Departamento de Arquitectura de Computadores
Universidad de Málaga

UNIVERSIDAD
DE MÁLAGA

PH.D. THESIS

Aceleración de la computación en altas prestaciones mediante FPGA

Sergio David Muñoz Capó

Málaga, 29 de Junio de 2017

UNIVERSIDAD DE MÁLAGA

AUTOR: Sergio David Muñoz Capó

http://orcid.org/0000-0002-8480-3067

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga

Dr. Francisco Javier Hormigo Aguilar
Profesor Titular del Departamento
de Arquitectura de Computadores
de la Universidad de Málaga

CERTIFICAN:

Que la memoria titulada *"Aceleración de la computación en altas prestaciones mediante FPGA"*, ha sido realizada por D. Sergio David Muñoz Capó bajo nuestra dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga y concluye la Tesis que presenta para optar al grado de Doctor en Ingeniería de Informática.

Málaga, 29 de Junio de 2017

Fdo: Dr. Francisco Javier Hormigo Aguilar
Director de la Tesis Doctoral

A mis padres

A mis hermanas

A mi futura esposa

Agradecimientos

De mi época como estudiante de doctorado sólo puedo decir que tengo muy buenos recuerdos, la oportunidad que se me dio fue magnífica y de incalculable valor, de esas cosas que jamás se olvidan por lo buenas que fueron.

En el transcurso de la tesis tuve la ocasión de conocer a muchas personas que de algún modo u otro forman parte de esta tesis. Por ello, me gustaría mencionarlas y agradeciéndoles su aportación por minúscula que fuera.

En primer lugar, agradecer a Javier Hormigo por su voto de confianza y permitir que trabajara con él durante unos cuantos años. La verdad que fue privilegio, me encantan esas personas que hace fáciles las cosas más complejas, y si su punto de vista difiere del tuyo no dudan en decírtelo aportando siempre su coherente razonamiento. También he disfrutado buscando soluciones y resolviendo problemas con una persona tan creativa e innovadora. Su espíritu luchador dice mucho de él al igual que su extensa capacidad inventiva.

Agradecer también al departamento de Arquitectura de Computadores de la Universidad de Málaga por la oportunidad ofrecida y confianza depositada en mí, especialmente a Emilio Zapata y al actual director del departamento Oscar Plata. Otra persona a la que doy las gracias por su trabajo es Carmen Donoso, me encanta la energía que siempre presenta y su preocupación por el buen funcionamiento del departamento es Carmen Donoso. La verdad que debo agradecerles a todos los miembros del departamento: becarios, profesores, secretarios, técnicos, etc. en la realización de la tesis, porque en cierto modo, todos ellos has participado en este ciclo. No cito nombres porque necesitaría una gran cantidad de folios para completarlo, pero estoy seguro que al leerlo, ellos perfectamente saben a quienes me refiero.

Si que voy a hacer una especial mención a quienes fueron mis compañeros de laboratorio 2.3.8, esas personas con las que pasábamos juntos nuestro día a día: Francisco J. Jaime, Sonia González, Carlos García, Antonio Muñoz, Iván Romero, etc., la huella que han dejado saben de sobra que perdurará para siempre. También agradecer los años que hemos pasado juntos a nuestros vecinos del laboratorio 2.3.10, en especial a Lidia, Alberto, Miguel Ángel, Ricardo, Pirlo, Nono, Manolo, Juan, Oscar, Antonio, Alejandro, etc. y por ese segundo puesto que nos llevamos como equipo de fútbol sala.

Pasando al terreno personal, quiero agradecerle a mucha gente que de forma consciente o inconsciente han tenido su repercusión en mí. Comenzando por mis padres, Pepe y Toñi, que siempre me han apoyado, luchando mucho para que pueda conseguir cosas como esta. Seguidamente a mis hermanas y mi sobrino, Guedi, Nuria y Víctor, por compartir todos esos momentos tan especiales, esos días en los que uno necesita de esas emociones que sólo saben dar esas personas que han compartido una vida desde niños. No me olvido nunca del gran favor que de una forma u otra mi abuela, mis tíos, y mis primos siempre han sido partícipes.

Dar las gracias a María Jesús, mi Gurifrisquita, quién de sobra ha sabido comprender a qué me enfrentaba, ayudándome y haciendo que todo fuera más sencillo, a pesar de que estos últimos meses casi no haya podido hacer estar con ella todo el tiempo que hubiera deseado. Sinceramente, su comprensión y comportamiento ha sido de 10, siempre dándome ánimos en esos momentos que uno se plantea como continuar.

Dar las gracias también a Paco Jaime y Rosa por la energía tan positiva que siempre desprenden, si no fuera por ellos, estoy seguro que esta tesis se hubiera alargado muchísimo más en el tiempo. He de añadir cómo anécdota de la que Paco es partícipe,a finales de octubre nos encontrábamos cenando en un bar María Jesús, Paco Jaime, mi padre y yo, en la conversación surgió el tema de la tesis, ese día, me abrieron los ojos entre todos, y gracias a ellos, volví al camino del que me había alejado, luchando con más ganas que nunca. Gracias a ellos centré todos mis esfuerzos en conseguir acabarla, así que muchas gracias de corazón.

