,31,34,40,41,
50]
  saidas do bloco:[ctrmux2,nenaet,nename,nescet,nmuxet,
pa8,pleit1,pleit2,plepet]
  entradas do bloco:[nitb1c,nitb6c,nitb7l,nitb8c,nmepet,
nponvoz,p2m1c,p2ml,pa8,pglec,pitb2c]


      ####   Realimenta"c-c^"oes detetadas:


   ####         Fim de relatorio



   ###                  SMAuT
   ###    Sistema Modular de Auditoria de Testabilidade
   ###    Prototipo realizado por: Bernadete A.L.Oliveira
   ###
   ###          Arquivo de falhas
   ###
   ###    CI analisado:   ctrmem
   ###
   ###    Elementos de circuito fora das regras:
   ###    Clocks nao controlaveis:
   ### Celulas sequenciais fora de scan:
   ###    Instancias sequenciais fora de qualquer scan:
  instancia'tipo,numero: ffd,26

   ###    Falhas de scanin:


   ###    Falhas de scanout:


   ####         Fim de relatorio


%%%%%%%%%%    combrx    %%%%
%** *  NX-HILO Version 1.35
%** *  Copyright (C]) 1985,1987 -
Personal CAD Systems,Inc.
%** *  File In  : COMBRX.NLT
%** *  File Out : COMBRX.HDL
%  data : 12 - 04 - 91
%CCT
%COMBRX(MRX,PMER,PMDR,WF,XF,YF,ZF,MXR,MRC,
PRDMOR,PRES,
QW,QX,QY,QZ)
po([mrx,pmer,pmdr,wf,xf,yf,zf]).
pi([mxr,mrc,prdmor,pres,qw,qx,qy,qz]).
nand(61,nqw,[qw]).
nand(3,g3ny,[qw,qy,nqz,nmxr,prdmor]).
nand(6,g6ny,[qw,qx,qy,nqz,nmxr]).
nand(21,g21ny,[qw,nqy]).
nand(46,g46ny,[g12ny,g42ny]).
nand(35,g35ny,[g24ny,g25ny]).
nor(5,g5ny,[qw,nqx,nqy,qz,nmxr])
nand(68,i8nb,[g24ny]).
nand(7,g7ny,[qw,nqx,nqy,mxr])
nand(44,g44ny,[g12ny,g40ny])
nand(23,g23ny,[nqw,nqy,nmrc]).
nand(27,g27ny,[qw,qy]).
nand(32,g32ny,[g7ny,g14ny]).
nand(31,g31ny,[g23ny,g27ny]).
nand(40,g40ny,[i8nb,g26ny])
nand(14,g14ny,[qw,nqx,qz,mxr]).
nor(22,g22ny,[qx,qz,nmxr]).
nand(43,g43ny,[g39ny,g12ny]).
nand(4,g4ny,[nqx,qy,qz,nmxr,amrc]).
nand(66,nmrc,[mrc]).
nand(13,g13ny,[qw,nqx,qy,qz,nmxr]).
nand(33,pmdr,[g4ny,g13ny]).
nand(11,g11ny,[qw,qy,nqz,nmxr,mrc]).
nand(34,g34ny,[g14ny,g13ny,g11ny]).
nand(19,g19ny,[qx,mxr]).
nand(65,nmxr,[mxr]).
nand(63,nqy,[qy]).
nand(18,g18ny,[nqy,qz]).
nand(17,g17ny,[qx,nqy]).
nand(16,g16ny,[qx,qz]).
nand(15,g15ny,[nqw,mrc]).
nand(10,g10ny,[qw,nqx,nmxr,nrdmor]).
nand(62,nqx,[qx]).
nand(64,nqz,[qz]).
or(49,yf,[g34ny,g45ny]).
nand(45,g45ny,[g41ny,g8ny]).
nand(67,nrdmor,[prdmor]).
nand(41,g41ny,[g35ny,g26ny]).
nor(26,g26ny,[qw,qx,qz,mrc]).
nand(8,g8ny,[qy,nqz,nmxr,nmrc]).
nand(37,g37ny,[g18ny,g19ny,g21ny]).
nand(24,g24ny,[qy,mxr]).
nand(25,g25ny,[nqy,nmxr,prdmor]).
nand(39,g39ny,[g22ny,g31ny])
nand(36,g36ny,[g10ny,g15ny,g16ny,g17ny]).
nand(12,g12ny,[nqw,nqx,qy,mxr,qz]).
nor(28,g28ny,[qx,qw,qx,qy,mrc]).
nand(29,g29ny,[prdmor,nmxr]).
nand(38,g38ny,[nmxr,g29ny]).
nand(42,g42ny,[g28ny,g38ny]).
nand(30,xf,[g3ny,g6ny]).
or(51,mrx,[g36ny,g34ny,g37ny,g46ny]).
nor(52,zf,[pres,g48ny]).
or(50,wf,[g43ny,g34ny,g36ny,g37ny]).
nor(48,g48ny,[g32ny,g44ny]).
or(47,pmer,[g5ny,g43ny]).


