UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

GABRIEL AMMES PINHO

Análise de Métodos de Síntese de Funções Booleanas

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. Renato Perez Ribas

Porto Alegre
2018

“Some quotations are greatly improved by lack of context.”

— JOHN WYNDHAM

AGRADECIMENTOS

Gostaria de iniciar os agradecimentos pela minha mãe, Tanira, e meu dois pais, Nauro e Silvio, que mesmo estando longe, sempre me apoiaram antes, durante, e com certeza apoiarão depois da minha graduação. Continuando, agradeço a minha namorada Elisandra, que sempre esteve comigo durante esta jornada, sempre foi uma companheira para todas as horas e sempre aguentou minhas atividades acadêmicas madrugada a dentro, esta inclusive, e aos meus sogros, Nilda e Dalvo, que me acolheram como filho desde o dia que os conheci.

Gostaria de agradecer também aos professores Renato Ribas e André Reis, pela oportunidade de participar do laboratório e pelo conhecimento inserido neste ambiente por eles. Agradeço a todos que passaram pelo laboratório nos últimos três anos, que sempre me ajudaram em tudo que eu precisei, me possibilitaram o futebol semanal do laboratório, que continua até hoje, e se tornaram uma parte fundamental da minha formação, tanto atual quanto futura.

Finalmente, gostaria de agradecer a todos, professores, colegas de graduação e colegas de laboratório, que se tornaram amigos que levarei para toda minha vida.

RESUMO

Diversos métodos de síntese de funções Booleanas foram propostos na literatura mas o relacionamento entre elas não é claramente definido e uma comparação se torna útil para decidir qual método é mais indicado para uma dada aplicação. Este trabalho apresenta e implementa diversos métodos de síntese de funções Booleanas. Dentre eles, métodos que utilizam operações AND e OR, métodos que utilizam operações AND, OR e XOR, métodos que resultam em expressões com estrutura dois níveis e métodos que resultam em expressões com estrutura multinível. Ao final, é feita uma comparação destes métodos considerando métricas como número de literais, custo de implementação, profundidade lógica e tempo de execução.

Palavras-chave: Funções Booleanas. Síntese lógica. Ferramentas de EDA. Otimização lógica. Circuitos digitais.

A Comparative Analysis of Different Boolean Function Synthesis Methods

ABSTRACT

Several logic function synthesis approaches have been proposed but the relationship between them is not clearly defined. Such a comparison becomes quite useful for the decision of which one is more suitable for a given application. This work presents and implements a set of Boolean function synthesis methods. This set of methods contains methods based in AND and OR operations, methods based in AND, OR and XOR operations, two-level synthesis methods and multilevel synthesis methodos. At the end of this work, a comparison is done considering metrics as literals number, implementation cost, logic depth and execution time.

Keywords: Boolean function. Logic synthesis. CAD tool. Logic optimization. Digital circuit.

LISTA DE ABREVIATURAS E SIGLAS

CI	Circuito Integrado
EDA	*Eletronic Design Automation*
ERMPF	Expressão de Reed-Muller de Polaridade Fixa
ERMPP	Expressão de Reed-Muller de Polaridade Positiva
EPSDKRO	Expressão Pseudo Kroenecker
EPSDRM	Expressão Pseudo Reed-Muller
KRO	Expressão de Kronecker
MUX	Multiplexador
POS	*Product-of-Sums*
RTL	*Register Transfer Level*
SOP	*Sum-of-Products*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuits*

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Desde a criação do primeiro CI em 1958, o número de transistores por área de silício vem aumentando exponencialmente com o passar dos anos. Este comportamento for previsto por Moore em 1965 (MOORE, 1965), sendo popularmente conhecido como a Lei de Moore. Inicialmente, a Lei de Moore predizia que o número de transistores por área de silício, a um mesmo custo, dobraria a cada dois anos. Anos depois esta taxa foi corrigida para o dobro de transistores a cada 18 meses, sendo que esta taxa é vista até os dias atuais.

Com o aumento exponencial do número de transistores em um CI, além do aumento da complexidade da fabricação do mesmo, fez com que o desenvolvimento manual de CIs se tornasse impraticável. Desta forma, é necessária a utilização de ferramentas computacionais para este desenvolvimento. Estas ferramentas são chamadas de ferramentas de EDA (*Eletronic Design Automation*).

Geralmente, ferramentas de EDA executam três grandes etapas: a *síntese de auto-nível*, a *síntese lógica* e a *síntese física* (MICHELI, 1994).

A síntese de alto-nível consiste em transformar uma descrição algorítmica do comportamento desejado ao circuito em uma descrição de hardware, como RTL (*Register Transfer Lever*), que implementa este comportamento. A descrição algorítmica é comumente feita utilizando formatos como System C, VHDL e Verilog.

A síntese lógica tem como objetivo transformar uma descrição em RTL em uma estrutura primitiva de um circuito, como uma representação por portas lógicas. Esta etapa é tipicamente dividida em três fases: otimização independente de tecnologia, mapeamento tecnológico e otimização dependente de tecnologia. A primeira fase aplica transformações que não dependem da tecnologia do circuito final, dependendo apenas do comportamento funcional desejado, e gera uma estrutura intermediária, que pode ser, por exemplo, uma expressão Booleana, uma rede Booleana ou um grafo. Em seguida, o mapeamento tecnológico mapeia partes da estrutura intermediária em células com informações relativas a tecnologia do circuito final. Por fim, a otimização dependente de tecnologia realiza otimizações nas células mapeadas, como redimensionamento de células e duplicação de lógica.

A síntese física é tipicamente dividida em duas fazes: o posicionamento, que tem como objetivo posicionar as células dentro da área do chip de forma com que não haja sobreposições de células, e o roteamento destas células, realizando a conexão destas células

através de fios com o objetivo de minimizar métricas como comprimento total de fios e diminuição do atraso causados por estes fios, por exemplo.

Uma possível forma de realizar a síntese independente de tecnologia é utilizando métodos de síntese de funções Booleanas. Um método de síntese de funções Booleanas busca representar um dado comportamento lógico, geralmente representado por uma tabela verdade, em uma expressão Booleana contendo literais e operadores Booleanos.

Existem diversos métodos de síntese de funções Booleanas e cada método pode apresentar diferentes abordagens. Uma função Booleana pode ser sintetizada em uma estrutura de dois níveis, como somas-de-produtos (SoP) ou produto-de-somas (PoS) , ou em uma expressão fatorada de estrutura multi-nível (MICHELI, 1994). Existem abordagens onde a representação de funções pode ser restringido ao uso de operações de AND e OR, enquanto outras abordagens utilizam também a operação de OU-Exclusivo (XOR).

1.1 Motivação

Cada método de síntese de funções Booleanas apresenta características interessantes mas também desvantagens que devem ser consideradas. Apesar destas características serem conhecidas, o relacionamento entre diferentes métodos não é bem estabelecido. Usualmente, alguns métodos apresentam resultados melhores para área mas apresentam uma grande profundidade lógica, enquanto outros apresentam melhores resultados em profundidade lógica mas apresentam resultados piores em área. Logo, o quanto um método ganha ou perde em diferentes aspectos em relação a outros métodos, se torna uma questão interessante de ser respondida.

1.2 Objetivo

Este trabalho tem como objetivo implementar diferentes métodos de síntese de funções Booleanas e comparar os seus resultados em relação ao número de literais, custo de implementação, a profundidade lógica da expressão resultante e a tempo de execução destes métodos.

1.3 Organização do Texto

O restante do texto está organizado da seguinte forma. No capítulo 2, são apresentados diversos métodos de síntese de funções Booleanas, explicitando suas características e como o método realiza a síntese. No capítulo 3, é apresentada a implementação dos métodos apresentados no capítulo anterior, detalhando estrutura de dados e estratégias utilizadas para realizar a implementação dos mesmos. No capítulo 4, são apresentados os resultados deste trabalho, comparando os métodos apresentados em relação ao número de literais, custo de implementação, profundidade lógica e tempo de execução para diversas funções Booleanas. Por fim, no capítulo 5 é feito um resumo e a conclusão deste trabalho.

2 MÉTODOS DE SÍNTESE

Em vários métodos apresentado neste trabalho, a entrada será uma tabela verdade de n variáveis. Uma tabela verdade de n variáveis possui 2^n linhas, onde cada linha representa uma possível valoração das variáveis. Cada linha possui uma ou mais saídas, que representam se aquela valoração das variáveis é verdadeira, falsa ou *don't care*, para uma dada função. Neste trabalho somente serão utilizadas tabelas verdades de uma saída, com os possíveis valores de saída sendo verdadeiro ou falso. Um exemplo de tabela verdade é apresentados na Figura 2. Está tabela verdade vai ser utilizada como entrada de diversos métodos apresentados neste capitulo, para um estudo de caso.

Figura 2.1: Exemplo de tabela verdade.

A	B	C	OUT
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Uma tabela verdade pode ser representada por seus mintermos e seus maxtermos. Um mintermo é um produto de literais onde cada variável aparece uma vez em forma complementar ou não-complementar. Uma linha onde a saída é verdadeira resulta em um mintermo positivo, enquanto uma saída falsa resulta em um mintermo negativo. A conversão de valores em binário de uma tabela verdade para os mintermos é feita considerando as valorações de variáveis na respectiva linha. Se a valoração da variável é positiva, o literal presente no mintermo terá polaridade positiva, caso contrário terá polaridade negativa. Esta conversão, considerando todas as valorações de 3 variáveis, é mostrada na Figura 2.

Figura 2.2: Conversão de bits para mintermos.

$$
\begin{aligned}
&abc \\
000 &\longrightarrow \overline{a}\cdot\overline{b}\cdot\overline{c} \\
001 &\longrightarrow \overline{a}\cdot\overline{b}\cdot c \\
010 &\longrightarrow \overline{a}\cdot b\cdot\overline{c} \\
011 &\longrightarrow \overline{a}\cdot b\cdot c \\
100 &\longrightarrow a\cdot\overline{b}\cdot\overline{c} \\
101 &\longrightarrow a\cdot\overline{b}\cdot c \\
110 &\longrightarrow a\cdot b\cdot\overline{c} \\
111 &\longrightarrow a\cdot b\cdot c
\end{aligned}
$$

Um maxtermo é uma soma de literais onde cada variável aparece uma vez em forma complementar ou não-complementar. Uma linha onde a saída é verdadeira resulta em um maxtermo negativo, enquanto uma saída falsa resulta em um maxtermo positivo. No caso dos maxtermos, se a valoração da variável é positiva, o literal presente no maxtermo terá polaridade negativa, caso contrário terá polaridade positiva. Essa conversão, considerando todas as valorações de 3 variáveis, é mostrada na Figura 2.