Para ir acabando, y no por ello menos importante, no me quiero olvidar de esas personas que siempre se ha preocupado por mí, han estado pendientes de cómo llevaba las cosas, ofreciéndome su ayuda, sus favores, etc. y que siempre te dan esos momentos en la vida que sólo una amistad de verdad sabe dar. Sé que si algún día ellos leen esto sabrán de sobra que me refiero a ellos, pero no está de más que cite sus nombres, así que gracias a Reme, Mili, Michele, Rosa, Víctor, Tolo, Noelia, Nacho, Irene, Eli, David, Silvia, Salva, etc.

Gracias a todos de corazón.

Resumen

Actualmente, cada vez es más común ver algoritmos implementados para arquitecturas heterogéneas, en las que se distinguen más de un tipo de elemento procesador. En esta tesis se quieren presentar las increíbles ventajas que pueden aportar las FPGA a este tipo de computación, ya que permiten la posibilidad de actuar como aceleradores de algoritmos.

La principal ventaja de las FPGA es su capacidad de reprogramación, es decir, su diseño lógico no es estático y fijo como ocurre con la mayor parte de los circuitos integrados (ASIC). Esta característica es dada en las FPGAs gracias a que se componente esencial es una matriz de bloques lógicos programables interconectados a través de rutas también programables. Una vez se describe la funcionalidad deseada en un lenguaje de descripción Hardware, el software de programación se encarga de habilitar las conexiones y rutas necesarias para su implementación. Las ventajas son múltiples, ya que permiten construir diseños hardware sin el coste de desarrollo que conlleva una implementación ASIC.

En esta tesis se presentan varias implementaciones en FPGA para dos tipos de algoritmos bastante bien diferenciados: la factorización de matrices QR a través del métodos de rotaciones de Givens y la resolución de los problemas de planificación Job Shop Scheduling (JSSP) a través de un algoritmo genético. En cada uno de los casos se muestran varios aspectos que ayudan a crear arquitecturas en FPGA que permiten optimizar su implementación a nivel de productividad y/o utilización de recursos.

Para la descomposición QR por el método Givens se presentan dos modelos hardware diferentes aplicados a contextos distintos. El primero de ellos cuenta con un diseño iterativo y aplicable a tamaños de matrices que abarca un amplio rango, desde la matriz más simple de un tamaño 4×4 hasta algunas de una envergadura que ronda un tamaño de 256×256. En el se diferencian claramente dos secciones, una encargada de la rotación y otra del cómputo de ángulos de rotación. En cada una de ellas se ha utilizado una configuración CORDIC específica para la computación de la rotaciones y vectorizaciones. Manifestar también la diferencia de cómputo existente entre las matrices Q y R, siendo de un orden superior la primera matriz. Por tal motivo se han analizado arquitecturas descompensadas en cuanto a elementos rotadores, favorable a la sección de la matriz Q que requiere de un mayor número de operaciones.

El segundo modelo destinado a la factorización de matrices QR de tamaño mucho más pequeño con el objetivo de mejorar el rendimiento, utiliza un nuevo tipo de arquitectura sistólica donde se han optimizado los nodos de procesamiento. La arquitectura se presenta como una matriz sistólica bidimensional que hace uso del algoritmo CORDIC segmentado y rediseñado para acelerar el cómputo. Dicho enfoque permite un flujo continuo y una producción constante de factorización de matrices. La arquitectura se ha implementado en formato punto fijo, siendo estudiada y analizada para matrices de un tamaño 4×4 aunque es totalmente exportable a otras dimensiones.

En ambas arquitecturas se ha estudiado la reducción de la cantidad de rotaciones realizadas

por los algoritmos CORDIC, parte esencial y de gran influencia en la precisión de los algoritmos. Resaltar también la importancia del solapamiento a nivel de operaciones que permite reducciones de en los tiempos de procesamiento, como por ejemplo el solapamiento de instrucciones y paralelización del cómputo de los ángulos con las rotaciones.

Por último, recalcar la diferencia en el uso de recursos entre ambas arquitecturas, iterativa y sistólica, siendo esta primera mucho más liviana que la segunda, aunque la capacidad de cómputo es superior en la sistólica.

Otro diseño propuesto en esta tesis es la implementación FPGA para la resolución de problemas de planificación JSSP por medio de un algoritmo genético. Los problemas JSSP son problema NP-completos y pertenecen a unos de los grupos más complejos. En este caso se trata de un problema de minimización de tiempos en el momento de ordenar una secuencia de tareas y trabajos a realizar.

La implementación del algoritmo genético ha sido diseñada por completo, partiendo del formato de representación de la solución, una secuencia de los trabajos a realizar. El diseño se ha realizado utilizando parámetros que permitan ajustar el diseño del problema para conseguir la resolución de múltiples configuraciones, acotando de este modo la cantidad de recursos utilizados.

