

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Emulação de Circuitos Quânticos em Placa FPGA

Heron Aragão Monteiro

Campina Grande, Paraíba, Brasil

©Heron Aragão Monteiro, 31/05/2012

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Emulação de Circuitos Quânticos em Placa FPGA

Heron Aragão Monteiro

Dissertação submetida à Coordenação do Curso de Pós-Graduação em
Ciência da Computação da Universidade Federal de Campina Grande -
Campus I como parte dos requisitos necessários para obtenção do grau
de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Metodologia e Técnicas da Computação

Bernardo Lula Jr.

(Orientador)

Francisco Marcos de Assis

(Orientador)

Campina Grande, Paraíba, Brasil

©Heron Aragão Monteiro, 31/05/2012



DIGITALIZAÇÃO:
SISTEMOTECA - UFCG

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

M775e Monteiro, Heron Aragão.
Emulação de circuitos quânticos em placa FPGA / Heron Aragão
Monteiro. - Campina Grande, 2012.
109f.: il.

Dissertação (Mestrado em Ciência da Computação) – Universidade
Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Orientadores: Prof. Bernardo Lula Jr., Prof. Francisco Marcos de Assis.
Referências.

1. Circuitos de Processamento. 2. Emulação. 3. Circuitos Quânticos.
4. FPGA. I. Título.

CDU 004.31 (043)

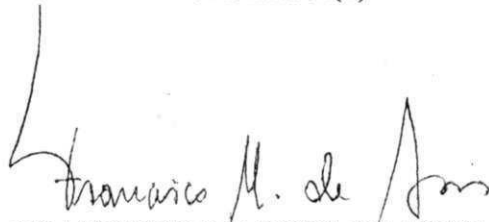
"EMULAÇÃO DE CIRCUITOS QUÂNTICOS EM PLACA FPGA"

HERON ARAGÃO MONTEIRO

DISSERTAÇÃO APROVADA EM 31/05/2012



BERNARDO LULA JUNIOR, Dr.
Orientador(a)



FRANCISCO MARCOS DE ASSIS, Dr.
Orientador(a)



ELMAR UWE KURT MELCHER, Dr.
Examinador(a)



ADRIANO DE ALBUQUERQUE BATISTA, Ph.D
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Com o avanço da nanotecnologia, a computação quântica tem recebido grande destaque no meio científico.

Utilizando os fundamentos da mecânica quântica, têm sido propostos diversos algoritmos quânticos. E, até então, os mesmos têm apresentado ganhos significativos com relação às suas versões clássicas.

Na intenção de poder ser verificada a eficiência dos algoritmos quânticos, diversos simuladores vêm sendo desenvolvidos, visto que a confecção de um computador quântico ainda não foi possível.

Há duas grandes vertentes de simuladores: os simuladores por software e os simuladores por hardware, chamados de emuladores. Na primeira classe se encontram os programas desenvolvidos em um computador clássico, procurando implementar os fundamentos da mecânica quântica, fazendo uso das linguagens de programação clássicas. Na segunda, são utilizados recursos que não estejam vinculados à plataforma do computador clássico.

Dentre os emuladores, particularmente, estudos têm sido realizados fazendo uso de hardware dedicado (mais especificamente, FPGA's¹).

O presente trabalho propõe a verificação da real utilidade da plataforma FPGA, com a intenção de se desenvolver um emulador universal, que permita a emulação de qualquer classe de circuitos, e que os mesmos possam ser implementados com um maior número de q-bits em relação aos circuitos tratados nos trabalhos anteriores.

¹Field-Programmable Gate Arrays

Abstract

With the progress of nanotechnology, quantum computing has received great emphasis in scientific circles.

Using the basis of quantum mechanics, different quantum algorithms have been proposed. And so far, they have presented significant gains with respect to its classic versions.

In order to verify the efficiency of quantum algorithms, several simulators have been developed, since the construction of a quantum computer is not yet possible.

There are two major classes of simulators, simulators via software and via hardware. The latter being also called emulators. In the first class, programs are developed in a classical computer, attempting to implement the fundamentals of quantum mechanics, making use of classic programming languages. In the second, resources are used that are not related to the classic computer platform.

Among the emulators, in particular, studies have been made using dedicated hardware (more specifically, FPGA's²).

The present work proposes the use of the FPGA boards in emulation of quantum circuits aiming a gain scale in relation to the alternatives presented so far. The present work proposes checking the usefulness of the FPGA with the intention of developing an universal emulator that is able to emulate any type of circuit, and that they can be implemented with a larger number of q-bit in respect to the circuits treated in the previous works.

²Field-Programmable Gate Arrays

Agradecimentos

A Deus.

Aos meus pais, início de tudo.

À minha esposa, meu porto seguro.

Aos meus filhos, incentivo de vida.

Aos meus irmãos, cunhadas, cunhado e sobrinhos.

Aos meus sogros, acolhimento total.

Aos meus orientadores e professores, guias nessa jornada.

À HP.

Ao IQuanta/UFCG.

Ao LAD/UFCG.

Aos colegas de curso.

Conteúdo

1	Introdução	1
2	Computação Quântica	4
2.1	Emaranhamento quântico	7
2.2	Paralelismo quântico	7
3	Circuitos Quânticos	8
3.1	Portas Quânticas	9
3.1.1	Portas quânticas de 1 q-bit	9
3.1.2	Operações Controladas	12
4	Algoritmos Quânticos	13
4.1	Algoritmo de Deutsch	13
4.2	Algoritmos Aritméticos	15
4.3	Transformada de Fourier Quântica	15
4.4	Algoritmo de Grover	15
5	Simulação e Emulação de Circuitos Quânticos	21
5.1	Simuladores Universais	24
5.2	Simuladores Não-universais	25
5.3	Emuladores	25
6	Recursos para Emulação de Circuitos Quânticos	28
6.1	Placas FPGA	28
6.2	Ferramentas de Desenvolvimento	30

6.3	Linguagens de Descrição de Hardware	31
6.4	O Simulador Zeno	33
7	Solução Proposta	34
7.1	Decisões de Implementação	34
7.2	Recursos Computacionais	35
7.3	A Solução	36
7.3.1	Q-bits	36
7.3.2	Registrador Quântico	37
7.3.3	Portas Lógicas	38
7.3.4	Eficiência da Implementação	46
8	Resultados	48
8.1	Circuitos Aritméticos	50
8.1.1	Circuito 3_17tc	50
8.1.2	Circuito ham3tc	51
8.1.3	Circuito Graycode	52
8.1.4	Circuito rd53d2	53
8.1.5	Circuito 6sym	55
8.1.6	Circuito gf2 ⁴ mult	57
8.2	O Circuito da TFQ	59
8.3	O Circuito de Grover	66
8.4	Análise dos Resultados	71
9	Conclusão	75
A	Mudança de Proposta de Trabalho	84
B	Circuito de Grover para 3 q-bits	86
C	Circuito de Grover para 7 q-bits	89

Lista de Símbolos

CAS - *Computer Algebra System*

DPI - *Direct Programming Interface*

FPGA - *Field-Programmable Gate Arrays*

TQF - *Transforma Quântica de Fourier*

VHDL - *VHSIC Hardware Description Language*

VHSIC - *Very High Speed Integrated Circuits*

Lista de Figuras

1.1	Lei de Moore.	1
2.1	Esfera de Bloch.	6
3.1	Circuito Quântico.	8
4.1	Circuito quântico do Algoritmo de Deutsch.	14
4.2	Atuação de U_f sobre o estado $ \psi\rangle$	19
4.3	Reflexão de $ \psi_1\rangle$	20
5.1	Etapas do processo de simulação.	22
6.1	Célula Lógica.	29
6.2	Interconexão entre células lógicas.	29
6.3	Esquema de construção de bloco lógico.	30
6.4	Ligação entre células lógicas vizinhas.	30
6.5	Linguagem SystemVerilog.	32
7.1	Placa DE2-70.	35
7.2	Representação interna das amplitudes.	37
8.1	Circuito Quântico.	49
8.2	Circuito Aritmético <i>3_17tc</i>	50
8.3	Circuito Aritmético <i>ham3tc</i>	51
8.4	Circuito Aritmético Graycode.	53
8.5	Circuito Aritmético <i>rd53d2</i>	54
8.6	Circuito Aritmético <i>6sym</i>	56
8.7	Circuito Aritmético <i>gf2^4mult</i>	58

8.8	Circuito da TFQ de 3 q-bits.	59
8.9	Circuito da TFQ de 4 q-bits.	60
8.10	Circuito da TFQ de 5 q-bits.	60
8.11	Circuito da TFQ de 6 q-bits.	60
8.12	Circuito Quântico de Grover para 4 q-bits.	66
8.13	Grupo iterador de Grover para 4 q-bits.	67

Lista de Códigos Fonte

7.1	Precisão	37
7.2	Quantidade de q-bits	37
7.3	Amplitudes	38
7.4	Apontadores para os vetores de amplitudes	38
7.5	Primeiro passo do operador de Hadamard	39
7.6	Inverso da raiz quadrada de dois	39
7.7	Função Multiplica	39
7.8	Segundo passo do operador de Hadamard	40
7.9	Porta lógica Not	40
7.10	Porta lógica CNot	41
7.11	Porta Toffoli	42
7.12	Porta lógica CCNot_m	43
7.13	Porta lógica Z	44
7.14	Porta lógica Z	44
7.15	Porta lógica Rotação	45
7.16	Porta S	45
7.17	Variável porta lógica	46
7.18	Exemplo de Chamada de Porta Lógica	47
8.1	Inicialização dos estados quânticos	49
8.2	Código do Circuito Aritmético <i>3_17tc</i>	50
8.3	Código do Circuito Aritmético <i>ham3tc</i>	51
8.4	Código do Circuito Aritmético Graycode	52
8.5	Código do Circuito Aritmético <i>rd53d2</i>	54
8.6	Código do Circuito Aritmético <i>6sym</i>	55

8.7	Código do Circuito Aritmético $gf2^4$ mult	57
8.8	Código da TFQ de 3 q-bits	60
8.9	Código da TFQ de 4 q-bits	61
8.10	Código da TFQ de 5 q-bits	62
8.11	Código da TFQ de 6 q-bits	64
8.12	Circuito de Grover com 4 q-bits	67
	DE2_70_TOP_07q.sv	89

Capítulo 1

Introdução

Em 1965, o então presidente da Intel, Gordon Moore, previu que a quantidade de transistores dos chips dobraria a cada dezoito meses, sem aumento de custo [59]. Em 1975 o próprio Gordon reformulou sua lei afirmando que essa quantidade dobraria a cada dois anos [32].

Nos últimos anos a sua previsão vem se confirmando, como pode ser visto na Figura 1.1. Porém pesquisadores apontam para o fim dessa lei, principalmente pelo fato de haver limites físicos para que os transistores reduzam de tamanho.

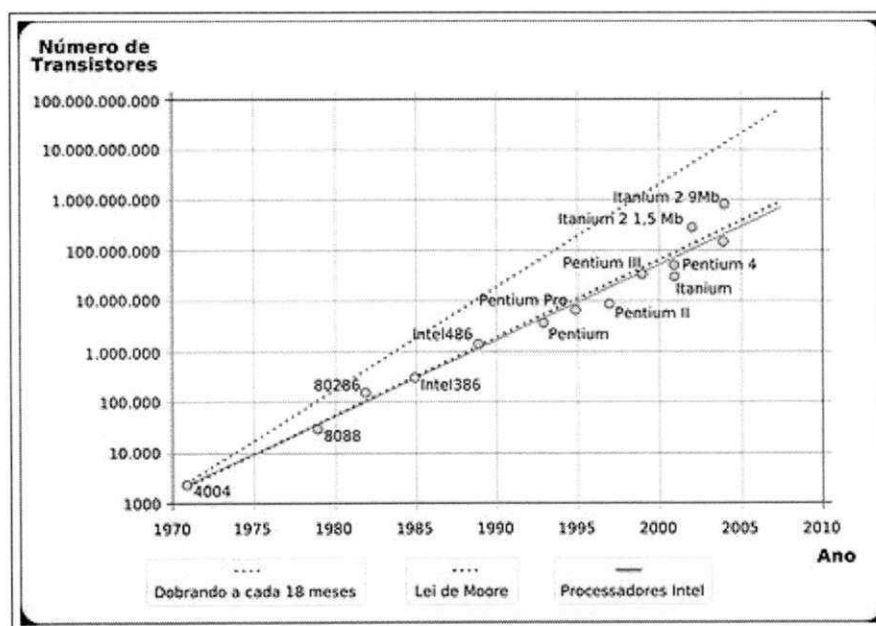


Figura 1.1: Lei de Moore.

Fonte: http://pt.wikipedia.org/wiki/Ficheiro:Lei_de_moore_2006.svg.png, em 24/01/2012.

Mesmo antes de ter contato com a lei de Moore, em 1959, Richard Feynman, no encontro anual da Sociedade Americana de Física, declarou não haver obstáculo, a nível teórico, para a construção de pequenos dispositivos, podendo inclusive chegar à escala atômica. Sendo assim proposta a nanotecnologia [60].

Feynman, em 1982, chegou à conclusão que os sistemas clássicos existentes não seriam capazes de modelar, eficientemente, os sistemas mecânicos quânticos, sugerindo que computadores baseados nas leis da mecânica quântica deveriam ser usados para modelar tais sistemas [9]. Como definido por Fahdil [11] e Viamontes [56], um computador quântico só poderá ser assim chamado quando usar, para executar operações em dados, os fenômenos quânticos de superposição e emaranhamento.

Mesmo não sendo alcançada a construção de tal dispositivo, estudos na área de informação quântica têm avançado e diversos algoritmos quânticos vêm sendo propostos [36], sendo utilizado o modelo matemático para sua construção. Porém, esse modelo não é de fácil assimilação e manipulação por pessoas que não estejam muito familiarizadas com os fundamentos da matemática e com as leis da mecânica quântica.

Como alternativa à abstração do modelo matemático, com o interesse de disseminar os conhecimentos adquiridos, verificar a funcionalidade dos algoritmos propostos e auxiliar no desenvolvimento de novos algoritmos, têm sido desenvolvidas soluções a nível de software (simuladores) e a nível de hardware (emuladores).

Os simuladores utilizam a linguagem de circuitos quânticos para a descrição de algoritmos quânticos. Dentre eles há uma grande preocupação com a facilidade de uso, sendo fornecidas interfaces que permitem a rápida construção e teste dos circuitos, sendo, na maioria dos casos, suas estruturas internas convertidas para um modelo matemático, representado geralmente por matrizes. Porém, a execução desse modelo matemático consome uma grande quantidade de recursos computacionais, impossibilitando a construção de circuitos com uma quantidade maior de q-bits, além de não contemplar o paralelismo quântico.

Na intenção de implementar o paralelismo quântico, e conseqüentemente, possibilitar a manipulação de circuitos quânticos com um maior número de q-bits, vêm sendo propostas soluções de emulação, fazendo uso de placas FPGA. Nas soluções apresentadas é possível se observar que há uma preocupação com a implementação dos fundamentos da mecânica quântica. São implementados algoritmos quânticos que, quando comparados com simulado-

res, apresentam resultados significativamente superiores, quanto ao tempo de execução. As soluções implemetadas lidam com circuitos com poucos q-bits, mas os autores, nas seções de trabalhos futuros, vislumbram a possibilidade de serem manipulados circuitos com um número significativo de q-bits, tornando a plataforma atrativa.

O objetivo do presente trabalho¹ é verificar se realmente a emulação, em placas FPGA, pode manipular circuitos quânticos com um número considerável de q-bits, tendo como pressuposto o paralelismo quântico, sem abrir mão da universalidade da solução, ou seja, da possibilidade da implementação de qualquer algoritmo quântico.

O presente trabalho está assim organizado: no capítulo 2 é fornecida uma introdução à computação quântica e seus conceitos fundamentais.

No capítulo 3 é apresenta a linguagem de circuitos quânticos, descrevendo suas notações e principais estruturas.

Os algoritmos quânticos são explorados no capítulo 4, dando ênfase ao algoritmo de Grover, objeto de estudo desse trabalho.

No capítulo 5 são descritas as técnicas de simulação e emulação de circuitos quânticos, destacando algumas soluções e suas características principais.

O capítulo 6 destaca os recursos de hardware e software necessários para a emulação de circuitos quânticos.

No capítulo 7 é apresentada a solução proposta, sendo seus resultados analisados no capítulo 8.

Finalmente, são feitas as conclusões do trabalho no capítulo 9.

¹Veja apêndice A.

Capítulo 2

Computação Quântica

Enquanto que na computação clássica a unidade de representação da informação é o bit, na computação quântica essa unidade recebe o nome de qubit (quantum bit) ou q-bit [36]. Um bit, utilizando a base computacional, pode assumir um valor 0 ou um valor 1. Já o q-bit pode assumir um valor 0, um valor 1 ou uma combinação linear desses dois valores, chamados de estados básicos [4].

Utilizando a notação de Dirac¹, o q-bit é dado pela equação

$$|\psi\rangle = \alpha_1|0\rangle + \alpha_2|1\rangle, \quad (2.1)$$

onde $|0\rangle$ e $|1\rangle$ são vetores que pertencem ao espaço vetorial complexo e formam uma base ortonormal desse espaço [34] e α_1 e α_2 (eq. 2.1), chamados de amplitudes, são números complexos² que satisfazem a relação:

$$|\alpha_1|^2 + |\alpha_2|^2 = 1. \quad (2.2)$$

$|\alpha_1|^2$ e $|\alpha_2|^2$ representam as probabilidades do q-bit estar, respectivamente, nos estados $|0\rangle$ ou $|1\rangle$, após a aplicação de uma operação de medição [36]. O estado quântico que satisfaz a equação 2.2 é dito normalizado. Quando as amplitudes α_1 e α_2 são diferentes de zero, $|\psi\rangle$ (eq. 2.1) é um estado que está em superposição dos elementos da base.

Em notação matricial [22], cada estado da base computacional pode ser assim representado:

¹Paul Adrien Maurice Dirac (1902-1984), físico teórico britânico.

²Números na forma $x + iy$, onde x e $y \in \mathbb{R}$ e $i^2 = -1$.

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{e} \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \quad (2.3)$$

Conseqüentemente, um estado $|\psi\rangle$ qualquer pode ser escrito como:

$$|\psi\rangle = \alpha_1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \alpha_2 \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix}. \quad (2.4)$$

Tendo como exemplo um sistema de dois q-bits, seu estado pode ser representado pela equação:

$$|\psi\rangle = \alpha_1 \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \alpha_2 \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \alpha_3 \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \alpha_4 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{pmatrix}. \quad (2.5)$$

De uma forma geral, um sistema que possua um registrador de n q-bits poderá armazenar até 2^n estados simultaneamente, sendo representado pelas equações 2.6 e 2.7.

$$|\psi\rangle = \alpha_1 |00 \dots 0\rangle_{2^n} + \alpha_2 |01 \dots 0\rangle_{2^n} + \alpha_{2^n} |11 \dots 1\rangle_{2^n} = \sum_{i=1}^{2^n} \alpha_i |\psi_i\rangle \quad (2.6)$$

$$|\psi\rangle = \alpha_1 \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}_{2^n} + \alpha_2 \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix}_{2^n} + \dots + \alpha_{2^n} \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}_{2^n} = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{2^n} \end{pmatrix}. \quad (2.7)$$

Alternativamente, um q-bit pode ser representado geometricamente, em \mathfrak{R}^3 , podendo seu estado quântico ser dado pela equação:

$$|\psi\rangle = e^{i\gamma} (\cos(\theta/2)|0\rangle + e^{i\varphi} \sin(\theta/2)|1\rangle), \quad (2.8)$$

onde γ, θ e $\varphi \in \mathfrak{R}$.

A equação 2.8 pode ser reescrita na forma abaixo (eq. 2.9), visto que o elemento $e^{i\gamma}$, conhecido como fator de fase global, não possui efeito físico observável, já que $|e^{i\gamma}| = 1$ [25].

$$|\psi\rangle = \cos(\theta/2)|0\rangle + e^{i\varphi} \sin(\theta/2)|1\rangle \quad (2.9)$$

Cada elemento dessa equação é conhecido como vetor de Bloch e o conjunto desses vetores forma a esfera de Bloch, unitária em \mathfrak{R}^3 , esquematizada na figura 2.1, onde o ângulo $\theta \in [0, \pi/2]$ indica as quantidades de $|0\rangle$ e $|1\rangle$ do estado $|\psi\rangle$ e o ângulo $\varphi \in [0, \pi/2]$ é a fase relativa e é medido tomando como base o eixo X e a projeção de $|\psi\rangle$ no plano XY .

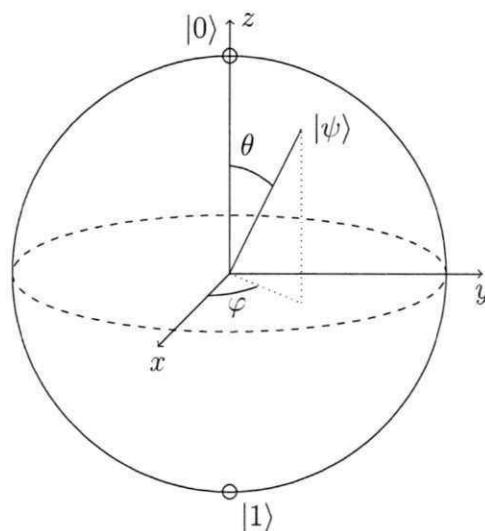


Figura 2.1: Esfera de Bloch.

Um sistema quântico, em cada instante t , é representado por um estado $|\psi\rangle$. Dessa forma, é dito que o estado quântico é uma função direta do tempo [22], como segue:

$$|\psi(t)\rangle \quad (2.10)$$

Assim, o sistema quântico evolui com o tempo. E, de acordo com a teoria quântica, essa evolução é linear, ou seja, se aplica a todos os estados do sistema quântico.

Como afirma o segundo postulado da mecânica quântica, a evolução de um sistema quântico fechado se dá por meio de transformações unitárias [36], representadas por operadores unitários³. Então, para que um sistema quântico evolua de um estado $|\psi_1\rangle$ para um estado $|\psi_2\rangle$ é necessária a aplicação de um operador unitário U a esse primeiro estado.

$$|\psi_2\rangle = U|\psi_1\rangle \quad (2.11)$$

Considerando $|\psi_1\rangle$ da forma dada pela equação 2.6, tem-se:

³Um operador é dito unitário quando satisfizer a equação $U^\dagger U = I$.

$$U|\psi_1\rangle = U \sum_{i=1}^{2^n} \alpha_i |\psi_i\rangle = \sum_{i=1}^{2^n} \alpha_i U|\psi_i\rangle = |\psi_2\rangle. \quad (2.12)$$

Cada operador U deve ter a dimensão do espaço de estados.

2.1 Emaranhamento quântico

De acordo com o postulado da composição de sistemas, dois sistemas isolados podem formar um sistema combinado, tendo como espaço de estados resultante o produto tensorial de cada um dos espaços de estados originais. Se dois sistemas possuem seus respectivos espaços de estados $|a\rangle$ e $|b\rangle$, o sistema combinado desses dois subsistemas terá, como espaço de estados:

$$|a\rangle \otimes |b\rangle. \quad (2.13)$$

Contudo, essa representação só será válida se os dois subsistemas forem preparados e mantidos isolados. Se houver alguma interação entre eles, poderá acontecer o caso de não ser possível escrever o espaço de estados como produto tensorial dos estados originais. E, nesse caso, é dito que o estado está emaranhado. Assim, um estado $|\psi\rangle$ é dito emaranhado quando não for possível escrevê-lo na forma:

$$|\psi\rangle = |a\rangle \otimes |b\rangle \quad (2.14)$$

2.2 Paralelismo quântico

Como dito inicialmente, um sistema quântico pode armazenar, ao mesmo tempo, mais de um estado da base. Se o sistema for composto de n q-bits, o seu registrador poderá armazenar, simultaneamente, até 2^n estados distintos e é dito que estes estados estão em superposição.

