UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

TIAGO MÜLLER GIL CARDOSO

Exploração de Reordenamento de ROBDDs no Mapeamento Tecnológico de Circuitos Integrados

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. André Inácio Reis
Orientador

Porto Alegre, maio de 2007.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Cardoso, Tiago Müller Gil

Exploração de Reordenamento de ROBDDs no Mapeamento Tecnológico de Circuitos Integrados / Tiago Muller Gil Cardoso – Porto Alegre: Programa de Pós-Graduação em Computação, 2007.

92 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2007. Orientador: Dr. André Inácio Reis.

1. BDDs. 2. mapeamento tecnológico. 3. reordenamento de transistores. 4. gerador de células. I. Reis, André Inácio. II. Título.

AGRADECIMENTOS

Agradeço ao meu orientador André Inácio Reis, por toda paciência, dedicação e sabedoria compartilhada, sem os quais este trabalho de mestrado seria impossível.

Agradeço aos meus colegas Leomar Rosa Jr. e Felipe Marques, pela orientação técnica e pela amizade, fundamentais para que todas as dificuldades fossem superadas.

Agradeço aos meus colegas de laboratório: Milene Händel por toda a colaboração, especialmente durante o período de disciplinas; Paulo Butzen pela ajuda nas simulações SPICE; Eduardo Flores pela contribuição e uso do pacote de BDDs em seu trabalho de diplomação; Carlos Klock e Mateus Gomes por contribuírem com os visualizadores de BDDs. Também agradeço aos demais colegas que contribuíram direta ou indiretamente para este trabalho, Rodrigo Mancuso, Carlos Afonso Ferreira, Mário Osório, Felipe Schneider, Giovani Sartori, Vinicius Correia, Guilherme Schlinker, João Daniel Togni, Simone Bavaresco e Caio Alegretti. Um agradecimento especial ao Prof. Renato Ribas por manter "a casa em ordem" enquanto o Prof. André esteve fora do Brasil.

Agradeço aos amigos, por contribuírem para a formação de meu caráter e personalidade, estando presentes nos momentos mais importantes da vida, sejam comemorações ou períodos de dificuldade.

Agradeço à minha família, por me oferecerem todo apoio, necessário para que eu chegasse até aqui. Um forte abraço aos meus irmãos Mauro, Mateus, Luciano, suas esposas e seus filhos. Um abraço todo especial aos meus pais, Marzi e Saul, por me fornecerem a vida, por me educarem e por todos estes anos de amizade e amor incondicional.

Agradeço à Elizangela, por se tornar a pessoa mais importante em minha vida e por vencermos os longos períodos em que estivemos separados pela distância entre as capitais deste Sul.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ALU	Arithmetic and Logic Unit
API	Application Programming Interface
ASVO	Application Specific Variable Ordering
BDD	Binary Decison Diagram
BLIF	Berkeley Logic Interchange Format
CAD	Computer Aided Design
CMOS	Complementary Metal-Oxide-Semiconductor
CPTL	Complementary Pass Transistor Logic
DAC	Design Automation Conference
DCPTL	Disjoint Complementary Pass Transistor Logic
DED	Double-Error-Detecting
DVO	Dynamic Variable Ordering
FPGA	Field Programmable Grid Array
ITE	If-Then-Else
LUT	Look-Up Table
MOS	Metal-Oxide-Semiconductor
MUX	Multiplexador
NCV	Non-Controlling Value
NP	Non-Deterministic Polynomial
NPTL	NMOS Pass Transistor Logic
OBDD	Ordered Binary Decision Diagram
ROBDD	Reduced Ordered Binary Decision Diagram
SBDD	Shared Binary Decision Diagram
SEC	Single-Error-Correcting
UFRGS	Universidade Federal do Rio Grande do Sul
VLSI	Very Large Scale Integration

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Os ROBDDs são estruturas utilizadas com sucesso em ferramentas de CAD para microeletrônica. Estas estruturas permitem a representação canônica de funções booleanas ao se estabelecer um ordenamento fixo de variáveis. No contexto de um gerador automático de células lógicas para circuitos integrados, os ROBDDs podem servir de base para a derivação de redes de transistores cujo comportamento elétrico equivale ao comportamento lógico de uma função booleana desejada. Nas redes de transistores derivadas de ROBDDs, o posicionamento relativo dos transistores é determinado pelo ordenamento de variáveis. O efeito do reordenamento de transistores já foi estudado na década de noventa e sabe-se de sua influência sobre características de área, atraso e potência de um circuito digital. Entretanto, estes estudos limitam-se à topologia CMOS complementar série/paralelo, que é a topologia de redes de transistores mais comum. Neste trabalho, explora-se o efeito do reordenamento de variáveis nas características de área e atraso de circuitos mapeados com seis famílias lógicas diferentes, cujas células constituem redes de transistores derivadas de ROBDDs. Em geral, os resultados dos experimentos indicam que, para estas famílias lógicas, selecionar ordenamentos, onde transistores controlados por sinais mais críticos posicionam-se relativamente mais próximos à saída da célula, pode levar ao mapeamento de circuitos com atraso 16,4% inferior, em média, ao atraso do circuito equivalente com ordenamentos selecionados para obtenção da menor área possível e ignorando-se os atrasos de chegada nas entradas de uma célula.

Palavras-Chave: BDDs, mapeamento tecnológico, reordenamento de transistores, gerador de células.

Exploration of ROBDD Reordering on Technology Mapping for Integrated Circuits

ABSTRACT

The ROBDDs are structures that have been successfully used in CAD tools for microelectronics. These structures allow canonical representation of boolean functions when established a fixed variable ordering. In the context of an automatic logic cell generator for integrated circuits, ROBDDs may serve as a base for deriving transistor networks from which electrical behavior is equivalent to the logic behavior of a specified boolean function. With ROBDD derived transistor networks, the relative placement of transistors is determined by variable ordering. The effect of transistor reordering was already studied in the nineties and we know about its influence over area, delay and power characteristics of an integrated circuit. However, these studies were limited to complementary series/parallel CMOS topology, which is the standard for transistor networks topology. In this work, the effect of variable reordering is explored over area and delay characteristics of circuits mapped to six different logic families, where cells are designed with ROBDD derived transistor networks. Experimental results indicate that, in general, placing transistors controlled by the most critical signals closer to cell output may lead to a circuit mapping with an average 16.4% less delay than an equivalent circuit where orderings for smallest possible area are selected and input arrival times of a cell are ignored.

Keywords: BDDs, technology mapping, transistor reordering, cell generator.

1 INTRODUÇÃO

De acordo com uma avaliação respondida por freqüentadores da 42ª Design Automation Conference (www.dac.com), os três principais fluxos de projeto de interesse são: ASIC (36.3% de referências), Custom (25.7% de referências) seguido de FPGA (13.5% de referências). Somando-se ASIC e Custom tem-se 62% de referências. Assim sendo, este mercado ainda é importante dentro do cenário de ferramentas de CAD, considerando-se que o DAC é a referência nesta área. Note-se que tanto circuitos ASIC como circuitos Custom oferecem ampla liberdade em todas as camadas de projeto. Este tipo de fluxo de projeto permite explorar a flexibilidade na geração de redes de transistores para células combinacionais. Esta liberdade advém do fato de todos os níveis de máscaras poderem ser usados sem restrições.

Muitas abordagens foram propostas para a geração de redes de transistores a partir de estruturas como diagramas de decisão binária (BDD - *Binary Decision Diagram*, em inglês). Uma revisão de métodos para geração de redes de transistores a partir de BDDs pode ser encontrada em (ROSA JR, 2006). A proposta deste trabalho é adicionar um nível extra de liberdade aos métodos apresentados em (ROSA JR, 2006). Trabalhos anteriores (CARLSON, 1992) (CARLSON, 1995) consideram o reordenamento de transistores em uma célula, tendo como base a sua utilização (instâncias individuais) em um circuito, para obter ganhos em desempenho. Estes trabalhos concentram-se em células que apresentam redes de transistores na topologia CMOS estático complementar série/paralelo (WESTE, 1999) que é a topologia mais utilizada na implementação de circuitos digitais. Nosso objetivo aqui é estender este tipo de estudo para redes derivadas de BDDs, o que ainda não existe na literatura.

O contexto geral deste trabalho considera então os seguintes fatos:

- o interesse majoritário de freqüentadores do DAC por fluxos que utilizam todos os níveis de máscaras;
- a disponibilidade de métodos de geração de redes de transistores a partir de BDDs (ROSAJR, 2006);
- a disponibilidade de geradores de células que aceitam como entrada redes de transistores (www.inf.ufrgs.br/lagarto) (LEFEBVRE, 1997);
- a existência de companhias que baseiam seus fluxos de projeto ou oferecem ferramentas para geração automática de células (www.nangate.com) (ROY, 2005).

Baseado neste contexto, deseja-se verificar o efeito do reordenamento de células individuais derivadas de BDDs na área e no atraso de circuitos integrados. A abordagem

proposta assume que o circuito final será programado por todas as máscaras e pode, portanto, ser usada em metodologias ASIC ou Custom.

Esta dissertação está organizada da seguinte forma:

Capítulo 2: apresenta as hipóteses de base desta dissertação. Ele fala sobre os efeitos explorados e uma visão resumida de como se espera que tais efeitos influenciem a área e atraso de um circuito integrado digital.

Capítulo 3: revisa conceitos teóricos e de projeto relacionados a BDDs e pacotes de BDDs.

Capítulo 4: apresenta algumas das características encontradas em seis diferentes famílias lógicas que empregam redes de transistores derivadas de ROBDDs.

Capítulo 5: apresenta as decisões de projeto relacionadas ao software desenvolvido como parte deste trabalho.

Capítulo 6: apresenta os resultados dos experimentos realizados para avaliar o efeito do ordenamento de variáveis nas características de área e atraso em circuitos mapeados com células lógicas derivadas de ROBDDs.

Capítulo 7: apresenta considerações finais sobre o trabalho desenvolvido e sugestões para trabalhos futuros.

2 PROPOSTA DA DISSERTAÇÃO

A síntese de um circuito integrado deve levar em conta área, atraso e potência. Em determinado fluxo de projeto, deseja-se reduzir estes três parâmetros através de uma função de custo que combine os três. São especialmente desejáveis métodos que possam ocasionar ganhos em algum dos parâmetros, sem implicar perdas em outro. Assim, modificações locais que não interfiram com ganhos obtidos em outros níveis são bastante desejáveis, com destaque para ganhos resultantes da mudança na topologia de transistores.

Existem companhias que desenvolveram ferramentas voltadas para fluxos de projeto que incluem a geração automática de células. Como exemplo, a já extinta *Cadabra* (comprada pela *Numerical Technologies* e depois adquirida pela *Synopsys*, que ainda comercializa os produtos da *Cadabra*) e outras ainda ativas, tais como *Prolific*, *Zenasis* e *Nangate*.

Portanto, a geração de redes de transistores é um elemento chave em fluxos de projeto baseados na geração automática de células ou na geração automática de bibliotecas de células. A contribuição deste trabalho se dará na etapa de geração de redes de transistores para células.

Este capítulo apresenta as hipóteses de base desta dissertação. Ele fala sobre os efeitos explorados e uma visão resumida de como se espera que tais efeitos influenciem a área e atraso de um circuito integrado digital. A seção 2.1 apresenta, em linhas gerais, o problema do mapeamento tecnológico. A seção 2.2 apresenta o papel do ordenamento de variáveis no mapeamento tecnológico com células lógicas derivadas de ROBDDs. A seção 2.3 apresenta o objetivo deste trabalho. A seção 2.4 apresenta a metodologia utilizada para alcançar o objetivo proposto. A seção 2.5 apresenta uma revisão com os trabalhos mais importantes relacionados ao tema desta dissertação. A seção 2.6 destaca a contribuição deste trabalho para fluxos de projeto que empregam células lógicas derivadas de ROBDDs.

2.1 Mapeamento tecnológico

Em linhas gerais, o mapeamento tecnológico tem como objetivo transformar uma descrição lógica previamente otimizada do circuito em uma descrição funcionalmente equivalente contendo células-padrão (leiaute). As células-padrão são provenientes de uma biblioteca de células que é projetada para uma tecnologia específica. A Figura 2.1 apresenta o contexto do mapeamento tecnológico dentro de um fluxo de projeto completo. Uma revisão de métodos de mapeamento tecnológico pode ser encontrada em (CORREIA, 2005).

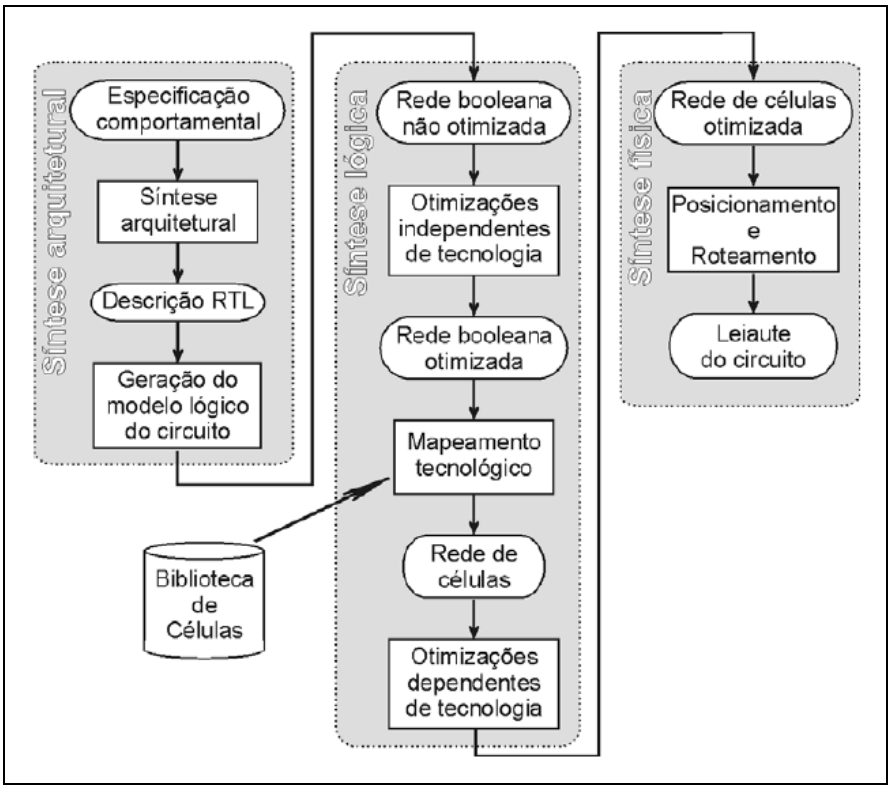

Figura 2.1: Contexto do mapeamento tecnológico (CORREIA, 2005)

No mapeamento tecnológico tradicional, cada célula da biblioteca tem seu leiaute projetado cuidadosamente por uma equipe de projetistas experientes. As medidas de área, atraso e potência são extraídas sob diferentes contextos e são dispostas junto à biblioteca de células para alimentar a função de custos de um método de mapeamento tecnológico.

As bibliotecas de células que possuem um conjunto fixo de células previamente projetadas e caracterizadas são conhecidas como *bibliotecas estáticas*. Normalmente, uma biblioteca estática oferece poucas células em conseqüência do alto custo de projeto e caracterização destas células. Diferentes pesquisadores (KEUTZER, 1999) (SECHEN, 2003) defendem a tese de que este tipo de biblioteca leva ao mapeamento de circuitos com qualidade inferior à que poderia ser obtida caso toda célula fosse otimizada de acordo com o contexto individual da célula no circuito.

Como alternativa ao uso de bibliotecas estáticas, a geração automática de células permite métodos de mapeamento tecnológico voltados ao uso de *bibliotecas dinâmicas* (ou *bibliotecas virtuais*). Em geral, as bibliotecas dinâmicas possuem um número maior de células e são induzidas através de restrições estruturais, tais como: *número de entradas* ou *número de transistores em série* de uma célula. Com um maior número de versões para células funcionalmente equivalentes, o mapeamento tecnológico com bibliotecas dinâmicas (ou livre de biblioteca) permite maior flexibilidade na determinação da melhor célula em cada ponto específico do circuito, levando possivelmente ao projeto automático de circuitos com melhores características de área, atraso e potência. A Tabela 2.1 resume as características de bibliotecas estáticas e dinâmicas.

Tabela 2.1: Bibliotecas de células: Estática x Dinâmica

Característica	Biblioteca estática	Biblioteca dinâmica
Número de células	Pequeno	Grande
Caracterização das células	Precisa	Simples
Complexidade das células	Baixa	Alta
Portabilidade	Custo alto	Custo baixo
Projeto das células	Manual	Automático

Fonte: CORREIA, 2005

2.2 Ordenamento de ROBDDs

Os ROBDDs são uma representação canônica de funções booleanas quando um ordenamento fixo de variáveis é estabelecido. A Figura 2.2 apresenta exemplos de ROBDDs representando algumas das funções booleanas mais simples. Um caminho que vai de um nodo raiz até um nodo terminal representa uma ou mais linhas de uma tabela-verdade, como ilustrado na Figura 2.3.

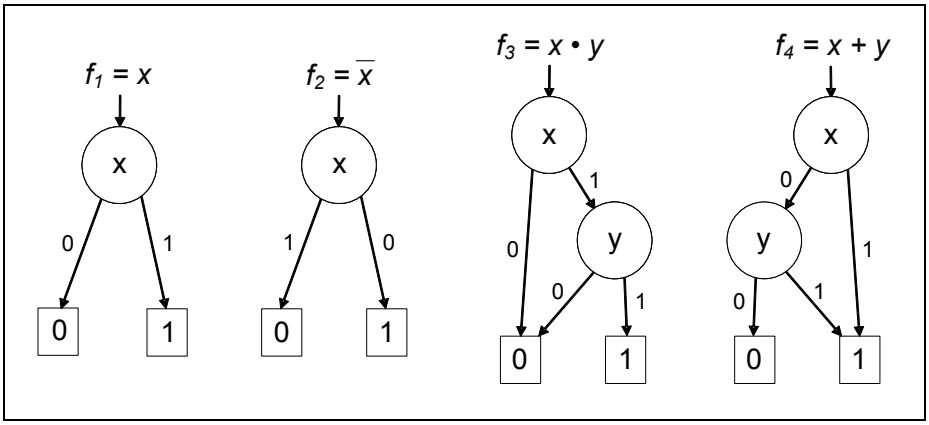

Figura 2.2: Exemplo de ROBDDs representando funções booleanas

A estrutura de ROBDDs que representam a mesma função pode variar de acordo com a ordem pretendida para as variáveis (ver Figura 2.4). O tamanho do ROBDD pode variar de linear a exponencial em relação ao número de variáveis, dependendo da função representada e do ordenamento de variáveis. Foi demonstrado que encontrar a ordem que leva o ROBDD ao mínimo exato de nodos é um problema NP-completo (BOLLIG, 1996), embora possam ser obtidas soluções para pequenas instâncias do problema (até dez ou vinte variáveis) em tempo aceitável. Na prática são utilizados métodos heurísticos como *Sifting* (RUDELL, 1993) para se obter bons ordenamentos em tempo hábil. A quantidade de ordenamentos (permutações) possíveis cresce rapidamente e é determinado pelo fatorial do número de variáveis da função. O capítulo 3 apresenta mais detalhes sobre a estrutura de ROBDDs.

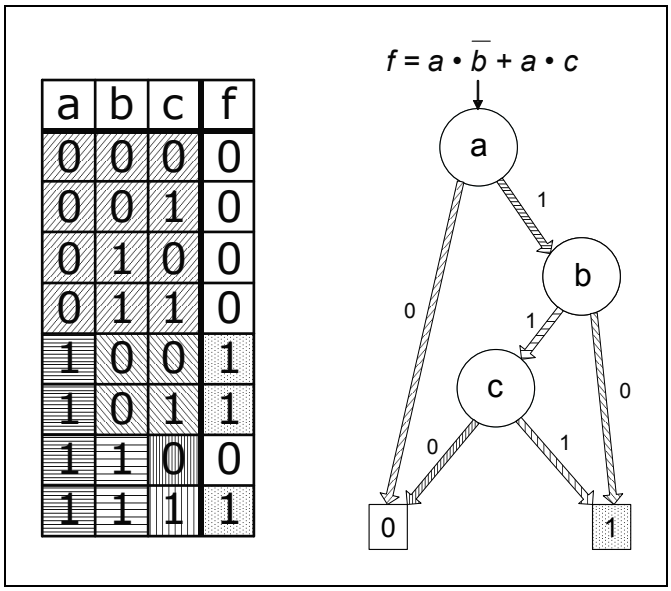

Figura 2.3: Caminhos de um ROBDD representam linhas de uma tabela-verdade

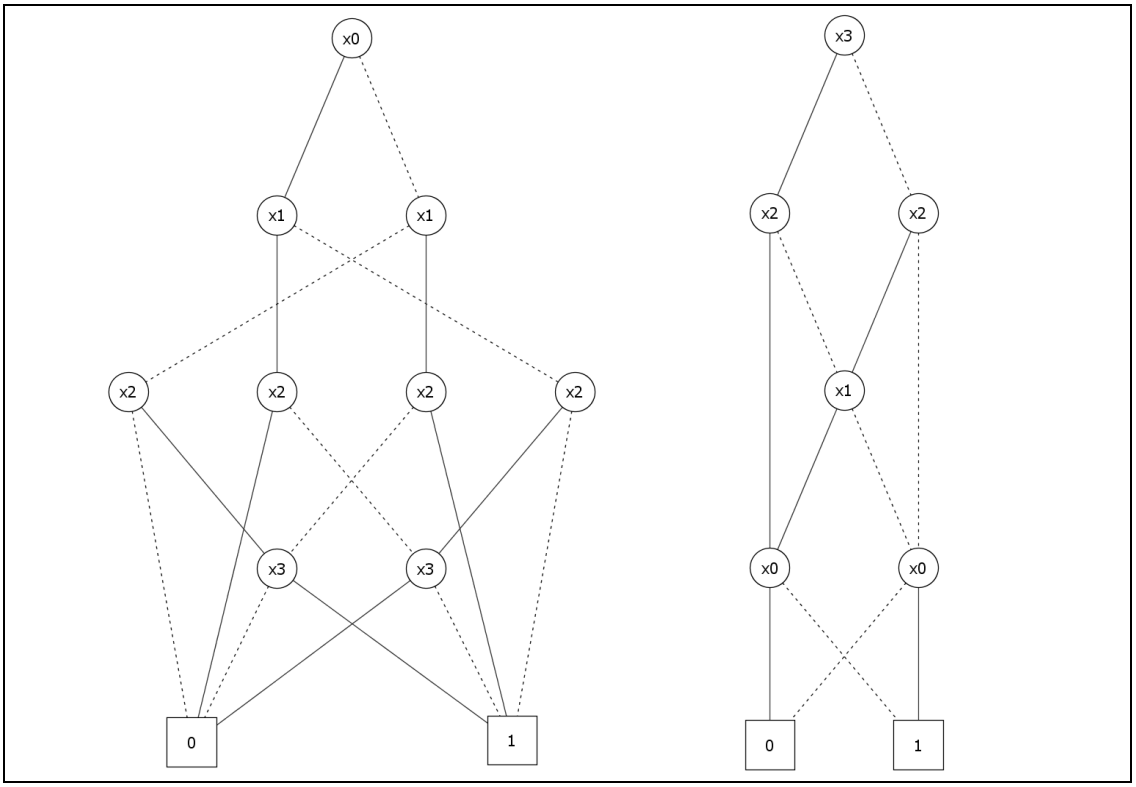

Figura 2.4: ROBDDs para a mesma função com ordenamentos distintos

Neste trabalho explora-se o reordenamento de transistores em famílias lógicas cujas redes de transistores são derivadas a partir da associação de transistores a arcos de ROBDDs. O ordenamento de variáveis influi diretamente na topologia das redes derivadas (ver Figura 2.5). A minimização do ROBDD é importante para métodos que buscam circuitos com área mínima, mas o ordenamento de variáveis também afeta o posicionamento relativo dos transistores, influenciando outras características do circuito como atraso e potência.