Para la implementación del algoritmo genético ha sido necesario el diseño de operando específicos para el algoritmo genético óptimos para un desarrollo que permita el solapamiento de instrucciones. Por ello se ha seguido un diseño modular donde todos los módulos se ejecutan de forma paralela. Destacar que en la operación de cruce de soluciones se ha utilizado un operador segmentado que habilita el cómputo paralelo de la posterior evaluación de los datos.

Los resultados para solventar problemas JSSP son muy positivos, puesto que el área ocupada permite un replicación hasta más de 12 veces el mismo algoritmo en la FPGA utilizada (FPGA Virtex-6 XV6VLX240T velocidad -2), obteniendo de este modo hasta 12 ejecuciones paralelas del algoritmo en busca de la solución. Se ha presentado una comparación con una CPU, donde la FPGA se muestra entre 5 y 18 veces más rápida, dependiendo del problema analizado y la cantidad de iteraciones. También, indicar que la cantidad de aciertos ha sido bastante favorable para la FPGA, aunque es cierto que hallar la mejor solución puede depender de la casuística y la probabilidad.

Para finalizar con el resumen, indicar que aún se quedan varios frentes abiertos en los se puede seguir investigando. Como por ejemplo utilizar nuevos tipos de operandos en el algoritmo genético, analizar el consumo de las FPGA, o probar nuevas técnicas que afectan a la mejora del conjunto de soluciones.

Índice

Índice de figuras

Índice de tablas

1 Introducción

En este primer capítulo se desea introducir al lector en el tema, dar los elementos más básicos para comprender los objetivos de esta tesis y el motivo. A lo largo del desarrollo, se comenzará mostrando los aspectos más simplistas para guiar al lector hacia los más complejos, incrementando la complejidad de forma paulatina. Cierto es, que varios apartados ya pueden ser conocidos por el lector, como por ejemplo los conceptos de FPGA, factorización QR, CORDIC, etc., pero es el punto de partida para lograr una coherencia y compresión textual.

1.1 Aceleración de Algoritmos mediante FPGA

Situados en la casilla de inicio, el primer aspecto a definir de forma genérica es el significado del término FPGA (Field Programmable Gate Array). Las FPGA son dispositivos que pertenecen al conjunto de los sistemas embebidos, diseñados para realizar una o algunas pocas funciones dedicadas, frecuentemente utilizadas en sistemas de computación en tiempo real. Ya de inicio, las FPGAs forman un concepto que distan bastante del propósito genérico de un ordenador donde la CPU cubre un amplio rango de necesidades genéricas. En el caso de las FPGA, éstas se centran en potenciar ciertos aspectos y obviar otros, a fin de dedicarse a un cometido más exclusivo.

Añadir que las FPGAs no son los únicos dispositivos embebidos del mercado, también existes los procesadores de propósito general adaptados a placas embebidas, DSP (procesadores específicos para procesamiento de señales), microcontroladores, circuito integrado de aplicación específica (ASICs), etc.

Los dispositivos FPGA permiten la implementación de todo el hardware y software de un sistema digital en un circuito integrado configurable, permitiendo desarrollos conocidos como sistemas programables en el chip, todo ello gracias a su diseño de forma matricial de bloques lógicos

configurables. El desarrollo de sistemas embebidos modernos demanda la implementación de funciones sofisticadas en plazos de diseño cortos, la ventaja de las FPGAs es que posibilitan una gran flexibilidad en el diseño e implementación de estos sistemas.

Pensando en la meta, una de las preguntas que siempre nos cuestionado al crear un algoritmo es: ¿cómo podría modificarlo para que acabe antes?, especialmente en aquellos algoritmos en el que su cómputo sobrepasa la paciencia de uno mismo o bien se requiere duración similar al tiempo real. Durante muchos años, la única opción para acelerar un algoritmo era esperar a la aparición de nuevos procesadores con nuevos tipos de instrucciones y velocidades de procesamiento cada vez más elevadas.

Dicha estrategia funcionó durante un buen tiempo, si embargo cada vez resultaba más inviable, debido entre otros motivos, al elevado consumo de energía[1]. En los últimos 20 años las mejoras han crecido siempre de forma exponencial, sin embargo, desde hace unos años los nuevos diseños se encuentran con barreras difíciles de superar. En la Fig. 1.1 se presentan varios aspectos de gran relevancia, por ejemplo, se aprecia como el rendimiento de la ejecución secuencial está en decremento del rendimiento de la ejecución paralela. Además, se muestra un estancamiento de la potencia y frecuencia, en contra, el número de transistores y núcleos presenta un crecimiento exponencial.

