UNIVERSIDAD NACIONAL DE MAR DEL PLATA

TESIS DE GRADO

Emulador de algoritmos cuánticos en FPGA utilizando herramientas de diseño de alto nivel

Autor:
Agustin SILVA

Supervisor:
Dr. Omar Gustavo ZABALETA

Universidad Nacional de Mar del Plata
Facultad de Ingeniería
Departamento de Electrónica
Laboratorio de Sistemas Caóticos

23 de agosto de 2018

«*Busca por el agrado de buscar, no por el de encontrar...*»

Jorge Luis Borges

UNIVERSIDAD NACIONAL DE MAR DEL PLATA

Resumen

Facultad de Ingeniería

Departamento de Electrónica

Ingeniería Electrónica

Emulador de algoritmos cuánticos en FPGA utilizando herramientas de diseño de alto nivel

by Agustin SILVA

A pesar de los avances de los últimos años en materia de tecnología cuántica, uno de los campos de investigación en los cuales grandes empresas internacionales están invirtiendo una gran cantidad de tiempo y recursos, no existe hoy una computadora cuántica a gran escala. La ausencia de hardware cuántico dificulta el diseño de nuevos algoritmos. Por otra parte, una alternativa que permite avanzar en esta linea es la utilización de emuladores que permitan modelar los algoritmos existentes y diseñar nuevas alternativas. La principal ventaja de la computación cuántica por sobre la clásica es la posibilidad de procesar datos en forma paralela naturalmente. En tal sentido, resulta conveniente disponer de un método que emule el comportamiento de la forma más exacta posible. En este trabajo se propone un método que realice esta tarea de manera flexible aprovechando la tecnología FPGA (field-programmable gate array) y que reduzca los tiempos tanto de diseño como de procesamiento. Se utilizan herramientas del entorno Vivado® que permiten programar bloques a alto nivel que luego son sintetizados a código RTL. El potencial del emulador es puesto a prueba mediante la implementación de dos de los algoritmos de mayor importancia: la transformada cuántica de Fourier (QFT) y el algoritmo de búsqueda de Grover. Se describe con detalle cada paso del diseño y se estudia el rendimiento del circuito para diferentes casos.

Acknowledgements

A mi madre.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Las computadoras cuánticas prometen resolver problemas, que, de otro modo, serían intratables con las computadoras convencionales (Marinescu y Marinescu, 2005). Su importancia práctica surge de la aceleración exponencial en el cálculo de ciertas tareas algorítmicas sobre el cálculo clásico (Cleve y col., 1998). Por lo tanto, dejaron de ser interesantes sólo para físicos y matemáticos, para ser de interés de muchos otros investigadores tales como biólogos (Hagan, Hameroff y Tuszyński, 2002), economistas (Piotrowski y Sładkowski, 2017), ingenieros (Kumar y col., 2015; Arizmendi y Zabaleta, 2012; Zabaleta y Arizmendi, 2014; Zabaleta, Barrangú y Arizmendi, 2017), entre otros.

La superposición cuántica y el entaglement son las características principales que la computación cuántica aprovecha para superar el rendimiento clásico. Estas propiedades de las computadoras cuánticas son claramente una gran ventaja, pero un reto importante para poner en práctica a gran escala. El principal inconveniente de la computación cuántica es la decoherencia cuántica. El estado de coherencia, fundamental para una operación de ordenadores cuánticos, se destruye cuando es afectado por la interacción con el entorno. Como consecuencia, las necesidades físicas de manipular un sistema a escala cuántica son considerables en los dominios del superconductor, la nanotecnología y la electrónica cuántica. Aunque muchos líderes tecnológicos (IBM, 2017) están invirtiendo un gran esfuerzo en el desarrollo de ordenadores cuánticos con avances cada día más notorios, será necesario esperar un poco más para que una computadora cuántica sea capaz de reemplazar a un ordenador clásico en la realización de cálculos importantes. Esto no impide que otras lineas de investigación avancen en el área del software que aproveche al máximo el potencial de estas computadoras.

La construcción de algoritmos eficientes para resolver problemas clásicamente inmanejables es una tarea sensible y difícil en el campo de la computación cuántica. Además, en ausencia de una computadora física cuántica, es necesaria una tecnología alternativa para analizar el rendimiento y la sensibilidad a errores de circuitos cuánticos, y dónde implementar algoritmos heurísticos cuánticos diseñados. Muchos enfoques se han hecho en esa dirección, cada uno de ellos con ventajas y desventajas, (Khalid, Zilic y Radecka, 2004; Khalil-Hani, Lee y Marsono, 2015; Garcia y Markov, 2015). La cuestión principal a

tratar es el paralelismo cuántico, que no es posible a través de la simulación de software debido a la naturaleza secuencial de éstas. Por otro lado, los emuladores de hardware pueden imitar el paralelismo cuántico más de cerca, pero a expensas de una alta cantidad de recursos, por lo que es esencial en este caso la elección correcta de la arquitectura de hardware.

La mejor manera de describir los algoritmos cuánticos es mediante circuitos cuánticos. Del mismo modo que los circuitos convencionales, los circuitos cuánticos consisten en compuertas conectadas por canales que transportan información de un lugar a otro (Barenco y col., 1995). Debido a que los estados cuánticos crecen exponencialmente a medida que aumenta el número de bits cuánticos (qubits), es un gran reto producir una técnica eficiente para imitar a los operadores cuánticos en hardware clásico.

En este trabajo propongo un emulador de circuitos cuánticos que se centra en los diseños de algoritmos cuánticos que pueden aprovechar las capacidades de FPGA (computación paralela) con mínimos conocimiento de programación a bajo nivel (RTL ó register transfer language). Las herramientas de diseño de alto nivel son promovidas por los proveedores de las FPGA, tales como Vivado High Level Synthesis (HLS), proporcionado por Xilinx, OpenCL por Altera SDK, Stratus HLS por Cadence, Synphony C Compiler por Synopsys, etc., véase (Qin y Berekovic, 2015) y referencias en el mismo. En este caso se decidió trabajar con Xlinx Vivado® design suite (*Vivado Design Suite User Guide* 2016), que permite sintetizar código C/C++ a RTL (VHDL o Verilog), para diseñar un emulador de circuitos cuánticos. El diseño es evaluado con dos de los circuitos cuánticos más importante: el circuito QFT (Transformada Cuántica de Fourier) se elige para probar el emulador cuántico que estamos presentando, debido a su papel en algunos de los algoritmos más importantes de computación cuántica: la factorización de Shor, el hallazgo de fase y el logaritmo discreto. El otro circuito elegido para probar el emulador es el Algoritmo de Búsqueda de Grover, otro de los algoritmos cuánticos más importantes que reduce el tiempo de una búsqueda de n elementos desordenados.

Todas las medidas se realizan trabajando con la placa de desarrollo ZedBoard Zynq-7000 ARM/FPGA SoC Development Board from Xilinx que funciona con un sistema de chip modelo XC7Z020-CLG484-1 y un microprocesador Dual-core ARM Cortex^{TM}-A9.

El resto del trabajo está organizado de la siguiente manera: Sección 2 da una descripción de la mecánica y la computación cuántica. En la sección 3 se describen las herramientas principales con las que se trabajó. En la sección 4 el desarrollo realizado y su implementación en FPGA. En la sección 5 se analiza el rendimiento del diseño en la FPGA y se comparan los resultados con los mismos algoritmos implementados en el microprocesador integrado en la placa. Finalmente, la sección 6 concluye el trabajo.

En el marco de este trabajo final se presentó el artículo (Silva y Zabaleta, 2017).

Capítulo 2

Marco Teórico

2.1. Computación cuántica

Durante los últimos años la teoría de la mecánica cuántica ha encontrado un nuevo campo de aplicación en el ámbito de la información y la computación. La física cuántica permite codificar la información de una manera no-local clásicamente imposible, así como el procesamiento de información con una eficiencia que sobrepasa ampliamente a las computadoras clásicas (Galindo y Martin-Delgado, 2002). Algunas lineas en las cuales ha avanzado fuertemente la teoría cuántica de información son: El teletransporte cuántico, la codificación densa, y la criptografía cuántica. En lo que sigue de esta sección se resumirán los aspectos básicos y necesarios de la mecánica cuántica y luego daremos un recorrido por los algoritmos cuánticos que se implementarán en este trabajo.

2.1.1. El estado de un sistema cuántico

El estado cuántico es la descripción del estado físico de un sistema cuántico. De acuerdo con el primer postulado de la mecánica cuántica: El estado de cualquier sistema físico cerrado puede ser descrito por medio de un vector de estado v con coeficientes complejos y longitud unitaria en un espacio de Hilbert H, es decir un espacio lineal complejo (espacio de estados) equipado con un producto interno. Para describir modelos realistas de computación cuántica, estaremos interesados en grados de libertad para los cuales el estado es descrito por un vector en un espacio de Hilbert complejo finito. En particular, estamos interesados en sistemas compuestos por sistemas individuales de dos niveles. El estado de cada sistema de dos niveles se describe por medio de un vector en un espacio bidimensional de Hilbert. Para representar un vector de estado utilizaremos, de aquí en adelante, la notación de Dirac $|v\rangle$ donde v indica el autovalor correspondiente a cierto estado cuántico y se pronuncia 'ket v'. De este modo se definen $|0\rangle$ y $|1\rangle$ como los estados base del qubit, el equivalente cuántico del bit. Esto es análogo a lo que ocurre en computación tradicional donde un sistema biestable puede estar representado por la tensión en un capacitor de +5V y otro de 0V . De esta forma

podemos codificar un bit asignando, por ejemplo, el valor lógico '1' al estado en el cual el voltaje es +5V y '0' al estado en el cual el voltaje en el capacitor es 0V . Por lo tanto a la base $|0\rangle$, $|1\rangle$ para el estado de un qubit se la llama comúnmente la base computacional. Como ejemplos de sistemas cuánticos de dos estados podemos pensar en estados spin-up y spin-down de un electrón. Estos bits, tomados a partir del spin de las partículas reciben el nombre de qubits (bits cuánticos).

También, un electrón orbitando un núcleo, por ejemplo, puede describirse por un vector en un espacio bidimensional de Hilbert. Según la mecánica cuántica, los valores de energía que puede tomar el electrón están cuantizados, es decir que en lugar de poder tomar cualquier valor de energía, el electrón esta restringido a tomar solo valores de un conjunto discreto. Además, el electrón estará en general en el estado de más baja energía o con menor probabilidad en el primer nivel excitado, pero la cantidad de energía necesaria para excitar el sistema a los próximos niveles es tan alta que es muy poco probable que encontremos niveles de energía mayores al primer excitado. Para un caso como éste es posible ignorar, para fines prácticos, el subespacio que comprende los niveles mayores al primer estado excitado de energía, y por lo tanto tenemos un sistema descrito por un vector bidimensional en un espacio comprendido por los dos niveles más bajos de energía.

El estado de una partícula se determina a través de la asignación de una probabilidad, no podemos hablar de un estado 0 ó 1 claramente determinado para partículas cuánticas. Esta es la ventaja que tiene la computación cuántica respecto a la clásica: La lógica de un bit es 0 ó 1, mientras que un qubit contiene el concepto de ambos a la vez.

Para realizar cálculos no triviales son necesarios más de un qubit. Si tomamos por ejemplo dos bits, sus estados posibles son cuatro: 00, 01, 10, 11 y para representar estos estados son necesarios cuatro vectores ortogonales en cuatro dimensiones, formados por el producto tensorial entre $|0\rangle$ y $|1\rangle$.

$$|0\rangle \otimes |0\rangle, |0\rangle \otimes |1\rangle, |1\rangle \otimes |0\rangle, |1\rangle \otimes |1\rangle \tag{2.1}$$

Clásicamente, son necesarios cuatro pares de bits para representar la misma información que un solo par de qubits. En el texto se utilizará una nomenclatura más compacta para representar los estados de múltiples qubits,

$$|00\rangle, |01\rangle, |10\rangle, |11\rangle \tag{2.2}$$

El estado general de un qubit $|\psi\rangle$ es una combinación lineal pesada de los estados de la base, también llamada superposición de estados,

$$|\psi\rangle = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{2.3}$$

donde los pesos $\alpha, \beta \epsilon C$ son denominados amplitudes de probabilidad, por lo tanto deben satisfacer $|\alpha|^2 + |\beta|^2 = 1$. Consideremos el ejemplo del electrón orbitando: en general el electrón está en un estado que es combinación lineal entre el estado de menor energía y el primer estado excitado. El tercer postulado especifica que, siendo $|\psi\rangle$ el estado del sistema, la probabilidad de que al medir el qubit el estado sea $|0\rangle$ es $|\alpha|^2$ por lo tanto la probabilidades de que sea $|1\rangle$ es $|\beta|^2$. Cabe aclarar aquí un punto muy importante: Este estado ambiguo del sistema donde, con cierta probabilidad coexisten ambos estados ($|0\rangle$ y $|1\rangle$) tiene como condición que el sistema sea cerrado, es decir que no haya intervención del medio. Dicho de otra manera, si se mide el estado del sistema lo que se obtendrá como resultado es uno de los dos estados $|0\rangle$ ó $|1\rangle$. Se dice que el sistema colapsa hacia uno de los estados de la base. Esta característica difiere claramente de lo que ocurre en la mecánica clásica que establece por ejemplo que una moneda está en uno de los dos estados lógicos (cara ó cruz) antes de la medición y luego la medición lo que hace es revelar este hecho.

Por último, de acuerdo con el cuarto postulado de la mecánica cuántica, el espacio de estados de un sistema compuesto es el producto tensorial de los sistemas físicos que lo componen. Como ocurre en la ciencia de la computación clásica, un conjunto de n qubits forman un registro. Desde el punto de vista físico, si consideramos qubits numerados del 1 al n, y que el qubit i está en el estado $|\psi_i\rangle$, entonces el estado del registro cuántico esta dado por $|\psi_1\rangle \otimes |\psi_2\rangle \otimes ...|\psi_n\rangle$, aunque usualmente se escribe de manera abreviada $|\psi_1\rangle|\psi_2\rangle...|\psi_n\rangle$.

2.1.2. Evolución de un sistema

La dinámica de un sistema de partículas está determinada por la ecuación de Schrödinger 2.4

$$H\psi = i\hbar \frac{\partial}{\partial t}\psi \tag{2.4}$$

Si tenemos un problema de valor inicial $\psi(t = 0) = \psi_0$ podemos definir un operador $U(t)$ que cumpla

$$HU(t)|\psi_0\rangle = i\hbar \frac{\partial}{\partial t}U(t)|\psi_0\rangle \qquad y \qquad U(0)\psi = \psi \tag{2.5}$$

De esta forma se obtiene una ecuación de operadores $HU = i\hbar \frac{\partial}{\partial t}U$, cuya solución es $e^{-i\frac{H}{\hbar}t}$. Por lo tanto U es el *Operador de la evolución temporal* y satisface el criterio

$$|\psi(t + t_0)\rangle = U(t)|\psi(t_0)\rangle \tag{2.6}$$

2.1.3. Superposición cuántica

Si bien se ha mencionado esta propiedad de manera superficial, la superposición cuántica es un fenómeno que merece ser presentado con más detalle. En las computadoras clásicas, las señales eléctricas tales como voltajes representan los estados de 0 y 1 de un bit de información. En cambio, en computadoras cuánticas los estados de un electrón se pueden usar como un qubit. La orientación del spin hacia arriba o hacia abajo representa los dos estados 0 y 1 respectivamente. Usando qubits, las computadoras cuánticas pueden realizar operaciones lógicas y aritméticas como lo hacen las computadoras clásicas. Sin embargo, la diferencia importante es que un qubit puede representar la superposición de los estados 0 y 1 (Kanamori y col., 2006). Esta característica de las computadoras cuánticas hace posible la computación paralela de manera "natural". Como cada qubit representa dos estados al mismo tiempo, dos qubits pueden representar cuatro estados simultáneamente, de forma general, n qubits pueden representar 2^n estados. Por ejemplo, cuando usamos dos qubits que son la superposición de los estados 0 y 1 como entrada para una operación, podemos obtener el resultado de cuatro operaciones para cuatro entradas con solo un paso de cálculo, comparado con las cuatro operaciones necesarias en una computadora clásica.

2.1.4. Entaglement

La capacidad computacional de procesamiento paralelo de la computación cuántica, es enormemente incrementada por el procesamiento masivamente en paralelo, debido a una interacción puramente cuántica. Mientras el estado de un sistema clásico se puede especificar por medio de los estados de todos los sistemas que lo constituyen, en la teoría cuántica un sistema combinado puede tener propiedades adicionales, en cuyo caso los sistemas que lo componen se dice que están entangled o "enredados" unos con otros.

El entanglement cuántico ocurre cuando partículas tales como fotones, electrones e incluso pequeños diamantes (Lee y col., 2011) interactúan físicamente y luego son separados; el tipo de interacción es tal que cada miembro resultante de un par es descrito de manera apropiada por el mismo estado, que es indefinido en términos de factores importantes tales como posición, momento, spin, polarización, etc. Los estados entangled fueron discutidos por primera vez en el famoso trabajo de Einstein, Podolsky and Rosen (EPR), en el cual ellos intentaban demostrar la incompletitud de la mecánica cuántica (Einstein, Podolsky y Rosen, 1935).

Una buena medida para cuantificar cuán entangled está un sistema de dos qubits es una magnitud llamada Concurrencia. Partiendo del estado más general de dos qubits,

$$|\psi\rangle = \alpha|HH\rangle + \beta|HV\rangle + \gamma|VH\rangle + \delta|VV\rangle \tag{2.7}$$

se define $C = 2(|\alpha\delta - \beta\gamma|)$.

De acuerdo a este cálculo, solamente si C = 0 el estado es separable. Si C = 1 (su máximo valor), el estado está completamente entangled.

Supongamos un experimento en el cual se pretende medir el spin de dos electrones emitidos en direcciones opuestas decayendo a un único estado (single state) cuyo spin total es cero. Esto es, los electrones que conforman el estado tienen diferentes números cuánticos $s = +1/2$ y $s = -1/2$ y de esta manera el spin total del estado singlet es nulo. Para un estado como este la conservación del momento angular requiere que los vectores spin estén orientados en direcciones opuestas. El estado que describe esta situación es 2.8, que si se compara con el estado general 2.7 se tiene que $\alpha = \delta = 0$ mientras $\beta \neq 0$ y $\gamma \neq 0$. Si se decide por ejemplo medir el primer qubit se obtendrá aleatoriamente $|0\rangle$ ó $|1\rangle$ con probabilidad $|\beta|^2$ y $|\gamma|^2$ respectivamente. Pero lo interesante del asunto es que si el resultado de la medición resultó ser 0 sabemos con seguridad que una medición del otro qubit resultará 1 y de forma similar, si la medición en primer qubit resulta ser 1, la observación del segundo qubit, con seguridad, arrojará 0 como resultado. Pareciera ser que existe una "misteriosa conexión" entre las dos partículas, pero lo que es más impresionante aún es que experimentos específicos han probado que el fenómeno se mantiene incluso cuando los qubits de ϕ son trasladados a dos sitios distintos muy alejados. Recientemente investigadores de la ciudad de Viena han establecido el récord en teletransportación cuántica, logrando transmitir datos a través de una distancia aproximada de 144 km (Ursin y col., 2007).

$$|\psi\rangle = \beta|\uparrow\downarrow\rangle + \gamma|\downarrow\uparrow\rangle = \beta|01\rangle + \gamma|10\rangle \tag{2.8}$$

2.1.5. Compuertas cuánticas

El operador evolución temporal satisface la condición

$$U^{\dagger}(t)U(t) = e^{i\frac{H}{\hbar}t} \cdot e^{-i\frac{H}{\hbar}t} = 1 \tag{2.9}$$

Los operadores U que cumplen con $U^{\dagger} = U^{(-1)}$ son denominados *unitarios*. Debido a que la evolución temporal de un sistema cuántico está gobernada por un operador unitario y $U^{\dagger} = U(-t)$ se deduce que el comportamiento temporal de un sistema cuántico es *reversible*, siempre y cuando no se realice una medición, es decir que el sistema colapse.

Por lo expresado, la evolución temporal de los sistemas cuánticos está matemáticamente descrita por operadores unitarios. Las compuertas de los circuitos cuánticos asociados a los algoritmos cuánticos son operadores unitarios. Un circuito cuántico tiene la misma función que su contraparte clásico, es decir consta de una entrada, una salida y en medio de estas un canal por donde se transmite la información. En el camino, la información ingresada podrá ser modificada por medio de compuertas cuya tarea es transformar el estado

de los qubits a su entrada, el estado del sistema a la salida de la compuerta podrá ser parte de la entrada a otra compuerta, tal como ocurre en los circuitos clásicos. De hecho la señal siempre es modificada debido a que el canal de comunicaciones no es ideal. Por tal razón es común asociar las compuertas con operadores unitarios.

