Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Projeto e Otimização de Circuitos Digitais por Técnicas de Evolução Artificial

Vitor Coimbra de Oliveira

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Marcus Vinicius Lamar

Brasília
2015

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Homero Luiz Piccolo

Banca examinadora composta por:

Prof. Dr. Marcus Vinicius Lamar (Orientador) — CIC/UnB
Prof. Dr. Ricardo Pezzuol Jacobi — CIC/UnB
Prof. Dr. Márcio da Costa Pereira Brandão — CIC/UnB

CIP — Catalogação Internacional na Publicação

Oliveira, Vitor Coimbra de.

Projeto e Otimização de Circuitos Digitais por Técnicas de Evolução Artificial / Vitor Coimbra de Oliveira. Brasília : UnB, 2015.
183 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2015.

1. algoritmo genético, 2. hardware evolutivo, 3. circuito digital, 4. cgp

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Projeto e Otimização de Circuitos Digitais por Técnicas de Evolução Artificial

Vitor Coimbra de Oliveira

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Marcus Vinicius Lamar (Orientador)
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Márcio da Costa Pereira Brandão
CIC/UnB

Prof. Dr. Homero Luiz Piccolo
Coordenador do Bacharelado em Ciência da Computação

Brasília, 02 de agosto de 2015

Dedicatória

Dedico a meus pais, Marianita e Oswaldo, e meus irmãos, Artur e Luiza, que nunca deixaram de me apoiar.

Agradecimentos

Agradeço ao meu orientador, Marcus Lamar, pelo grande apoio e suporte desde o princípio deste trabalho.

Agradeço também ao meu colega e amigo Matheus Pimenta por sua dedicação e interesse à área da computação.

Resumo

Técnicas de projeto de circuitos digitais atualmente se baseiam principalmente em métodos *top-down*, que utilizam um conjunto de regras e restrições para auxiliar a construção do projeto. Por conta disso, ainda há um espaço desconhecido de soluções para vários problemas. Algoritmos genéticos, por outro lado, constroem soluções utilizando uma metodologia *bottom-up*, e provaram-se úteis para problemas de alta complexidade e de otimização.

Este trabalho propõe um novo algoritmo, denominado HMC-CGP, com convergência e otimização eficientes para problemas de síntese circuitos digitais. Em sua essência, esse algoritmo opera encontrando uma solução funcional utilizando a técnica MC-CGP e, em seguida, buscando otimizá-la utilizando CGP.

Os circuitos utilizados para testes foram somadores de 1 e 2 *bits*, multiplicador de 2 *bits* e decodificador para *display* de 7 segmentos. Os resultados observados mostram que o método converge mais rapidamente para uma resposta válida do que o método CGP convencional e atinge patamares de até 50% de redução para números de portas e transistores.

Palavras-chave: algoritmo genético, hardware evolutivo, circuito digital, cgp

Abstract

Current digital circuit design techniques are based on top-down methods, which depend on a set of rules and restrictions made to help the design process. Because of that, there is still an unknown space of solutions for many problems. Genetic algorithms, on the other hand, build solutions by using a bottom-up methodology and have proven themselves useful for high complexity and optimization problems.

This work proposes a new algorithm, called HMC-CGP, for efficient convergence and optimization of digital circuit sintesis problems. In its essence, this algorithm first finds a functionally correct solution by making use of a technique called MC-CGP and, in its next phase, optimizes it by using another technique called CGP.

The circuits of the one and two bits adders, two bits multiplier and 7 segment display are used to verify the algorithm. The results show that the proposed algorithm converges faster to a functional circuit than the common CGP technique. The optimization phase was able to reduce gate and transistor counts by up to 50%.

Keywords: genetic algorithm, evolving hardware, digital circuit, cgp

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ASIC Application-Specific Integrated Circuit. 19

BCD Binary Coded Decimal. 50

CGP Cartesian Genetic Programming. iii, iv, 2, 27, 29–31, 36, 41–43, 45–49, 51, 52, 65, 73, 74

CPU Central Processing Unit. 45

DNA Ácido Desoxirribonucleico. 1

EEPROM Electrically Erasable Programmable Read-Only Memory. 23

FPGA Field-Programmable Gate Array. 1–4, 19–21, 23, 33, 35–37, 39–41, 45, 67, 74, 75

FPL Field-Programmable Logic. 20

HDL Hardware Description Language. 19, 34

HMC-CGP Hybrid Multi-Chromosome Cartesian Genetic Programming. iii, iv, 2, 3, 41–45, 52, 54, 56, 58, 59, 63, 65, 71, 73–75

I/O Input / Output. 19, 21

LAB Logic Array Block. 21, 22

LE Logic Element. 20, 21, 67

LED Light emitter diode. 23, 51

LUT Look-up Table. 19, 21

MC-CGP Multi-Chromosome Cartesian Genetic Programming. iii, iv, 2, 31, 36, 41, 42, 45–49, 52, 65, 72–74

PC Personal Computer. 75

PLL Phase Locked Loop. 21

RAM Random Access Memory. 45

SDRAM Synchronous Dynamic Random Access Memory. 23

SRAM Synchronous Random Access Memory. 23

tpd propagation delay time. 5

USB Universal Serial Bus. 23, 75

Capítulo 1

Introdução

Tradicionalmente, os seres humanos projetam sistemas de alta complexidade, tais como prédios, computadores e carros, utilizando um conjunto complexo de regras que visam cumprir uma série de requerimentos. Por natureza, esse processo é feito de uma sistemática *top-down*, ou seja, se parte de uma especificação abstrata do problema e segue-se através da construção de sistemas menores e mais específicos. Esta sistemática se apresenta em forte contraste com o mecanismo que resultou na concepção da imensa diversidade de seres vivos hoje encontrados. Os seres vivos são a consequência de um conjunto de instruções simples, codificadas por DNA, que constroem componentes mais complexos. Um organismo é criado após inúmeras reações químicas causadas pelo DNA. Organismos complexos são então formados através de um processo evolutivo. Segundo Darwin (Darwin [1859]), uma das bases da evolução é o mecanismo de Seleção Natural. Este mecanismo é baseado em uma população composta de diversos indivíduos, cada um com características próprias definidas pelos genes. A sobrevivência de um indivíduo e sua capacidade de transferir seus genes para seus descendentes são diretamente relacionados à sua adaptação em relação ao ambiente em que se encontra.

Com o advento da computação, diversas técnicas para design de componentes digitais para auxiliar esse evento e solucionar diferentes problemas surgiram nos últimos anos.

Dentre eles, surgiram os chamados dispositivos reconfiguráveis, cuja principal característica é a sua capacidade de ser configurado para a realização de tarefas específicas. Alguns desses dispositivos podem ter partes reconfiguradas enquanto outras estão executando suas funções. Um tipo especial de circuito reconfigurável chamado Field-Programmable Gate Array (FPGA) é utilizado neste trabalho.

Formado essencialmente por uma grade de células lógicas, cada qual interconectada com suas vizinhas e cujo objetivo é controlar se dois fios estão eletricamente ligados ou não, torna-se possível a implementação de um vasto conjunto de funções Booleanas (Garcia et al. [2006]).

Paralelamente, outra maneira de solução de problemas, chamada de *Programação Genética*, baseada em algoritmos genéticos, já vem há muito tempo sendo utilizada para problemas relacionados a *software*. Algoritmos genéticos se baseiam fortemente nos princípios da seleção natural para resolução de problemas (Melanie [1999]). Os indivíduos de uma população representam as possíveis soluções e são codificados em uma sequência de *bits* chamada *bitstream*, atuando com o papel de cromossomos, descrevendo essa solução. Novas populações são formadas e testadas utilizando a chamada função de *fitness*, em

que cada indivíduo é avaliado e associado a um *score* baseado no quão perto do resultado desejado ele chegou. Esse *score* determina suas chances de se reproduzirem e passarem seus cromossomos para as populações seguintes (Melanie [1999]). Dessa maneira, à medida que novas iterações são feitas, indivíduos pouco adaptados tendem a não sobreviver e os mais adaptados a se reproduzir. Ao longo do tempo, devido aos mecanismos de *crossover* e mutação geralmente encontrados em tais algoritmos, pequenas modificações podem ser observadas, resultando em novas soluções e possivelmente gerando também um aumento no *fitness* da população e, consequentemente, uma aproximação maior à solução adequada.

A motivação principal deste trabalho foi a união dos dois campos apresentados e possivelmente a exploração de um espaço de soluções ainda não bem explorado. A utilização de ambas as técnicas pode ser aplicada tanto a circuitos digitais quanto analógicos (Burian [2009]). Neste trabalho serão explorados apenas a aplicação de algoritmos genéticos a circuitos digitais.

1.1 Objetivo Geral

Este trabalho possui como objetivo principal o estudo e implementação de técnicas de evolução genética no projeto de circuitos digitais combinacionais e sequenciais em FPGA.

1.2 Objetivos Específicos

De forma a verificar a eficácia da abordagem genética a projetos de circuitos digitais, as seguintes metas foram estabelecidas:

- Implementar os algoritmos CGP e MC-CGP, comumente utilizados para a evolução de circuitos digitais;
- Projetar os circuitos a serem avaliados: somadores de 1 e 2 *bits*, multiplicador de 2 *bits* e *display* de 7 segmentos;
- Propor uma nova estratégia evolutiva HMC-CGP;
- Mostrar a eficácia do método proposto por meio de comparação dos resultados nos circuitos projetados.

1.3 Justificativa

Thompson [1998] mostra que esse tipo de abordagem oferece algumas vantagens interessantes quando se utiliza um *hardware* reconfigurável como uma FPGA. O fato de ser reconfigurável torna viável a rápida execução de diferentes circuitos, permitindo a flexibilidade necessária normalmente exigida em algoritmos evolucionários. Das vantagens mencionadas, algumas são: um ambiente mais rico proveniente da falta de restrições usualmente aplicadas para a conveniência de *designers* e, por consequência, o aproveitamento de características sutis da física dos semicondutores dos componentes. Como resultado

disso, circuitos então evoluídos geneticamente podem apresentar melhor uso da área oferecida, menor gasto de energia e maior eficiência na execução da tarefa dada quando comparados a circuitos feitos utilizando o método de projeto convencional.

Este trabalho está organizado da seguinte forma: no capítulo 2 é apresentada a revisão bibliográfica sobre os conceitos de álgebra booleana, circuitos digitais, FPGAs e algoritmos genéticos e suas abordagens. O capítulo 3 apresenta um detalhamento sobre a metodologia utilizada e desenvolvimento da proposta de evolução HMC-CGP. No capítulo 4 são mostrados os resultados obtidos pela aplicação da metodologia proposta. Finalmente, o capítulo 5 apresenta as conclusões e trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo apresentará conceitos básicos utilizados neste trabalho. Primeiramente, serão mostrados os fundamentos da Álgebra Booleana, juntamente com técnicas de otimização de expressões lógicas. Uma breve explicação sobre os dispositivos FPGAs é dada, em seguida, e, por último, é feita uma revisão sobre a base teórica de algoritmos genéticos.

2.1 Álgebra Booleana

Inicialmente proposta por George Boole (Nelson et al. [1995]), a área hoje denominada Álgebra Booleana trata de problemas relacionados à lógica moderna. Essa lógica também forma a base para computação em sistemas modernos. Qualquer algoritmo ou circuito eletrônico digital pode ser representado por um sistema de equações booleanas (Hyde [2001]).

Nesta álgebra, variáveis assumem apenas um conjunto finito de valores. Mais precisamente, apenas dois valores distintos são representados: $\{0, 1\}$. Operações são realizadas sobre essas variáveis utilizando operadores unários, que possuem uma variável como entrada, e binários, que possuem duas.

2.1.1 Operadores Básicos

As funções básicas e principais que compõem todas as outras possíveis são denominadas: *AND*, *OR* e *NOT*, formando então um conjunto completo de conectivos.

Operador *AND*

O resultado da operação *AND* pode ser definido como 1 apenas se ambas as entradas são 1 e 0 senão. Essa operação normalmente é representada utilizando o símbolo $\cdot$ e sua tabela verdade é mostrada na figura 2.1.

Operador *OR*

O operador *OR* tem como sua saída 1 se quaisquer das entradas for 1, senão é definida como 0. Sua tabela verdade pode ser escrita conforme mostra a tabela na figura 2.2. Associa-se a este operador o símbolo $+$.

Operador *NOT*

A operação *NOT* é a simples inversão do valor passado. Portanto, se sua entrada for 1, a saída é 0 e vice-versa como mostra a tabela na figura 2.3. Se uma variável é denominada A, então sua inversão é definida como $\overline{A}$.

2.1.2 Composição de Funções

Funções em Álgebra Booleana são criadas a partir de composições utilizando os três operadores básicos apresentados interespaçados pelas constantes 0, 1 ou variáveis. Um exemplo de função seria $f(A, B) = A \cdot \overline{B} + \overline{A} \cdot B$ cujo resultado se limita ao conjunto $\{0, 1\}$. Por consequência, uma característica favorável é conseguir representar essas funções a partir de tabelas verdade, como mostra a tabela 2.1 para a função descrita. Essa função também é conhecida como o operador *XOR* (*eXclusive OR*) binário.

Tabela 2.1: Tabela verdade da função $f(A, B) = \overline{A} \cdot \overline{B} + A \cdot B$

A	B	$\overline{A}$	$\overline{B}$	$A \cdot \overline{B}$	$\overline{A} \cdot B$	$A \cdot \overline{B} + \overline{A} \cdot B$
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	0	1
1	1	0	0	0	0	0

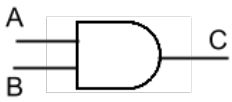
2.1.3 Circuitos Combinacionais

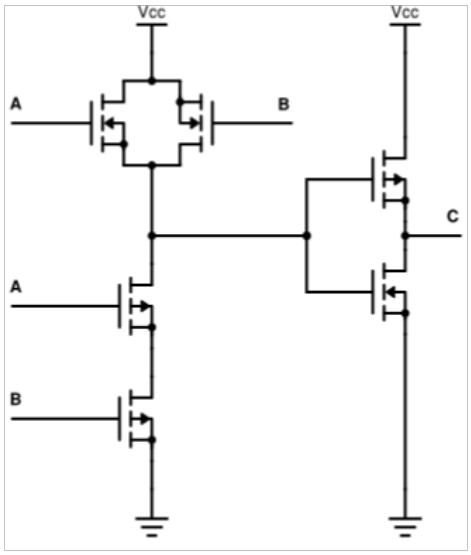
Outra maneira de representar funções booleanas é por meio de portas lógicas. Atualmente, circuitos digitais combinacionais, ou seja, aqueles cujas saídas dependem exclusivamente de suas entradas, são abstraídos dessa forma.

Representação física

Fisicamente, portas lógicas são implementadas utilizando-se elementos básicos de circuitos elétricos, como resistores, diodos e transistores (Agarwal and Lang [2005]). O nível lógico 1 é considerado como aproximadamente $5V$ de tensão, e para cerca de $0V$, considera-se como o nível lógico 0.

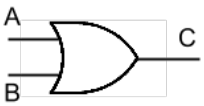
Utilizando as variáveis A e B para entradas e C para saída, as figuras fig. 2.1, fig. 2.2, fig. 2.3, fig. 2.4, fig. 2.5, fig. 2.6, fig. 2.7 mostram os símbolos das portas lógicas, uma possível construção física utilizando transistores bipolares e suas respectivas tabelas verdade.

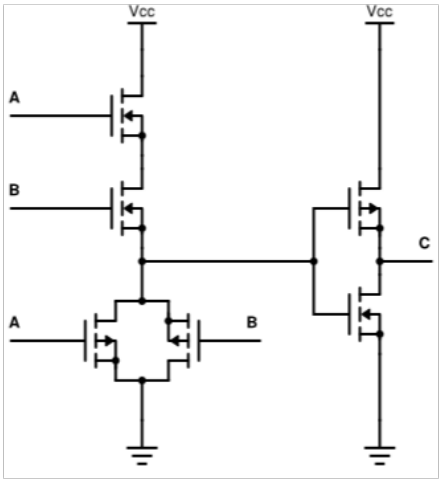




A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

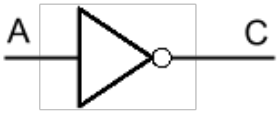
Figura 2.1: Porta lógica *AND*, circuito e tabela verdade.

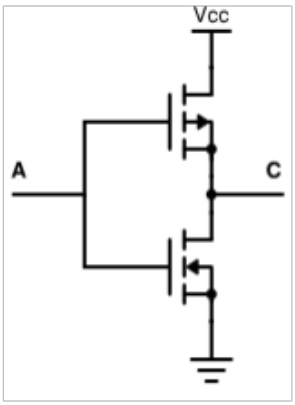




A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

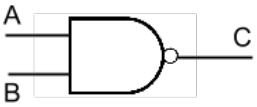
Figura 2.2: Porta lógica *OR*, circuito e tabela verdade.

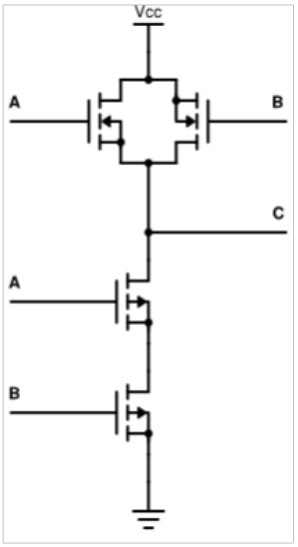




A	$\overline{A}$
0	1
1	0

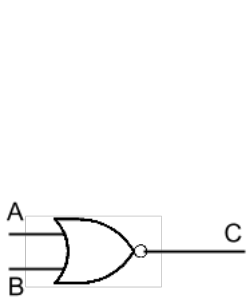
Figura 2.3: Porta lógica *NOT*, circuito e tabela verdade.

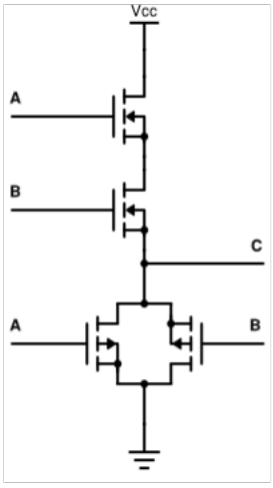




A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

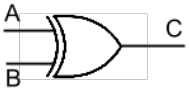
Figura 2.4: Porta lógica *NAND*, circuito e tabela verdade.

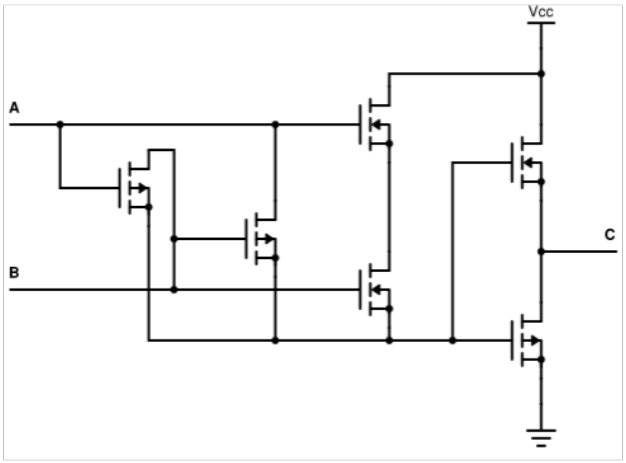




A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

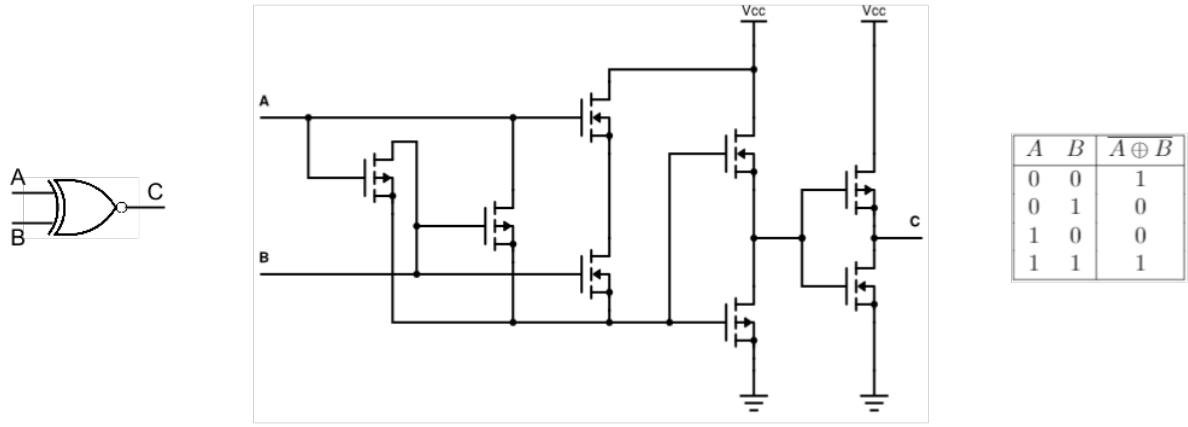
Figura 2.5: Porta lógica *NOR*, circuito e tabela verdade.





A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Figura 2.6: Porta lógica *XOR*, circuito e tabela verdade.



A	B	$\overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

Figura 2.7: Porta lógica *XNOR*, circuito e tabela verdade.

Observa-se que, dependendo da porta lógica, a sua implementação pode requerer diferentes números de transistores, sendo que a porta *NOT* a mais simples, e as portas *XOR* e *XNOR* as mais complexas.

Tempo de propagação

Enquanto expressões booleanas são suficientes para expressar a funcionalidade de circuitos digitais combinacionais, uma característica importante para *designs* mais complexos é o tempo de propagação propagation delay time (tpd). Ele pode ser definido como o tempo necessário para que, após uma mudança na entrada do circuito, a saída se estabilize, ou seja se torne válido, em 0 ou 1 (Agarwal and Lang [2005]).

Como exemplo, uma mudança na entrada A do circuito da figura 2.8 terá seu resultado refletido na saída R somente após a propagação do sinal estável em cada uma das portas lógicas *NOT*, resultando em um tempo de propagação total de:

$$tpd_{total} = tpd_{N_1} + tpd_{N_2} + tpd_{N_3} + tpd_{N_4}$$

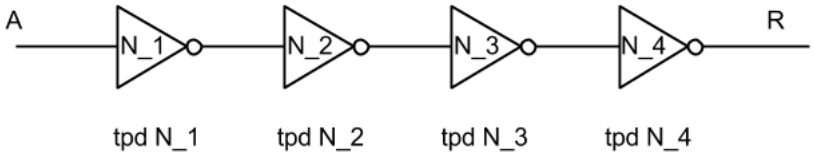


Figura 2.8: Circuito combinacional exemplificando tempo de propagação.

Deste modo, quanto mais camadas de portas lógicas são necessárias para a tarefa desejada, maior será o atraso para a obtenção da resposta, limitando assim a máxima frequência de utilização do circuito.

Tipicamente, fios possuem aproximadamente um atraso de $1ns$ para cada $15cm$ de comprimento, enquanto uma porta lógica pode variar de cerca de $10ns$ até à casa de picossegundos, dependendo da tecnologia utilizada (Balch [2003]).

2.1.4 Autossuficiência

Autossuficiência para o caso de portas lógicas significa que é possível formar qualquer função booleana com uma combinação delas. Para isto, basta mostrar que é possível que sejam geradas as portas *AND*, *OR* e *NOT*, que compõem um dos conjuntos completos de operações, a partir de uma única porta lógica. Se isso for verdade, então, já que se pode representar todas as funções booleanas com apenas essas 3 funções (Nelson et al. [1995]), a porta lógica em questão também o faz. Esse tipo de característica também é conhecido como porta lógica universal.

É o caso da função *NAND*, funcionalmente idêntica a $f(A, B) = \overline{A \cdot B}$. Sua tabela verdade pode ser vista na tabela da figura 2.4. O operador *NOT* é reproduzido a partir da função de apenas *NANDs* como $f(A) = \overline{A \cdot A}$. A função *OR* pode ser defininida como $g(A, B) = \overline{\overline{A \cdot A} \cdot \overline{B \cdot B}}$. Para *AND*, define-se $h(A, B) = \overline{\overline{A \cdot B} \cdot \overline{A \cdot B}}$.

Outro exemplo é o da função *NOR* (tabela verdade na figura 2.5). Sua representação em funções booleanas se dá por uma porta *OR* seguida de uma *NOT*, $f(A, B) = \overline{A + B}$. Para mostrar que *NOR* também possui essa completude funcional, sua representação para portas *OR* se dá como $\overline{\overline{A + B} + \overline{A + B}}$. Para portas *AND*, $\overline{\overline{A + A} + \overline{B + B}}$ e, *NOT*, $\overline{A + A}$.

Portas *NAND* e *NOR* são desejáveis não só por serem funcionalmente completas, mas também por sua facilidade de fabricação quando comparadas a outras portas lógicas. Como consequência, elas são utilizadas no conjunto das portas básicas em muitas famílias de *IC*s de lógica digital (Mano [2006]).

2.1.5 Forma Canônica

A criação de funções booleanas se mostra ser bastante flexível e diversificada. Porém, nem todas são únicas. É possível que duas funções distintas produzam a mesma saída, como por exemplo $f(A, B) = A \cdot B$ e $g(A, B) = \overline{\overline{A} + \overline{B}}$.

Neste caso, há uma infinidade de funções booleanas para uma dada tabela verdade. Torna-se necessário uma padronização visando apenas uma função booleana possível para cada configuração de tabela verdade.

Isto pode ser alcançado utilizando-se de formas canônicas de expressões lógicas. Há duas formas principais de para expressões canônicas, conhecidas como a forma soma de produtos e a forma produto de somas (Agarwal and Lang [2005]). A forma de utilização mais comum é o formato de soma de produtos, também chamado de soma de *mini-termos*.

Para um circuito com entradas $A_1, A_2, \ldots, A_n$, um *mini-termo* é um produto em que cada variável de entrada ou seu complemento aparecem apenas uma vez (Nelson et al. [1995]). Para um circuito com entradas A, B, C e D, um mini-termo possível é $A \cdot B \cdot \overline{C} \cdot D$, enquanto que $\overline{A} \cdot \overline{B} \cdot C$ não pode ser considerado um. Para a tabela verdade 2.2, só há a função canônica $f(A, B, C) = A \cdot B \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot \overline{C}$ que a representa.

