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

MARIA FLÁVIA BORRAJO TONDO

Geração de memórias de programa em Verilog compatível com a plataforma Pitanga

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

Orientador: Prof. Dr. André Inácio Reis
Co-orientador: Leonardo Droves Silveira

Porto Alegre
2023

“Tudo foi calculado,
exceto como viver.”
— JEAN PAUL SARTRE

AGRADECIMENTOS

Queria deixar registrado meu agradecimento especial a todas as pessoas que tornaram este TCC possível. Cada um de vocês fez a diferença e foi fundamental nessa etapa crucial da minha vida acadêmica.

Primeiramente, agradeço aos meus orientadores André Reis e Leonardo Droves Silveira, pelo apoio, pela orientação e pela compreensão ao longo desse processo. Suas dicas e conselhos foram essenciais para dar forma a este trabalho.

Ao Instituto de Informática da UFRGS, agradeço por proporcionar um ambiente acadêmico propício ao aprendizado, à pesquisa e ao desenvolvimento, que foi imprescindível para a minha formação e realização deste trabalho.

Aos meus amigos e colegas de curso, pela amizade, pela parceria e pelo incentivo durante todo o período de estudo. É bom saber que muitos de vocês irão permanecer ao meu lado depois que este ciclo estiver encerrado.

À minha família, pelo cuidado, apoio constante e pela compreensão. Sem vocês, nada disso teria sido possível.

Por fim, um agradecimento a todos que, de alguma maneira, apoiaram e contribuíram com este trabalho, seja com conselhos, materiais ou simplesmente estando presentes durante esta jornada. Cada palavra de incentivo e gesto de apoio fez toda a diferença!

RESUMO

Este estudo compara o uso de Diagramas de Decisão Binária (BDDs) e técnicas de redução de circuitos combinacionais na geração de descrições de hardware em Verilog, a partir da memória de processadores didáticos. Os BDDs oferecem representações eficientes de funções Booleanas, enquanto as técnicas de redução simplificam circuitos. Explorou-se o processador Neander para aplicação desses métodos e geração de código Verilog, sendo feita a avaliação no âmbito da otimização de circuitos combinacionais na plataforma Pitanga/inPlace.

Palavras-chave: Verilog. Processadores didáticos. Diagrama de decisão binária. Circuito combinacional.

Generation of program memories using Verilog compatible with the Pitanga platform

ABSTRACT

This study compares the use of Binary Decision Diagrams (BDDs) and reduction techniques of combinational circuits in generating hardware descriptions in Verilog, from memory of didactic processors. BDDs offer efficient representations of Boolean functions, while reduction techniques simplify circuits. We explored the Neander processor to apply these methods and generate Verilog code, with the evaluation being carried out in the context of optimizing combinational circuits on the Pitanga/inPlace platform.

Keywords: Verilog, Didactic processors, Binary decision diagram, Combinational circuit.

LISTA DE ABREVIATURAS E SIGLAS

BDD	Binary Decision Diagram
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Arrays
ISA	Industry Standard Architecture
PLA	Programmable Logic Array
RAM	Random Access Memory
ROM	Read-Only Memory
VLSI	Very Large Scale Integration

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Nos domínios da ciência da computação e da engenharia de sistemas digitais, a eficiência e otimização dos circuitos combinacionais são elementos cruciais para o desenvolvimento de sistemas eletrônicos avançados e complexos. A representação e manipulação eficazes de funções Booleanas são fundamentais para alcançar esse objetivo.

Este estudo se propõe a explorar e comparar dois enfoques fundamentais: a utilização de Diagramas de Decisão Binária (BDDs) e técnicas de redução de circuitos combinacionais, com o intuito de gerar descrições de *hardware* em Verilog, utilizando como estudo de caso e arquivo de entrada a memória de processadores didáticos. Tais processadores, embora didáticos, servem como modelos significativos para a compreensão e o desenvolvimento de sistemas computacionais.

Os BDDs, como uma estrutura de dados bem estabelecida, têm demonstrado eficiência na representação compacta de funções booleanas. Sua aplicação na criação de modelos de circuitos combinacionais pode proporcionar uma representação eficaz e facilitar a análise e a otimização desses circuitos. Por outro lado, as reduções de circuitos combinacionais buscam simplificar as expressões booleanas, reduzindo o número de portas lógicas e, consequentemente, a complexidade dos circuitos.

Além disso, a geração de código Verilog é um passo crucial no projeto e implementação de sistemas digitais. Através desta geração, as funcionalidades dos circuitos são traduzidas em descrições compreensíveis e implementáveis em *hardware*.

O próximo capítulo apresentará uma revisão de conceitos acerca dos processadores, enfatizando o processador didático Neander, da composição de um circuito combinacional, sobre BDDs e sua utilização em técnicas de simplificações de circuitos e a estrutura da linguagem Verilog. O terceiro capítulo abordará os detalhes da implementação, desde o tratamento de dados do arquivo de entrada até a geração do arquivo de saída, em Verilog. Após, o quarto capítulo discorre e analisa os resultados obtidos. E, por fim, o último capítulo sintetiza o que este trabalho conseguiu explorar e suas limitações, assim como, apresenta as possibilidades de trabalhos futuros.

2 REVISÃO DE CONCEITOS

2.1 Processadores

Um processador, também conhecido como CPU (*Central Processing Unit*), é responsável por executar instruções e manipular dados em um formato binário. É o componente central em um sistema de computação, interpretando e executando as operações necessárias para que um computador funcione corretamente.

Sua arquitetura é complexa, composta por diversas unidades funcionais, incluindo a Unidade de Controle e a Unidade Lógica e Aritmética. A Unidade de Controle é responsável por coordenar a execução de instruções, controlar a transferência de dados e gerenciar a operação global do processador. Já a Unidade Lógica e Aritmética executa operações aritméticas (como adição e subtração) e lógicas (como AND, OR e NOT) (HENNESSY, 2012).

O processador opera em um ciclo de busca, interpretação, execução e escrita, conhecido como ciclo de instrução. Durante o ciclo de busca, o processador busca a próxima instrução na memória principal. Na etapa de interpretação, a instrução é decodificada para entender qual operação precisa ser executada. A execução envolve a realização efetiva da operação indicada pela instrução. Por fim, na etapa de escrita, os resultados da operação são armazenados na memória ou em registradores internos.

Para uma boa eficiência operacional, os processadores possuem memória cache e registradores. A memória cache é frequentemente usada para acelerar o acesso aos dados e instruções mais usados. E os registradores internos, que armazenam temporariamente dados e instruções, são cruciais para a eficiência operacional do processador.

2.2 Processador hipotético Neander

O processador hipotético Neander é um modelo teórico e simplificado de computador, frequentemente usado para fins didáticos em cursos introdutórios de arquitetura e organização de computadores. Ao contrário dos computadores reais, que possuem diversas instruções e componentes, o Neander foi projetado para ter um conjunto limitado, tornando mais fácil para os alunos compreenderem os conceitos básicos da computação, como o ciclo de instrução, a manipulação de dados e a execução de operações (WEBER, 2012).

A memória do Neander é composta por 256 endereços. Em cada endereço, é possível armazenar um *byte* de dados, variando de -128 a +127, já que ele funciona com a representação em complemento de dois. Ele possui um conjunto de instruções básicas, totalizando 11, como por exemplo: operação de soma, operações lógicas, de carregamento e armazenamento de dados e instruções de desvio. Todas as instruções podem ser vistas na Tabela 2.1. Além disso, ele possui um Acumulador (AC) de 8 *bits*, um apontador de programa (PC) e um registrador de estado que possui dois códigos de condição para verificar se o valor do acumulador é negativo ou zero (WEBER, 2012).

Tabela 2.1: Conjunto de instruções do Neander.

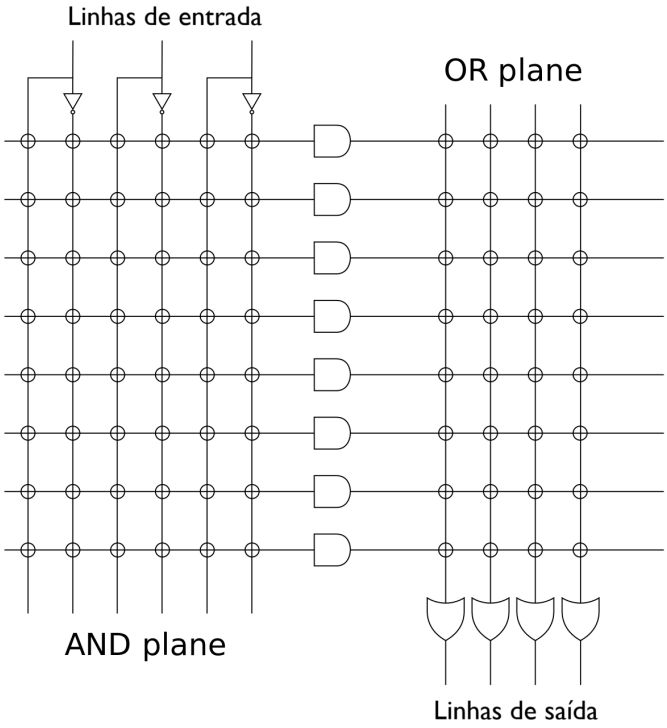
Código (Decimal)	**Instrução**	**Operação**
0	NOP	Nenhuma operação
16	STA end	Armazena acumulador - (*store*)
32	LDA end	Carrega acumulador - (*load*)
48	ADD end	Soma
64	OR end	"ou"lógico
80	AND end	"e"lógico
96	NOT	Inverte (complementa) acumulador
128	JMP end	Desvio incondicional - (*jump*)
144	JN end	Desvio condicional - (*jump on negative*)
160	JZ end	Desvio condicional - (*jump on zero*)
240	HLT	Término de execução - (*halt*)

2.3 *Programmable Logic Array* (PLA)

Embora este trabalho não implemente PLA, é fundamental reconhecer sua ampla utilização no campo da eletrônica digital e circuitos integrados, tornando sua revisão relevante. O PLA possui uma arquitetura que se baseia em uma matriz bidimensional de interconexões de entrada e saída, acompanhada por uma matriz de elementos lógicos programáveis. A estrutura é fundamental para a flexibilidade e a capacidade de personalização desse componente.

A matriz de interconexões, como mostra a Figura 2.1, é composta por linhas de entrada (horizontais) e colunas de saída (verticais). As entradas estão conectadas às linhas horizontais, chamadas de "linhas de entrada", enquanto as saídas estão interligadas às colunas verticais, denominadas "linhas de saída". Cada ponto de interseção entre uma linha de entrada e uma linha de saída representa um local onde uma função lógica específica pode ser implementada.

Figura 2.1: Exemplo da estrutura de um PLA.



Por sua vez, a matriz de elementos lógicos programáveis é composta por elementos que são, na maioria das vezes, portas AND e OR. Esses elementos permitem a configuração da função lógica desejada. Geralmente, cada elemento é uma porta AND seguida de uma porta OR, proporcionando flexibilidade para criar diversas funções lógicas a partir das entradas disponíveis.

A flexibilidade oferecida pela estrutura do PLA permite a programação das conexões entre as linhas de entrada, os elementos lógicos programáveis e as linhas de saída. Isso possibilita a personalização das funções lógicas implementadas no dispositivo, tornando-o uma ferramenta valiosa para o design e a implementação de circuitos digitais complexos e adaptáveis, atendendo a uma variedade de necessidades no campo da eletrônica.