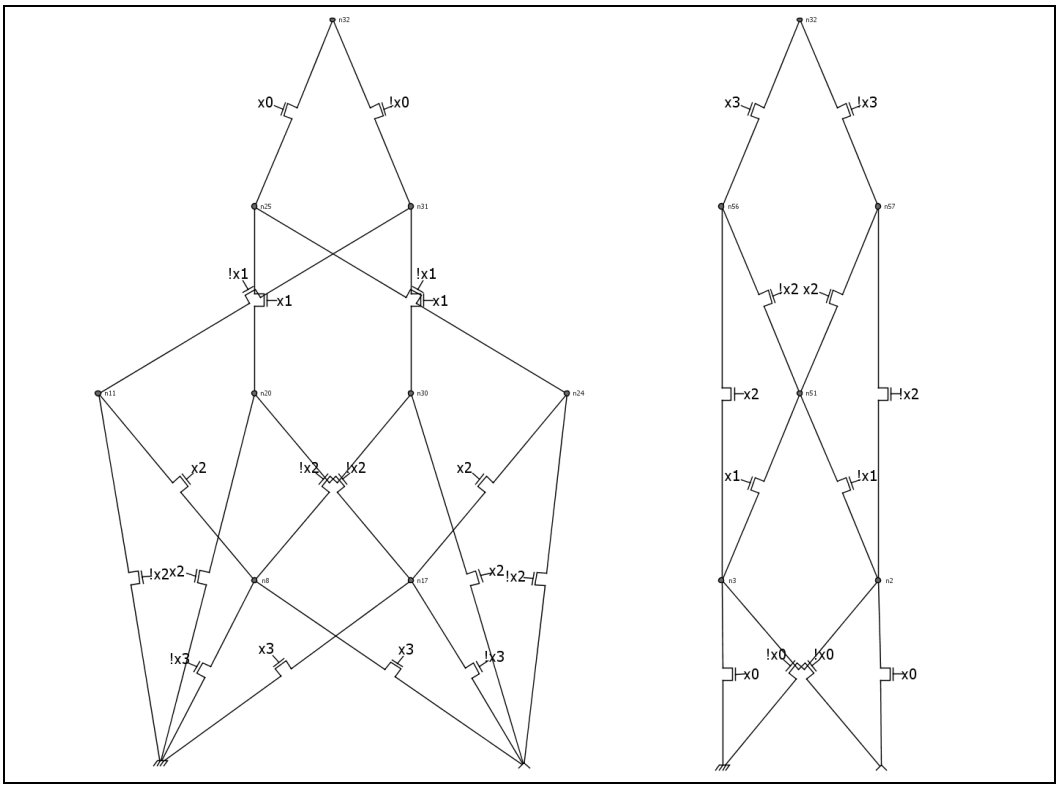

Figura 2.5: O ordenamento do ROBDD influi diretamente na rede derivada

2.3 Objetivo

O reordenamento de ROBDDs é um problema amplamente estudado e sabe-se que o ordenamento de variáveis pode ter influência significativa no tamanho da representação. No mapeamento tecnológico com células derivadas de ROBDDs, quanto maior a estrutura, mais transistores aparecem na rede derivada, dando origem a células lógicas que ocupam maior área, acrescentam maior capacitância parasita e estão sujeitas a um maior consumo estático de potência (*leakage)*. Entretanto, ao considerar que as variáveis do ROBDD representam os sinais de entrada de uma célula e que os tempos de chegada destes sinais podem ser diferentes, torna-se interessante o ordenamento das variáveis de modo a favorecer a propagação daqueles sinais que possuem os maiores atrasos de chegada em cada célula. O objetivo é diminuir o atraso de propagação no caminho crítico do circuito ao favorecer os sinais mais críticos localmente em cada célula.

O objetivo deste trabalho é investigar o efeito do ordenamento de variáveis nas características de área e atraso, considerando circuitos mapeados com redes de transistores derivadas a partir de ROBDDs. O tempo de chegada dos sinais de entrada é utilizado como critério para o ordenamento das variáveis no ROBDD. Assume-se a disponibilidade de um gerador de células que recebe a descrição SPICE da rede derivada e gera o leiaute para uma célula equivalente, embora este componente não tenha sido utilizado nos experimentos.

2.4 Metodologia

Para explorar o efeito do reordenamento de variáveis na área e atraso de circuitos mapeados com famílias lógicas derivadas de ROBDDs, a seguinte metodologia é proposta:

- Partindo de uma descrição lógica do circuito combinacional no formato BLIF, obter uma decomposição do circuito com funções booleanas de até quatro variáveis. Isto pode ser realizado com auxílio da ferramenta SIS (SENTOVICH, 1992).
- Cada função booleana obtida após a decomposição pode ser representada por um ROBDD de até quatro variáveis. Para investigar o efeito do ordenamento de variáveis no desempenho dos circuitos mapeados com células lógicas derivadas de ROBDDs, são definidos quatro critérios de ordenamento de variáveis como enumerado a seguir:
  - (i) *Ordenamentos para topo crítico*: os sinais de entrada com os maiores tempos de chegada são posicionados mais próximos à saída da célula, ou seja, próximos ao topo do ROBDD. Se os tempos de chegada forem iguais então é mantido o ordenamento original da descrição lógica. A Figura 2.6 apresenta um circuito hipotético (a) e um caminho potencialmente crítico com ordenamentos de variáveis para topo crítico (b). Os tempos de chegada são definidos através do modelo de atrasos unitários (ver subseção 5.3.1).
  - (ii) *Ordenamentos para base crítica*: utiliza-se o ordenamento inverso ao obtido em (i). Por exemplo, se o ordenamento (i) é (*a*,*b*,*c*,*d*) então o ordenamento (ii) será (*d*,*c*,*b*,*a*).
  - (iii) *Ordenamentos para menor área*: seleciona-se um ordenamento de variáveis que leva ao número mínimo de nodos na estrutura. Para ordenamentos diferentes que levam ao mesmo número de nodos, seleciona-se aquele que resultar na maior quantidade de arcos em direção aos nodos terminais. Permanecendo empate, seleciona-se o ordenamento que resultar no maior número de arcos em direção ao nodo terminal zero.
  - (iv) *Ordenamentos para menor área e topo crítico no caminho mais longo*: Para as células que pertencem ao caminho mais longo do circuito seleciona-se o ordenamento obtido em (i) e para as demais células seleciona-se o ordenamento obtido em (iii).
- Diferentes modelos de atrasos (ver subseção 5.3.1) são empregados na busca de dez caminhos potencialmente críticos em cada um dos quatro mapeamentos do circuito.
- As células que pertencem a cada caminho potencialmente crítico são isoladas e valores não-controladores são associados às entradas laterais para sensibilizar o caminho desde uma entrada primária até a saída primária (ver Figura 2.6 (b)).
- São geradas descrições SPICE prontas para a simulação de cada caminho potencialmente crítico. Esta descrição contém as células e suas instâncias encadeadas e com os vetores de entrada para a sensibilização de todo o caminho.

- A simulação SPICE fornece uma medida precisa do atraso de propagação em cada caminho potencialmente crítico. Assume-se que o caminho crítico para cada versão do circuito seja aquele que possuir o maior atraso de propagação entre os dez caminhos simulados.
- A razão entre o atraso do caminho crítico obtido no circuito mapeado com os critérios de ordenamento (i), (ii) ou (iv) e o atraso do caminho crítico obtido no circuito mapeado com o critério de ordenamento (iii) determina o efeito do reordenamento no atraso do circuito.
- A razão entre a área do circuito mapeado com os critérios de ordenamento (i), (ii) ou (iv) e a área do circuito mapeado com o critério de ordenamento (iii) determina o efeito do reordenamento sobre a área do circuito.

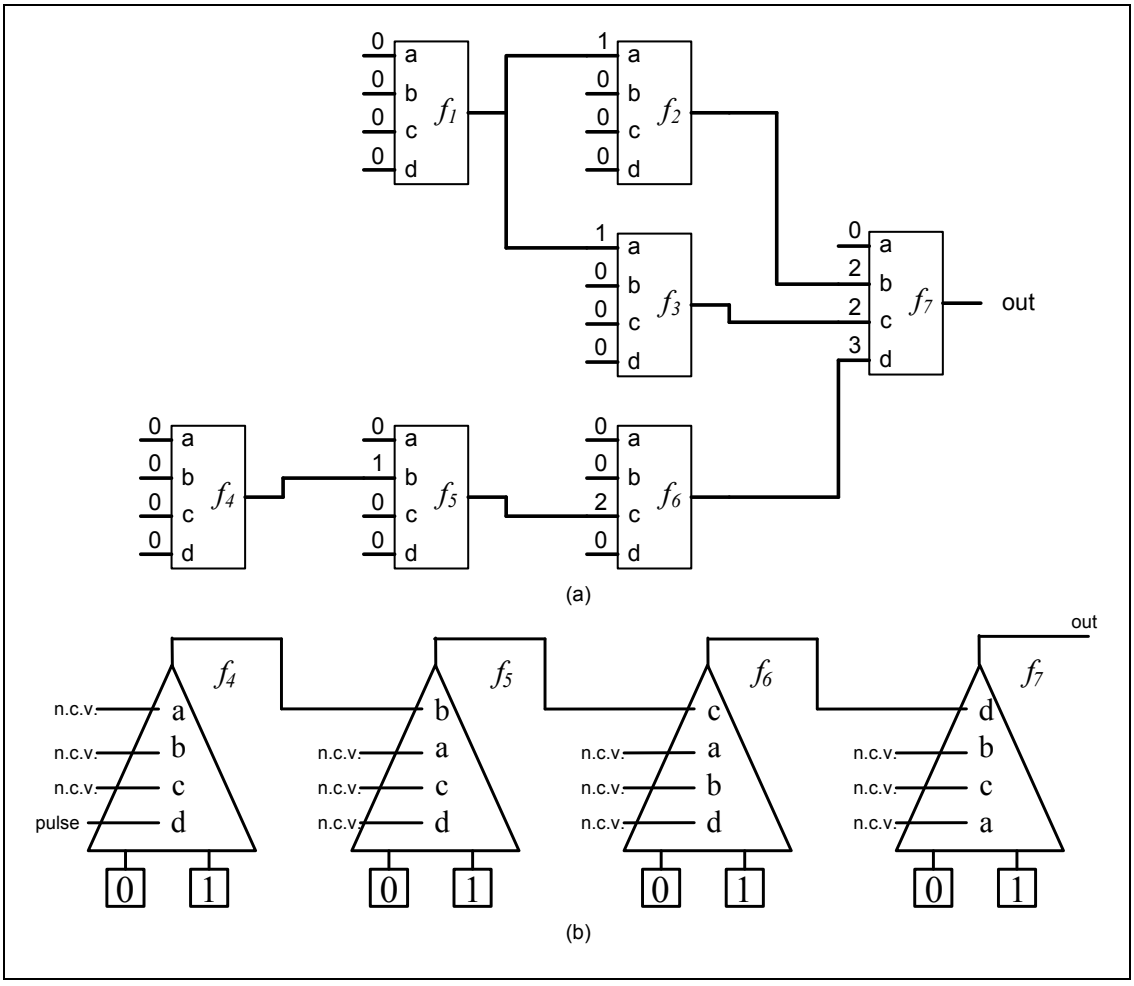

Figura 2.6: Ordenamento de variáveis para topo crítico

A mesma metodologia foi aplicada para avaliar seis diferentes famílias lógicas que empregam células com redes de transistores derivadas de ROBDDs. As medidas de área são obtidas pela contagem de transistores nas células do circuito mapeado. O fluxo da metodologia proposta é mostrado na Figura 2.7.

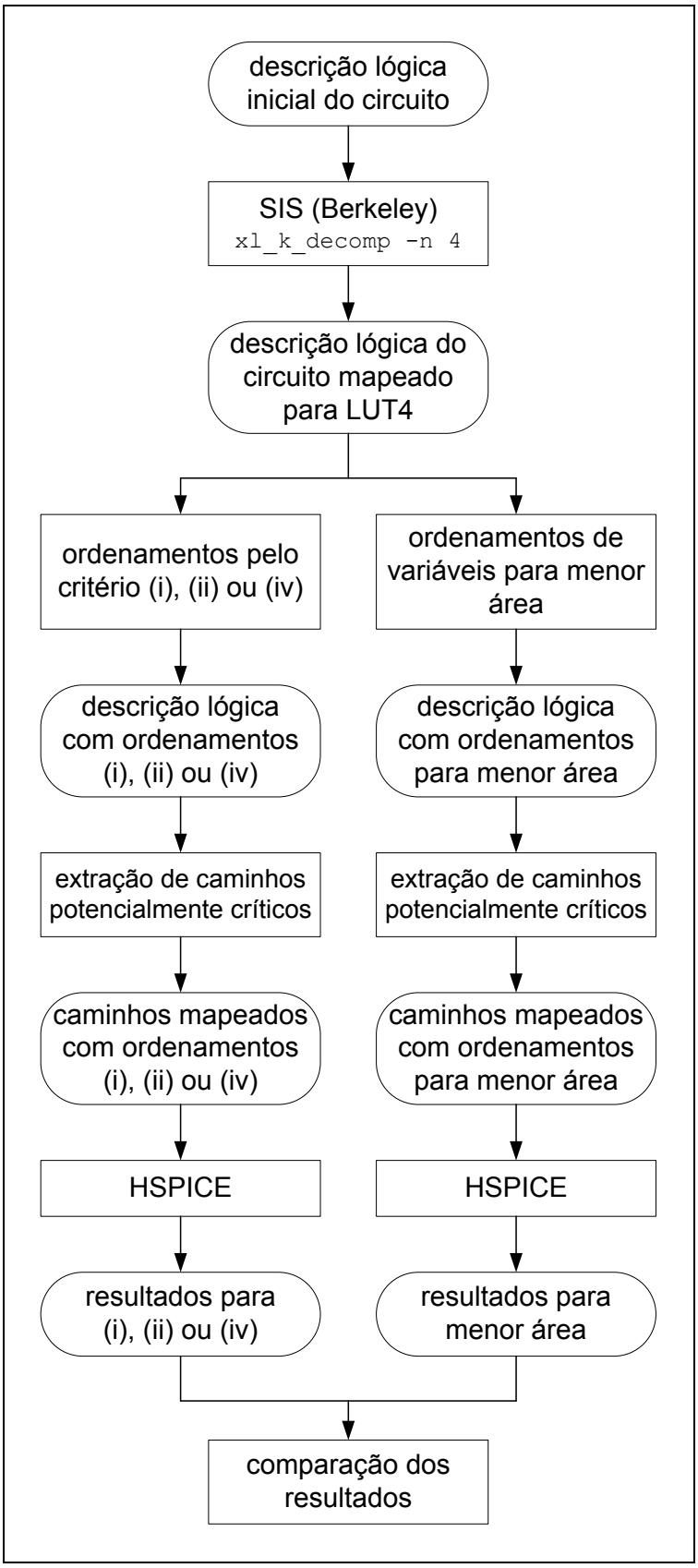

Figura 2.7: Fluxo da metodologia proposta

2.5 Trabalhos relacionados

Um estudo sobre o efeito do reordenamento de transistores (CARLSON, 1992) demonstrou que o ordenamento de transistores pode ter efeito significativo no atraso de células lógicas implementadas com redes de transistores na topologia CMOS estático complementar série/paralelo. Os autores simularam diferentes implementações de portas NAND variando parâmetros como número de entradas, capacitância de carga, dimensionamento de transistores ou inclinação do sinal de entrada. São considerados dois ordenamentos: os transistores controlados pelos sinais mais críticos são posicionados mais próximos à saída (*top-critical*) ou então mais próximos às fontes de tensão (*bottom-critical*). Os resultados demonstram que o efeito do reordenamento pode variar em função dos parâmetros e que nem sempre o ordenamento *top-critical* leva a melhores resultados do que o ordenamento *bottom-critical*.

Uma metodologia para a otimização do atraso de circuitos através do reordenamento de transistores é proposta em (CARLSON, 1995). Para diminuir o atraso do circuito, cada célula no caminho crítico é simulada em ambos os ordenamentos (*top-critical* e *bottom-critical*) e seleciona-se aquela que obteve o menor atraso. Os resultados indicam redução média em torno de 9% no atraso de circuitos que empregam células na topologia CMOS complementar série-paralelo.

Diferentes autores utilizam ROBDDs como estrutura base para a derivação de circuitos com lógica de transistor de passagem. Em (YANO, 1996) foram obtidos ganhos em área, atraso e potência em relação à topologia CMOS complementar série/paralelo para circuitos pequenos. Um SBDD monolítico é construído para a lógica do circuito sem nenhuma preocupação com o ordenamento de variáveis. O problema é que a construção do BDD monolítico torna-se inviável para circuitos grandes, especialmente quando a ordem das variáveis é ignorada. O SBDD é mapeado para uma biblioteca de células formada por células MUX e inversores com diferentes dimensionamentos.

Um fluxo mais elaborado é apresentado em (BUCH, 1997). Evita-se a construção do BDD monolítico inserindo-se variáveis intermediárias quando o número de nodos ou a profundidade atinge determinado limite na construção do BDD. Em vez de um único SBDD para o circuito todo, são obtidos vários BDDs com tamanho limitado e encadeados numa estrutura denominada *decomposed* BDDs. O ordenamento de variáveis é independente para cada um dos BDDs. Os autores sugerem dois critérios para o ordenamento das variáveis. No critério para menor área, procura-se minimizar o número de nodos utilizando *sifting* (RUDELL, 1993). No critério para menor potência, minimiza-se o número de nodos rotulados pela variável com maior probabilidade de chaveamento e/ou as variáveis com menor probabilidade de chaveamento são posicionadas mais próximas aos nodos terminais. Os autores não exploram o ordenamento de variáveis para reduzir o atraso do circuito.

Em (CHAUDHRY, 1998) é apresentado um fluxo orientado para a síntese automática de circuitos PTL com área mínima. Foram adaptadas técnicas de otimização multi-nível como eliminação e minimização de *don't cares*. O método proposto depende de *sifting* para minimização dos BDDs, levando a resultados sub-ótimos. Após a síntese do circuito como uma rede de células MUX e inversores, é realizada uma etapa de remoção de redundâncias com auxílio da ferramenta SIS (SENTOVICH, 1992). Como o circuito é decomposto em BDDs de tamanho limitado, poderia ser viável a aplicação de uma abordagem de otimização exata (EBENDT, 2004) em vez de *sifting*.

Um fluxo orientado à síntese automática de circuitos PTL com atraso mínimo é apresentado em (LIU, 1999). Este fluxo considera o mapeamento de nodos de BDDs para células MUX e inversores. Os autores utilizam um modelo de atrasos baseado em estimativas de resistência e de capacitância na saída da célula, além do atraso intrínseco da propagação de sinais através de um multiplexador, seja como sinal de controle ou como sinal de passagem. Os atrasos de chegada dos sinais são considerados na estimativa do atraso de pior caso. São aplicadas otimizações de síntese multi-nível, semelhantes às apresentadas em (CHAUDHRY, 1998). O algoritmo de *sifting* foi modificado para considerar o atraso de pior caso (ou o produto área-atraso) como métrica de minimização.

Os autores em (SCHOLL, 2000) também utilizam uma versão modificada do algoritmo de *sifting* que considera, além do número de nodos, a profundidade do ROBDD como métrica para minimização. O ROBDD (ou *decomposed* BDD) serve apenas como estrutura de dados para o algoritmo de síntese e a estrutura final do circuito não possui relacionamento um para um com a estrutura canônica, permitindo circuitos mais compactos e de menor profundidade lógica do que abordagens ingênuas de geração de redes de transistores derivadas de BDDs.

Um fluxo de síntese PTL semelhante ao apresentado em (YANO, 1996) é proposto em (HSIAO, 2000). Os autores adicionaram uma estratégia de ordenamento de variáveis para minimizar a área do circuito e consideram a possibilidade dos sinais de entrada não possuírem os mesmos tempos de chegada, permitindo ao usuário especificar um ordenamento diferente de variáveis do que o obtido através da minimização do ROBDD.

O fluxo de síntese PTL apresentado em (SHELAR, 2005) é voltado para otimização do atraso do circuito. Os autores exploram um método de bipartição recursivo que reduz pela metade a profundidade do BDD em cada recursão. O ponto de corte é selecionado criteriosamente, avaliando estimativas de atraso após o corte e selecionando o ponto de corte que leve ao menor aumento em área (*max-flow min-cut*). O modelo de atrasos usado para as estimativas é o modelo de atraso de Elmore, permitindo estimativas em tempo linear no número de transistores. Os BDDs resultantes da decomposição são mapeados para células NPTL com entradas em drenos. O mapeamento considera a utilização de multiplexadores com codificação *one-hot* para obter circuitos mais rápidos. O método foi aplicado aos benchmarks ISCAS'85 com resultados inconsistentes de atraso em relação aos circuitos mapeados com tecnologia CMOS convencional. As variáveis são ordenadas para minimizar o tamanho do BDD através de *sifting*.