   ###
   ###                  SMAuT
   ###
   ###    Sistema Modular de Auditoria de Testabilidade
   ###
   ###    Prototipo realizado por  Bernadete A.L.Oliveira
   ###
   ###
   ###    Circuito verificado:    combrx
   ###
   ###    Caminhos identificados/verificados:
   ###    Logica Combinacional em blocos.

Instancias combinacionais do CI verificado contidas em
 blocos:
     [3,4,5,6,7,8,10,11,12,13,14,15,16,17,18,19,21,22,23,
24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,
43,44,45,46,47,48,49,50,51,52,61,62,63,64,65,66,67,68]

Instancias combinacionais do CI verificado e  nao
contidas em outras :
     [[3,4,5,6,7,8,10,11,12,13,14,15,16,17,18,19,21,22,23,
24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,
43,44,45,46,47,48,49,50,51,52,61,62,63,64,65,66,67,68]]


Blocos combinacionais:
instancias do bloco:[3,4,5,6,7,8,10,11,12,13,14,15,16,17,
18,19,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,61,62,63,64,
65,66,67,68]
  saidas do bloco:[g10ny,g12ny,g13ny,g14ny,g24ny,g26ny,
g31ny,g34ny,g36ny,g37ny,g39ny,g40ny,g41ny,g42ny,g43ny,
g44ny,mrx,nmrc,nmxr,nqw,nqx,nqy,nqz,pmdr,pmer,wf,xf,yf,zf]
  entradas do bloco:[g10ny,g12ny,g13ny,g24ny,g26ny,g31ny,
g34ny,g39ny,g40ny,g41ny,g42ny,g44ny,mrc,mxr,nmrc,nmxr,nqw,
nqx,nqy,nqz,prdmor,pres,qw,qx,qy,qz]

instancias do bloco:[3,4,5,6,7,8,10,11,12,13,14,15,16,17,
18,19,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,61,62,63,64,
65,66,67,68]
  saidas do bloco:[g10ny,g12ny,g13ny,g14ny,g24ny,g26ny,
g31ny,g34ny,g36ny,g37ny,g39ny,g40ny,g41ny,g42ny,g43ny,
g44ny,mrx,nmrc,nmxr,nqw,nqx,nqy,nqz,pmdr,pmer,wf,xf,yf,zf]
  entradas do bloco:[g10ny,g12ny,g13ny,g24ny,g26ny,g31ny,
g34ny,g39ny,g40ny,g41ny,g42ny,g44ny,mrc,mxr,nmrc,nmxr,nqw,
nqx,nqy,nqz,prdmor,pres,qw,qx,qy,qz]

instancias do bloco:[3,4,5,6,7,8,10,11,12,13,14,15,16,17,
18,19,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,61,62,63,64,
65,66,67,68]
  saidas do bloco:[g10ny,g12ny,g13ny,g14ny,g24ny,g26ny,
g31ny,g34ny,g36ny,g37ny,g39ny,g40ny,g41ny,g42ny,g43ny,
g44ny,mrx,nmrc,nmxr,nqw,nqx,nqy,nqz,pmdr,pmer,wf,xf,yf,zf]
  entradas do bloco:[g10ny,g12ny,g13ny,g24ny,g26ny,g31ny,
g34ny,g39ny,g40ny,g41ny,g42ny,g44ny,mrc,mxr,nmrc,nmxr,nqw,
nqx,nqy,nqz,prdmor,pres,qw,qx,qy,qz]