Estando um sistema quântico em superposição de estados da base, a aplicação de um operador unitário, como demonstrado pela equação 2.12, se dá em cada um desses estados, simultaneamente, em uma única operação, o que caracteriza o paralelismo quântico.

Com o paralelismo quântico é possível ser feita a aplicação de uma função, uma única vez, para todas as entradas em superposição, produzindo todas as possíveis saídas, também em superposição [49].

Capítulo 3

Circuitos Quânticos

Um circuito quântico é um modelo usado na computação quântica para representar a aplicação de uma sequência finita de operadores a uma determinada quantidade de q-bits de entrada.

Em um circuito quântico, como esquematizado na figura 3.1, o ponto de partida é o seu estado inicial, que pode estar ou não emaranhado, representado por $|\psi\rangle$. Cada linha representa a evolução do q-bit no tempo; diferentemente da representação clássica, onde cada linha representa um fio por onde o bit trafega.

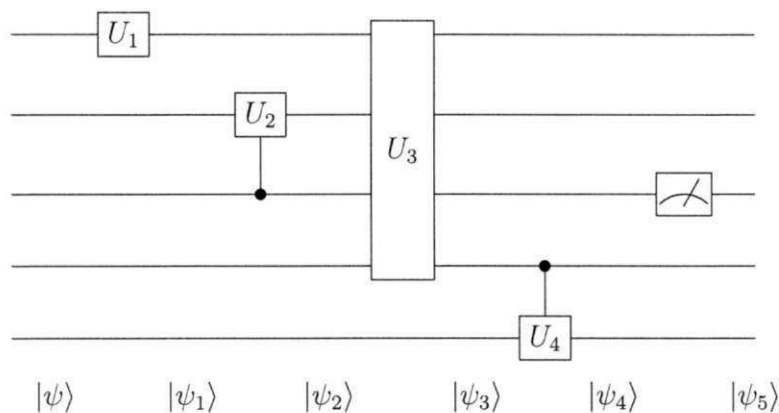


Figura 3.1: Circuito Quântico.

O símbolo $\boxed{U_1}$ representa uma porta lógica quântica, que executa uma operação unitária, aplicada ao q-bit associado à esta linha. As linhas verticais indicam que a execução de uma porta lógica está condicionada aos valores dos outros q-bits marcados com um ponto (portas lógicas $\boxed{U_2}$ e $\boxed{U_4}$).

O símbolo representado pela caixa de nome U_3 indica sobre quais q-bits o operador

dessas portas lógicas.

Porta NOT quântica

Classicamente a porta lógica NOT tem o objetivo de inverter o valor do bit de entrada ($0 \rightarrow 1$ e $1 \rightarrow 0$). Quanticamente a ideia é a mesma, mas deve ser considerado que o q-bit onde a porta atua pode estar em uma superposição de seus estados da base. Assim, a porta lógica NOT¹, atuando num estado $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, resultará no estado $|\psi_1\rangle = \beta|0\rangle + \alpha|1\rangle$, evidenciando que o operador atua linearmente em todos os estados da base, já que os valores das amplitudes dos dois estados foram trocados. A porta X representa uma rotação da esfera de Bloch no eixo X.

Utilizando a notação matricial, a porta NOT e sua aplicação podem ser representadas pelas equações 3.2 e 3.3, respectivamente.

$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (3.2)$$

$$X \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix} \quad (3.3)$$

Como pode ser visto, um operador de um q-bit pode ser representado por uma matriz quadrada de duas dimensões, que tem como única restrição a unitariedade, visando a preservação da condição de normalização dos estados quânticos.

Portas de Pauli

Juntamente com a porta X, as portas I, Y e Z compõem o conjunto das chamadas portas de Pauli.

A porta lógica I é o operador identidade, que preserva, após a sua atuação, inalterado um estado quântico, e é representada pela matriz:

$$I \equiv \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (3.4)$$

¹Também conhecida como porta de Pauli X

As portas lógicas Y e Z executam uma rotação da esfera de Bloch nos eixos Y e Z, respectivamente, tendo sua representação matricial da forma das equações 3.5 e 3.6.

$$Y \equiv \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad (3.5)$$

$$Z \equiv \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad (3.6)$$

Porta Hadamard

A porta de Hadamard é amplamente utilizada nos circuitos quânticos. Suas representações matricial e algébrica são dadas pelas equações 3.7 e 3.8.

$$H \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (3.7)$$

$$H|\psi\rangle = \alpha \frac{|0\rangle + |1\rangle}{\sqrt{2}} + \beta \frac{|0\rangle - |1\rangle}{\sqrt{2}} \quad (3.8)$$

Ao ser aplicado o operador de Hadamard a um q-bit no estado puro ($|0\rangle$ ou $|1\rangle$), é obtido um estado que é a superposição dos estados $|0\rangle$ ou $|1\rangle$ com igual probabilidade, como representado nas equações 3.9 e 3.10.

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad (3.9)$$

$$H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \quad (3.10)$$

Porta S

Porta S, ou porta de fase, introduz uma fase relativa $i = e^{i\pi/2}$ ao estado do q-bit onde a mesma é aplicada [25], tendo sua atuação representada pela matriz abaixo.

$$S \equiv \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \quad (3.11)$$

Porta T

Porta T, também conhecida como porta $\frac{\pi}{8}$, corresponde a uma rotação de $\frac{\pi}{4}$ radianos em relação ao eixo z.

$$T \equiv \begin{bmatrix} 1 & 0 \\ 0 & \exp(i\pi/4) \end{bmatrix} \quad (3.12)$$

Portas de Rotação

As portas S e T são exemplos particulares das portas de rotação, ou de fase. Generalizando, tais portas podem ser representadas como descrito abaixo.

$$R_k \equiv \begin{bmatrix} 1 & 0 \\ 0 & \exp(2\pi i/2^k) \end{bmatrix} \quad (3.13)$$

3.1.2 Operações Controladas

O modelo de porta lógica quântica controlada é a porta c-NOT (não-controlado²). Ela possui dois q-bits de entrada conhecidos como controle e alvo. Nessa porta o q-bit alvo terá seu valor alterado apenas quando o q-bit de controle tiver valor $|1\rangle$, como representado abaixo (eq. 3.14):

$$|00\rangle \rightarrow |00\rangle; |01\rangle \rightarrow |01\rangle; |10\rangle \rightarrow |11\rangle; |11\rangle \rightarrow |10\rangle; \quad (3.14)$$

Matricialmente essa porta pode ser assim representada (eq. 3.15):

$$U_{cN} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (3.15)$$

Uma extensão da porta c-NOT é a porta Toffoli, com três q-bits de entrada, sendo dois controles e um alvo.

²Do inglês Controlled-NOT

Capítulo 4

Algoritmos Quânticos

Analogamente aos algoritmos clássicos, os algoritmos quânticos são procedimentos finitos, executados passo-a-passo, utilizando um conjunto limitado de instruções, visando a solução de um problema, nesse caso em particular, usando os conceitos e paradigmas da computação quântica.

Geralmente os algoritmos quânticos são descritos usando o modelo de circuitos quânticos, visto no capítulo anterior.

Apesar do número de algoritmos quânticos ainda ser limitado, é inquestionável a sua capacidade de resolver problemas computacionais mais rapidamente que as soluções clássicas disponíveis [18].

A seguir serão destacados alguns desses algoritmos que, com exceção do primeiro, serão objeto de estudo do presente trabalho.

4.1 Algoritmo de Deutsch

O primeiro algoritmo quântico foi proposto por Deutsch¹ em 1985 [10]. O problema consistia em decidir se uma função binária $f : \{0, 1\} \rightarrow \{0, 1\}$ era constante ou balanceada. Uma função f é dita balanceada quando há a mesma quantidade de 0s e 1s na saída.

Classicamente, para se decidir que f é balanceada ou não, considerando x um argumento binário da função, é preciso calcular $f(x)$ para seus dois possíveis valores, ou seja, é preciso executar a função duas vezes. Deutsch provou ser possível uma solução quântica fazendo

¹David Deutsch, físico israelense.

uma única execução do algoritmo [33].

Usando o modelo de circuitos quânticos, o algoritmo de Deutsch pode ser esquematizado como na figura abaixo.

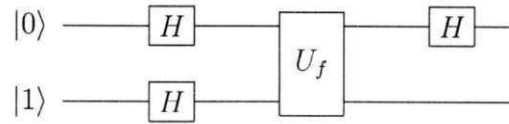


Figura 4.1: Circuito quântico do Algoritmo de Deutsch.

Nesse circuito a função é representada pela caixa preta rotulada de U_f . Geralmente essa caixa preta é denominada de oráculo, já que não é necessário saber como a mesma é implementada, e sim quais são os resultados que ela gera.

Estando o sistema quântico binário em seu estado inicial (eq. 4.1), é aplicado o operador de Hadamard aos dois q-bits, resultando no estado intermediário $|\psi_1\rangle$ (eq. 4.2) que, ao passar pelo oráculo, gera o estado $|\psi_2\rangle$ (eq. 4.3). Concluindo, ao aplicar o operador de Hadamard no primeiro q-bit, tem-se como resultado final o estado quântico $|\psi_3\rangle$ (eq. 4.4).

$$|\psi_0\rangle = |01\rangle \quad (4.1)$$

$$|\psi_1\rangle = \left[\frac{|0\rangle + |1\rangle}{\sqrt{2}} \right] \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right] \quad (4.2)$$

$$|\psi_2\rangle = \begin{cases} \pm \left[\frac{|0\rangle + |1\rangle}{\sqrt{2}} \right] \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right] & \text{se } f(0) = f(1) \\ \pm \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right] \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right] & \text{se } f(0) \neq f(1) \end{cases} \quad (4.3)$$

$$|\psi_3\rangle = \begin{cases} \pm |0\rangle \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right] & \text{se } f(0) = f(1) \\ \pm |1\rangle \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right] & \text{se } f(0) \neq f(1) \end{cases} \quad (4.4)$$

De modo geral, o estado $|\psi_3\rangle$ pode ser escrito na forma da equação 4.5 e, dessa forma, após ser feita uma medição no primeiro q-bit, aparecerá o valor 0 se a função for constante e 1 se a mesma for balanceada, gerando o resultado desejado com apenas uma avaliação da função $f(x)$.

$$|\psi_3\rangle = \pm |f(0) \oplus f(1)\rangle \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right] \quad (4.5)$$

4.2 Algoritmos Aritméticos

É indiscutível que os algoritmos quânticos têm sido objeto de grande interesse de estudo por parte dos pesquisadores. Porém o computador quântico, quando construído, deverá estar preparado para executar também tarefas mais simples como as operações aritméticas.

Dessa forma, diversas soluções de simuladores e emuladores se utilizam dessa classe de algoritmos para demonstrar a versatilidade de suas soluções.

4.3 Transformada de Fourier Quântica

A Transformada de Fourier Quântica (TFQ) é a versão quântica da Transformada Discreta de Fourier. Ela é a peça principal para muitos algoritmos quânticos, dentre eles o algoritmo de fatoração quântica e o algoritmo do cálculo da ordem de um número [36].

A TFQ é executada pelo operador unitário F , sob um estado quântico $|\psi\rangle$, representado pela equação 4.6[26].

$$F|\psi\rangle = \sum_{k=0}^{N-1} b_k |k\rangle, \text{ onde}$$

$$b_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} a_j e^{2\pi i j k / N} \quad (4.6)$$

Segundo demonstrado em estudos sobre a TFQ ([26], [36], [39], dentre outros), a mesma pode ser implementada fazendo-se uso das portas de Hadamard e de fase (ou rotação), portas essas citadas no capítulo anterior.

4.4 Algoritmo de Grover

Em 1996 Grover [17] desenvolveu o algoritmo quântico de busca, cujo objetivo é encontrar, em uma lista desordenada, um elemento específico que está nessa lista.

Dada uma lista L de tamanho $N - 1$, onde $N = 2^n$ e $n \in \mathbb{N}$, a função f (eq. 4.7), definida na forma da equação 4.8, retornará um valor 1 caso o elemento w procurado esteja na lista L e retornará 0 caso contrário.

$$f : |0, 1, \dots, N - 1\rangle \rightarrow |0, 1\rangle \quad (4.7)$$

$$f(w) = \begin{cases} 1, & \text{se } w \in L, \\ 0, & \text{se } w \notin L. \end{cases} \quad (4.8)$$

Considerando que há apenas uma única solução w para a função f , classicamente, no pior caso, as soluções são encontradas, com probabilidade de no mínimo $\frac{2}{3}$, após a aplicação de $\Omega(2^n)$ consultas à lista [22].

Quanticamente, com a utilização do algoritmo de Grover, a quantidade de buscas é de $O(\sqrt{2^n})$.

O algoritmo de Grover tem como entradas dois registradores: um com n q-bits, que contem os elementos da lista onde será efetuada a busca; e outro com 1 q-bit.

Ao primeiro registrador, inicializado com o estado $|0 \dots 0\rangle$, é aplicado o operador de Hadamard, com o objetivo de formar uma superposição de todos os estados da base, gerando o estado $|\psi\rangle$, que conterà, nesse caso, todos os elementos de busca da lista L .

A ideia central do algoritmo de Grover é fazer a separação dos elementos que são solução dos que não são solução do problema. De modo geral, a atuação do operador U_f pode ser assim esquematizada:

$$\frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle|0\rangle \xrightarrow{U_f} \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |w\rangle|1\rangle + \sqrt{\frac{N-1}{N}} |\psi\rangle|0\rangle, \quad (4.9)$$

onde $|\psi\rangle$ representa todos os estados da base com exceção do elemento w .

Como dito no capítulo anterior, quando é aplicado um operador de Hadamard a um q-bit no estado $|1\rangle$ é obtido o estado quântico $\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)$, que pode ser representado como $|-\rangle$.

A aplicação de um operador U_f qualquer a um único estado quântico $|x\rangle$ (no primeiro registrador), concatenado com o estado $|-\rangle$ (no segundo registrador), pode ser representado como segue:

$$\begin{aligned}
U_f : |x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) &= \left(\frac{U_f|x\rangle|0\rangle - U_f|x\rangle - |1\rangle}{\sqrt{2}} \right) \\
&= \left(\frac{|x\rangle|0 \oplus f(x)\rangle - |x\rangle|1 \oplus f(x)\rangle}{\sqrt{2}} \right) \\
&= |x\rangle \left(\frac{|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle}{\sqrt{2}} \right) \tag{4.10}
\end{aligned}$$

Analisando o termo $\left(\frac{|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle}{\sqrt{2}} \right)$, tem-se:

$$\begin{aligned}
\left(\frac{|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle}{\sqrt{2}} \right) &= \frac{|0\rangle - |1\rangle}{\sqrt{2}} \quad \text{se } f(x) = 0 \\
\left(\frac{|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle}{\sqrt{2}} \right) &= - \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \quad \text{se } f(x) = 1
\end{aligned}$$

ou seja,

$$\left(\frac{|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle}{\sqrt{2}} \right) = (-1)^{f(x)} \tag{4.11}$$

Aplicando o resultado da equação 4.11 em 4.10, tem-se:

$$U_f : |x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) = (-1)^{f(x)} |x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \tag{4.12}$$

Dessa forma, tendo um primeiro registrador de entrada do algoritmo de Grover como uma superposição de todos os estados da base (entradas da lista L), se o segundo registrador for inicializado com $|1\rangle$, e for aplicado o operador de Hadamard nesse segundo registrador, a atuação do operador U_f poderá ser assim representada:

$$\begin{aligned}
U_f(|\psi\rangle|-\rangle) &= U_f \left(\left(\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle \right) |-\rangle \right) \\
&= U_f \left(\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle |-\rangle \right) \\
&= \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} U_f(|i\rangle|-\rangle) \tag{4.13}
\end{aligned}$$

Aplicando o resultado obtido na equação 4.12, a atuação do operador U_f , dada pela equação 4.13, fará com que os registradores permaneçam com seus valores inalterados, sendo

apenas alterada a amplitude do elemento procurado, passando de $\frac{1}{\sqrt{N}}$ para $-\frac{1}{\sqrt{N}}$ (eq. 4.14). Aqui é observado claramente o paralelismo quântico, já que a função f é aplicada uma única vez, atuando em todos os estados simultaneamente.

$$U_f(|\psi\rangle|-\rangle) = \left(\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} (-1)^{f(i)} |i\rangle \right) |-\rangle \quad (4.14)$$

Após a aplicação do operador U_f , o elemento procurado é identificado no nível quântico, já que é o único com amplitude negativa. Porém essa informação não está acessível classicamente, já que todas as probabilidades dos estados quânticos são iguais (eq. 4.15).

$$\left| \frac{1}{\sqrt{N}} \right|^2 = \left| -\frac{1}{\sqrt{N}} \right|^2 \quad (4.15)$$

O estado $|\psi_1\rangle$, resultante da aplicação do operador U_f ao estado inicial do primeiro registrador $|\psi\rangle$, será dado por

$$|\psi_1\rangle = \frac{1}{\sqrt{N}} \sum_{i=0, i \neq w}^{N-1} |i\rangle - \frac{1}{\sqrt{N}} |w\rangle, \quad (4.16)$$

ou seja,

$$|\psi_1\rangle = |\psi\rangle - \frac{2}{\sqrt{N}} |w\rangle \quad (4.17)$$

Para que uma operação de medição possa retornar a informação desejada é preciso aumentar a amplitude do elemento procurado, causando, conseqüentemente, a diminuição da amplitude dos demais elementos. Esse aumento de amplitude é conseguido aplicando-se ao estado intermediário $|\psi_1\rangle$ (eq 4.17) o operador

$$2|\psi\rangle\langle\psi| - I, \quad (4.18)$$

onde I é o operador identidade.

O estado $|\psi_2\rangle$ resultante será:

$$\begin{aligned} |\psi_2\rangle &= (2|\psi\rangle\langle\psi| - I)|\psi_1\rangle \\ &= (2|\psi\rangle\langle\psi| - I) \left(|\psi\rangle - \frac{2}{\sqrt{N}} |w\rangle \right) \\ &= (2\langle\psi|\psi\rangle)|\psi\rangle - \left(\frac{4}{\sqrt{N}} \langle\psi|w\rangle \right) |\psi\rangle - |\psi\rangle + \frac{2}{\sqrt{N}} |w\rangle \end{aligned} \quad (4.19)$$

Sabendo que $\langle \psi | w \rangle = \frac{1}{\sqrt{N}}$ e usando esse resultado na equação 4.19, $|\psi_2\rangle$ será dado por:

$$|\psi_2\rangle = \frac{N-4}{N}|\psi\rangle + \frac{2}{\sqrt{N}}, \quad (4.20)$$

sendo a amplitude do estado $|w\rangle$ dada por

$$|w\rangle = \left(\frac{N-4}{N}\right) \times \left(\frac{1}{\sqrt{N}}\right) + \left(\frac{2}{\sqrt{N}}\right) = \left(\frac{3N-4}{N\sqrt{N}}\right). \quad (4.21)$$

A amplitude de cada um dos demais estados $|i\rangle$ será:

$$|i\rangle = \left(\frac{N-4}{N}\right) \times \left(\frac{1}{\sqrt{N}}\right) = \left(\frac{N-4}{N\sqrt{N}}\right). \quad (4.22)$$

O operador de Grover G é formado pela composição do operador U_f e do operador da equação 4.18, como segue:

$$G = ((2|\psi\rangle\langle\psi| - I) \otimes I) U_f \quad (4.23)$$

Utilizando a representação geométrica, a atuação do operador de Grover pode ser analisada, inicialmente, sob o foco de cada um dos seus componentes isoladamente.

A atuação de U_f pode ser representada como uma reflexão do estado inicial $|\psi\rangle$ sobre o subespaço $|u\rangle$, ortogonal ao subespaço $|w\rangle$, da solução procurada, como indicado na figura 4.2.

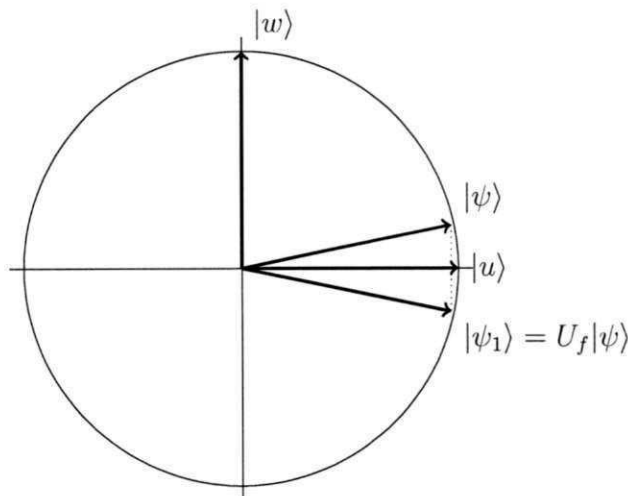


Figura 4.2: Atuação de U_f sobre o estado $|\psi\rangle$.

A representação do operador $2|\psi\rangle\langle\psi| - I$ pode também ser esquematizada como mostrado na figura 4.3, onde o estado $|\psi_2\rangle$ é obtido fazendo-se uma reflexão do estado $|\psi_1\rangle$ sob o estado inicial $|\psi\rangle$.

A atuação de um operador de Grover G (eq 4.23), de acordo com resultados demonstrados em [39] e [3], leva o sistema quântico do estado inicial $|\psi\rangle$ para o estado intermediário $|\psi_2\rangle$, que é rotacionado em θrad em relação ao estado inicial, em direção ao subespaço $|w\rangle$ da solução procurada.

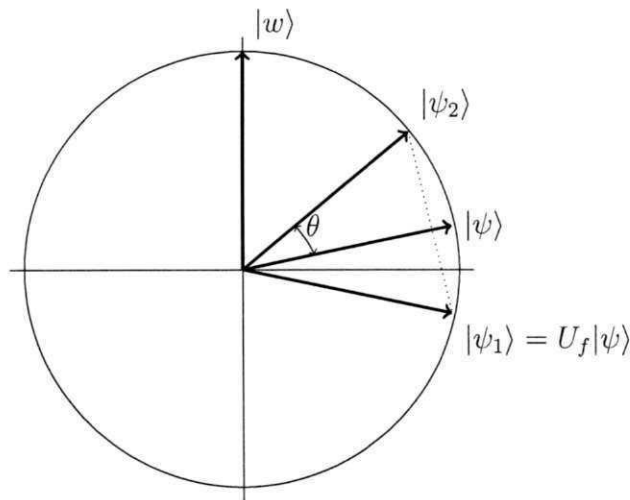


Figura 4.3: Reflexão de $|\psi_1\rangle$.

Como visto na figura 4.3, um operador de Grover rotaciona o estado quântico em θrad . Porém a aplicação desse operador deve se restringir a uma quantidade k de vezes que aproxime o estado da solução procurada. Como demonstrado em [39], [3], [36] e outros, essa quantidade não deve ser superior a $\frac{\pi}{4}\sqrt{N}$, quando houver apenas uma solução de busca, e não superior a $\frac{\pi}{4}\sqrt{\frac{N}{M}}$, quando houver M soluções no espaço de busca.

Capítulo 5

Simulação e Emulação de Circuitos

Quânticos

A simulação é um processo de criação e utilização de modelos de um sistema real ou hipotético onde, através da experimentação, é procurado entender o comportamento desse sistema modelado [51].

Segundo Savory e Mackulak, o processo de simulação (figura 5.1) deve envolver os seguintes passos: formulação, criação, codificação e execução do modelo de simulação; resumo dos resultados e análise e interpretação da saída.