2.6 A contribuição deste trabalho

A contribuição deste trabalho é destacar a importância do ordenamento de variáveis para fluxos de síntese PTL que utilizam ROBDDs como estrutura para o mapeamento. A maioria dos fluxos de síntese PTL propostos na literatura não consideram o efeito do ordenamento de variáveis sobre o atraso do circuito sintetizado. Os experimentos realizados como parte deste trabalho demonstram que é possível obter circuitos mais rápidos ao se favorecer o sinal mais lento em cada célula do caminho crítico. O efeito do reordenamento de variáveis é semelhante ao efeito do reordenamento de transistores em células CMOS convencionais. A diferença é que o número de transistores pode

variar no reordenamento com ROBDDs, não ocorrendo o mesmo no reordenamento de transistores para células CMOS complementar série/paralelo.

Outra questão a ser observada é o uso de métodos heurísticos para minimizar ROBDDs de tamanho limitado. Apesar da minimização de ROBDDs ser um problema NP-completo, instâncias pequenas do problema podem ser tratadas em tempo aceitável. Nos experimentos realizados neste trabalho, os ROBDDs estão limitados a quatro variáveis, em uma estrutura equivalente a *decomposed* BDDs (BUCH, 1997). Com este número de variáveis é possível utilizar a estratégia mais simples de minimização exata: construir ROBDDs para todos os ordenamentos e selecionar aquele que leva ao menor número de nodos. Entretanto, existem métodos mais elaborados de minimização exata que podem tratar ROBDDs com até vinte variáveis em tempo aceitável (EBENDT, 2004).

3 DIAGRAMAS DE DECISÃO BINÁRIA

Os BDDs são considerados o estado da arte em estruturas de dados para representação e manipulação eficiente de funções booleanas. Os BDDs são empregados com sucesso em ferramentas de CAD VLSI em aplicações de síntese, verificação, testes, simulação e decomposição funcional, entre outras. A ordem pretendida para as variáveis pode ter forte impacto no tamanho da representação com ROBDDs, porém determinar o ordenamento de variáveis que minimiza o tamanho da estrutura é considerado um problema intratável. Neste trabalho, os ROBDDs são utilizados como estrutura base para a derivação de redes de transistores que implementam uma função booleana qualquer. Neste caso, a minimização do ROBDD é importante para diminuir o número de transistores nas redes derivadas. Entretanto, o ordenameto de variáveis pode influir também na velocidade de propagação dos sinais através da rede derivada, afetando o atraso crítico de um circuito combinacional mapeado com estas redes.

Este capítulo revisa conceitos teóricos e de projeto relacionados a BDDs e pacotes de BDDs. A seção 3.1 apresenta o conceito de BDDs e a terminologia básica associada a estas estruturas. A seção 3.2 apresenta a formulação recursiva de ITE, utilizada na síntese eficiente de BDDs. A seção 3.3 apresenta o conceito de ROBDDs e as regras de redução que reduzem um OBDD qualquer a uma representação canônica de funções booleanas. A seção 3.4 mostra as vantagens que podem ser obtidas com o compartilhamento de nodos. A seção 3.5 apresenta considerações sobre o problema do ordenamento de variáveis em ROBDDs. A seção 3.6 destaca o problema específico de trocar duas variáveis de posição após um ROBDD estar construído. A seção 3.7 apresenta técnicas de programação utilizadas em pacotes de BDDs eficientes. A seção 3.8 apresenta as estruturas de dados encontradas em um pacote de BDDs eficiente. A seção 3.9 apresenta o algoritmo da operação ITE. A seção 3.10 apresenta um algoritmo para o problema da permuta de variáveis adjacentes.

3.1 Diagramas de decisão binária

Um *diagrama de decisão binária* (BDD) sobre um conjunto X_n de variáveis é um grafo acíclico dirigido $G = (V, A)$ com um vértice (ou nodo) raiz e as seguintes propriedades:

- Um elemento de V é um vértice *não-terminal* ou um vértice *terminal*.
- Cada vértice não-terminal v é rotulado por uma variável de X_n, referenciada por *índice*(v), onde v possui exatamente dois sucessores em V, referenciados por *zero*(v) e *um*(v).

- Cada vértice terminal v é rotulado por um valor $valor(v) \in \{0, 1\}$ e não possui sucessores.

O arco entre v e $zero(v)$ é denominado *arco-zero*(v) e o arco entre v e $um(v)$ é denominado *arco-um*(v). O *tamanho* de um BDD G é dado pela sua quantidade de vértices e pode ser representado por $|G|$. A *profundidade* de G é dada pelo maior comprimento de um caminho partindo da raiz de G até um vértice terminal e é representada por *profundidade*(G). O conjunto de nodos rotulados com a variável x_i é chamado de *nível*(x_i).

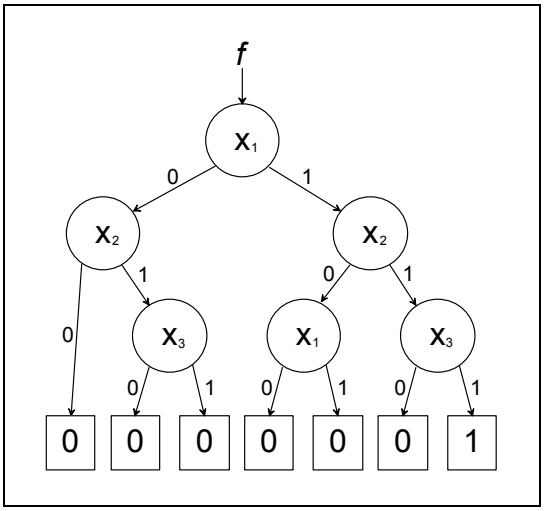

Figura 3.1: Exemplo de BDD para $f = x_1 \cdot x_2 \cdot x_3$

Cada vértice v de um BDD representa graficamente uma *decomposição de Shannon* em respeito à variável *índice*(v). Se o BDD possui um vértice raiz v, então este BDD representa uma função booleana f_v (ver Figura 3.1) definida da seguinte maneira:

- Se v é um vértice terminal e $valor(v) = 1$ ($valor(v) = 0$), então $f_v = 1$ ($f_v = 0$).
- Se v é um vértice não-terminal e $índice(v) = x_i$, então $f_v = \overline{x_i} \cdot f_{zero(v)} + x_i \cdot f_{um(v)}$, onde $f_{zero(v)} = f_v(x_1, ..., x_{i-1}, 0, x_{i+1}, ..., x_n)$ e $f_{um(v)} = f_v(x_1, ..., x_{i-1}, 1, x_{i+1}, ..., x_n)$.

Algumas restrições estruturais são aplicadas à definição de BDDs para eliminar elementos indesejados e permitir algoritmos de manipulação mais eficientes. Por exemplo, um BDD G é dito *livre* se cada variável é encontrada no máximo uma vez em cada caminho em G, partindo da raiz até um vértice terminal. Um BDD G é dito *ordenado* se ele é livre e as variáveis são encontradas na mesma ordem para qualquer caminho em G, partindo da raiz até um vértice terminal.

A ordem das variáveis em um BDD ordenado é um mapeamento $\pi: X_n \rightarrow \{1, ..., n\}$, onde $\pi(x_i)$ representa a posição da variável nesta ordem. Para uma ordem π, considera-se $x_i < x_j$ se e somente se $\pi(x_i) < \pi(x_j)$. Se uma ordem π não é definida, assume-se a ordem natural $\pi(x_i) = i$ ($i \in \{1, ..., n\}$). Para reforçar que um BDD é ordenado, usa-se o termo OBDD para referenciá-lo. O conjunto de nodos rotulados com a variável x_i é chamado de *nível*(x_i).

Cada vértice pode ser referenciado por uma letra maiúscula F e descrito por uma tripla (x_i, G, H), onde x_i é a variável *índice*(F), G é o vértice apontado pelo *arco-um*(F) e H é o vértice apontado pelo *arco-zero*(F). A variável x_i é chamada de *variável topo* de

F. A variável topo x_i de um conjunto de vértices é aquela que aparece primeiro na ordem, ou seja, aquela que tem o menor valor para $\pi(x_i)$.

3.2 Formulação recursiva da ITE

De forma equivalente à decomposição de Shannon, cada vértice F = (x_i, G, H) equivale à operação lógica terciária *Se* F *Então* G *Senão* H (ITE) como formalizado na seguinte expressão:

$$ITE(F, G, H) = F \cdot G + \overline{F} \cdot H$$

Esta operação pode ser utilizada para calcular operações booleanas de duas variáveis. Por exemplo, as operações conjunção, disjunção e negação podem ser calculadas respectivamente como mostra a Tabela 3.1.

Tabela 3.1: Cálculo de operações boolenas através da operação ITE

$F \cdot G$	ITE(F, G, 0)
$F + G$	ITE(F, 1, G)
$\overline{F}$	ITE(F, 0, 1)

Seja F = (w, T, E) e considerando $\pi(v) \leq \pi(w)$. Encontrar $F_{v=1}$ (cofator positivo) ou $F_{v=0}$ (cofator negativo) de F em relação a v pode ser obtido da seguinte maneira:

- Se $\pi(v) < \pi(w)$ então $F_{v=1}$ = F. Se $v = w$ então $F_{v=1}$ = T.
- Se $\pi(v) < \pi(w)$ então $F_{v=0}$ = F. Se $v = w$ então $F_{v=0}$ = E.

Sejam F, G e H os vértices de um BDD e v a variável topo destes vértices, então calcular ITE(F, G, H) pode ser realizado através da seguinte formulação recursiva:

$$ITE(F, G, H) = (v, ITE(F_{v=1}, G_{v=1}, H_{v=1}), ITE(F_{v=0}, G_{v=0}, H_{v=0}))$$

Os casos terminais para a recursão são:

$$ITE(1, F, G) = ITE(0, G, F) = ITE(F, 1, 0) = F$$

A formulação recursiva da ITE em conjunto com técnicas de programação (ver seção 3.7), formam a base para a síntese e manipulação eficiente de funções booleanas representadas com BDDs.

3.3 Diagramas de decisão binária ordenados e reduzidos

Um algoritmo para eliminação de redundâncias em OBDDs é apresentado em (BRYANT, 1986). Um OBDD processado por este algoritmo constitui uma representação canônica de funções booleanas, dado um ordenamento fixo de variáveis. Esta representação é atualmente conhecida como *diagrama de decisão binária*

ordenado e reduzido (ver Figura 3.2) e pode ser obtida com a aplicação exaustiva de apenas duas regras de redução em OBDDs (MINATO, 1996):

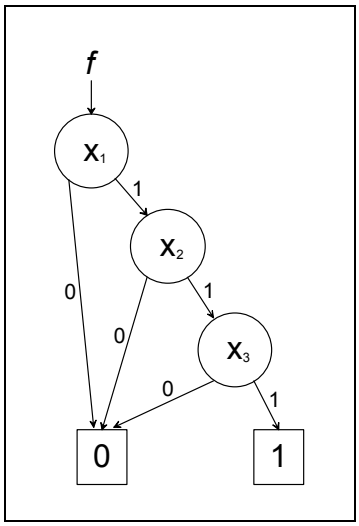

Figura 3.2: Exemplo de ROBDD para $f = x_1 \cdot x_2 \cdot x_3$

- Eliminar todos os vértices em que os dois arcos apontam para um mesmo vértice (ver Figura 3.3).

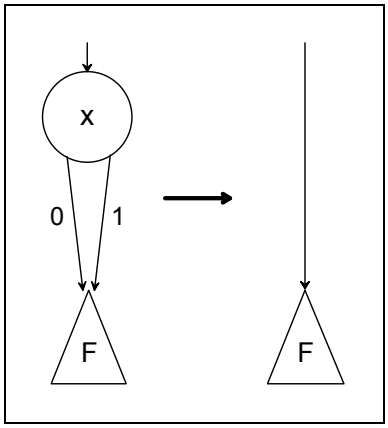

Figura 3.3: Redução por eliminação de vértice

- Compartilhar todos os subgrafos isomórficos (ver Figura 3.4).

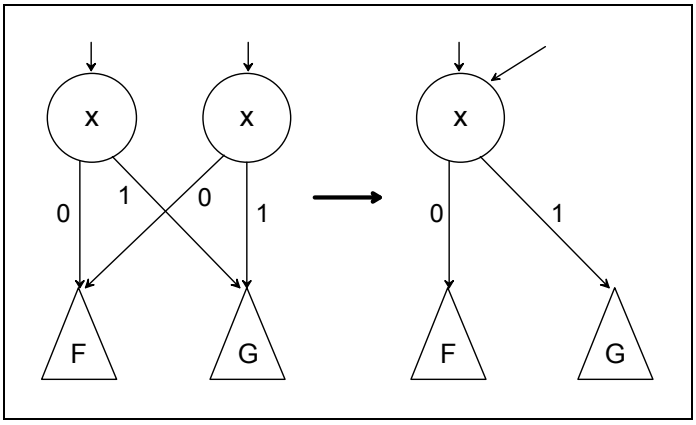

Figura 3.4: Redução por compartilhamento de vértice

Deve-se observar que a representação é canônica apenas para uma ordem fixa das variáveis, ou seja, dois ROBDDs *F* e *G* com as respectivas ordens π_1 e π_2, representando uma mesma função *f*, serão isomórficos se e somente se $\pi_1 = \pi_2$. Uma representação canônica é importante para aplicações práticas porque permite verificar a equivalência de duas funções booleanas simplesmente verificando-se o isomorfismo dos seus ROBDDs.

Em geral, é aceito que os ROBDDs apresentam um bom compromisso entre tamanho da representação e eficiência de manipulação, pois permitem que as principais operações de manipulação sejam tratadas por algoritmos de complexidade polinomial em relação ao número de vértices ou de variáveis. No entanto, mesmo com a exaustiva aplicação das regras de redução, um ROBDD pode apresentar tamanho exponencial em relação à quantidade de variáveis, como acontece na representação de multiplicadores.

O tamanho do ROBDD pode variar de acordo com o tipo de função representada e com a ordem pretendida para as variáveis, enquanto uma tabela-verdade possui tamanho invariavelmente exponencial em relação ao número de variáveis. Observa-se que o espaço ocupado por um nodo de BDD é muitas vezes maior do que uma posição de tabela-verdade, pois esta pode ser representada com um único bit.

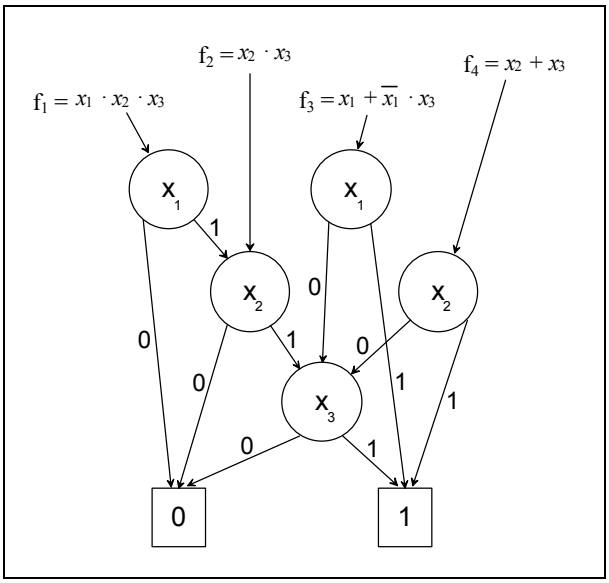

Figura 3.5: Exemplo de ROBDDs com compartilhamento de vértices

3.4 Compartilhamento de ROBDDs

Um conjunto de ROBDDs pode ser representado por um único *BDD compartilhado* (SBDD). Um SBDD é formado por ROBDDs em que os vértices equivalentes são compartilhados (ver Figura 3.5). Quando os subgrafos isomórficos são totalmente compartilhados, não existem dois vértices equivalentes num mesmo SBDD e cada vértice é a raiz de uma função booleana diferente. Com a utilização de SBDDs, as seguintes vantagens são obtidas:

- A verificação de equivalência pode ser realizada imediatamente, apenas verificando os vértices raízes.
- Economiza-se tempo e espaço evitando a redundância de vértices equivalentes.

Para garantir que dois vértices equivalentes nunca coexistam, as regras de redução são verificadas antes que um novo vértice seja criado. Se o arco-um e o arco-zero têm o mesmo destino (redução por eliminação) ou se já existe um vértice equivalente (redução por compartilhamento) então não se cria um novo vértice, mas aponta-se para o vértice existente equivalente.

Se for atribuído um *identificador único*, como um número natural positivo diferente para cada vértice, então o teste de equivalência entre as funções *f* e *g* é um teste escalar entre os identificadores únicos dos vértices que representam *f* e *g*.

3.5 Ordenamento de variáveis

O ordenamento de variáveis é um problema fundamental para o uso efetivo de ROBDDs. Dependendo da ordem escolhida para as variáveis, o tamanho da representação pode variar de linear a exponencial para algumas funções. Entretanto, a minimização exata, ou seja, determinar qual ordem resultará em tamanho mínimo, é um problema *NP-completo* (BOLLIG, 1996). Na prática, muitas funções com mais de vinte variáveis não podem ser minimizadas de forma exata, mesmo pelos algoritmos mais eficientes que se conhece (EBENDT, 2004). Entretanto, métodos heurísticos permitem obter ordens suficientemente boas em tempo aceitável para aplicações reais. A Figura 3.6 apresenta dois ROBDDs que representam a mesma função, porém com ordenamentos diferentes para as variáveis.

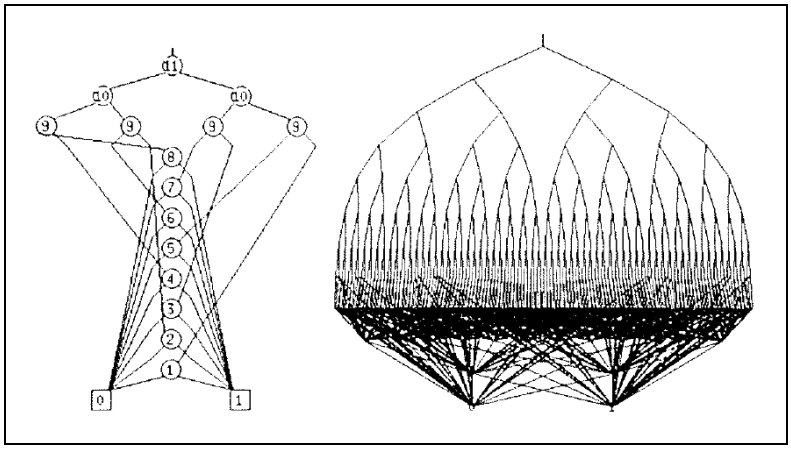

Figura 3.6: A mesma função em ordenamentos diferentes (MINATO, 1996)

O problema da minimização de ROBDDs foi intensamente pesquisado por diferentes autores e muitos métodos foram propostos na literatura. Em (MINATO, 1996) são apresentadas duas propriedades empíricas sobre o efeito do ordenamento de variáveis sobre o tamanho de um ROBDD:

- *Computação localizada*: As entradas com computação localizada devem ser mantidas próximas na ordem, ou seja, entradas fortemente relacionadas devem ser mantidas próximas umas às outras.
- *Poder para controlar a saída*: As entradas com grande influência na função devem ficar mais próximas à raiz na ordem.

Com a utilização de heurísticas para estas propriedades é possível obter representações mais compactas. No entanto, (MINATO, 1996) indica que as duas

propriedades tendem a entrar em conflito, dificultando sua aplicação. Em (DRECHSLER, 1998) os métodos de minimização heurística são classificados em dois tipos:

- *Ordenamento de Variáveis por Aplicações Específicas* (ASVO): A ordem das variáveis é determinada antes da construção do BDD, utilizando informações disponíveis sobre a instância do problema. Por exemplo, para representação de circuitos, estas informações são obtidas através das propriedades estruturais do circuito.
- *Ordenamento Dinâmico de Variáveis* (DVO): A construção do BDD começa com uma ordem inicial. Se necessário, uma fase de *reordenamento* é disparada ao se atingir determinada condição desfavorável no consumo de memória.

Uma estratégia favorável é combinar ambos os tipos de métodos, ou seja, um método de ASVO determina o ordenamento inicial, mas um método de reordenamento é disparado conforme necessário. O método mais popular de DVO, conhecido por *Sifting*, é apresentado em (RUDELL, 1993). Este método consiste em deslocar uma variável seqüencialmente por todos os níveis do ROBDD e, numa estratégia gulosa, fixá-la na posição que deixe o ROBDD com a aparente menor quantidade de nodos. Este método pode apresentar melhores resultados quando aplicado por duas vezes consecutivas. Em geral, os métodos de reordenamento são baseados na operação básica de permuta de variáveis adjacentes, como apresentada na seção 3.6.

3.6 Permuta de variáveis adjacentes

A solução para o problema da *permuta de variáveis adjacentes* fornece uma operação fundamental aos métodos de reordenamento. O problema pode ser definido formalmente da seguinte maneira:

> *Definição*: Seja G_1 um π_1-BDD com o conjunto de variáveis X_n e a ordem π_1: $X_n \rightarrow \{1, ..., n\}$ $(1 \leq k < k+1 \leq n)$ e seja $\pi_1(x_k) = k$ e $\pi_1(x_{k+1}) = k+1$, obter um π_2-BDD G_2 que represente as mesmas funções de G_1 e onde $\pi_2(x_k) = \pi_1(x_{k+1})$ e $\pi_2(x_{k+1}) = \pi_1(x_k)$.

A solução para este problema pode ser observada realizando-se a decomposição de Shannon para duas variáveis em ordem alternada, como ilustrado a seguir:

$$f_{x_k x_{k+1}} = \bar{x_k} \cdot \bar{x_{k+1}} \cdot f_{00} + \bar{x_k} \cdot x_{k+1} \cdot f_{01} + x_k \cdot \bar{x}_{k+1} \cdot f_{10} + x_k \cdot x_{k+1} \cdot f_{11}$$

$$f_{x_{k+1} x_k} = \bar{x_k} \cdot \bar{x_{k+1}} \cdot f_{00} + \bar{x_k} \cdot x_{k+1} \cdot f_{10} + x_k \cdot \bar{x}_{k+1} \cdot f_{01} + x_k \cdot x_{k+1} \cdot f_{11}$$

Portanto, a permuta de variáveis pode ser realizada simplesmente trocando-se f_{10} por f_{01} e vice-versa. Esta operação é local em relação aos níveis envolvidos, ou seja, pode ser realizada sem a necessidade de alterar os vértices que estão acima do nível k ou abaixo do nível k+1.