instancias do bloco:[3,4,5,6,7,8,10,11,12,13,14,15,16,17,
18,19,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,61,62,63,64,
65,66,67,68]
  saidas do bloco:[g10ny,g12ny,g13ny,g14ny,g24ny,g26ny,
g31ny,g34ny,g36ny,g37ny,g39ny,g40ny,g41ny,g42ny,g43ny,
g44ny,mrx,nmrc,nmxr,nqw,nqx,nqy,nqz,pmdr,pmer,wf,xf,yf,zf]
  entradas do bloco:[g10ny,g12ny,g13ny,g24ny,g26ny,g31ny,
g34ny,g39ny,g40ny,g41ny,g42ny,g44ny,mrc,mxr,nmrc,nmxr,nqw,
nqx,nqy,nqz,prdmor,pres,qw,qx,qy,qz]

instancias do bloco:[3,4,5,6,7,8,10,11,12,13,14,15,16,17,
18,19,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,61,62,63,64,
65,66,67,68]
  saidas do bloco:[g10ny,g12ny,g13ny,g14ny,g24ny,g26ny,
g31ny,g34ny,g36ny,g37ny,g39ny,g40ny,g41ny,g42ny,g43ny,
g44ny,mrx,nmrc,nmxr,nqw,nqx,nqy,nqz,pmdr,pmer,wf,xf,yf,zf]
  entradas do bloco:[g10ny,g12ny,g13ny,g24ny,g26ny,g31ny,
g34ny,g39ny,g40ny,g41ny,g42ny,g44ny,mrc,mxr,nmrc,nmxr,nqw,
nqx,nqy,nqz,prdmor,pres,qw,qx,qy,qz]
```

```
instancias do bloco:[3,4,5,6,7,8,10,11,12,13,14,15,16,17,
18,19,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,61,62,63,64,
65,66,67,68]
  saidas do bloco:[g10ny,g12ny,g13ny,g14ny,g24ny,g26ny,
g31ny,g34ny,g36ny,g37ny,g39ny,g40ny,g41ny,g42ny,g43ny,
g44ny,mrx,nmrc,nmxr,nqw,nqx,nqy,nqz,pmdr,pmer,wf,xf,yf,zf]
  entradas do bloco:[g10ny,g12ny,g13ny,g24ny,g26ny,g31ny,
g34ny,g39ny,g40ny,g41ny,g42ny,g44ny,mrc,mxr,nmrc,nmxr,nqw,
nqx,nqy,nqz,prdmor,pres,qw,qx,qy,qz]

instancias do bloco:[3,4,5,6,7,8,10,11,12,13,14,15,16,17,
18,19,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,61,62,63,64,
65,66,67,68]
  saidas do bloco:[g10ny,g12ny,g13ny,g14ny,g24ny,g26ny,
g31ny,g34ny,g36ny,g37ny,g39ny,g40ny,g41ny,g42ny,g43ny,
g44ny,mrx,nmrc,nmxr,nqw,nqx,nqy,nqz,pmdr,pmer,wf,xf,yf,zf]
  entradas do bloco:[g10ny,g12ny,g13ny,g24ny,g26ny,g31ny,
g34ny,g39ny,g40ny,g41ny,g42ny,g44ny,mrc,mxr,nmrc,nmxr,nqw,
nqx,nqy,nqz,prdmor,pres,qw,qx,qy,qz]

instancias do bloco:[3,4,5,6,7,8,10,11,12,13,14,15,16,17,
18,19,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,61,62,63,64,
65,66,67,68]
  saidas do bloco:[g10ny,g12ny,g13ny,g14ny,g24ny,g26ny,
g31ny,g34ny,g36ny,g37ny,g39ny,g40ny,g41ny,g42ny,g43ny,
g44ny,mrx,nmrc,nmxr,nqw,nqx,nqy,nqz,pmdr,pmer,wf,xf,yf,zf]
  entradas do bloco:[g10ny,g12ny,g13ny,g24ny,g26ny,g31ny,
g34ny,g39ny,g40ny,g41ny,g42ny,g44ny,mrc,mxr,nmrc,nmxr,nqw,
nqx,nqy,nqz,prdmor,pres,qw,qx,qy,qz]


      ####    Realimenta~c~c~~oes detetadas:
reali(nmxr).

reali(g34ny).

reali(nqz).