Tabela 2.2: Tabela verdade com mini-termos da função canônica $f(A, B, C) = A \cdot B \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot \overline{C}$

A	B	C	$f(A, B, C)$	Mini-termo
0	0	0	1	$\overline{A} \cdot \overline{B} \cdot \overline{C}$
0	0	1	0	
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	0	
1	1	0	1	$A \cdot B \cdot \overline{C}$
1	1	1	0	

2.1.6 Minimizando Funções Booleanas

Muitas vezes, quando se projeta um circuito combinacional, é desejável obter sua função mínima, ou seja aquela cujo número de operações é o menor possível, para que seja utilizado apenas o necessário, economizando recursos. Para isto existem técnicas de minimização como *Mapas de Karnaugh*, *Método de Quine-McCluskey*, dentre outros (Gudise and Venayagamoorthy [2003]).

Para otimizações de implementação, outras técnicas têm surgido, principalmente envolvendo conceitos de programação genética que serão discutidas mais adiante neste trabalho. Esta seção se limita somente à minimização lógica.

Mapas de *Karnaugh*

Mapas de *Karnaugh* são uma maneira de representar uma tabela verdade de tal modo que ela exponha características redundantes de um circuito. Eles se utilizam da forma canônica para suas construções. Suas células são compostas de valores 0 ou 1, sendo cada 1 um mini-termo desejado da função. Como exemplo, a função $f(A, B, C) = \overline{A} \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot C$ tem uma tabela verdade como mostra a tabela 2.3 e sua representação em mapa de *Karnaugh* se dá como mostra a tabela 2.4.

Tabela 2.3: Tabela verdade com mini-termos da função $f(A, B, C) = \overline{A} \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot C$

A	B	C	$f(A, B, C)$	Mini-termo
0	0	0	0	
0	0	1	1	$\overline{A} \cdot \overline{B} \cdot C$
0	1	0	0	
0	1	1	1	$\overline{A} \cdot B \cdot C$
1	0	0	0	
1	0	1	1	$A \cdot \overline{B} \cdot C$
1	1	0	0	
1	1	1	0	

Tabela 2.4: Mapa de *Karnaugh* de $f(A,B,C) = \overline{A}\cdot B\cdot C + \overline{A}\cdot\overline{B}\cdot C + A\cdot\overline{B}\cdot C$

$f(A,B,C)$:

B \ A,C	00	01	11	10
0	0 (0)	1 (1)	1 (5)	0 (4)
1	0 (2)	1 (3)	0 (7)	0 (6)

A tabela 2.5 exibe células que podem ser otimizadas da tabela 2.4. Isso ocorre pois o mapa de *Karnaugh* foi projetado para que células adjacentes difiram somente em um valor da entrada. Dessa forma, é possível retirar termos desnecessários da expressão lógica, se utilizando do fato que $A + \overline{A} = 1$ e $B + \overline{B} = 1$.

Uma das otimizações marcadas na tabela 2.5 que pode ser feita é sobre os termos $\overline{A}\cdot B\cdot C$ e $\overline{A}\cdot\overline{B}\cdot C$. Em ambos, a única variável que se altera é B. Por causa disso, é possível juntá-los, resultando no termo $\overline{A}\cdot C$. O mesmo ocorre para $\overline{A}\cdot\overline{B}\cdot C$ e $A\cdot\overline{B}\cdot C$, ao retirar a variável A de ambos, o resultado é $\overline{B}\cdot C$.

No final, após todas as otimizações terem sido feitas, somam-se todos os termos gerados. No exemplo da tabela 2.5, seu circuito mínimo é $\overline{A}\cdot C + \overline{B}\cdot C$.

Tabela 2.5: Mapa de *Karnaugh* exibindo células passíveis de otimização.

$f(A,B,C)$:

B \ A,C	00	01	11	10
0	0 (0)	1 (1)	1 (5)	0 (4)
1	0 (2)	1 (3)	0 (7)	0 (6)

A principal desvantagem de se usar mapas de *Karnaugh* é que o algoritmo funciona bem para até 6 variáveis de entrada, tornando-se impraticável para valores maiores. O número excessivo de células dificulta uma seleção razoável de saídas adjacentes (Mano [2006]).

Método de *Quine-McCluskey*

Uma função booleana também pode ser expressa de uma forma numérica. Cada mini-termo é representado por um número, sendo ele a conversão binária de seus dígitos para decimal. Por exemplo, o mini-termo de quatro variáveis $\overline{A}\cdot B\cdot C\cdot D$ é expressado como 0111 em base binária, resultando no número 7 na base decimal.

Uma função booleana, portanto, também pode ser expressa como um somatório de mini-termos em sua representação decimal. Usando como exemplo a função da tabela 2.3, temos que ela pode ser expressa como $\sum m_1 + m_3 + m_5$, também sendo representada como $\sum m(1,3,5)$.

O método de *Quine-McCluskey*, também conhecido como método dos implicantes primos, visa amenizar a dificuldade vista para o método de *Karnaugh* para otimização de

funções com mais de seis variáveis. Ele consiste de duas partes: a primeira é achar todos os termos implicantes primos, ou seja, aqueles que são candidatos para inclusão no resultado final simplificado. A segunda é escolher, dentre esse termos, os que dão a expressão com o menor número de implicantes primários (Mano [2006]).

Para determinar quais termos são primos implicantes, cada mini-termo é comparado com todos os outros. Se eles diferem em apenas uma variável, ela é removida e um novo termo é formado. Isso se repete até que não seja mais possível combinar mais termos.

A tabela 2.6 demonstra isso para a função da tabela 2.3.

Tabela 2.6: Tabela de implicantes primos para a função $\sum m(1,3,5)$.

Número de 1s	Mini-termo	Binário	Implicante (Tamanho 2)
1	m_1	001	$m(1,3)0-1$ $m(1,5)-01$
2	m_3 m_5	011 101	

O passo seguinte monta o gráfico dos implicantes primos, como pode ser visto na tabela 2.7.

Tabela 2.7: Gráfico de implicantes primos para a função $\sum m(1,3,5)$. Construído após o primeiro passo do algoritmo de *Quine-McCluskey*.

	1	3	5	Implicante	Expressão
$m(1,3)$	X	X		$0-1$	$\overline{A}\cdot C$
$m(1,5)$	X		X	-01	$\overline{B}\cdot C$

Ao final, todos os termos encontrados são somados. Para este exemplo, como visto na tabela 2.7, o resultado é $\overline{A}\cdot C+\overline{B}\cdot C$, chegando então na mesma expressão encontrada pelo método de *Karnaugh*.

2.2 Lógica Sequencial

Até agora, a saída dos circuitos vistos na seção anterior dependiam apenas de suas entradas. Circuitos sequenciais utilizam-se tanto de circuitos combinacionais quanto de circuitos de retroalimentação. Isso possibilita a aplicação de uma abstração chamada *estado*. Dessa forma, torna-se muito comum a utilização e produção de chamadas *máquinas de estado*. A figura 2.9 mostra esse tipo de abordagem.

Um exemplo comum de circuito sequencial é o contador. Um contador é utilizado para a numeração de tíquetes de uma fila bancária, por exemplo.

2.2.1 Elementos Básicos

Para que circuitos consigam manter estados, uma estrutura básica denominada *flip-flop* é utilizada. Nesta seção, será mencionado apenas o *flip-flop* tipo D, porém outros tipo podem ser encontrados.

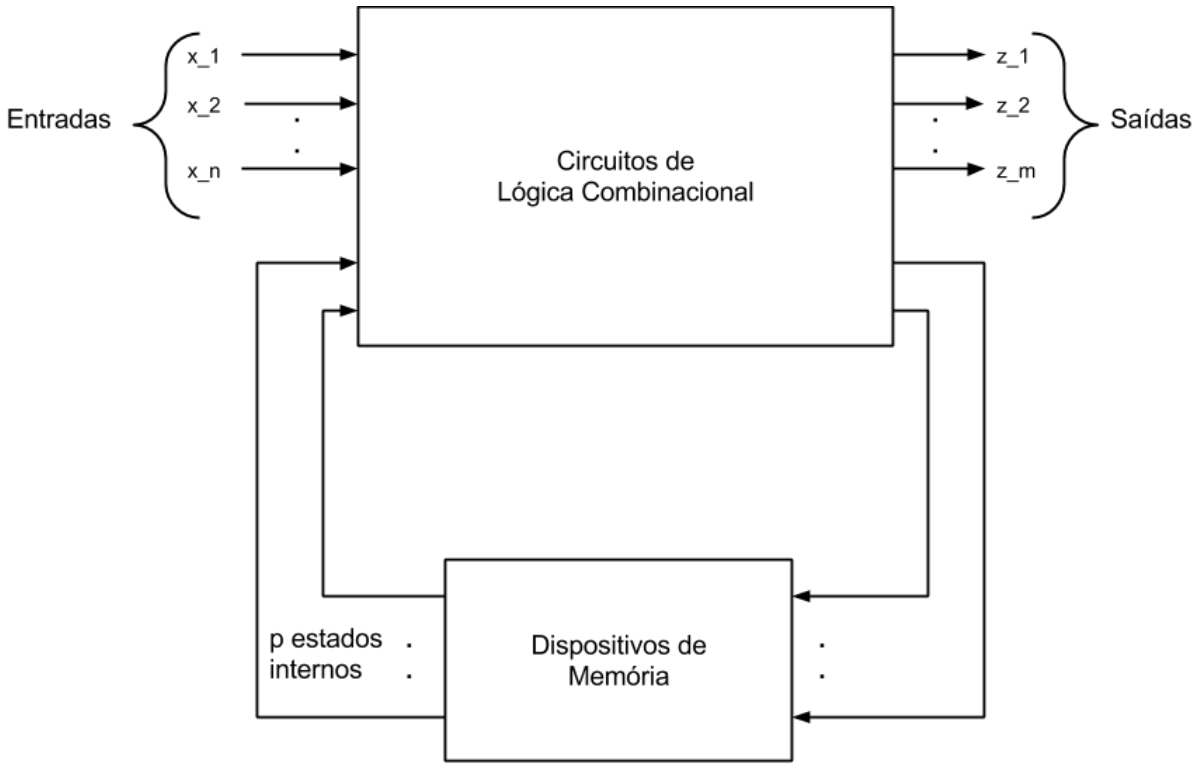


Figura 2.9: Estrutura conceitual de uma máquina de estados.

O *flip-flop* é um elemento capaz de guardar o valor de 1 *bit* (Tarnoff [2007]). Um pequeno circuito retroalimentado pode ser usado para alcançar este objetivo, apresentado na figura 2.10.

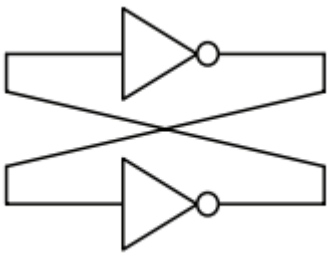


Figura 2.10: Estrutura realimentada estável independente de entradas (Tarnoff [2007]).

É possível ver que as saídas de ambas as portas está estável, tornando-as ideais para guardarem valores. A desvantagem desse circuito é a impossibilidade de alteração do valor armazenado, já que é completamente auto-contido.

A porta *NAND*, apresentada na seção 2.1.4, atua como um simples inversor se uma de suas saídas for mantida em 1. Dessa maneira, é possível replicar a estrutura da figura 2.10 utilizando portas *NAND* como mostra a figura 2.11.

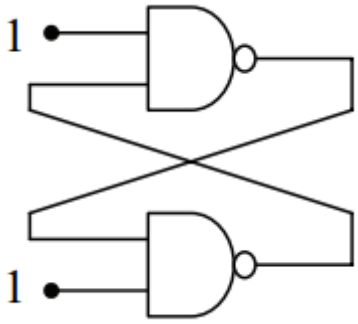


Figura 2.11: Estrutura de retroalimentação utilizando portas NAND (Tarnoff [2007]).

Alterando os valores de uma das entradas, é possível alterar a saída das portas lógicas, com o circuito se estabilizando novamente no momento em que as duas entradas retornarem para 1.

Caso a entrada superior receba o rótulo S e a inferior R, correspondente às saídas Q e $\overline{Q}$, o elemento é chamado de *latch RS*, e é utilizado na maior parte de circuitos que necessitam armazenar dados. Seu diagrama pode ser visto na figura 2.12.

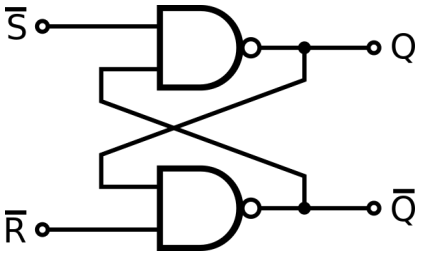


Figura 2.12: Diagrama de um *latch RS* (Wikipedia [2014b]).

Latch D

Um possível módulo que pode ser formado pelo *latch RS* é o *latch* tipo D, que tem a interface definida na figura 2.13, onde a saída Q é o valor armazenado no circuito, E é o valor que determina se o valor D é armazenado no *latch*.

A entrada E funciona como uma porta. Se estiver no nível 1 (aberta), qualquer alteração no valor D causa uma alteração na saída do circuito. Se estiver em 0 (fechada), o valor da saída se mantém estável mesmo que o valor D se altere. Seu comportamento completo pode ser visto na tabela verdade 2.8.

Flip-flop D

Latches são normalmente categorizadas como sensíveis a nível, ou seja, se seu nível de *enable* estiver em alto, todas as alterações de D são vísiveis na saída do *latch*. *Flip-flops*, diferentemente, só causam alterações na saída nas transições $0 \rightarrow 1$ ou $1 \rightarrow 0$ do *enable*, também chamado de *clock* no caso de *flip-flops*. A essas duas categorias dão-se os nomes de *positive edge triggered* e *negative edge triggered* (Tarnoff [2007]).

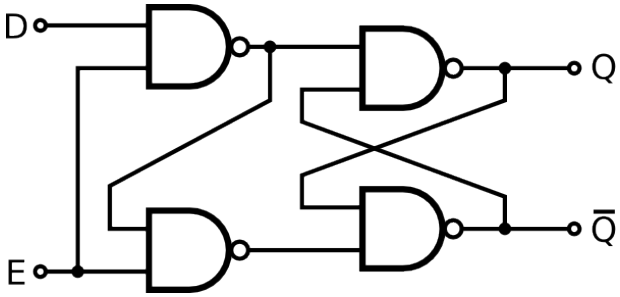


Figura 2.13: Diagrama de um *latch* D (Wikipedia [2014b]).

Tabela 2.8: Tabela verdade da estrutura *latch* D.

Entradas		Saídas	
E	D	Q	$\overline{Q}$
0	X	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

A estrutura interna de um *flip-flop* D de *negative edge*, ou seja, com mudança de saída apenas durante a transição $1 \rightarrow 0$, pode ser vista na figura 2.14.

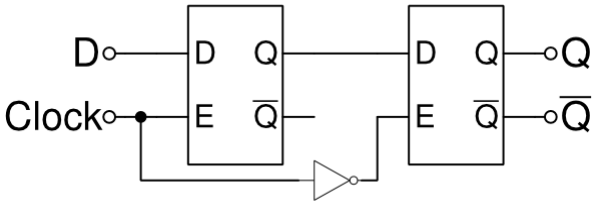


Figura 2.14: Diagrama de um *flip-flop* D (Wikipedia [2014b]).

Este tipo de *flip-flop*, em especial, é chamado de *master-slave*. Com o circuito montado dessa maneira, sua tabela verdade pode ser vista na tabela 2.9.

Tabela 2.9: Tabela verdade do *flip-flop* tipo D.

Entradas		Saídas	
Clock	D	Q	$\overline{Q}$
$1 \rightarrow 0$	0	0	1
$1 \rightarrow 0$	1	1	0
0	X	Q	$\overline{Q}$
1	X	Q	$\overline{Q}$

2.2.2 Máquina de Estados Finitos

A abstração e modelagem de problemas que se utilizam de circuitos sequenciais normalmente se dá por meio das chamadas máquinas de estados.

Uma máquina de estados pode ser definida como uma quíntupla $M = (Y, X, Z, \alpha, \beta)$. Y é o conjunto de todos os estados existentes na máquina. X é o conjunto de entradas para a máquina. Z, o conjunto de todos os estados de saída possíveis da máquina. α é a função $\alpha : (Y \times X) \rightarrow Y$, que define o próximo estado com base no estado atual e entrada atual. A função β define a saída atual da máquina, sua definição formal depende da classificação da máquina como Moore ou Mealy (Belgasem [2003]).

Máquina de Moore

Para máquinas de Moore, a função β é definida como:

$$\beta : Y \rightarrow Z,$$

ou seja, a saída atual do circuito depende apenas de seu estado atual.

Máquina de Mealy

No caso de máquinas de Mealy, a saída atual da máquina depende tanto do estado atual quanto da entrada atual. Ou seja, sua função β é da forma

$$\beta : (Y \times X) \rightarrow Z.$$

Comparação

Máquinas de estados são frequentemente representadas utilizando grafos especiais chamados Diagramas de Transições e Estados (Belgasem [2003]). Um exemplo de uma máquina de Mealy com uma entrada e duas saídas pode ser visto na figura 2.15, onde cada nó representa um estado (com seu rótulo) e os números nas arestas indicam as entradas e as saídas associadas. Sua tabela de transições associada está descrita na tabela 2.10.

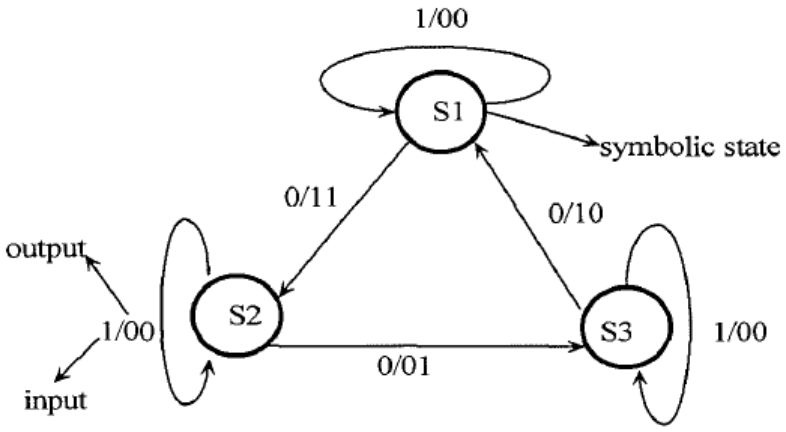


Figura 2.15: Exemplo de Diagrama de Transições e Estados (Belgasem [2003]).

Tabela 2.10: Exemplo de tabela de transição de estados.

Entrada	Estado Atual	Próx. Estado	Saída
0	S_1	S_2	11
0	S_2	S_3	01
0	S_3	S_1	10
1	S_1	S_1	00
1	S_2	S_2	00
1	S_3	S_3	00

A mesma máquina implementada utilizando o paradigma de Moore teria um diagrama de transições e estados como mostra a figura 2.16, com uma tabela de transições descrita na tabela 2.11.

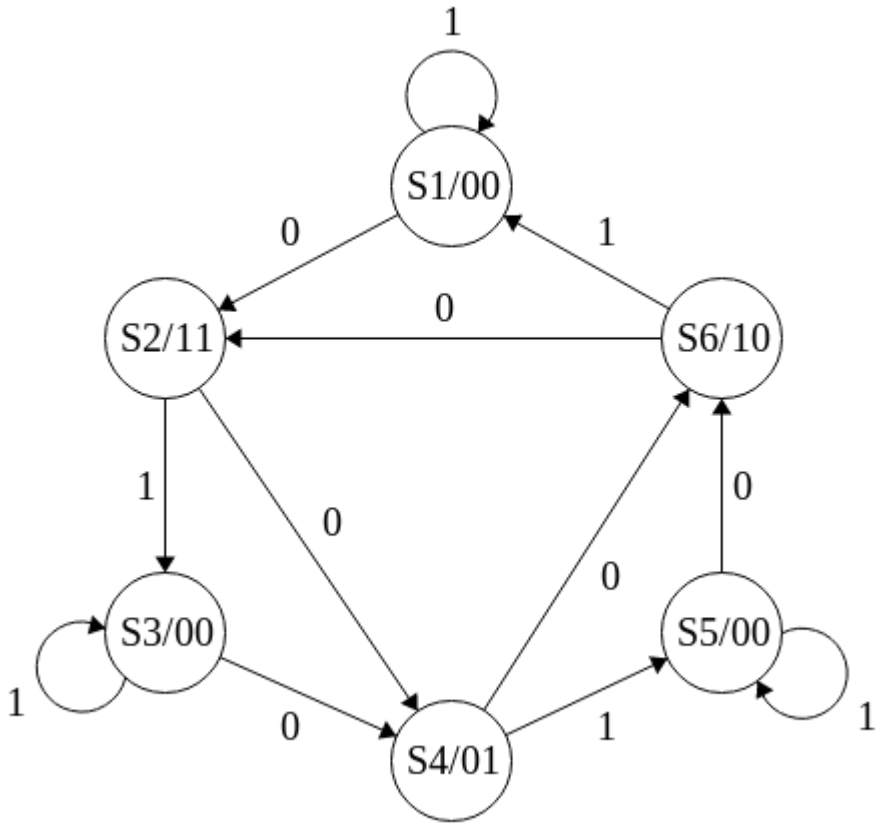


Figura 2.16: Exemplo de Diagrama de Transições e Estados utilizando o paradigma de Moore.

Em geral, máquinas de Moore possuem uma saída mais simples de ser entendida, já que depende apenas de um fator. Além disso, são mais seguras pois se a entrada for assíncrona com o *clock* da máquina, então ocorrem condições de corrida. Em contrapartida, máquinas de Mealy são significativamente mais compactas, possuindo menos estados em geral.

Tabela 2.11: Tabela de transições de estados para o exemplo utilizando o paradigma de Moore.

Estado Atual/Saída	Prox. Estado	
	0	1
$S_1/00$	S_2	S_1
$S_2/11$	S_4	S_3
$S_3/00$	S_4	S_3
$S_4/01$	S_6	S_5
$S_5/00$	S_6	S_5
$S_6/10$	S_2	S_1

2.2.3 Processo de *Design*

De acordo com Belgasem [2003], o processo de projeto de um circuito sequencial, assim como a abordagem tradicional para outros problemas, é dividir a especificação em sub-etapas. Cada uma deve ter, idealmente, pouca ou nenhuma relação com as outras. O objetivo é conseguir, a partir da especificação de um problema, desenvolver uma máquina de estados finitos que o resolva. Um esquema de como isso é normalmente feito está na figura 2.17.

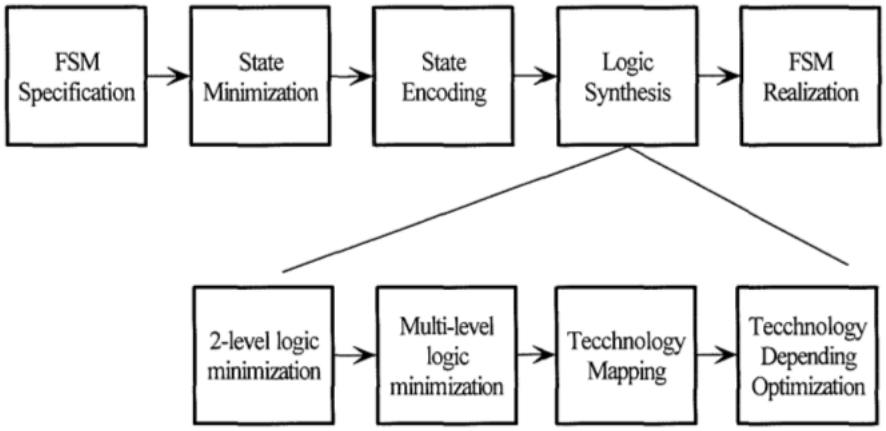


Figura 2.17: Processo de projeto de um circuito sequencial (Belgasem [2003]).

A razão disso vem do fato de que o problema de sintetizar um circuito sequencial é um problema considerado difícil. Belgasem [2003] cita a grande quantidade de possíveis implementações possíveis para um dado problema, além de outros fatores como área que o circuito ocupa e seu gasto de energia para a determinação de uma boa solução.

Especificação da Máquina

Este passo consiste na configuração inicial desenvolvida pelos projetistas do circuito. Normalmente contém o diagrama de estados e transições com uma tabela de transições de estados.

Minimização de Estados

Muitas vezes, a máquina inicialmente gerada não possui um número ótimo de estados. Um método clássico para realizar isso é incorporar *don't cares*, ou seja, valores que em um dado instante não são importantes, na tabela de transições e, consequentemente, no diagrama de transições e estados.

Em contrapartida, outras abordagens de minimização focam não só no número de estados mas na complexidade lógica da minimização ótima, Avedillo et al. [1991] menciona que nem sempre é desejável se ter o número mínimo de estados.

Codificação de Estados

Em implementações de máquinas de estados, os estados são representados por *strings* de *bits*. O problema que estuda a relação de mapeamento entre estados e *strings* de *bits* que possui o menor custo é chamado de Problema da Codificação de Estados (Amaral et al. [1995]).

Para essa etapa, diversos autores já tentaram diferentes abordagens (Belgasem [2003]). Em geral, são utilizadas heurísticas para solucionar esse tipo de problema. Dentre elas, algoritmos genéticos foram usados com sucesso (Amaral et al. [1995]), porém o problema ainda é considerado difícil.

Síntese do Circuito

Atualmente, existe uma vasta gama de ferramentas dedicadas a síntese a partir de uma descrição da máquina. As formas mais comuns de se descrever circuitos sequenciais são pelas linguagens denominadas Hardware Description Language (HDL), como *VHDL* e *Verilog* (Belgasem [2003]).

As máquinas são simuladas e testadas em dispositivos específicos denominados reconfiguráveis. Com os objetivos alcançados, é feito um Application-Specific Integrated Circuit (ASIC) a partir da descrição resultante.

