Portada

Índice

[1. Introducción 3](#_Toc341885542)

[2. Antecedentes 4](#_Toc341885543)

[2.1 Estado de la cuestión 4](#_Toc341885544)

[2.2 Conocimientos necesarios 4](#_Toc341885545)

[2.2.1 Física e Información cuántica 4](#_Toc341885546)

[2.2.2 La arquitectura MIPS 12](#_Toc341885547)

[3. Gestión del proyecto 13](#_Toc341885548)

[4. Objetivo 14](#_Toc341885549)

[5. Desarrollo del proyecto 15](#_Toc341885550)

[5.1 Simulando los sistemas cuánticos 15](#_Toc341885551)

[5.2 Simulando hardware en Java 20](#_Toc341885552)

[5.3 Diseñando el procesador qMIPS 26](#_Toc341885553)

[6. Conclusiones 27](#_Toc341885554)

[7. Futuras líneas 28](#_Toc341885555)

[8. Bibliografía 29](#_Toc341885556)

[9. Anexo A: Código fuente 30](#_Toc341885557)

# Introducción

La miniaturización de los componentes que conforman los dispositivos electrónicos actuales se acerca cada vez más a un límite físico infranqueable: llegará un momento en el que los componentes serán tan pequeños que los efectos de la física cuántica serán más relevantes que los de la física clásica y nos será imposible manejar la información tal y como lo hacemos hasta ahora.

El efecto túnel, por ejemplo, permitiría a una corriente de electrones saltar de un conductor a otro aun estando separados por una barrera clásicamente infranqueable, lo que haría muy complicado manejarla.

Este límite nos obliga a buscar otras vías de avanzar en la implementación de computadores cada vez más rápidos. La idea básica de la computación cuántica es, en vez de ver la física cuántica como un límite, utilizar sus, a menudo contraintuitivas propiedades, para construir máquinas más potentes que las actuales.

Efectos como la superposición de estados cuánticos, que permite a una partícula estar en varios estados al mismo tiempo; o el entrelazamiento cuántico, que liga las partículas por muy separadas que estén, permite realizar computaciones y comunicar información de forma exponencialmente más rápida y totalmente segura.

Por supuesto todo esto dista de ser sencillo. Los estados cuánticos son extraordinariamente frágiles, tienden a interaccionar con su entorno de forma que se pierden sus características cuánticas para pasar a ser estados clásicos cuyas propiedades nos dejan de ser útiles. Por esto no observamos en el día a día los efectos microscópicos de las partículas, aun siendo todo un gran conjunto de ellas.

A día de hoy, la computación cuántica está en un estado muy temprano de su desarrollo, ya que se trata de un paradigma de la computación muy joven.

Las primeras ideas de utilizar las propiedades de la física cuántica para realizar computaciones que superasen a las clásicas las planteó Richard Feynman en 1982, al observar lo difícil que parecía para los ordenadores de su época simular sistemas cuánticos. Feynman postuló que un ordenador que utilizara las leyes de la física cuántica para funcionar, sería capaz de simular eficientemente sistemas cuánticos. Este postulado está afirmando, al fin y al cabo, que un ordenador tal y como los conocemos hoy en día sería incapaz de realizar eficientemente ciertas tareas que un computador cuántico realizaría con facilidad.

En los años 80, David Deutsch, desarrolló el marco matemático actual de la computación cuántica, expuso el concepto de un computador cuántico universal y demostró con un sencillo ejemplo que podría realizar ciertas tareas más rápidamente que cualquier ordenador clásico, utilizando el algoritmo que lleva su nombre. Este algoritmo se explicará en detalla en las secciones siguientes.

En aquel momento la computación cuántica era un paradigma puramente científico, lejos de tener una aplicación práctica más allá de la simulación de otros sistemas cuánticos. Esta idea cambió radicalmente cuando Peter Shor propuso su algoritmo de descomposición de números primos. Apoyándose en el algoritmo de Deutsch-Josza, que permite obtener el periodo de una función, propuso un algoritmo para descomponer en factores primos, que supondría una mejora de orden exponencial sobre el mejor algoritmo clásico.

Dado que el extendido sistema de criptografía RSA se apoya en la dificultad que supone descomponer un número muy grande en sus factores primos, este descubrimiento supone, si se desarrollara un computador cuántico totalmente funcional, la caída de este sistema de criptografía.

Posteriormente se han ido descubriendo muchas otras aplicaciones de la información cuántica, hasta el punto que se ha dividido en varias ramas de desarrollo:

* Computación cuántica: el desarrollo de computaciones que superen de alguna forma a sus contrapartidas clásicas. Tanto el desarrollo teórico de *algoritmos cuánticos*, como la implementación física de computadores cuánticos propiamente dichos. Es en este punto en el que se centrará el proyecto. Hasta la fecha, el número de *bits cuánticos* que somos capaces de controlar en laboratorio es del orden de diez, muy lejos de lo necesario para superar a los sistemas actuales.
* Criptografía cuántica: el desarrollo de sistemas que permitan el envío de información de forma totalmente segura utilizando las propiedades de la física cuántica. El ejemplo clave de este tipo de algoritmos es el protocolo BB84. Dispositivos físicos de este tipo ya se comercializan, aunque tienen muchas limitaciones.
* Teleportación cuántica: el envío de información de forma más rápida haciendo uso del entrelazamiento cuántico. Teóricamente, se puede enviar la cantidad infinita de información contenida en el estado de una partícula de forma instantánea entre dos puntos del espacio, da igual lo distantes que estén. Este punto lleva a confusión en muchos casos: primero, es información lo que se envía no materia; segundo, requiere el envío de dos bits clásicos, luego la información no viaja en ningún caso más rápido que la luz. En el momento de escribir esto, el record de distancia es de 143km entre la Palma y Tenerife [http://www.agenciasinc.es/Noticias/Record-mundial-de-teleportacion-cuantica-en-Canarias].

Aunque no exista aun la tecnología para desarrollar un computador cuántico que supere a los sistemas de información actuales, el aspecto matemático de la computación cuántica ha sido desarrollado en detalle. Se tiene constancia de que, si bien un computador cuántico podría realizar cualquier operación que realizara uno clásico, es más lógico que un ordenador actual controle a la máquina cuántica. La gran ventaja es que podemos hacer un sistema de computación cuántica mucho más específico, lo justo para que la máquina conjunta sea totalmente universal en el sentido descrito por Deutsch, facilitando en gran medida su construcción.

Este es el punto fundamental de este proyecto: integrar en una arquitectura real de un procesador actual un núcleo que permita realizar una serie de operaciones sobre estados cuánticos. Este núcleo será, por supuesto, una simulación clásica de un estado cuántico.

En las siguientes secciones se detallara la construcción de este sistema y se dará pequeña introducción matemática a la información cuántica, de forma que sea más fácil comprender el resto del proyecto.

# Antecedentes

## Implementación física de un computador cuántico

El verdadero reto existente hoy en día en este campo es desarrollar un computador cuántico con unos tiempos de *decoherencia* suficientes para operar y con una tecnología que permita incrementar el número de qubits de forma arbitraria, es decir, que sea escalable.

Desarrollar un computador cuántico para computaciones pequeñas es relativamente fácil. Un espectrómetro de resonancia magnética nuclear (RMN), disponible en muchos laboratorios, se puede utilizar como un pequeño computador cuántico. La característica diferenciadora de este tipo de implementación es que la computación se ejecuta sobre una muestra con un gran conjunto de moléculas y no sobre una en particular.

En el 2001 un equipo de IBM realizo con éxito la factorización del número 15 en 3 y 5 mediante el algoritmo de Shor, utilizando un espectrómetro RMN sobre una muestra que contenía moléculas sintéticas, con cinco núcleos de y dos de . El espectrómetro utilizaba el espín de los núcleos como qubits, luego la máquina disponía de 7 qubits para operar.

Ilustración 1 Espectrómetro RMN del mismo modelo que el utilizado por IBM. [Fuente: http://www.mckscientific.com/]

Esta implementación es fácil de realizar y tiene tiempos de *decoherencia* relativamente largos. Aun así, no es un buen candidato para implementar un computador cuántico que supere a los ordenadores actuales ya que es tremendamente complicado aumentar el número de qubits disponibles.