   ####          Fim de relatorio

   ###                SMAuT
   ###    Sistema Modular de Auditoria de Testabilidade
   ###    Prototipo realizado por: Bernadete A.L.Oliveira
   ###
   ###          Arquivo de falhas
   ###
   ###    CI analisado:   combrx
   ###
   ###    Elementos de circuito fora das regras:
   ###    Clocks nao controlaveis:
   ### Celulas sequenciais fora de scan:
   ###    Instancias sequenciais fora de qualquer scan:

   ###    Falhas de scanin:


   ###    Falhas de scanout:


   ####          Fim de relatorio

%%%%%%%%%%%%%    exemplo      %%
%instancias
%inst([1:52])


%modos de opera~c~c~~ao(teste)
modo scan([[n1,0],[reex,1],[sa20,1],[nen,0],[nte,0],
[nrst,1]])

%pinos de entrada
pi([nen,sel1,sel2,date20,date,nte,nrst,nt,si~c~c~k,vdd,
reex,sa20]).

%pinos de saida
po([q52,so,so2,so3]).

%pinos bi-direcionais
pio([so4,so5]).

%iobuf(n,instancia,Y,Q,A,NEN)
iobuf(n,43,so4,qso4,q42,nen).

%tribuffers
tribuf(n,44,da20,date20,nen).
tribuf(n,45,so5,q39,nen).


%ffd(instancia,q,qn,d~c~c~k)
ffd(9,s6,ns6,q21~c~c~k).
ffd(19,q19,nq19,d19~c~c~k). %%%% clock ns0 antes
ffd(34,q34,'',d34~c~c~k).
ffd(28,q28,'',d28~c~c~k)
ffd(31,q31,'',d31~c~c~k).
ffd(38,q38,'',q20~c~c~k).
ffd(40,q40,'',d40~c~c~k).
ffd(41,q41,'',q40~c~c~k).
ffd(42,q42,'',q41~c~c~k).
ffd(46,da47,'',da46~c~c~k).
ffd(47,da48,'',da47~c~c~k).
ffd(48,da49,'',da48~c~c~k).
ffd(49,q49,'',da49~c~c~k).

%ffdr(n,instancia,q,qn,d~c~c~lk,rst)
ffdr(n,2,s2,ns2,s8,s4,vdd).
ffdr(n,39,q39,'',q38~c~c~k,vdd).

%mx21(instancia,q,a,b,sel)
mx21(11,sia,[sim,s9],[nt]).
mx21(51,d40,[a51,b51],[sel1]).

mx41(52,q52,[q19,nq19,d19,ns0],[sel1,sel2]).
mx41(35,so2,[q31,qb35,qc35,qd35],[rst30,pt]).

%bufl(instancia,q,a)
bufl(5,s4~c~c~k).
bufl(6,s5~c~c~k).
bufl(7,s6,reex).
bufl(13,sim,si).
bufl(34,ntb24,nt).
ibufl(36,nte36,nte).
ibufl(26,da27,date).

%obufl(instancia,q,a).
obufl(15,so,sob).

%mxffd(n,instancia,q,qn,da,db,sel~c~c~lk,rst)
mxffd(n,8,s0,ns0,sia,nsa,pt~c~c~k,vdd).
mxffd(n,20,q20,nq20,da20,db20,sa20~c~c~k,vdd).
mxffd(n,21,q21,nq21,s1,db21,pt21~c~c~k,vdd).
mxffd(n,27,d28,'',da27,q34,pte36~c~c~k,vdd).
mxffd(n,30,d31,'',q28,b29,sa20~c~c~k,vdd).
mxffd(n,3,sob,ns3,sia,s2,nsa,s5,s6).
mxffd(n,1,s1,ns1,si,s12a,pt21~c~c~k,nre).