Figura 2.3: Conversão de bits para maxtermos.

$$
\begin{array}{l}
abc \\
000 \longrightarrow a+b+c \\
001 \longrightarrow a+b+\overline{c} \\
010 \longrightarrow a+\overline{b}+c \\
011 \longrightarrow a+\overline{b}+\overline{c} \\
100 \longrightarrow \overline{a}+b+c \\
101 \longrightarrow \overline{a}+b+\overline{c} \\
110 \longrightarrow \overline{a}+\overline{b}+c \\
111 \longrightarrow \overline{a}+\overline{b}+\overline{c}
\end{array}
$$

Neste trabalho serão utilizados na maior parte dos casos os mintermos positivos e os maxtermos positivos. Logo, ocultarei a expressão positivo quando me referir a eles e explicitarei se forem usados os mintermos ou maxtermos negativos.

Os métodos para síntese de funções lógicas aqui apresentados não estão completos, existem muitos outros. As abordagens discutidas a seguir são umas das mais utilizadas na síntese lógica de circuitos digitais.

2.1 Expressões com operações AND e OR

As operações AND e OR são as operações mais comuns na síntese de funções Booleanas. A grande maioria dos métodos utiliza estas operações na expressão Booleana resultante.

Nesta seção, serão apresentado cinco métodos que utilizam somente operações AND e OR. Dentro destes cinco métodos, os três primeiros apresentam um estrutura dois níveis e os outros dois uma estrutura multinível.

2.1.1 Expressões Canônicas

A forma mais direta de representar uma tabela verdade em uma expressão lógica é sua forma normal canônica (MICHELI, 1994). A forma normal canônica pode ser construída de duas maneiras distintas: a forma normal canônica disjuntiva, composta de produtos de literais unidos por somas, onde cada produto de literais é chamado de cubo, e a forma normal canônica conjuntiva, composta de somas de literais unidas por produtos, onde cada soma de literais é chamado de cláusula. Devido à forma estrutural destas formas, a primeira é chamada de soma-de-produtos canônica, ou SOP canônica, e a segunda de produto-de-somas canônica, ou POS canônica.

Para a construção de uma expressão baseada nestas duas possíveis formas, são utilizados, respectivamente, os mintermos e os maxtermos de uma tabela verdade. Uma SOP é construída ligando todos os mintermos de uma função por operações OR enquanto uma POS é construída ligando todos os maxtermos de uma função por operações AND. Para a tabela verdade apresentada na Figura 2, sua SOP é mostrada abaixo:

$$F = a \cdot \overline{b} \cdot \overline{c} + a \cdot \overline{b} \cdot c + a \cdot b \cdot \overline{c} + \overline{a} \cdot b \cdot c, \tag{2.1}$$

e sua POS a seguir:

$$F = a + b + c \cdot a + b + \overline{c} \cdot a + \overline{b} + c \cdot \overline{a} + \overline{b} + \overline{c}. \tag{2.2}$$

2.1.2 Síntese de dois níveis utilizando lógica de MUX

Um multiplexador, ou MUX, é um bloco combinacional que seleciona um dos sinais de entrada para ser conectado a uma única saída (BROWN; VRANESIC, 2008). Em outras palavras, a saída de um MUX corresponde a um dos sinais de entrada de acordo com o sinal de seleção, sendo composto por 2^n entradas e n sinais de seleção.

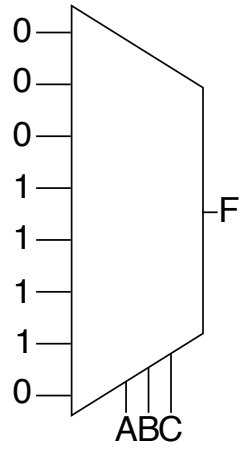
Um multiplexador pode ser logicamente implementado utilizando operações AND e OR. Cada entrada é ligada por uma operação AND aos sinais de seleção e estas operações AND são ligadas por uma operação OR. Desta forma, um MUX pode ser implementado por uma SOP, onde cada cubo corresponde a uma condição de seleção e a respectiva entrada selecionada. A expressão genérica de um MUX com quatro entradas, representadas por e0, e1, e2 e e3, e dois sinais de seleção, representados por a e b, é mostrada na

Equação a seguir:

$$F = \overline{a} \cdot \overline{b} \cdot e0 + \overline{a} \cdot b \cdot e1 + a \cdot \overline{b} \cdot e2 + a \cdot b \cdot e3 \tag{2.3}$$

Devido às semelhanças topológicas e de implementação, um MUX pode representar uma tabela verdade onde a saída de cada linha da tabela verdade é um sinal de entrada do MUX e as variáveis são os sinais de seleção. É necessário um MUX de n sinais de seleção para implementar uma tabela verdade de n variáveis, seguindo esta lógica. A implementação da tabela verdade utilizada como estudo de caso em um MUX 8x1 está na Figura 2.4 e sua expressão lógica é a mesma SOP da tabela da tabela verdade.

Figura 2.4: Implementação da tabela verdade em um MUX de oito entradas.



A representação de uma tabela verdade utilizando MUX mostrada anteriormente é bastante direta, mas não implementa nenhum tipo de minimização. É possível implementar uma função de n variáveis utilizando um MUX com 2^{n-1} sinais de entrada e $n-1$ sinais de seleção. Para fazer esta minimização, $n-1$ variáveis são utilizadas como sinais de seleção deste MUX com 2^{n-1} entradas e em cada entrada deste MUX haverá um MUX de duas entradas, sendo o seu sinal de seleção a variável restante. A Figura 2.5 mostra esta modificação no MUX de oito entradas apresentado anteriormente.

Dado um MUX de duas entradas existem três formas de simplificá-lo: se suas entradas apresentarem o mesmo valor, este MUX pode ser substituído por este valor; se a primeira entrada for 0 e a segunda entrada for 1, este MUX pode ser substituído pelo sinal de seleção com a polaridade positiva; se a primeira entrada for 1 e a segunda for 0, o MUX pode ser substituído pelo sinal de seleção com a polaridade negativa. Essa otimização é mostrada na Figura 2.6 e sua expressão lógica está abaixo:

$$F = \overline{a} \cdot b \cdot c + a \cdot \overline{b} + a \cdot b \cdot \overline{c}. \tag{2.4}$$

Figura 2.5: Utilização de MUX de duas entradas para simplificar transformar o MUX original em um MUX de quatro entradas.

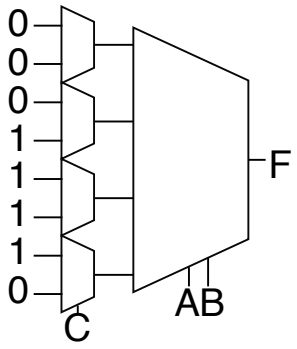
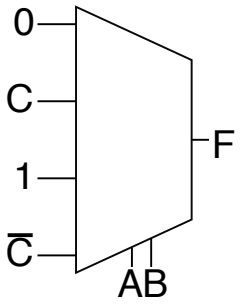


Figura 2.6: Função original implementada em um MUX de quatro entradas.



2.1.3 Algorítimo de Quine-McCluskey

O algoritmo de Quine-McCluskey é um dos mais difundidos métodos de síntese de funções Booleanas (MCCLUSKEY, 1956). Este algoritmo encontra uma SOP com o menor número de cubos de uma dada tabela verdade de n variáveis, podendo também ser utilizado para encontrar uma POS com o menor número de cláusulas. Estas expressões são conhecidas como ISOP, ou soma-de-produtos irredundante, e IPOS, ou produto-de-soma irredundante.

O algoritmo consiste em dois passos principais: encontrar todos os cubos implicantes primos da função e utilizar estes cubos implicantes primos para cobrir os mintermos desta função.

Um cubo implicante é um cubo que consegue representar, ou cobrir, um ou mais mintermos e é chamado de primo, ou redundante, se não pode coberto por nenhum outro cubo. Um implicante primo é chamado de essencial quando existe um mintermo que só é coberto por este cubo.

O algoritmo inicia utilizando os mintermos como cubos implicantes iniciais. Se

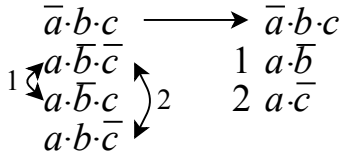
dois cubos possuem os mesmos literais e a única diferença entre estes cubos é a polaridade de um literal, este literal pode ser considerado como *don't care*, ou seja, pode ser removido do cubo. Esta simplificação é feita da seguinte forma:

$$a \cdot \overline{b} \cdot c + \overline{a} \cdot \overline{b} \cdot \overline{c} \equiv \overline{a} \cdot \overline{b} \cdot (c + \overline{c}) \equiv \overline{a} \cdot \overline{b} \cdot 1 \equiv \overline{a} \cdot \overline{b}, \tag{2.5}$$

e este novo cubo cobre os dois cubos anteriores.

Esta etapa é feita até que todos os cubos sejam comparados e a próxima etapa realiza as mesma ações que a anterior, mas considera os cubos simplificados e os que não foram utilizados em nenhuma simplificação na etapa anterior. Quando não existe nenhuma simplificação entre um etapa e outra, este passo está completo e os cubos resultantes são chamados de cubos implicantes primos. O lado esquerdo da Figura 2.7 mostra os cubos referentes a tabela verdade utilizada até aqui, as flechas representam quais os cubos que podem ser combinados. No lado direito desta figura estão os cubos 1 e 2 resultantes das combinações e o primeiro cubo que se manteve inalterado, pois não foi utilizado em nenhuma combinação. Após as combinações mostradas, nenhuma combinação pode ser feita e esta etapa é encerrada.

Figura 2.7: Combinação dos cubos e cubos resultantes.



A cobertura dos mintermos é feita fixando os implicantes primos essenciais e utilizando os outros implicantes primos para cobrir os mintermos restantes. Para fazer a cobertura dos cubos implicantes primos não essenciais, é utilizada uma tabela de cobertura. Cada linha desta tabela representa um cubo e cada coluna representa um mintermo que deve ser coberto. Quando um cubo cobre um mintermo, a posição da tabela referente ao cubo e ao mintermo é marcada. Um conjunto de linhas onde todas as colunas possuem pelo menos uma marcação, é considerada uma cobertura. A cobertura que possuir o menor número de linhas, ou seja de cubos implicantes primos é a saída do algoritmo.

