Lenguajes de Programación Cuántico

Brandon Prieto

September 2018

Índice general

1. Introducción **2**

2. Estado del Arte y Marco Teórico **4**
2.1. Lenguaje de programación 4
2.2. IBM Experiencia Cuántica 4
2.2.1. Quantum Computer Information Software Kit 5
2.3. Lenguaje de Programación Imperativo 8
2.4. Quantum Computation Language (QCL) 8
2.5. Lenguaje Q 8
2.6. Programación Cuántica 9
2.6.1. Lenguaje Ensamblador Cuántico 9
2.7. QRAM y Lenguaje Ensamblador Cuántico 11
2.8. Comparación de QCL y Q 17
2.8.1. Deutsch en QCL 17
2.8.2. Deutsch en Q 18

3. Compilador Lenguaje Cuántico Ensamblador (Compilador LQE) **19**
3.1. Compilador LQE 19
3.1.1. BNF 22
3.1.2. EBNF 23
3.1.3. JavaCC QuantumAssembly.jj 23
3.1.4. Algorimo de Deutsch LQE 24

4. Conclusiones **26**

Capítulo 1

Introducción

Para hablar de lenguajes de programación debemos remontarnos aproximadamente al año 1800 donde Charles Baggage definió la máquina analítica, cuyo diseño se basaba en el telar de Joseph Marie Jacquard, el cual usaba tarjetas perforadas para realizar diseños en el tejido. Charles Babbage adaptó su diseño para conseguir calcular funciones analíticas, con él trabajaba como colaboradora Ada Lovedby, la cual es considerada como la primera programadora de la historia, pues realizo programas para aquélla supuesta máquina de Babagge, en tarjetas perforadas, aunque como la máquina pudo construirse hasta después de un siglo. En 1943 apareció ENIAC (Electronic Numerical Integrator And Calculator), su programación se basaba en el cambio de componentes físicos, es decir, que se programaba, cambiando directamente el Hardware de la máquina, exactamente lo que sé hacia era cambiar cables de sitio para conseguir así la programación de la máquina. La entrada y salida de datos se realizaba mediante tarjetas perforadas.

Un Lenguaje de Programación es un conjunto estructurado de reglas, notaciones, símbolos y caracteres que permiten a un programador poder expresar el procesamiento de datos y sus resultados por medio del uso de computadores. Cada lenguaje posee una sintaxis propia, lo cual los hace muy diferentes entre si. Se usan para crear programas que contengan algoritmos por los cuales, el programador puede comunicarse con la computadora y así controlar las acciones a realizar por está, existen varios tipos de paradigmas de programación, por ejemplo, imperativos, funcionales, declarativos, orientado a objetos. Donde se puede encontrar una gran variedad de lenguajes mencionando algunos, Java, Python, C , C++, Pascal, Javascript, C#, PHP.

Hay lenguajes de programación de alto nivel y de bajo nivel, donde los de alto nivel permiten que se pueda escribir y leer de una forma más sencilla, ya que, son muy parecidos al lenguaje humano simplificando su entendimiento. Permiten expresarse en un nivel y estilo de escritura fácilmente legible y comprensible por otros programadores, además no dependen del tipo de máquina. Y los de bajo nivel son lenguajes totalmente dependientes de la máquina, es decir que el programa que se realiza con este tipo de lenguajes no se pueden migrar o utilizar en otras maquinas. Acá se encuentra el lenguaje de máquina el usa operaciones fundamentales para su funcionamiento de la máquina, y el lenguaje ensamblador el cual es un derivado del lenguaje máquina y esta formado por abreviaturas de letras y números llamadas mnemotécnicos. Con la aparición de este lenguaje se crearon los programas traductores para poder pasar los programas escritos en lenguaje ensamblador a lenguaje máquina.

La computación cuántica comenzó a surgir a partir del año 1980 donde se comenzó a proponer el uso de física cuántica para realizar cálculos computaciones, y David Deutsch describiera el primer computador cuántico, de aquí se empezó la idea de generar algoritmos para este mismo, en 2017 IBM presentó el primer computador cuántico, el cuál puede ser usado por medio de la nube, el cual permite crear circuitos cuánticos a través del modelo de arrastrar y soltar compuertas cuánticas, para más información sobre está vea [1] capítulo 5, o también a través de su lenguaje ensamblador OpenQASM [2], y para programadores más avanzados pueden realizar uso de uso API desarrollada en Python. En el siguiente documento se mostrará las funciones del lenguaje ensamblador cuántico, el modelo de computación QRAM, el cual usa el poder de una computadora clásica y el lenguaje ensamblador cuántico para añadir más potencia al momento de resolver problemas, y se tocarán las diferencias de dos lenguajes de programación cuánticos QCL [3] y Q [4].

Capítulo 2

Estado del Arte y Marco Teórico

2.1. Lenguaje de programación

Un Lenguaje de Programación es un conjunto estructurado de reglas, notaciones, símbolos y caracteres que permiten a un programador poder expresar el procesamiento de datos y sus resultados por medio del uso de computadores. Cada lenguaje posee una sintaxis propia, lo cual los hace muy diferentes entre si. Se usan para crear programas que contengan algoritmos por los cuales, el programador puede comunicarse con la computadora y así controlar las acciones a realizar por está.

2.2. IBM Experiencia Cuántica

IBM Q Experience es una plataforma en línea que brinda a los usuarios del público en general acceso a un conjunto de prototipos de procesadores cuánticos de IBM a través de la nube. Se puede decir que es computación cuántica basada en la nube. A partir de mayo de 2018, hay tres procesadores en la experiencia IBM Q: dos procesadores de 5 cúbits y un procesador de 16 cúbit. Este servicio se puede utilizar para ejecutar algoritmos y experimentos, y explorar tutoriales y simulaciones sobre lo que podría ser posible con la computación cuántica.

