Portada

Índice

[1. Introducción 3](#_Toc358732901)

[2. Antecedentes 6](#_Toc358732902)

[2.1 Implementación física de un computador cuántico 6](#_Toc358732903)

[2.2 La unión entre el procesador cuántico y el clásico 8](#_Toc358732904)

[2.3 Conocimientos necesarios 8](#_Toc358732905)

[2.2.1 Física e Información cuántica 9](#_Toc358732906)

[2.2.2 La arquitectura MIPS 24](#_Toc358732907)

[3 Gestión del proyecto 27](#_Toc358732908)

[4 Objetivo 28](#_Toc358732909)

[5 Desarrollo del proyecto 29](#_Toc358732910)

[5.1 Simulando los sistemas cuánticos 29](#_Toc358732911)

[5.2 El simulador de circuitos cuánticos Qubit101 33](#_Toc358732912)

[5.3 Simulando hardware en Java 39](#_Toc358732913)

[5.4 Diseñando el procesador qMIPS 44](#_Toc358732914)

[5.5 Manual de uso 61](#_Toc358732915)

[5.5.1 Simulador de circuitos cuánticos Qubit101 61](#_Toc358732916)

[5.5.2 Simulador del procesador cuántico qMIPS 68](#_Toc358732917)

[6 Conclusiones 74](#_Toc358732918)

[6.1 Ejemplos 74](#_Toc358732919)

[6.1.1 El algoritmo de Deutsch 74](#_Toc358732920)

[7 Futuras líneas 78](#_Toc358732921)

[8 Bibliografía 79](#_Toc358732922)

[9 Anexo A: Código fuente 80](#_Toc358732923)

# 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 que sea capaz de mantener los estados totalmente aislados de forma que no pierdan sus propiedades cuánticas al interaccionar con su entorno, es decir, 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 hasta su estado fundamental utilizando un haz láser de frecuencia adecuada. Los qubits son los estados energéticos de los iones atrapados, luego tendremos un qubit por cada uno. Un haz laser por cada ión 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 publicó un artículo [http://arxiv.org/abs/1009.6126] en el que describían un método para mantener entrelazados, es decir, mantenidos en un estado cuántico coherente, 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, muchos de ellos de una enorme complejidad, sin bien ninguno se acerca aún al rendimiento de un ordenador moderno. 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 simulador de este proyecto, si bien desde una perspectiva distinta: una implementación física de un computador cuántico de dos qubits cuya arquitectura este basada en la de Von Neumann, pilar principal de la mayoría de los computadores clásicos modernos [Mariantoni et al.]. Este 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 del procesador clásico 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.

Premisas que definen el mundo como lo conocemos macroscópicamente, 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 y la comunidad científica acepta de forma general la violación de las desigualdades de Bell, si bien no se ha conseguido desarrollar ningún experimento sin lagunas hasta el momento.

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 utilizada para representar algoritmos cuánticos: los circuitos cuánticos. En un circuito cuántico un qubit se representa simplemente como una línea continua:

Es importante señalar que estos *circuitos* no son en ningun caso circuitos físicos como pueden ser los eléctricos, se trata simplemente de una representacion gráfica de un algoritmo que en su aplicación a un computador cuántico real se traducirian en una serie de pulsos laser, de radiofracuencia, etc; dependiendo de la implementación física de dicha máquina.

En los siguientes puntos se irá completando la descripción de este método para representar algoritmos 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 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 .

El producto de tensorial en su forma matricial se calcula de la siguiente forma, para un producto de un vector de dimensión dos por uno de cuatro:

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), por ejemplo:

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.

El producto tensorial de dos matrices se calcula, para una matriz 2x2 por otra 2x2, de la forma:

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 (*Controlled-NOT, Negación controlada*), 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 o la operación XOR, ya que si c = 0 deja inalterado a *t* () y si c = 1 lo invierte ().

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 puede realizar una puerta AND cuántica? Sí 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*. Invierte el qubit inferior tan solo si los dos superiores valen 1. Introduciendo un estado en el qubit inferior realiza la misma función que la operación *NAND*. 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.

#### El algoritmo de Deutsch:

El primer algoritmo que se desarrolló que demostró que los computadores cuánticos pueden ser superiores a los clásicos fue desarrollado por David Deutsch en [Año?]. El algoritmo no tiene ninguna aplicación práctica más allá de demostrar esta superioridad.

El *problema de Deutsch* que resuelve este algoritmo es el siguiente: dado un *oráculo* que ejecuta una de las cuatro funciones binarias de un bit, determinar si se trata de una de las dos funciones constantes: o ; o una de las funciones equilibradas: o .

Un *oráculo* es una función cuyo comportamiento interno es desconocido o simplemente no es importante (una caja negra). Algunos algoritmos cuánticos se basan en este tipo de funciones para operar.

El algoritmo de Deutsch es capaz de resolver el problema con tan solo una llamada al *oráculo* contra el mínimo de dos que necesita un computador clásico.

Para ejecutar este algoritmo tan solo son necesarios dos qubis.

El estado de entrada al algoritmo debe ser el . El primer paso es aplicar una puerta de Hadamard a ambos qubits de forma que el estado quede:

El oráculo se aplica negando el segundo qubit si el resultado de aplicar la función sobre el primero da 1. En forma de circuito:



De esta forma, el estado del sistema sería, tras la llamada al oráculo:

Y si, por último, aplicamos una puerta de Hadamard al primer qubit:

Ya es fácil observar que si medimos el primer qubit y obtenemos un 0, nos aseguramos de que y que por tanto la función es de tipo constante. Sin embargo si obtenemos un 1, se tiene que y la función es equilibrada.

El circuito completo del algoritmo quedaría:



Este algoritmo al fin y al cabo está calculando una propiedad global de la función *f(x)*: su periodo, con tan solo una ejecución de dicha función. Esto será fundamental en la ejecución del algoritmo de factorización de Shor.

#### El algoritmo de Grover:

Este algoritmo cuántico fue desarrollado por Lov K. Grover en 1996. Trata de buscar en una colección desordenada de datos uno o varios elementos que cumplan una cierta propiedad. Aquí particularizaré para el caso en el que tan solo exista una solución pero es sencillo generalizarlo para varias de ellas [NielsenChuang]. Este algoritmo es útil para aquellos problemas en los que identificar una solución es sencillo pero encontrarla es tremendamente complicado. Es capaz de encontrar dicha solución en un tiempo de orden , donde N es el número de elementos de la colección, contra el mejor algoritmo clásico que lo realizaría en un tiempo de orden . Se trata de un algoritmo que en general es probabilístico y su objetivo será intentar que la solución correcta sea la más probable.

El primer paso del algoritmo es colocar qubits en superposción utilizando puertas de Hadamard. Además, se debe utilizar un qubit más para el subespacio del oráculo preparado como en el algoritmo de Deutsch en el estado: :



Ahora hay que repetir veces el *operador de Grover*:



¿Cómo funciona este operador? Los *n* qubits superiores están inicialmente en el estado: , así que, si es el estado solución y es el estado conjunto del resto de componentes, podemos separarlo de la forma:

Por simplicidad, .

Al estar el qubit auxiliar en el estado preparado, el efecto del oráculo será simplemente el de cambiar la fase de la componente solución: . Si visualizamos las componentes en su forma vectorial, podemos entender este efecto como una *reflexión* en el plano definido por y sobre el vector :



El siguiente paso del algoritmo es aplicar el operador . El operador de cambio de fase invierte todas las componentes de la superposición excepto la componente , es decir, realiza la operación . El efecto conjunto del operador sería , que geométricamente es una vez más una reflexión, esta vez sobre el vector :



Lo que hemos conseguido con esto es girar el vector para aproximarlo hacia , vector solución del problema. Es fácil ver que si repetimos estas reflexiones un número adecuado de veces, llegara un punto de acercamiento máximo al vector solución y, por tanto, una probabilidad máxima de obtener la respuesta correcta en la medida. Se puede demostrar que el número de veces que se debe aplicar el operador de Grover para garantizar la probabilidad máxima es [NielsenChuang], es decir, el orden de complejidad del algoritmo es .

El circuito completo del algoritmo quedaría:



Este algoritmo no sirve tan solo para buscar un dato en una base de datos desordenada, también puede ayudar a acelerar la solución de problemas de la clase NP. Por ejemplo, se puede utilizar para determinar si un grafo tiene un ciclo Hamiltoniano, si encontramos un oráculo que los identifique, podemos encontrarlos en un tiempo .

#### Algoritmo de Shor:

Posiblemente el algoritmo cuántico más importante y conocido actualmente es el algoritmo de factorización en números primos de Shor. Ideado por Peter Shor en 1994, significó el inicio de la computación cuántica con aplicaciones fuera del ámbito científico ya que podría utilizarse para derribar el sistema criptográfico RSA tan extendido hoy en día.