2.4 Circuito combinacional

Um circuito combinacional é formado por uma interconexão de portas lógicas que transformam informações binárias de entrada em saídas desejadas. Eles são caracterizados pela sua capacidade de produzir uma saída baseada apenas no seu estado de entrada atual, sem levar em consideração seus estados anteriores. Isso contrasta com os circuitos

sequenciais, onde a saída é determinada tanto pelas entradas atuais quanto por estados anteriores, permitindo-lhes ter uma "memória"de eventos passados.

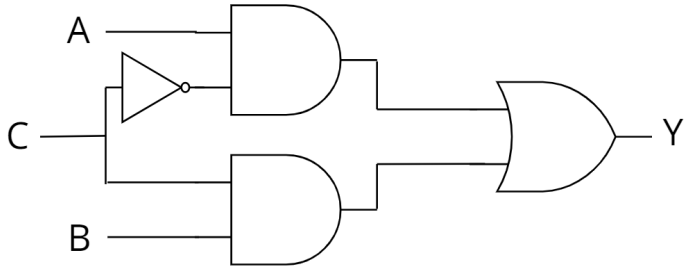
Existem vários circuitos combinacionais conhecidos usados no *design* de sistemas digitais, como somadores, subtratores, comparadores, decodificadores, codificadores e multiplexadores. Estes circuitos, disponíveis como componentes padrão em circuitos integrados, são essenciais no *design* de sistemas digitais e também são usados em circuitos VLSI complexos. Eles podem ser especificados por uma tabela-verdade, que lista os valores de saída para cada combinação de variáveis de entrada, ou por funções booleanas (MANO; MICHAEL et al., 2013).

2.4.1 Multiplexador 2x1

O multiplexador 2x1 é um componente fundamental em eletrônica digital usado para direcionar um de dois sinais de entrada para uma única saída, com base em um sinal de controle. Esse tipo de multiplexador possui duas entradas de dados (A e B), uma entrada de controle (geralmente chamada de seletor) e uma saída.

Quando o sinal de controle está em um estado específico (0 ou 1), o multiplexador direciona a entrada correspondente (A ou B) para a saída. Se o sinal de controle for 0, a entrada A será transmitida para a saída; se for 1, a entrada B será transmitida, como mostra a Figura 2.2.

Figura 2.2: Multiplexador 2x1.



A expressão booleana para a saída Y de um multiplexador 2x1 é:

$$Y = A * !C + C * B$$

onde A e B são as entradas de dados, C é o sinal de controle, !C é a negação de C, "$*$"representa a operação lógica AND e "+"representa a operação lógica OR.

Os multiplexadores 2x1 são amplamente utilizados em circuitos digitais para seleção de dados, roteamento de sinais e implementação de diversas funções lógicas. Podem ser combinados para criar multiplexadores de maior ordem, como os de 4x1 ou 8x1.

Essencialmente, o multiplexador 2x1 é uma ferramenta valiosa para projetar circuitos que precisam alternar entre diferentes fontes de dados de maneira controlada, o que o torna um elemento-chave na construção de sistemas digitais eficientes e flexíveis.

2.5 *Binary Decision Diagram* (BDD)

Um BDD (Binary Decision Diagram ou Diagrama de Decisão Binária) é uma representação gráfica eficiente de funções booleanas, frequentemente usada em projetos de circuitos digitais e verificação de *hardware*. Ele consiste em nodos, arcos e terminais 0 (T0) ou 1 (T1), onde os nodos representam variáveis de controle e os arcos indicam escolhas (BRYANT, 1986). Neste trabalho, usaremos a notação de arco sólido representando o filho 1 e arco tracejado representando o filho 0 do nodo.

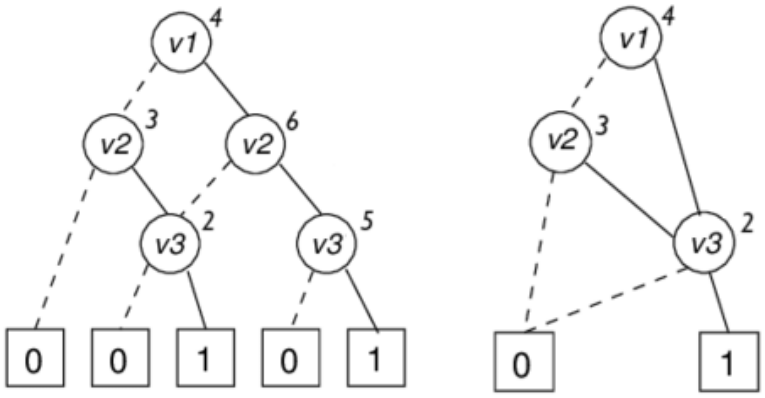
2.5.1 *Reduced Ordered Binary Decision Diagram* (ROBDD)

Os ROBDDs, ou Diagramas Binários de Decisão Reduzidos (do inglês, Reduced Ordered Binary Decision Diagrams), são uma representação gráfica eficiente e poderosa para manipulação de funções booleanas. Em comparação com outras representações clássicas, como tabelas-verdade, mapas de Karnaugh e formas canônicas de soma de produtos, os ROBDDs oferecem vantagens significativas, representando funções booleanas como grafos direcionados acíclicos, com restrições particulares na ordem das variáveis de decisão nos vértices (KNUTH, 2011). Isso permite o desenvolvimento de algoritmos eficientes para manipular as representações de maneira otimizada.

As principais vantagens dos ROBDDs incluem oferecer uma representação canônica única para uma função, independentemente da ordem das variáveis. Essa representação também elimina a redundância estrutural, resultando em uma representação mais compacta. Operações como AND, OR, NOT são otimizadas no ROBDD, proporcionando eficiência na manipulação da função booleana.

Na Figura 2.3, é possível notar que, no processo de criação de um ROBDD, um novo nodo é criado somente se já não existe um nodo que implementa a função desejada (v4 e v5 representam a mesma função). Isso previne a duplicação de informações, mantendo-se canônica. Além disso, no ROBDD, um nodo é redundante se seus dois sucessores apontam para o mesmo nodo (v3), sendo omitido do diagrama final.

Figura 2.3: Diagramas equivalentes.

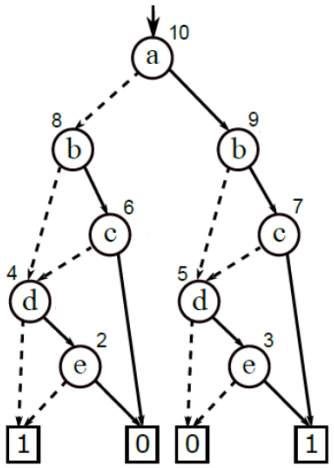


(a) BDD. (b) ROBDD.

2.5.2 Simplificações

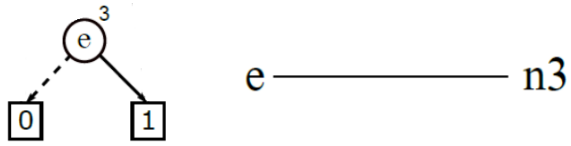
Neste estudo, serão utilizadas as simplificações de multiplexadores propostas pelo grupo LogiCS (BRANDÃO et al., 2022a) para gerar circuitos a partir de BDDs. Dada a Figura 2.4, tem-se 9 nodos, onde cada um é representado, inicialmente, por um multiplexador.

Figura 2.4: BDD completo (BRANDÃO et al., 2022a).



A primeira redução se aplica quando o arco 0 aponta para T0 e o arco 1 aponta para T1 (Figura 2.5). Deste modo, podemos simplificar o multiplexador a um fio, onde a saída, neste caso *n3*, é o valor da variável de controle.

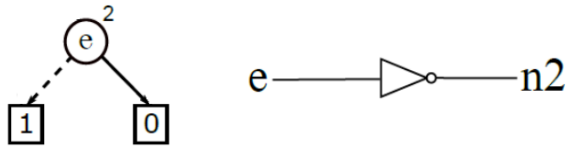
Figura 2.5: Exemplo da primeira simplificação de um multiplexador a partir de um nodo de BDD (BRANDÃO et al., 2022a).



(a) Nodo 3. (b) Redução 1.

Já a segunda redução se aplica quando o arco 0 aponta para T1 e o arco 1 aponta para T0 (Figura 2.6). Assim, podemos simplificar o multiplexador a um inversor, onde a saída, neste caso *n2*, é o inverso do valor da variável de controle.

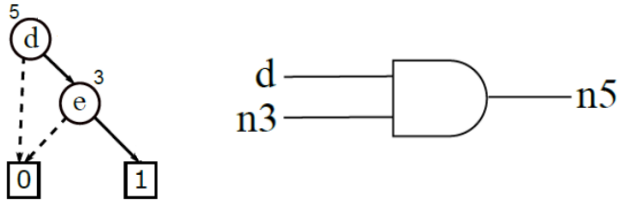
Figura 2.6: Exemplo da segunda simplificação de um multiplexador a partir de um nodo de BDD (BRANDÃO et al., 2022a).



(a) Nodo 2. (b) Redução 2.

Na terceira redução, temos o arco 0 apontando para T0 (Figura 2.7). Com essa informação, podemos simplificar o multiplexador a uma porta AND, onde a saída, neste caso *n5*, é um produto da variável de controle com o nodo apontado pelo arco 1.

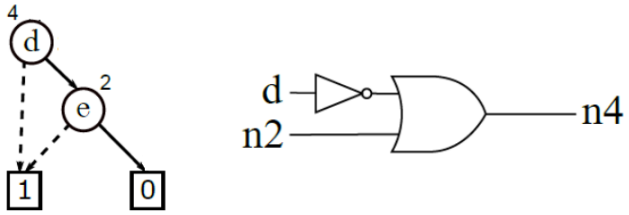
Figura 2.7: Exemplo da terceira simplificação de um multiplexador a partir de um nodo de BDD (BRANDÃO et al., 2022a).



(a) Nodo 5. (b) Redução 3.

Na quarta redução, agora, temos o arco 0 apontando para T1 (Figura 2.8). Sabendo disso, podemos simplificar o multiplexador a um inversor e uma porta OR, onde a saída, neste caso *n4*, é uma soma do inverso da variável de controle com o nodo apontado pelo arco 1.

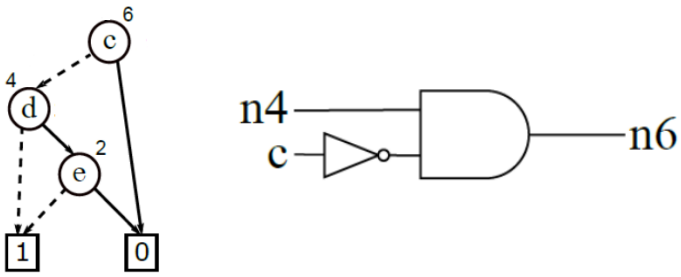
Figura 2.8: Exemplo da quarta simplificação de um multiplexador a partir de um nodo de BDD (BRANDÃO et al., 2022a).



(a) Nodo 4. (b) Redução 4.