%inversor(instancia,q,a)
nbuf(10,pt,nnt).
nbuf(16,nnt,ptt).
nbuf(17,ptt,nt).
nbuf(22,pt21,nt).
nbuf(37,pte36,nte36).

%nandX(instancia,q,[entradas])
nand(18,s12a,[a18,s0]).
nand(23,nre,[ns3,ntb24]).

%orX(instancia,q,[entradas])
or(33,rst30,[sia,nrst,si33]).
or(32,so3,[s32,d31]).


%andX(instancia,Q,[entradas])
and(50,nsa,[ck,reex]).


   ###
   ###                SMAuT
   ###
   ###    Sistema Modular de Auditoria de Testabilidade
   ###
   ###    Prototipo realizado por: Bernadete A.L.Oliveira
   ###
   ###
   ###    Circuito verificado:    exemplo
   ###
   ###    Caminhos identificados/verificados:


SCAN :  4
Numero de instancias sequenciais do scan:  5
lista de sinais de clock:  [ck,s4,s5]
lista de sinais de reset:  [nre,so,vdd]
lista de sinais de selecao:  [nsa,pt21,vdd]
instancias do scan 4:[1,21,9,2,3,15]

SCANIN atraves de :  si  si

SCANOUT atraves de :  sob  so

 Modo scan dado pelos sinais e respectivos niveis
listados a seguir:

[nsa,0] com [reex,0]

[pt21,1] com [nt,0]
```

```
 rst:[nre,1]  com [nt,0]

 rst:[s6,1]  com [reex,1]

 Niveis associados com o scan 4:

---


 SCAN :   3
 Numero de instancias sequenciais do scan:  4
 lista de sinais de clock:  [ck]
 lista de sinais de reset:  [vdd]
 lista de sinais de selecao:  [vdd]
 instancias do scan 3:[46,47,48,49]


 SCANOUT atraves de :   49  q49

 Modo scan dado pelos sinais e respectivos niveis
listados a seguir:

 Niveis associados com o scan 3:

---


 SCAN :   2
 Numero de instancias sequenciais do scan:  4
 lista de sinais de clock:  [ck]
 lista de sinais de reset:  [vdd]
 lista de sinais de selecao:  [pt,pte36,rst30,sa20,vdd]
 instancias do scan 2:[27,28,30,31,35]

 SCANIN atraves de :   da27  date

 SCANOUT atraves de :   so2  '129165

 Modo scan dado pelos sinais e respectivos niveis
listados a seguir:

 [pt,1]  com [nt,0]

 [pte36,1]  com [nte,0]

 [rst30,1]  com [nrst,1]

 [sa20,1]  com [sa20,1]

 Niveis associados com o scan 2:

---


 SCAN :   1
 Numero de instancias sequenciais do scan:  3
 lista de sinais de clock:  [ck]
 lista de sinais de reset:  [vdd]
 lista de sinais de selecao:  [nen,vdd]
 instancias do scan 1:[40,41,42,43]


 SCANOUT atraves de :   so4  '129109

 Modo scan dado pelos sinais e respectivos niveis
listados a seguir:


 Niveis associados com o scan 1:

 instancia: 43  nen  n


---


 SCAN :   0
 Numero de instancias sequenciais do scan:  3
 lista de sinais de clock:  [ck]
 lista de sinais de reset:  [vdd]
 lista de sinais de selecao:  [nen,sa20,vdd]
 instancias do scan 0:[20,38,39,45]

 SCANIN atraves de :   da20  date20

 SCANOUT atraves de :   so5  '129156

 Modo scan dado pelos sinais e respectivos niveis
listados a seguir:

 [sa20,1]  com [sa20,1]

 Niveis associados com o scan 0:

 instancia: 45  nen  n

---


  ###  Possivel saida de scan atraves da instancia 3:
sob  so


  ###  Possivel saida de scan atraves da instancia 39:
q39  so5


  ###  Possivel saida de scan atraves da instancia 42:
q42  so4

  ###  Logica Combinacional em blocos:

Instancias combinacionais do CI verificado contidas em
blocos:
    [10,16,17,22,23,24]