A seguinte seção trata de um dos tipos mais utilizados de dispositivos reconfiguráveis, extremamente útil para o desenvolvimento de circuitos não só sequenciais, mas de forma geral.

2.3 Dispositivo Reconfigurável - FPGA

Um Field-Programmable Gate Array (FPGA) é um dispositivo lógico que consiste de um arranjo bidimensional de chamadas células lógicas e chaves programáveis. Cada célula lógica pode ser programada para exercer uma função booleana, cujo valor é disponibilizado para células vizinhas através dessas chaves (Chu [2008]). Células especiais, principalmente as que estão na borda do arranjo, se conectam a portas destinadas a canais de I/O.

Em um dado momento, a configuração da FPGA é determinada por uma memória *on-chip*, que pode ser modificada por *software*. Apesar da configuração do circuito não se modificar, as mudanças realizadas a essa memória fazem com que as ligações entre células lógicas e suas configurações se modifiquem, portanto, exercendo a função de um novo circuito (Thompson [1998]), como mostra a figura 2.18.

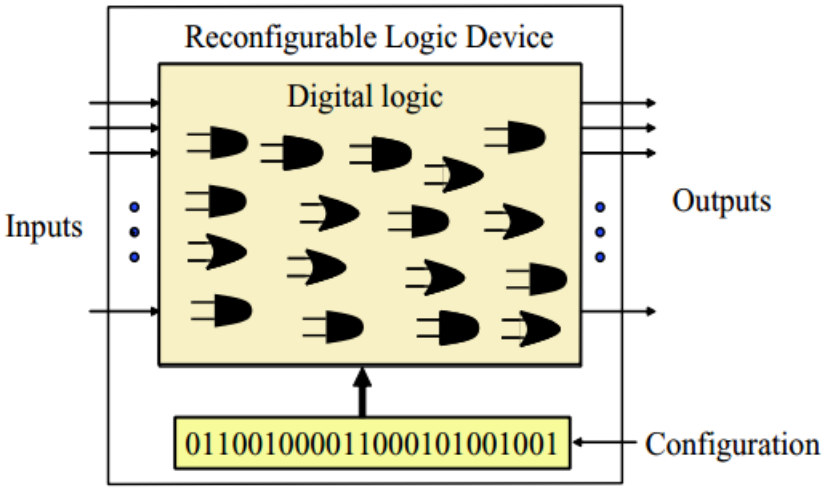


Figura 2.18: Configuração de memória mapeando para um circuito em um dispositivo reconfigurável (Torresen [2004]).

Cada célula lógica tipicamente consiste de uma Look-up Table (LUT) configurável de 3 ou 4 entradas, ou seja uma tabela verdade que descreve uma função booleana, e um *flip-flop* tipo-D. Por exemplo, a célula da família de FPGAs *Cyclone II* pode ser vista na figura 2.19.

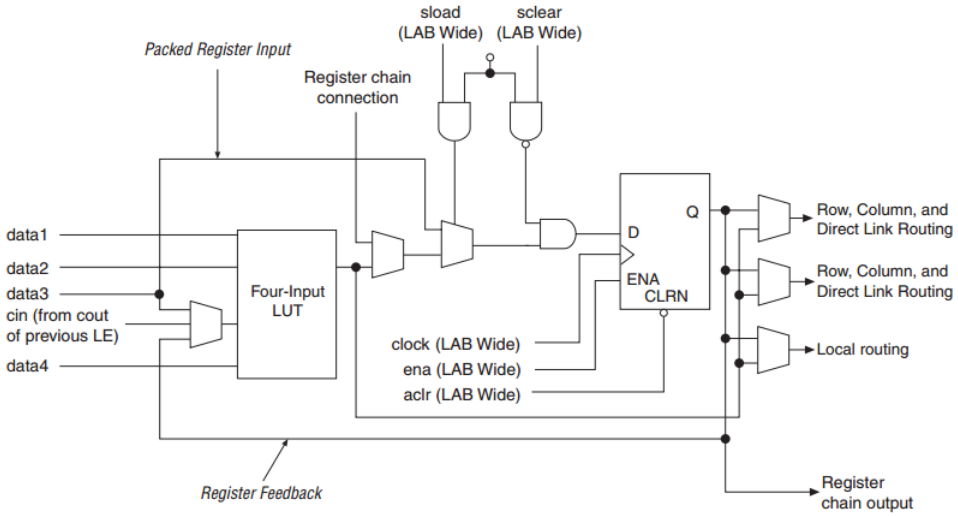


Figura 2.19: Logic Element (LE) da família de FPGAs *Cyclone II* (Altera [2007]).

FPGAs pertencem a uma classe de dispositivos chamada Field-Programmable Logic (FPL) (Meyer-Baese [2007]), pois o processo de programação pode ser feito várias vezes após a produção do dispositivo.

A figura 2.20 mostra conceitualmente como o arranjo de células é feito.

Existe uma variedade grande de diferentes tipos de dispositivos reconfiguráveis. Para este trabalho, o foco será apenas em FPGAs por suas facilidades de reconfiguração apresentadas.

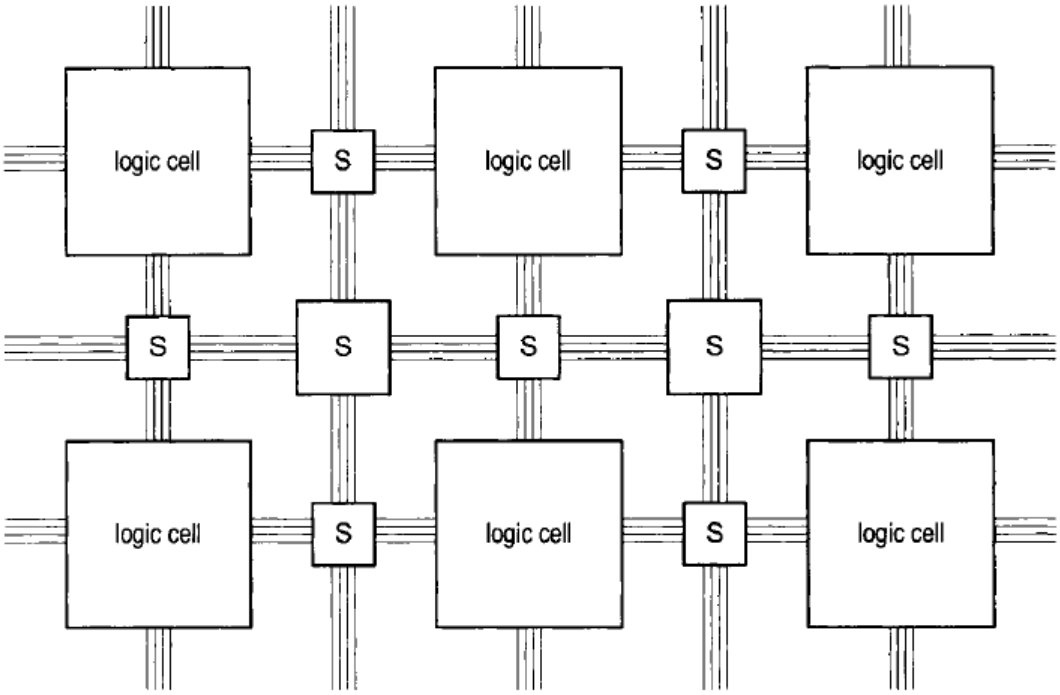


Figura 2.20: Arranjo conceitual de elementos lógicos e chaves programáveis (Chu [2008]).

2.3.1 FPGA Altera Cyclone IV EP4CE115

Esta seção detalhará brevemente a FPGA Altera Cyclone IV EP4CE115 que será utilizada para este trabalho. Consiste de 114,480 elementos lógicos básicos cujas estruturas podem ser vistas na figura 2.21. É possível observar que o elemento lógico básico constitui de LUTs de 4 entradas, blocos de memória e multiplicadores (Altera [2013]).

Cada elemento lógico está contido em um conjunto denominado Logic Array Block (LAB). A troca de informações dentro de um mesmo LAB ocorre por um barramento local, como mostra a figura 2.22. Esse barramento também é usado para trocar informações com estruturas adjacentes.

Além disso, outras características relevantes ao *chip* são:

- 3888 K*bits* de memória embarcada;
- 266 Multiplicadores 18×18 *bits*;
- 4 circuitos PLL de livre uso;
- 528 pinos de I/O para usuário.

A FPGA está integrada à placa de desenvolvimento *Altera DE2-115*, cujo *layout* pode ser visto na figura 2.23.

O *chip* principal apresenta diversas interfaces para dispositivos externos. Dentre outros é possível listar (Altera [2010]), para memórias:

- 128MB (32Mx32*bit*) SDRAM;

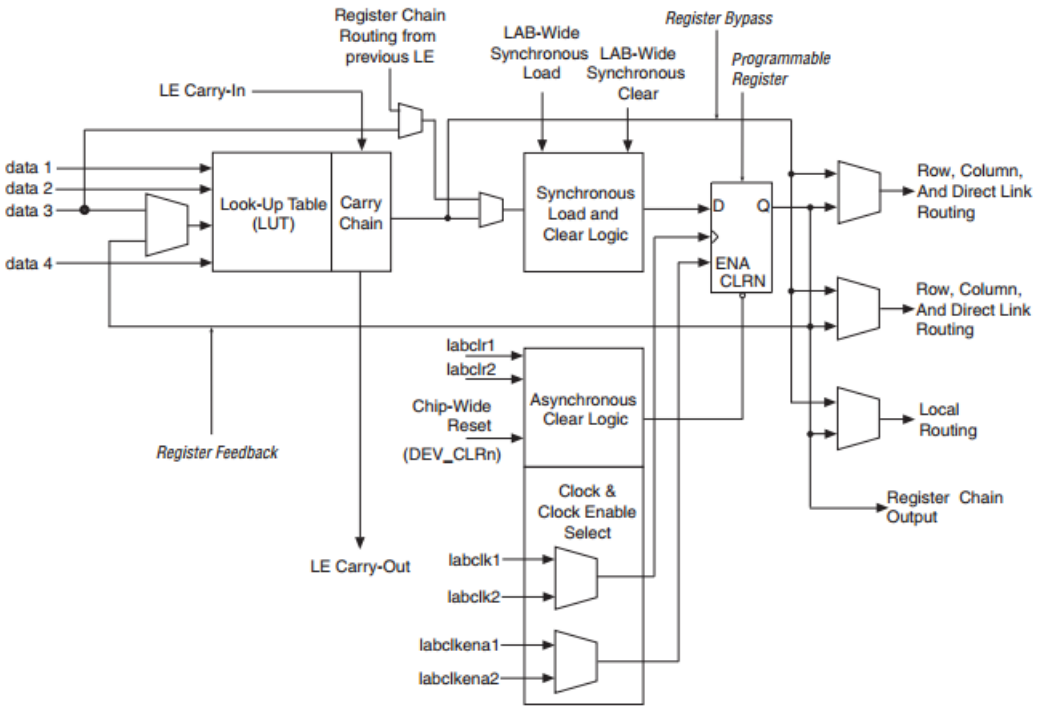


Figura 2.21: LE da família de FPGAs *Cyclone IV* (Altera [2013]).

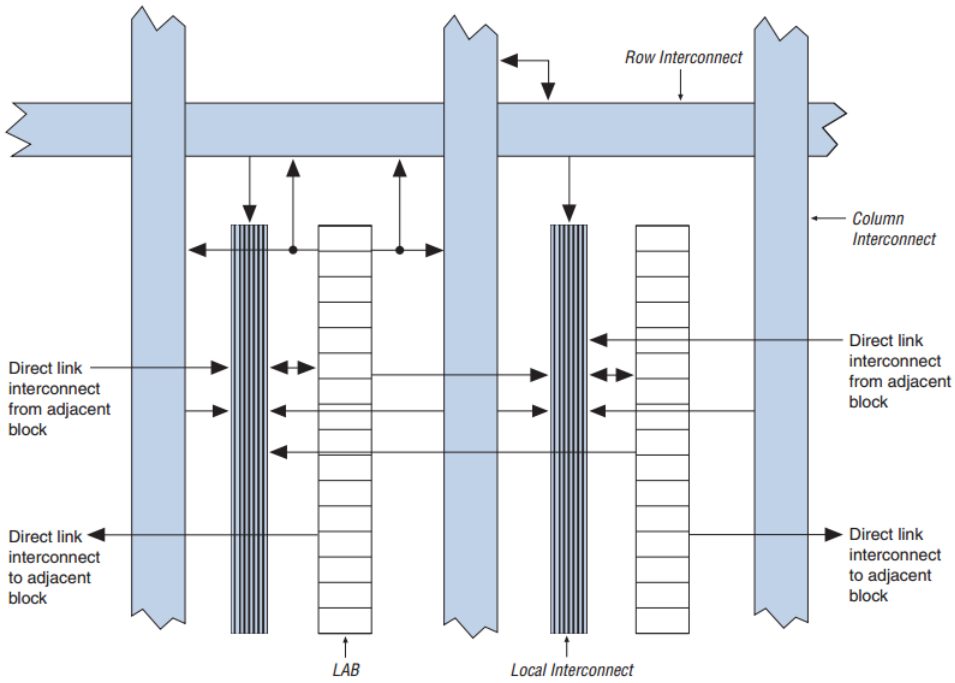


Figura 2.22: LAB com barramento de interconexão local (Altera [2013]).

- 2MB (1Mx16) SRAM;
- 8MB (4Mx16) *Flash* com modo 8-*bits*;
- 32KB EEPROM.

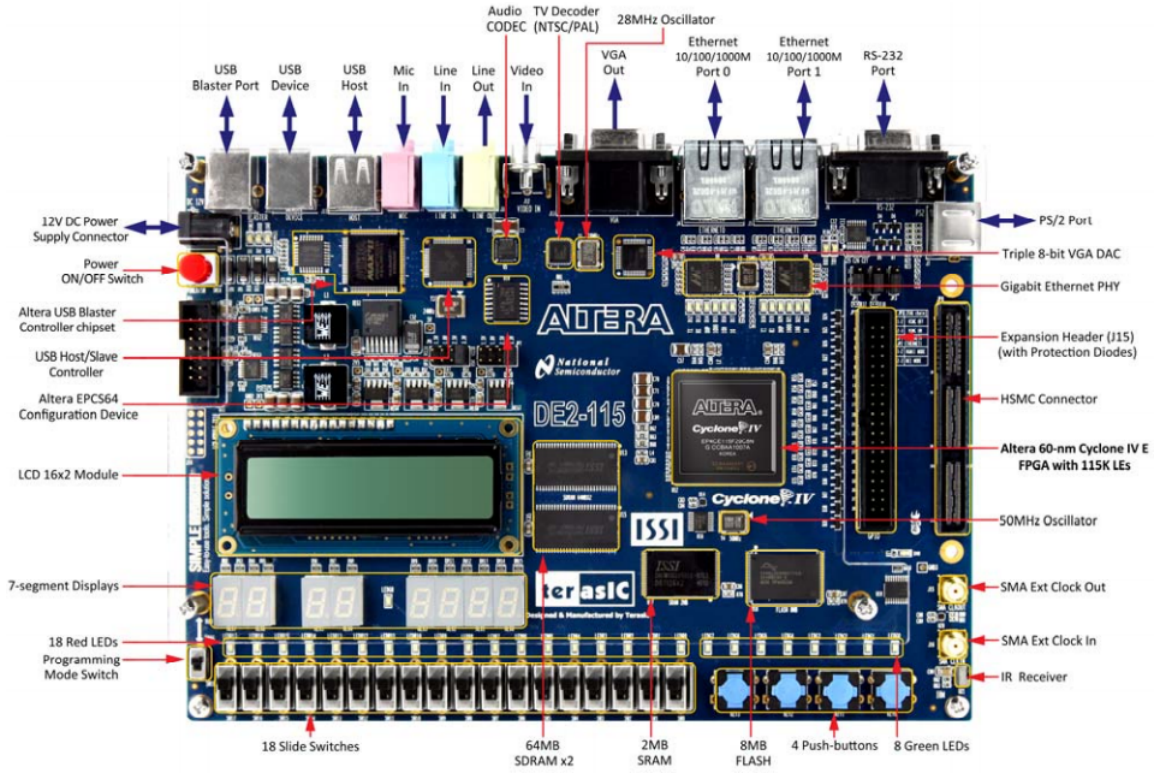


Figura 2.23: *Layout* da placa de desenvolvimento *Altera DE2-115* (Altera [2010]).

Três *clocks* de 50MHz, conectores USB tipos A e B, uma porta de expansão de 40 pinos.

Para interface direta com o usuário,

- 18 chaves deslizantes e 4 de botão;
- 18 LEDs vermelhos e 9 verdes;
- 8 *displays* de 7 segmentos.

Todos esses recursos tornam a placa de desenvolvimento e a FPGA ferramentas convenientes para projetar circuitos. Com a capacidade de gerar qualquer sistema computacional e ser reconfigurável torna este ambiente ideal para rápidas iterações de configurações, tarefa necessária para que este estudo seja viável.

Processo de *Design*

Empresas fabricantes de FPGAs tipicamente oferecem *softwares* que facilitam o desenvolvimento em suas plataformas. Em geral, esses programas têm a responsabilidade de auxiliar a etapa de compilação (exemplo na figura 2.24) da descrição de um circuito para o *bitstream* de configuração específico de alguma FPGA, passando pelas fases de análise, síntese e montagem do projeto. Para os chips do fabricante *Altera*, esse *software* de desenvolvimento se chama *Quartus II*.

Após a compilação do circuito, esses programas costumam oferecer também mecanismos de *download* das descrições geradas para o *chip*, configurando-o com a funcionalidade desejada.

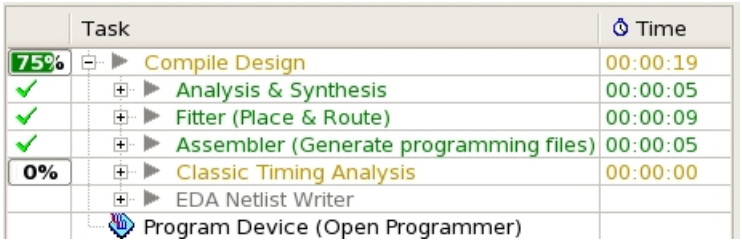


Figura 2.24: Etapas típicas de compilação para *bitstream* alvo.

2.4 Algoritmos Genéticos

De acordo com o trabalho de Darwin [1859], as características das espécies são determinadas não por intervenções divinas como se acreditava na época anterior à publicação desse trabalho, mas sim por um mecanismo denominado *seleção natural*.

Darwin [1859] argumenta que indivíduos mais adaptados de uma população têm maiores chances de se reproduzirem. Como consequência disso, seus genes têm uma probabilidade maior de serem passados para as gerações seguintes. Devido a isso, a cada nova geração, os genes da população tendem a se convergir para aquele que está mais adaptado ao ambiente.

Pode se ver que, aplicado em escalas de milhões de anos, esse mecanismo de seleção natural exibe características bastante interessantes nas espécies. Há muito se busca uma maneira de imitar esse tipo de comportamento e aplicá-lo a outros tipos de problemas, dando origem à área de Algoritmos Genéticos.

Conceito inventado por John Holland na década de 60 (Melanie [1999]), seu objetivo era estudar o fenômeno de adaptação que ocorre na natureza visando desenvolver mecanismos similares para sistemas computacionais.

Antes de aplicar uma abordagem genética a um problema, é preciso torná-lo apto a isso. Um dos primeiros requisitos é conseguir descrevê-lo com base em todos os parâmetros necessários, ou seja, suas possíveis soluções. Estas são denominadas os indivíduos da população. Cada indivíduo é representado por um cromossomo, ou seja, um conjunto de genes, que formam os parâmetros mencionados. Por sua vez, genes são representados por um conjunto de "alelos" possíveis, que no contexto computacional pode ser descrito como os dois valores válidos para um *bit*: 0 e 1. Dessa forma é possível resumir um indivíduo em uma lista de 0s e 1s.

2.4.1 Operadores Genéticos

Nesta seção, serão introduzidos operações comumente utilizadas em algoritmos genéticos.

Reprodução ou *Crossover*

Esta operação busca imitar o fenômeno da recombinação genética. Um ponto de separação do cromossomo é escolhido aleatoriamente e dois novos indivíduos são gerados a partir da permuta da informação genética entre os dois cromossomos selecionados. A figura 2.25 ilustra essa operação.

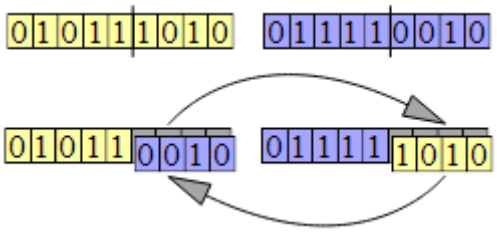


Figura 2.25: Exemplo de crossover entre dois cromossomos (Wikipedia [2014a]).

Mutação

A mutação visa simular a situação em que o cromossomo é alterado por erros durante a divisão celular. Essas mudanças são essenciais para o surgimento de novas características.

No caso de algoritmos genéticos, essa mudança se representa por uma probabilidade p_m. Em geral, a mutação atua na construção de novos indivíduos. Ou seja, em sua criação, há uma probabilidade de p_m de haver uma mutação. Mutações costumam alterar um ou mais alelos aleatórios como mostra a tabela 2.12.

Tabela 2.12: Exemplo de operação de mutação em um cromossomo.

Cromossomo Inicial	0110010111001011
Cromossomo Mutante	0110010111011011

Função de *fitness*

As funções de *fitness* têm como entrada um cromossomo e saída um valor de avaliação, dependendo do quão perto ele chega da solução, ou seja, atuam definindo aqueles que estão mais ou menos aptos a ter seus genes passados adiante. A função de *fitness* é considerada um dos pontos mais sensíveis de um algoritmo genético.

Uma vez definidos os operadores básicos, um algoritmo genético pode ser definido na série de passos apresentada na figura 2.26.

2.4.2 Operação de Seleção

Essa operação consiste em selecionar cromossomos mais adaptados para a reprodução. Quanto mais adaptado, maior é sua chance de ser escolhido, mas não há garantia de que isso ocorra. Dentre os diversos métodos de seleção, os mais populares são as estratégias de seleção por roleta ou por torneio descritos a seguir.

Seleção por Roleta

A seleção por roleta se assemelha bastante a uma roleta real de cassino, de onde seu nome é derivado. Em geral, nela, indivíduos que possuem *fitness* maiores possuem uma maior probabilidade de serem escolhidos, porém não se garante a escolha. A probabilidade é distribuída relativamente para cada indivíduo, ou seja, a soma de todas as probabilidades deve ser 1, e deve ser proporcional ao *fitness* de cada um.

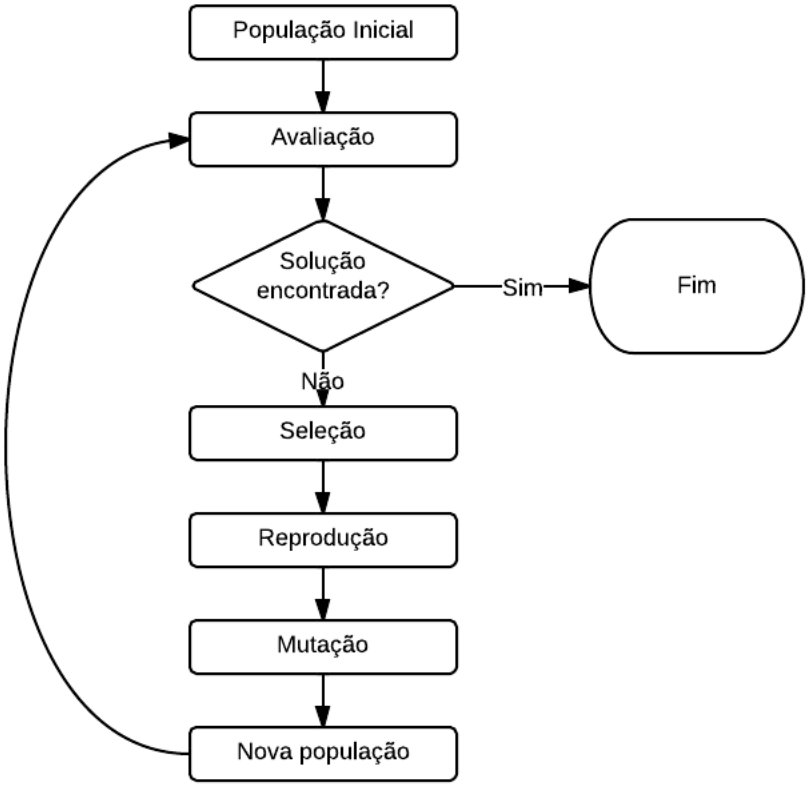


Figura 2.26: Passos gerais para um algoritmo genético.

Dessa maneira, a forma de se calcular o *fitness relativo* do n-ésimo indivíduo é dada por

$$f_r(x_n) = \begin{cases} \frac{f(x_n)}{\sum_{i=1}^{N} f(x_i)}, & \text{se } n = 1 \\ f_r(x_{n-1}) + \frac{f(x_n)}{\sum_{i=1}^{N} f(x_i)}, & \text{senão} \end{cases} \tag{2.1}$$

onde N é o tamanho da população.

Após o cálculo dos *fitness* relativos, gera-se um número aleatório $0 \leq r < 1$, escolhendo aquele indivíduo em que r pertence ao intervalo entre um *fitness* relativo e outro. Uma representação visual deste processo pode ser visto na figura 2.27.

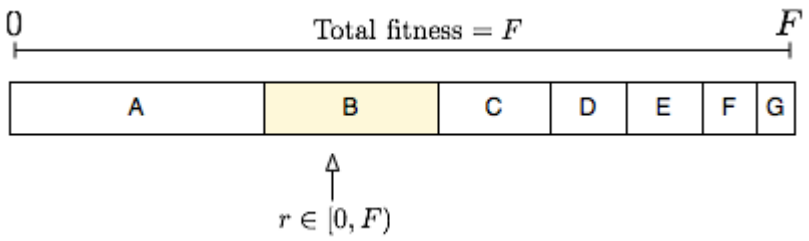


Figura 2.27: Exemplo da seleção por roleta (Wikipedia [2014c]).