A simulação pode ser utilizada quando se deseja estudar o comportamento de uma característica específica de um sistema complexo; quando é inviável fazer experimentos no sistema real; no projeto de sistemas inexistentes, através da criação de protótipos; dentre outras utilidades.

Enquanto que simulação se utiliza basicamente de ferramentas de software para a construção e execução dos modelos propostos, a emulação se utiliza de ferramentas de hardware e/ou software para o tratamento desses modelos.

Para que os processos de simulação ou emulação tenham êxito é necessário possuir conhecimentos sobre a metodologia de simulação/emulação; saber formular bem o problema; escolher as ferramentas adequadas para a confecção do simulador/emulador; estabelecer métricas de validade do modelo; e dominar os procedimentos de captura, análise e interpretação dos resultados.

Além dos conhecimentos listados acima é preciso estar ciente que o modelo construído

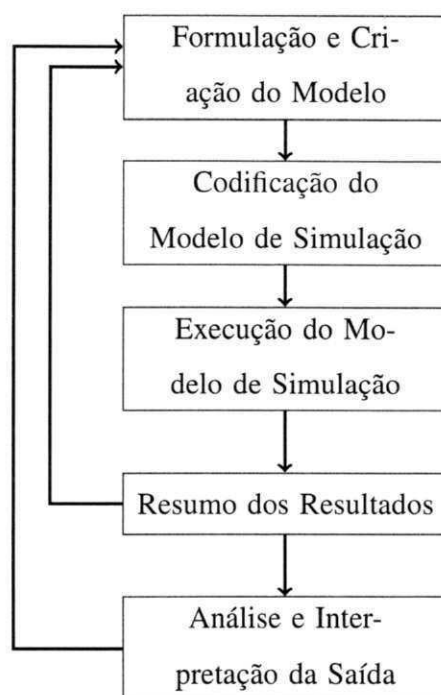


Figura 5.1: Etapas do processo de simulação.

é uma abstração ao sistema real, e como tal, possui limitações. É preciso identificar se as variáveis modeladas possuem comportamento estocástico ou determinístico.

Se as variáveis são estocásticas o resultado será tão bom quanto forem os dados de entrada e o tamanho da amostra.

Se as variáveis forem determinísticas o resultado só evidenciará as respostas para os casos testados, não sendo viável, na maioria dos casos, a cobertura total dos casos de testes.

E, independentemente da modelagem dos dados, é preciso identificar quantas execuções dos experimentos deverão ser efetuadas para que se possa atingir valores estatísticos confiáveis.

Como o computador quântico ainda não foi construído, a confecção de simuladores e emuladores tem sido adotada com intenções de verificar a aplicabilidade dos algoritmos quânticos; de auxiliar no ensino da computação quântica; como ferramenta para possibilitar a proposição de novos algoritmos; e como laboratório para a proposição de tecnologias para a construção do computador em si.

De modo geral, os simuladores e emuladores quânticos se baseiam no modelo de circuitos quânticos, que é uma poderosa e eficiente linguagem de descrição de algoritmos [36].

Com a intenção de auxiliar na compreensão e proposição de novos algoritmos diversos simuladores¹ e emuladores quânticos têm sido propostos.

Nos simuladores há uma preocupação com a facilidade de uso. São fornecidas interfaces que permitem que um circuito possa rapidamente ser construído e testado. Os q-bits de entrada podem ser editados para que os circuitos executem com diferentes dados. Podem ser incluídos novos operadores e alterados os existentes. Tudo isso para que os circuitos possam ser melhor estudados.

Apesar dos simuladores permitirem uma abstração ao modelo matemático, internamente é esse modelo quem dá suporte à execução dos circuitos. E, geralmente, tanto os estados quânticos como as operações sobre esses estados são representados por matrizes.

Exemplificando, para um sistema de n q-bits os estados quânticos são representados por uma matriz coluna² de ordem $(2^n \times 1)$ e cada operador será uma matriz quadrada de ordem $(2^n \times 2^n)$, onde cada elemento da matriz é um número complexo c (como definido no Cap. 2).

Considerando a existência de m operadores diferentes, *qtde_variaveis* (eq. 5.1) representa a quantidade de variáveis de memória (do tipo real) necessárias para o armazenamento de todo o circuito [31].

$$qtde_variaveis = (2^n \times 2) \times ((2^n \times m) + 1) \quad (5.1)$$

Cada aplicação de um operador a um circuito se dá pela multiplicação da matriz coluna (de estados) pela matriz quadrada (do operador), resultando em uma outra matriz coluna (novo estado), sendo necessárias *qtde_mult_oper* (eq. 5.2) multiplicações e *qtde_adic_oper* (eq. 5.3) adições de números complexos para a sua execução. Dessa forma, para um circuito que tenha p aplicações de operadores, serão necessárias *qtde_mult_tot* (eq. 5.4) multiplicações e *qtde_adic_tot* (eq. 5.5) adições [31].

$$qtde_mult_oper = 2^n \times 2^n \quad (5.2)$$

¹Uma extensa lista de simuladores desenvolvidos pode ser encontrada em http://www.quantiki.org/wiki/List_of_QC_simulators.

²Também chamada de registrador.

$$qtde_adic_oper = 2^n \times (2^n - 1) \quad (5.3)$$

$$qtde_mult_tot = (2^n \times 2^n \times p) \quad (5.4)$$

$$qtde_adic_tot = 2^n \times (2^n - 1) \times p \quad (5.5)$$

Dos dados acima, é possível ser observado que há um grande consumo de recursos computacionais (memória e processamento) quando da simulação de circuitos quânticos, tornando ineficiente, até mesmo impossível, a simulação de circuitos que necessitem manipular uma maior quantidade de q-bits. Além do mais, os computadores clássicos não são capazes, de forma eficiente, de simular o paralelismo presente nos algoritmos quânticos [24].

Para solucionar, ou pelo menos amenizar esse problema, diversas técnicas têm sido propostas. Citando alguns exemplos, Viamontes et al. [56] propuseram a utilização de estruturas de dados baseados em grafos que permitem, segundo os autores, a compressão dos dados, facilitando as operações de álgebra linear. Uma estrutura hierárquica, similar a uma árvore com zero ou mais níveis de ponteiros e um único nível contendo os valores de estados, foi usada para representar o vetor de estados [37]. Samad et al. [50] propuseram a utilização de uma única matriz representativa de todo o circuito, sendo disponibilizada, em memória, uma coluna de cada vez, que era multiplicada ao vetor de estados, gerando um resultado intermediário, que era somado ao resultado da multiplicação da próxima coluna da matriz com o vetor de estados, até que fosse obtido o vetor de estados resultante.

Independentemente da forma de armazenamento e execução dos circuitos quânticos, os simuladores podem ser classificados em dois grandes grupos: os simuladores universais e os não-universais.

Os simuladores universais, como o nome sugere, se propõem a manipular quaisquer algoritmos quânticos, enquanto que os não-universais lidam com algoritmos específicos.

5.1 Simuladores Universais

A grande maioria dos simuladores universais se utiliza do modelo de circuitos quânticos para a construção dos algoritmos a manipular, implementando interfaces que possibilitem o

desenho do circuito quântico de forma fácil e intuitiva. Eles permitem a execução do circuito em um único passo ou a execução passo-a-passo, possibilitando que o usuário verifique a evolução do sistema quântico em cada execução de uma porta lógica.

Com o intuito de manipular diversos algoritmos quânticos, esses simuladores implementam a maioria das portas lógicas (principalmente as portas universais), sendo que alguns deles [6] permitem que o usuário crie suas portas lógicas particulares.

Existem simuladores que são disponibilizados em uma versão on-line [21] [45], mas a maioria precisa de uma instalação prévia para sua execução.

Os simuladores pertencentes a essa classe geralmente manipulam circuitos quânticos que possuam poucas dezenas de q-bits ([6], [21], [46] e [20]), chegando, nos melhores casos, a manipular 40 q-bits [47].

Alternativamente, existem simuladores que não usam o modelo de circuitos quânticos, podendo fazer uso de pseudo-linguagens quânticas para expressar os algoritmos a manipular ([41], [43], [40] e [42]). Geralmente os resultados são mostrados utilizando a álgebra simbólica, podendo os algoritmos ser solucionados simbólica ou numericamente ([44]).

5.2 Simuladores Não-universais

Como dito acima, os simuladores não-universais procuram fornecer maior eficiência e usabilidade com relação aos simuladores universais, principalmente no que se refere à quantidade de q-bits a manipular e à performance. Diversas são as soluções que podem chegar à quantidade de centenas de q-bits.

O simulador CHP (*CNOT-Hadamard-Phase*) [8], por exemplo, é um simulador que implementa circuitos estabilizadores, fazendo uso apenas das portas lógicas C-Not, Hadamard, fase e medição de 1 q-bit, podendo manipular, segundo os autores, centenas de q-bits facilmente.

5.3 Emuladores

Os emuladores se utilizam de recursos de hardware nos modelos quânticos. A ideia central da emulação é a utilização de uma plataforma de hardware que não o computador clássico.

Diversas têm sido as técnicas de emulação: ressonância magnética, armadilha de íons, hardware dedicado (placas FPGA³), dentre outras.

A vantagem na utilização dessas diversas tecnologias reside no fato de poderem ser implementados os fundamentos da mecânica quântica (superposição, emaranhamento, descoerência, ...), não possíveis de serem implementados em computadores clássicos.

Sarthour et al. [38] realizaram experimentos com ressonância magnética nuclear, implementado, dentre outros, os algoritmos de Grover e de Shor, manipulando 2 e 7 q-bits respectivamente. Em estudos mais recentes, Grinolds et al.[16] e Goto [15] demonstram que será possível um ganho de escalabilidade por conta de um maior controle quântico de spins individuais em sistemas de matéria condensada.

Muitos pesquisadores acreditam que a armadilha de íons é uma das técnicas de maior destaque na manipulação de algoritmos quânticos [27] [19]. Essa é uma técnica que permite uma maior escalabilidade, sendo publicados resultados onde é possível a manipulação de algoritmos com até 300 q-bits [52].

Em sua primeira tentativa de emular um computador quântico utilizando FPGA, Fujishima [13] observou que determinadas classes de algoritmos tinham seus estados quânticos com valores 0 ou $1/m$ (sendo m a quantidade de estados diferentes de zero) durante toda a execução do circuito. Assim pensando, ao invés de armazenar amplitudes, ele optou por armazenar os estados com amplitudes diferentes de zero. O autor propôs a criação de elementos de processamento (que possuem um conjunto de registradores, uma unidade lógica e uma memória) que são conectados em paralelo, em uma rede. Para essa solução particular foi possível manipular circuitos com até 75 q-bits, sendo usados como exemplo os algoritmos de Grover e de satisfatibilidade (SAT).

Dando seguimento em sua solução de emulação, Fujishima et al. [14] incluíram, na estrutura anteriormente proposta, um bit de controle para sinalizar a ocorrência de descoerência, que nesse caso em particular, era implementada como uma operação NOT no q-bit marcado. Como exemplo foi implementado um circuito com 16 q-bits para o problema da satisfatibilidade, se mostrando 275 vezes mais rápido que a simulação por software.

Khalid et al. [24] propuseram o uso da tecnologia FPGA para modelar circuitos quânticos. Foi criada uma biblioteca de elementos básicos dos circuitos quânticos, usada para a

³Field-Programmable Gate Arrays

construção dos algoritmos, requerendo o menor esforço do desenvolvedor na construção de circuitos mais complexos. As operações dos elementos são executadas por uma sequência de comandos ao invés de ser utilizada a representação matricial.

Foram emulados os algoritmos da Transformada Quântica de Fourier (QFT) e de Grover, com 3 q-bits (dois para o espaço de busca e um para controle). Para validação de seu trabalho os tempos de execução dos algoritmos emulados foram comparados com os tempos de execução dos mesmos algoritmos no simulador Libquantum, apresentando, sua solução, ganhos significativos. Na conclusão do seu trabalho, os autores afirmaram ser possível a manipulação de circuitos mais complexos e com maior escalabilidade.

Já em sua dissertação de mestrado, Khalid [23] demonstrou seus resultados com circuitos da QFT e de Grover com 3, 4 e 5 qubits, comparando seus resultados com o simulador QuIDD.

Em 2008, Aminian et al. [4] também propuseram a emulação de circuitos quânticos com FPGA. Sua solução se baseou na solução de Khalid, tendo como diferencial a inclusão de bits de controle visando a economia de recursos e de tempo de execução. Cada q-bit, anteriormente apenas representado como um par de números complexos, agora possui bits de controle (cuja quantidade varia de acordo com a porta lógica implementada).

Como primeiro exemplo da eficiência do método proposto, foi implementado o circuito da QFT com 3 q-bits que, de acordo com os dados apresentados, atingiu uma frequência de clock maior que a solução anterior e foi mais eficiente que o simulador Libquantum.

Ainda como exemplo foram implementados alguns circuitos aritméticos booleanos, com no máximo 8 q-bits (usando portas NOT, CNOT e CCNOT), onde foi demonstrada uma economia de recursos lógicos utilizados e também uma significativa redução do tempo de execução dos mesmos, ainda quando comparado com a solução proposta por Khalid.

Monteiro et al. [30] também propuseram a criação de uma biblioteca de componentes básicos de hardware que são compostos de portas lógicas, flip-flops e registradores, que permitem a construção de elementos mais complexos. Além desses componentes, foi criada a biblioteca qExVHDL, que contém métodos para descrição de estados, principais operadores e um conjunto de portas universais. Foram utilizados como estudo de caso o interferômetro Mach-Zehnder e o algoritmo de Deutsch.

permitindo a criação de blocos de lógica que possibilitam a execução de funções complexas (figura 6.2) [35].

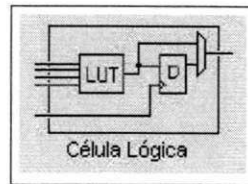


Figura 6.1: Célula Lógica.

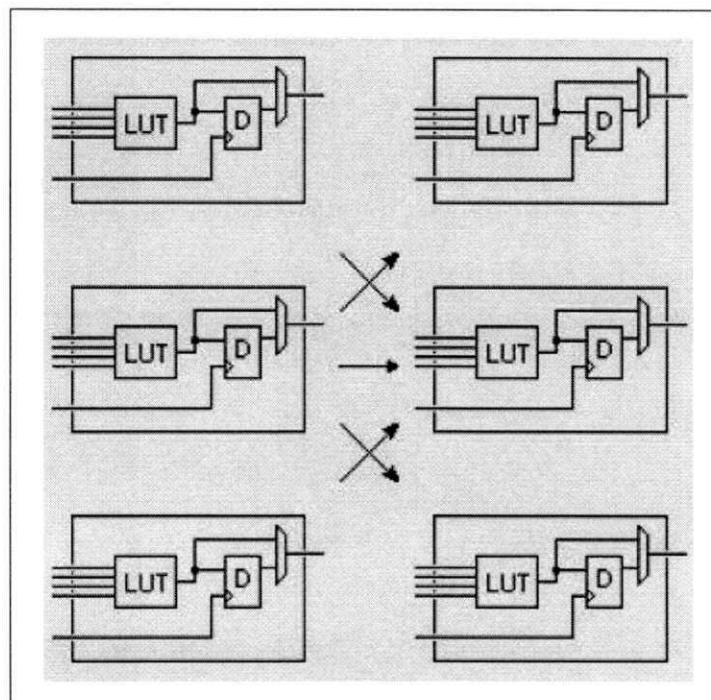


Figura 6.2: Interconexão entre células lógicas.

Circundando o dispositivo FPGA estão os blocos de entrada e saída que podem ser conectados aos blocos construídos. Esses blocos fazem a ligação entre os blocos de lógica e os pinos da placa FPGA (figura 6.3). Visando dar maior agilidade na construção de funções do tipo contadores e somadores, as células lógicas ainda possuem linhas dedicadas (carry chains) que ligam, mais eficientemente células vizinhas (figura 6.4) [35].

Capítulo 6

Recursos para Emulação de Circuitos Quânticos

Como visto no capítulo anterior, pesquisadores têm procurado fazer uso das placas FPGA na modelagem, construção e execução de circuitos quânticos. Nas seções seguintes são descritos os recursos necessários para emulação de circuitos quânticos.

6.1 Placas FPGA

FPGAs são placas lógicas digitais programáveis que, como o próprio nome indica, são placas compostas de uma grande quantidade de elementos programáveis, que podem simular o comportamento de um circuito [1].

O primeiro dispositivo FPGA foi lançado em 1985 pela empresa Xilinx Inc. A FPGA é composta basicamente por blocos de entrada e saída, elementos lógicos configuráveis e chaves de interconexão [57].

Os elementos lógicos são componentes padrões que podem ser livremente conectados pelas chaves de interconexão, através da utilização de uma matriz de trilhas condutoras e chaves (switches) programáveis. O vetor de blocos lógicos e a matriz de interconexão podem ser programados pelo usuário para a construção de circuitos integrados complexos [7].

A unidade básica de uma FPGA é a célula lógica (figura 6.1), que executa pequenas funções lógicas, sendo capaz de armazenar um único valor lógico (zero ou um) [57]. Cada célula lógica pode ser conectada a outras células através de fios que circundam essas células,

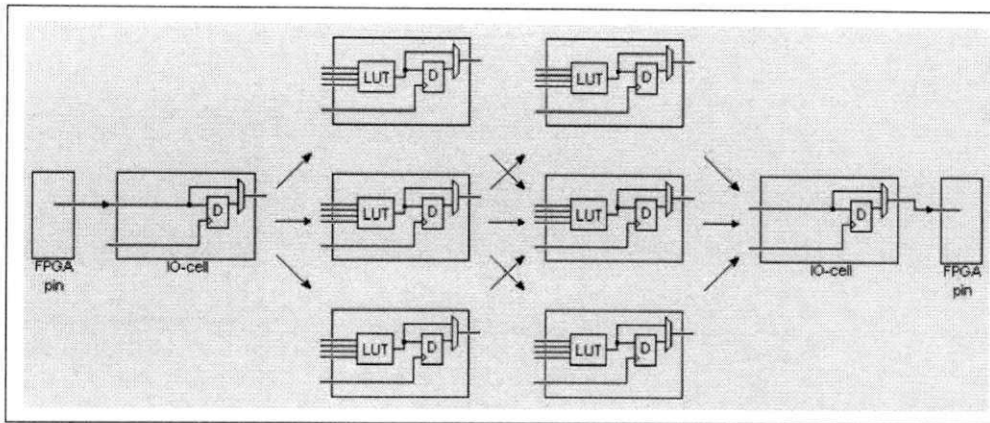


Figura 6.3: Esquema de construção de bloco lógico.

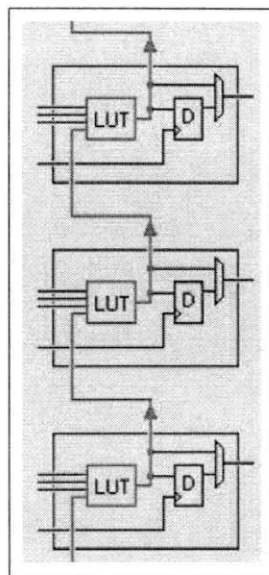


Figura 6.4: Ligação entre células lógicas vizinhas.

6.2 Ferramentas de Desenvolvimento

Geralmente o dispositivo FPGA é usado como protótipo para a criação de circuitos integrados onde, com o auxílio de poderosas ferramentas de desenvolvimento e modelagem, é feita a programação desse dispositivo. Como o dispositivo possui uma memória estática interna, ele pode ser reprogramado quantas vezes se queira, podendo o mesmo ser programado de forma definitiva caso não se queira produzir o circuito integrado em larga escala [7].

A utilização das ferramentas de desenvolvimento fornece um nível de abstração tal que

permite que o desenvolvedor se preocupe menos em como os circuitos serão configurados, se dedicando mais em quais funções o mesmo irá desempenhar. Dentre essas ferramentas, a mais utilizada é o Quartus II[®], produzido pela empresa Altera.

O Quartus II[®] é uma ferramenta que possibilita que o desenvolvedor compile seus projetos, execute análise de tempo, examine relatórios de compilação, simule o projeto com diferentes dados de entrada e, por fim, configure o dispositivo FPGA [2]. O mesmo está disponível em duas versões: Web Edition, uma versão gratuita, que executa um número limitado de funções e permite a configuração de um número restrito de dispositivos FPGA; e a versão completa, que depende da aquisição de uma licença para seu uso.

O pacote pode ser executado em diversos sistemas operacionais (Windows 7 Vista e XP, Red Hat Enterprise Linux 4 e 5, SUSE Linux Enterprise 10 e 11 e CentOS 4 e 5).

6.3 Linguagens de Descrição de Hardware

Como dito acima, a utilização de ferramentas de desenvolvimento permite uma abstração no que se refere à construção dos circuitos integrados. Essa abstração se deve, em parte, à possibilidade de uso de linguagens HDL¹, que são linguagens formais para descrição de hardware [12].

As linguagens HDL contemplam o comportamento espacial e temporal dos sistemas eletrônicos, incluindo estruturas sintáticas e semânticas que possibilitam a implementação de concorrência e tratamento de tempo, atributos primordiais dos circuitos integrados [58].

Dentre as linguagens de descrição de hardware, as mais utilizadas são a VHDL² e a SystemVerilog.

A linguagem VHDL foi desenvolvida, em 1980, sob o comando do Departamento de Defesa dos Estados Unidos, se propondo a documentar o projeto VHSIC. Ela facilita o projeto de um circuito, permitindo que o mesmo seja descrito em forma de algoritmo, contemplando ainda as fases de documentação, síntese, simulação, teste, verificação formal e ainda compilação da solução. Sua sintaxe se assemelha à das linguagens Pascal e Ada[61]. Atualmente os direitos autorais da linguagem estão em poder da IEEE, que lançou, em 2008, sua versão

¹Hardware Description Language

²VHSIC (Very High Speed Integrated Circuits) Hardware Description Language.

mais atual (IEEE 1076-2008).

A linguagem VHDL permite múltiplos níveis de hierarquia, podendo ser definidos projetos estruturais e comportamentais; permite a execução simultânea de comandos através de declarações de comandos concorrentes e não é *case-sensitive*.

Entre os anos de 1983 e 1984 foi desenvolvida, por Phil Moorby e Prabhu Goel, a linguagem Verilog, comprada pela Cadence Design Systems em 1990 e transformada em linguagem de domínio público em 1995, sob o controle da organização Accellera. Essa linguagem pode ser usada nas tarefas de descrição, simulação e síntese de circuitos digitais. A sua versão atual é o padrão IEEE 1364-2005 (Verilog 2005).

A linguagem SystemVerilog (IEEE 1800) teve como origem a linguagem Verilog (figura 6.5³). Foi inicialmente desenvolvida pela empresa Accellera [55]. É uma linguagem que tem sido adotada por centenas de empresas envolvidas com projetos de semicondutores. Ela fornece um alto nível de abstração para as fases de projeto e verificação, sendo essa última fase drasticamente melhorada em relação às demais linguagens de descrição de hardware [54].