Se puede realizar construcción de circuitos cuánticos a tráves del Quantum Composer, el cuál es una herramienta de interfaz gráfica en la que puede

arrastrar y soltar diferentes operaciones para controlar cúbits. El Quantum Composer le permite desarrollar sus propios algoritmos cuánticos, que llamamos puntuaciones cuánticas.

También posee el Quantum Score, el cual es el conjunto de instrucciones, o algoritmo, para una computadora cuántica. Es una serie de puertas contra el tiempo jugadas en diferentes cúbits, muy parecido a una partitura musical. También se le llama un circuito cuántico.

Los usuarios interactúan con un procesador cuántico a través del modelo de circuito cuántico de cálculo, aplicando puertas cuánticas en los qubits utilizando una GUI llamada el compositor cuántico, escribiendo código de lenguaje de ensamblador cuántico o mediante QISKit.

2.2.1. Quantum Computer Information Software Kit

Quantum Computer Information Software Kit (QISkit) es el framework de código abierto para computación cuántica publicado por IBM bajo la licencia apache desarrollado en Python, está diseñado teniendo en cuenta la modularidad y la extensibilidad. basado en circuitos especificados a través del lenguaje intermediario OpenQASM, compila y ejecuta el SDK en varios backends por medio del procesador de 16 bits cuánticos o un simulador local y en línea ambos desarrollados también por IBM. Para el backend de QISKIT en línea, se usa como puente el API conector de a IBM Quantum Experience. QISKit permite a los desarrolladores realizar exploraciones en la Experiencia Quantum de IBM usando una interfaz Python. Esta interfaz le permite trabajar con circuitos cuánticos y ejecutar múltiples circuitos en un lote eficiente de experimentos.

QISKit está compuesto de cuatro elementos fundamentales:

- QISKit Terra
- QISKit Aqua
- QISKit Ignis
- QISKit Aer

QISKit Terra: Es la base sobre la que se basa el resto del software. Terra proporciona una base para la composición de programas cuánticos a nivel de circuitos y pulsos, para optimizarlos en base a las restricciones de un dispositivo en particular, y para administrar la ejecución de lotes de experimentos en dispositivos de acceso remoto. Terra define las interfaces para una

experiencia deseable para el usuario final, así como el manejo eficiente de las capas de optimización, programación de pulsos y comunicación backend.

QISKit Aqua: Aqua, es el elemento de la vida. Para hacer que la computación cuántica esté a la altura de sus expectativas, necesitamos encontrar aplicaciones del mundo real. Aqua es donde los algoritmos para NISQ son realizados. Estos algoritmos se pueden utilizar para crear aplicaciones para la computación cuántica. Aqua es accesible para expertos en dominios de química, optimización o inteligencia artificial, donde van a poder explorar los beneficios de usar computadoras cuánticas como aceleradores para tareas computacionales específicas, sin tener que preocuparse de cómo traducir el problema al lenguaje de las máquinas cuánticas.

Nota: Noisy Intermediate-Scale Quantum (NISQ) es un termino introducido por John Preskil donde dice, "las computadoras cuánticas puede realizar tareas que superan las capacidades de las computadoras digitales clásicas de hoy, pero el ruido en las puertas cuánticas limitará el tamaño de los circuitos cuánticos que pueden ejecutarse de manera confiable". Para más información vea [7].

QISKit Ignis: Ignis, está dedicado a combatir el ruido y los errores. Mejora la caracterización de errores, de las compuertas, y la computación en presencia de ruido. Ignis está destinado a aquellos que desean diseñar códigos de corrección de errores cuánticos, o que deseen estudiar formas de caracterizar errores a través de métodos como la tomografía, o incluso encontrar una mejor manera de usar las compuertas lógicas cuánticas mediante la exploración del desacoplamiento dinámico y el control óptimo.

QISKit Aer: Aer, impregna todos los elementos de Qiskit. Para acelerar realmente el desarrollo de las computadoras cuánticas necesitamos mejores simuladores, emuladores y depuradores. Aer ayuda a comprender los límites de los procesadores clásicos al demostrar en qué medida pueden imitar el cálculo cuántico. Además, Aer se puede usar para verificar que las computadoras cuánticas actuales y futuras funcionen correctamente. Esto se puede hacer estirando los límites de la simulación para acomodar más de 50 bits cuánticos con una profundidad razonablemente alta, y simulando los efectos del ruido realista en el cálculo.

En resumen, QISKit Terra es la base del código, para componer programas cuánticos a nivel de circuitos y pulsos, Aqua esta para algoritmos y aplicaciones de construcción, Ignis para tratar el ruido y los errores y Aer sirve

para acelerar el desarrollo a través de simuladores, emuladores y depuradores.

2.2.1.1. Open Quantum Assembly Languague

QISKit OpenQASM o "Quantum Assembly Languague.es uno de los proyectos de código abierto que componen QISKit, básicamente un lenguaje ensamblador cuántico el cual permite hacer la representación de circuitos cuánticas simples. Además dispone de un articulo públicado en enero del 2017 donde se especifica el lenguaje, y un repositorio en github. Para más información de uso de OpenQASM revise [8].

Ejemplo 2.1.1.1.1 Algoritmo de Deutsch para f(x) = x en OpenQASM para dos bits cuánticos.

```
⋮
include "qelib1.inc";
qreg q[2];
creg c[2];
x q[1];
h q[0];
h q[1];
cx q[0],q[1];
h q[0];
measure q[0] ->c[0];
⋮
```

2.2.1.2. Software Developer's Kit

El Software Developer's Kit o SDK proporciona soporte para la fase de generación de circuitos de en IBM Quantum Experience y utiliza la API de QISKit para acceder al hardware y simuladores de IBM Quantum Experience.

