Universidade Federal de Mato Grosso – UFMT
Campus Universitário de Várzea Grande
Faculdade de Engenharia
Engenharia de Computação

Filipe Chagas Ferraz

DLQ - um projeto de linguagem de programação declarativa de alto nível de abstração para computação quântica

Várzea Grande
2023

Filipe Chagas Ferraz

DLQ - um projeto de linguagem de programação declarativa de alto nível de abstração para computação quântica

Trabalho de Conclusão de Curso apresentado à Faculdade de Engenharia como parte dos requisitos para a obtenção do título de Bacharel em Engenharia de Computação.
Orientador: Ronaldo Luiz Alonso
Coorientador: Frank Eduardo da Silva Steinhoff

Várzea Grande
2023

Resumo

Chagas, Ferraz. **DLQ - um projeto de linguagem de programação declarativa de alto nível de abstração para computação quântica**. 134 p. Trabalho de Conclusão de Curso – Engenharia de Computação, Faculdade de Engenharia, Universidade Federal de Mato Grosso, Brasil, 2023.

Neste trabalho de conclusão de curso, desenvolveu-se uma linguagem de programação declarativa, de alto nível de abstração e de fácil aprendizagem, para a programação de computadores quânticos. A linguagem, batizada como DLQ (*Declarative Language for Quantum*), tem como elementos principais: as variáveis numéricas, os operadores aritméticos, lógicos e relacionais, e por fim, as declarações terminadoras. Ao longo deste trabalho, os circuitos quânticos correspondentes a cada um desses elementos, a gramática da linguagem e o processo de compilação dos códigos são explicados de forma detalhada. Exemplos de código na linguagem e considerações finais sobre o trabalho realizado são apresentados nos últimos capítulos.

Palavras-chave: Computação Quântica. Algoritmos Quânticos. Linguagem de Programação. Alto nível de abstração..

Abstract

Chagas, Ferraz. **DLQ - a project of high abstraction level declarative programming language for quantum computing**. 134 p. Undergraduate Dissertation – Computing Engineering, Engineering Faculty, Federal University of Mato Grosso, Brazil, 2023.

In this end-of-course work, a declarative programming language was developed, with a high level of abstraction and easy to learn, for the programming of quantum computers. The language, baptized as DLQ (Declarative Language for Quantum), has as main elements: numerical variables; arithmetic, logical and relational operators; and finally, terminator statements. Throughout this paper, the quantum circuits corresponding to each of these elements, the grammar of the language, and the process of compiling the codes are explained in detail. Code examples in the language and final considerations about the work done are presented in the last chapters.

Keywords: Quantum Computing. Quantum Algorithms. Programming Language. High Abstraction Level..

Lista de ilustrações

Lista de tabelas

Lista de siglas

AST *Abstract Syntax Tree*

DLQ *Declarative Language for Quantum*

GAS *Grover Adaptive Search*

NISQ *Noisy Intermediate-Scale Quantum*

QAOA *Quantum Approximate Optimization Algorithm*

SIMD "*Single Instruction, Multiple Data*"

TCC trabalho de conclusão de curso

TRL *Technology Readiness Level*

Sumário

Capítulo 1

Introdução

1.1 Visão geral

Computação Quântica é uma área em ascensão que visa o desenvolvimento de artifícios baseados em mecânica quântica para a resolução de problemas computacionais. Esta área tem vivenciado um crescimento abrupto de interesse e atividade industrial (PRESKILL, 2018), e grandes empresas como IBM, Google, Microsoft e Intel competem por posições relevantes nela (HUANG et al., 2020).

Embora o "*boom*" de interesse e atividade industrial seja recente, como apontado por Preskill (2018), o início do desenvolvimento desta área não é. No início da década de 1980, Benioff (1982) apresentou um modelo quântico de máquina de Turing. Paralelamente, Feynman (1982) abordou a possibilidade de simular sistemas quânticos utilizando máquinas quânticas hipotéticas, as quais foram chamadas de computadores quânticos ou simuladores quânticos universais pelo mesmo. Posteriormente, Deutsch (1985) apresentou um modelo de computador quântico universal que era a realização do computador quântico hipotético de Feynman, argumentando que a máquina de Turing quântica proposta por Benioff não apresentava vantagens computacionais em relação às máquinas de Turing tradicionais. Na década de 1990, diversos algoritmos para esta classe de computadores foram propostos. Dentre eles, destacam-se o algoritmo de Deutsch e Jozsa (1992), que foi criado para demonstrar que há problemas matemáticos que um computador quântico pode resolver de forma mais eficiente do que um clássico; o algoritmo de Shor (1997), criado para decompor números inteiros grandes em produtos de números primos em tempo polinomial; e o algoritmo de Grover (1996), um rápido algoritmo de busca capaz de resolver problemas NP-completos de satisfatibilidade com aceleração quadrática. Dentre as aplicações introduzidas no século XXI, destacam-se os passeios quânticos (CHILDS et al., 2003), o algoritmo quântico de otimização aproximada (FARHI; GOLDSTONE; GUTMANN, 2014) e o aprendizado de máquina quântico (LLOYD; MOHSENI; REBENTROST, 2013).

A construção de computadores quânticos reais (físicos) não se dá por um único caminho. Há diversas formas de armazenar e processar dados fisicamente em sistemas quânticos. Atualmente, destacam-se três tecnologias utilizadas para construir hardware quântico: supercondução (HUANG et al., 2020), armadilha de íons (HäFFNER; ROOS; BLATT, 2008) e sistemas óticos lineares (KOK et al., 2007).

Os computadores quânticos da era atual, a qual Preskill (2018) chama de *Noisy Intermediate-Scale Quantum* (NISQ), possuem limitações consideráveis. Computadores quânticos NISQ têm poucos *qubits*, o que é análogo a dizer que têm pouca memória; têm altos níveis de ruído, o que significa que os resultados que estes computadores dão têm altos índices de erro; e não são capazes de executar grandes quantidades de operações devido ao ruído. No entanto, diversos estudos sobre correção e mitigação de erros criam perspectivas para uma futura era de computadores quânticos tolerantes a falhas (AHARONOV; BEN-OR, 1997) (PRESKILL, 1997).

Os algoritmos quânticos, que inicialmente eram descritos de forma puramente teórica, atualmente podem ser implementados com o auxílio de ferramentas como Q#, Qiskit, Cirq, Quipper e Scaffold (HEIM et al., 2020). Todas essas ferramentas requerem conhecimentos razoáveis em matemática e física ao programador, já que não provêm níveis de abstração altos ao ponto de "esconder" as bases em mecânica quântica. Neste trabalho de conclusão de curso (TCC), foi desenvolvida uma nova linguagem de programação com o intuito de resolver parcialmente este problema. A linguagem, batizada como *Declarative Language for Quantum* (DLQ), é entendível para programadores que não conhecem mecânica quântica, e pode ser utilizada para resolver problemas de satisfatibilidade envolvendo operações aritméticas, lógicas e relacionais básicas. Esta classe de problemas pode ser descrita pela seguinte sentença: considerando uma função $f : \mathbb{N}^n \mapsto \mathbb{B}$, onde $\mathbb{B}$ é o conjunto booleano ($\mathbb{B} = \{0, 1\}$), encontre uma n-upla $(x_1, x_2, ..., x_n) \in \mathbb{N}^n$ que satisfaça $f(x_1, x_2, ..., x_n) = 1$. A solução desses problemas com computação quântica é concebida através da implementação das operações aritméticas, lógicas e relacionais como circuitos quânticos (que são explicados mais adiante), e da aplicação do algoritmo de busca de Grover (que também é explicado mais adiante).

No estágio atual de desenvolvimento da linguagem, há suporte apenas para algumas (poucas) operações com números naturais, mas esta limitação pode ser superada no futuro implementando as mesmas operações (e mais outras) para números inteiros, usando notação de complemento de 2, e números com ponto flutuante, talvez usando a norma IEEE 754.

1.2 Organização

Esta monografia está organizada da seguinte forma:

- O capítulo 2 aborda conceitos básicos de computação quântica: *qubits*, estados de Bell, portas lógicas quânticas e circuitos quânticos.

- No capítulo 3, as implementações das operações lógicas (*not*, *and* e *or*), aritméticas (soma, subtração, multiplicação e potenciação) e relacionais (=, >, e <) para computadores quânticos são apresentadas. Além disso, o algoritmo de busca de Grover, bem como a forma como este algoritmo é usado para resolver problemas de satisfatibilidade, é explicado.
- No capítulo 4, a implementação do compilador de DLQ – o *DLQpiler* – é abordada.
- No capítulo 5, dois exemplos de aplicação simples da linguagem DLQ, e seus resultados em ambiente simulado, são apresentados.
- No capítulo 6, considerações finais sobre o trabalho realizado são feitas.

Capítulo 2

Introdução à computação quântica

2.1 Por que usar computação quântica?

A principal vantagem que esta classe de computadores apresenta é o **paralelismo quântico**. Uma comparação com a computação clássica pode esclarecer o conceito de paralelismo quântico: computadores clássicos com instruções do tipo "*Single Instruction, Multiple Data*" (SIMD) são capazes de processar múltiplas tuplas de dados com uma única instrução, e eles o fazem dispondo essas tuplas para serem processadas paralelamente por múltiplos co-processadores (LEE, 1995). Computadores quânticos são dotados de um potencial semelhante de processamento paralelo. Na verdade, o poder de processamento paralelo dos computadores quânticos é significativamente maior do que o de computadores clássicos com instruções SIMD, porém, este paralelismo não é concebido nem usado da mesma forma em ambos os tipos de computador.

A diferença mais básica entre os dois tipos de computador (clássico e quântico) está no tipo de informação que estes processam: computadores clássicos processam informação clássica, e computadores quânticos processam informação quântica. Computadores clássicos armazenam informações em sequências de *bits*. Um *bit* é um termo binário, que vale 0 ou 1. Com uma sequência de N *bits*, é possível obter 2^N diferentes combinações de 0s e 1s. Por outro lado, computadores quânticos armazenam informações em sequências de *qubits* (*bits* quânticos), que podem valer 0, 1, ou ambos os valores simultaneamente. Enquanto uma sequência de N *bits* só pode assumir uma das 2^N possíveis combinações de 0s e 1s por vez, uma sequência de N *qubits* pode assumir várias ao mesmo tempo, e este fenômeno é conhecido como **superposição quântica**. Além disso, a forma como o processamento dessas informações é feito também difere em ambas as formas de computação: computadores clássicos processam *bits* utilizando portas lógicas *not*, *and*, *or* e *xor*, enquanto computadores quânticos processam *qubits* utilizado portas lógicas quânticas, que são operadores lineares unitários. Estes operadores são distributivos, o que significa que, quando são aplicados a *qubits* em superposição, todas as informações superpostas são processadas paralelamente (ao mesmo tempo), e esta capacidade de processamento paralelo de informações superpostas é o chamado paralelismo quântico.

Ao longo deste capítulo, conceitos básicos de computação quântica serão explicados de forma detalhada, tomando como base, principalmente, a obra de Nielsen e Chuang (2000), bem como outras que também são citadas.

2.2 *Qubits*

2.2.1 Definição

Como já mencionado na seção 2.1, *qubits* são *bits* quânticos. Fisicamente, *qubits* são concebidos como sistemas quânticos de dois níveis, como fótons polarizados e partículas de spin-1/2 (D'ALESSANDRO; DAHLEH, 2001). Algebricamente, o estado de um *qubit* é definido como:

$$|\psi\rangle = w_0|0\rangle + w_1|1\rangle, \quad \|w_0\|^2 + \|w_1\|^2 = 1 \tag{1}$$

onde $|0\rangle$ e $|1\rangle$ são vetores ortonormais em $\mathbb{C}^2$ que representam os estados independentes do sistema de dois níveis. Esses vetores são:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2}$$

O símbolo $|\cdot\rangle$ é denominado *ket*. Um *ket* é uma notação de Dirac que representa um estado quântico (um vetor ou uma função de onda) associado a um símbolo ou valor numérico. Os vetores $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ e $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ são estados quânticos associados a, respectivamente, 0 e 1, e por este motivo são grafados como $|0\rangle$ e $|1\rangle$. Também existe a notação $\langle\cdot|$, denominada *bra*, que representa a transposta conjugada de $|\cdot\rangle$, e a notação $\langle\cdot|\cdot\rangle$, chamada *braket*, que representa um produto interno de dois vetores pertencentes a um espaço vetorial $\mathbb{C}^n$ ou duas funções pertencentes a um espaço de Hilbert.

Como também já foi dito na seção 2.1, um *qubit* pode assumir os valores 0, 1, ou ambos simultaneamente através do fenômeno da superposição. Na eq. 1, o *qubit* vale 0 quando $(w_0, w_1) = (1, 0)$, ou seja, $|\psi\rangle = |0\rangle$; e vale 1 quando $(w_0, w_1) = (0, 1)$, ou seja, $|\psi\rangle = |1\rangle$. A superposição dos valores 0 e 1 ocorre quando ambos os pesos w_0 e w_1 são diferentes de zero.

Quando o estado de um *qubit* em superposição é aferido experimentalmente, ocorre um fenômeno conhecido como **colapso do estado quântico** ou **colapso da função de onda**, que faz com que o estado se manifeste de forma indeterminística como 0 ou 1. Nessa aferição, a probabilidade do valor 0 ser observado é $\|w_0\|^2$, e a probabilidade do valor 1 ser observado é $\|w_1\|^2$. Desse modo, o colapso do estado $|\psi\rangle$ (eq. 1) pode ser interpretado como uma variável aleatória de alfabeto $\Omega_\psi = \{0, 1\}$ e probabilidades $P_\psi(0) = \|w_0\|^2$ e $P_\psi(1) = \|w_1\|^2$.

Em geral, o estado $|\psi\rangle$ de um *qubit* é uma combinação linear de dois estados independentes $|s_0\rangle \in \mathbb{C}^2$ e $|s_1\rangle \in \mathbb{C}^2$, o que significa que $\{|s_0\rangle, |s_1\rangle\}$ é uma base do espaço de estados de um *qubit* para qualquer par L.I ($|s_0\rangle, |s_1\rangle$). O conjunto $\{|0\rangle, |1\rangle\}$ é conhecido como **base computacional**. Existem infinitas outras bases. Outros dois exemplos de bases são $\{|+\rangle, |-\rangle\}$ e $\{|i\rangle, |-i\rangle\}$, onde:

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \quad (3) \qquad |-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \quad (4)$$

$$|i\rangle = \frac{|0\rangle + i|1\rangle}{\sqrt{2}} \quad (5) \qquad |-i\rangle = \frac{|0\rangle - i|1\rangle}{\sqrt{2}} \quad (6)$$

Para qualquer base $\{|s_0\rangle, |s_1\rangle\}$, os pesos w_0 e w_1 da combinação linear $|\psi\rangle = w_0|s_0\rangle + w_1|s_1\rangle$ podem ser definidos como $w_0 = \langle s_0|\psi\rangle$ e $w_1 = \langle s_1|\psi\rangle$.

2.2.2 Representação gráfica

O estado de um *qubit* pode ser definido em função de dois parâmetros $\theta \in \mathbb{R}$ e $\phi \in \mathbb{R}$ da seguinte forma:

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\sin\left(\frac{\theta}{2}\right)|1\rangle \tag{7}$$

O parâmetro ϕ da eq. 7 é frequentemente chamado de "ângulo de fase" do estado $|1\rangle$, enquanto $e^{i\phi}$ é chamado de "fator de fase" de $|1\rangle$. Este parâmetro não altera as probabilidades $P_\psi(0)$ e $P_\psi(1)$ do colapso de $|\psi\rangle$, mas é importante para muitos algoritmos.

A partir da eq. 7, é possível representar o estado de um *qubit* graficamente em coordenadas esféricas, como mostram as figuras 1 e 2. Esta representação é conhecida como **esfera de Bloch**.

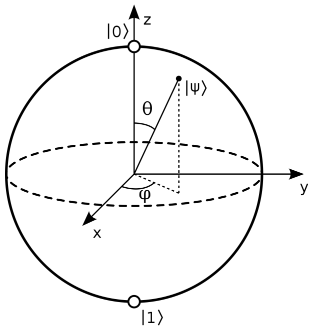

Figura 1 – Esfera de Bloch (Figura extraída de Wikimedia Commons, de autoria de Smite-Meister, sob lincença CC BY-SA 3.0 <https://creativecommons.org/licenses/by-sa/3.0>)

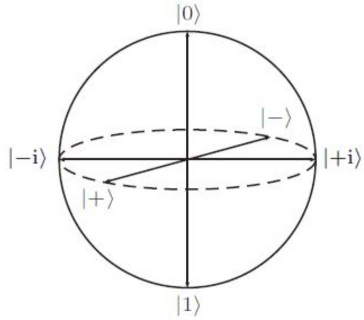

Figura 2 – Estados canônicos na esfera de Bloch (Figura extraída de Wikimedia Commons, de autoria de Kevin Garapo, Mhlambululi Mafu e Francesco Petruccione, sob lincença CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>).

2.3 Registradores quânticos

Em computação clássica, sequências de *bits* são utilizadas para codificar valores de conjuntos discretos não binários. Os conjuntos discretos mais simples de codificar são os subconjuntos dos naturais. Qualquer valor $\boldsymbol{x} \in \{\lambda \in \mathbb{N} : \lambda < 2^n\}$ para $n \in \mathbb{N}$ pode ser codificado como uma sequência de n valores binários $[x_1, x_2, ..., x_n]$ tal que:

$$\boldsymbol{x} = \sum_{i=1}^{n} 2^{i-1} x_i \tag{8}$$

O conjunto de todas as possíveis sequências de n valores binários $[x_1, x_2, ..., x_n]$ é definido como $\underset{1}{\{0,1\}} \times \underset{2}{\{0,1\}} \times ... \times \underset{n}{\{0,1\}} = \{0,1\}^n$, onde $\cdot \times \cdot$ é o produto cartesiano.

Em computação quântica, sequências de *qubits* são chamadas de registradores quânticos. Tal qual os *qubits*, os estados dos registradores quânticos são vetores. O estado $|x_1, x_2, ..., x_n\rangle$ de um registrador de n *qubits* independentes é definido como $|x_1, x_2, ..., x_n\rangle = |x_1\rangle \otimes |x_2\rangle \otimes ... \otimes |x_n\rangle = \bigotimes_{i=1}^{n} |x_i\rangle$, onde $|x_i\rangle \in \mathbb{C}^2$ é o estado do i-ésimo *qubit* e $\cdot \otimes \cdot$ é o produto de Kronecker. Pode-se dizer que o produto de Kronecker é um análogo do produto cartesiano para espaços vetoriais.

O produto de Kronecker de um par de matrizes (A, B) é definido como:

$$A \otimes B = \begin{bmatrix} a_{1,1}B & \dots & a_{1,m}B \\ \vdots & \ddots & \vdots \\ a_{n,1}B & \dots & a_{n,m}B \end{bmatrix} \tag{9}$$

No caso de um par de estados $|a\rangle = [a_1, a_2]^\intercal$ e $|b\rangle = [b_1, b_2]^\intercal$, o produto de Kronecker é:

$$|a, b\rangle = |a\rangle \otimes |b\rangle = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \otimes \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1 \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \\ a_2 \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_1b_1 \\ a_1b_2 \\ a_2b_1 \\ a_2b_2 \end{bmatrix} \tag{10}$$

Considerando $a \in \{0, 1\}$ e $b \in \{0, 1\}$, temos os seguintes possíveis estados $|a, b\rangle$:

$$|0, 0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{11}$$

$$|0, 1\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{12}$$

$$|1, 0\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \tag{13}$$

$$|1, 1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{14}$$

Em geral, para qualquer sequência de valores binários $[x_1, x_2, ..., x_n]$, o vetor equivalente a $|x_1, x_2, ..., x_n\rangle$ é $[\lambda_1, \lambda_2, ..., \lambda_{2^n}]^\intercal$ tal que:

$$\lambda_i = \begin{cases} 1, & i = 1 + \boldsymbol{x} \\ 0, & i \neq 1 + \boldsymbol{x} \end{cases} \quad , \qquad \boldsymbol{x} = \sum_{j=1}^{n} 2^{j-1} x_j \tag{15}$$

Um *bra* ou *ket* também pode conter um valor numérico correspondente a uma sequência de valores binários. Exemplos: $|\mathbf{0}\rangle = |0, 0\rangle$, $|\mathbf{1}\rangle = |0, 1\rangle$, $|\mathbf{2}\rangle = |1, 0\rangle$, $|\mathbf{3}\rangle = |1, 1\rangle$. Especificamente neste trabalho, o uso de negrito em números ou símbolos pode indicar que o número ou símbolo corresponde a uma sequência de valores binários.

Algumas propriedades importantes do produto de Kronecker são:

- $A \otimes (\sum_{i=1}^{n} B_i) = \sum_{i=1}^{n} (A \otimes B_i)$, onde A e B_i são matrizes.
- $(\sum_{i=1}^{n} A_i) \otimes B = \sum_{i=1}^{n} (A_i \otimes B)$, onde A_i e B são matrizes.

- $(\sum_{i=1}^{n} A_i) \otimes (\sum_{j=1}^{m} B_j) = \sum_{i=1}^{n} \sum_{j=1}^{m} (A_i \otimes B_j)$, onde A_i e B_j são matrizes.
- $(kA) \otimes B = A \otimes (kB) = k(A \otimes B)$, onde A e B são matrizes e k é um escalar.

2.4 Emaranhamento quântico

Quando um sistema de n *qubits* possui um estado quântico que não pode ser definido como um produto de Kronecker de n estados independentes em $\mathbb{C}^2$, é dito que este sistema está **emaranhado** ou **entrelaçado**. Um exemplo clássico de emaranhamento quântico com dois *qubits* é o estado de Bell, definido na equação 16.

$$|\mathbf{Bell}\rangle = \frac{|0,0\rangle + |1,1\rangle}{\sqrt{2}} \tag{16}$$

Não existe nenhum par de estados $(|q_1\rangle, |q_2\rangle)$ tal que $|q_2\rangle \otimes |q_1\rangle = |\mathbf{Bell}\rangle$, e portanto $|\mathbf{Bell}\rangle$ é um estado de dois *qubits* emaranhados. Em geral, para sistemas de dois *qubits*, é possível determinar se há um emaranhamento no estado verificando as amplitudes de probabilidade deste. Para fins explicativos, considere o seguinte estado de 2 *qubits*:

$$\begin{aligned} |a,b\rangle &= (\alpha_0|0\rangle + \alpha_1|1\rangle) \otimes (\beta_0|0\rangle + \beta_1|1\rangle) \\ &= \alpha_0\beta_0|0,0\rangle + \alpha_0\beta_1|0,1\rangle + \alpha_1\beta_0|1,0\rangle + \alpha_1\beta_1|1,1\rangle \end{aligned} \tag{17}$$

Considerando $\omega_0 = \alpha_0\beta_0$, $\omega_1 = \alpha_0\beta_1$, $\omega_2 = \alpha_1\beta_0$ e $\omega_3 = \alpha_1\beta_1$, temos:

$$|a,b\rangle = \omega_0|0,0\rangle + \omega_1|0,1\rangle + \omega_2|1,0\rangle + \omega_3|1,1\rangle \tag{18}$$

É óbvio constatar que $(\alpha_0\beta_0)(\alpha_1\beta_1) = (\alpha_0\beta_1)(\alpha_1\beta_0)$, e dada esta afirmação, é possível deduzir que:

$$\omega_0\omega_3 = \omega_1\omega_2 \tag{19}$$

A equação 19 é uma regra que deve ser satisfeita para que se prove que um estado de 2 *qubits* com amplitudes $\omega_0, \omega_1, \omega_2$ e ω_3 é um produto tensorial de dois estados independentes $|a\rangle \in \mathbb{C}^2$ e $|b\rangle \in \mathbb{C}^2$. O estado Bell descrito na equação 16 tem coeficientes $\omega_0 = \omega_3 = \frac{1}{\sqrt{2}}$ e $\omega_1 = \omega_2 = 0$, que não satisfazem a equação 19.

De modo geral, um estado $|\psi\rangle$ de n *qubits* não possui emaranhamento caso satisfaça:

$$|\psi\rangle = \bigotimes_{i=1}^{n} |q_i\rangle, \qquad |q_i\rangle \in \mathbb{C}^2 \tag{20}$$

2.5 Portas lógicas quânticas

Como já foi dito na seção 2.1, as ditas portas lógicas quânticas são operadores lineares unitários (matrizes unitárias, mais especificamente). Uma matriz unitária é, basicamente, uma matriz quadrada cuja transposta conjugada é também sua inversa. Em linguagem matemática, essa afirmação pode ser descrita como $U^\dagger = U^{-1}$, onde U é uma matriz quadrada, $U^\dagger$ é sua transposta conjugada e U^{-1} é sua inversa. Em alguns casos, a matriz U é sua própria transposta conjugada e inversa; nesses casos, a matriz é dita autoadjunta. Exemplos de matrizes autoadjuntas são as matrizes de Pauli, a porta Hadamard e a porta *Controlled-NOT*.

Existe uma infinidade de portas lógicas quânticas. Em teoria, qualquer matriz unitária cujas quantidades de linhas e colunas são potências de 2 pode ser considerada uma porta lógica quântica. No entanto, existe um conjunto de portas lógicas quânticas que são universais, que inclui as portas Clifford (*Controlled-NOT*, Hadamard, S, Pauli-X, Pauli-Y e Pauli-Z) e a porta T. Além dessas, existem as portas parametrizadas $R_x(\theta)$, $R_y(\theta)$ e $R_z(\theta)$. Todas essas portas lógicas serão descritas nas próximas seções.

2.5.1 Matrizes de Pauli

As matrizes Pauli-X (eq. 28), Pauli-Y (eq. 22) e Pauli-Z (eq. 29) são portas lógicas quânticas básicas de 1 *qubit*. O papel dessas portas é rotacionar o estado do *qubit* em π radianos na esfera de Bloch em torno dos respectivos eixos x, y e z. A porta X, em particular, é frequentemente usada como um análogo quântico da porta lógica clássica *NOT*, pois faz os mapeamentos $|0\rangle \mapsto |1\rangle$ e $|1\rangle \mapsto |0\rangle$. Essas três matrizes podem ser grafadas como (X, Y, Z), $(\sigma_x, \sigma_y, \sigma_z)$ ou $(\sigma_1, \sigma_2, \sigma_3)$. Em computação quântica, a primeira notação é a mais usada.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{21}$$

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \tag{22}$$

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{23}$$

2.5.2 Hadamard

A porta Hadamard (eq. 24) equivale a uma transformada discreta de Fourier 2×2. Esta porta é frequentemente utilizada para transformar um estado da base computacional em uma superposição, pois mapeia os estados $|0\rangle$ e $|1\rangle$ para, respectivamente, $|+\rangle$ e $|-\rangle$. Outro uso comum da porta Hadamard é como parte da transformada de Fourier quântica, que será apresentada mais adiante.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{24}$$

2.5.3 S e T

As portas S (também chamada de Phase ou P) e T são, assim como a porta Pauli-Z, matrizes que rotacionam o estado de um *qubit* em torno do eixo z na esfera de Bloch. Enquanto a porta Pauli-Z faz rotações de π radianos, as portas S e T fazem rotações de, respectivamente, $\pi/2$ radianos e $\pi/4$ radianos.