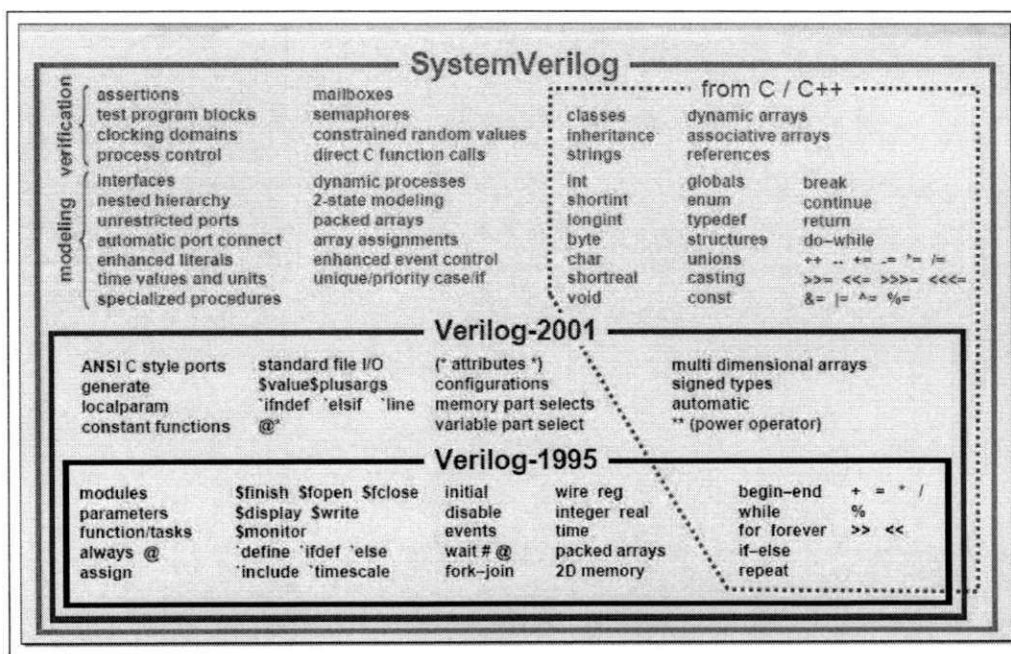


Figura 6.5: Linguagem SystemVerilog.

³Fonte: http://www.systemverilog.org/pdf/5a_V2K_SV_compatibility.pdf

SystemVerilog permite a programação modular, possibilitando que novos módulos sejam facilmente incorporados ao código já existente, mesmo que esse tenha sido escrito em Verilog, minimizando os riscos e custos da utilização de uma nova linguagem de verificação [54].

SystemVerilog necessita de um número menor de linhas de código para produzir o mesmo efeito que um código em VHDL, reduzindo o trabalho, diminuindo a possibilidade de erros e tornando o código mais fácil de entender [28].

SystemVerilog possui sintaxe semelhante a linguagem C e permite que códigos escritos em C, C++, SystemC e Verilog sejam usados conjuntamente, através do uso da interface DPI⁴[54].

SystemVerilog possui um grande suporte a testes. Sendo orientação a objetos um grande atrativo deste suporte.

6.4 O Simulador Zeno

O simulador Zeno[6] foi utilizado com base para a construção dos circuitos a serem testados e para a verificação dos resultados obtidos na solução proposta.

Esse simulador, implementado em Java, é um software livre, sob licença GPL. A sua primeira versão datou do ano de 2004, tendo como objetivo principal permitir a modelagem e simulação de algoritmos quânticos usando o modelo de circuitos quânticos, de forma fácil e intuitiva, usando elementos gráficos. O mesmo traz em sua biblioteca um conjunto universal de portas, permitindo a definição de outras portas lógicas. É permitida a simulação envolvendo estados puros ou mistos.

Em 2007 foi disponibilizada uma nova versão[5] onde foi incluído o módulo CAS⁵ com o objetivo de permitir a criação e manipulação de descrições matemáticas. Foram feitas ainda algumas modificações no código, que resultaram numa melhoria de performance em relação à versão anterior, conforme verificado nos testes efetuados.

⁴Direct Programming Interface

⁵Computer Algebra System

Capítulo 7

Solução Proposta

Como dito no capítulo 5, diversos foram os trabalhos que abordaram a emulação de circuitos quânticos utilizando as placas FPGA, sendo utilizadas diversas abordagens em suas implementações, visando principalmente uma melhor performance e também uma economia na quantidade de recursos físicos utilizados.

Todos os trabalhos se mostraram eficientes quando aplicados os métodos propostos, quando comparados com soluções de simulação em software ou mesmo quando comparados com outras soluções emuladas.

Porém, como pôde ser observado, foram emulados circuitos com pequena quantidade de q-bits, e, na conclusão de todos os trabalhos, era afirmada a possibilidade de serem manipulados circuitos mais significativos, ou seja, com uma maior quantidade de q-bits, sendo essa tarefa deixada como sugestão de trabalhos futuros.

Com o objetivo melhor utilizar os recursos de FPAG, foi proposto o presente trabalho, com a intenção de aumentar o número de q-bits dos circuitos emulados principalmente por meio da utilização de estruturas que possibilitem que os mesmos sejam implementados consumindo a menor quantidade de células lógicas de FPGA possível.

7.1 Decisões de Implementação

Para a condução do trabalho foram estabelecidas as seguintes condições:

1. Teria que ser utilizada, como recurso de hardware, uma placa FPGA.

2. A solução deveria ser capaz de emular qualquer circuito quântico. Ou seja, teria que ser um emulador universal.
3. As portas quânticas teriam que ser aplicadas individualmente, permitindo a execução passo-a-passo do circuito.
4. As amplitudes dos estados quânticos deveriam ser números complexos, evitando perda de informação.

7.2 Recursos Computacionais

Para a execução desse trabalho, primeiramente, foi escolhida a placa de FPGA a ser utilizada.

Através de uma parceria entre o IQuanta/UFCG e o LAD/UFCG, foi disponibilizada uma placa DE2-70 (figura 7.1) Cyclone[®] II 2C70 da empresa Altera.

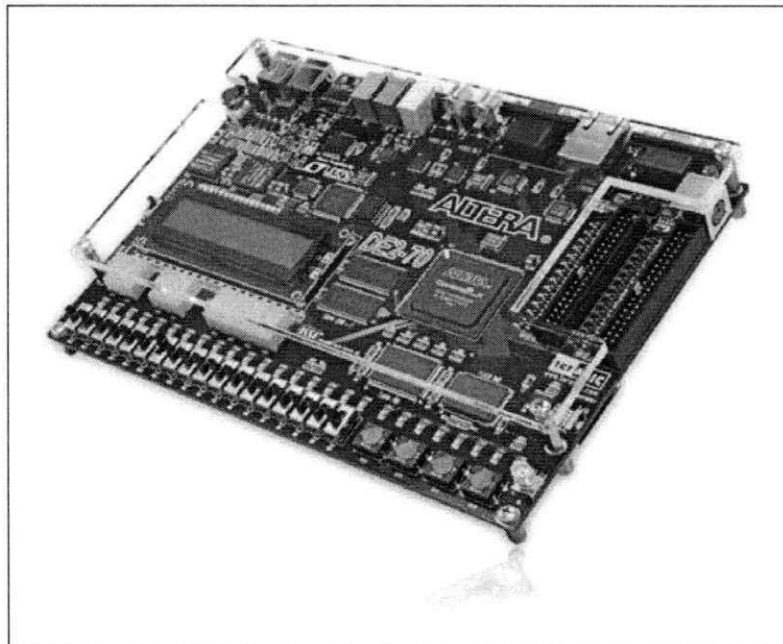


Figura 7.1: Placa DE2-70.

Dentre outras configurações, a placa DE2-70 possui:

- Porta USB-Blaster - para a configuração da placa
- Entrada/Saída de microfone e vídeo

- RS232 - para a troca serial de dados binários
- Portas de infravermelho, teclado e mouse PS/2, USB 2.0 A e B
- Memória (MB): 64 SDRAM, 2 SSRAM e 8 Flash
- Slot para cartão SD
- 8 visores de sete segmentos
- Um visor de LCD 16×2
- 18 interruptores, 18 luzes vermelhas, 8 luzes verdes e 4 botões *pushbottom*
- Dois *clocks* internos de 50 e 27 MHz e uma entrada para *clock* externo
- Possui 68.416 elementos lógicos

Como ferramenta de desenvolvimento foi utilizado o Quartus II®. Como a versão Web Edition não contempla todos os recursos da ferramenta, foi utilizada a versão completa¹, já que o LAD/UFCG possui licença de uso da mesma.

A linguagem de descrição de hardware adotada foi a SystemVerilog.

Foi utilizado um computador com processador Intel(R) Core(TM)2 de 3.00GHz, com sistema operacional CentOS versão 5.7 de 32 bits com 2 GB de memória RAM.

Para a verificação da solução foram escolhidos uma série de algoritmos quânticos aritméticos, a Transformada de Fourier Quântica e o algoritmo de Grover.

7.3 A Solução

As seções seguintes mostram como os componentes do emulador foram implementados.

7.3.1 Q-bits

Cada q-bit está representado por um número complexo, armazenando sua amplitude. Internamente ele é representado por dois números reais. Um para a parte real e outro para a parte

¹Full Version 10.1 - Service Pack 1.

imaginária desse complexo. Como esses números reais nunca são maiores que 1 e nem menores que -1 , foi adotada a representação de ponto fixo, sendo o primeiro bit a representação do inteiro (0 ou 1) e os demais, a parte fracionária (figura 7.2). Por conta do tipo de dado escolhido, a representação do sinal é codificada em complemento de dois, não havendo a necessidade de reservar um bit para tal informação.

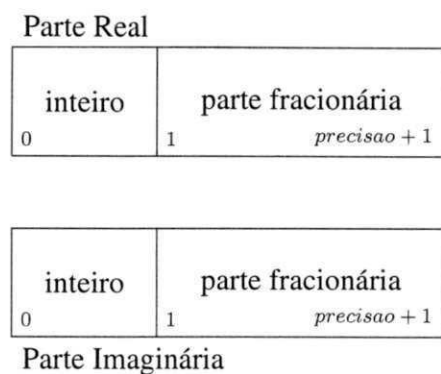


Figura 7.2: Representação interna das amplitudes.

A precisão dos números reais (parte fracionária) é um argumento da solução que pode ser alterado em tempo de compilação (código 7.1).

Código Fonte 7.1: Precisão

```
parameter int precisao
```

A quantidade de q-bits também é uma variável que pode ser atribuída em tempo de compilação, que recebe um valor inteiro de acordo com o circuito a emular (código 7.2).

Código Fonte 7.2: Quantidade de q-bits

```
parameter int qtde_qbits
```

7.3.2 Registrador Quântico

Todas as operações quânticas são realizadas a nível de registrador quântico, que é uma estrutura de vetor, cuja dimensão está diretamente relacionada ao número de q-bits (2^{qtde_q-bits}).

Foram implementados dois vetores: um para armazenar a parte real do número complexo e outro para armazenar a sua parte imaginária. As operações em um q-bit são efetuadas nas

partes real e imaginária da amplitude desse q-bit.

Como o número de q-bits e a precisão são variáveis, os vetores que armazenam as amplitudes variam em função desses números. Os vetores de amplitudes são representados como no trecho do código 7.3.

Código Fonte 7.3: Amplitudes

```
logic signed [precisao+1:0] amp_real [0:(1<<qtde_qbits) - 1];  
logic signed [precisao+1:0] amp_imag [0:(1<<qtde_qbits) - 1];
```

Para endereçar esses vetores de amplitudes são usadas duas variáveis: *origem* e *destino*, que também são função da quantidade de q-bits (código 7.4).

Código Fonte 7.4: Apontadores para os vetores de amplitudes

```
bit [qtde_qbits:0] origem, destino;
```

Para acessar cada q-bit particular dentro de um estado quântico é utilizada a variável *posicao*. Como no Quartus os vetores começam com índice 0 e na linguagem de circuitos os q-bits são numerados a partir de 1, para fazer referência a um determinado q-bit, o parâmetro posição, internamente, é decrescido de uma unidade, como listado nos exemplos do próximo capítulo.

7.3.3 Portas Lógicas

Os operadores lógicos, nessa solução, são chamados de portas lógicas.

Cada porta lógica é implementada para representar a atuação de um operador no registrador quântico. Como a intenção é implementação do paralelismo quântico, essa atuação é executada em um único passo², independentemente da quantidade de estados quânticos afetados pelo operador.

A solução não contempla todos os operadores quânticos possíveis. Foram implementados apenas os operadores necessários à execução dos algoritmos testados.

²Exceto para o operador de Hadamard, descrito na seção seguinte.

Operador de Hadamard

Para a implementação do operador de Hadamard, foi decidido que o mesmo seria executado em dois passos. Num primeiro momento todas as amplitudes, diferentes de zero, são multiplicadas pelo inverso da raiz quadrada de dois, como descrito no código 7.5.

Código Fonte 7.5: Primeiro passo do operador de Hadamard

```
task Hadamard_1;
// primeiro passo hadamard
// multiplica todas as amplitudes por 1/sqrt(2)
for (origem = 0; origem < (1<<qtde_qbits); origem++)
  if ((amp_real[origem] != 0) || (amp_imag[origem] != 0))
    begin
      amp_real[origem] <= Multiplica(amp_real[origem], inv_raiz2);
      amp_imag[origem] <= Multiplica(amp_imag[origem], inv_raiz2);
    end;
endtask
```

A variável *inv_raiz2* contém o valor do inverso da raiz quadrada de dois, com a precisão passada como parâmetro. Essa variável é obtida da constante *max_inv_raiz2* que contém a mesma informação com uma precisão máxima de 32 bits (código 7.6).

Código Fonte 7.6: Inverso da raiz quadrada de dois

```
assign max_inv_raiz2 = 32'b00101101010000010011110011001101;
assign inv_raiz2 = max_inv_raiz2 >>> (30 - precisao);
```

Multiplica é uma função que retorna o valor da multiplicação de dois números passados como parâmetro, sendo o seu resultado ajustado ao tamanho da precisão escolhida (código 7.7).

Código Fonte 7.7: Função Multiplica

```
function logic signed [precisao+1:0] Multiplica
  (logic signed [precisao+1:0] a, b);
  temp = (a * b);
  temp = temp >>> (precisao - 1);
  if (temp[0]) // testa se penultimo bit igual a um, para arredondar
    if (temp > 0)
      temp = temp + 1;
```

```

    else
        temp = temp - 1;
    Multiplica = temp >>> 1;
endfunction

```

Para a conclusão do operador de Hadamard, é executado o segundo passo (código 7.8). O parâmetro de entrada *posicao* é necessário para que seja informado em qual q-bit está sendo aplicado o operador. Cada q-bit no circuito é identificado por um número inteiro, sendo a contagem feita de baixo para cima, iniciando em um.

Código Fonte 7.8: Segundo passo do operador de Hadamard

```

task Hadamard_2 (input int posicao);
// segundo passo hadamard
// soma as amplitudes ja multiplicadas de acordo com o qubit passado como
  parametro
  for (origem = 0; origem < (1<<qtde_qbits); origem++) begin
    destino = origem;
    destino[posicao - 1] = ~destino[posicao - 1];
    if (origem[posicao - 1]) begin // testa se esta aplicando hadamard a um
      q-bit 1
      amp_real[origem] <= amp_real[destino] - amp_real[origem]; //se q-
        bit 1
      amp_imag[origem] <= amp_imag[destino] - amp_imag[origem];
    end
  else begin
    amp_real[origem] <= amp_real[origem] + amp_real[destino]; //se q-
      bit 0
    amp_imag[origem] <= amp_imag[origem] + amp_imag[destino];
  end
endtask

```

Operador de Negação

O operador de negação é implementado pela porta lógica Not (código 7.9). Seu objetivo é fazer a negação de q-bits, através da troca dos valores das amplitudes dos q-bits envolvidos.

Código Fonte 7.9: Porta lógica Not

```

task Not (input int posicao);
    origem = 0;
    for (m = 1; m <= ((1<<qtde_qbits)/(1<<posicao)); m++)
    begin
        for (n = 1; n <= (1<<(posicao - 1)); n ++)
        begin
            destino = origem;
            destino[posicao - 1] = ~destino[posicao - 1];
            amp_real[origem] <= amp_real[destino];
            amp_real[destino] <= amp_real[origem];
            amp_imag[origem] <= amp_imag[destino];
            amp_imag[destino] <= amp_imag[origem];
            origem = origem + 1;
        end; // for n
        origem = origem + n - 1;
    end; // for m
endtask // Not

```

Aqui também é passada, como parâmetro, a posição, ou seja, o q-bit alvo do operador. Como a troca é feita aos pares (*origem* \rightleftharpoons *destino*), é preciso ter cuidado para que um q-bit reconhecido como destino em uma troca, não seja identificado como origem num momento seguinte, ocasionando a anulação da troca.

Essa restrição foi solucionada na lógica de programação, sem a necessidade de se criar bits de controle para identificar os q-bits já trocados.

Como particularidade das linguagens de descrição de hardware e do seu modo de execução, pode ser observado, no código acima (código 7.9), que não há a necessidade de ser definida, explicitamente, uma área de transferência quando se quer fazer a troca entre duas variáveis, como ocorre comumente nas outras linguagens de programação.

Operador de Negação Controlada

A porta lógica CNot (código 7.10) tem o mesmo objetivo da porta lógica Not, sendo que a negação de um q-bit, indicado pelo parâmetro *posicao*, é controlado pela informação contida no parâmetro *controle*, como descrito no capítulo 3.

```

task CNot (input int posicao , controle);
    origem = 0;
    for (m = 1; m <= ((1<<qtde_qbits)/(1<<posicao)); m++)
    begin
        for (n = 1; n <= (1<<(posicao-1)); n ++ )
        begin
            if (origem[controle - 1]) // testa se q-bit de controle igual a 1
            begin
                destino = origem;
                destino[posicao -1] = ~destino[posicao -1];
                amp_real[origem] <= amp_real[destino];
                amp_real[destino] <= amp_real[origem];
                amp_imag[origem] <= amp_imag[destino];
                amp_imag[destino] <= amp_imag[origem];
            end; // if controle
            origem = origem + 1;
        end; // for n
        origem = origem + n -1;
    end; // for m
endtask //CNot

```

A Porta Toffoli, ou CCNot (código 7.11), é uma porta de negação, que funciona analogamente à porta CNot, sendo acrescido mais um q-bit de controle.

Código Fonte 7.11: Porta Toffoli

```

1 task CCNot (input int posicao , controle1 , controle2);
2     origem = 0;
3     for (m = 1; m <= ((1<<qtde_qbits)/(1<<posicao)); m++)
4     begin
5         for (n = 1; n <= (1<<(posicao-1)); n ++ )
6         begin
7             if ((origem[controle1]) && (origem[controle2])) // testa se
8                 controles iguais a 1
9             begin
10                destino = origem;
11                destino[posicao -1] = ~destino[posicao -1];
12                amp_real[origem] <= amp_real[destino];
                amp_real[destino] <= amp_real[origem];

```



```

13         amp_imag[origem] <= amp_imag[destino];
14         amp_imag[destino] <= amp_imag[origem];
15     end; // if controle
16     origem = origem + 1;
17 end; // for n
18 origem = origem + n - 1;
19 end; // for m
20 endtask //CCNot

```

Para certos algoritmos, como o de Grover, é necessária a presença de um operador de negação com múltiplos controles. O mesmo pode ser implementado como uma sequência de portas Toffoli e um operador CNot, ou diretamente, como um único operador com todos os controles fornecidos. Para essa solução foi escolhida a segunda opção. Assim, a porta lógica CCNot_m (código 7.12) é usada quando se deseja fazer uma operação de negação com múltiplos controles, sendo que esses controles não são passados como parâmetro e sim inseridos diretamente no código (código 7.12, linhas 7 e 8), sendo alterado para cada situação particular, na fase de programação.

Código Fonte 7.12: Porta lógica CCNot_m

```

1 task CCNot_m (input int posicao);
2     origem = 0;
3     for (m = 1; m <= ((1<<qtde_qbits)/(1<<posicao)); m++)
4     begin
5         for (n = 1; n <= (1<<(posicao-1)); n++)
6         begin
7             if (origem[1] && origem[2] && origem[3] &&
8                 origem[4]) // testa se q-bits de controle iguais a 1
9                 begin
10                    destino = origem;
11                    destino[posicao-1] = ~destino[posicao-1];
12                    amp_real[origem] <= amp_real[destino];
13                    amp_real[destino] <= amp_real[origem];
14                    amp_imag[origem] <= amp_imag[destino];
15                    amp_imag[destino] <= amp_imag[origem];
16                end; // if controle
17            origem = origem + 1;

```

```

18     end; // for n
19     origem = origem + n - 1;
20     end; // for m
21 endtask //CCNot_m

```

Porta de Pauli Z

A porta lógica Z (código 7.13), que executa uma rotação no eixo Z da esfera de Bloch, é assim codificada:

Código Fonte 7.13: Porta lógica Z

```

task Porta_Z (input int posicao);
//synthesis loop_limit 8000
  for (origem = 0; origem < (1<<qtde_qbits); origem++)
    if (origem[posicao - 1]) begin
      amp_real[origem] <= (amp_real[origem] * -1);
      amp_imag[origem] <= (amp_imag[origem] * -1);
    end;
endtask; // Porta_Z

```

Analogamente à porta CCNot_m, foi criada a porta lógica Porta_Z_m (código 7.14), que executa o operador Z com múltiplos controles, também inseridos diretamente no código, para cada caso particular.

Código Fonte 7.14: Porta lógica Z

```

task Porta_Z (input int posicao);
//synthesis loop_limit 8000
  for (origem = 0; origem < (1<<qtde_qbits); origem++)
    if (origem[1] && origem[2] && origem[3] &&
        origem[4]) // testa se q-bits de controle iguais a 1
      if (origem[posicao - 1]) begin
        amp_real[origem] <= (amp_real[origem] * -1);
        amp_imag[origem] <= (amp_imag[origem] * -1);
      end;
endtask; // Porta_Z_m

```

Portas de Rotação

As portas de rotação, amplamente utilizadas no algoritmo da TFQ, foram implementadas usando o código 7.15. Já que no algoritmo da TFQ todas as portas de rotação aplicadas são portas controladas, as implementações dessas portas foram também implementadas em suas versões controladas.

Como parâmetros são passados o q-bit alvo (posição), o controle e os fatores de multiplicação das partes real e imaginária, resultantes do cálculo da expressão $\exp(2\pi i/2^k)$. Essa implementação faz uso da operação de multiplicação, já listada acima.

Código Fonte 7.15: Porta lógica Rotação

```
task Rotacao (input int posicao , controle , logic signed [precisao+1:0] a ,
             b);
// synthesis loop_limit 8000
for (origem = 0; origem < (1<<qtde_qbits); origem++)
  if (origem[posicao-1] && origem[controle])
    begin
      amp_real[origem] <= Multiplica(amp_real[origem], a) -
                          Multiplica(amp_imag[origem], b);
      amp_imag[origem] <= Multiplica(amp_real[origem], a) +
                          Multiplica(amp_imag[origem], b);
    end;
endtask // Rotacao
```

Porta S

A porta S, que é um caso particular das portas de rotação, foi implementada com o código 7.16.

Código Fonte 7.16: Porta S

```
task Porta_S (input int posicao , controle);
// synthesis loop_limit 8000
for (origem = 0; origem < (1<<qtde_qbits); origem++)
  if (origem[posicao-1] && origem[controle])
    begin
      amp_real[origem] <= (amp_imag[origem] * -1);
    end;
endtask
```

```
    amp_imag[origem] <= amp_real[origem];  
    end;  
endtask // Porta_S
```

7.3.4 Eficiência da Implementação

Uma solução de emulação deve estar focada em alguns requisitos para que possa ser comprovada a eficiência da mesma.

Um dos pontos mais enfatizado está relacionado ao tempo de execução do circuito emulado. Nessa solução proposta o tempo sempre estará em função da quantidade de portas lógicas executadas, visto que cada uma é executada em um único ciclo de clock da placa emuladora, com exceção da porta de Hadamard, que precisa de dois ciclos de clock para a sua execução.

Dessa forma, o tempo de execução dependerá exclusivamente do tamanho do circuito e da capacidade de processamento da placa FPGA. Assim, tempos melhores serão alcançados se placas com maior velocidade de processamento forem utilizadas.

Outra questão fundamental é a quantidade de espaço físico que é necessária para a implementação do circuito, visto que, como em todo recurso computacional, as placas FPGA possuem limitações de armazenamento. Soluções que consumam menos elementos ou células lógicas, sem perder em representatividade de informação e eficiência em relação ao tempo de execução, estarão em vantagem em relação às outras soluções.