2.2.1.3. QISKit API

Es un wrapper de Python alrededor de la API HTTP de Quantum Experience que le permite conectarse y ejecutar el código OpenQASM. Los

desarrolladores pueden interactuar directamente con el experimento y los simuladores de Quantum Experience. La API utiliza OpenQASM, que admite un conjunto de herramientas de circuitos cuánticos, lo que abre más capacidades para el hardware cuántico subyacente en las versiones posteriores

2.3. Lenguaje de Programación Imperativo

Los lenguajes de programación imperativos pueden ser definidos como un conjunto de instrucciones, por los cuales el programador puede indicar a la computadora la manera en que este mismo desea que resuelva un problema, las instrucciones definidas se ejecutan de manera secuencial, es decir, una a una, existen varios tipos de lenguajes, por ejempo, Java, Python, C, C++.

2.4. Quantum Computation Language (QCL)

QCL fue el primer lenguaje de cuántico de programación. Creado por Bernhard Ömer, quien quería que este fuera parecido a C y Pascal. QCL contiene todas las características de un lenguaje de programación clásico tales como, variables, ciclos, condicionales, lo cual va a facilitar la transición de los lenguajes de programación clásicos a los lenguajes de programación cuánticos. También cuenta con soporte para funciones ya construidas, algunas de estas son cos, sin, print.

También cuenta con todos los operadores aritméticos conocidos. Para toda la explicación se supone que el lector ya tiene conocimientos de C, ya que tienen una sintaxis parecida y facilitará la tarea de entender el código. Para más información vea [3]

2.5. Lenguaje Q

Este lenguaje puede expresar de forma compacta los algoritmos cuánticos existentes y reducirlos a secuencias de operaciones elementales; También se presta fácilmente a sistemas automáticos, independientes de hardware, simplificación del circuito. Para más información vea [4]

2.6. Programación Cuántica

Programar un computador significa hacer que la computadora lleve acabo ciertas acciones en un lenguaje específico, ya sea directamente o través de un interprete intermediario. La manera básica de como manejar datos esta representada de la siguiente manera, con una entrada de datos y un control.

```
DATOS + CONTROL = PROGRAMACIÓN
```

Control significa que el programa está construido por un conjunto de instrucciones básicas y un conjunto de estructuras tales como condicionales, ciclos, saltos. Todo esto se transpone al mundo cuántico, primero se debe suponer que la computadora consiste de un dispositivo cuántico, con una entrada de datos cuántica, la cual es representada por un conjunto direccionable de cúbits, junto a un conjunto de operaciones ya construido que permiten la manipulación de datos, operaciones que son de dos tipos:

1. Unitarias las cuales evolucionaran los datos cuánticos.
2. Medición las cuales inspeccionaran el valor de los datos.

También se asumirá que se pueden ensamblar más operaciones fuera de las básicas. Esto se representará de la siguiente manera:

```
DATOS CUÁNTICOS + CONTROL = PROGRAMACIÓN CUÁNTICA
```

Sabiendo esto ahora nos podemos enfocar en abordar uno de los temas más importarte para los lenguajes de programación cuánticos, el lenguaje emsablador cuántico, en donde la programación cuántica se realizará a bajo nivel.

2.6.1. Lenguaje Ensamblador Cuántico

Hoy un día existen múltiples lenguajes de programación para las computadoras clásicas, donde los desarrolladores no tienen preocupación alguna sobre la arquitectura sobre la máquina en la que están trabajando, aquí el compilador se ocupa de todo el trabajo, pero alguien tuvo que preocuparse de esto en algún momento por como construir estos interpretes, hace unas décadas el lenguaje ensamblador era el único lenguaje existente. Ahora no nos preocuparemos por esto ya que se espera que un programador en cuántica

tenga una gran experiencia, así como el programador de hoy en día no tiene un amplio conocimiento sobre hardware.

Para especificar los algoritmos cuánticos existentes se necesita al menos un ensamblador cuántico. Aunque no se necesita conocer específicamente el hardware cuántico por el cual la computadora esta compuesta, pero si se necesita saber cual es su tipo de arquitectura.

Existen tres tipos diferentes de modelos de computación cuántica que pueden ser equivalentes entre si.

- Modelo de circuitos cuánticos.
- Máquina de Turing cuántica.
- Modelo de memoria cuántica de acceso aleatorio (Modelo QRAM).

Estos modelos serán explicados a continuación en el orden en el que han sido nombrados anteriormente. Empezando por el modelo de circuitos cuánticos, el cuál es un circuito que opera con bits cuánticos haciendo uso de compuertas lógicas cuánticas, las cuales son representadas mediante matrices unitarias. Este modelo requiere tres elementos que son necesario para su funcionamiento.

- Un dispositivo de entrada el cual recibirá datos cuánticos.
- Un conjunto básico de compuertas, que pueden usarse paralelamente y secuencialmente.
- Un dispositivo el cual permite realizar mediciones.

Normalmente en el mundo clásico el programador revisa ver las variables en cualquier lugar durante la computación, pero en la parte cuántica las mediciones se realizan al final por lo cual puede resultar extraño.

El segundo modelo es la máquina de Turing cuántica, este es una analogía de la máquina de Turing clásica, este modelo es ideal para tratar problemas de complejidad de clases, pero no cuando se trata de diseño de algoritmos o de lenguajes de programación. Este modelo consta de tres elementos.

- Una cinta de memoria infinita en donde cada elemento es un cúbit.
- Un procesador finito.
- Un cabezal.