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \tag{25}$$

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix} \tag{26}$$

2.5.4 $R_x(\theta)$, $R_y(\theta)$ e $R_z(\theta)$

As portas $R_x(\theta)$ (eq. 28), $R_y(\theta)$ (eq. 22) e $R_z(\theta)$ (eq. 29) são matrizes que fazem rotações de ângulo livre na esfera de Bloch, onde o ângulo de rotação é dado por θ. Os eixos de rotação das três portas são, respectivamente, x, y e z. Pode-se dizer que essas portas são generalizações das matrizes de Pauli, uma vez que $R_x(\pi) = X$, $R_y(\pi) = Y$ e $R_z(\pi) = Z$. Além disso, essas três matrizes de rotação de ângulo livre são definidas a partir das matrizes de Pauli como $R_x(\theta) = e^{-i\theta X/2}$, $R_y(\theta) = e^{-i\theta Y/2}$ e $R_z(\theta) = e^{-i\theta Z/2}$.

$$R_x(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \tag{27}$$

$$R_y(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \tag{28}$$

$$R_z(\theta) = \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix} \tag{29}$$

2.5.5 *Controlled-NOT* e demais portas controladas

Controlled-NOT (eq. 30 e tab. 1) é a porta de múltiplos *qubits* mais básica. Fisicamente, o uso dessa porta lógica induz um emaranhamento entre os *qubits* operandos. Do ponto de vista computacional, o emaranhamento criado pela porta *Controlled-NOT* pode ser interpretado como um análogo quântico da porta lógica clássica *XOR* ou como uma aplicação da porta X sobre um *qubit* alvo condicionada por um *qubit de controle*. Essas duas interpretações se devem ao fato de que, para um registrador de estado $|c, t\rangle$, onde

$|c\rangle$ é o estado do *qubit* de controle e $|t\rangle$ é o estado do *qubit* alvo, o mapeamento de CNOT pode ser definido como $|c, t\rangle \mapsto |c, t \oplus c\rangle$.

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{30}$$

$\|c, t\rangle$	$\text{CNOT}\|c, t\rangle$
$\|0, 0\rangle$	$\|0, 0\rangle$
$\|0, 1\rangle$	$\|0, 1\rangle$
$\|1, 0\rangle$	$\|1, 1\rangle$
$\|1, 1\rangle$	$\|1, 0\rangle$

Tabela 1 – Tabela-verdade da porta *Controlled-NOT*

De modo geral, é possível criar uma versão controlada de qualquer unitário U, inclusive com múltiplos *qubits* de controle, combinando as portas descritas acima. Há dois exemplos de portas controladas que, por serem utilizadas no presente trabalho, devem ser mencionadas: a porta *Toffoli* (eq. 31), que é similar à porta *Controlled-NOT*, porém com 2 *qubits* de controle, fazendo o mapeamento $|c_1, c_2, t\rangle \mapsto |c_2, c_2, t \oplus c_1 c_2\rangle$, e a porta CR_z (eq. 32), que é a versão controlada da porta R_z.

$$\text{Toffoli} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{31}$$

$$CR_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^{-i\theta/2} & 0 \\ 0 & 0 & 0 & e^{i\theta/2} \end{bmatrix} \tag{32}$$

2.6 Circuitos quânticos

Circuitos quânticos são modelos diagramáticos de algoritmos quânticos onde *qubits* são representados por linhas retas horizontais, e portas lógicas quânticas são representadas por blocos e linhas verticais sobre os *qubits*. Por serem análogos a circuitos digitais reversíveis, os circuitos quânticos são representações gráficas relativamente amigáveis para pessoas que já têm familiaridade com lógica digital. Uma outra forma bastante útil de representar algoritmos quânticos graficamente é por meio do Cálculo-ZX (WETERING, 2020). No entanto, Cálculo-ZX só costuma ser usado para fins de otimização.

Em circuitos quânticos, cada porta lógica quântica tem seu próprio símbolo, como mostra a figura 3.

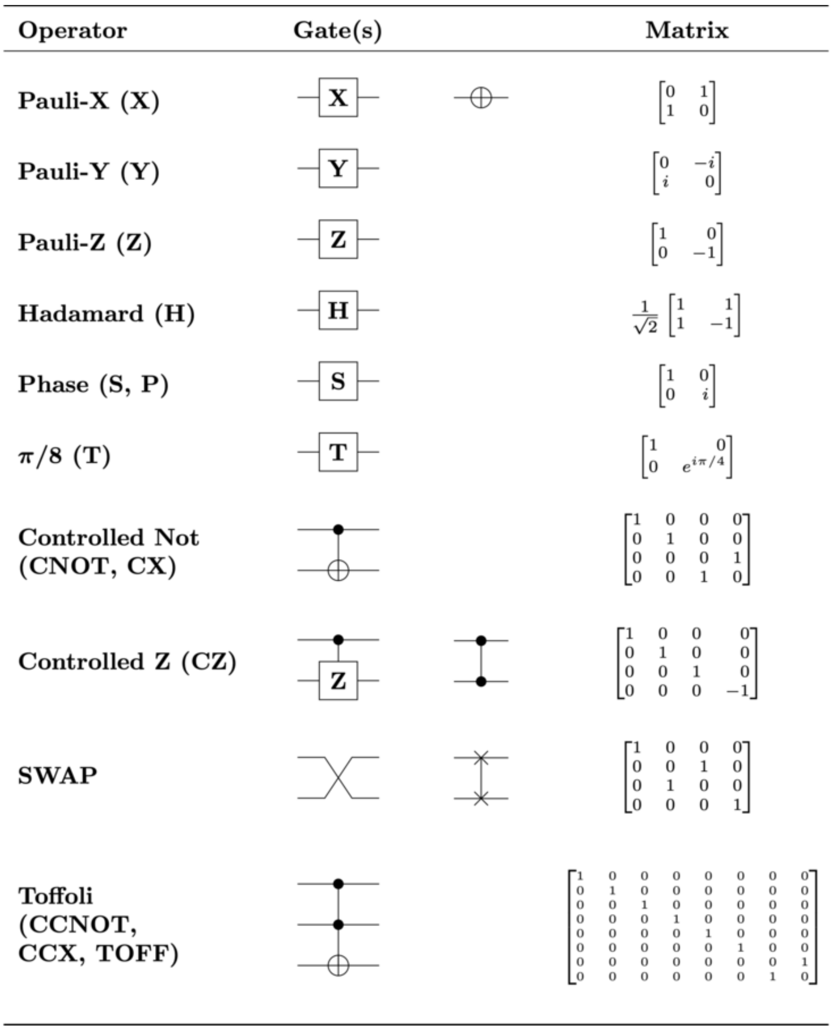

Figura 3 – Simbologia das portas lógicas quânticas em circuitos quânticos (figura extraída de Wikimedia Commons, de autoria de Rxtreme, sob licença CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>)

Usemos, como exemplo introdutório, um circuito quântico para criar um estado Bell:

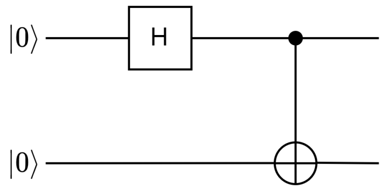

Figura 4 – Circuito quântico que gera um estado Bell

Como pode ser visto na figura acima, o circuito gerador do estado Bell é composto por um registrador de dois *qubits* inicializado como $|0,0\rangle$, uma porta Hadamard sobre o primeiro *qubit*, e uma porta *Controlled-NOT* tomando o primeiro *qubit* como controle e o segundo *qubit* como alvo. Este circuito pode ser descrito algebricamente como um operador U tal que $U|0,0\rangle = (|0,0\rangle + |1,1\rangle)/\sqrt{2}$, definido como $U = \text{CNOT}_{(q_1,q_2)} \times (I \otimes H)$, onde I é uma matriz identidade 2×2 e $\text{CNOT}_{(q_1,q_2)}$ é a matriz da porta *Controlled-NOT* tomando o primeiro *qubit* como controle e o segundo como alvo.

Como segundo exemplo introdutório, tomemos a decomposição da porta Toffoli em portas *Controlled-NOT*, Hadamard, T e $T^\dagger$:

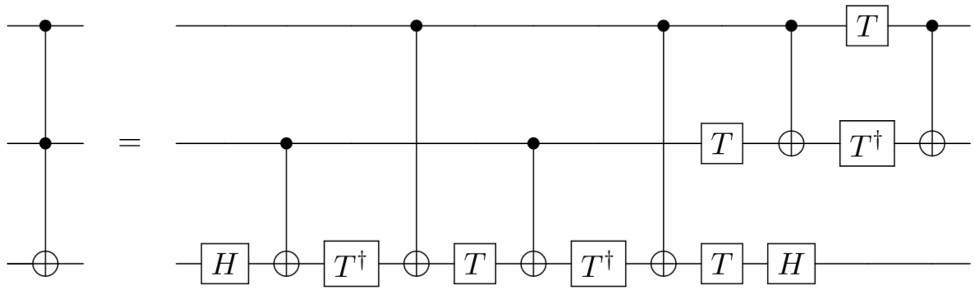

Figura 5 – Decomposição da porta lógica quântica Toffoli (figura extraída de Wikimedia Commons, de autoria de Geek3, sob licença CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>)

Observe que as linhas verticais com pontos escuros indicam que há *qubits* de controle na porta. Também há casos em que os pontos são claros, indicando que *qubits* de controle estão negados, como mostra a figura a seguir:

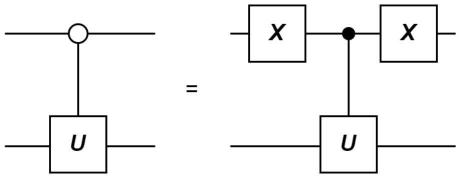

Figura 6 – Porta genérica com *qubit* de controle negado

Capítulo 3

Operações lógicas, aritméticas e relacionais em computação quântica

Como já mencionado no primeiro capítulo, uma das ideias centrais da linguagem DLQ é a "tradução" de operações lógicas, aritméticas e relacionais em circuitos quânticos. Na versão atual do compilador *DLQpiler*, há suporte para a síntese dos operadores *not*, *and*, *or*, soma ($+$), subtração ($-$), multiplicação ($*$), potenciação (ˆ), maior-que ($>$), menor-que ($<$), igual ($=$) e diferente ($!=$). Há três modalidades de síntese para esses operadores: **registrador-por-registrador**, onde a operação é feita entre duas variáveis; **registrador-por-constante**, onde a operação é feita entre uma variável do lado esquerdo e uma constante do lado direito; e **constante-por-registrador**, onde a operação é feita entre uma constante do lado esquerdo e uma variável do lado direito. Há também casos em que uma operação é feita com apenas operadores constantes. Nesses casos, o resultado da operação é computado pelo próprio compilador.

Neste capítulo, serão apresentados os circuitos quânticos utilizados para computar as operações citadas acima. Além disso, a forma como o algoritmo de Grover é usado para obter resultados relevantes é explicada.

3.1 Operações lógicas

Implementar as operações lógicas *not*, *and* e *or* como circuitos quânticos é bastante simples. Em primeiro lugar, deve-se pensar em um *qubit* como uma variável booleana. Seguindo este raciocínio, é trivial constatar que a porta lógica quântica Pauli-X pode ser usada como o operador *not*. Já o operador *and* pode ser traduzido em uma porta lógica Toffoli, uma vez que esta porta faz o mapeamento $|c_1, c_2, 0\rangle \mapsto |c_1, c_2, c_1c_2\rangle$. Neste caso, toma-se como alvo um *qubit* auxiliar inicializado como $|0\rangle$, o qual chamamos de *qubit* resultado. Por fim, a partir da junção dos operadores *not* e *and*, é possível criar um operador *or* com base no teorema de De Morgan, segundo o qual $A \vee B = \overline{\overline{A} \wedge \overline{B}}$. A figura 7 mostra os circuitos quânticos dos operadores lógicos *not*, *and* e *or*.

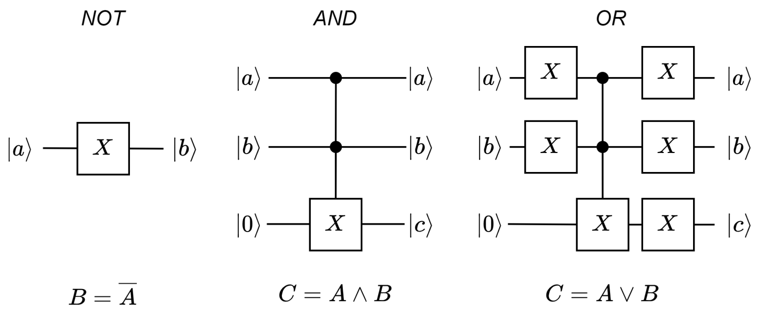

Figura 7 – As três operações lógicas (*not*, *and* e *or*), e seus respectivos circuitos quânticos equivalentes.

3.2 Transformada de Fourier Quântica (QFT)

Antes de partir para as operações aritméticas, é necessário abordar a transformada de Fourier quântica (QFT), que é usada como parte dos circuitos quânticos aritméticos que são apresentados adiante. Este artificio é, basicamente, uma implementação da transformada discreta de Fourier em forma de circuito quântico, proposta inicialmente por Coppersmith (1994). Esta transformada mapeia o espaço vetorial da base computacional para outro espaço vetorial onde é mais fácil fazer algumas operações aritméticas. Costuma-se chamar a base do contra-domínio da QFT de "base de Fourier". O uso de QFT em aritmética já foi explorado em trabalhos como os de Draper (2000) e Ruiz-Perez & Garcia-Escartin (2017).

Algebricamente, a QFT faz o mapeamento descrito na equação 33, podendo ser definida como um operador U_{QFT} na notação de Dirac (eq. 34) ou na forma matricial (eq. 35). Nessas três equações, $\omega = e^{\frac{2\pi i}{2^N}}$, l é o índice-coluna (começando em 0) e r é o índice-linha (também começando em 0).

$$|j\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \omega_N^{jk} |k\rangle \tag{33}$$

$$U_{\text{QFT}} = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} \omega_N^{jk} |k\rangle\langle j| \tag{34}$$

$$U_{\text{QFT}} = \frac{1}{\sqrt{2^N}} \begin{bmatrix} \omega^0 & \dots & \omega^{0l} & \dots & \omega^{0(2^N-1)} \\ \vdots & \ddots & \vdots & & \vdots \\ \omega^{0r} & \dots & \omega^{lr} & \dots & \omega^{r(2^N-1)} \\ \vdots & & \vdots & \ddots & \vdots \\ \omega^{0(2^N-1)} & \dots & \omega^{l(2^N-1)} & \dots & \omega^{(2^N-1)^2} \end{bmatrix} \tag{35}$$

O circuito quântico tradicional da QFT tem complexidade $\mathcal{O}(n^2)$, onde n é o número de *qubits*. Este circuito é composto majoritariamente por portas Hadamard e R_z controladas.

A figura 8 mostra um exemplo de circuito quântico de QFT de 3 *qubits*, e o algoritmo 1 mostra o procedimento de construção da QFT para um número arbitrário de *qubits* n.

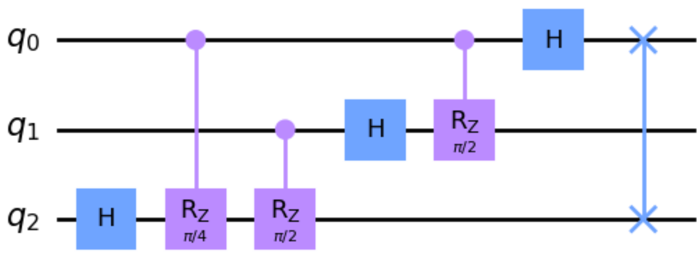

Figura 8 – Circuito quântico de uma QFT de 3 *qubits*

Algoritmo 1 Construção algorítmica da QFT para qualquer número de *qubits*

Data: $n \in \mathbb{Z} \cap [1; \infty)$ #Número de qubits
Data: $q_1, q_2, ..., q_n$ #Qubits alvo
$m \leftarrow n$
while $m > 0$ **do**
  $H(\text{alvo} = q_{m-1})$ #Porta Hadamard
  for $i \in [0, ..., m-1]$ **do**
    $CR_z(\theta = \frac{\pi}{2^{m-1-i}}, \text{controle} = q_i, \text{alvo} = q_{m-1})$ #Porta R_z controlada
  end
  $m \leftarrow m - 1$
end
for $m \in [0, ..., \lfloor n/2 \rfloor]$ **do**
  $\text{Swap}(\text{alvo}_1 = q_m, \text{alvo}_2 = q_{n-m-1})$ # Porta Swap
end

3.3 Operações aritméticas

3.3.1 Soma e subtração registrador-por-constante

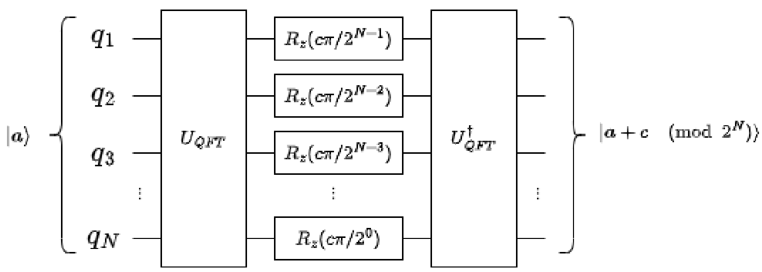

Figura 9 – Circuito quântico do somador registrador-por-constante

O circuito quântico aritmético mais básico utilizado neste trabalho é o somador registrador-por-constante. Este circuito quântico, que aqui é representado algebricamente por $U_+(c) \in$

$\mathbb{C}^{2^n \times 2^n}$, faz o mapeamento $|\boldsymbol{x}\rangle \mapsto |\boldsymbol{x}+\boldsymbol{c} \pmod{\mathbf{2^n}}\rangle$, onde c é o valor constante inteiro que deseja-se somar ao registrador, e n é o número de *qubits* no registrador. A implementação do somador registrador-por-constante que foi utilizada, apresentada na figura 9, é uma modificação, original do presente trabalho, do somador baseado em QFT proposto por Draper (2000). Algebricamente, define-se este somador simplificado da seguinte forma:

$$U_{+}(c) = U_{\mathrm{QFT}}^{\dagger} \times U_{\phi(+)}(c) \times U_{\mathrm{QFT}} \tag{36}$$

Os operadores U_{QFT} e $U_{\mathrm{QFT}}^{\dagger}$ são, respectivamente, a transformada de Fourier quântica e a transformada inversa de Fourier quântica, e o operador $U_{\phi(+)}(c)$ é o somador registrador-por-constante para uma constante c na base de Fourier. O operador $U_{\phi(+)}(c)$ é definido para n *qubits* como:

$$U_{\phi(+)}(c) = \bigotimes_{\tau=1}^{n} R_z\left(\frac{c\pi}{2^{n-t}}\right) = R_z\left(\frac{c\pi}{2^{n-n}}\right) \otimes ... \otimes R_z\left(\frac{c\pi}{2^{n-1}}\right) \tag{37}$$

onde $t = N - (\tau - 1)$.

Uma vez que se tem os somadores $U_{+}(c)$ e $U_{\phi(+)}(c)$, é trivial constatar que os subtratores são simplesmente as transposições dos somadores, ou seja, $U_{-}(c) = U_{+}(c)^{\dagger}$ e $U_{\phi(-)}(c)^{\dagger}$.

A corretude do somador registrador-por-constante pode ser demonstrada analisando-o na forma matricial. Primeiramente, expandimos o estado de n *qubits* $|a\rangle$ tal que $a \in \{x \in \mathbb{N} : 0 \leq x \leq 2^n - 1\}$ na forma matricial:

$$|a\rangle = [\lambda_j] \in \mathbb{C}^{2^N \times 1}, \quad \lambda_j = \begin{cases} 1, & j = a \\ 0, & j \neq a \end{cases} \tag{38}$$

Na equação 38, a expressão $[\lambda_j] \in \mathbb{C}^{2^N \times 1}$ corresponde a uma matriz de 2^n linhas e 1 coluna. A variável j corresponde ao índice-linha da respectiva entrada λ_j da matriz, tal que $0 \leq j \leq 2^n - 1$.

Em seguida, expandimos U_{QFT} na forma matricial como:

$$U_{\mathrm{QFT}} = \frac{1}{\sqrt{2^n}}[\omega^{jk}] \in \mathbb{C}^{2^n \times 2^n} \tag{39}$$

Na equação 39, a expressão $\frac{1}{\sqrt{2^n}}[\omega^{jk}] \in \mathbb{C}^{2^n \times 2^n}$ corresponde a uma matriz de 2^n linhas e 2^n colunas cujas entradas são ω^{jk}, onde j é o índice-linha da entrada e k é o índice-coluna da entrada. O símbolo ω é uma constante definida como $\omega = e^{\frac{2\pi i}{2^n}}$.

A operação $U_{\mathrm{QFT}}|\boldsymbol{a}\rangle$ resulta em uma matriz coluna equivalente à coluna de índice $\boldsymbol{a}$ da matriz U_{QFT}. Descrevendo esta operação algebricamente, temos:

$$U_{\text{QFT}}|a\rangle = \frac{1}{\sqrt{2^n}}[\omega^{ja}] \in \mathbb{C}^{2^n \times 1} \tag{40}$$

Na equação 40, a expressão $\frac{1}{\sqrt{2^n}}[\omega^{ja}] \in \mathbb{C}^{2^n\times 1}$ corresponde a uma matriz coluna com 2^n linhas cujas entradas são $\frac{1}{\sqrt{2^n}}\omega^{ja}$, onde j é o índice-linha da entrada. Esta expressão é equivalente a $\frac{1}{\sqrt{2^n}}[..., \omega^{ja}, ...]^\intercal$.

A aplicação de $U^\dagger_{\text{QFT}}$ sobre $U_{\text{QFT}}|\boldsymbol{a}\rangle$ resulta no próprio $|\boldsymbol{a}\rangle$, o que nos leva a seguinte constatação:

$$U^\dagger_{\text{QFT}}\left(\frac{1}{\sqrt{2^n}}[..., \omega^{ja}, ...]^\intercal\right) = |a\rangle, \qquad 0 \le a \le 2^n - 1 \tag{41}$$

A equação 41 é restrita a $0 \le \boldsymbol{a} \le 2^n - 1$. Porém, para qualquer $\boldsymbol{a} \in \mathbb{N}$, a expressão $U^\dagger_{\text{QFT}}\left(\frac{1}{\sqrt{2^n}}[..., \omega^{j\boldsymbol{a}}, ...]^\intercal\right)$ resulta em $|\boldsymbol{a} \pmod{2^n}\rangle$, e isto se deve à periodicidade de $\omega^{j\boldsymbol{a}}$.

Baseando-se na equação 41, é possível deduzir que:

$$\begin{gathered} U^\dagger_{\text{QFT}}\left(\frac{1}{\sqrt{2^N}}[..., \omega^{j(a+c)}, ...]^\intercal\right) = U^\dagger_{\text{QFT}}\left(\frac{1}{\sqrt{2^N}}[..., \omega^{ja}\omega^{jc}, ...]^\intercal\right) = |a + c\rangle, \\ 0 \le a + c \le 2^N - 1 \end{gathered} \tag{42}$$

Em geral, para qualquer $|\boldsymbol{x}\rangle$ tal que $\boldsymbol{x} \in \mathbb{N}$, consideramos que $U_{\text{QFT}}|\boldsymbol{x}\rangle = |\phi(\boldsymbol{x})\rangle$ e $U^\dagger_{\text{QFT}}|\phi(\boldsymbol{x})\rangle = |\boldsymbol{x}\rangle$, onde $|\phi(\boldsymbol{x})\rangle$ é simplesmente uma notação para o resultado da aplicação da transformada de Fourier quântica sobre um estado $|\boldsymbol{x}\rangle$ qualquer.

Com base na constatação apresentada na equação 42, conclui-se que, para criar o mapeamento $|\phi(a)\rangle \mapsto |\phi(a+c)\rangle$, é necessário multiplicar cada entrada $\frac{1}{\sqrt{2^n}}\omega^{ja}$ de $|\phi(a)\rangle$ por um fator ω^{jc}. Esta operação pode ser descrita como um produto matricial $U_{\phi(+)}(c)|\phi(a)\rangle$, uma vez que o operador $U_{\phi(+)}(c)$ pode ser expandido na forma matricial como mostra a seguinte equação:

$$U_{\phi(+)}(c) = [\lambda_{(j,k)}] \in \mathbb{C}^{2^n \times 2^n}, \qquad \lambda_{(j,k)} = \begin{cases} \omega^{jc}, & j = k \\ 0, & j \ne k \end{cases} \tag{43}$$

A equação 43 define $U_{\phi(+)}(c)$ como uma matriz diagonal cujas entradas diagonais são ω^{jc}, onde j e k são, respectivamente, o índice-linha e o índice-coluna da entrada. Esta definição também pode ser escrita como $U_{\phi(+)}(c) = \text{diag}[..., \omega^{jc}, ...]$.

3.3.2 Soma registrador-por-registrador

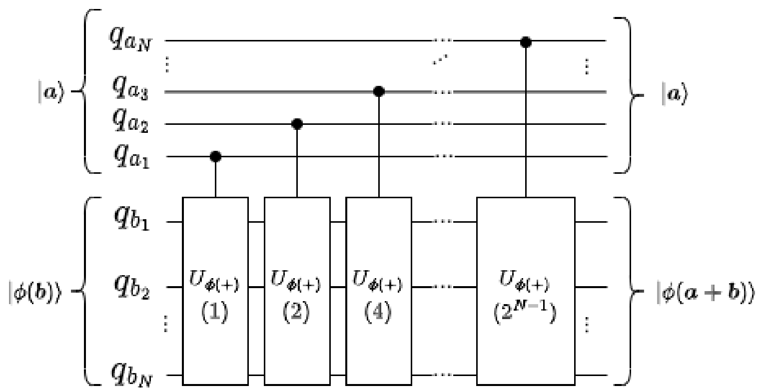

Figura 10 – Circuito quântico de um somador registrador-por-registrador

Com base no somador registrador-por-constante, é possível criar um somador registrador-por-registrador, ou seja, que faz o mapeamento $|\boldsymbol{b}, \boldsymbol{a}\rangle \mapsto |\boldsymbol{a} + \boldsymbol{b} \pmod{\mathbf{2}^{n}}, \boldsymbol{a}\rangle$, partindo do princípio de que é possível definir $b + a$ como $b + \sum_{j=1}^{n} a_j 2^{j-1}$, onde $a_j \in \{0, 1\}$ e n é o número de *qubits* alvo. A estrutura deste somador consiste em uma sequência de somadores $U_{\phi(+)}(2^{j-1})$ controlados por $|a_j\rangle$ aplicados sobre o registrador alvo inicializado como $U_{\text{QFT}}|\boldsymbol{b}\rangle$. Após a aplicação das n operações $U_{\phi(+)}(2^{j-1})$ controladas, aplica-se a QFT inversa sobre o registrador alvo para que o estado final do circuito se torne $|\boldsymbol{a} + \boldsymbol{b} \pmod{\mathbf{2}^{n}}, \boldsymbol{a}\rangle$. Uma representação diagramática deste somador registrador-por-registrador é apresentada na figura 10.