A quinta redução se aplica quando o arco 1 aponta para T0 (Figura 2.9). Dessa forma, podemos simplificar o multiplexador a um inversor e uma porta AND, onde a saída, neste caso *n6*, é um produto do nodo apontado pelo arco 0 com o inverso da variável de controle.

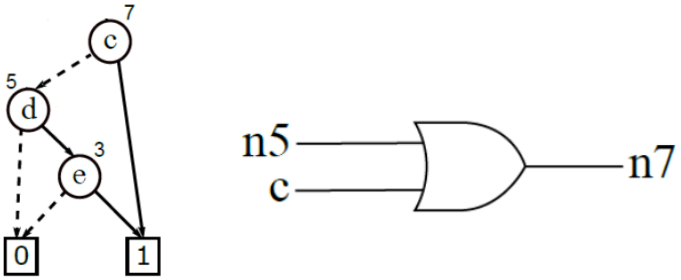
Figura 2.9: Exemplo da quinta simplificação de um multiplexador a partir de um nodo de BDD (BRANDÃO et al., 2022a).



(a) Nodo 6. (b) Redução 5.

Já a sexta redução se aplica quando o arco 1 aponta para T1 (Figura 2.10). Nesse sentido, podemos simplificar o multiplexador a uma porta OR, onde a saída, neste caso *n7*, é uma soma do nodo apontado pelo arco 0 com a variável de controle.

Figura 2.10: Exemplo da sexta simplificação de um multiplexador a partir de um nodo de BDD (BRANDÃO et al., 2022a).

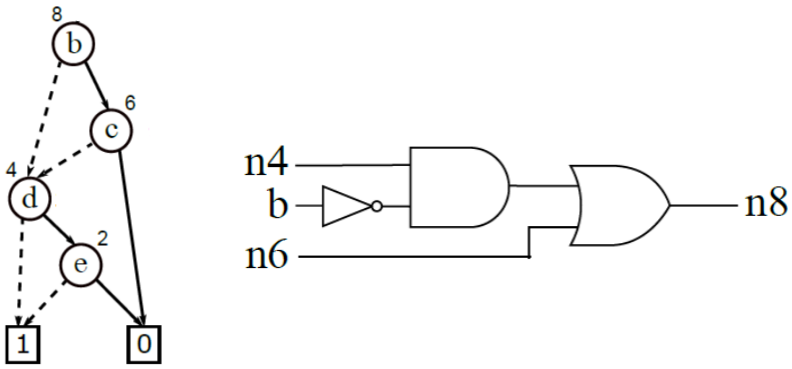


(a) Nodo 7. (b) Redução 6.

Na sétima redução, temos que a função do nodo é "*negative unate*", ou seja, sempre que o filho 1 for igual a 1, o filho 0 terá que ser 1 também (Figura 2.11). Consequente-

mente, podemos simplificar o multiplexador excluindo uma das portas AND. Com isso, a saída, neste caso *n8*, é o produto do nodo apontado pelo arco 0 com o inverso da variável de controle, somado com o nodo apontado pelo arco 1.

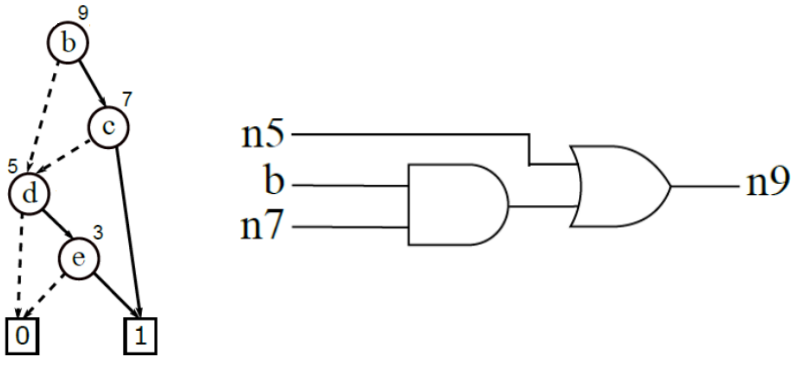
Figura 2.11: Exemplo da sétima simplificação de um multiplexador a partir de um nodo de BDD (BRANDÃO et al., 2022a).



(a) Nodo 8. (b) Redução 7.

Por fim, na oitava redução, temos que a função do nodo é "*positive unate*", ou seja, sempre que o filho 1 for igual a 0, o filho 0 terá que ser 0 também (Figura 2.12). Portanto, podemos simplificar o multiplexador excluindo uma das portas AND. Com isso, a saída, neste caso *n9*, é o produto da variável de controle com o nodo apontado pelo arco 1, somado com o nodo apontado pelo arco 0.

Figura 2.12: Exemplo da oitava simplificação de um multiplexador a partir de um nodo de BDD (BRANDÃO et al., 2022a).



(a) Nodo 9. (b) Redução 8.

Para os casos que não se encaixam em nenhuma das reduções, como o *n10* da Figura 2.4, segue a aplicação do caso geral e mantém-se o multiplexador na sua forma original.

2.6 Linguagem Verilog

Verilog é uma linguagem de descrição de *hardware* utilizada para descrever sistemas digitais em diversos níveis de abstração. Foi criado no final de 1983 como uma ferramenta proprietária de simulação e verificação e é uma linguagem que auxilia desde os estágios iniciais de *design*, com sua abstração comportamental, até as etapas de implementação, com suas abstrações estruturais, permitindo aos projetistas gerenciar a complexidade do *design* através de estruturas hierárquicas (THOMAS; MOORBY, 2008).

Atualmente, a linguagem Verilog é uma ferramenta fundamental na indústria de *design* de *hardware*. Com a rápida evolução da tecnologia e o surgimento de dispositivos inteligentes, sistemas embarcados e soluções de IoT, a necessidade de *hardware* personalizado e eficiente vem crescendo muito. Devido à sua versatilidade e capacidade de descrever *hardware* em níveis granulares, a linguagem tornou-se fundamental para desenvolver soluções inovadoras e competitivas. Assim, para profissionais e estudantes da área, o domínio do Verilog é importante, sendo uma habilidade fundamental para inovação e desenvolvimento no setor.

2.7 Relação com trabalhos do grupo LogiCS

Esta seção discute a relação deste TCC com os trabalhos do grupo LogiCS. LogiCS Lab, abreviação de Logic Circuit Synthesis Lab, é um grupo de pesquisa localizado na Universidade Federal do Rio Grande do Sul (UFRGS).

Este trabalho de conclusão tem suas raízes nos conceitos básicos de projeto de circuitos lógicos (ROSA et al., 2003; WAGNER; REIS; RIBAS, 2006) e síntese lógica (REIS; MATOS, 2018) (REIS; DRECHSLER, 2018), incluindo métodos adaptados para diferentes classes de funções lógicas (CORREIA; REIS, 2001), investigados pelo grupo LogiCS. A investigação contínua ao longo dos anos tem explorado diversas áreas, incluindo a geração automática de células lógicas para bibliotecas de células (TOGNI et al., 2002; MARTINS et al., 2015), mapeamento tecnológico (REIS et al., 1997; REIS, 1999; CORREIA; REIS, 2004; MARQUES et al., 2007), geração de circuitos a partir de BDDs (PERALTA et al., 2021a; BRANDÃO et al., 2022b; PERALTA et al., 2023a) e síntese de netlist de transistores para células lógicas (JUNIOR et al., 2006; da Silva; REIS; RIBAS, 2009; ROSA et al., 2007; CALLEGARO et al., 2010; BUTZEN et al., 2010a; BUTZEN et al., 2012; ROSA et al., 2009).

O grupo de pesquisa também propôs técnicas eficientes de síntese de redes de transistores para otimizar área (POLI et al., 2003), variabilidade (da Silva; REIS; RIBAS, 2009; BUTZEN et al., 2010a; BUTZEN et al., 2012) e potência (BUTZEN et al., 2010b). Alguns métodos propostos no laboratório LogiCS, como a síntese baseada em corte KL (MACHADO et al., 2012) e a síntese baseada em composição funcional (MARTINS; RIBAS; REIS, 2012), são usados em diferentes aplicações. Estes incluem o uso utilizado em novas tecnologias (NEUTZLING et al., 2013; MARRANGHELLO et al., 2015; NEUTZLING et al., 2015; NEUTZLING et al., 2018; NEUTZLING et al., 2019), circuitos robustos (GOMES et al., 2014; GOMES et al., 2015), e circuitos assíncronos (MOREIRA et al., 2014).

Embora este trabalho de conclusão não contribua com novos métodos de síntese lógica, ele absorve a experiência e a abordagem científica preservadas e aprimoradas ao longo da existência do laboratório LogiCS. Os trabalhos deste trabalho de conclusão baseiam-se no conhecimento prévio de estruturas de dados como Binary Decision Diagrams (BDDs). Esse conhecimento prévio construído ao longo dos anos foi combinado com uma proposta de projeto feita pela inPlace Design Automation. Este TCC inova então como um esforço cooperado, entre a academia e a indústria, a fim de entregar não apenas uma contribuição acadêmica, mas a prova de conceito para gerar a descrição Verilog de um hardware digital de memórias ROM de programa da máquina hipotética Neander. O verilog utilizado é compatível com a abordagem de placa virtual Pitanga (COSTA; SILVEIRA; REIS, 2022a; COSTA; SILVEIRA; REIS, 2023a) para prototipar circuitos digitais. Porém a parte de RAM não foi implementada no escopo deste Trabalho de Conclusão.

3 DETALHES DE IMPLEMENTAÇÃO

Após uma introdução mais aprofundada dos conceitos necessários para o entendimento do funcionamento desta ferramenta, aqui começa, de fato, a descrição da implementação desenvolvida para o trabalho proposto. A primeira seção abordará o tratamento de dados realizado a partir do arquivo de entrada. A segunda seção discute o processo de criação de BDDs para as funções definidas. A terceira trata acerca da construção de um circuito combinacional representando todos os nodos do diagrama. Finalmente, a quarta seção apresenta a estrutura do arquivo de saída com o programa traduzido para linguagem Verilog.

Como dito anteriormente, essa aplicação recebe um arquivo de memória contendo instruções que executam no processador hipotético Neander. Assim, para ajudar na contextualização de cada etapa da explicação, será utilizado o conjunto de instruções de entrada abaixo (Tabela 3.1):

Tabela 3.1: Exemplo de programa executado no processador hipotético Neander.

Endereço	**Dado**	**Mnemônico**	**Dado (Hexa)**	**Dado (Binário)**
0	32	LDA 7	20	00100000
1	7	-	07	00000111
2	48	ADD 7	30	00110000
3	7	-	07	00000111
4	16	STA 8	10	00010000
5	8	-	08	00001000
6	240	HLT	F0	11110000
7	5	NOP	05	00000101

3.1 Tratamento de dados

Considerando o arquivo de entrada com as instruções definidas acima, tem-se esta estrutura (representação em hexadecimal):

4e03 5244 0020 0007 0030 0007 0010 0008
00f0 0005 0000 0000 0000 0000 0000 0000

Os quatro primeiros *bytes* são referentes a inicialização do programa e aparecem no início de todos os arquivos lidos. A partir disso, cada dado é composto por dois *bytes*, porém, pelo fato do Neander ser um processador de 8 *bits*, o *byte* mais significativo nunca é utilizado. Logo, será armazenado em um vetor apenas os *bytes* menos significativos (em binário), que contém as informações necessárias para a tradução. Assim que finalizada a leitura do arquivo, o vetor resultante pode ser interpretado como uma tabela-verdade (Tabela 3.2).