Los operadores en el espacio bidimensional de Hilbert se pueden representar con matrices 2×2. Un operador lineal queda completamente determinado por su acción sobre una base, veamos por ejemplo el caso de una del operador NOT que mapea el estado $|0\rangle$ a $|1\rangle$ y $|1\rangle$ a $|0\rangle$. Es más, por ser un operador lineal, mapea la combinación lineal de entradas a una combinación lineal de las salidas, o bien expresado de otra forma, la compuerta NOT mapea el estado general

$$\alpha|0\rangle + \beta|1\rangle \tag{2.10}$$

al estado

$$\alpha|1\rangle + \beta|0\rangle \tag{2.11}$$

En notación matricial:

$$NOT = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2.12}$$

Recordando que los estados de la base tienen su representación como vectores columna, veamos como actúa la compuerta sobre uno de estos estados:

$$NOT|0\rangle \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \equiv |1\rangle \tag{2.13}$$

La compuerta NOT también es conocida por su nombre en inglés "*flip-gate*" ó como *compuerta X*, una de las cuatro compuertas de *Pauli*. En computación clásica la compuerta equivalente es la compuerta *inversora*. La segunda compuerta que correspondería al *buffer* en computación clásica, ya que mantiene en la salida el valor de la entrada, es la compuerta I cuya representación matricial es la matriz identidad. Las otras dos compuertas que aquí presentamos, no tienen sus predecesores en la computación clásica, estas son Y y Z. En lo que sigue presentamos como actúan estas sobre una entrada arbitraria $|\psi\rangle = a|0\rangle + b|1\rangle$.

$$Y|\psi\rangle \equiv \begin{bmatrix} 0 & -j \\ j & 0 \end{bmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} -jb \\ ja \end{pmatrix} \equiv -jb|0\rangle + ja|1\rangle \tag{2.14}$$

Como se puede observar el efecto que causa Y sobre el estado de entrada es multiplicar por j e intercambiar las amplitudes de probabilidad, donde

$j = \sqrt{-1}$ es la unidad imaginaria. Por su parte Z genera sobre la entrada un cambio de fase. Es común recordar a esta compuerta como la que realiza un cambio de signo en el estado $|1\rangle$ de la base computacional.

$$Z|\psi\rangle \equiv \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} a \\ -b \end{pmatrix} \equiv a|0\rangle - b|1\rangle \tag{2.15}$$

La compuerta *Hadamard* H es una de las más utilizadas ya que aplicada sobre alguno de los estados puros de la base crea un estado en superposición uniforme,

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.16}$$

$$\begin{aligned} H|1\rangle &\mapsto \tfrac{1}{\sqrt{2}}(|0\rangle - |1\rangle), \\ H|0\rangle &\mapsto \tfrac{1}{\sqrt{2}}(|0\rangle + |1\rangle). \end{aligned} \tag{2.17}$$

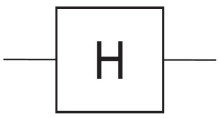


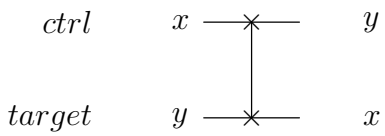
FIGURA 2.1: Compuerta de Hadamard

La compuerta cuántica R_k cuyo nombre en inglés es *Phase Gate*. Como su nombre hace intuir, esta realiza un cambio de fase, o dicho de otra forma, una rotación de fase y su aplicación sobre un estado arbitrario es la siguiente,

$$R(k)|\psi\rangle = \begin{bmatrix} 1 & 0 \\ 0 & e^{j\frac{2\pi}{2^k}} \end{bmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} a \\ e^{j\frac{2\pi}{2^k}} b \end{pmatrix} = a|0\rangle + e^{j\frac{2\pi}{2^k}} b|1\rangle. \tag{2.18}$$

Por último se presentan tres compuertas que son de mucha utilidad en los algoritmos cuánticos, la compuerta SWAP, la compuerta $CNOT$, la compuerta de rotación y la compuerta C-$\widehat{U}_f$, más información se puede adquirir en libros tales como (Leung y col., 1997)

La compuerta SWAP es una compuerta de dos qubits que intercambia el valor de dos estados cuánticos.



$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La compuerta Control NOT es una compuerta de dos qubits cuya representación matricial y su diagrama circuital son los siguientes.

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.19}$$

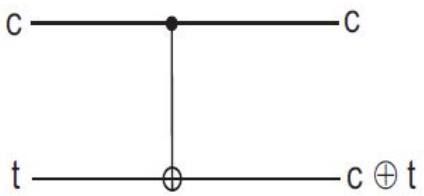


FIGURA 2.2: La compuerta NOT controlada cambia el estado de $|t\rangle$ solamente si la entrada de control $|c\rangle$ es $|1\rangle$

Esta compuerta actúa sobre dos qubits, el qubit de control y el qubit denominado target. La acción de $CNOT$ está dada por $|c\rangle|t\rangle \rightarrow |c\rangle|c \bigoplus t\rangle$ esto es, la compuerta realiza la operación NOT sobre el qubit target $|t\rangle$ si el qubit de control está en estado $|1\rangle$, en otro caso el estado de t no cambia.

Siendo U una compuerta que actúa sobre un solo qubit, la compuerta $Uctrl$ es nuevamente una compuerta de dos qubits con un qubit de control y un qubit target. Si se acciona el bit de control entonces U se aplica al qubit de target, en otro caso éste mantiene su estado, esto es $|c\rangle|t\rangle \rightarrow |c\rangle U^c|t\rangle$. En la figura 2.3 se muestra el diagrama circuital para la U controlada.

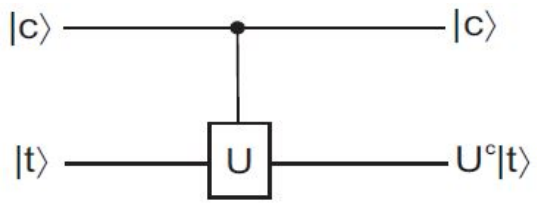


FIGURA 2.3: La compuerta U controlada cambia el valor de $|t\rangle$ a $U|t\rangle$ solamente si la entrada de control c se activa

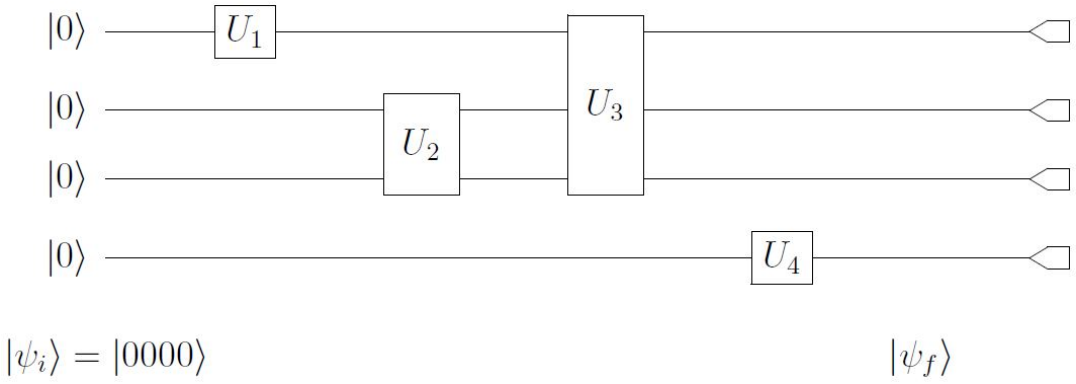


FIGURA 2.4: Modelo de circuito cuántico

2.1.6. Circuitos cuánticos

Un diagrama circuital que describa las secuencias de operaciones cuánticas y mediciones que intervienen en la descripción de algoritmos cuánticos complejos puede resultar muy eficiente. Las compuertas simples presentadas previamente pueden ser ensambladas de forma que formen un arreglo tipo red que nos permita realizar operaciones cuánticas más complicadas que las que pueden realizar cada una de ellas por separado (Deutsch, 1989). Un modelo de circuito cuántico es el que muestra en la figura 2.4. Como en el caso clásico, el circuito cuántico consiste en compuertas conectadas por "cables". Los qubits que ingresan desde la izquierda son transportados en el tiempo a través de estos cables y las compuertas, representadas por bloques rectangulares, van actuando sobre ellos.

En este ejemplo el estado de 4 qubits $|\psi_i\rangle = |0000\rangle$ entra en el circuito para ser procesado por las compuertas U_1, U_2, U_3, U_4. Como salida del circuito tenemos el estado de 4 qubits (posiblemente entangled) $|\psi_i\rangle$. Se realiza una medición qubit por qubit en la base computacional $|0000\rangle, |0001\rangle, .., |1110\rangle, |1111\rangle$, aunque en algunos casos puede que sea necesario hacer una medición conjunta. En muchos casos, en lo que se está realmente interesado no es el estado cuántico de salida, sino en la información clásica que indica qué resultado se produjo.

A continuación se presentan los algoritmos de interés junto con su modelo circuital en función de los conceptos y compuertas anteriormente descritas.

2.2. Transformada Cuántica de Fourier

La Transformada de Fourier es una transformación usada para convertir una señal del dominio del tiempo al dominio de la frecuencia, siendo muy importante para una gran variedad de procesos de ingeniería, por ejemplo, en las telecomunicaciones. Su versión discreta (ver Fig. 2.5), llamada Transformada Discreta de Fourier (DFT), tiene una implementación computacionalmente muy eficiente en la forma de la transformada rápida de Fourier (FFT).

2.2.1. Descripción del algoritmo

La Transformada de Fourier Cuántica es análoga a la DFT clásica. Es un algoritmo crucial debido a su implementación en otras aplicaciones prácticas con ventajas en cuanto a la aceleración frente a sus homologos clásicos (algoritmos de Shore, algoritmo de Hallgreen, algoritmos de logaritmos discretos, etc). Mediante la explotación de las ventajas del paralelismo cuántico, este algoritmo puede ejecutarse en mucho menos tiempo cuando se ejecuta en un ordenador cuántico.

Vamos a definir el vector de estado de entrada $|\Psi\rangle$ como:

$$|\Psi\rangle = \frac{1}{\sqrt{2^n}} * \sum_{j=0}^{2^n-1} f(j\Delta t) * |j\rangle \tag{2.20}$$

donde $f(j\Delta t)$, son los estados de amplitudes de probabilidad. Entonces, la QFT en $|\psi\rangle$ se define mediante la siguiente expresión:

$$QFT \rightarrow \frac{1}{\sqrt{2^n}} * \sum_{k=0}^{2^n-1} \sum_{j=0}^{2^n-1} f(j\Delta t) * e^{2\pi i(\frac{jk}{2^n}))} * |j\rangle \tag{2.21}$$

2.2.2. Modelo Circuital

Re-ordenando la ecuación 2.21 apropiadamente (Imre y Balazs, 2013), es posible construir el circuito QFT de la figura 2.6 en función de las compuertas cuánticas presentadas en las sección 2.1.5

$$\begin{aligned} QFT_{2^n}|j\rangle = \frac{1}{\sqrt{2^n}}(|0\rangle + e^{2\pi i 0.j_n}|1\rangle)(|0\rangle + e^{2\pi i 0.j_{n-1}j_n}|1\rangle)... \\ ...(|0\rangle + e^{2\pi i 0.j_1 j_2 ... j_n}|1\rangle), \end{aligned} \tag{2.22}$$

2.2.3. Matlab y simplificación

Utilizando el programa Matlab se estudió el comportamiento de cada una de las compuertas utilizadas para implementar la QFT. Se programó, en función de su composición matricial, scripts para cada una de las compuertas de

$$X(f) = \int_{-\infty}^{\infty} x(t)\, e^{-i2\pi ft} dt, \quad X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi(kn/N)}.$$

FIGURA 2.5: Izquierda: Transformada de Fourier. Derecha: Transformada discreta de Fourier.

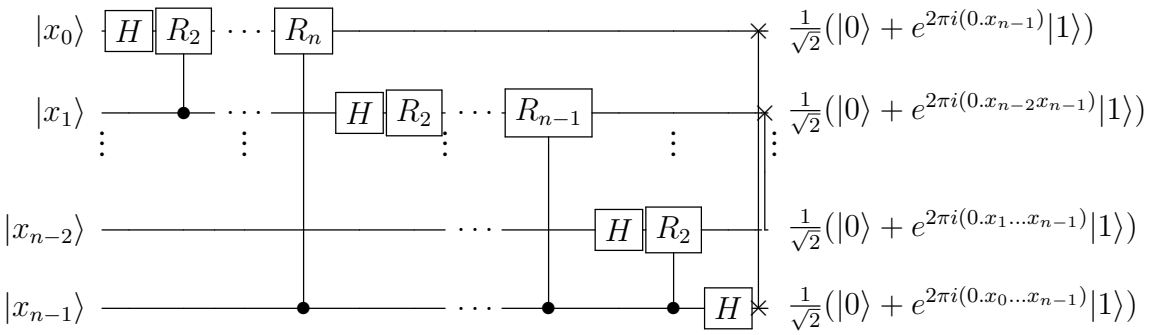


FIGURA 2.6: Circuito de QFT de N-qubits

manera genérica, es decir, para una cantidad de qubits N.

Para comprender los parámetros de entrada de las compuertas, ver fig 2.6. La compuerta Hadamard pide como parámetro: la cantidad de qubits del sistema y a cuál de los qubits va a ser aplicada la compuerta. La compuerta SWAP pide como parámetro: la cantidad de qubits del sistema, y los dos cables a los cuales esta compuerta está conectada. Finalmente la compuerta rotación pide como parámetro: la cantidad de qubits del sistema, el qubit al cuál será aplicada la compuerta, el qubit de control y un valor k, que indicara el angulo θ de giro, $\theta = \frac{2\pi}{2^k}$, (cada uno de los códigos se encuenta en el anexo, sección 7.1.1).

Finalmente se implementó el algoritmo de la QFT, aplicando las compuertas correspondientes en su debido orden y se verificó que el funcionamiento sea el deseado. El problema surge cuando se aumenta el orden de la cantidad de qubits de entrada, la cantidad de compuertas necesarias es $\frac{n(n+1)}{2}$, es decir, esta cantidad crece $O(n^2)$.

A la hora de simular, esto consume mucho tiempo de procesamiento, y a la hora de implementarlo, muchos recursos. Debido a que el objetivo del proyecto es poder emular la QFT para la mayor cantidad de qubits de entrada posible, se tomó la decisión de estudiar el comportamiento de todo el sistema completo. Es decir, se agruparon las compuerta y se encontró un sistema equivalente general que puede reemplazar al conjunto de compuertas por una única matriz (2.23) compleja de ($2^n x 2^n$):

$$M^{QFT} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & w & w^2 & .. & w^{N-1} \\ 1 & w^2 & w^4 & .. & w^{2(N-1)} \\ 1 & : & : & .. & : \\ 1 & : & : & .. & : \\ 1 & w^{N-1} & w^{2(N-1)} & .. & w^{(N-1)(N-1)} \end{bmatrix} \quad w = e^{\frac{2\pi i}{N}} \tag{2.23}$$

Se programó en Matlab dicho algoritmo y se verificó que su comportamiento sea exactamente igual al realizado anteriormente con todas las compuertas

por separado. Los tiempos de simulación en software disminuyen notablemente con éste cambio aparentemente trivial.

2.3. Algoritmo de Grover

Un típico problema informatico es el de identificar un elemento en un vector desordenado. El problema es de fácil solución, sin embargo la eficiencia es un factor que se debe tener en cuenta, ya que muchas veces puede ser una limitación. El objetivo de estos algoritmos es enviar una secuencia desordenada en un vector de tamaño n, una condición y que la salida sea un vector del mismo tamaño que la entrada tal que: Vector$_{Salida}$(i) = 1 si Vector$_{Entrada}$(i) cumple la condición y Vector$_{Salida}$(i) = 0 si Vector$_{Entrada}$(i) no cumple la condición.

En la informatica clásica se requieren $\frac{n}{2}$ consultas en promedio para encontrar un elemento en un vector de n elementos desordenados. En la computación cuántica, utilizando el algoritmo de Grover, se requiere una cantidad de consultas del orden de $\sqrt{n}$ o $\sqrt{2^N}$ (siendo N la cantidad de qubits). Si bien la mejora no es exponencial, como en la mayoría de los casos cuánticos, sigue siendo una mejora significante para un algoritmo tan utilizado.

2.3.1. Descripción del algoritmo

Las operaciones principales en el algoritmo de Grover son: inversión de fase e inversión sobre la media. La inversión de fase, invierte la fase de un estado de interés, la inversión sobre la media refuerza la diferencia entre el elemento de interés y los otros elementos en el vector.

Se calcula el valor medio de todos los elementos y se invierte en función de éste sin cambiar el valor medio del vector completo. Matemáticamente se puede describir como: $E^{'} = -E + 2 * M$, siendo M el valor medio, E un elemento genérico del vector, $E^{'}$ el futuro valor del elemento luego de la inversión.

Para que la probabilidad de encontrar el elemento deseado sea alta, el proceso debe iterarse $\frac{\pi}{4}\sqrt{n}$ veces. A continuación se describe el pseudicódigo del algoritmo de Grover.

1. Inicializar el sistema con N qubits en $|0\rangle$, esto es, $|\psi_{ini}\rangle = |00.,00\rangle \equiv |0\rangle^{\bigotimes n}$.
2. Aplicar la compuerta de Hadamard de $n \times n$ ($n = 2^N$) sobre ψ_{ini} para crear un estado $\psi_1 = \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle$ superposición de todos los estados de la base.
3. Aplicar la iteración de Grover G (inversión de fase e inversión de la media), un número $K = \frac{\pi}{4}\sqrt{n}$ de veces.

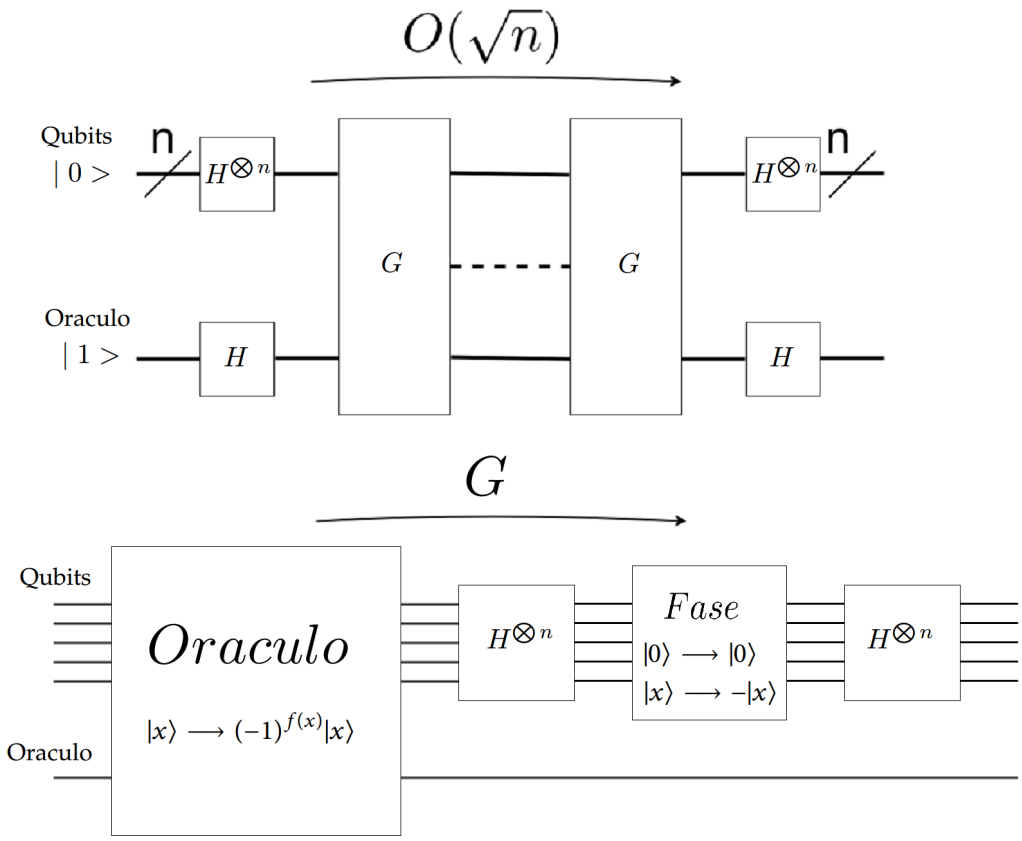


FIGURA 2.7: Circuito de Algoritmo de Grover

4. Medir el estado resultante.

2.3.2. Modelo Circuital

La representación circuital del algoritmo de Grover se puede observar en la figura 2.7. Los $n + 1$ qubits actúan como entradas a una compuerta especial, llamada Oráculo, una caja negra cuyo trabajo es "marcar"la solución del problema. Los n primeros qubits actúan como líneas de control y el restante es el objetivo.

El rol del módulo Oráculo es reconocer la solución a un problema de búsqueda particular en la operación de inversión de fase. Al monitorear el qubit oráculo, se puede detectar una solución al problema de búsqueda a través de los cambios de éste. El diseño del módulo de Oráculo varía con las diferentes aplicaciones de búsqueda.

Si bien este circuito se puede representar en su mayoría por las compuertas anteriormente explicadas, contiene la denominada compuerta "$Oráculo$", la cual no se puede representar de manera matricial. Por lo tanto, a la hora de

trabajar con el algoritmo de Grover se trabaja con el algoritmo en sí, y no con su representación circuital o matricial.

2.3.3. Matlab y simplificación

Como se mencionó anteriormente, no se puede trabajar por completo con el algoritmo de manera matricial debido a la imposibilidad de representar el oráculo. Por lo tanto, la parte correspondiente al oráculo será calculada con antelación, es decir, no se emulará su comportamiento, sino que se trabajará suponiendo la salida de esta compuerta como conocida.

Teniendo esto en cuenta, se simulo dicho algoritmo en matlab tomando como ejemplo un simple detector de números en una lista de n (2^N) elementos aleatorios, se envía como parámetro el tamaño de la lista (o la cantidad de qubits) y la condición, la devolución es uno de los valores que cumplen la condición y la probabilidad de error.

En un algoritmo clásico esto tomaría una cantidad de secuencias del orden de $O(\frac{n}{2})$, sin embargo con este algoritmo se puede encontrar el valor en una cantidad de iteraciones del orden de $O(\sqrt{n})$

Capítulo 3

Zedboard, Zynq7000 y Vivado

3.1. Placa Zedboard

El trabajo fue desarrollado utilizando la placa *ZedBoard Development Board*. ZedBoard Development Board es una placa de desarrollo para el SoC programable Xilinx Zynq-7000 (AP SoC). La placa requiere de como mínimo, dos cables USB y el cable de alimentación. Uno de los cables USB es utilizado para la programación del SoC y el otro para la comunicación entre la UART y un COM virtual. La placa, además, cuenta con las siguiente características:

- Xilinx Zynq-7000 AP SoC XC7Z020-CLG484
- Dual-core ARM Cortex™-A9
- Memoria Ram de 512 MB DDR3
- Programador USB-JTAG
- USB-UART y USB OTG 2.0
- 10/100/1000 Ethernet
- ADAU1761 SigmaDSP® Stereo, 96 kHz, 24-Bit Audio Codec
- ADV7511 225 MHz HDMI (1080p HDMI, 8-bit VGA, 128x32 OLED)
- Expansiones I/O PS y PL (FMC, Pmod, XADC)
- Memoria Flash de 256 MB Quad-SPI

En la figura 3.1 se muestra una imagen de la placa utilizada para el desarrollo del proyecto.