3.3.3 Multiplicação e potenciação

Também é possível criar um circuito de multiplicação com base no somador registrador-por-constante, partindo do pressuposto de que um produto $cx_1x_2...x_m$ pode ser definido como:

$$\left(\sum_{j_1=1}^{n_1} x_{(1,j_1)} 2^{j_1-1}\right) \times ... \times \left(\sum_{j_k=1}^{n_m} x_{(m,j_m)} 2^{j_m-1}\right) \times c = \sum_{j_1=1}^{n_1} ... \sum_{j_m=1}^{n_m} x_{(1,j_1)}...x_{(m,j_m)} c 2^{j_1+...+j_m-m} \tag{44}$$

onde c é uma constante natural, cada x_i é um fator natural, e cada $x_{(i,j)}$ é o j-ésimo bit (do menos ao mais significativo) de x_i. A partir desta formulação, pode-se implementar um circuito quântico de multiplicação aplicando uma série de operações $U_{\phi(+)}(c2^{j_1+...+j_m-m})$ multi-controladas pelas sequências de *qubits* $[|x_{(1,j_1)}\rangle, ..., |x_{(m,j_m)}\rangle]_{j_1,...,j_m}$ sobre um registrador alvo inicializado como $U_{\text{QFT}}|\mathbf{0}\rangle$, e em seguida aplicando a QFT inversa sobre o mesmo registrador. O circuito quântico resultante faz o mapeamento $|0, x_1, ..., x_m\rangle \mapsto |c \prod_i^m x_i \pmod{2^{n_t}}, x_1, ..., x_m\rangle$, onde n_t é o número de *qubits* do registrador alvo.

Seguindo a mesma lógica do circuito quântico de multiplicação, é possível criar circuitos de potenciação com expoente constante, ou híbridos de multiplicação e potenciação. Isto pode ser feito repetindo os fatores com expoente maior que 1 na formulação da equação 44. A expressão a^2b^3, por exemplo, fica:

$$\sum_{j_1=1}^{n_1}\sum_{j_2=1}^{n_2}\sum_{j_3=1}^{n_3}\sum_{j_4=1}^{n_4}\sum_{j_5=1}^{n_5} a_{j_1}a_{j_2}b_{j_3}b_{j_4}b_{j_5}2^{j_1+j_2+j_3+j_4+j_5-5}$$

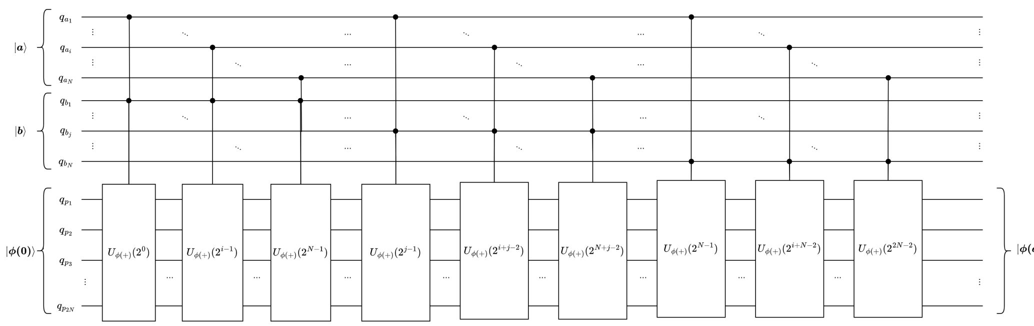

Figura 11 – Exemplo de circuito quântico multiplicador que faz o mapeamento $|\phi(\mathbf{0}), \boldsymbol{b}, \boldsymbol{a}\rangle \mapsto |\phi(\boldsymbol{ab}), \boldsymbol{b}, \boldsymbol{a}\rangle$

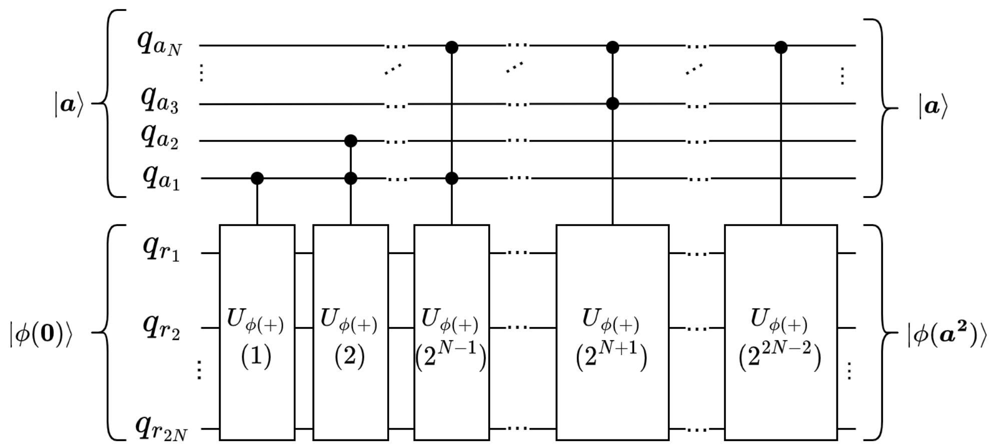

Figura 12 – Exemplo de circuito quântico de potenciação que faz o mapeamento $|\phi(\mathbf{0}), \boldsymbol{a}\rangle \mapsto |\phi(\boldsymbol{a^2}), \boldsymbol{a}\rangle$

3.4 Operações relacionais

3.4.1 Igualdade e não-igualdade

Introduzindo a parte das operações relacionais, que são operações de comparação entre operandos, temos as operações de igualdade (=) e não-igualdade ($\neq$). As implementações dessas duas operações em forma de circuitos quânticos baseiam-se no fato de que, para um par de operandos $(\boldsymbol{a}, \boldsymbol{b}) \in \mathbb{Z}^2$ tais que $\boldsymbol{a} = \sum_{i=1}^{n} a_i 2^{i-1}$ e $\boldsymbol{b} = \sum_{i=1}^{n} b_i 2^{i-1}$, onde a_i e b_i são bits, a proposição $\boldsymbol{a} = \boldsymbol{b}$ é verdadeira se $a_i = b_i$ para qualquer $1 \leq i \leq n$. Portanto, a proposição $\boldsymbol{a} = \boldsymbol{b}$ pode ser verificada de forma algorítmica fazendo, primeiro, uma atribuição $v_i \leftarrow \begin{cases} 1, & a_i = b_i \\ 0, & a_i \neq b_i \end{cases}$ para cada $1 \leq i \leq n$, onde $v_1, v_2, ..., v_n$ são variáveis auxiliares. Em seguida, toma-se $\boldsymbol{a} = \boldsymbol{b}$ como verdadeira caso $v_i = 1$ para qualquer $1 \leq i \leq n$. Este algoritmo pode ser facilmente traduzido na expressão booleana $(a_1 \oplus b_1)(a_2 \oplus b_2)...(a_n \oplus b_n)$, que por sua vez pode ser facilmente traduzida no seguinte circuito quântico:

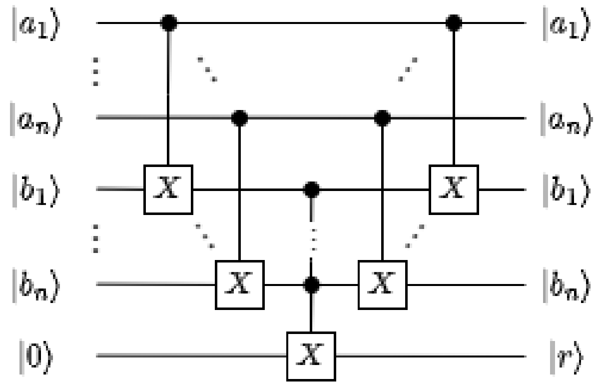

Figura 13 – Circuito quântico da operação de igualdade, onde $|r\rangle$ é o resultado

Observe que, no circuito quântico da figura 13, uma sequência de portas *Controlled-Not* são utilizadas para transformar os estados $|b_i\rangle$ em $|a_i \oplus b_i\rangle$. Em seguida, uma porta X multi-controlada é aplicada sobre o *qubit* de resultado para transformar o estado deste em $|(a_1 \oplus b_1)...(a_n \oplus b_n)\rangle$. Por fim, as portas *Controlled-Not* são aplicadas novamente de forma inversa para fazer os *qubits* do operando $\boldsymbol{b}$ voltarem aos seus estados originais. Esta é, no entanto, apenas a implementação do operador de igualdade no modo registrador-por-registrador. No modo registrador-por-constante, ao invés de portas *Controlled-Not*, utilizam-se portas X. Se, por exemplo, $\boldsymbol{a}$ é um operando constante, uma porta X será aplicada sobre cada *qubit* de estado inicial $|b_i\rangle$ tal que $a_i = 1$.

Uma vez que se tem o operador de igualdade implementado, a obtensão da implementação do operador de não-igualdade é trivial: basta adicionar, por último, uma porta X sobre o *qubit* de resultado para transformar $|(a_1 \oplus b_1)...(a_n \oplus b_n)\rangle$ em $|\overline{(a_1 \oplus b_1)...(a_n \oplus b_n)}\rangle$.

3.4.2 Maior-que e menor-que

As implementações dos operadores relacionais maior-que (>) e menor-que (<) em forma de circuitos quânticos partem do princípio de que, para um par de operandos $(\boldsymbol{a}, \boldsymbol{b}) \in \mathbb{Z}^2$, a proposição $\boldsymbol{a} < \boldsymbol{b}$ é verdadeira se $\boldsymbol{a} - \boldsymbol{b} < 0$. Por mais obvia que esta afirmação seja, ela tem um papel importante, que é o de reduzir o problema de comparar dois inteiros positivos a um problema menor: comparar um inteiro com 0. Considerando que a subtração $\boldsymbol{a} - \boldsymbol{b}$ pode resultar em valores menores que 0, para comparar este resultado a 0, é necessário utilizar a **notação de complemento de 2**. Este artifício é um sistema numérico utilizado em computação para representar números inteiros com sinal. O complemento de 2 de um número inteiro é obtido invertendo todos os seus bits e somando 1 ao resultado. Como exemplo, considere o número no sistema binário 0101. Para encontrar o complemento de dois deste número, primeiro inverte-se cada um de seus bits para obter 1010, e então soma-se 1 para obter o complemento de dois 1011. Na notação de complemento de 2, o bit mais significativo do número representa o sinal (0 para números positivos e 1 para números negativos), podendo ser utilizado para verificar se o número é menor que 0.

A implementação algorítmica do operador menor-que consiste nas seguintes etapas:

1. Toma-se 2 registradores: o primeiro para armazenar $\boldsymbol{a}$ e o segundo para armazenar $\boldsymbol{b}$;

2. Acrescenta-se um novo bit mais significativo ao primeiro registrador para tornar não-modular a operação de subtração da próxima etapa;

3. Aplica-se uma subtração para transformar o estado $\boldsymbol{a}$ do primeiro registrador em $\boldsymbol{a} - \boldsymbol{b}$;

4. Verifica-se o valor do bit mais significativo do primeiro registrador: caso seja 1, então $\boldsymbol{a} < \boldsymbol{b}$, e caso seja 0, então $\boldsymbol{a} \geq \boldsymbol{b}$.

O procedimento descrito acima pode ser traduzido em circuito quântico, em modo registrador-por-registrador, utilizando um subtrator registrador-por-registrador e uma porta *Controlled-Not*, como mostra a seguinte figura:

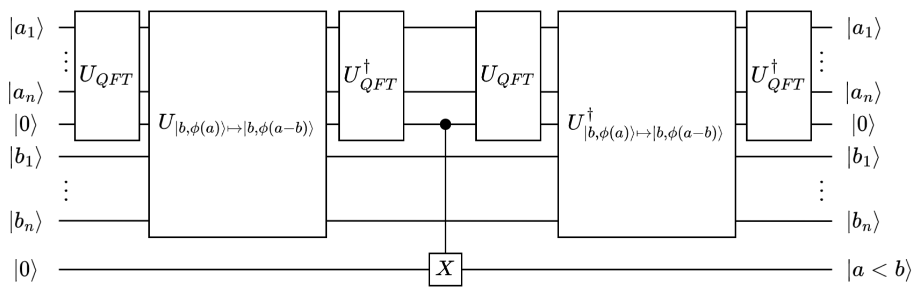

Figura 14 – Circuito quântico do operador menor-que, modo registrador-por-registrador

Já no modo registrador-por-constante (fig. 15), o circuito quântico proposto faz uso do operador $U_+(\cdot)$, e o número de *qubits* adicionais no registrador operando depende da constante com a qual o valor do registrador será comparado. No caso em que a comparação

$\boldsymbol{a} < \boldsymbol{b}$ é feita com $\boldsymbol{b}$ constante, o número de *qubits* adicionais necessários pode ser calculado como $n - \max\{n, \lceil \log_2 b \rceil\} + 1$.

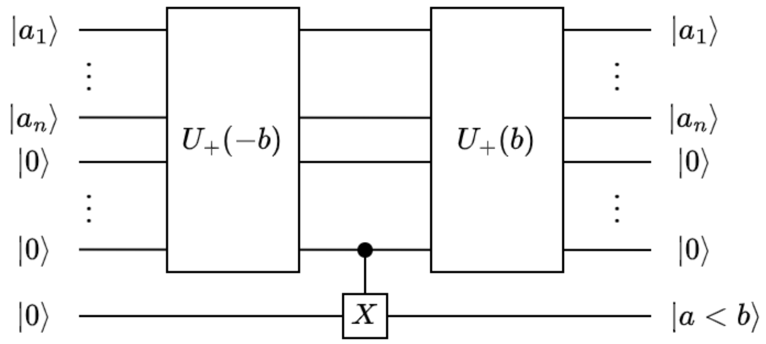

Figura 15 – Circuito quântico do operador menor-que, modo registrador-por-constante

A partir das implementações do operador relacional menor-que apresentadas acima, é possível criar implementações do operador maior-que, dado que $a > b$ equivale a $\neg(a - 1 < b)$ para $a \in \mathbb{N}$. Na figura a seguir, o circuito quântico do operador maior-que, em modo registrador-por-constante, é apresentado.

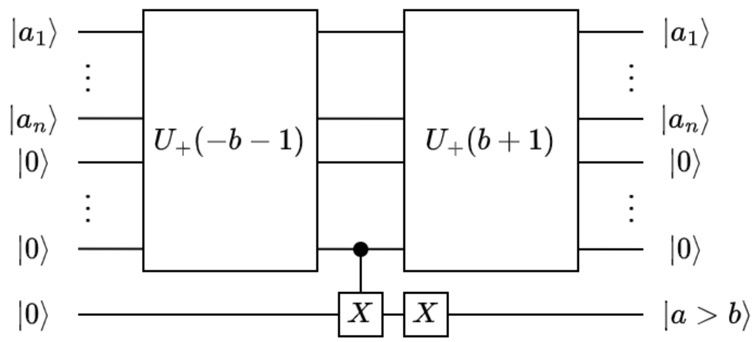

Figura 16 – Circuito quântico do operador maior-que, modo registrador-por-constante

3.5 Soluções de problemas com o algoritmo Grover

Como já foi dito no capítulo 2, uma vez que se usa um computador quântico, espera-se alguma vantagem proveniente do paralelismo quântico. Executar operações lógicas, aritméticas e relacionais em computadores quânticos, por si só, não é vantajoso, pois embora estas operações possam ser usadas para processar quantidades enormes de dados paralelamente, apenas uma tupla de valores "selecionada" aleatoriamente pode ser obtida em cada rodada do programa quântico. No entanto, quando estas operações são associadas ao algoritmo de Grover, torna-se possível aumentar a probabilidade de obtenção de tuplas de valores que satisfaçam determinadas condições.

O algoritmo de Grover é um algoritmo quântico de busca proposto por Lov K. Grover (1996). Este algoritmo resolve problemas de busca em conjuntos de dados não-ordenados com complexidade $\mathcal{O}(\sqrt{N})$, onde N é o tamanho do espaço de busca.

Considere um espaço de busca $X = \{x_1, x_2, ..., x_N\}$ para o qual deseja-se encontrar um x_i que satisfaça $f(x_i) = 1$, tal que $f(\cdot) \in \{0, 1\}$. O algoritmo de Grover pode ser empregado da seguinte forma para resolver este problema:

1. Toma-se um registrador quântico de n *qubits* e estado inicial $|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{i=1}^{N} |x_i\rangle$;

2. Aplica-se um operador U_O, chamado oráculo, sobre o registrador. Este operador irá multiplicar os estados $|x_i\rangle$ que satisfazem uma determinada condição por -1;

3. Aplica-se um operador de difusão $U_D = 2|\psi\rangle\langle\psi| - I_n$ sobre o registrador para aumentar a probabilidade de obtenção dos estados $|x_i\rangle$ "marcados" pelo oráculo;

4. Repete-se os passos 2 e 3 até que as probabilidades dos estados marcados sejam ótimas (ou, pelo menos, altas o suficiente). Considerando que há uma quantidade s de estados marcados pelo oráculo, a quantidade ótima de iterações dos passos 2 e 3 é $\left\lfloor \frac{\pi}{4} \sqrt{\frac{N}{s}} \right\rfloor$.

O operador de difusão U_D pode ser implementado em forma de circuito quântico, considerando U_ψ o operador que transforma $|0\rangle^{\otimes n}$ em $|\psi\rangle$, como mostra a seguinte figura:

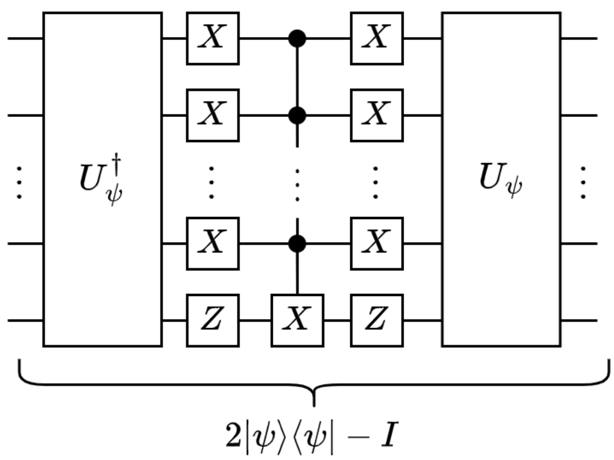

Figura 17 – Implementação do operador de difusão de Grover em forma de circuito quântico.

Já o oráculo U_O não tem uma formula geral de implementação. No presente trabalho, os oráculos são circuitos quânticos construídos para, a princípio, resolver expressões compostas por operadores lógicos, aritméticos e relacionais, e marcar os estados do espaço de busca que representem soluções. Neste caso, o circuito quântico do oráculo é uma composição dos circuitos apresentados nas seções anteriores deste capítulo. Um exemplo de oráculo criado para marcar as soluções da expressão $2a+b=7$ é apresentado na figura a seguir:

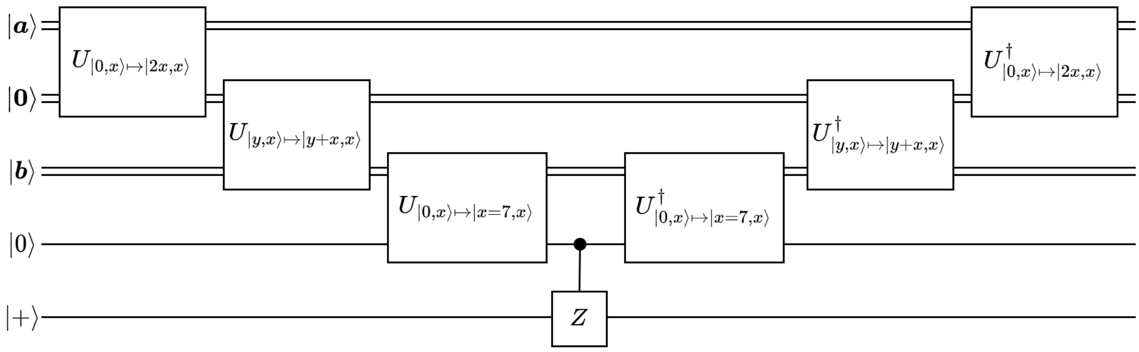

Figura 18 – Exemplo de oráculo que marca as soluções da expressão $2a+b=7$

Observe que o circuito apresentado acima é composto pelos seguintes operadores aritméticos e relacionais: $U_{|0,x\rangle\mapsto|2x,x\rangle}$ (multiplicação por constante), $U_{|y,x\rangle\mapsto|y+x,x\rangle}$ (soma registrador-por-registrador) e $U_{|0,x\rangle\mapsto|x=7,x\rangle}$ (igualdade registrador-por-constante). Além desses operadores, há uma porta lógica *Controlled-Z*. Uma vez que o papel do operador $U_{|0,x\rangle\mapsto|x=7,x\rangle}$ é atribuir um resultado ao *qubit* alvo invertendo o estado deste de $|0\rangle$ para $|1\rangle$, quem se encarrega de multiplicar o estado geral da sequência de registradores por -1 é a porta *Controlled-Z*. Ignorando o último *qubit* (aquele inicializado como $|+\rangle$), o procedimento do circuito acima consiste em 5 etapas:

1. transformar o estado inicial $|\boldsymbol{a},\boldsymbol{0},\boldsymbol{b},0\rangle$ em $|\boldsymbol{a},2\boldsymbol{a},\boldsymbol{b},0\rangle$ usando o operador $U_{|0,x\rangle\mapsto|2x,x\rangle}$;

2. transformar o estado $|\boldsymbol{a},2\boldsymbol{a},\boldsymbol{b},0\rangle$ em $|\boldsymbol{a},2\boldsymbol{a},2\boldsymbol{a}+\boldsymbol{b},0\rangle$ usando o operador $U_{|y,x\rangle\mapsto|y+x,x\rangle}$;

3. transformar o estado $|\boldsymbol{a},2\boldsymbol{a},2\boldsymbol{a}+\boldsymbol{b},0\rangle$ em $|\boldsymbol{a},2\boldsymbol{a},2\boldsymbol{a}+\boldsymbol{b},1\rangle$ caso $2a+b=7$, e $|\boldsymbol{a},2\boldsymbol{a},2\boldsymbol{a}+\boldsymbol{b},0\rangle$ caso $2a+b\neq 7$, usando o operador $U_{|0,x\rangle\mapsto|x=7,x\rangle}$;

4. Aplicar a porta *Controlled-Z* para multiplicar o estado conjunto de todos os registradores do circuito por -1 caso $2a+b=7$;

5. Reverter os passos 1, 2 e 3.

No caso em que se deseja resolver a equação $2a+b=7$ para, por exemplo, $a\in\{1,3,5\}$ e $b\in\{2,4,6\}$, o primeiro e o terceiro registrador da figura 18 devem ser inicializados como, respectivamente, $|\boldsymbol{a}\rangle=\frac{|\mathbf{1}\rangle+|\mathbf{3}\rangle+|\mathbf{5}\rangle}{\sqrt{3}}$ e $|\boldsymbol{b}\rangle=\frac{|\mathbf{2}\rangle+|\mathbf{4}\rangle+|\mathbf{6}\rangle}{\sqrt{3}}$.

Observe também que o circuito quântico da figura 18 é composto, para além dos registradores básicos que armazenam os operandos e o resultado da expressão $2a+b=7$, de *qubits* e registradores auxiliares, comumente chamados de *ancillas* na literatura, responsáveis por armazenar resultados temporários, como $2a$ e $2a+b$. Reduzir a quantidade de registradores e *qubits* auxiliares necessários para resolver uma determinada expressão é um importante desafio de otimização, pois tanto os simuladores como os computadores quânticos da era atual são muito limitados em relação à quantidade de *qubits* com que podem trabalhar. A estratégia adotada no presente trabalho para otimizar o uso de registradores e *qubits* auxiliares é a constante "limpeza" destes. O termo "limpeza" refere-se a reversão de um estado corrente para o estado inicial $|0,..,0\rangle$ após o devido uso da informação temporária armazenada no auxiliar, o que é feito aplicando, de forma inversa, todos os operadores que alteraram seu estado para diferente de $|0,..,0\rangle$.

Capítulo 4

Implementação do compilador *DLQpiler*

Compiladores são ferramentas fundamentais no desenvolvimento de *software* moderno. Estes são, basicamente, ferramentas que traduzem código de alto nível de abstração – ou seja, de fácil interpretação por humanos – em código intermediário ou de máquina, facilitando o trabalho dos programadores. Dentre os exemplos clássicos, pode-se citar os compiladores das linguagens: Fortran, criado pela equipe de John Backus na IBM durante a década de 1950 (BACKUS et al., 1957); COBOL (*Common Business-Oriented Language*), criado por Grace Hopper e sua equipe, também durante a década de 1950 (SAMMET, 1978); C, criado por Dennis Ritchie na década de 1970 (KERNIGHAN; RITCHIE, 1978); e Pascal, criado por Niklaus Wirth na década de 1970 (WIRTH, 2000). Exemplos notáveis de compiladores modernos da computação clássica são: GCC (GNU *Compiller Collection*), uma coleção de compiladores de código aberto para as linguagens C, C++, *Objective-C*, Ada e Fortran; CLang, um compilador de código aberto baseado no projeto LLVM (*Low Level Virtual Machine*), com suporte às linguagens C, C++, *Objective-C* e *Swift*; e Javac (Java *Compiler*), o compilador da linguagem Java, que traduz o código de alto-nível em *bytecode* para a Máquina Virtual Java (JVM).

O compilador *DLQpiler* é um dos principais frutos do presente trabalho. Escrito em linguagem Python, o compilador faz uso do SDK de computação quântica Qiskit, desenvolvido pela IBM, e da biblioteca PLY, responsável pela análise léxica e sintática dos códigos fontes em DLQ. O processo de compilação do código DLQ se dá pelas seguintes etapas:

1. **análise léxica**: nesta etapa, os lexemas da linguagem (*tokens*) são reconhecidos no texto e sequenciados;

2. **análise sintática**: nesta etapa, verifica-se a acordância da sequência de lexemas com a gramática da linguagem e, concomitantemente, gera-se uma estrutura de árvore sintática;

3. **síntese**: nesta etapa, um código intermediário – que, neste caso, é um circuito quântico – é gerado a partir de uma análise recursiva da árvore sintática.

Ao longo deste capítulo, serão apresentados: o paradigma, a sintaxe e a estrutura do compilador *DLQpiler*.

4.1 Paradigma

Como já dito em capítulos anteriores, DLQ é uma linguagem declarativa, o que quer dizer que o código de alto nível é uma descrição do problema a ser resolvido, e não necessariamente o passo-a-passo da resolução do problema, como no caso das linguagens imperativas. Em geral, linguagens declarativas não admitem estados mutáveis nem estruturas de controle: é o caso da DLQ. Esta linguagem, em sua atual versão, é pensada para ser utilizada na resolução de problemas de satisfatibilidade envolvendo expressões algébricas simples. Os principais elementos da linguagem são:

- **Variáveis** - objetos matemáticos que são definidos em termos de conjuntos de pertinência ou de expressões algébricas, e que não necessariamente possuem valor definido até que o programa seja executado;

- **Operadores** - símbolos que compõem as expressões algébricas, podendo ser aritméticos, relacionais ou lógicos;

- **Terminadores** - sentença final do código que especifica o problema a ser resolvido.