Na prática, a operação é realizada percorrendo-se os vértices do nível k e aplicando a modificação necessária conforme o caso. Os casos triviais ocorrem quando (a) um vértice do nível k não possui sucessores em k+1 ou então (b) um vértice do nível k+1 não possui antecessores no nível k. Para estes casos, nenhuma alteração é necessária além da permuta do nível dos vértices, como ilustrado na Figura 3.7.

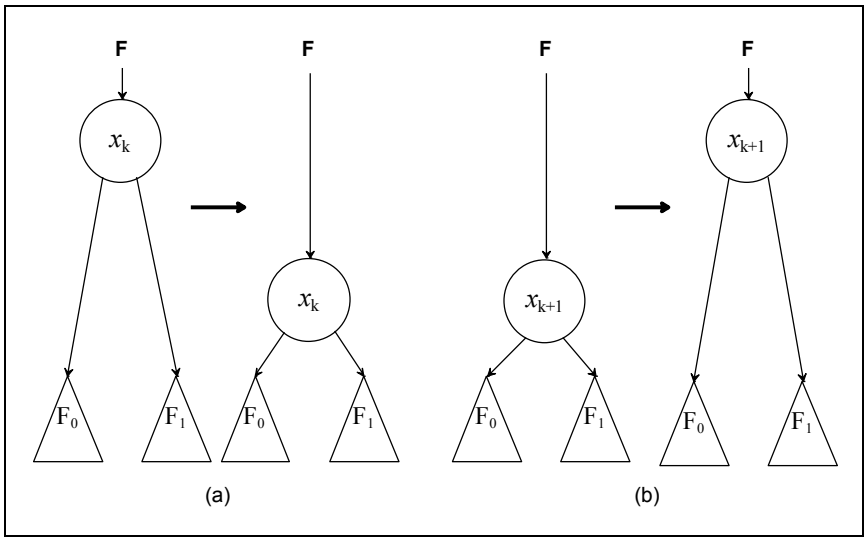

Figura 3.7: Casos triviais da permuta de variáveis adjacentes

Os demais casos, onde o vértice do nível *k* possui pelo menos um sucessor em *k*+1, podem ser deduzidos do caso geral apresentado na Figura 3.8.

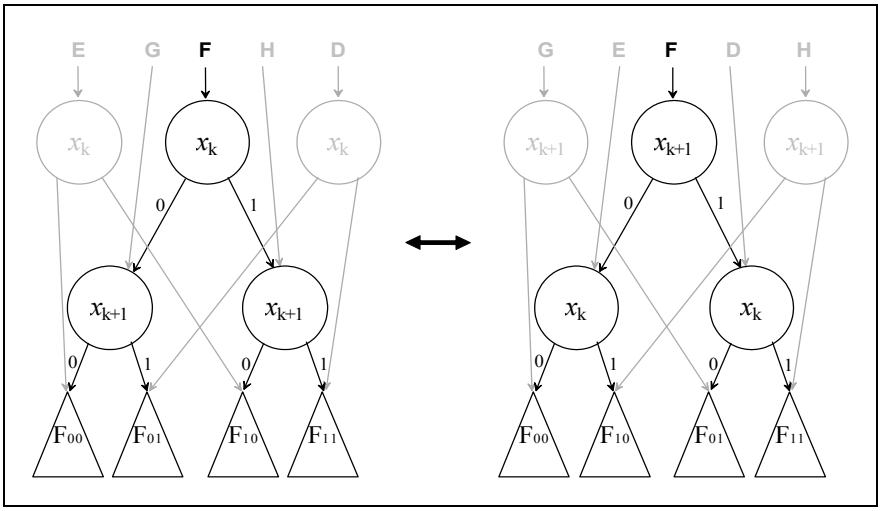

Figura 3.8: Caso geral da permuta de variáveis adjacentes

O caso geral pode ser tratado através das seguintes operações:

- A variável de F é atualizada para x_{k+1}
- Se o vértice G = *zero*(F) não está no nível *k*+1, então considerar $F_{00} = F_{01} = G$.
- Se o vértice H = *um*(F) não está no nível *k*+1, então considerar $F_{10} = F_{11} = H$.
- Se $F_{00} \neq F_{10}$ então o *arco-zero*(F) passa a apontar para o vértice $E = (x_k, F_{00}, F_{10})$. Se o vértice não existir então ele deve ser criado. Se $F_{00} = F_{10}$ então o *arco-zero*(F) deverá apontar para $F_{00} = F_{10}$ devido à regra de redução por eliminação.
- Se $F_{01} \neq F_{11}$ então o *arco-um*(F) passa a apontar para o vértice $D = (x_k, F_{01}, F_{11})$. Se o vértice não existir então ele deve ser criado. Se $F_{01} = F_{11}$ então o *arco-um*(F) deverá apontar para $F_{01} = F_{11}$ devido à regra de redução por eliminação.
- Se os vértices G e H não possuírem antecessores após a permuta então eles podem ser removidos.

3.7 Técnicas de programação para um pacote de BDDs eficiente

Uma *tabela hash* associa um *valor* a uma *chave*. Uma *função de hash*, aplicada à chave, seleciona qual de *N* listas ligadas está armazenado o par (*chave, valor)*. O *fator de carga* de uma tabela hash é definido como $\alpha = n / N$, onde *n* é o número de chaves armazenadas na tabela.

Uma *função de memória* para a função F é uma tabela de valores (x, F(x)) que já foram calculados. Se F é chamada com o argumento x novamente então F(x) é retornada sem necessidade de recálculo.

Um *cache baseado em hash* é uma tabela hash onde não se usam listas ligadas para resolver as colisões. Em vez disso, no momento da inserção, o elemento existente na posição em particular é substituído pelo novo. No momento da consulta, se o elemento não coincidir com a chave armazenada então ocorre uma falta de cache e nenhum elemento é retornado.

Uma *forma fortemente canônica* é uma forma que reduz a complexidade de um teste de equivalência entre elementos de um conjunto. Um *identificador único* é associado a cada elemento do conjunto, então o teste de equivalência é um simples teste escalar entre os identificadores únicos de cada elemento.

A *coleta de lixo* é uma classe de técnicas para liberar periodicamente a memória que não é mais utilizada por funções de interesse do usuário.

3.8 Estruturas de um pacote de BDDs eficiente

Além da estrutura dos vértices, que deve ser projetada de forma a não desperdiçar memória, outras estruturas são importantes para a eficiência de um pacote de BDDs. Em (BRACE, 1990), uma tabela hash é utilizada para manter uma forma fortemente canônica, garantindo que cada vértice represente uma função booleana diferente de forma eficiente. Esta tabela é denominada tabela-única, como definido a seguir.

A *tabela-única* mapeia uma tripla (x_i, G, H) para um vértice F = (x_i, G, H). Cada vértice do BDD possui uma entrada na tabela-única. Antes de se adicionar um vértice ao BDD, é realizada uma consulta à tabela-única para determinar se um vértice equivalente já não existe. Se existir então o vértice existente será usado, senão o novo vértice será adicionado ao BDD e uma nova entrada é criada para ele na tabela-única. Assume-se que quando for criado um novo vértice F, os vértices G e H tenham sido previamente criados. Portanto, a função F está representada no BDD se e somente se a tripla (x_i, G, H) está presente na tabela-única.

A *tabela-calculada* é uma função de memória para acelerar a operação ITE. Uma vez calculada a operação ITE(F, G, H), seu resultado fica armazenado em alguma posição da tabela-calculada. Desta maneira, quando houver uma nova chamada à ITE, com estes mesmos argumentos, será retornado o resultado previamente calculado e armazenado na tabela-calculada, dispensando qualquer recálculo da ITE. Em (BRACE 1990) a tabela-calculada é implementada como um *cache baseado em hash*, onde a *chave* é a tripla (F, G, H) e o *valor* é o resultado da ITE(F, G, H).

Um *vetor de variáveis* (ou *mapa de variáveis*) pode ser utilizado para manter o relacionamento entre o nome de uma variável e sua representação numérica interna.

Na prática, (BRACE, 1990) sugere que a tabela-única e o BDD sejam combinados numa única estrutura de dados. Para isto, utiliza-se um campo adicional nos vértices que aponta o próximo vértice na cadeia de colisões da tabela-única. Assim, a estrutura básica de um vértice necessita de campos para as seguintes informações:

- Variável
- Vértice apontado pelo arco-zero
- Vértice apontado pelo arco-um
- Vértice seguinte na cadeia de colisão da tabela-única

Dependendo da implementação, podem existir outros campos na estrutura dos vértices, além dos citados. Por exemplo, é comum utilizar-se um campo para manter a *contagem de referências* a partir de outros vértices ou funções do usuário (*fórmulas*). Este campo pode ser usado na coleta de lixo, observando que os vértices que não representam funções de interesse do usuário possuem zero referências.

3.9 Algoritmo da ITE

Considerando as definições das seções anteriores, apresenta-se o algoritmo da operação ITE. A rotina ITE recebe três vértices de BDD como argumentos e retorna o vértice que representa o resultado da operação. O algoritmo apresentado na Figura 3.9 é adaptado de (BRACE, 1990).

```
ITE(F,G,H){
  se (caso terminal da recursão){
      retornar resultado;
  } senão se (a tabela-calculada tem a entrada {F,G,H}){
        retornar resultado;
  } senão {
      seja v a variável topo de {F,G,H};
      T := ITE(F_{v=1},G_{v=1},H_{v=1});
      E := ITE(F_{v=0},G_{v=0},H_{v=0});
      Se (T = E) retornar T;
      R := ache_ou_adicione_à_tabela_única(v,T,E);
      insira_na_tabela_calculada({F,G,H},R);
      retornar R
  }
}
```

Figura 3.9: Algoritmo da ITE

3.10 Algoritmo da permuta de variáveis adjacentes

Considerando um pacote de BDDs projetado com as estruturas da seção 3.8, definiu-se um algoritmo para realizar a permuta de variáveis. A rotina permutaAdjacentes recebe, além do nível k, duas listas como argumentos. A primeira (*listaPai*) contém todos os vértices do nível *k* e a segunda (*listaFilho*) todos os vértices do nível *k*+1. O algoritmo da Figura 3.10 é baseado no método geral apresentado na seção 3.6.

```
permutaAdjacentes(k, listaPai, listaFilho) {
  para cada vértice F da listaPai {
    T := um(F);
    E := zero(F);
    se (índice(T) = x_{k+1}) ou (índice(E) = x_{k+1}) {
      retire F da listaPai;
      retire F da tabela-única;
      se (índice(T) = x_{k+1}) {
        F_11 := um(T);
        F_10 := zero(T);
      } senão {
        F_11 := T;
        F_10 := T;
      }
      se (índice(E) = x_{k+1}) {
        F_01 := um(E);
        F_00 := zero(E);
      } senão {
        F_01 := E;
        F_00 := E;
      }
      se (F_11 = F_01) um(F) := F_11;
      senão {
        um(F) := ache_ou_adicione_à_tabela_única(x_k,F_11,F_01);
        se (não está) coloque um(F) na listaPai;
      }
      se (F_10 = F_00) zero(F) := F_10;
      senão {
        zero(F) := ache_ou_adicione_à_tabela_única(x_k,F_10,F_00);
        se (não está) coloque zero(F) na listaPai;
      }
      adicione_à_tabela_única(F);
      se (T não é mais necessário) coloque T na cesta de lixo;
      se (E não é mais necessário) coloque E na cesta de lixo;
    }
  }
  para cada vértice F da listaPai {
    índice(F) := x_{k+1};
    acerte índice(F) na tabela_única para x_{k+1};
  }
  para cada vértice F da listaFilho {
    índice(F) := x_k;
    acerte índice(F) na tabela_única para x_k;
  }
  atualize o vetor (ou mapa) de variáveis para a nova ordem;
}
```

Figura 3.10: Algoritmo da permuta de variáveis adjacentes

4 FAMÍLIAS LÓGICAS DERIVADAS DE BDDS

Os circuitos combinacionais podem ser vistos como uma rede de funções booleanas interconectadas, onde cada porta lógica (ou célula, considerando-se leiaute) implementa uma função booleana específica. Cada função booleana pode ser representada por um BDD e através da associação de transistores a arcos de BDD (ver Figura 4.1) pode-se derivar uma rede de transistores que implementa a função desejada. Este capítulo apresenta as características de seis famílias lógicas onde as redes de transistores são derivadas de BDDs.

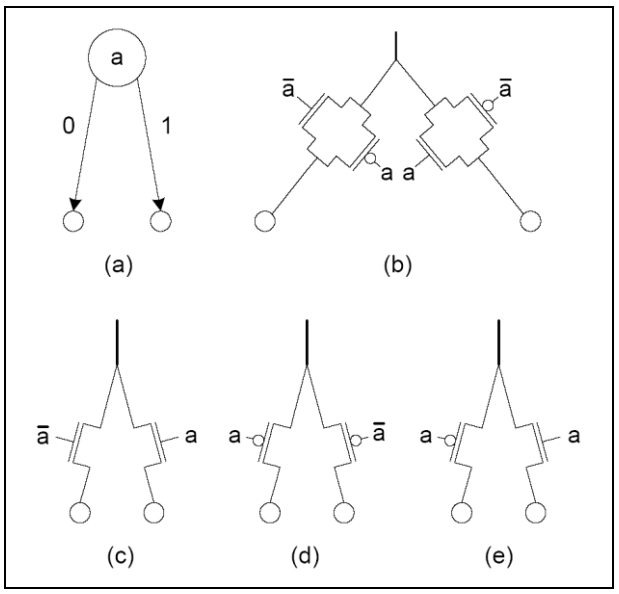

Figura 4.1: Associando transistores a arcos de BDDs

São obtidas três famílias diretamente a partir da estrutura de um BDD, enquanto outras três famílias são obtidas aplicando-se otimizações às redes de transistores da família simples correspondente. As famílias que possuem otimizações tendem a levar ao mapeamento de circuitos com menos transistores e menor atraso em relação à família equivalente sem otimizações.

Embora se saiba que a decomposição inicial da rede booleana tenha forte influência sobre as características finais de um circuito, este trabalho não explora nenhum método novo para melhorar esta decomposição. Em vez disso, utiliza-se uma decomposição inicial fixa dos circuitos, obtida com auxílio da ferramenta SIS (Berkeley). Esta decomposição inicial da rede booleana é equivalente à utilizada no mapeamento para FPGAs com LUTs de quatro entradas (LUT4), onde cada nodo da rede é uma função booleana de quatro entradas.

Uma vantagem de uma decomposição com funções de até quatro entradas é que qualquer célula derivada de ROBDDs terá no máximo quatro transistores em série. Esta limitação dispensa a inserção de *buffers* no interior de redes de transistores como é realizada em (BUCH, 1997). Outra vantagem é que algoritmos exatos, intratáveis para funções com muitas variáveis, são aplicados em tempo aceitável para funções com quatro variáveis.

Este capítulo apresenta as características de seis famílias lógicas derivadas de BDDs. As famílias NPTL, CPTL e DCPTL são apresentadas respectivamente nas seções 4.1, 4.2 e 4.4. A seção 4.3 apresenta a otimização por entradas em drenos, aplicada às famílias NPTL e CPTL. A seção 4.5 apresenta a versão da família DCPTL que possui um esquema de otimização diferenciado.

4.1 Derivação da família NPTL

A derivação de redes de transistores para a família NPTL é a mais direta possível. A partir de um ROBDD qualquer, associa-se um transistor NMOS para cada arco, como ilustrado na Figura 4.2.

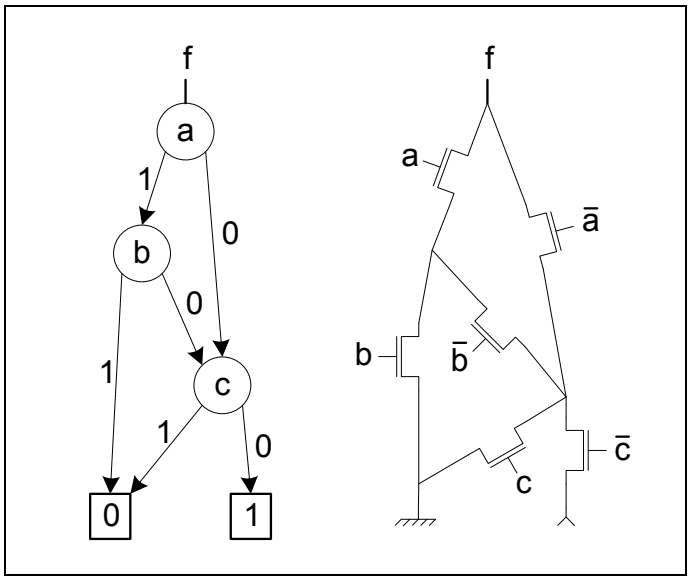

Figura 4.2: Associando transistores NMOS aos arcos do ROBDD

Como pode ser observado na figura acima, para cada arco-um (arco-zero) de um vértice *v* foi associado um transistor NMOS controlado pelo literal positivo (negativo) da variável *índice*(*v*). O critério para escolha da polaridade do sinal de controle é conseqüência da natureza elétrica de um transistor NMOS. O comportamento ideal deste tipo de transistor é permitir a passagem do sinal quando recebe o valor lógico positivo (booleano um, ou tensão de V_{dd}) em seu terminal de controle e cortar a passagem do sinal quando receber o valor lógico negativo (booleano zero, ou tensão de *GND)*. Portanto, associar o arco-um (arco-zero) a um transitor NMOS controlado pelo literal positivo (negativo), manterá o comportamento elétrico conforme idealizado por um arco de BDD.

Uma característica importante a ser observada na família NPTL é a degradação do sinal na saída da célula. Esta degradação é conseqüência do uso exclusivo de transistores NMOS, visto que a constituição física deste tipo de transistor não oferece

boa condução elétrica do valor lógico positivo (booleano um). Neste trabalho, a estratégia adotada para contornar o problema da degradação do sinal foi adicionar um inversor CMOS antes da saída da célula, assim restabelecendo a intensidade adequada do sinal na saída. A possibilidade da lógica estar invertida em decorrência deste inversor é considerada durante o mapeamento do circuito.

Outra característica da família NPTL é o potencial para circuitos que ocupam menos área do que para as outras famílias, devido à baixa contagem de transistores nas redes derivadas.

4.2 Derivação da família CPTL

A derivação de redes de transistores para a família CPTL é um pouco mais elaborada do que para a família NPTL. Na família CPTL são associadas chaves CMOS (*transmission gates*) aos arcos de um BDD em vez de um único transistor (ver Figura 4.3). Uma chave CMOS é composta por dois transistores em paralelo, sendo um deles NMOS e outro PMOS. A atribuição do literal ao terminal de controle do transistor NMOS é realizada da mesma forma como é feita para a família NPTL. Entretanto, para o transistor PMOS, utiliza-se o literal na polaridade inversa, visto que o comportamento de um transistor PMOS é o oposto do transistor NMOS, em se tratando da variável de controle.

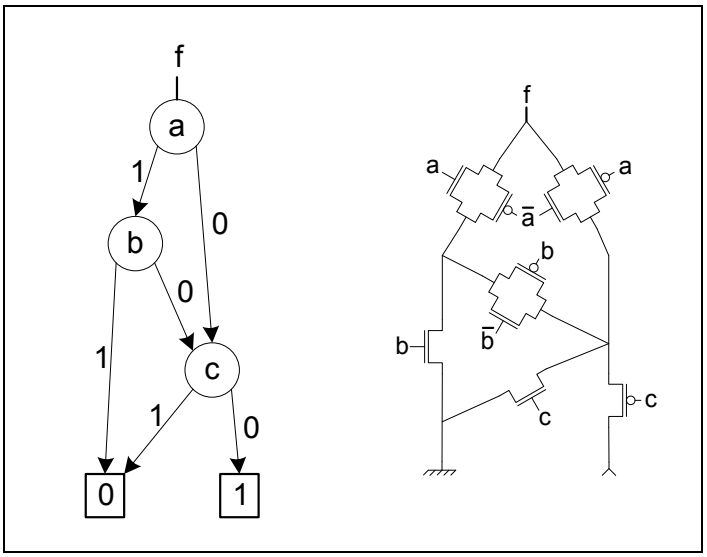

Figura 4.3: Associando chaves CMOS aos arcos do ROBDD

A vantagem com o uso de chaves CMOS em vez de um único transistor é que uma chave CMOS oferece boa condução elétrica para ambos os valores lógicos positivo e negativo, dispensando-se a necessidade de adicionar um inversor com o propósito de restabelecer o sinal na saída da célula, como ocorre na família NPTL. Em contrapartida, a família CPTL necessita de quase duas vezes mais transistores do que a família NPTL, refletindo em células que ocupam mais área e possuem mais capacitâncias parasíticas.

Um outro aspecto importante a ser observado na derivação na família CPTL é que os arcos que levam aos terminais são substituídos por um único transistor em vez de uma chave CMOS. Os arcos que levam ao terminal-0 derivam somente transistores NMOS,

enquanto os arcos que levam ao terminal-1 derivam somente transistores PMOS. Assim, economiza-se um transistor para cada arco que leva a um nodo terminal.

4.3 Otimização para as famílias NPTL e CPTL

A partir das famílias simples NPTL e CPTL, podem ser obtidas outras duas famílias aplicando-se uma otimização trivial. Esta otimização consiste na remoção dos transistores que são controlados por uma variável específica, introduzindo o sinal desta variável diretamente na rede como uma variável de passagem. A Figura 4.4 ilustra a aplicação desta otimização.

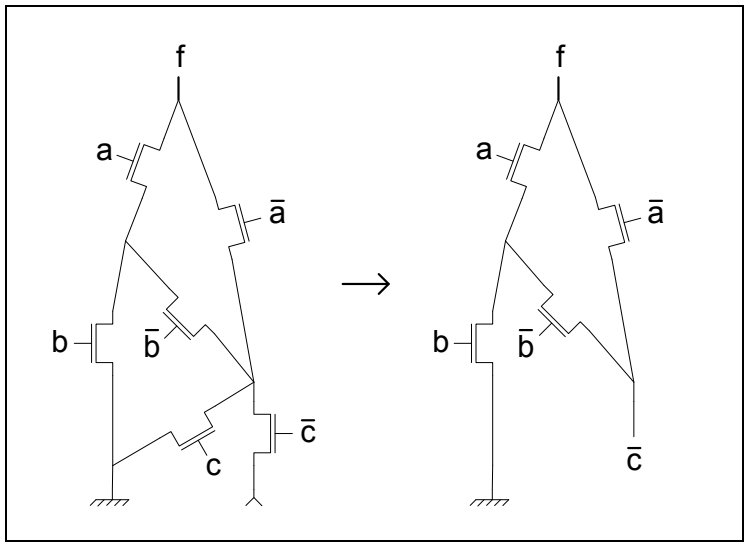

Figura 4.4: Otimização da rede transistores com entradas em drenos

A conseqüência imediata desta otimização é a derivação de células que possuem menor quantidade de transistores. Entretanto, a quantidade de transistores em série torna-se um problema para a variável de passagem, como ilustrado na Figura 4.5.