Na figura 2.27, o individuo A possui o maior *fitness*, tendo portanto uma maior probabilidade de ser selecionado. Por outro lado, o indivíduo G é o menos adaptado, possuindo a menor probabilidade de seleção.

Seleção por Torneio

A seleção por torneio busca simular o fato de que uma população inteira, salvo poucas exceções, nunca está competindo contra si como um todo, mas sim por pequenas competições localizadas.

Miller et al. [1995] cita que algumas vantanges de implementações desse tipo de seleção são: a facilidade de escrita, eficiência do ponto de vista de paralelismo e providenciam um maior controle sobre a *pressão da seleção*. Essa pressão é definida como o grau em que indivíduos mais adaptados são favorecidos. Trabalhos anteriores como o de Miller et al. [1995] também citam que o sucesso de um algoritmo genético depende, em maior parte, da pressão da seleção. Se a pressão for muito baixa, a taxa de convergência será desnecessariamente baixa. Se for muito alta, é grande a chance de que a convergência ocorra em uma solução não-ideal.

Dada uma população P, e um tamanho de torneio k, a seleção por torneio seleciona um conjunto aleatório de k indivíduos de P para competirem entre si, e, para cada torneio, escolhendo aquele mais adaptado. Seu pseudo-código correspondente pode ser visto no algoritmo 1.

Input: P, k
Result: Indivíduo mais adaptado
$melhor.fitness \leftarrow 0$;
for *1 to* k **do**
  $competidor \leftarrow P[random()]$;
  if $competidor.fitness > melhor.fitness$ **then**
    $melhor \leftarrow competidor$;
  end
end
return *melhor*

Algoritmo 1: Algoritmo para seleção de um indivíduo por torneio.

A escolha do melhor indivíduo de uma população é realizada pela simples escolha do indivíduo com maior *fitness* em torneio de tamanho k.

Apesar do tamanho ideal do torneio variar de acordo com o problema (Xie and Zhang [2009]), normalmente são escolhidos valores não muito altos para k. De fato, muitas vezes $k = 2$ é o valor utilizado, chamado de torneio binário. Observe que para $k = 1$, a seleção por torneio se reduz a uma seleção aleatória.

2.5 Programação Genética Cartesiana

A Programação Genética Cartesiana (Cartesian Genetic Programming (CGP)), tem como característica principal a linearização de um cromossomo para representar uma rede, utilizando-se números inteiros. O termo *cartesiano* vem do fato do método consistir em uma grade bidimensional de nós (Miller [1999]).

O cromossomo é dividido em duas seções distintas: nós e saídas. Cada gene da seção de nós identifica dois componentes básicos de um nó:

1. A operação realizada pelo nó;
2. As entradas usadas para o cálculo.

A seção de saídas indica a que nó cada saída deve se ligar para produzir seu resultado.

A descrição que o cromossomo representa é conhecida como *genótipo*. O programa resultante da decodificação desse genótipo é chamado de fenótipo. Para CGP, enquanto o genótipo possui tamanho fixo, descrevendo completamente a grade, o fenótipo efetivo varia com o número de nós utilizados desde zero até todos os disponíveis no genótipo (Miller [2011]).

2.5.1 Forma Geral

Se n_i é o número de entradas necessárias para o programa e n_o o número de saídas, um sistema genético cartesiano de $N = n_r \times n_c$ nós, cada qual com r entradas, tem seu endereçamento de cromossomos da seguinte maneira:

$$F_0 C_{0,0} \cdots C_{0,r-1} F_1 C_{1,0} \cdots C_{1,r-1} \cdots F_{N-1} C_{n_c * n_r,0} \cdots C_{n_c * n_r,r-1} O_0 \cdots O_{n_o-1}$$

onde n_r indica o número de linhas, n_c o número de colunas, F_k é a função que o nó de número k exerce, C é o endereço de onde provém o valor de entrada e os valores O_l indicam de qual nó a saída l retira seu valor.

Um parâmetro adicional l indica o número de colunas anteriores que a coluna j pode usar para valores de entradas de seus nós, limitando sua escolha de colunas para $[max(0, j-l), \cdots, (j-1)]$. Dessa forma, não são permitidas retroalimentações, gerando uma topologia com apenas ligações *feedforward*. Uma visualização gráfica da forma geral pode ser vista na figura 2.28.

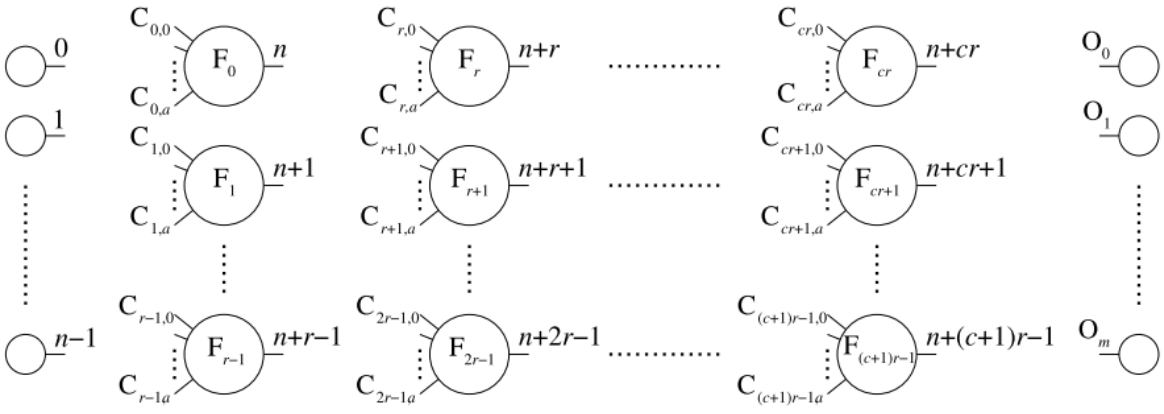


Figura 2.28: Forma geral da estrutura de um sistema genético cartesiano (Miller [2011]).

O exemplo da figura 2.29 mostra um sistema com $n_i = 6$, $n_o = 2$, $n_r = 2$, $n_c = 3$, $r = 2$ e $l = 2$. É possível observar a tradução que ocorre entre a descrição do genótipo para o fenótipo. No fenótipo, é instanciado um grafo de elementos lógicos, com suas ligações e

funções definidas pelo genótipo. As saídas do circuito constituem os dois últimos elementos do genótipo.

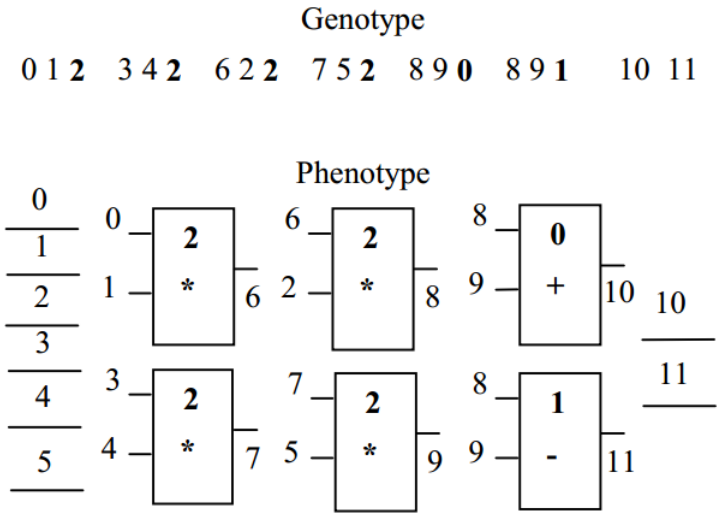


Figura 2.29: Exemplo de sistema cartesiano genético. Evidencia relação entre genótipo e fenótipo (Miller [1999]).

2.5.2 Restrição de Valores

Para a variável F, que indica a função do nó, seja n_f o número de funções possíveis do sistema, então

$$0 \leq F < n_f.$$

Para as entradas do nó $C_{i,j}$, deve se respeitar o valor l de níveis atingíveis. Portanto,

$$n_i + (j - l)n_r \leq C_{i,j} \leq n_i + jn_r$$

para $j \geq l$ e

$$0 \leq C_{i,j} \leq n_i + jn_r$$

para $j < l$. Para os valores de saída O, segue-se a seguinte restrição:

$$0 \leq O < n_i + N.$$

Tipicamente, conforme apresentado na seção 2.5.1, sistemas genéticos cartesianos não possuem ciclos. Para que o objetivo de liberação de restrições seja melhor atendido, não será usada a variável l que indica o número de colunas anteriores atingíveis. Para este trabalho colunas poderão usar quaisquer outras como possíveis valores. Dessa forma, torna-se possível a retroalimentação, estrutura imprescindível em circuitos capazes de armazenar dados, tais como *latches* e *flip-flops*.

2.5.3 Estratégia evolucionária $(1 + \lambda)$

Primeiramente definido por Beyer [1993] o algoritmo $(1 + \lambda)$ para geração e seleção de uma nova população vem sendo utilizado pela comunidade científica (Hilder et al. [2010], Walker et al. [2009]) como escolha de estratégia evolutiva em problemas CGP. Miller et al. [2000] o descreve como uma estratégia eficiente e de implementação simples. Seu nome vem do fato de haver somente 1 indivíduo, gerando λ outros indivíduos por mutações. Seu algoritmo está descrito no algoritmo 2.

Input: B, cromossomo com maior *fitness*.
Result: Melhor indivíduo encontrado.
$C \leftarrow \lambda$ mutações de B;
$M = \{c | c \in C, c.fitness \geq B.fitness\}$;
if $M \neq \emptyset$ **then**
  $B \leftarrow max(M)$;
end
return B

Algoritmo 2: Algoritmo que descreve a estratégia $(1 + \lambda)$.

Uma visualização gráfica pode ser vista na figura 2.30, onde $\lambda = 4$. Nela, o indivíduo 1, que gera suas λ mutações, está sempre no topo da gerações. Na geração 0, como nenhum indivíduo gerado a partir do 1 possui um *fitness* igual ou maior a ele, ele é passado como o 1 da seguinte geração. Na geração 1, um dos indivíduos gerados possui um *fitness* igual a seu gerador, tornando-o indivíduo 1 da geração 2. Esse processo se repete até que o objetivo desejado seja alcançado.

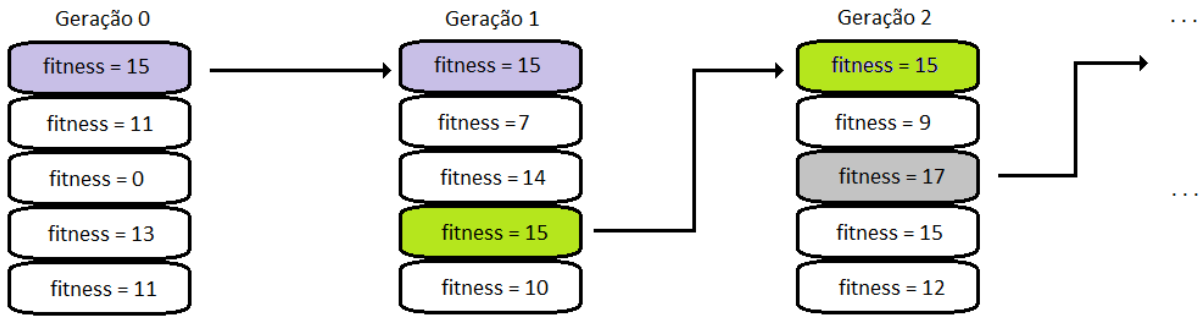


Figura 2.30: Exemplo de execução do algoritmo $(1 + \lambda)$.

A principal diferença entre esse algoritmo e os descritos anteriormente é que seu operador principal é a mutação, não havendo operações de *crossovers*. Todos os λ indivíduos são gerados a partir de mutações do individuo pai. De acordo com Miller et al. [2000], tipicamente $\lambda = 4$.

O algoritmo $(1 + \lambda)$ é naturalmente elitista. O indivíduo com maior *fitness* sempre é preservado para gerações seguintes.

2.5.4 Neutralidade

CGP representa o genótipo como uma lista de tamanho fixo de números inteiros que indicam as propriedades de cada nó em um grafo. O fenótipo resultante, porém, é resultado da aplicação das ligações desses nós no genótipo. Portanto, o fenótipo possui um tamanho variável, havendo então genes inativos, ou seja, que não possuem efeito algum na avaliação de seu *fitness*. À existência de genes inativos dá-se o nome de *neutralidade*.

O fenômeno da neutralidade permite que o cromossomo se mova pelo espaço de soluções sem afetar sua avaliação, e já foi minuciosamente estudado por (Walker et al. [2009], Miller and Smith [2006], Vassilev and Miller [2000]), entre outros. Os estudos realizados chegaram à conclusão que a neutralidade é um efeito extremamente benéfico à eficiência do processo evolutivo.

2.5.5 *MC-CGP*

Conceito introduzido por Walker et al. [2009], Multi-Chromosome Cartesian Genetic Programming (MC-CGP) difere da representação CGP convencional por dividir o genótipo em seções de tamanho igual chamadas de *cromossomos*. Cada cromossomo é ligado a uma saída do circuito. Isso permite que problemas com múltiplas saídas sejam divididos em vários subproblemas menores de uma saída apenas. A ideia principal da proposta é tornar a convergência mais fácil.

Cada cromossomo contém uma quantidade igual de nós e é tratado como um problema distinto a ser resolvido. A figura 2.31 mostra um exemplo de um problema que possui 4 saídas, portanto dividindo o cromossomo total em 4 seções. Cada cromossomo c_0, c_1, c_2, c_3 possui uma saída correspondente $O_{c_0}, O_{c_1}, O_{c_2}, O_{c_3}$ respectivamente.

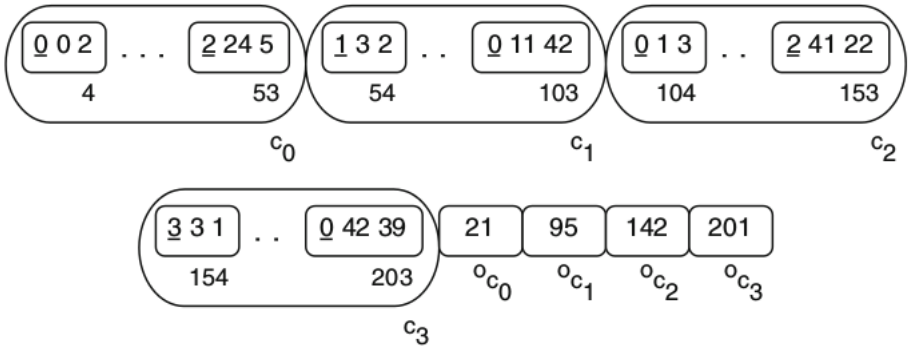


Figura 2.31: Exemplo exibindo comportamento isolado de cada cromossomo em um ambiente MC-CGP (Walker et al. [2009]).

Devido à independência de cada subproblema, sua avaliação paralela reduz em parte a natureza sequencial dos algoritmos genéticos, permitindo uma implementação eficiente em hardware.

2.6 Hardware Evolutivo

Nesta seção, as duas áreas distintas descritas nas seções 2.3 e 2.4 serão unidas em um nova, recentemente fundada, chamada *Hardware Evolutivo*.

Atualmente, a indústria de produção de circuitos eletrônicos atinge uma vasta gama de aplicações. Dessa forma, há uma crescente demanda de circuitos cada vez mais complexos. Técnicas como as apresentadas na seção 2.1.6 são úteis para o projeto de circuitos combinacionais. Por outro lado, o projeto de circuitos sequenciais, que representam uma grande parte das aplicações, pouco tiram proveito (Belgasem [2003]). Consequentemente, inovações em técnicas de projeto são pesquisadas diariamente. Dentre elas, apresentam-se os chamados sistemas bio-inspirados (Gordon and Bentley [2002]).

Hardware Evolutivo pode ser definido como um circuito ou sistema capaz de modificar sua arquitetura e, portanto, seu comportamento dinamicamente e de forma independente por meio de interações com seu ambiente (Belgasem [2003]).

2.6.1 Aplicações

A área de *hardware evolutivo* busca resolver problemas de maneira equivalente àquela descrita na seção 2.4. Ou seja, através da inspiração em mecanismos da natureza, busca-se obter um *hardware* mais flexível.

Tolerância a Falhas

Nenhum sistema se mantém estático. Falhas, internas ou externas (interferência eletromagnética, por exemplo), são eventualmente introduzidas ao longo do tempo, podendo em algum momento, em conjunto com outros fatores, surgir como erros. Esse problema é agravado quando posto em paralelo com a necessidade de sistemas cada vez mais complexos. Falhas podem, então, surgir em diversas partes, tornando impraticável a verificação total e constante do circuito.

Dessa maneira, no projeto de um sistema, pode-se utilizar da abordagem evolutiva para embutir falhas no processo da seleção, como mudanças de temperatura para melhor se adaptar a diferentes ambientes (Thompson [1998]).

Canham and Tyrrell [2002] evoluíram um oscilador, mostrando certa viabilidade para esse tipo de método. Utilizando uma simples função de *fitness* que levava em conta apenas a saída do circuito. Após um indivíduo atingir um nível aceitável, integrou-se um novo fator para a seleção. Uma falha era introduzida, um elemento lógico aleatório tinha sua saída travada para 1 ou 0. Uma queda súbita no *fitness* médio foi observada, porém ao longo das gerações o nível anterior se restabeleceu. O resultado foi um circuito robusto, com certa redundância e boa resistência a falhas.

Soluções Inovadoras

Uma das principais motivações para se utilizar algoritmos genéticos para desenvolver circuitos vem do fato de não conhecermos todo o espaço de soluções possíveis. De fato, *design* de circuitos lida com um conjunto infinito de possíveis soluções, principalmente para aqueles que podem ser classificados como sequenciais, como a seção 2.2 descreve.

Projetistas frequentemente se utilizam de abstrações para achar soluções de problemas. Por exemplo, no *design* de processadores de propósito geral, a menor abstração utilizada tipicamente se mantém no nível de portas lógicas. Portas lógicas, porém, são formadas por um conjunto de transistores, que por sua vez, possuem sua própria abstração física. Seria completamente impraticável o projeto de algo assim levando em conta os absolutos mínimos detalhes.

Por não lidar com essas abstrações impostas para a conveniência de projetistas, *hardware* evolutivo possui um espaço de soluções ainda não completamente explorado a sua disposição. A figura 2.32 ilustra a afirmação.

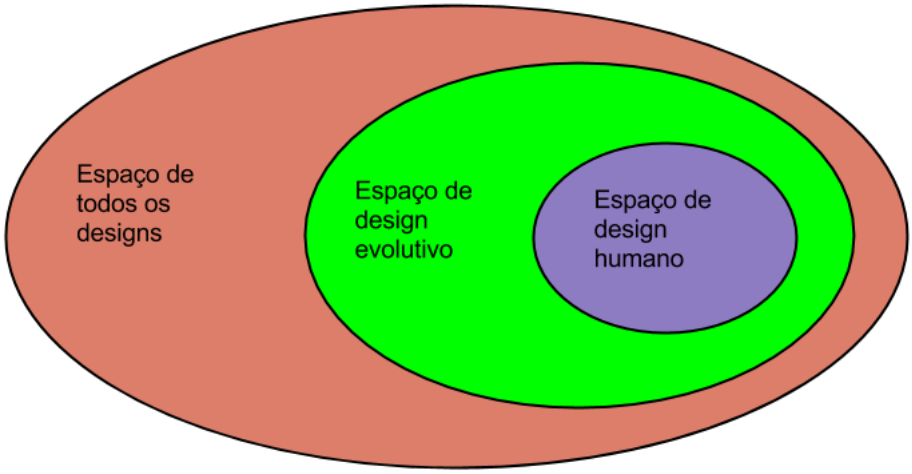


Figura 2.32: Espaço de soluções e alcances das abordagens evolutiva e humana.

Um detalhe deve ser destacado, porém. Enquanto a remoção de restrições pode ressaltar soluções antes não vistas, é possível também que não se chegue a ela em um tempo prático ou que nem se chegue nela (Belgasem [2003]). É importante encontrar um equilíbrio entre liberdade e restrição para que uma solução eficiente seja encontrada.

2.6.2 Formas de Evolução

Para *hardware* evolutivo, existem duas abordagens principais para a projeção do circuito final.

Extrínseca

Na abordagem extrínseca, o fenótipo dos circuitos é avaliado em *software*, com o resultado final, então, sendo implementado em um circuito real. Como simular todas as características do mundo real é computacionalmente muito caro, *softwares* geralmente optam por se utilizarem de modelos de abstrações com características bem definidas para evitar cálculos mais detalhados (Thompson [1998]).

Intrínseca

Na abordagem intrínseca, a avaliação dos indivíduos da população é feita no ambiente físico determinado, normalmente uma FPGA. Dessa forma, cada indivíduo é carregado, testado e avaliado enquanto está sendo executado na placa. Também chamada de evolução *on-line* (Torresen [1997]).

2.6.3 Evoluindo Circuitos

Neste trabalho, circuitos serão evoluidos utilizando o método geral mostrado na figura 2.33. O cromossomo de cada indivíduo é mapeado para a descrição das ligações dos elementos lógicos da FPGA. Após a avaliação de cada indivíduo, uma nova população é formada utilizando algum critério de seleção (2.4.2). Esse processo se repete até que uma solução aceitável seja alcançada.

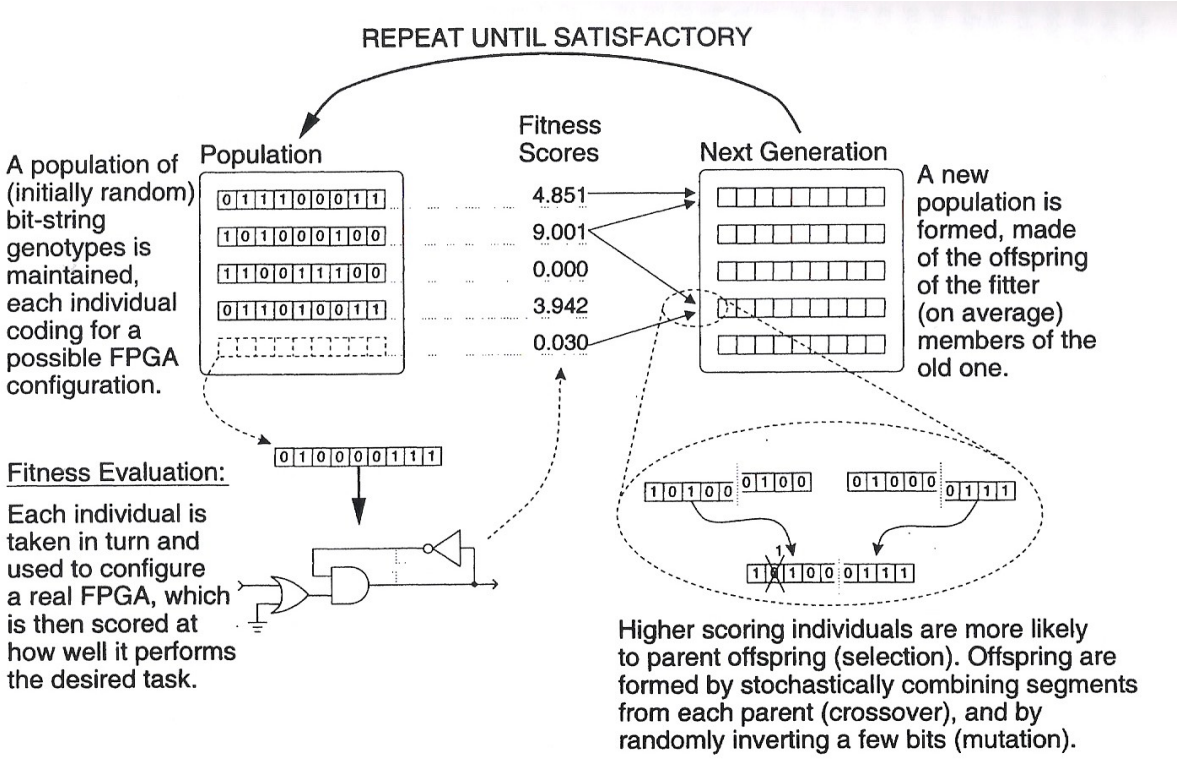


Figura 2.33: Ciclo geral da evolução de circuitos intrînsecos (Thompson [1998]).

2.6.4 Níveis de Abstração e Granularidade

A classificação de *hardware* se dá por camadas de abstração, como se pode ver na figura 2.34. De maneira similar, a abordagem evolutiva em *hardware* normalmente acontece em uma dessas abstrações.

De acordo com Stoica [1997], o nível de granularidade pode ser dividido entre as classes extrínseca e intrínseca.

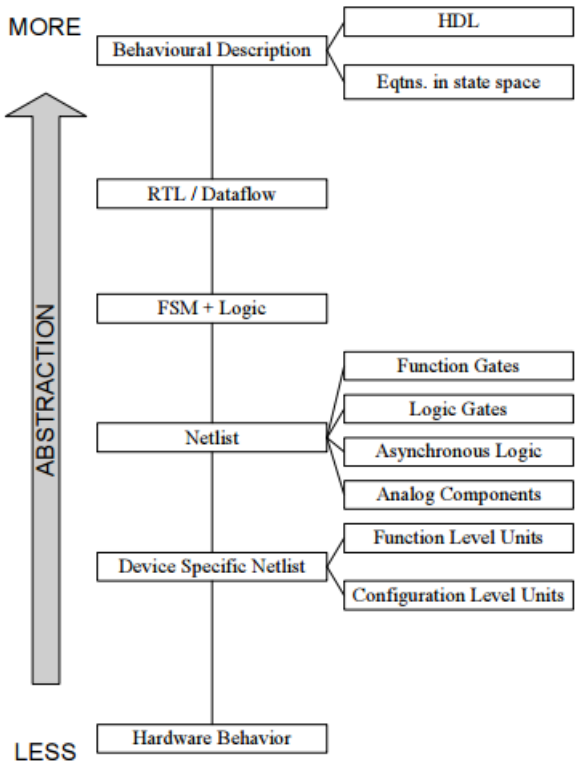


Figura 2.34: Níveis de abstração presentes em classificações de *hardware* (Gordon and Bentley [2002]).