3.2. Xilinx Zynq 7000

La familia Zynq-7000 está basada en la arquitectura de los system-on-chip All Programmable de Xilinx. Estos productos integran, en un único dispositivo,

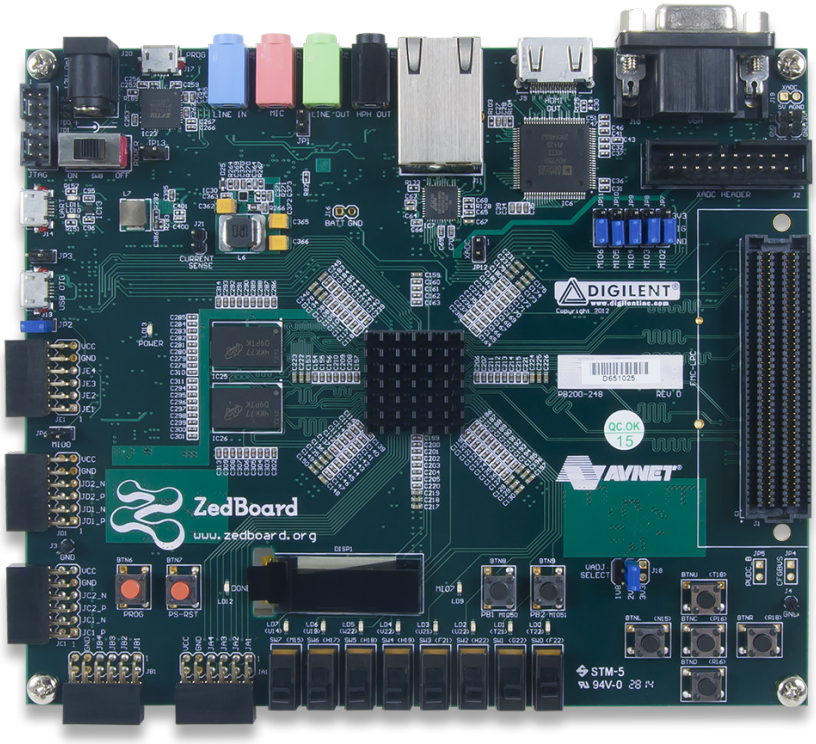


FIGURA 3.1: ZedBoard Zynq™-7000 Development Board

un sistema de procesamiento basado en un ARM® Cortex™-A9 doble núcleo y una lógica programable de Xilinx de 28nm. Los CPU del ARM Cortex-A9 son la base del sistema de procesamiento que también incluye: memoria en el chip, interfaces de memoria externa y un amplio conjunto de interfaces de conectividad periférica. Dando como resultado una herramienta extremadamente flexible y de alto rendimiento.

3.2.1. Caracteristicas generales

En la figura 3.2 se puede observar un diagrama general donde figura de manera separada el sistema de procesamiento (PS) y la lógica programable (PL). En el PS se encuentra: el procesador ARM Cortex-A9 doble núcleo, las memorias caches e integradas (On-Chip), el controlador DMA y un gran conjunto de interfaces y periféricos de entrada y salida. El PL, a su vez cuenta con bloque de lógica programable (CLB), bloques RAM, DSPs e I/O, conexión JTAG, un transceptor y dos ADC de 12 bit.

- **Descripción de las caracteristicas del PS:**
  - Unidad de Procesamiento de Aplicación: Incluye el procesador doble núcleo ARM Cortex-A9, las memorias chaché, on-chip, el controlador DMA, interruptores y timers.
  - Interfaces de Memoria: Incluye un controlador de memoria dinámica (DDR3, DDR3L, DDR2 y LPDDR2) y módulos de interfaz de

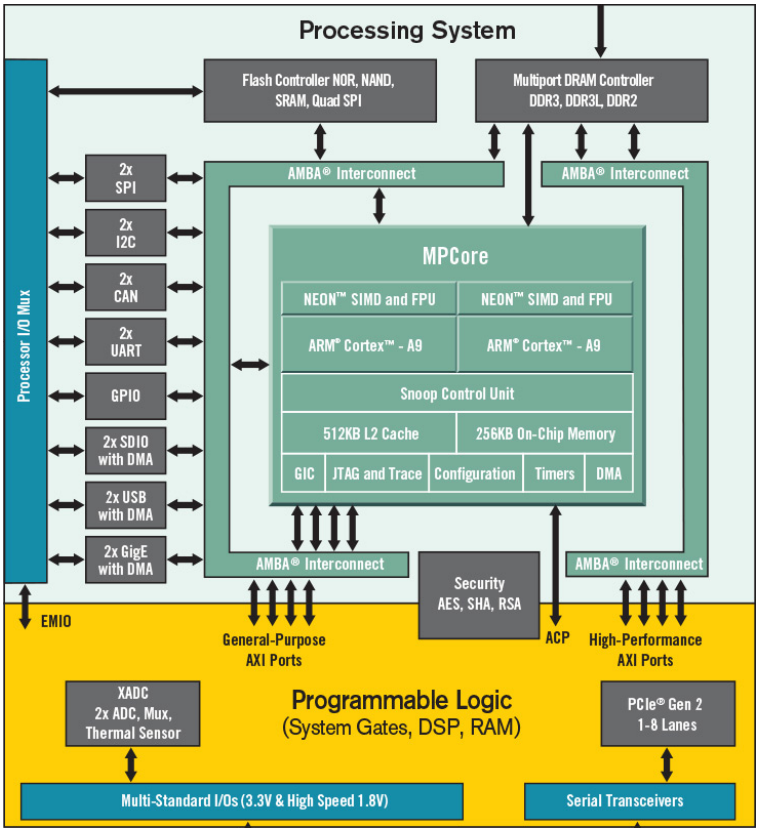


FIGURA 3.2: Diagrama de bloques del Zynq7000

memoria estática (flash NAND, flash Quad-SPI, un bus de datos paralelo y flash NOR).

- Periféricos I/O:
  - Dos periféricos MAC de tres modos de Ethernet.
  - Dos periféricos USB 2.0 OTG.
  - Dos SD/SDIO 2.0 con DMA incorporado.
  - Dos UART.
  - Dos interfaces maestro y esclavo I2C.
- Interfaces:
  - Externas:
    - Clock, reset, modo de arranque y referencia de voltaje.
    - Hasta 54 pines dedicados I/O multiusos (MIO), configurables por software para conectarse a los periféricos de I/O internos y controladores de memoria estática.
    - Memorias DDR2/DDR3/DDR3L/LPDDR2 de 32 o 16 bits.
  - Con el PL:

  - ◇ AMBA, interface AXI para comunicación de datos primarios.
  - ◇ DMA, interruptores y señales de evento.
  - ◇ Las I/O de multiplexado extensible (EMIO) permite a los periféricos PS no asignados acceder a la I/O PL.
  - ◇ Clocks y resets.
  - ◇ Configuración y varios: PCAP, XADC, JTAG, etc.

- **Descripción de las características del PL:**
  - Bloques de lógica programable (CLB): Ocho LUT por CLB configurables como RAM de 64x1 o 32x2 bit o registro de desplazamiento (SRL).
  - BRAM: Bloques RAM doble puerto con hasta 36 bits de ancho configurables como RAM de hasta 18Kb.
  - DSP: 18x25 multiplicadores con signo y sumador de 48 bits.
  - Bloques programables I/O: Compatibilidad con estándares comunes de I/O, incluidos LVCMOS, LVDS y SSTL de 1.2V a 3.3V con retardo programable incorporado.
  - Transceptores serie de baja potencia en dispositivos seleccionados.
  - XADC: Dos convertidores de 12-bit analógico/digital, voltaje y temperatura en el chip.

Esta diferenciación entre Hardware y Software permite una programación flexible y un trabajo por separado de ellos. Luego se debe trabajar en interrelacionar el sistema mediante la comunicación y sincronización de las partes. La posibilidad de poder trabajar con un microprocesador y una FPGA permite elegir como sintetizar un proceso, es decir, cuándo resulta óptimo sintetizar un proceso paralelizandolo (implementar en FPGA) o no (implementar en microprocesador). Por lo tanto, esto abre un abanico de posibilidades y un nuevo enfoque a la hora de realizar aplicaciones. En la figura 3.3 se puede ver una distribución más detallada de los elementos del Zynq-7000.

3.2.2. Interfaz AXI

La constante comunicación entre PS y PL y la dependencia de una sincronización precisa para trabajar, tanto con la FPGA como con el microprocesador, requieren de un estándar de comunicación entre ellos que facilite la relación entre los bloques pre-diseñados y/o los bloques creados por el usuario denominados propiedades intelectuales (IPCores). Este estándar es la interfaz AXI, una familia de buses que establece una comunicación tipo Maestro/Esclavo.

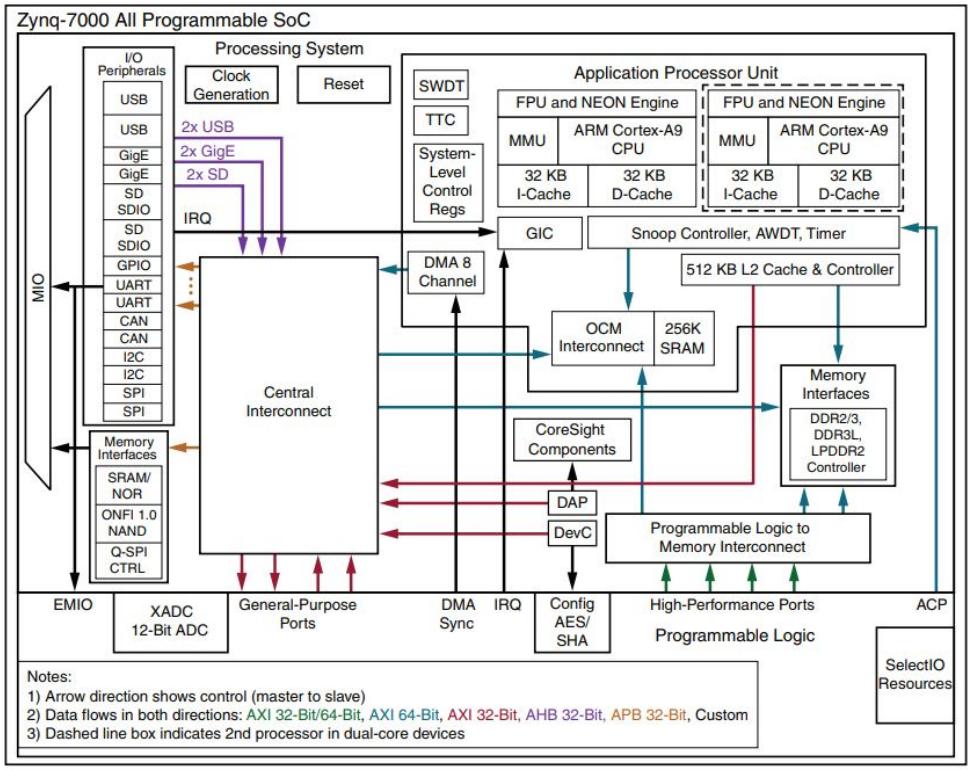


FIGURA 3.3: Diagrama detallado del chip Zynq7000

Existen tres tipos de comunicaciones AXI dependiendo de la aplicación y el flujo de datos en el bus:

- AXI4-Full: Funciona con mapeo de memoria con dirección única y datos múltiples. Hasta 256 datos de transferencia como ráfaga con una longitud de datos entre 32 a 1024 bits.
- AXI4-Lite: Funciona también con mapeo de dirección única pero con datos únicos. Utiliza menos recursos que AXI4 y trabaja con datos de una longitud entre 32 y 64 bits.
- AXI4-Stream: No trabaja con direcciones mapeadas para transmitir datos, unicamente con transmisión de datos en ráfaga lo cual resulta ventajoso en aplicaciones de transferencia continua de información. Longitud de datos ilimitada.

En los protocolos que utilizan mapeo de memoria, como AXI4-Full y AXI4-Lite, la información se mueve en ambas direcciones, tanto en la lectura como en la escritura. Al ser una transacción de tipo Maestro/Esclavo el maestro iniciará la comunicación y el esclavo esperará y responderá a ella. En las figuras 3.4 y 3.5 se puede ver el esquema de lectura y escritura para una comunicación de tipo AXI4-Full o AXI4-Lite. Sin embargo, la comunicación se comporta distinto para el caso de AXI-Stream donde las transmisiones ocurren en un único sentido como se puede ver en la figura 3.6.

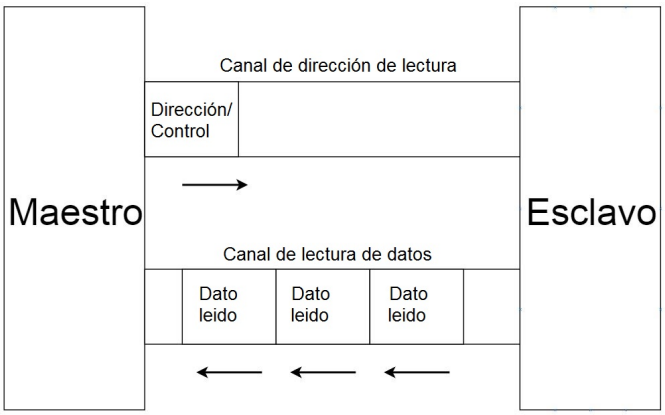


FIGURA 3.4: Esquema de lectura Maestro/Esclavo

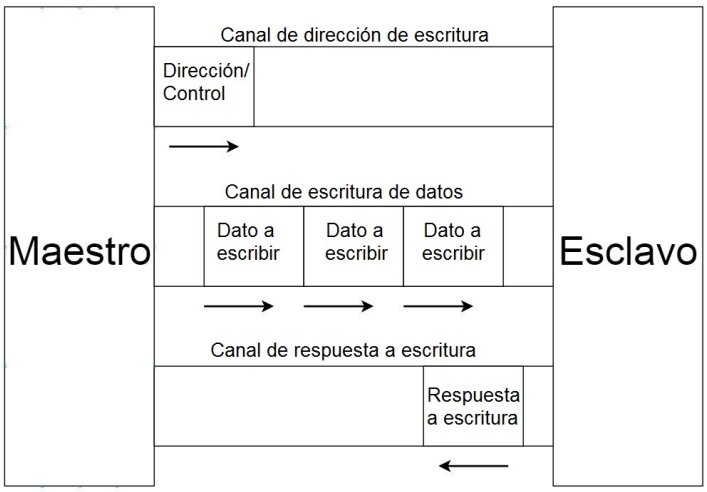


FIGURA 3.5: Esquema de escritura Maestro/Esclavo

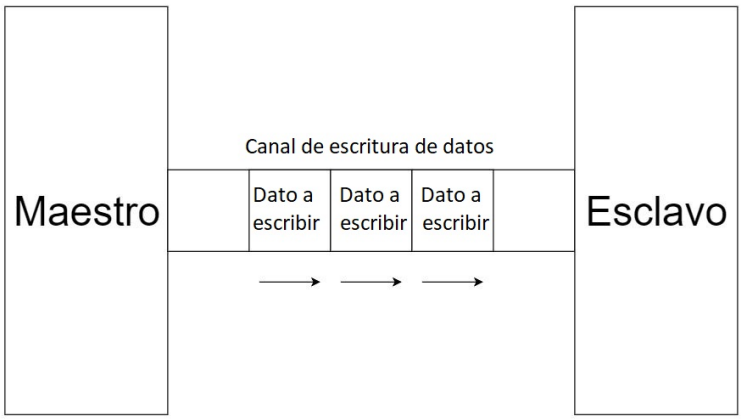


FIGURA 3.6: Esquema de escritura AXI-Stream

3.3. Paquete Vivado

El paquete de Vivado ofrece un conjunto de programas que permiten trabajar y manipular por completo las herramientas anteriormente descritas. Este paquete de programas es brindado por Xilinx y, a pesar de ser una herramienta relativamente nueva, cuenta con una numerosa variedad de opciones y le brinda al diseñador una gran libertad a la hora de tomar decisiones.

El paquete de Vivado incluye tres herramientas principales, los cuales fueron utilizados y serán descritos a continuación:

- Vivado HLS
- Vivado Design
- Vivado SDK

3.3.1. Vivado HLS

Una de las principales ventajas de trabajar con el paquete Vivado por sobre otros softwares, es que brinda la posibilidad de convertir un algoritmo, un bloque o una función escrita en lenguaje C (alto nivel), a lenguaje RTL (nivel de transferencia de registros, bajo nivel). Esto es posible gracias al programa Vivado HLS (High Level Sintesis), éste da la posibilidad de sintetizar funciones complejas en una FPGA sin la necesidad de trabajar con lenguajes como VDHL o Verilog, podemos abstraernos y pensar algoritmos en alto nivel, los cuales serán sintetizados luego para poder ser implementados en la FPGA y aprovechar sus ventajas.

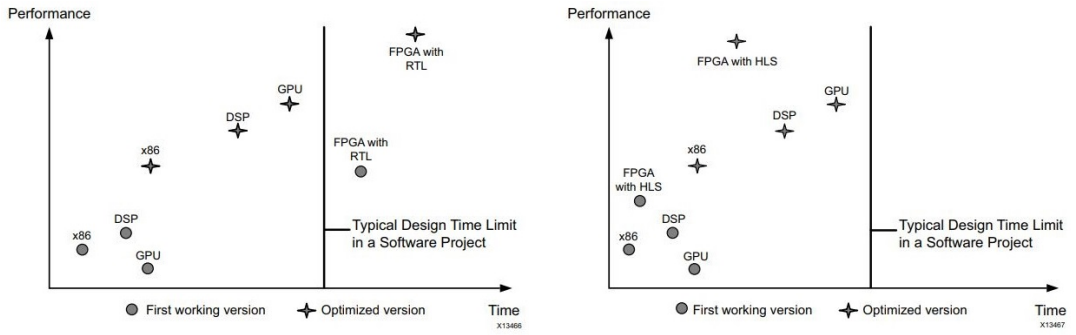


FIGURA 3.7: (a) Izquierda, características para diseño con RTL.
(b) Derecha, características para diseño con HLS.

Si bien el código es realizado en C, la programación no es cien por ciento libre, hay algunas restricciones y desventajas. A continuación se detallan algunas de estas limitaciones:

- En primer lugar, no se puede trabajar con memorias dinámicas, es decir, todos los arreglos deben ser estáticos. Esto se debe a que el sintetizador debe conocer el tamaño de los vectores o matrices con los que va a trabajar para una mayor optimización.

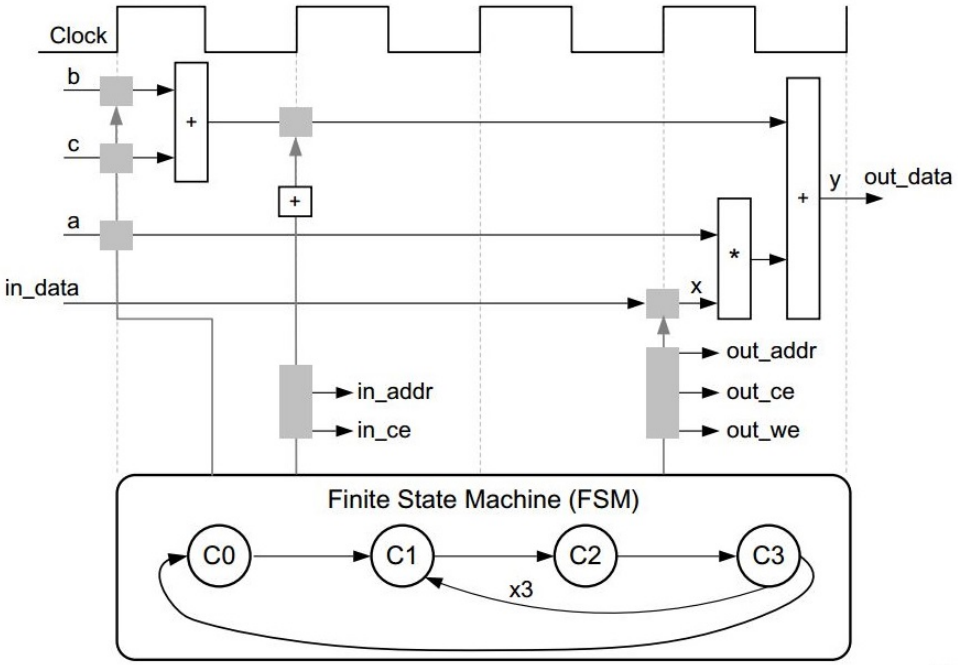


FIGURA 3.8: Creación de máquinas de estado

- En segundo lugar, se recomienda no utilizar funciones recursivas, ya que el sintetizador suele tener problemas, si bien esto no está completamente prohibido, es una recomendación evitar código recursivo.
- En tercer lugar, el RTL generado no es de fácil lectura, a veces se quiere tener más control sobre el código generado para realizar modificaciones, lo cual resulta complejo en Vivado HLS.

A pesar de las desventajas mencionadas, los puntos positivos, comenzando por la disminución en tiempo de diseño, son notables. Las ventajas a la hora de trabajar con una FPGA por sobre un microprocesador son conocidas: se puede obtener una mayor performance debido a su capacidad de procesar datos en paralelo. Pero como se ve en la figura 3.7 (a), en general, los tiempos de trabajo con RTL son altos, ésta es la principal razón por la cual no se trabaja en general con FPGA. Sin embargo, como se ve en la figura 3.7 (b) estos tiempos bajan trabajando con HLS, y podemos obtener una alta performance en un menor tiempo de trabajo una vez que aprendemos a dominar las herramientas de Vivado.

Si bien no se tiene un control completo sobre el código RTL (VHDL o Verilog), se puede orientar e indicar la manera en la que éste se va a implementar. Es decir, mediante instrucciones (directivas), dentro del código C que indicarán como se desea implementar cierta instrucción. Por ejemplo: dentro de un loop, se puede definir si se implementará mediante Pipeline, Unroll, Dependeces, etc. Se detallan las particularidades de cada una de estas técnicas más adelante.

La manera en la que Vivado HLS sintetiza el código C en RTL es mediante la creación de una máquina de estados a partir del algoritmo en C (Fig. 3.8).

```
int foo(char x, char a, char b, char c) {
  char y;
  y = x*a+b+c;
  return y
}
```

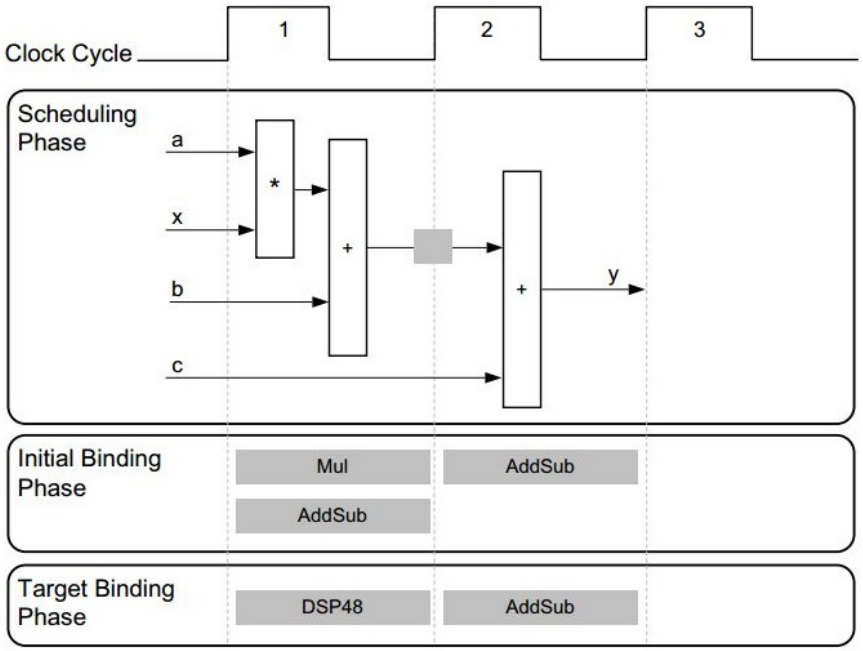


FIGURA 3.9: Binding

Luego en función de los recursos disponibles divide la máquina de estados en sub-funciones. Después, crea un flujo de control y lo mapea en ciclos de reloj en función de la placa y recursos utilizados (Scheduling). Por último, selecciona el Hardware que realizará cada una de las tareas dependiendo de la placa utilizada (Binding), ver figura 3.9. El código finalmente sintetizado puede ser en formato: Verilog, VHDL o SystemC.