Como continuação do exemplo da Figura 2.7, o primeiro mintermo só é coberto por ele mesmo, o terceiro mintermo só é coberto pelo cubo 1 e o quarto mintermo só é coberto pelo cubo 2. Sendo assim, todos os cubos resultantes são cubos implicantes primos essenciais e não tem necessidade de utilizar a tabela de cobertura. Um exemplo da aplicação da tabela de cobertura é visto na Seção 2.2. A expressão que resulta da

aplicação deste algoritmo na tabela verdade de exemplo é a seguinte:

$$F = a \cdot \overline{b} + \overline{a} \cdot b \cdot c + a \cdot \overline{c}. \tag{2.6}$$

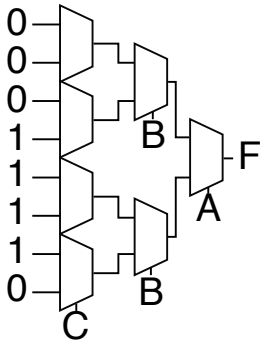
Também é possível encontrar uma expressão na forma de POS utilizando o algoritmo de Quine-McCluskey. Para isso, são utilizados os mintermos negativos da tabela verdade. O funcionamento do algoritmo é basicamente o mesmo, sendo a única diferença que a SOP resultante deve ser negada. Por fim, são aplicadas o teorema de De Morgan para transformar esta SOP negada em uma POS (HURLEY, 2011). A POS resultante da aplicação deste algoritmo utilizando a mesma tabela verdade é mostrada a seguir:

$$F = (a + b) \cdot (\overline{a} + \overline{b} + \overline{c}) \cdot (a + c). \tag{2.7}$$

2.1.4 Síntese Multinível Utilizando MUX

Até agora, foram apresentados métodos que resultam em expressões com dois níveis, sendo este o primeiro que apresenta resultados multinível. Neste método são utilizados os conceitos apresentados na Seção 2.1.2 referentes a substituição de um MUX de duas entradas por um único sinal e diminuição de tamanho de um MUX utilizando MUXs de duas entradas. Este método consistem em transformar um MUX de 2^n em $2^n - 1$ MUXs de duas entradas, divididos em n níveis. A representação da tabela verdade utilizada como exemplo em MUXs de duas entradas é mostradas na Figura 2.8.

Figura 2.8: Função original implementada em um MUX de duas entradas.



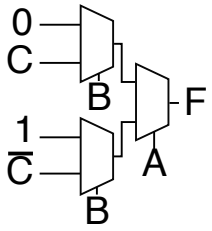
As simplificações de MUX de duas entradas são feitas e em caso de dois MUX tiverem as mesmas entradas e o mesmo sinal de seleção, um deles pode ser desconsiderado e a saída do restante estará ligada nas respectivas entradas. A expressão resultante será construída a partir o MUX mais próximo a saída, considerando sua expressão genérica,

mostrada na Equação 2.3.

A rede de MUX resultantes considerando a tabela verdade utilizada como exemplo é mostrada na Figura 2.9. Apesar de serem as mesmas entradas que eram utilizadas no MUX de 4 entradas na Seção 2.1.2, a expressão final será mais simplificada, pois o literal a é posto em evidência, como mostrado a seguir:

$$F = (\overline{a} \cdot b \cdot c) + a \cdot (\overline{b} + (b \cdot \overline{c})). \tag{2.8}$$

Figura 2.9: Rede de MUX de duas entradas após otimizações.



2.1.5 Fatoração de funções Booleana

Fatorar uma função Booleana é uma das operações básicas dos algoritmos de síntese lógica. De forma geral, fatorar uma expressão Booleana é modificá-la com o intuito de diminuir o número de literais utilizados. Existem diferentes abordagens de fatoração Booleana, porém não existe um método para qualquer função com mais que 4 variáveis que garanta uma expressão com número ótimo de literais (MICHELI, 1994). O método escolhido para fazer parte deste trabalho é um método que fatora uma função Booleana utilizando particionamento de grafos (MINTZ; GOLUMBIC, 2005).

Este método consiste em reduzir a expressão de entrada utilizando particionamento de grafos de forma recursiva, até que um subgrafo represente uma função *read-once*. Uma função é chamada de *read-once* se for possível representar esta função em uma expressão onde cada literal aparece apenas uma vez. Esta classe de função é de interesse pois é possível obter uma expressão com número ótimo de literais se a função for *read-once* (GOLUMBIC; MINTZ; RODICS, 2005).

Para uma dada expressão de entrada, que consiste em uma SOP ou uma POS, um grafo é criado para esta expressão. Um nodo é adicionado ao grafo para cada cubo/clausula da expressão. Dois nodos são ligados por uma aresta se os mesmos tiverem literais em comum.

Além do grafo da expressão original, também é criado um grafo para sua expressão *dual*, ou seja, se a entrada for uma SOP, é necessário gerar uma POS equivalente, e vice-versa. Em seguida, estes grafos são analisados para escolher o particionamento que minimiza o número de arestas entre dois subgrafos. As expressões referentes a estes subgrafos são utilizadas como entrada para próxima chamada da recursão.

As expressões resultantes destas chamadas recursivas serão ligadas utilizando a operação principal da expressão que originou o grafo utilizado. Desta forma, se o grafo foi gerado a partir de uma SOP, as expressões serão ligadas por uma operação OR, e caso contrário, por uma operação AND. A recursão acaba quando uma função *reed-once* é encontrada e sua expressão resultante será gerada a partir do algoritmo mostrado em (GOLUMBIC; MINTZ; RODICS, 2005). A expressão resultante para este método, utilizando a SOP canônica mostrada na Equação 2.1 como entrada, está a seguir:

$$F = (\overline{a} \cdot c) + (a \cdot (\overline{b} + \overline{c})). \tag{2.9}$$

Este é o único método deste trabalho que não foi implementado pelo autor. O método tinha sido previamente implementado por membros do laboratório de pesquisa e constava no repositório do mesmo. Desta forma, sua implementação não será apresentada no capítulo seguinte.

2.2 Expressões com operações AND, OR e XOR

Além das operações AND e OR utilizadas no métodos anteriores, a operação XOR é uma operação bastante utilizada, principalmente em circuitos aritméticos. A sua utilização em métodos de síntese de funções Booleanas pode melhorar os resultados da síntese (SASAO, 1993b), devido a possibilidade de utilizar somente uma operação para representar o mesmo comportamento lógico que necessitava de três operações se forem consideradas apenas operações AND e OR.

2.2.1 Extensão do algoritmo de Quine-McCluskey

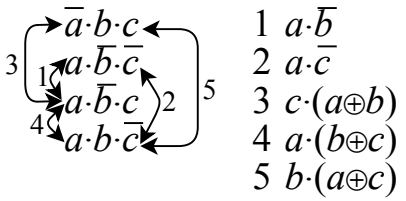
Uma expansão ao algoritmo de Quine-McCluskey é apresentada em (TURTON, 1996). Esta expansão considera que os cubos podem conter operações XOR, além das operações AND que são utilizadas usualmente. Para a explicação desta expansão consi-

derarei que um cubo pode conter operações AND e XOR.

A diferença da implementação desta expansão em relação a implementação do algoritmo original está na combinação dos cubos. Além de combinar como no algoritmo original, onde apenas um literal tem a polaridade diferente, também são combinados cubos que possuem dois literais com polaridade diferente. Caso estes literais tenham polaridades inversas em cada um dos cubos, será feita uma XOR entre estas duas variáveis. Caso contrário, ou seja, um cubo tem estes dois literais com polaridade positiva e o outro cubo com polaridade negativa, será feita uma XNOR destas duas variáveis.

Utilizando a tabela verdade utilizada como exemplo, a Figura 2.10 mostra a simplificação dos cubos neste algoritmo. Na esquerda desta imagem são mostrados os mintermos, e as flechas representam quais os cubos que tem dois literais com polaridades diferentes e podem ser simplificados. A direita da imagem estão os cubos resultantes, onde os cubos 1 e 2 foram obtidos combinando cubos com apenas um literal com polaridade diferente, e apresentam apenas operações AND, e os cubos 3, 4 e 5 foram obtidos combinando cubos com dois literais com polaridade diferente, e apresentam as operações AND e XOR.

Figura 2.10: Combinação dos cubos e cubos resultantes.



Todos os mintermos são cobertos por mais de um cubo implicante primo, desta forma, não existe nenhum cubo implicante primo essencial e é necessário utilizar a tabela de cobertura para obter a expressão final. A Figura 2.11 mostra a tabela de cobertura para este problema, considerando seus cubos implicantes primos e seus mintermos.

Figura 2.11: Tabela de cobertura.

Cubos \ Mintermos	$\overline{a}\cdot b\cdot c$	$a\cdot\overline{b}\cdot\overline{c}$	$a\cdot\overline{b}\cdot c$	$a\cdot b\cdot\overline{c}$
$a\cdot\overline{b}$		X	X	
$a\cdot\overline{c}$		X		X
$c\cdot(a\oplus b)$	X		X	
$a\cdot(b\oplus c)$			X	X
$b\cdot(a\oplus c)$	X			X

Existem diversos conjuntos de linhas da tabela que fazem com que todas as colunas sejam marcadas, porém só dois utilizam o menor número possível de linhas. O

primeiro conjunto de linhas contém a primeira e a quinta linha e o segundo conjunto de linhas contém segunda e a terceira linha. A expressão resultante destas duas coberturas estão a baixo:

$$F = (a \cdot \overline{b}) + b \cdot (a \oplus c), \tag{2.10}$$

$$F = (a \cdot \overline{c}) + c \cdot (a \oplus b). \tag{2.11}$$

2.2.2 Síntese Utilizando Expansões

Outra forma de obter uma expressão Booleana com características específicas é aplicando expansões a uma expressão inicial. Diversas classes de expressões que utilizam operações XOR são mostradas em (SASAO, 1993a). Dentro deste conjunto de classes, 5 classes são obtidas aplicando expansões em uma expressão de entrada.