Tabela 3.2: Tabela-verdade dos dados de entrada.

A	**B**	**C**	**f0**	**f1**	**f2**	**f3**	**f4**	**f5**	**f6**	**f7**
0	**0**	**0**	0	0	1	0	0	0	0	0
0	**0**	**1**	0	0	0	0	0	1	1	1
0	**1**	**0**	0	0	1	1	0	0	0	0
0	**1**	**1**	0	0	0	0	0	1	1	1
1	**0**	**0**	0	0	0	1	0	0	0	0
1	**0**	**1**	0	0	0	0	1	0	0	0
1	**1**	**0**	1	1	1	1	0	0	0	0
1	**1**	**1**	0	0	0	0	0	1	0	1

Neste caso, como tem oito linhas de dados, serão considerados três *bits* de entrada (A, B e C), correspondendo ao endereço de entrada de cada instrução. Mais adiante, elas serão usadas como variáveis de controle. No momento, direcionaremos nossa atenção para outro ponto: perceba que cada coluna, ou *bit* da saída, é representada por uma função (f0 a f7). Para cada função dessas, será gerado um BDD, sendo este algoritmo explicado a seguir.

3.2 Criação de BDDs

O principal objetivo para o uso do algoritmo desta seção é assegurar que todas as combinações das variáveis de entrada resultem nas saídas corretas para cada coluna da tabela-verdade. Isso é crucial para garantir que o circuito combinacional e, consequentemente, o arquivo de saída em Verilog funcionem de maneira equivalente ao programa executado no Neander. A seguir, na Tabela 3.3, está a lista das funções identificadas na

seção anterior que serão representadas na forma de BDD.

Tabela 3.3: Funções identificadas, através da tabela-verdade dos dados de entrada.

f0	00000010	**f4**	00000100
f1	00000010	**f5**	01010001
f2	10100010	**f6**	01010000
f3	00101010	**f7**	01010001

Para cada função, será executado o pseudocódigo abaixo.

Algorithm 1 Criação de BDD

```
1:  procedure CREATEBDD(f)
2:      nodes ← f
3:      variable ← 0
4:      while size(nodes) > 1 do            ▷ Quando for igual à 1, chegamos na raiz
5:          steps ← size(nodes)/2
6:          for step in steps do
7:              f0, f1 ← firstPair(nodes)
8:                          ▷ Após cada atribuição, é retirado o par de dados de nodes
9:              new_node ← bdd.createNode(variable, f1, f0)
10:             nodes.append(new_node)      ▷ O novo nodo é inserido no fim de nodes
11:         end for
12:         variable ← variable + 1
13:     end while
14:     bdd.setRoot(first(nodes))           ▷ O último nodo de nodes é a raiz deste bdd
15:     return bdd
16: end procedure
```

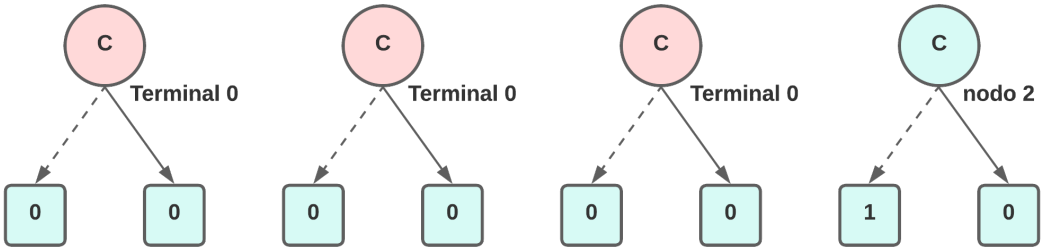
O algoritmo recebe como parâmetro uma função (por exemplo, f0), onde cada *bit* da função é colocado numa fila de nodos (começando do *bit* mais significativo) e é inicializado um atributo que representa a variável de entrada atual (começando pelo *bit* menos significativo). Enquanto houver dados na fila de nodos, será retirado o primeiro par de dados da fila (alocando-os aos atributos *f0* e *f1*, respectivamente) e criado um novo nodo, sendo *variable* a variável de controle, *f1* o filho do arco 1 do nodo e *f0* o filho do arco 0 do nodo. Aqui vale ressaltar que, se um novo nodo é igual a um já existente, será atribuído a ele o mesmo identificador do nodo já existente. Também é importante dizer que, se o nodo aponta para o terminal 0 através de ambos os arcos 0 e 1, o retorno da criação do nodo será 0, assim como, se o nodo aponta para o terminal 1 através de ambos os arcos 0 e 1, o retorno da criação do nodo será 1 (com isso em mente, o identificador de novos nodos inicia em 2). Após a atribuição de um novo nodo, este valor é inserido no

final da fila de nodos e isso se repetirá até que a rodada esteja completa, ou seja, que todos os pares de dados da fila de nodos inicial (considerando o início de cada rodada) tenham gerado novos nodos. A cada nova rodada a variável de controle é atualizada e, quando o *loop* finaliza, o nodo que restou na fila é identificado como a raiz do *BDD*.

Posto isso, vamos analisar a execução desse algoritmo de forma gráfica, utilizando a f0 (00000010) como parâmetro.

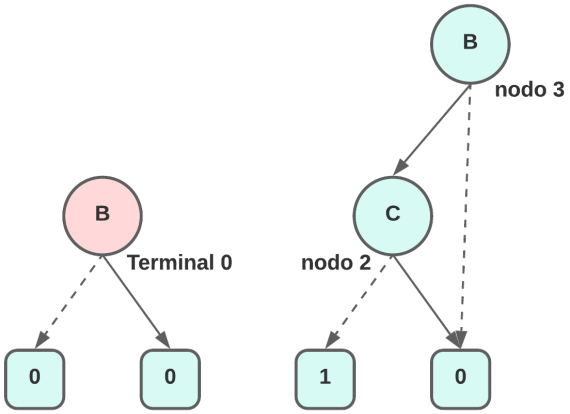
Primeira rodada: *variable* 0 é equivalente à variável de entrada C; temos os pares de dados iniciais 00, 00, 00 e 10. A Figura 3.1 apresenta a execução da primeira rodada, onde as três primeiras tentativas de criar um novo nodo resultam no retorno do inteiro 0, pois ambos os arcos 0 e 1 apontam para o terminal 0. Já o quarto par de dados irá gerar um novo nodo, cujo identificador é 2.

Figura 3.1: Primeira rodada da criação de BDD para f0. Fonte: a autora.



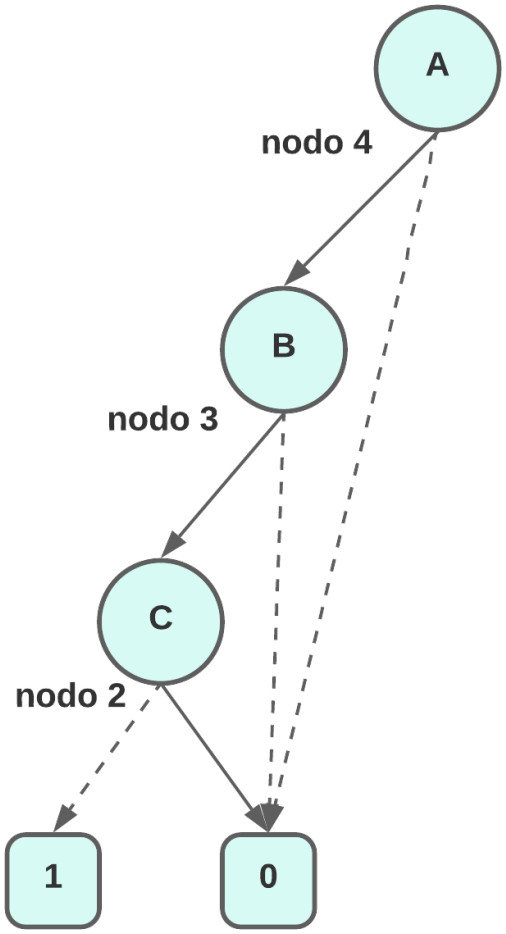
Segunda rodada: *variable* 1 é equivalente à variável de entrada B; finalizada a rodada anterior, agora temos os pares de dados 00 e 02. A Figura 3.2 mostra a execução desta rodada, onde, mais uma vez, temos um retorno 0 da função que cria novos nodos, consequência dos arcos 1 e 0 apontarem para o terminal 0 com o primeiro par de dados. Por outro lado, o segundo par de dados gerará um novo nodo, sendo 3 o seu identificador.

Figura 3.2: Segunda rodada da criação de BDD para f0. Fonte: a autora.



Terceira rodada: *variable* 2 é equivalente à variável de entrada A; por fim, nos restou o par de dados 03, decorrente da rodada passada. A Figura 3.3 exibe o BDD resultante da terceira rodada, onde o único par de dados na fila irá gerar um novo nodo, com identificador 4.

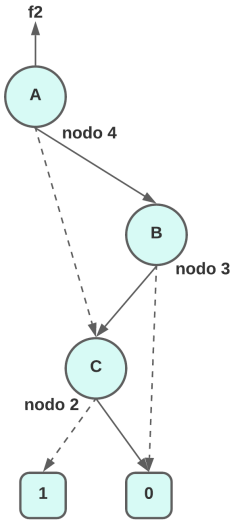
Figura 3.3: Terceira rodada da criação de BDD para f0. Fonte: a autora.



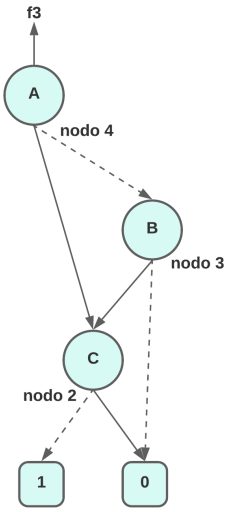
No final da terceira rodada, teremos apenas o nodo 4 na fila de nodos, logo, este será identificado como o nodo raiz e o BDD de f0 estará criado.

De forma análoga, a Figura 3.4 reúne os BDDs resultantes gerados para as demais funções, porém, como f1 é uma cópia de f0, seu BDD foi omitido. Pode ser observado que, em alguns dos BDDs, houve a tentativa de criar um novo já existente (nodo 2) e, por esse motivo, ele aparece mais de uma vez no diagrama de algumas funções.

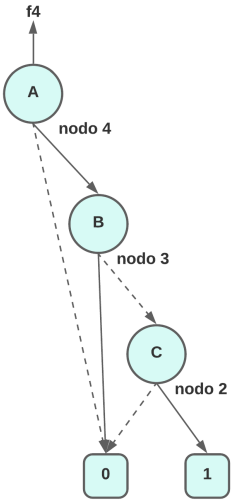
Figura 3.4: BDDs para as funções f2 até f7. Fonte: a autora.



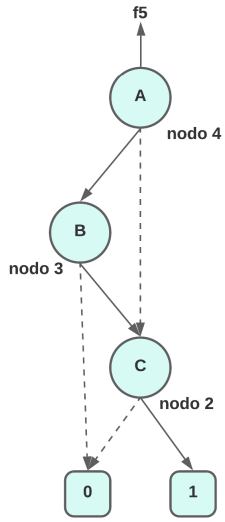
(a) BDD para f2.