El proceso de sintetización no es solo en función del código C, sino que también puede incluir directivas, como ya se mencionó anteriormente. Si bien incluir directivas es opcional, cumplen un rol fundamental en el diseño con Vivado HLS. Se pueden aplicar directivas de optimización a los siguientes objetivos: Interfaces, Funciones, Loops, Arreglos, Regiones.

Por último se muestra en la figura 3.10 el entorno Vivado HLS y se describen las principales partes de éste.

3.3.2. Vivado Design

En el software Vivado Design se realizan todas las conexiones del sistema. Esta etapa es de gran importancia, ya que todo los factores de sincronización dependerán de ella. En general, se inicia el proceso agregando el Zynq7000 en la pantalla, y luego, de manera visual, se agregan todos los bloques necesario para el funcionamiento adecuado del sistema. En la figura 3.11 se ve

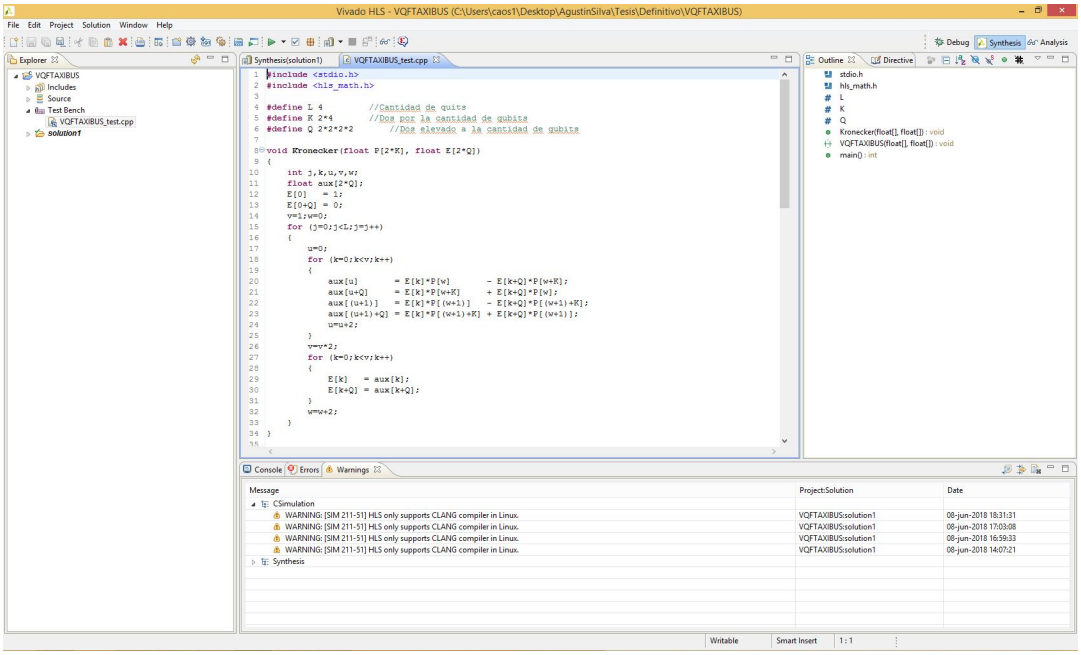


FIGURA 3.10: Ventana de trabajo en Vivado HLS. Izquierda: Librerías y explorador de archivos. Centro: Ventana de programación. Derecha: Lugar de selección de directivas. Abajo: Consola.

la ventana de trabajo del Vivado Design Suite, especificando las principales partes del sotfware.

Uno de los posible tipos de bloques son los generados previamente en Vivado HLS, los cuales habrá que conectar de manera adecuada con el resto del sistema para su correcto funcionamiento. Otra gran herramienta, es la posibilidad de crear inmediatamente bloques en código RTL (VHDL o Verilog). Aparte de los bloques anteriormente mencionados, el software cuenta con una gran cantidad de bloques que pueden ser de alta utilidad, tanto para el sistema en sí, como para la interconexión de las partes o asignación de entradas y salidas.

Luego del diseño de bloques (más su sincronización y sus controladores), se verifica su correcto funcionamiento, finalmente el programa sintetiza el sistema completo y crea el Bitstream. Esta síntesis lleva una gran cantidad de minutos y se le puede indicar al programa que realice cierta cantidad de síntesis distintas y elija la más optima en función de los indicadores con los que trabaja (tiempos, potencia y recursos utilizados).

3.3.3. Vivado SDK

Vivado SDK de Xilinx® es un entorno de desarrollo de software integrado para los procesadores integrados Xilinx. SDK se basa en la plataforma de código abierto Eclipse. El software SDK incluye las siguietes caracteristicas:

- Editor de código C/C++

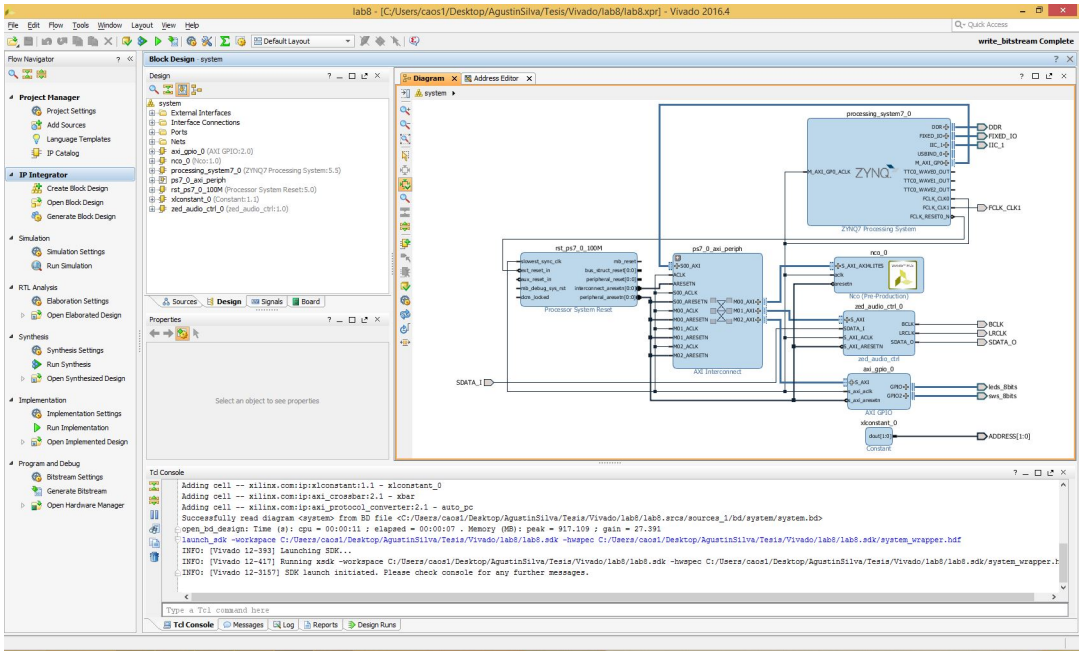


FIGURA 3.11: Ventana de trabajo en Vivado Design. Izquierda: Etapas de trabajo para la sintetización. Centro: Explorador de bloques. Derecha: Lugar de diseño. Abajo: Consola.

- Configuración de compilación de aplicaciones y generación automática de Makefile
- Debugeador de sistemas Xilinx (XSDB).
- Programador de FPGA (bitstream).
- Generador de gestor de arranque simple.

A pesar de no ser un sistema operativo de tiempo real, contiene todos los drivers, tanto de Xilinx como los creados por el usuario. Aquí se programará el microprocesador, ARM Cortex-A9 en este caso, en el cual se podrá proporcionar como función los distintos bloques asignados en el Vivado Design, incluyendo los bloques creados por el usuario en el software Vivado HLS. También se puede asignar manualmente punteros a direcciones de memoria conocidas (BRAMs, por ejemplo) y trabajar con ellas como si fuesen arreglos desde el punto de vista del software.

En la figura 3.12 se ve la ventana principal de trabajo del SDK indicando las principales partes del mismo.

3.3.4. Modalidad de trabajo en Vivado

A continuación se describe, de manera resumida, modo de trabajo con este entorno de desarrollo.

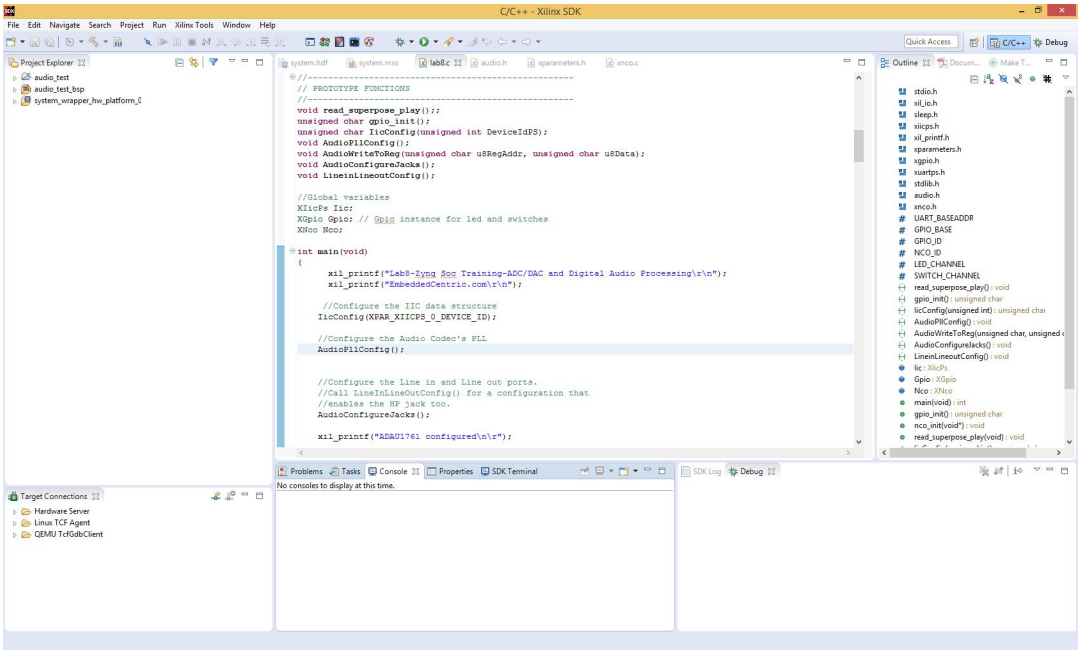


FIGURA 3.12: Ventana de trabajo en Vivado SDK. Izquierda: Explorador de archivos. Centro: Ventana de programación. Abajo: Consola.

En la figura 3.13 observamos un diagrama de los pasos a seguir, tanto los obligatorios como los opcionales, para el desarrollo de un IPCore, comenzando por el código C/C++ y las posibles directivas. Luego, utilizando los denominados Test Bench, se realiza una simulación del código de software. Después de comprobar el funcionamiento del algoritmo, se sintetiza a RTL y se realiza una co-simulación, la cual será una simulación de nuestro algoritmo pero en su versión ya sintetizada (RTL). Finalmente, una vez corroborado su funcionamiento, se exporta el bloque IPCore diseñado hacia el programa Vivado Design Suite.

Por último, en la figura 3.14 se puede ver un diagrama con el funcionamiento del Vivado SDK, desde la creación e importación en el Vivado Design Suite; la creación del código de la aplicación, la compilación y el posible debugeo. Este procedimiento se debe repetir tantas veces como se desee hasta que el rendimiento del sistema sea el esperado.

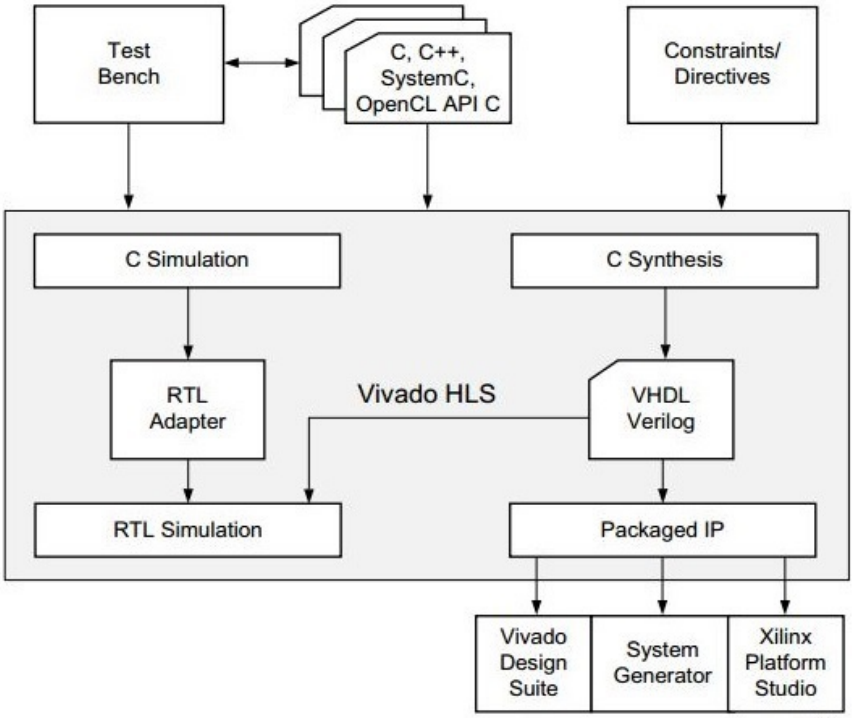


FIGURA 3.13: Esquema de trabajo en Vivado HLS

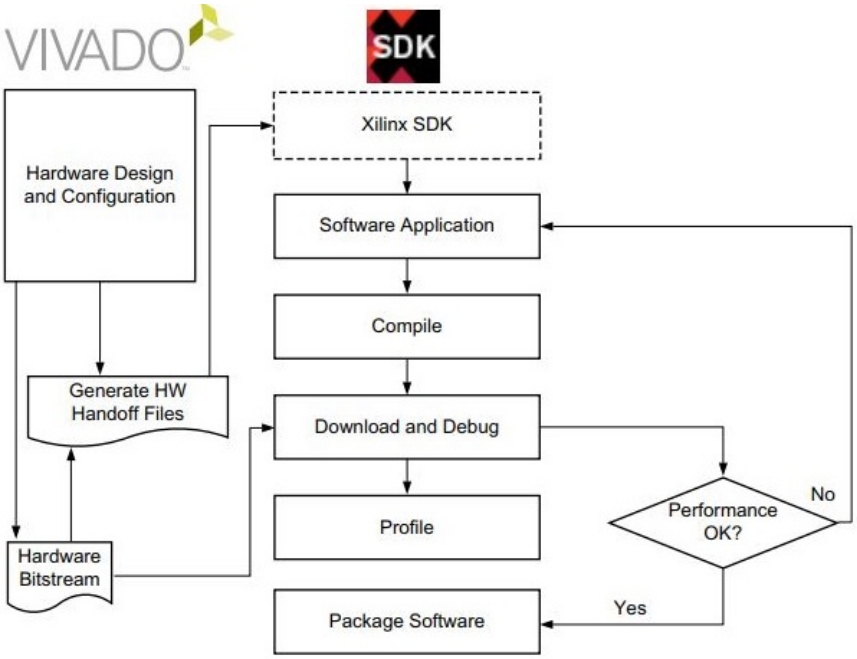


FIGURA 3.14: Esquema de trabajo en Vivado SDK

Capítulo 4

Implementación en FPGA

A pesar de que la mayoría de los algoritmos cuánticos se ejecutan exponencialmente más rápido que su equivalente clásico, esta ventaja no puede ser aprovechada en ninguno de los simuladores por software. Si bien en estos simuladores es posible estudiar y diseñar algoritmos cuánticos, no es posible en ellos estudiar su respuesta temporal, es decir, las mejoras temporales frente a sus equivalentes clásicos. Otra gran desventaja de los simuladores en software es que a medida que aumenta la cantidad de qubits de entrada, el tiempo de procesamiento crece de manera exponencial.

El objetivo de este proyecto es crear un emulador cuántico que no solo simule los algoritmos, sino que también imite el comportamiento físico en una FPGA. Esto resolverá los problemas de los simuladores en software: imposibilidad de estudiar el comportamiento temporal de los algoritmos y el tiempo de procesamiento no crecerá de manera exponencial con la cantidad de qubits debido a la posibilidad de imitar la naturaleza cuántica: realizar múltiples tareas en simultaneo.

La implementación de circuitos basados únicamente en FPGA es una tarea usualmente realizada por ingenieros debido a la necesidad de trabajar con un lenguaje de bajo nivel y manejar detalles a nivel de hardware. Por otro lado, la re-configurabilidad y el procesamiento en paralelo han trasformado a las FPGA en una herramienta adecuada para investigar la computación cuántica. Sin embargo, debido a la complejidad de estos algoritmos y la necesidad de variar parámetros constantemente, no resulta práctico trabajar con lenguajes de bajo nivel como VHDL ó Verilog.

En este contexto se estudió la posibilidad de implementar un Emulador Cuántico en FPGA diseñado a alto nivel utilizando el entorno de Xilinx denominado Vivado. Esto permitirá:

- Desarrollar un emulador cuántico
- Imitar la naturaleza de la cuántica (simultaneidad)
- Trabajar a alto nivel para poder modelizar sistemas complejos

A continuación se explicará, cada una de las partes del proceso.

4.1. Vivado HLS

En Fig. 4.1 se observa un diagrama en bloque general del IPCore (Intellectual Property Core), ésta será la manera en la que trataremos los datos, enviaremos el vector de entrada del circuito y un vector con ceros que luego representará la salida. En el IPCore trabajaremos mediante la lógica de arreglos programada y enviaremos como resultado, tanto el vector de entrada, como el de salida.

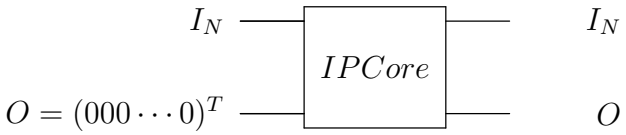


FIGURA 4.1: Diagrama en bloque general para un IPCore

En esta etapa el IPCore es generado. El circuito completo descrito por las compuertas presentadas en Fig. 2.6 y Fig. 2.7 puede ser representadas satisfactoriamente por un código que se implementará en la FPGA. Por el lado de la QFT, este algoritmo puede ser representado por un producto matricial entre la entrada y una matriz compleja equivalente que se deberá calcular en función de la cantidad de qubits de entrada. Para el caso de Grover se deberá ejecutar un algoritmo base que se deberá repetir sobre el vector de estado tantas veces como iteraciones sean necesarias (una cantidad que también deberá ser calculada en función de la cantidad de qubits).

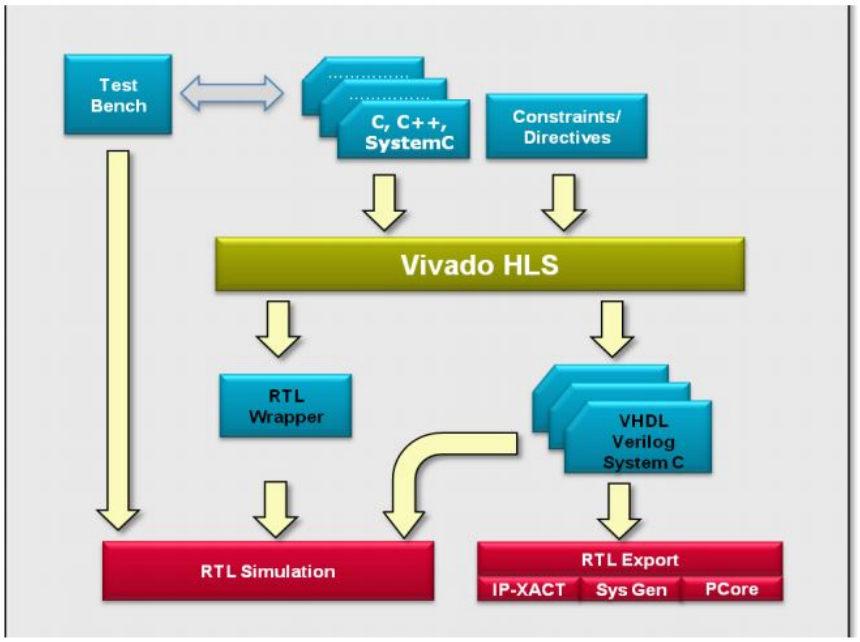


FIGURA 4.2: Esquema de trabajo en Vivado HLS

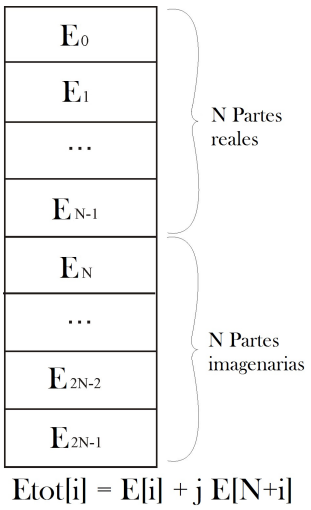


FIGURA 4.3: Parte real e imaginaria de los elementos del vector de entrada complejo

Los códigos son escritos en C sumados a directivas que definirán como éste se convertirá en código RTL (VHDL o Verilog). Mediante una herramienta denominada 'test bench', podremos hacer pruebas en nuestro código antes de que éste sea exportado, tanto en su versión C como VHDL. Probando con distinta cantidad de qubits y de entradas, verificaremos que su funcionamiento sea correcto.

La principal razón por la cual se decidió contemplar ambos algoritmos como un bloque y no como un conjunto de compuertas, es para reducir espacio en la FPGA y poder realizar emulaciones para una mayor cantidad de qubits, es decir un tamaño mayor del vector de entrada. Si intentamos emular el pasaje del vector de entrada por cada una de las compuertas requeridas por el algoritmo hasta llegar al final del circuito, necesitaremos una cantidad de recursos mucho mayor.

Cabe destacar que el hecho de programar a alto nivel permite realizar un código flexible. Es decir, debido a que el código realizado en Vivado es en C, se lo programó de la manera más genérica posible. Supongamos el caso de Fourier, vamos a diseñar el código para entrada de N qubits, luego si queremos implementar N = 4, solamente debemos cambiar el valor de N y ya podremos estudiar el caso en particular. Todas las indicaciones para la sintetización y el diseño del circuito se realizan una sola vez. Este método es muy complejo de realizar con lenguaje RTL, debido a la dificultad extra que genera trabajar a niveles más bajos.