As expansões que serão utilizadas para esta síntese são: a Expansão de Davio Positiva, representada pela expressão abaixo:

$$F = 1 \cdot f_0 \oplus x_1 \cdot f_2, \tag{2.12}$$

a Expansão de Davio Negativa, representada pela expressão abaixo:

$$F = \overline{x_1} \cdot f_2 \oplus 1 \cdot f_1 \tag{2.13}$$

e a Expansão de Shannon, representada pela expressão abaixo:

$$F = \overline{x_1} \cdot f_0 \oplus x_1 \cdot f_1, \tag{2.14}$$

onde F= $f(x_1, x_2, x_3, ..., x_n)$, $f_0 = f(0, x_2, x_3, ..., x_n)$, $f_1 = f(1, x_2, x_3, ..., x_n)$e $f_2 = f_0 \oplus f_1$.

As expressões abordadas neste trabalho estão definidas a seguir:

Expressão Reed-Muller de Polaridade Positiva: É aplicada a Expansão de Davio Positiva para cada variável da expressão de entrada. Esta expressão é única para uma dada expressão e é composta de somente literais com polaridade positiva. A representação da Equação 2.1 está abaixo:

$$F = a \oplus (b \cdot (a \oplus a)) \oplus (c \cdot (a \oplus (b \cdot (a \oplus a)) \oplus a \oplus b)). \tag{2.15}$$

Expressão Reed-Muller de Polaridade Fixa: Podem ser aplicadas tanto a Expansão de Davio Positivo quanto de Davio Negativo para cada variável da expressão de entrada. Desta forma, serão geradas 2^n possíveis expressões para uma dada expressão de entrada. Estas expressões terão os literais ou com polaridade positiva ou com polaridade negativa. Uma possível representação da Equação 2.1 está abaixo:

$$F = a \oplus (c \cdot (a \oplus (a \oplus b))). \tag{2.16}$$

Expressão Kronecker: Podem ser aplicadas a Expansão de Davio Positivo, de Davio Negativo ou de Shannon a cada variável da expressão de entrada. Desta forma, serão geradas 3^n possíveis expressões para uma dada expressão de entrada, sem nenhuma restrição na polaridade dos literais. Uma possível representação da Equação 2.1 está abaixo:

$$F = (\overline{c} \cdot a) \oplus (c \cdot (a \oplus b)). \tag{2.17}$$

Expressão Pseudo Reed-Muller: Quando uma das expansões mostradas é aplicada em uma expressão são geradas duas sub-expressões conectadas pelo operador XOR. No caso da Expressão Pseudo Reed-Muller, podem ser aplicadas a Expansão de Davio Positiva ou de Davio Negativa para cada variável e para cada sub-expressão. Desta forma, serão geradas 2^{2^n} possíveis expressões para uma dada expressão de entrada, sem nenhuma restrição na polaridade dos literais. Uma possível representação da Equação 2.1 está abaixo:

$$F = a \oplus (c \cdot b). \tag{2.18}$$

Expressão Pseudo Kronecker: Na Expressão Pseudo Kroneker, podem ser aplicadas a Expansão de Davio Positiva, de Davio Negativa ou de Shannon para cada variável e para cada sub-expressão. Desta forma, serão geradas 3^{2^n} possíveis expressões para uma dada expressão de entrada, sem nenhuma restrição na polaridade dos literais. Uma possível representação da Equação 2.1 está abaixo:

$$F = a \oplus (c \cdot b). \tag{2.19}$$

3 IMPLEMENTAÇÃO

A implementação deste trabalho foi feita utilizando a linguagem de programação Java e a validação dos resultados, ou seja, se as expressões da saída correspondiam logicamente a tabela verdade da entrada, foi feita utilizando o *framework* Karma (KLOCK et al., 2007). É possível implementar os métodos mostrados no capítulo anterior de diversas maneiras. As implementações mostradas neste capítulo não são o estado-da-arte, sendo assim, existem implementações mais eficientes na literatura.

A tabela verdade, entrada de diversos métodos, é representada por dois vetores de inteiro, um contendo os valores das linhas onde a função vale 1, ou seja, os mintermos positivos, e o outro contendo os valores das linhas onde a função vale 0, os maxtermos positivos. Todos os métodos apresentam a expressão de saída em uma String e alguns destes, utilizam estas expressões como entrada e saída.

3.1 Expressões Canônicas

Por serem a forma mais direta de sintetizar uma funções, possuem também uma implementação bastante direta. Para cada inteiro do vetor de mintermos, é feita a leitura bit a bit para determinar qual a polaridade de cada literal e são inseridos caracteres para indicar a operação AND entre os literais na String de saída. Quando o bit vale 0, o literal terá polaridade negativa, e caso valer 1, o literal terá polaridade positiva. Ao fim da leitura de cada inteiro, é adicionado um caractere para indicar a operação OR entre o cubos. A POS canônica tem funcionamento semelhante, modificando a entrada, as operações e as polaridades. É utilizado o vetor referente aos maxtermos neste caso. Na leitura bit a bit dos inteiros do vetor, o literal terá polaridade positiva se o bit for 0 e polaridade negativa se o bit for 1. Os literais são separados por caracteres que indicam a operação OR e as clausulas são separadas por caracteres que indicam a operação AND.

3.2 Síntese de Dois Níveis Utilizando Lógica de MUX

Para a síntese dois níveis utilizando a lógica de MUX, é utilizado como entrada o vetor referente aos mintermos. É utilizado um vetor de inteiros de $2^{(n-1)}$ posições como estrutura intermediária, com todos os valores do vetor inicialmente iguais a 0. Este vetor

pode conter quatro diferentes valores, onde o valor 0 representa o valor lógico 0, o valor 1 representa o valor lógico 1, o valor 2 representa a variável presente na seleção com polaridade positiva e o valor 3 representa esta variável com polaridade negativa.

Quando é necessário inserir um valor ao vetor intermediário, isto é feito na posição referente a divisão inteira por 2 do inteiro. Para cada inteiro do vetor de entrada os seguintes passos são seguidos: se o inteiro for impar, seu valor é armazenado em uma variável temporária; se o valor for par, o inteiro armazenado é testado e caso seja igual ao inteiro atual menos 1, o valor 1 será adicionado ao vetor intermediário, caso contrário, o valor adicionado referente a este inteiro será 2 e o deve ser adicionado o valor 3 referente ao inteiro armazenado e seu valor deve ser apagado.

Cada posição deste vetor intermediário corresponde a respectiva entrada de um MUX com $2^{(n-1)}$ entradas. A expressão de saída é gerada considerando o funcionamento lógico de um MUX, como mostrado na Equação 2.3.

A seleção de qual a variável será utilizada na entrada afeta o resultado da síntese, desta forma, este método é executado n vezes, utilizando todas as variáveis, uma de cada vez, como uma possível entrada, e é selecionada a menor expressão de saída como resultado do método. As expressões resultantes são avaliadas e a expressão com menos literais é escolhida.

3.3 Algoritmo de Quine-McCluskey

A entrada deste método é o vetor de inteiros referente aos mintermos. Os mintermos são convertidos em uma estrutura para representar cubos. Esta estrutura é composta por inteiros, mas as manipulações são feitas bit a bit. Um mintermo com n literais, precisa de $2 * n$ bits para ser representado nesta estrutura, onde cada literal é representado por dois bits. Isso é necessário pois é preciso representar os literais que são *don't care* nos cubos. Cada literal negativo é representado pelos bits 01, os literais positivos são representados pelos bits 10 e os *don't cares* por 11, mantendo a ordem dos literais originais. Esta codificação é chamada de *positional-cube notation* (MICHELI, 1994).

Os cubos são divididos considerando a quantidade de literais positivos para facilitar a combinação entre eles. Desta forma, os cubos somente são comparados com os cubos que possuem um literal positivo de diferença. Para comparar dois cubos, é feita uma operação XOR entre eles. Se no resultado existir somente dois bits 1 e estes bits estiverem em posições referentes a um literal, os cubos podem ser combinados.

Após obter todos os cubos implicantes primos, os cubos implicantes essenciais são identificados e colocados na solução final. Caso ainda restem mintermos a serem cobertos, é necessário realizar a cobertura destes.

Para a cobertura é utilizada um pacote chamado de BitSet, por ser uma boa estrutura para gerar a tabela de cobertura (RUDELL; SANGIOVANNI-VINCENTELLI, 1989). Um BitSet pode ser considerado como um vetor onde os possíveis valores são 0 e 1, mas é considerada pelo compilador como um número inteiro. Diversas operações binárias estão disponíveis neste pacote. Para a cobertura, é feito um vetor de BitSet para representar a tabela de cobertura. Cada BitSet representa um cubo, e cada valor neste representa se um mintermo é coberto ou não por ele.

O algoritmo inicia ordenando os cubos em relação ao número de mintermos que eles cobrem e em seguida é feita uma estimativa de qual o número mínimo de cubos que podem ser utilizados para cobrir os mintermos, que chamarei de c. Por exemplo, se os quatro maiores cubos cobrem quatro mintermos e existem 12 mintermos, a menor cobertura possível, teoricamente falando, teria três cubos, ou seja, o valor de c seria três. São feitas as combinações de c cubos e caso todas as combinações tenham sido feitas e nenhuma solução tenha sido encontrada, o valor de c é incrementado, até que uma cobertura seja encontrada.

Uma otimização intermediária é feita quando são escolhidos $(c - 1)$ cubos. Caso número de mintermos que precisam ser cobertos seja maior que o número de mintermos que o próximo cubo cobre, não é mais necessário continuar esta combinação. Como os cubos estão em ordem decrescente em relação ao número de mintermos que cada cubo cobre, se o próximo cubo cobre menos mintermos que o necessário, nenhum cubo depois dele cobrirá.

Como as combinações foram feitas de forma crescente em relação ao número de cubos, quando uma cobertura encontrada, todas as coberturas contendo o número c atual de cubos, serão as coberturas com o número ótimo de cubos. As expressões resultantes são avaliadas e a expressão com menos literais é escolhida.

3.4 Síntese Multinível Utilizando Lógica de MUX

Para a implementação desde método são utilizados três estruturas principais, um vetor de inteiros com $2^{(n-1)}$ posições, que orientará o funcionamento do algoritmo, mapa dos MUXs utilizados, que será utilizado para reutilizar um MUX existente, e um vetor

String, que será utilizado para gerar a expressão de saída. Nos dois vetores, a seguinte codificação é utilizada: os valores lógicos 0 e 1 são representadas pelos valores 0 e 1; a variável mais significativa é representada pelo valor 3 e sua negação pelo valor 4, e assim sucessivamente até a variável menos significativa, que terá o valor $2 * n$ e sua negação o valor $2 * n + 1$. Quando um MUX for criado, será dado um identificador igual ao último valor utilizado na codificação somado um, no caso do primeiro MUX criado, seu identificador será $2 * n + 2$.