Para elucidar o funcionamento da linguagem, um exemplo: deseja-se encontrar dois números primos de 0 a 10 que, se multiplicados, resultam em 15. Para resolver este problema, utiliza-se o seguinte código DLQ:

```
p1[4] in {2, 3, 5, 7};
p2[4] in {2, 3, 5, 7};
y[1] := p1*p2=15;
amplify y 2 times
```

O código acima é composto pelas variáveis `p1`, `p2` e `y`. `p1` e `p2` possuem, ambas, 4 bits, e são definidas em termos de conjuntos de pertinência. As definições dessas duas primeiras variáveis indicam que elas podem valer 2, 3, 5 ou 7. Já a variável `y` possui apenas 1 bit e é definida como a expressão `p1*p2=15`. A última linha contém o terminador *amplify*, indicando que o algoritmo de busca de Grover deve ser aplicado com 2 iterações para buscar os valores de `p1` e `p2` para os quais `y=1`. A síntese deste código resultará em um circuito quântico em que `p1`, `p2` e `y` são registradores de, respectivamente, 4, 4 e 1 *qubits*, havendo também uma série de *qubits* auxiliares. No próximo capítulo, os possíveis resultados deste código, bem como de outros códigos, serão apresentados.

4.2 Gramática

Linguagens de programação são idiomas formais com gramáticas bem definidas. De acordo com a hierarquia de Chomsky, as gramáticas formais são classificadas como: regulares (tipo 3), livres de contexto (tipo 2), sensíveis ao contexto (tipo 1) e, por fim, com estruturas de frase (tipo 0).

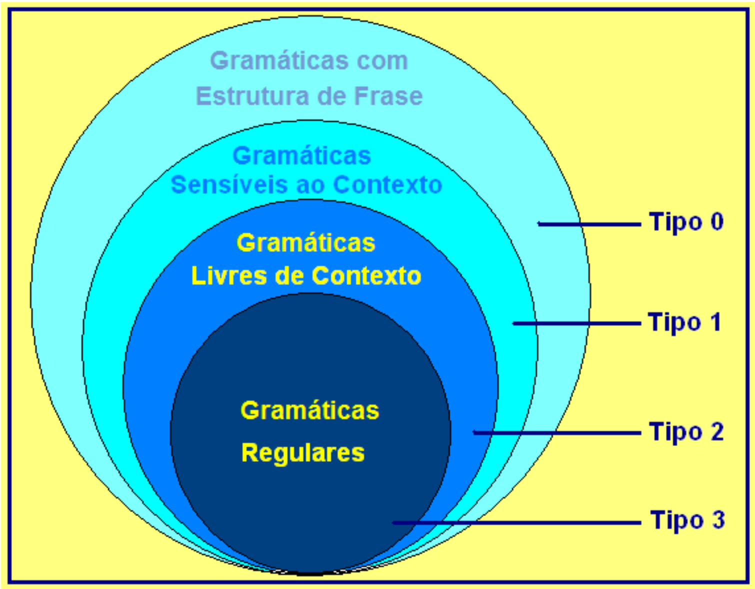

Figura 19 – Representação diagramática da hierarquia de Chomsky. Imagem retirada de Wikimedia Commons, de autoria de Ricardo Ferreira de Oliveira, sob licença CC BY-SA 3.0 <https://creativecommons.org/licenses/by-sa/3.0/>

O compilador *DLQpiler* implementa uma gramática livre de contexto para a análise sintática e um conjunto de gramáticas regulares para a análise léxica.

4.2.1 Gramáticas regulares e análise léxica

Gramática regular é um conceito fundamental da teoria de linguagens formais. Estas gramáticas descrevem conjuntos de *strings* (sequências de caracteres) que podem ser reconhecidas e geradas por máquinas de estados finitos. Em um compilador, a definição de uma gramática regular se dá, em geral, por uma expressão regular.

O dicionário de símbolos e expressões regulares do analisador léxico do compilador *DLQpiler* é o seguinte:

Nome	Símbolo ou expressão regular
NUMBER	`[0-9]+`
ID	`[a-zA-Z_][a-zA-Z_0-9]*`
PLUS	`+`
MINUS	`-`
MUL	`*`
DIVIDE	`/`
HAT	`^`
EQUAL	`=`
NEQ	`!=`
LT	`<`
GT	`>`
COMMA	`,`
ASSIGN	`:=`
LPAREN	`(`
RPAREN	`)`
LCURLY	`{`
RCURLY	`}`
LBRACKET	`[`
RBRACKET	`]`
SEMICOLON	`;`

Tabela 2 – Dicionário de símbolos e expressões regulares do *DLQpiler*

A expressão regular `NUMBER` engloba todas as *strings* formadas por dígitos de 0 a 9, enquanto `ID` (identificador) engloba as strings que iniciam com letras ou *underline*, e prosseguem com letras, *underlines* e dígitos de 0 a 9.

Há também um conjunto de palavras reservadas da linguagem, que são reconhecidas como `ID` pelo analisador léxico, mas que fazem parte da sintaxe da linguagem e não devem ser usadas para nomear variáveis. São elas: `in`, `and`, `or`, `not`, `true`, `false`, `amplify` e `times`.

4.2.2 Gramáticas livre de contexto e analise sintática

Uma gramática livre de contexto é um sistema formal usado para descrever a estrutura e a sintaxe de uma linguagem, consistindo em um conjunto de regras de produção que define como sentenças válidas podem ser construídas nesta linguagem.

Gramáticas livres de contexto são constituídas por sequências de regras de produção na forma $A \to \alpha$, onde A é um símbolo não-terminal e α é uma cadeia de símbolos terminais ou não-terminais. Uma gramática livre de contexto composta por múltiplas regras de produção é analisada de forma recursiva, resultando em um grafo-árvore com símbolos do texto original. Veja o seguinte exemplo:

$$C \rightarrow B\text{é}A$$
$$B \rightarrow \text{Fulano}|\text{Ciclano}|\text{Beltrano}$$
$$A \rightarrow \text{cuiabano}|\text{mineiro}|\text{paulista}|\text{carioca}$$

As regras de produção descritas acima formam uma gramática livre de contexto que contém frases como “Fulano é cuiabano”, “Ciclano é mineiro”, “Beltrano é carioca”, entre outras. Repare que, na regra de produção $C \rightarrow B\text{é}A$, B e A são símbolos não-terminais que representam, respectivamente, um nome e um gentílico, enquanto os símbolos terminais da gramática são “Fulano”, “Ciclano”, “Beltrano”, “cuiabano”, “mineiro”, “paulista”, “carioca” e “é”. Uma análise da frase “Fulano é cuiabano”, com base nesta gramática, resulta no seguinte grafo-árvore:

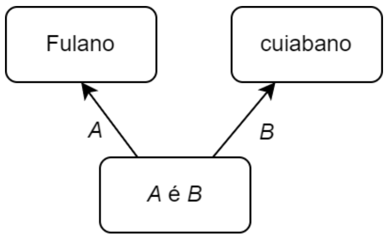

Figura 20 – Árvore resultante da análise da frase ”Fulano é cuiabano” de acordo com uma gramática livre de contexto

No contexto das linguagens computacionais, esse tipo de grafo é denominado *Abstract Syntax Tree* (AST), ou simplesmente árvore sintática. Ferramentas de análise sintática automática, como a biblioteca PLY, são utilizadas para gerar ASTs a partir de *strings*. Nestas ferramentas, a gramática é descrita usando o formalismo de Backus-Naur (com pequenas variações), onde as regras de produção são grafadas na forma “`<símbolo> ::= <cadeia>`”. O exemplo de gramática apresentado acima fica, no formalismo de Backus-Naur, da seguinte forma:

```
<C> ::= <B> "é" <A>
<B> ::= "Fulano" | "Ciclano" | "Beltrano"
<A> ::= "Cuiabano" | "Mineiro" | "Paulista" | "Carioca"
```

A gramática livre de contexto da linguagem DLQ, no formalismo de Backus-Naur, é a seguinte:

```
<fullcode> ::= <regdefseq> <amplifyterm>
<amplifyterm> ::= "amplify" ID NUMBER "times"
<regdefseq> ::= <regdef> ";" <regdefseq> | <regdef> ";"
<regdef> ::= <regdefs> | <regdefx>
```

```
<regdefs> ::= ID "[" NUMBER "]" "in" "{" <expseq> "}"
<regdefx> ::= ID "[" NUMBER "]" ":=" <expression>
<expseq> ::= <expseq> "," <expression> | <expression>
<expression> ::= <expression> "or" <expression>
             | <expression> "and" <expression>
             | "not" <expression>
             | <expression> "=" <expression>
             | <expression> "!=" <expression>
             | <expression> "<" <expression>
             | <expression> ">" <expression>
             | <expression> "+" <expression>
             | <expression> "-" <expression>
             | <expression> "*" <expression>
             | <expression> "^" <expression>
             | <expression> "/" <expression>
             | "-" <expression> (unary minus)
             | "(" <expression> ")"
             | "false"
             | "true"
             | NUMBER
             | ID
```

Esta gramática, em particular, requer um recurso adicional denominado Tabela de Precedência, que define a ordem de prioridade dos operadores e as regras de associatividade destes. A tabela de precedência da linguagem DLQ é a seguinte:

Prioridade	Operadores	Associatividade
9ª	or	esquerda
8ª	and	esquerda
7ª	not	direita
6ª	<, >	esquerda
5ª	=, !=	esquerda
4ª	+, -	esquerda
3ª	*, /	esquerda
2ª	- (*unary minus*)	direita
1ª	^	direita

Tabela 3 – Tabela de precedência da linguagem DLQ

4.3 Estrutura do compilador

O código-fonte do compilador é composto pelos seguintes módulos funcionais:

- **utils** - pequeno conjunto de recursos que são utilizados em todo o código;
- **lexer** - módulo onde a tabela de símbolos e expressões regulares é definida;
- **parser** - módulo onde as regras de produção da gramática e os procedimentos de geração de AST são definidos;
- **ast** - módulo onde definem-se as classes que representam os nós da AST;
- **qunits** - módulo onde os circuitos quânticos aritméticos, lógicos e relacionais apresentados no capítulo 3 são implementados;
- **synth** - módulo responsável pela síntese dos circuitos quânticos finais;
- **main** - módulo raiz do compilador, onde os pontos de entrada do usuário são definidos.

Este código fonte é parcialmente orientado a objetos, especialmente nos módulos **ast** e **synth**. A orientação a objetos é utilizada para tornar o código mais organizado, fácil de compreender e fácil de modificar. No módulo **ast**, os nós das árvores sintáticas geradas pelo analisador sintático são definidos como classes, e cada classe contém um pequeno conjunto de métodos recursivos de geração de circuito quântico. A hierarquia das classes do módulo **ast** é apresentada no diagrama da figura 21. As descrições dessas classes são as seguintes:

- ***ASTNode*** - A superclasse que engloba, em geral, os nós de AST. Esta classe é abstrata, ou seja, não deve ser instanciada diretamente. Seu principal atributo é a propriedade *line*, que armazena a numeração da linha de código correspondente do nó. Herda apenas **object**.
- ***Expression*** - A classe dos nós de expressões aritméticas, lógicas e relacionais. Herda a classe ***ASTNode***. Seus principais métodos são:

  get_leafs - método recursivo que retorna todos os identificadores usados na expressão;

  n_result_qubits - método recursivo que calcula a quantidade de *qubits* necessária para armazenar o resultado da expressão;

  alloc_result_qubits - método que aloca *qubits* auxiliares no circuito quântico para armazenar o resultado da expressão;

  release_result_qubits - método que libera os *qubits* de resultados quando já foram utilizados e seus estados já foram revertidos para $|0, ..., 0\rangle$;

  pre_build - método recursivo que faz a preparação da árvore para que os métodos ***n_result_qubits***, ***build*** e ***reverse*** possam ser utilizados;

build - método recursivo que gera um circuito quântico para resolver a expressão da árvore;

reverse - método que gera o circuito inverso do que é gerado pelo método *build*, necessário para reverter os estados dos *qubits* de resultado para $|0, ..., 0\rangle$.

- ***Identifier*** - Classe que representa os identificadores (símbolos terminais que nomeiam variáveis). Herda ***Expression***. Seu principal atributo é ***label***, que armazena o símbolo terminal em forma de texto;
- ***Parenthesis*** - Classe que representa os parênteses das expressões. Herda ***Expression***. Seu principal atributo é ***inner_expr***, que armazena o objeto ***Expression*** da expressão interna.
- ***ArithmeticExpression*** - Classe que representa os nós de expressões aritméticas. Herda ***Expression***.
- ***UnaryMinus*** - Classe que representa os usos unários do operador *minus* ("-"). Herda ***ArithmeticExpression***.
- ***Power*** - Classe que representa as operações de potenciação. Herda a classe ***ArithmeticExpression***. Seus principais atributos são: ***base_expr***, que contém o nó raiz da expressão base, e ***exponent***, que contém o expoente inteiro.
- ***Product*** - Classe que representa os produtórios (produtos de multiplos operandos). Herda ***ArithmeticExpression***. Seus principais atributos são: ***operands***, que armazena uma lista de nós operandos, e ***exponents***, que armazena uma lista de expoentes inteiros.
- ***Summation*** - Classe que representa os somatórios. Herda ***ArithmeticExpression***. Seus principais atributos são: ***operands***, que armazena uma lista de nós operandos, e ***signals***, que armazena uma lista de sinais ("+" ou "-").
- ***RelationalExpression*** - Classe que representa as operações relacionais. Herda ***Expression***. Seus principais atributos são: ***left***, que armazena o nó raiz da expressão à esquerda do operador relacional, e ***right***, que armazena o nó raiz da expressão à direita do operador relacional.
- ***Equal***, ***NotEqual***, ***LessThan*** e ***GreaterThan*** - Classes que representam os operadores relacionais. Herdam ***RelationalExpression***.
- ***LogicExpression*** - Classe que representa as operações lógicas (booleanas). Herda ***Expression***.
- ***Not*** - Classe que representa o operador lógico *not*. Herda ***LogicExpression***. Seu principal atributo é ***operand***, que armazena o nó raiz do operando.

- ***And*** e ***Or*** - Classes que representam os operadores lógicos *and* e *or*. Herdam ***LogicExpression***. Possuem, ambas, o atributo ***operands***, que contém uma lista de 2 ou mais operandos.

- ***RegisterDefinition*** - Classe que representa um *statement* de definição de variável. Herda ***ASTNode***. Seus atributos são: ***name***, que contém o nome da variável definida, e ***n***, que contém o tamanho da variável em ***qubits***.

- ***RegisterExpressionDefinition*** - Classe que representa a definição de uma variável como uma expressão. Herda ***RegisterDefinition***. Seus principais métodos são: ***build***, que gera o circuito quântico da expressão, e ***reverse***, que gera o circuito quântico inverso da expressão. Seu principal atributo é ***expr***, que contém o nó raiz da expressão.

- ***RegisterSetDefinition*** - Classe que representa a definição de uma variável em termos de um conjunto de valores. Herda ***RegisterDefinition***. Seu principal atributo é ***values***, que contém os valores do conjunto de pertinência da variável.

- ***Terminator*** - Classe que representa os terminadores. Herda ***ASTNode***.

- ***Amplify*** - Classe que representa o terminador *amplify*. Herda ***Terminator***. Seus principais atributos são: ***target***, que contém o nome a variável alvo da busca de Grover, e ***it***, que contém o número de iterações da busca de Grover.

- ***FullCode*** - Classe que representa o nó raiz da AST. Herda ***ASTNode***. Seus principais atributos são: ***regdefseq***, que contém uma lista de nós do tipo ***RegisterDefinition***, e ***terminator***, que contém um nó do tipo ***Terminator***.

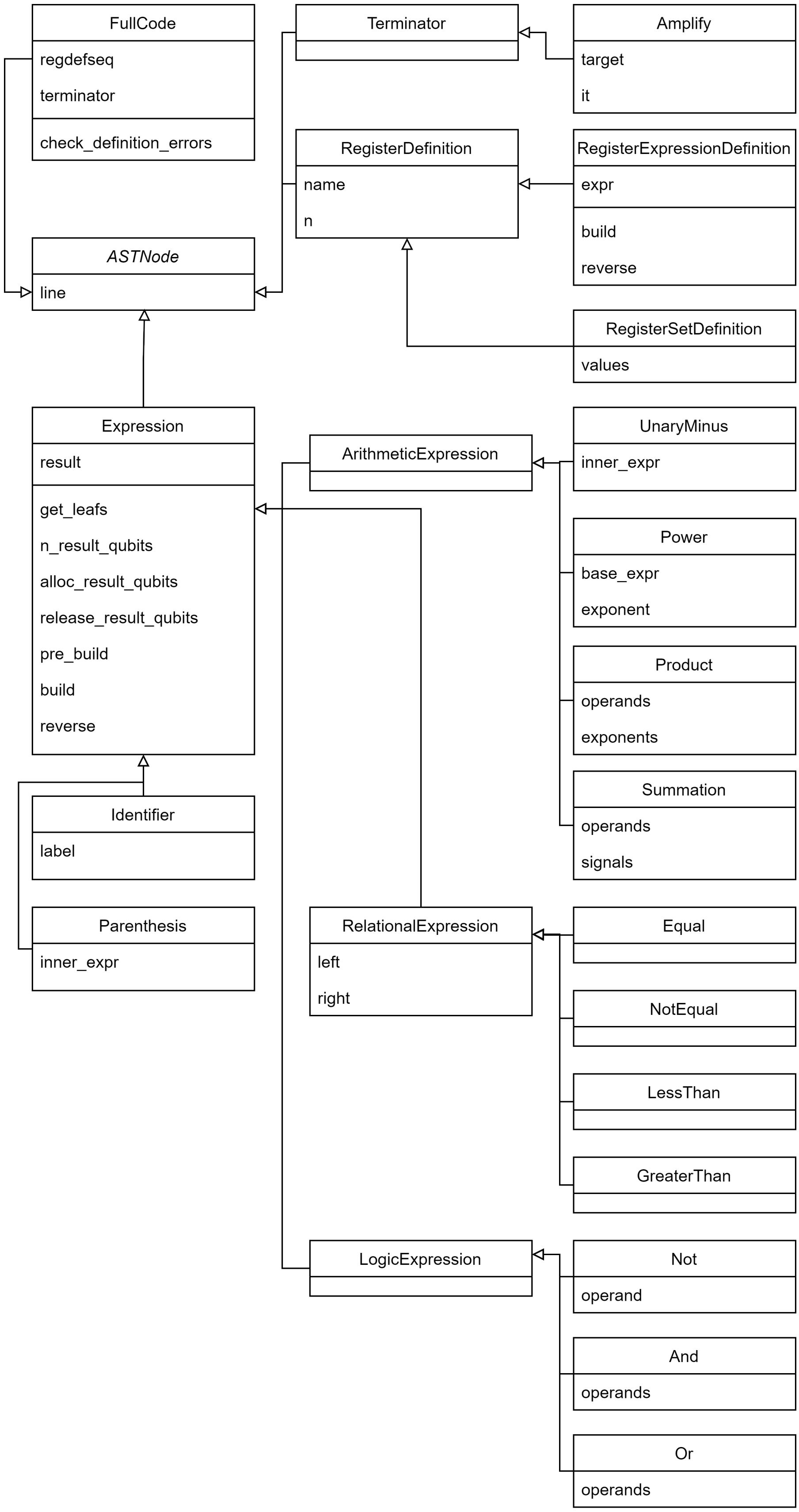

Figura 21 – Diagrama de classes do módulo **ast**

O módulo **synth** contém a classe ***QuantumEvaluator***. Enquanto os métodos ***build*** e ***reverse*** da classe ***Expression*** são responsáveis pela geração dos circuitos aritméticos, lógicos e relacionais, que constituem os oráculos da busca de Grover, os métodos da classe ***QuantumEvaluator*** são responsáveis por construir as buscas de Grover completas, além de gerenciar os recursos do circuito quântico como um todo.

O código fonte completo do compilador *DLQpiler* está presente nos anexos desta monografia.

4.4 Processo de síntese de circuito quântico

Como já mencionado na seção anterior, a geração dos circuitos quânticos é feita pelos métodos ***build*** das classes ***Expression*** e ***QuantumEvaluator***, sendo que, na classe ***Expression***, este método é responsável pela construção do circuito que resolve uma expressão, enquanto na classe ***QuantumEvaluator***, o método é responsável por construir a busca de Grover completa. De modo superficial, o método ***build*** da classe ***QuantumEvaluator*** pode ser descrito da seguinte forma:

1. Crie um *qubit* auxiliar chamado *phase* e aplique sobre ele uma porta Hadamard.

2. Para cada nó do tipo ***RegisterDefinition*** na AST, faça:

    a) Se o nó é do tipo ***RegisterSetDefinition***, com especificação de tamanho n e conjunto de valores $\{v_1, v_2, ..., v_m\}$, crie um registrador quântico de n *qubits* e, sobre ele, aplique um operador unitário para transformar seu estado inicial $|0, ..., 0\rangle$ em $|\psi\rangle = \frac{1}{\sqrt{m}}(|v_1\rangle + |v_2\rangle + ... + |v_m\rangle)$. Para isso, pode-se usar a porta ***StatePreparation***, da plataforma Qiskit.

    b) Se o nó é do tipo ***RegisterExpressionDefinition***, com especificação de tamanho n, crie um registrador de n *qubits*, defina-o como registrador de resultado do nó ***Expression*** interno, execute o método ***pre_build*** do nó interno e, depois, execute o método ***build*** do nó interno para construir o circuito que resolve a expressão.

3. Obtenha o *qubit* mais significativo do registrador correspondente à variável alvo do programa (cujo nome é especificado no atributo ***target*** do nó ***Amplify***) e aplique uma porta Z controlada por este *qubit* sobre o *phase qubit*.

4. Reverta os circuitos gerados no passo 2.

5. Este é o passo de geração do operador de difusão. Sobre cada *qubit* do circuito, exceto o *phase qubit*, aplique uma porta X, e sobre o *phase qubit*, aplique uma porta Z. Em seguida, aplique sobre o *phase qubit* uma porta X controlada por todos os outros *qubits*. Por fim, aplique novamente uma porta Z sobre o *phase qubit* e portas X sobre os demais *qubits*.

6. Considerando que, no nó ***Amplify*** da AST, a especificação de número de iterações contida no atributo ***it*** é n_i, repita os passos 2, 3, 4 e 5 mais $n_i - 1$ vezes.

7. Repita o passo 2.

8. Por fim, adicione portas de medição sobre os registradores não-auxiliares.

A construção do circuito de resolução de expressão, desempenhada pelo método ***Expression.build***, é feita de forma recursiva em intra-ordem – ou seja, para cada nó da expressão na AST, gera-se, primeiro, os circuitos das sub-árvores, e depois gera-se o circuito do nó atual. A figura 22 mostra um exemplo ilustrado de geração de circuito quântico de resolução de expressão a partir de uma AST.

Quando necessária, a alocação dos *qubits* de resultado de cada nó é feita, algumas vezes, no método ***build***, e outras vezes no método ***pre_build***, a depender da classe do nó.

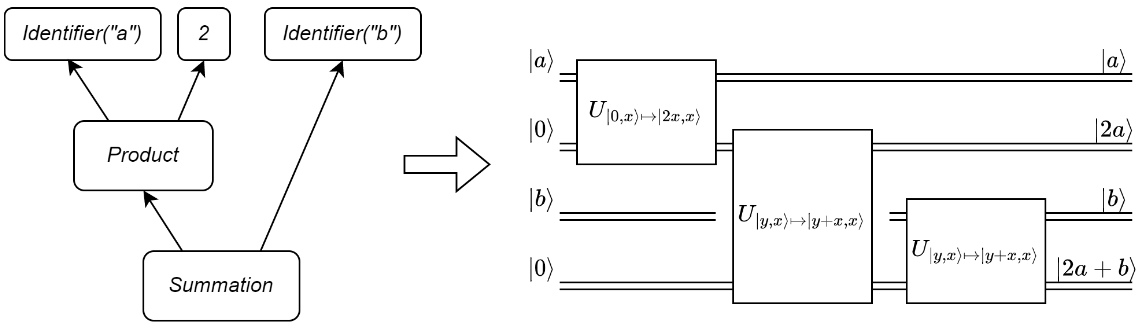

Figura 22 – Ilustração da transformação de uma AST em um circuito quântico para resolver uma expressão

Capítulo 5

Exemplos de códigos

Neste capítulo, dois exemplos simples de aplicação da linguagem DLQ, bem como os resultados obtidos nas simulações dos circuitos quânticos gerados pelo compilador, são apresentados. Estes são exemplos didáticos e também provas de conceito do presente projeto.

5.1 Exemplo 1 - satisfatibilidade booleana

O problema da satisfatibilidade booleana consiste em verificar se uma expressão booleana de n variáveis binárias é satisfatível. Este foi o primeiro problema computacional a ser provado NP-completo. Como exemplo, usemos a expressão $(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee \neg x_4)$. O código para resolver este problema é o seguinte:

```
x1[1] in {false, true};
x2[1] in {false, true};
x3[1] in {false, true};
x4[1] in {false, true};
y[1] := (x1 or not x3 or x4) and (not x2 and x3 and not x4);
amplify y 3 times
```

Repare que, neste código, utiliza-se 3 iterações da busca de Grover. Este número de iterações pode ser calculado a partir da fórmula $\left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{s}} \right\rfloor$, considerando que o tamanho do espaço de busca é $N = 2^4 = 16$ e supondo que o número de soluções é $s = 1$. Mais ou menos iterações que isto resultará em menores probabilidades de uma solução ser encontrada.

Ao fazer a compilação do código e executar o circuito quântico obtido usando o simulador `aer_simulator`, da plataforma Qiskit, com 1024 *shots*, obteve-se a seguinte tabela de resultados:

	x1	x2	x3	x4	y	$freq
0	1	0	1	0	1	931
1	1	1	1	1	0	10
2	0	0	1	0	0	9
3	0	1	1	1	0	9
4	1	1	0	0	0	8
5	1	1	0	1	0	7
6	0	1	1	0	0	7
7	0	1	0	0	0	6
8	0	0	0	1	0	6
9	0	1	0	1	0	6
10	1	0	0	0	0	5
11	1	0	1	1	0	4
12	0	0	1	1	0	4
13	1	1	1	0	0	4
14	0	0	0	0	0	4
15	1	0	0	1	0	4

Tabela 4 – Tabela de resultados do exemplo 1

Cada *shot* é uma execução do circuito no simulador. Em cada uma dessas execuções, uma tupla contendo um valor para cada variável do código é obtida de forma indeterminística (e esse indeterminismo se deve ao fenômeno do colapso do estado quântico). Se o número de iterações especificadas no terminador *amplify* for apropriado, espera-se que a probabilidade de obtenção de uma tupla de valores que representem a solução do problema seja alta. A tabela de resultados mostra as tuplas de valores obtidas em um determinado número de execuções do circuito quântico, cada uma com sua frequência absoluta.

Observe que, de acordo com a tabela, a probabilidade da solução $(x_1, x_2, x_3, x_4) = (1, 0, 1, 0)$ ser obtida como resultado é de, aproximadamente, 91%, o que é um bom resultado.

O circuito quântico resultante da compilação do código deste exemplo é exibido na figura 23, nos anexos.