En cuanto a la programación, se puede notar que en el caso de la QFT, tanto el vector de entrada como el de salida pueden ser números complejos (no así en el algoritmo de Grover que la salida es un número real). Como se menciono anteriormente, Vivado tiene ciertas restricciones a la hora de trabajar con C, entre ellas, no provee una librería para trabajar con números complejos.

Por lo tanto, se creó una librería para trabajar con número complejos, donde

se incluyeron todas las funciones necesarias (producto matricial, producto tensorial, etc). La manera de almacenar los datos es siempre en un único vector, dónde la primer mitad corresponde a la parte real y la segunda a la parte imaginaria, como se ve en la Fig. 4.3. La decisión de almacenar todos los datos en un mismo vector es consecuencia de que cada vector será almacenado en una RAM propia (como se explicará más adelante), trabajar con la menor cantidad de vectores concluye en una menor cantidad de Hardware.

Teniendo esto en cuenta, la Transformada Cuántica de Fourier se puede considerar como el producto matricial entre el vector de estados de entrada con una matriz equivalente a todo el conjunto de compuertas que aparecen en el circuito. Y, el algoritmo de Grover, como un proceso iterativo, definido en la sección 2.3.1, que se ejecutara X veces, siendo $X = O(\sqrt{estados})$.

4.1.1. Directivas

Tal vez las directivas sean el concepto más importante en Vivado HLS, porque el diseño final difiere mucho dependiendo de las directivas elegidas. Por ejemplo, se puede optar por un diseño orientado a utilizar la menor cantidad de recursos posibles, a mejorar la velocidad de procesamiento, a que la paralelización sea máxima, entre otros. Siendo el último nuestro caso de interés.

En este trabajo se utilizan tres tipos de directivas: de protocolo de comunicación, de modo de almacenamiento, de implementación de loops. En cuanto al protocolo de comunicación optado, se eligió por AXILITE, las razones de esta elección van a ser desarrolladas luego.

En cuanto a las directivas de almacenamiento, utilizando bloques RAM creados dentro de la FPGA en vez de utilizar las RAM externas que posee la placa se puede disminuir notablemente el tiempo de procesamiento ya que la comunicación deja de ser con la parte externa de la FPGA sino que es interna. La disminución en el tiempo de procesamiento es debido a que no es necesario pasar por un controlador para comunicarse desde el IPCore hacia la RAM. Esto se realiza con directivas, asignándole a cada variable de entrada y salida (vector) un BRAM con una dirección correspondiente.

Las directivas de implementación de loops son las que puede variar más el tiempo de procesamiento y la cantidad de recursos necesarias. Sin embargo no todas las directivas pueden ser utilizadas para todos los casos. Las maneras de implementar los loops van desde, un posible caso secuencial (pocos recursos, mínima paralelización) hasta una directivas llamada unroll (donde la cantidad de recursos es máxima y la paralelización es máxima). Los directivas más comunes se describen a continuación:

- Allocation: Especifica un límite para la cantidad de operaciones utilizadas. Esto puede forzar los recursos de hardware y puede aumentar la latencia.

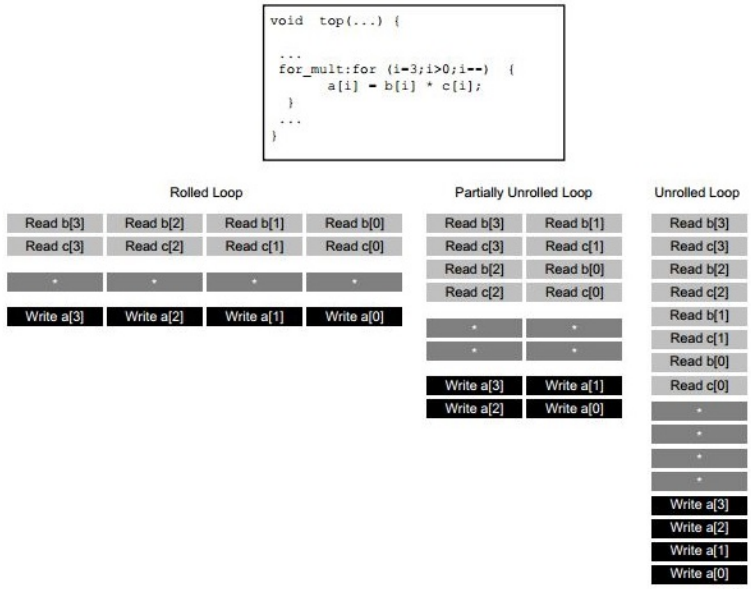


FIGURA 4.4: Directiva: Unroll

- Dataflow: Permite la canalización de nivel de tareas, permitiendo que funciones y bucles se ejecuten simultáneamente. Se usa para minimizar la latencia.
- Dependence: Se usa para proporcionar información adicional que puede superar las dependencias de bucle-acarreo y permitir que los bucles se canalicen (o se canalicen con intervalos más bajos).
- Pipeline: Reduce el intervalo de inicio al permitir la ejecución concurrente de operaciones dentro de un bucle o función.
- Unroll: Desenrolla bucles for para crear múltiples operaciones independientes en lugar de una sola colección de operaciones.

4.1.2. QFT

Después de estudiar todos los diferentes tipos de directivas, el método denominado 'unroll' (Fig. 4.4) resultó ser el que reduce en mayor grado el tiempo de procesamiento y maximiza la paralelización, ya que al ser básicamente un producto matricial entre un vector complejo y una matriz compleja, cada uno de los elementos del vector de salida se puede calcular de manera independiente. Teniendo esto en cuenta, esta directiva es la que permite realizar la mayor cantidad posible de tareas en paralelo.

4.1.3. Algoritmo de Grover

Sin embargo en casos como para el algoritmo de Grover, donde es necesario iterar, el método 'unroll' deja de ser optimo, ya que en este caso los distintos

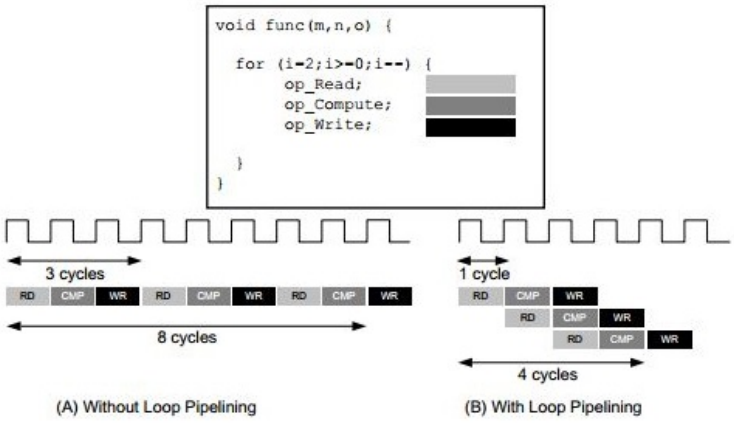


FIGURA 4.5: Directiva: Pipeline

ciclos del for se realizan forzosamente en paralelo. En Grover, para comenzar el ciclo $N+1$ se necesita información calculada en el ciclo N. Cuando hace falta esperar operaciones de una iteración para comenzar la siguiente resulta optimo trabajar con la directiva 'pipeline' (Fig. 4.5). Ésta nos permite optimizar tiempo cuando se trata de implementar tareas que no se pueden realizar completamente en simultaneo.

En el siguiente código se puede observar como con $\#pragma$s el usuario puede proporcionar las directivas de almacenamiento, comunicación y loops sobre ciertas partes especificas del código.

```
#define DOSPI 6.282958984375
#define M 16      //Cantidad de estados

void VQFTAXIBUS(float E[2*M], float S[2*M])
{
#pragma HLS INTERFACE s_axilite port=return bundle=CRTL_BUS
//Directiva de comunicacion
#pragma HLS RESOURCE variable=E core=RAM_1P_BRAM
#pragma HLS INTERFACE bram port=E //Directiva almacenamiento
#pragma HLS RESOURCE variable=S core=RAM_1P_BRAM
#pragma HLS INTERFACE bram port=S //Directiva almacenamiento
int j,k;
float m,n;
for (j=0;j<M;j++){
#pragma HLS UNROLL   //Directiva de loop
S[j]=0;
S[j+M]=0;
for (k=0;k<M;k++){
#pragma HLS UNROLL   //Directiva de loop
m = hls::cos(DOSPI*j*k/M)/hls::sqrt(M);
n = hls::sin(DOSPI*j*k/M)/hls::sqrt(M);
S[j] = S[j] + (m) * E[k] - (n) * E[k+M];
S[j+M] = S[j+M] + (m) * E[k+M] + (n) * E[k];
}}}
```

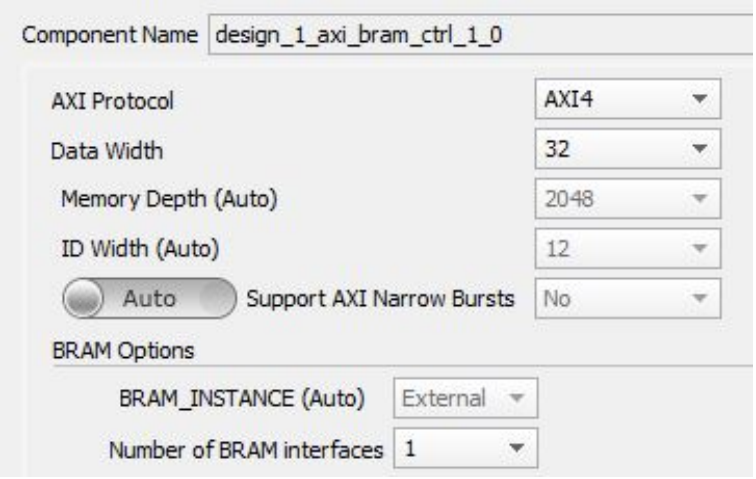


FIGURA 4.6: Parámetros a configurar en controlador de BRAM

4.2. Vivado Design

Una vez que la creación del IPCore deseado está completa, es necesario realizar las conexiones para comunicar el microprocesador (ARM Cortex-A9, es nuestro caso) y el IPCore. Si bien los algoritmos son creados en Vivado HLS, el diseño del emulador donde se implementarán se realiza en Vivado Design. En este emulador se podrán importar cada uno de los bloques diseñados anteriormente y los que se deseen diseñar en el futuro siempre y cuando respeten ciertas condiciones.

4.2.1. Block RAM y Axi Timer

Los bloques de RAM necesarios para almacenar las información de los vectores de entrada y salida del IPCore tienen un inconveniente. Si bien estos generan que la comunicación entre el IPCore y la RAM sea mucho más rápido, la característica por defecto de estos bloques imposibilita la comunicación entre el microprocesador y el BRAM. Debido a que nosotros queremos leer y escribir en las RAMs tanto desde el IPCore como desde el microprocesador se deben realizar ciertas modificaciones manuales como las que se describen a continuación.

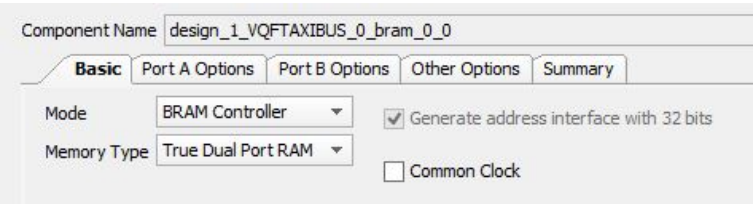


FIGURA 4.7: Parámetros a configurar en BRAM

En primer lugar, los BRAM tienen por defecto un solo puerto (para comunicarse con el IPCore), este valor debe ser modificado manualmente a $TrueDualPortRAM$

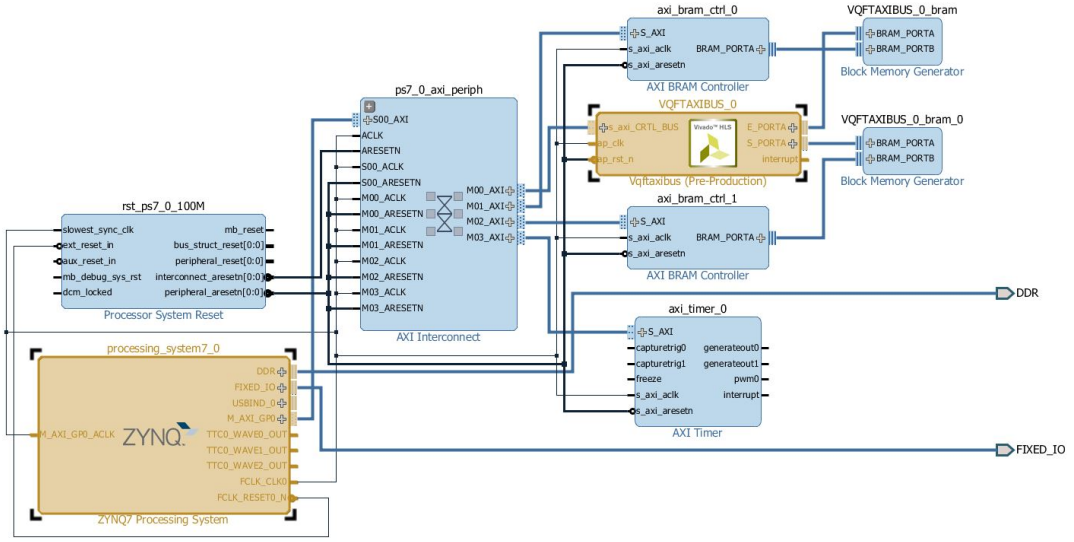


FIGURA 4.8: Diagrama en bloque del diseño en Vivado

(ver Fig. 4.7). Luego, para que el microprocesador se comunique y se entienda con el BRAM es necesario crear un controlador BRAM y conectarlo al segundo puerto del BRAM que se creó anteriormente. Debido a que siempre trabajaremos con ráfagas de datos de tamaño de 32 bits, se definió que el tipo de comunicación entre el controlador BRAM y el BRAM será AXI4-Full, (ver Fig. 4.6).

Otro de los bloques que se agregó al diseño es un AXI Timer, éste permite medir con gran precisión (64 bits) cada uno de los tiempos de procesamiento. El tipo de comunicación que se seleccionó para la transmisión entre el microprocesador y el timer, es AXI4-Lite, debido a que la cantidad del intercambio de datos entre ambos es baja.

4.2.2. Bus de control y AXI

El protocolo adecuado para la comunicación entre ZYNQ7000 y el IPCore es Axi4-Lite. La razón principal de esta elección es que el bus de control que comunica el IPCore con el ZYNQ7000 trabaja con un tipo de comunicación que maneja una cantidad pequeña de datos y tampoco se utiliza constantemente. El bus maneja acciones como: notificar que la comunicación va a comenzar ó terminar, que datos van a ser enviados, que el programa va a esperar hasta que cierta tarea termine, entre otros.

Otra razón por la cual es conveniente usar este protocolo es que Vivado da la opción de 'auto-routear' una parte importante de las conexiones si se trabaja con AXI ahorrando una gran cantidad de tiempo de diseño. A pesar de que algunas de las conexiones deberán ser modificadas (como se explicó anteriormente), gran parte de las conexiones principales son realizaras por el sistema automáticamente.

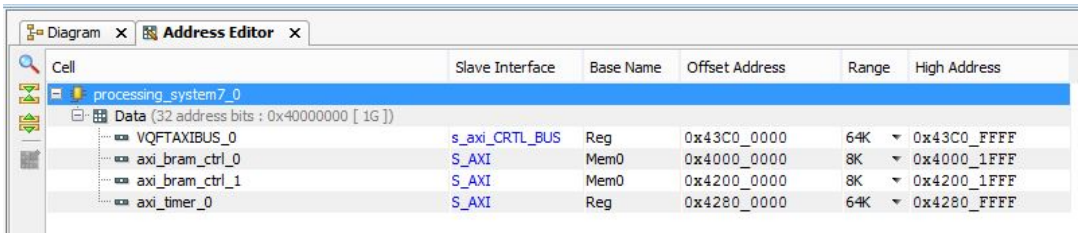


FIGURA 4.9: Dirección en las que el controlador deberá escribir para utilizar las BRAM

4.2.3. Conexiones

En cuánto a diseño, la parte más importante sea probablemente la de realizar las conexiones para comunicar de manera correcta el chip (ZYNQ7000) y el IPCore.

El diseño final es el que se puede observar en la figura 4.8, éste nos permite no solo trabajar con los algoritmos particulares con los que hemos trabajado, sino que simplemente intercambiando el IPCore en uso, por uno nuevo, el diseño funcionará sin problemas. Solo se debe recordar que los IPCore diseñados deben cumplir con las características básicas anteriormente mencionadas (almacenar los vectores de entrada y salida en BRAM y trabajar con un protocolo AXI4-Lite en el bus de control).

4.3. Vivado SDK

Vivado SDK es la última etapa donde el algoritmo a estudiar va a ser ejecutado tanto en el microprocesador como en la FPGA. Utilizando los drivers y realizando ciertas especificaciones sobre el hardware, podremos utilizar al IPCore y a cada uno de los bloques agregados en el Vivado Design como una función más desde el punto de vista del microprocesador.

Con este programa y utilizando C vamos a asignar las direcciones de memoria RAM a las variables correspondiente. Para establecer los valores de entrada a la FPGA, vamos a crear punteros y asignarlos a las direcciones que correspondan a cada una de las BRAM, las cuales podemos visualizar en Vivado Design (Fig. 4.9), junto con el tamaño de cada una de las BRAM (8Kb).

```
float *EHW = (float*)0x40000000;
float *SHW = (float*)0x42000000;
```

Luego de hacer esto, estas RAMs puede ser escritas o leídas simplemente modificando estos punteros. Esta es la idea general de Vivado, diseñar hardware y modelizarlo de manera tal que éste pueda ser tratado como software desde el punto de vista del microprocesador.

4.3.1. Drivers

La manipulación correcta de los drivers es lo que nos permitirá utilizar bloques de hardware desde el microprocesador. Si bien los drivers son generados automáticamente, estos deben ser inicializados por nosotros utilizando los prototipos de funciones disponible.

IP Core

En el caso del IPCore, los drivers no son solo útiles para inicializar el procesador sino también para controlarlo. El código de inicialización del IP Core se adjunta a continuación (ejemplo para Fourier):

```
void init_vqftaxibusCore()
{
  int status = 0;

  doVqftaxibus_cfg =
     XVqftaxibus_LookupConfig(XPAR_XVQFTAXIBUS_0_DEVICE_ID);
  if (doVqftaxibus_cfg)
  {
    status =
       XVqftaxibus_CfgInitialize(&doVqftaxibus,doVqftaxibus_cfg);
    if (status != XST_SUCCESS)
    {
      printf("Error a inicializar doVqftaxibus\n");
    }
  }
}
```

Funciones como $XVqftaxibus_IsDone(\&doVqftaxibus)$ (esta función solo se pone en 1 cuando el IPCore terminó su procesamiento), para el caso Fourier, nos permitirán manipular el hardware. Todas los prototipos de funciones disponibles se muestran a continuación.

```
void XVqftaxibus_Start(XVqftaxibus *InstancePtr);
u32 XVqftaxibus_IsDone(XVqftaxibus *InstancePtr);
u32 XVqftaxibus_IsIdle(XVqftaxibus *InstancePtr);
u32 XVqftaxibus_IsReady(XVqftaxibus *InstancePtr);
void XVqftaxibus_EnableAutoRestart(XVqftaxibus *InstancePtr);
void XVqftaxibus_DisableAutoRestart(XVqftaxibus
   *InstancePtr);

void XVqftaxibus_InterruptGlobalEnable(XVqftaxibus
   *InstancePtr);
void XVqftaxibus_InterruptGlobalDisable(XVqftaxibus
   *InstancePtr);
void XVqftaxibus_InterruptEnable(XVqftaxibus *InstancePtr,
   u32 Mask);
```

```
void XVqftaxibus_InterruptDisable(XVqftaxibus *InstancePtr,
   u32 Mask);
void XVqftaxibus_InterruptClear(XVqftaxibus *InstancePtr,
   u32 Mask);
u32 XVqftaxibus_InterruptGetEnabled(XVqftaxibus
   *InstancePtr);
u32 XVqftaxibus_InterruptGetStatus(XVqftaxibus *InstancePtr);
```

AxiTimer

Usando los prototipos de funciones generados en los drivers del bloque AxiTimer, se creó una librería para facilitar las repetidas mediciones que se realizaran en esta última etapa.

```
class AxiTimerHelper {
public:
  AxiTimerHelper();
  virtual ~AxiTimerHelper();
  unsigned int getElapsedTicks();
  double getElapsedTimerInSeconds();
  unsigned int startTimer();
  unsigned int stopTimer();
  double getClockPeriod();
  double getTimerClockFreq();
private:
  XTmrCtr m_AxiTimer;
  unsigned int m_tickCounter1;
  unsigned int m_tickCounter2;
  double m_clockPeriodSeconds;
  double m_timerClockFreq;
};
```

4.3.2. Formato de datos

Debido que los datos del vector de entrada serán enviados desde el microprocesador a la FPGA, debemos tener en cuenta que la manera en la cual el hardware interpreta los datos no es la misma que la del microprocesador. El microrprocesador trabaja con $float$, en cambio, la FPGA, trabaja con $uint32$ (unsigned 32 bits). Por lo tanto, se programaron funciones que transformen de float a uint32 y de uint32 a float para que cada vez que se desee enviar datos de la FPGA al microprocesador y viceversa la comunicación sea correcta.

```
unsigned int float_to_u32(float val)
{
  unsigned int result;
  union float_bytes {
```

```
    float v;
    unsigned char bytes[4];
  } data;
  data.v = val;

  result = (data.bytes[3] << 24) + (data.bytes[2] << 16) +
     (data.bytes[1] << 8) + (data.bytes[0]);
  return result;
}

float u32_to_float(unsigned int val)
{
union {
    float val_float;
    unsigned char bytes[4];
  } data;
  data.bytes[3] = (val >> (8*3)) & 0xff;
  data.bytes[2] = (val >> (8*2)) & 0xff;
  data.bytes[1] = (val >> (8*1)) & 0xff;
  data.bytes[0] = (val >> (8*0)) & 0xff;
  return data.val_float;
}
```

4.3.3. Modo de trabajo

Con todo lo explicado anteriormente, la metodología va a ser la siguiente: primero se copia el código en C (utilizado previamente en Vivado HLS para implementar el IP Core) sin las directivas y lo utilizaremos como una función en el microprocesador. Luego, utilizando la librería del AXI Timer, se mide y almacena en una variable el tiempo que tarda el microprocesador en ejecutar el código. Finalmente, se llama (pasándole los parámetros necesarios) al bloque IP Core (hardware), se mide con el AXI Timer y almacena el tiempo de procesamiento en otra variable. Realizaremos esta tarea para distinta entradas, y distinta cantidad de qubits.

```
myTimerSW.startTimer(); // Inicio de medicion 1
VQFTAXIBUS(ESW,SSW); // Ejecucion software
myTimerSW.stopTimer(); // Fin medicion 1

myTimerHW.startTimer();    //Inicio medicion 2
XVqftaxibus_Start(&doVqftaxibus); //Ejecucion hardware
while (!XVqftaxibus_IsDone(&doVqftaxibus));
myTimerHW.stopTimer();    //Fin medicion 2
```

Capítulo 5

Análisis de resultados

El análisis de resultados se divide en dos enfoques, por un lado se analiza la validez los resultados de los algoritmos propuestos tanto en hardware como en software y se interpretan en función de sus características. Por otro lado, se hace un análisis la respuesta temporal, es decir, la diferencia entre el algoritmo implementado en FPGA y el mismo algoritmo corriendo en el microprocesador.