O vetor de inteiros é inicializado utilizando o método mostrado na Seção 3.2 com o vetor referentes aos mintermos como entrada. Este vetor poderá conter os valores 0, 1, $2*n$ e $2*n+1$. O vetor de String é inicializado da posição 0 até $2*n+1$, com uma String referente a codificação mostrada anteriormente, ou seja, constantes e literais referentes a codificação.

Cada iteração computa um nível de MUX, ou seja, os MUXs que utilizam uma dada variável como seleção. Após cada iteração, o tamanho do vetor será reduzido pela metade. Duas posição sequentes do vetor são comparadas, caso os valores representem uma das combinações mostradas na Seção 2.1.4, o valor resultante da combinação será adicionado no vetor utilizado na próxima iteração. Caso os valores não representem nenhuma das combinações, é necessário criar um novo MUX. O mapa de MUXs é consultado para verificar se este novo MUX já foi criado. Caso o mapa contenha o MUX, o mesmo identificador é utilizado no vetor de inteiro e não é necessária nenhuma outra modificação. Caso o mapa não contenha este MUX, é gerado um identificador para este MUX, é computada a String referente a este MUX e é adicionada uma entrada no mapa referente a ele.

O algoritmo terminará quando o vetor de inteiros possuir apenas uma posição e o valor presente nesta posição será o referente a expressão Booleana resultante na vetor de String.

Assim como o método apresentado na Seção 3.2, a ordem das variáveis de seleção afeta o resultado da síntese. Foram utilizadas $2 * n$ permutações na ordem das variáveis de forma com que exista, pelo menos, uma permutação com cada variável como a variável mais significativa e uma permutação com cada variável como a variável menos significativa. As expressões resultantes são avaliadas e a expressão com menos literais é escolhida.

3.5 Extensão do Algoritmo de Quine-McCluskey

Para realizar a extensão do algoritmo de Quine-McCluskey é necessário alterar a etapa de obtenção dos cubos, inclusive a estrutura de dados. O vetor de mintermos é utilizado como entrada deste método.

Como é necessário representar a existência de uma XOR e e quais são as variáveis de entrada da mesma, um número fixo de bits como no algoritmo original não pode ser utilizado. Para a representação do cubos é utilizado um vetor de inteiros, onde o valor do inteiro é 2 elevado a sua posição no vetor. A polaridade é representada pelo sinal do inteiro e as operações XOR e XNOR são representadas pela soma dos valores nas respectivas posições (TURTON, 1996).

Desta forma, quando é feita a passagem dos inteiros presentes no vetor de entrada para cubos, o bit menos significativo estará na posição 0 do vetor e terá o valor 1 (2^0), o segundo bit menos significativo estará na posição 1 do vetor e tera o valor 2 (2^1) e assim sucessivamente. Caso o bit seja 0 seu valor será negativo e caso seja 1 se valor será positivo. Quando existir uma operação XOR ou XNOR entre duas ou mais posições, é feita a soma dos valores entre estas duas posições e o sinal da soma é definido por qual operação é feita, se for a operação XOR, o sinal é positivo, se for a operação XNOR, será negativo. Por exemplo, se existir uma operação XOR entre os dois literais representados pelas posições 0 e 1, os valores destas duas posições será 3, e se existir uma operação XNOR entre os literais representados pelas posições 2 e 3, os valores destas duas posições seriam -12. Os literais que são *don't care*, são representados pelo valor 0.

Os cubos são divididos considerando o número de literais positivos, assim como no anterior, porém, é necessário comparar cubos com 1 literal positivo de diferença, assim como no caso anterior, e cubos com 0 e 2 literais de diferença, que são os casos onde ocorre a inserção da operação XOR e XNOR, respectivamente.

A comparação é feita fazendo a soma dos valores nos vetores, considerando cada posição. O vetor resultante é analisado considerando qual a diferença de literais positivos entre estes dois cubos. Se esta diferença for 1, os cubos são combinados caso exista somente um valor do vetor resultante igual a 0 e todas as outras posições sendo igual ao dobro do valor original. Os casos onde a diferença é 0 e 2 são analisados da mesma forma. Caso existam dois valores do vetor igual a zero e todas as outros valores sendo igual ao dobro do valor original a combinação pode ser feita. A diferença está no sinal, que é positivo se a diferença for 0 e negativo se a diferença for 2.

A cobertura é feita da mesma forma que foi apresentada na Seção 3.3 e as expressões resultantes são avaliadas e a expressão com menos literais é escolhida.

3.6 Síntese Utilizando Expansões

A expressão de entrada é uma expressão canônica gerada pelo algoritmo mostrado na Seção 3.1, sendo esta representa por uma *LogicTree*. Uma *LogicTree* é uma estrutura de dados, desenvolvida pelo laboratório de pesquisa e presente no repositório do mesmo, que consiste em uma árvore onde cada nó interno representa uma função Booleana e cada folha representa uma variável de entrada ou uma constante, 0 ou 1. Dentre diversos métodos implementados utilizando esta estrutura, os utilizados nesta etapa do trabalho foram o *buildTree*, que constrói a árvore utilizando uma expressão Booleana como entrada, *replaceVariableName*, que substitui uma variável por outra variável ou por uma constante, *constantPropagator*, para a propagação de constantes e os métodos de criação de nós, como o *createAnd* e *createXor*, que criam, respetivamente, nós com operações AND e XOR com entradas que podem ser outros nós, variáveis ou constantes.

Inicialmente é utilizada o método *buildTree* para criar a *LogicTree* inicial. Para implementar as expansões é necessário a implementação das funções $f0$, $f1$ e $f2$, utilizadas nas expansões, como mostrado na Seção 2.2.2. Para a implementação das funções $f0$ e $f1$ é utilizada o método *replaceVariableName* para trocar uma dada variável por 0 ou por 1, respectivamente. Para a implementação da função $f2$ é utilizada o método *createXor* para criar um nó XOR com as funções $f0$ e $f1$ como entradas deste nó.

Para a implementação das expansões, é utilizado método *createAnd* para criar dois nós AND, com as entradas adequadas, e utilizado o método *createXor* para criar um nó XOR com os dois nós AND criados como entrada.

Para a obtenção das expressões Reed-Muller de polaridade positiva, Reed-Muller de polaridade fixa e de Kroneker, são aplicadas as expansões necessárias em relação a primeira variável e as expressões resultantes serão a entrada para a próxima etapa, onde são aplicadas as expansões em relação a segunda variável, e assim por diante até a última variável. As expressões resultantes da última etapa são avaliadas e a expressão com menos literais é escolhida.

Para as expressões Pseudo Reed-Muller e Pseudo-Kroeneker, são aplicadas as expansões considerando a primeira variável assim como as expressões anteriores, porém, a partir da segunda variável o funcionamento é diferente. As árvores resultantes da apli-

cação das expansões são divididas em duas subárvores, uma para cada nó filho do nó da raiz, que representa a operação XOR. Em seguida cada expansão é aplicada em cada subárvore em relação a segunda variável. As etapas de divisão em subárvores e de aplicação de expansões são feitas até a última variável.

As expressões resultantes são obtidas escolhendo uma das possíveis expressões geradas a partir da aplicação de expansões em cada subárvore, a cada variável. As expressões resultantes são avaliadas e a expressão com menos literais é escolhida.

4 RESULTADOS

Para a comparação dos métodos apresentados, foram utilizadas mil funções de 3, 4, 5 e 6 variáveis, onde cada função foi gerada aleatoriamente. As métricas utilizadas para comparar as expressões resultantes de cada método foram o número de literais, o custo de implementação e a profundidade lógica. Além disso, o tempo para executar as implementações dos métodos também será abordado.

Durante a análise dos resultados, foi observado que o número de literais de uma expressão é igual a soma das operações AND, OR e XOR da mesma mais 1. Desta forma, não existe nenhum ganho em mostrar informações referentes a estas duas métricas. A métrica custo de implementação é uma alternativa ao número de operações e o seu valor é a soma do número de operações AND e OR somada com o dobro do número de operações XOR. Este valor foi obtido se considerarmos que a tecnologia utilizada para a implementação destas portas lógicas seja CMOS, com o mesmo número de transistores na rede *pull-up* e na rede *pull-down*. Desta forma, é possível implementar uma porta AND, ou uma porta OR, utilizando 3 transistores NMOS e 3 transistores PMOS, totalizando 6 transistores, e uma porta XOR utilizando 6 transistores NMOS e 6 transistores PMOS, totalizando 12 transistores.

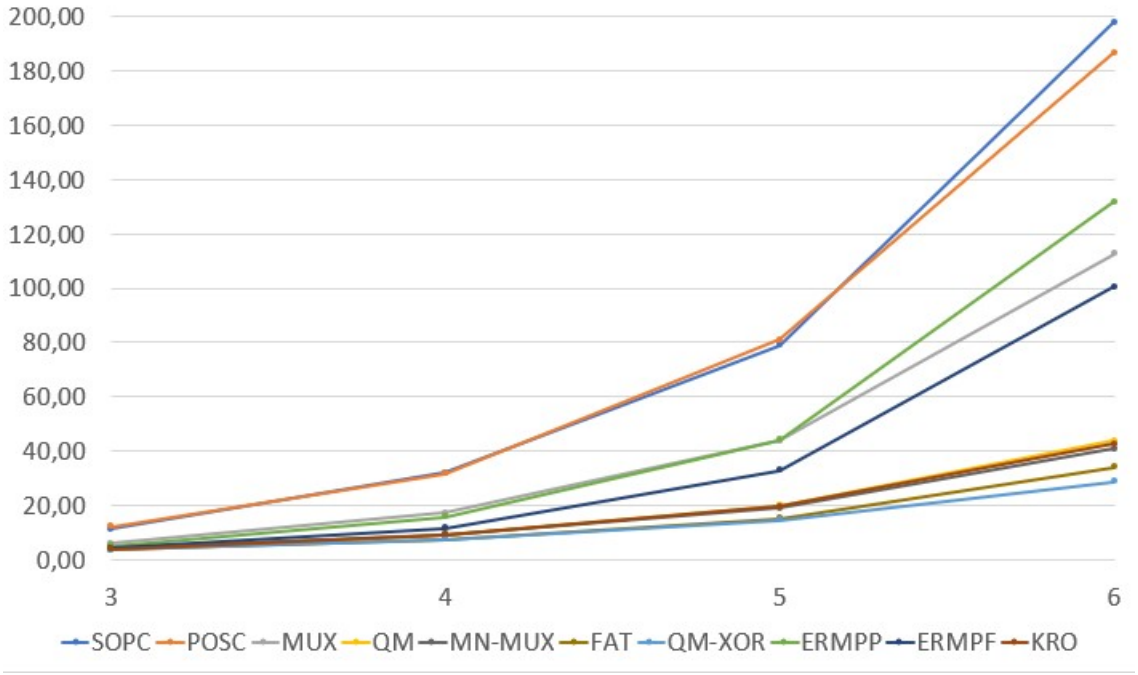
O método para obtenção de expressões Pseudo Reed-Muller só pode ser aplicado em funções de 3 e 4 variáveis e o método para obtenção de expressões Pseudo Kronecker só pode ser aplicado em funções de até 3 variáveis. Por este motivo, os resultados do primeiro método só aparecerão na Tabela 4.1 e na Tabela 4.2 e do segundo método só aparecerão na Tabela 4.1.