A alternativa para a economia de células lógicas foi a decisão de ser feita apenas uma chamada para cada porta lógica diferente. Para tal, todo o circuito é executado tendo como base a variável *porta_logica* (código 7.17), que é inicializada com zero e é incrementada a cada execução de uma nova porta, com exceção da porta de Hadamard, que, como dito anteriormente, é executada em dois passos.

Código Fonte 7.17: Variável porta lógica

```
int porta_logica = 0;
```

Por exemplo, se for necessária a execução da porta de negação com o primeiro q-bit como alvo em várias etapas do circuito, sua codificação será como segue abaixo (código 7.18), onde o símbolo `||` é o operador lógico *ou*.

Código Fonte 7.18: Exemplo de Chamada de Porta Lógica

```
if ((porta_logica == 5) ||
    (porta_logica == 17) ||
    (porta_logica == 29) ||
    (porta_logica == 32)
)
    Not(1);
```

E, finalmente, outra característica que pode ser considerada como requisito de eficiência é a capacidade da solução de implementar vários tipos de circuitos. E, como será demonstrado no capítulo seguinte, com a solução proposta foi possível a implementação de circuitos aritméticos, circuitos da TFQ e circuitos do algoritmo de Grover.

Capítulo 8

Resultados

Tomando como base os recursos e ferramentas citados no capítulo anterior, procurando verificar qual a capacidade máxima de emulação suportada, foram feitos vários testes da solução proposta.

Inicialmente foi verificado se as portas lógicas haviam sido codificadas corretamente. Para tal, foram construídos circuitos hipotéticos que foram codificados no emulador. Os resultados de saída foram comparados aos resultados obtidos quando da execução do mesmo circuito no simulador Zeno.

No simulador Zeno é possível que se percorra o circuito passo-a-passo, verificando os estados quânticos a cada aplicação de uma porta lógica. O emulador foi também assim preparado, através da instalação um clock manual, podendo também ser acompanhada a execução a cada aplicação de uma porta lógica. Dessa forma, foi possível fazer a comparação das duas soluções, sendo verificado que ambas apresentaram os mesmos valores para os circuitos testados.

Num segundo momento foram feitos testes que comprovaram a reversibilidade das portas lógicas implementadas. Para tal, foram construídos circuitos quânticos, como o da figura 8.1, onde o estado quântico final, como era de se esperar, foi o mesmo do estado quântico antes da execução do circuito.

Estando as portas lógicas corretamente codificadas, foram iniciados os testes com os circuitos específicos.

A primeira tarefa a ser executada, para todos os circuitos, é a atribuição de um valor inicial para o estado quântico (código 8.1). Nesse exemplo as amplitudes de todos os estados

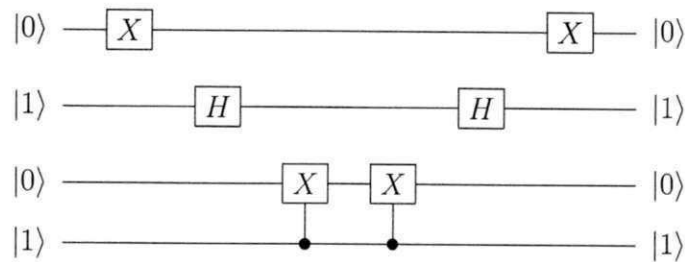


Figura 8.1: Circuito Quântico.

quânticos (reais e imaginárias) são setadas para 0, com exceção do estado $|0 \dots 1\rangle$, onde a amplitude real é setada para 1.

Código Fonte 8.1: Inicialização dos estados quânticos

```
task Inicia_estados;
  for (i = 0; i < 1 << qtde_qbits; i ++)
    begin
      amp_real[i] <= '0;
      amp_imag[i] <= '0;
    end;
  amp_real[1] <= 010<<<(precisao -1);
  amp_imag[1] <= '0;
endtask // Inicia_estados
```

Após a inicialização do estado quântico é dada sequência à execução dos algoritmos quânticos propriamente ditos, conforme destacado nas seções seguintes.

Para a visualização dos resultados (as amplitudes dos estados quânticos) foram utilizados os visores de sete segmentos, as luzes verdes e vermelhas e os dezoito interruptores da placa FPGA.

Em todos os circuitos implementados foi utilizada a ferramenta *TimeQuest Timing Analyzer*, disponibilizada no Quartus, que analisa o circuito e o clock informado e verifica se o mesmo está adequado. A informação *Slack* do *Report Setup Summary* indica se o clock escolhido é ideal para a execução do circuito. Quanto mais próximo de zero, mais ajustado estará o clock. Se o valor for negativo o clock foi subestimado, sendo necessário aumentar sua frequência. Caso contrário, o mesmo foi superestimado, devendo a frequência ser diminuída.

8.1 Circuitos Aritméticos

A lista dos algoritmos aritméticos é extensa [48].

Para efeito de testes foram escolhidos os algoritmos *3_17tc*, *ham3tc*, *graycode*, *rd53d2*, *6sym* e $gf2 \wedge 4mult$.

Para todos os circuitos acima foram implementadas versões com 8, 16 e 32 bits de precisão para os números complexos.

8.1.1 Circuito 3_17tc

O circuito *3_17tc*, de três q-bits, representado pela figura 8.2, é um circuito booleano aritmético reversível, muito utilizado como circuito de teste, que representa o pior caso possível para uma função de 3 variáveis, apresentado em [29]; foi implementado conforme o código 8.2.

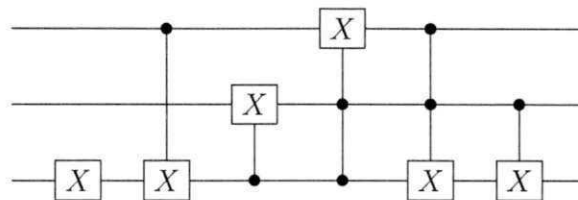


Figura 8.2: Circuito Aritmético *3_17tc*.

Código Fonte 8.2: Código do Circuito Aritmético *3_17tc*

```
...
int porta_logica = 0;

always @(negedge reset_n or posedge iCLK_50)
  if (!reset_n)
    porta_logica <= 0;
  else
    porta_logica <= porta_logica + 1;

always_ff @ (posedge iCLK_50)
  if (porta_logica == 0)
    Inicia_estados; else
    if ((porta_logica == 1)
```



```

Not(1); else
  if ((porta_logica == 2)
CNot(1,3); else
  if ((porta_logica == 3)
CNot(2,1); else
  if ((porta_logica == 4)
CCNot(3,1,2); else
  if ((porta_logica == 5)
CCNot(1,2,3); else
  if ((porta_logica == 6)
CNot(1,2);

```

...

Conforme relatório de compilação da ferramenta Quartus, para a emulação do circuito, foram necessários 48 elementos lógicos, para todas as precisões.

Para as precisões de 8 e 16 bits foi estimado um clock de 255MHz e para a precisão de 32 bits, um clock de 253,3MHz. Como o circuito é executado em sete pulsos de clock, para as precisões 8 e 16 são necessários 27,4ns para a execução e para a precisão 32 são necessários 27,6ns.

8.1.2 Circuito ham3tc

O circuito ham3tc, de três q-bits (figura 8.3), que implementa a função de codificação de Hamming, foi implementado conforme o código 8.3.

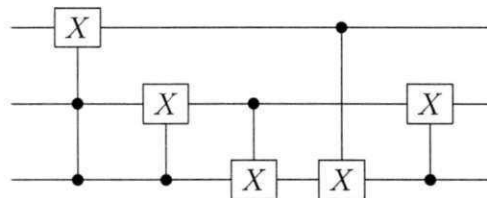


Figura 8.3: Circuito Aritmético *ham3tc*.

Código Fonte 8.3: Código do Circuito Aritmético *ham3tc*

...

```
int porta_logica = 0;
```

```
always @(negedge reset_n or posedge iCLK_50)
```

```

if (!reset_n)
    porta_logica <= 0;
else
    porta_logica <= porta_logica + 1;

always_ff @ (posedge iCLK_50)
    if (porta_logica == 0)
        Inicia_estados; else
        if ((porta_logica == 1)
            CCNot(3,1,2); else
        if ((porta_logica == 2)
            CNot(2,1); else
        if ((porta_logica == 3)
            CNot(1,2); else
        if ((porta_logica == 4)
            CNot(1,3); else
        if ((porta_logica == 5)
            CNot(2,1);

```

...

Para essa emulação foram necessários 10 elementos lógicos, independentemente da precisão utilizada, sendo o mesmo executado em seis pulsos de clock, gastando 11ns de execução, chegando a uma frequência de 509MHz.

8.1.3 Circuito Graycode

O circuito graycode, de seis q-bits, representado pela figura 8.4, que transforma um número K (no intervalo $0 \dots 2^6 - 1$) no k-ésimo padrão do código Gray, foi implementado conforme o código 8.4.

Código Fonte 8.4: Código do Circuito Aritmético Graycode

```

...
int porta_logica = 0;

always @(negedge reset_n or posedge iCLK_50)
    if (!reset_n)
        porta_logica <= 0;

```

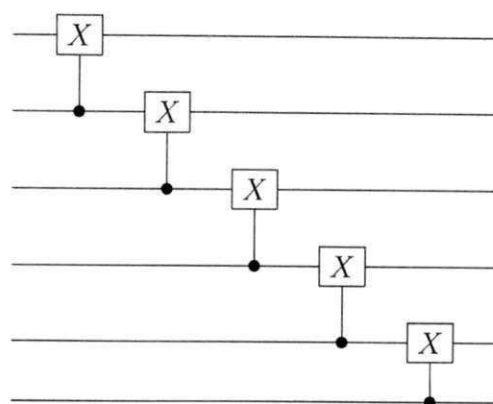


Figura 8.4: Circuito Aritmético Graycode.

```
else
    porta_logica <= porta_logica + 1;
```

```
always_ff @ (posedge iCLK_50)
    if (porta_logica == 0)
        Inicia_estados; else
        if ((porta_logica == 1)
            CNot(6,5); else
        if ((porta_logica == 2)
            CNot(5,4); else
        if ((porta_logica == 3)
            CNot(4,3); else
        if ((porta_logica == 4)
            CNot(3,2); else
        if ((porta_logica == 5)
            CNot(2,1);
```

...

Para a emulação do circuito graycode foram necessários 11 elementos lógicos, para as três configurações de precisão, sendo o mesmo também executado em seis pulsos de clock, levando 12 nanossegundos, a uma frequência de 491,2MHz.

8.1.4 Circuito rd53d2

O circuito de 8 q-bits, rd53d2 (figura 8.5), cujo objetivo é gerar como saída a codificação binária da quantidade de uns do número binário oferecido como entrada, foi implementado

como listado no código 8.5.

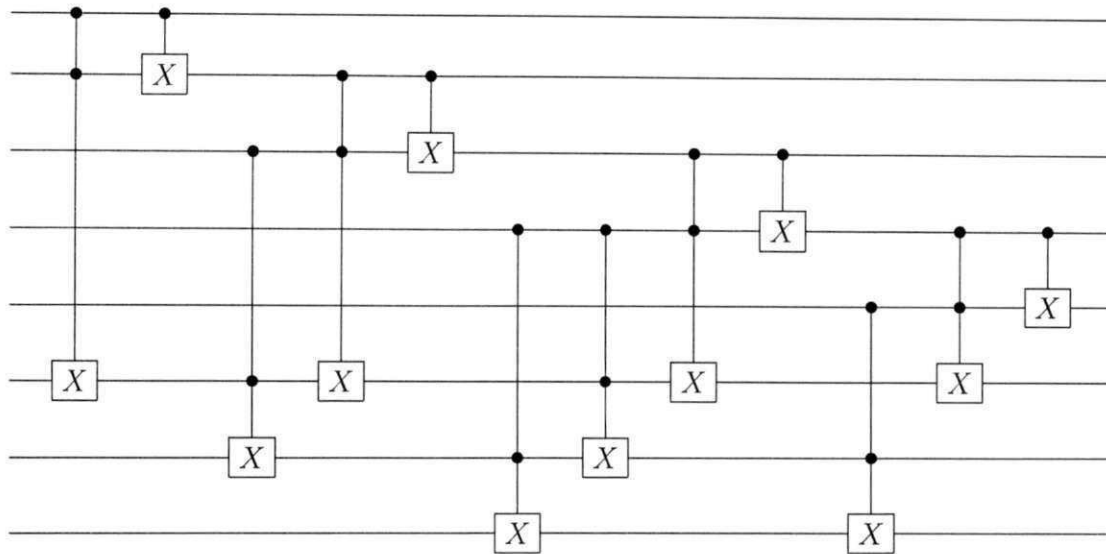


Figura 8.5: Circuito Aritmético rd53d2.

Código Fonte 8.5: Código do Circuito Aritmético rd53d2

```
...
int porta_logica = 0;

always @(negedge reset_n or posedge iCLK_50)
  if (!reset_n)
    porta_logica <= 0;
  else
    porta_logica <= porta_logica + 1;

always_ff @ (posedge iCLK_50)
  if (porta_logica == 0)
    Inicia_estados; else
    if ((porta_logica == 1)
    CCNot(3,7,8); else
    if ((porta_logica == 2)
    CNot(7,8); else
    if ((porta_logica == 3)
    CCNot(2,3,6); else
    if ((porta_logica == 4)
    CCNot(3,6,7); else
```

```

    if ((porta_logica == 5)
    CNot(6,7); else
    if ((porta_logica == 6)
    CCNot(1,2,5); else
    if ((porta_logica == 7)
    CCNot(2,3,5); else
    if ((porta_logica == 8)
    CCNot(3,5,6); else
    if ((porta_logica == 9)
    CNot(5,6); else
    if ((porta_logica == 10)
    CCNot(1,2,4); else
    if ((porta_logica == 11)
    CCNot(3,4,5); else
    if ((porta_logica == 12)
    CNot(4,5);

```

...

No circuito rd53d2 foram utilizados 11 elementos lógicos, independentemente da precisão, levando 24 nanossegundos para sua execução, a uma frequência de 509MHz, já que foram necessários doze pulsos de clock.

8.1.5 Circuito 6sym

O circuito de 10 q-bits 6sym (figura 8.6), uma função simétrica de seis entradas e uma saída, que retorna o valor um se e somente se, a entrada for 2, 4 ou 6; foi implementado como demonstrado no código 8.6.

Código Fonte 8.6: Código do Circuito Aritmético 6sym

```

...
int porta_logica = 0;

always @(negedge reset_n or posedge iCLK_50)
    if (!reset_n)
        porta_logica <= 0;
    else
        porta_logica <= porta_logica + 1;

```

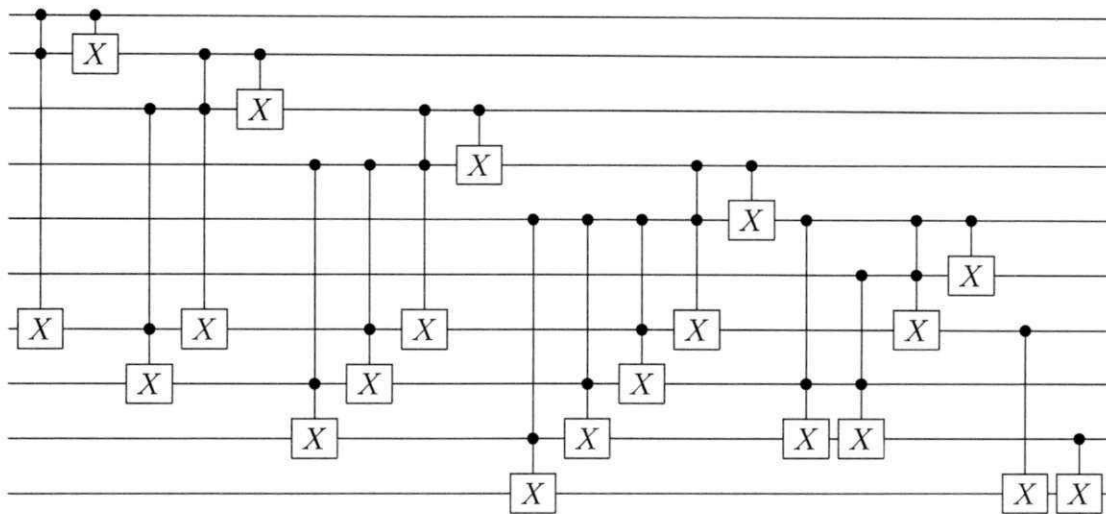


Figura 8.6: Circuito Aritmético 6sym.

```

always_ff @ (posedge iCLK_50)
  if (porta_logica == 0)
    Inicia_estados; else
    if ((porta_logica == 1)
CCNot(4,9,10); else
    if ((porta_logica == 2)
CNot(9,10); else
    if ((porta_logica == 3)
CCNot(3,4,8); else
    if ((porta_logica == 4)
CCNot(4,8,9); else
    if ((porta_logica == 5)
CNot(8,9); else
    if ((porta_logica == 6)
CCNot(2,3,7); else
    if ((porta_logica == 7)
CCNot(3,4,7); else
    if ((porta_logica == 8)
CCNot(4,7,8); else
    if ((porta_logica == 9)
CNot(7,8); else
    if ((porta_logica == 10)
CCNot(1,2,6); else

```

```

    if ((porta_logica == 11)
    CCNot(2,3,6); else
    if ((porta_logica == 12)
    CCNot(3,4,6); else
    if ((porta_logica == 13)
    CCNot(4,6,7); else
    if ((porta_logica == 14)
    CNot(6,7); else
    if ((porta_logica == 15)
    CCNot(1,2,5); else
    if ((porta_logica == 16)
    CCNot(2,3,5); else
    if ((porta_logica == 17)
    CCNot(4,5,6); else
    if ((porta_logica == 18)
    CNot(5,6); else
    if ((porta_logica == 19)
    CNot(1,4); else
    if ((porta_logica == 20)
    CNot(1,2);

```

...

Essa emulação consumiu 11 elementos lógicos, em todas as precisões, sendo executado em 42 nanossegundos, com 20 pulsos de clock, a uma frequência de 500MHz.

8.1.6 Circuito $gf2^4$ mult

Como último exemplo de circuito aritmético, foi implementado (código 8.7) o circuito de 12 q-bits $gf2^4$ mult (figura 8.7), que implementa a função de multiplicação de campo finito ou campo de Galois.

Código Fonte 8.7: Código do Circuito Aritmético $gf2^4$ mult

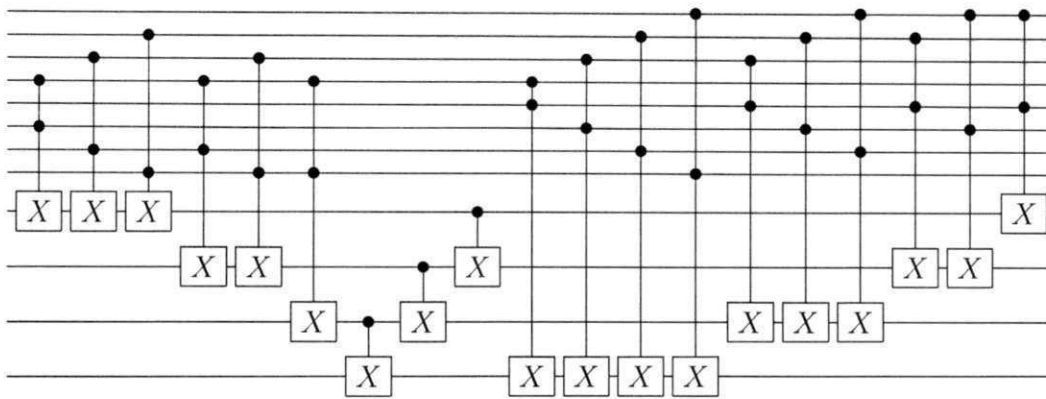
...

```

int porta_logica = 0;

always @(negedge reset_n or posedge iCLK_50)
    if (!reset_n)

```

Figura 8.7: Circuito Aritmético $gf2^4$ mult.

```

        porta_logica <= 0;
    else
        porta_logica <= porta_logica + 1;

always_ff @ (posedge iCLK_50)
    if (porta_logica == 0)
        Inicia_estados; else
        if ((porta_logica == 1)
            CCNot(4,7,9); else
        if ((porta_logica == 2)
            CCNot(4,6,10); else
        if ((porta_logica == 3)
            CCNot(4,5,11); else
        if ((porta_logica == 4)
            CCNot(3,6,9); else
        if ((porta_logica == 5)
            CCNot(3,5,10); else
        if ((porta_logica == 6)
            CCNot(2,5,9); else
        if ((porta_logica == 7)
            CNot(1,2); else
        if ((porta_logica == 8)
            CNot(2,3); else
        if ((porta_logica == 9)
            CNot(3,4); else
        if ((porta_logica == 10)
            CCNot(1,8,9); else

```



```

if ((porta_logica == 11)
CCNot(1,7,10); else
if ((porta_logica == 12)
CCNot(1,6,11); else
if ((porta_logica == 13)
CCNot(1,5,12); else
if ((porta_logica == 14)
CCNot(2,8,10); else
if ((porta_logica == 15)
CCNot(2,7,11); else
if ((porta_logica == 16)
CCNot(2,6,12); else
if ((porta_logica == 17)
CCNot(3,8,11); else
if ((porta_logica == 18)
CCNot(3,7,12); else
if ((porta_logica == 19)
CCNot(4,8,12);

```

...

Essa emulação consumiu 11 elementos lógicos, em todas as precisões, sendo executado em 42 nanossegundos, com 20 pulsos de clock, a uma frequência de 509MHz.

8.2 O Circuito da TFQ

Para o circuito da TFQ foram feitas emulações com 3 (figura 8.8), 4 (figura 8.9), 5 (figura 8.10) e 6 (figura 8.11) q-bits, para precisões de 8, 16 e 32 bits.

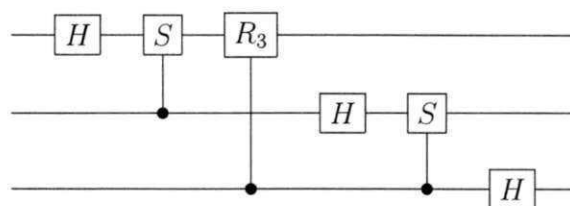


Figura 8.8: Circuito da TFQ de 3 q-bits.

Para a emulação da TFQ de 3 q-bits (código 8.8) foram necessários 2.341 elementos lógicos para precisão 8, 3.002 para precisão 16 e 2.799 para precisão 32.

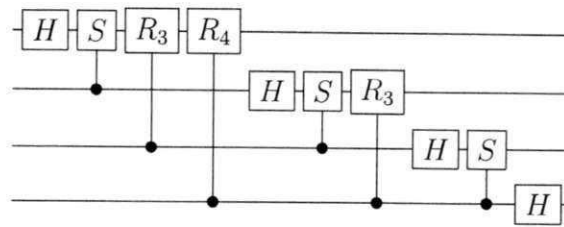


Figura 8.9: Circuito da TFQ de 4 q-bits.

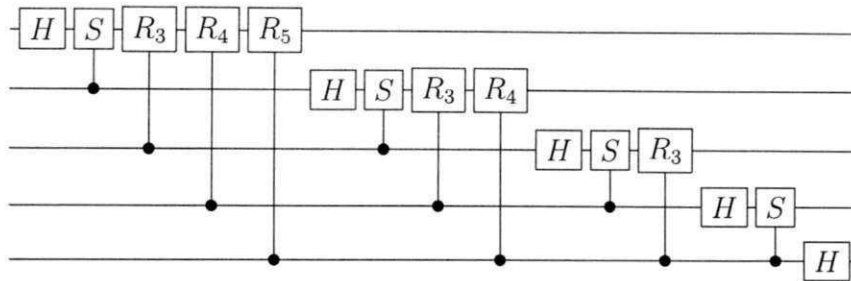


Figura 8.10: Circuito da TFQ de 5 q-bits.