La lectura de datos y los gráficos obtenidos fueron analizados en Matlab. La placa tiene una salida UART por la cual se envían los datos, tanto de entrada, salida, como tiempos de procesamiento. Estos valores son leídos, interpretados y finalmente graficados mediante un script para verificar que el resultado haya sido el esperado.

El funcionamiento general del sistema es el siguiente: se escribe el programa (versión C) en el microprocesador (seteando los vectores de entrada), se ejecuta el algoritmo en el software (se obtiene la salida de software), luego se ejecuta el algoritmo en la FPGA (se obtiene la salida de hardware).

Estos valores (junto con los tiempos de procesamiento correspondientes) se envían desde el microprocesador por la UART a la PC, estos son cargados en Matlab y finalmente analizados. El único parametro a setear a ambos lados de la comunicación es el baud-rate. El baud-rate utilizado es 115200bps debido a que es la máxima velocidad soportada por el protocolo. A continuación se muestra la sección del script en Matlab encargada de la lectura de datos:

```
s = serial('COM3','BaudRate',115200);
Q = 4;   //cantidad de qubits
M = 2^Q; //cantidad de estados
fopen(s);
x=[];
for i=1:(3*M+2)  // 3M (3 vectores: entrada, salidaSW y
   salida HW) + 2 (tiempos: SW y HW)
x = [x fscanf(s,'%f')];
end
fclose(s);
```

5.1. Tamaño de datos

Antes de pasar al análisis de datos, vale la pena hacer un comentario sobre el tamaño de las variables con las que se trabajó. Como se mencionó anteriormente, tanto en hardware como en software se utilizan datos de 32 bits. Sin embargo, las razones de trabajar con esta precisión se justifican a continuación.

Vivado HLS brinda la posibilidad de trabajar el hardware de una manera bastante flexible. Una de las opciones que brinda, es la de elegir el tamaño en que la FPGA almacenará los datos. Aunque el código esté escrito en C y se utilicen variables float (de 32 bits), es posible adaptar el código para que en hardware se almacene la cantidad de bits que se necesite en función del balance entre precisión y hardware requerido.

Debido a que los datos con los se trabaja tienen como valor máximo (en valor absoluto) 1 (recordemos que los datos almacenados son probabilidades), resulta tentador utilizar punto fijo. Por lo tanto, se estudiaron distintas posibilidades.

Por ejemplo para el caso de QFT con 4 qubits de entrada se realizó la siguiente prueba: se calcularon y compararon la cantidad de hardware requerido y la resolución de la salida del mismo IPCore pero diseñado con tres tamaños de datos distintos: punto fijo 16 bits, punto fijo 32 bits y punto flotante 32 bits.

CUADRO 5.1: Resolución de los datos de salida para una misma entrada en 4QFT para punto fijo 16 bits, punto fijo 32 bits, punto flotante 32 bits y matlab 64 bits.

Punto fijo (16 bits)	Punto fijo (32 bits)	Punto flotante (32 bits)	Matlab (64 bits)
Salida	Salida	Salida	Salida
0.250000+0.000000i	0.250000+0.000000i	0.250000+0.000000i	0.250000+0.000000i
-0.095703+0.230957i	-0.095655+0.230977i	-0.095655+0.230977i	-0.095670+0.230969i
-0.177002-0.176758i	-0.176802-0.176752i	-0.176802-0.176752i	-0.176776-0.176776i
0.230713-0.095947i	0.230950-0.095720i	0.230950-0.095720i	0.230969-0.095670i
0.000000+0.249756i	0.000071+0.250000i	0.000071+0.250000i	0.000000+0.250000i
-0.231201-0.095703i	-0.231004-0.095589i	-0.231004-0.095589i	-0.230969-0.095670i
0.176514-0.177002i	0.176702-0.176852i	0.176702-0.176852i	0.176776-0.176776i
0.095703+0.230713i	0.095785+0.230922i	0.095785+0.230922i	0.095670+0.230969i
-0.250000+0.000000i	-0.250000+0.000141i	-0.250000+0.000141i	-0.250000+0.000000i
0.095459-0.231201i	0.095524-0.231031i	0.095524-0.231031i	0.095670-0.230969i
0.176758+0.176514i	0.176902+0.176652i	0.176902+0.176652i	0.176776+0.176776i
-0.230957+0.095703i	-0.230895+0.095851i	-0.230895+0.095851i	-0.230969+0.095670i
-0.000244-0.250000i	-0.000212-0.250000i	-0.000212-0.250000i	-0.000000-0.250000i
0.230957+0.095215i	0.231058+0.095458i	0.231058+0.095458i	0.230969+0.095670i
-0.176758+0.176758i	-0.176602+0.176952i	-0.176602+0.176952i	-0.176776+0.176776i
-0.095947-0.230957i	-0.095916-0.230868i	-0.095916-0.230868i	-0.095670-0.230969i

En cuanto a la resolución de los datos, en la tabla 5.1 se muestran los datos registrados donde se ingresó al sistema con una misma entrada y se midió la salida resultante. Para cada uno de los casos se trabajo con un tamaño y formato de datos diferente.

Observando detalladamente, se observa que recién comienza a haber diferencia a partir del cuarto dígito después de la coma. Es decir, comparando los cuatro casos, en los primero tres dígitos después de la coma siempre obtenemos el mismo resultado.

Por lo tanto, en función de la resolución de los datos no resulta una limitación trabajar con 16, 32 o 64 bits, ya que se considera que el hecho de que los valores recién cambien en el cuarto dígito no es un problema a fines prácticos.

El problema surge a la hora de implementar, debido a la cantidad de recursos necesarios en cada uno de los casos. En la figura 5.1 vemos estos números extraídos del reporte entregado por Vivado los tres casos considerados: punto fijo 16 y 32 bits y punto flotante 32 bits, tal como observamos en los dos primeros casos la cantidad de DSPs es superior (en rojo) a la disponible en la placa. Por contrario, para el caso de punto flotante, la cantidad de recursos es más que suficiente. Esto se debe a que el sintetizador de Vivado tiene una manera de implementar las funciones trigonométricas más eficiente en el caso del punto flotante. Esto es, utiliza Look-Up Tables para disminuir la cantidad de operaciones a realizar.

En cambio, en el caso de punto fijo, la relación entre la parte entera y la fraccionaria son asignadas por el usuario y tal como se ve en la figura 5.1 se requieren numerosos DSP para cumplir con la misma tarea. Lo cual hace el sistema menos eficiente. Finalmente, en función de este análisis se decidió trabajar con punto flotante de 32 bits.

5.2. Transformada Cuántica de Fourier

5.2.1. Eficacia del emulador

En este apartado se analiza cuan cercanos son los datos obtenidos de la FGPA frente a los obtenidos del microprocesador. Los vectores a la salida son los mismos debido, principalmente, a que el tamaño de los datos utilizados en la FPGA es, como se explicó anteriormente, de 32 bits, un tamaño que no solo es lo suficientemente grande como para mostrar los resultados con un alto grado de precisión sino que también es el mismo que se utiliza en el microprocesador, float (32 bits).

En las imágenes 5.2 y 5.3, se puede ver como es la respuesta de la transformada de Fourier para una entrada senoidal y un pulso respectivamente. La respuesta obtenida es la misma que se obtendría al aplicarle la DFT (Discrete Fourier transform) a la entrada. Para una entrada senoidal obtenemos dos

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	300	-	-
Expression	-	-	0	13832
FIFO	-	-	-	-
Instance	0	-	36	40
Memory	-	-	-	-
Multiplexer	-	-	-	949
Register	-	-	11726	-
Total	0	300	11762	14821
Available	280	220	106400	53200
Utilization (%)	0	136	11	27

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	30000
FIFO	-	-	-	-
Instance	0	3044	36	40
Memory	-	-	-	-
Multiplexer	-	-	-	1196
Register	-	-	43854	-
Total	0	3044	43890	31236
Available	280	220	106400	53200
Utilization (%)	0	1383	41	58

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	0	50	3516	7150
Memory	-	-	-	-
Multiplexer	-	-	-	7128
Register	-	-	2629	-
Total	0	50	6145	14278
Available	280	220	106400	53200
Utilization (%)	0	22	5	26

FIGURA 5.1: Cantidad de recursos necesarios para implementar 4QFT. 1) Punto fijo 16 bits, 2) Punto fijo 32 bits, 3) Punto flotante 32 bits.

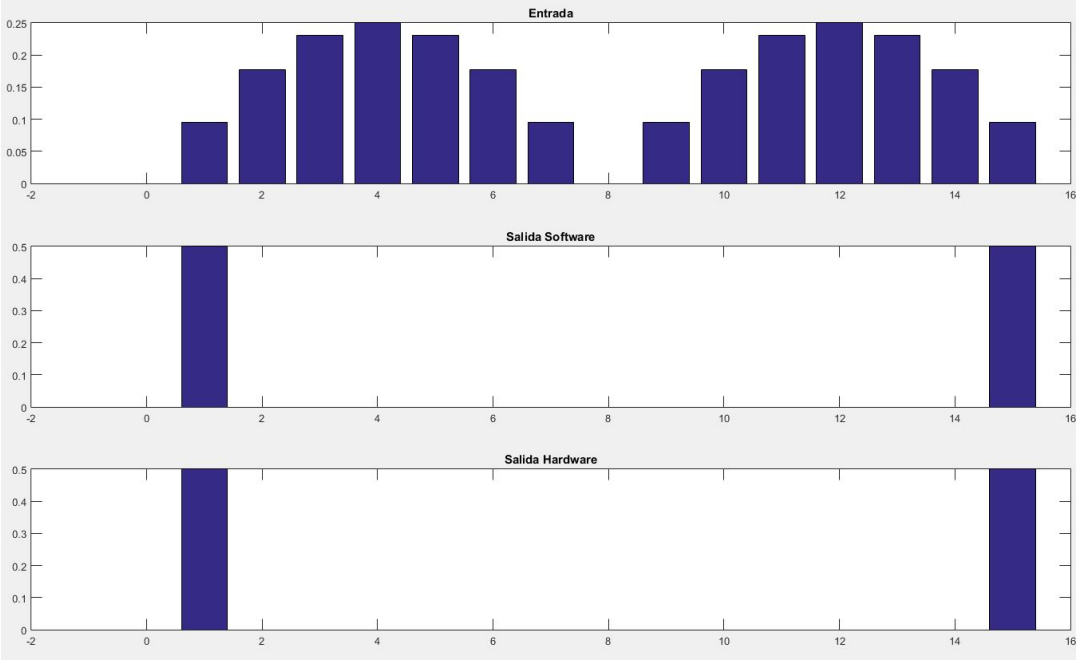


FIGURA 5.2: Entrada senoidal y salida de QFT en el Hardware

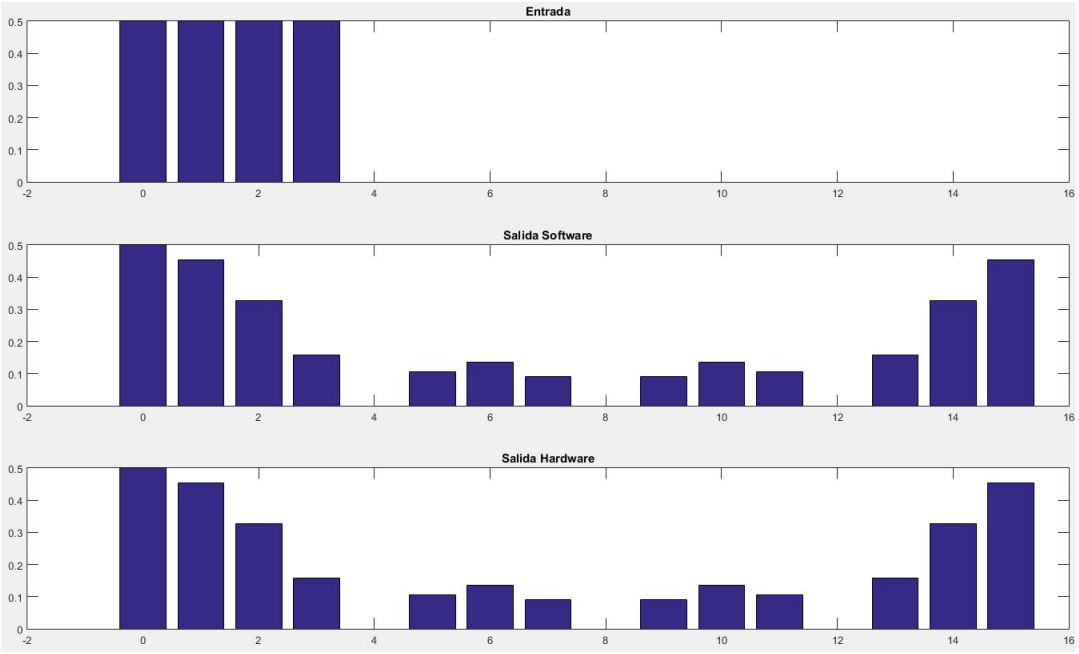


FIGURA 5.3: Entrada pulso y salida de QFT en el Hardware

deltas simétricas y para un pulso obtenemos una sinc a la salida. Sin embargo, el proceso es realizado mediante el algoritmo cuántico de la transformada de Fourier.

Estas igualdades en los resultados entre la FPGA y el micrprocesador, verifican el correcto funcionamiento del sistema a la hora de emular el algoritmo en hardware. Cabe destacar que cuando se comparan los valores de salida de QFT de la FPGA con los de las primeras simulaciones de Matlab hay una diferencia en los bits menos significativos, esto se debe a que Matlab trabaja con 64 bits. Esta diferencia se podría corregir aumentando el tamaño de los datos de la FPGA, lo cual no sería práctico debido a que no se busca tal grado de precisión y generaría un aumento excesivo en la cantidad de hardware requerido.

5.2.2. Tiempos

El análisis más importante es el realizado para calcular la respuesta temporal entre el algoritmo en hardware y software. Si bien es importante que el funcionamiento de los algoritmos sea el correcto, la ventaja frente a los simuladores del emulador es la capacidad de imitar el comportamiento temporal de los cuánticos de una forma más eficiente. Este eficiencia es analizada a través de una comparación entre los tiempos de cálculo.

En la tabla 5.2 podemos ver los tiempos de procesamiento para cada una de las NQFT (siendo N el número de qubits de entrada) en el microprocesador (ARM Cortex™-A9) y el diseño en nuestro IPCore (FPGA).

CUADRO 5.2: Tiempos obtenidos para las distintas NQFT

	IPCore (FPGA) (μseg)	ARM Cortex™-A9 (μseg)
1QFT	1.2	6.18
2QFT	1.6	13.67
3QFT	2.4	60.53
4QFT	3.9	236.06
5QFT	4.7	986.41
6QFT	5.8*	4609.74

Se debe aclarar que los valores para la FPGA de 6QFT, no pudieron ser medidos debido que los requisitos de hardware necesarios son mayores a los disponibles en la placa con la que se trabajó. Sin embargo, Vivado HLS contiene una herramienta que permite estimar el tiempo de procesamiento (latencia, en función de los ciclos de reloj) que requerirá el IPCore en caso de éste poder ser implementado.

Las mejoras en el tiempo de procesamiento son notables debido a la principal ventaja que tienen las FPGA sobre los microprocesadores: su capacidad para realizar diferentes tareas en paralelo, imposible en todos los dispositivos que funcionan secuencialmente.

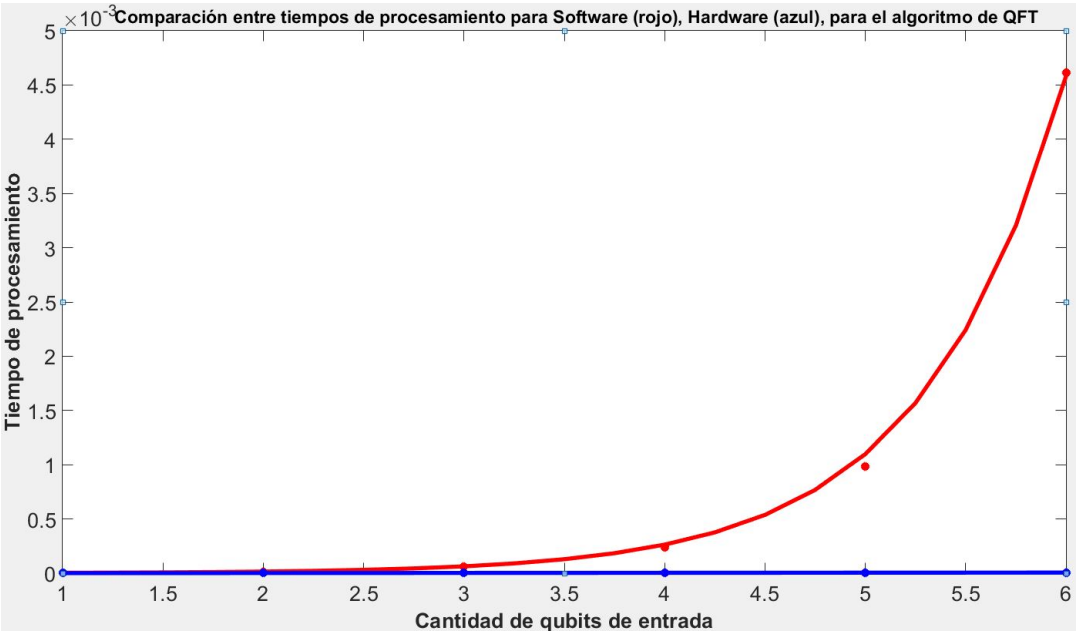


FIGURA 5.4: Tiempo de procesamiento en función de qubits para la QFT

$$\textit{Tiempo Software} = 8{,}702 * 10^{-7} * e^{1{,}428*N}(R - square : 0{,}9991) \tag{5.1}$$

$$\textit{Tiempo Hardware} = 9{,}677 * 10^{-7} * N - 1{,}42 * 10^{-7}(R - square : 0{,}9772) \tag{5.2}$$

Los tiempos en la FPGA no son solo significativamente menores a los del microprocesador, la diferencia más importante está en el orden de crecimiento del tiempo de procesamiento en función de la cantidad de qubits. Éste, para la simulación en software, muestra un incremento exponencial (ver 5.1) con N (cantidad de qubits) en el microprocesador mientras que el incremento correspondiente a la FPGA es lineal (ver 5.2), cumpliendo con el objetivo inicialmente planteado.

Este comportamiento se puede ver de manera más clara en la figura 5.4. En la curva roja (software) podemos observar cómo, interpolando los 6 puntos con los que contamos, vemos una relación exponencial. Y, haciendo zoom en la curva azul (hardware) de la figura 5.4, se obtiene la figura 5.5, donde se ve que el crecimiento responde a una recta.

Todas las interpolaciones fueron calculadas con Matlab tomando como parametro $R - square$ para obtener la curva que mejor se ajuste a la secuencia de puntos.

A pesar de las fehacientes mejoras que plantea el sistema, hay un inconveniente. Las mejoras en el pasaje de un crecimiento exponencial a uno lineal, se compensan con un incremento en la cantidad de hardware necesario para

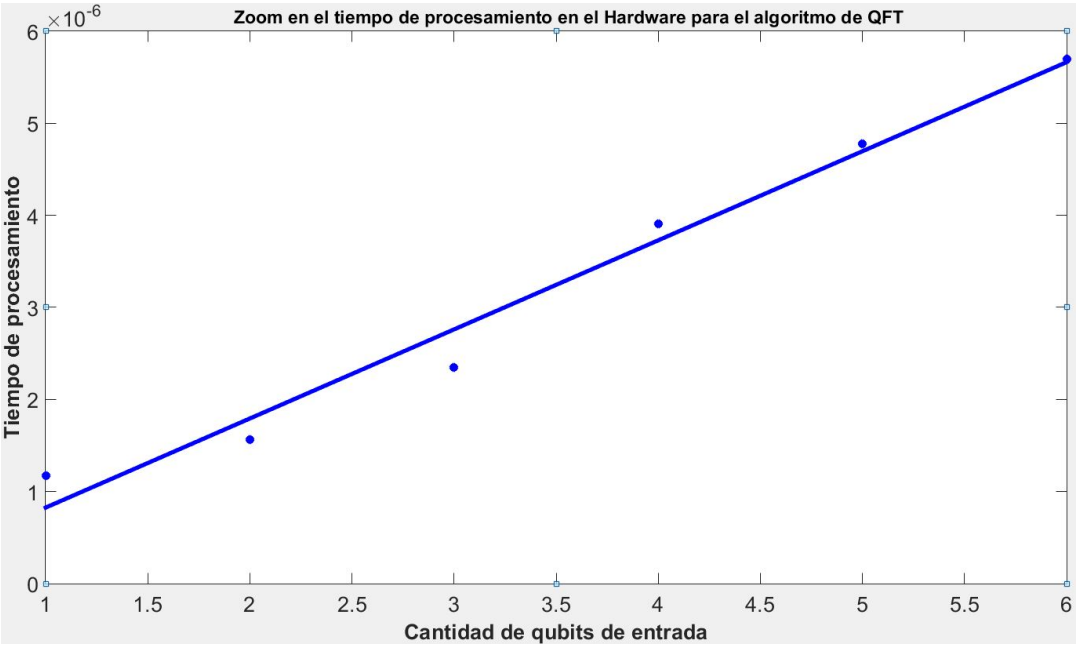


FIGURA 5.5: Tiempo de procesamiento en FPGA (μseg) vs Número de Qubits.

implementar el algoritmo en función de la cantidad de qubits. Es decir, los recursos requeridos en la FPGA crecen de manera exponencial. Este hecho limita de manera notable la cantidad máxima de qubits con la que el sistema puede trabajar.

En el caso de la QFT, el recurso que crece exponencialmente son las LUTs (LookUp-Tables), esto se debe a que la mayor parte de las operaciones se deben a funciones trigonometricas. La manera más eficiente que tiene Vivado de implementar estas funciones es almacenando salida para cada una de las entradas posibles y, así evitar calcularlas. A medida que crece la cantidad de qubits, crece exponencialmente la cantidad de cuentas a realizar y, por lo tanto, la cantidad de LUTs necesarias. En la tabla 5.5 se ve la cantidad de LUTs y DSPs requeridas, junto con el porcentaje en relación a la cantidad disponible en nuestra FPGA.

