Portada

Índice

[1. Introducción 3](#_Toc341885542)

[2. Antecedentes 4](#_Toc341885543)

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

[2.2 Conocimientos necesarios 4](#_Toc341885545)

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

[2.2.2 La arquitectura MIPS 12](#_Toc341885547)

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

[4. Objetivo 14](#_Toc341885549)

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

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

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

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

[6. Conclusiones 27](#_Toc341885554)

[7. Futuras líneas 28](#_Toc341885555)

[8. Bibliografía 29](#_Toc341885556)

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

# Introducción

# Antecedentes

## Estado de la cuestión

## Conocimientos necesarios

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

### 2.2.1 Física e Información cuántica

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

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

#### Los postulados de la física cuántica

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

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

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 .

Estos estados, como vectores que son, tienen su representación matricial:

Esta base se denomina *base computacional* y se usará a menudo por comodidad.

Si ponemos un bit lógico en contraposición a un qubit, podríamos hacer el símil:

0 lógico =

1 lógico =

Esto nos lleva a un hecho sorprendente, que es la base de la potencia de la computación cuántica. Para nosotros es relativamente fácil imaginar el estado , pensando por ejemplo en el espín de un electrón haciendo o en los niveles de energía del electrón como . Pero entonces, ¿qué significa algo del tipo ?, ¿está el espín del electrón arriba o abajo? La respuesta es que no está ni arriba ni abajo, sino en ambas al mismo tiempo, combinadas en los que se conoce como una *superposición cuántica de estados*. Esto nos permitirá, como veremos más adelante, ejecutar funciones para toda una gama de entradas al mismo tiempo en una sola operación, explotando lo que se denomina *paralelismo cuántico*.

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 que sean ficticios; podemos obtener, si bien no todos los resultados posibles de una función, sí una propiedad general de esta, como puede ser su periodo. Esto será fundamental para los algoritmos cuánticos más famosos, que superan ampliamente a las maquinas clásicas.

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

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

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

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

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. 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:

.”

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 .”

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

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

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

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

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

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

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

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

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

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

0

1

0

1

0

0

0

0

0

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

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

Necesitamos, como en la figura, un qubit auxiliar.

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

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

Si realizamos el producto interno de estas dos ecuaciones obtenemos:

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

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

a

a

a

No se puede realizar en una computación cuántica.

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

### 2.2.2 La arquitectura MIPS

# Gestión del proyecto

# Objetivo

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

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

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

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

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

# Desarrollo del proyecto

## Simulando los sistemas cuánticos

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

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

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

#### Representación matricial de los sistemas

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

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

En esta representación la computación:

H

X

se calcularía:

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

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

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

Si observamos las matrices del diseño anterior para un número más grande de componentes, como por ejemplo:

X

X

Observamos que la matriz está casi vacía. Las operaciones de las cuales podríamos decir que son más *clásicas* en cuanto a que es más fácil que las ejecute un computador clásico, suelen exhibir este comportamiento. El ejemplo superior representa tan solo dos inversores unos sobre el qubit superior y otro sobre el inferior.

Si olvidamos el punto de vista matricial, podemos ver por qué se da este comportamiento:

Comparándolo con una operación puramente cuántica:

Vemos que las operaciones clásicas no cambian el número de componentes en superposición cosa que si hacen las operaciones cuánticas.

Teniendo esto en cuenta, podemos trasladar la complejidad del sistema al tamaño de la superposición en vez de al número de qubits. Esto será siempre una mejora ya que dicho tamaño será siempre como mucho y en muchos casos será inferior.

Para implementarlo en el programa seguiremos los mismos puntos que en el caso anterior:

* Definimos los estados. En el último ejemplo se puede ver la estructura típica de un estado cuántico en computación cuántica: un coeficiente complejo asociado a un valor binario. Al ver este tipo de asociaciones, es fácil pensar en un mapa de valores que asocie componente (clave) a coeficiente (valor). Este mapa tendrá tantos elementos como componentes haya en la superposición, independientemente del número de qubits.

En la implementación, algo del tipo se denomina *estado clásico* (clase Java *ClassicState*) y un estado cuántico (clase *QuantumState*) contiene un mapa que asocia números complejos (clase *Complex*) a estados clásicos. Esta implementación del mapa delega en la clase de la API de Java *TreeMap* que utiliza un tipo de mapa en árbol muy eficiente (mapa rojo-negro).



Imagen 1 Clases de los estados cuánticos

Utilizando este mapa podemos definir toda la aritmética de los estados cuánticos. Sumar un estado cuántico otro es añadir a un mapa los elementos del otro, sumando los coeficientes si están en ambos estados y eliminando los que obtengan un coeficiente de 0. Multiplicar el estado por un escalar seria recorrer todo el mapa y multiplicar cada coeficiente por el escalar. Por ejemplo el estado:

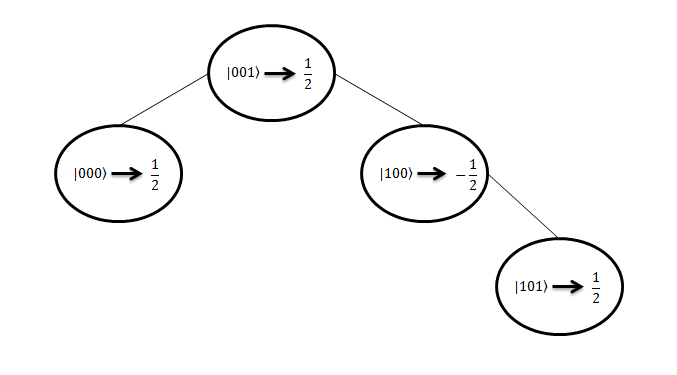


Imagen 2 Ejemplo de mapa de componentes

Quedaría como un árbol:

Ahora queda ver como actuarían las puertas sobre este tipo de estados.