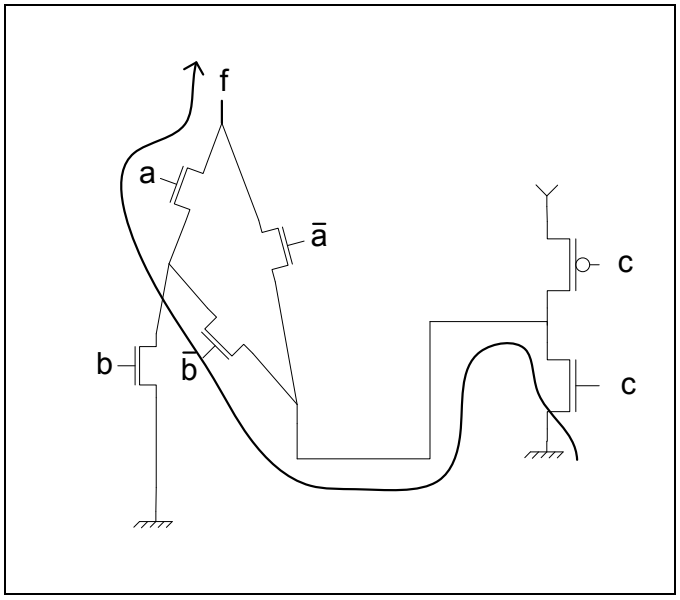

Figura 4.5: Transistores em série encadedados em mais de uma célula

Como indicado pela seta, a condução da variável de passagem não se restringe aos transistores em série de uma única célula, forçando a inserção de buffers ou inversores entre células para evitar cadeias longas de transistores em série. Devido à possibilidade da formação de longas cadeias de transistores em série, torna-se obrigatória a inserção de buffers entre as células da família CPTL com otimização.

4.4 Derivação da família DCPTL

A derivação de redes de transistores para a família DCPTL é semelhante a da família CPTL, com a diferença que os transistores PMOS e NMOS são derivados em planos disjuntos. A Figura 4.6 ilustra a derivação da família DCPTL.

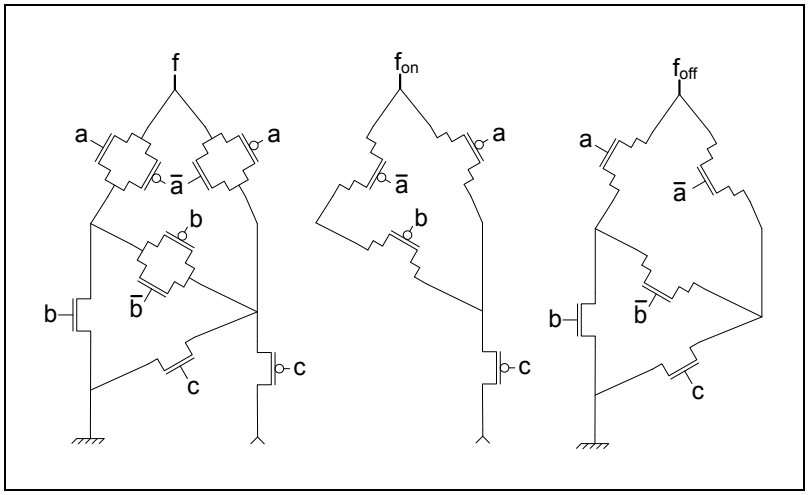

Figura 4.6: Associando transistores PMOS e NMOS em planos disjuntos

Para derivar o plano P (plano N), associam-se transistores PMOS (NMOS) aos arcos, com exceção dos arcos que levam ao terminal-0 (terminal-1). As saídas de cada plano são conectadas formando uma rede que possui qualidades semelhantes às de redes CMOS estáticas convencionais, com a diferença da não-complementaridade e da estrutura não necessariamente série/paralela de ambos os planos. Existe a possibilidade de derivar ambos os planos empregando ordenamentos diferentes de variáveis, embora este efeito não seja explorado neste trabalho.

O comprimento da cadeia mais longa de transistores em série será o mesmo em ambos os planos derivados e equivale ao comprimento do caminho mais longo no ROBDD para a função desejada. Isto representa uma vantagem em relação à topologia CMOS estática complementar série/paralelo para algumas funções. Por exemplo, uma função XOR de quatro entradas pode ser implementada com apenas quatro transistores em série na família DCPTL, enquanto uma topologia série/paralelo equivalente precisaria de oito transistores em série.

Assim como na família CPTL, existe boa condução do sinal até a saída da célula, dispensando o uso de inversores para restaurar o sinal enquanto certo limite de transistores em série seja obedecido.

4.5 Otimização para a família DCPTL

A técnica de otimização aplicada à família DCPTL é mais complexa do que as apresentadas para as famílias NPTL e CPTL. A técnica de otimização que explora a propriedade de *unateness* foi introduzida em (POLI, 2003) e posteriormente empregada em (ROSA JR, 2006). Esta técnica explora a propriedade de *unateness* em cada nodo do ROBDD e substitui alguns dos arcos por circuito-fechado ou circuito-aberto em vez de transistores. Em relação a família DCPTL simples, a família DCPTL otimizada oferece ganhos em área, atraso e potência. A validação das redes é feita através de simulação funcional para todos os vetores de entrada, consumindo mais tempo de processamento para gerar cada célula.

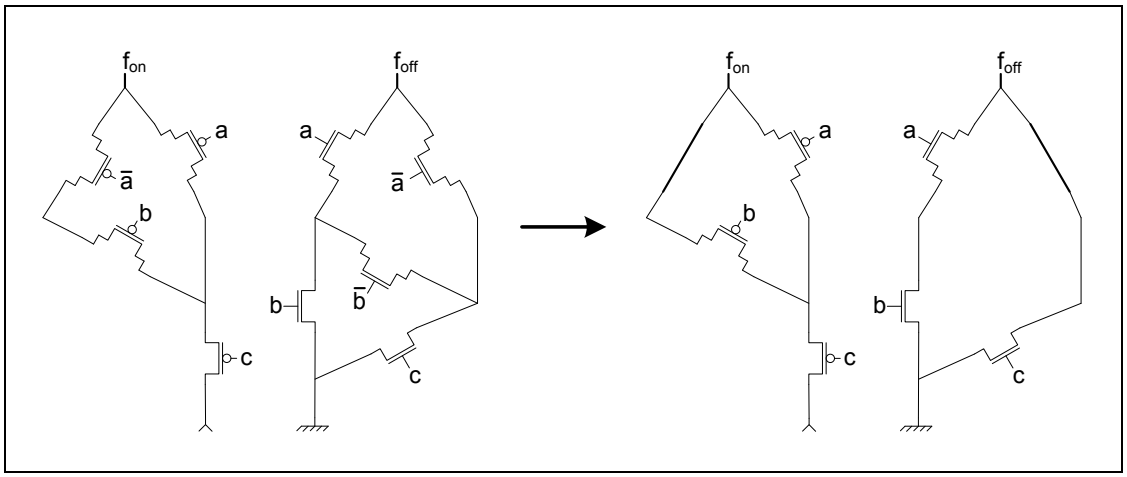

Figura 4.7: Otimização para a família DCPTL

As células lógicas desta família apresentam as mesmas qualidades da DCPTL simples, porém com menor quantidade de transistores. Além de células mais compactas, diminui-se também a necessidade de inversores para se obter os sinais de controle na polaridade complementar. A Figura 4.7 mostra uma célula DCPTL antes e depois de aplicar as otimizações. As células oferecem boa condução elétrica de ambos os valores lógicos positivo e negativo. As otimizações não causam a formação de longas cadeias de transistores em série como acontece nas otimizações por entradas em drenos e, portanto o inversor na saída da célula é dispensável, respeitando-se certo limite de transistores em série. Outra vantagem é a eliminação dos inversores que seriam necessários para gerar o sinal de controle nos transistores removidos.

Existe a possibilidade de uma célula DCPTL otimizada implementar a função booleana invertida caso a cadeia mais longa no plano P seja maior que a cadeia mais longa no plano N. Procura-se manter o comprimento da cadeia mais longa no plano P sempre menor ou igual ao comprimento da cadeia mais longa no plano N, invertendo-se a polaridade da função quando necessário. Esta inversão ocorre porque a resistência associada a transistores PMOS é geralmente maior do que a resistência de transistores NMOS.

5 SOFTWARE DESENVOLVIDO

Para a realização dos experimentos propostos neste trabalho tornam-se necessários diferentes componentes de software. Para avaliar o ordenamento de ROBDDs é necessário, em primeiro lugar, um pacote de BDDs que forneça suporte a este tipo de estrutura. Diferentes pacotes de BDDs podem ser encontrados na Internet (www.bdd-portal.org), cada qual com peculiaridades que nem sempre vão de encontro às necessidades do usuário (JANSSEN, 2003). Neste trabalho, optou-se pelo desenvolvimento de um novo pacote de BDDs, direcionado principalmente às necessidades imediatas da geração de redes de transistores.

Um outro pesquisador do mesmo grupo de pesquisa utilizou o novo pacote de BDDs para implementar um pacote de geração de redes de transistores. Este software permite a geração de redes de transistores para diferentes famílias lógicas derivadas de ROBDDs (ROSA JR, 2006). A interação entre os pesquisadores no desenvolvimento deste software foi importante para a validação e testes do novo pacote de BDDs.

Para alcançar o objetivo proposto nesta dissertação, foi desenvolvida uma ferramenta de extração de caminhos potencialmente críticos de um circuito mapeado com células derivadas de ROBDDs. Esta ferramenta é capaz de ler a descrição lógica de um circuito no formato BLIF, extrair caminhos potencialmente críticos do circuito (onde as redes de transistores são ordenadas conforme a metodologia proposta) e gerar a descrição SPICE pronta para a simulação transiente de cada caminho potencialmente crítico. A geração da descrição SPICE para células individuais é baseada no gerador de redes de transistores desenvolvido em (ROSA JR, 2006).

Uma variação da ferramenta de extração de caminhos foi implementada para obter a medida de área dos circuitos. O critério de ordenamento de variáveis foi aplicado no mapeamento do circuito inteiro, ou seja, a contagem de transistores é feita para todas as células em vez de apenas aquelas que pertencem a caminhos potencialmente críticos. O total de transistores é utilizado como medida de área do circuito.

Todo o software foi desenvolvido com a linguagem de programação Java. O motivo da opção por Java foi de que boa parte do software do grupo de pesquisa está implementado com esta linguagem. Isto facilitou o reuso de alguns componentes, como o leitor de descrições BLIF.

Este capítulo apresenta o software que foi desenvolvido para a realização dos experimentos propostos nesta dissertação. A seção 5.1 apresenta as características e decisões de projeto para a implementação do novo pacote de BDDs. A seção 5.2 apresenta os componentes de software associado à geração de redes de transistores. A seção 5.3 apresenta detalhes da ferramenta que foi desenvolvida para extração dos caminhos potencialmente críticos.

5.1 Pacote de BDDs

Um novo pacote de BDDs foi desenvolvido totalmente com a linguagem de programação Java. Apesar da principal aplicação em vista para o novo pacote ser a geração de redes de transistores, outras funcionalidades de interesse foram implementadas. A lista abaixo apresenta as principais funcionalidades implementadas:

- *Síntese de ROBDDs* (criar variável, entrada por expressão com parênteses balanceados, entrada por tabela-verdade, operação ITE, operações binárias AND, OR, etc.)
- *Manipulação de ROBDDs* (obter os sucessores de um nodo, obter a variável de um nodo, a posição de uma variável na ordem, SAT-count, etc.)
- *Permuta de variáveis adjacentes* (usada repetidamente para colocar uma variável em qualquer nível, após a construção do BDD)
- *Minato-ISOP* (MINATO, 1996)
- *Cálculo do número mínimo de transistores em série para implementar uma porta lógica complexa* (SCHNEIDER, 2005)

5.1.1 Organização do pacote de BDDs implementado

O pacote é formado por três classes principais *BDDManager*, *BDDNode* e *ComputedTriple*. Outras duas classes auxiliares *BDD2SOP* e *Expression2BDDParser* são utilizadas para o processamento de expressões booleanas. A Figura 5.1 ilustra a organização do pacote.

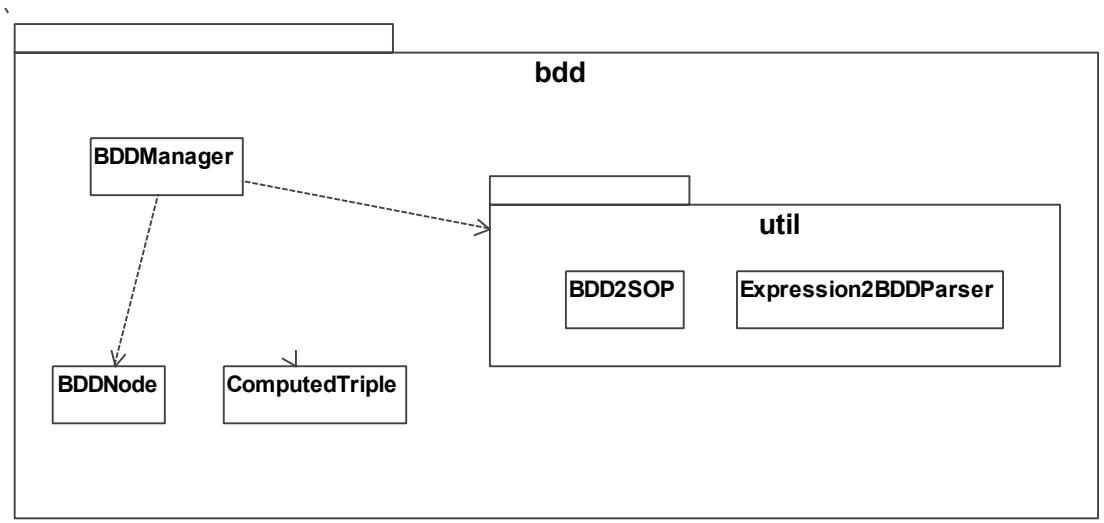

Figura 5.1: Organização do novo pacote de BDDs

A classe principal do pacote é a classe *BDDManager*. Esta classe contém as estruturas de dados para as variáveis, a tabela-única e a tabela-calculada. É a interface desta classe que o usuário deve conhecer para utilizar o pacote. As demais classes são utilizadas pela classe *BDDManager* e não precisam ser conhecidas pelo usuário. A documentação para os métodos implementados na classe *BDDManager* pode ser encontrada no ANEXO A.

5.1.2 Estrutura dos nodos

As estruturas de dados são elementos críticos para o desempenho e consumo de memória de um pacote de BDDs. Algumas aplicações necessitam de BDDs com alguns milhões de nodos e, portanto, a estrutura dos nodos deve consumir a menor quantidade de palavras de memória possíveis, envolvendo complexas manipulações de campos de bits. No entanto, para a geração de redes de transistores, dificilmente serão encontrados BDDs que chegam a uma centena de nodos. Por isso, a estrutura dos nodos é bastante simples no pacote implementado. Cada nodo possui quatro campos contendo valores inteiros (um inteiro em Java sempre tem sinal e ocupa 32 bits) como mostrado abaixo:

```
int index; // a variable index
   int low; // edge-0 (ELSE)
  int high; // edge-1 (THEN)
int refCount; // reference counter
```

É possível uma implementação em Java que seja baseada na manipulação de campos de bits com auxílio de operações de *bit-shifting*. Foram feitos alguns experimentos com sucesso utilizando nodos que possuíam apenas oito bytes. Porém, optou-se por uma implementação mais simples, mais fácil de manter e provavelmente mais rápida, ocupando 16 bytes (cada inteiro ocupa quatro bytes).

Deve-se observar que esta quantidade de memória não considera dados adicionais associados ao gerenciamento de instâncias e coleta de lixo da máquina virtual Java. Sabe-se que pelo menos mais quatro bytes são alocados para a referência do nodo no vetor de nodos e outros oito bytes são alocados para a referência e o índice do nodo na tabela-única. Portanto, o consumo de memória será de pelo menos 28 bytes por nodo. O consumo real pode ser ainda maior, considerando que podem existir mais de uma entrada por nodo na tabela-calculada e cada entrada nesta tabela consome pelo menos mais 16 bytes.

5.1.3 Gerenciamento de variáveis

Cada variável é representada por um identificador único que corresponde à posição desta variável em um vetor de nomes de variáveis. A seguinte linha de código Java define este vetor:

```
ArrayList<String> variableNames;
```

Portanto, a primeira variável criada receberá o identificador com o valor zero, correspondente à primeira posição no vetor de variáveis. Esta variável inicial é criada no nível zero, que é o nível mais alto para uma variável no pacote.

Utiliza-se um outro vetor para manter a posição de cada variável no ordenamento de variáveis, como definido a seguir:

```
ArrayList<Integer> variableOrdering;
```

Cada posição deste vetor representa um nível e o conteúdo da posição é um identificador de variável. Para obter o mapeamento inverso, ou seja, a partir de um identificador de variável, obter a posição desta no ordenamento, implementou-se outro

vetor onde cada posição representa um identificador de variável e o conteúdo da posição é o nível que a variável ocupa, como definido a seguir:

```
ArrayList<Integer> variableLevels;
```

Estes dois últimos vetores têm seus elementos trocados de posição após operações de permuta de variáveis, porém o mesmo não ocorre para o vetor de nomes. Poderia-se utilizar um único vetor para manter o ordenamento de variáveis no pacote, porém optou-se pela utilização de dois vetores para acelerar as consultas.

5.1.4 Tabela-única e vetor-único de nodos

O uso das estruturas padrão do Java torna o projeto e a manutenção do pacote mais simples, apesar de não oferecerem o desempenho máximo que poderia ser obtido com uma implementação específica. A seguinte linha de código Java define a estrutura da tabela-única no pacote implementado:

```
HashMap<BDDNode, Integer> uniqueTable;
```

Em (BRACE, 1990), definiu-se que a tabela-única é uma tabela hash onde a chave é uma tripla-única que identifica um nodo de BDD e o valor retornado é a referência (ou ponteiro) para o nodo do BDD. Entretanto, na implementação deste trabalho, utilizou-se um esquema diferente, baseado em idéias apresentadas em (JANSSEN, 2001). Cada nodo possui um índice único (um valor inteiro) que o identifica. O valor deste índice corresponde à posição que o nodo ocupa em um vetor de nodos, como definido a seguir:

```
ArrayList<BDDNode> nodeVector;
```

Na tabela-única implementada, o valor retornado pela tabela hash é o índice do nodo e a chave é curiosamente uma instância de BDDNode. O método `boolean equals(Object o)` da classe BDDNode (a classe HashMap chama este método para comparar a chave de pesquisa com a chave armazenada) foi sobrescrito para comparar somente os três campos do nodo referentes a uma tripla-única, excluindo-se o campo de contagem de referências na comparação. Este esquema economiza a memória que seria utilizada para o armazenamento das chaves na tabela-única (a classe HashMap armazena chaves para resolver colisões), já que um nodo ocupa memória de qualquer forma.

Uma única instância da classe BDDNode é utilizada como chave-única e tem o valor de seus atributos alterados para cada consulta à tabela-única. Isto evita o custo desnecessário de instanciar uma nova chave para cada consulta.

5.1.5 Tabela-calculada

A função da tabela-calculada é acelerar a operação ITE, mantendo o resultado de operações anteriores e eliminando a necessidade de recalcular uma operação ITE com os mesmos argumentos. Esta estrutura foi implementada como definido a seguir:

```
HashMap<ComputedTriple, Integer> computedTable;
```

Assim como na tabela-única, utiliza-se uma única instância da classe ComputedTriple como chave de consultas à tabela-calculada. Esta classe contém três

campos inteiros correspondentes aos argumentos de uma operação ITE, como definidos abaixo:

```
 int If;
int Then;
int Else;
```

Na implementação atual, cada chamada à ITE cujos argumentos (F,G,H) não sejam encontrados na tabela-calculada implicam em uma nova entrada nesta tabela. Isto pode exigir um consumo excessivo de memória em aplicações que utilizam milhões de nodos, mas não implica em maiores problemas para a geração de redes de transistores.

5.1.6 Aplicações desenvolvidas com o novo pacote de BDDs

O pacote de BDDs desenvolvido como parte deste trabalho serviu de base para a implementação de diferentes aplicações. A principal aplicação desenvolvida foi um pacote para geração de redes de transistores, desenvolvido como parte do trabalho de outro pesquisador do grupo (ROSA JR, 2006). Este software permite a geração de redes de transistores para diferentes famílias lógicas. A classe principal desse pacote é a classe *SwitchNetwork*. Não se entrará em maiores detalhes sobre a implementação desta classe, porém a mesma é utilizada neste trabalho para gerar as redes de transistores com os diferentes ordenamentos de variáveis.

Em paralelo com o núcleo eficiente do pacote, foi desenvolvido um ambiente de linha de comando para testar as principais funcionalidades do pacote. Este ambiente oferece as funcionalidades apresentadas no início da seção 5.1 e foi integrado a algumas funcionalidades adicionais, como visualização de BDDs. O visualizador de BDDs chamado BDDeiro (GOMES, 2006) foi utilizado para gerar a Figura 2.4 e a Figura 2.5 desta dissertação. Uma listagem com os comandos do ambiente de linha de comando pode ser obtida através do comando “help”, cujo resultado é apresentado na Figura 5.2.

Além do pacote de geração de redes de transistores e do ambiente de linha de comando, outras aplicações utilizaram o mesmo pacote de BDDs.

Um método de decomposição funcional não-disjunta objetivando mapeamento tecnológico com células lógicas que respeitam o número mínimo de transistores em série é apresentado em (FLORES, 2006). O método é fortemente baseado em BDDs e utiliza os recursos de síntese e manipulação de BDDs, incluindo reordenamento de variáveis e cálculo do número mínimo de transistores em série.