CUADRO 5.3: LUT usadas en función del tamaño de la QFT

	1QFT	2QFT	3QFT	4QFT	5QFT	6QFT
LUTs	1846	3760	8116	14278	51870	124062
LUT(%)	3	7	15	26	97	233
DSPs	10	20	40	50	90	–
DSP(%)	4	9	18	22	40	–

Finalmente en la figura 5.6 se puede analizar de manera visual como la cantidad de hardware necesario para implementar el IPCore aumenta de manera exponencial con la cantidad de qubits, desde uno a cinco. La parte roja representa el IPCore, la azul las BRAM y lo amarillo las inter-conexiones y el timer.

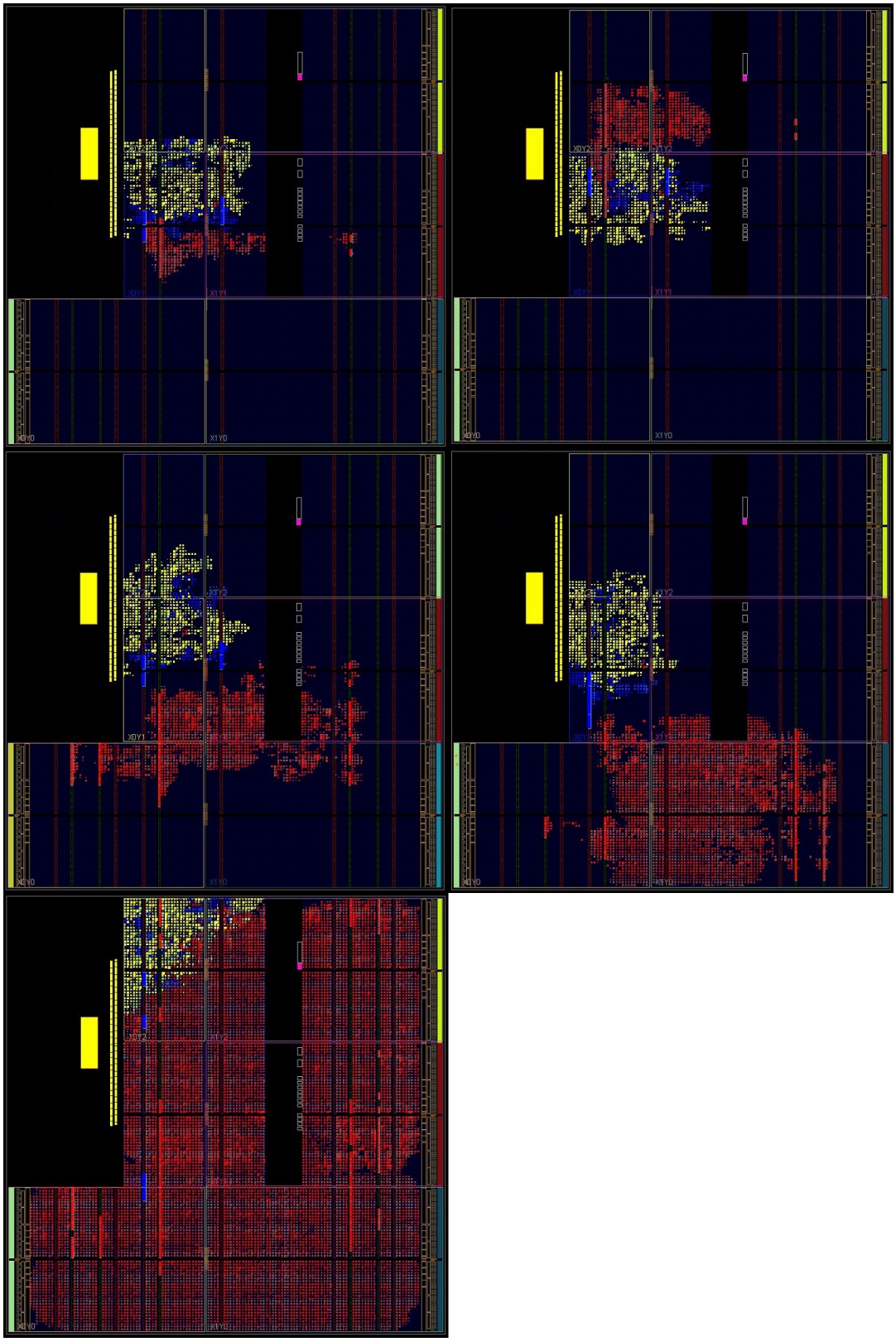


FIGURA 5.6: Crecimiento del Hardware en función de la cantidad de Qubits. Rojo: IPCore. Azul: BRAM y controladores. Amarillo: Inter-conexiones y timer.

5.3. Algoritmo de Grover

5.3.1. Eficacia del emulador

El emulador fue evaluado esta vez con el algoritmo de búsqueda de Grover. La eficiencia se verificó nuevamente comparando los datos obtenidos de la FPGA con los datos a la salida del microprocesador. El tamaño de los datos almacenados y el diseño se mantienen, y los valores finales son los mismos hasta en los bits menos significativos.

La manera de presentar los datos es la siguiente: trabajando con una lista de 32 elementos, se muestran en un gráfico los valores de los elementos del vector de salida. Recordemos que el vector de salida muestra la probabilidad de que cierto elemento sea encontrado luego de una búsqueda.

En las figuras 5.7 y 5.8 se pueden observar dos casos posibles. En el primero, de las lista de 32 elementos, dos cumplen la condición requerida (elementos 11 y 27), por lo tanto, estos elementos tendrán una probabilidad mayor (48,066 %) que el resto que no cumplen la condición. En cambio, en la segunda figura, cuatro de los 32 elementos cumplen con la condición, por lo tanto, la probabilidad de que estos elementos sean encontrados es de un 23,633 %. Cabe destacar que la probabilidad de ser encontrados de los elementos que cumplen la condición es siempre la misma entre ellos.

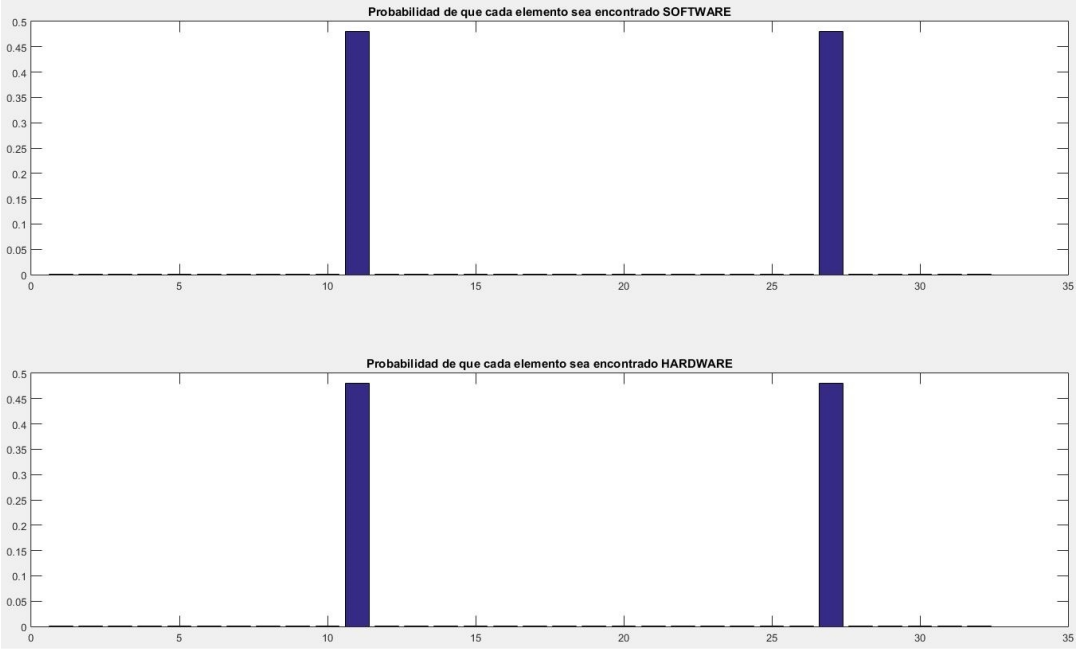
El IPCore correspondiente al algoritmo de Grover fue creado de forma independiente. Esto significa que el correcto funcionamiento del diseño del emulador no depende del algoritmo con el que se trabaje, siempre y cuando se mantengan las características generales anteriormente mencionadas (ver 4.2.3).

5.3.2. Tiempos

Si bien la eficiencia obtenida en este caso arroja resultados equivalentes al caso de la QFT, el análisis de la respuesta temporal tiene algunas diferencias notables debido a que las características de los algoritmos son muy diferentes. En la tabla comparativa 5.4 se observan los tiempos de ejecución tanto para la FPGA como para el microprocesador para distintos tamaños de espacios de búsqueda.

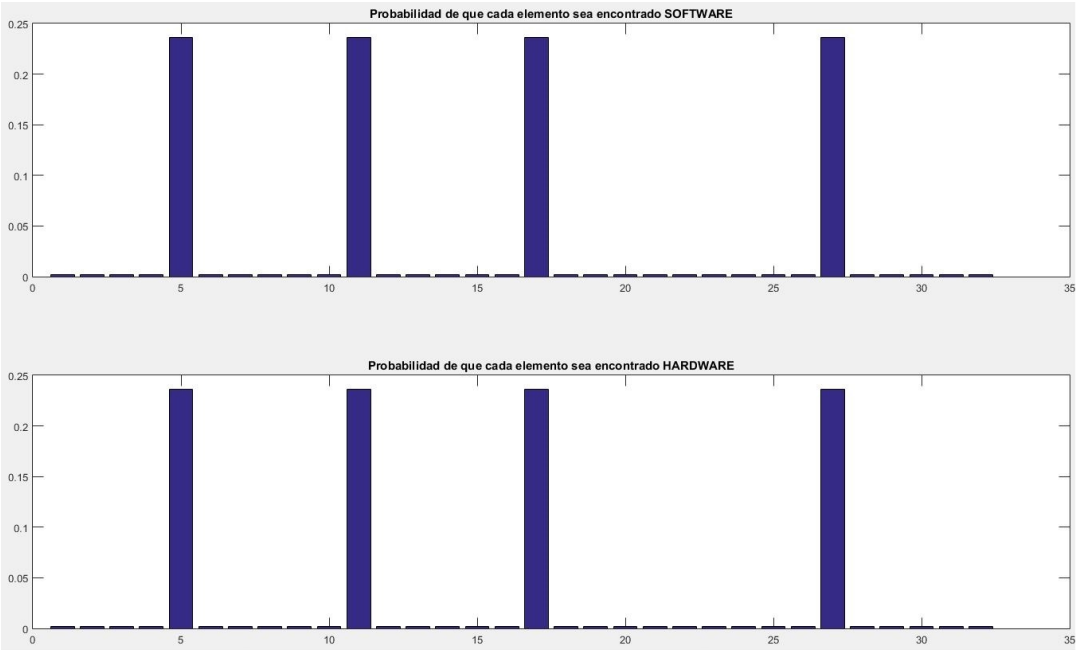
El máximo tamaño del espacio de búsqueda que se logra implementar con el hardware disponible es $2^6 = 64$ elementos. Pero el tiempo de procesamiento para el caso de 128 elementos se calculó en función de los parámetros entregados por Vivado (latencia y frecuencia clock).

Como se observa en la tabla 5.4 para los dos primeros casos, Grover4 y Grover8, los tiempos de procesamiento son más altos en FPGA que en el microprocesador. En el caso de 16 elementos los tiempos son comparables, y de ahí



El numero buscado es el 22, en la posicion 11, con probabilidad 0.480659
El numero buscado es el 19, en la posicion 27, con probabilidad 0.480659

FIGURA 5.7: Probabilidad de encontrar dos elementos en una lista de 32



El numero buscado es el 17, en la posicion 5, con probabilidad 0.236328
El numero buscado es el 22, en la posicion 11, con probabilidad 0.236328
El numero buscado es el 21, en la posicion 17, con probabilidad 0.236328
El numero buscado es el 19, en la posicion 27, con probabilidad 0.236328

FIGURA 5.8: Probabilidad de encontrar cuatro elementos en una lista de 32

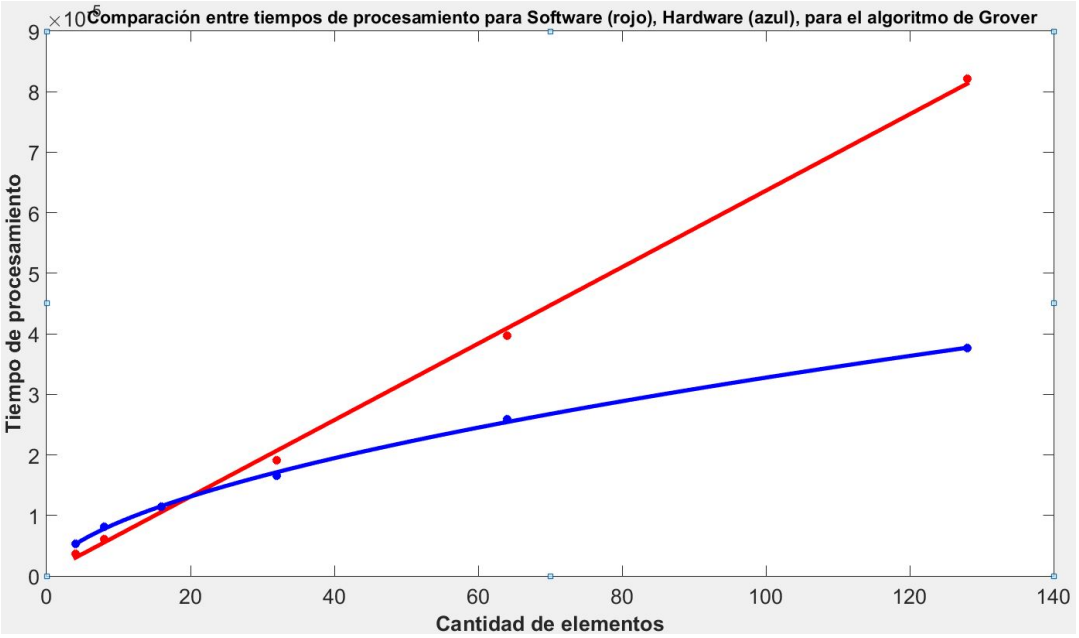


FIGURA 5.9: Tiempo de procesamiento en función de cantidad de elementos Grover

en adelante las mejoras de la FPGA frente al microprocesador comienzan a ser notables.

$$\textit{Tiempo Software} = 6{,}31 * 10^{-7} * n + 4{,}87 * 10^{-7}(R - square : 0{,}9987) \quad (5.3)$$

$$\textit{Tiempo Hardware} = 2{,}387 * 10^{-6} n^{0{,}5688}(R - square : 0{,}9992) \quad (5.4)$$

Si se grafica el tiempo de procesamiento en función de la cantidad de estados, se obtiene la figura 5.9, donde se observa cómo el tiempo de procesamiento crece de manera lineal (ver 5.3) con la cantidad de estados para el caso del software (rojo) mientras que para el caso de hardware (azul), esto ocurre en el orden $O(\sqrt{n})$ (siendo n la cantidad de estados) (ver 5.4).

CUADRO 5.4: Tiempos obtenidos para distinta cantidad de elementos en el algoritmo de Grover

Elementos (Q)	IPCore (FPGA) (μseg)	ARM Cortex™-A9 (μseg)
4 (2)	5.32	3.7
8 (3)	8.12	6.1
16 (4)	11.44	11.3
32 (5)	16.59	19.1
64 (6)	25.85	39.7
128 (7)	37.63*	82.2

Este comportamiento verifica lo teóricamente predicho, ya que el algoritmo de Grover, no propone una mejora exponencial como la QFT, sino de $O(\sqrt{n})$ como se mencionó en el marco teórico. Esto se debe principalmente a que es un proceso iterativo y no aprovecha al máximo la capacidad de paralelizar que ofrece la computación cuántica.

Otra manera de explicar matemáticamente por qué el tiempo de procesamiento crece con $O(\sqrt{n})$, es que la cantidad de iteraciones óptima para calcular la probabilidad de encontrar a un elemento, responde a la siguiente expresión: $K = \frac{\pi}{4}\sqrt{n}$, siendo n la cantidad de estados.

Finalmente, cabe destacar que a pesar de que la mejora en el tiempo no fue de orden exponencial, el hardware necesario sí aumenta de manera exponencial con la cantidad de qubits de entrada. Es decir, esta limitación sigue presente (y lo seguirá en todos los algoritmos cuánticos). En la tabla 5.5 se ve como crece el hardware necesario de manera total y en función del los recursos disponibles.

CUADRO 5.5: LUTs y DSPs usados en función del tamaño de la lista de elementos

Elementos (Q)	4 (2)	8 (3)	16 (4)	32 (5)	64 (6)	128 (7)
LUTs	11841	14141	18654	27887	47021	66211
LUT(%)	22	26	35	52	88	124
DSPs	36	44	60	92	156	174
DSP(%)	16	20	27	41	70	89

Capítulo 6

Conclusiones

El procesamiento paralelo hace que la emulación cuántica en FPGA sea más eficiente que la simulación en software. Un sistema de hardware como éste y fácil de manejar puede ser muy útil para aquellos investigadores de computación cuántica que tienen conocimientos mínimos de RTL (VHDL o Verilog), pero que estén familiarizados con un lenguaje de programación más usual tal como C/C++. En este marco se desarrollo un sistema de emulación de circuitos cuánticos que aprovecha las herramientas de síntesis de alto nivel proporcionadas por Vivado ®, permitiendo al usuario programar la FPGA a partir de un algoritmo escrito en C, más ciertas directivas relacionadas con el hardware. Además, con la metodología propuesta el diseñador puede estudiar desde el más básico IPCore cuántico, como una puerta cuántica simple, hasta un algoritmo más complejo, como lo son la QFT o el algoritmo de Grover.

El rendimiento del emulador se analizó implementando dos de los algoritmos más importantes de la computación cuántica, que son la transformada cuántica de Fourier para una cantidad de qubits de 1 a 6, y el el algoritmo de búsqueda de Grover para 4..128 elementos. Los tiempos de procesamiento obtenidos del circuito que ejecuta el algoritmo en FPGA se compararon con el mismo algoritmo ejecutado en el microprocesador, y se concluye que las propiedades del paralelismo de la FPGA fueron aprovechadas eficientemente. El tamaño máximo de datos que podemos gestionar está claramente restringido por el tamaño de la FPGA. En nuestro caso, la FPGA coloca de forma óptima el circuito QFT hasta N = 5 y Grover con 64 elementos. Sin embargo, aunque el sistema no tenga éxito al colocar los otros casos mayores, esto se puede resolver fácilmente trabajando en otra FPGA con más recursos.

Por último, aunque los emuladores pueden mejorar el rendimiento de una manera notable en comparación con las simulaciones de software, el principal inconveniente de las emulaciones cuánticas es el aumento exponencial de la cantidad de recursos en función de la cantidad de qubits de entrada. Lejos de ser una situación pesimista es un fenómeno que evidencia aún más la necesidad de una computadora cuántica.

Capítulo 7

Apendice

7.1. Códigos

7.1.1. Matlab

Hadamard

```
function [G]=hadamard(q,l)
q1=2^q;
I=eye(2);
H=(1/sqrt(2)).*[1 1;1 -1];
G=1;
for j=1:q
if j==l
G=kron(G,H);
else
G=kron(G,I);
end
end
G;
```

Rotación

```
function [R]=rotacion(q,g,c,f)
q1=2^q;
R=eye(q1);
k1=zeros(1,q);
for j=1:q1
for k=1:q1
j1=dec2bin(j-1);

for l=numel(j1):-1:1
k1(1,q+1-l)=str2num(j1(numel(j1)+1-l));
end
```

```
if (k1(1,g)==1)&&(k1(1,c)==1)&&(j==k)
R(j,k)=exp(2.*pi.*1i/(2^f));
end
end
end
R;
```

SWAP

```
function [S]=swap(q,c1,c2)
q1=2^q;
S=zeros(q1);
k1=zeros(1,q);
for j=1:q1
k=dec2bin(j-1);
for j1=numel(k):-1:1
k1(1,q+1-j1)=str2num(k(numel(k)+1-j1));
end
if k1(1,c1)==k1(1,c2)
S(j,j)=1;
else
m=k1(1,c1);
n=k1(1,c2);
k1(1,c1)=n;
k1(1,c2)=m;
k2=num2str(k1);
j1=bin2dec(k2)+1;
S(j1,j)=1;
k1=zeros(1,q);
end
end
S;
```

Transformada de Fourier

```
function [t]=transformada(q)
E=entrada(q);
%E=[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]'
%E=[-0.25 0.25*i  -0.25 0.25*i -0.25 0.25*i -0.25 0.25*i -0.25*i -0.25

a=cputime;
T=E;
for j=1:q
T=hadamard(q,j)*T;
for k=j+1:q
```

```
T=rotacion(q,j,k,k-j+1)*T;
end
end
for j=1:q/2
T=swap(q,j,q-j+1)*T;
end
S=T
t=cputime-a;
```

Transformada rápida de Fourier

```
function [E,S]=furia(q)
E=entrada(q);
% E=[0.5*i 0.5*i 0.5*i 0.5*i 0 0 0 0 0 0 0 0 0 0 0 0]'
%E=[-0.25; 0.25*i;  -0.25; 0.25*i; -0.25; 0.25*i; -0.25; 0.25*i; -0.25*
%E=[0.707;0.707;0;0;0;0;0;0;0;0;0;0;0;0;0;0;]

a=cputime;
x=2^q;
T=zeros(x);
for j=0:x-1
for k=0:x-1
T(j+1,k+1)=(1/sqrt(x))*(cos(2*pi*j*k/x)+i*sin(2*pi*j*k/x));
end
end
T;
S=T*E
t=cputime-a;
```

Algoritmo de Grover

```
function D=grover(q)
E = 2^q; %tamaño del vector
%E1 = [3 1 8 17 6 3 5 5] %lista en busqueda

% Creación de la lista con algunos elementos por encima de la condición
for i=1:E
E1(i)= randi(10);
end
E1(3)=18;
E1(7)=19;

% Creación del vector con los elementos que complen la condición (1) o
% (0)
M=0;
for i=1:E
```

```
B(i)=1/sqrt(E);
if E1(i)>15  %condicion
V(i)=1;
M = M+1;
else
V(i)=0;
end
end

% Cálculo de la cantidad optima de iteraciones
K = round (pi/(4*(asin(sqrt(M/E)))))

for i=1:K

%Inversión de fase sobre los elementos que cumplen la condición
for j=1:E
B(j) = (-1)^V(j) * B(j);
end

%Cálculo de la media
A=0;
for j=1:E
A=B(j)+A;
end
A=A/E;

%Inversión sobre la media
for j=1:E
B(j)=2*A-B(j);
end
end


X = max (B);
for i=1:E
if X==B(i)
fprintf(1,'El numero buscado es el %d, en la posicion %d, con probabili
end
end
B;
```