Os métodos suas abreviações que estão presentes nos gráficos e nas tabelas deste capítulo são as seguintes: SOP canônica (SOPC), POS canônica (POSC), síntese de dois níveis utilizando a lógica de MUX (MUX), Quine-McCluskey(QM), síntese multinível utilizando lógica de MUX (MN-MUX), fatoração de funções Booleanas (FAT), extensão do algoritmo de Quine-McCluskey para utilizzar operações XOR (QM-XOR), Expressão Reed-Muller de Polaridade Positiva (ERMPP), Expressão Reed-Muller de Polaridade Fixa (ERMPF), Expressão Kronecker (KRO), Expressão Pseudo Reed-Muller (EPSDRM) e Expressão Pseudo Kronecker (EPSDK).

O gráfico presente na Figura 4.1 apresenta o número médio de literais de cada método em relação ao número de variáveis das funções de entrada. Todos os métodos tendem a ter um crescimento exponencial, acompanhando o número de linhas das tabelas

verdade de entrada. Os dois métodos que resultam em expressões canônicas apresentam o maior número de literais. Este comportamento era esperado, pois estes métodos criam um cubo, ou uma clausula, para cada mintermo, ou maxtermo, sem executar nenhuma simplificação. A síntese dois níveis baseado em MUX e as expressões Reed-Muller de polaridade positiva e polaridade fixa apresentam um número de literais intermediário. Os cinco métodos restantes apresentam um comportamento bastante próximo, com a extensão de Quine-McCluskey apresentando o menor número médio de literais, seguido pela fatoração de funções Booleanas.

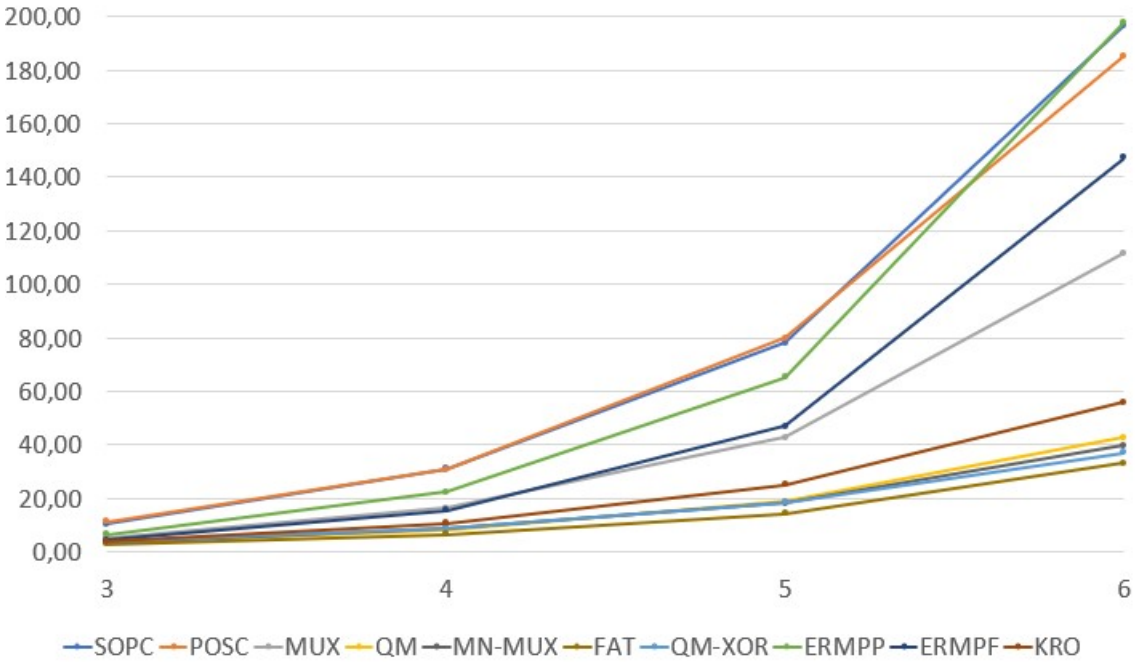
Figura 4.1: Gráfico comparando o número médio de literais por número de variáveis das funções de entrada.



No gráfico da Figura 4.2 são mostrados os valores para o custo de implementação médio de cada método em relação ao número de variáveis das funções da entrada. Como dito anteriormente, o soma das operações é muito próxima do número de literais, logo, o peso das operações XOR na soma é dobrado para ser mais próximo ao custo real de usar cada operação. Em relação ao gráfico da Figura 4.1, os métodos que não utilizam operações XOR tem basicamente o mesmo comportamento neste gráfico, enquanto os métodos que utilizam operações XOR são "penalizadas"por isso. As duas mudanças mais significativas em relação ao gráfico anterior são que a expressão Reed-Muller de polaridade positiva apresenta o maior custo de implementação, junto com a POS canônica, e a fatoração de funções Booleanas apresenta o menor custo de implementação.

O gráfico mostrado na Figura 4.3 apresenta a profundidade lógica média de cada

Figura 4.2: Gráfico comparando o custo de implementação médio por número de variáveis das funções de entrada.

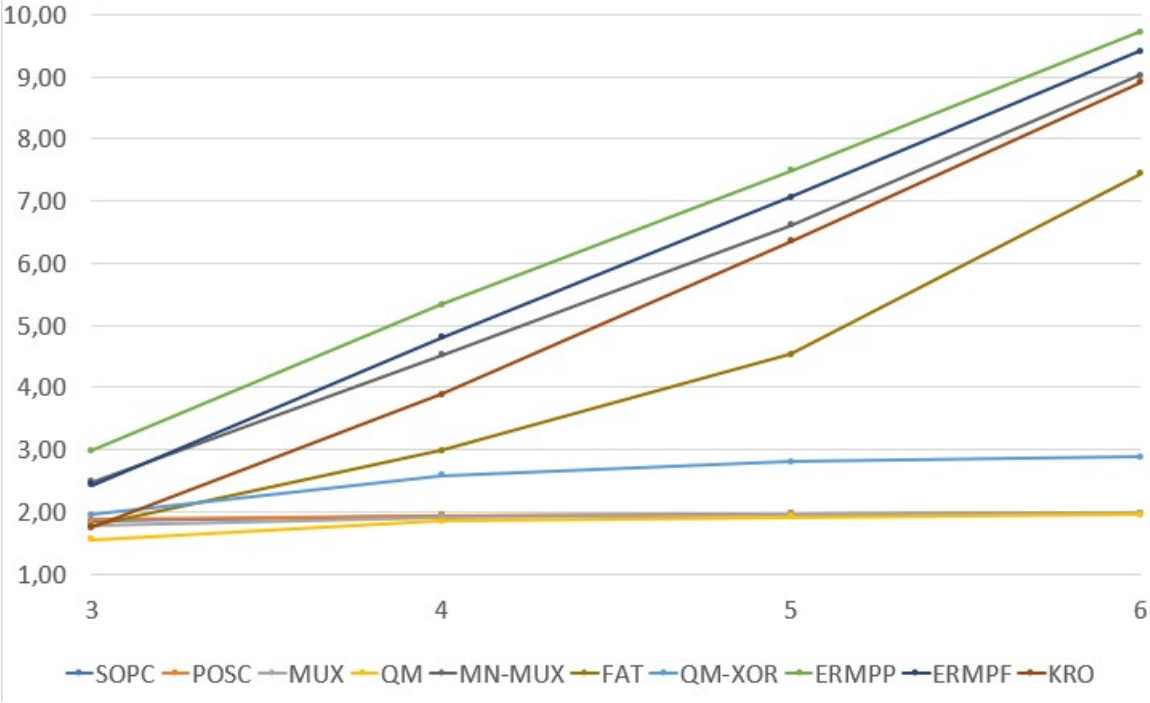


um dos métodos em relação ao número de variáveis nas funções de entrada. Os métodos para obter expressões canônicas, síntese de dois níveis utilizando MUX e o algoritmo de Quine-McCluskey são conhecidamente métodos com profundidade lógica igual a 2, e o gráfico confirma esta característica. Os métodos baseados em aplicação de expansões e o método para síntese multinível utilizando lógica de MUX apresentam as maiores profundidades lógicas. Ainda considerando os métodos de síntese multinível, o que apresenta menor profundidade é a fatoração de funções Booleanas. Apesar da expansão do algoritmo de Quine-McCluskey não ser uma síntese dois níveis, ela também não é uma síntese multinível. Apesar de não ser uma nomenclatura comum, este método poderia ser considerado uma síntese três níveis.

Para ser possível enxergar em um gráfico todos os valores relacionado ao tempo de execução com detalhes, os métodos foram divididos em três grupos: o primeiro grupo contém os métodos que apresentam o tempo de execução médio para 6 variáveis menor que 1 segundo, o segundo grupo contém os métodos que apresentam o tempo de execução médio para 6 variáveis maior que 1 segundo e menor que 100 segundos e o terceiro grupo contém os métodos restantes. Além disso, alguns métodos estão em mais de um grupo, mesmo que o seu tempo seja relativo a somente um deles, para que possa se ter um referencial do gráfico de um grupo em relação ao gráfico do outro grupo.