* Las puertas cuánticas, en esta representación, son pequeños algoritmos que definen como actúan sobre cada componente en superposición. El sistema recorre el estado y aplica el algoritmo sobre cada componente, construyendo un nuevo estado resultante. Por ejemplo, el código de la puerta X sería:

Imagen Operacion de PauliXGate

Tan solo hay que definir esta función y una plantilla se encarga del resto. Todas las puertas cuánticas, unitarias o no unitarias se definen de este modo, aunque existe una plantilla para cada tipo.

El programa dispone de una serie de puertas cuánticas definidas:

* *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.

Estas puertas forman un conjunto universal, de hecho es suficiente con menos, pero todas ellas forman un conjunto cómodo para realizar algoritmos de cierta complejidad.



Imagen 4 Clases de las puertas cuánticas

Por último, quedaría definir como se forman estados de muchos qubits:

* Los estados de muchos qubits se crean simplemente introduciendo en el mapa un estado clásico de mayor tamaño. Incluso se dispone de una puerta para añadir más qubits al sistema si fuera necesario.

El proyecto base del trabajo importará el simulador y hará uso de las clases que se ha indicado.

## 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 Clases para definir dispositivos

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

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

Mostraré como ejemplo como construir un registro síncrono:

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

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

#### Sincronizando los dispositivos en ejecución

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

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

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

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

## Diseñando el procesador qMIPS

El procesador diseñando en el proyecto está basado en una versión simplificada del MIPS clásico sin encadenamiento de instrucciones, que utiliza una cantidad variable de ciclos por instrucción, presentado de forma educativa en el libro [HennessyPatterson].

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.
* Sólo se puede acceder a la memoria de palabra en palabra de 32 bits. No se pueden cargar o guardar otro tipo de datos.

#### Las instrucciones y el compilador

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

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

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

Los programas contienen una o varias directivas

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

* Instrucciones aritmético-lógicas:
  + add Rd, Rs, Rt: Suma de los registros Rs y Rt en Rd. Lanza una excepción si hay desbordamiento.
  + addu Rd, Rs, Rt: Suma de los registros Rs y Rt en Rd. Ignora el desbordamiento.
  + sub Rd, Rs, Rt: Resta de los registros Rs y Rt en Rd. Lanza una excepción si hay desbordamiento.
  + subu Rd, Rs, Rt: Resta de los registros Rs y Rt en Rd. Ignora el desbordamiento.
  + mult Rd, Rs, Rt: Multiplica Rs y Rt en Rd. Si Rs y Rt contienen más de 16 bits, el resultado puede ocupar más de los 32 bits que caben en Rd, se tendrán mostraran los bits menos significativos.
  + div Rd, Rs, Rt: Realiza la división entera de Rs entre Rt en Rd.
  + divu Rd, Rs, Rt: Realiza la división entera de Rs entre Rt en Rd.
  + and Rd, Rs, Rt: Realiza la operación logica Y entre Rs y Rt en Rd.
  + or Rd, Rs, Rt: Realiza la operación logica O entre Rs y Rt en Rd.
  + xor Rd, Rs, Rt: Realiza la operación logica XOR entre Rs y Rt en Rd.
  + nor Rd, Rs, Rt: Realiza la operación logica O negada entre Rs y Rt en Rd.
  + slt Rd, Rs, Rt: Coloca un 1 en Rd si Rs es menor a Rt.
* Operaciones con operando inmediato:
  + addi Rd, Rs, C: Suma Rs y C en Rd. Lanza una excepción si hay desbordamiento.
  + addiu Rd, Rs, C: Suma Rs y C en Rd. Ignora el desbordamiento.
  + ori Rd, Rs, C: Realiza la operación O entre Rs y C en Rd.
  + slti Rd, Rs, C: Coloca un 1 en Rd si Rs es menor a C.
* Operaciones de memoria:
  + lw Rd, C(Rs): Carga en Rd el contenido de la memoria en la dirección Rs + C.
  + sw C(Rd), Rs: Escribe en la dirección Rd + C de la memoria el contenido de Rs.
* Instrucciones de salto, aceptan tanto una dirección como una etiqueta en el código:
  + j C: Salta a la dirección especificada en C.
  + beq Rs, Rt, C: Si Rs y Rt son iguales, avanza o retrocede el número de instrucciones especificados en C.
  + bne Rs, Rt, C: Si Rs y Rt son distintos, avanza o retrocede el número de instrucciones especificadas en C.
* Excepciones:
  + trap C: Lanza la excepción de código C. La instrucción TRAP 0 finaliza el programa.
* Operaciones cuánticas; Qt y Qc son qubit objetivo y de control, si son iguales la operación es no controlada:
  + qhad Qt, Qc: Puerta de Hadamard.
  + qx Qt, Qc: Puerta X de Pauli. Inversor.
  + qy Qt, Qc: Puerta Y de Pauli.
  + qz Qt, Qc: Puerta Z de Pauli.
  + qphs Qt, Qc, A: Cambio de fase especificado por el parámetro A (alfa) de 5 bits.
  + qmea Qt, Rs, S: Mide el qubit Qt y lo vuelca en Rs desplazado el valor de S, de 5 bits.
  + qrst Rs: Destruye el estado cuántico y vuelca en contenido de Rs.

# Conclusiones

# Futuras líneas

# Bibliografía

# Anexo A: Código fuente