Granularidades para Classe Extrínseca

As abstrações definidas por Stoica [1997] para circuitos simulados em *software* são:

- Dispositivos Primitivos – Geralmente considerado no nível de descrição de transistores por meio de tabelas e equações.
- Macro-modelo Funcional – Nível de abstração que se utiliza de portas lógicas como menor objeto.
- Comportamental – Utiliza-se de linguagens de descrição de *hardware* (HDLs).

Granularidades para Classe Intrínseca

Para circuitos de fato executados, uma outra lista de abstrações que dita o bloco construtor é associada:

- Nível de configuração – Nível que descreve a configuração de cada elemento lógico em FPGAs.
- Transistor – Descreve a ligação de dispositivos analógicos, como o transistor.
- Portas Lógicas – Bloco básico descreve portas lógicas (2.1).
- Funcional – Blocos descrevem circuitos mais complexos, como multiplexadores, ou funções matemáticas.

2.6.5 Limitações em *Hardware* Evolutivo

Torresen [2004] relata alguns problemas sobre escalabilidade conhecidos na área de *hardware* evolutivo. Dentre eles está o problema do tamanho do cromossomo.

Para se conseguir representar circuitos complexos, um cromossomo de tamanho considerável deve ser utilizado. A consequência disso é uma abertura muito grande do espaço de soluções, tornando a aplicação dessa técnica viável, atualmente, apenas a circuitos pequenos. Dessa forma, a maior parte das pesquisas atuais têm um foco maior em acelerar o processo de evolução de um circuito.

Diferentes abordagens têm surgido para amenizar esse problema. Uma delas é utilizar cromossomos de tamanho variável. Outra envolve utilizar abstrações maiores. Uma técnica chamada evolução incremental também pode ser usada. Nela, o circuito alvo é separado em diferentes módulos, com cada um sendo evoluído separadamente, o que traz um espaço de buscas mais simples e menor quando comparada a evolução do circuito completo sem separações.

Neste capítulo, foram apresentados conceitos básicos necessários para o entendimento do restante do trabalho.

Foram vistos fundamentos sobre a Álgebra Booleana na seção 2.1, necessária para modelar circuitos utilizando equações. Além disso, foram vistas técnicas existentes de minimização de equações booleanas.

Foi apresentada a área de circuitos sequenciais na seção 2.2, seus blocos básicos e como eles podem ser representados por máquinas de estado.

Também foram vistos dispositivos reconfiguráveis na seção 2.3 e suas características, com um foco especial em FPGAs e como elas podem auxiliar o desenvolvimento de circuitos.

Sobre a área de algoritmos genéticos na seção 2.4, suas operações básicas e suas relações entre si foram apresentadas. Uma técnica mais especializada, chamada de Programação Genética Cartesiana e descrita na seção 2.5, muito utilizada para representar circuitos digitais. MC-CGP foi introduzido e comparado ao CGP convencional como forma de ajuda ao processo evolutivo.

Finalmente, todos esses conceitos são unidos na seção 2.6, que apresenta e descreve a área de *Hardware* Evolutivo, mostrando as principais características e limitações, como tempo viável para a convergência de soluções, que serão abordadas nos capítulos seguintes.

Capítulo 3

Metodologia Proposta

Este trabalho tem como objetivo principal avaliar a abordagem genética no desenvolvimento e otimização de circuitos digitais.

Neste capítulo serão detalhados todos os passos feitos para o desenvolvimento dos experimentos e desenvolvimento da proposta deste trabalho. Primeiramente, todos os diferentes métodos de avaliação de indivíduos são descritos, explicitando dificuldades e soluções encontradas. Em seguida, a estratégia de evolução proposta é apresentada.

3.1 Avaliação do indivíduo

As seguintes subseções tratam da etapa de cálculo de *fitness* do indivíduo. Essa etapa faz parte do passo *Avaliação*, visto na figura 2.26. O problema é como, a partir da representação de um indivíduo por seu cromossomo, avaliar seu grau de aptidão.

3.1.1 Manipulação direta do *Bitstream*

Dado que o dispositivo FPGA é configurado através de um *stream* de *bits*, é natural a utilização desse *bitstream* como cromossomo de um determinado indivíduo. Desde modo a avaliação de sua aptidão se reduz à aplicação das entradas e verificação das correspondentes saídas do circuito configurado. A manipulação genética do cromossomo necessita a modificação direta desse *bitstream*. Nos FPGAs do fabricante *Altera*, o *bitstream* somente pode ser gerado pelo uso do *software Quartus II*, a partir de uma descrição do circuito desejado.

Diversos problemas foram encontrados para a aplicação dessa abordagem, dentre eles:

- Como saber se uma determinada configuração poderá ou não causar danos ao dispositivo FPGA (Thompson [1998]);
- Informações insuficientes acerca da formação do *bitstream* de configuração por parte do fabricante do dispositivo;

Tentou-se obter informações sobre o *bitstream* através de engenharia reversa e análise dos arquivos gerados pelo software de síntese, porém não se obteve nenhum padrão que pudesse ser facilmente identificado.

Embora seja a solução mais natural de aplicação dos modelos genéticos em dispositivos FPGA, devido às dificuldades encontradas, o método não pôde ser adotado.

3.1.2 Modelo em *Verilog*

A segunda proposta visa, a partir do cromossomo, criar um modelo do indivíduo usando a linguagem de descrição de hardware *Verilog*. Desta forma, o *bitstream* do indivíduo pode ser gerado através do software de síntese *Quartus II*.

O modelo *Verilog* do indivíduo é compilado juntamente com um circuito de interface. Esse circuito tem a responsabilidade de gerenciar a entrada de dados para a avaliação do indivíduo e enviar sua saída para o computador *host*, de acordo com a figura 3.1.

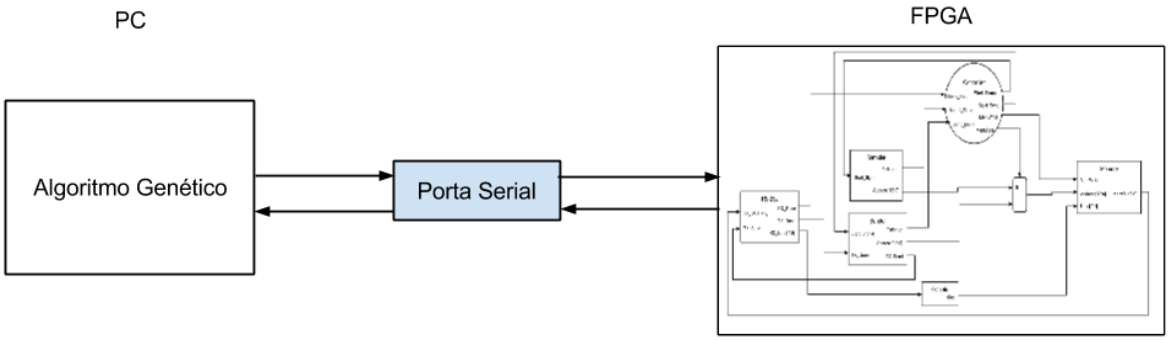


Figura 3.1: Visão de alto nível da comunicação de dados.

O caminho de dados detalhado do circuito de interface é apresentado na figura 3.2

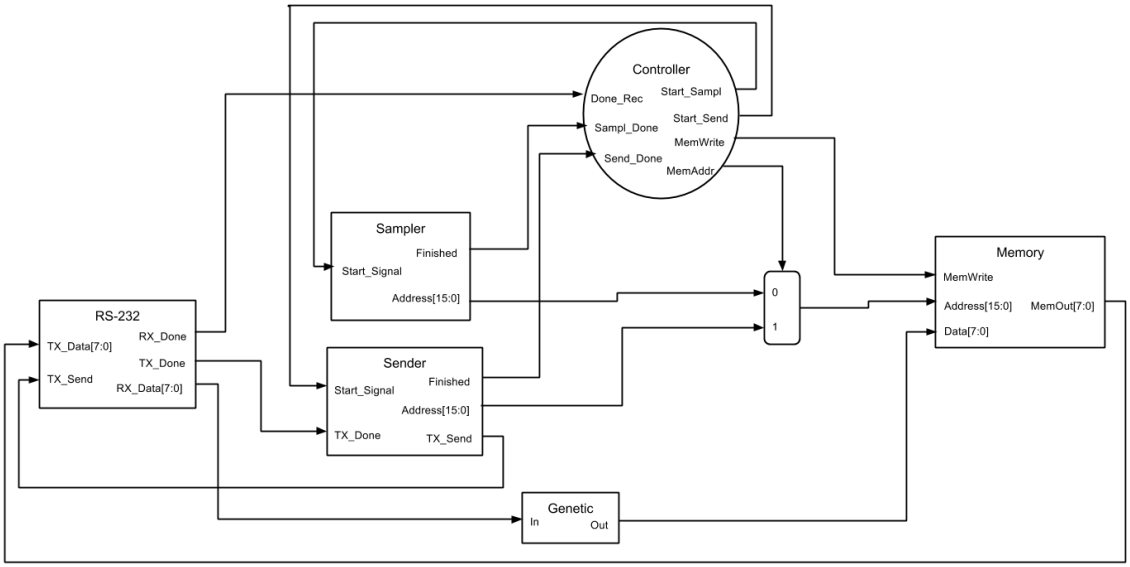


Figura 3.2: Caminho de dados do circuito de interface

O bloco *Genetic* consiste do circuito correspondente ao indivíduo a ser avaliado modelado em *Verilog*. As funcionalidades de cada módulo são as seguintes:

- *RS-232*: enviar e receber dados pela porta serial utilizando o protocolo RS-232.

- *Sampler*: contador básico, responsável por incrementar o endereço no qual escrever a saída do indivíduo.
- *Sender*: enviar os dados da memória para o computador *host*. Espera cada dado ser enviado com sucesso antes de enviar o próximo.
- *Controller*: controlador que possui a máquina de estados do programa, controlando quais sinais ativar ou desativar.
- *Memory*: memória de 65536 *bytes*.
- *Genetic*: circuito genético criado a partir da descrição de um indivíduo.

A figura 3.3 apresenta a máquina de estados principal do circuito de interface.

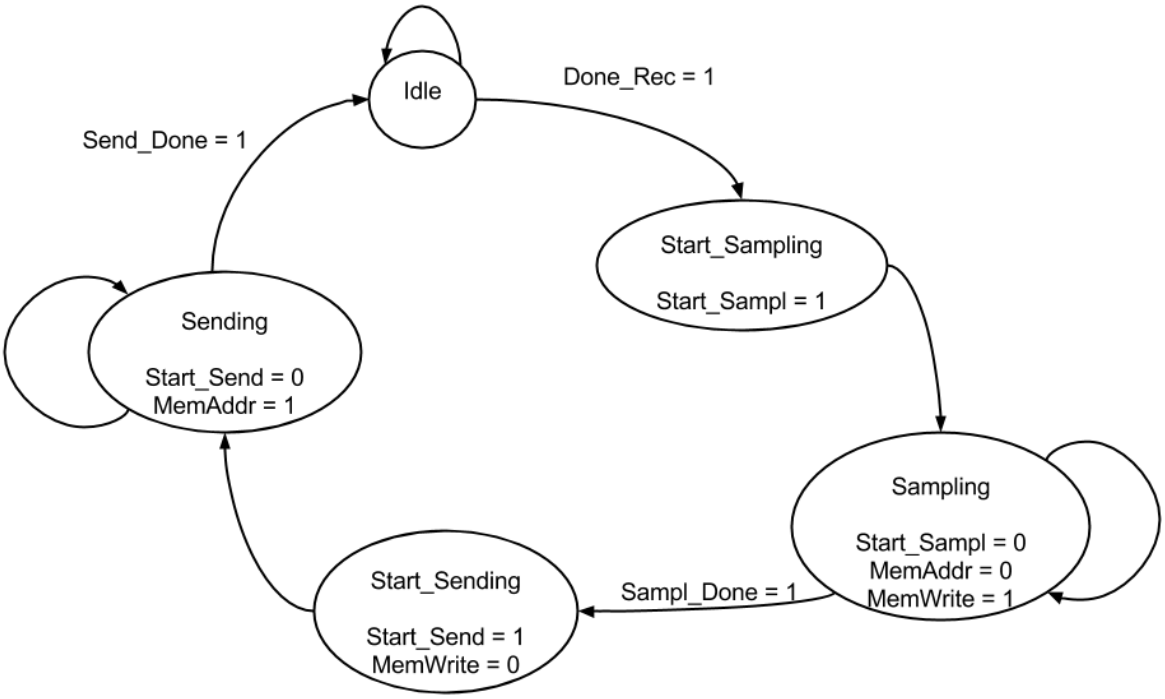


Figura 3.3: Máquina de estados do circuito que gerencia a comunicação entre o computador e o FPGA.

O circuito auxiliar foi projetado para lidar com indivíduos cujas descrições formam circuitos sequenciais, cujas saídas são variáveis ao longo do tempo. Para solucionar esse problema, a saída do indivíduo é amostrada a uma frequência fixa de modo a se obter informações precisas de seu comportamento no tempo.

Ao receber um dado do computador, indicando a entrada atual do indivíduo, o circuito amostra a saída do circuito genético a uma taxa de 50 MHz. A memória, de 65536 *bytes*, então, é completamente preenchida com essas amostras. Em seguida, elas são enviadas para o computador utilizando o protocolo RS-232 de comunicação serial a 115kbps.

Este método provou-se funcional. Porém a avaliação de 1 geração de uma população de 5 indivíduos necessita em média 15 minutos, com a compilação do *bitstream* demandando

cerca de 2 minutos para cada indivíduo. Deste modo, para a realização das várias iterações necessárias à convergência do algoritmo genético, necessita-se de um grande tempo de processamento, tornando o método impraticável. Por exemplo, a avaliação de 10000 gerações demandaria cerca de 105 dias de processamento contínuo.

Para tentar reduzir o tempo de avaliação por geração, todos os indivíduos da população foram inseridos no circuito a ser compilado, evitando várias recompilações além do tempo utilizado para configuração de cada indivíduo na FPGA. Dessa forma, o circuito utiliza um multiplexador para iterar em todos os indivíduos, gravando suas saídas sequencialmente a cada análise. O tempo de avaliação apresentou melhora de 9 minutos, reduzindo-se para um tempo total de 6 minutos por geração, ainda utilizando mais tempo que o desejado, ou 41 dias para um único experimento com 10000 gerações.

Visando ainda a redução do tempo de avaliação dos indivíduos, o próximo método consiste em criar um circuito virtual reconfigurável, ou seja, uma grade de elementos lógicos interconectados entre si, conforme apresentado na figura 2.28. Sua descrição é uma lista de números inteiros indicando qual configuração cada elemento deve assumir. Com isso, se torna possível enviar a descrição do indivíduo serialmente, evitando-se os procedimentos de recompilação e reconfiguração total da FPGA. O elemento lógico básico utilizado pode ser visto na figura 3.4.

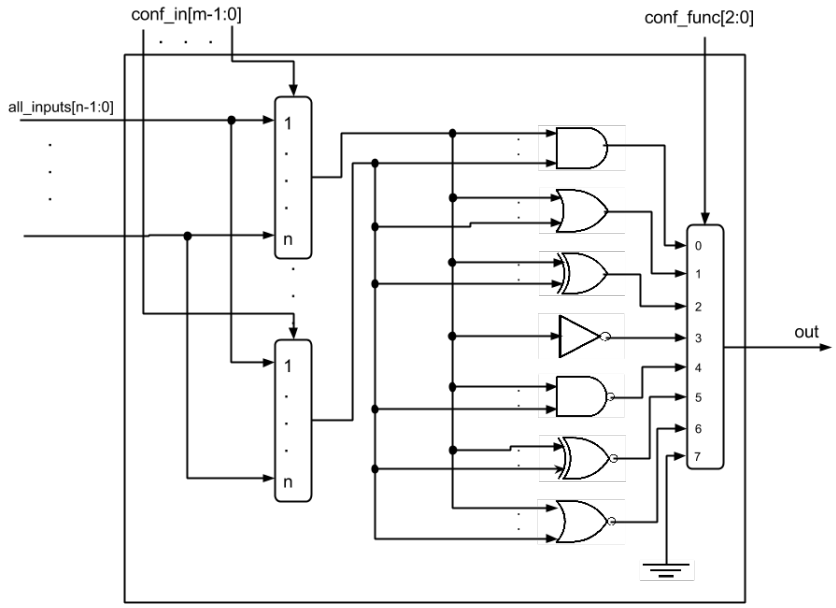


Figura 3.4: Elemento lógico básico que compõe a grade do circuito virtual reconfigurável.

Neste circuito, a entrada é o conjunto de todos os pinos do circuito reconfigurável, ou seja, a união entre as entradas do circuito e os pinos de saída de cada elemento lógico, incluindo seu próprio. Isso resulta em um circuito flexível do ponto de vista de configuração, podendo representar inclusive circuitos realimentados. As entradas superiores do bloco indicam as configurações de cada multiplexador interno. Suas funções possíveis são $\{AND, OR, XOR, NOT, NAND, XNOR, NOR\}$.

O algoritmo, que executa no computador *host*, foi, então, alterado para enviar a descrição do indivíduo, isto é, sua lista de números inteiros ou cromossomo, serialmente e o circuito alterado para carregá-la no módulo reconfigurável. Com isso, há apenas os custos da compilação e *download* iniciais e a comunicação serial.

O tempo de execução, para esta última tentativa, chegou a cerca de 4 segundos por geração para uma população de 5 indivíduos. Deste modo a execução de um único experimento com 10000 gerações, necessita pouco mais de 1 hora de processamento. Este tempo, embora mais viável, ainda é inadequado para uma etapa de investigação, onde um grande número de testes devem ser realizados a fim de validar algoritmos e propostas.

3.1.3 Simulação por *software*

Devido aos problemas encontrados nas implementações em *hardware*, principalmente o tempo necessário para a convergência do algoritmo genético, buscou-se uma solução extrínseca. Assim, evitam-se custos de compilação para *bitstream*, de *download* dos indivíduos para a FPGA, de comunicação de dados e ainda permite a possibilidade de avaliação paralela.

O *software* de simulação *Icarus Verilog* (Williams [2015]) permite que descrições *Verilog* sejam compiladas e simuladas de maneira rápida em um máquina virtual chamada *vvp*. Uma desvantagem fundamental, porém, é que este programa não simula circuitos com realimentação como se espera obter quando um circuito está atuando ou evoluindo. Dessa forma, reduziu-se o escopo do trabalho para cobrir apenas circuitos combinacionais, onde realimentações não são necessárias.

Uma simulação inicial utilizando *Icarus Verilog* obteve resultados mais rapidamente, com uma taxa de cerca de 20 gerações por segundo. Deste modo um experimento evolutivo de 10000 gerações pode ser realizada em pouco mais de 8 minutos. Como algoritmos genéticos são inerentemente algoritmos estocásticos, necessitando de uma análise estatística através da repetição dos experimentos, essa taxa ainda não mostrou-se adequada para a realização dessa tarefa em tempo hábil.

A solução encontrada foi realizar a simulação específica do individuo, tratando o circuito como um dígrafo acíclico (Bondy and Murty [1976]), calculando a expressão booleana (equivalente ao circuito combinacional) por sua travessia. Com isso, a taxa de avaliações atinge cerca de 300 gerações por segundo, portanto um experimento de 10000 gerações necessita apenas 33 segundos para ser concluído.

3.2 Algoritmo proposto

Visando a redução dos custos computacionais da aplicação de estratégia evolucionária na síntese de circuitos digitais e a obtenção de soluções com melhor qualidade em relação a alguma característica de interesse, neste trabalho é proposto o algoritmo Hybrid Multi-Chromosome Cartesian Genetic Programming (HMC-CGP). Este novo algoritmo visa aliar uma forma ágil de busca por uma solução válida com uma etapa explícita de otimização da solução encontrada. Sendo portanto uma estratégia hibrida, ainda não encontrada na literatura, pois associa eficientemente os algoritmos CGP e MC-CGP.

Em ambas etapas, propõem-se o uso da estratégia de evolução $(1+\lambda)$, já de reconhecida eficiência (Hilder et al. [2010]).

3.2.1 Etapa 1: Solução funcional

Walker et al. [2009] mostra que o método MC-CGP converge para uma solução funcional utilizando 3 a 392 vezes menos gerações que o método CGP convencional. Propõe-se, então, a utilização do MC-CGP em uma primeira etapa de modo a obter uma solução funcional para o problema.

De acordo com o MC-CGP, cada saída do problema a ser resolvido é tratada separadamente. Neste trabalho essa separação se dá pela criação de várias populações distintas, que são avaliadas em paralelo para um ganho ainda maior de desempenho.

A função de erro do fenótipo de um individuo com cromossomo C, para uma função de saída desejada D, pode ser definida como

$$\varepsilon(C, D) = \sum_{i=0}^{\rho-1} |S_i(C) - D_i| \tag{3.1}$$

onde $S_i(C)$ é a saída obtida pelo circuito gerado pelo cromossomo C para a entrada i, D_i a saída desejada, n_{in} o número de entradas do circuito e $\rho = 2^{n_{in}}$ o número de linhas da tabela verdade. Em problemas de síntese de circuitos combinacionais, a saída desejada D geralmente é representada pela tabela verdade que soluciona o problema.

Nesta primeira etapa a função de *fitness* $f_1(C, D)$ é definida como

$$f_1(C, D) = \frac{1}{5 \times 10^{-7} + \varepsilon(C, D)} \tag{3.2}$$

onde o valor 5×10^{-7} foi arbitrariamente escolhido de maneira a limitar o valor máximo da função *fitness*.

Esta primeira fase se encerra quando um conjunto de indivíduos que atende corretamente aos requerimentos do problema é encontrado, isto é, a função de *fitness* atinge seu valor máximo para todas as populações.

Uma das maiores desvantagens encontrada no método MC-CGP é que, embora a solução encontrada seja correta, os circuitos de cada saída são evoluídos separadamente. Por consequência, os circuitos gerados costumam ser maiores do que o necessário (Asha and Hemamalini [2015]). Portanto, o algoritmo MC-CGP prioriza o desempenho (número de gerações) em detrimento de possíveis soluções que utilizem o circuito completo de maneira mais eficiente.

3.2.2 Etapa 2: Otimização

O algoritmo HMC-CGP se utiliza de uma segunda etapa de evolução com o objetivo de otimizar a solução encontrada com base em um critério previamente escolhido. Neste trabalho será utilizado como critério de otimização uma combinação das três características definidas

1. Número de portas lógicas;
2. Número de camadas de portas;
3. Número de transistores.

Estas características foram escolhidas por representarem a eficiência do projeto de um circuito digital. O número de portas lógicas caracteriza a complexidade do circuito gerado, portanto a área necessária a sua construção, sendo então relacionado ao custo do circuito integrado. O número de camadas máxima de portas lógicas, ou caminho critico do circuito, reflete a velocidade com que o circuito responde a alterações das suas entradas, impactando diretamente na sua frequência máxima de utilização. O número de transistores está diretamente relacionado com o consumo energético do circuito integrado, bem como também ao seu custo.

Nesta segunda etapa do HMC-CGP, todos os indivíduos distintos que resolvem separadamente o problema obtidos na primeira etapa, são unidos em apenas uma população, por linhas, conforme apresentado na figura 3.5. Além disso, o tamanho da população recebe o incremento de um fator α de modo que $\lambda' = \lambda \cdot \alpha$ proporcione a geração de uma maior diversidade de indivíduos buscando mais caminhos pelo espaço de soluções.

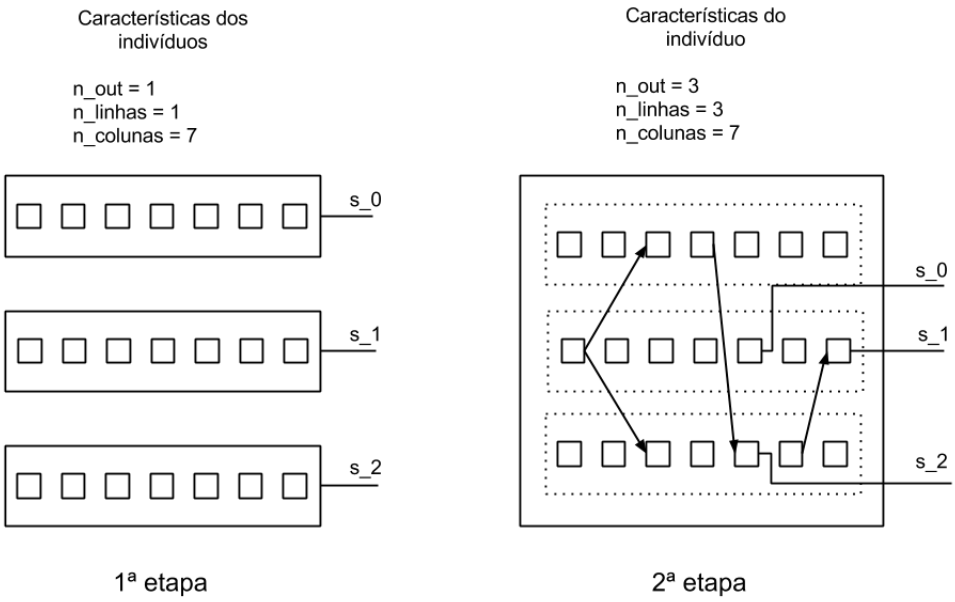


Figura 3.5: Visão geral do algoritmo HMC-CGP.

A figura 3.5 apresenta um exemplo de um problema com 3 saídas. Na primeira etapa, os indivíduos são evoluídos separadamente. Na segunda, os indivíduos são unidos, com cada solução inicialmente representando uma linha dos indivíduos que constituem a população da segunda etapa. À medida que a otimização ocorre, isso permite que a, antes impossível, interação entre os genes dos indivíduos seja possível, isto é, as portas lógicas de cada solução podem então ser compartilhadas. Nesta fase, portanto, propomos a aplicação do algoritmo CGP ordinário para a geração do melhor individuo que representa o circuito digital final.