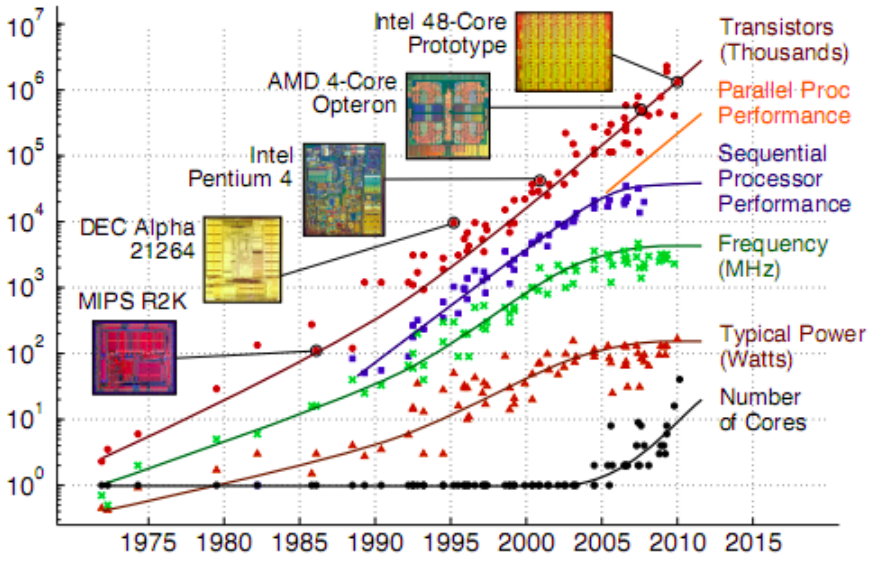


Figura 1.1: Evolución de los procesadores.

Todos estos cambios han llevado a pensar a los grandes expertos en futuras arquitectura de computadores que se tiene que explotar el paralelismo para optimizar los programas. Debido a ciertas barreras que presentan los procesadores apreciables en la Fig. 1.1, todo ello quiere indicar que la solución pasa por alternativas híbridas donde otras arquitecturas mitiguen los problemas de los procesadores. El objetivo es realizar la computación a partir de dos enfoques, una parte encargada del proceso general gestionada por el propio procesador, y otra parte gestionada por un dispositivo distinto que trabaje como acelerador para aquellas tareas específicas que los procesadores no operan de forma óptima.

La aceleración de algoritmos ya existe desde hace tiempo, especialmente para mejoras en la ejecución a nivel de operación en el código fuente con técnicas de ejecución concurrente, desplegar bucles, etc. Sin embargo, otro formato para poder realizar dicha aceleración es utilizar algún

otro medio que mejore la rapidez del algoritmo. Los tres grandes formatos de hoy en día son utilizando Graphics Processing Unit (GPU), Application-Specific Integrated Circuit (ASIC) y Field Programmable Gate Array (FPGA).

Los procesadores gráficos GPUs son arquitecturas que fueron diseñadas para el procesamiento de imágenes gráficas, aunque en la actualidad se han utilizado para propósitos mucho más generales. Inicialmente las GPUs presentaban un procesamiento basado en cauces segmentado cuyas etapas realizaban tareas fijas, explotando el paralelismo a nivel de dato y tarea, aspecto que ha cambiado en la actualidad. A día de hoy su potencial permite computar varios hilos de datos a la vez gracias a su composición escalable de multiprocesadores. Su flexibilidad ha permitido resolver un amplio rango de complejos problemas a través de la aceleración de cómputos a través de la unión de la GPU y la CPU como por ejemplo problemas de química cuántica, modelos atómicos, dinámica molecular, inteligencia artificial, simulaciones, gobierno y defensa, electrodinámica y electromagnetismo, etc.

El otro gran bloque de aceleradores son los Circuitos Integrado para Aplicaciones Específicas (ASIC) que a través de un diseño particular permiten optimizar ciertas partes del procesamiento. Su utilización se centra en optimizar ciertas funciones específicas como por ejemplo chips diseñados únicamente para ser utilizados en telefonía. Los chips ASIC han crecido hasta alcanzar valores cercanos a los 100 millones de puertas lógicas y algunos ya incluyen ligeros procesadores de 32-bits, bloques RAM, etc., denominados System-on-Chip (SoC).

Centrándonos en el acelerador que nos interesa, las FPGAs tienen numerosas aplicaciones en la informática moderna donde uno de los más importantes es la aceleración algorítmica. Sus inicios comienzan en el campo de las telecomunicaciones y redes, pero a lo largo de los años 90, sus aplicaciones en otros campos fueron ampliándose, fortaleciendo sus propiedades como aceleradores [2].

Lo cierto es que las FPGA presentan grandes posibilidades debido a la capacidad de reconfiguración, la ventaja prioritaria de las FPGAs con respecto a otros aceleradores. Sin embargo, el bajo uso de la reprogramabilidad que se les da, hacen que sigan siendo aún demasiado estáticas. Aspecto que ha mejorado extensamente en proyectos como Stratix 10[3] donde se ha demostrado la eficiencia de las FPGAs combinadas con un procesador de bajo consumo ARM.

1.1.1 Futuro de las FPGAs

Las grandes empresas de internet no tienen suficiente con los servidores genéricos basados en el hardware tradicional. Todos ellos comienzan a migrar gradualmente a una estrategia en la que la creación de servidores diseñados a medida permita ganar enteros en cuanto a eficiencia y potencia.