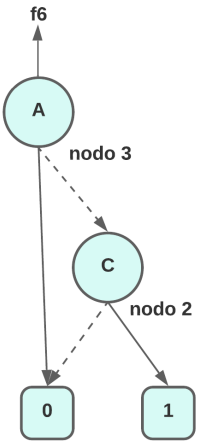
(b) BDD para f3.



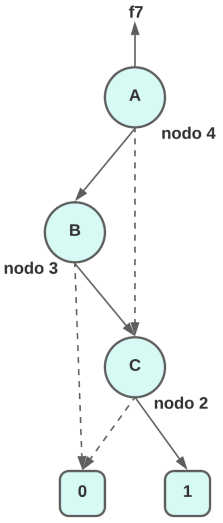
(c) BDD para f4.



(d) BDD para f5.



(e) BDD para f6.

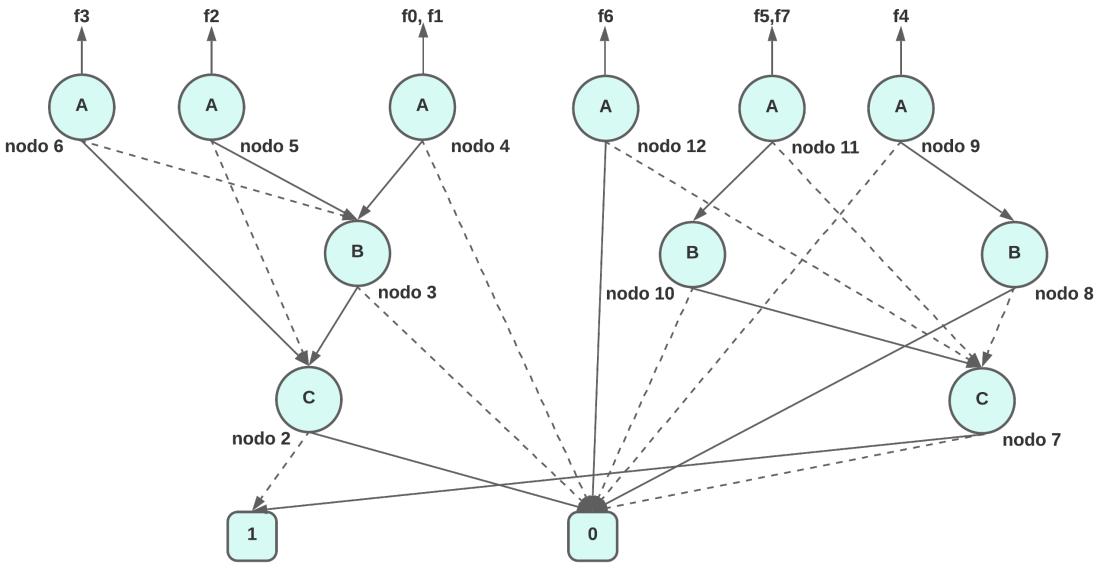


(f) BDD para f7.

A implementação considera, também, a criação de um BDD compartilhado entre todas as funções, tirando proveito da característica de que vários nodos se repetem entre as diferentes funções, conseguindo diminuir, dessa forma, o número total de nodos criados. A lógica da criação continua a mesma, a principal diferença é que, no momento da criação de um novo nodo, será considerado os nodos já criados pelas funções anteriores. Por exemplo, a execução do algoritmo de criação de BDD para f1 não irá acrescentar nenhum novo nodo, visto que f0 == f1.

Assim, após rodar o algoritmo para todas as funções, teremos o BDD apresentado na Figura 3.5 como resultado, com oito raízes, sendo que a raiz de f1 é igual a de f0 (nodo 4) e a raiz de f7 é igual a de f5 (nodo 11).

Figura 3.5: BDD compartilhado. Fonte: a autora.



3.3 Construção do circuito combinacional

Agora que todas as funções estão contempladas pelo BDD, passaremos para a próxima etapa. Para construção do circuito combinacional, serão levadas em conta as reduções de multiplexadores obtidas a partir das simplificações vistas na seção 2.5.2 do capítulo anterior. Para relembrar quais são estas reduções, a Tabela 3.4 apresenta uma síntese:

Tabela 3.4: Resumo das simplificações de multiplexadores em nodos de BDDs.

Caso geral: $nodo\_i = filho\_0 * !v\_j + v\_j * filho\_1$		
	Quando aplicar?	**Equivalência**
Redução 1	arco 0 aponta para T0 e arco 1 aponta para T1	$nodo\_i = v\_j$
Redução 2	arco 0 aponta para T1 e arco 1 aponta para T0	$nodo\_i = !v\_j$
Redução 3	arco 0 aponta para o T0	$nodo\_i = v\_j * filho\_1$
Redução 4	arco 0 aponta para o T1	$nodo\_i = !v\_j + filho\_1$
Redução 5	arco 1 aponta para T0	$nodo\_i = filho\_0 * !v\_j$
Redução 6	arco 1 aponta para T1	$nodo\_i = filho\_0 + v\_j$
Redução 7	a função do nodo é "*negative unate*"	$nodo\_i = filho\_0 * !v\_j + filho\_1$
Redução 8	a função do nodo é "*positive unate*"	$nodo\_i = filho\_0 + v\_j * filho\_1$

Com um BDD separado para cada função, podemos produzir os circuitos combinacionais com as reduções da Tabela 3.5 (note que para o nodo 2, que é nodo folha, sempre será aplicada a Redução 1 ou 2):

Tabela 3.5: Reduções aplicadas para cada função.

Função	**nodo 2**	**Equivalência**	**nodo 3**	**Equivalência**	**nodo 4**	**Equivalência**
f0	Redução 2	$n2 = !v0$	Redução 3	$n3 = v1 * n2$	Redução 3	$n4 = v2 * n3$
f1	Redução 2	$n2 = !v0$	Redução 3	$n3 = v1 * n2$	Redução 3	$n4 = v2 * n3$
f2	Redução 2	$n2 = !v0$	Redução 3	$n3 = v1 * n2$	Caso geral	$n4 = n2 * !v2 + v2 * n3$
f3	Redução 2	$n2 = !v0$	Redução 3	$n3 = v1 * n2$	Caso geral	$n4 = n3 * !v2 + v2 * n2$
f4	Redução 1	$n2 = v0$	Redução 5	$n3 = n2 * !v1$	Redução 3	$n4 = v2 * n3$
f5	Redução 1	$n2 = v0$	Redução 3	$n3 = v1 * n2$	Caso geral	$n4 = n2 * !v2 + v2 * n3$
f6	Redução 1	$n2 = v0$	Redução 5	$n3 = n2 * !v2$	-	-
f7	Redução 1	$n2 = v0$	Redução 3	$n3 = v1 * n2$	Caso geral	$n4 = n2 * !v2 + v2 * n3$

No entanto, se utilizarmos um único BDD, compartilhado entre as funções, os circuitos combinacionais criados para os nodos, com as reduções, serão como mostra a Tabela 3.6 a seguir. Perceba que, como este BDD é a união dos BDDs separados por função, toda redução contida aqui deve aparecer, também, em pelo menos um nodo de alguma função listada acima.

Tabela 3.6: Reduções aplicadas para os nodos do BDD compartilhado.

Nodo	Redução	Equivalência
nodo 2	Redução 2	$n2 = !v0$
nodo 3	Redução 3	$n3 = v1 * n2$
nodo 4	Redução 3	$n4 = v2 * n3$
nodo 5	Caso geral	$n5 = n2 * !v2 + v2 * n3$
nodo 6	Caso geral	$n6 = n3 * !v2 + v2 * n2$
nodo 7	Redução 1	$n7 = v0$
nodo 8	Redução 5	$n8 = n7 * !v1$
nodo 9	Redução 3	$n9 = v2 * n8$
nodo 10	Redução 3	$n10 = v1 * n7$
nodo 11	Caso geral	$n11 = n7 * !v2 + v2 * n10$
nodo 12	Redução 5	$n12 = n7 * !v2$

3.4 Geração do arquivo final em Verilog

Finalmente, alcançamos a última etapa da implementação. Neste momento, todos os circuitos combinacionais definidos na seção anterior serão traduzidos para linguagem Verilog e incorporados em um módulo padrão, conforme mostrado na Figura 3.6.

O módulo de memória ROM recebe como entrada um endereço (*address*), e fornece como saída o dado (*data*) correspondente ao endereço. Assim, o módulo padrão apresenta a porta de entrada *address* e a porta de saída *data*. A porta de entrada *address* é um barramento de até 8 *bits*, onde cada bit do endereço representará uma variável de entrada. A porta de saída *data*, também um barramento de 8 *bits*, terá cada *bit* assinalado à tabela-verdade de uma função Booleana associada a saída correspondente.

A partir do módulo padrão, ilustrado na Figura 3.6, várias implementações distintas da memória ROM podem ser criadas. A seguir, veremos dois exemplos distintos para a geração de um *hardware* em Verilog, correspondentes a funções Booleanas representadas através de BDDs.

Em um primeiro exemplo, a Figura 3.7 mostra a descrição em Verilog para o *hardware* correspondente a memória de programa, considerando que as funções têm seu BDD próprio individual. Neste caso, foi adicionado um prefixo com a função da qual o nodo

Figura 3.6: Módulo padrão de memória, no qual é inserido um circuito combinacional correspondente a ROM de um programa.

```
module memoria (address, data);

  input [7:0] address;
  output [7:0] data;
  assign v0 = address[0];
                ...
  assign v7 = address[7];

  //comeco do circuito combinacional
  // Aqui sao inseridos os circuitos combinacionais criados
                ...
  //fim do circuito combinacional

  assign data[7] = nodo_raiz_f0;
                ...
  assign data[0] = nodo_raiz_f7;

endmodule
```

pertence para impedir conflitos no nome dos nodos e erros no arquivo Verilog. Isso acontece porque os nodos do BDD são identificados como números inteiros e as variáveis no código Verilog fazem referência a estes números. Deste modo, aparecem nodos distintos, porém com a mesma numeração, em BDDs independentes. Um exemplo disso é o nodo *n2*, correspondente ao inteiro *2*, que ocorre em todos os BDDs independentes que deram origem ao código na Figura 3.7. A partir dos diversos nodos com a numeração *2*, são criadas oito variáveis com o nome *fn_n2*, uma para cada função Booleana *fn*, ou seja: *f0_n2*, *f1_n2*, e assim por diante até *f7_n2*.

Em um segundo exemplo, a Figura 3.8 mostra a descrição em Verilog para o *hardware* correspondente a memória de programa, considerando que as funções são representadas por um BDD compartilhado, onde cada função de saída corresponde a um nodo do BDD. Note que neste caso, o nodo com numeração *2* é único para todas as funções. Assim, as variáveis do código Verilog não tem prefixo, e se usa simplesmente *n2*, ao invés de *fn_n2*, conforme era necessário para o código resultante de BDDs independentes mostrado anteriormente na Figura 3.7.