El objetivo del algoritmo es dado un número cualquiera, encontrar uno de sus factores primos. Es capaz de realizarlo en un tiempo con *N* el número a factorizar, es decir, en un tiempo polinómico cuando lo mejores algoritmos clásicos se mueven en tiempos exponenciales.

Como ya se describió en el algoritmo de Deutsch, los algoritmos cuánticos pueden utilizarse para obtener una propiedad global de una función evaluándola tan solo una vez. Shor descubrió que el periodo de cierta función generada a partir de un número, estaba asociada a los factores de dicho número. Dado que el periodo es una propiedad global de las funciones, se puede utilizar un computador cuántico para obtenerlo y a partir de este calcular los factores primos.

El algoritmo está pensado para ejecutarse en una máquina clásico-cuántica ya que tiene una serie de pasos fácilmente ejecutables en la máquina clásica y una subrutina que debe ejecutarse en la cuántica. Si se quiere factorizar el número *M* el algoritmo consiste en:

1. Se genera un número aleatorio *a* desde 1 hasta *M*. Para ejecutar este paso también se puede utilizar la máquina cuántica. Si *a*  y *M* son coprimos seguimos adelante, si no, hemos encontrado un factor de *M* y el algoritmo finaliza.
2. Se utiliza la máquina cuántica para ejecutar la función sobre una superposición de todas las posibles entradas y ejecuta una transformada cuántica de Fourier inversa para obtener el periodo.

### 2.2.2 La arquitectura MIPS

La compañía “MIPS Computer Systems Inc.” nació en 1984 a partir de un proyecto de la Universidad de Stanford, para comercializar la arquitectura que el grupo MIPS CPU había desarrollado.

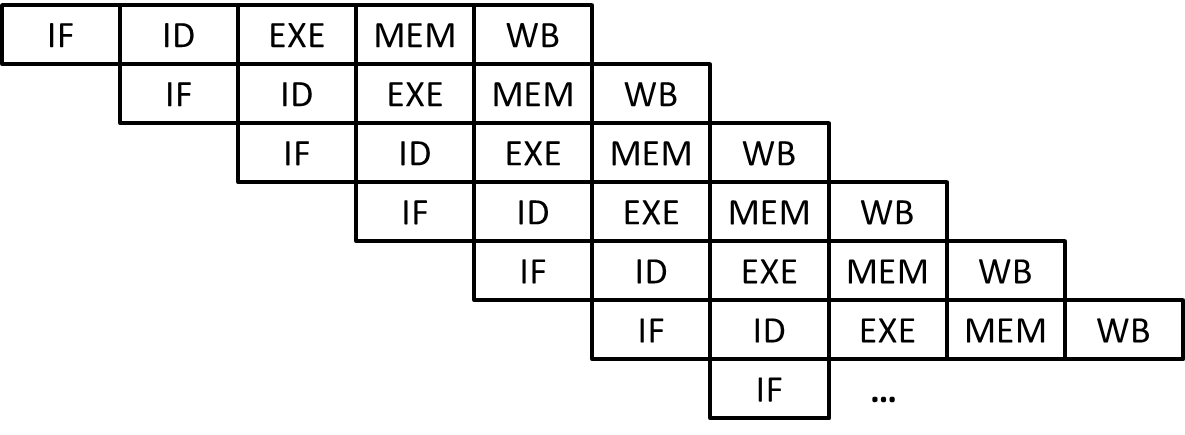
Se trataba de una arquitectura RISC con un encadenamiento estricto de 5 etapas. Estas dos características se combinan muy bien, para poder garantizar un flujo constante de instrucciones todas deben ser del mismo tamaño y por tanto, no muy complejas ya que no hay suficiente espacio en la instrucción para complicar las instrucciones [http://encore.fama.us.es/iii/encore/record/C\_\_Rb1951779].

Una arquitectura RISC (Reduced Instruction Set Computing) es aquella que en contraposición a las CISC (Complex Instruction Set Computing) tiene un conjunto de instrucciones que tiende a ser muy simple, de forma que el hardware necesita muy pocos ciclos para ejecutar una instrucción completa y los compiladores pueden optimizar la ejecución de forma fácil ya que no suelen utilizar operaciones muy complejas y específicas que proporciona un procesador CISC. Este tipo de arquitectura ha tenido tanto éxito que ninguna otra desarrollada posteriormente al 1985 ha seguido otra filosofía. Los procesadores RISC no suelen permitir operar directamente con la memoria, sino que debemos cargar el dato, operar con él y guardarlo de nuevo, estas arquitecturas se denominan *load/store* y es el caso de la arquitectura MIPS*.*

Los procesadores con encadenamiento dividen las instrucciones en una serie de pasos en cuya ejecución el hardware utilizado no se necesita en ningún otro paso, de forma que se pueden lanzar todas al mismo tiempo para acelerar la ejecución. El procesador MIPS I utilizaba una segmentación en 5 etapas:

* Fase IF (*Instruction Fetch*): La siguiente instrucción se obtiene de la memoria (o caché de instrucciones).
* Fase ID (*Instruction Decode*): Se decodifica la instrucción y se leen los operandos del fichero de registros.
* Fase EXE (*Execution*): Se utiliza la ALU para realizar los cálculos correspondientes ya sea una operación de tipo aritmético-lógica o el cálculo de una dirección efectiva en memoria.
* Fase MEM (*Memory*): Se realizan las operaciones con memoria (o caché de datos) ya sea lectura o escritura de los datos correspondientes. Aproximadamente tres de cada cuatro instrucciones no harán nada en esta etapa, pero mantenerla evita que dos instrucciones intenten acceder a la memoria al mismo tiempo.
* Fase WB (*Write Back*): Se escribe en el registro correspondiente el resultado de la ejecución de la instrucción.

Y las ejecuta de la siguiente forma:



Se puede ver que tras las cinco primeras instrucciones y, en el caso ideal, el procesador ejecutaría cinco instrucciones al mismo tiempo en cada ciclo. Es solo en el caso ideal porque las dependencias de datos (que una instrucción posterior necesite un dato de la anterior) o las dependencias de control (saltos y ramificaciones), provocan que hagan falta ciclos de bloqueo para que el programa se ejecute correctamente. Además, la idea de que cada fase dure exactamente un ciclo de reloj no es muy realista, ya que por ejemplo, buscar los datos en memoria que no estén en la caché o realizar operaciones de división o multiplicación puede ser demasiado costoso para ejecutarse en un ciclo.

Ahora podemos ver la justificación de que utilicemos una filosofía *load-store* ya que los datos se leen de memoria en la fase 4, donde ya es demasiado tarde para que se pueda operar con ellos en la ALU. Además las instrucciones RISC de un tamaño constante nos proporcionan un flujo más o menos constante en la fase IF.

El procesador del proyecto imita el conjunto de instrucciones del MIPS I con algunas variaciones, pero no es un procesador encadenado. Una ejecución de este tipo complica en gran manera la tarea ya bien del programador, del compilador o de la unidad de control, que deben detectar y prevenir los fallos por dependencias.

# 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. En cada ciclo de ejecución las instrucciones deben modificar el estado interno del procesador (registros y memoria) y ejecutarse en una serie de dispositivos combinacionales simulados, tal y como lo harían en un procesador real
* 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

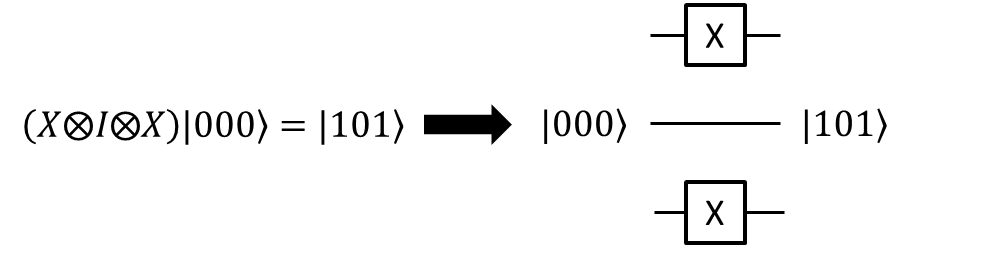
El programa del proyecto utiliza una implementación radicalmente distinta. Esta implementación trata de alejarse del principal problema de la anterior: matrices de un tamaño desmesurado.