Con la compra de Altera por parte de Intel en el verano de 2015, se demostró el gran interés por parte de Microsoft por la combinación de los chips genéricos de los servidores junto con los chips programables que contienen las FPGAs, una nueva tendencia por la que ha apostado muy fuerte.

Las primeras pruebas iniciadas con el buscador Bing de Microsoft presenta una mayor rapidez en los algoritmos de aprendizaje automático utilizados para las búsquedas, en concreto las búsquedas son hasta 100 veces más veloces. Ello hace decantar la balanza en favor de los procesadores con FPGA incrustadas en lugar de los convencionales chips.

El proyecto denominado Project Catapult [4], iniciado en el año 2010, ha transformado la visión de la empresa en la que se priorizan ciertos servicios en los servidores mejorando su versatilidad, potencia y eficiencia, a través de la combinación de procesamiento CPU y FPGA. De hecho, los

investigadores del proyecto aseguran que en los próximos años prácticamente cada nuevo servidor de Microsoft incluirá un FPGA.

Los esfuerzos del Project Catapult no sólo se han centrado en esas mejoras, también han tenido éxito en la reducción de la infraestructura de servidores en la que se eliminan componentes que no se necesitan. Realmente el coste ha disminuido alrededor de un 30% del resto de componentes en el servidor, consumiendo menos del 10% del consumo total y procesando los datos el doble de rápido que los servidores habituales lo harían.

En el caso de la empresa Google fue de las primeras en mostrar cómo sus servidores podrían tener grandes ventajas para sus operaciones en la red. Sus servidores se convirtieron en uno de los secretos mejor guardados, donde el principal factor que se potenció fue el hardware [5]. Google ya no quiere mejorar a través de sus propios servidores, ahora está trabajando en nuevos diseños de microprocesadores heterogéneos para gobernarlos.

1.2 Motivación, metas y logros de la Tesis

Una vez presentado qué es una FPGA y conocidos sus principales aplicaciones, ya se dispone de los conocimientos para poder entender y comprender las motivaciones de esta tesis.

El primer punto es aclarar el motivo por el que hemos decidido utilizar un dispositivo FPGA en vez de los procesadores comunes a los que tan acostumbrados estamos. La respuesta es sencilla y simple, aunque la eficiencia de un procesador es indiscutible, todos ellos cuentan con una serie de instrucciones y estructuras prediseñadas que en algunos cómputos pueden ir en su contra. Por ejemplo, en aquellas secciones donde un algoritmo se repite una y otra vez de forma iterativa y además cuenten con cómputos muy específicos en el que las cachés presentan una eficiencia nula ante tales cálculos.

La amplia versatilidad de las FPGAs y su facilidad para la especialización de componentes en cálculos específicos son características magníficas para poder realizar una arquitectura personalizada; de este modo es fácil conseguir tiempos de respuesta más cortos a través de este asequible medio. Optimizando el algoritmo, o partes de él, a través de una FPGA conseguimos potenciar aquellos aspectos donde la CPU flaquea, consiguiendo que la combinación CPU y FPGA presente una mejor eficiencia.

Una vez alcanzado este punto ya se puede entrever la motivación y el porqué de la utilización de FPGAs. Las FPGAs pueden servir como acelerador de algoritmos, por lo que se pretende aplicar este concepto en algunos campos. Es complicado conseguir estrategias de diseño general, pero resultará de gran utilizad extraer pautas a partir de la experiencia y experimentos que ayuden en futuros diseños aplicados a múltiples campos. Ahora falta conocer en qué campos y qué contribuciones han sido posibles, anunciar de antemano que las aportaciones que se van a enunciar ahora han tratado temáticas muy distintas.

El grueso principal ha sido para una metodología de gran importancia en el álgebra lineal, el algoritmo de factorización de matrices QR a través de rotaciones de Givens. El objetivo es experimentar con dicha metodología desde varios ámbitos y perspectivas con el fin de exprimir al máximo el rendimiento y recursos en el momento de la factorización, a fin de buscar nuevas formas de optimizar la factorización de matrices utilizando un dispositivo FPGA. Por ese motivo, será habitual ver comparaciones con otras arquitecturas basadas en FPGA y con la propia CPU.

Inicialmente las aportaciones parten de una base muy simplista donde los algoritmos de factorización QR son aplicados tal cual se han descrito matemáticamente, creando un hilo conductor

del que se ha conseguido datos muy positivos.