O primeiro grupo contém os métodos para obtenção da SOP canônica, da POS

Figura 4.3: Gráfico comparando profundidade lógica de cada método por número de variáveis das funções de entrada.



canônica e da expressão Reed-Muller de polaridade positiva, o método de síntese de dois níveis e multinível utilizando a lógica de MUX. O segundo grupo contém os métodos para obtenção da expressão Reed-Muller de polaridade positiva e Kronecker, o método de síntese multinível utilizando a lógica de MUX e a Fatoração de funções Booleanas. O terceiro grupo contém o método para obtenção da expressão Kronecker e os métodos de Quine-McCluskey original e sua expansão para utilização de funções XOR.

Os tempos totais de execução, em segundos, de cada grupo de métodos em relação ao número de variáveis nas 1000 funções de entrada são mostrados, respectivamente, na Figura 4.4, na Figura 4.5 e na Figura 4.6.

O gráfico apresentado na Figura 4.7 mostra o tempo total de execução, em segundos, em escala logarítmica, de todos os métodos em relação ao número de variáveis das funções de entrada. Nesta imagem é possível comparar todos os métodos em um só gráfico, podendo notar-se que a maioria dos métodos possuem um comportamento linear neste gráfico, significando um comportamento exponencial em um gráfico com escala linear. As exceções são os dois métodos baseados no algoritmo de Quine-McCluskey. Isto acontece porque, além do tamanho do problema dobrar a cada variável, o algoritmo para resolução da tabela de cobertura possui não possui uma complexidade polinomial, pois é um problema conhecidamente NP-completo.

Figura 4.4: Gráfico comparando o tempo de execução médio do primeiro grupo de métodos por número de variáveis das funções de entrada.

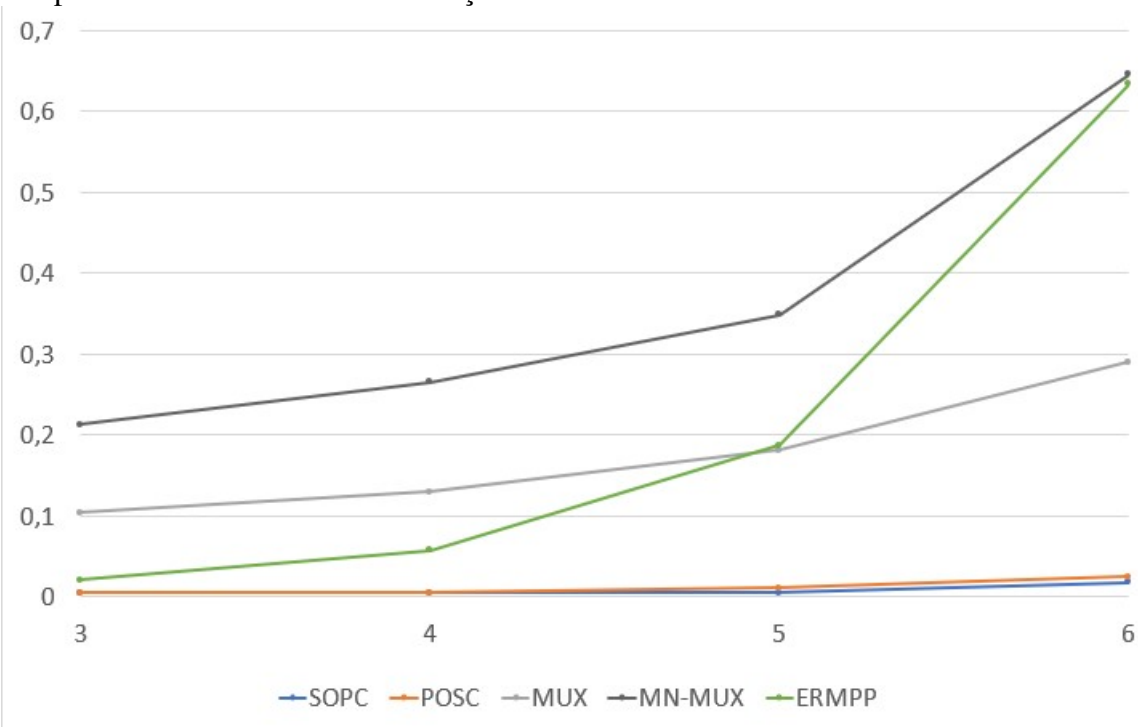


Figura 4.5: Gráfico comparando o tempo de execução médio do segundo grupo de métodos por número de variáveis das funções de entrada.

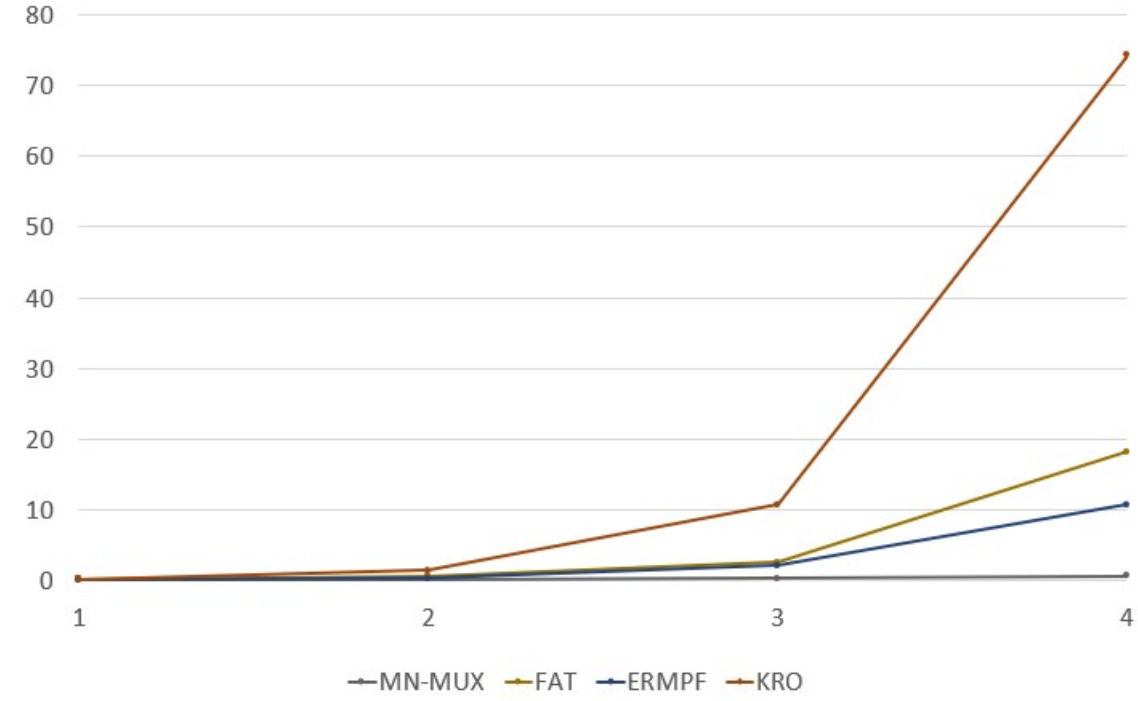


Figura 4.6: Gráfico comparando o tempo de execução médio o terceiro grupo de métodos por número de variáveis das funções de entrada.

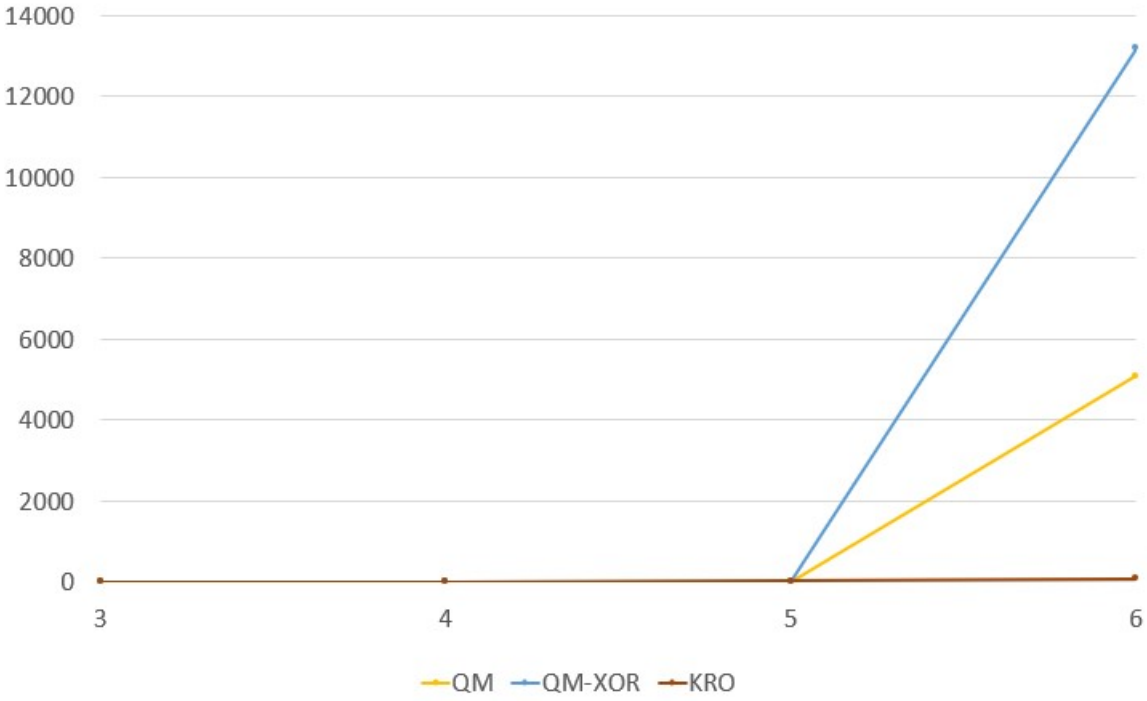
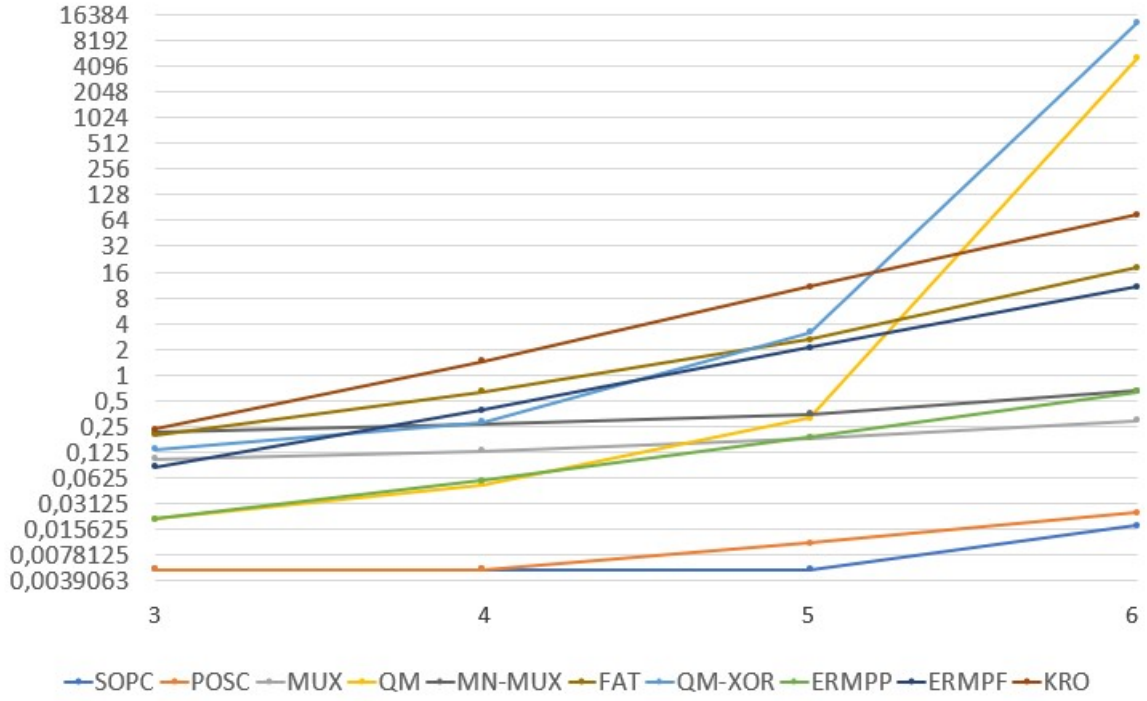