Lectura desde UART

```
s = serial('COM3','BaudRate',115200);
Q = 4;
M = 2^Q;
```

```
fopen(s);
x=[];
for i=1:(3*M+2)
x = [x fscanf(s,'%f')];
end
fclose(s);

for i=1:M
t(i) = i;
ent(i) = x(i);
sal(i) = x(i+M);
salSW(i) = x(i+2*M);
end
tSoft = x(3*M+1);
tHard = x(3*M+2);

ttot = 1:0.25:16;
HWsalida = spline(t,sal,ttot);
SWsalida = spline(t,salSW,ttot);

figure(1)
subplot(3,1,1)
stairs(t,ent,'b','LineWidth',2)
title('Entrada Hardware y Software')
subplot(3,1,2)
plot(t,sal,'b o',ttot,HWsalida,'r','LineWidth',2)
title('Salida Hardware')
subplot(3,1,3)
plot(t,salSW,'b o',ttot,SWsalida,'g','LineWidth',2)
title('Salida Software')

fprintf('QFT Software tardó: %f segundos\n', tSoft)
fprintf('QFT Hardware tardó: %f segundos\n', tHard)
```

7.1.2. Vivado HLS

Transformada de Fourier, TestBench

```
#include <stdio.h>
#include <hls_math.h>

#define L 4 //Cantidad de quits
#define K 2*4 //Dos por la cantidad de qubits
#define Q 2*2*2*2 //Dos elevado a la cantidad de qubits

void Kronecker(float P[2*K], float E[2*Q])
```

```
{
int j,k,u,v,w;
float aux[2*Q];
E[0]   = 1;
E[0+Q] = 0;
v=1;w=0;
for (j=0;j<L;j=j++)
{
u=0;
for (k=0;k<v;k++)
{
aux[u]       = E[k]*P[w]       - E[k+Q]*P[w+K];
aux[u+Q]     = E[k]*P[w+K]     + E[k+Q]*P[w];
aux[(u+1)]   = E[k]*P[(w+1)]   - E[k+Q]*P[(w+1)+K];
aux[(u+1)+Q] = E[k]*P[(w+1)+K] + E[k+Q]*P[(w+1)];
u=u+2;
}
v=v*2;
for (k=0;k<v;k++)
{
E[k]   = aux[k];
E[k+Q] = aux[k+Q];
}
w=w+2;
}
}

void VQFTAXIBUS(float E[2*Q], float S[2*Q]);

int main ()
{
int j;
float P[2*K];
float E[2*Q];
float S[2*Q];

E[0]=0; E[0+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[1]=0; E[1+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[2]=0; E[2+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[3]=0; E[3+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[4]=0; E[4+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[5]=1; E[5+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[6]=0; E[6+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[7]=0; E[7+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[8]=0; E[8+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[9]=0; E[9+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[10]=0; E[10+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
```

```
E[11]=0; E[11+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[12]=0; E[12+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[13]=0; E[13+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[14]=0; E[14+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]
E[15]=0; E[15+Q]=0; // Parte real y parte compleja Er[i] Ei[i+M]

//Kronecker(P,E);
VQFTAXIBUS(E,S);

printf("\nE = \n");
for (j=0;j<Q;j++)
{
printf("%f + i* %f\n",(float)E[j],(float)E[j+Q]); // Parte real y parte
}

printf("\nS = \n");
for (j=0;j<Q;j++)
{
printf("%f + i* %f\n",(float)S[j],(float)S[j+Q]); // Parte real y parte
}
return 0;
}
```

Transformada de Fourier, IPCore

```
#include <hls_math.h>
// #include <ap_fixed.h>
// typedef ap_fixed<16,4> fix_t;

#define DOSPI 6.282958984375
#define M 16 //Cantidad de estados

void VQFTAXIBUS(float E[2*M], float S[2*M])
{
#pragma HLS INTERFACE s_axilite port=return bundle=CRTL_BUS
//Directiva de control
#pragma HLS RESOURCE variable=E core=RAM_1P_BRAM
#pragma HLS INTERFACE bram port=E //Directiva almacenamiento
#pragma HLS RESOURCE variable=S core=RAM_1P_BRAM
#pragma HLS INTERFACE bram port=S //Directiva almacenamiento
int j,k;
float m,n;
for (j=0;j<M;j++)
{
#pragma HLS UNROLL //Directiva de loop
S[j]=0;
```

```
S[j+M]=0;
for (k=0;k<M;k++)
{
#pragma HLS UNROLL //Directiva de loop
m = hls::cos(DOSPI*j*k/M)/hls::sqrt(M);
n = hls::sin(DOSPI*j*k/M)/hls::sqrt(M);
S[j]   = S[j]   + (m)   *   E[k]   - (n)   *   E[k+M];
S[j+M] = S[j+M] + (m)   *   E[k+M] + (n)   *   E[k];
}
}
}
```

Algoritmo de Grover, TestBench

```
#include <stdio.h>
#include <math.h>

#define E 64 // tamaño del vector

void NGrover64(float E1[E], float B[E], float &C);

int main()
{
int i,j;
float X,C;
float B[E],E1[E];

for (i=0;i<E;i++)
{
B[i]=1/sqrt(E);
E1[i] = 5;
}
E1[13] = 19;
E1[24] = 17;
E1[32] = 19;
E1[45] = 17;
C=10; //condicion

NGrover64(E1,B,C);

X = B[0];
for (i=1;i<E;i++)
{
if (X<B[i])
{
X = B[i];
```

```
}
}
j = 0;
for (i=1;i<E;i++)
{
if (X==B[i])
{
printf("El numero buscado es el %f, en la posicion %d, con probabilidad
j = j+1;
}
}
printf("Exito: %f\n", X*X*j);
printf("La cantidad de iteraciones es %f\n", C);
}
```

Algoritmo de Grover, IPCore

```
#include <math.h>

#define PI 3.1416
#define L 64   // tamaño del vector

void NGrover64(float E1[L], float B[L], float &C)
{
#pragma HLS INTERFACE s_axilite port=C bundle=CRTL_BUS
#pragma HLS INTERFACE s_axilite port=return bundle=CRTL_BUS
#pragma HLS RESOURCE variable=E1 core=RAM_1P_BRAM
#pragma HLS INTERFACE bram port=E1
#pragma HLS RESOURCE variable=B core=RAM_1P_BRAM
#pragma HLS INTERFACE bram port=B
int i,j;
bool V[L];
float A,K,M,K1,K2;

M=0;
for (i=0;i<L;i++)
{
#pragma HLS UNROLL
if (E1[i]>C)  //condicion
{
V[i]=1;
M=M+1;

}else{
V[i]=0;
}
```

```
}
K2=sqrt(M/L);
K1=K2+(1/6)*pow(K2,3)+(3/40)*pow(K2,5);
K = floor (PI/(4*(K1)));
//K = 3;
C = K;

for (i=0;i<K;i++)
{
#pragma HLS PIPELINE
for (j=0;j<L;j++)
{
#pragma HLS UNROLL
if (V[j]==1)
{
B[j] = (-1) * B[j];
}
}
A=0;
for (j=0;j<L;j++)
{
#pragma HLS UNROLL
A=B[j]+A;
}
A=A/L;

for (j=0;j<L;j++)
{
#pragma HLS UNROLL
B[j]=2*A-B[j];
}
}
}
```

7.1.3. Vivado SDK

Transformada de Fourier

```
#include <xparameters.h>
#include <xvqftaxibus.h>
#include <stdio.h>
#include <math.h>
#include "AxiTimerHelper.h"

#define PI 3.1416
#define L 4 //Cantidad de quits
```

```
#define K 2*4 //Dos por la cantidad de qubits
#define M 2*2*2*2 //Dos elevado a la cantidad de qubits

float *EHW = (float*)0x40000000;
float *SHW = (float*)0x42000000;

XVqftaxibus doVqftaxibus;
XVqftaxibus_Config *doVqftaxibus_cfg;

void Kronecker(float P[2*K], float E[2*M])
{
int j,k,u,v,w;
float aux[2*M];
E[0]   = 1;
E[0+M] = 0;
v=1;w=0;
for (j=0;j<L;j++)
{
u=0;
for (k=0;k<v;k++)
{
aux[u]       = E[k]*P[w]       - E[k+M]*P[w+K];
aux[u+M]     = E[k]*P[w+K]     + E[k+M]*P[w];
aux[(u+1)]   = E[k]*P[(w+1)]   - E[k+M]*P[(w+1)+K];
aux[(u+1)+M] = E[k]*P[(w+1)+K] + E[k+M]*P[(w+1)];
u=u+2;
}
v=v*2;
for (k=0;k<v;k++)
{
E[k]   = aux[k];
E[k+M] = aux[k+M];
}
w=w+2;
}
}

void VQFTAXIBUS(float E[2*M], float S[2*M])
{
int j,k;
float m,n;
for (j=0;j<M;j++)
{
S[j]=0;
S[j+M]=0;
for (k=0;k<M;k++)
{
```

```
m = cos((2*PI*j*k)/(M))/sqrt(M);
n = sin((2*PI*j*k)/(M))/sqrt(M);
S[j]   = S[j]   + (m)   *   E[k]   - (n)   *   E[k+M];
S[j+M] = S[j+M] + (m)   *   E[k+M] + (n)   *   E[k];
}
}
}


void init_vqftaxibusCore()
{
int status = 0;

doVqftaxibus_cfg = XVqftaxibus_LookupConfig(XPAR_XVQFTAXIBUS_0_DEVICE_I
if (doVqftaxibus_cfg)
{
status =XVqftaxibus_CfgInitialize(&doVqftaxibus,doVqftaxibus_cfg);
if (status != XST_SUCCESS)
{
printf("Error a inicializar doVqftaxibus\n");
}
}
}

int main()
{
init_vqftaxibusCore();

int j;
float P[2*K];
float ESW[2*M];
float SSW[2*M];
float MESW[M], MEHW[M];
float MSSW[M], MSHW[M];
AxiTimerHelper myTimerSW, myTimerHW;

ESW[0]=0; ESW[0+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
ESW[1]=0; ESW[1+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
ESW[2]=0; ESW[2+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
ESW[3]=0.707; ESW[3+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
ESW[4]=0; ESW[4+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
ESW[5]=0; ESW[5+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
ESW[6]=0; ESW[6+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
ESW[7]=0; ESW[7+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
ESW[8]=0; ESW[8+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
ESW[9]=0; ESW[9+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
ESW[10]=0; ESW[10+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
```

```
ESW[11]=0; ESW[11+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
ESW[12]=0; ESW[12+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
ESW[13]=0.707; ESW[13+M]=0; // Parte real y parte compleja Er[i] Ei[i+M
ESW[14]=0; ESW[14+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
ESW[15]=0; ESW[15+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]

EHW[0]=0; EHW[0+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[1]=0; EHW[1+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[2]=0; EHW[2+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[3]=0.707; EHW[3+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[4]=0; EHW[4+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[5]=0; EHW[5+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[6]=0; EHW[6+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[7]=0; EHW[7+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[8]=0; EHW[8+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[9]=0; EHW[9+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[10]=0; EHW[10+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[11]=0; EHW[11+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[12]=0; EHW[12+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[13]=0.707; EHW[13+M]=0; // Parte real y parte compleja Er[i] Ei[i+M
EHW[14]=0; EHW[14+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]
EHW[15]=0; EHW[15+M]=0; // Parte real y parte compleja Er[i] Ei[i+M]

//Kronecker(P,ESW);
//Kronecker(P,EHW);

myTimerSW.startTimer();
VQFTAXIBUS(ESW,SSW);
myTimerSW.stopTimer();

myTimerHW.startTimer();
XVqftaxibus_Start(&doVqftaxibus);
while (!XVqftaxibus_IsDone(&doVqftaxibus));
myTimerHW.stopTimer();

for (j=0;j<M;j++)
{
MESW[j] = sqrt( pow(ESW[j],2.0) + pow(ESW[j+M],2.0) );
MEHW[j] = sqrt( pow(EHW[j],2.0) + pow(EHW[j+M],2.0) );
MSSW[j] = sqrt( pow(SSW[j],2.0) + pow(SSW[j+M],2.0) );
MSHW[j] = sqrt( pow(SHW[j],2.0) + pow(SHW[j+M],2.0) );
}

//printf("\nEntrada \n");
for (j=0;j<M;j++)
{
// printf("ESW = %f + i* %f // EHW = %f + i* %f\n",ESW[j],ESW[j+M],EHW[
```

```
}
//printf("\nSalida \n");
for (j=0;j<M;j++)
{
// printf("SHW = %f + i* %f\n",SHW[j],SHW[j+M]);
}
//printf("QFT Software tardó: %f segundos\n", myTimerSW.getElapsedTimer
//printf("QFT Hardware tardó: %f segundos\n", myTimerHW.getElapsedTimer

for (j=0;j<M;j++)
{
printf("%f\n", MEHW[j]);
}
for (j=0;j<M;j++)
{
printf("%f\n", MSHW[j]);
}
for (j=0;j<M;j++)
{
printf("%f\n", MSSW[j]);
}

printf("%f\n", myTimerSW.getElapsedTimerInSeconds());
printf("%f\n", myTimerHW.getElapsedTimerInSeconds());

return 0;
}
```

Algoritmo de Grover

```
#include <xparameters.h>
#include <xngrover64.h>
#include <stdio.h>
#include <math.h>
#include "AxiTimerHelper.h"

#define PI 3.1416
#define L 64   // tamaño del vector

float *E1HW = (float*)0x40000000;
float *BHW = (float*)0x42000000;

XNgrover64 doNgrover64;
XNgrover64_Config *doNgrover64_cfg;
```

```
unsigned int float_to_u32(float val)
{
unsigned int result;
union float_bytes {
float v;
unsigned char bytes[4];
} data;
data.v = val;

result = (data.bytes[3] << 24) + (data.bytes[2] << 16) + (data.bytes[1]
return result;
}

void NGrover64(float E1[L], float B[L], float C)
{
int i,j;
bool V[L];
float A,K,M,K1,K2;

M=0;
for (i=0;i<L;i++)
{
if (E1[i]>C)  //condicion
{
V[i]=1;
M=M+1;

}else{
V[i]=0;
}
}
K2=sqrt(M/L);
K1=K2+(1/6)*pow(K2,3)+(3/40)*pow(K2,5);
K = round (PI/(4*(K1)));

for (i=0;i<K;i++)
{
for (j=0;j<L;j++)
{
if (V[j]==1)
{
B[j] = (-1) * B[j];
}
}
A=0;
for (j=0;j<L;j++)
{
```

```
A=B[j]+A;
}
A=A/L;

for (j=0;j<L;j++)
{
B[j]=2*A-B[j];
}
}
}

void init_ngrover64Core()
{
int status = 0;

doNgrover64_cfg = XNgrover64_LookupConfig(XPAR_XNGROVER64_0_DEVICE_ID);
if (doNgrover64_cfg)
{
status =XNgrover64_CfgInitialize(&doNgrover64,doNgrover64_cfg);
if (status != XST_SUCCESS)
{
printf("Error a inicializar doNgrover64\n");
}
}
}

int main()
{
init_ngrover64Core();
int i;
float XSW,XHW,C;
float BSW[L],E1SW[L];
AxiTimerHelper myTimerSW, myTimerHW;


for (i=0;i<L;i++)
{
BSW[i]=1/sqrt(L);
BHW[i]=1/sqrt(L);
E1SW[i] = 5;
E1HW[i] = 5;
}
E1SW[15] = 15;
E1SW[24] = 16;

E1HW[15] = 15;
E1HW[24] = 16;
```

```
C=10; //condicion

myTimerSW.startTimer();
NGrover64(E1SW,BSW,C);
myTimerSW.stopTimer();

XNgrover64_Set_C(&doNgrover64,float_to_u32(C));
myTimerHW.startTimer();
XNgrover64_Start(&doNgrover64);
while (!XNgrover64_IsDone(&doNgrover64));
myTimerHW.stopTimer();

XSW = BSW[0];
XHW = BHW[0];
for (i=1;i<L;i++)
{
if (XSW<BSW[i])
{
XSW = BSW[i];
}
if (XHW<BHW[i])
{
XHW = BHW[i];
}
}
for (i=1;i<L;i++)
{
if (XSW==BSW[i])
{
printf("El numero buscado es el %f, en la posicion %d, con probabilidad
}
if (XHW==BHW[i])
{
printf("El numero buscado es el %f, en la posicion %d, con probabilidad
}
}
printf("\nGrover Software tardó: %f segundos\n", myTimerSW.getElapsedTi
printf("Grover Hardware tardó: %f segundos\n", myTimerHW.getElapsedTime
return 0;
}
```

AXI Timer

```
#include "AxiTimerHelper.h"

AxiTimerHelper::AxiTimerHelper() {
```

```
XTmrCtr_Initialize(&m_AxiTimer, XPAR_TMRCTR_0_DEVICE_ID);

m_timerClockFreq = (double) XPAR_AXI_TIMER_0_CLOCK_FREQ_HZ;
m_clockPeriodSeconds = (double) 1/m_timerClockFreq;
}

AxiTimerHelper::~AxiTimerHelper(){
}

unsigned int AxiTimerHelper::getElapsedTicks(){
return m_tickCounter2 - m_tickCounter1;
}

unsigned int AxiTimerHelper::startTimer(){
XTmrCtr_Reset(&m_AxiTimer,0);
m_tickCounter1 = XTmrCtr_GetValue(&m_AxiTimer, 0);
XTmrCtr_Start(&m_AxiTimer, 0);
return m_tickCounter1;
}

unsigned int AxiTimerHelper::stopTimer(){
XTmrCtr_Stop(&m_AxiTimer, 0);
m_tickCounter2 = XTmrCtr_GetValue(&m_AxiTimer, 0);
return m_tickCounter2 - m_tickCounter1;
}

double AxiTimerHelper::getElapsedTimerInSeconds(){
double elapsedTimeInSeconds = (double)(m_tickCounter2 - m_tickCounter1)
return elapsedTimeInSeconds;
}

double AxiTimerHelper::getClockPeriod(){
return m_clockPeriodSeconds;
}
```

Bibliografía

Arizmendi, C. M. y O. G. Zabaleta (2012). «Stability of couples in a quantum dating market». En: *special IJAMAS issue "Statistical Chaos and Complexity* 26, págs. 143-149.

Barenco, Adriano y col. (1995). «Elementary gates for quantum computation». En: *Phys. Rev. A* 52 (5), págs. 3457-3467. DOI: 10.1103/PhysRevA.52.3457.

Cleve, R. y col. (1998). «Quantum algorithms revisited». En: *Proc. R. Soc. Lond.* 454, págs. 339-354.

Deutsch, David Elieser (1989). «Quantum computational networks». En: *Proc. R. Soc. Lond. A* 425.1868, págs. 73-90.

Einstein, Albert, Boris Podolsky y Nathan Rosen (1935). «Can quantum-mechanical description of physical reality be considered complete?» En: *Physical review* 47.10, pág. 777.

Galindo, Alberto y Miguel Angelo Martin-Delgado (2002). «Information and computation: Classical and quantum aspects». En: *Reviews of Modern Physics* 74.2, pág. 347.

Garcia, Hector J e Igor L Markov (2015). «Simulation of quantum circuits via stabilizer frames». En: *IEEE Transactions on Computers* 64.8, págs. 2323-2336.

Hagan, Scott, Stuart R Hameroff y Jack A Tuszyński (2002). «Quantum computation in brain microtubules: Decoherence and biological feasibility». En: *Physical Review E* 65.6, pág. 061901.

IBM (2017). *Paving the Path to Universal Quantum Computing*. URL: https://www.ibm.com/blogs/think/2017/03/ibm-quantum/.

Imre, Sandor y Ferenc Balazs (2013). *Quantum Computing and Communications: an engineering approach*. John Wiley & Sons.

Kanamori, Yoshito y col. (2006). «A short survey on quantum computers». En: *International Journal of Computers and Applications* 28.3, págs. 227-233.

Khalid, Ahmed Usman, Zeljko Zilic y Katarzyna Radecka (2004). «FPGA emulation of quantum circuits». En: *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*. IEEE, págs. 310-315.

Khalil-Hani, M, YH Lee y MN Marsono (2015). «An accurate FPGA-based hardware emulation on quantum fourier transform». En: *Quantum* 1, a1b3.

Kumar, Kunal y col. (2015). «A survey on quantum computing with main focus on the methods of implementation and commercialization gaps». En: *Computer Science and Engineering (APWC on CSE), 2015 2nd Asia-Pacific World Congress on*. IEEE, págs. 1-7.

Lee, Ka Chung y col. (2011). «Entangling macroscopic diamonds at room temperature». En: *Science* 334.6060, págs. 1253-1256.

Leung, Debbie W. y col. (1997). «Approximate quantum error correction can lead to better codes». En: *Physical Review A* 56 (4), págs. 2567-2573. DOI: 10.1103/PhysRevA.56.2567.

Marinescu, Dan C. y Gabriela M. Marinescu (2005). *Approaching Quantum Computing*. Upper Saddle River, NJ: Pearson Prentice Hall.

Piotrowski, Edward W y Jan Sładkowski (2017). «Quantum Game Theoretical Frameworks in Economics». En: *The Palgrave Handbook of Quantum Models in Social Science*. Springer, págs. 39-57.

Qin, Shaodong y Mladen Berekovic (2015). «A Comparison of High-Level Design Tools for SoC-FPGA on Disparity Map Calculation Example». En: *arXiv preprint arXiv:1509.00036*.

Silva, Agustin y Omar Gustavo Zabaleta (2017). «FPGA quantum computing emulator using high level design tools». En: *Embedded Systems (CASE), 2017 Eight Argentine Symposium and Conference on*. IEEE, págs. 1-6.

Ursin, Rupert y col. (2007). «Entanglement-based quantum communication over 144 km». En: *Nature physics* 3.7, nphys629.

Vivado Design Suite User Guide (2016). URL: www.xilinx.com/support/documentation/sw$_$manuals/xilinx2016$_$4/ug893-vivado-ide.pdf.

Zabaleta, O. G. y C. M. Arizmendi (2014). «Quantum Game Techniques Applied to Wireless Networks Communications». En: *Journal of Advances in Applied and Computational Mathematics* 1, págs. 3-7.

Zabaleta, O.G., J.P. Barrangú y C.M. Arizmendi (2017). «Quantum game application to spectrum scarcity problems». En: *Physica A: Statistical Mechanics and its Applications* 466, págs. 455 -461. ISSN: 0378-4371. DOI: http://dx.doi.org/10.1016/j.physa.2016.09.054. URL: http://www.sciencedirect.com/science/article/pii/S0378437116306793.