Considerando o tamanho resultante para os circuitos combinacionais, o circuito da Figura 3.7 tem 23 linhas com o comando *assign fn_nn* (entre o começo e o fim do circuito combinacional); enquanto o circuito da Figura 3.8 tem apenas 11 linhas com o comando *assign nn* (entre o começo e o fim do circuito combinacional).

Figura 3.7: Módulo padrão preenchido com um circuito combinacional, onde cada uma das 8 funções Booleanas foi gerada como um BDD independente.

```
module memoria (address, data);

  input [2:0] address;
  output [7:0] data;
  assign v0 = address[0];
  assign v1 = address[1];
  assign v2 = address[2];

  //comeco do circuito combinacional
  assign f0_n2 = ~v0;
  assign f0_n3 = v1 & f0_n2;
  assign f0_n4 = v2 & f0_n3;
  assign f1_n2 = ~v0;
  assign f1_n3 = v1 & f1_n2;
  assign f1_n4 = v2 & f1_n3;
  assign f2_n2 = ~v0;
  assign f2_n3 = v1 & f2_n2;
  assign f2_n4 = f2_n2 & ~v2 | v2 & f2_n3;
  assign f3_n2 = ~v0;
  assign f3_n3 = v1 & f3_n2;
  assign f3_n4 = f3_n3 & ~v2 | v2 & f3_n2;
  assign f4_n2 = v0;
  assign f4_n3 = f4_n2 & ~v1;
  assign f4_n4 = v2 & f4_n3;
  assign f5_n2 = v0;
  assign f5_n3 = v1 & f5_n2;
  assign f5_n4 = f5_n2 & ~v2 | v2 & f5_n3;
  assign f6_n2 = v0;
  assign f6_n3 = f6_n2 & ~v2;
  assign f7_n2 = v0;
  assign f7_n3 = v1 & f7_n2;
  assign f7_n4 = f7_n2 & ~v2 | v2 & f7_n3;
  //fim do circuito combinacional

  assign data[7] = f0_n4;
  assign data[6] = f1_n4;
  assign data[5] = f2_n4;
  assign data[4] = f3_n4;
  assign data[3] = f4_n4;
  assign data[2] = f5_n4;
  assign data[1] = f6_n3;
  assign data[0] = f7_n4;

endmodule
```

3.5 Contribuições deste capítulo

Este capítulo apresentou como os circuitos de memória são gerados, partindo de um arquivo de programa. A partir deste arquivo, é gerado uma descrição das funções Booleanas da ROM na forma de BDD. Este BDD pode ser individual para cada função

Figura 3.8: Módulo padrão preenchido com um circuito combinacional, onde todas as 8 funções Booleanas foram geradas a partir de um único BDD compartilhado, apresentando múltiplas saídas.

```
module memoria (address, data);

  input [2:0] address;
  output [7:0] data;
  assign v0 = address[0];
  assign v1 = address[1];
  assign v2 = address[2];

  //comeco do circuito combinacional
  assign n2 = ~v0;
  assign n3 = v1 & n2;
  assign n4 = v2 & n3;
  assign n5 = n2 & ~v2 | v2 & n3;
  assign n6 = n3 & ~v2 | v2 & n2;
  assign n7 = v0;
  assign n8 = n7 & ~v1;
  assign n9 = v2 & n8;
  assign n10 = v1 & n7;
  assign n11 = n7 & ~v2 | v2 & n10;
  assign n12 = n7 & ~v2;
  //fim do circuito combinacional

  assign data[7] = n4;
  assign data[6] = n4;
  assign data[5] = n5;
  assign data[4] = n6;
  assign data[3] = n9;
  assign data[2] = n11;
  assign data[1] = n12;
  assign data[0] = n11;

endmodule
```

Booleana ou pode ser um BDD partilhado entre todas as funções. Com isso concluído, equações são geradas para cada nodo e inseridas em um cabeçalho padrão em Verilog compatível com a placa Pitanga. Essas equações podem ser geradas usando ou não as simplificações propostas anteriormente pelo grupo LogiCS (BRANDÃO et al., 2022a). Deste modo, quatro variações de síntese são possíveis.

O método apresentado neste capítulo corresponde a principal contribuição deste TCC: geração, compatíveis com a placa virtual Pitanga, de memórias de programa ROM do computador hipotético Neander. No próximo capítulo são apresentados os resultados obtidos no escopo deste trabalho.

4 RESULTADOS

Neste momento, será feito um comparativo entre as diferentes combinações dos métodos para geração de circuitos combinacionais apresentados no capítulo anterior. Testes foram realizados com programas de diferentes tamanhos (ver Apêndices A-E), e a Tabela 4.1 detalha a quantidade de portas lógicas utilizadas em cada método para cada programa.

Tabela 4.1: Quantidade de portas lógicas utilizadas em cada método para cada programa.

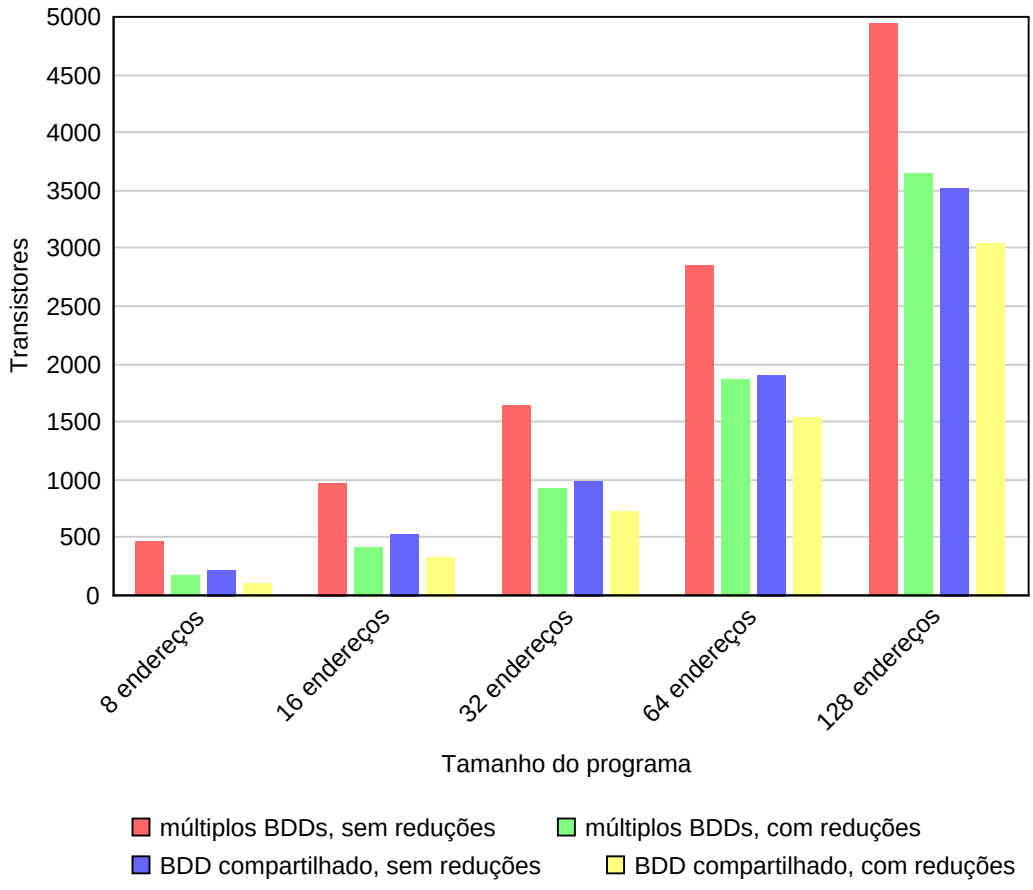
Tamanho	**Porta**	**Múltiplos BDDs**		**BDD compartilhado**	
		sem reduções	**com reduções**	**sem reduções**	**com reduções**
8 endereços	**and**	46	19	22	12
	or	23	4	11	3
	not	23	10	11	6
Total de portas		**92**	**33**	**44**	**21**
16 endereços	**and**	96	44	52	34
	or	48	15	26	15
	not	48	30	26	20
Total de portas		**192**	**89**	**104**	**69**
32 endereços	**and**	164	97	98	75
	or	82	36	49	32
	not	82	57	49	40
Total de portas		**328**	**190**	**196**	**147**
64 endereços	**and**	284	196	190	158
	or	142	78	95	70
	not	142	109	95	83
Total de portas		**568**	**383**	**380**	**311**
128 endereços	**and**	494	378	352	309
	or	247	162	176	145
	not	247	203	176	161
Total de portas		**988**	**743**	**704**	**615**

Percebe-se um padrão entre os BDDs sem reduções aplicadas, onde, para cada programa, temos o dobro de portas **and**, em relação às portas **or** e **not**, sendo que essas duas últimas aparecem em quantidades iguais. Isso se dá pela configuração de um multiplexador, que é o caso geral utilizado para os nodos sem redução. Já nos métodos com reduções, observamos uma queda significativa no número de portas utilizadas, no geral,

e as portas **not** estão em uma quantidade maior que as portas **or**, pela característica das reduções aplicadas.

Conforme a tecnologia CMOS, cada porta **and** e **or** possui seis transistores e cada porta **not** possui dois. Assim, na Figura 4.1, podemos comparar quantos transistores cada método utilizaria para gerar a tradução de cada programa do Neander. É possível observar que, somente aplicando as reduções, já obtemos uma queda grande no total de transistores. Indo mais além, ao utilizarmos um BDD compartilhado, aliado às reduções, este valor é ainda mais reduzido. E, mesmo sem as reduções, o BDD compartilhado já consegue mitigar a quantidade de transistores necessários para o circuito, pelo fato das funções compartilharem muitos nodos iguais.

Figura 4.1: Quantidade de transistores utilizados por cada método. Fonte: a autora.



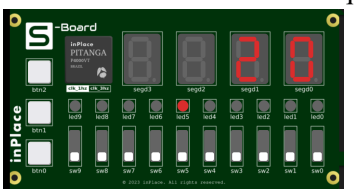
Considerando que os circuitos gerados podem ser implementados em FPGAs, há alguns pontos a serem observados em relação à importância de reduzir o número de transistores ou, mais genericamente, a complexidade das implementações. FPGAs têm recursos limitados, como número de células lógicas e blocos de RAM, e podem existir restrições de *design* específicas, como limites de consumo de energia ou requisitos de

área. Reduzir o número de transistores ajuda a atender a essas restrições e permite uma melhor utilização desses recursos. Além disso, quanto mais eficiente e compacta for a implementação, mais flexibilidade haverá para reprogramar a FPGA para diferentes funcionalidades, já que haverá mais recursos disponíveis. Sabe-se, também, que circuitos menos complexos são mais fáceis de manter e atualizar, o que é especialmente importante em sistemas sujeitos a mudanças frequentes.

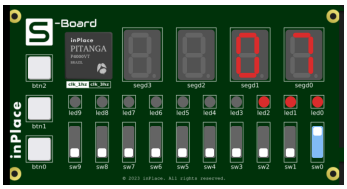
Quanto ao desempenho, implementações mais enxutas geralmente resultam em menor tempo de rota, menor consumo de energia e menor latência, o que é crítico em muitas aplicações FPGA. Reduzir a complexidade pode levar, também, a tempos de síntese e compilação mais curtos, acelerando o ciclo de desenvolvimento e teste do projeto. Ademais, a quantidade de recursos utilizados pode influenciar diretamente o custo de produção ou a escolha da FPGA. Portanto, reduzir a complexidade pode levar a uma escolha mais econômica.