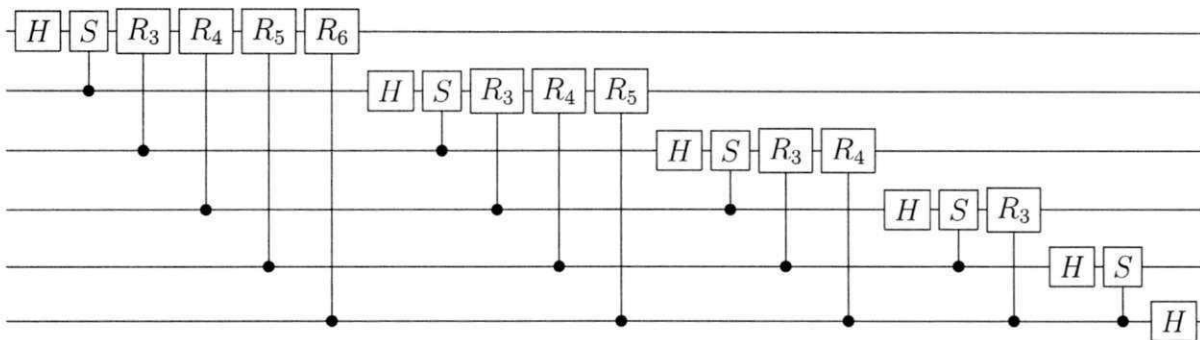


Figura 8.11: Circuito da TFQ de 6 q-bits.

A emulação ocorre em dez pulsos de clock¹, trabalhando a uma frequência de 86,9MHz, 68,5MHz e 75,6MHz, para as precisões 8, 16 e 32, respectivamente, com tempos de execução de 115, 146 e 132 nanossegundos, sendo necessários 2314, 3002 e 2800 elementos lógicos para cada configuração.

Código Fonte 8.8: Código da TFQ de 3 q-bits

```
...
int porta_logica = 0;

always @(negedge reset_n or posedge iCLK_50)
    if (!reset_n)
        porta_logica <= 0;
```

¹Lembrando que cada porta de Hadamard consome dois pulsos.

```

else
    porta_logica <= porta_logica + 1;

always_ff @ (posedge iCLK_50)
    if (porta_logica == 0)
        Inicia_estados; else
    if ((porta_logica == 1) || (porta_logica == 5) || (porta_logica
        == 8)
        )
        Hadamard_1; else
    if ((porta_logica == 2))
        Hadamard_2(6); else
    if ((porta_logica == 3))
        Porta_S(3, 2); else
    if ((porta_logica == 4))
        Rotacao(3, 1, rot3a, rot3b); else
    if ((porta_logica == 6))
        Hadamard_2(2); else
    if ((porta_logica == 7))
        Porta_S(2, 1); else
    if ((porta_logica == 9))
        Hadamard_2(4);

...

```

Para o circuito da TFQ de 4 q-bits (código 8.9) foram utilizados 5.120, 6.413 e 6.276 elementos lógicos, para as respectivas precisões 8, 16 e 32, com clocks de 70,2MHz, 61,6MHz e 60,4MHz e tempos de execução de 200ns, 228ns e 232ns, necessários para a execução dos 14 pulsos de clock.

Código Fonte 8.9: Código da TFQ de 4 q-bits

```

...
int porta_logica = 0;

always @(negedge reset_n or posedge iCLK_50)
    if (!reset_n)
        porta_logica <= 0;
    else

```

```
    porta_logica <= porta_logica + 1;

always_ff @ (posedge iCLK_50)
    if (porta_logica == 0)
        Inicia_estados; else
    if ((porta_logica == 1) || (porta_logica == 6) || (porta_logica
        == 10) ||
        (porta_logica == 13)
        )
        Hadamard_1; else
    if ((porta_logica == 2))
        Hadamard_2(4); else
    if ((porta_logica == 3))
        Porta_S(4, 3); else
    if ((porta_logica == 4))
        Rotacao(4, 2, rot3a, rot3b); else
    if ((porta_logica == 5))
        Rotacao(4, 1, rot4a, rot4b); else
    if ((porta_logica == 7))
        Hadamard_2(3); else
    if ((porta_logica == 8))
        Porta_S(3, 2); else
    if ((porta_logica == 9))
        Rotacao(3, 1, rot3a, rot3b); else
    if ((porta_logica == 11))
        Hadamard_2(2); else
    if ((porta_logica == 12))
        Porta_S(2, 1); else
    if ((porta_logica == 14))
        Hadamard_2(1);
...

```

A emulação da TFQ de 5 q-bits (código 8.10) utilizou 10.936, 13.836 e 14.564 elementos lógicos, trabalhando com frequências de 62,5MHz, 61,7MHz e 61,71MHz, com tempos de execução de 320ns, 325ns e 325ns para as respectivas precisões 8, 16 e 32, sendo consumidos 20 pulsos de clock.

Código Fonte 8.10: Código da TFQ de 5 q-bits

```
...
int porta_logica = 0;

always @(negedge reset_n or posedge iCLK_50)
  if (!reset_n)
    porta_logica <= 0;
  else
    porta_logica <= porta_logica + 1;

always_ff @(posedge iCLK_50)
  if (porta_logica == 0)
    Inicia_estados; else
  if ((porta_logica == 1) || (porta_logica == 7) || (porta_logica
    == 12) ||
    (porta_logica == 16) || (porta_logica == 19)
    )
    Hadamard_1; else
  if ((porta_logica == 2))
    Hadamard_2(5); else
  if ((porta_logica == 3))
    Porta_S(5, 4); else
  if ((porta_logica == 4))
    Rotacao(5, 3, rot3a, rot3b); else
  if ((porta_logica == 5))
    Rotacao(5, 2, rot4a, rot4b); else
  if ((porta_logica == 6))
    Rotacao(5, 1, rot5a, rot5b); else
  if ((porta_logica == 8))
    Hadamard_2(4); else
  if ((porta_logica == 9))
    Porta_S(4, 3); else
  if ((porta_logica == 10))
    Rotacao(4, 2, rot3a, rot3b); else
  if ((porta_logica == 11))
    Rotacao(4, 1, rot4a, rot4b); else
  if ((porta_logica == 13))
```

```

        Hadamard_2(3); else
    if ((porta_logica == 14))
        Porta_S(3, 2); else
    if ((porta_logica == 15))
        Rotacao(3, 1, rot3a, rot3b); else
    if ((porta_logica == 17))
        Hadamard_2(2); else
    if ((porta_logica == 18))
        Porta_S(2, 1); else
    if ((porta_logica == 20))
        Hadamard_2(1);

```

...

A TFQ de 6 q-bits (código 8.11), que é executada em 29 pulsos de clock, consumiu 23.105 elementos lógicos para a precisão 8, com clock de 66,7MHz e tempo de execução de 434 nanossegundos.

Para a precisão 16 foram necessários 29.224 elementos lógicos, sendo executada em 516 nanossegundos, a uma frequência de 56,2MHz.

Não foi possível a conclusão da compilação para o circuito com 32 bits de precisão que, apesar de consumir 33.185 elementos lógicos (49% da capacidade da placa), não pode ser acomodado na placa, retornando a mensagem "*Can't fit design in device.*", após 1:40 h de compilação.

Código Fonte 8.11: Código da TFQ de 6 q-bits

...

```

int porta_logica = 0;

always @(negedge reset_n or posedge iCLK_50)
    if (!reset_n)
        porta_logica <= 0;
    else
        porta_logica <= porta_logica + 1;

always_ff @ (posedge iCLK_50)
    if (porta_logica == 0)
        Inicia_estados; else

```

```
if ((porta_logica == 1) || (porta_logica == 8) || (porta_logica
    == 14) ||
    (porta_logica == 19) || (porta_logica == 23) || (
        porta_logica == 26)
    )
    Hadamard_1; else
if ((porta_logica == 2))
    Hadamard_2(6); else
if ((porta_logica == 3))
    Porta_S(6, 5); else
if ((porta_logica == 4))
    Rotacao(6, 4, rot3a, rot3b); else
if ((porta_logica == 5))
    Rotacao(6, 3, rot4a, rot4b); else
if ((porta_logica == 6))
    Rotacao(6, 2, rot5a, rot5b); else
if ((porta_logica == 7))
    Rotacao(6, 1, rot6a, rot6b); else
if ((porta_logica == 9))
    Hadamard_2(5); else
if ((porta_logica == 10))
    Porta_S(5, 4); else
if ((porta_logica == 11))
    Rotacao(5, 3, rot3a, rot3b); else
if ((porta_logica == 12))
    Rotacao(5, 2, rot4a, rot4b); else
if ((porta_logica == 13))
    Rotacao(5, 1, rot5a, rot5b); else
if ((porta_logica == 15))
    Hadamard_2(4); else
if ((porta_logica == 16))
    Porta_S(4, 3); else
if ((porta_logica == 17))
    Rotacao(4, 2, rot3a, rot3b); else
if ((porta_logica == 18))
    Rotacao(4, 1, rot4a, rot4b); else
if ((porta_logica == 20))
```

```

    Hadamard_2(3); else
if ((porta_logica == 21))
    Porta_S(3, 2); else
if ((porta_logica == 22))
    Rotacao(3, 1, rot3a, rot3b); else
if ((porta_logica == 24))
    Hadamard_2(2); else
if ((porta_logica == 25))
    Porta_S(2, 1); else
if ((porta_logica == 27))
    Hadamard_2(1);

```

...

8.3 O Circuito de Grover

A título de ilustração, na figura 8.12 está representado um circuito de Grover com 4 q-bits, sendo 3 q-bits referentes ao espaço de busca acrescido de mais um q-bit auxiliar, como representado na figura 8.12.

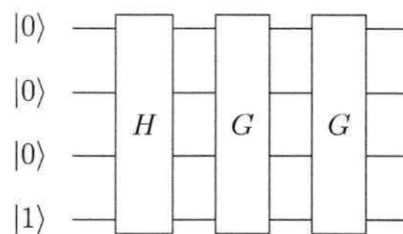


Figura 8.12: Circuito Quântico de Grover para 4 q-bits.

O símbolo \boxed{G} representa um grupo iterador de Grover, expandido na figura 8.13, que para esse caso particular, deve ser executado duas vezes.

Para otimizar a execução da sequência das portas lógicas e evitar que o mesmo código (cada iteração do algoritmo de Grover) fosse repetido no momento da codificação, foram utilizadas as variáveis *qtde_iteracoes*, *iteracao* e *repeticao_iteracao*. Dessa forma foi necessário identificar, em cada circuito particular, a quantidade de iterações necessárias, qual a execução está sendo executada e a quantidade de portas lógicas presente em cada grupo iterador, ficando o algoritmo codificado como listado a seguir (código 8.12).

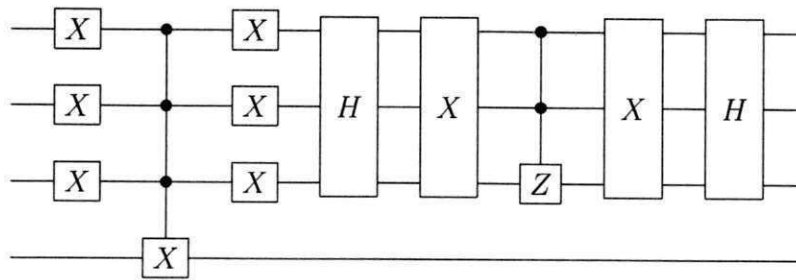


Figura 8.13: Grupo iterador de Grover para 4 q-bits.

Código Fonte 8.12: Circuito de Grover com 4 q-bits

```

int porta_logica = 0;
int iteracao = 0;
int repeticao_iteracao = 26; // alterar para cada novo circuito
int qtde_iteracoes = 2;    // alterar para cada novo circuito

always @(negedge reset_n or posedge iCLK_50)
    if (!reset_n)
        porta_logica <= 0;
    else
        porta_logica <= porta_logica + 1;

always_ff @ (posedge iCLK_50)
    if (porta_logica == 0)
        Inicia_estados; else
        if (
            (porta_logica == 1) ||
            (porta_logica == 3) ||
            (porta_logica == 5) ||
            (porta_logica == 7) ||
            (porta_logica == (16 + (iteracao *
                repeticao_iteracao))) ||
            (porta_logica == (18 + (iteracao *
                repeticao_iteracao))) ||
            (porta_logica == (20 + (iteracao *
                repeticao_iteracao))) ||
            (porta_logica == (29 + (iteracao *
                repeticao_iteracao))) ||
            (porta_logica == (31 + (iteracao *

```

```

        repeticao_iteracao))) ||
        (porta_logica == (33 + (iteracao *
        repeticao_iteracao)))
    )
    Hadamard_1; else
if ((porta_logica == 2) ||
    (porta_logica == (17 + (iteracao *
    repeticao_iteracao))) ||
    (porta_logica == (30 + (iteracao *
    repeticao_iteracao)))
    )
    Hadamard_2(4); else
if ((porta_logica == 4) ||
    (porta_logica == (19 + (iteracao * repeticao_iteracao
    ))) ||
    (porta_logica == (32 + (iteracao *
    repeticao_iteracao)))
    )
    Hadamard_2(3); else
if ((porta_logica == 6) ||
    (porta_logica == (21 + (iteracao * repeticao_iteracao
    ))) ||
    (porta_logica == (34 + (iteracao *
    repeticao_iteracao)))
    // ultima execucao do primeiro iterador
    )
begin
    Hadamard_2(2);
        if ((porta_logica == (34 + (iteracao *
        repeticao_iteracao)))
        && (iteracao < qtde_iteracoes))
            iteracao ++;
        end
    else
if (porta_logica == 8)
Hadamard_2(1); else
if ((porta_logica == (9 + (iteracao * repeticao_iteracao

```

```
    ))) ||
    (porta_logica == (15 + (iteracao * repeticao_iteracao
    ))) ||
    (porta_logica == (22 + (iteracao * repeticao_iteracao
    ))) ||
    (porta_logica == (28 + (iteracao * repeticao_iteracao
    )))
    )
    Not(4); else
if ((porta_logica == (10 + (iteracao * repeticao_iteracao
))) ||
    (porta_logica == (14 + (iteracao * repeticao_iteracao
    ))) ||
    (porta_logica == (23 + (iteracao * repeticao_iteracao
    ))) ||
    (porta_logica == (27 + (iteracao * repeticao_iteracao
    )))
    )
    Not(3); else
if ((porta_logica == (11 + (iteracao * repeticao_iteracao
))) ||
    (porta_logica == (13 + (iteracao * repeticao_iteracao
    ))) ||
    (porta_logica == (24 + (iteracao * repeticao_iteracao
    ))) ||
    (porta_logica == (26 + (iteracao * repeticao_iteracao
    )))
    )
    Not(2); else
    if ((porta_logica == (12 + (iteracao * repeticao_iteracao
    )))
    )
    CCNot_m(1); else
    if ((porta_logica == (25 + (iteracao * repeticao_iteracao
    )))
    )
    Porta_Z_m(2);
```

Inicialmente foi codificado um circuito de Grover com 3 q-bits. Para as precisões 8, 16 e 32 foram necessárias 2.527, 3.397 e 3.224 elementos lógicos, respectivamente, com frequências de 79,8MHz, 60,2MHz e 69,7MHz, sendo os circuitos executados em 314, 416 e 359 nanossegundos.

Posteriormente foi implementado um circuito com 4 q-bits, que utilizou, fazendo as duas iterações necessárias do operador de Grover, 5.723, 7.093 e 8.174 elementos lógicos, com frequências de 50MHz, 48,8MHz e 46,8MHz e tempos de execução de 1, 22, 1, 25 e 1, 31 microssegundos, para, respectivamente, as precisões 8, 16 e 32.

Na emulação do circuito de Grover com 5 q-bits, para a precisão 8 foram necessários 12.531 elementos lógicos, a uma frequência de 47,98MHz e levando $2,335 \times 10^{-6}$ s. Para 16 bits de precisão foram utilizados 17.331 elementos lógicos, com frequência de 41,6MHz e tempo de execução de $2,716 \times 10^{-6}$ s. E finalmente, para 32 bits, foram consumidos 18.869 elementos lógicos, a 44,86MHz e com execução de $2,519 \times 10^{-6}$ s.

Para emulação de um circuito com 7 q-bits foi alcançada a precisão de 8 bits sendo executadas seis iterações do operador de Grover. Nessa configuração foram consumidos 61.416 elementos lógicos².

Quando foi tentada uma precisão de 12 bits foi levantada a necessidade de 73, 450 elementos lógicos, o que excedeu a capacidade da placa. O tempo de síntese da ferramenta Quartus, para tal configuração, foi de 19:47 h.

Na tentativa de melhorar a saída da ferramenta Quartus foi usada a ferramenta Synplify[53] que tem por objetivo realizar uma otimização do código, em alto nível, antes da fase de síntese do circuito. Essa ferramenta atua diretamente no código fonte, gerando um arquivo com a extensão *vqm*, que é então fornecido à ferramenta Quartus para a síntese do circuito. Porém, da forma que o código foi escrito, a ferramenta não acrescentou melhoras ao código sintetizado, não apresentando diminuição do tempo de síntese e nem da quantidade de elementos lógicos utilizados, sendo essa ferramenta descartada.

Como visto pelos resultados acima, a placa FPGA escolhida estava impossibilitando que fosse possível a construção de um circuito de Grover com um maior número de q-bits e com a precisão e a quantidade de iterações ideais.

Mesmo não possuindo fisicamente uma placa de maior capacidade, a ferramenta Quar-

²O código completo está no apêndice B.

tus permite que sejam feitas sínteses para diversas placas cadastradas em seu sistema. Foi tentada, então, a emulação de circuitos usando, hipoteticamente, uma placa de maior quantidade de elementos lógicos. Foi escolhida uma placa da família STRATIX IV que possui uma quantidade de elementos lógicos pelo menos dez vezes maior que a placa DE2-70.

Na primeira tentativa de emular o circuito de Grover de 7 q-bits, com precisão de 16 bits, depois de cinquenta e um minutos de execução, a ferramenta cancelou o processo dando uma mensagem de estouro de memória. Ou seja, o computador que estava executando a ferramenta não dispunha de memória suficiente para dar seguimento ao processo de síntese do circuito.

Tentando ainda fazer a síntese do circuito acima foi utilizado outro computador com 4GB de RAM. Mesmo assim a ferramenta não conseguiu concluir a síntese do circuito, ainda por problemas de memória.

Como última alternativa, foi utilizado um computador com o sistema operacional CentOS de 64 bits, com 4GB de memória RAM e mais 16GB de memória virtual.

Nessa configuração foi possível a síntese de circuitos de Grover com 12 q-bits, com precisão de 16 bits, executando 50 iterações, sendo necessárias mais de 15 h para a conclusão do processo. Porém, não foi possível a execução da solução por não ser viável a aquisição da placa escolhida.

8.4 Análise dos Resultados

Como pôde ser visto dos resultados expostos acima, a solução aqui proposta apresentou resultados bastante satisfatórios, principalmente quando comparados com os resultados obtidos em trabalhos anteriores.

No trabalho apresentado por Aminian et al [4], foram emulados circuitos quânticos aritméticos e a TFQ.

Com relação aos circuitos quânticos aritméticos (vide tabela 8.1), excetuando o circuito *3_17tc*, cujo trabalho proposto consumiu 48 elementos lógicos contra 24 do trabalho anterior, as demais implementações tiveram ganhos consideráveis, com maior destaque para o circuito *rd53d2* que na solução proposta necessitou de 10 elementos lógicos contra 3.072 do referido trabalho. Ainda foi possível a emulação do circuito *6sym* com 10 q-bits e do circuito $gf2 \wedge$

Circuito	Precisão	Elementos Lógicos [4]	Elementos Lógicos Nossa Solução	Tempo Execução [4]	Tempo Execução Nossa Solução (ns)
3_17tc 3 q-bits	8	24	48	4,48	27,4
	16				27,4
	32				27,6
ham3tc 3 q-bits	8	24	10	4,48	11
	16				
	32				
graycode 6 q-bits	8	320	11	4,45	12
	16				
	32				
rd53d2 8 q-bits	8	3072	10	7,5	24
	16				
	32				
6sym 10 q-bits	8	-	11	-	42
	16				
	32				
gf2^4mult 12 q-bits	8	-	11	-	37,3
	16				
	32				
TFQ 3 q-bits	8	3905	2341	-	115
	16	8197	3002		146
	32	-	2800		132
TFQ 6 q-bits	8	-	12531	-	2335
	16		17331		2716
	32		18869		2519

Tabela 8.1: Comparação dos Resultados de Emulação dos Circuitos Quânticos

$4mult$ com 12 q-bits, ambos com precisão de até 32 bits, sendo que no outro trabalho foi implementado um circuito com 8 q-bits e precisão máxima de 16 bits.

Com relação aos tempos de execução dos circuitos aritméticos, a solução proposta apresentou um desempenho inferior, mas que ainda permaneceu na casa das dezenas de nanossegundos.

Para o circuito da TFQ de 3 q-bits a solução aqui proposta apresentou bons resultados. Foi obtida uma precisão maior, chegando a 32 bits. A quantidade de elementos lógicos foi significativamente reduzida, de 3.905 para 2.341 (redução de 40%) para precisão 8 e de 8.197 para 3.002 (redução de 63,4%) para a precisão de 16. Porém, como nos circuitos aritméticos,

Circuito	Qtde q-bits	Precisão [23] e [24]	Elementos Lógicos [23] e [24]	Precisão Nossa Solução	Elementos Lógicos Nossa Solução
TFQ	3	12	11271	16	3002
	4	12	16687	16	6413
	5	12	21898	16	13836
	6	-	-	16	29224
Grover	3	12	14284	16	3397
	4	12	23525	16	7093
	5	12	30121	16	17331
	7	-	-	8	61416
	12	-	-	16	-

Tabela 8.2: Comparação dos Resultados de Emulação dos Circuitos Quânticos

a frequência de execução atingida aqui foi mais baixa, caindo de $137,9MHz$ para $86,9MHz$ e de $131,3MHz$ para $68,5MHz$ para as mesmas precisões. No presente trabalho foi possível a implementação da TFQ de 6 q-bits não presente em trabalhos anteriores.

Os resultados aqui obtidos foram também comparados com os resultados de [23] e [24] (vide tabela 8.2), que implementaram os circuitos de Grover de da TFQ.

Em [23] foram emulados os circuitos da TFQ e de Grover, com precisão de 12 bits, para 3, 4 e 5 q-bits. Para os circuitos da TFQ foram necessários 11.271, 16.687 e 21.898 elementos lógicos, respectivamente. Para os circuitos de Grover foram utilizados 14.284, 23.525 e 30.121 elementos lógicos. Aqui foi possível a emulação da TFQ com 6 q-bits e 16 bits de precisão, não presente nos outros trabalhos.

Comparando os resultados da TFQ com os resultados aqui obtidos, mesmo considerando a precisão de 16 bits, para o circuito de 3 q-bits foram necessários 3.002 elementos lógicos, representando uma redução de 73,4%. Para 4 q-bits, foram utilizados 6.413 elementos lógicos, com uma redução de 61,5%. E, para 5 q-bits, foi observada uma redução de 58,2%, sendo necessários 13.836 elementos lógicos.

Quando feita a comparação com o circuito de Grover, também considerando a precisão de 16 bits, houve uma redução de 76,2% elementos lógicos, saindo de 14.284 para 3.397, para o circuito de 3 q-bits, uma redução de 51,1% elementos lógicos para o circuito de 4 q-bits, caindo de 23.525 para 7.093 e uma redução de 42,5% para o circuito de 5 q-bits, com 30.121 elementos lógicos contra 17.331 da solução aqui proposta.