Otra rama de avance en la implementación física de este tipo de máquinas se basa en lo que se denomina “Trampas de iones”.

Las trampas de iones utilizan una combinación de campos electromagnéticos para confinar iones individuales en una región del espacio.

Ilustración 2 Esquema de una trampa de iones

En 1995 J. I. Cirac y P. Zoller de la Universidad de Innsbruck, propusieron un modelo de computador cuántico utilizando trampas de iones que era capaz de realizar la operación “NOT-controlada” (se describirá más adelante). Para realizar las computaciones, los iones se enfrían utilizando un haz laser de frecuencia adecuada hasta su estado fundamental. Los qubits son los estados energéticos de los iones atrapados, luego tendremos un qubit por cada uno. Un haz laser por cada ion se encarga de operar con ellos, excitándolos o enfriándolos según sea necesario. En el modelo de Cirac y Zoller los iones que interaccionan no tienen por qué estar juntos.

En mayo de 2011 un grupo de investigadores de varias universidades público un artículo [http://arxiv.org/abs/1009.6126] en el que describían un método para mantener entrelazados hasta 14 iones en una trampa. Hasta la fecha es el mayor número de qubits mantenidos en una trampa iones.

Existen varios modelos más hoy día, algunos de gran complejidad y no es el punto central de este proyecto. Remito a la bibliografía para obtener más información [BIBLIOGRAFÍA].

## La unión entre el procesador cuántico y el clásico

El concepto de unión entre computador clásico y cuántico, concepto clave de este proyecto, está siendo explotado intensamente en el ámbito de la investigación. De hecho, se han realizado experimentos reales de computadores cuánticos a muy pequeña escala que combinan de forma efectiva ambos paradigmas.

Investigadores de la Universidad de California desarrollaron en el 2011 un concepto similar al que aquí se presenta: un computador cuántico con la arquitectura de Von Neumann [Mariantoni et al.]. El computador disponía de una unidad de procesamiento cuántico con dos qubits unidos por un bus de acoplamiento, una memoria cuántica de otros dos qubits y dos registros de puesta a cero, todo ello integrado en un chip superconductor. La ventaja de esta arquitectura es que podemos pasar el estado de los qubits a la memoria cuántica donde tienen tiempos de decoherencia mucho más altos (sobre 1µs) que en los qubits sobre los que se opera (sobre 400ns), de esta forma se pueden almacenar en memoria mientras se realizan otras operaciones en la unidad de procesamiento. El programa a ejecutar sobre los qubits está almacenado en un ordenador corriente, que emite los pulsos de microondas necesarios.

Hoy en día es prácticamente unánime en la comunidad científica que este es el camino a seguir. Dado que el sistema cuántico solo es superior en escenarios muy específicos, es más lógico utilizar un ordenador corriente, mucho más fácil de desarrollar y programar, para realizar la mayoría de las tareas. El computador cuántico entraría en acción a petición del clásico, cuando se dé una situación para la que este es más efectivo.

## Conocimientos necesarios

Para la correcta comprensión del Proyecto que se va a exponer, es necesario introducir al lector en el campo de la Física Cuántica. Se enfocará claramente hacia la rama de la Información Cuántica y de forma resumida, el lector interesado encontrará una extensa guía en [NielsenChuang]. Además, se presentará la arquitectura MIPS expuesta en [HenessyPatterson], en la que se apoya el Proyecto.

### 2.2.1 Física e Información cuántica

La física cuántica no es más que un marco matemático para el desarrollo de teorías físicas. Se basa en una serie de postulados empíricos, obtenidos prácticamente por ensayo y error, que aun así han resultado en una importantísima rama de la física de una precisión impresionante, con tan solo algunos problemas que se han ido refinando en sucesivas teorías. Aquí no necesitaremos tal nivel de precisión y nos apoyaremos en el marco matemático clásico de la Física Cuántica.

Los postulados de la Física Cuántica, que definen dicho marco matemático, cambian dependiendo de la fuente que se consulte y de a qué rama se enfoque dicha fuente. Por supuesto todos vienen a decir lo mismo solo que planteado de diversas formas. Dado que aquí buscamos el punto de vista de la Información Cuántica, expondré dichos postulados citando a [NielsenChuang]. Me apoyaré en ellos para explicar los conceptos que sean necesarios para la comprensión del Proyecto.

#### Los postulados de la física cuántica

* **Primer postulado: El espacio de estados**

“Asociado a cualquier sistema físico aislado existe un espacio vectorial complejo con un producto interno definido (es decir, un espacio de Hilbert) denominado **espacio de estados** del sistema. El sistema queda completamente descrito por su **vector de estado**, que es un vector unitario en el espacio de estados del sistema”

*-[NielsenChuang]*

Como se puede observar la definición de este *espacio de estado* y su  *vector de estados* deja mucho sin fijar. ¿Cuál es el espacio de estados de un sistema físico dado? La pregunta dista mucho de ser trivial, muchos espacios de estados de sistemas físicos estudiados hoy día son de una complejidad tremenda.

Afortunadamente, nosotros solo necesitamos el espacio de estados más simple que se puede plantear, lo que se denomina un **qubit.** Un qubit tiene un espacio de estados de tan solo dos dimensiones complejas. Tomando como base de dicho espacio dos vectores ortogonales, llamémosles y , cualquier vector queda definido como:

Con α y β dos números complejos de forma que el vector sea unitario, es decir .

Esta notación se denomina *notación de Dirac*, los estados se representan con símbolos del tipo (*ket*) y se pueden identificar con *vectores columna*, cada uno de ellos tiene asociado un vector dual, con el símbolo (*bra*), que se pueden identificar, a su vez, con vectores fila. Uno se obtiene a partir del otro transponiéndolo y complejo-conjugándolo (conjugación hermítica). El producto escalar de dos vectores de este tipo se escribe de forma sencilla: .

Estos estados se pueden representar de forma matricial. Como tenemos un espacio de dos dimensiones complejas podemos tomar una base que defina este espacio de forma arbitraria, es decir, cualesquiera dos vectores complejos ortogonales y unitarios definirán una base válida.

Etiquetando las bases como y es fácil relacionarlas con los bits clásicos. El significado físico de estos estados dependerá de la implementación física correspondiente: el espín de los núcleos en un espectrómetro RMN, el estado de excitación de los iones en una trampa, etc. Pero los cálculos matemáticos relevantes son independientes de la implementación, luego podemos abstraernos de la implementación física y aun así proponer un marco matemático coherente para la computación cuántica.

Una base del espacio complejo de dos estados que simplifica los cálculos, al menos para esta introducción, es la que se denomina *base computacional*, definida como:

De forma que un vector arbitrario queda definido como:

Siempre y cuando .

Para intentar entender las similitudes y diferencias entre las computaciones clásica y cuántica, podemos relacionar sus bases de la forma:

0 lógico =

1 lógico =

De esa forma podríamos operar con los qubits, a simple vista, como si de lógica binaria se tratase. Pero el estado de un qubit no se reduce simplemente a o , existen estados que son perfectamente válidos como por ejemplo: , de los cuales no podemos decir que estén en el estado ni en el . La interpretación que se debe dar a este tipo de estados ni siquiera está clara hoy en día, se podría pensar que es la falta de información sobre el sistema, debida quizás a que la teoría cuántica es incompleta, la que nos está llevando a estados absurdos, a una simple representación probabilística de nuestro desconocimiento, es decir, la existencia de *variables ocultas* que no se están incluyendo en la teoría. Contra este enfoque, John Stewart Bell, publicó en 1964 un artículo en el que, partiendo tan solo de dos premisas:

* Realidad: los sistemas físicos tienen valores definidos para sus propiedades aunque no haya nadie observándolos.
* Localidad: la información, es decir, cualquier efecto físico, se propaga a una velocidad finita.