Cada conjunto de instrucciones se aplica sobre el elemento de la cinta señalado por el cabezal. El resultado va a depender del cúbit en la cinta y el estado en cual este el procesador en el momento de medición.

El tercer modelo es el más conveniente para nosotros, es conocido como el modelo QRAM (Modelo de memoria cuántica de acceso aleatorio), este modelo esta compuesto de dos partes.

- Un computador clásico que juega el rol de maestro.
- Un dispositivo cuántico, que pueda ser accedido por el maestro en el momento que lo requiera.

2.7. QRAM y Lenguaje Ensamblador Cuántico

El objetivo del modelo QRAM, es que cualquier programador pueda escribir código clásico en cualquier lenguaje, por ejemplo, Python, y cuando requiere de poder adicional de la máquina para lograr resolver un problema, sea capaz de añadir algunas lineas de código ensamblador cuántico. El ensamblador cuántico va a ser el puente para poder acceder y hacer uso de nuestro dispositivo cuántico. Por cierto, como ya pudo haber notado el programador no requiere tener conocimiento alguno sobre la composición interna del dispositivo cuántico, y además, tampoco necesita saber la manera de como almacenan, inicializan o se miden los bits cuánticos. Solo debería preocuparse por cosas como la capacidad de almacenamiento, el tamaño de la memoria cuántica. Todo lo demás va a pasar a través de la **Interfaz de Hardware Cuántico**, la cuál va a encargarse de traducir los comandos de ensamblador cuántico introducidos en acciones explicitas que va realizar el dispositivo 2 cuántico.

Resumiendo el modelo QRAM puede explotar recursos cuánticos y, al mismo tiempo, pueden ser utilizados. Para realizar cualquier tipo de computación clásica. Nos permite controlar las operaciones realizadas en registros cuánticos. y proporciona el conjunto de instrucciones para su definición.

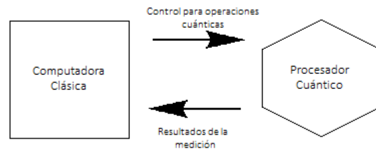

Figura 2.1: Representación básica del modelo QRAM.

Observado la Figura 2.1 Tenemos dos dispositivos el primero es una computadora clásica, dentro de la computadora maestra, el registro de control se utilizará para almacenar la instrucción en lenguaje ensamblador cuántico, note que las propias instrucciones son codificables como secuencias de bits. Cuando el puntero de control se encuentra en una u otra de las instrucciones cuánticas, el maestro utilizará la interfaz de hardware cuántica para enviarla al dispositivo cuántico. En el segundo dispositivo hay dos cosas: un conjunto de registros de almacenamiento de datos cuánticos y las utilidades que aplican operaciones en el almacenamiento. Lo primero que hará el programador es pedirle al dispositivo cuántico a través de la interfaz de hardware cuántica que inicie una secuencia direccionable de bits cuánticos. Estas transacciones se realizan a través de una interfaz conocida como el registro cuántico o registro-q.

Ya que la anterior explicación puede no haber sido lo suficientemente clara vamos a definir un **registro cuántico**. Un registro cuántico es una interfaz para una secuencia direccionable de bits cuánticos. Cada registro q tiene un identificador único por el cual es referido.

En el teorema de no clonación: El teorema de no clonación evitará que el ensamblador cuántico tenga instrucciones de copia. No hay forma de copiar el contenido de un registro a otro, una operación familiar y generalizada en los ensambladores ordinarios.

Por ahora vamos a pensar en un registro cuántico como un arreglo adyacente de cúbits. Tampoco es necesario preocuparse por donde o como se almacenan los bits cuánticos. Después de que un registro cuántico ha sido inicializado y manipulado por el programador, esté puede emitir un comando que medirá las partes seleccionadas del mismo. El dispositivo cuántico va a realizar la medición solicitada y devolverá un valor clásico que se puede mostrar y almacenar en algún lugar del programa principal. En este modelo, la medición se intercala con otros comandos: el programador puede solicitar en cualquier momento después de la inicialización el valor de una sección

arbitraria del registro. Todo esto puede verse en la Figura 2.2

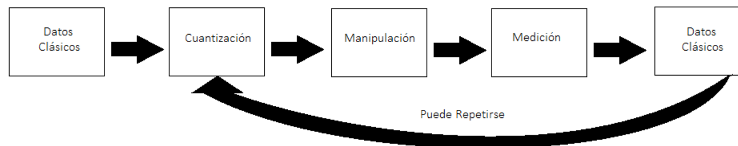

Figura 2.2: Control Cuántico.

Ahora vamos usar el lenguaje básico basado en lenguaje ensamblador propuesto por [1] el cual no es un estándar, pero nos va a permitir mostrar de manera sencilla como son las instrucciones del lenguaje ensamblador, dado que un lenguaje ensamblador que tenga propósitos de uso en la vida real puede ser más complejo revise [13].

Los identificadores de los registros cuánticos se van a nombrar *R1*,*R2*,...,R_n. También supondremos, para una mejor comodidad, que todos los registros son de tamaño 8, es decir, pueden almacenar el análogo cuántico de un byte, un **qubyte**. El prefijo R representará el código de un registro q no especificado. Dicho lo anterior comencemos a enumerar las instrucciones del lenguaje ensamblador propuesto. La arquitectura QRAM permite que el programa de llamada transfiera cada instrucción individual al procesador cuántico de una en una. Ya hablamos de como identificar los registros cuánticos, pero ahora necesitamos una forma de inicializarlos. Más específicamente, necesitamos pasar una matriz de bits del programa principal a un registro q dado, y pedirle al dispositivo cuántico que lo inicie.