Instancias combinacionais do CI verificado e  nao
contidas em outras :
    [5,6,7,11,13,15,18,26,32,33,35,36,37,43,44,45,50,51,
52,[10,16,17,
22,23,24]]

Blocos combinacionais:
instancias do bloco:[10,16,17,22,23,24]
 saidas do bloco:[nre,pt,pt21]
 entradas do bloco:[ns3,nt]


     ####  Realimenta"c~c""oes detetadas:


  ####        Fim de relatorio

  ###              SMAuT
  ###  Sistema Modular de Auditoria de Testabilidade
  ###  Prototipo realizado por: Bernadete A.L.Oliveira
  ###
  ###        Arquivo de falhas
  ###
  ###  CI analisado:   exemplo
  ###
  ###  Lista de sinais de clock verificada:
       SCAN(4) :   [ck,s4,s5]

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal s4, numero de buffers: 1
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal s5, numero de buffers: 1
 e para o sinal ck, numero de buffers:0


  ###  Elementos de circuito fora das regras:
  ###  Clocks nao controlaveis:


 ### ERRO  ###: Clock alimenta entrada de dados da
instancia:  8
 sinal de clock: ck  entrada de dados:  nsa


  ### C~elulas sequenciais fora de scan:
  ###  Instancias sequenciais fora de qualquer scan:
    instancia`tipo,numero:  mxffd,8
    instancia`tipo,numero:  ffd,34
    instancia`tipo,numero:  ffd,19


 ### Detetado por cla'elo'mxa = instancia   19
 ### observavel mas nao controlavel ###

 ### Substituir as seguintes instancias por MXFFD:
ffd  40  e  MUX : 51
 ### Substituir as seguintes instancias por MXFFD:
mxffd  8  e  MUX : 11
```

```
   ###   Falhas de scanin:

   ### Detetada falha de scanin por cla'sc'scin:
     SCAN:  3     instancia da celula:  46
     Possiveis entradas de scan:  da46
   ### Detetada falha de scanin por cla'sc'scin:
     SCAN:  1     instancia da celula:  40
     Possiveis entradas de scan:  d40

   ###   Falhas de scanout:

   ### Detetada falha de scanout por cla'sc'scout:
     SCAN:  3     instancia da celula:  49
      ### Possivel saida de scan:  q49


   ####          Fim de relatorio

%%%%%%%%%     clktree      %%%%%%%%%%%%%%%%%%
%arvore de clock para teste de pesqclk

bufl(1"c-c"kc"c-c"kb).
bufl(2"c-c"kd"c-c"kb).
bufl(3"c-c"kf"c-c"ke).
bufl(4"c-c"ke"c-c"ka).
bufl(5"c-c"kb"c-c"ka).
bufl(6"c-c"ka"c-c"k).
bufl(7"c-c"ki"c-c"kg).
bufl(8"c-c"kg"c-c"k).

clock([ck"c-c"ka"c-c"kb"c-c"kc"c-c"kd,
       cke"c-c"kf"c-c"kg"c-c"ki]).
pi([ck]).

ffd(10,q10,',d10"c-c"kc).
ffd(11,q11,',q10"c-c"kd).
ffd(12,q12,',q11"c-c"kf).
ffd(13,q13,',q12"c-c"ki).
ffd(14,q14,',q13"c-c"kb)
ffd(15,q15,',q14"c-c"ke).
ffd(16,q16,',q15"c-c"kg).
ffd(17,q17,',q16"c-c"ka).
ffd(18,q18,',q17"c-c"k).