Asumiendo ambas premisas, las cuales definen el mundo como lo conocemos macroscópicamente, Bell obtuvo una desigualdad. Siguiendo el mismo camino, pero utilizando el marco matemático de la física cuántica, descubrió que dicha desigualdad se violaba, lo que quiere decir que si la física cuántica describe la naturaleza, la asunción de *realismo local* es falsa. En contra a lo que se podría pensar, experimentos llevados a cabo para poner a prueba estas desigualdades se han decantado siempre del lado de la física cuántica, aunque sin ser hasta la fecha del todo concluyentes.

Por tanto, aquí asumiremos que los estados del tipo están representando que el nivel energético, espín, etc. se encuentra en ambos estados al mismo tiempo. Esto se denomina *superposición cuántica de estados*, y es una de las claves de la potencia de la computación cuántica.

El problema es que ese tipo de estado es inobservable. Como señalaré en el tercer postulado, el hecho de observar un estado cuántico, en general, lo destruye. Pero que no podamos observar dichos estados no significa, como ya se ha mostrado, que sean ficticios; podemos obtener, si bien no todos los resultados posibles de una función, sí una propiedad general de esta, como puede ser su periodo. Esto será fundamental para los algoritmos cuánticos más famosos, que superan ampliamente a las maquinas clásicas.

Por último, comentar la forma más famosa para representar algoritmos cuánticos: los circuitos cuánticos. En un circuito cuántico un qubit se representa como una línea continua:

En las siguientes secciones completaré la descripcion de los circuitos cuánticos.

* **Segundo postulado: La evolución de los estados**

**“La evolución de un sistema cuántico cerrado viene descrito por una transformación unitaria. Es decir, el estado**  del sistema en el instante está relacionado con el estado en el instante por un **operador unitario** U que depende únicamente de los instantes y .”

*-[NielsenChuang]*

Esta evolución es una evolución discreta, nos lleva del instante al instantáneamente. Si queremos obtener cómo evoluciona el estado de forma continua tendremos que recurrir a la ecuación de Schrödinger pero por ahora no nos será necesaria.

Este postulado nos da intrínsecamente un elemento clave, podemos hacer evolucionar a los estados a voluntad, es decir, podemos operar con ellos siempre y cuando garanticemos que las operaciones que se realizan son unitarias.

Una vez más, cabe aclarar que esta la forma matemática de esta evolución se puede definir sin necesidad de elegir una implementación física. A partir de esta forma matemática podremos, posteriormente, obtener las ecuaciones concretas para una implementación física dada. Pero esto dista mucho de ser trivial y escapa al alcance de este proyecto.

Cuando un operador de este tipo se aplica a un qubit se denomina *puerta cuántica.*

Un operador es unitario cuando cumple que:

Un ejemplo de una puerta cuántica muy útil sería la contrapartida cuántica del inversor clásico (puerta NOT), la puerta cuántica X:

,

,

En la base computacional:

; ;

Es fácil comprobar que es unitaria.

La puerta cuántica quizás más importante se denomina *puerta de Hadamard* y nos permite hacer evolucionar los estados base hacia una superposición cuántica, ya que realiza:

y

Su representación matricial es:

En la representación de circuitos cuánticos, las puertas se representan como cajas etiquetadas con el nombre de la puerta:

H

X

El tiempo pasa de izquerda a derecha de la linea. En el ejeplo quedaría que .

* **Tercer postulado: Las medidas**

**“Las medidas cuánticas vienen descritas por una colección de operadores de medida. Estos operadores actúan sobre el espacio del sistema bajo medida. El subíndice ‘m’ indica una de las posibles respuestas de la medida. Si el estado previo a la medida es el** , la probabilidad de obtener un resultado ‘m’ será:

.

y el estado tras la medida quedará:

.”

*-[NielsenChuang]*

En el segundo postulado vimos como evolucionaba un estado cuántico en un sistema cerrado. Pero si queremos obtener resultados de los experimentos no tenemos más remedio que interferir en él. El hecho de observar un estado cuántico lo convierte en un sistema abierto y la evolución unitaria ya no es válida.

Los operadores de medida son proyectores sobre el subespacio sobre el que se realiza la medida; véase el subespacio definido por para un resultado de 0 y para un resultado de 1. Entonces vemos que, como era de esperar, si medimos el estado es imposible obtener un 0 como resultado ya que su proyección sobre el subespacio 0 es nula. Esta proyección es la que da la probabilidad de obtener un resultado u otro.

Por ejemplo, si suponemos el estado y lo medimos obtendremos:

Y tras la medida el estado quedará:

Sabiendo que el proyector y .

En resumen, el módulo al cuadrado de los coeficientes que acompañan a cada vector en un estado cuántico nos da la probabilidad de encontrarnos dicho estado tras una medida y por tanto de obtener su resultado asociado.

El símbolo asociado a las medidas en un circuito cuántico es:

* **Cuarto postulado: los sistemas físicos compuestos**

“El espacio de estados de un sistema físico compuesto es el producto tensorial de los espacios de estados componentes. Es más, si tenemos los estados numerados de 1 a ‘n’ y el sistema ‘i’ se prepara en el estado , el estado conjunto del sistema completo será .”

*-[NielsenChuang]*

Hasta ahora habíamos trabajado con espacios tan solo de un qubit. ¿Cómo representamos estados en los que está involucrado más de un qubit? Realizando el producto tensorial de ambos estados. Por ejemplo si tenemos el estado y el el estado total resultante será: , o simplemente .

Un sistema de dos qubits ya no está inmerso en un espacio complejo de dos dimensiones, sino de cuatro. En forma matricial podemos hacer (en la base computacional):

Como vemos los vectores que representan el estado de dos qubits tienen dimensión cuatro. En general, la dimensión del sistema completo será de , con ‘q’ el número de qubits.

Por otra parte, las puertas cuánticas que afectan a un espacio de varios qubits ya no pueden ser de dimensión dos. Se da que los operadores que operan sobre este espacio son el producto tensorial de los operadores que actúan sobre los subespacios correspondientes, por ejemplo, aplica la puerta X al primer qubit y la puerta H al segundo. Nótese que la operación no es conmutativa, aplica H al primer qubit y X al segundo y su representación matricial será distinta en general.

En computación clásica, un conjunto de instrucciones de vital importancia son las condicionales, que ejecutan una operación solo si se cumple una determinada condición. En computación cuántica también existen este tipo de operaciones. La más simple y más utilizada es la puerta cuántica CNOT, que ejecuta la operación X (NOT) sobre un qubit denominado *objetivo* solo si otro denominado *control* está en el estado . Es decir, si tenemos un par de qubits , la operación CNOT realiza: , siendo el operador una suma módulo 2.

Un circuito cuántico tiene una línea horizontal por cada qubit que se quiera representar. La puerta CNOT tiene una representación especial, un punto negro indica el qubit control y el símbolo indica el qubit objetivo, ambos conectados por una línea:

#### Detalles a la hora de construir algoritmos cuánticos:

Dados los postulados vistos en el apartado anterior, encontramos una serie de restricciones para la construcción de *programas cuánticos*, la restricción más fuerte nos la da que nuestras puertas, que hacen evolucionar el estado, deben ser unitarias y por ende:

1. **Deben ser reversibles**: Al cumplir las puertas que , lo que estamos afirmando es que su *conjugada hermítica* (†) es su inversa, es decir, su inversa siempre existe. Por tanto, siempre debemos ser capaces de reconstruir la entrada de una puerta a partir de su salida.

Si se observan algunas puertas clásicas, como por ejemplo la puerta AND, resulta que es imposible obtener la entrada a partir de la salida para ciertos casos:

0

1

0

1

0

0

0

0

0

Como se observa a partir de su salida 0 es imposible saber que entrada se introdujo.

Entonces, ¿no se pueden realizar una puerta AND cuántica? Si se puede construir, pero debemos conservar información suficiente para reconstruir la entrada y para ello no es suficiente con dos entradas para esta puerta en concreto:

Necesitamos, como en la figura, un qubit auxiliar.

Esta puerta reversible con dos qubits de control se conoce como *puerta de Toffoli* y permite ejecutar la operación *NAND* (sin más que introducir un 1 en vez de un 0 como qubit auxiliar). Como la operación *NAND*  es universal para las puertas clásicas, queda demostrado que cualquier puerta clásica se puede ejecutar utilizando una combinación de puertas reversibles.