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:

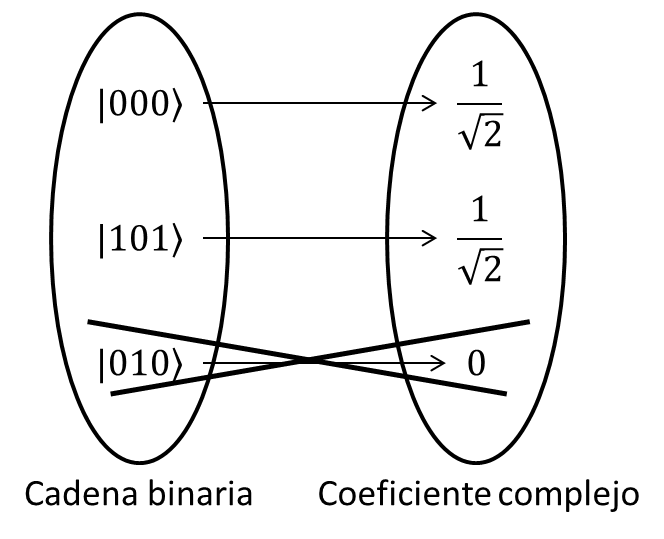


La situación es más complicada si tenemos un estado en superposición. En la representación matricial, un estado de este tipo quedaría representado como un vector con dos componentes o más componentes no nulas:

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.

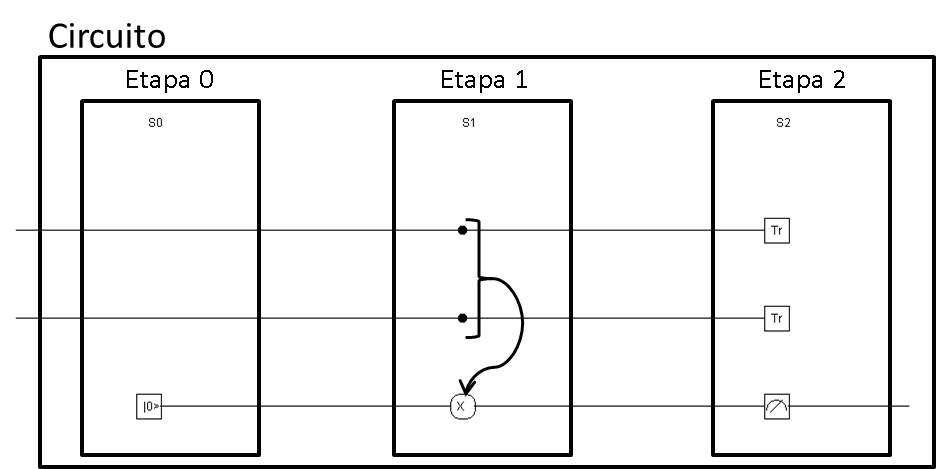
## 5.2 El simulador de circuitos cuánticos Qubit101

El punto de inicio del proyecto es el simulador de circuitos cuánticos Qubit101. Este simulador tiene implementado el motor de simulación de estados cuánticos que se presentó en la sección anterior.

Este simulador es, básicamente, un editor gráfico de circuitos cuánticos que es capaz de generar, a partir de dicha representación gráfica, una sucesión de puertas cuánticas que simulen el efecto del circuito.

Un circuito es en la implementación un *array dinámico de etapas*, junto con la lógica de edición necesaria.

Estas etapas son, a su vez, colecciones de puertas cuánticas que se ejecutan, al menos conceptualmente, al mismo tiempo. Un qubit de control afectará a todas las puertas controladas de su misma etapa:

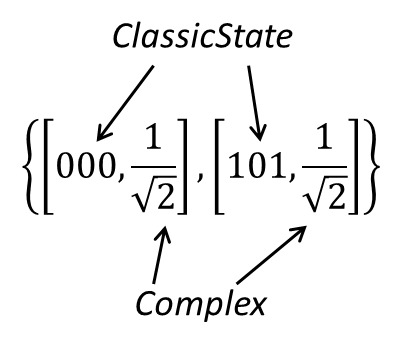


Estas estructuras están implementadas en las clases *Stage* para las etapas y *Circuit* para los circuitos:

[Imagen estructura de paquetes]

Las etapas además, tienen implementada la lógica de simulación en sus métodos *Simulate*, luego para ejecutar un circuito completo tan solo tenemos que ir ejecutando etapa a etapa pasando la salida de una a la entrada de la siguiente.

Esta simulación se realiza utilizando el esquema de mapa descrito en la sección anterior. Concretamente, las cadenas binarias *clave* son de la clase *ClassicState* y los coeficientes binarios  *valor*  son instancias de la clase *Complex*:



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 una forma muy parecida y tienen una plantilla que pueden extender. Esta plantilla encapsula la lógica para recorrer las componentes o utilizar qubits de control y el programador tan solo debe definir cómo actúa la puerta sobre cada componente.

La plantilla es la clase *UnitaryGateTemplate*. Una vez extendida solo habría que implementar el método abstracto *singleComponentOperatio,* que, por ejemplo, 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 del simulador Qubit101 será importado y 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 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 cómo 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 qué 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 se está 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*. Qué 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.



1. Las tareas recién despertadas no se ejecutan de inmediato sino que esperan en una cola a que todos los dispositivos anteriores hayan terminado.



1. Cada vez que una tarea concluya informa a la clase de sincronización que lleva una cuenta del número de ellas que están en funcionamiento.
2. Cuando no quedan más tareas en funcionamiento se activan las que estaban en espera que a su vez pondrán en la cola más tareas.
3. En el momento en el que no queden más tareas en 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 ejecutarán en el orden correcto y se podrá esperar si alguna lleva un tiempo muy largo. Además el 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 diseñado 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.

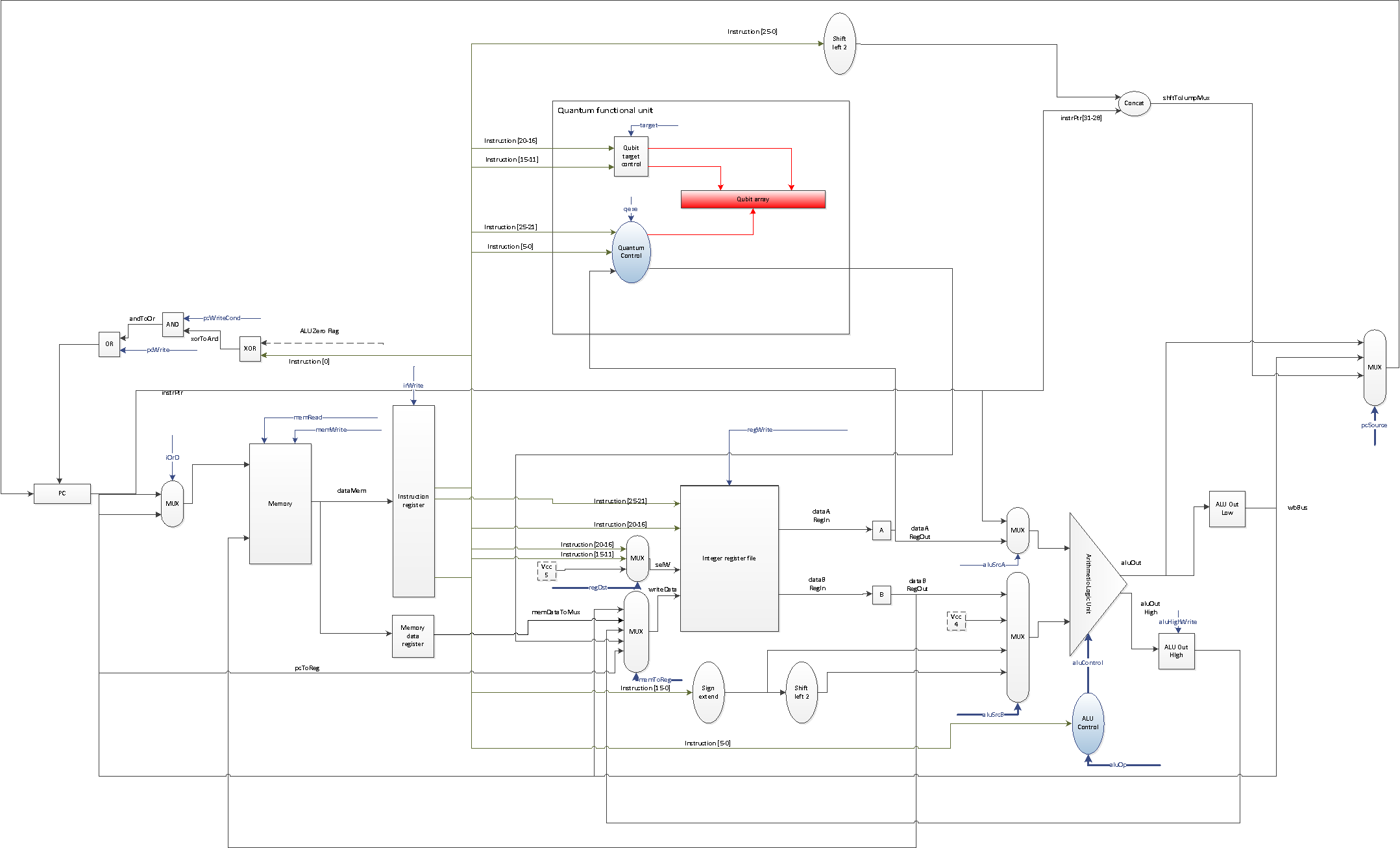
#### Arquitectura “física” del sistema

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:

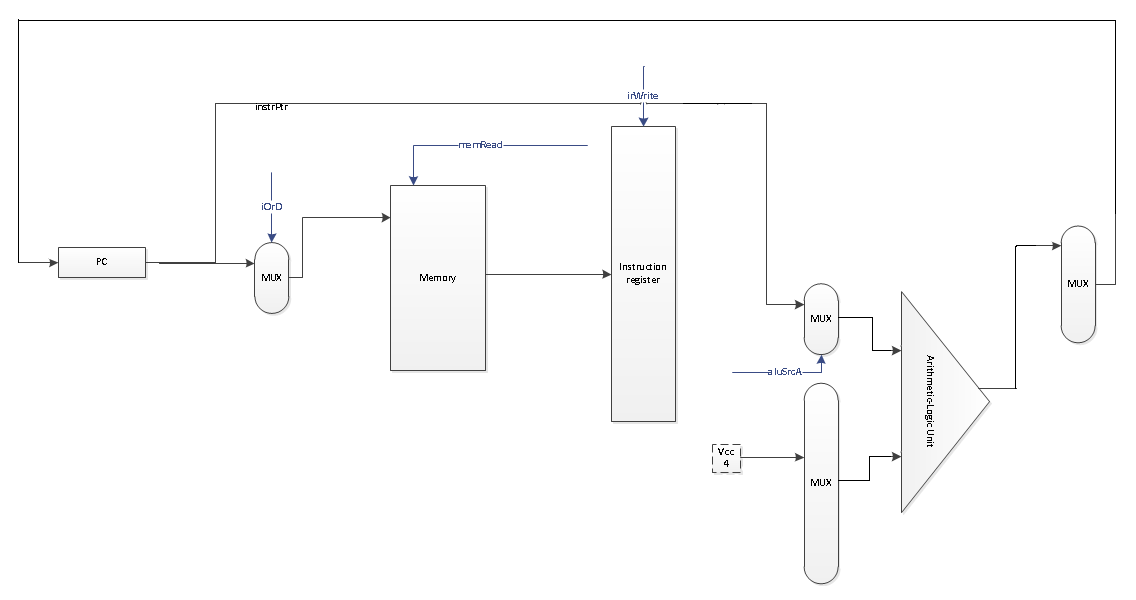


En esta figura las líneas verdes representan a la instrucción de memoria, las azules son señales de control, las rojas son operaciones cuánticas y el resto son negras.

#### Fases de la unidad de control:

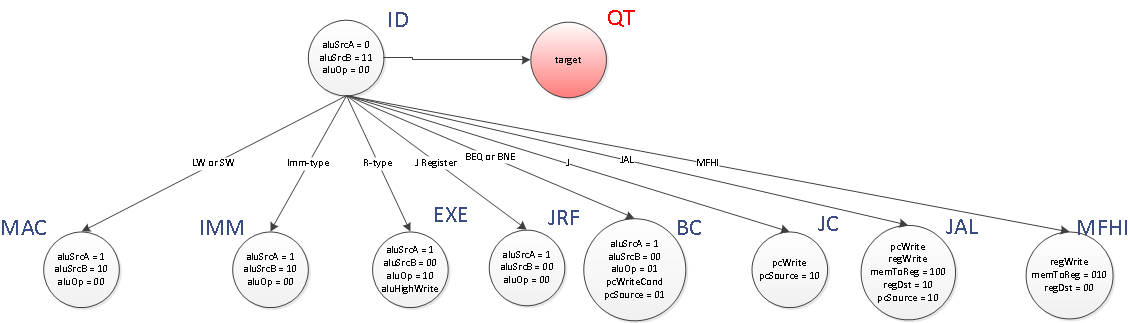
El procesador qMIPS, a diferencia del MIPS I real, no utiliza encadenamiento de instrucciones, pero tampoco una ejecución de tipo monociclo. Las instrucciones se segmentan tal y como si existiera encadenamiento pero ya que tenemos todo el hardware disponible para cada instrucción, podemos adecuar su longitud en ciclos a lo estrictamente necesario. Por tanto, la unidad de control está programada para seguir un árbol de estados provocando que cada instrucción realice tan solo las operaciones que necesita.

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 ciclo de ejecución se denomina fase ID (Instruction Decode). En esta fase la unidad de control debe decidir a partir del código de operación qué 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.

* Operaciones con memoria: La primera fase de este tipo de operaciones consiste en calcular la dirección efectiva donde se debe operar (Fase MAC). Para ello se indica a la ALU que sume el contenido del registro indicado con el dato inmediato que aparece en la instrucción. A continuación, según sea la instrucción de lectura (Fase MAR) o de escritura (Fase MAW) se activa la señal correspondiente de la memoria. Por último, si se trata de una operación de lectura, debemos escribir el resultado de la operación en el registro indicado (Fase MRC).



* Suma con operando inmediato: En su primera fase (Fase IMM) se ordena a la ALU que sume el dato inmediato de 16 bits de memoria con su signo extendido con el contenido del registro indicado. En la segunda fase simplemente se escribe el resultado en el registro correspondiente (Fase REW).



* Operación lógico-aritmética básica: Este tipo de instrucción es muy similar al anterior. Se ordena a la ALU que realice la operación indicada en los últimos 5 bits de la instrucción sobre los registros indicados (Fase EXE) y a continuación se escribe el dato en el fichero de registros (Fase RC).



* Saltos a dirección apuntada por registro: En la primera fase (Fase JRF) se ordena a la ALU que deje pasar el valor del registro indicado hacia el contador de programa. En el siguiente ciclo simplemente se escribe en dicho contador el valor correspondiente (Fase JRC).



* Salto condicional: Dado que la dirección efectiva del salto se calculó en la fase ID, en este ciclo (Fase BC) tan solo debemos comprobar utilizando la ALU si la condición del salto se cumple y escribir el contador de programa en tal caso.



* Salto incondicional: Igualmente que en el caso anterior la dirección efectiva del salto ya está calculada, así que en este caso tan solo debemos escribir sobre el contador de programa (Fase JC).



* Salto a subrutina: Idéntico al salto incondicional salvo que en este caso debemos indicarle al fichero de registros que escriba el puntero de instrucción actual en el registro 31, para poder realizar el retorno de subrutina (Fase JAL).



* Mover desde el registro de parte alta: En esta fase (MFHI) tan solo debemos mover el dato contenido en el registro de parte alta de la ALU hacia el registro indicado en la instrucción.



* Por último quedarían las operaciones de tipo cuántico. Estas fases se describen en detalle en la sección siguiente.



El árbol de estado completo de la unidad de control quedaría, por tanto, de la siguiente forma:



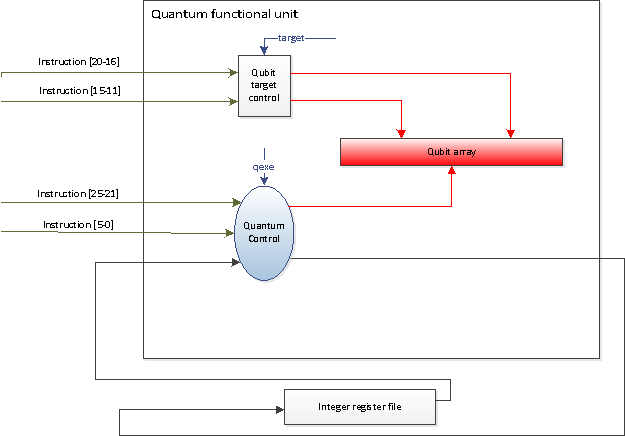
#### La unidad funcional cuántica

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 más 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 láser se encargan de enviar señales electromagnéticas adecuadas para operar con cada ión de la trampa.