Para fins de validação, as descrições de *hardware* geradas em Verilog foram executadas na S-board, placa de prototipagem virtual da empresa inPlace (COSTA; SILVEIRA; REIS, 2022b; COSTA; SILVEIRA; REIS, 2023b). Constata-se, na Figura 4.2, que as instruções são as mesmas para o conjunto de dados do Neander visto no capítulo anterior.

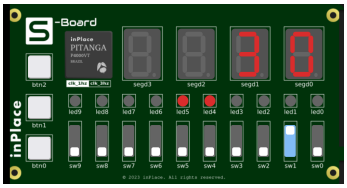
Figura 4.2: Emulação do arquivo Verilog de memória (saída) na placa Pitanga. O programa Neander de entrada é aquele mostrado na Tabela 3.1.



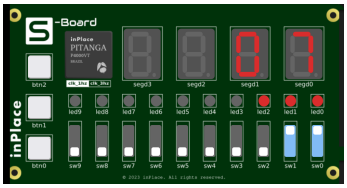
(a) Instrução 20 Endereço 0.



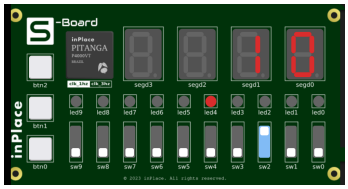
(b) Imediato 07 Endereço 1.



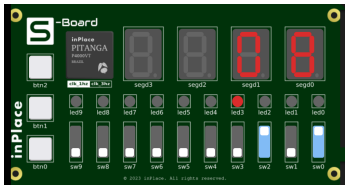
(c) Instrução 30 Endereço 2.



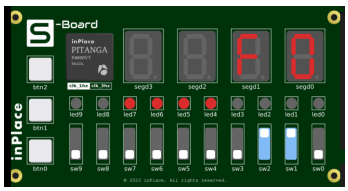
(d) Imediato 07 Endereço 3.



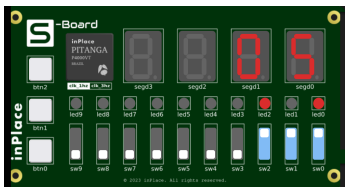
(e) Instrução 10 Endereço 4.



(f) Imediato 08 Endereço 5.



(g) Instrução F0 Endereço 6.



(h) Dado 05 Endereço 7.

A figura 4.2 mostra oito capturas distintas da placa Pitanga para o conteúdo de memória da Tabela 3.1. Em cada captura, o endereço é dado pelas chaves e o conteúdo da memória é mostrado nos LEDs e, também, nos *displays* de sete segmentos. Pode-se perceber que, para cada endereço nas chaves, o conteúdo é o esperado, a partir da Tabela 3.1, demonstrando que a geração da memória foi feita corretamente para o exemplo e é compatível com a placa Pitanga.

5 CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma ferramenta para geração de memórias de programa compatíveis com a ISA do processador didático hipotético Neander. A partir do formato de armazenamento de programas do simulador do Neander, são geradas memórias ROM compatíveis com ferramentas comerciais, tais como a plataforma Pitanga da empresa inPlace ou com os softwares Vivado da Xilinx e Quartus II da Intel (antiga Altera). Esta ferramenta permite que alunos de Circuitos Digitais convertam seus programas Neander diretamente para Verilog para uso em projetos físicos do Neander em diversas plataformas. O programa Neander é convertido em uma ROM, onde as funções Booleanas correspondentes ao conteúdo da memória do programa são descritas na forma de BDD, que é depois convertido para um conjunto de equações correspondentes aos seus nodos. A ferramenta proposta pode usar BDDs individuais para cada *bit* de saída da memória ROM ou usar um BDD compartilhado entre as memórias, apresentando múltiplas saídas. As equações dos nodos de BDDs podem ser geradas usando ou não as simplificações propostas, anteriormente, pelo grupo LogiCS (BRANDÃO et al., 2022a). Deste modo, quatro variações de síntese são possíveis.

Considerando-se trabalhos anteriores, algumas observações podem ser feitas. Em primeiro lugar, já houve um TCC anterior (PISONI, 2021), orientado pelo professor André Reis, sobre geração de memórias Neander em Verilog, todavia, este trabalho não focava em gerar um Verilog compatível com a placa Pitanga. Neste sentido, o trabalho aqui proposto é um avanço, pois os Verilogs gerados são compatíveis com inPlace/Pitanga, Intel/Quartus e Xilinx/Vivado. Em segundo lugar, as simplificações formuladas pelo grupo LogiCS, vistas anteriormente, (BRANDÃO et al., 2022a) não haviam sido consideradas neste TCC anterior (PISONI, 2021). Assim, o trabalho aqui apresentado é o primeiro a aplicar estas reduções (BRANDÃO et al., 2022a) de multiplexadores a partir de BDDs, no contexto de geração automática de memórias do Neander em Verilog.

5.1 Limitações

Em termos de limitações deste trabalho, pode-se apontar duas principais. A primeira delas é que a maneira mais óbvia de sintetizar uma ROM seria, provavelmente, em forma de soma de produtos com uma PLA. Para uma maior perspectiva de implementação, este caso de base deveria ter sido implementado para comparação, caso houvesse

mais tempo. Porém, no contexto de um TCC, é aceitável que esta implementação não tenha sido realizada. Uma segunda limitação é o não atendimento a parte RAM da memória.

5.2 Trabalhos futuros

A sugestão de trabalhos futuros passa por atacar as limitações do projeto, listadas na seção anterior. Assim, um primeiro trabalho futuro seria implementar a geração de equações em soma de produtos para ter PLA como caso de base. Um segundo trabalho futuro seria implementar o suporte as palavras que necessitam de escrita, em alguma forma de RAM. Outros trabalhos futuros poderiam considerar que memórias têm partes que não estão preenchidas e considerar estas como *don´t cares*. Uma possibilidade seria usar os trabalhos recentes de Peralta e co-autores (PERALTA et al., 2021b) (PERALTA et al., 2023b), que consideram *don´t cares* em BDDs.

5.3 Concluindo em uma nota positiva

Este trabalho de conclusão de curso de Engenharia de computação cumpre os requisitos esperados para um TG2. São integrados vários conceitos tais como arquiteturas de memória, processadores, formato de descrição de memória, linguagem de descrição de *hardware*, funções Booleanas e sua representação através de BDDs, bem como a geração das equações a partir do BDD. Estes conceitos são explorados para implementar uma ferramenta de síntese de memórias para uso em plataformas virtuais (inPlace/Pitanga) ou de FPGA (Intel/Altera ou Xilix/Vivado). Desta forma, os requisitos do TG2 são plenamente alcançados.

REFERÊNCIAS

BRANDÃO, E. D. et al. Possible reductions to generate circuits from bdds. In: **2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.: s.n.], 2022. p. 406–409.

BRANDÃO, E. D. et al. Possible reductions to generate circuits from bdds. In: **2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.: s.n.], 2022. p. 406–409.

BRYANT. Graph-based algorithms for boolean function manipulation. **IEEE Transactions on Computers**, C-35, n. 8, p. 677–691, 1986.

BUTZEN, P. et al. Design of cmos logic gates with enhanced robustness against aging degradation. **Microelectronics Reliability**, v. 52, n. 9, p. 1822–1826, 2012. ISSN 0026-2714. SPECIAL ISSUE 23rd EUROPEAN SYMPOSIUM ON THE RELIABILITY OF ELECTRON DEVICES, FAILURE PHYSICS AND ANALYSIS. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0026271412002892>.

BUTZEN, P. F. et al. Transistor network restructuring against nbti degradation. **Microelectronics Reliability**, v. 50, n. 9, p. 1298–1303, 2010. ISSN 0026-2714. 21st European Symposium on the Reliability of Electron Devices, Failure Physics and Analysis. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0026271410004130>.

BUTZEN, P. F. et al. Standby power consumption estimation by interacting leakage current mechanisms in nanoscaled cmos digital circuits. **Microelectronics Journal**, v. 41, n. 4, p. 247–255, 2010. ISSN 0026-2692. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S002626921000042X>.

CALLEGARO, V. et al. Switchcraft: A framework for transistor network design. In: **Proceedings of the 23rd Symposium on Integrated Circuits and System Design**. New York, NY, USA: Association for Computing Machinery, 2010. (SBCCI '10), p. 49–53. ISBN 9781450301527. Available from Internet: <https://doi.org/10.1145/1854153.1854167>.

CORREIA, V.; REIS, A. Advanced technology mapping for standard-cell generators. In: **Proceedings of the 17th Symposium on Integrated Circuits and System Design**. New York, NY, USA: Association for Computing Machinery, 2004. (SBCCI '04), p. 254–259. ISBN 1581139470. Available from Internet: <https://doi.org/10.1145/1016568.1016636>.

CORREIA, V. P.; REIS, A. I. Classifying n-input boolean functions. In: **VII Workshop Iberchip (Vol. 58).** [S.l.: s.n.], 2001.

COSTA, A. S.; SILVEIRA, L. D.; REIS, A. I. A virtual board approach for prototyping and teaching digital design. In: **2022 35th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.: s.n.], 2022. p. 1–6.

COSTA, A. S.; SILVEIRA, L. D.; REIS, A. I. A virtual board approach for prototyping and teaching digital design. In: **2022 35th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.: s.n.], 2022. p. 1–6.

COSTA, A. S.; SILVEIRA, L. D.; REIS, A. I. Live demonstration: Pitanga platform for virtual fpga remote laboratories. In: **2023 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.: s.n.], 2023. p. 1–5.

COSTA, A. S.; SILVEIRA, L. D.; REIS, A. I. Live demonstration: Pitanga platform for virtual fpga remote laboratories. In: **2023 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.: s.n.], 2023. p. 1–5.

da Silva, D. N.; REIS, A. I.; RIBAS, R. P. Cmos logic gate performance variability related to transistor network arrangements. **Microelectronics Reliability**, v. 49, n. 9, p. 977–981, 2009. ISSN 0026-2714. 20th European Symposium on the Reliability of Electron Devices, Failure Physics and Analysis. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0026271409002546>.

GOMES, I. A. C. et al. Methodology for achieving best trade-off of area and fault masking coverage in atmr. In: **2014 15th Latin American Test Workshop - LATW**. [S.l.: s.n.], 2014. p. 1–6.

GOMES, I. A. C. et al. Using only redundant modules with approximate logic to reduce drastically area overhead in tmr. In: **2015 16th Latin-American Test Symposium (LATS)**. [S.l.: s.n.], 2015. p. 1–6.

HENNESSY, D. A. P. J. L. **Computer Architecture: A Quantitative Approach 5th edition**. [S.l.]: Elsevier, 2012.

JUNIOR, L. S. da R. et al. Fast disjoint transistor networks from bdds. In: **Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design**. New York, NY, USA: Association for Computing Machinery, 2006. (SBCCI '06), p. 137–142. ISBN 1595934790. Available from Internet: <https://doi.org/10.1145/1150343.1150381>.

KNUTH, D. E. **The Art of Computer Programming: Combinatorial Algorithms, Part 1**. 1st. ed. [S.l.]: Addison-Wesley Professional, 2011. ISBN 0201038048.