   ###
   ###                 SMAuT
   ###
   ###   Sistema Modular de Auditoria de Testabilidade
   ###
   ###   Prototipo realizado por: Bernadete A.L.Oliveira
   ###
   ###
   ###   Circuito verificado:    clktree
   ###
   ###   Caminhos identificados/verificados:


  SCAN :  0
  Numero de instancias sequenciais do scan:  9
  lista de sinais de clock:  [ck"c-c"ka"c-c"kb"c-c"kc"c-c"kd"c-c"ke,
  "kf"c-c"kg"c-c"ki]
  lista de sinais de reset:  [vdd]
  lista de sinais de selecao:  [vdd]
  instancias do scan 0:[10,11,12,13,14,15,16,17,18]


  SCANOUT atraves de :   18  q18

  Modo scan dado pelos sinais e respectivos niveis
listados a seguir:

  Niveis associados com o scan 0:


—

   ###   Logica Combinacional em blocos:

Instancias combinacionais do CI verificado contidas em
blocos:
    [1,2,3,4,5,6,7,8]

Instancias combinacionais do CI verificado e  nao
contidas em outras :
    [[1,2,3,4,5,6,7,8]]

Blocos combinacionais:
instancias do bloco:[1,2,3,4,5,6,7,8]
  saidas do bloco:[cka"c-c"kb"c-c"kc"c-c"kd,
                   cke"c-c"kf"c-c"kg"c-c"ki]
  entradas do bloco:[ck]

instancias do bloco:[1,2,3,4,5,6,7,8]
  saidas do bloco:[cka"c-c"kb"c-c"kc"c-c"kd,
                   cke"c-c"kf"c-c"kg"c-c"ki]
  entradas do bloco:[ck]

   ####   Realimenta"c-c""oes detetadas.


   ####          Fim de relatorio

   ###                 SMAuT
   ###   Sistema Modular de Auditoria de Testabilidade
   ###   Prototipo realizado por: Bernadete A.L.Oliveira
   ###
   ###          Arquivo de falhas
   ###
   ###   CI analisado:    clktree
   ###
   ###   Lista de sinais de clock verificada:
         SCAN(0) :   [ck"c-c"ka"c-c"kb"c-c"kc,
                      ckd"c-c"ke"c-c"kf"c-c"kg,
                      cki]

### mesma fase e possivel distorcao de clock.
 Pino associado aos sinais de clock: ck

 Para o sinal cka, numero de buffers: 1
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock
 Pino associado aos sinais de clock: ck

 Para o sinal ckb, numero de buffers: 2
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal ckc, numero de buffers: 3
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal ckd, numero de buffers: 3
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal cke, numero de buffers: 2
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock.
 Pino associado aos sinais de clock: ck

 Para o sinal ckf, numero de buffers: 3
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock
 Pino associado aos sinais de clock: ck

 Para o sinal ckg, numero de buffers: 1
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock.
 Pino associado aos sinais de clock: ck

 Para o sinal cki, numero de buffers: 2
 e para o sinal ck, numero de buffers:0


   ###   Elementos de circuito fora das regras.
   ###   Clocks nao controlaveis:
   ### Celulas sequenciais fora de scan:
   ###   Instancias sequenciais fora de qualquer scan:

   ###   Falhas de scanin:

   ### Detetada falha de scanin por cla'sc'scin.
     SCAN:  0     instancia da celula:  10
     Possiveis entradas de scan:  d10

   ###   Falhas de scanout:

   ### Detetada falha de scanout por cla'sc'scout:
     SCAN:  0     instancia da celula:  18
      ### Possivel saida de scan:  q18


   ####          Fim de relatorio

%%%%%%     clknbuf     %%%%%%%%%%%
%

ibufl(1"c-c"k1"c-c"k).
bufl(2"c-c"k2"c-c"k1).
bufl(4"c-c"k4"c-c"k3).
```

```
buf1(7"c-c"k7"c-c"k2).
buf1(6"c-c"k6"c-c"k7).
nbuf(3"c-c"k3"c-c"k2).
nbuf(5"c-c"k5"c-c"k3).
nbuf(8"c-c"k8"c-c"k1).
nbuf(9"c-c"k9"c-c"k8).
nbuf(10"c-c"k10"c-c"k9).
pi([ck]).
ffd(10,q10,',d10"c-c"k3).
ffd(11,q11,',q10"c-c"k4).
ffd(12,q12,',q11"c-c"k6).
ffd(13,q13,',q12"c-c"k9).
ffd(14,q14,',q13"c-c"k2).
ffd(15,q15,',q14"c-c"k5).
ffd(16,q16,',q15"c-c"k8).
ffd(17,q17,',q16"c-c"k1).
ffd(18,q18,',q17"c-c"k).
```