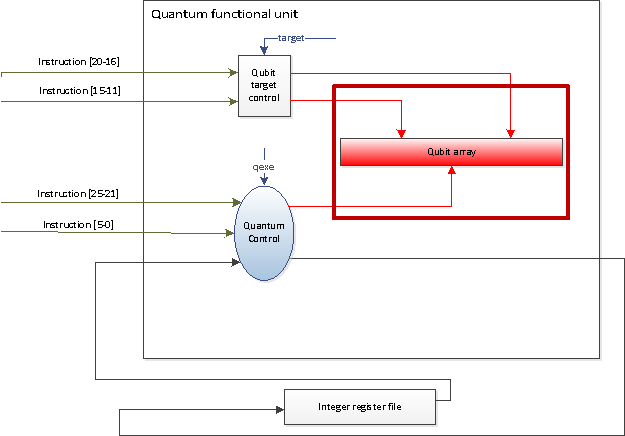
La imagen superior representaría una trampa de iones de 4 qubits.

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

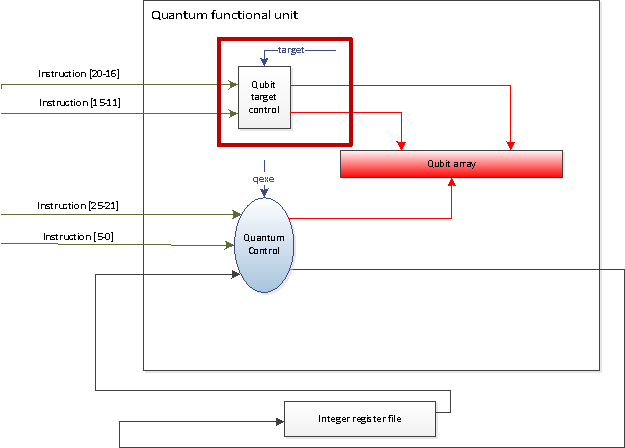


Consta de tres partes principales:

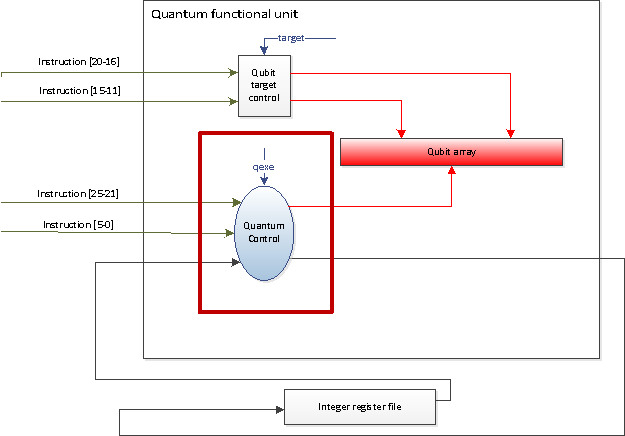
* Un *array* *de qubits* que en el caso del simulador contiene 32 qubits, cada uno de ellos se etiqueta con una ‘Q’ seguida de un número, de forma que para referirnos al primer qubit escribiremos Q0, Q1 para el segundo, etc. Hasta llegar al Q31. Cada uno de estos qubits sería un ión de la trampa.



* Un sistema de apuntado, que se encarga de desplazar el láser sobre el qubit correspondiente. Este sistema obtiene información sobre a qué 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 ordena 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 qué qubit es el objetivo. Los cables del 15 al 11 (otros 5) indican cuál es el de control. Si ambos 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-cuántica* 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 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 [Referencia 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 clásicas que acepta el compilador; Rd, Rs y Rt son registros, C representa un número entero con signo de 16 bits:

|  |  |  |  |
| --- | --- | --- | --- |
| **Instrucción** | **C. Op.** | **Func.** | **Resumen** |
| *Instrucciones Aritmético-Lógicas* | | | |
| add Rd, Rs, Rt | 0x0 | 0x20 | Suma de los registros Rs y Rt en Rd. Lanza una excepción si hay desbordamiento. |
| addu Rd, Rs, Rt | 0x0 | 0x21 | Suma de los registros Rs y Rt en Rd. Ignora el desbordamiento. |
| sub Rd, Rs, Rt | 0x0 | 0x22 | Resta de los registros Rs y Rt en Rd. Lanza una excepción si hay desbordamiento. |
| subu Rd, Rs, Rt | 0x0 | 0x23 | Resta de los registros Rs y Rt en Rd. Ignora el desbordamiento. |
| mult Rd, Rs, Rt | 0x0 | 0x18 | 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 y la parte alta se almacenara en el registro *High*. |
| div Rd, Rs, Rt | 0x0 | 0x1A | Realiza la división entera de Rs entre Rt en Rd. Almacena el resto en el registro *High*. |
| divu Rd, Rs, Rt | 0x0 | 0x1B | Realiza la división entera de Rs entre Rt en Rd. Almacena el resto en el registro *High*. |
| and Rd, Rs, Rt | 0x0 | 0x24 | Realiza la operación logica Y entre Rs y Rt en Rd. |
| or Rd, Rs, Rt | 0x0 | 0x25 | Realiza la operación logica O entre Rs y Rt en Rd. |
| xor Rd, Rs, Rt | 0x0 | 0x26 | Realiza la operación logica XOR entre Rs y Rt en Rd. |
| nor Rd, Rs, Rt | 0x0 | 0x27 | Realiza la operación logica O negada entre Rs y Rt en Rd. |
| slt Rd, Rs, Rt | 0x0 | 0x2A | Coloca un 1 en Rd si Rs es menor a Rt. |
| *Instrucciones con operando inmediato* | | | |
| addi Rd, Rs, C | 0x8 | - | Suma Rs y C en Rd. Lanza una excepción si hay desbordamiento. |
| *Operaciones con memoria* | | | |
| lw Rd, C(Rs) | 0x23 | - | Carga en Rd el contenido de la memoria en la dirección Rs + C. |
| sw C(Rd), Rs | 0x2B | - | Escribe en la dirección Rd + C de la memoria el contenido de Rs. |
| *Instrucciones de salto* | | | |
| jr Rs | 0x1B | - | Salta a la dirección almacenada en el registro Rs. |
| j C (o etiqueta) | 0x02 | - | Salta a la dirección especificada en C. |
| jal C (o etiqueta) | 0x03 | - | 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 (o etiqueta) | 0x04 | - | Si Rs y Rt son iguales, avanza o retrocede el número de instrucciones especificados en C. |
| bne Rs, Rt, C (o etiqueta) | 0x05 | - | Si Rs y Rt son distintos, avanza o retrocede el número de instrucciones especificadas en C. |
| *Excepciones* | | | |
| trap C | 0x1A | - | Lanza la excepción de código C. |
| *Especiales* | | | |
| mfhi Rs | 0x1C | - | Mueve el valor del registro *High* al registro Rs. |

Las operaciones cuánticas que acepta el compilador son las siguientes. Qt y Qc son qubit objetivo y de control respectivamente, si son el mismo la operación es no controlada:

|  |  |  |  |
| --- | --- | --- | --- |
| **Instrucción** | **C. Op.** | **Func.** | **Resumen** |
| *Operaciones cuánticas unitarias* | | | |
| qhad Qt, Qc | 0x0C | 0x00 | Puerta de Hadamard. |
| qx Qt, Qc | 0x0C | 0x01 | Puerta X de Pauli. Inversor. |
| qy Qt, Qc | 0x0C | 0x02 | Puerta Y de Pauli. |
| qz Qt, Qc | 0x0C | 0x03 | Puerta Z de Pauli. |
| qphs Qt, Qc, Rs | 0x0C | 0x10 | Cambio de fase especificado en el registro Rs. |
| *Operaciones cuánticas no unitarias* | | | |
| qmea Qt, Rs, S | 0x0F | 0x1A | Mide el qubit Qt y lo vuelca en Rs desplazado a la izquierda el valor de S, de 5 bits. |
| qrst Rs | 0x0C | 0x1B | Limpia el estado cuántico y vuelca en él contenido de Rs. |

Aunque con las instrucciones anteriores la máquina cuántica ya es universal, las siguientes dos instrucciones hacen mucho más sencillo la ejecución de ciertos algoritmos, sobre todo las subrutinas cuánticas:

|  |  |  |  |
| --- | --- | --- | --- |
| **Instrucción** | **C. Op.** | **Func.** | **Resumen** |
| *Operaciones cuánticas de control* | | | |
| qoff Rs | 0x0C | 0x1D | Utiliza como qubit 0 el qubit señalado por los últimos 5 bits del registro, es decir, si Rs contiene un 7, Q0 pasará a ser el que anteriormente era el Q7; el Q1 pasara al Q8, etc. Para deshacer los cambios simplemente hay que hacer qoff R0. |
| qcnt Rs | 0x0C | 0x1C | Marca el qubit apuntado por los últimos 5 bits del registro Rs como qubit de control, por ejemplo, si el el registro Rs contiene un 5, el qubit Q5 controlara el resto de operaciones cuánticas unitarias que sigan a esa, a no ser que el qubit objetivo sea también Q5. Se pueden marcar tantos qubits de control al mismo tiempo como se quiera. Para desmarcar uno de ellos, se repite la instrucción sobre el mismo qubit. |

El compilador, además, acepta dos directivas:

* *.word* A B: Almacena en la dirección de memoria A el dato B de 32 bits. Se utiliza para guardar datos en memoria antes de iniciar la ejecución.
* *.text* A: Tras esta directiva se deben colocar las instrucciones del programa. La primera instrucción estará en la dirección A de memoria.