5.2 Exemplo 2 - fatoração prima

O problema da fatoração prima consiste em encontrar, para um $x \in \mathbb{N}$ qualquer, um par de números primos (p_1, p_2) tal que $p_1 p_2 = x$. Como exemplo, tomemos $x = 15$. O código para resolver este problema é o seguinte:

```
p1[4] in {2, 3, 5, 7};
p2[4] in {2, 3, 5, 7};
y[1] := p1*p2=15;
```

```
amplify y 2 times
```

O número de iterações é calculado considerando um tamanho de espaço de busca $N = 4^2 = 16$ e um número de soluções $s = 2$. A tabela de resultados da simulação do circuito quântico obtido, usando o simulador `aer_simulator`, com 1024 *shots* de simulação, é mostrada a seguir.

	p1	p2	y	$freq
0	5	3	1	474
1	3	5	1	455
2	3	2	0	12
3	3	7	0	12
4	5	7	0	10
5	5	2	0	9
6	2	3	0	9
7	7	5	0	7
8	2	5	0	7
9	2	7	0	6
10	7	7	0	6
11	3	3	0	6
12	7	3	0	4
13	5	5	0	4
14	2	2	0	2
15	7	2	0	1

Tabela 5 – Tabela de resultados do exemplo 2

Repare que, de acordo com a tabela, as soluções $(p_1, p_2) = (5, 3)$ e $(p_1, p_2) = (3, 5)$ são obtidas com, respectivamente, 46% e 44% de frequência relativa. Considerando que as duas soluções, juntas, têm 90% de frequência relativa, é um bom resultado.

O circuito quântico resultante da compilação do código deste exemplo é exibido na figura 24, nos anexos.

Capítulo 6

Considerações finais

O presente trabalho apresenta um projeto de linguagem de programação de aplicação específica, de fácil aprendizagem e fácil interpretação, para o desenvolvimento de programas de computador quântico. O projeto é, a princípio, simples e de aplicabilidade limitada, porém extensível, e portanto deve ser visto apenas como um protótipo. Dentre as melhorias que podem ser feitas na linguagem, pode-se citar as seguintes:

- **Mais tipos de dados** - na versão atual, a linguagem não conta com um sistema de tipos, de modo que o único tipo de dado reconhecido é o inteiro sem sinal (natural). Uma das principais melhorias que podem ser feitas na linguagem é a implementação de um sistema de tipos forte e estático, bem como a adição de mais tipos de dados, como inteiros com sinal, números com ponto flutuante e *strings*.

- **Mais operadores** - a versão atual da linguagem conta apenas com as operações de adição (+), subtração (-), multiplicação (*), potenciação (^), igualdade (=), não-igualdade ($\neq$), maior-que (>), menor-que (<), *not*, *or* e *and*. Uma melhoria que pode ser feita é o acréscimo de mais operadores. Exemplos de operadores numéricos que podem ser adicionados são: exponenciação (que, diferente da potenciação, tem expoente variável), valor absoluto, divisão, raiz, seno, cosseno, etc.

- **Elementos de programação funcional** - recursos como funções puras, listas, tuplas, operadores ternários (como *if* e *for*), e operadores de processamento de listas, como *map*, *reduce* e *filter*, podem ser incorporados à linguagem DLQ em versões futuras.

- **Mais terminadores** - o único terminador presente na versão atual da linguagem é o *amplify*, que é uma indicação de que o algoritmo de busca de Grover deve ser aplicado, com especificação manual de número de iterações. Desse modo, a única aplicação da linguagem é a resolução de problemas de satisfatibilidade. No entanto, mais terminadores, baseados em outros algoritmos quânticos conhecidos, podem ser incorporados para ampliar o leque de aplicações da DLQ. Por exemplo: é possível criar um terminador de contagem de soluções, ou mesmo um terminador de busca de soluções sem especificação manual de número de iterações, tomando como base o algoritmo quântico de contagem de Brassard et. al (1998). Outra possibilidade é a incorporação de terminadores de otimização, busca de valor mínimo e busca de valor máximo, baseando-se em algoritmos como *Quantum Approximate Optimization Algorithm* (QAOA) (FARHI; GOLDSTONE; GUTMANN, 2014), *Grover Adaptive Search* (GAS) (GILLIAM; WOERNER; GONCIULEA, 2021), ou o algoritmo de busca de valor mínimo de Durr e Hoyer (1999).

- **Mais otimização** - uma das principais qualidades de um compilador é seu poder de otimização (ou seja, sua capacidade de minimizar a complexidade, o número de instruções e o consumo de memória de um programa). O compilador *DLQpiler* tem otimizações modestas, e, portanto, a incorporação de técnicas mais sofisticadas de otimização é uma possibilidade relevante de melhoria.

- **Robustez ao ruído** - na versão atual do *DLQpiler*, não há métodos de redução e mitigação de ruídos quânticos (decoerência) implementados, fazendo com que o uso deste só seja possível em ambientes ideais simulados. Para que a linguagem DLQ se torne útil na prática, é fundamental que métodos de mitigação e redução de ruídos sejam adicionados ao compilador.

Conclui-se que o projeto DLQ ainda não tem o nível de maturidade necessário para o uso comercial, mas já tem prova de conceito, e pode ser útil para fins acadêmicos. Sua maturação dependerá de seu desenvolvimento contínuo e da formação de uma comunidade de usuários e contribuidores. Espera-se que, nos próximos anos, a demanda por tecnologias baseadas em computação quântica aumente, os sistemas computacionais quânticos evoluam e tornem-se mais acessíveis, e, por conseguinte, o interesse por ferramentas como a linguagem DLQ cresça, favorecendo a formação de uma comunidade em torno do projeto, ou até mesmo de projetos similares.

Uma forma de medir a maturidade de um projeto de engenharia é a escala *Technology Readiness Level* (TRL), desenvolvida na NASA (TZINIS, 2015). Os níveis de maturidade, nesta escala, são definidos da seguinte forma:

- TRL1 - princípios básicos observados e reportados;

- TRL2 - conceito e aplicação da tecnologia formulados;

- TRL3 - prova experimental de conceito;

- TRL4 - tecnologia validada em ambiente simulado;

- TRL5 - tecnologia validada em ambiente relevante;

- TRL6 - protótipo do sistema demonstrado em ambiente relevante;

- TRL7 - demonstração do protótipo do sistema em ambiente operacional;
- TRL8 - sistema completo e qualificado;
- TRL9 - sistema real comprovado em ambiente operacional.

Na escala TRL, o compilador *DLQpiler* está situado em TRL3, uma vez que tem provas experimentais de conceito, mas ainda não é perfeito em ambiente simulado, havendo *bugs* para serem corrigidos e testes mais rigorosos a serem feitos.

Atualmente, há diversos serviços de computação quântica em nuvem (como o IBM *Quantum* e o AWS *Braket*). No entanto, os sistemas de computação quântica mais avançados, como o IBM *Osprey*, ainda são de acesso restrito, fazendo com que os testes do *DLQpiler* se limitem às simulações, o que dificulta o desenvolvimento da linguagem DLQ e faz com que seja inviável implementar recursos muito específicos e custosos a curto prazo. A maturação do projeto DLQ depende diretamente da maturação da computação quântica em geral.

Referências

AHARONOV, D.; BEN-OR, M. Fault-tolerant quantum computation with constant error. In: **Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing**. New York, NY, USA: Association for Computing Machinery, 1997. (STOC '97), p. 176–188. ISBN 0897918886. Disponível em: <https://doi.org/10.1145/258533.258579>.

BACKUS, J. W. et al. The fortran automatic coding system. In: **Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability on - IRE-AIEE-ACM '57 (Western)**. Los Angeles, California: ACM Press, 1957. p. 188–198. Disponível em: <http://portal.acm.org/citation.cfm?doid=1455567.1455599>.

BENIOFF, P. Quantum mechanical hamiltonian models of turing machines. **Journal of Statistical Physics**, v. 29, n. 3, p. 515–546, 1982.

BRASSARD, G.; HØYER, P.; TAPP, A. Quantum counting. In: LARSEN, K. G.; SKYUM, S.; WINSKEL, G. (Ed.). **Automata, Languages and Programming**. Berlin, Heidelberg: Springer, 1998. (Lecture Notes in Computer Science), p. 820–831. ISBN 978-3-540-68681-1.

CHILDS, A. M. et al. Exponential algorithmic speedup by a quantum walk. **Proceedings of the thirty-fifth ACM symposium on Theory of computing - STOC '03**, 2003.

COPPERSMITH, D. **An approximate Fourier transform useful in quantum factoring**. 1994.

D'ALESSANDRO, D.; DAHLEH, M. Optimal control of two-level quantum systems. **IEEE Transactions on Automatic Control**, v. 46, n. 6, p. 866–876, 2001.

DEUTSCH, D. Quantum theory, the church–turing principle and the universal quantum computer. **Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences**, v. 400, n. 1818, p. 97–117, Jul 1985.

DEUTSCH, D.; JOZSA, R. Rapid solution of problems by quantum computation. **Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences**, v. 439, n. 1907, p. 553–558, Dec 1992.

DRAPER, T. G. **Addition on a Quantum Computer**. 2000.

DURR, C.; HOYER, P. A quantum algorithm for finding the minimum. arXiv, n. arXiv:quant-ph/9607014, Jan 1999. ArXiv:quant-ph/9607014. Disponível em: <http://arxiv.org/abs/quant-ph/9607014>.

FARHI, E.; GOLDSTONE, J.; GUTMANN, S. **A Quantum Approximate Optimization Algorithm**. arXiv, 2014. Disponível em: <https://arxiv.org/abs/1411.4028>.

FEYNMAN, R. P. Simulating physics with computers. **International Journal of Theoretical Physics**, v. 21, n. 6-7, p. 467–488, 1982.

GILLIAM, A.; WOERNER, S.; GONCIULEA, C. Grover adaptive search for constrained polynomial binary optimization. **Quantum**, Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften, v. 5, p. 428, Apr 2021.

GROVER, L. K. A fast quantum mechanical algorithm for database search. **Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96**, 1996.

HEIM, B. et al. Quantum programming languages. **Nature Reviews Physics**, v. 2, n. 12, p. 709–722, 2020.

HUANG, H.-L. et al. Superconducting quantum computing: A review. **Science China Information Sciences**, v. 63, n. 8, 2020.

HäFFNER, H.; ROOS, C.; BLATT, R. Quantum computing with trapped ions. **Physics Reports**, v. 469, n. 4, p. 155–203, 2008. ISSN 0370-1573. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0370157308003463>.

KERNIGHAN, B. W.; RITCHIE, D. M. **The C programming language**. 1978.

KOK, P. et al. Linear optical quantum computing with photonic qubits. **Rev. Mod. Phys.**, American Physical Society, v. 79, p. 135–174, Jan 2007. Disponível em: <https://link.aps.org/doi/10.1103/RevModPhys.79.135>.

LEE, R. Accelerating multimedia with enhanced microprocessors. **IEEE Micro**, v. 15, n. 2, p. 22–32, 1995.

LLOYD, S.; MOHSENI, M.; REBENTROST, P. **Quantum algorithms for supervised and unsupervised machine learning**. arXiv, 2013. Disponível em: <https://arxiv.org/abs/1307.0411>.

NIELSEN, M. A.; CHUANG, I. L. **Quantum Computation and Quantum Information**. [S.l.]: Cambridge University Press, 2000.

PRESKILL, J. **Fault-tolerant quantum computation**. arXiv, 1997. Disponível em: <https://arxiv.org/abs/quant-ph/9712048>.

______. Quantum computing in the nisq era and beyond. **Quantum**, v. 2, p. 79, 2018.

RUIZ-PEREZ, L.; GARCIA-ESCARTIN, J. C. Quantum arithmetic with the quantum fourier transform. **Quantum Information Processing**, v. 16, n. 6, p. 152, Apr 2017. ISSN 1573-1332.

SAMMET, J. E. The early history of cobol. In: _____. **History of programming languages**. New York, NY, USA: Association for Computing Machinery, 1978. p. 199–243. ISBN 978-0-12-745040-7. Disponível em: <https://doi.org/10.1145/800025.1198367>.

SHOR, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. **SIAM Journal on Computing**, v. 26, n. 5, p. 1484–1509, 1997.

TZINIS, I. **Technology Readiness Level**. Brian Dunbar, 2015. Disponível em: <http://www.nasa.gov/directorates/heo/scan/engineering/technology/technology_readiness_level>.

WETERING, J. van de. Zx-calculus for the working quantum computer scientist. arXiv, n. arXiv:2012.13966, Dec 2020. ArXiv:2012.13966 [quant-ph]. Disponível em: <http://arxiv.org/abs/2012.13966>.

WIRTH, N. The development of procedural programming languages personal contributions and perspectives. In: **Modular Programming Languages**. Springer, Berlin, Heidelberg, 2000. p. 1–10. Disponível em: <https://link.springer.com/chapter/10.1007/10722581_1>.

Anexos

ANEXO A

Circuitos quânticos dos exemplos

Figura 23 – Circuito quântico resultante da compilação do código fonte do exemplo 1 (atente-se à numeração das partes)

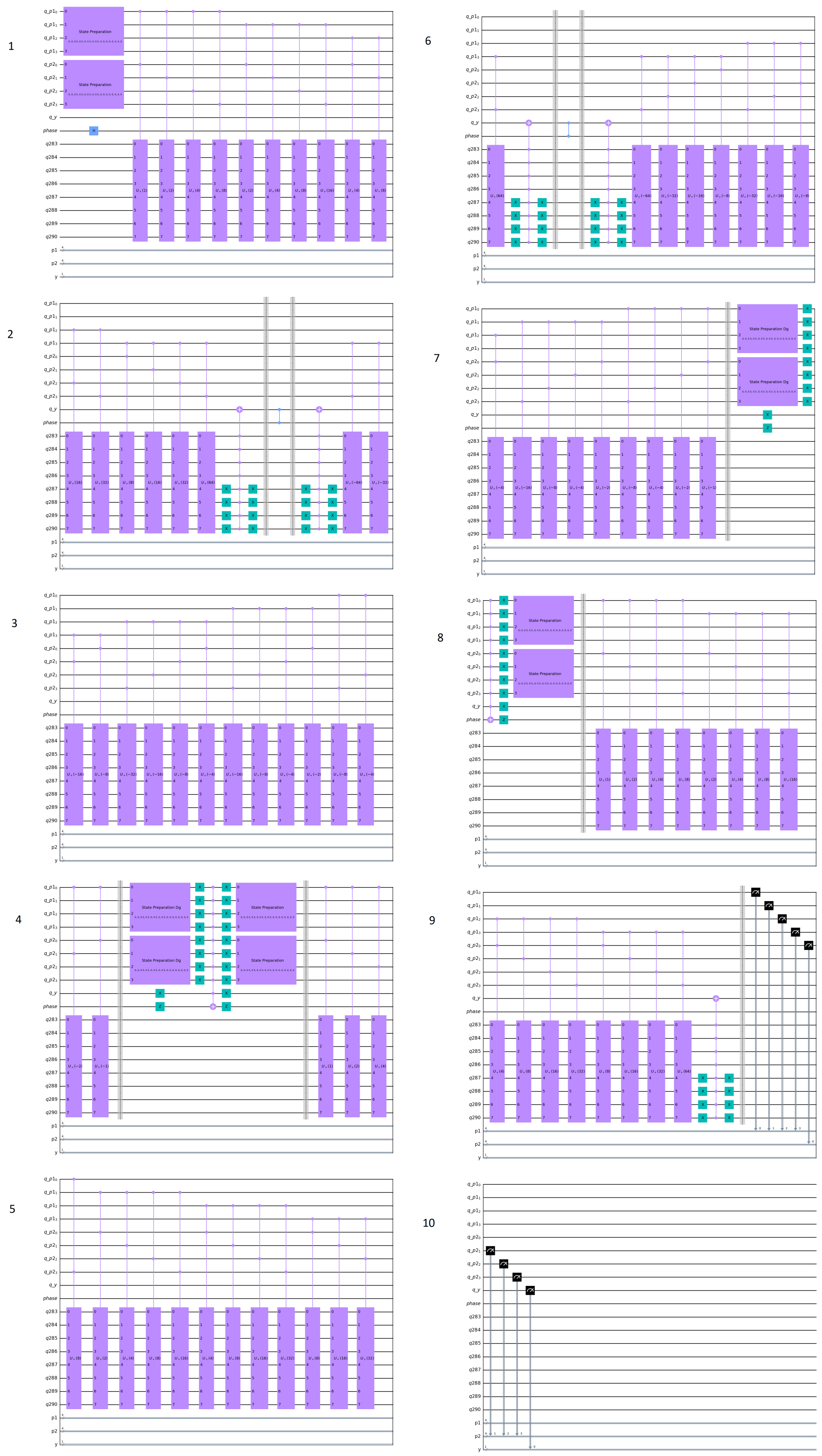

Figura 24 – Circuito quântico resultante da compilação do código fonte do exemplo 2 (atente-se à numeração das partes)

ANEXO B

Código fonte do DLQpiler

B.1 Módulo *utils.py*

```
#Filipe Chagas, 2023

from typing import *
from math import sqrt

def is_none(obj) -> bool:
    """
    :param obj: Object or None
    :type obj: Any type
    :return: True if obj is None
    :rtype: bool
    """
    return isinstance(obj, type(None))

def natural_to_binary(x: int, n: int) -> List[bool]:
    """Returns x in the binary system.
    :param x: Natural (including zero) to convert.
    :type x: int
    :param n: Number of bits.
    :type n: int
    :return: List of bits. Each bit is a bool. Most significant bit last
    .
    :rtype: List[bool]
    """
    assert n > 0
    x = x % 2**n
    return [bool((x//2**i)%2) for i in range(n)]

def binary_to_natural(x: List[bool]) -> int:
    """Returns x as a natural number (including zero).
    :param x: List of bits. Most significant bit last.
```

```python
    :type x: List[bool]
    :return: Natural number (including zero).
    :rtype: int
    """
    n = len(x)
    return sum([(2**i)*int(x[i]) for i in range(n)])

def set_to_statevector(values: Set[int], size: int) -> List[float]:
    """Return the statevector of a register initialized with a set of
   positive integer values.

    :param values: Superposed values
    :type values: Set[int]
    :param size: Register's size
    :type size: int
    :return: Statevector as a list of float values
    :rtype: List[float]
    """
    assert all([isinstance(v, int) for v in values])
    assert all([v >= 0 for v in values])
    assert all([v < 2**size for v in values])
    assert isinstance(size, int) and size > 0

    psi = [0 for i in range(2**size)]

    for v in values:
        psi[v] = 1/sqrt(len(values))

    return psi
```

Listing B.1 – utils.py

B.2 Módulo *qunits.py*

```python
#Filipe Chagas, 2023

import qiskit
from itertools import product
from dlqpiler.utils import natural_to_binary
from math import *
from typing import *

def qft(n: int) -> qiskit.circuit.Gate:
    """Returns a QFT gate for n qubits.

    :param n: Number of target qubits.
    :type n: int
    :return: QFT gate.
```

```
    :rtype: qiskit.circuit.Gate
    """
    def rotations(my_circuit: qiskit.circuit.Gate, m: int):
        if m == 0:
            return my_circuit
        else:
            my_circuit.h(m-1) #Add a Haddamard gate to the most
  significant qubit

            for i in range(m-1):
                my_circuit.crz(pi/(2**(m-1-i)), i, m-1)

            rotations(my_circuit, m-1)

    my_circuit = qiskit.QuantumCircuit(n, name='QFT')

    rotations(my_circuit, n)

    for m in range(n//2):
        my_circuit.swap(m, n-m-1)

    return my_circuit.to_gate()

# --- Arithmetic circuits ---

def register_by_constant_addition(n: int, c: int) -> qiskit.circuit.Gate
  :
    """
    Register-by-constant addition gate (simplified draper adder).
    Get a gate to perform an addition of a constant $c$ to a integer
  register.
    No ancillary qubits needed.

    :param n: Number of target qubits.
    :type n: int
    :param c: Constant to add.
    :type c: int
    :return: RCA gate.
    :rtype: qiskit.circuit.Gate
    """
    assert n > 0

    my_circuit = qiskit.QuantumCircuit(n, name=f'$U_+({c})$')

    my_qft = qft(n)
    my_circuit.append(my_qft, list(range(n)))

```

```python
    for i in range(n):
        theta = c * (pi / (2**(n-i-1)))
        my_circuit.rz(theta, i)

    my_circuit.append(my_qft.inverse(), list(range(n)))

    return my_circuit.to_gate()

def register_by_register_addition(circ: qiskit.QuantumCircuit, src_reg: List[qiskit.circuit.Qubit], target_reg: List[qiskit.circuit.Qubit]):
    """Build a register-by-register addition circuit

    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param src_reg: Operand register
    :type src_reg: List[qiskit.circuit.Qubit]
    :param target_reg: Target register
    :type target_reg: List[qiskit.circuit.Qubit]
    """
    for i in range(len(src_reg)):
        controlled_addition = register_by_constant_addition(len(target_reg), 2**i).control(1)
        circ.append(controlled_addition, [src_reg[i]]+target_reg)

def register_by_register_addition_dg(circ: qiskit.QuantumCircuit, src_reg: List[qiskit.circuit.Qubit], target_reg: List[qiskit.circuit.Qubit]):
    """Build an inverse register-by-register addition circuit

    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param src_reg: Operand register
    :type src_reg: List[qiskit.circuit.Qubit]
    :param target_reg: Target register
    :type target_reg: List[qiskit.circuit.Qubit]
    """
    for i in range(len(src_reg))[::-1]:
        controlled_addition = register_by_constant_addition(len(target_reg), 2**i).inverse().control(1)
        circ.append(controlled_addition, [src_reg[i]]+target_reg)

def register_by_register_subtraction(circ: qiskit.QuantumCircuit, src_reg: List[qiskit.circuit.Qubit], target_reg: List[qiskit.circuit.Qubit]):
    """Build a register-by-register subtraction circuit

    :param circ: Target quantum circuit
```

```
    :type circ: qiskit.QuantumCircuit
    :param src_reg: Operand register
    :type src_reg: List[qiskit.circuit.Qubit]
    :param target_reg: Target register
    :type target_reg: List[qiskit.circuit.Qubit]
    """
    for i in range(len(src_reg)):
        controlled_addition = register_by_constant_addition(len(target_reg), -2**i).control(1)
        circ.append(controlled_addition, [src_reg[i]]+target_reg)

def register_by_register_subtraction_dg(circ: qiskit.QuantumCircuit, src_reg: List[qiskit.circuit.Qubit], target_reg: List[qiskit.circuit.Qubit]):
    """Build an inverse register-by-register subtraction circuit

    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param src_reg: Operand register
    :type src_reg: List[qiskit.circuit.Qubit]
    :param target_reg: Target register
    :type target_reg: List[qiskit.circuit.Qubit]
    """
    for i in range(len(src_reg))[::-1]:
        controlled_addition = register_by_constant_addition(len(target_reg), -2**i).inverse().control(1)
        circ.append(controlled_addition, [src_reg[i]]+target_reg)

def multiproduct(circ: qiskit.QuantumCircuit, bases: List[List[qiskit.circuit.Qubit]], exponents: List[int], result: List[qiskit.circuit.Qubit], constant: int = 1):
    """Build a circuit that perform a productory with constant exponents

    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param bases: List of registers that contains base values
    :type bases: List[List[qiskit.circuit.Qubit]]
    :param exponents: List of exponents
    :type exponents: List[int]
    :param result: Target register
    :type result: List[qiskit.circuit.Qubit]
    :param constant: Constant factor, defaults to 1
    :type constant: int, optional
    """
    #The powers and the productory must be calculated as a sequence of controlled const additions
    factors_indexes = [] #This list contains a sequence with the index
```

```
of each factor E times, where E is the respective exponent
    for i in range(len(exponents)):
        factors_indexes += [i]*exponents[i]

    for t in product(*[list(range(len(bases[factors_indexes[i]]))) for i in range(len(factors_indexes))]): #Each tuple t have the indexes of the qubits that must be used as control of each register
        ctrl_idx = {factor_index:set() for factor_index in range(len(bases))} #This dict will map each factor's index to a set with it's control qubit's indexes
        #fill the ctrl dict
        for i in range(len(t)):
            ctrl_idx[factors_indexes[i]].add(t[i])

        ctrl_qubits = [] #Control qubits
        #fill the ctrl_qubits list
        for factor_index in ctrl_idx.keys():
            for qubit_index in ctrl_idx[factor_index]:
                ctrl_qubits.append(bases[factor_index][qubit_index])

        c = constant*2**sum(t) #Constant to add

        #Append controled const addition
        my_const_adder = register_by_constant_addition(len(result), c)
        my_const_adder = my_const_adder.control(len(ctrl_qubits))
        circ.append(my_const_adder, ctrl_qubits + result)


def multiproduct_dg(circ: qiskit.QuantumCircuit, bases: List[List[qiskit.circuit.Qubit]], exponents: List[int], result: List[qiskit.circuit.Qubit], constant: int = 1):
    """Build the inverse product circuit

    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param bases: List of registers that contains base values
    :type bases: List[List[qiskit.circuit.Qubit]]
    :param exponents: List of exponents
    :type exponents: List[int]
    :param result: Target register
    :type result: List[qiskit.circuit.Qubit]
    :param constant: Constant factor, defaults to 1
    :type constant: int, optional
    """
    #The powers and the productory must be calculated as a sequence of controlled const additions
    factors_indexes = [] #This list contains a sequence with the index
```

```python
of each factor E times, where E is the respective exponent
    for i in range(len(exponents)):
        factors_indexes += [i]*exponents[i]

    instructions = []

    for t in product(*[list(range(len(bases[factors_indexes[i]]))) for i in range(len(factors_indexes))]): #Each tuple t have the indexes of the qubits that must be used as control of each register
        ctrl_idx = {factor_index:set() for factor_index in range(len(bases))} #This dict will map each factor's index to a set with it's control qubit's indexes
        #fill the ctrl dict
        for i in range(len(t)):
            ctrl_idx[factors_indexes[i]].add(t[i])

        ctrl_qubits = [] #Control qubits
        #fill the ctrl_qubits list
        for factor_index in ctrl_idx.keys():
            for qubit_index in ctrl_idx[factor_index]:
                ctrl_qubits.append(bases[factor_index][qubit_index])

        c = constant*2**sum(t) #Constant to add

        #Append controled const addition
        my_const_adder = register_by_constant_addition(len(result), -c)
        my_const_adder = my_const_adder.control(len(ctrl_qubits))
        instructions.append((my_const_adder, ctrl_qubits + result))

    for ins, qbts in instructions[::-1]:
        circ.append(ins, qbts)

# --- Relational circuits ---

def register_less_than_register(circ: qiskit.QuantumCircuit, left: List[qiskit.circuit.Qubit], right: List[qiskit.circuit.Qubit], aux: List[qiskit.circuit.Qubit], result: qiskit.circuit.Qubit):
    """Build the quantum circuit of the less-than operation

    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param left: left operand
    :type left: List[qiskit.circuit.Qubit]
    :param right: right operand
    :type right: List[qiskit.circuit.Qubit]
    :param aux: ancilla qubit
    :type aux: qiskit.circuit.Qubit
```

```python
    :param result: result qubit
    :type result: qiskit.circuit.Qubit
    """
    xleft = left + aux
    register_by_register_subtraction(circ, right, xleft)
    circ.cx(aux, result)
    register_by_register_subtraction_dg(circ, right, xleft)

def register_less_than_register_dg(circ: qiskit.QuantumCircuit, left: List[qiskit.circuit.Qubit], right: List[qiskit.circuit.Qubit], aux: List[qiskit.circuit.Qubit], result: qiskit.circuit.Qubit):
    """Build the inverse quantum circuit of the less-than operation

    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param left: left operand
    :type left: List[qiskit.circuit.Qubit]
    :param right: right operand
    :type right: List[qiskit.circuit.Qubit]
    :param aux: ancilla qubit
    :type aux: qiskit.circuit.Qubit
    :param result: result qubit
    :type result: qiskit.circuit.Qubit
    """
    xleft = left + aux
    register_by_register_addition(circ, right, xleft)
    circ.cx(aux, result)
    register_by_register_addition_dg(circ, right, xleft)

def register_greater_than_register(circ: qiskit.QuantumCircuit, left: List[qiskit.circuit.Qubit], right: List[qiskit.circuit.Qubit], aux: qiskit.circuit.Qubit, result: qiskit.circuit.Qubit):
    """Build the quantum circuit of the greater-than operation

    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param left: left operand
    :type left: List[qiskit.circuit.Qubit]
    :param right: right operand
    :type right: List[qiskit.circuit.Qubit]
    :param aux: ancilla qubit
    :type aux: qiskit.circuit.Qubit
    :param result: result qubit
    :type result: qiskit.circuit.Qubit
    """
    register_less_than_register(circ, right, left, aux, result)