- Inicializar el registro R
  `INITIALIZE R [INPUT]`

El `[INPUT]` opcional es un arreglo clásico de bits, cuyo tamaño coincide con el tamaño de la registro, que como habíamos dicho antes es de un byte. Además si no se especifica, se va supone que se rellena automáticamente con ceros.

Ejemplo 2.6.1 El siguiente ejemplo inicializa un registro R1 de ocho bits cuánticos, el cual que daría inicializado así [00000000], pero entonces se

vuelve a inicializar usando como entrada el arreglo de bits $G = [11011011]$.

```
⋮
var G = [11011011]
⋮
INITIALIZE R1
INITIALIZE R1 B
⋮
```

Como se suposición se tendrá que la inicialización envía todos los bits cuánticos a su estado fundamental $|0\rangle$. Es decir, si inicializamos un registro cuántico de tamaño 5, el estado conjunto es $|00000\rangle$. Si, por otro lado, proporcionamos un `INPUT` tal como [00101], el sistema se encargará de inicializar nuestro registro a $|00101\rangle$.

Una vez que tenemos un registro inicializado, podemos acceder a sus cúbits individuales para la manipulación. Por ejemplo, R [0] indicará su primer cúbit, y así sucesivamente. Ahora se expandirá el ensamblador con la capacidad de seleccionar una variable de subregistro, se denotará con la letra s del prefijo S.

- Seleccionar del subregistro R compuesto por un número qubits NUMQUBIT a partir de R[OFFSET]. Almacenando la dirección en la variable S.
  `SELECT S R OFFSET NUMQUBIT`

Ejemplo 2.6.2 Se inicializa un registro cuántico R1, y se extrae un subregistro formado por los índices 2 y 3 representados por la variable S.

```
⋮
INITIALIZE R1
SELECT S R1 2 3
⋮
```

Ejemplo 2.6.3 Se inicializa un registro cuántico R1, y se extrae un subregistro formado por los índices 2 y 4 representados por la variable S1, y luego se extraen los índices 0 y 2 representados por la variable S2.

```
⋮
INITIALIZE R1 [01110001]
SELECT S1 R1 2 4
SELECT S2 S1 0 2
```

⋮

¿Cuáles cúbits de R1 seleccionamos en S2?

Para responder está pregunta revisemos las instrucciones paso a paso. Primero inicializamos el registro R1 con [01110001]. Luego seleccionamos los índices 2 y 4, es decir, los cúbits 1 y 0, respectivamente, para que se entienda mejor ya que nuestro arreglo está compuesto de 0's y 1's seleccionamos [00**11**0**0**001], ahora tenemos en S1 el registro cuántico [**10**000000], para esto estamos suponiendo que los demás números del registro se rellenan con 0's. Ahora almacenamos en S2 los índices 0 y 2, es decir, 1 y 0, [**1**0**0**0000], note que al final solo permanecerá 1 del registro R1 ya que los 0's son diferentes, así que no se confunda.

Ahora volvemos a las compuertas cuánticas las cuales ya se habían nombrado anteriormente.

$$\textbf{GATES} = [G_0, G_1, \ldots, G_n]$$

Se pueden realizar diferentes elecciones de compuertas cuánticas básicas, siempre que el conjunto **GATES** sea un conjunto universal de compuertas, es decir, genere todas las transformaciones unitarias de dimensión finita mediante aplicaciones sucesivas de composición y producto tensor. En la práctica. Algunos ejemplos de compuertas cuánticas que se podrían usar del conjunto.

$$\textbf{GATES} = [H, R_\theta, I_n, CNOT]$$

donde H, R_θ, I_n y $CNOT$ denotan el Hadamard, el cambio de fase en un ángulo θ,la matriz de identidad $n \times n$, y la compuerta NOT controlado, respectivamente.

- La instrucción básica para aplicar una compuerta se ve de la siguiente forma.
  `APPLY U R`

Donde U es la compuerta a aplicar y R el registro cuántico.

Ejemplo 2.6.4 Se inicializa un registro cuántico R1, y se aplica la compuerta de Hadamard.

⋮

```
INITIALIZE R1
APPLY H R1
⋮
```

También tenemos instrucciones para realizar concatenación, producto tensor e inversa.

- La operación de composición, que se ejecuta secuencialmente de derecha a izquierda dos transformaciones unitarias U_1 y U_2, y guarda el resultado en una variable U.
  `U CONCAT U1 U2`

- U es el resultado de realizar producto tensor de U_1 y U_2.
  `U TENSOR U1 U2`

- El inverso: U es el resultado de tomar el inverso de U1, es decir, la transformación que "deshace"U_1.
  `U INVERSE U1`

Hasta aquí hemos hablado de varias instrucciones pero como se supone que vamos obtener nuestra respuesta, para esto vamos a usar la instrucción de medición.

- Medición del registro R1 y guardado del resultado en la variable clásica RES.
  `MEASURE R RE`

Hasta ahora, no se ha hablado de estructuras de control, como los saltos, condicionales. La razón es que son prescindibles. Si el programador desea implementar un if - then - else, entonces podría emitir una declaración de medición, recuperar una matriz de bits y usar una estructura condicional clásica (if, while, case). Y podría escribir un comando así por ejemplo:

```
IF(RES==[01]) THEN APPLY H R ELSE TENSOR H R
WHILE(ANS<=[01]) APPLY H R
```

El lenguaje propuesto hasta el momento, está compuesto por nada más que cadenas binarias, pero es de utilidad para expresar algunos algoritmos

cuánticos.