## Manual de uso

El proyecto se compone de dos partes, el simulador de circuitos cuánticos Qubit101 y el simulador del procesador MIPS con funciones cuánticas qMIPS. A continuación se describen ambas herramientas a modo de manual de uso.

### Simulador de circuitos cuánticos Qubit101

La pantalla principal del simulador siguiente:



Se compone de un menú superior:



Una barra de herramientas:



Y una zona de trabajo que ocupa el resto de la pantalla.

#### Diseñando un nuevo circuito

Para empezar a diseñar un nuevo circuito, se debe pulsar el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\1.gif o el elemento del menú *Nuevo…*

Esto abrirá una ventana que nos pedirá que introduzcamos un nombre para el circuito así como el número de qubits de los que dispondrá (se pueden añadir o quitar posteriormente). Al aceptar se abrirá en el espacio de trabajo una nueva ventana con el circuito creado:



Los circuitos se organizan por etapas. En cada etapa se puede añadir una puerta cuántica por qubit y las etapas se pueden añadir o eliminar utilizando los botones **S+** y **S-**.

Para añadir una puerta debemos colocarnos sobre un qubit y una etapa con el ratón hasta que se muestre un pequeño recuadro punteado y hacer click, esto resaltara el qubit en la etapa seleccionada y nos permitirá añadir puertas cuánticas con los botones de la barra de herramientas.

Las puertas cuánticas unitarias simples de las que se dispone para formar circuitos son las siguientes:

* C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\hadamard.gif: Puerta de Hadamard
* C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\PauliX.gif: Puerta X de Pauli.
* C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\PauliY.gif: Puerta Y de Pauli.
* C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\PauliZ.gif: Puerta Z de Pauli.
* C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\Phase.gif: Puerta de cambio de fase. Para esta puerta se debe especificar un parámetro α para la fase.

Todas las puertas anteriores se pueden hacer controladas marcando la casilla antes de añadirlas al circuito. Para identificarlas como controladas se mostraran de forma circular en vez de cuadradas.

Pulsando el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\Control.gif se marcan uno o varios qubits como qubits de control que controlarán a todas las puertas controladas de su misma etapa.

Se dispone de tres puertas no unitarias:

* C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\Measure.gif: Realiza una medida en el qubit y la etapa seleccionadas.
* C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\AddQubit.gif: Añade un qubit auxiliar a la etapa seleccionada.
* C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\Trace.gif: Mide y descarta el qubit seleccionado en esa etapa.

Con estas últimas dos puertas se debe cuidar que al finalizar una etapa e iniciarse la siguiente el número de qubits debe coincidir. Si no coinciden el programa lo señalara con un aspa roja entre las etapas conflictivas:



Se pueden añadir o quitar qubits de cada etapa utilizando los botones **W+** y **W-**.

Por último, para eliminar una puerta cuántica basta seleccionarla y pulsar el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\RemGate.gif.

#### Guardando y cargando circuitos

El programa permite guardar los circuitos en formato QCML (*Quantum Circuit Markup Language,* extensión de XML). Pulsando el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\22.gif se guarda el circuito seleccionado y con el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\22b.gif se guardan todos los circuitos abiertos.

Para cargar un circuito se pulsa el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\52.gif y al seleccionarlo se abrirá una nueva ventana con el circuito guardado.

#### Utilizando circuitos cuánticos como puertas

El simulador permite utilizar un circuito guardado como si de una puerta cuántica más se tratase. Primero se debe cargar el circuito utilizando el botón … y el circuito seleccionado aparecerá en el menú desplegable:



Si se selecciona el circuito en dicho menú y se pulsa el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\circuitGate.gif se añadirá el circuito como puerta.

Dado que el circuito que se utilice puede tener más de un qubit, se debe cuidar que el circuito seleccionado quepa en el actual y no sobresalga por debajo.

Además, el circuito seleccionado puede crear o eliminar qubits en su interior y, por tanto, puede tener diferente número de qubits a la entrada y a la salida, así que habrá que adecuar el tamaño de las etapas donde se vaya a añadir.

El contenido del circuito añadido se puede ver haciendo doble click sobre él, lo que abrirá una nueva ventana con su contenido. Se debe tener en cuenta que lo que se abre es una copia del circuito anterior, así que si se guarda no se guardará en el interior del circuito contenedor sino en un archivo aparte.

Por último, el circuito añadido se puede marcar como controlado tal y como si fuera una puerta común marcando la casilla correspondiente. Los qubits de control de su misma etapa controlarán a todas las puertas unitarias contenidas en él.

#### Simulando los circuitos

La ventana de simulación se abre pulsando el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\play.png:



Si el circuito tiene errores de diseño la ventana no se abrirá.

Esta ventana muestra el circuito a simular en el marco principal, una barra de herramientas en la parte izquierda:



Una barra de entrada en la que se indica el estado inicial en binario, decimal o hexadecimal según la casilla que se marque:



Y la zona donde se mostrarán los resultados.

Existen dos tipos de simulación:

* C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\quick-play.gif: Realiza una simulación rápida, mostrando únicamente el resultado de las medidas.
* C:\Users\Jaime\Repositorios GIT\qMIPS101\Qubit101\src\icons\play.png: Realiza una simulación detallada mostrando las medidas realizadas así como la evolución del estado etapa a etapa.

Se debe tener en cuenta que si el estado es muy grande en alguna etapa al sistema le puede resultar muy complicado imprimirlo por pantalla y un estado tan grande rara vez es representativo para el usuario. En estos casos se recomienda utilizar la simulación rápida.

Una vez simulado el sistema los resultados aparecerán en las pestañas inferiores, particularizados para cada etapa. Nos podemos desplazar por las etapas con los botones C:\Users\Jaime\Desktop\56.gif y C:\Users\Jaime\Desktop\57.gif y la etapa seleccionada se irá mostrando en gris.

### Simulador del procesador cuántico qMIPS

La ventana principal del simulador QMIPS es la siguiente:



Tiene un menú superior:



Una barra de herramientas:



Una consola de información:



Y el espacio de trabajo.

El programa inicialmente arranca sin haber construido aun la circuitería del simulador, lo que en el programa se llama “Sistema”, luego el primer paso será construir dicho sistema pulsando el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\qMIPS\src\qmips\icons\build.png o el ítem del menú “*System -> Build*”. El simulador construirá el sistema e informará por la consola de información si todo ha ido bien. En ese momento mostrará en la ventana de trabajo todos los dispositivos relevantes para la ejecución.

#### Las vistas de los dispositivos

Cada dispositivo relevante tiene su propia vista de información:



Se muestran una serie de registros importantes del sistema:

* El registro de instrucción: que guarda temporalmente la instrucción que se ha traído de memoria.
* El contador del programa: que almacena la dirección de la siguiente instrucción que se debe traer de memoria.
* El registro de dato de memoria: que almacena temporalmente el dato que se trae de la memoria.

Los registros se muestran de la siguiente forma:

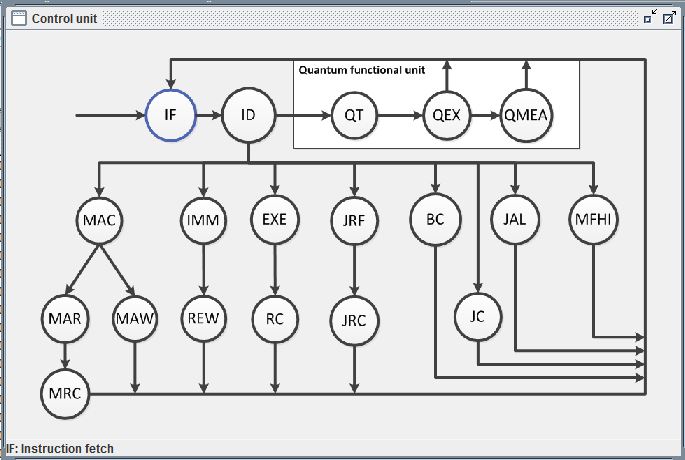


El valor del registro aparece en binario y entre paréntesis en decimal.