MACHADO, L. et al. Kl-cut based digital circuit remapping. In: **NORCHIP 2012**. [S.l.: s.n.], 2012. p. 1–4.

MANO, M. M.; MICHAEL, D. et al. **Digital Design With an Introduction to the Verilog HDL FIFTH EDITION**. [S.l.]: Pearson, 2013. P. 125-126.

MARQUES, F. S. et al. Dag based library-free technology mapping. In: **Proceedings of the 17th ACM Great Lakes Symposium on VLSI**. New York, NY, USA: Association for Computing Machinery, 2007. (GLSVLSI '07), p. 293–298. ISBN 9781595936059. Available from Internet: <https://doi.org/10.1145/1228784.1228857>.

MARRANGHELLO, F. S. et al. Factored forms for memristive material implication stateful logic. **IEEE Journal on Emerging and Selected Topics in Circuits and Systems**, v. 5, n. 2, p. 267–278, 2015.

MARTINS, M. et al. Open cell library in 15nm freepdk technology. In: **Proceedings of the 2015 Symposium on International Symposium on Physical Design**. New York, NY, USA: Association for Computing Machinery, 2015. (ISPD '15), p. 171–178. ISBN 9781450333993. Available from Internet: <https://doi.org/10.1145/2717764.2717783>.

MARTINS, M. G. A.; RIBAS, R. P.; REIS, A. I. Functional composition: A new paradigm for performing logic synthesis. In: **Thirteenth International Symposium on Quality Electronic Design (ISQED)**. [S.l.: s.n.], 2012. p. 236–242.

MOREIRA, M. et al. Semi-custom ncl design with commercial eda frameworks: Is it possible? In: **2014 20th IEEE International Symposium on Asynchronous Circuits and Systems**. [S.l.: s.n.], 2014. p. 53–60.

NEUTZLING, A. et al. A simple and effective heuristic method for threshold logic identification. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 37, n. 5, p. 1023–1036, 2018.

NEUTZLING, A. et al. Synthesis of threshold logic gates to nanoelectronics. In: **2013 26th Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.: s.n.], 2013. p. 1–6.

NEUTZLING, A. et al. Effective logic synthesis for threshold logic circuit design. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 38, n. 5, p. 926–937, 2019.

NEUTZLING, A. et al. Threshold logic synthesis based on cut pruning. In: **2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2015. p. 494–499.

PERALTA, R. D. et al. A method to join the on-set and off-set of an incompletely boolean function into a single bdd. In: **2021 34th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.: s.n.], 2021. p. 1–6.

PERALTA, R. D. et al. A method to join the on-set and off-set of an incompletely boolean function into a single bdd. In: **2021 34th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.: s.n.], 2021. p. 1–6.

PERALTA, R. D. et al. An improved method to join bdds for incompletely specified boolean functions. In: **2023 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.: s.n.], 2023. p. 1–5.

PERALTA, R. D. et al. An improved method to join bdds for incompletely specified boolean functions. In: **2023 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.: s.n.], 2023. p. 1–5.

PISONI, F. P. **Ferramenta para gerar Verilog a partir de memórias de processadores didáticos**. Porto Alegre: UFRGS - Trabalho de Conclusão do curso de Engenharia de Computação, 2021.

POLI, R. et al. Unified theory to build cell-level transistor networks from bdds [logic synthesis]. In: **16th Symposium on Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings.** [S.l.: s.n.], 2003. p. 199–204.

REIS, A. Covering strategies for library free technology mapping. In: **Proceedings. XII Symposium on Integrated Circuits and Systems Design (Cat. No.PR00387)**. [S.l.: s.n.], 1999. p. 180–183.

REIS, A.; MATOS, J. Physical awareness starting at technology-independent logic synthesis. In: REIS, A.; DRECHSLER, R. (Ed.). **Advanced Logic Synthesis**. [S.l.]: Springer, 2018.

REIS, A. I.; DRECHSLER, R. **Advanced logic synthesis**. Springer, 2018. ISBN 978-3-319-88407-3. Available from Internet: <https://doi.org/10.1007/978-3-319-67295-3>.

REIS, A. I. et al. Library free technology mapping. In: ____. **VLSI: Integrated Systems on Silicon: IFIP TC10 WG10.5 International Conference on Very Large Scale Integration 26–30 August 1997, Gramado, RS, Brazil**. Boston, MA: Springer US, 1997. p. 303–314. ISBN 978-0-387-35311-1. Available from Internet: <https://doi.org/10.1007/978-0-387-35311-1_25>.

ROSA, L. S. et al. Scheduling policy costs on a java microcontroller. In: MEERSMAN, R.; TARI, Z. (Ed.). **On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 520–533. ISBN 978-3-540-39962-9.

ROSA, L. S. da et al. A comparative study of cmos gates with minimum transistor stacks. In: **Proceedings of the 20th Annual Conference on Integrated Circuits and Systems Design**. New York, NY, USA: Association for Computing Machinery, 2007. (SBCCI '07), p. 93–98. ISBN 9781595938169. Available from Internet: <https://doi.org/10.1145/1284480.1284511>.

ROSA, L. S. da et al. Switch level optimization of digital cmos gate networks. In: **2009 10th International Symposium on Quality Electronic Design**. [S.l.: s.n.], 2009. p. 324–329.

THOMAS, D.; MOORBY, P. **The Verilog® hardware description language**. [S.l.]: Springer Science & Business Media, 2008.

TOGNI, J. et al. Automatic generation of digital cell libraries. In: **Proceedings. 15th Symposium on Integrated Circuits and Systems Design**. [S.l.: s.n.], 2002. p. 265–270.

WAGNER, F. R.; REIS, A. I.; RIBAS, R. P. **Fundamentos de circuitos digitais**. [S.l.]: Sagra Luzzatto, Porto Alegre, 2006.

WEBER, R. F. **Fundamentos de arquitetura de computadores**. 4. ed. Porto Alegre: Bookman, 2012.

APÊNDICE A — PROGRAMAS DO NEANDER PARA TESTES

Tabela A.1: Arquivo de entrada com 8 endereços.

End.	Dado	Mnemônico
0	32	LDA 7
1	7	-
2	48	ADD 7
3	7	-
4	16	STA 8
5	8	-
6	240	HLT
7	5	NOP

Tabela A.2: Arquivo de entrada com 16 endereços.

End.	Dado	Mnemônico
0	32	LDA 130
1	130	-
2	48	ADD 131
3	131	-
4	160	JZ 15
5	15	-
6	96	NOT
7	16	STA 131
8	131	-
9	32	LDA 129
10	129	-
11	48	ADD 130
12	130	-
13	130	JMP 2
14	2	-
15	240	HLT

Tabela A.3: Arquivo de entrada com 32 endereços.

End.	**Dado**	**Mnemônico**
0	32	LDA 130
1	130	-
2	48	ADD 131
3	131	-
4	160	JZ 31
5	31	-
6	96	NOT
7	16	STA 131
8	131	-
9	32	LDA 129
10	129	-
11	48	ADD 130
12	130	-
13	26	STA 130
14	130	-
15	80	AND 128
16	128	-
17	160	JZ 2
18	2	-
19	32	LDA 131
20	131	-
21	96	NOT
22	16	STA 131
23	131	-
24	144	JN 31
25	31	-
26	48	ADD 133
27	133	-
28	128	JMP 2
29	2	-
30	0	NOP
31	240	HLT

Tabela A.4: Arquivo de entrada com 64 endereços.

End.	Dado	Mnemônico
0	32	LDA 128
1	128	-
2	160	JZ 122
3	122	-
4	80	AND 133
5	133	-
6	160	JZ 10
7	10	-
8	128	JMP 140
9	140	-
10	32	LDA 133
11	133	-
12	16	STA 138
13	138	-
14	32	LDA 129
15	129	-
16	16	STA 135
17	135	-
18	32	LDA 130
19	130	-
20	16	STA 136
21	136	-
22	32	LDA 129
23	129	-
24	144	JN 32
25	32	-
26	32	LDA 130
27	130	-
28	144	JN 49
29	49	-
30	128	JMP 38
31	38	-
32	32	LDA 130
33	130	-
34	144	JN 38
35	38	-
36	128	JMP 57
37	57	-
38	32	LDA 130
39	130	-
40	96	NOT
41	48	ADD 133
42	133	-
43	48	ADD 129
44	129	-
45	144	JN 49
46	49	-
47	128	JMP 57
48	57	-
49	32	LDA 129
50	129	-
51	16	STA 136
52	136	-
53	32	LDA 130
54	130	-
55	16	STA 135
56	135	-
57	32	LDA 135
58	135	-
59	96	NOT
60	16	STA 137
61	137	-
62	0	NOP
63	240	HLT

Tabela A.5: Arquivo de entrada com 128 endereços.

End.	Dado	Mnemôn.
0	32	LDA 128
1	128	-
2	160	JZ 122
3	122	-
4	80	AND 133
5	133	-
6	160	JZ 10
7	10	-
8	128	JMP 140
9	140	-
10	32	LDA 133
11	133	-
12	16	STA 138
13	138	-
14	32	LDA 129
15	129	-
16	16	STA 135
17	135	-
18	32	LDA 130
19	130	-
20	16	STA 136
21	136	-
22	32	LDA 129
23	129	-
24	144	JN 32
25	32	-
26	32	LDA 130
27	130	-
28	144	JN 49
29	49	-
30	128	JMP 38
31	38	-
32	32	LDA 130
33	130	-
34	144	JN 38
35	38	-
36	128	JMP 57
37	57	-
38	32	LDA 130
39	130	-
40	96	NOT
41	48	ADD 133
42	133	-
43	48	ADD 129
44	129	-
45	144	JN 49
46	49	-
47	128	JMP 57
48	57	-
49	32	LDA 129
50	129	-
51	16	STA 136
52	136	-
53	32	LDA 130
54	130	-
55	16	STA 135
56	135	-
57	32	LDA 135
58	135	-
59	96	NOT
60	48	ADD 133
61	133	-
62	16	STA 137
63	137	-
64	32	LDA 128
65	128	-
66	144	JN 86
67	86	-
68	48	ADD 137
69	137	-
70	160	JZ 100
71	100	-
72	144	JN 140
73	140	-
74	128	JMP 68
75	68	-
76	32	LDA 128
77	128	-
78	48	ADD 137
79	137	-
80	160	JZ 100
81	100	-
82	144	JN 78
83	78	-
84	128	JMP 114
85	114	-
86	32	LDA 135
87	135	-
88	144	JN 92
89	92	-
90	128	JMP 76
91	76	-
92	32	LDA 128
93	128	-
94	48	ADD 137
95	137	-
96	160	JZ 100
97	100	-
98	128	JMP 140
99	140	-
100	32	LDA 138
101	138	-
102	160	JZ 122
103	122	-
104	32	LDA 136
105	136	-
106	16	STA 135
107	135	-
108	32	LDA 132
109	132	-
110	16	STA 138
111	138	-
112	128	JMP 57
113	57	-
114	48	ADD 137
115	137	-
116	160	JZ 100
117	100	-
118	144	JN 140
119	140	-
120	128	JMP 114
121	114	-
122	32	LDA 133
123	133	-
124	16	STA 131
125	131	-
126	0	NOP
127	240	HLT