1. **Es imposible clonar un estado cuántico desconocido:** Lo que se conoce como *teorema de no-clonación*. La prueba es simple; imaginemos dos estados y a copiar y una puerta C que realiza la copia:

Si realizamos el producto interno de estas dos ecuaciones obtenemos:

Pero la ecuación tan solo tiene dos soluciones, o x=1 o x=0. Por tanto los estados o son iguales o son ortogonales. Así que es imposible construir una puerta que clone un estado cuántico cualquiera.

Para los acostumbrados a la computación clásica, esto es un resultado impactante. Algo tan visto en cualquier circuito digital como:

a

a

a

No se puede realizar en una computación cuántica.

Con estos dos puntos, pensando en circuitos cuánticos, concluimos que ignorando puertas no unitarias como la de la medida, siempre tendremos a la salida el mismo numero de qubits que teniamos a la entrada.

### 2.2.2 La arquitectura MIPS

# Gestión del proyecto

# Objetivo

El objetivo general del proyecto es diseñar y simular en Java un procesador real, con la característica de que tendrá una unidad funcional capaz de ejecutar instrucciones de computación cuántica.

Este conjunto de instrucciones debe ser suficiente para ejecutar cualquier algoritmo cuántico que se desee, en más o menos tiempo, es decir, debe ser una máquina cuántica universal en la que se pueda ejecutar cualquier operación combinando las instrucciones implementadas.

El simulador desarrollado, por tanto, debe tener las siguientes propiedades:

* Ser capaz de simular los estados cuánticos de una forma lo más óptima posible, de forma que sea posible utilizar una cantidad de qubits razonable para la ejecución de una serie de algoritmos de prueba, que serán:
  + El algoritmo de Deutsch, demostración de la eficiencia del computador.
  + El algoritmo de Shor, de factorización en números primos.
  + El algoritmo de Grover, de búsqueda en una base de datos desordenada.
* No debe ser simplemente un intérprete de instrucciones de un código maquina extendido, sino un simulador del procesamiento de las instrucciones por un computador real, con una arquitectura bien definida y realista, al menos en los componentes clásicos del procesador.
* Para ello imitará a los lenguajes de descripción de hardware, como VHDL, lo que permitirá diseñar el computador componente a componente de acuerdo a su arquitectura.
* Con respecto a la unidad funcional cuántica, se deben hacer las mínimas asunciones posibles con respecto a su funcionamiento, ya que se trata de una unidad ficticia, irrealizable con la tecnología actual, y desconocemos cómo se comportará cuando se desarrolle.

Por último se presentará una interfaz gráfica que permita hacer un seguimiento del estado del procesador en cada componente relevante, incluyendo el estado cuántico de la nueva unidad funcional.

# Desarrollo del proyecto

## Simulando los sistemas cuánticos

Este punto del proyecto es crítico, de ello depende la eficiencia del programa final en gran parte.

Realmente, no se conoce ningún algoritmo eficiente para simular los sistemas cuánticos (si se descubriera, invalidaría la ventaja de los computadores cuánticos), así que habrá que buscar una solución lo más eficiente posible a fin de minimizar el retraso en estos cálculos.

Este parte del proyecto está programada en el proyecto Java *Qubit101*, que es un simulador de circuitos cuánticos que utiliza la implementación que se explicará y de la que se aprovecha el proyecto *qMIPS*, centro de este trabajo.

#### Representación matricial de los sistemas

Una primera idea surge rápidamente a raíz de la explicación expuesta sobre la computación cuántica. Los estados se pueden representar como vectores y los operadores como matrices, y es trivial realizar esta implementación en prácticamente cualquier lenguaje de programación:

* Se elige una representación para los estados, que serán vectores como hemos dicho. Por comodidad, lo más lógico es elegir la base computacional, que realiza simplemente:
* Se busca una representación de las puertas cuánticas, que serán matrices como hemos dicho y se aplicaran multiplicándolas por el vector de estado correspondiente:
* Y se obtiene una representación para sistemas de muchos qubits. El producto tensorial en las matrices se conoce como *producto de Kronecker*  y es relativamente fácil de implementar.

En esta representación la computación:

H

X

se calcularía:

Es en el último punto donde encontramos el problema de esta implementación: el producto de Kronecker hace crecer a las matrices y a los estados con el número de qubits en el sistema, para adecuarse a los nuevos grados de libertad. De hecho, para un número *q* de qubits, tendremos vectores de estado de elementos y matrices para los operadores de elementos, es decir, una complejidad exponencial en el número de qubits.

Esta primera implementación se realizó y fue descartada ya que con aproximadamente 10 qubits en el sistema, la máquina virtual de java era incapaz de reservar suficiente memoria para los estados y las matrices. Con 8 o 9 el tiempo de cálculo rondaba los minutos por operación.

#### Representación en mapa de componentes de los estados

El programa del proyecto utiliza una implementación radicalmente distinta. Esta implementación trata de alejarse del principal problema de la anterior: matrices de un tamaño desmesurado.

Si observamos los estados cuánticos en su forma matricial y su representación en notación de Dirac, por ejemplo:

Podemos ver que parece mucho más cómodo trabajar con la notación de Dirac, donde solo utilizamos tres bits para representar lo mismo para lo que necesitamos ocho en la representación matricial.

Además, realizar operaciones sobre el vector requeriría una matriz con 64 datos para aplicarle, por ejemplo, una puerta X al tercer qubit:

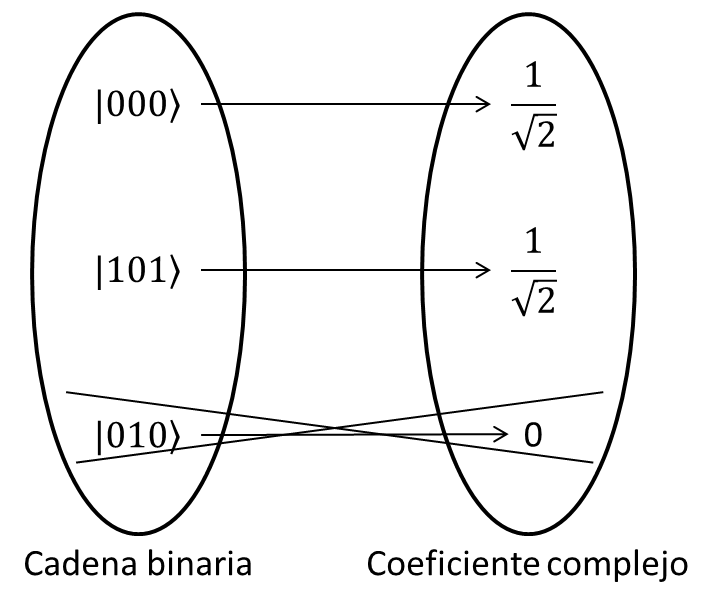
En la notación de Dirac tan solo necesitamos *interpretar* un símbolo que nos diga que puerta debemos aplicar:

La situación es más complicada si tenemos un estado en superposición. En la representación matricial, un estado de este tipo quedaría representado como un vector con dos componentes o más componentes no nulas:

Equivalente en la notación de Dirac a:

Vemos que aquí la ventaja de esta notación se ha diluido ligeramente. La situación se agravará según aumentemos el *tamaño de la superposición* pero, al fin y al cabo, tendremos como límite máximo *datos* en total para la superposición máxima y en la notación matricial tendremos esa cantidad de *datos*  sea como sea el tamaño de la superposición: hemos trasladado la complejidad del algoritmo a ese tamaño en vez del al número de qubits*.*

Pero, ¿Cómo son esos *datos* en cada representación? En la representación matricial estos datos serán una serie de coeficientes complejos en forma de vector. En la representación de Dirac tendremos una serie de parejas: una cadena binaria asociada a un coeficiente. Si este coeficiente es cero dicha pareja no aparece en la representación:



Así, tendremos qua almacenar estas parejas para tener toda la información del estado, de forma que un estado como:

Quedaría almacenado de la forma:

En programación, la estructura de datos relevante es un *mapa*. Los mapas son estructuras de datos que asocian a cada *clave* un *valor*, de modo que si queremos obtener un valor concreto nos referimos a su clave asociada. Aquí, nuestras claves serán estas cadenas binarias y los valores asociados los coeficientes complejos.