Nesta fase de otimização a função de *fitness* deve ser redefinida. Dada a função de erro em 3.1, define-se, então, a função de *fitness* $f_2(C, D)$ desta fase como

$$f_2(C, D) = \begin{cases} 0, & \text{se } \varepsilon(C, D) \neq 0 \\ \frac{1}{1+n_p(C)} + \frac{1}{1+n_c(C)} + \frac{1}{1+n_t(C)}, & \text{senão.} \end{cases} \tag{3.3}$$

onde $n_p(C)$ é o número de portas do fenótipo do individuo C, $n_c(C)$ o maior número de camadas de portas e $n_t(C)$ o número total de transistores.

De acordo com a equação 3.3, indivíduos gerados que possuam quaisquer divergências com a saída desejada são descartados, recebendo o menor valor de *fitness* possível. Mantém-se e avaliam-se apenas aqueles indivíduos que atuam corretamente.

Como nesta fase não se procura uma solução correta, a única condição de parada é atingir o número de gerações definido para a otimização.

Neste capítulo foi vista a metodologia principal na qual este trabalho se baseia. Primeiramente, foram discutidas todas as diversas tentativas para avaliação de indivíduos, culminando na avaliação extrínseca adotada.

A parte seguinte apresentou o algoritmo HMC-CGP, projetado como forma de solução possível para o problema da escalabilidade dos algoritmos genéticos.

O próximo capítulo apresenta os resultados obtidos em diversos experimentos com o algoritmo HMC-CGP e sua comparação com estratégias evolutivas convencionais.

Capítulo 4

Resultados Obtidos

Neste capítulo, serão apresentados os resultados dos principais experimentos realizados. O primeiro resultado mostrado é a comparação entre as estratégias evolucionárias CGP e MC-CGP para convergência em alguns experimentos. Na seguinte seção, os resultados e comparações com o método evolutivo proposto HMC-CGP serão detalhados. Por último, a mesma abordagem de otimização usada no HMC-CGP é aplicada partindo-se de um circuito gerado por técnicas clássicas de projetos de circuitos digitais.

A máquina utilizada para execução dos experimentos possui um processador *Intel(R) Core(TM) i5-3570 CPU @ 3.40GHz*, com 4 núcleos, 4 *threads* e *Cache L1* de $4 \times 32KB$, memória RAM de 8GB e com sistema operacional *Windows 7 Professional*.

O projeto foi implementado utilizando a linguagem *C++*, as bibliotecas *OpenMP* de paralelismo e *RS-232 for Linux, FreeBSD and Windows* (van Beelen [2015]) para comunicação serial da parte do computador *host* e o módulo *Verilog Simple RS232 UART* (Bourdeauducq [2010]) para a comunicação serial na FPGA. Todo o código gerado e usado para a obtenção dos seguintes resultados está disponível *online* (Coimbra [2014]).

4.1 *CGP* e *MC-CGP*

Inicialmente pretende-se verificar se, de fato, problemas que utilizam do algoritmo MC-CGP convergem mais eficientemente do que aqueles que utilizam a técnica CGP convencional. Nos experimentos apresentados nesta seção não estamos interessados na estrutura ou qualidade do circuito gerado como solução válida, características estas que serão analisadas na próxima seção.

Realizou-se experimentos de síntese dos seguintes circuitos lógicos

- Somador completo de 1 *bit* com *carry*;
- Somador de 2 *bits*;
- Multiplicador de 2 *bits*;
- Decodificador para *display* de 7 segmentos.

Para fins de comparação da eficiência das estratégias evolucionárias testadas, define-se taxa de sucesso η por

$$\eta = 100 \times \frac{n_s}{n_{total}}, \tag{4.1}$$

onde n_s indica o número de experimentos que obtiveram sucesso em convergir para uma solução correta e n_{total} o número de repetições dos experimentos realizados com o algoritmo.

Para todos os experimentos foram realizadas $n_{total} = 20$ repetições, com limites do número de gerações definidas em 2000, 5000 e 10000. Nos experimentos, os elementos lógicos das matrizes CGP e MC-CGP utilizadas possuem apenas 2 entradas, com a matriz MC-CGP fixada em 1 linha e 10 colunas. O algoritmo de mutação seleciona e modifica 3 genes aleatoriamente para a geração de um novo indivíduo. Cada elemento lógico é capaz de realizar uma entre as funções lógicas $\{AND, OR, XOR, NOT, NAND, XNOR, NOR\}$.

4.1.1 Somador completo de 1 *bit* com *carry*

O somador completo de 1 *bit* é a célula básica utilizada na construção de circuitos somadores de N *bits* por cascateamento, e é definido por

$$\{C_{out}, S\} = A + B + C_{in} \tag{4.2}$$

onde entradas são definidas pelos *bits* A e B, o valor do *carry-in* por C_{in}, o operador $+$ corresponde à adição aritmética e $\{\cdot, \cdot\}$ à operação de concatenação. O resultado da soma S e o *bit* de *carry-out* C_{out} são as saídas do módulo.

A figura 4.1 apresenta este módulo com suas entradas e saídas.

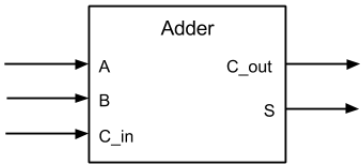


Figura 4.1: Módulo somador de 1 *bit* com *carry-in*.

A tabela 4.1 apresenta a tabela verdade do circuito a ser sintetizado.

Tabela 4.1: Tabela verdade para o circuito somador de 1 *bit*.

Entradas			Saídas	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Neste experimento, a matriz CGP era composta de 1 linha e 10 colunas, fazendo com que 10000 gerações necessitassem de cerca de 6 segundos. Os resultados obtidos pelos algoritmos CGP e MC-CGP para este problema são mostrados na tabela 4.2.

Tabela 4.2: Resultados para somador de 1 *bit* com *carry-in*.

	CGP		*MC-CGP*	
Geração limite	Média gerações	Taxa sucesso	Média gerações	Taxa sucesso
2000	851.0	10%	601.9	100%
5000	2701.1	50%	115.9	100%
10000	4802.0	95%	462.0	100%

A tabela 4.2 apresenta para cada estratégia evolucionária, o número médio de gerações necessário para que uma solução correta seja encontrada e a taxa de sucesso obtida quando se limita o número máximo de gerações.

Comparando os resultados, é possível observar que o CGP mostrou baixas taxas de sucesso de 10% e 50% para limites de 2000 e 5000 gerações respectivamente, e um resultado melhor de 95% para 10000 gerações. Por outro lado, o método MC-CGP apresentou uma taxa de sucesso de 100% para todos os limites de gerações.

É interessante notar que as médias de gerações necessárias para o algoritmo MC-CGP são muito menores que as necessárias ao CGP, podendo aquele algoritmo ser cerca de 10 vezes mais rápidos para altas taxas de sucesso.

4.1.2 Somador de 2 *bits*

A função de um somador de 2 *bits* é realizar a soma de dois números binários de 2 *bit* cada, tendo como resultado um número de 2 *bits* e um *bit* de *carry-out*.

A figura 4.2 apresenta o módulo somador de 2 *bits*, com suas entradas e saídas.

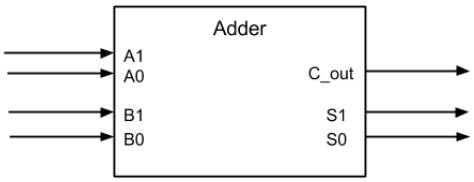


Figura 4.2: Módulo somador de 2 *bits*.

As entradas correspondem aos números A e B, cada um composto de 2 *bits*, a saída S também é um número de 2 *bits* e o *carry-out*, C_{out} conforme

$$\{C_{out}, S\} = A + B \tag{4.3}$$

onde o operador + corresponde à adição aritmética.

A definição deste problema pode ser realizada através da sua tabela verdade apresentada na tabela 4.3.

Tabela 4.3: Tabela verdade para o somador de 2 *bits*.

Entradas				Saídas		
A_1	A_0	B_1	B_0	C_{out}	S_1	S_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

A tabela 4.4 apresenta os resultados obtidos na evolução de soluções para o somador de 2 *bits*, com a matriz CGP definida por 3 linhas e 10 colunas, portanto fazendo com que o tempo necessário para 10000 gerações fosse cerca de 16 segundos.

Tabela 4.4: Resultados para o somador de 2 *bits*.

	CGP		*MC-CGP*	
Geração limite	Média gerações	Taxa sucesso	Média gerações	Taxa sucesso
2000	N/A	0%	1576.44	45%
5000	N/A	0%	1886.26	75%
10000	N/A	0%	3456.85	100%

Nota-se que o método CGP, para 20 repetições do experimento, não conseguiu obter uma solução válida para limites de até 10000 gerações. Por outro lado, o método MC-CGP obteve boas taxas de sucesso a partir do limite de 5000 gerações.

4.1.3 Multiplicador de 2 *bits*

O módulo multiplicador de 2 *bits* é apresentado na figura 4.3.

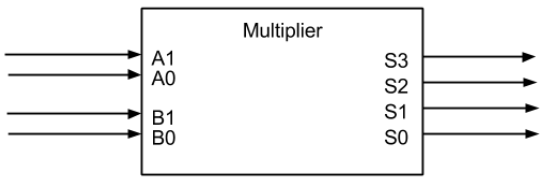


Figura 4.3: Módulo multiplicador de 2 *bits*.

Sua função é multiplicar suas entradas binárias A e B de 2 *bits* de acordo com

$$S = A \times B \tag{4.4}$$

resultando uma saída S de 4 *bits*.

Este problema pode ser definido através da tabela verdade apresentada na tabela 4.5.

Tabela 4.5: Tabela verdade para o multiplicador de 2 *bits*.

Entradas				Saídas			
A_1	A_0	B_1	B_0	S_3	S_2	S_1	S_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Com a matriz CGP possuindo 3 linhas e 10 colunas, o tempo aproximadamente utilizado para 10000 gerações é 16 segundos. Os resultados obtidos pelos algoritmos evolutivos são mostrados pela tabela 4.6.

Tabela 4.6: Resultados para o multiplicador de 2 *bits*.

	CGP		*MC-CGP*	
Geração limite	Média gerações	Taxa sucesso	Média gerações	Taxa sucesso
2000	N/A	0%	1436.2	40%
5000	3429.0	5%	2502.5	90%
10000	6529.0	5%	4022.6	100%

Observa-se que o método CGP não atingiu uma solução válida para um limite de 2000 gerações e obteve, para 20 tentativas, uma única solução para 5000 e 10000 gerações. Por outro lado, com boas taxas para os limites de 5000 e 10000, o método MC-CGP mostra-se superior em termos de convergência.

4.1.4 Decodificador para *display* de 7 segmentos

Os módulos decodificadores para *display* de 7 segmentos são muito utilizados para representar dígitos a partir de um número binário de 4 *bits*. Tradicionalmente, *displays* de 7 segmentos são divididos em duas categorias de representação Binary Coded Decimal (BCD) e hexadecimal.

A categoria BCD utiliza 4 *bits* de entrada para definir os dígitos decimais de 0 a 9, conforme apresentado na tabela 4.7.

Tabela 4.7: Dígitos representados por codificação *BCD*.

Dígito	Binário
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Nesta codificação, as representações para valores superiores, de 1001(10) a 1111(15), são realizadas usando dois dígitos, sendo portanto, estas entradas indefinidas em um módulo, podendo ser consideradas *don't cares*, facilitando enormemente o projeto do circuito.

A codificação hexadecimal, por outro lado, permite representar todos os 16 dígitos associados a uma entrada de 4 *bits*. Os valores de 1001(A) a 1111(F) são definidos conforme a tabela 4.8.

Tabela 4.8: Dígitos superiores na codificação hexadecimal.

Dígito	Binário
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Este experimento busca realizar um circuito decodificador de *display* de 7 segmentos hexadecimal. A tabela verdade do problema é apresentada na tabela 4.9.

Tabela 4.9: Tabela verdade para o decodificador *display* de 7 segmentos hexadecimal.

Entradas				Saídas						
A_3	A_2	A_1	A_0	S_6	S_5	S_4	S_3	S_2	S_1	S_0
0	0	0	0	0	1	1	1	1	1	1
0	0	0	1	0	0	0	0	1	1	0
0	0	1	0	1	0	1	1	0	1	1
0	0	1	1	1	0	0	1	1	1	1
0	1	0	0	1	1	0	0	0	1	1
0	1	0	1	1	1	0	1	1	0	1
0	1	1	0	1	1	1	1	1	0	1
0	1	1	1	0	0	0	0	1	1	1
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1
1	0	1	0	1	1	1	0	1	1	1
1	0	1	1	1	1	1	1	1	0	0
1	1	0	0	0	1	1	1	0	0	1
1	1	0	1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1	0	0	1
1	1	1	1	1	1	1	0	0	0	1

Neste caso, o *bit* 1 representa um LED do *display* ligado e o *bit* 0 desligado. A figura 4.4 exibe a correspondência dos LEDs e os *bits* do sinal de saída S.

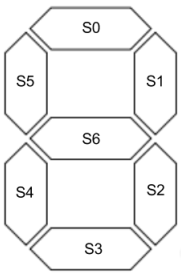


Figura 4.4: Associação das saídas aos LEDs de um *display* de 7 segmentos.

Os resultados obtidos pelos métodos evolutivos para este problema são mostrados na tabela 4.10, utilizando uma matriz CGP definida por 5 linhas e 10 colunas, tomando cerca de 31 segundos para 10000 gerações.

Tabela 4.10: Resultados para o decodificador *display* de 7 segmentos hexadecimal.

	CGP		*MC-CGP*	
Geração limite	Média gerações	Taxa sucesso	Média gerações	Taxa sucesso
2000	N/A	0%	N/A	0%
5000	N/A	0%	N/A	0%
10000	N/A	0%	7072.5	30%

Como é possível observar na tabela 4.10, também neste experimento o método CGP não conseguiu atingir soluções válidas nas 3×20 repetições do experimento com os diferentes limites. Devido à complexidade do problema, o método MC-CGP conseguiu obter apenas 6 soluções válidas neste experimento, considerando um limite de 10000 gerações.

O método evolutivo MC-CGP apresenta-se como uma abordagem naturalmente mais aplicável a várias classes de problemas quando comparado ao método CGP, dentre elas, o projeto de circuitos digitais com múltiplas saídas (Walker et al. [2009]).Os experimentos realizados comprovaram essa afirmação, e justificam a escolha do método MC-CGP para a primeira etapa do método HMC-CGP proposto neste trabalho. A seguinte seção apresentará os resultados obtidos com a aplicação do método proposto.

4.2 *HMC-CGP*

Esta seção visa mostrar os resultados obtidos para a aplicação do algoritmo HMC-CGP proposto, isto é, encontrar uma solução válida e sua otimização. Para cada experimento é definido um número máximo de gerações para ser otimizado de acordo com os critérios discutidos na seção 3.2.2.

Os resultados serão apresentados de forma que seja possível analisar e comparar as características de número de portas, maior número de camadas (caminho crítico) e número de transistores obtidos na primeira etapa e na segunda etapa da estratégia HMC-CGP, para os experimentos

- Somador completo de 1 *bit* com *carry*;
- Somador de 2 *bits*;
- Multiplicador de 2 *bits*;
- Decodificador para *display* de 7 segmentos;
- Somador de 2 *bits* com *carry*.

Os quatro primeiros experimentos já foram definidos e apresentados na seção (4.1).

Cada experimento foi realizado 30 vezes de modo a se obter dados estatisticamente significativos. Nesses experimentos definiu-se um máximo de 50000 gerações, a matriz CGP com 1 linha e 10 colunas, cada elemento lógico possuindo apenas 2 entradas. O parâmetro de incremento entre as populações da primeira e segunda etapas $\alpha = 3.5$ foi definido. O algoritmo de mutação seleciona e modifica 3 genes aleatoriamente. Apenas uma entre as funções lógicas $\{AND, OR, XOR, NOT, NAND, XNOR, NOR\}$ é permitida para cada elemento lógico.

A tabela 4.11 apresenta o número de transistores considerados na construção de cada porta lógica, de acordo com o estudo apresentado na seção 2.1.3.

Tabela 4.11: Tabela de funções lógicas e número de transistores

Função lógica	Número de transistores
AND	6
OR	6
NOT	2
XOR	6
NAND	4
NOR	4
XNOR	8

É interessante notar que esses números de transistores podem ser livremente alterados de acordo com a tecnologia de integração a ser utilizada para a implementação dos circuitos, sem impacto na metodologia ora proposta.

4.2.1 Somador completo de 1 *bit* com *carry*

A figura 4.5 mostra a solução clássica normalmente utilizada pelos projetistas e estudada nos cursos de graduação.

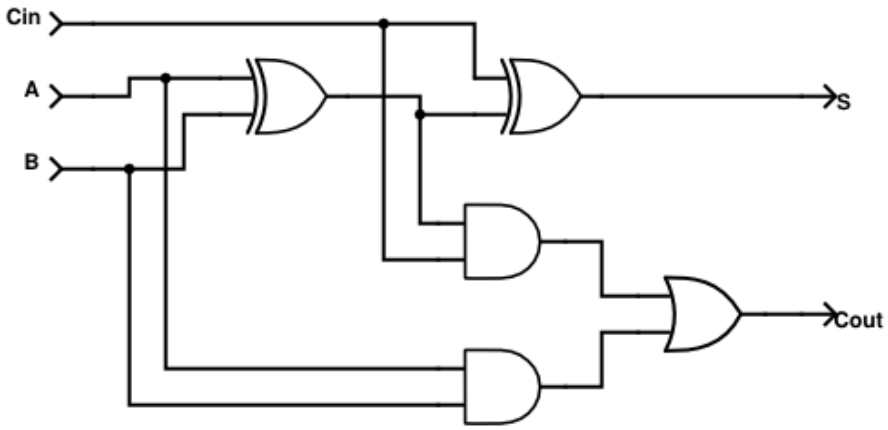


Figura 4.5: Somador completo de 1 *bit* clássico.

É possível observar que trata-se de um circuito intuitivo e modular. Esse circuito possui as seguintes características para o número de portas lógicas (n_p), número máximo de camadas (n_c) e o número total de transistores (n_t) utilizados pela solução

- $n_p = 5$;
- $n_c = 3$;
- $n_t = 30$.

A figura 4.6 mostra a solução sintetizada pelo *software* de projeto *Quartus II*.

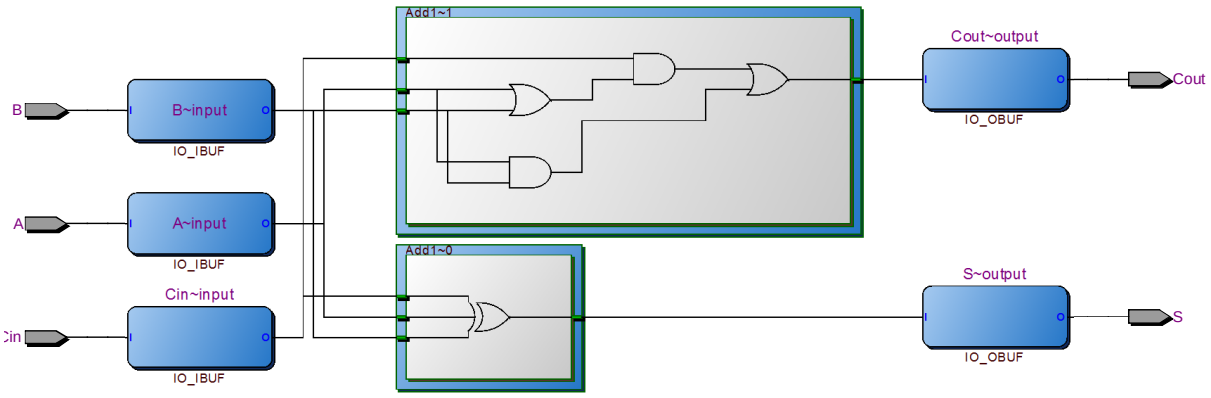


Figura 4.6: Somador completo de 1 *bit* sintetizado pelo *software Quartus II*.

A solução obtida pelo *Quartus II* necessita de uma porta XOR de 3 entradas, que pode ser sintetizada de maneira simplificada usando o cascateamento de duas portas XOR de 2 entradas, resultando assim em $n_p = 6$, $n_c = 3$ e $n_t = 36$.

Os resultados obtidos nas 30 repetições das duas etapas da estratégia HMC-CGP são apresentados na tabela 4.12.

Tabela 4.12: Resultados dos experimentos realizados para o somador completo de 1 *bit*.

	Etapa 1			Etapa 2			
Experimento	n_p	n_c	n_t	n_p	n_c	n_t	Tempo (s)
1	9	3	56	5	3	24	93
2	8	3	48	5	3	24	94
3	9	4	52	5	3	24	107
4	7	3	42	5	3	24	145
5	7	3	40	5	3	24	141
6	11	6	58	5	3	28	139
7	7	4	46	5	3	28	88
8	10	4	48	5	3	24	99
9	8	3	48	5	3	28	98
10	9	5	52	6	3	30	90
11	7	4	42	5	3	30	99
12	11	5	74	5	3	28	95
13	8	3	44	5	3	28	99
14	7	3	42	5	3	30	90
15	7	4	42	5	3	24	89
16	7	3	40	5	3	24	96
17	9	5	58	5	3	30	97
18	8	4	46	5	3	24	92
19	10	4	60	5	3	24	87
20	10	4	52	5	3	24	89
21	9	4	54	5	3	24	89
22	7	3	34	5	3	28	91
23	7	4	38	5	3	24	92
24	6	3	40	5	3	24	88
25	6	3	38	5	3	30	97
26	7	4	38	5	3	28	103
27	6	3	36	5	3	24	98
28	10	4	54	5	3	24	99
29	7	3	42	5	3	24	96
30	11	6	64	5	3	24	94
Média	8.16	3.8	47.6	5.03	3.0	25.93	99.13
Desvio Padrão	1.55	0.88	9.27	0.18	0.00	2.49	15.20

A tabela 4.12 apresenta, para cada repetição do experimento, as características n_p, n_c e n_t obtidas na primeira etapa do algoritmo e após a otimização feita na segunda etapa, bem como o tempo total, em segundos, necessário para a obtenção da solução final. Esta tabela sumariza ainda, apresentando os valores médios e desvios padrão esperados para cada característica e o tempo.

Para se quantificar o grau de otimização atingido em cada critério, define-se o percentual de otimização como

$$r = \frac{n_{original} - n_{otimizado}}{n_{original}} \times 100 \tag{4.5}$$

onde r é a redução percentual obtida pela otimização de uma característica, $n_{original}$ o valor original da característica antes da otimização e $n_{otimizado}$ o valor após a otimização.

Dessa forma, para este experimento, as reduções médias observadas são de r_p : 38.3% para o número de portas lógicas, r_c : 21.0% para o maior caminho crítico e r_t : 45.5% para o número de transistores.

A tabela 4.12 nos mostra que, considerando o número de transistores n_t, inúmeras soluções são mais eficientes do que a solução clássica. A figura 4.7 apresenta um exemplo de circuito evoluído geneticamente com $n_t = 24$.

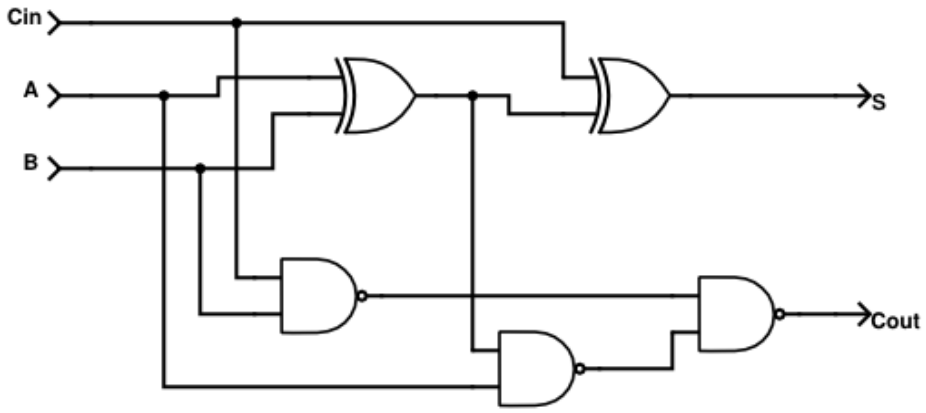


Figura 4.7: Exemplo de solução encontrada pelo HMC-CGP para o circuito somador de 1 *bit*.

Fazendo uso de portas *NAND*, que possuem menos transistores do que portas *AND* e *OR*, o cálculo do C_{out} é feito utilizando menos recursos do que o circuito da figura 4.5. O circuito ainda utiliza portas *XOR* para o cálculo da soma S, pois mesmo tendo mais transistores do que outras portas como *NAND* ou *NOR*, é uma ferramenta mais eficiente para a soma de dois *bits*.

Uma das técnicas utilizada para o projeto de circuitos somadores de N *bits* consiste cascatear múltiplos somadores de 1 *bit*. Dessa forma, se o somador geneticamente evoluído for utilizado para a construção de um somador de 32 *bits*, por exemplo, haverá uma redução de aproximadamente 192 transistores, reduzindo custos de produção e potência consumida.

4.2.2 Somador de 2 *bits*

O bloco somador de 2 *bits* clássico, apresentado na literatura, é mostrado na figura 4.8.