Entre todos los algoritmos utilizados para la factorización QR de matrices, el algoritmo de Givens ha sido el seleccionado para realizar un estudio a fin de optimizar varios diseños sobre un dispositivo FPGA. Se han desarrollado múltiples configuraciones y pruebas del algoritmo QR sobre la FPGA diferenciados por su modo de operación, distinguiéndose entre algoritmos iterativos y algoritmos sistólicos. Hacer hincapié en la efectividad que cada uno de los tipos de algoritmos tiene según el tamaño de matriz al que hacen frente. En el primero de ellos se ha trabajado con un tamaño de matriz mucho mayor que en los sistólicos, en contraposición se han desarrollado varios algoritmos y configuración para la factorización de matrices de un menor tamaño. Ambos tienen sus ventajas e inconvenientes, donde el tamaño de la matriz afecta directamente a los tiempos de resolución y capacidad de factorización. De este modo, un algoritmo sistólico es aconsejable para matrices pequeñas e un índice de factorización elevado, mientras que para conseguir factorización una matriz de grandes dimensiones se recomienda utilizar un algoritmo iterativo. Otra característica destacable es la parametrización de los distintos algoritmos, gracias a ella es posible crear un algoritmo que utilice los recursos de la FPGA de una forma óptima, sin necesidad de acaparar excesivos componentes para su posterior inactividad.

Otra meta que se ha presentado durante la tesis ha sido poder probar y estudiar nuevos formatos numéricos que mejoran las técnicas aritméticas para conseguir mejores resultados. En este caso hemos probado una técnica redondeo numérico hacia el valor más cercano basado en una nueva representación numérica llamada HUB (Half-Unit Biased). Gracias a la cual ha sido posible ahorrar hasta un bit y reducir la utilización de recursos den la FPGA obteniendo los mismos resultados.

Por último, se ha realizado un trabajo muy diferente al anterior, se ha investigado sobre un problema de optimización NP-completo denominado Job Shop Scheduling (JSSP) a través de algoritmos genéticos para lograr su resolución. El propósito del JSSP es asignar de forma óptima un conjunto limitado de trabajos a un conjunto cerrado de máquinas para su realización, todo ello en el menor tiempo posible. El objetivo en este caso es resolver este tipo de problemas que presentan una complejidad computacional elevada en dispositivos FPGA a fin de conseguir una aceleración en los tiempos de resolución y optimización en la búsqueda de soluciones. Se han conseguido resultados muy competentes en cuanto a búsqueda de soluciones y tiempos de procesamiento, especialmente para la resolución de problemas de dimensiones grandes.

1.2.1 Objetivos principales

A modo de resumen se quiere resaltar los objetivos principales de esta tesis obtenidos a partir de las motivaciones y metas definidas anteriormente:

- Demostración de las ventajas del uso de dispositivos FPGAs como aceleradores en la computación de algoritmos, implementando dicha aceleración sobre diversos tipos algoritmos bastante dispares.

- Estudio del algoritmo de factorización QR de matrices a través del método de las rotaciones de Givens en dispositivos FPGA.

- Optimización e implementación del algoritmo de factorización QR de Givens en FPGA con distintos formatos, sistólicos e iterativos, y distintos objetivos, matrices pequeñas y de tamaño medio, para contrastar las ventajas o inconvenientes de cada uno de ellos.

- Creación de diseños parametrizable y ajustables con el objetivo de poder sintetizar algoritmos que consuman el espacio justo para la resolución de problemas, aprovechando los recursos justos para ello.

- Diseño e implementación de algoritmos genéticos parametrizables capaces de solventar problemas de Job Shop Scheduling en dispositivos FPGA, actualmente escasos en la literatura. Se pretende diseñar los operandos de forma modular a fin de poder ser modificados por otros futuros. Como nota adicional se quiere hacer uso de una batería de problemas definida por Lawrence cuyas soluciones son conocidas.

- Obtención de pautas de diseño que favorezcan la creación de nuevos algoritmos que mejoren el rendimiento computacional y ocupacional en los dispositivos FPGA, con el objetivo de lograr una mayor paralelización operacional optimizando los recursos utilizados de los dispositivos.

- Estudio, prueba e implementación del novedoso formato numérico HUB que aporta un valor adicional en las operaciones matemáticas, consiguiendo el redondeo computacional de forma sencilla y mitigando tiempos de procesamiento a su vez.

1.2.2 Contribuciones principales

A lo largo de esta tesis se han generado las siguientes contribuciones científicas:

- El artículo [52] denominado "FPGA implementation of QR decomposition for medium size matrices" fue presentado en la conferencia XXVIIth Conference on Design of Circuits and Integrated Systems en el año 2012 que tuvo lugar en Avignon. En él se presenta una estructura parametrizable básica para FPGA que permite factorizar un amplio rango de tamaños de matrices a través del método de QR de Givens. Es el hilo conductor que sirve como punto de partida de nuevos trabajos presentados en esta tesis.

- En el artículo [61] denominado "High-throughput FPGA implementation of QR decomposition" fue publicado en la revista IEEE Transactions on Circuits and Systems II: Express Briefs en Septiembre de 2015. En este artículo se presenta una novedosa arquitectura sistólica parametrizable capaz conseguir la factorización QR de matrices siguiendo el método de Givens sobre matrices 4×4. Las ventajas presentadas son el ahorro varias etapas sobre dicho algoritmo además de una arquitectura que ahorra recursos en comparación con otras. Dicha arquitectura se basa en arrays sistólicos y el diseño especial de un algoritmo CORDIC que permite aplicar la vectorización y rotación en ciclos de reloj continuos. Destacar el elevado rendimiento presentado y la gran cantidad de matrices que es capaz de factorizar el diseño propuesto.