El fichero de registros tiene su propia vista que muestra el contenido de cada registro así como que registro está seleccionado a cada momento para leer:



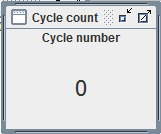
Para la unidad de control se muestra el árbol de estados completo. El estado actual del dispositivo aparecerá resaltado y señalado en la barra inferior de forma que sea sencillo seguir la ejecución de las instrucciones:



La unidad funcional cuántica muestra en un listado las componentes en superposición del estado interno de la unidad así como el *offset* actual y los qubits que están marcados como de control:



Por último, se dispone de una pequeña ventana que muestra el número de ciclos que se han ejecutado:



Las ventanas se pueden organizar automáticamente por la pantalla utilizando el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\qMIPS\src\qmips\icons\arrange.png. Si las ventanas no caben en la pantalla el programa las organizará en cascada para que el usuario las ubique como desee.

#### Cargando el código fuente

La memoria ahora mismo está vacía así que se debe cargar algún código fuente para que el simulador se ejecute. Para ello se pulsa el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\qMIPS\src\qmips\icons\load_source.png o el ítem del menú “*File -> Load source…*”. Podemos ejecutar el paso anterior y este de una vez utilizando el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\qMIPS\src\qmips\icons\build_load_source.png o el ítem del menú “*File -> Build and load source…”.*

Ahora el programa intentará compilar el código fuente a la memoria, si se produce algún error, el programa mostrará los errores en la consola de información y seguirá adelante, pero el programa podría no haberse cargado correctamente.

Ahora se mostrarán dos nuevas ventanas:

* La ventana de edición de código fuente: muestra el código fuente tal y como fue escrito. Permite editar el código directamente, guardarlo con el botón  y recompilar el código guardado con el botón . Además permite revertir las modificaciones realizadas hasta el documento guardado con el botón .



* La vista de progreso del programa: muestra una tabla con la dirección la operación y los parámetros de cada instrucción del código, resaltando la instrucción que se está ejecutando en cada momento para que sea más fácil seguir la evolución del programa.



#### Simulando el sistema

El programa permite simular el comportamiento del procesador utilizando los tres botones de la barra de herramientas siguientes:

* El botón C:\Users\Jaime\Repositorios GIT\qMIPS101\qMIPS\src\qmips\icons\play_one.png ejecuta un ciclo del programa cargado en memoria.
* El botón C:\Users\Jaime\Repositorios GIT\qMIPS101\qMIPS\src\qmips\icons\play_trap.png ejecuta el programa hasta que encuentra una excepción de programa. Típicamente se ejecuta hasta que se alcanza la instrucción “trap 0” que suele indicar el final del programa.
* El botón C:\Users\Jaime\Repositorios GIT\qMIPS101\qMIPS\src\qmips\icons\play_cycles.png muestra una pequeña ventana desplegable donde podemos indicarle el número de ciclos que queremos que ejecute.

Cada vez que se ejecute un ciclo la información de cada ventana se modificará según vaya cambiando la información del sistema en tiempo real.

Si queremos reiniciar el sistema, pulsaremos el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\qMIPS\src\qmips\icons\reset.png que dispara una señal de reset.

Durante la ejecución del programa pueden lanzarse varias excepciones, si esto ocurre el programa se bloquea en la excepción correspondiente hasta que el usuario pulsa el botón C:\Users\Jaime\Repositorios GIT\qMIPS101\qMIPS\src\qmips\icons\continue.png. Esto permite colocar varias instrucciones “trap 0” a modo de *puntos de interrupción* y ejecutar el programa de excepción en excepción.

# Conclusiones

## 6.1 Ejemplos

A continuación se exponen una serie de ejemplos significativos del funcionamiento del simulador.

### 6.1.1 El algoritmo de Deutsch

El algoritmo de Deutsch fue descrito en detalle en la sección [sección].

El algoritmo está implementado de forma que dependiendo de un índice en memoria llama a uno u otro oráculo. El resultado se almacenará en el registro R1 de la forma:

* R1 = 0: El oráculo es de tipo constante, es decir, o .
* R1 = 1: El oráculo es de tipo equilibrado, es decir, o
* R1 = -1: El índice de memoria no corresponde a ningún oráculo.

El primer paso será definir dicha posición de memoria y cargar el valor.

.word 0x200 1 //El número de la posición 200 indicará que oráculo

// se utiliza

.text 0

addi R2, R0, 0x200

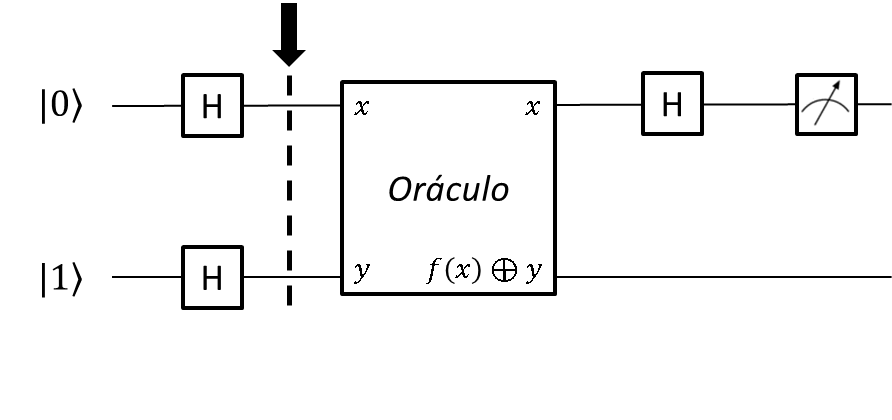
lw R2, 0(R2) //Se carga el índice de memoria.

qrst R0 //Limpiamos el registro cuántico

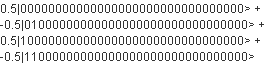
qx Q1, Q1 //Inicializamos el Segundo qubit a 1

qhad Q0, Q0 //Aplicamos la puerta de Hadamard a ambos qubits qhad Q1, Q1 // para crear la superposición.

El programa está imitando, básicamente, el circuito cuántico que representa el algoritmo de Deutsch. Hasta este momento hemos ejecutado:



Y el estado de la unidad cuántica en este punto es:



El siguiente paso sería realizar la llamada al oráculo correspondiente:

addi R3, R0, 0

beq R3, R2, const0 //El oráculo es f(x) = 0

addi R3, R0, 1

beq R3, R2, const1 //El oráculo es f(x) = 1

addi R3, R0, 2

beq R3, R2, ident //El oráculo es f(x) = x

addi R3, R0, 3

beq R3, R2, not //El oráculo es f(x) = not x

addi R1, R0, -1 //No existe una función con ese índice.

// devuelve error

trap 0

Ahora existen una serie de subrutinas que ejecutan cada oráculo:

* Para la función , la subrutina no tiene que hacer nada ya que el resultado será siempre cero y no tendrá que invertir el segundo qubit nunca:

|  |  |  |
| --- | --- | --- |
| **Índice** | **Código** | **Circuito** |
| 0 | const0:  j continue |  |

* Para la función , la subrutina debe invertir el qubit inferior siempre ya que el resultado es siempre 1:

|  |  |  |
| --- | --- | --- |
| **Índice** | **Código** | **Circuito** |
| 1 | const1:  qx Q1, Q1  j continue |  |

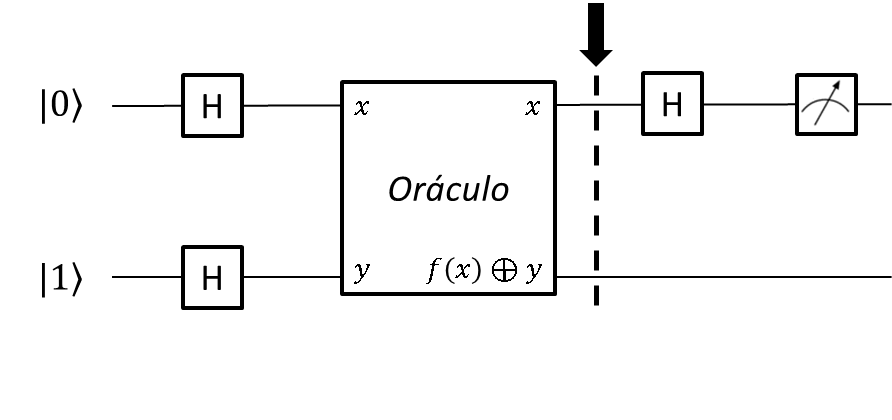
* En la función , habrá que invertir el qubit inferior si el superior vale 1, es decir, la operación CNOT:

|  |  |  |
| --- | --- | --- |
| **Índice** | **Código** | **Circuito** |
| 2 | ident:  qx Q1, Q0  j continue |  |

* La función , tendrá que invertir el qubit inferior, si el superior vale cero. Esto se puede conseguir negando el qubit superior, ejecutando un CNOT y volviendo a negarlo para que vuelva a su estado original:

|  |  |  |
| --- | --- | --- |
| **Índice** | **Código** | **Circuito** |
| 3 | not:  qx Q0, Q0  qx Q1, Q0  qx Q0, Q0  j continue |  |

Hemos ejecutado hasta ahora:



Solo queda ejecutar la última puerta de Hadamard y realizar la medida para obtener el resultado:

continue:

//La puerta de Hadamard final

qhad Q0, Q0

//Volcamos la medida del primer qubit sobre el registro R1

qmea Q0, R1, 0

trap 0

En el registro R1 quedará el resultado de la forma antes descrita con tan sólo una llamada al oráculo.