Concretamente, en el proyecto, las cadenas binarias *clave* son de la clase *ClassicState* y los coeficientes binarios  *valor*  son instancias de la clase *Complex*.

Ahora la eficiencia de la simulación dependerá en gran medida de la implementación de los mapas que se utilice.

Java proporciona en su API una implementación de mapas ordenados en forma de un tipo muy eficiente de árbol binario que se conoce como *árbol rojo-negro*, esta implementación nos asegura un tiempo logarítmico en el número de elementos para obtenerlos o insertarlos. La clase de la API de Java que representa estos árboles se llama TreeMap.

En la clase *QuantumState*, está implementada la lógica de los estados cuánticos. Al añadir un componente a la superposición el algoritmo busca si esta componente ya está en el estado y suma los coeficientes. Si no estuviera simplemente añade la componente al mapa. En todo caso, lo elimina si el coeficiente resulta ser cero.

Así están ubicadas estas tres clases ya descritas en la estructura de paquetes del proyecto:

Queda ver como se aplican las puertas cuánticas sobre esta representación de estados. Como hemos dicho anteriormente, en notación de Dirac representamos las puertas con símbolos y tan solo necesitamos interpretarlos para aplicarlas. En este caso, solo necesitamos decirle al simulador como actúa la puerta sobre cada componente de la superposición, de forma que crear el estado resultante consista en recorrer todo el estado de entrada generando los nuevos componentes. De esta forma, el mismo trozo de código que representa una puerta X para un solo qubit, funciona igualmente para cientos de ellos.

Entrando en la implementación, todas las puertas *unitarias* actúan de la misma forma y tienen una plantilla que pueden extender y definir así, tan solo, como se actúa sobre cada componente y la lógica para recorrer el estado o hacer las operaciones utilizando qubits de control queda encapsulada en la plantilla, esta plantilla está en la clase *UnitaryGateTemplate*. Una vez extendida la plantilla solo habría que implementar el método abstracto *singleComponentOperation,* que para la puerta X sería:



Las puertas que no son unitarias no pueden implementar esta plantilla ya que no pueden ser operaciones controladas. Por suerte, no se utilizan muchas de ellas.

El simulador tiene suficientes puertas cuánticas para ser una máquina cuántica universal y de hecho es suficiente con menos, pero sería incomodo trabajar con un conjunto de puertas muy reducido. Estas puertas ya implementadas son las siguientes:

* *AddQubit:*  Se encarga de añadir un qubit más al sistema, en la posición que se le indique. Esta puerta no es unitaria.
* *CircuitGate:* Esta es una puerta especial, encapsula todo un circuito cuántico en su interior a modo de subrutina.
* *HadamardGate:* La puerta de Hadamard.
* *Measure:* Se encarga de realizar las medidas sobre los estados. Dispone de una variable consultable con el resultado de la medida. Esta puerta no es unitaria.
* *PauliXGate:* La puerta X, correspondiente a un inversor clásico.
* *PauliYGate:* La puerta correspondiente a la matriz Y de Pauli.
* *PauliZGate:* La puerta correspondiente a la matriz Z de Pauli.
* *PhaseShiftGate:* Puerta de cambio de fase, rota la fase del estado dependiendo de un parámetro de entrada.
* *SwapGate:* Intercambia el estado de dos qubits del sistema.
* *TraceOut:* Descarta un qubit del sistema. Previamente lo mide para romper cualquier entrelazamiento entre el qubit afectado y los demás. Esta puerta no es unitaria.

En la estructura de paquetes del simulador estas puertas están almacenadas en las clases de la imagen:

En esta representación se han conseguido simular hasta 22 qubits en superposición antes de que la máquina virtual de Java se quede sin memoria. El sistema tarda unos 4 segundos en crear dicha superposición:

Se puede observar aquí claramente el comportamiento exponencial de la complejidad del algoritmo. En cuanto a la memoria que ocupa el estado el comportamiento es igualmente exponencial, y la máquina virtual de Java lanza una excepción cuando se alcanza un tamaño por defecto de aproximadamente un GB:

Este motor de simulación será el utilizado por el proyecto qMIPS para simular la unidad funcional cuántica.

## Simulando hardware en Java

El sistema debe ser capaz de imitar el hardware de la maquina sin poner ninguna restricción con respecto a qué tipo de componente de está simulando. Para ello se ha construido un marco en el que resulta sencillo construir estos componentes, usando un diseño parecido a lo que aportan los lenguajes específicamente para descripción de hardware como VHDL o Verilog.

#### Definiendo los dispositivos

En el sistema, todo componente debe heredar de la clase *Device*, una clase abstracta que encapsula toda la lógica común a todos los componentes. El programador solo debe encargarse de definir cómo reacciona el componente a los cambios en sus entradas.

**Los componentes se interconectan con buses, que pueden tener uno o varios cables. Son instancias de la clase *Bus*. En los bus se escriben y se leen vectores lógicos; arrays de datos booleanos que indican el nivel de tensión del cada cable. Estos arrays son instancias de la clase *LogicVector.*

Imagen 5 Clases para definir dispositivos

Cada componente debe de tener como parámetros de su constructor los buses, tanto de entrada como de salida, de los que quiera disponer y se interconectan compartiendo los buses al instanciarlos.

Para definir el comportamiento de los buses, se debe implementar dentro de los hijos de *Device* la función *defineBehavior()* que debe ser llamada al final del constructor además. Dentro de esta función tendremos una o varias llamadas a la función *behavior(…)*, donde se indicará a los cambios de que buses debe reaccionar el componente y qué hacer ante esos cambios.

Mostraré como ejemplo como construir un registro síncrono:

* Se declara la clase heredada de Device y se colocan los buses como atributos, para poder utilizarlos para definir el comportamiento:
* **Se define el constructor, con todos los buses como parámetros de entrada, se asignan a sus atributos y se llama al método *defineBehavior():*
* Ahora definimos el comportamiento del dispositivo, implementando la función *defineBehavior().* En su interior encontramos dos llamadas a *behavior(..).* La primera, indica que ocurre al darse un ciclo de subida en el reloj. Para eso colocamos como primer parámetro un array con tan solo el bus correspondiente al reloj (*clk*), ya que solo debe reaccionar así ante cambios en este bus. Como segundo parámetro se introduce el comportamiento en si, como una clase hija de *Behavior* de la que debemos implementar el método *task()*. El método comprueba que realmente en un ciclo de subida leyendo el reloj, además comprueba que la señal de habilitación está activada. Si se dan ambas, escribe y mantiene en su salida lo que tenga a la entrada.
* La segunda llamada a *behavior(…)* se usa para reiniciar el registro si se activa la señal de reset.

Cabe señalar que el reloj que se usa en el sistema no hereda de la clase *Device* ya que se trata de un dispositivo especial, que no debe reaccionar a ningún evento de escritura sino que debe disparar dichos eventos de forma espontánea.

#### Sincronizando los dispositivos en ejecución

Cada dispositivo de un sistema construido con este marco se ejecuta en su propio hilo. Al registrarse las tareas que debe realizar un dispositivo, se asocian a los buses correspondientes dichas tareas, de forma que si algo escribe en uno de ellos esas tareas se realizan y provocarán otras escrituras que, a su vez, pondrán en marcha más.

Cuando un sistema de un gran tamaño está en ejecución, puede tener un gran número de tareas corriendo al mismo tiempo y que deben ejecutarse en un orden estricto. El proyecto está construido de forma que es fácil cambiar la forma en la que se sincronizan los dispositivos, se dispone de una o varias clases de sincronización que implementan la interfaz *Synchronization*. Que implementación se utiliza se define en la clase *SyncShortcut* y en el proyecto se utiliza la clase *PoolSync.*

En dicha clase se utiliza lo que se denomina una *piscina de hilos* para administrar los hilos en ejecución. Una piscina de hilos es una herramienta a la que se envían una serie de tareas y se encarga de ejecutarlas en una serie de hilos que se crean tan sólo una vez y si no tienen nada que hacer se duermen. Existen piscinas con un número de hilos fijo, donde las tareas esperan a tener un hilo disponible para ejecutarse y piscinas que crecen con el número de tareas. En el proyecto se usa esta segunda implementación, de forma que si hay más tareas que hilos se crean nuevos hilos y si ocurre al contrario habrá hilos durmiendo a la espera de trabajo. Si pasa una cantidad fijada de tiempo y algún hilo no ha trabajado se destruye.