```

```python
def register_greater_than_register_dg(circ: qiskit.QuantumCircuit, left: List[qiskit.circuit.Qubit], right: List[qiskit.circuit.Qubit], aux: qiskit.circuit.Qubit, result: qiskit.circuit.Qubit):
    """Build the inverse quantum circuit of the greater-than operation

    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param left: left operand
    :type left: List[qiskit.circuit.Qubit]
    :param right: right operand
    :type right: List[qiskit.circuit.Qubit]
    :param aux: ancilla qubit
    :type aux: qiskit.circuit.Qubit
    :param result: result qubit
    :type result: qiskit.circuit.Qubit
    """
    register_less_than_register_dg(circ, right, left, aux, result)

def register_less_than_constant(circ: qiskit.QuantumCircuit, reg: List[qiskit.circuit.Qubit], constant: int, aux: List[qiskit.circuit.Qubit], result: qiskit.circuit.Qubit):
    """Build the quantum circuit of the less-than operation with an constant right operand

    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param reg: left operand
    :type reg: List[qiskit.circuit.Qubit]
    :param constant: right operand
    :type constant: int
    :param aux: ancilla qubits
    :type aux: List[qiskit.circuit.Qubit]
    :param result: result qubit
    :type result: qiskit.circuit.Qubit
    """
    xreg = reg + aux
    adder = register_by_constant_addition(len(xreg), -constant)
    circ.append(adder, xreg)
    circ.cx(reg[-1], result)
    circ.append(adder.inverse(), xreg)

def register_less_than_constant_dg(circ: qiskit.QuantumCircuit, reg: List[qiskit.circuit.Qubit], constant: int, aux: List[qiskit.circuit.Qubit], result: qiskit.circuit.Qubit):
    """Build the inverse quantum circuit of the less-than operation with an constant right operand

```

```
    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param reg: left operand
    :type reg: List[qiskit.circuit.Qubit]
    :param constant: right operand
    :type constant: int
    :param aux: ancilla qubits
    :type aux: List[qiskit.circuit.Qubit]
    :param result: result qubit
    :type result: qiskit.circuit.Qubit
    """
    xreg = reg + aux
    adder = register_by_constant_addition(len(xreg), -constant)
    circ.append(adder.inverse(), xreg)
    circ.cx(reg[-1], result)
    circ.append(adder, xreg)

def register_greater_than_constant(circ: qiskit.QuantumCircuit, reg: List[qiskit.circuit.Qubit], constant: int, aux: List[qiskit.circuit.Qubit], result: qiskit.circuit.Qubit):
    """Build the quantum circuit of the greater-than operation with an constant right operand

    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param reg: left operand
    :type reg: List[qiskit.circuit.Qubit]
    :param constant: right operand
    :type constant: int
    :param aux: ancilla qubits
    :type aux: List[qiskit.circuit.Qubit]
    :param result: result qubit
    :type result: qiskit.circuit.Qubit
    """
    xreg = reg + aux
    adder = register_by_constant_addition(len(xreg), -constant-1)
    circ.append(adder, xreg)
    circ.cx(reg[-1], result)
    circ.x(result)
    circ.append(adder.inverse(), xreg)

def register_greater_than_constant_dg(circ: qiskit.QuantumCircuit, reg: List[qiskit.circuit.Qubit], constant: int, aux: List[qiskit.circuit.Qubit], result: qiskit.circuit.Qubit):
    """Build the inverse quantum circuit of the greater-than operation with an constant right operand

```

```
    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param reg: left operand
    :type reg: List[qiskit.circuit.Qubit]
    :param constant: right operand
    :type constant: int
    :param aux: ancilla qubits
    :type aux: List[qiskit.circuit.Qubit]
    :param result: result qubit
    :type result: qiskit.circuit.Qubit
    """
    xreg = reg + aux
    adder = register_by_constant_addition(len(xreg), -constant-1)
    circ.append(adder.inverse(), xreg)
    circ.x(result)
    circ.cx(reg[-1], result)
    circ.append(adder, xreg)

def register_equal_register(circ: qiskit.QuantumCircuit, left: List[
   qiskit.circuit.Qubit], right: List[qiskit.circuit.Qubit], aux: List[
   qiskit.circuit.Qubit], result: qiskit.circuit.Qubit):
    """Build a quantum circuit to the Equal operation between two
   registers

    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param left: left operand
    :type left: List[qiskit.circuit.Qubit]
    :param right: right operand
    :type right: List[qiskit.circuit.Qubit]
    :param aux: ancilla qubits
    :type aux: List[qiskit.circuit.Qubit]
    :param result: result qubit
    :type result: qiskit.circuit.Qubit
    """
    if len(left) >= len(right):
        xleft = left
        xright = right + aux
    else:
        xleft = left + aux
        xright = right
    assert len(xleft) == len(xright)

    for i in range(len(xleft)):
        circ.cx(xleft[i], xright[i])

    circ.mcx(xright, result)
```

```python

    for i in range(len(xleft))[::-1]:
        circ.cx(xleft[i], xright[i])

def register_equal_register_dg(circ: qiskit.QuantumCircuit, left: List[
    qiskit.circuit.Qubit], right: List[qiskit.circuit.Qubit], aux: List[
    qiskit.circuit.Qubit], result: qiskit.circuit.Qubit):
    """Build an inverse quantum circuit to the Equal operation between
    two registers

    :param circ: Target quantum circuit
    :type circ: qiskit.QuantumCircuit
    :param left: left operand
    :type left: List[qiskit.circuit.Qubit]
    :param right: right operand
    :type right: List[qiskit.circuit.Qubit]
    :param aux: ancilla qubits
    :type aux: List[qiskit.circuit.Qubit]
    :param result: result qubit
    :type result: qiskit.circuit.Qubit
    """
    register_equal_register(circ, left, right, aux, result)

def register_equal_constant(circ: qiskit.QuantumCircuit, reg: List[
    qiskit.circuit.Qubit], constant: int, aux: List[qiskit.circuit.Qubit
    ], result: qiskit.circuit.Qubit):
    """Build a quantum circuit to the Equal operation between a register
     and a constant

    :param circ: Target circuit
    :type circ: qiskit.QuantumCircuit
    :param reg: Register operand
    :type reg: List[qiskit.circuit.Qubit]
    :param constant: Const operand
    :type constant: int
    :param aux: Ancilla qubits
    :type aux: List[qiskit.circuit.Qubit]
    :param result: Result qubit
    :type result: qiskit.circuit.Qubit
    """
    xreg = reg + aux
    bconst = natural_to_binary(constant, len(reg)+len(aux))
    for i in range(len(bconst)):
        if not bconst[i]:
            circ.x(xreg[i])
    circ.mcx(reg+aux, result)
    for i in range(len(bconst)):
```

```
        if not bconst[i]:
            circ.x(xreg[i])

def register_equal_constant_dg(circ: qiskit.QuantumCircuit, reg: List[
    qiskit.circuit.Qubit], constant: int, aux: List[qiskit.circuit.Qubit
    ], result: qiskit.circuit.Qubit):
    """Build an inverse quantum circuit to the Equal operation between a
    register and a constant

    :param circ: Target circuit
    :type circ: qiskit.QuantumCircuit
    :param reg: Register operand
    :type reg: List[qiskit.circuit.Qubit]
    :param constant: Const operand
    :type constant: int
    :param aux: Ancilla qubits
    :type aux: List[qiskit.circuit.Qubit]
    :param result: Result qubit
    :type result: qiskit.circuit.Qubit
    """
    register_equal_constant(circ, reg, constant, aux, result)

def register_not_equal_register(circ: qiskit.QuantumCircuit, left: List[
    qiskit.circuit.Qubit], right: List[qiskit.circuit.Qubit], aux: List[
    qiskit.circuit.Qubit], result: qiskit.circuit.Qubit):
    """Build a quantum circuit to the Not-Equal operation between two
    registers

    :param circ: Target circuit
    :type circ: qiskit.QuantumCircuit
    :param reg: Register operand
    :type reg: List[qiskit.circuit.Qubit]
    :param constant: Const operand
    :type constant: int
    :param aux: Ancilla qubits
    :type aux: List[qiskit.circuit.Qubit]
    :param result: Result qubit
    :type result: qiskit.circuit.Qubit
    """
    register_equal_register(circ, left, right, aux, result)
    circ.x(result)

def register_not_equal_register_dg(circ: qiskit.QuantumCircuit, left:
    List[qiskit.circuit.Qubit], right: List[qiskit.circuit.Qubit], aux:
    List[qiskit.circuit.Qubit], result: qiskit.circuit.Qubit):
    """Build an inverse quantum circuit to the Not-Equal operation
    between two registers
```

```

    :param circ: Target circuit
    :type circ: qiskit.QuantumCircuit
    :param reg: Register operand
    :type reg: List[qiskit.circuit.Qubit]
    :param constant: Const operand
    :type constant: int
    :param aux: Ancilla qubits
    :type aux: List[qiskit.circuit.Qubit]
    :param result: Result qubit
    :type result: qiskit.circuit.Qubit
    """
    circ.x(result)
    register_equal_register_dg(circ, left, right, aux, result)

def register_not_equal_constant(circ: qiskit.QuantumCircuit, reg: List[
    qiskit.circuit.Qubit], constant: int, aux: List[qiskit.circuit.Qubit
    ], result: qiskit.circuit.Qubit):
    """Build a quantum circuit to the Not-Equal operation between a
    register and a constant

    :param circ: Target circuit
    :type circ: qiskit.QuantumCircuit
    :param reg: Register operand
    :type reg: List[qiskit.circuit.Qubit]
    :param constant: Const operand
    :type constant: int
    :param aux: Ancilla qubits
    :type aux: List[qiskit.circuit.Qubit]
    :param result: Result qubit
    :type result: qiskit.circuit.Qubit
    """
    register_equal_constant(circ, reg, constant, aux, result)
    circ.x(result)

def register_not_equal_constant_dg(circ: qiskit.QuantumCircuit, reg:
    List[qiskit.circuit.Qubit], constant: int, aux: List[qiskit.circuit.
    Qubit], result: qiskit.circuit.Qubit):
    """Build an inverse quantum circuit to the Not-Equal operation
    between a register and a constant

    :param circ: Target circuit
    :type circ: qiskit.QuantumCircuit
    :param reg: Register operand
    :type reg: List[qiskit.circuit.Qubit]
    :param constant: Const operand
    :type constant: int
```

```
    :param aux: Ancilla qubits
    :type aux: List[qiskit.circuit.Qubit]
    :param result: Result qubit
    :type result: qiskit.circuit.Qubit
    """
    circ.x(result)
    register_equal_constant_dg(circ, reg, constant, aux, result)
```

Listing B.2 – qunits.py

B.3 Módulo *lexer.py*

```
#Filipe Chagas, 2023

import ply.lex as lex

#This is a dictionary of reserved language words.
#It is necessary to create this dictionary so that lexer does not return
    these tokens as generic identifiers.
reserved = {
    'true': 'TRUE',
    'false': 'FALSE',
    'and': 'AND',
    'or': 'OR',
    'not': 'NOT',
    'in': 'IN',
    'amplify': 'AMPLIFY',
    'times': 'TIMES',
}

#This is a list with the names of the tokens.
tokens = [
   'NUMBER',
   'ID',
   # --- arithmetic ---
   'PLUS',
   'MINUS',
   'MUL',
   'DIVIDE',
   'HAT',
   # --- relational ---
   'EQUAL',
   'NEQ',
   'LT',
   'GT',
   # --- others ---
   'COMMA',
   'ASSIGN',
```

```
    'SEMICOLON',
    'LPAREN',
    'RPAREN',
    'LCURLY',
    'RCURLY',
    'LBRACKET',
    'RBRACKET'
] + list(reserved.values())

#Next, we define the regular expressions for the tokens
# --- arithmetic ---
t_PLUS     = r'\+'
t_MINUS    = r'-'
t_MUL   = r'\*'
t_DIVIDE   = r'/'
t_HAT  = r'\^'

# --- relational ---
t_EQUAL  = r'='
t_NEQ = r'!='
t_LT = r'<'
t_GT = r'>'

# --- others ---
t_COMMA = r','
t_ASSIGN   = r':='
t_LPAREN   = r'\('
t_RPAREN   = r'\)'
t_LCURLY   =  r'\{'
t_RCURLY   =  r'\}'
t_LBRACKET   =  r'\['
t_RBRACKET   =  r'\]'
t_SEMICOLON   =  r';'

#Next, we define the regex of numbers and identifiers (or reserved words
    )
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value,'ID') #Check for reserved words
    return t

#Define a rule so we can track line numbers
```

```python
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

#A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

#Defines a custom exception for lexical errors
class LexicalError(Exception):
    def __init__(self, token) -> None:
        super().__init__(f'Illegal character {token.value[0]}')

#Error handling rule
def t_error(t):
    raise LexicalError(t)

#Build the lexer
lexer = lex.lex()
```

Listing B.3 – lexer.py

B.4 Módulo *parser.py*

```python
#Filipe Chagas, 2023

from dlqpiler.lexer import *
from dlqpiler import ast
import ply.yacc as yacc

#Defines a custom exception for parsing errors
class ParsingError(Exception):
    def __init__(self, line: int, message: str) -> None:
        self.line = line
        self.message = message
        if isinstance(line, int): #Check if line is not None
            super().__init__(f'Parsing error at line {line}: {message}')
        else:
            super().__init__(f'Parsing error at EOF: {message}')

#Defines the precedence and associativity of unary and binary operators
precedence = (
    ('left', 'OR'),
    ('left', 'AND'),
    ('right', 'NOT'),
    ('left', 'LT', 'GT'),
    ('left', 'EQUAL', 'NEQ'),
    ('left', 'PLUS', 'MINUS'),
    ('left', 'MUL', 'DIVIDE'),
```

```
    ('right', 'UMINUS'),
    ('right', 'HAT'),
)

# --- Parsing rules to statements ---

#Root parsing rule
def p_full_code(p):
    'fullcode : regdefseq amplifyterm'
    p[0] = ast.FullCode(p.lineno(0), p[1], p[2])

#Syntax of the amplify terminator
def p_amplify_terminator(p):
    'amplifyterm : AMPLIFY ID NUMBER TIMES'
    target = p[2]
    it = p[3]

    if it < 0:
        raise ParsingError(p.lineno(0), 'The number of amplify iterations must be greater or equal to 0')

    p[0] = ast.Amplify(p.lineno(0), target, it)

#Syntax of a sequence of register definitions separated by semicolons
def p_regdef_sequence_body(p):
    'regdefseq : regdef SEMICOLON regdefseq'
    p[0] = [p[1]] + p[3]

def p_regdef_sequence_tail(p):
    'regdefseq : regdef SEMICOLON'
    p[0] = [p[1]]

#A register definition statement, that can be by set or by expression
def p_register_definition(p):
    '''regdef : regdefs
              | regdefx'''
    p[0] = p[1]

#Statement for the definition of a register as a set
#Example: "myreg[8] in {1, 2, 3}" defines an 8-bit register as a superposition of 1, 2 and 3
def p_register_definition_set(p):
    'regdefs : ID LBRACKET NUMBER RBRACKET IN LCURLY expseq RCURLY'
    id = p[1]
    n = p[3]
    seq = p[7]

```

```
    if n <= 0:
        raise ParsingError(p.lineno(0), 'Register\'s size must be greater than 0')

    if not all([isinstance(v, int) for v in seq]):
        raise ParsingError(p.lineno(0), 'A set must be composed only of constant values')

    p[0] = ast.RegisterSetDefinition(p.lineno(0), id, n, set(seq))

#Statement for the definition of a register as an expression
#Example: "myreg[8] := b^2 - 4*a*c" defines an 8-bit register as b^2-4*a*c
def p_register_definition_expression(p):
    'regdefx : ID LBRACKET NUMBER RBRACKET ASSIGN expression'
    id = p[1]
    n = p[3]
    expr = p[6]

    if n <= 0:
        raise ParsingError(p.lineno(0), 'Register\'s size must be greater than 0')

    if isinstance(expr, (ast.Identifier, int)):
        raise ParsingError(p.lineno(0), 'dlqpiler currently does not accept direct assignments or constants in registers, only logical, arithmetic and relational expressions.')

    p[0] = ast.RegisterExpressionDefinition(p.lineno(0), id, n, expr)


# --- Parsing rules to expression sequences ---
def p_expression_sequence_fork(p):
    'expseq : expseq COMMA expression'
    p[0] = p[1] + [p[3]]

def p_expression_sequence_tail(p):
    'expseq : expression'
    p[0] = [p[1]]

# --- Parsing rules to logic expressions ---
def p_expression_or(p):
    'expression : expression OR expression'
    if isinstance(p[1], int) and isinstance(p[3], int):
        p[0] = int(bool(p[1] % 2) or bool(p[3] % 2))
    elif isinstance(p[1], ast.Expression) and isinstance(p[3], int):
        p[0] = 1 if bool(p[3] % 2) else p[1]
```

```python
    elif isinstance(p[1], int) and isinstance(p[3], ast.Expression):
        p[0] = 1 if bool(p[1] % 2) else p[3]
    elif isinstance(p[1], ast.Expression) and isinstance(p[3], ast.Expression):
        p[0] = ast.Or.merge(p.lineno(0), p[1], p[3])
    else:
        raise ParsingError(p.lineno(0), f'It is not possible to apply the OR operator to types {(type(p[1]), type(p[3]))}')

def p_expression_and(p):
    'expression : expression AND expression'
    if isinstance(p[1], int) and isinstance(p[3], int):
        p[0] = int(bool(p[1] % 2) and bool(p[3] % 2))
    elif isinstance(p[1], ast.Expression) and isinstance(p[3], int):
        p[0] = p[1] if bool(p[3] % 2) else 0
    elif isinstance(p[1], int) and isinstance(p[3], ast.Expression):
        p[0] = p[3] if bool(p[1] % 2) else 0
    elif isinstance(p[1], ast.Expression) and isinstance(p[3], ast.Expression):
        p[0] = ast.And.merge(p.lineno(0), p[1], p[3])
    else:
        raise ParsingError(p.lineno(0), f'It is not possible to apply the AND operator to types {(type(p[1]), type(p[3]))}')

def p_expression_not(p):
    'expression : NOT expression'
    if isinstance(p[2], int):
        p[0] = int(not bool(p[2] % 2))
    elif isinstance(p[2], (ast.Expression, int)):
        p[0] = ast.Not(p.lineno(0), p[2])
    else:
        raise ParsingError(p.lineno(0), f'It is not possible to apply the NOT operator to type {type(p[2])}')

# --- Parsing rules to relational expressions ---

#Parsing rule to the equal operator ('=')
def p_expression_equal(p):
    'expression : expression EQUAL expression'
    if isinstance(p[1], int) and isinstance(p[3], int):
        p[0] = p[1] == p[3]
    elif isinstance(p[1], (ast.Expression, int)) and isinstance(p[3], (ast.Expression, int)):
        p[0] = ast.Equal(p.lineno(0), p[1], p[3])
    else:
        raise ParsingError(p.lineno(0), f'It is not possible to apply the equal operator to types {(type(p[1]), type(p[3]))}')
```

```
#Parsing rule to the not-equal operator ('!=')
def p_expression_not_equal(p):
    'expression : expression NEQ expression'
    if isinstance(p[1], int) and isinstance(p[3], int):
        p[0] = p[1] != p[3]
    elif isinstance(p[1], (ast.Expression, int)) and isinstance(p[3], (ast.Expression, int)):
        p[0] = ast.NotEqual(p.lineno(0), p[1], p[3])
    else:
        raise ParsingError(p.lineno(0), f'It is not possible to apply the not-equal operator to types {(type(p[1]), type(p[3]))}')

#Parsing rule to the less-than operator ('<')
def p_expression_less_than(p):
    'expression : expression LT expression'
    if isinstance(p[1], int) and isinstance(p[3], int):
        p[0] = p[1] < p[3]
    elif isinstance(p[1], (ast.Expression, int)) and isinstance(p[3], (ast.Expression, int)):
        p[0] = ast.LessThan(p.lineno(0), p[1], p[3])
    else:
        raise ParsingError(p.lineno(0), f'It is not possible to apply the less-than operator to types {(type(p[1]), type(p[3]))}')

#Parsing rule to the greater-than operator ('>')
def p_expression_greater_than(p):
    'expression : expression GT expression'
    if isinstance(p[1], int) and isinstance(p[3], int):
        p[0] = p[1] > p[3]
    elif isinstance(p[1], (ast.Expression, int)) and isinstance(p[3], (ast.Expression, int)):
        p[0] = ast.GreaterThan(p.lineno(0), p[1], p[3])
    else:
        raise ParsingError(p.lineno(0), f'It is not possible to apply the greater-than operator to types {(type(p[1]), type(p[3]))}')

# --- Parsing rules to arithmetic expressions ---

#Parsing rule to the addition operator ('+')
def p_expression_add(p):
    'expression : expression PLUS expression'
    if isinstance(p[1], int) and isinstance(p[3], int):
        p[0] = p[1] + p[3]
    elif isinstance(p[1], (ast.Expression, int)) and isinstance(p[3], (ast.Expression, int)):
        p[0] = ast.Summation.merge_add(p.lineno(0), p[1], p[3])
```

```
    else:
        raise ParsingError(p.lineno(0), f'It is not possible to apply the addition operator to types {(type(p[1]), type(p[3]))}')

#Parsing rule to the subtraction operator ('-')
def p_expression_sub(p):
    'expression : expression MINUS expression'
    if isinstance(p[1], int) and isinstance(p[3], int):
        p[0] = p[1] - p[3]
    elif isinstance(p[1], (ast.Expression, int)) and isinstance(p[3], (ast.Expression, int)):
        p[0] = ast.Summation.merge_sub(p.lineno(0), p[1], p[3])
    else:
        raise ParsingError(p.lineno(0), f'It is not possible to apply the subtraction operator to types {(type(p[1]), type(p[3]))}')

#Parsing rule to the multiplication operator ('*')
def p_expression_mul(p):
    'expression : expression MUL expression'
    if isinstance(p[1], int) and isinstance(p[3], int):
        p[0] = p[1] * p[3]
    elif isinstance(p[1], (ast.Expression, int)) and isinstance(p[3], (ast.Expression, int)):
        p[0] = ast.Product.merge(p.lineno(0), p[1], p[3])
    else:
        raise ParsingError(p.lineno(0), f'It is not possible to apply the product operator to types {(type(p[1]), type(p[3]))}')

#Parsing rule to the division operator ('^')
def p_expression_power(p):
    'expression : expression HAT expression'
    if isinstance(p[3], int):
        if isinstance(p[1], ast.Expression):
            p[0] = ast.Power(p.lineno(0), p[1], p[3])
        elif isinstance(p[1], int):
            p[0] = p[1]**p[3]
        else:
            raise ParsingError(p.lineno(0), f'It\'s not possible to apply the power operator to a base of type {type(p[1])}')
    else:
        raise ParsingError(p.lineno(0), 'The power operator can only be used with constant exponent')

#Parsing rule to the division operator ('/')
def p_expression_division(p):
    'expression : expression DIVIDE expression'
    if isinstance(p[1], int) and isinstance(p[3], int):
```

```
        p[0] = p[1]//p[3]
    else:
        raise ParsingError(p.lineno(0), 'The division operator can only
    be applied to constant numeric values')

#Parsing rule to unary minus
def p_expression_uminus(p):
    'expression : MINUS expression %prec UMINUS'
    if isinstance(p[2], ast.Expression):
        p[0] = ast.UnaryMinus(p.lineno(0), p[2])
    elif isinstance(p[2], int):
        p[0] = -p[2]
    else:
        raise ParsingError(p.lineno(0), f'It\'s not possible to apply
    the unary minus operator to {type(p[2])}')

#Parsing rule to expressions in parentheses
def p_expression_parentheses(p):
    'expression : LPAREN expression RPAREN'
    if isinstance(p[2], int): #If the inner expression is a value,
    return the value
        p[0] = p[2]
    else:
        p[0] = ast.Parentheses(p.lineno(0), p[2])

# --- Parsing rules to values and identifiers ---

def p_expression_false(p):
    'expression : FALSE'
    p[0] = 0

def p_expression_true(p):
    'expression : TRUE'
    p[0] =  1

def p_expression_number(p):
    'expression : NUMBER'
    p[0] = p[1]

def p_expression_id(p):
    'expression : ID'
    p[0] = ast.Identifier(p.lineno(0), p[1])

#Defines a function to handle syntax errors
def p_error(p):
    if p:
        raise ParsingError(p.lineno(0), 'Invalid syntax')
```

```
    else:
        raise ParsingError(None, 'Invalid syntax')

#Build the parser
parser = yacc.yacc()
```

Listing B.4 – parser.py

B.5 Módulo *ast.py*

```
#Filipe Chagas, 2023

from typing import *
from enum import Enum
from typing import Set
from dlqpiler import utils, qunits
import qiskit
import math

n_bits_const = lambda c: int(math.ceil(math.log2(c))) if c > 0 else 0

class SynthError(Exception):
    def __init__(self, line: int, description: str) -> None:
        self.line = line
        self.description = description
        super().__init__(f'Synthesis error at line {line}: {description}')

class Signal(Enum):
    POS = True #Positive signal
    NEG = False #Negative signal

class ASTNode():
    def __init__(self, line: int) -> None:
        """
        :param line: Line of code
        :type line: int
        """
        assert isinstance(line, int)
        self.line = line

    def to_dict(self) -> Dict[str, object]:
        """
        Recursively convert the AST to a tree of dictionaries

        :return: Tree of dictionaries
        :rtype: Dict[str, object]
        """
```

```
        return dict()

# --- Expression AST nodes ---

class Expression(ASTNode):
    def __init__(self, line: int) -> None:
        super().__init__(line)
        self.result = None

    def get_leafs(self) -> Set[str]:
        """Return a set with all identifiers used in the expression

        :return: self-descriptive
        :rtype: Set[str]
        """
        return set()

    def needs_result_allocation(self) -> bool:
        """
        :return: True if the ASTNode needs allocation of result qubits
        :rtype: bool
        """
        return True

    def n_result_qubits(self, quantum_evaluator) -> int:
        """
        :param quantum_evaluator: Parent quantum evaluator
        :type quantum_evaluator: QuantumEvaluator
        :raises NotImplementedError: self-descriptive
        :return: Number of necessary result qubits
        :rtype: int
        """
        raise NotImplementedError()

    def alloc_result_qubits(self, quantum_evaluator):
        """Allocate the result qubits from the quantum evaluator

        :param quantum_evaluator: Parent quantum evaluator
        :type quantum_evaluator: QuantumEvaluator
        """
        assert utils.is_none(self.result)
        self.result = [quantum_evaluator.alloc_ancilla() for i in range(
    self.n_result_qubits(quantum_evaluator))]

    def release_result_qubits(self, quantum_evaluator):
        """Release the clean result qubits