Foi possível a emulação de um circuito de Grover com 7 q-bits, mesmo com 8 bits de precisão, implementado na placa de FPGA disponível e ainda, foi feita a síntese de um circuito de Grover de 12 q-bits, com precisão de 16 bits, tendo como base uma placa de FPGA não disponível para testes, motivo que não viabilizou a execução do mesmo. Ainda aqui, para os circuitos executados, o tempo de execução da solução proposta se apresentou inferior ao referido trabalho.

Já no trabalho [24] foram tratados apenas circuitos de 3 q-bits, com precisão de 16 bits, onde a TFQ foi emulada com 5.076 e Grover com 12.636 elementos lógicos, que tiveram seus valores reduzidos em 40,9% e 73,12%, respectivamente. Já em relação à frequência de execução, no referido trabalho a mesma ficou em 82,1MHz para os dois circuitos, enquanto que aqui foram obtidas frequências de 68,5MHz para a TFQ e 48,8MHz para Grover.

Capítulo 9

Conclusão

Com os recursos disponíveis para o desenvolvimento do emulador quântico e seguindo as características definidas para a solução, foi possível a emulação de diferentes classes de circuitos e com precisões variadas, sendo alcançado o objetivo da universalidade da solução, ou seja, da capacidade de se emular circuitos diferentes sem a necessidade de grandes esforços e mudanças.

Os resultados das técnicas de emulação de trabalhos anteriores, com relação ao tempo de execução dos circuitos, quando comparados com simuladores, foram bastante satisfatórios ao emular circuitos com pequena quantidade de q-bits. Sendo assim, os autores previam, e sugeriram como trabalho futuro, que seria possível o tratamento de circuitos com maior número de q-bits.

Como descrito no capítulo anterior, foi observada uma redução bastante significativa com relação ao número de elementos lógicos necessários para a emulação dos circuitos quânticos, mesmo comprometendo os tempos de execução, que ficaram superiores aos tempos de execução dos emuladores anteriormente implementados, porém ainda muito melhores que os tempos dos simuladores. Graças a essa redução foi possível a emulação de circuitos com mais q-bits que as soluções anteriores, como o circuito aritmético $gf2 \wedge 4mult$, com 12 q-bits e a TFQ de 6 q-bits (com precisão 16).

Para o algoritmo de Grover, foi possível a emulação de um circuito com 7 q-bits, porém com uma precisão de apenas 8 bits, representando, ainda assim, uma evolução em relação aos trabalhos anteriores.

Particularmente, para o circuito de Grover, foram tentadas outras configurações, que

só seriam possíveis serem implementadas se fossem disponibilizados recursos tecnológicos mais avançados (a nível de computador e placa FPGA) .

Nessa solução foram preservados o paralelismo e o emaranhamento quânticos, também levantos como requisitos, deixando a solução mais próxima da realidade quântica, sem a necessidade da utilização de estruturas clássicas para simular essas características, principalmente com relação ao paralelismo, onde cada porta quântica é aplicada simultaneamente em todos os estados, mesmo que para isso, como já dito, tenha causado um comprometimento no tempo de execução dos circuitos.

Além do mais, o presente trabalho permitiu a aquisição de *expertise* com relação à utilização da plataforma FPGA para tratamento de circuitos quânticos, no IQuanta - Instituto de Estudos em Computação e Informação Quânticas.

Porém, apesar do trabalho aqui proposto atingir seus objetivos: a implementação de um emulador universal, preservando os fundamentos da mecânica quântica e possibilitando a implementação de circuitos com uma maior quantidade de q-bits; é possível ser verificado, pelos resultados obtidos, que a utilização das placas FPGA não é a melhor alternativa para o tratamento de circuitos quânticos, pois existem simuladores, como por exemplo o simulador Zeno, que podem mais facilmente implementar circuitos quânticos, de maneira mais intuitiva, com mais q-bits e sem a necessidade de recursos computacionais avançados.

Bibliografia

- [1] 2010. Disponível em: [http : //www.altera.com/products/fpga.html](http://www.altera.com/products/fpga.html). Acessado em 05/04/2011.
- [2] Altera quartus, 2010. Disponível em: [http : //en.wikipedia.org/wiki/Altera_Quartus](http://en.wikipedia.org/wiki/Altera_Quartus). Acessado em 11/04/2011.
- [3] Rafael S. de Oliveira Alves. Álgebra geométrica e o algoritmo de grover. Master's thesis, Universidade Estadual de Campinas . Instituto de Matemática, Estatística e Computação Científica, 2008.
- [4] M. Aminian, M. Saeedi, M.S. Zamani, and M. Sedighi. Fpga-based circuit model emulation of quantum algorithms. In *Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual*, pages 399 –404, april 2008.
- [5] Alexandre de A. Barbosa. Um simulador simbólico de circuitos quânticos. Master's thesis, Universidade federal de Campina Grande., 2007.
- [6] Gustavo Eulálio Miranda Cabral. Uma ferramenta para projeto e simulação de circuitos quânticos. Master's thesis, Universidade Federal de Campina Grande, 2004.
- [7] Fabbryccio A. C. M. Cardoso and Marcelo A. C. Fernandes. Introdução a fpga, 2007. Disponível em: [http : //www.decom.fee.unicamp.br/cardoso/ie344b/Introducao_FPGA_Fluxo_de_Projeto.pdf](http://www.decom.fee.unicamp.br/cardoso/ie344b/Introducao_FPGA_Fluxo_de_Projeto.pdf). Acessado em 05/04/2011.
- [8] 2005. Disponível em: [http : //www.scottaaronson.com/chp/](http://www.scottaaronson.com/chp/). Acessado em 14/06/2011.

- [9] Bruno Leonardo Martins de Melo and Túlio Vinícius Duarte Christofolletti. Computação quântica: Estado da arte, 2003. Disponível em <http://www.inf.ufsc.br/barreto/trabaluno/TCBrunoTulio.pdf>. Acessado em 4/3/2010.
- [10] David Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. In *Proceedings of the Royal Society.*, volume 400, pages 97–117, 1985.
- [11] Moayad A. Fahdil, Ali Foud Al-Azawi, and Sammer Said. Operations algorithms on quantum computer. In *International Journal of Computer Science and Network Security*, volume 10, pages 85–95, Janeiro 2010.
- [12] Tarcísio Fisher. Hdl. hardware description language, 2011. Disponível em: <http://www.pet.inf.ufsc.br/sites/default/files/hdl.pdf>. Acessado em 05/04/2011.
- [13] M. Fujishima. Fpga-based high-speed emulator of quantum computing. In *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, pages 21 – 26, dec. 2003.
- [14] M. Fujishima, K. Saito, M. Onouchi, and H. Hoh. High-speed processor for quantum-computing emulation and its applications. In *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, volume 4, pages IV–884 – IV–887 vol.4, may 2003.
- [15] Atsushi Goto. A new operation principle for solid-state nuclear magnetic resonance (nmr) quantum computers. a step toward scalable nmr quantum computations., Julho 2011. Disponível em <http://www.nims.go.jp/eng/news/press/2011/07/p201107060.html>. Acessado em 16/08/2011.
- [16] M. S. Grinolds, P. Maletinsky, S. Hong, M. D. Lukin, R. L Walsworth, and A. Yacoby. Quantum control of proximal spins using nanoscale magnetic resonance imaging., Março 2011. Disponível em <http://arxiv.org/abs/1103.0546>. Acessado em 10/05/2011.

- [17] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM.
- [18] Kuk-Hyun Han and Jong-Hwan Kim. Genetic quantum algorithm and its application to combinatorial optimization problem. In *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, volume 2, pages 1354–1360 vol.2, 2000.
- [19] Winfried K. Hensinger. Quantum information: Microwave ion-trap quantum computing., Agosto 2011. Disponível em [http : //www.nature.com/nature/journal/v476/n7359/full/476155a.html](http://www.nature.com/nature/journal/v476/n7359/full/476155a.html). Acessado em 10/09/2011.
- [20] 2000. Disponível em: [http : //www.eng.buffalo.edu/phygons/jaQuzzi/index.html](http://www.eng.buffalo.edu/phygons/jaQuzzi/index.html). Acessado em 11/06/2011.
- [21] 2011. Disponível em: [http : //jqquantum.sourceforge.net/](http://jqquantum.sourceforge.net/). Acessado em 11/06/2011.
- [22] Phillip Kaye, Raymond Laflamme, and Michele Mosca. *An Introduction to Quantum Computing*. Oxford University Press, Inc., New York, NY, USA, 2007.
- [23] Ahmed Usman Khalid. Fpga emulation of quantum circuits. Master's thesis, McGill University, 2005.
- [24] Ahmed Usman Khalid, Zeljko Zilic, and Katarzyna Radecka. Fpga emulation of quantum circuits. In *Proceedings of the IEEE International Conference on Computer Design*, pages 310–315, Washington, DC, USA, 2004. IEEE Computer Society.
- [25] Bernardo Lula Jr and Aécio Ferreira de Lima, 2006. Disponível em: [http : //ppginf.ucpel.tche.br/weciq/CD/Mini - Cursos/BernardoLulaJr/mini - curso - bernardo - lula.pdf](http://ppginf.ucpel.tche.br/weciq/CD/Mini - Cursos/BernardoLulaJr/mini - curso - bernardo - lula.pdf). Acessado em 16/09/2010.
- [26] Suenne Riguetto Machado. Implementação da transformada de fourier quântica em núcleos quadrupolares. Master's thesis, Centro Brasileiro de Pesquisas Físicas - Rio de Janeiro, 2008.

- [27] Miranda Marquit. Ion-trap quantum computing., Maio 2009. Disponível em [http : //www.physorg.com/news161348276.html](http://www.physorg.com/news161348276.html). Acessado em 16/08/2011.
- [28] Elmar Melcher. Systemverilog. curso do brazil-ip., 2011. Disponível em: [http : //www.scribd.com/doc/77110795/SVerilog](http://www.scribd.com/doc/77110795/SVerilog). Acessado em 11/03/2011.
- [29] D. Michael Miller, Dmitri Maslov, and Gerhard W. Dueck. A transformation based algorithm for reversible logic synthesis. In *Proceedings of the 40th annual Design Automation Conference, DAC '03*, pages 318–323, New York, NY, USA, 2003. ACM.
- [30] E. Monteiro, D Jaccottety, E. Costa, and R. Reiser. Modelagem de circuitos quânticos em vhdl. In *Anais do CNMAC*, volume 2, pages 229–230, 2009.
- [31] Heron Aragão Monteiro and Bernardo Lula Jr. Uma alternativa de armazenamento e manipulação de estruturas quânticas para simuladores universais. In Carlile Lavor, Renato Portugal, Franklin de Lima Marquezino, and Demerson Nunes Gonçalves, editors, *Anais / WECIQ 2010*, pages 137–146. Laboratório Nacional de Computação Científica, Outubro 2010.
- [32] Carlos Morimoto. O verdadeiro sentido da lei de moore, 11 2009. Disponível em: [http : //www.gdhpress.com.br/blog/moore/](http://www.gdhpress.com.br/blog/moore/). Acessado em 04/02/2010.
- [33] Michele Mosca, 2008. Disponível em: [http : //arxiv.org/abs/0808.0369v1](http://arxiv.org/abs/0808.0369v1). Acessado em 12/08/2010.
- [34] Valéria S. Motta, Luiz Mariano Carvalho, and Nelson Maculan, 2010. Disponível em: [http : //www.sbmac.org.br/eventos/cnmac/cd_xxviii_cnmac/resumosextendidos/valeria_motta_S](http://www.sbmac.org.br/eventos/cnmac/cd_xxviii_cnmac/resumosextendidos/valeria_motta_S). Acessado em 22/10/2011.
- [35] Jean P. Nicolle, 2010. Disponível em: [http : //www.fpga4fun.com/](http://www.fpga4fun.com/). Acessado em 11/04/2011.
- [36] Michael A. Nielsen and Issac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

- [37] Kevin M. Obenland and Alvin M. Despain. A parallel quantum computer simulator, 1998. Disponível em [http : //arxiv.org/PS_cache/quant - ph/pdf/9804/9804039v1.pdf](http://arxiv.org/PS_cache/quant-ph/pdf/9804/9804039v1.pdf). Acessado em 22/2/2010.
- [38] I.S. Oliveira, J.D. Bulnes, and R.S. Sarthour. Computação quântica via ressonância magnética nuclear. In *Anais da IV Escola do CBPF. 1 ed.*, volume 1, pages 303–330, 2003.
- [39] Renato Portugal. Uma introdução à computação quântica. In Eliana X.L. Andrade, Rubens Sampaio, and Geraldo Silva, editors, *Notas em Matemática Aplicada*, volume 8, 2005.
- [40] 1998. Disponível em: [http : //tph.tuwien.ac.at/ oemer/qcl.html](http://tph.tuwien.ac.at/oemer/qcl.html). Acessado em 14/06/2011.
- [41] 2003. Disponível em: [http : //www.het.brown.edu/people/andre/qlambda/](http://www.het.brown.edu/people/andre/qlambda/). Acessado em 11/06/2011.
- [42] 1999. Disponível em: [http : //library.wolfram.com/infocenter/MathSource/1893/](http://library.wolfram.com/infocenter/MathSource/1893/). Acessado em 14/06/2011.
- [43] 2006. Disponível em: [http : //sneezy.cs.nott.ac.uk/qml/](http://sneezy.cs.nott.ac.uk/qml/). Acessado em 14/06/2011.
- [44] 2000. Disponível em: [http : //web.archive.org/web/20060116174553/http : //userpages.umbc.edu/ cmccub1/quacs/quacs.html](http://web.archive.org/web/20060116174553/http://userpages.umbc.edu/cmccub1/quacs/quacs.html). Acessado em 14/06/2011.
- [45] 2011. Disponível em: [http : //homepage.cem.itesm.mx/lgoomez/quantum/index.htm](http://homepage.cem.itesm.mx/lgoomez/quantum/index.htm). Acessado em 11/06/2011.
- [46] 2000. Disponível em: [http : //iaks - www.ira.uka.de/home/matteck/QuaSi/aboutquasi.html](http://iaks-www.ira.uka.de/home/matteck/QuaSi/aboutquasi.html). Acessado em 11/06/2011.
- [47] 2009. Disponível em: [http : //vlsicad.eecs.umich.edu/Quantum/qp/index.html](http://vlsicad.eecs.umich.edu/Quantum/qp/index.html). Acessado em 11/06/2011.
- [48] Reversible logic synthesis benchmarks, 2009. Disponível em: [http : //webhome.cs.uvic.ca/ dmaslov/](http://webhome.cs.uvic.ca/dmaslov/). Acessado em 11/06/2011.

- [49] Bart Rylander, Terry Soule, James Foster, and Jim Alves-Foss, 2000. Disponível em: [http : //people.ibest.uidaho.edu/ foster/Papers/40147.pdf](http://people.ibest.uidaho.edu/foster/Papers/40147.pdf). Acessado em 05/11/2010.
- [50] Wissam Abdel Samad, Roy Ghandour, and Mohamad Nabil Hajj Chehade. Memory efficient quantum circuit simulator based on linked list architecture. *Arxiv preprint quantph0511079*, (1 0):8, 2005.
- [51] Paul Savory and Gerald Mackulak. The science of simulation modeling. In *InternationalConferenceonSimulationinEngineeringEducation(ICSEE94)*, volume 26, pages 115–119, 1994.
- [52] Andrew M. Steane. How to build a 300 bit, 1 giga-operation quantum computer. In *Quantum Information Computation*, volume 7, pages 171–183. Rinton Press, Incorporated0, Dezembro 2004.
- [53] 2011. Disponível em: [http : //www.synopsys.com/tools/implementation/fpgaimplementation](http://www.synopsys.com/tools/implementation/fpgaimplementation). Acessado em 20/08/2011.
- [54] Welcome to systemverilog., 2011. Disponível em: [http : //www.systemverilog.in/overview.php](http://www.systemverilog.in/overview.php). Acessado em 11/03/2011.
- [55] Systemverilog overview, 2011. Disponível em: [http : //www.systemverilog.org/](http://www.systemverilog.org/). Acessado em 15/03/2011.
- [56] George F. Viamontes, Manoj Rajagopalan, Igor L. Markov, and John P. Hayes. Gate-level simulation of quantum circuits. In *Los Alamos Quantum Physics Archive*, Aug. 2002, pages 295–301, 2003.
- [57] Field-programmable gate array, 2010. Disponível em: [http : //pt.wikipedia.org/wiki/Field – programmable_gate_array](http://pt.wikipedia.org/wiki/Field_programmable_gate_array). Acessado em 05/04/2011.
- [58] Hardware description language, 2010. Disponível em: [http : //en.wikipedia.org/wiki/Hardware_description_language](http://en.wikipedia.org/wiki/Hardware_description_language). Acessado em 11/04/2011.

-
- [59] 2009. Disponível em: [http : //pt.wikipedia.org/wiki/Lei_de_Moore](http://pt.wikipedia.org/wiki/Lei_de_Moore). Acessado em 04/02/2010.
- [60] 2010. Disponível em: [http : //pt.wikipedia.org/wiki/RichardFeynman](http://pt.wikipedia.org/wiki/RichardFeynman). Acessado em 04/02/2010.
- [61] Vhdl, 2010. Disponível em: [http : //pt.wikipedia.org/wiki/VHDL](http://pt.wikipedia.org/wiki/VHDL). Acessado em 11/03/2011.

Apêndice A

Mudança de Proposta de Trabalho

Inicialmente o trabalho tinha como proposta¹ a emulação, em placas FPGA, de circuitos quânticos mais flexíveis, onde os argumentos e parâmetros de execução pudessem ser separados da lógica de processamento dos mesmos, como acontece com todos os circuitos, onde o oráculo, que implementa uma função a ser tratada ou um argumento a ser buscado, está inserido no próprio circuito, havendo a necessidade do mesmo ser re-implementado a cada mudança dessa função ou argumento.

Para que esse objetivo fosse alcançado seria necessário que os circuitos quânticos fossem alterados, através da inclusão de q-bits auxiliares.

Porém, para que seja feita a inclusão de um q-bit a um circuito, é necessário o dobro da capacidade de armazenamento das estruturas quânticas, o que desencoraja tal inclusão.

Para que esse efeito fosse minimizado, foi proposta [31] a utilização de uma estrutura alternativa de armazenamento e manipulação de estruturas quânticas, proposta essa submetida, aceita apresentada e publicada nos anais do III Workshop-Escola de Computação e Informação Quântica, em outubro de 2010.

A intenção seria a utilização da menor quantidade de recursos computacionais possível, viabilizando a inclusão de q-bits auxiliares ser causar aumento significativo na utilização desses recursos.

Com a implementação dessa proposta ficaria viável, então, a proposição de circuitos quânticos mais flexíveis, através da inclusão de q-bits auxiliares, sem resultar num aumento

¹Essa proposta foi apresentada e aceita, em novembro de 2010, de acordo com o regimento do Programa de Pós-Graduação em Ciências da Computação da Universidade Federal de Campina Grande-PB.

dos recursos computacionais necessários.

A ideia central da proposta era a de armazenar, em memória, apenas as amplitudes dos estados quânticos que fossem diferentes de zero, alocação essa que seria feita em tempo de execução, à medida que esses estados fossem sendo requisitados.

Porém, quando foi iniciada a codificação da solução, em placa FPGA, com a linguagem de descrição de hardware SystemVerilog, utilizando a ferramenta Quartus, foi constatado que não seria possível fazer essa alocação de memória em tempo de execução, apesar da linguagem fornecer comandos para tal.

Dessa forma, não poderia ser utilizada a técnica de alocação de memória, e consequentemente, dos estados quânticos, em tempo de execução, à medida que os mesmos fossem sendo alcançados, impossibilitando a inclusão de q-bits extras sem aumento de recursos computacionais. Assim, diante da impossibilidade de continuidade da proposta defendida e desejando explorar a plataforma escolhida, foi dado outro rumo ao trabalho.

Apêndice B

Circuito de Grover para 3 q-bits

- 0 -> Inicia_estados
- 1 -> Hadamard1
- 2 -> Hadamard2(4)
- 3 -> Hadamard1
- 4 -> Hadamard2(3)
- 5 -> Hadamard1
- 6 -> Hadamard2(2)
- 7 -> Hadamard1
- 8 -> Hadamard2(1)
- > Início da primeira iteração de Grover
- 9 -> Not(4)
- 10 -> Not(3)
- 11 -> Not(2)
- 12 -> CCNot_m(1)
- 13 -> Not(2)
- 14 -> Not(3)
- 15 -> Not(4)
- 16 -> Hadamard1
- 17 -> Hadamard2(4)
- 18 -> Hadamard1
- 19 -> Hadamard2(3)

20 -> Hadamard1
21 -> Hadamard2(2)
22 -> Not(4)
23 -> Not(3)
24 -> Not(2)
25 -> Porta_Z_m(2)
26 -> Not(2)
27 -> Not(3)
28 -> Not(4)
29 -> Hadamard1
30 -> Hadamard2(4)
31 -> Hadamard1
32 -> Hadamard2(3)
33 -> Hadamard1
34 -> Hadamard2(2)
-> Início da segunda iteração de Grover
35 -> Not(4)
36 -> Not(3)
37 -> Not(2)
38 -> CCNot_m(1)
39 -> Not(2)
40 -> Not(3)
41 -> Not(4)
42 -> Hadamard1
43 -> Hadamard2(4)
44 -> Hadamard1
45 -> Hadamard2(3)
46 -> Hadamard1
47 -> Hadamard2(2)
48 -> Not(4)
49 -> Not(3)

50 -> Not(2)
51 -> Porta_Z_m(2)
52 -> Not(2)
53 -> Not(3)
54 -> Not(4)
55 -> Hadamard1
56 -> Hadamard2(4)
57 -> Hadamard1
58 -> Hadamard2(3)
59 -> Hadamard1
60 -> Hadamard2(2)
-> Fim do Iterador de Grover

Apêndice C

Circuito de Grover para 7 q-bits

Segue, abaixo, o código completo da solução implementada para o algoritmo de Grover com 7 q-bits.

```
//Grover 7 qubits
// 6 iteracoes
// precisao 8
// -----
// Copyright (c) 2007 by Terasic Technologies Inc.
// -----
// Permission:
// Terasic grants permission to use and modify this code for use
// in synthesis for all Terasic Development Boards and Altera
// Development Kits made by Terasic. Other use of this code,
// including the selling ,duplication , or modification of any
// portion is strictly prohibited.
// Disclaimer:
// This VHDL/Verilog or C/C++ source code is intended as a design
// reference which illustrates how these types of functions can
// be implemented. It is the user's responsibility to verify their
// design for consistency and functionality through the use of
// formal verification methods. Terasic provides no warranty
```

```
// regarding the use or functionality of this code.
// -----
//           Terasic Technologies Inc
//           356 Fu-Shin E. Rd Sec. 1. JhuBei City ,
//           HsinChu County , Taiwan
//           302
//           web: http://www.terasic.com/
//           email: support@terasic.com
// -----
// Major Functions:      DE2_70 TOP LEVEL
// -----
// Revision History :
// -----
// Ver  :| Author      :| Mod. Date :| Changes Made:
// V1.0 :| Johnny FAN  :| 07/07/09  :| Initial Revision
// V1.1 :| Johnny FAN  :| 07/07/09  :| 1.change fpga device
//           from ep2c70896c8 to ep2c70896c6
//           2.change module name
//           from DE2P_TOP to DE2_70_TOP
// V1.2 :| Johnny FAN  :| 07/07/09  :| 1. change for pcb v1.1
//           2. modify PS2 interface
//           i/o
//           3. modify GPIO,VGA,
//           and UART interface
//           pin assignment
//           4. change GPIO_CLKOUT
//           from output logic to
//           inout
// V1.21 :| Johnny FAN  :| 07/07/09  :| 1.remove ssram address
//           bus oSRAM_A[19]and
//           oSRAM_A[20]
//           2.remove flash address
//           bus oFLASH_A[25:22]
```



```

// V2.0 :| Perly HU      :| 08/12/2010:| Revision to Quartus
//
//                                     II 10.0
// -----
parameter int precisao = 8; / qtde de casas decimais da amplitude
parameter int qtde_qbits = 7; // numero de qubits do circuito
module DE2_70_TOP
    (
//=====
// PORT declarations
//=====
//////////              Clock Input logic              //////////
input logic    iCLK_28,           // 28.63636 MHz
input logic    iCLK_50,           // 50 MHz
input logic    iCLK_50_2,         // 50 MHz
input logic    iCLK_50_3,         // 50 MHz
input logic    iCLK_50_4,         // 50 MHz
input logic    iEXT_CLOCK,        // External Clock
//////////              Push Button              //////////
input logic    [3:0] iKEY,         // Pushbutton [3:0]
//////////              DPDT Switch              //////////
input logic    [17:0]iSW,         // Toggle Switch [17:0]
//////////              7-SEG Dispaly              //////////
output logic   [6:0]  oHEX0_D,     // Seven Segment Digit 0
output logic   oHEX0_DP,          // Seven Segment Digit 0
//                                     decimal point
output logic   [6:0]  oHEX1_D,     // Seven Segment Digit 1
output logic   oHEX1_DP,          // Seven Segment Digit 1
//                                     decimal point
output logic   [6:0]  oHEX2_D,     // Seven Segment Digit 2
output logic   oHEX2_DP,          // Seven Segment Digit 2
//                                     decimal point
output logic   [6:0]  oHEX3_D,     // Seven Segment Digit 3
output logic   oHEX3_DP,          // Seven Segment Digit 3