Utilizar una piscina de hilos:

1. Evita la carga de trabajo extra que significa crear los hilos. Tan solo se crean en la primera iteración del programa y en las sucesivas están ya creados y disponibles.
2. Simplifica el manejo de los hilos. El programador ya no tiene que preocuparse de crear los hilos y sincronizarlos ya que está todo encapsulado en la piscina.

En esta implementación los ciclos de reloj tienen una duración muy variable, desde microsegundos para operaciones aritméticas hasta minutos u horas para operaciones de tipo cuántico, por tanto no es válido que el reloj emita flancos de forma periódica en el tiempo. El funcionamiento de la clase *PoolSync* es el siguiente:

1. El reloj emite un flanco por alguna interacción externa, que normalmente será la interfaz de usuario, activando una serie de tareas de los dispositivos que reaccionan al reloj, los síncronos.
2. Las tareas recién despertadas no se ejecutan de inmediato sino que esperan en una cola a que todos los dispositivos anteriores hayan terminado.
3. Cada vez que una tarea concluya informa a la clase de sincronización que lleva una cuenta del número de ellas que están en funcionamiento.
4. Cuando no quedan más tareas en funcionamiento se activan las que estaban en espera que a su vez pondrán en la cola más tareas.
5. En el momento en el que no queden más tareas en ni funcionamiento ni en la cola de espera, se considera que el sistema ha llegado a un estado estable y se despierta al reloj para que emita otro flanco.

Utilizando este sistema se garantiza que las tareas se ejecutaran en el orden correcto y se podrá esperar si alguna lleva un tiempo muy largo. Además en número de hilos en la piscina tras el primer ciclo será tan grande como el número máximo de tareas que se lancen a la vez y en general no tendrá que crecer.

## Diseñando el procesador qMIPS

El procesador diseñando en el proyecto está basado en una versión simplificada del MIPS clásico, sin encadenamiento de instrucciones, que utiliza una cantidad variable de ciclos por instrucción, presentado de forma educativa en el libro [HennessyPatterson]. Al presentado en el libro se han añadido varias instrucciones, siendo bastante cercano al procesador MIPS I.

El procesador tiene las siguientes características:

* Bus de instrucciones y bus de datos de 32 bits.
* Fichero de registros de 32 registros de 32 bits. El registro 0 no contiene ningún valor, siempre vale cero.
* Memoria común de datos e instrucciones. No se ha implementado jerarquía de memoria y se supone que responde en un solo ciclo.
* Todas las operaciones son sobre 32 bits. No se aceptan instrucciones de *byte* o *halfword*.
* No dispone de unidad de punto flotante, solo usa aritmética entera.

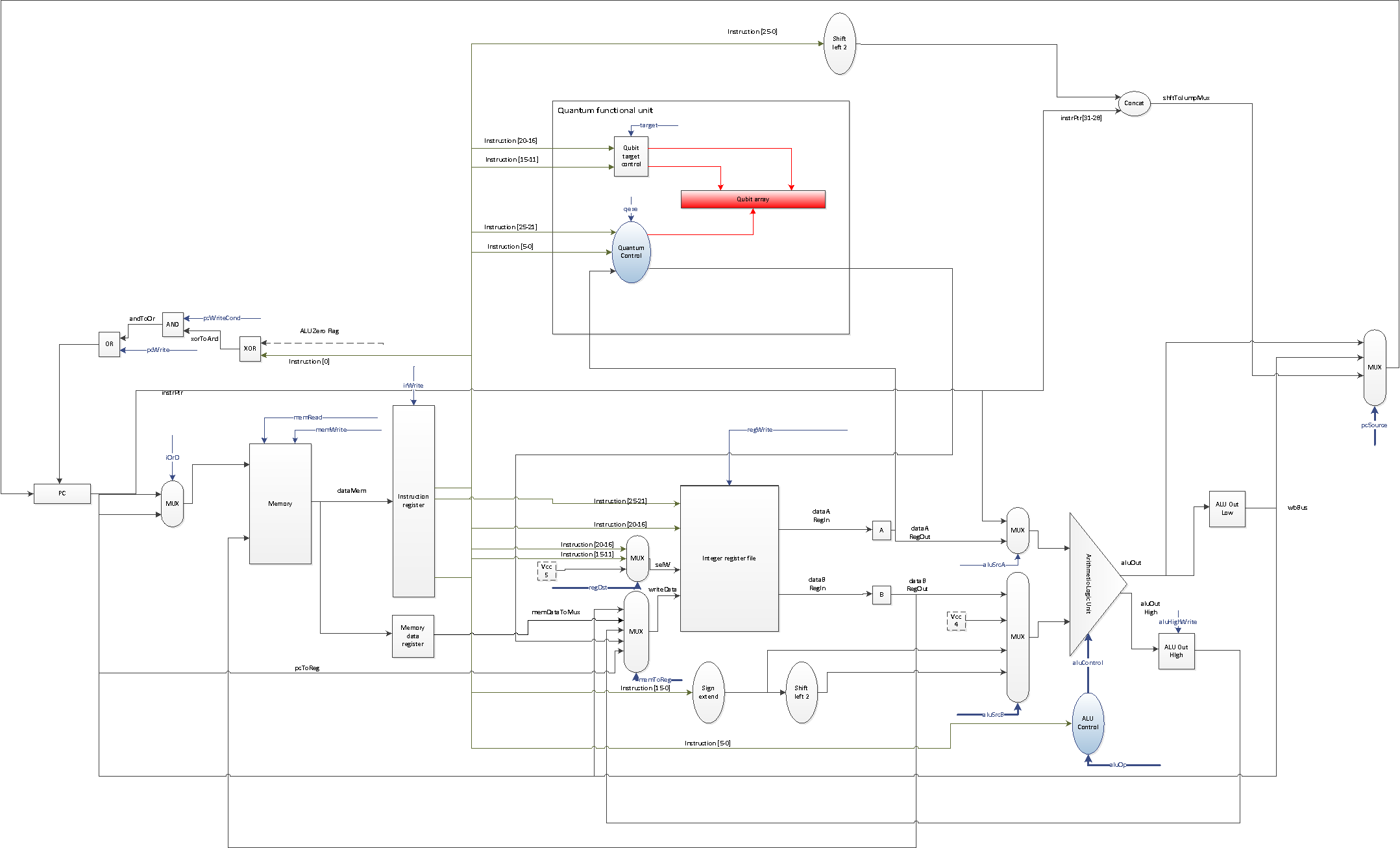
#### Arquitectura “física” del sistema

Uno de los objetivos del proyecto es presentar una arquitectura real de un procesador, implementable físicamente, no simplemente un intérprete de instrucciones. Esta arquitectura está programada en la clase *QuantumMIPS*:



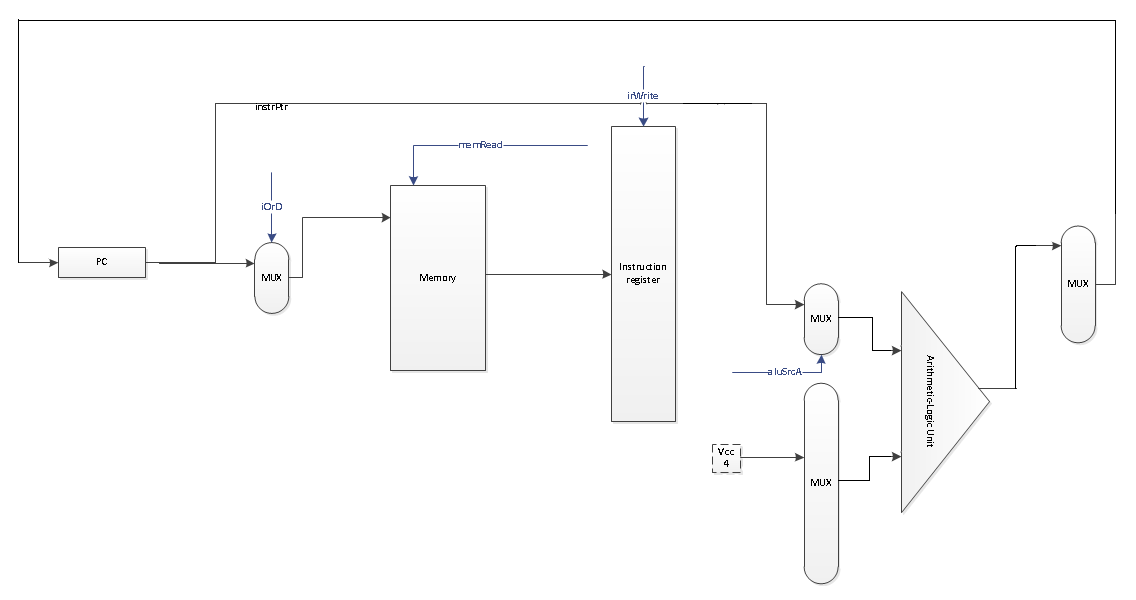
En esa clase se interconectan todos los dispositivos que conforman el procesador y se asignan todas las variables que necesita la interfaz gráfica para mostrar la información relevante al usuario.

Esta interconexión se entiende más fácilmente en el siguiente esquema:



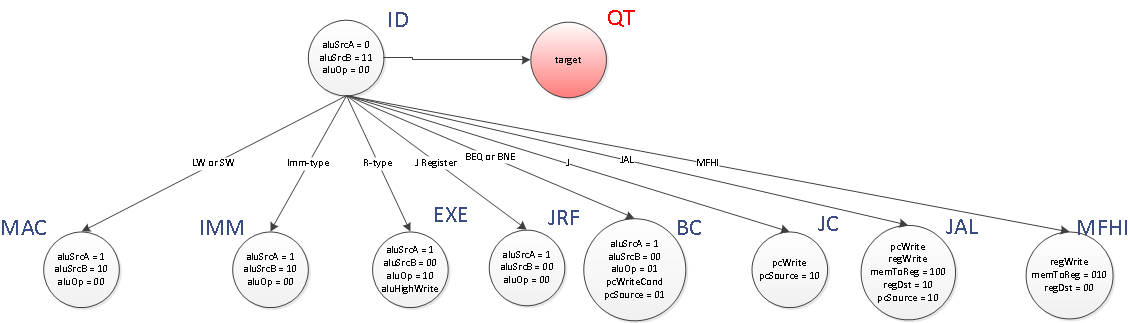
En esta figura las líneas verdes representan a la instrucción de memoria, las azules son señales de control, las rojas son operaciones cuánticas y el resto son negras.

El primer ciclo de ejecución se denomina fase IF (Instruction Fetch). Consiste siempre en traer la instrucción siguiente de la memoria y calcular el puntero de instrucción a la siguiente asumiendo que no hay ningún salto.





El segundo de ejecución se denomina fase ID (Instruction Decode). En esta fase la unidad de control debe decidir a partir del código de operación que señales debe activar para poner en funcionamiento la instrucción que corresponda. Además, se precalcula la dirección de un posible salto.



A partir de este momento la fase siguiente es muy distinta dependiendo del tipo de instrucción. En la siguiente imagen se detalla el árbol de estados de la unidad de control así como que señal se activa en cada una de ellas.



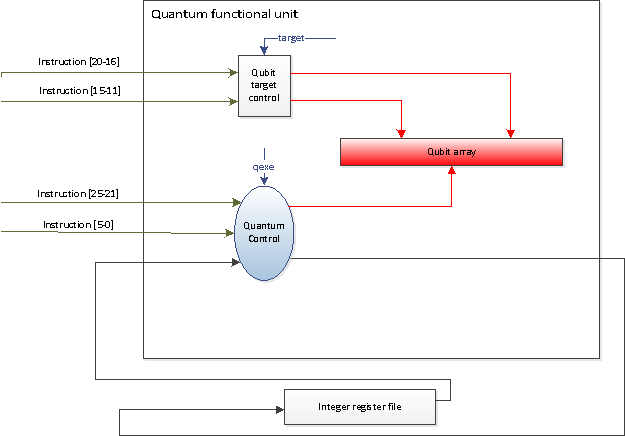
#### La unidad funcional cuántica

Este punto es el punto fundamental en el desarrollo del proyecto. Dado que no podemos hacer ninguna asunción sobre el funcionamiento de la unidad cuántica, no podemos fijarnos a ninguna implementación física concreta. Aun así, para que sea mas fácil visualizar su comportamiento, vamos a asumir que se trata de una trampa de iones.

En las trampas de iones se dispone de una serie de átomos atrapados por campos electromagnéticos oscilantes, que confinan cada partícula espacialmente. Una serie de haces laser se encargan de enviar señales electromagnéticas adecuadas para operar con cada ion de la trampa.

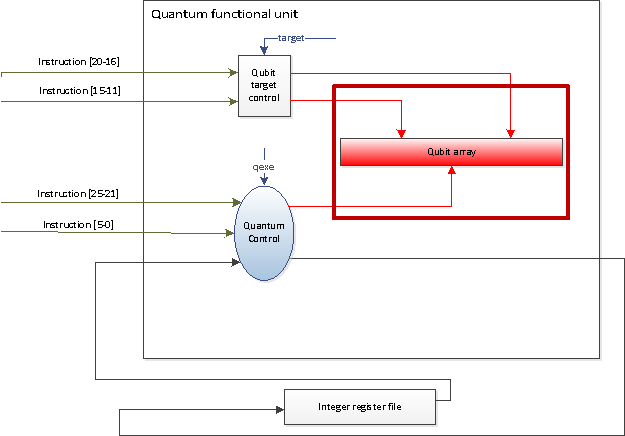
La imagen superior representaría una trampa de iones de 4 qubits.

La unidad funcional cuántica implementada en el proyecto tiene la siguiente estructura:

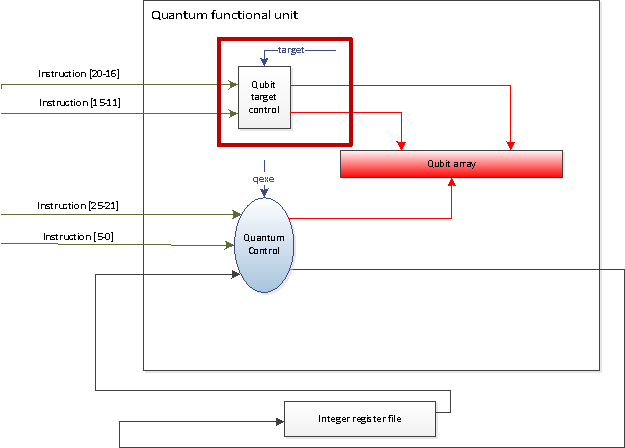


Consta de tres partes principales:

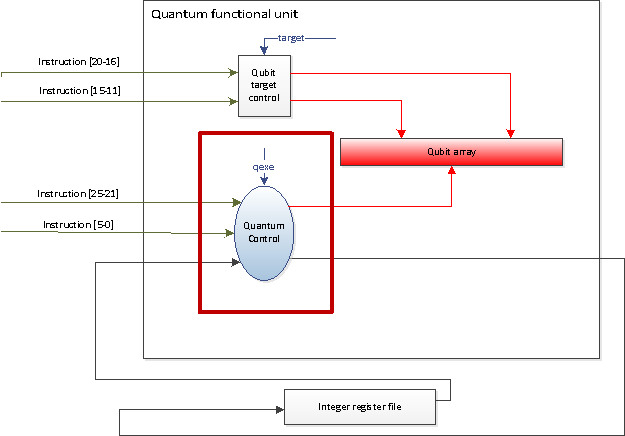
* Un *array* *de qubits* que en el caso del simulador contiene 32 qubits, cada uno de ellos se etiqueta con una ‘Q’ seguida de un número, de forma que para referirnos al primer qubit escribiremos Q0, Q1 para el segundo, etc. Hasta llegar al Q31. Cada uno de estos qubits sería un ión de la trampa.