Um outro pacote de software implementado com o novo pacote de BDDs visa o mapeamento tecnológico com células lógicas geradas sob demanda que respeitam o número mínimo de transistores em série (MARQUES, 2006). O pacote utiliza o mesmo gerador de redes de transistores utilizado neste trabalho e também utiliza a rotina de cálculo do número mínimo de transistores em série para funções com até oito variáveis. Este método permite obter redes lógicas que podem levar a resultados de maior qualidade no mapeamento de ASICs do que o método utilizado para decompor a rede lógica nesta dissertação (a decomposição nesta dissertação é obtida limitando-se o número de entradas com auxílio da ferramenta SIS).

```
add | a -> add function from truth table or expression (prefix
  "0x" for hexa)
and -> apply AND operator for two boolean function root nodes
b -> balance of a boolean funtion
cmos -> print a CMOS netlist from a root node
d -> density of a boolean function
e -> add function from expression
exit | quit -> exit from console
help | h -> display help on commands
inv -> find complement of a function
isop -> print a minato-morreale irredundant sum of products
  equation given a root node
lb -> print series transistor lower bound of a boolean function
movevar | mv -> move a variable to another level
nc -> negative cofactor a function with respect to a variable
nptl -> print a nPTL netlist from a root node
ocmos -> print an optimized CMOS netlist from a root node
onptl -> print an optimized nPTL netlist from a root node
optl -> print an optimized PTL netlist from a root node
or -> apply OR operator for two boolean function root nodes
pc -> positive cofactor a function with respect to a variable
print | p -> print the ROBDD from a root node
ptl -> print a PTL netlist from a root node
reset -> reset all ROBDD data structures
view | v -> view the ROBDD from a root node
w -> weight of a boolean function (SAT-COUNT)
xor -> apply XOR operator for two boolean function root nodes
nxor -> apply NXOR operator for two boolean function root nodes
```

Figura 5.2: Comandos do ambiente de linha de comando

5.2 Geração de redes de transistores

Foi criada uma classe adaptadora *SwitchNetworkWrapper* para encapsular a interface e o protocolo da classe *SwitchNetwork* do pacote de geração de redes de transistores desenvolvida como parte de outro trabalho (ROSA JR, 2006) e cujos detalhes de implementação não serão mencionados nesta dissertação. O construtor da *SwitchNetworkWrapper* recebe uma instância de *BDDManager* como argumento. A função booleana para a qual se quer derivar uma rede de transistores deve estar cadastrada na instância de *BDDManager* com o ordenamento de variáveis desejado. O método principal da classe *SwitchNetworkWrapper* é o método `getBDDSpiceSubckt()` com a seguinte assinatura:

```
public String getBDDSpiceSubckt(int root, String logicFamily)
```

O argumento `root` é o índice do nodo do BDD, retornado pela instância de *BDDManager*, que representa a função booleana da qual se quer derivar uma rede de transistores. O argumento `logicFamily` contém o nome da família lógica para a qual a

rede será derivada. O método retorna uma *String* contendo a descrição da rede como um sub-circuito SPICE. Por exemplo, para determinada função OR de três entradas, a descrição da Figura 5.3 foi fornecida pelo método:

```
* x9527+x47425+x47851
.subckt  gate_x47533  x47851  x9527  x47425  nx47851  nx9527  nx47425  nout
m0  6  nx47851  7  gnd  nmos w=wn l=lt ad=3.2e-12 as=3.2e-12 pd=8.4u ps=8.4u
m1  vdd  x47851  7  gnd  nmos w=wn l=lt ad=3.2e-12 as=3.2e-12 pd=8.4u ps=8.4u
m2  4  nx9527  6  gnd  nmos w=wn l=lt ad=3.2e-12 as=3.2e-12 pd=8.4u ps=8.4u
m3  vdd  x9527  6  gnd  nmos w=wn l=lt ad=3.2e-12 as=3.2e-12 pd=8.4u ps=8.4u
m4  gnd  nx47425  4  gnd  nmos w=wn l=lt ad=3.2e-12 as=3.2e-12 pd=8.4u ps=8.4u
m5  vdd  x47425  4  gnd  nmos w=wn l=lt ad=3.2e-12 as=3.2e-12 pd=8.4u ps=8.4u
xinv  7  nout  invsubckt
* PU,PD: 3,3
.ends gate_x47533
```

Figura 5.3: Descrição SPICE de uma célula lógica gerada automaticamente

O nome para o sub-circuito (o identificador encontrado logo após os comandos `subckt` e `ends`) pode ser alterado antes de gerar a descrição da rede com o método `setSubcktName(String)`. Se nenhum nome é fornecido então a rede é gerada com o nome padrão `gate_x`, onde `x` equivale ao índice do nodo no BDD da função representada. O nome do sub-circuito é importante no momento de instanciação do sub-circuito. Para se obter o nome do sub-circuito gerado utiliza-se o método `getSubcktName()`.

O exemplo apresentado é uma rede da família NPTL, portanto composto somente por transistores do tipo NMOS. O nome dos dispositivos (`nmos` no exemplo) e os parâmetros de dimensionamento (no exemplo `w=wn l=lt ad=3.2e-12 as=3.2e-12 pd=8.4u ps=8.4u`) podem ser definidos com uma única *String* através do método `setNMOSComplement(String)` ou `setPMOSComplement(String)` para transistores NMOS ou PMOS respectivamente.

A família NPTL é uma das famílias que possuem degradação do sinal na saída da célula, por isso foi colocada uma instância de inversor no interior do sub-circuito para restabelecer o sinal na saída. Em conseqüência deste inversor, a função implementada é na verdade o complemento da função esperada e este fato é reforçado pelo identificador `nout` no nó de saída do sub-circuito. A família DCPTL otimizada também pode ter a saída negada apesar de não possuir o inversor na saída. Isto ocorre no caso em que a função é invertida para impedir que o comprimento da maior cadeia de transistores em série no plano P seja maior que no plano N.

Um outro método importante oferecido pela classe *SwitchNetworkWrapper* é o método `getSensibilizationVector(String)`. Este método recebe o nome de uma das variáveis de entrada como parâmetro e retorna um *Map<String, Boolean>* que contém os valores lógicos não-controladores para as demais variáveis. O método serve para determinar os valores lógicos que serão aplicados às entradas laterais das células que compõem um caminho potencialmente crítico. Assim, um pulso de entrada é conduzido desde uma entrada primária até uma saída primária e o atraso de propagação é medido precisamente através da simulação elétrica. A descrição SPICE para a

simulação transiente é gerada automaticamente pela ferramenta de extração de caminhos apresentada na seção 5.3.

Para obter o número de transistores de uma rede gerada, utiliza-se os métodos `int getTransistorNCount()` ou `int getTransistorPCount()` para obter o número de transistores NMOS ou PMOS, respectivamente. Os transistores de um possível inversor dentro do sub-circuito da rede não estão incluídos nesta contagem. Pode-se descobrir se um sub-circuito gerado possui tal inversor através do método `hasInversor()`.

5.3 Ferramenta de extração de caminhos para simulação SPICE

A freqüência máxima de operação de um circuito digital síncrono é determinada basicamente pelo maior atraso de propagação de um sinal que vai de uma entrada primária até uma saída primária de um circuito combinacional. A seqüência de células lógicas e conectores pela qual este sinal se propaga é considerada o caminho crítico do circuito. Entretanto, determinar exatamente qual o caminho crítico de um circuito não é tarefa simples.

5.3.1 Modelos de atrasos

A estratégia deste trabalho para encontrar o caminho crítico de um circuito é baseada na solução apresentada em (CORREIA, 2005). São utilizados diferentes modelos de atrasos para estimar o atraso de cada sinal gerado por uma célula em relação às entradas primárias do circuito. Foram definidos dez modelos de atrasos como enumerados a seguir:

(i) *Atrasos unitários*: Cada célula lógica tem atraso unitário. Este modelo serve para encontrar o caminho mais longo do circuito. Este modelo também foi utilizado para obter o tempo de chegada dos sinais para o método de ordenamento de variáveis.

(ii) *Cadeia de transistores em série*: Cada porta lógica tem seu atraso determinado pelo comprimento da maior cadeia de transistores em série da porta. O número de transistores em série é importante para o atraso porque serve de abstração para a resistência elétrica que determina a velocidade de carga ou descarga da capacitância associada à saída da porta. Este modelo serve para encontrar o caminho cuja soma dos comprimentos da maior cadeia de transistores em série de cada porta seja a maior do circuito.

(iii) *Número mínimo de transistores em série*: Cada porta lógica tem seu atraso determinado pelo número mínimo de transistores em série para implementar a porta lógica de acordo com a função booleana desejada (SCHNEIDER, 2005). Este modelo serve para encontrar o caminho cuja soma dos comprimentos mínimos teóricos da maior cadeia de transistores em série de cada porta seja a maior do circuito.

(iv) *Fanout*: Cada porta lógica tem seu atraso determinado pelo seu *fanout*, ou seja, o número de portas conectadas à saída da porta. O *fanout* é importante porque fornece uma abstração para a capacitância associada às portas conectadas à saída da célula. Este modelo serve para encontrar o caminho cuja soma dos *fanouts* de cada porta seja a maior do circuito.

(v) *Cadeia de transistores em série e Fanout*: Cada porta lógica tem seu atraso determinado pela multiplicação do atraso definido em (ii) pelo atraso definido em (iv).

Este modelo serve para encontrar o caminho cuja soma do produto de (ii) por (iv) de cada porta seja a maior do circuito.

(vi) *Número mínimo de transistores em série e fanout*: Cada porta lógica tem seu atraso determinado pela multiplicação do atraso definido em (iii) pelo atraso definido em (iv). Este modelo serve para encontrar o caminho cuja soma do produto de (iii) por (iv) de cada porta seja a maior do circuito.

(vii) *Quadrado da cadeia de transistores em série*: Cada porta lógica tem seu atraso determinado pelo quadrado do comprimento da maior cadeia de transistores em série da porta. Este modelo serve para encontrar o caminho cuja soma dos quadrados dos comprimentos da maior cadeia de transistores em série de cada porta seja a maior do circuito.

(viii) *Quadrado do número mínimo de transistores em série*: Cada porta lógica tem seu atraso determinado pelo quadrado do número mínimo de transistores em série para implementar a porta lógica de acordo com a função booleana desejada. Este modelo serve para encontrar o caminho cuja soma dos quadrados dos comprimentos mínimos teóricos da maior cadeia de transistores em série de cada porta seja a maior do circuito.

(ix) *Quadrado da cadeia de transistores em série e fanout*: Cada porta lógica tem seu atraso determinado pela multiplicação do atraso definido em (vii) pelo atraso definido em (iv). Este modelo serve para encontrar o caminho cuja soma do produto de (vii) por (iv) de cada porta seja a maior do circuito.

(x) *Quadrado do número mínimo de transistores em série e fanout*: Cada porta lógica tem seu atraso determinado pela multiplicação do atraso definido em (viii) pelo atraso definido em (iv). Este modelo serve para encontrar o caminho cuja soma do produto de (viii) por (iv) de cada porta seja a maior do circuito.

Para a implementação dos modelos de atrasos foi criado o pacote *delaymodel*. A Figura 5.4 apresenta um diagrama de classes para o pacote *delaymodel*. Este pacote contém a interface *DelayModel* e a classe abstrata *AbstractDelayModel*. A interface *DelayModel* define o método `float getDelay(String output)` que deve receber o nome de um sinal como parâmetro de entrada e retornar o tempo de chegada associado ao sinal. A classe *AbstractDelayModel* implementa a interface *DelayModel* e define o protocolo básico para os demais modelos de atrasos implementados. Ela contém uma estrutura para o mapeamento de nomes de sinais para valores de atraso como definido a seguir:

```
protected Map<String,Float> delays;
```

Este mapa é utilizado para associar ou consultar rapidamente o valor correspondente ao atraso de chegada de um sinal, utilizando seu nome como chave. Na implementação do método `float getDelay(String output)` da classe *AbstractDelayModel*, o método busca recursivamente os atrasos dos sinais de entrada e retorna a soma do maior atraso entre os sinais de entrada com o atraso associado à própria célula cujas entradas estão sendo avaliadas. O valor absoluto para o atraso da célula é obtido através de uma chamada ao método abstrato `float getCellDelay(String output)`. Cada subclasse da classe *AbstractDelayModel* deve implementar este método para retornar o atraso associado a uma célula conforme um método de cálculo específico.

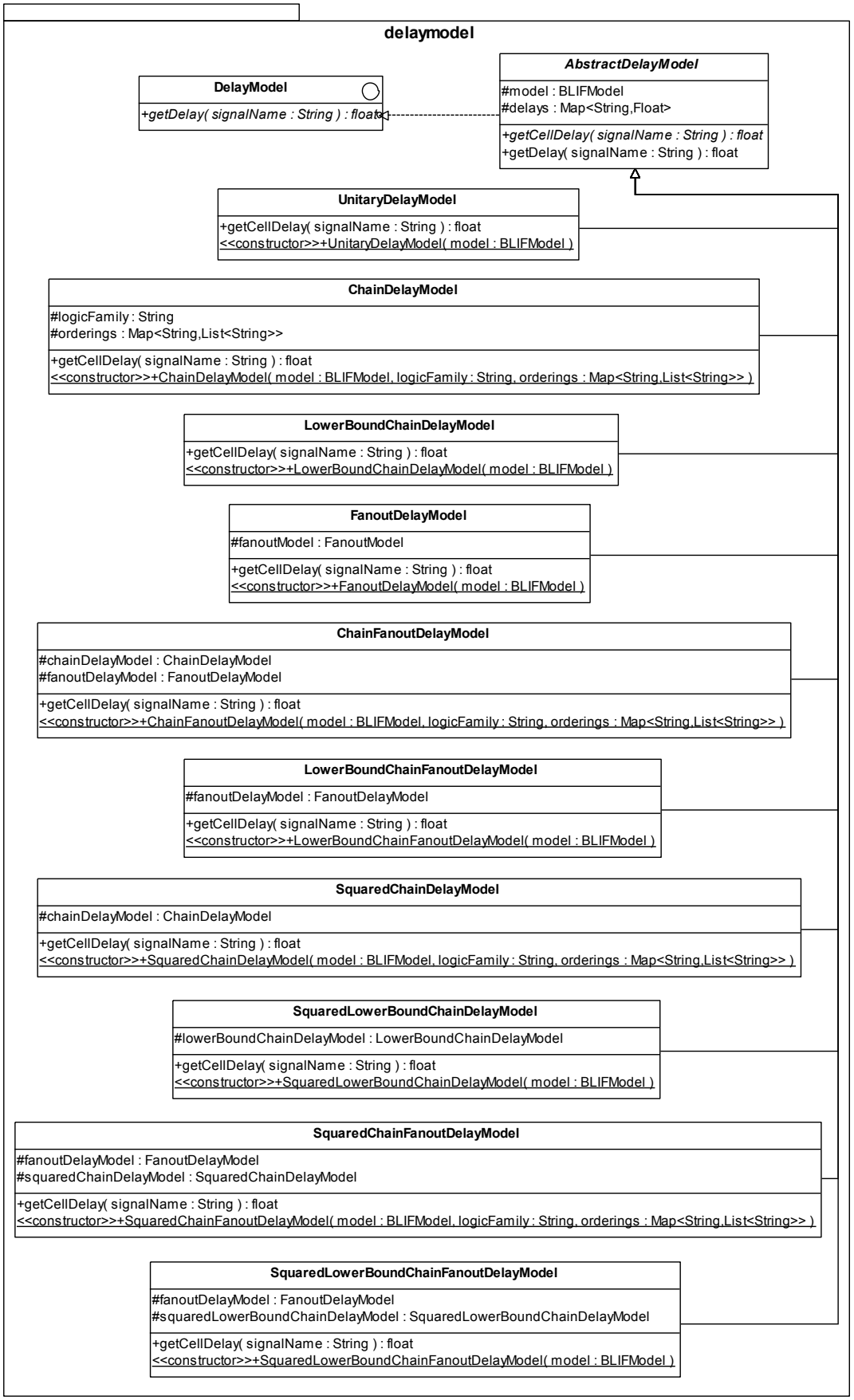

Figura 5.4: Um diagrama de classes para o pacote *delaymodel*

5.3.2 Busca de caminhos potencialmente críticos

A execução do método `public static void main()` da classe *BLIFCriticalPathExtractor* abre uma janela para o usuário selecionar os arquivos com extensão BLIF ou as pastas contendo os arquivos a serem processados. A ferramenta gera as descrições SPICE dos dez caminhos potencialmente críticos nas seis famílias lógicas com os quatro critérios de ordenamento para cada circuito.

Em cada circuito selecionado pelo usuário, inicialmente a ferramenta utiliza um *parser* para fazer a leitura da descrição BLIF, carregando as informações em uma estrutura de dados (uma instância da classe *BLIFModel*). Esta é a mesma estrutura de dados utilizada na construção dos modelos de atrasos. Adicionalmente, para um critério de ordenamento por vez, é construída uma estrutura de dados para mapear o nome de uma célula para a lista com as variáveis de entrada ordenadas, conforme é definida a seguir:

```
protected Map<String, List<String>> orderings;
```

A extração dos caminhos considera os ordenamentos armazenados nesta estrutura para decidir os casos em que há empate no atraso de duas ou mais entradas. Os modelos de atrasos que calculam o número de transistores em série também dependem desta estrutura, visto que o ordenamento pode influir neste parâmetro.

Para determinar um caminho potencialmente crítico, a saída primária com o maior atraso é escolhida como a saída potencialmente crítica. No caso de empate entre os atrasos de duas ou mais saídas primárias, por simplicidade escolhe-se a saída primária que é encontrada primeiramente na descrição BLIF.

Para encontrar um caminho potencialmente crítico, utiliza-se chamadas ao método `float getDelay(String signalName)` na instância de uma classe que implementa a interface *DelayModel*. A partir da célula que gera o sinal da saída primária considerada crítica, o método `void findCriticalPathGates(String output)` avalia os atrasos de chegada dos sinais de entrada e faz uma chamada recursiva para a porta que gera o sinal de entrada com o maior atraso. A recursão termina ao se alcançar uma porta onde todas as entradas são entradas primárias. No caso de empate entre os atrasos de sinais de entrada, considera-se crítica aquela entrada que esteja mais distante da saída (mais próxima dos nodos terminais do ROBDD) de acordo com a posição definida pelo critério de ordenamento. Após a chamada recursiva, a porta atual é adicionada em uma lista de portas como definido a seguir:

```
protected LinkedHashSet<String> criticalPathGates;
```

5.3.3 Geração da descrição SPICE de um caminho

Após determinar as portas que compõem um caminho potencialmente crítico, cada porta do caminho é processada para gerar a descrição SPICE. A classe *SwitchNetworkWrapper* apresentada na seção 5.2 é utilizada para gerar a descrição SPICE do sub-circuito (*.subckt*) da rede de transistores. Cada sub-circuito é instanciado, conectado e sensibilizado com um vetor de entrada que permita a passagem de um pulso aplicado à entrada considerada crítica. A Figura 5.5 apresenta as instanciações geradas automaticamente para um caminho do circuito *c2670* mapeado com ordenamentos topo-crítico.

```
x0 sigHigh  sigHigh  sigPulse  sigLow  sigLow  nsigPulse  out1  gate_x7376
x1 sigLow  sigLow  sigLow  out1  sigHigh  sigHigh  sigHigh  nout1  out2  gate_x13011
x2 sigHigh  out2  sigLow  sigHigh  sigLow  nout2  sigHigh  sigLow  out3  gate_x13156
x3 out3  sigHigh  nout3  sigLow  out4  gate_x7272
x4 out4  sigHigh  sigHigh  sigHigh  nout4  sigLow  sigLow  sigLow  out5  gate_x7253
x5 sigHigh  out5  sigHigh  sigLow  sigLow  nout5  sigLow  sigHigh  out6  gate_x7420
x6 out6 sigHigh  sigHigh  sigLow  nout6  sigLow  sigLow  sigHigh  out7  gate_xn_n492
x7 out7  sigHigh  sigLow  sigLow  nout7  sigLow  sigHigh  sigHigh  out8  gate_x7426
x8 out8  sigHigh  sigLow  sigHigh  nout8  sigLow  sigHigh  sigLow  out9  gate_x13208
x9 out9  sigHigh  sigHigh  sigLow  nout9  sigLow  sigLow  sigHigh  out10  gate_x7431
x10 out10  sigHigh  sigLow  nout10  sigLow  sigHigh  out11  gate_x13222
x11 out11 sigLow sigHigh sigLow  nout11  sigHigh  sigLow  sigHigh  out12  gate_x7452
x12 out12 sigLow sigHigh sigLow  nout12  sigHigh  sigLow  sigHigh  out13  gate_x7449
x13 out13 sigLow sigLow sigLow  nout13  sigHigh  sigHigh  sigHigh  out14  gate_x7446
x14 out14  sigLow  sigHigh  nout14  sigHigh  sigLow  out  gate_p_329_1414_
```

Figura 5.5: Exemplo de instanciações de células geradas automaticamente

O pulso de entrada (*sigPulse*) e os sinais lógicos (*sigLow* e *sigHigh*) são definidos em um rodapé (apresentado na Figura 5.6) que é adicionado a todas as descrições SPICE. A fonte com o pulso de entrada (*vpulse*) está definida para inclinação máxima e está conectada a um *buffer* com dois inversores (*xinvp1* e *xinvp2*) que tornam o sinal mais próximo de um comportamento elétrico real. Como as células lógicas das famílias avaliadas podem exigir ambas as polaridades de um sinal de entrada, conectou-se um terceiro inversor (*xinvpulse*) para fornecer o sinal negado do pulso de entrada. Por simplicidade, a ferramenta não avalia se o sinal é efetivamente consumido em ambas as polaridades e sempre insere um inversor para obter um sinal complementado. Por este motivo, podem aparecer inversores desnecessários aumentando a capacitância em um caminho extraído.

Além dos inversores adicionados para gerar os sinais complementares, foram adicionados inversores para simular o efeito do *fanout* em cada porta. O *fanout* da célula é avaliado e são conectados tantos inversores à saída quanto o número esperado de portas ligadas ao terminal de saída da célula. A mesma proporção de inversores é conectada tanto ao sinal direto como para o sinal complementado. A ferramenta avalia a polaridade da função gerada em cada célula e toma o devido cuidado para conectar na polaridade correta o terminal de entrada da próxima célula do caminho.