Ejemplo 2.6.5 Algoritmo de Deutsch para f(x) = x.

```
⋮
INITIALIZE R1 [01000000]
SELECT S1 R1 0
SELECT S2 R1 1
TENSOR S S1 S2
APPLY H S
APPLY CNOT S
APPLY H S
MEASURE S ANS
⋮
```

2.8. Comparación de QCL y Q

Para realizar la comparación de ambos lenguajes se va a mostrar la implementación del algoritmo de Deutsch para f(x) = x, el cual es sencillo pero nos va a permitir ver la esencia de ambos lenguajes. Para más información sobre el algoritmo de Deutsch ver [1] capítulo 6.

2.8.1. Deutsch en QCL

```
operator Uf(qureg x,qureg y){
   //Para f(x) no hay necesidad de un uf
}

procedure deutsch() {
   qureg registro1[1];
   qureg registro2[1];
   int m;
   CNot(registro2);
   H(registro1);
   H(registro2);
   Uf(registro1,registro2);
   H(registro1);
```

```
   measure registro1,m;
}
```

2.8.2. Deutsch en Q

```
Qop Uf(qureg x,qureg y){
   //Para f(x) no hay necesidad de un uf
}

int Deutsch() {
   Qreg registro1(1,0);
   Qreg registro2(1,1);
   Qop h1 = QHadamard(registro1);
   Qop h2 = QHadamard(registro2);
   Uf(registro1, registro2);
   h1.QHadarmard();
   int m = h1.measure();
   return m;
}
```

En QCL las operaciones unitarias son funciones (qufunct u operator) y en Q son objetos, por lo que en QCL su manipulación está sujeta a la sintaxis de la función llamada; por lo tanto, la simplificación son más complicadas de implementar, en Q esto se facilita por el mismo hecho de que son objetos. Aunque en ambos lenguajes los registros cuánticos se manejan de manera parecida, en QCL además del qureg, existen otros dos tipos de registro, quvoid y quscratch, que, para una correcta verificación de tipos, requieren el conocimiento del estado del dispositivo cuántico. Su sintaxis puede ser diferir ya que QCL esta basado en C y Q en C++. Y por ultimo Q está más enfocado en el paradigma orientado a objetos, por lo cual al hacer el uso de compuertas cuánticas las funciones propuestas van a ser diferentes.

Capítulo 3

Compilador Lenguaje Cuántico Ensamblador (Compilador LQE)

3.1. Compilador LQE

Este compilador ha sido construido usando javacc y está basado en el lenguaje propuesto en [1], haciendo uso de la clase QuantumAssembly.jj la cual contiene los tokens básicos del ensamblador. Si se realizan cambios a este archivo se deberan ejecutar los siguientes comandos.

```
$javacc QuantumAssembly.jj
$javac *.java
$java QuantumAssembly 〈 codigoEnsamblador.txt
```

No se debe preocupara por la primera y la segunda linea, estas solo generar error, si no se maneja la sintaxis de javacc y de java. La tercera linea generara el código apartir del lenguaje ensamblador del archivo txt en java creado un nuevo archivo Main.java

- `INITIALIZE registerName [qubitNumber]`
- `SELECT varName registerName indexMin:indexMax`
- `TENSOR tensorAnswer register1 register2`
- `APPLY GATE registerName`

- `INVERSE varAnswerName registerName`
- `CONCAT varAnswerName register1 register2`
- `MEASURE registerName varAnswer`

Se tienen matrices predeterminadas de Hadarmard, Not Controlado, Identidad.

- Hada -⟩ Hadamard
- Iden -⟩ Identidad
- Cnot -⟩ Not Controlado

3.1.0.1. Initialize

Permite inicializar un registro de bits cuánticos de la siguiente manera, donde el usuario deberá tomar cada el registro como un arreglo para mayor comodidad, por debajo se crearán ArrayList¡ComplexNumber[]¿para almacenar los cúbits.

```
INITIALIZE myQubitRegister [0010101]
```

Donde cada número en el arreglos representa un bit cuántico y estado, por tal si solo requiere usar un cúbit, por ejemplo, debería hacer esto.

```
INITIALIZE myQubitRegister [1]
```

3.1.0.2. Select

Permite inicializar un registro tomando los bits cuánticos de un registro creado con anterioridad, seleccionando los índices en los que desea sacar el subregistro, tenga en cuenta que primero va el índice menor y luego el mayor, y van separados por ":"para mayor comodidad y no crear confunciones, por debajo se crearán ArrayList⟨ComplexNumber[]⟩ para almacenar los cúbits.

```
INITIALIZE myQubitRegister [00001111]
SELECT mySelectVar myQubitRegister 0:3
```

Al realizar esto la variable myQubitRegister que quedaría igual, es decir, con un registro [00001111], la variable mySelectVar quedaría con [0000]

3.1.0.3. Apply

Permite aplicar las siguientes compuertas H (Hadamard),CNOT (Not Controllado),NOT (Negación),I (Identity) a un registro de cúbits, tenga en cuenta que la compuerta se aplicará a cada cúbit del registro, si requiere aplicar a un solo cúbit deberá usar SELECT, aplicar la compuerta con APPLY, y luego CONCAT para volverla a concatenar al registro, el simulador multiplicara el cúbit por la matriz correspondiente a la compuerta para así aplicara.

```
INITIALIZE myQubitRegister [10001111]
APPLY H myQubitRegister
```

3.1.0.4. Inverse

Almacena la inversa de la variable requerida por el usuario, se hace aplicando la operacion inverse definida el simulador,

```
INVERSE myInverse Q
```

Guardando la inversa de la Q en la variable myInverse, tenga en cuenta que la inversa se aplicara a todos los bits cuánticos dentro del registro.