* Un sistema de apuntado, que se encarga de desplazar el láser sobre el qubit correspondiente. Este sistema obtiene información sobre a que qubit debe apuntar de la instrucción. Se encarga de señalar el qubit objetivo y, si no son ambos el mismo, el qubit de control. Debe quedar claro que esta imagen de los láseres es ficticia, se trata simplemente de *preparar* los qubits sobre los que se va a actuar.



* Una unidad de control cuántica, que dependiendo de la instrucción actual decida qué *onda* deben enviar los láseres de forma que tenga el efecto de la puerta cuántica requerido en el qubit apuntado.



El proceso a la hora de ejecutar una instrucción de tipo cuántico sería el siguiente:

1. La instrucción se trae de memoria (IF) y se decodifica (ID) como cualquier otro tipo de instrucción.
2. La unidad de control detecta la instrucción de tipo cuántico y ordenad a la unidad de apuntado que debe preparar los qubits seleccionados (fase QT). Estos qubits vienen identificados en la propia instrucción: los cables del 20 al 16 (5 cables), indican que qubit es el objetivo. Los cables del 15 al 11 (otros 5) indican cual es el de control. Si ambas son iguales la unidad de apuntado interpreta que no hay qubit de control.



1. En el siguiente ciclo la unidad de control ordena a la unidad de control cuántica que realice las operaciones seleccionadas (fase QEX). Toma la operación seleccionada de los cables de la instrucción del 0 al 5. Además, esta unidad necesita los cables 21 al 25 por si se utiliza algún parámetro que forma parte de la instrucción así como acceso al fichero de registros por si el parámetro procede de este. De esta unidad parte el único bus de salida de la unidad cuántica que escribirá en el registro correspondiente los resultados de una posible medida.



1. Si la instrucción era una operación cuántica corriente, se procede a ejecutar la siguiente instrucción. Si se trata de una medida, se inicia la fase de medida (fase QMEA). En esta fase se ordena al fichero de registros que escriba el resultado de la medida.



1. Se ejecuta la instrucción siguiente.

Cabe señalar que la idea más generalizada en este tipo de combinaciones de computación *clásico-cúantica* es que se utilicen *registros cuánticos*, pequeñas agrupaciones de qubits de un tamaño definido o variable para almacenar los estados y que cada operación se pueda ejecutar sobre un registro de completo de ese tipo. Aquí se está operando sobre qubits independientes, es como si en un computador clásico se ejecutaran las instrucciones aritméticas típicas bit a bit: sumar el bit 2 con el 3, restar el 4 del 1, etc. ¿Por qué se ha elegido entonces esta representación? Por tres razones:

* La complejidad de la simulación no nos permite crear superposiciones de más de 22 qubits sin que la máquina virtual de Java deje de funcionar. Esta simulación almacena aun así, 32 qubits. Se podrían utilizar, por ejemplo, 4 registros de 8 qubits pero parece obvio que son demasiado pocos registros para realizar operaciones de una cierta complejidad. En el caso contrario nos encontraríamos con muchos registros pero demasiado pequeños para que sean verdaderamente útiles.
* La mayoría de los algoritmos cuánticos están muy bien definidos en forma de circuitos cuánticos y estos se suelen referir a qubits independientes en muchos momentos. Si utilizáramos qubits agrupados, tendríamos que utilizar todo un registro para usar un solo qubit y con un número tan pequeño de ellos no sería práctico.
* También con respecto a los circuitos cuánticos, referirnos a cada uno de ellos independientemente nos da una transformación de circuito a código máquina prácticamente directa.

#### Las instrucciones y el compilador

El procesador presentado en la fuente [HennessyPatterson] aceptaba tan solo un subconjunto de las instrucciones que se han implementado. Aun así este no es un procesador MIPS clásico propiamente dicho ya que algunas instrucciones funcionan de una forma ligeramente distinta.

El compilador se ha desarrollado con la herramienta ANTLR. A esta herramienta se le proporcionan gramáticas en un lenguaje específico y construye “parseadores” en Java automáticamente.

Es bastante complejo construir gramáticas de una cierta complejidad con esta herramienta y, al no ser este el objetivo del trabajo, el compilador realiza muy pocas comprobaciones más allá de que la gramática del programa sea la correcta y es bastante rígido. Si el programa está mal escrito o contiene alguna incoherencia como por ejemplo una dirección negativa o fuera de rango, el programa fallará.

A continuación se muestran las instrucciones que acepta el compilador; Rd, Rs y Rt son registros, C representa un número entero con signo de 16 bits:

* Instrucciones aritmético-lógicas:
  + add Rd, Rs, Rt: Suma de los registros Rs y Rt en Rd. Lanza una excepción si hay desbordamiento.
  + addu Rd, Rs, Rt: Suma de los registros Rs y Rt en Rd. Ignora el desbordamiento.
  + sub Rd, Rs, Rt: Resta de los registros Rs y Rt en Rd. Lanza una excepción si hay desbordamiento.
  + subu Rd, Rs, Rt: Resta de los registros Rs y Rt en Rd. Ignora el desbordamiento.
  + mult Rd, Rs, Rt: Multiplica Rs y Rt en Rd. Si Rs y Rt contienen más de 16 bits, el resultado puede ocupar más de los 32 bits que caben en Rd, se tendrán mostraran los bits menos significativos.
  + div Rd, Rs, Rt: Realiza la división entera de Rs entre Rt en Rd.
  + divu Rd, Rs, Rt: Realiza la división entera de Rs entre Rt en Rd.
  + and Rd, Rs, Rt: Realiza la operación logica Y entre Rs y Rt en Rd.
  + or Rd, Rs, Rt: Realiza la operación logica O entre Rs y Rt en Rd.
  + xor Rd, Rs, Rt: Realiza la operación logica XOR entre Rs y Rt en Rd.
  + nor Rd, Rs, Rt: Realiza la operación logica O negada entre Rs y Rt en Rd.
  + slt Rd, Rs, Rt: Coloca un 1 en Rd si Rs es menor a Rt.
* Operaciones con operando inmediato:
  + addi Rd, Rs, C: Suma Rs y C en Rd. Lanza una excepción si hay desbordamiento.

* Operaciones de memoria:
  + lw Rd, C(Rs): Carga en Rd el contenido de la memoria en la dirección Rs + C.
  + sw C(Rd), Rs: Escribe en la dirección Rd + C de la memoria el contenido de Rs.
* Instrucciones de salto, aceptan tanto una dirección como una etiqueta en el código:
  + jr Rs: Salta a la dirección almacenada en el registro Rs.
  + j C: Salta a la dirección especificada en C.
  + jal C: Salta a la dirección especificada en C y almacena en R31 la dirección de la siguiente instrucción. Se utiliza para realizar saltos a subrutina. Para realizar el retorno de subrutina se ejecuta jr R31.
  + beq Rs, Rt, C: Si Rs y Rt son iguales, avanza o retrocede el número de instrucciones especificados en C.
  + bne Rs, Rt, C: Si Rs y Rt son distintos, avanza o retrocede el número de instrucciones especificadas en C.
* Excepciones:
  + trap C: Lanza la excepción de código C. La instrucción TRAP 0 finaliza el programa.
* Operaciones cuánticas unitarias; Qt y Qc son qubit objetivo y de control, si son iguales la operación es no controlada:
  + qhad Qt, Qc: Puerta de Hadamard.
  + qx Qt, Qc: Puerta X de Pauli. Inversor.
  + qy Qt, Qc: Puerta Y de Pauli.
  + qz Qt, Qc: Puerta Z de Pauli.
  + qphs Qt, Qc, A: Cambio de fase especificado por el parámetro A (alfa) de 5 bits.
* Operaciones cuánticas no unitarias:
  + qmea Qt, Rs, S: Mide el qubit Qt y lo vuelca en Rs desplazado el valor de S, de 5 bits.
  + qrst Rs: Destruye el estado cuántico y vuelca en él contenido de Rs.
  + qoff Rs:

# Conclusiones

# Futuras líneas

# Bibliografía

# Anexo A: Código fuente