- En artítulo [71] denominado "Improving fixed-point implementation of QR decomposition by rounding-to-nearest" fue presentado y publicado en el congreso IEEE International Symposium on Consumer Electronics (ISCE) a mediados de 2015 en Madrid. En la presentación se muestra cómo aplicando el formato numérico HUB permite ahorrar un bit menos en una arquitectura basada en FPGA capaz de factorizar matrices de tamaño 4×4 a través del método Givens QR, todo ello gracias a que consigue el redondeo numérico en lugar del típico truncamiento de una forma sencilla.

1.2.3 Metodología de trabajo

En el desarrollo de este trabajo se han utilizado distintos enfoques metodológicos de los que se pueden abstraer cuatro grandes fases por la que todas las aportaciones han transcurrido.

1. **Documentación y Análisis**. Es el paso previo que todo nuevo proyecto requiere, antes de comenzar con cualquier investigación, se han captado todas esas ideas que circulan por los diversos medios como son: artículos, sitios web, capítulos de libros, etc. para ver cómo conectan con el proyecto que queremos llevar a cabo. En este punto se agradecen tener herramientas software que aporten validez a los diseños, y ya de paso, sirvan para contrastar que nuestro modelo es factible. Por ejemplo, diseñar un prototipo de los algoritmos a través de las herramientas software como Matlab o entornos de programación son de utilidad para realizar el diseño sobre la FPGA.

2. **Diseño y Modelado**. Una vez conocido qué se quiere realizar, el siguiente paso es trabar en el cómo. En los diversos modelados siempre se ha trabajado con una filosofía modular o por bloques. Para ello, nos hemos servidos especialmente de un esquema dónde se distinguen los diversos bloques diferenciados por sus funcionalidades, el objetivo es mostrar la arquitectura de diseño desde un punto de vista general formado por cajas negras. Aunque cada uno de los módulos es desarrollado posteriormente a través de diversos gráficos fundamentados en una descripción de los componentes requeridos junto con sus correspondientes diagramas de flujo. Los diagramas de flujos has resultado de especial interés en el momento del diseño de máquinas de estados encargadas del control. La división de los algoritmos en partes aporta compresión en los diagramas y esquemas, por lo que se puede decir que nos hemos decantado por diseño conocido como Top-Bottom, en el que se realiza el prototipo desde un alto nivel de abstracción, subdividiendo el diseño jerárquicamente en nuevos sub-componentes.

3. **Desarrollo**. Este es el instante en el que se comienza a construir una primera versión implementada mediante el lenguaje de modelado hardware VHDL. En este punto la experiencia es de vital importancia, ya que en la construcción o programación hay factores que pueden afectar al comportamiento de la FPGA. Es importante trabajar en un diseño en el que la sincronización sea satisfactoria y permita un funcionamiento correcto de nuestro diseño. Realmente, en este paso no hay que centrarse sólo en conseguir el funcionamiento correcto, se ha de llegar a un escalón más, hay que pensar en el desarrollo para que el algoritmo se comporte de la mejor forma en el dispositivo FPGA, aprovechando al máximo los recursos y optimizando el rendimiento. También es importante conocer cuál va a ser el medio de transferencia de información y señales al dispositivo FPGA, el que es de vital importancia conocer para un correcto desarrollo y funcionamiento.

4. **Análisis y validación** Una vez llegado a este punto, ya hay que comenzar a contrastar y valorar resultados. Evaluar si el funcionamiento es el esperado, definido en paso inicial. Es el punto en el que se lleva a cabo una gran variedad de pruebas para poder juzgar la efectividad y eficiencia del diseño en el que la objetividad es quién gobierna.

Añadir en última instancia que el diseño de algoritmos sobre un dispositivo hardware acostumbra a seguir las pautas anteriores de forma escalonada, aunque es cierto, que la construcción del algoritmo presenta un comportamiento cíclico donde en cada iteración se busca el perfeccionamiento y mejora del algoritmo. La ventaja de trabajar mediante iteraciones es que ya se dispone de una base con la que es posible realizar una comparación, de tal forma que al realizar una mejora en los pasos anteriores nos permite acercarnos hacia la solución óptima.

1.3 Estructura de la Tesis

La tesis continua con los siguientes capítulos:

Capítulo 2. Conocimientos previos. En este capítulo se introducen las bases para la comprensión de varios aspectos de la tesis. Inicialmente se presenta qué es una FPGA, donde se definen sus ventajas y desventajas. Además presenta la idea de reconfiguración, optimización de recursos, latencia, etc., aspecto más que importante en el diseño de arquitecturas hardware. También se lleva a cabo una introducción a la factorización de matrices QR, justificando el uso de las rotaciones de Givens; una de las piezas fundamentales para completar esta tesis. En la parte final, se presenta otro gran pilar, el algoritmo CORDIC, en él se muestran las distintas variantes e implementaciones. Es importante conocer de cerca el funcionamiento del algoritmo CORDIC, ya que en los siguientes capítulos se ha optimizado para el beneficio de nuestra propuestas.