```

```
        :param quantum_evaluator: Parent quantum evaluator
        :type quantum_evaluator: QuantumEvaluator
        """
        assert not utils.is_none(self.result)
        for qb in self.result:
            quantum_evaluator.free_ancilla(qb)
        self.result = None

    def pre_build(self, quantum_evaluator):
        """Do the necessary actions before the build operation

        :param quantum_evaluator: Parent quantum evaluator
        :type quantum_evaluator: QuantumEvaluator
        """
        pass

    def build(self, quantum_evaluator):
        """Build the quantum circuit recursively

        :param quantum_evaluator: Parent quantum evaluator
        :type quantum_evaluator: QuantumEvaluator
        :raises NotImplementedError: self-descriptive
        """
        raise NotImplementedError()

    def reverse(self, quantum_evaluator):
        """Build the inverse quantum circuit recursively

        :param quantum_evaluator: Parent quantum evaluator
        :type quantum_evaluator: QuantumEvaluator
        :raises NotImplementedError: self-descriptive
        """
        raise NotImplementedError()

class Identifier(Expression):
    def __init__(self, line: int, label: str) -> None:
        """
        :param line: Line of code
        :type line: int
        :param label: Text of the identifier
        :type label: str
        """
        super().__init__(line)
        assert isinstance(label, str)
        self.label = label

    def get_leafs(self) -> Set[str]:
```

```
        return {self.label}

    def n_result_qubits(self, quantum_evaluator) -> int:
        n = quantum_evaluator.get_register_size(self.label)
        if utils.is_none(n):
            raise SynthError(self.line, f'Identifier "{self.label}" not defined')
        return n

    def needs_result_allocation(self) -> bool:
        return False

    def to_dict(self) -> Dict[str, object]:
        return {'type': 'Identifier', 'label': self.label}

    def pre_build(self, quantum_evaluator):
        qreg = quantum_evaluator.get_qiskit_register(self.label)
        if utils.is_none(qreg):
            raise SynthError(self.line, f'Identifier "{self.label}" not defined')
        self.result = [qubit for qubit in qreg]

class Parentheses(Expression):
    def __init__(self, line: int, inner_expr: Expression) -> None:
        """
        :param line: Line of code
        :type line: int
        :param inner_expr: Expression in parentheses
        :type inner_expr: Expression
        """
        super().__init__(line)
        assert isinstance(inner_expr, Expression)
        self.inner_expr = inner_expr

    def get_leafs(self) -> Set[str]:
        return self.inner_expr.get_leafs()

    def pre_build(self, quantum_evaluator):
        self.inner_expr.pre_build(quantum_evaluator)

    def n_result_qubits(self, quantum_evaluator) -> int:
        return self.inner_expr.n_result_qubits(quantum_evaluator)

    def needs_result_allocation(self) -> bool:
        return False

    def to_dict(self) -> Dict[str, object]:
```

```python
        return {'type': 'Parentheses', 'inner_expr': self.inner_expr.to_dict()}

    @staticmethod
    def bypass(expr: Expression) -> Expression:
        """If expr is a Parentheses node or a chain of multiple Parentheses nodes, it returns the first inner not-Parentheses expression.

        :param expr: self-descriptive
        :type expr: Expression
        :return: Inner non-Parentheses expression
        :rtype: Expression
        """
        x = expr
        while isinstance(x, Parentheses):
            x = x.inner_expr
        return x

# --- Arithmetic expression AST nodes ---

class ArithmeticExpression(Expression):
    def __init__(self, line: int) -> None:
        """
        :param line: Line of code
        :type line: int
        """
        super().__init__(line)

class UnaryMinus(ArithmeticExpression):
    def __init__(self, line: int, inner_expr: Expression) -> None:
        """
        :param line: Line of code
        :type line: int
        :param inner_expr: Expression to which the unary minus is being applied
        :type inner_expr: Expression
        """
        super().__init__(line)
        assert isinstance(inner_expr, Expression)
        self.inner_expr = inner_expr

    def needs_result_allocation(self) -> bool:
        return False

    def pre_build(self, quantum_evaluator):
        self.inner_expr.pre_build(quantum_evaluator)
```

```

    def get_leafs(self) -> Set[str]:
        return self.inner_expr.get_leafs()

    def to_dict(self) -> Dict[str, object]:
        return {'type': 'UnaryMinus', 'inner_expr': self.inner_expr.
    to_dict()}

class Power(ArithmeticExpression):
    def __init__(self, line: int, base_expr: Expression, exponent: int)
    -> None:
        """
        :param line: Line of code
        :type line: int
        :param base_expr: Expression that is used as base
        :type base_expr: Expression
        :param exponent: Integer that is used as exponent
        :type exponent: int
        """
        super().__init__(line)
        assert isinstance(base_expr, Expression)
        assert isinstance(exponent, int)
        self.base_expr = base_expr
        self.exponent = exponent

    def get_leafs(self) -> Set[str]:
        return self.base_expr.get_leafs()

    def to_dict(self) -> Dict[str, object]:
        return {'type': 'Power', 'base_expr': self.base_expr.to_dict(),
    'exponent': self.exponent}

    def n_result_qubits(self, quantum_evaluator) -> int:
        nb = self.base_expr.n_result_qubits(quantum_evaluator)
        return nb*self.exponent

    def pre_build(self, quantum_evaluator):
        self.base_expr.pre_build(quantum_evaluator)
        self.base_expr = Parentheses.bypass(self.base_expr)

    def build(self, quantum_evaluator) -> List[qiskit.circuit.Qubit]:
        if self.base_expr.needs_result_allocation():
            self.base_expr.alloc_result_qubits(quantum_evaluator)

        if not isinstance(self.base_expr, Identifier):
            self.base_expr.build(quantum_evaluator)

```

```
        qunits.multiproduct(quantum_evaluator.quantum_circuit, [self.
    base_expr.result], [self.exponent], self.result)

    def reverse(self, quantum_evaluator) -> List[qiskit.circuit.Qubit]:
        qunits.multiproduct_dg(quantum_evaluator.quantum_circuit, [self.
    base_expr.result], [self.exponent], self.result)

        if not isinstance(self.base_expr, Identifier):
            self.base_expr.reverse(quantum_evaluator)

        if self.base_expr.needs_result_allocation():
            self.base_expr.release_result_qubits(quantum_evaluator)

class Product(ArithmeticExpression):
    def __init__(self, line: int, operands: List[Union[Expression, int
    ]]) -> None:
        """
        :param line: Line of code
        :type line: int
        :param operands: Operands of the productory
        :type operands: List[Union[Expression, int]]
        :param signals: List with the signals of the operands
        :type signals: List[Signal]
        """
        super().__init__(line)
        assert isinstance(operands, list)
        assert all([isinstance(op, (Expression, int)) for op in operands
    ])
        self.operands = operands
        self.exponents = [1]*len(self.operands)
        self.const_factor = 1
        self.filtered_operands = []
        self.filtered_exponents = []

    @staticmethod
    def merge(line: int, left: Union[Expression, int], right: Union[
    Expression, int]) -> object:
        """Return a Product object to a pair of operands

        :param line: Line of code
        :type line: int
        :param left: Left operand
        :type left: Union[Expression, int]
        :param right: Right operand
        :type right: Union[Expression, int]
        :return: Product object
        :rtype: Product
```

```
        """
        assert isinstance(left, (Expression, int))
        assert isinstance(right, (Expression, int))
        left_operands_list = left.operands if isinstance(left, Product) else [left]
        right_operands_list = [right]
        return Product(line, left_operands_list + right_operands_list)

    def get_leafs(self) -> Set[str]:
        leafs = set()
        for op in self.operands:
            if isinstance(op, Expression):
                leafs = leafs.union(op.get_leafs())
        return leafs

    def to_dict(self) -> Dict[str, object]:
        return {
            'type': 'Product',
            'operands': [(op.to_dict() if isinstance(op, Expression) else op) for op in self.operands]
        }

    def n_result_qubits(self, quantum_evaluator) -> int:
        return sum([self.filtered_operands[i].n_result_qubits(quantum_evaluator)*self.filtered_exponents[i] for i in range(len(self.filtered_operands))]) + n_bits_const(self.const_factor)

    def pre_build(self, quantum_evaluator):
        #--- merge power operations ---
        for i in range(len(self.operands)):
            if isinstance(self.operands[i], int):
                self.const_factor *= self.operands[i]
            else:
                self.operands[i].pre_build(quantum_evaluator)
                while isinstance(self.operands[i], (Power, Parentheses)):
                    self.operands[i] = Parentheses.bypass(self.operands[i])
                    if isinstance(self.operands[i], Power):
                        self.exponents[i] *= self.operands[i].exponent
                        self.operands[i] = self.operands[i].base_expr
                self.filtered_operands.append(self.operands[i])
                self.filtered_exponents.append(self.exponents[i])

    def build(self, quantum_evaluator):
        for op in self.filtered_operands:
            if op.needs_result_allocation():
```

```
                op.alloc_result_qubits(quantum_evaluator)

            if not isinstance(op, Identifier):
                op.build(quantum_evaluator)

        qunits.multiproduct(quantum_evaluator.quantum_circuit, [op.result for op in self.filtered_operands], self.filtered_exponents, self.result, self.const_factor)

    def reverse(self, quantum_evaluator):
        qunits.multiproduct_dg(quantum_evaluator.quantum_circuit, [op.result for op in self.filtered_operands], self.filtered_exponents, self.result, self.const_factor)

        for op in self.filtered_operands[::-1]:
            if not isinstance(op, Identifier):
                op.reverse()

            if op.needs_result_allocation():
                op.release_result_qubits(quantum_evaluator)

class Summation(ArithmeticExpression):
    def __init__(self, line: int, operands: List[Union[Expression, int]], signals: List[Signal]) -> None:
        """
        :param line: Line of code
        :type line: int
        :param operands: Operands of the summation
        :type operands: List[Union[Expression, int]]
        :param signals: List with the signals of the operands
        :type signals: List[Signal]
        """
        super().__init__(line)
        assert isinstance(operands, list)
        assert isinstance(signals, list)
        assert all([isinstance(op, (Expression, int)) for op in operands])
        assert all([isinstance(sig, Signal) for sig in signals])
        self.operands = operands
        self.signals = signals
        self.const_term = 0
        self.filtered_operands = []
        self.filtered_signals = []

    @staticmethod
    def merge_add(line: int, left: Union[Expression, int], right: Union[Expression, int]) -> object:
```

```python
        """Return a Summation object to a pair of addition operands

        :param line: Line of code
        :type line: int
        :param left: Left operand
        :type left: Union[Expression, int]
        :param right: Right operand
        :type right: Union[Expression, int]
        :return: Summation object
        :rtype: Summation
        """
        assert isinstance(left, (Expression, int))
        assert isinstance(right, (Expression, int))
        left_operands_list = left.operands if isinstance(left, Summation) else [left]
        right_operands_list = [right]
        left_signals_list = left.signals if isinstance(left, Summation) else [Signal.POS]
        right_signals_list = [Signal.POS]
        return Summation(line, left_operands_list + right_operands_list, left_signals_list + right_signals_list)

    @staticmethod
    def merge_sub(line: int, left: Union[Expression, int], right: Union[Expression, int]) -> object:
        """Return a Summation object to a pair of subtraction operands

        :param line: Line of code
        :type line: int
        :param left: Left operand
        :type left: Union[Expression, int]
        :param right: Right operand
        :type right: Union[Expression, int]
        :return: Summation object
        :rtype: Summation
        """
        assert isinstance(left, (Expression, int))
        assert isinstance(right, (Expression, int))
        left_operands_list = left.operands if isinstance(left, Summation) else [left]
        right_operands_list = [right]
        left_signals_list = left.signals if isinstance(left, Summation) else [Signal.POS]
        right_signals_list = [Signal.NEG]
        return Summation(line, left_operands_list + right_operands_list, left_signals_list + right_signals_list)

```

```
    def get_leafs(self) -> Set[str]:
        leafs = set()
        for op in self.operands:
            if isinstance(op, Expression):
                leafs = leafs.union(op.get_leafs())
        return leafs

    def to_dict(self) -> Dict[str, object]:
        return {
            'type': 'Summation',
            'operands': [(op.to_dict() if isinstance(op, Expression)
else op) for op in self.operands],
            'signals': [('+' if sig == Signal.POS else '-') for sig in
self.signals]
        }

    def pre_build(self, quantum_evaluator):
        for i in range(len(self.operands)):
            if isinstance(self.operands[i], int):
                self.const_term += self.operands[i] if self.signals[i]
== Signal.POS else -self.operands[i]
            else:
                self.operands[i].pre_build(quantum_evaluator)
                while isinstance(self.operands[i], (Parentheses,
UnaryMinus)):
                    self.operands[i] = Parentheses.bypass(self.operands[
i])
                    if isinstance(self.operands[i], UnaryMinus):
                        self.operands[i] = self.operands[i].inner_expr
                        self.signals[i] = Signal.POS if self.signals[i]
== Signal.NEG else Signal.NEG
                self.filtered_operands.append(self.operands[i])
                self.filtered_signals.append(self.signals[i])

    def n_result_qubits(self, quantum_evaluator) -> int:
        return max([op.n_result_qubits(quantum_evaluator)+1 for op in
self.filtered_operands] + [n_bits_const(self.const_term)])

    def build(self, quantum_evaluator):
        for i in range(len(self.filtered_operands)):
            if self.filtered_operands[i].needs_result_allocation():
                self.filtered_operands[i].alloc_result_qubits(
quantum_evaluator)

            if not isinstance(self.filtered_operands[i], Identifier):
                self.filtered_operands[i].build(quantum_evaluator)

```

```
                if self.filtered_signals[i] == Signal.POS:
                    qunits.register_by_register_addition(quantum_evaluator.quantum_circuit, self.filtered_operands[i].result, self.result)
                else:
                    qunits.register_by_register_subtraction(quantum_evaluator.quantum_circuit, self.filtered_operands[i].result, self.result)

    def reverse(self, quantum_evaluator):
        for i in range(len(self.filtered_operands)):
            if self.filtered_signals[i] == Signal.POS:
                qunits.register_by_register_addition_dg(quantum_evaluator.quantum_circuit, self.filtered_operands[i].result, self.result)
            else:
                qunits.register_by_register_subtraction_dg(quantum_evaluator.quantum_circuit, self.filtered_operands[i].result, self.result)

            if not isinstance(self.filtered_operands[i], Identifier):
                self.filtered_operands[i].reverse(quantum_evaluator)

            if self.filtered_operands[i].needs_result_allocation():
                self.filtered_operands[i].release_result_qubits(quantum_evaluator)

# --- Relational expression AST nodes ---

class RelationalExpression(Expression):
    def __init__(self, line: int, left: Union[Expression, int], right: Union[Expression, int]) -> None:
        """
        :param line: Line of code
        :type line: int
        :param left: Left operand
        :type left: Union[Expression, int]
        :param right: Right operand
        :type right: Union[Expression, int]
        """
        super().__init__(line)
        assert isinstance(left, (Expression, int))
        assert isinstance(right, (Expression, int))
        self.left = left
        self.right = right
        self.aux = None
        self.mode = '' #can be '', 'rr', 'rc' or 'cr'