3.1.0.5. Tensor

Guarda el resultado de realizar producto tensor, entre 2 registros, o matrices especiales, Hada (Hadamard), Iden (Identity), Cnot (Controlled Not). Se realiza producto tensor de los cada cúbit de los registros en orden. Recuerde que se realiza producto tensor entre matrices o entre registros que serían arreglos, por lo cual deberian ser del mismo tamaño.

```
TENSOR myTensor R1 R2
```

Almacena en myTensor un nuevo registro el resultado de realizar producto tensor de R1 y R2.

3.1.0.6. Measure

Almacena el resultado de realizar una medición de un determinado registro en una nueva variable.

MEASURE Q answer

Almacena en la varible answer el resultado de realizar la medición de Q.

3.1.1. BNF

Notación en Backus-Naur Form usada para definir la sintaxis del lenguaje ensamblador. <start>::= <reg> <eol> | <reg> <expr> <eol>

<expr>::= <reg>|<sel>|<apply>|<tensor>|<concat>|<inv>|<measure>

<reg>::= ‘initialize’ <identifier>
| ‘initialize’ <identifier> <size>
| ‘initialize’ <identifier> <expr>
| ‘initialize’ <identifier><size> <expr>

<sel>::= ‘select’ <identifier> <identifier> <index>‘:’<index> |
‘select’ <identifier> <identifier> <index>‘:’<index>
<expr>

<apply>::= ‘apply’ <gate> <identifier>
| ‘apply’ <gate> <identifier> <expr>

<tensor>::= ‘tensor’ <identifier> <identifier> <identifier>
| ‘tensor’ <identifier> <identifier> <identifier> <expr>

<concat>::= ‘concat’ <identifier> <identifier> <identifier>
| ‘concat’ <identifier> <identifier> <identifier> <expr>

<inv>::= ‘inverse’ <identifier> <identifier>
| ‘inverse’ <identifier> <identifier> <expr>

<measure>::= ‘measure’ <identifier> <identifier>
| ‘measure’ <identifier> <identifier><expr>

<gate>::= ‘h’ | ‘phase’ | ‘identity’ | ‘cnot’ | ‘not’

<identifier>::= | <letter> <identifier> | <letter> <digit> <identifier>

<size>::= ‘[’ <qubit>‘]’ | ‘[’ <qubit> <multisize>‘]’

<multisize>::= <qubit> | <qubit> <multisize>

<qubit>::= ‘0’ | ‘1’

<letter>::= ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’ | ‘H’ | ‘I’ | ‘J’
| ‘K’ | ‘L’ | ‘M’ | ‘N’ | ‘O’ | ‘P’ | ‘Q’ | ‘R’ | ‘S’ | ‘T’
| ‘U’ | ‘V’ | ‘W’ | ‘X’ | ‘Y’ | ‘Z’ | ‘a’ | ‘b’ | ‘c’ | ‘d’
| ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i’ | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’
| ‘o’ | ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘v’ | ‘w’ | ‘x’
| ‘y’ | ‘z’

```
<digit>::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<index>::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
```

3.1.2. EBNF

Notación en Extended Backus-Naur Form usada para definir la sintaxis del lenguaje ensamblador.

```
start = reg [expr] EOL
expr = reg | sel | apply | tensor | concat | inv | measure
reg = 'initialize' identifier [size] [expr]
sel = 'select' identifier identifier index':'index [expr]
apply = 'apply' gate identifier [expr]
tensor= 'tensor' identifier identifier identifier [expr]
concat= 'concat' identifier identifier identifier [expr]
inv = 'inverse' identifier identifier [expr]
measure = 'measure' identifier identifier [expr]
gate = 'h' | 'phase' | 'identity' | 'cnot' | 'not'
identifier = letter {letter} {digit}
size = '[' qubit {qubit} ']'
qubit = '0' | '1'
letter = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K'
| 'L'          | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V'
| 'W' | 'X'          | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g'
| 'h' | 'i' | 'j'          | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r'
| 's' | 't'| 'u' | 'v'          | 'w' | 'x' | 'y' | 'z'
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
index = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
```

3.1.3. JavaCC QuantumAssembly.jj

Esta es la clase que se usar para cambiar los tokens, añadir nuevas compuertas, además la cuál genera el código en java aplicado al simulador cuántico. El código escrito en formato java se encarga de crear un archivo Main.java a partir de la entrada con formato txt. Y en javacc se definen los tokens con los comandos especiales, compuertas, tamaño de los registros, caracteres que pueden llevar como nombre las variables teniendo en cuenta que sea un carácter o una cadena. Para modificaciones hay que tener en cuenta la estructura del EBNF para así no dañar la estructura principal del lenguaje.

La representación en javacc es exactamente la misma propuesta en ebnf, por supuesto teniendo en cuenta que la sintaxis de ambas partes es diferente.

3.1.4. Algorimo de Deutsch LQE

En esta implematación del algoritmo de Deutsch en LQE para f(x) = x, se crean ambos bits cuánticos, se entrelazan realizando producto tensor, se crea la matriz de hadamard del tamaño para poderlos poner en superposición, se hace tensor de hadamard identidad para que la matriz alcance el tamaño indicando, luego se realiza tensor de hadamard y los qubits entrelazados para ponerlos en estado de superposición, se aplica CNOT a los qubits en superposición en este caso este seria nuestra matriz uf por lo cual es menos cosos usar CNOT y no crearla manualmente, y tensor de de este y la identidad, y luego medimos.

```
INITIALIZE R1 [0]
INITIALIZE R2 [1]
TENSOR QUBIT R2 R1
TENSOR Thada Hada Hada
TENSOR Tiden Hada Iden
TENSOR R0 Thada QUBIT
TENSOR RE1 Cnot R0
TENSOR RE2 Tiden RE1
MEASURE RE2 ANS
```