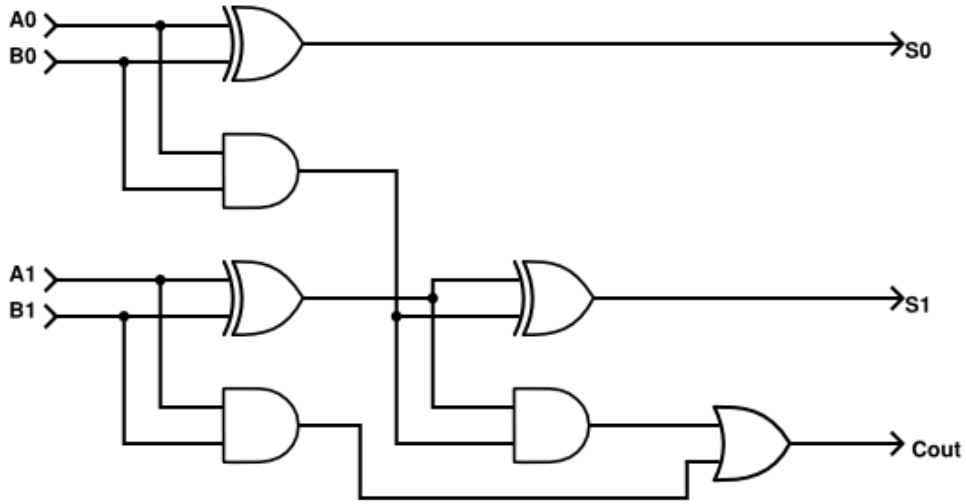


Figura 4.8: Somador de 2 *bits* como normalmente projetado.

Este circuito necessita de 7 portas lógicas, 3 camadas e 42 transistores para ser construído.

O circuito sintetizado a partir da descrição comportamental em *Verilog* utilizando o *software Quartus II* é apresentado na figura 4.9.

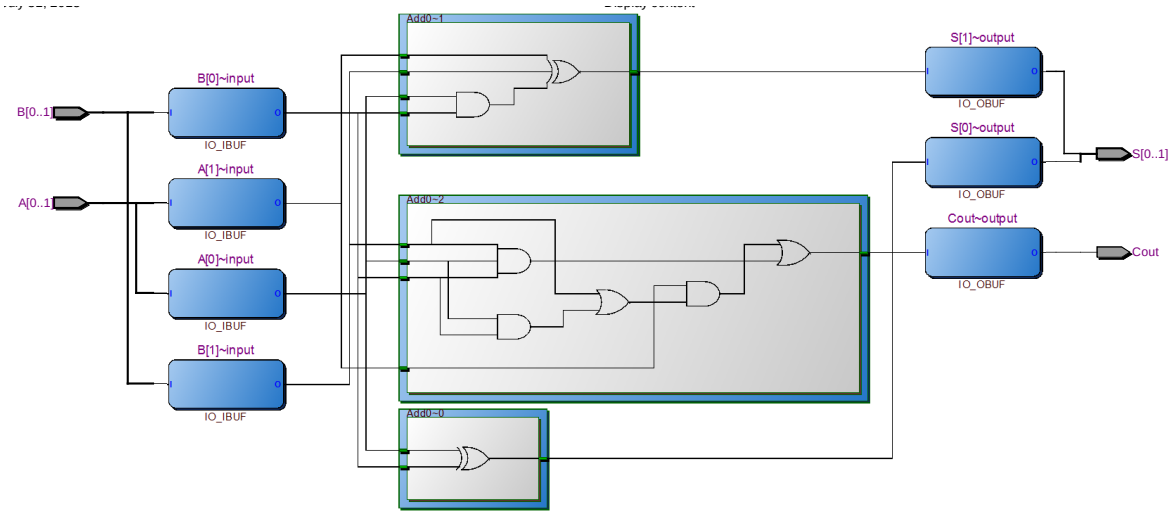


Figura 4.9: Somador de 2 *bits* projetado pelo *Quartus II*.

Este circuito necessita de 8 portas lógicas, 4 camadas e 48 transistores para ser construído.

Os resultados obtidos para a evolução de um somador de 2 *bits* pela estratégia proposta são apresentados na tabela 4.13.

Tabela 4.13: Resultados dos experimentos realizados para o somador de 2 *bits*.

	Etapa 1			Etapa 2			
Experimento	n_p	n_c	n_t	n_p	n_c	n_t	Tempo (s)
1	15	5	70	7	3	38	279
2	16	4	76	7	3	36	281
3	12	4	64	7	3	38	300
4	12	3	62	7	3	38	298
5	11	4	66	7	3	36	292
6	17	5	98	7	3	36	287
7	11	5	58	7	3	36	288
8	10	5	56	8	3	48	307
9	14	4	80	7	3	36	299
10	14	5	88	7	3	40	302
11	10	4	60	7	3	40	303
12	15	5	86	7	4	40	316
13	10	6	64	7	4	42	299
14	11	5	68	7	4	42	305
15	14	5	88	7	3	36	269
16	13	6	68	7	3	38	268
17	16	5	88	7	3	36	264
18	10	4	52	7	3	42	269
19	10	3	54	7	3	36	263
20	10	3	56	7	3	36	264
21	12	4	72	7	3	36	287
22	14	6	80	7	3	36	277
23	13	6	76	7	3	38	293
24	11	5	60	7	3	38	270
25	12	4	64	8	3	40	292
26	14	5	74	7	4	40	288
27	14	4	80	7	3	40	293
28	15	4	84	7	3	36	291
29	11	5	58	8	3	42	301
30	17	5	104	7	3	40	323
Média	12.8	4.6	71.8	7.1	3.13	38.53	288.93
Desvio Padrão	2.23	0.86	13.54	0.31	0.35	2.83	15.84

Para este projeto obtivemos uma redução média de $r_p = 44.5\%$, $r_c = 31.9\%$ e r_t : 46.3%, entre os circuitos evoluídos geneticamente nas duas etapas do algoritmo HMC-CGP.

Um dos circuitos mais otimizado contendo 7 portas, 3 camadas e 36 transistores, pode ser visto na figura 4.10.

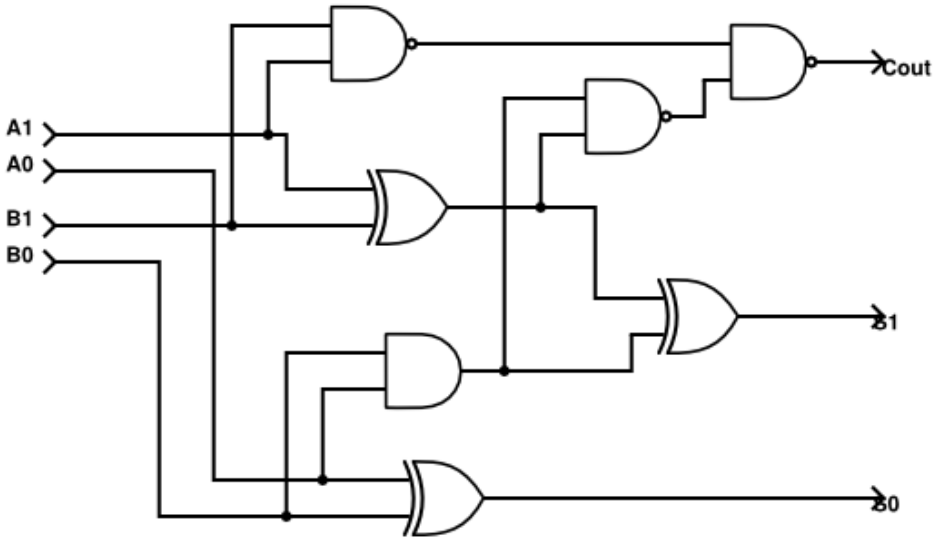


Figura 4.10: Somador de 2 *bits* projetado pelo HMC-CGP.

A solução genética, apesar de ter o mesmo número de portas, possui 6 transistores a menos que a solução clássica. A evolução deste circuito segue uma estratégia de utilizar portas *NAND*s similar àquela vista no experimento do somador de 1 *bit*, apresentado na figura 4.7, para minimizar o número de transistores utilizados.

4.2.3 Multiplicador de 2 *bits*

A síntese do circuito multiplicador de 2 *bits*, definido por 4.4, pelo método clássico de Mapa de *Karnaugh* gera o seguinte conjunto de equações para as saídas (32x8 [2015])

$$\begin{aligned} S_3 &= A_1 A_0 B_1 B_0 \\ S_2 &= A_1 \overline{A_0} B_1 + A_1 B_1 \overline{B_0} \\ S_1 &= \overline{A_1} A_0 B_1 + A_0 B_1 \overline{B_0} + A_1 \overline{A_0} B_0 + A_1 \overline{B_1} B_0 \\ S_0 &= A_0 B_0 \end{aligned} \tag{4.6}$$

O circuito resultante dessas equações pode ser visto na figura 4.11.

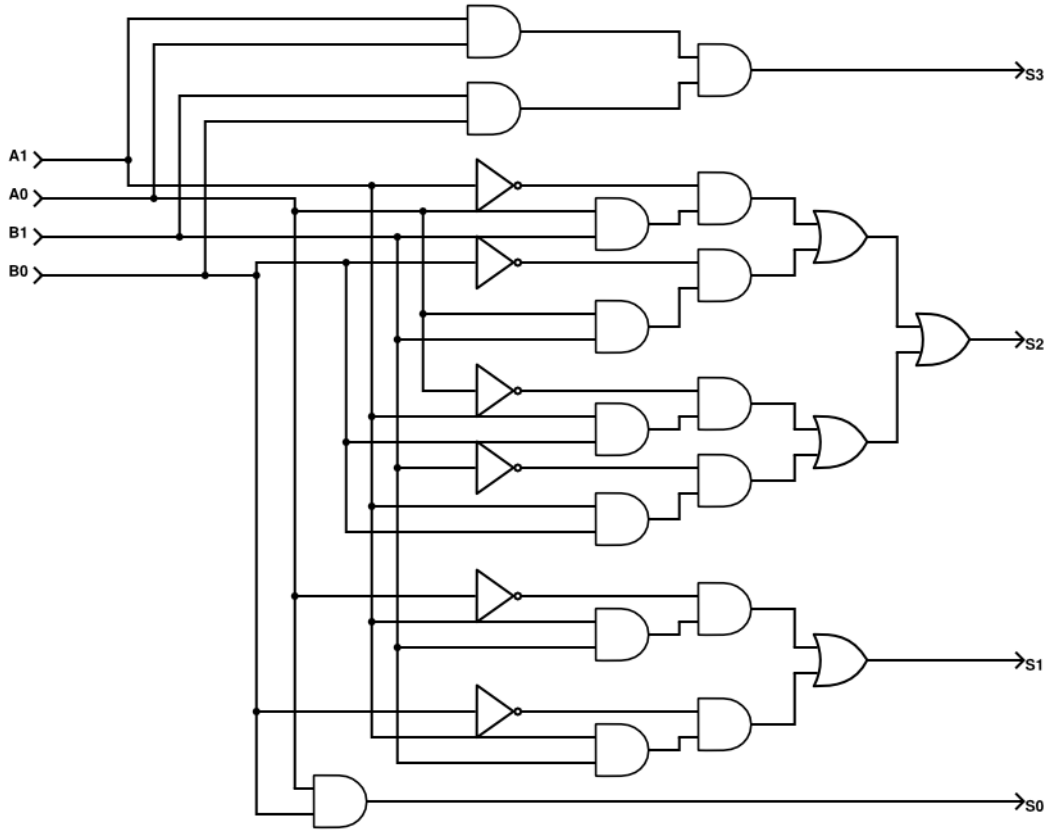


Figura 4.11: Multiplicador de 2 *bits* gerado por minimização por Mapa de *Karnaugh*.

O circuito obtido consiste em 26 portas lógicas dispostas em um máximo de 4 camadas, totalizando 132 transistores para sua construção. Isso mostra como a síntese por Mapa de *Karnaugh* é capaz de modelar cada saída com a menor equação lógica possível, porém a síntese do conjunto de saídas gera circuitos não otimizados.

O circuito feito por síntese do *Quartus II* está na figura 4.12.

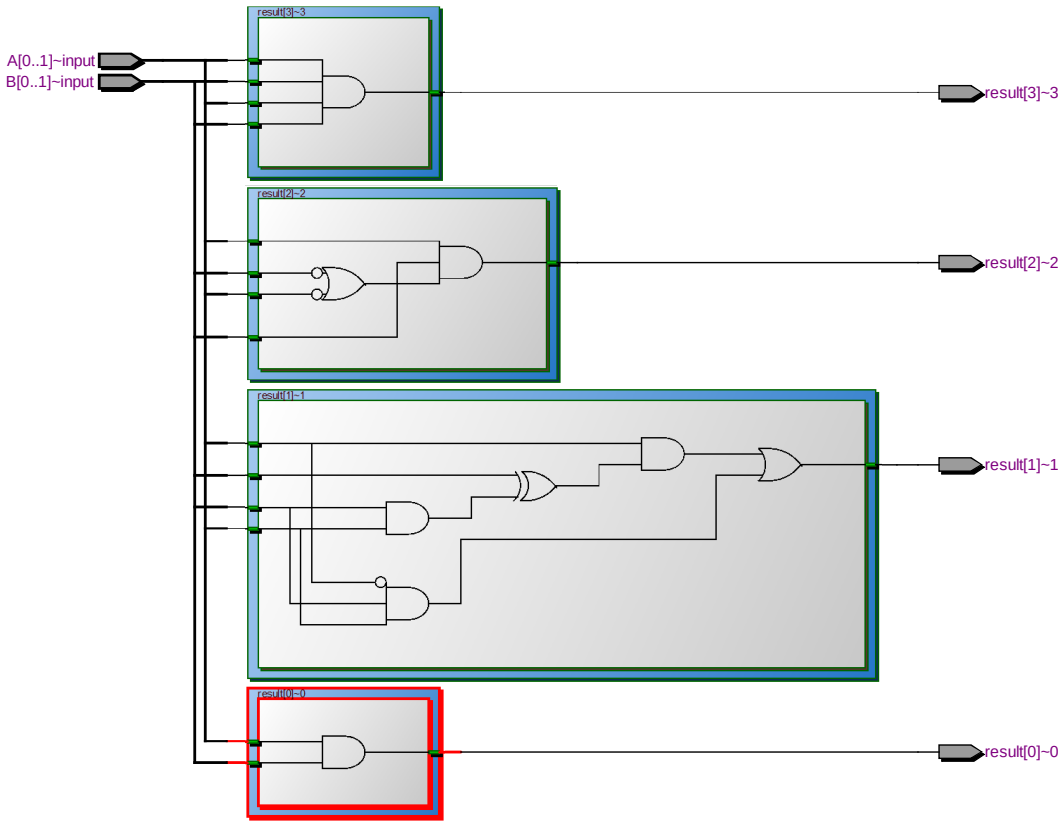


Figura 4.12: Circuito multiplicador de 2 *bits* sintetizado pelo *Quartus II*.

Os resultados obtidos pela evolução genética para o multiplicador de 2 *bits* são apresentados na tabela 4.14.

Tabela 4.14: Resultados dos experimentos realizados para multiplicador de 2 *bits*.

	Etapa 1			Etapa 2			
Experimento	n_p	n_c	n_t	n_p	n_c	n_t	Tempo (s)
1	13	4	62	7	2	32	370
2	19	6	98	7	2	38	360
3	15	5	72	8	2	42	369
4	15	5	68	8	2	36	361
5	15	5	78	7	2	38	353
6	18	4	88	7	2	32	353
7	14	3	66	9	3	40	363
8	14	3	74	7	2	32	356
9	11	3	52	8	2	36	356
10	15	4	78	8	2	36	356
11	13	4	74	7	2	32	357
12	12	3	64	8	2	36	362
13	14	4	74	8	2	42	358
14	17	4	90	8	2	42	353
15	11	4	64	8	2	42	355
16	17	5	106	11	3	46	357
17	20	5	102	10	4	50	358
18	14	5	84	8	2	36	359
19	13	3	68	7	2	38	355
20	11	4	62	7	2	38	359
21	16	4	80	7	2	38	358
22	14	4	68	7	2	38	359
23	13	4	68	8	3	36	359
24	13	3	72	7	2	38	356
25	11	4	66	8	3	36	350
26	12	3	64	7	2	38	354
27	15	5	72	7	2	32	357
28	14	5	64	7	2	32	359
29	15	3	82	8	2	36	356
30	12	3	70	7	2	38	355
Média	14.2	4.03	74.33	7.7	2.2	37.53	357.76
Desvio Padrão	2.33	0.85	12.51	0.95	0.48	4.22	4.26

A tabela nos mostra que, em média, pode-se obter as significativas reduções percentuais das características analisadas quando comparada à versão evoluída não otimizada. Redução no número de portas lógicas de $r_p = 45.7\%$, no número de camadas (caminho crítico) de $r_c = 45.4\%$ e no número de transistores do circuito de $r_t = 49.5\%$.

O resultado mais significativo a mostrar neste experimento é a considerável otimização realizada em média para todos os aspectos. Em especial, o número de transistores conseguiu ser reduzido em quase 50%. A maior parte das evoluções convergiram para uma solução com 7 portas lógicas estruturadas em 2 camadas e 32 transistores, cujo circuito é mostrado na figura 4.13.

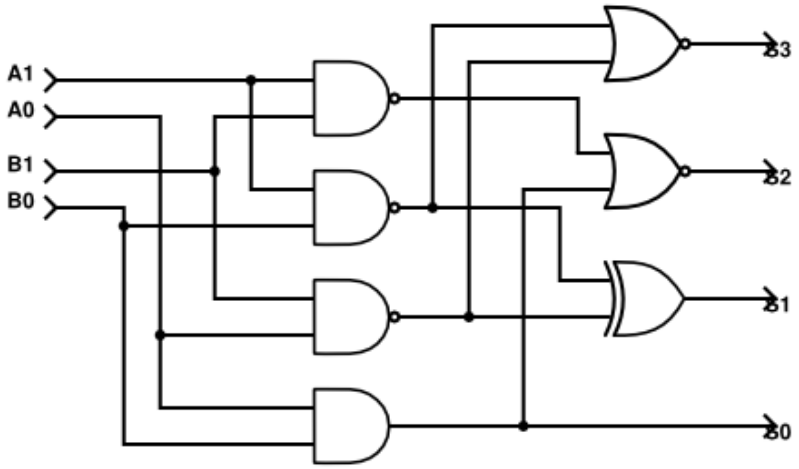


Figura 4.13: Multiplicador de 2 *bits* obtido pelo HMC-CGP

Em comparação com o circuito apresentado em 4.11, a solução ótima encontrada apresenta uma redução de $r_p = 73.0\%$, $r_c = 50.0\%$ e $r_t = 75.7\%$.

4.2.4 Decodificador para *display* de 7 segmentos

A tabela 4.15 apresenta os resultados obtidos na evolução de circuitos de decodificação para *display* de 7 segmentos em hexadecimal.

Tabela 4.15: Resultados dos experimentos realizados para decodificador para *display* de 7 segmentos hexadecimal.

	Etapa 1			Etapa 2			
Experimento	n_p	n_c	n_t	n_p	n_c	n_t	Tempo (s)
1	45	5	254	31	5	166	716
2	54	7	288	35	5	190	713
3	50	5	286	36	5	186	724
4	45	5	270	36	4	194	750
5	46	6	274	33	4	186	701
6	48	5	278	30	5	164	684
7	47	5	274	31	5	176	708
8	47	7	264	33	5	174	704
9	50	5	282	35	5	186	754
10	46	6	262	35	5	192	771
11	43	5	236	29	5	166	778
12	48	5	284	33	5	176	704
13	49	6	272	32	6	158	777
14	49	5	256	31	5	162	727
15	51	7	284	33	5	168	744
16	44	6	240	31	5	170	731
17	47	6	288	29	5	152	677
18	45	5	256	32	4	170	741
19	48	7	264	30	5	162	763
20	50	6	288	31	5	170	742
21	46	5	264	34	5	180	749
22	46	6	272	36	6	182	795
23	44	6	270	31	5	186	789
24	49	5	260	36	5	184	824
25	49	7	282	34	6	186	716
26	47	5	276	33	5	178	725
27	48	8	268	34	5	188	773
28	50	8	282	36	5	182	747
29	49	6	280	35	5	198	785
30	46	5	260	31	5	160	707
Média	47.53	5.83	270.46	32.86	5.0	176.4	740.63
Desvio Padrão	2.40	0.95	13.60	2.22	0.45	11.92	35.37

Por esta tabela podemos observar uma redução percentual média de $r_p = 30.8\%$ no número de portas lógicas, $r_c = 14.2\%$ no número de camadas e $r_t = 39.1\%$ no número de transistores.

Neste experimento, é interessante observar as diferentes otimizações realizadas. No experimento 17, foi encontrado um circuito capaz de realizar a tarefa com 29 portas, 5 camadas e 152 transistores. Um outro circuito com 32 portas e 170 transistores, mas com 4 camadas foi encontrado no experimento 18, mostrando que em diferentes casos, a evolução seguiu caminhos diferentes. Outro detalhe que se destaca é que, diferente dos

experimentos anteriores, não há uma solução que domine a maior parte das otimizações, evidenciando essa diversidade.

O circuito obtido no experimento 17 é mostrado na figura 4.14.

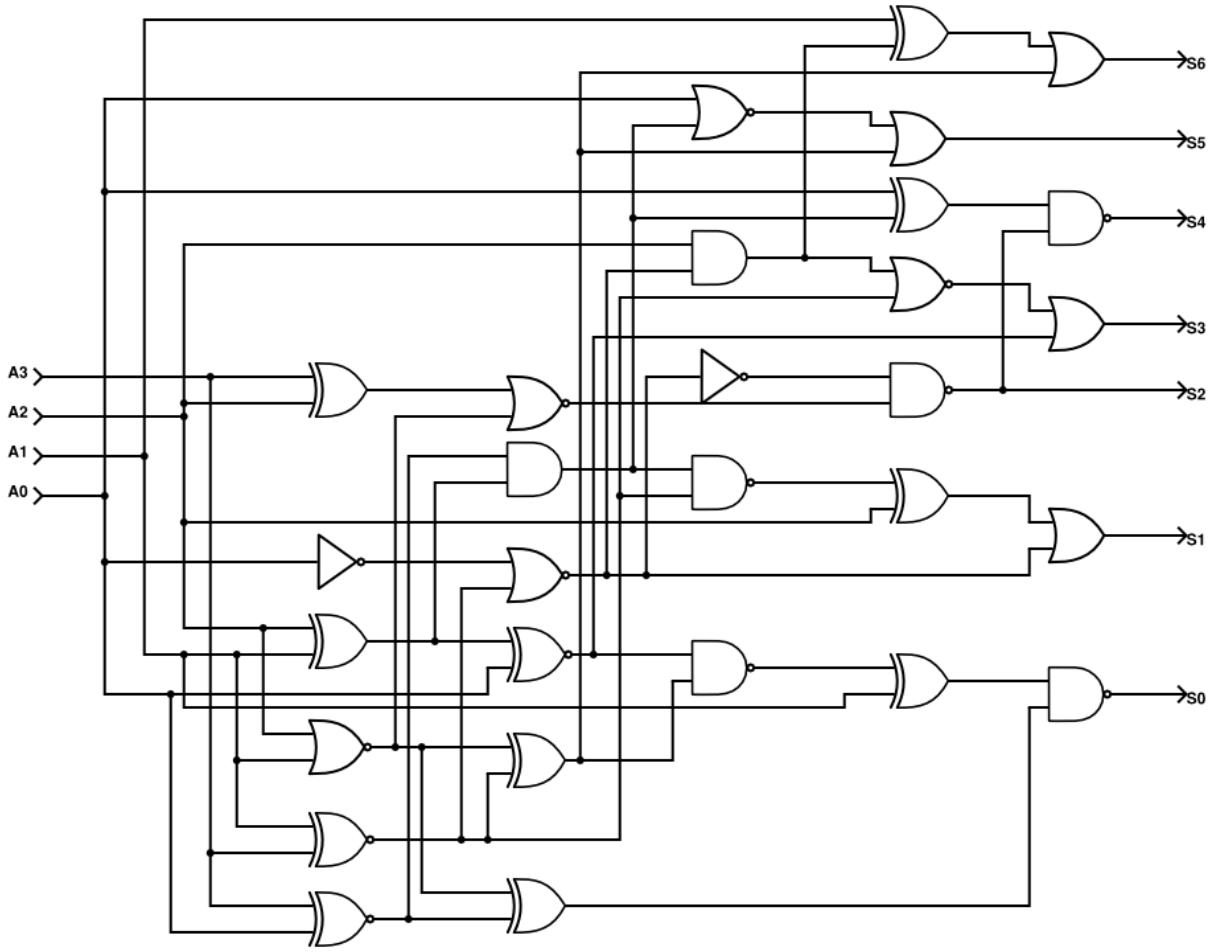


Figura 4.14: Circuito decodificador para *display* de 7 segmentos com 29 portas, 5 camadas e 152 transistores obtido pelo HMC-CGP

A figura mostra um circuito complexo em que há um forte reaproveitamento de cálculos já realizados entre si para computar a resposta corretamente. Esse reaproveitamento foi mencionado anteriormente na seção 3.2.2, como uma das possíveis desvantagens da estratégia MC-CGP usada na primeira etapa do algoritmo, e que é explorada na aplicação da estratégia CGP na segunda etapa do algoritmo HMC-CGP proposto.

O circuito gerado pelo *software Quartus II* é apresentado na figura 4.15.