```
  ###
  ###                SMAuT
  ###
  ###   Sistema Modular de Auditoria de Testabilidade
  ###
  ###   Prototipo realizado por: Bernadete A.L.Oliveira
  ###
  ###
  ###   Circuito verificado:    :lknbuf
  ###
  ###   Caminhos identificados/verificados:

 SCAN :   0
 Numero de instancias sequenciais do scan:  9
 lista de sinais de clock:  [ck"c-c"k1"c-c"k2,
ck3"c-c"k4"c-c"k5"c-c"k6,
ck8"c-c"k9]
 lista de sinais de reset:  [vdd]
 lista de sinais de selecao:  [vdd]
 instancias do scan 0:[10,11,12,13,14,15,16,17,18]

 SCANOUT atraves de :   18  q18

 Modo scan dado pelos sinais e respectivos niveis
listados a seguir:

 Niveis associados com o scan 0:

--

  ###   Logica Combinacional em blocos:

Instancias combinacionais do CI verificado contidas em
blocos:
    [2,3,4,5,6,7,8,9,10]

Instancias combinacionais do CI verificado e  nao
contidas em outras :
    [1,[2,3,4,5,6,7,8,9,10]]

Blocos combinacionais:
instancias do bloco:[2,3,4,5,6,7,8,9,10]
 saidas do bloco:[ck10"c-c"k2"c-c"k3"c-c"k4"c-c"k5"c-c"k6"c-c"k8"c-c"k9]
 entradas do bloco:[ck1]

instancias do bloco:[2,3,4,5,6,7,8,9,10]
 saidas do bloco:[ck10"c-c"k2"c-c"k3"c-c"k4"c-c"k5"c-c"k6"c-c"k8"c-c"k9]
 entradas do bloco:[ck1]

    ####   Realimenta"c-c""oes detetadas:

 ####        Fim de relatorio

  ###              SMAuT
  ###   Sistema Modular de Auditoria de Testabilidade
  ###   Prototipo realizado por: Bernadete A.L.Oliveira
  ###
  ###         Arquivo de falhas
  ###
  ###   CI analisado:   :lknbuf
  ###
  ###   Lista de sinais de clock verificada:
      SCAN(0) :   [ck"c-c"k1"c-c"k2,
ck3"c-c"k4"c-c"k5"c-c"k6"c-c"k8,
ck9]

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal ck1, numero de buffers: 1
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal ck2, numero de buffers: 2
 e para o sinal ck, numero de buffers:0

### ERRO ###: diferentes fases de clock no caminho
 Pino associado aos sinais de clock: ck
 Para o sinal ck3 numero de buffers inversores: 1
  e para o sinal ck, numero de buffers inversores:0

### ERRO ###: diferentes fases de clock no caminho
 Pino associado aos sinais de clock: ck
 Para o sinal ck4 numero de buffers inversores: 1
  e para o sinal ck, numero de buffers inversores:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal ck5, numero de buffers: 2
 e para o sinal ck, numero de buffers:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal ck6, numero de buffers: 4
 e para o sinal ck, numero de buffers:0

### ERRO ###: diferentes fases de clock no caminho
 Pino associado aos sinais de clock: ck
 Para o sinal ck8 numero de buffers inversores: 1
  e para o sinal ck, numero de buffers inversores:0

### mesma fase e possivel distorcao de clock:
 Pino associado aos sinais de clock: ck

 Para o sinal ck9, numero de buffers: 1
 e para o sinal ck, numero de buffers:0

  ###   Elementos de circuito fora das regras:
  ###   Clocks nao controlaveis:
### Celulas sequenciais fora de scan:
  ###   Instancias sequenciais fora de qualquer scan:

  ###   Falhas de scanin:

### Detetada falha de scanin por cla'sc'scin:
   SCAN:  0    instancia da celula:  10
   Possiveis entradas de scan:  d10

  ###   Falhas de scanout:

### Detetada falha de scanout por cla'sc'scout:
   SCAN:  0    instancia da celula:  18
    ### Possivel saida de scan:  q18

 ####        Fim de relatorio
```