O dimensionamento dos transistores e as tensões de V_{dd} e *GND* são definidos em um cabeçalho que é adicionado a todas as descrições SPICE geradas, como apresentado na Figura 5.7.

```
* Input signal sources
vpulse sPulse gnd pulse(vdd gnd 0 .1n .1n 40n 80n)
vlowSignal sLow gnd pulse(vdd gnd 0 .1n .1n 400n 400n)
vhighSignal sHigh gnd pulse(gnd vdd 0 .1n .1n 400n 400n)

* Input signal buffers
xinvp1    sPulse    nsPulse    invsubckt
xinvp2    nsPulse    sigPulse    invsubckt
xinvl1    sLow    nsLow    invsubckt
xinvl2    nsLow    sigLow    invsubckt
xinvh1    sHigh    nsHigh    invsubckt
xinvh2    nsHigh    sigHigh    invsubckt

* Inverter to input pulse signal
xinvpulse    sigPulse    nsigPulse    invsubckt

* Measure for inverter resulting logic function
.MEASURE TRAN tdF TRIG V(sigPulse) VAL = 0.6 RISE=1 TD = 38n TARG V(out) VAL=0.6 FALL=1
.MEASURE TRAN tdR TRIG V(sigPulse) VAL = 0.6 FALL=1 TD = 38n TARG V(out) VAL=0.6 RISE=1

* Measure for non-inverter resulting logic function
.MEASURE TRAN tdF1 TRIG V(sigPulse) VAL = 0.6 FALL=1 TD = 38n TARG V(out) VAL=0.6 FALL=1
.MEASURE TRAN tdR1 TRIG V(sigPulse) VAL = 0.6 RISE=1 TD = 38n TARG V(out) VAL=0.6 RISE=1

* Options
.tran 8p 115n
.option post

.end
```

Figura 5.6: Rodapé adicionado às descrições SPICE

```
.include /home/gme/tmgcardoso/hspice/tech/ibm_013u.mod

* Parameters
.param vdd = 1.2
.param gnd = 0

.global gnd
.global vdd

vsrc vdd gnd vdd
.global gnd
.global vdd

.param wn = 1u
.param wp = 1.6u
.param lt = 0.13u
```

Figura 5.7: Cabeçalho adicionado às descrições SPICE

6 RESULTADOS

O ordenamento de variáveis tem influência na topologia e conseqüentemente no comportamento elétrico das redes de transistores derivadas de ROBDDs. Nos experimentos realizados são empregados diferentes critérios de ordenamento de variáveis no mapeamento de circuitos do conjunto de benchmarks ISCAS'85. A descrição SPICE para dez caminhos potencialmente críticos é extraída e sensibilizada automaticamente pela ferramenta de extração de caminhos (ver seção 5.3). Entre os dez caminhos, considera-se crítico aquele que tem o maior atraso obtido através da simulação SPICE.

Este capítulo apresenta os resultados dos experimentos realizados para avaliar o efeito do ordenamento de variáveis nas características de área e atraso em circuitos mapeados com células lógicas derivadas de ROBDDs. A seção 6.1 apresenta informações gerais sobre a organização dos resultados. A seção 6.2 apresenta os resultados obtidos através de simulações SPICE para o efeito do ordenamento de variáveis no atraso dos circuitos avaliados. A seção 6.3 apresenta os resultados de área obtidos para os circuitos avaliados. A seção 6.4 apresenta os resultados para o produto área x delay nos circuitos avaliados. A seção 6.5 apresenta considerações sobre os resultados obtidos.

6.1 Organização dos resultados

Em todos os experimentos, os circuitos foram mapeados para uma tecnologia IBM de 0,13 μm com tensão de alimentação de 1,2 volts. O dimensionamento dos transistores é fixado com o comprimento mínimo de 0,13 μm para transistores PMOS e NMOS. A largura utilizada para os transistores PMOS é de 1,6 μm enquanto a de transistores NMOS é de 1 μm. As simulações foram realizadas com a ferramenta HSPICE (Synopsys).

Foram avaliados dez circuitos do conjunto de benchmarks ISCAS'85 (ver Tabela 6.1). Para cada circuito são obtidos mapeamentos em seis famílias lógicas com células derivadas de ROBDDs. Os circuitos são mapeados utilizando quatro diferentes critérios para o ordenamento de variáveis. Para cada versão são extraídos dez caminhos potencialmente críticos para simulação transiente SPICE. A seguinte fórmula ilustra o total de simulações realizadas para os experimentos.

10 circuitos * 6 famílias * 4 ordenamentos * 10 caminhos = **2400 simulações**

Foi desenvolvido um conjunto de *scripts de shell* para automatizar a execução das simulações e também um *script Perl* para reunir os resultados de todas as simulações

em um único arquivo de texto. O arquivo de resultados alimenta um programa Java criado especificamente para encontrar o pior atraso entre os dez caminhos simulados e gerar as tabelas de resultados. Os resultados são impressos em formato separado por vírgulas, com uma tabela por família lógica onde cada linha é composta pelo nome do circuito de benchmark e pelos atrasos críticos obtidos para cada mapeamento desse circuito. Finalmente, os resultados são manipulados em uma planilha eletrônica para a montagem das tabelas finais. Os resultados finais são apresentados como a razão do atraso obtido em um dos mapeamentos do circuito pelo atraso obtido no circuito equivalente mapeado com ordenamentos para menor área. As três famílias com o sufixo PTL1 são aquelas derivadas diretamente a partir de ROBDDs e as outras três famílias com sufixo PTL2 correspondem às famílias com otimizações apresentadas nas seções 4.3 e 4.5.

Tabela 6.1: Características dos circuitos de benchmark ISCAS'85

Circuito	Função	Número de entradas	Número de saídas	Número de portas lógicas
c432	Controlador de interrupções com 27 canais	36	7	160
c499	Circuito SEC de 32 bits	41	32	202
c880	ALU de 8 bits	60	26	383
c1355	Circuito SEC de 32 bits	41	32	546
c1908	Circuito SEC / DED de 16 bits	33	25	880
c2670	ALU de 12 bits e controlador	233	140	1193
c3540	ALU de 8 bits	50	22	1669
c5315	ALU de 9 bits	178	123	2307
c6288	Multiplicador de 16 bits	32	32	2406
c7552	Somador / comparador de 32 bits	207	108	3512

Fonte: HANSEN, 1999

6.2 Resultados das simulações SPICE

Os resultados obtidos através das simulações SPICE indicam que o efeito do ordenamento de variáveis é significativo no atraso crítico de circuitos mapeados com as seis famílias lógicas avaliadas.

A Tabela 6.2 apresenta os resultados de circuitos mapeados com ordenamentos para topo crítico. Os ordenamentos para topo crítico levaram aos melhores resultados de atraso, sendo 16,4% em média menores do que o atraso do circuito equivalente com ordenamentos para menor área. Os únicos resultados negativos aparecem nas famílias NPTL (com e sem otimização) para os circuitos c880 e c2670. Curiosamente, o circuito c2670 apresentou a maior redução de atraso em comparação ao mapeamento para menor área, sendo 36,3% menor nas famílias CPTL1 e DCPTL1.

A Tabela 6.3 apresenta os resultados de circuitos mapeados com ordenamentos para base crítica. Em contraste com os ordenamentos para topo crítico, estes ordenamentos levaram aos piores resultados de atraso, sendo 42,1% em média maiores do que o atraso do circuito equivalente com ordenamentos para menor área. O único resultado positivo foi o do circuito c7552 na família NPTL2.

Tabela 6.2: Efeito no atraso com ordenamentos para topo crítico

circuito	NPTL1	CPTL1	DCPTL1	NPTL2	CPTL2	DCPTL2	*média*
c432	0,885	0,896	0,896	0,914	0,928	0,954	*0,912*
c499	0,945	0,800	0,800	0,923	0,694	0,784	*0,824*
c880	1,065	0,897	0,897	1,033	0,894	0,964	*0,959*
c1355	0,912	0,803	0,803	0,929	0,723	0,841	*0,835*
c1908	0,847	0,691	0,691	0,846	0,674	0,884	*0,772*
c2670	1,015	0,637	0,637	1,028	0,667	0,755	*0,790*
c3540	0,917	0,829	0,829	0,788	0,835	0,892	*0,848*
c5315	0,904	0,800	0,800	0,935	0,775	0,883	*0,850*
c6288	0,889	0,725	0,725	0,902	0,743	0,851	*0,806*
c7552	0,788	0,678	0,678	0,907	0,641	0,905	*0,766*
média	*0,917*	*0,775*	*0,775*	*0,920*	*0,757*	*0,871*	***0,836***

Tabela 6.3: Efeito no atraso com ordenamentos para base crítica

circuito	NPTL1	CPTL1	DCPTL1	NPTL2	CPTL2	DCPTL2	*média*
c432	1,383	1,733	1,733	1,200	1,597	1,178	*1,471*
c499	1,368	1,516	1,516	1,312	1,580	1,407	*1,450*
c880	1,307	1,659	1,659	1,123	1,580	1,332	*1,443*
c1355	1,173	1,114	1,114	1,119	1,185	1,132	*1,139*
c1908	1,450	1,785	1,785	1,268	1,979	1,389	*1,609*
c2670	1,411	1,591	1,591	1,222	1,603	1,181	*1,433*
c3540	1,290	1,751	1,751	1,015	1,991	1,431	*1,538*
c5315	1,199	1,516	1,516	1,142	1,513	1,059	*1,324*
c6288	1,209	1,923	1,923	1,073	1,785	1,457	*1,562*
c7552	1,146	1,401	1,401	0,939	1,284	1,260	*1,238*
média	*1,294*	*1,599*	*1,599*	*1,141*	*1,610*	*1,283*	***1,421***

Tabela 6.4: Efeito no atraso com ordenamentos para menor área e topo crítico no caminho mais longo

circuito	NPTL1	CPTL1	DCPTL1	NPTL2	CPTL2	DCPTL2	*média*
c432	0,891	0,896	0,896	0,914	0,928	0,959	*0,914*
c499	1,001	1,000	1,000	1,002	1,012	1,008	*1,004*
c880	1,065	0,897	0,897	1,033	0,912	0,976	*0,963*
c1355	0,951	0,891	0,891	0,976	0,950	0,969	*0,938*
c1908	0,847	0,758	0,758	0,846	0,720	0,884	*0,802*
c2670	1,029	0,718	0,718	1,055	0,730	0,890	*0,857*
c3540	0,961	0,950	0,950	0,810	0,898	0,966	*0,923*
c5315	0,904	0,945	0,945	0,935	0,994	0,883	*0,934*
c6288	0,897	1,005	1,005	0,913	1,004	0,983	*0,968*
c7552	0,767	0,678	0,678	0,861	0,641	0,905	*0,755*
média	*0,931*	*0,874*	*0,874*	*0,934*	*0,879*	*0,942*	***0,906***

A Tabela 6.4 apresenta os resultados de circuitos mapeados com ordenamentos onde apenas as células que pertencem ao caminho mais longo são ordenadas para topo crítico, enquanto as demais células são ordenadas para menor área. Este critério de ordenamento levou a bons resultados de atraso, sendo 9,4% em média menores do que o atraso do circuito equivalente com ordenamentos para menor área.

6.3 Resultados de área

Além dos resultados de desempenho obtido através das simulações SPICE, os resultados de área foram obtidos a partir da contagem total de transistores em cada célula do circuito mapeado.

Tabela 6.5: Efeito na área com ordenamentos para topo crítico

circuito	NPTL1	CPTL1	DCPTL1	NPTL2	CPTL2	DCPTL2	*média*
c432	1,107	1,145	1,145	1,063	1,128	1,106	*1,116*
c499	1,050	1,044	1,044	1,069	1,052	1,032	*1,048*
c880	1,109	1,122	1,122	1,063	1,098	1,113	*1,104*
c1355	1,062	1,057	1,057	1,092	1,071	1,046	*1,064*
c1908	1,111	1,107	1,107	1,108	1,104	1,081	*1,103*
c2670	1,095	1,112	1,112	1,154	1,154	1,100	*1,121*
c3540	1,123	1,138	1,138	1,105	1,131	1,115	*1,125*
c5315	1,162	1,183	1,183	1,220	1,226	1,120	*1,182*
c6288	1,116	1,086	1,086	1,135	1,090	1,026	*1,090*
c7552	1,175	1,178	1,178	1,215	1,202	1,127	*1,179*
média	*1,111*	*1,117*	*1,117*	*1,123*	*1,125*	*1,087*	***1,113***

A Tabela 6.5 mostra que os ordenamentos para topo crítico, quando aplicados em todas as células do circuito, levam a um aumento de 11,3% em média na quantidade de transistores, comparando-se ao circuito mapeado com ordenamentos para menor área. A família DCPTL2 apresentou o menor aumento em área, de apenas 8,7% em média.

Tabela 6.6: Efeito na área com ordenamentos para base crítica

circuito	NPTL1	CPTL1	DCPTL1	NPTL2	CPTL2	DCPTL2	*média*
c432	1,116	1,160	1,160	1,070	1,142	1,110	*1,126*
c499	1,212	1,184	1,184	1,354	1,250	1,165	*1,225*
c880	1,111	1,140	1,140	1,070	1,123	1,115	*1,117*
c1355	1,258	1,223	1,223	1,382	1,279	1,225	*1,265*
c1908	1,139	1,142	1,142	1,162	1,156	1,085	*1,138*
c2670	1,088	1,107	1,107	1,116	1,131	1,087	*1,106*
c3540	1,147	1,198	1,198	1,134	1,203	1,141	*1,170*
c5315	1,090	1,121	1,121	1,132	1,158	1,089	*1,118*
c6288	1,040	1,068	1,068	1,021	1,065	1,022	*1,047*
c7552	1,192	1,228	1,228	1,309	1,307	1,168	*1,239*
média	*1,139*	*1,157*	*1,157*	*1,175*	*1,181*	*1,121*	***1,155***

A Tabela 6.6 mostra que os ordenamentos para base crítica, quando aplicados em todas as células do circuito, levam a um aumento de 15,5% em média na quantidade de transistores. A família CPTL2 apresentou o maior aumento em área, de 18,1% em média.

Tabela 6.7: Efeito na área com ordenamentos para menor área e topo crítico no caminho mais longo

circuito	NPTL1	CPTL1	DCPTL1	NPTL2	CPTL2	DCPTL2	*média*
c432	1,022	1,026	1,026	1,006	1,017	1,026	*1,021*
c499	1,002	1,001	1,001	1,003	1,002	1,000	*1,002*
c880	1,002	1,003	1,003	1,000	1,002	1,002	*1,002*
c1355	1,013	1,012	1,012	1,016	1,014	1,010	*1,013*
c1908	1,013	1,014	1,014	1,010	1,012	1,011	*1,012*
c2670	1,007	1,007	1,007	1,008	1,008	1,003	*1,007*
c3540	1,007	1,007	1,007	1,006	1,006	1,006	*1,006*
c5315	1,006	1,007	1,007	1,009	1,009	1,005	*1,007*
c6288	1,006	1,006	1,006	1,001	1,003	1,003	*1,004*
c7552	1,002	1,003	1,003	1,003	1,005	1,001	*1,003*
média	*1,008*	*1,009*	*1,009*	*1,006*	*1,008*	*1,007*	***1,008***

A Tabela 6.7 mostra que os ordenamentos para menor área e topo crítico no caminho mais longo levam a um aumento de 0,8% em média na quantidade de transistores. Entre os critérios de ordenamento avaliados, este apresenta o menor aumento de área em relação ao mapeamento para menor área.

6.4 Produto área x atraso

O produto área x atraso pode ser usado para avaliar simultaneamente as medidas de área e atraso do circuito mapeado. Quanto menor o produto, maior a qualidade do mapeamento.

Tabela 6.8: Produto área x atraso com ordenamentos para topo crítico

circuito	NPTL1	CPTL1	DCPTL1	NPTL2	CPTL2	DCPTL2	*média*
c432	0,980	1,026	1,026	0,971	1,046	1,055	*1,017*
c499	0,992	0,835	0,835	0,987	0,730	0,809	*0,864*
c880	1,181	1,006	1,006	1,099	0,982	1,073	*1,058*
c1355	0,968	0,848	0,848	1,015	0,775	0,880	*0,889*
c1908	0,941	0,765	0,765	0,938	0,743	0,956	*0,851*
c2670	1,111	0,708	0,708	1,186	0,770	0,831	*0,886*
c3540	1,030	0,943	0,943	0,870	0,944	0,995	*0,954*
c5315	1,051	0,947	0,947	1,141	0,950	0,989	*1,004*
c6288	0,993	0,787	0,787	1,024	0,810	0,873	*0,879*
c7552	0,926	0,799	0,799	1,103	0,771	1,020	*0,903*
média	*1,017*	*0,866*	*0,866*	*1,033*	*0,852*	*0,948*	***0,931***

A Tabela 6.8 mostra que os ordenamentos para topo crítico, quando aplicados em todas as células do circuito, levam a uma redução de 16,9% em média no produto área x atraso, comparando-se ao circuito mapeado com ordenamentos para menor área. Os circuitos c432, c880 e c5315 apresentaram resultados negativos em média, assim como as famílias NPTL1 e NPTL2.

Tabela 6.9: Produto área x atraso com ordenamentos para base crítica

circuito	NPTL1	CPTL1	DCPTL1	NPTL2	CPTL2	DCPTL2	*média*
c432	1,542	2,010	2,010	1,283	1,825	1,308	*1,663*
c499	1,659	1,796	1,796	1,777	1,976	1,640	*1,774*
c880	1,452	1,891	1,891	1,202	1,775	1,485	*1,616*
c1355	1,476	1,363	1,363	1,546	1,515	1,386	*1,441*
c1908	1,653	2,039	2,039	1,474	2,288	1,508	*1,833*
c2670	1,535	1,762	1,762	1,364	1,813	1,284	*1,587*
c3540	1,481	2,097	2,097	1,151	2,396	1,632	*1,809*
c5315	1,307	1,699	1,699	1,293	1,751	1,153	*1,484*
c6288	1,257	2,054	2,054	1,096	1,900	1,490	*1,642*
c7552	1,365	1,719	1,719	1,230	1,679	1,472	*1,531*
média	*1,473*	*1,843*	*1,843*	*1,342*	*1,892*	*1,436*	***1,638***

A Tabela 6.9 mostra que os ordenamentos para base crítica, quando aplicados em todas as células do circuito, levam a um aumento de 63,8% em média no produto área x atraso, comparando-se ao circuito mapeado com ordenamentos para menor área.

Tabela 6.10: Produto área x atraso com ordenamentos para menor área e topo crítico no caminho mais longo

circuito	NPTL1	CPTL1	DCPTL1	NPTL2	CPTL2	DCPTL2	*média*
c432	0,911	0,919	0,919	0,919	0,944	0,985	*0,933*
c499	1,003	1,001	1,001	1,005	1,014	1,008	*1,005*
c880	1,068	0,900	0,900	1,033	0,914	0,978	*0,965*
c1355	0,963	0,902	0,902	0,992	0,963	0,979	*0,950*
c1908	0,858	0,768	0,768	0,855	0,729	0,894	*0,812*
c2670	1,037	0,723	0,723	1,063	0,735	0,892	*0,862*
c3540	0,967	0,957	0,957	0,815	0,903	0,971	*0,928*
c5315	0,910	0,951	0,951	0,944	1,003	0,887	*0,941*
c6288	0,902	1,011	1,011	0,914	1,007	0,987	*0,972*
c7552	0,768	0,681	0,681	0,864	0,644	0,906	*0,757*
média	*0,939*	*0,881*	*0,881*	*0,940*	*0,886*	*0,949*	***0,913***

A Tabela 6.10 mostra que os ordenamentos para menor área e topo crítico no caminho mais longo levam a uma redução de 8,7% em média no produto área x atraso, comparando-se ao circuito mapeado com ordenamentos para menor área.

6.5 Considerações sobre os resultados

Os resultados obtidos com os experimentos demonstram o efeito do ordenamento de variáveis na área e no atraso dos circuitos mapeados para as seis famílias lógicas avaliadas. Apesar dos caminhos potencialmente críticos simulados poderem representar falsos caminhos, ou seja, caminhos que não podem ser sensibilizados com o uso real do circuito, é esperado um efeito semelhante no verdadeiro caminho crítico do circuito. Assim, espera-se que o reordenamento de variáveis proporcione ganhos efetivos de atraso em relação a ordenamentos aleatórios de variáveis ou ordenamentos para menor área. Ao ordenar somente as células que pertencem ao caminho crítico para melhor atraso e ordenar para menor área as demais células, pode-se obter circuitos mais rápidos sem aumentar significativamente a área do circuito mapeado. A dificuldade é a determinação do caminho crítico a ser otimizado e qual o ordenamento que leva ao menor atraso (topo crítico ou base crítica) em cada célula deste caminho.

Outro aspecto a ser observado é que os experimentos realizados para esta dissertação não consideram as capacitâncias e resistências associadas ao roteamento interno e externo à célula. O comprimento dos fios condutores possui forte influência no atraso de um circuito, sendo cada vez mais importante devido à constante miniaturização dos dispositivos. Portanto, o efeito geral do reordenamento de transitores pode ser menos significativo no atraso de um circuito ao considerar o atraso adicionado pela condução dos sinais através de metal ou polisilício. A disponibilidade de um gerador de leiaute para as células permitiria a extração das capacitâncias e resistências associadas ao roteamento no interior da célula. Adicionalmente, uma ferramenta de posicionamento e roteamento poderia ser usada para sintetizar o circuito, permitindo extrair estes parâmetros no roteamento global.

7 CONCLUSÃO

O ordenamento de variáveis em ROBDDs determina a topologia das redes de transistores derivadas a partir destas estruturas. O posicionamento relativo dos transistores é determinado diretamente pelo ordenamento de variáveis. Com auxílio de simulações SPICE, avaliou-se o efeito do ordenamento de variáveis sobre o atraso crítico de circuitos mapeados em seis famílias lógicas diferentes que empregam células com redes de transistores derivadas de ROBDDs. O critério de ordenamento de variáveis explorado neste trabalho que em geral levou ao mapeamento de circuitos mais rápidos foi o de posicionar os transistores controlados pelos sinais com maior atraso de chegada mais próximos relativamente à saída da célula.

Nos experimentos realizados, a descrição lógica para diferentes circuitos foi inicialmente decomposta em funções booleanas de até quatro variáveis. Em seguida, é construído automaticamente um modelo de atrasos onde cada função da rede booleana corresponde ao atraso de uma unidade de tempo. As entradas primárias possuem tempos de chegada iguais a zero e os tempos de chegada para outros sinais correspondem ao número de funções booleanas no caminho mais longo em direção a uma entrada primária. Este modelo de atrasos (modelo de atrasos unitários) foi utilizado para determinar o ordenamento de variáveis em cada célula dos circuitos.

Foram utilizados dois mapeamentos para cada circuito em uma tecnologia de 130 nanômetros. Na primeira versão, os transistores controlados pelos sinais com maiores tempos de chegada são posicionados mais próximos às saídas das células. Na segunda versão, o ordenamento utilizado em cada célula é o inverso ao utilizado na primeira versão. Para avaliar o atraso, foram extraídos dez caminhos potencialmente críticos em cada versão do circuito.