Código gerado en java

```java
1 ArrayList<?> hadamardMatrix = QuantumCompilerMethods.
    createHadamardMatrix();
2   ArrayList<?> controlledNotMatrix = QuantumCompilerMethods.
      createControlledNotMatrix();
3   ArrayList<?> identityMatrix = QuantumCompilerMethods.
      createIdentityMatrix();
4   ArrayList<ComplexNumber[]> R1 = new ArrayList();
5   R1 = QuantumCompilerMethods.createQubitRegister("0");
6   ArrayList<ComplexNumber[]> R2 = new ArrayList();
7   R2 = QuantumCompilerMethods.createQubitRegister("1");
8   ArrayList<?> QUBIT = QuantumCompilerMethods.tensorProduct(
      R2,R1);
9   ArrayList<ComplexNumber[]> applyR2H =
      QuantumCompilerMethods.quantumApply((ComplexNumber[][])
```

```
        hadamardMatrix.get(0),R2);
10    ArrayList<?> Thada = QuantumCompilerMethods.tensorProduct((
        ComplexNumber[][]) hadamardMatrix.get(0),(ComplexNumber
        [][]) hadamardMatrix.get(0));
11    ArrayList<?> Tiden = QuantumCompilerMethods.tensorProduct((
        ComplexNumber[][]) hadamardMatrix.get(0),(ComplexNumber
        [][]) identityMatrix.get(0));
12    ArrayList<?> R0 = QuantumCompilerMethods.tensorProduct(
        Thada,QUBIT);
13    ArrayList<?> RE1 = QuantumCompilerMethods.tensorProduct(
        controlledNotMatrix,R0);
14    ArrayList<?> RE2 = QuantumCompilerMethods.tensorProduct(
        Tiden,RE1);
15    double ANS = QuantumCompilerMethods.measure(RE2);
```

Ya que un registro de cúbits se ve como un arreglo en este caso, se manejara usando ArrayList, el tipo es ? debido a que al realizar producto tensor se puede realizar entre 2 vectores o entre 2 matrices, usando instanceof y por dentro se identifica cual es el tipo de ambas entradas para realizar el producto tensor correspondiente, ya que el código varia entre ambos, para aplicar las compuertas se multiplican las matrices por dentro, al crear el codigo se inicializan automaticamente las matrices de hadamard, identidad y not controlado.

Capítulo 4

Conclusiones

Aún falta bastante tiempo para poder ver un lenguaje de programación imperativo cuántico, debido a la falta de disponibilidad de computadores cuánticos, los avances de implementación de lenguajes ensambladores cuánticos son muy grandes, y ya se puede usar al menos uno con un computador cuántico real, el QASM.

Hay bastantes simuladores de computadores cuánticos, pero nunca serán tan eficiente como uno real, debido a la inmensa capacidad de recursos requerida para la creación bits cuánticos.

Se ha implementado el compilador básico usando javacc, haciendo el análisis léxico, y creando una clase la cual puede ser usado por el simulador cuántico construido en java, creando una clase Main que ejecutara los métodos ya definidos en este, solo con que el usuario sea capaz de entender y usar el LQE básico.

Bibliografía

[1] Mirco A Mannucci Noson S. Yanofsky. *Quantum Computing for Computer Scientists*. Cambridge University Press, 2008.

[2] IBM Developer. Qiskit openqasm, November 2018. URL `https://developer.ibm.com/code/open/projects/qiskit/qiskit-openqasm`.

[3] Bernhard Ömer. Quantum programming in qcl, May 2003. URL `http://tph.tuwien.ac.at/~oemer/doc/quprog.pdf`.

[4] T. Calarco S. Bettelli, L. Serafini. Toward an architecture for quantum programming, March 2001. URL `arXiv:cs/0103009v3`.

[5] Paola Camacho. Historia y definición de los lenguajes de programación. URL `https://www.monografias.com/trabajos99/historia-y-definicion-lenguajes-programacion/historia-y-definicion-lenguajes-programacion.shtml`.

[6] Sara Alvarez. Tipos de lenguajes de programación, February 2006. URL `https://desarrolloweb.com/articulos/2358.php`.

[7] John Preskill. Quantum computing in the nisq era and beyond, July 2018. URL `arXiv:1801.00862v3`.

[8] John A. Smolin Jay M. Gambetta Andrew W. Cross, Lev S. Bishop. Open quantum assembly language, July 2017. URL `https://arxiv.org/abs/1707.03429`.

[9] Jay M. Gambetta Ali Javadi-Abhari. Qiskit and its fundamental elements, July 2018. URL `https://medium.com/qiskit/qiskit-and-its-fundamental-elements-bcd7ead80492`.

[10] IBM Research and the IBM QX team. Ibm experience documentation, 2017. URL https://quantumexperience.ng.bluemix.net/qx/tutorial?sectionId=beginners-guide&page=introduction.

[11] Bernhard Ömer. Classical concepts in quantum programming, April 2003. URL https://arxiv.org/pdf/quant-ph/0211100.pdf.

[12] IBM Research and the IBM QX team. Ibm experience documentation, 2017. URL https://quantumexperience.ng.bluemix.net/qx/tutorial?sectionId=full-user-guide&page=introduction.

[13] David William Rajagopal Nagarajan, Nikolaos Papanikolaou. Simulating and Compiling Code for the Sequential Quantum Random Access Machine. 170:101–124, 2007. doi: https://doi.org/10.1016/j.entcs.2006.12.014.

[14] JavaCC Oracle. Javacc documentation, October 1996. URL https://javacc.org/doc.