Figura 4.7: Gráfico comparando o tempo de execução de todos os métodos, utilizando escala logarítmica, por número de variáveis de entrada.



A Tabela 4.1 e a Tabela 4.2 demostram com detalhe os dados obtidos para mil funções de 3 e 4 variáveis. O principal objetivo da apresentação destas tabelas é comparar os métodos para obter expressões Pseudo Reed-Muller e Pseudo Kronecker com os demais métodos, que já foram comparados entre si. Para 3 variáveis, as expressões Pseudo Kronecker e Pseudo Reed-Muller apresentam os melhores resultados para número de literais e profundidade lógica entre todos os métodos. Para 4 variáveis, as expressões Pseudo Reed-Muller apresentam o terceiro menor número de literais e a menor profundidade lógica entre os métodos multinível.

Tabela 4.1: Mil funções aleatórias de 3 variáveis.

Método	**Literais**	**AND**	**OR**	**XOR**	**Profundidade**	**Tempo (s)**
SOPC	11.63	7.76	2.88	0.00	1.86	0.005
POSC	12.37	3.12	8.24	0.00	1.88	0.005
MUX	6.27	3.73	1.54	0.00	1.78	0.104
QM	4.19	2.04	1.14	0.00	1.56	0.020
MN-MUX	4.66	2.50	1.16	0.00	2.48	0.213
FAT	3.89	1.65	1.24	0.00	1.77	0.197
QMXOR	3.88	1.16	0.80	0.92	1.96	0.135
ERMPP	5.49	1.93	0.41	2.15	2.98	0.020
ERMPF	4.35	1.36	0.70	1.29	2.43	0.083
KRO	4.06	2.26	0.06	0.74	1.74	0.228
EPSDRM	3.65	1.38	0.45	0.83	1.54	0.301
PSDKRO	3.63	1.43	0.42	0.78	1.54	0.410

Tabela 4.2: Mil funções aleatórias de 4 variáveis.

Método	**Literais**	**AND**	**OR**	**XOR**	**Profundidade**	**Tempo (s)**
SOPC	32.02	24.01	7.00	0.00	1.94	0.005
POSC	31.98	7.00	23.99	0.00	1.94	0.005
MUX	17.37	12.29	4.07	0.00	1.92	0.129
QM	9.33	5.75	2.58	0.00	1.85	0.052
MN-MUX	9.60	5.77	2.84	0.00	4.52	0.265
FAT	7.72	3.47	3.25	0.00	2.99	0.634
QMXOR	7.77	2.92	1.62	2.24	2.59	0.280
ERMPP	15.96	6.23	1.21	7.53	5.33	0.057
ERMPF	11.82	4.22	1.83	4.77	4.81	0.384
KRO	9.38	5.62	0.26	2.50	3.89	1.445
EPSDRM	8.92	4.84	0.70	2.38	2.04	126.282

5 CONCLUSÃO

No primeiro capítulo é apresentada uma introdução a este trabalho. Neste capítulo, foram apresentadas as motivações da utilização e desenvolvimento de ferramentas de EDA e quais são as etapas básicas destas ferramentas. Foi dada uma introdução a métodos de síntese de funções Booleanas e como estes podem ser inseridos na etapa de Síntese Lógica. Para finalizar este capítulo, foram apresentadas as motivações e objetivos deste trabalho e a organização deste texto.

No segundo capítulo são apresentados diversos métodos de síntese de funções Booleanas e diversas definições que eram necessárias para a explicação dos mesmo. No total, foram apresentados cinco métodos que utilizam operações AND e OR, onde três delas resultam em expressões dois níveis e as outras duas em expressões multinível, e dois métodos que apresentam operações AND, OR e XOR, onde um destes métodos resulta em expressões de três níveis e o outro, que é dividido em cinco diferentes expressões, que apresenta expressões multinível.

No terceiro capitulo é mostrado como cada método foi implementado, considerando quais as estrutura de dados foram utilizadas e algumas decisões que foram tomadas para as implementações.

No quarto capítulo são apresentados os dados referentes a diferentes métricas, onde estas métricas foram número de literais, custo de implementação, profundidade lógica e tempo de execução. As seguintes conclusões podem ser tomadas, considerando as características dos métodos apresentados.

Os métodos para obter expressões canônicas apresentam o maior número de literais, como esperado, e apresentam os métodos com o menor tempo de execução.

Os métodos que utilizam a lógica de MUX para seu desenvolvimento são bastante rápidos e apresentam resultados interessantes. O método voltado para síntese dois níveis apresenta uma diminuição no número de literais interessante em relação as expressões canônicas, mas ficam longe do apresentado no algoritmo de Quine-McCluskey. O método multinível apresenta resultados muito bons em relação ao número de literais, somente a fatoração de funções Booleanas e a extensão do algoritmo de Quine-McCluskey apresentam resultados melhores, mas apresenta a terceira maior profundidade lógica entre os métodos analisados. Estes métodos são bastante interessantes para mostrar a utilização de outros blocos combinacionais para orientar a síntese de funções Booleanas.

Os métodos que resultam em expressões com o número ótimo de cubos, como o

algoritmo de Quine-McCluskey e sua expansão para integrar a operação XOR ao método apresentam, respectivamente, o menor número de literais médio considerando os métodos para síntese dois níveis e entre todos os métodos. Porém, os tempos de execução destes dois métodos são muito maiores que o tempo dos outro métodos para funções de 6 variáveis. Desta forma, podem ser opções bastantes indicadas de métodos com um número fixo de níveis para funções de até 5 variáveis.

O método de fatoração de funções Booleanas apresentam o segundo menor número de literais e o menor custo de implementação entre todos os métodos. Além disso, apresenta a menor profundidade lógica entre os métodos multinível e um tempo de execução intermediário entre os métodos. Este método, para um propósito geral, aparenta ser o mais indicado para sintetizar um função Booleana qualquer.

Dentro das expressões obtidas através da aplicação de expansões, a classe de expressões Pseudo Kronecker aparenta ter resultados mais interessante, mas a sua obtenção a partir da aplicação de expansões não apresenta um tempo de execução factível para ser utilizado de forma geral. Logo, sua obtenção através de outros métodos pode ser interessante (SASAO, 1993a).

REFERÊNCIAS

BROWN, S.; VRANESIC, Z. **Fundamentals of Digital Logic With Vhdl Design**. [S.l.]: McGraw-Hill Education, 2008.

GOLUMBIC, M.; MINTZ, A.; RODICS, U. Factoring and recognition of read-once functions using cographs and normality. In: 38TH DESIGN AUTOMATION CONFERENCE, 2001, Las Vegas, NV, USA. [S.l.], 2005.

HURLEY, P. J. **A Concise Introduction to Logic**. [S.l.]: Wadsworth Publishing, 2011.

KLOCK, C. et al. Karma: A didactic tool for two-level logic synthesis. In: IEEE INTERNATIONAL CONFERENCE ON MICROELECTRONIC SYSTEMS EDUCATION, 2007, San Diego, CA, USA. [S.l.], 2007.

MCCLUSKEY, E. J. Minimization of boolean functions. **The Bell System Technical Journal**, v. 35, n. 6, p. 1417–1444, Nov 1956.

MICHELI, G. D. **Synthesis and Optimization of Digital Circuits**. [S.l.]: McGraw-Hill Higher Education, 1994.

MINTZ, A.; GOLUMBIC, M. Factoring boolean functions using graph partitioning. **Discrete Applied Mathematics**, v. 149, p. 131 – 153, Aug 2005.

MOORE, G. E. Cramming more components onto integrated circuits. **Eletronics**, v. 38, n. 8, 1965.

RUDELL, R. L.; SANGIOVANNI-VINCENTELLI, A. Logic synthesis for vlsi design. (PhD Thesis). 1989.

SASAO, T. And-exor expressions and their optimization. In: **Logic Synthesis and Optimization**. [S.l.]: Springer US, 1993. p. 287–312.

SASAO, T. Logic synthesis with exor gates. In: **Logic Synthesis and Optimization**. [S.l.]: Springer US, 1993. p. 259–285.

TURTON, B. Extending quine-mccluskey for exclusive-or logic synthesis. **IEEE Transactions on Education**, v. 39, n. 1, p. 81–85, Feb 1996.