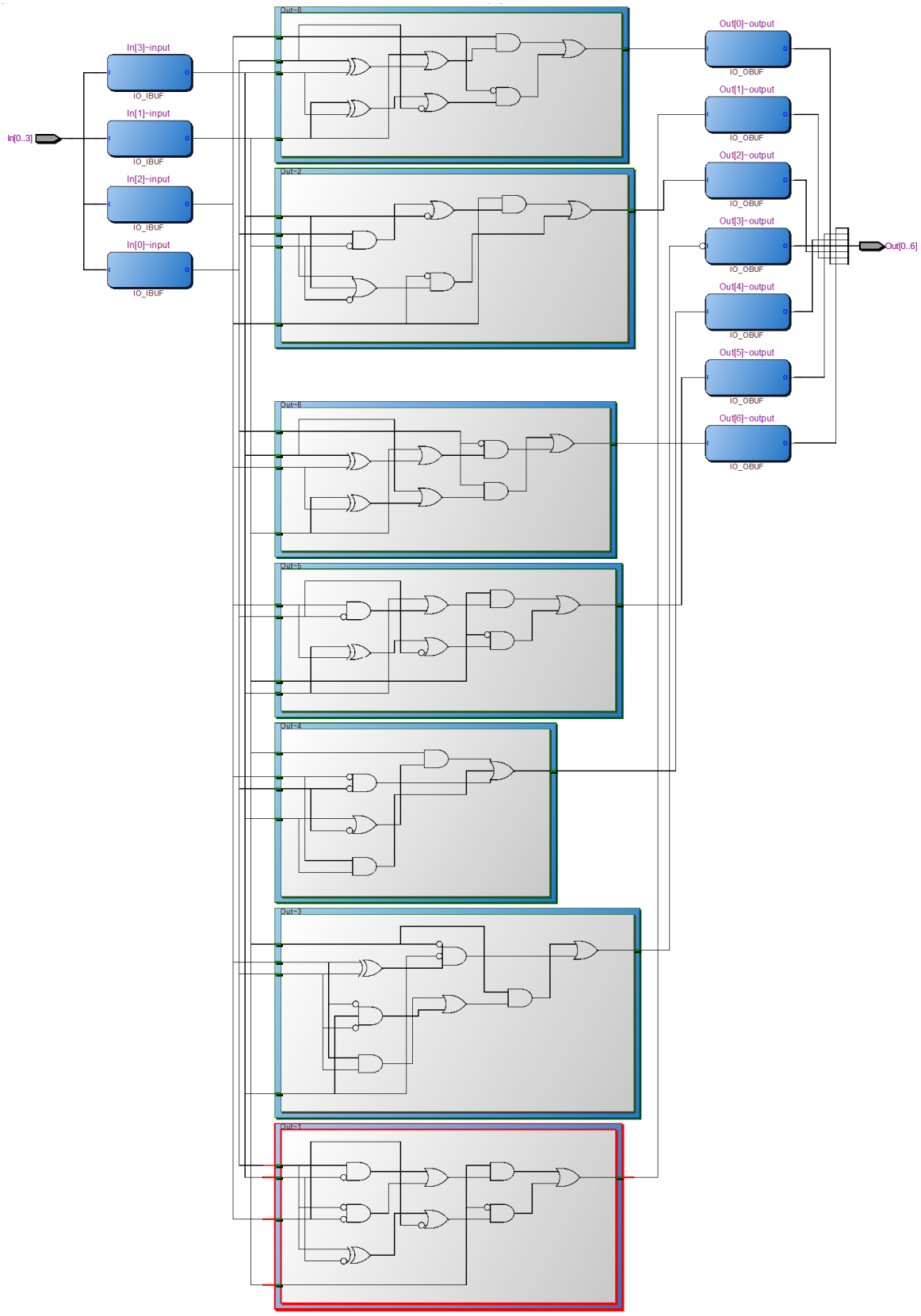


Figura 4.15: Circuito gerado por síntese pelo *software Quartus II*.

A figura 4.15 evidencia que o programa aloca um LE da FPGA para cada saída necessária. O circuito em portas lógicas total equivalente, considerada a soma dos circuitos equivalentes implementados por cada LE, é 74. O número máximo de camadas é 5 e o número de transistores, 352. Se comparado com a solução vista na figura 4.14, as reduções observadas, então, são $r_p = 60.8\%$, $r_c = 0.0\%$ e $r_t = 56.8\%$.

4.2.5 Somador de 2 *bits* com *carry*

O objetivo principal deste experimento, em particular, é comparar a estratégia de criação de circuitos somadores de N *bits* pelo cascateamento de módulos somadores de 1 *bit* (4.2.1) e a evolução direta de um somador de N *bits*. Desde modo, neste experimento o módulo somador de 2 *bits* com *carry* é projetado e comparado com o cascateamento das soluções obtidas na seção 4.2.1.

Esse circuito possui uma funcionalidade similar ao somador de 2 *bits*, porém com a adição de uma entrada a mais, C_{in}, representando o valor *carry-in*. A tabela verdade deste problema é apresentada pela tabela 4.16.

Tabela 4.16: Tabela verdade para o módulo somador de 2 *bits* com *carry*.

Entradas					Saídas		
C_{in}	A_0	A_1	B_0	B_1	C_{out}	S_0	S_1
0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1
0	0	0	1	0	0	1	0
0	0	0	1	1	0	1	1
0	0	1	0	0	0	0	1
0	0	1	0	1	0	1	0
0	0	1	1	0	0	1	1
0	0	1	1	1	1	0	0
0	1	0	0	0	0	1	0
0	1	0	0	1	0	1	1
0	1	0	1	0	1	0	0
0	1	0	1	1	1	0	1
0	1	1	0	0	0	1	1
0	1	1	0	1	1	0	0
0	1	1	1	0	1	0	1
0	1	1	1	1	1	1	0
1	0	0	0	0	0	0	1
1	0	0	0	1	0	1	0
1	0	0	1	0	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	0	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	0	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	0	0	1	1
1	1	0	0	1	1	0	0
1	1	0	1	0	1	0	1
1	1	0	1	1	1	1	0
1	1	1	0	0	1	0	0
1	1	1	0	1	1	0	1
1	1	1	1	0	1	1	0
1	1	1	1	1	1	1	1

O circuito sintetizado pelo *software Quartus II* pode ser visto na figura 4.16.

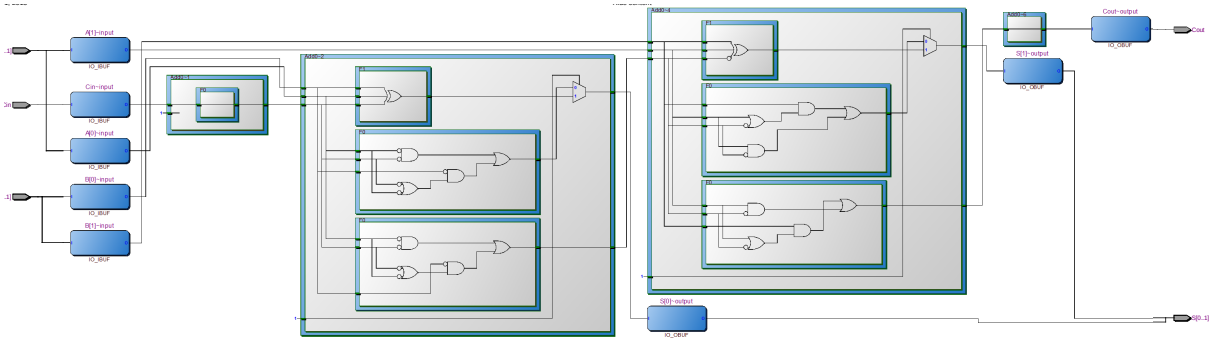


Figura 4.16: Circuito somador de 2 *bits* com *carry* sintetizado por *Quartus II*.

Os resultados obtidos neste experimento são apresentados na tabela 4.17.

Tabela 4.17: Resultados dos experimentos realizados para o somador de 2 *bits* com *carry*.

	Etapa 1			Etapa 2			
Experimento	n_p	n_c	n_t	n_p	n_c	n_t	Tempo (s)
1	17	4	96	13	4	68	758
2	18	6	102	11	5	54	742
3	17	6	106	10	6	58	695
4	17	5	96	11	5	58	771
5	17	6	94	11	5	60	844
6	23	5	136	11	5	60	761
7	18	5	102	11	5	58	635
8	21	6	114	10	5	54	749
9	19	5	112	12	5	74	769
10	16	5	90	11	5	54	781
11	20	5	124	11	5	60	850
12	19	5	122	11	5	62	777
13	18	5	110	10	5	52	709
14	18	5	106	11	5	54	757
15	20	6	122	11	5	58	746
16	18	5	106	12	5	64	715
17	17	6	104	10	6	58	658
18	19	5	114	14	4	78	653
19	16	4	98	11	4	62	655
20	17	5	98	12	5	58	670
21	17	5	100	10	5	58	640
22	16	5	106	11	5	64	686
23	16	4	90	11	4	58	697
24	18	5	100	10	5	48	740
25	16	5	104	10	5	58	628
26	21	5	114	14	4	76	680
27	19	5	120	10	5	56	705
28	18	5	108	12	4	68	644
29	19	6	114	10	5	58	608
30	17	6	106	10	6	54	564
Média	18.06	5.16	107.13	11.06	4.9	60.06	709.56
Desvio Padrão	1.70	0.59	10.67	1.11	0.55	6.92	67.46

Neste experimento observou-se uma redução média de $n_p = 38.7\%$ no número de portas lógicas, de $n_c = 5.0\%$ no numero de camadas e de $n_t = 43.9\%$ no número de transistores necessários.

O melhor resultado em termos de n_t, com 48 transistores, não se mostra mais eficiente do que o cascateamento de dois do melhor encontrado pelo somador de 1 *bit*, cada com 24 transistores. Em contrapartida, o circuito cascateado possui 5 camadas enquanto há resultados que mostram circuitos com apenas 4 camadas. O circuito obtido pode ser visto na figura 4.17.

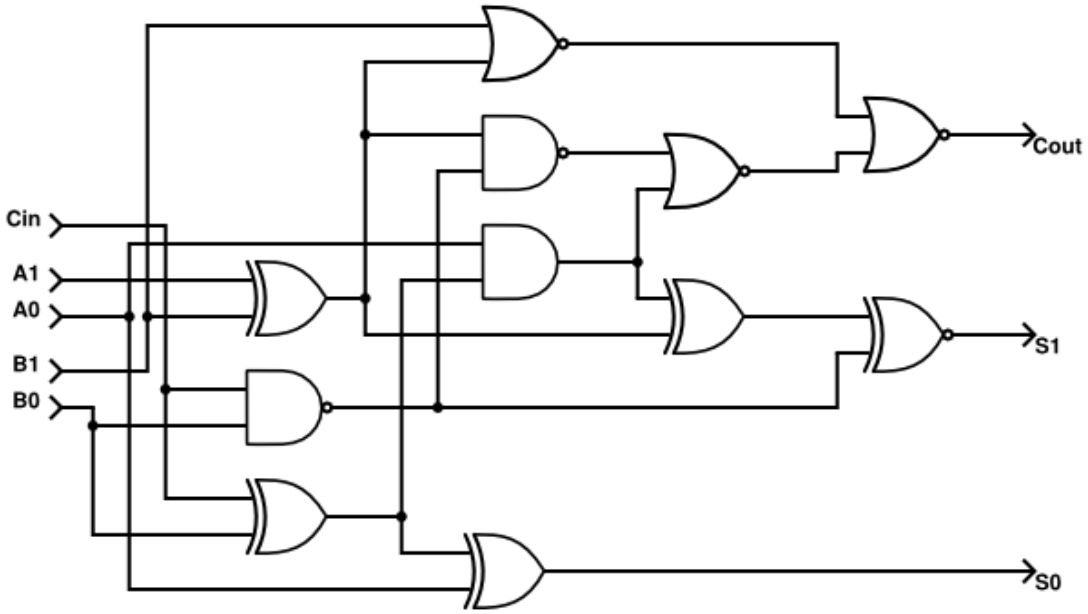


Figura 4.17: Circuito mais otimizado encontrado em termos de n_c para somador de 2 *bits* com *carry* pelo HMC-CGP.

Dessa forma, é possível observar que foram obtidos resultados mais eficientes em termos de n_c nos circuitos evoluídos, quando comparados ao simples cascateamento de somadores de 1 *bit*.

4.3 Otimização a partir de técnicas clássicas

Um experimento adicional foi realizado para investigar a hipótese acerca da eficácia de uma otimização a partir de circuitos gerados por técnicas como Mapa de *Karnaugh* ou o método de *Quine-McCluskey*, ambos discutidos na seção 2.1.6.

A motivação principal vem da dificuldade das técnicas evolutivas convergirem para uma solução correta para circuitos que possuem 6 ou mais entradas. Como existem métodos determinísticos já definidos para a geração desses circuitos, seu uso também tornaria a primeira etapa do algoritmo HMC-CGP determinística.

O circuito escolhido para isso foi o somador de 1 *bit* com *carry*. Apesar de não ser um circuito considerado grande para algoritmos genéticos, buscou-se apenas verificar a hipótese inicial.

Para o circuito escolhido, as seguintes expressões booleanas formam a solução mínima encontrada por Mapa de *Karnaugh*:

$$S = \overline{A}\overline{B}C_{in} + \overline{A}B\overline{C_{in}} + A\overline{B}\overline{C_{in}} + ABC_{in} \tag{4.7}$$

$$C_{out} = BC_{in} + AC_{in} + AB \tag{4.8}$$

Os resultados encontrados estão na tabela 4.18.

Tabela 4.18: Resultados dos experimentos realizados para o somador completo de 1 *bit* com *carry* gerado por Mapa de *Karnaugh*.

	Etapa 1			Etapa 2			
Experimento	n_p	n_c	n_t	n_p	n_c	n_t	Tempo (s)
1	19	6	102	5	3	24	138
2	19	6	102	5	3	28	130
3	19	6	102	5	3	24	131
4	19	6	102	5	3	24	140
5	19	6	102	5	3	30	128
6	19	6	102	5	3	30	129
7	19	6	102	5	3	24	128
8	19	6	102	5	3	24	134
9	19	6	102	5	3	24	134
10	19	6	102	5	3	24	130
11	19	6	102	5	3	24	135
12	19	6	102	5	3	24	134
13	19	6	102	5	3	24	142
14	19	6	102	5	3	24	127
15	19	6	102	5	3	24	130
16	19	6	102	5	3	24	128
17	19	6	102	5	3	24	126
18	19	6	102	5	3	24	127
19	19	6	102	5	3	24	127
20	19	6	102	5	3	24	133
21	19	6	102	5	3	24	138
22	19	6	102	6	3	32	128
23	19	6	102	5	3	24	133
24	19	6	102	5	3	24	128
25	19	6	102	5	3	24	129
26	19	6	102	5	3	24	125
27	19	6	102	5	3	24	129
28	19	6	102	5	3	24	129
29	19	6	102	5	3	24	126
30	19	6	102	5	3	24	126
Média	19	6	102	5.03	3	24.8	130.73
Desvio Padrão	0.00	0.00	0.00	0.18	0.00	2.14	4.44

Enquanto é possível observar que o método clássico gera circuitos grandes quando comparados àqueles gerados pelo método MC-CGP para o mesmo problema (tabela 4.12), a otimização ainda ocorre de forma bem sucedida. Neste caso, a redução alcançada no número de portas lógica é de n_p : 73.5%, no número de camadas de n_c : 50% e no número de transistores de n_t : 75.6%. Ou seja, ocorre uma considerável otimização, mesmo com os grandes circuitos iniciais.

Dessa forma, é possível concluir que otimizar circuitos gerados a partir de métodos clássicos é uma técnica viável, alcançando resultados semelhantes àqueles encontrados

pelo método HMC-CGP (seção 4.2.1).

Neste capítulo, primeiramente foi realizada uma verificação da afirmação feita por Walker et al. [2009] de que a estratégia MC-CGP é superior à estratégia evolutiva CGP para circuitos com múltiplas saídas em termos de eficiência na convergência. Como esperado, a resposta se mostrou positiva.

Em seguida, estratégia evolutiva HMC-CGP proposta é avaliada nos mesmos experimentos, buscando, além da solução correta a otimização das características selecionadas números de portas, número de camadas (caminho crítico) e número de transistores. Para cada experimento o método HMC-CGP apresentou melhorias consideráveis, até mesmo para circuitos complexos como o decodificador para *display* de 7 segmentos hexadecimal, reduzindo, em alguns casos, em até 50% o número de transistores quando comparada à estratégia MC-CGP.

Pelos experimentos realizados neste capítulo, é possível, então, comprovar a eficácia do algoritmo HMC-CGP proposto, que alcança soluções que não seriam possíveis pelo simples uso do método evolutivo MC-CGP, ou que demandariam um número impraticável de gerações caso apenas a estratégia CGP fosse utilizada.

No capítulo seguinte serão apresentadas as conclusões deste trabalho e indicações de trabalhos futuros.

Capítulo 5

Conclusões

Este trabalho procurou responder questões sobre a possibilidade de utilização de algoritmos genéticos para realizar a tarefa de projetar circuitos digitais. Enquanto um dos objetivos principais, o estudo de soluções sequenciais, não pôde ser realizado devido aos problemas técnicos descritos, resultados interessantes surgiram mesmo apenas para o projeto de circuitos combinacionais.

Questões como "é possível utilizar algoritmos genéticos para projetar circuitos digitais?", "esses projetos possuem alguma característica destoante de projetos humanos?", "além de encontrar a funcionalidade desejada, eles também conseguem otimizá-la?", foram os principais motivadores para a realização deste trabalho.

Primeiramente, foi conduzido um estudo detalhado sobre como álgebra booleana e seus métodos clássicos de otimização são formulados. Em seguida, estudou-se uma plataforma reconfigurável comum para prototipação e verificação de circuitos digitais, a FPGA. Conceitos sobre algoritmos genéticos foram apresentados, mostrando classificações, algoritmos comumente utilizados e os métodos evolutivos principais vistos neste trabalho, o CGP e o MC-CGP. Ao final, todas essas ideias foram reunidas para definir conceitos relacionados à área de *Hardware Evolutivo*.

Foi proposto um novo algoritmo evolutivo, denominado HMC-CGP, para realizar a tarefa de projeto e otimização de circuitos digitais. Considerando a dificuldade de convergência dos métodos evolutivos convencionais em atingir uma solução válida e ótima, o algoritmo proposto opera em duas etapas. A primeira etapa busca obter uma solução funcional que atenda aos requerimentos do problema em um tempo prático. A segunda etapa visa otimizar a solução encontrada em relação a três critérios básicos, o número de portas lógicas, o número de camadas (caminho crítico) e o número de transistores necessário à sua implementação em uma dada tecnologia. Deste modo espera-se obter uma solução otimizada que não seria viável ser encontrada por métodos convencionais.

Os problemas escolhidos para os experimentos foram a síntese dos circuitos somadores de 1 e 2 *bits*, multiplicador de 2 *bits* e decodificador *display* de 7 segmentos hexadecimal.

Para circuitos pequenos como o somador completo de 1 *bit*, a otimização alcançou, em média, reduções de 38.3% para o número de portas, 21.0% para o maior caminho crítico e 45.5% para o número de transistores a partir da solução funcional encontrada pela primeira etapa do algoritmo. Para o circuito decodificador para *displays* de 7 segmentos, as reduções encontradas foram 30.8%, 14.2% e 39.1% respectivamente. Os resultados

encontrados indicam, portanto, que o método HMC-CGP se mostra eficaz no projeto e otimização de circuitos digitais.

Muitos caminhos ainda podem ser investigados para que a evolução intrínseca ainda seja viável. O maior problema encontrado neste trabalho, o tempo necessário para a avaliação da população de indivíduos, pode ser melhorado utilizando comunicação USB entre o PC *host* e o dispositivo FPGA. Outra possível solução é a implementação de todo o algoritmo genético proposto diretamente na FPGA. Essas soluções demandam estudos mais aprofundados, porém possibilitariam a evolução de circuitos mais complexos como os necessários em sistemas digitais sequenciais.

Referências

32x8. Online karnaugh map solver with circuit for up to 8 variables. http://www.32x8.com/, 2015. [Online; acessado 30-07-2015]. 60

Anant Agarwal and Jeffrey H. Lang. *Foundations of Analog and Digital Electronic Circuits*. Denise E. M. Penrose, 1st edition, 2005. 5, 9, 10

Altera. 2. cyclone ii architecture. http://www.altera.com/literature/hb/cyc2/cyc2_cii51002.pdf, 2007. Acessado: 13-04-2014. vii, 21

Altera. De2-115 user manual. ftp://ftp.altera.com/up/pub/Altera_Material/12.1/Boards/DE2-115/DE2_115_User_Manual.pdf, 2010. Acessado: 18-04-2014. vii, 22, 24

Altera. Cyclone iv device handbook, volume 1. http://www.altera.com/literature/hb/cyclone-iv/cyclone4-handbook.pdf, 2013. Acessado: 18-04-2014. vii, 22, 23

José Nelson Amaral, Kagan Tumer, and Joydeep Ghosh. Designing genetic algorithms for the state assignment problem. *Systems, Man and Cybernetics, IEEE Transactions on*, 25(4):687–694, 1995. 20

S Asha and Dr. R Rani Hemamalini. Synthesis of adder circuit using cartesian genetic programming. *WSEAS Transactions on Circuits and Systems*, 14:83–88, 2015. 43

MJ Avedillo, JM Quintana, and JL Huertas. Smas: A program for the concurrent state reduction and state assignment of finite state machines. In *Circuits and Systems, 1991., IEEE International Symposium on*, pages 1781–1784. IEEE, 1991. 20

Mark Balch. *Complete digital design: a comprehensive guide to digital electronics and computer system architecture*. McGraw-Hill, Inc., 2003. 10

Ali Belgasem. *Evolutionary Algorithms for Synthesis and Optimization of Sequential Logic Circuits*. PhD thesis, Edinburgh Napier University, 2003. vii, 17, 19, 20, 33, 34

H. Beyer. Towards a theory of 'evolution strategies'. some asymptotical results from the (1,+lambda)-theory. *Evolutionary Computation*, 1:165–188, 1993. 31

John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory with applications*, volume 290. Macmillan London, 1976. 42

Sebastien Bourdeauducq. Simple rs232 uart. http://opencores.org/project,mmuart, 2010. [Online; acessado em 26-03-2015]. 46

Petr Burian. Evolvable hardware implemented by fpga. *Computer Applications in Electrical Engineering*, 2009. 2

RO Canham and AM Tyrrell. Evolved fault tolerance in evolvable hardware. In *Proceedings of IEEE Congress on Evolutionary Computation*. Citeseer, 2002. 33

Pong P. Chu. *FPGA Prototyping by Verilog Examples*. Wiley, 3rd edition, 2008. vii, 20, 22

Vitor Coimbra. Vitorcbsb/hw-verilog. https://github.com/VitorCBSB/hw-verilog, 2014. [Online; última atualização em 03-03-2015]. 46

Charles Darwin. On the origin of species. *Murray, London*, page 360, 1859. 1, 25

Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP Journal on Embedded Systems*, 2006. doi: 10.1155/ES/2006/56320. URL https://wiki.ittc.ku.edu/eecs700/images/8/80/Survey.pdf. 1

GW Timothy Gordon and J Peter Bentley. On evolvable hardware. In *Soft Computing in Industrial Electronics*, pages 279–323. Springer, 2002. viii, 33, 36

Venu G. Gudise and Ganesh K. Venayagamoorthy. Evolving digital circuits using particle swarm. *Neural Networks, 2003. Proceedings of the International Joint Conference on*, 2003. doi: 10.1109/IJCNN.2003.1223391. 11

James Hilder, James Alfred Walker, and Andy Tyrrell. Use of a multi-objective fitness function to improve cartesian genetic programming circuits. In *Adaptive Hardware and Systems (AHS), 2010 NASA/ESA Conference on*, pages 179–185. IEEE, 2010. 31, 42

Randall Hyde. *The Art of Assembly Language*. no starch press, 2001. 4

M. Morris Mano. *Digital Logic and Computer Design*. Prentice-Hall, 1st edition, 2006. 10, 12, 13

Mitchell Melanie. *An Introduction to Genetic Algorithms*. Bradford, 5th edition, 1999. 1, 2, 25

Uwe Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. Springer, 3rd edition, 2007. 21

Brad L. Miller, Brad L. Miller, David E. Goldberg, and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995. 28

Julian F Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142, 1999. vii, 28, 30

Julian F Miller. *Cartesian Genetic Programming*. Springer, 1st edition, 2011. vii, 29

Julian F Miller and Stephen L Smith. Redundancy and computational efficiency in cartesian genetic programming. *Evolutionary Computation, IEEE Transactions on*, 10(2): 167–174, 2006. 32

Julian F Miller, Dominic Job, and Vesselin K Vassilev. Principles in the evolutionary design of digital circuits—part i. *Genetic programming and evolvable machines*, 1(1-2): 7–35, 2000. 31

Victor P. Nelson, H. Troy Nagle, J. David Irwin, and Bill D. Carroll. *Digital Logic Circuit Analysis & Design*. Prentice-Hall, Inc., 1st edition, 1995. 4, 10

Adrian Stoica. On hardware evolvability and levels of granularity. 1997. 35, 36

David Tarnoff. *Computer Organization and Design Fundamentals*. Lulu, 2007. vii, 14, 15

Adrian Thompson. *Hardware Evolution*. Springer, 1998. vii, 2, 20, 33, 34, 35, 38

Jim Torresen. Evolvable hardware-a short introduction. In *Proceedings of the 1997 International Conference on Neural Information Processing and Intelligent Information Systems*, volume 1, pages 674–677, 1997. 35

Jim Torresen. An evolvable hardware tutorial. In *Field Programmable Logic and Application*, pages 821–830. Springer, 2004. vii, 21, 37

Teunis van Beelen. Rs-232 for linux, freebsd and windows. http://www.teuniz.net/RS-232/, 2015. [Online; acessado em 26-03-2015]. 46

Vesselin K Vassilev and Julian F Miller. The advantages of landscape neutrality in digital circuit evolution. In *Evolvable systems: from biology to hardware*, pages 252–263. Springer, 2000. 32

James Alfred Walker, Katharina Völk, Stephen L Smith, and Julian Francis Miller. Parallel evolution using multi-chromosome cartesian genetic programming. *Genetic Programming and Evolvable Machines*, 10(4):417–445, 2009. vii, 31, 32, 43, 53, 74

Wikipedia. Recombinação (computação evolutiva) — Wikipédia, a enciclopédia livre. http://pt.wikipedia.org/wiki/Recombina%C3%A7%C3%A3o_%28computa%C3%A7%C3%A3o_evolutiva%29, 2014a. [Online; acessado 19-04-2014]. vii, 26

Wikipedia. Flip-flop (electronics) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Flip-flop_(electronics), 2014b. vii, 15, 16

Wikipedia. Fitness proportionate selection — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Fitness_proportionate_selection, 2014c. vii, 27

Stephen Williams. Icarus verilog. http://iverilog.icarus.com, 2015. [Online; acessado em 01-08-2015]. 42

Huayang Xie and Mengjie Zhang. *Tuning Selection Pressure in Tournament Selection*. School of Engineering and Computer Science, Victoria University of Wellington, 2009. 28