Cada um dos dez caminhos potencialmente críticos é determinado em função de um modelo de atrasos diferente. Os modelos de atrasos consideram diferentes características que influenciam o atraso de uma célula, tais como número de transistores em série e fanout. A descrição SPICE pronta para a simulação transiente de cada caminho potencialmente crítico é gerada automaticamente.

Foram realizadas com sucesso duas mil e quatrocentas simulações SPICE para avaliar o efeito do ordenamento de variáveis sobre o atraso de dez circuitos em seis famílias lógicas. O caminho potencialmente crítico com o maior atraso entre os dez caminhos simulados é considerado o caminho crítico daquela versão do circuito. Os resultados dos experimentos indicam que o ordenamento de variáveis para topo crítico leva a um ganho de 16,4% em média no atraso, comparando-se com ordenamentos para menor área.

Como sugestão para trabalhos futuros, considera-se a utilização de um modelo de atrasos mais preciso. Este modelo poderia ser usado para determinar os sinais mais críticos com maior exatidão do que o modelo de atrasos unitários. Além disso, este modelo de atrasos pode ser usado para testar as duas versões (*top-critical* ou *bottom-critical*) de uma célula. Avaliando-se individualmente cada célula, em vez de aplicar um único critério para todas células, provavelmente levará a um efeito mais expressivo com o reordenamento do que o obtido nos experimentos realizados.

Uma outra questão a ser tratada em trabalhos futuros é o dimensionamento dos transistores. Nos experimentos realizados, todos os transistores de um mesmo tipo (PMOS ou NMOS) possuem o mesmo tamanho. O dimensionamento de transistores possui forte impacto no desempenho dos circuitos. Entretanto, acredita-se que o ordenamento de variáveis pode levar a circuitos mais rápidos e com menor área do que circuitos otimizados unicamente através do dimensionamento de transistores. Esta questão pode ser comprovada através de experimentos que não foram realizados neste trabalho.

REFERÊNCIAS

BOLLIG, B.; WEGENER, I. Improving the Variable Ordering of OBDDs is NP-Complete. **IEEE Transactions on Computers**, Washington, DC, USA: IEEE Computer Society, v. 45, n. 9, p. 993-1002, September, 1996.

BRACE, K.; RUDELL, R.; BRYANT, R. Efficient Implementation of a BDD Package. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, DAC, 1990. **Proceedings**… New York, NY, USA: ACM, 1990. p. 40-45.

BRYANT, R. Graph-Based Algorithms for Boolean Function Manipulation. **IEEE Transactions on Computers**, Washington, DC, USA: IEEE Computer Society, v. 35, n. 8, p. 677-691, August, 1986.

BUCH, P. et al. Logic synthesis for large pass transistor circuits. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, ICCAD, 1997. **Proceedings**… Washington, DC, USA: IEEE Computer Society, 1997. p. 663-670.

CARLSON, B.; CHEN, C. Effects of transistor reordering on the performance of MOS digital circuits. In: MIDWEST SYMPOSIUM ON CIRCUITS AND SYSTEMS, 1992. **Proceedings**… [S.l.: s.n], August, 1992. v. 1, p. 121-124.

CARLSON, B.; SUH-JUCH, L. Delay optimization of digital CMOS VLSI circuits by transistor reordering. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.: s.n], v. 14, n. 10, p. 1183-1192, October, 1995.

CHAUDHRY R. et al. Area-oriented synthesis for pass transistor logic. In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN, ICCD, 1998. **Proceedings**... Washington, DC: IEEE Computer Society, p. 160-167.

CORREIA, V. **Mapeamento Tecnológico para Bibliotecas Virtuais Simétricas e Assimétricas com Minimização da Profundidade Lógica do Circuito**. 2005. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

DRECHSLER, R.; BECKER, B. **Binary Decision Diagrams:** Theory and Implementation. [S.l.]: Kluwer Academic Publishers, 1998.

EBENDT, R.; GUNTHER, W.; DRECHSLER, R. Combining Ordered Best-First Search with Branch and Bound for Exact BDD Minimization. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2004. **Proceedings**… Piscataway, NJ, USA: IEEE Press, 2004. p. 875-878.

FLORES, E.; REIS A.; CARDOSO, T. Non-disjoint functional decomposition. In: SOUTH SYMPOSIUM ON MICROELECTRONICS, SIM, 2006. **Proceedings**... Porto Alegre: SBC, 2006, p. 219-223.

GOMES, M. V. N. et al. BDDeiro: BDD Visualization Tool. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 2006, Ouro Preto, MG. **Student Forum**... [S.l.]: SBC, 2006.

HANSEN, M. C.; YALCIN, H.; HAYES, J. P. Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering. **IEEE Design and Test of Computers**, Los Alamitos, CA, USA: IEEE Computer Society Press, v. 16, n. 3, p. 72-80, September, 1999.

HSIAO S. F.; YEH J. S.; CHEN D. Y. High-performance multiplexer-based logic synthesis using pass-transistor logic. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 2000. **Proceedings**... [S.l.: s.n], 2000, v. 2, p. 325-328.

JANSSEN, G. Design of a Pointerless BDD Package. In: INTERNATIONAL WORKSHOP ON LOGIC SYNTHESIS, IWLS, 2001, Lake Tahoe, CA, USA. **Proceedings**… [S.l.: s.n], 2001.

JANSSEN, G. A Consumer Report on BDD Packages. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 2003. **Proceedings**... Los Alamitos, CA, USA: IEEE Computer Society Press, p. 217-222.

KEUTZER, K.; GIRCZYC, E. Cell libraries - build vs. buy; static vs. dynamic. In: DESIGN AUTOMATION CONFERENCE, DAC, 1999. **Proceedings**... [S.l.: s.n], 1999, p. 341-342.

LEFEBVRE, M.; MARPLE, D.; SECHEN, C. The future of custom cell generation in physical synthesis. In: DESIGN AUTOMATION CONFERENCE, DAC, 1997. **Proceedings**... New York, NY, USA: ACM, 1997, p. 446-451.

LIU T.-H. et al. Performance driven synthesis for pass transistor logic. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, 1999. **Proceedings**... Washington, DC: IEEE Computer Society, p. 372-377.

MARQUES, F. S. et al. Library-Less Technology Mapping based on DAGs. In: SOUTH SYMPOSIUM ON MICROELECTRONICS, SIM, 2006. **Proceedings**... Porto Alegre: SBC, 2006, p. 209-213.

MINATO, S. **Binary Decision Diagrams and Applications for VLSI CAD**. [S.l.]: Kluwer Academic Publishers, 1996.

POLI, R. E. B.; RIBAS R. P.; REIS A. I. Unified Theory to Build Cell-Level Transistor Networks from BDDs. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, SBCCI, 2003. **Proceedings**… Washington, DC, USA: IEEE Computer Society, 2003. p. 199–204.

ROSA, L. S. et al. Fast Disjoint Transistor Networks from BDDs. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, SBCCI, 19., 2006, Ouro Preto, Brasil. **Proceedings**… New York: ACM, 2006. p. 137-142

ROY, R.; BHATTACHARYA, D.; BOPPANA, V. Transistor-level optimization of digital designs with flex cells. **Computers**, Los Alamitos, CA, USA: IEEE Computer Society Press, v. 38, n. 2, p. 53-61, February, 2005.

RUDELL, R. L. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, ICCAD, 1993. **Proceedings**… Los Alamitos, CA, USA: IEEE Computer Society Press, 1993. p. 42-47.

SCHNEIDER, F. et al. Exact lower bound for the number of switches in series to implement a combinational logic cell. In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN, ICCD, 2005. **Proceedings**… Washington, DC, USA: IEEE Computer Society. p. 357-362.

SCHOLL C.; BECKER B. On the generation of multiplexer circuits for pass transistor logic. In: DESIGN AUTOMATION AND TEST IN EUROPE, DATE, 2000. **Proceedings**… New York, NY, USA: ACM. p. 372-379.

SECHEN, C. Libraries: lifejacket or straitjacket. In: DESIGN AUTOMATION CONFERENCE, DAC, 2003. **Proceedings**... New York: ACM, 2003. p. 642-643.

SENTOVICH, E. et al. **SIS**: A system for sequential circuit synthesis. Berkeley: EECS Department, University of California, 1992. (Technical Report n. UCB/ERL M92/41).

SHELAR, R. S.; SAPATNEKAR, S. S. BDD decomposition for delay oriented pass transistor logic synthesis. **IEEE Transactions on Very Large Scale Integration Systems**, [S.l: s.n.], v. 13, n. 8, p. 957-970, August, 2005.

WESTE, N.; HARRIS, D. **CMOS VLSI Design**: A Circuits and Systems Perspective. 3rd ed. Boston: Pearson/Addison Wesley, 2005.

YANO, K. et al. Top-down pass-transistor logic design. **IEEE Journal of Solid-State Circuits**, [S.l: s.n.], v. 31, n. 6, p. 792-803, June, 1996.

ANEXO A < Métodos da classe *BDDManager* >

---

addFromExpression

```
public int addFromExpression(java.lang.String anExpression)
```

Find or add a boolean function given a logic expression.

Parameters:

`anExpression` - a logic expression. Example: a*(b*!c+b*d).

Returns:

the root node representing the logic expression.

---

addFromTruthTable

```
public int addFromTruthTable(java.lang.String truthValues)
```

Find or add a boolean function given a truth table String representation in binary or hexadecimal format. The support variable's names are automatically created with a [Xn,...,Xm] pattern, where n is the n-th variable created at the specific manager's context, starting at zero, while m is n+(number of variables in truth-table).

Parameters:

`truthValues` - a truth table with same variable ordering as the current variable ordering for the ROBDD. The length of the binary String need to be a power of 2 with exponent equals to the size of variable support. The representation could be in hexadecimal format using the prefix *0x*.

Returns:

the indice of the node representing the given boolean function truth table.

---

addFromTruthTable

```
public int addFromTruthTable(java.lang.String truthValues,
                             java.util.Collection<java.lang.Integer> aSupport)
```

Find or add a boolean function given a truth table String representation in binary or hexadecimal format.

Parameters:

`truthValues` - a truth table with same variable ordering as the current variable ordering for the ROBDD. The length of the binary String need to be a power of 2 with exponent equals to the size of variable support. The representation could be in hexadecimal format using the prefix *0x*.

`aSupport` - the variables in the support set of given boolean function.

Returns:

the indice of the node representing the given boolean function truth table.

---

addReference

```
public void addReference(int aRoot)
```

Use this method to mark a boolean function root as a fuction of interest.

Parameters:

aRoot -

---

and

```
public int and(int aNode1,
               int aNode2)
```

Compute `and` operator for two boolean functions.

Parameters:

`aNode1` -

`aNode2` -

Returns:

the resulting boolean function root node.

---

balance

```
public java.math.BigInteger balance(int aRootNode,
                                    int aVariableIndex)
```

---

booleanDifference

```
public int booleanDifference(int aRootNode,
                             int aVariableIndex)
```

---

createVariable

```
public int createVariable(java.lang.String aName)
```

Creates a boolean variable at the bottom level (biggest level position).

Parameters:

`aName` - a mandatory unique name to the new variable.

Returns:

the indice of the node labeled with the variable name and with THEN and ELSE edges pointing to terminals ONE and ZERO respectively.

---

density

```
public java.math.BigDecimal density(int aRootNode)
```

---

density

```
public java.math.BigDecimal density(int aRootNode,
                                    int aVariableIndex)
```

---

explode

```
public static void explode()
```

---

findOrAdd

```
public int findOrAdd(int aVariable,
                     int aThen,
                     int anElse)
```

Searches the unique table for a node with top level variable `aVariable`, with the 1-edge pointing to `aThen` and the 0-edge pointing to `anElse`. If such a node isn't found then the node is created.

Parameters:

`aVariable` - the top level variable.

`aThen` - the edge-1 successor.

`anElse` - the edge-0 successor.

Returns:

the found or added node indice

---

getAnyMinterm

```
public int getAnyMinterm(int aRoot,
                         java.util.List<java.lang.Integer> aSupport)
```

---

getCubeRootFromLongestPath

```
public int getCubeRootFromLongestPath(int aRoot,
                                      boolean toTerminalOne)
```

Find the root node for a cube represented by the longest path to a terminal node (SAT-ONE-LONG)

Parameters:

`aRoot` -

`toTerminalOne` -

Returns:

the root node for the cube represented by a maximum length path

---

getCubeRootFromPath

```
public int getCubeRootFromPath(java.util.LinkedList<java.lang.Integer> aPath,
                               boolean toTerminalOne)
```

Find the root node for a cube represented by a path to a terminal node.

Parameters:

`aPath` - a linked-list with nodes from a path.

`toTerminalOne` - searching paths to one (zero) when receives `true` (`false`).

Returns:

the root node for the cube represented by a path

---

getCutParents

```
public java.util.HashSet<java.lang.Integer> getCutParents(int aRoot,
                                                          int aLevel)
```

---

getDAG

```
public java.util.ArrayList<java.lang.Integer> getDAG(int aRoot)
```

Get the indices of nodes reachable from a given root node.

Parameters:

`aRoot` - the indice of a root node.

Returns:

a Vector with the indices of nodes reachable from the root node.

---

getLongestPath

```
public java.util.LinkedList<java.lang.Integer> getLongestPath(int aRoot,

boolean toTerminalOne)
```

Parameters:

`aRoot` - the root node from the wanted path.

`toTerminalOne` - find path to terminal one (true) or zero (false).

Returns:

a list with nodes from the longest path to a terminal node.

---

getLowerBoundOffSet

```
public int getLowerBoundOffSet(int aRoot)
```

---

getLowerBoundOffSet

```
public int getLowerBoundOffSet(int aRoot,
                               int maxLowerBound)
```

---

getLowerBoundOnSet

```
public int getLowerBoundOnSet(int aRoot)
```

---

getLowerBoundOnSet

```
public int getLowerBoundOnSet(int aRoot,
                              int maxLowerBound)
```

---

getNextVariableIndex

```
public int getNextVariableIndex()
```

Get the variable index just before calling `createVariable(String)`.

Remember that every created variable has an index, i.e. an unique integer reflecting variable creation order, starting at *0* (zero). Also, a variable index is equal to the initial position (level) at variable ordering. After a ROBDD variable reordering, variable indexes and levels may differ from each other.

Returns:

a variable index.

---

getNodeElse

`public int` **`getNodeElse`**`(int aNodeIndice)`

Get the indice of a node pointed by the ELSE edge from a given node.

Parameters:

`aNodeIndice` - the indice from the node in which the ELSE is needed.

Returns:

the indice from a node pointed by the ELSE edge.

---

getNodeLevel

`public int` **`getNodeLevel`**`(int aNodeIndice)`

Get the position of a node label from the actual variable ordering.

Parameters:

`aNodeIndice` - a node indice.

Returns:

a node level.

---

getNodeThen

`public int` **`getNodeThen`**`(int aNodeIndice)`

Get the indice of a node pointed by the THEN edge from a given node.

Parameters:

`aNodeIndice` - the indice from the node in which the THEN is needed.

Returns:

the indice from a node pointed by the THEN edge.

---

getNodeVariable

`public int` **`getNodeVariable`**`(int aNodeIndice)`

Get the variable index of a node label.

Parameters:

`aNodeIndice` - a node indice.

Returns:

a variable index.

---

getPaths

```
public java.util.LinkedList<java.util.LinkedList<java.lang.Integer>>
getPaths(int aRoot,

boolean toTerminalOne)
```

Return a linked-list with all paths from root to choosen terminal node. (SAT-ALL)

Parameters:

`aRoot` - a root node.

`toTerminalOne` - searching paths to one (zero) when receives `true` (`false`).

Returns:

a linked-list with linked-lists of nodes of paths from root to choosen terminal node.

---

getProjectionsFromCube

```
public java.util.LinkedList<java.lang.Integer>
getProjectionsFromCube(int cube)
```

---

getProjectionsFromPath

```
public java.util.LinkedList<java.lang.Integer>
getProjectionsFromPath(java.util.LinkedList<java.lang.Integer> aPath,

boolean toTerminalOne)
```

---

getShortestPath

```
public java.util.LinkedList<java.lang.Integer> getShortestPath(int aRoot,

boolean toTerminalOne)
```

Parameters:

`aRoot` - the root node from the wanted path.

`toTerminalOne` - find path to terminal one (true) or zero (false).

Returns:

a list with nodes from the shortest path to a terminal node.

---

getSupport

```
public java.util.ArrayList<java.lang.Integer> getSupport(int aRootIndice)
```

Get the indexes of variables in the support set of a boolean function represented by a node. Variable ordering in this list reflects variable ordering at the ROBDD.

Parameters:

`aRootIndice` - the indice of a root node.

Returns:

a vector with the ordered support of a DAG.

---

getUnateness

`public int getUnateness(int aNodeIndice)`

Get unateness property of a boolean function.

Parameters:

`aNodeIndice` -

Returns:

the resulting boolean function root node

---

getVarCount

`public int getVarCount()`

refer to `getNextVariableIndex()`.

Returns:

a variable index.

---

getVariableIndex

`public int getVariableIndex(int aLevel)`

Get the variable index given a variable level.

Parameters:

`aLevel` - a variable level.

Returns:

a variable index.

---

getVariableIndex

`public int getVariableIndex(java.lang.String aName)`

Get the variable index given a variable name. Complexity: O(n) where "n" is number of variables. Variable ordering is insertion-order.

Parameters:

`aName` - the name of a variable.

Returns:

the index of a variable.

---

getVariableLevel

`public int getVariableLevel(int aVariableIndex)`

Get the variable level given a variable index.

Parameters:

`aVariableIndex` - a variable level.

Returns:

a variable index.

---

getVariableName

```
public java.lang.String getVariableName(int aVariableIndex)
```

Get the variable name given a variable index.

Parameters:

`aVariableIndex` - a variable index.

Returns:

a variable name.

---

getVariableNegativeProjection

```
public int getVariableNegativeProjection(int aVariableIndex)
```

---

getVariablePositiveProjection

```
public int getVariablePositiveProjection(int aVariableIndex)
```

---

getVarProjection

```
public int getVarProjection(int aVariableIndex,
                            boolean positive)
```

---

hex2bin

```
public static java.lang.String hex2bin(java.lang.String s)
```

---

inv

```
public int inv(int aNode)
```

Compute the complement of a boolean function represented by a node.

Parameters:

`aNode` -

Returns:

the indice of the root node for complemented boolean function.

---

isop

```
public java.lang.String isop(int aRootNode)
```

Get a Minato-Morreale irredundant sum-of-products for a given root node.

Parameters:

`aRootNode` - a root node.

Returns:

an ISOP expression.

---

isPositiveProjection

```
public boolean isPositiveProjection(int aRoot)
```

---

isProjection

```
public boolean isProjection(int aRoot)
```

---

isTerminal

```
public boolean isTerminal(int aNode)
```

Test if a node is a terminal node.

Parameters:

`aNode` -

Returns:

true if aNode is terminal, else *false.*

---

isTerminalOne

```
public boolean isTerminalOne(int aNode)
```

Test if a node is the 1-terminal node.

Parameters:

`aNode` -

Returns:

true if aNode is the 1-terminal, else *false.*

---

isTerminalZero

```
public boolean isTerminalZero(int aNode)
```

Test if a node is the 0-terminal node.

Parameters:

`aNode` -

Returns:

true if aNode is the 0-terminal, else *false.*

---

isValidNode

```
public boolean isValidNode(int aNode)
```

---

ite

```
public int ite(int anIf,
               int aThen,
               int anElse)
```

This is the core method to boolean function synthesis. All up to 2-input boolean function may be efficiently implemented with ITE.

Parameters:

`anIf` - the *if* or condition node indice.

`aThen` - the *then* node indice.

`anElse` - the *else* node indice.

Returns:

the indice of the node with the function F = anIf ^ aThen + !anIf ^ anElse.

---

nand

```
public int nand(int aNode1,
                int aNode2)
```

Compute `nand` operator for two boolean functions.

Parameters:

`aNode1` -

`aNode2` -

Returns:

the resulting boolean function root node.

---

negativeCofactor

```
public int negativeCofactor(int aRootNode,
                            int aVariableIndex)
```

Compute the negative cofactor of a boolean function with respect to a boolean variable.

Parameters:

`aRootNode` -

`aVariableIndex` -

Returns:

the resulting boolean function root node

---

nor

```
public int nor(int aNode1,
               int aNode2)
```

Compute `nor` operator for two boolean functions.

Parameters:

`aNode1` -

`aNode2` -

Returns:

the resulting boolean function root node.

---

nxor

```
public int nxor(int aNode1,
                int aNode2)
```

Compute nxor operator for two boolean functions.

Parameters:

`aNode1` -

`aNode2` -

Returns:

the resulting boolean function root node.

---

or

```
public int or(int aNode1,
              int aNode2)
```

Compute `or` operator for two boolean functions.

Parameters:

`aNode1` -

`aNode2` -

Returns:

the resulting boolean function root node.

---

positiveCofactor

```
public int positiveCofactor(int aRootNode,
                            int aVariableIndex)
```

Compute the positive cofactor of a boolean function with respect to a boolean variable.

Parameters:

`aRootNode` -

`aVariableIndex` -

Returns:

the resulting boolean function root node

---

printBDD

```
public java.lang.String printBDD(int aRoot)
```

Print information about a DAG given a root node.

Parameters:

`aRoot` - the indice of a root node.

Returns:

a String with information about a DAG.

---

removeReference

```
public void removeReference(int aRoot)
```

Use this method to unmark a boolean function root as a fuction of interest.

Parameters:

`aRoot` -

---

setVariableLevel

```
public void setVariableLevel(int aSourceLevel,
                             int aTargetLevel)
```

Move a variable in the ROBDD to a different level by repeateadly swap adjacent variable in the direction of a target level position.

Parameters:

`aSourceLevel` - the level with the variable to be moved.

`aTargetLevel` - the new level for the variable to be moved.

---

sop

```
public java.lang.String sop(int aRootNode)
```

Método pra gerar uma sop enquanto a isop não estava funcionando

Parameters:

`aRootNode` - um nodo

Returns:

uma String com uma sop redundante (SAT ALL)

---

swapLevels

```
public void swapLevels(int aParentLevel)
```

Swaps the variable in a level K with variable in level K+1.

Parameters:

`aParentLevel` - a variable level.

---

test

```
public static void test()
```

---

weight

```
public java.math.BigInteger weight(int aRootNode)
```

---

weight

```
public java.math.BigInteger weight(int aRootNode,
                                   java.util.List<java.lang.Integer> aSupport)
```

---

xor

```
public int xor(int aNode1,
               int aNode2)
```

Compute `xor` operator for two boolean functions.

Parameters:

`aNode1` -

`aNode2` -

Returns:

the resulting boolean function root node.