```

```
    def get_leafs(self) -> Set[str]:
        if isinstance(self.left, Expression) and isinstance(self.right, Expression):
            return self.left.get_leafs().union(self.right.get_leafs())
        elif isinstance(self.left, Expression) and isinstance(self.right, int):
            return self.left.get_leafs()
        else:
            return self.right.get_leafs()

    def to_dict(self) -> Dict[str, object]:
        return {
            'type': self.__class__.__name__,
            'left': self.left if isinstance(self.left, int) else self.left.to_dict(),
            'right': self.right if isinstance(self.right, int) else self.right.to_dict()
        }

    def n_result_qubits(self, quantum_evaluator) -> int:
        return 1

    def pre_build(self, quantum_evaluator):
        self.left = Parentheses.bypass(self.left)
        self.right = Parentheses.bypass(self.right)
        if isinstance(self.left, Expression) and isinstance(self.right, Expression):
            self.mode = 'rr'
        elif isinstance(self.left, Expression) and isinstance(self.right, int):
            self.mode = 'rc'
        else:
            self.mode = 'cr'

class Equal(RelationalExpression):
    def __init__(self, line: int, left: Expression | int, right: Expression | int) -> None:
        super().__init__(line, left, right)

    def pre_build(self, quantum_evaluator):
        super().pre_build(quantum_evaluator)
        if self.mode == 'rr':
            self.left.pre_build(quantum_evaluator)
            self.right.pre_build(quantum_evaluator)
            n = max([self.left.n_result_qubits(quantum_evaluator), self.right.n_result_qubits(quantum_evaluator)]) - min([self.left.n_result_qubits(quantum_evaluator), self.right.n_result_qubits(
```

```
quantum_evaluator)])
            self.aux = [quantum_evaluator.alloc_ancilla() for i in range(n)]
        elif self.mode == 'rc':
            self.left.pre_build(quantum_evaluator)
            n = max([math.ceil(math.log2(self.right)) - self.left.n_result_qubits(quantum_evaluator), 0])
            self.aux = [quantum_evaluator.alloc_ancilla() for i in range(n)]
        elif self.mode == 'cr':
            self.right.pre_build(quantum_evaluator)
            n = max([math.ceil(math.log2(self.left)) - self.right.n_result_qubits(quantum_evaluator), 0])
            self.aux = [quantum_evaluator.alloc_ancilla() for i in range(n)]
        else:
            raise Exception('Undefined mode')

    def build(self, quantum_evaluator):
        if self.mode == 'rr':
            if self.left.needs_result_allocation(): self.left.alloc_result_qubits(quantum_evaluator)
            if self.right.needs_result_allocation(): self.right.alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.left, Identifier): self.left.build(quantum_evaluator)
            if not isinstance(self.right, Identifier): self.right.build(quantum_evaluator)
            qunits.register_equal_register(quantum_evaluator.quantum_circuit, self.left.result, self.right.result, self.aux, self.result[0])
        elif self.mode == 'rc':
            if self.left.needs_result_allocation(): self.left.alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.left, Identifier): self.left.build(quantum_evaluator)
            qunits.register_equal_constant(quantum_evaluator.quantum_circuit, self.left.result, self.right, self.aux, self.result[0])
        elif self.mode == 'cr':
            if self.right.needs_result_allocation(): self.right.alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.right, Identifier): self.right.build(quantum_evaluator)
            qunits.register_equal_constant(quantum_evaluator.quantum_circuit, self.right.result, self.left, self.aux, self.result[0])
```

```python
        else:
            raise Exception('Undefined mode')

    def reverse(self, quantum_evaluator):
        if self.mode == 'rr':
            qunits.register_equal_register_dg(quantum_evaluator.quantum_circuit, self.left.result, self.right.result, self.aux, self.result[0])
            if not isinstance(self.left, Identifier): self.left.reverse(quantum_evaluator)
            if not isinstance(self.right, Identifier): self.right.reverse(quantum_evaluator)
            if self.left.needs_result_allocation(): self.left.release_result_qubits(quantum_evaluator)
            if self.right.needs_result_allocation(): self.right.release_result_qubits(quantum_evaluator)
        elif self.mode == 'rc':
            qunits.register_equal_constant_dg(quantum_evaluator.quantum_circuit, self.left.result, self.right, self.aux, self.result[0])
            if not isinstance(self.left, Identifier): self.left.reverse(quantum_evaluator)
            if self.left.needs_result_allocation(): self.left.release_result_qubits(quantum_evaluator)
        elif self.mode == 'cr':
            qunits.register_equal_constant_dg(quantum_evaluator.quantum_circuit, self.right.result, self.left, self.aux, self.result[0])
            if not isinstance(self.right, Identifier): self.right.reverse(quantum_evaluator)
            if self.right.needs_result_allocation(): self.right.release_result_qubits(quantum_evaluator)
        else:
            raise Exception('Undefined mode')

class NotEqual(RelationalExpression):
    def __init__(self, line: int, left: Expression | int, right: Expression | int) -> None:
        super().__init__(line, left, right)

    def pre_build(self, quantum_evaluator):
        super().pre_build(quantum_evaluator)
        if self.mode == 'rr':
            self.left.pre_build(quantum_evaluator)
            self.right.pre_build(quantum_evaluator)
            nl = self.left.n_result_qubits(quantum_evaluator)
            nr = self.right.n_result_qubits(quantum_evaluator)
```

```
            n = max([nl, nr]) - min([nl, nr])
            self.aux = [quantum_evaluator.alloc_ancilla() for i in range(n)]
        elif self.mode == 'rc':
            self.left.pre_build(quantum_evaluator)
            n = max([math.ceil(math.log2(self.right)) - self.left.n_result_qubits(quantum_evaluator), 0])
            self.aux = [quantum_evaluator.alloc_ancilla() for i in range(n)]
        elif self.mode == 'cr':
            self.right.pre_build(quantum_evaluator)
            n = max([math.ceil(math.log2(self.left)) - self.right.n_result_qubits(quantum_evaluator), 0])
            self.aux = [quantum_evaluator.alloc_ancilla() for i in range(n)]
        else:
            raise Exception('Undefined mode')

    def build(self, quantum_evaluator):
        if self.mode == 'rr':
            if self.left.needs_result_allocation(): self.left.alloc_result_qubits(quantum_evaluator)
            if self.right.needs_result_allocation(): self.right.alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.left, Identifier): self.left.build(quantum_evaluator)
            if not isinstance(self.right, Identifier): self.right.build(quantum_evaluator)
            qunits.register_not_equal_register(quantum_evaluator.quantum_circuit, self.left.result, self.right.result, self.aux, self.result[0])
        elif self.mode == 'rc':
            if self.left.needs_result_allocation(): self.left.alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.left, Identifier): self.left.build(quantum_evaluator)
            qunits.register_not_equal_constant(quantum_evaluator.quantum_circuit, self.left.result, self.right, self.aux, self.result[0])
        elif self.mode == 'cr':
            if self.right.needs_result_allocation(): self.right.alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.right, Identifier): self.right.build(quantum_evaluator)
            qunits.register_not_equal_constant(quantum_evaluator.quantum_circuit, self.right.result, self.left, self.aux, self.result[0])
```

```
        else:
            raise Exception('Undefined mode')

    def reverse(self, quantum_evaluator):
        if self.mode == 'rr':
            qunits.register_not_equal_register_dg(quantum_evaluator.
    quantum_circuit, self.left.result, self.right.result, self.aux, self.
    result[0])
            if not isinstance(self.left, Identifier): self.left.reverse(
    quantum_evaluator)
            if not isinstance(self.right, Identifier): self.right.
    reverse(quantum_evaluator)
            if self.left.needs_result_allocation(): self.left.
    release_result_qubits(quantum_evaluator)
            if self.right.needs_result_allocation(): self.right.
    release_result_qubits(quantum_evaluator)
        elif self.mode == 'rc':
            qunits.register_not_equal_constant_dg(quantum_evaluator.
    quantum_circuit, self.left.result, self.right, self.aux, self.result
    [0])
            if not isinstance(self.left, Identifier): self.left.reverse(
    quantum_evaluator)
            if self.left.needs_result_allocation(): self.left.
    release_result_qubits(quantum_evaluator)
        elif self.mode == 'cr':
            qunits.register_not_equal_constant_dg(quantum_evaluator.
    quantum_circuit, self.right.result, self.left, self.aux, self.result
    [0])
            if not isinstance(self.right, Identifier): self.right.
    reverse(quantum_evaluator)
            if self.right.needs_result_allocation(): self.right.
    release_result_qubits(quantum_evaluator)
        else:
            raise Exception('Undefined mode')

class LessThan(RelationalExpression):
    def __init__(self, line: int, left: Expression | int, right:
    Expression | int) -> None:
        super().__init__(line, left, right)

    def pre_build(self, quantum_evaluator):
        super().pre_build(quantum_evaluator)
        if self.mode == 'rr':
            self.left.pre_build(quantum_evaluator)
            self.right.pre_build(quantum_evaluator)
            nl = self.left.n_result_qubits(quantum_evaluator)
            nr = self.right.n_result_qubits(quantum_evaluator)
```

```
            n = (nr - nl if nr > nl else 0) + 1
            self.aux = [quantum_evaluator.alloc_ancilla() for i in range(n)]
        elif self.mode == 'rc':
            self.left.pre_build(quantum_evaluator)
            nl = self.left.n_result_qubits(quantum_evaluator)
            m = nl - max([nl, math.ceil(math.log2(self.right))])
            n = max([m, 0]) + 1
            self.aux = [quantum_evaluator.alloc_ancilla() for i in range(n)]
        elif self.mode == 'cr':
            self.right.pre_build(quantum_evaluator)
            nr = self.right.n_result_qubits(quantum_evaluator)
            m = nr - max([nr, math.ceil(math.log2(self.left))])
            n = max([m, 0]) + 1
            self.aux = [quantum_evaluator.alloc_ancilla() for i in range(n)]
        else:
            raise Exception('Undefined mode')

    def build(self, quantum_evaluator):
        if self.mode == 'rr':
            if self.left.needs_result_allocation(): self.left.alloc_result_qubits(quantum_evaluator)
            if self.right.needs_result_allocation(): self.right.alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.left, Identifier): self.left.build(quantum_evaluator)
            if not isinstance(self.right, Identifier): self.right.build(quantum_evaluator)
            qunits.register_less_than_register(quantum_evaluator.quantum_circuit, self.left.result, self.right.result, self.aux, self.result[0])
        elif self.mode == 'rc':
            if self.left.needs_result_allocation(): self.left.alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.left, Identifier): self.left.build(quantum_evaluator)
            qunits.register_less_than_constant(quantum_evaluator.quantum_circuit, self.left.result, self.right, self.aux, self.result[0])
        elif self.mode == 'cr':
            if self.right.needs_result_allocation(): self.right.alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.right, Identifier): self.right.build(quantum_evaluator)
            qunits.register_greater_than_constant(quantum_evaluator.
```

```
quantum_circuit, self.right.result, self.left, self.aux, self.result[0])
        else:
            raise Exception('Undefined mode')

    def reverse(self, quantum_evaluator):
        if self.mode == 'rr':
            qunits.register_less_than_register_dg(quantum_evaluator.quantum_circuit, self.left.result, self.right.result, self.aux, self.result[0])
            if not isinstance(self.left, Identifier): self.left.reverse(quantum_evaluator)
            if not isinstance(self.right, Identifier): self.right.reverse(quantum_evaluator)
            if self.left.needs_result_allocation(): self.left.release_result_qubits(quantum_evaluator)
            if self.right.needs_result_allocation(): self.right.release_result_qubits(quantum_evaluator)
        elif self.mode == 'rc':
            qunits.register_less_than_constant_dg(quantum_evaluator.quantum_circuit, self.left.result, self.right, self.aux, self.result[0])
            if not isinstance(self.left, Identifier): self.left.reverse(quantum_evaluator)
            if self.left.needs_result_allocation(): self.left.release_result_qubits(quantum_evaluator)
        elif self.mode == 'cr':
            qunits.register_greater_than_constant_dg(quantum_evaluator.quantum_circuit, self.right.result, self.left, self.aux, self.result[0])
            if not isinstance(self.right, Identifier): self.right.reverse(quantum_evaluator)
            if self.right.needs_result_allocation(): self.right.release_result_qubits(quantum_evaluator)
        else:
            raise Exception('Undefined mode')

class GreaterThan(RelationalExpression):
    def __init__(self, line: int, left: Expression | int, right: Expression | int) -> None:
        super().__init__(line, left, right)

    def pre_build(self, quantum_evaluator):
        super().pre_build(quantum_evaluator)
        if self.mode == 'rr':
            self.left.pre_build(quantum_evaluator)
            self.right.pre_build(quantum_evaluator)
```

```
            nl = self.left.n_result_qubits(quantum_evaluator)
            nr = self.right.n_result_qubits(quantum_evaluator)
            n = (nl - nr if nl > nr else 0) + 1
            self.aux = [quantum_evaluator.alloc_ancilla() for i in range(n)]
        elif self.mode == 'rc':
            self.left.pre_build(quantum_evaluator)
            nl = self.left.n_result_qubits(quantum_evaluator)
            m = nl - max([nl, math.ceil(math.log2(self.right))])
            n = max([m, 0]) + 1
            self.aux = [quantum_evaluator.alloc_ancilla() for i in range(n)]
        elif self.mode == 'cr':
            self.right.pre_build(quantum_evaluator)
            nr = self.right.n_result_qubits(quantum_evaluator)
            m = nr - max([nr, math.ceil(math.log2(self.left))])
            n = max([m, 0]) + 1
            self.aux = [quantum_evaluator.alloc_ancilla() for i in range(n)]
        else:
            raise Exception('Undefined mode')

    def build(self, quantum_evaluator):
        if self.mode == 'rr':
            if self.left.needs_result_allocation(): self.left.alloc_result_qubits(quantum_evaluator)
            if self.right.needs_result_allocation(): self.right.alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.left, Identifier): self.left.build(quantum_evaluator)
            if not isinstance(self.right, Identifier): self.right.build(quantum_evaluator)
            qunits.register_greater_than_register(quantum_evaluator.quantum_circuit, self.left.result, self.right.result, self.aux, self.result[0])
        elif self.mode == 'rc':
            if self.left.needs_result_allocation(): self.left.alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.left, Identifier): self.left.build(quantum_evaluator)
            qunits.register_greater_than_constant(quantum_evaluator.quantum_circuit, self.left.result, self.right, self.aux, self.result[0])
        elif self.mode == 'cr':
            if self.right.needs_result_allocation(): self.right.alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.right, Identifier): self.right.build(
```

```
quantum_evaluator)
            qunits.register_less_than_constant(quantum_evaluator.quantum_circuit, self.right.result, self.left, self.aux, self.result[0])
        else:
            raise Exception('Undefined mode')

    def reverse(self, quantum_evaluator):
        if self.mode == 'rr':
            qunits.register_greater_than_register_dg(quantum_evaluator.quantum_circuit, self.left.result, self.right.result, self.aux, self.result[0])
            if not isinstance(self.left, Identifier): self.left.reverse(quantum_evaluator)
            if not isinstance(self.right, Identifier): self.right.reverse(quantum_evaluator)
            if self.left.needs_result_allocation(): self.left.release_result_qubits(quantum_evaluator)
            if self.right.needs_result_allocation(): self.right.release_result_qubits(quantum_evaluator)
        elif self.mode == 'rc':
            qunits.register_greater_than_constant_dg(quantum_evaluator.quantum_circuit, self.left.result, self.right, self.aux, self.result[0])
            if not isinstance(self.left, Identifier): self.left.reverse(quantum_evaluator)
            if self.left.needs_result_allocation(): self.left.release_result_qubits(quantum_evaluator)
        elif self.mode == 'cr':
            qunits.register_less_than_constant_dg(quantum_evaluator.quantum_circuit, self.right.result, self.left, self.aux, self.result[0])
            if not isinstance(self.right, Identifier): self.right.reverse(quantum_evaluator)
            if self.right.needs_result_allocation(): self.right.release_result_qubits(quantum_evaluator)
        else:
            raise Exception('Undefined mode')

# --- logic expression AST nodes ---

class LogicExpression(Expression):
    def __init__(self, line: int) -> None:
        """
        :param line: Line of code
        :type line: int
        """
```

```
        super().__init__(line)

    def n_result_qubits(self, quantum_evaluator) -> int:
        return 1

class Not(LogicExpression):
    def __init__(self, line: int, operand: Expression) -> None:
        """
        :param line: line of code
        :type line: int
        :param operand: self-descriptive
        :type operand: Expression
        """
        super().__init__(line)
        assert isinstance(operand, Expression)
        self.operand = operand

    def to_dict(self) -> Dict[str, object]:
        return {
            'type': 'Not',
            'operand': self.operand if isinstance(self.operand, int) else self.operand.to_dict()
        }

    def pre_build(self, quantum_evaluator):
        Parentheses.bypass(self.operand)
        self.operand.pre_build(quantum_evaluator)

    def build(self, quantum_evaluator):
        if self.operand.needs_result_allocation():
            self.operand.alloc_result_qubits(quantum_evaluator)
        if not isinstance(self.operand, Identifier):
            self.operand.build(quantum_evaluator)
        quantum_evaluator.quantum_circuit.cx(self.operand.result[-1], self.result[-1])
        quantum_evaluator.quantum_circuit.x(self.result[-1])

    def reverse(self, quantum_evaluator):
        quantum_evaluator.quantum_circuit.x(self.result[-1])
        quantum_evaluator.quantum_circuit.cx(self.operand.result[-1], self.result[-1])
        if not isinstance(self.operand, Identifier):
            self.operand.reverse(quantum_evaluator)
        if self.operand.needs_result_allocation():
            self.operand.release_result_qubits(quantum_evaluator)

class And(LogicExpression):
```

```
    def __init__(self, line: int, operands: List[Expression]) -> None:
        """
        :param line: line of code
        :type line: int
        :param operands: self-descriptive
        :type operands: List[Expression]
        """
        super().__init__(line)
        assert all([isinstance(operand, Expression) for operand in
    operands])
        self.operands = operands

    @staticmethod
    def merge(line: int, left: Expression, right: Expression) -> object:
        """Return a And object to a pair of operands

        :param line: Line of code
        :type line: int
        :param left: Left operand
        :type left: Expression
        :param right: Right operand
        :type right: Expression
        :return: Product object
        :rtype: Product
        """
        assert isinstance(left, Expression)
        assert isinstance(right, Expression)
        left_operands_list = left.operands if isinstance(left, And) else
     [left]
        right_operands_list = [right]
        return And(line, left_operands_list + right_operands_list)

    def to_dict(self) -> Dict[str, object]:
        return {
            'type': 'And',
            'operands': [(op if isinstance(op, int) else op.to_dict())
    for op in self.operands]
        }

    def pre_build(self, quantum_evaluator):
        for i in range(len(self.operands)):
            self.operands[i] = Parentheses.bypass(self.operands[i])
            self.operands[i].pre_build(quantum_evaluator)

    def build(self, quantum_evaluator):
        for i in range(len(self.operands)):
            if self.operands[i].needs_result_allocation():
```

```
                self.operands[i].alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.operands[i], Identifier):
                self.operands[i].build(quantum_evaluator)
        quantum_evaluator.quantum_circuit.mcx([op.result[-1] for op in self.operands], self.result[-1])

    def reverse(self, quantum_evaluator):
        quantum_evaluator.quantum_circuit.mcx([op.result[-1] for op in self.operands], self.result[-1])
        for i in range(len(self.operands)):
            if not isinstance(self.operands[i], Identifier):
                self.operands[i].reverse(quantum_evaluator)
            if self.operands[i].needs_result_allocation():
                self.operands[i].release_result_qubits(quantum_evaluator)

class Or(LogicExpression):
    def __init__(self, line: int, operands: List[Expression]) -> None:
        """
        :param line: Line of code
        :type line: int
        :param operands: self-descriptive
        :type operands: List[Expression]
        """
        super().__init__(line)
        assert all([isinstance(operand, Expression) for operand in operands])
        self.operands = operands

    @staticmethod
    def merge(line: int, left: Expression, right: Expression) -> object:
        """Return a Or object to a pair of operands

        :param line: Line of code
        :type line: int
        :param left: Left operand
        :type left: Expression
        :param right: Right operand
        :type right: Expression
        :return: Product object
        :rtype: Product
        """
        assert isinstance(left, Expression)
        assert isinstance(right, Expression)
        left_operands_list = left.operands if isinstance(left, Or) else [left]
        right_operands_list = [right]
```

```python
        return Or(line, left_operands_list + right_operands_list)

    def to_dict(self) -> Dict[str, object]:
        return {
            'type': 'Or',
            'operands': [(op if isinstance(op, int) else op.to_dict()) for op in self.operands]
        }

    def pre_build(self, quantum_evaluator):
        for i in range(len(self.operands)):
            self.operands[i] = Parentheses.bypass(self.operands[i])
            self.operands[i].pre_build(quantum_evaluator)

    def build(self, quantum_evaluator):
        for i in range(len(self.operands)):
            if self.operands[i].needs_result_allocation():
                self.operands[i].alloc_result_qubits(quantum_evaluator)
            if not isinstance(self.operands[i], Identifier):
                self.operands[i].build(quantum_evaluator)
        quantum_evaluator.quantum_circuit.x([op.result[-1] for op in self.operands])
        quantum_evaluator.quantum_circuit.mcx([op.result[-1] for op in self.operands], self.result[-1])
        quantum_evaluator.quantum_circuit.x([op.result[-1] for op in self.operands])
        quantum_evaluator.quantum_circuit.x(self.result[-1])

    def reverse(self, quantum_evaluator):
        quantum_evaluator.quantum_circuit.x(self.result[-1])
        quantum_evaluator.quantum_circuit.x([op.result[-1] for op in self.operands])
        quantum_evaluator.quantum_circuit.mcx([op.result[-1] for op in self.operands], self.result[-1])
        quantum_evaluator.quantum_circuit.x([op.result[-1] for op in self.operands])
        for i in range(len(self.operands)):
            if not isinstance(self.operands[i], Identifier):
                self.operands[i].reverse(quantum_evaluator)
            if self.operands[i].needs_result_allocation():
                self.operands[i].release_result_qubits(quantum_evaluator)

# --- Register definition AST nodes ---
class RegisterDefinition(ASTNode):
    def __init__(self, line: int, name: str, n: int) -> None:
        """
```

```
        :param line: Line of code
        :type line: int
        :param name: Register's label/name
        :type name: str
        :param n: Register's size (number of qubits)
        :type n: int
        """
        super().__init__(line)
        assert isinstance(name, str)
        assert isinstance(n, int)
        self.name = name
        self.n = n

class RegisterExpressionDefinition(RegisterDefinition):
    def __init__(self, line: int, name: str, n: int, expr: Expression) -> None:
        """
        :param line: Line of code
        :type line: int
        :param name: Register's name
        :type name: str
        :param n: Register's size
        :type n: int
        :param expr: Target expression
        :type expr: Expression
        """
        super().__init__(line, name, n)
        assert isinstance(expr, Expression)
        self.expr = expr
        self.target = None

    def to_dict(self) -> Dict[str, object]:
        return {
            'type': 'RegisterExpressionDefinition',
            'name': self.name,
            'size': self.n,
            'expr': self.expr.to_dict()
        }

    def pre_build(self, quantum_evaluator):
        """Pre-build the inner expression tree

        :param quantum_evaluator: Parent quantum evaluator
        :type quantum_evaluator: QuantumEvaluator
        """
        self.target = [qbit for qbit in quantum_evaluator.get_qiskit_register(self.name)]
```

```
        self.expr.pre_build(quantum_evaluator)

    def build(self, quantum_evaluator):
        """Build the quantum circuit that evaluates the inner expression
    and asign the result to the target register

        :param quantum_evaluator: Parent quantum evaluator
        :type quantum_evaluator: QuantumEvaluator
        """
        #if self.expr.needs_result_allocation(): self.expr.
    alloc_result_qubits(quantum_evaluator)
        self.expr.result = self.target
        self.expr.build(quantum_evaluator)
        #qunits.register_by_register_addition(quantum_evaluator.
    quantum_circuit, self.expr.result, self.target)

    def reverse(self, quantum_evaluator):
        """Build the inverse quantum circuit

        :param quantum_evaluator: Parent quantum evaluator
        :type quantum_evaluator: QuantumEvaluator
        """
        #qunits.register_by_register_subtraction(quantum_evaluator.
    quantum_circuit, self.expr.result, self.target)
        self.expr.reverse(quantum_evaluator)
        #if self.expr.needs_result_allocation(): self.expr.
    release_result_qubits(quantum_evaluator)

class RegisterSetDefinition(RegisterDefinition):
    def __init__(self, line: int, name: str, n: int, values: Set[int])
    -> None:
        """
        :param line: Line of code
        :type line: int
        :param name: Register's name
        :type name: str
        :param n: Register's size
        :type n: int
        :param expr: Target expression
        :type expr: Expression
        """
        super().__init__(line, name, n)
        assert all([isinstance(v, int) for v in values])
        self.values = values

    def to_dict(self) -> Dict[str, object]:
        return {
```

```
            'type': 'RegisterSetDefinition',
            'name': self.name,
            'size': self.n,
            'values': self.values
        }

# --- Terminators ---

class Terminator(ASTNode):
    def __init__(self, line: int) -> None:
        """
        :param line: Line of code
        :type line: int
        """
        super().__init__(line)

class Amplify(Terminator):
    def __init__(self, line: int, target: str, iterations: int) -> None:
        """
        :param line: Line of code
        :type line: int
        :param target: Label of the target register
        :type target: str
        :param iterations: Number of iterations of the Grover's search
    algorithm
        :type iterations: int
        """
        super().__init__(line)
        assert isinstance(target, str)
        assert isinstance(iterations, int)
        self.target = target
        self.it = iterations

    def to_dict(self) -> Dict[str, object]:
        return {
            'type': 'Amplify',
            'target': self.target,
            'iterations': self.it
        }

# --- Full code AST node ---

class FullCode(ASTNode):
    def __init__(self, line: int, regdefseq: List[RegisterDefinition],
    terminator: Terminator) -> None:
        """
        :param line: Line of code
```

```
        :type line: int
        :param regdefseq: Register definition sequence
        :type regdefseq: List[RegisterDefinition]
        :param terminator: terminator AST node
        :type terminator: Terminator
        """
        super().__init__(line)
        assert isinstance(regdefseq, list)
        assert all([isinstance(x, RegisterDefinition) for x in regdefseq
    ])
        assert isinstance(terminator, Terminator)
        self.regdefseq = regdefseq
        self.terminator = terminator

    def to_dict(self) -> Dict[str, object]:
        return {
            'type': 'FullCode',
            'sequence': [x.to_dict() for x in self.regdefseq],
            'terminator': self.terminator.to_dict()
        }

    def get_reg_names_sizes_and_sets(self) -> List[Tuple[str, int, set
    ]]:
        """Return a list with a tuple for each defined register. Each
    tuple have a label, a size and a set of values.

        :return: List of tuples
        :rtype: List[Tuple[str, int, set]]
        """
        return [(reg.name, reg.n, reg.values) if isinstance(reg,
    RegisterSetDefinition) else (reg.name, reg.n, None) for reg in self.
    regdefseq]

    def check_definition_errors(self):
        """Check if there are errors of multiple definitions of the same
     identifier or missing identifier definitions

        :raises SynthError: Identifier already defined
        :raises SynthError: Identifier not defined
        """
        defined_registers = []
        for regdef in self.regdefseq:
            if regdef.name in defined_registers:
                raise SynthError(regdef.line, f'"{regdef.name}" is
    already defined')
            if isinstance(regdef, RegisterExpressionDefinition):
                for leaf in regdef.expr.get_leafs():
```

```
                    if not leaf in defined_registers:
                        raise SynthError(regdef.line, f'The identifier {leaf} is not defined')
            defined_registers.append(regdef.name)

        if isinstance(self.terminator, Amplify):
            if self.terminator.target not in defined_registers:
                raise SynthError(regdef.line, f'The target "{self.terminator.target}" specified at the amplify terminator is not defined')
```

Listing B.5 – ast.py

B.6 Módulo *synth.py*

```
#Filipe Chagas, 2023

import qiskit
from qiskit.providers.aer import AerSimulator
import pandas as pd
from qiskit.circuit.library.data_preparation.state_preparation import StatePreparation
from dlqpiler.ast import FullCode, RegisterExpressionDefinition, RegisterSetDefinition
from dlqpiler import utils
from typing import *

def reg_init_gate(values: Set[int], size: int) -> qiskit.circuit.Gate:
    """Returns the quantum initialization gate for a set of values

    :param values: Values    with    which the register should be initialized
    :type values: Set[int]
    :param size: Register size
    :type size: int
    :return: StatePreparation gate
    :rtype: qiskit.circuit.Gate
    """
    if utils.is_none(values) or len(values)==0:
        qc = qiskit.QuantumCircuit(size)
        return qc.to_gate()
    else:
        return StatePreparation(utils.set_to_statevector(values, size))

def organize_qiskit_result(result_counts: Dict[str, int], registers_names: List[str], test_function: Callable[[dict], str] = None) -> pd.DataFrame:
```

```python
    """Organize the results of the execution of a quantum circuit in a DataFrame.

    :param result_counts: Counts dict returned by Qiskit.
    :type result_counts: Dict[str, int]
    :param registers_names: List with the names of the registers.
    :type registers_names: List[str]
    :param registers_dtypes: List with the data types of the registers.
    :type registers_dtypes: List[QQDTypes]
    :param test_function: A function that receives a dict with a value for each register and return a status about it.  Default to None.
    :type test_function: Callable[[dict], str], optional.
    :return: DataFrame containing the value of each register in each outcome and the frequencies of the outcomes.
    :rtype: pd.DataFrame
    """
    out_dict = {key:[] for key in registers_names+['$freq'] + (['$comment'] if isinstance(test_function, Callable) else [])}

    for full_bit_string in result_counts.keys():
        freq = result_counts[full_bit_string] #absolute frequency of the current bit-string
        reg_bit_string = full_bit_string.split(' ') #separate registers

        #Convert bit-strings to naturals, integers or booleans
        reg_data_list = []
        for i in range(len(reg_bit_string)):
            reg_data_list.append(utils.binary_to_natural([True if c=='1' else False for c in reg_bit_string[i]][::-1]))

        #Append results to the output dictionary
        for i in range(len(reg_bit_string)):
            out_dict[registers_names[i]].append(reg_data_list[-1-i])

        #Append tests to the output dict
        if isinstance(test_function, Callable):
            d = {registers_names[i]:reg_data_list[-1-i] for i in range(len(reg_bit_string))}
            tr = test_function(d)
            out_dict['$comment'].append(tr)

        out_dict['$freq'].append(freq)

    return pd.DataFrame(out_dict, ).sort_values('$freq', ascending=False).reset_index(drop=True)

class QuantumEvaluator():
```

```
    def __init__(self, code: FullCode) -> None:
        """
        :param code: Root ASTNode generated by the PLY analysis
        :type code: FullCode
        """
        self.quantum_circuit = qiskit.QuantumCircuit()
        code.check_definition_errors()
        self.code = code

        # --- Main registers ---
        self.main_registers_list = [(name, size, qiskit.QuantumRegister(size, f'q_{name}'), reg_init_gate(values, size)) for name, size, values in code.get_reg_names_sizes_and_sets()]
        self.main_classical_registers_list = [(name, size, qiskit.ClassicalRegister(size, name), reg_init_gate(values, size)) for name, size, values in code.get_reg_names_sizes_and_sets()]
        for i in range(len(self.main_registers_list)):
            qr = self.main_registers_list[i][2]
            cr = self.main_classical_registers_list[i][2]
            self.quantum_circuit.add_register(qr)
            self.quantum_circuit.add_register(cr)

        # --- Phase qubits ---
        phase_reg = qiskit.QuantumRegister(size=1, name='phase')
        self.quantum_circuit.add_register(phase_reg)
        self.phase_qubit = phase_reg[0]

        # --- Target qubit ---
        self.target_qubit = self.get_qiskit_register(self.code.terminator.target)[-1]

        # --- Ancilla qubits ---
        self.ancilla_qubits = dict() # id-to-object mapping
        self.clean_ancillas = []

    def get_qiskit_register(self, label: str) -> qiskit.QuantumRegister:
        """Returns the respective circuit register

        :param label: Register's label (name)
        :type label: str
        :return: Qiskit register objects
        :rtype: qiskit.QuantumRegister
        """
        for name, size, reg, ig in self.main_registers_list:
            if name == label:
                return reg
        return None
```

```python

    def get_register_size(self, label: str) -> int:
        """Returns the respective register's size

        :param label: Register's label (name)
        :type label: str
        :return:
        :rtype: int
        """
        for name, size, reg, ig in self.main_registers_list:
            if name == label:
                return size
        return None

    def alloc_ancilla(self) -> qiskit.circuit.AncillaQubit:
        """Allocate an ancilla qubit.

        :return: Qubit object
        :rtype: qiskit.circuit.AncillaQubit
        """
        if len(self.clean_ancillas) > 0: #If there are available clean ancillas
            #Get an existent clean ancilla
            qubit = self.clean_ancillas[0]
            del self.clean_ancillas[0]
        else:
            reg = qiskit.QuantumRegister(1)
            qubit = reg[0]
            self.quantum_circuit.add_register(reg)
            self.ancilla_qubits[id(qubit)] = qubit
        return qubit

    def free_ancilla(self, qubit: qiskit.circuit.AncillaQubit):
        """Release a clean ancilla qubit

        :param qubit: Ancilla to release
        :type qubit: qiskit.circuit.AncillaQubit
        """
        assert id(qubit) in self.ancilla_qubits.keys(), f'The ancilla qubit {id(qubit)} does not belong to the circuit'
        assert qubit not in self.clean_ancillas, f'The ancilla qubit {id(qubit)} is not in use'
        self.clean_ancillas.append(qubit)

    def initialize_registers(self):
        """Append the initialization gates to the quantum circuit
        """
```

```
        for regdef in self.code.regdefseq:
            if isinstance(regdef, RegisterSetDefinition):
                self.quantum_circuit.append(reg_init_gate(regdef.values,
 regdef.n), self.get_qiskit_register(regdef.name))

    def revert_registers_initialization(self):
        """Append the inverce initialization gates to the quantum
circuit
        """
        for regdef in self.code.regdefseq[::-1]:
            if isinstance(regdef, RegisterSetDefinition):
                self.quantum_circuit.append(reg_init_gate(regdef.values,
 regdef.n).inverse(), self.get_qiskit_register(regdef.name))

    def build_evaluator(self):
        """Build a quantum circuit to evaluate the entire code
        """
        for regdef in self.code.regdefseq:
            if isinstance(regdef, RegisterExpressionDefinition):
                regdef.build(self)

    def revert_evaluator(self):
        """Build the inverse evaluation quantum circuit
        """
        for regdef in self.code.regdefseq[::-1]:
            if isinstance(regdef, RegisterExpressionDefinition):
                regdef.reverse(self)

    def append_measurements(self):
        """Append measurement gates
        """
        self.quantum_circuit.barrier()
        for i in range(len(self.main_registers_list)):
            qr = self.main_registers_list[i][2]
            cr = self.main_classical_registers_list[i][2]
            self.quantum_circuit.measure(qr, cr)

    def simulate(self, simulator: AerSimulator, shots=1024,
test_function: Callable[[dict], str] = None) -> pd.DataFrame:
        """Execute the circuit using a qiskit simulator.

        :param simulator: Qiskit simulator.
        :type simulator: AerSimulator
        :param shots: Number of simulation shots, defaults to 1024
        :type shots: int, optional
        :param test_function: A function that receives a dict with a
value for each register and return a status about it.  Default to
```

```
None.
        :type test_function: Callable[[dict], str], optional
        :return: DataFrame generated by the organize_qiskit_result function.
        :rtype: pd.DataFrame
        """
        result_counts = qiskit.execute(self.quantum_circuit, simulator, shots=shots).result().get_counts()
        return organize_qiskit_result(
                result_counts,
                registers_names=[reg[0] for reg in self.main_registers_list],
                test_function=test_function
            )

    def get_qubits(self) -> List[qiskit.circuit.Qubit]:
        """Returns a list with all the qubits in the circuit (except the phase qubit)

        :return: List of qubits
        :rtype: List[qiskit.circuit.Qubit]
        """
        qubits = []
        for name, size, reg, gate in self.main_registers_list:
            for qubit in reg:
                qubits.append(qubit)

        for key in self.ancilla_qubits.keys():
            qubits.append(self.ancilla_qubits[key])

        return qubits

    def build_grover_search(self, iterations: int = 0):
        """Build the Grover's quantum search algorithm

        :param iterations: number of search iterations, defaults to 0
        :type iterations: int, optional
        """
        assert iterations >= 0

        self.quantum_circuit.h(self.phase_qubit)
        self.initialize_registers()
        qubits = self.get_qubits()

        for i in range(iterations):
            self.build_evaluator()
            self.quantum_circuit.barrier()
```

```
            self.quantum_circuit.cz(self.target_qubit, self.phase_qubit)
            self.quantum_circuit.barrier()
            self.revert_evaluator()

            #Diffusion operator
            self.quantum_circuit.barrier()
            self.revert_registers_initialization()
            self.quantum_circuit.x(qubits)
            self.quantum_circuit.z(self.phase_qubit)
            self.quantum_circuit.mcx(qubits, self.phase_qubit)
            self.quantum_circuit.z(self.phase_qubit)
            self.quantum_circuit.x(qubits)
            self.initialize_registers()
            self.quantum_circuit.barrier()


        self.build_evaluator()

    def build_all(self):
        """Build the entire quantum circuit
        """
        for regdef in self.code.regdefseq:
            if isinstance(regdef, RegisterExpressionDefinition):
                regdef.pre_build(self)

        self.build_grover_search(self.code.terminator.it)
        self.append_measurements()
```

Listing B.6 – synth.py

B.7 Módulo *main.py*

```
#Filipe Chagas, 2023
from typer import Typer
from dlqpiler import synth
from dlqpiler import parser
from ply import yacc
from qiskit import Aer
import pandas as pd
from matplotlib import pyplot as plt

app = Typer()

def psim(code: str, shots: int) -> pd.DataFrame:
    qe = synth.QuantumEvaluator(yacc.parse(code))
    qe.build_all()
    return qe.simulate(Aer.get_backend('aer_simulator'), shots=shots)

```

```python
def pplot(code: str):
    qe = synth.QuantumEvaluator(yacc.parse(code))
    qe.build_all()
    qe.quantum_circuit.draw(output='mpl')
    plt.show()

def get_qqc(code: str):
    qe = synth.QuantumEvaluator(yacc.parse(code))
    qe.build_all()
    return qe.quantum_circuit

@app.command()
def sim(codefn: str, destfn: str, shots: int):
    """Execute a simulation of the given code and save it's results to a
    xlsx file

    :param codefn: Path to the code file
    :type codefn: str
    :param destfn: Path to the XLSX file
    :type destfn: str
    :param shots: Number of simulation shots
    :type shots: int
    """
    with open(codefn, 'r') as f:
        code = f.read()
        result = psim(code, shots)
        result.to_excel(destfn)

@app.command()
def plot(codefn: str):
    """Plot the quantum circuit generated for the given code

    :param codefn: Path to the code file
    :type codefn: str
    """
    with open(codefn, 'r') as f:
        code = f.read()
        pplot(code)
```

Listing B.7 – main.py