```



```

output logic          oDRAM1_WE_N,      // Write Enable
output logic          oDRAM0_CAS_N,     // Column Address Strobe
output logic          oDRAM1_CAS_N,     // Column Address Strobe
output logic          oDRAM0_RAS_N,     // Row Address Strobe
output logic          oDRAM1_RAS_N,     // Row Address Strobe
output logic          oDRAM0_CS_N,      // Chip Select
output logic          oDRAM1_CS_N,      // Chip Select
output logic          [1:0] oDRAM0_BA,  // Bank Address
output logic          [1:0] oDRAM1_BA,  // Bank Address
output logic          oDRAM0_CLK,       // Clock
output logic          oDRAM1_CLK,       // Clock
output logic          oDRAM0_CKE,       // Clock Enable
output logic          oDRAM1_CKE,       // Clock Enable
//////////////////// SDRAM Interface //////////////////////////////////////
//////////////////// Flash Interface //////////////////////////////////////
input                [14:0]FLASH_DQ,    // Data bus 15 Bits
//                                     (0 to 14)
input                FLASH_DQ15_AM1,    // Data bus Bit 15
//                                     or Address A-1
output logic          [21:0]oFLASH_A,    // Address bus 22 Bits
output logic          oFLASH_WE_N,       // Write Enable
output logic          oFLASH_RST_N,      // Reset
output logic          oFLASH_WP_N,       // Write Protect
//                                     /Programming Acceleration
input logic           iFLASH_RY_N,       // Ready/Busy output logic
output logic          oFLASH_BYTE_N,     // Byte/Word Mode
//                                     Configuration
output logic          oFLASH_OE_N,       // Output logic Enable
output logic          oFLASH_CE_N,       // Chip Enable
//////////////////// SRAM Interface //////////////////////////////////////
input                [31:0]SRAM_DQ,     // Data Bus 32 Bits
input                [3:0] SRAM_DPA,     // Parity Data Bus
output logic          [18:0]oSRAM_A,     // Address bus 21 Bits

```

```

output logic          oSRAM_ADSC_N,      //
Controller Address Status
output logic          oSRAM_ADSP_N,      //
Processor Address Status
output logic          oSRAM_ADV_N,        // Burst Address Advance
output logic          [3:0] oSRAM_BE_N,   // Byte Write Enable
output logic          oSRAM_CEI_N,        // Chip Enable
output logic          oSRAM_CE2,          // Chip Enable
output logic          oSRAM_CE3_N,        // Chip Enable
output logic          oSRAM_CLK,          // Clock
output logic          oSRAM_GW_N,         // Global Write Enable
output logic          oSRAM_OE_N,         // Output logic Enable
output logic          oSRAM_WE_N,         // Write Enable
//////////////////// ISP1362 Interface //////////////////////////////////////
input                 [15:0]OTG_D,        // Data bus 16 Bits
output logic          [1:0] oOTG_A,        // Address 2 Bits
output logic          oOTG_CS_N,          // Chip Select
output logic          oOTG_OE_N,          // Read
output logic          oOTG_WE_N,          // Write
output logic          oOTG_RESET_N,       // Reset
input                 OTG_FSPEED,         // USB Full Speed,
//                                         0 = Enable, Z = Disable
input                 OTG_LSPEED,         // USB Low Speed,
//                                         0 = Enable, Z = Disable
input logic           iOTG_INT0,          // Interrupt 0
input logic           iOTG_INT1,          // Interrupt 1
input logic           iOTG_DREQ0,         // DMA Request 0
input logic           iOTG_DREQ1,         // DMA Request 1
output logic          oOTG_DACK0_N,       // DMA Acknowledge 0
output logic          oOTG_DACK1_N,       // DMA Acknowledge 1
//////////////////// LCD Module 16X2 //////////////////////////////////////
input                 [7:0] LCD_D,         // Data bus 8 bits
output logic          oLCD_ON,            // Power ON/OFF

```

```

output logic      oLCD_BLON,      // Back Light ON/OFF
output logic      oLCD_RW,        // Read/Write Select ,
//                                     0 = Write , 1 = Read
output logic      oLCD_EN,        // Enable
output logic      oLCD_RS,        // Command/Data Select ,
//                                     0 = Command, 1 = Data
//////////////////// SD Card Interface //////////////////////////////////////
inout             SD_DAT,         // Data
inout             SD_DAT3,        // Data 3
inout             SD_CMD,         // Command Signal
output logic      oSD_CLK,        // Clock
//////////////////// I2C //////////////////////////////////////
inout             I2C_SDAT,       // Data
output logic      oI2C_SCLK,      // Clock
//////////////////// PS2 //////////////////////////////////////
inout             PS2_KBDAT,      // Keyboard Data
inout             PS2_KBCLK,      // Keyboard Clock
inout             PS2_MS DAT,     // Mouse Data
inout             PS2_MSCLK,     // Mouse Clock
//////////////////// VGA //////////////////////////////////////
output logic      oVGA_CLOCK,     // Clock
output logic      oVGA_HS,        // H_SYNC
output logic      oVGA_VS,        // V_SYNC
output logic      oVGA_BLANK_N,   // BLANK
output logic      oVGA_SYNC_N,    // SYNC
output logic      [9:0] oVGA_R,    // Red [9:0]
output logic      [9:0] oVGA_G,    // Green [9:0]
output logic      [9:0] oVGA_B,    // Blue [9:0]
//////////////////// Ethernet Interface //////////////////////////////////////
inout             [15:0]ENET_D,    // DATA bus 16Bits
output logic      oENET_CMD,      // Command/Data Select ,
//                                     0 = Command, 1 = Data
output logic      oENET_CS_N,     // Chip Select

```

```

output logic          oENET_IOW_N,      // Write
output logic          oENET_IOR_N,      // Read
output logic          oENET_RESET_N,    // Reset
input logic           iENET_INT,        // Interrupt
output logic          oENET_CLK,        // Clock 25 MHz
//////////////////// Audio CODEC //////////////////////////////////////
inout                 AUD_ADCLRCK,      // ADC LR Clock
input logic           iAUD_ADCDAT,      // ADC Data
inout                 AUD_DACLK,        // DAC LR Clock
output logic          oAUD_DACDAT,      // DAC Data
inout                 AUD_BCLK,         // Bit-Stream Clock
output logic          oAUD_XCK,         // Chip Clock
//////////////////// TV Devoder //////////////////////////////////////
input logic           iTD1_CLK27,      // Line_Lock
//                                     Output logic Clock
input logic           [7:0] iTD1_D,     // Data bus 8 bits
input logic           iTD1_HS,         // H_SYNC
input logic           iTD1_VS,         // V_SYNC
output logic          oTD1_RESET_N,     // Reset
input logic           iTD2_CLK27,      // Line_Lock
//                                     Output logic Clock
input logic           [7:0] iTD2_D,     // Data bus 8 bits
input logic           iTD2_HS,         // H_SYNC
input logic           iTD2_VS,         // V_SYNC
output logic          oTD2_RESET_N,     // Reset
//////////////////// GPIO //////////////////////////////////////
inout                 [31:0]GPIO_0,     // GI/O
input logic           GPIO_CLKIN_N0,    // Clock Input logic 0
input logic           GPIO_CLKIN_P0,    // Clock Input logic 1
inout                 GPIO_CLKOUT_N0,   // GClock Output logic 0
inout                 GPIO_CLKOUT_P0,   // Clock Output logic 1
inout                 [31:0]GPIO_1,     // I/O
input logic           GPIO_CLKIN_N1,    // Clock Input logic 0

```

```

input logic          GPIO_CLKIN_P1,    // Clock Input logic 1
inout                GPIO_CLKOUT_N1,   // Clock Output logic 0
inout                GPIO_CLKOUT_P1   // Clock Output logic 1
    );

logic signed [precisao+1:0] amp_real [0:(1<<qtde_qbits)-1];
logic signed [precisao+1:0] amp_imag [0:(1<<qtde_qbits)-1];
logic signed [precisao+1:0] inv_raiz2; // = 12b'010110101000; // 0.707106781
logic signed [31:0] max_inv_raiz2;
bit [qtde_qbits:0] origem, destino;

//In a list of dimensions, the right-most one varies most
//rapidly, as in C. However a packed dimension varies more
//rapidly than an unpacked one.
//=====
// REG/WIRE declarations
//=====
wire reset_n = iKEY[0];
int m, n;
int porta_logica = 0;
int iteracao = 0;
int repeticao_iteracao = 44; // alterar para cada novo circuito
int qtde_iteracoes = 6;    // alterar para cada novo circuito
logic my_clock;
assign estados = (1 << qtde_qbits);
assign max_inv_raiz2 = 32'b00101101010000010011110011001101;
assign inv_raiz2 = max_inv_raiz2 >>> (30 - precisao);
logic signed [2*precisao+1:0] temp; // para armazenar resultado multiplicacao
//=====
// Structural coding
//=====
always @(negedge iKEY[1] or
        negedge iKEY[2])
    if (~iKEY[1])

```

```
        my_clock <= 0;
    else my_clock <= 1;

always @(negedge reset_n or posedge iCLK_50)
    if (!reset_n)
        porta_logica <= 0;
    else
        porta_logica <= porta_logica + 1;

always_ff @ (posedge iCLK_50)
    if (porta_logica == 0)
        Inicia_estados; else
        if ((porta_logica == 1) ||
            (porta_logica == 3) ||
            (porta_logica == 5) ||
            (porta_logica == 7) ||
            (porta_logica == 9) ||
            (porta_logica == 11) ||
            (porta_logica == 13) ||
            (porta_logica == (22 + (iteracao * repeticao_iteracao))) ||
            (porta_logica == (24 + (iteracao * repeticao_iteracao))) ||
            (porta_logica == (26 + (iteracao * repeticao_iteracao))) ||
            (porta_logica == (28 + (iteracao * repeticao_iteracao))) ||
            (porta_logica == (30 + (iteracao * repeticao_iteracao))) ||
            (porta_logica == (32 + (iteracao * repeticao_iteracao))) ||
            (porta_logica == (47 + (iteracao * repeticao_iteracao))) ||
            (porta_logica == (49 + (iteracao * repeticao_iteracao))) ||
            (porta_logica == (51 + (iteracao * repeticao_iteracao))) ||
            (porta_logica == (53 + (iteracao * repeticao_iteracao))) ||
            (porta_logica == (55 + (iteracao * repeticao_iteracao))) ||
            (porta_logica == (57 + (iteracao * repeticao_iteracao)))
        )
        Hadamard_1; else
```



```
if ((porta_logica == 2) ||
    (porta_logica == (23 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (48 + (iteracao * repeticao_iteracao)))
)
    Hadamard_2(7); else
if ((porta_logica == 4) ||
    (porta_logica == (25 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (50 + (iteracao * repeticao_iteracao)))
)
    Hadamard_2(6); else
if ((porta_logica == 6) ||
    (porta_logica == (27 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (52 + (iteracao * repeticao_iteracao)))
)
    Hadamard_2(5); else
if ((porta_logica == 8) ||
    (porta_logica == (29 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (54 + (iteracao * repeticao_iteracao)))
)
    Hadamard_2(4); else
if ((porta_logica == 10) ||
    (porta_logica == (31 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (56 + (iteracao * repeticao_iteracao)))
)
    Hadamard_2(3); else
if ((porta_logica == 12) ||
    (porta_logica == (33 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (58 + (iteracao * repeticao_iteracao)))
)
    // ^
begin
    // :
    Hadamard_2(2); //ultima execucao do primeiro iterador
    if ((porta_logica == (58 + (iteracao * repeticao_iteracao)))
        && (iteracao < qtde_iteracoes))
```

```
        iteracao ++;
    end
    else
if (porta_logica == 14)
    Hadamard_2(1); else
if ((porta_logica == (34 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (41 + (iteracao * repeticao_iteracao))))
    )
    Not(7); else
if ((porta_logica == (15 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (19 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (35 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (42 + (iteracao * repeticao_iteracao))))
    )
    Not(6); else
if ((porta_logica == (36 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (43 + (iteracao * repeticao_iteracao))))
    )
    Not(5); else
if ((porta_logica == (16 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (20 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (37 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (44 + (iteracao * repeticao_iteracao))))
    )
    Not(4); else
if ((porta_logica == (38 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (45 + (iteracao * repeticao_iteracao))))
    )
    Not(3); else
if ((porta_logica == (17 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (21 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (39 + (iteracao * repeticao_iteracao))) ||
    (porta_logica == (46 + (iteracao * repeticao_iteracao))))
    )
```

```
        )
        Not(2); else
    if ((porta_logica == (18 + (iteracao * repeticao_iteracao)))
        )
        CCNot_m(1); else
    if ((porta_logica == (40 + (iteracao * repeticao_iteracao)))
        )
        Porta_Z_m(2);

task Inicia_estados;
// synthesis loop_limit 132000
for (int i=0; i < 1 << qtde_qbits; i ++)
begin
    amp_real[i] <= '0;
    amp_imag[i] <= '0;
end;
amp_real[1] <= 010<<<(precisao -1);
amp_imag[1] <= '0;
endtask // Inicia_estados

task Hadamard_1;
// primeiro passo hadamard multiplica todas as amplitudes por 1/sqtr(2)
// synthesis loop_limit 8000
for (origem = 0; origem < (1<<qtde_qbits); origem++)
    if ((amp_real[origem] != 0) || (amp_imag[origem] != 0))
        begin
            amp_real[origem] <= Multiplica(amp_real[origem], inv_raiz2);
            amp_imag[origem] <= Multiplica(amp_imag[origem], inv_raiz2);
        end;
endtask

task Hadamard_2 (input int posicao);
```

```

// soma as amplitudes ja multiplicadas de acordo com o qubit 0 ou 1
// synthesis loop_limit 8000
for (origem = 0; origem < (1<<qtde_qbits); origem++)
begin
    destino = origem;
    destino[posicao-1] = ~destino[posicao-1];
    if (origem[posicao-1]) begin // testa se 1
        amp_real[origem] <= amp_real[destino] - amp_real[origem]; //
        amp_imag[origem] <= amp_imag[destino] - amp_imag[origem]; //
    end
    else begin
        amp_real[origem] <= amp_real[origem] + amp_real[destino]; // se bit 0
        amp_imag[origem] <= amp_imag[origem] + amp_imag[destino]; // se bit 0
    end
end;
endtask

function logic signed [precisao+1:0] Multiplica
    (logic signed [precisao+1:0] a, b);

    temp = (a * b);
// **** para arredondamento
    temp = temp >>> (precisao - 1);
    if (temp[0]) // se penultimo bit igual a um
        if (temp > 0)
            temp = temp + 1; // para arredondamento
        else
            temp = temp - 1;
    Multiplica = temp >>> 1;
endfunction

task Not (input int posicao);
    origem = 0;
// synthesis loop_limit 8000

```

```

for (m = 1; m <= ((1<<qtde_qbits)/(1<<posicao)); m++)
begin
// synthesis loop_limit 8000
for (n = 1; n <= (1<<(posicao - 1)); n ++)
begin
destino = origem;
destino[posicao - 1] = ~destino[posicao - 1];
amp_real[origem] <= amp_real[destino];
amp_real[destino] <= amp_real[origem];
amp_imag[origem] <= amp_imag[destino];
amp_imag[destino] <= amp_imag[origem];
origem = origem + 1;
end; // for i
origem = origem + n - 1;
end; // for k
endtask // NOT

task CCNot_m (input int posicao);
origem = 0;
// synthesis loop_limit 8000
for (m = 1; m <= ((1<<qtde_qbits)/(1<<posicao)); m++)
begin
// synthesis loop_limit 8000
for (n = 1; n <= (1<<(posicao - 1)); n ++)
begin
if (origem[2] && origem[3] && origem[4] &&
origem[5] && origem[6] && origem[7]
)
begin
destino = origem;
destino[posicao - 1] = ~destino[posicao - 1];
amp_real[origem] <= amp_real[destino];
amp_real[destino] <= amp_real[origem];

```

```
        amp_imag[origem] <= amp_imag[destino];
        amp_imag[destino] <= amp_imag[origem];
        end; // if
        origem = origem + 1;
    end; // for n
    origem = origem + n - 1;
end; // for m
endtask // CCNot_m

task Porta_Z_m (input int posicao);
// synthesis loop_limit 8000
    for (origem = 0; origem < (1<<qtde_qbits); origem ++)
        if (origem[3] && origem[4] && origem[5] &&
            origem[6] && origem[7]
        )
            if (origem[posicao - 1]) begin // testa se 1 para inverter amplitud
                amp_real[origem] <= (amp_real[origem] * -1);
                amp_imag[origem] <= (amp_imag[origem] * -1);
            end;
    endtask // PortaZ_m

logic [4:0] j;
always_comb begin
    j <= iSW;
    oLEDR <= amp_real[j];
// oLEDR <= inv_raiz2;
    if (amp_real[j] >= 0)
        begin /// sinal
            oHEX7_D[0] = 1;
            oHEX7_D[1] = 1;
            oHEX7_D[2] = 1;
            oHEX7_D[3] = 1;
            oHEX7_D[4] = 1;
```

```
        oHEX7_D[5] = 1;
        oHEX7_D[6] = 1;
    end
else
    begin
        oHEX7_D[0] = 1;
        oHEX7_D[1] = 1;
        oHEX7_D[2] = 1;
        oHEX7_D[3] = 1;
        oHEX7_D[4] = 1;
        oHEX7_D[5] = 1;
        oHEX7_D[6] = 0;
    end
oHEX7_DP = 1; /// ponto
oHEX6_DP = 0;
oHEX5_DP = 1;
oHEX4_DP = 1;
oHEX3_DP = 1;
oHEX2_DP = 1;
oHEX1_DP = 1;
oHEX0_DP = 1;
if (~amp_real[j][precisao])
    begin
        oHEX6_D[0] = 0;
        oHEX6_D[1] = 0;
        oHEX6_D[2] = 0;
        oHEX6_D[3] = 0;
        oHEX6_D[4] = 0;
        oHEX6_D[5] = 0;
        oHEX6_D[6] = 1;
    end
end
else
    begin
```

```
        oHEX6_D[0] = 1;
        oHEX6_D[1] = 0;
        oHEX6_D[2] = 0;
        oHEX6_D[3] = 1;
        oHEX6_D[4] = 1;
        oHEX6_D[5] = 1;
        oHEX6_D[6] = 1;
    end
if (~amp_real[j][precisao -1])
    begin
        oHEX5_D[0] = 0;
        oHEX5_D[1] = 0;
        oHEX5_D[2] = 0;
        oHEX5_D[3] = 0;
        oHEX5_D[4] = 0;
        oHEX5_D[5] = 0;
        oHEX5_D[6] = 1;
    end
else
    begin
        oHEX5_D[0] = 1;
        oHEX5_D[1] = 0;
        oHEX5_D[2] = 0;
        oHEX5_D[3] = 1;
        oHEX5_D[4] = 1;
        oHEX5_D[5] = 1;
        oHEX5_D[6] = 1;
    end
end
if (~amp_real[j][precisao -2])
    begin
        oHEX4_D[0] = 0;
        oHEX4_D[1] = 0;
        oHEX4_D[2] = 0;
```



```
        oHEX4_D[3] = 0;
        oHEX4_D[4] = 0;
        oHEX4_D[5] = 0;
        oHEX4_D[6] = 1;
    end
else
    begin
        oHEX4_D[0] = 1;
        oHEX4_D[1] = 0;
        oHEX4_D[2] = 0;
        oHEX4_D[3] = 1;
        oHEX4_D[4] = 1;
        oHEX4_D[5] = 1;
        oHEX4_D[6] = 1;
    end
if (~amp_real[j][precisao - 3])
    begin
        oHEX3_D[0] = 0;
        oHEX3_D[1] = 0;
        oHEX3_D[2] = 0;
        oHEX3_D[3] = 0;
        oHEX3_D[4] = 0;
        oHEX3_D[5] = 0;
        oHEX3_D[6] = 1;
    end
else
    begin
        oHEX3_D[0] = 1;
        oHEX3_D[1] = 0;
        oHEX3_D[2] = 0;
        oHEX3_D[3] = 1;
        oHEX3_D[4] = 1;
        oHEX3_D[5] = 1;
```

```
        oHEX3_D[6] = 1;
    end
if (~amp_real[j][precisao -4])
    begin
        oHEX2_D[0] = 0;
        oHEX2_D[1] = 0;
        oHEX2_D[2] = 0;
        oHEX2_D[3] = 0;
        oHEX2_D[4] = 0;
        oHEX2_D[5] = 0;
        oHEX2_D[6] = 1;
    end
else
    begin
        oHEX2_D[0] = 1;
        oHEX2_D[1] = 0;
        oHEX2_D[2] = 0;
        oHEX2_D[3] = 1;
        oHEX2_D[4] = 1;
        oHEX2_D[5] = 1;
        oHEX2_D[6] = 1;
    end
if (~amp_real[j][precisao -5])
    begin
        oHEX1_D[0] = 0;
        oHEX1_D[1] = 0;
        oHEX1_D[2] = 0;
        oHEX1_D[3] = 0;
        oHEX1_D[4] = 0;
        oHEX1_D[5] = 0;
        oHEX1_D[6] = 1;
    end
else
```

```
begin
    oHEX1_D[0] = 1;
    oHEX1_D[1] = 0;
    oHEX1_D[2] = 0;
    oHEX1_D[3] = 1;
    oHEX1_D[4] = 1;
    oHEX1_D[5] = 1;
    oHEX1_D[6] = 1;
end
if (~amp_real[j][precisao - 6])
    begin
        oHEX0_D[0] = 0;
        oHEX0_D[1] = 0;
        oHEX0_D[2] = 0;
        oHEX0_D[3] = 0;
        oHEX0_D[4] = 0;
        oHEX0_D[5] = 0;
        oHEX0_D[6] = 1;
    end
else
    begin
        oHEX0_D[0] = 1;
        oHEX0_D[1] = 0;
        oHEX0_D[2] = 0;
        oHEX0_D[3] = 1;
        oHEX0_D[4] = 1;
        oHEX0_D[5] = 1;
        oHEX0_D[6] = 1;
    end
end
oLEDG <= porta_logica;
end
endmodule
```