El algoritmo se ejecuta en entre 48 y 80 ciclos de simulación dependiendo del oráculo y en un tiempo real casi despreciable.

### 6.1.2 El algoritmo de Grover

El algoritmo de búsqueda de Grover fue descrito en detalle en la sección [sección].

En esta implementación concreta el algoritmo buscará simplemente un número concreto dentro de la colección de los número del 0 al 31 (5 qubits). En la dirección 0x600 de memoria se almacena el número que el algoritmo debe buscar y el oráculo correspondiente se genera automáticamente.

El primer paso es obtener el número a buscar de memoria:

//Este es el número que el algoritmo buscará en la colección

.word 0x600 7

.text 0

//Buscamos el dato en memoria

addi R5, R5, 0x600

lw R5, 0(R5)

La primera parte del algoritmo de Grover es colocar los qubits del espacio de búsqueda en superposición y aplicar una puerta de Hadamard a un qubit auxiliar en el estado :

//Limpiamos el registro cuántico.

qrst R0

//Aplicamos la puerta de Hadamard a los 5 qubits del subespacio de

// búsqueda para colocarlos en superposición.

qhad Q0, Q0

qhad Q1, Q1

qhad Q2, Q2

qhad Q3, Q3

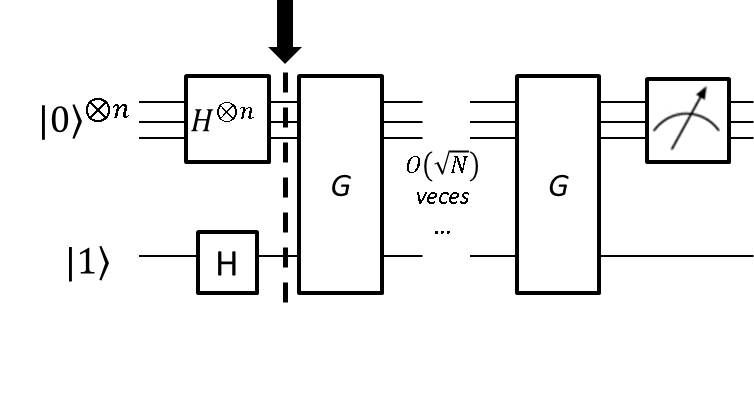
qhad Q4, Q4

//Ponemos el qubit auxuliar en el estado correspondiente

qx Q5, Q5

qhad Q5, Q5

Ya hemos inicializado el sistema para empezar a ejecutar el operador de Grover, estamos en el punto:



El programa debe saber cuántas iteraciones del ciclo del Grover son necesarias, se puede demostrar que para 32 datos hay que ejecutar iteraciones del operador de Grover:

//Contador de iteraciones a 5 en R2

addi R2, R0, 5

grover:

//Llamamos a la subrutina oracle que genera el oráculo correspondiente al // número elegido.

jal oracle

//Ejecutamos la inversión sobre la media: Hxn(Cambio de Fase)Hxn

qhad Q0, Q0

qhad Q1, Q1

qhad Q2, Q2

qhad Q3, Q3

qhad Q4, Q4

//La subrutina phase realiza el cambio de fase del operador

jal phase

qhad Q0, Q0

qhad Q1, Q1

qhad Q2, Q2

qhad Q3, Q3

qhad Q4, Q4

//Reducimos el contador de iteraciones y saltamos a la siguiente si no

// hemos acabado

addi R2, R2, -1

bne R2, R0, grover

La subrutina *oracle* genera un circuito que niega el qubit auxiliar si los otros cinco tienen el valor binario buscado, por ejemplo (el qubit superior es el menos significativo):

|  |  |  |
| --- | --- | --- |
| **Valor decimal** | **Valor binario** | **Circuito generado** |
| 5 | 00101 |  |
| 13 | 01101 |  |
| 28 | 11100 |  |

Funciona de la siguiente forma:

* Se mantiene en R7 un contador de desplazamientos, en R9 se almacena un 1 para utilizarlo de máscara y R8 guarda el número total de desplazamientos:

oracle:

addi R8, R0, 5

addi R9, R0, 1

add R7, R0, R0

* Se va desplazando el valor binario buscado a la derecha R7 veces y se enmascara con 1 para quedarnos con el bit menos significativo. Si este bit es cero, debemos ejecutar una puerta X sobre el qubit apuntado por R7. Al terminar deshacemos los cambios:

setnot:

srl R6, R5, R7

and R6, R6, R9

bne R6, R0, setisone

qoff R7

qx Q0, Q0

setisone:

addi R7, R7, 1

bne R7, R8, setnot

qoff R0

add R7, R0, R0

* Ahora se marcan todos los qubits menos el auxiliar como qubits de control, y se ejecuta la puerta X controlada por todos ellos sobre el auxiliar:

setcontrol:

qcnt R7

addi R7, R7, 1

bne R7, R8, setcontrol

qx Q5, Q5

* Finalmente, se repiten los pasos anteriores para que los qubits afectados dejen de ser de control y volver a aplicar la puerta X a los que la tuvieran:

add R7, R0, R0

unsetcontrol:

qcnt R7

addi R7, R7, 1

bne R7, R8, unsetcontrol

add R7, R0, R0

unsetnot:

srl R6, R5, R7

and R6, R6, R9

bne R6, R0, unsetisone

qoff R7

qx Q0, Q0

unsetisone:

addi R7, R7, 1

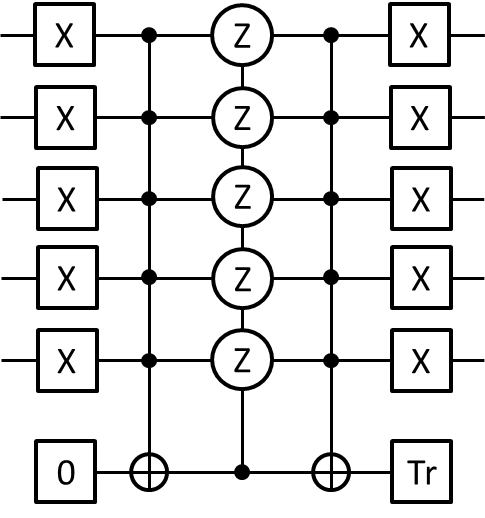
bne R7, R8, unsetnot

endgrover:

qoff R0

jr R31

La subrutina de cambio de fase (*phase*) ejecuta simplemente el siguiente circuito:



phase:

qx Q0, Q0

qx Q1, Q1

qx Q2, Q2

qx Q3, Q3

qx Q4, Q4

addi R20, R0, 0

qcnt R20

addi R20, R0, 1

qcnt R20

addi R20, R0, 2

qcnt R20

addi R20, R0, 3

qcnt R20

addi R20, R0, 4

qcnt R20

qx Q6, Q6

addi R20, R0, 1

qphs Q0, Q6, R20

qphs Q1, Q6, R20

qphs Q2, Q6, R20

qphs Q3, Q6, R20

qphs Q4, Q6, R20

qx Q6, Q6

addi R20, R0, 0

qcnt R20

addi R20, R0, 1

qcnt R20

addi R20, R0, 2

qcnt R20

addi R20, R0, 3

qcnt R20

addi R20, R0, 4

qcnt R20

qx Q0, Q0

qx Q1, Q1

qx Q2, Q2

qx Q3, Q3

qx Q4, Q4

qmea Q6, R20, 0

beq R20, R0, endphase

qx Q6, Q6

endphase:

jr R31

Lo último que queda en el algoritmo complete de Grover es medir los 5 qubits utilizados, volcándolos en el registro R1 bit a bit para obtener, con buena probabilidad, el número buscado:

qmea Q0, R1, 0

qmea Q1, R2, 1

or R1, R1, R2

qmea Q2, R2, 2

or R1, R1, R2

qmea Q3, R2, 3

or R1, R1, R2

qmea Q4, R2, 4

or R1, R1, R2

trap 0

# Futuras líneas

# Bibliografía

# Anexo A: Código fuente