Capítulo 3. Implementación iterativa para la descomposición QR sobre FPGA. En este apartado se presenta la primera aportación o propuesta para la descomposición QR de matrices. Para ello se ha hecho uso de una arquitectura iterativa, donde con un pequeña y limitada cantidad de recursos se consiguen la factorización a través de las rotaciones de Givens de tamaños de matrices muy elevados. El sincronismo entre componentes se realiza por medio de una unidad de control que dirige las dos secciones principales del diseño, una para el cálculo de ángulos y otro para la rotación de filas. Los resultados muestran como una arquitectura desbalanceada computacionalmente en favor de la matriz Q favorece al algoritmo.

Capítulo 4. Implementación sistólica para la descomposición QR sobre FPGA. En este capítulo se presenta una arquitectura distinta a la propuesta anterior con el objetivo de resolver el mismo problema, pero con una perspectiva distinta. En este caso la factorización se realiza consiguiendo un altísimo rendimiento con altas prestaciones, ya que la factorización de matrices alcanza una cantidad de más de 37 millones de matrices por segundo gracias al diseño sistólico. También se ha creado un rotación especial basada en el algoritmo CORDIC capaz de vectorizar y rotar elementos de forma paralela, permitiendo de este modo realizar dicho cálculo con un único componente. Además, se presenta un completo análisis de la medición de error y elección de variables que condicionan el circuito.

Capítulo 5. Descomposición QR sistólica sobre FPGA con formato HUB. En este apartado se presenta un novedoso formato de representación numérica denominado HUB. Su principal ventaja es que permite realizar el redondeo de forma muy sencilla gracias a su tipo de representación que utiliza un bit implícito a la derecha. Asimismo, permite ahorrar parte del hardware ya que las restas se simplifican cambiando la función del complemento a dos por una simple negación sin ningún tipo de acarreo extra. Su aplicación ha sido efectiva para nuestra propuesta sistólica con beneficiosos resultados, ya que se demuestra que es posible alcanzar los mismos resultados ahorrando un bit y minimizando el retardo máximo, en comparación con las propuestas anteriores.

Capítulo 6. Resolución de JSSP a través de GA en FPGA. En el capítulo se presenta una proposición propia para la resolución de problemas de planificación a través de algoritmos genéticos con base FPGA. La resolución de problemas JSSP no es sencilla, pero la arquitectura presenta muy buenos resultados tanto como por el acierto como en el rendimiento de la implementación. Los recursos que utilizan, a lo sumo, alcanzan un 8% de la capacidad de la

FPGA con una velocidad de procesamiento que ronda los 200 Mhz, positiva consecuencia producida por el óptimo diseño de los operandos. Los operandos están diseñados como módulos que pueden ser intercambiables por nuevos formatos sin que tenga repercusión en el algoritmo. Además, se ha realizado un diseño parametrizable en función de las dimensiones del problema con el objetivo de ajustar los recursos necesarios al diseño, sin excederse en hardware innecesario. También se ha realizado una comparación de la arquitectura propuesta con el mismo procesamiento en un CPU, mostrando un rendimiento computacional superior por parte de la FPGA, en concreto entre 18 y 6 veces más eficiencia computacional dependiendo del tamaño del problema.

Capítulo 7. Conclusiones. Es el capítulo final de la tesis dónde se muestran las conclusiones alcanzadas. Se ha dividido en tres secciones, una primera que agrupa las conclusiones generales. Las otras dos restantes enuncias varias conclusiones para la factorización QR de matrices y para la resolución de problemas de planificación.

2 Conocimientos previos

Debido al formato de la tesis y con el fin de proporcionar una mejor ubicación de los antecedentes y trabajos relacionados, este capítulo ha sido desmenuzado en varias secciones divididas por las distintas temáticas a tratar.

En la tesis se distinguen dos bloques bien diferenciados, por un lado la secciones que tienen que ver con la factorización QR de matrices y por otro la construcción de algoritmos genético para la resolución de problemas de Job Shop Scheduling (JSSP). Para alcanzar una compresión de los posteriores capítulos es necesario tener una compresión de las siguientes secciones.

Las secciones Dispositivos FPGA, Introducción a la Descomposición QR, Descomposición QR mediante rotaciones de Givens, Algoritmo CORDIC y Arquitectura generales del algoritmo CORDIC son necesarios para la compresión de los capítulos 3 Implementación iterativa para la descomposición QR sobre FPGA, capítulo 4 Implementación sistólica para la descomposición QR sobre FPGA y capítulo 5 Descomposición QR sistólica sobre FPGA con formato HUB.

Las secciones Problemas de planificación (Scheduling) y Descripción de la problemática Job Shop Scheduling presentan conceptos claramente orientados al capítulo 6 denominado Resolución de JSSP a través de GA en FPGA. En estos apartados se presenta qué tipo de problemas se van a resolver y cómo se a hacer. También se muestran con un ejemplo muy sencillo como se valora una solución a un problema JSSP, dando a conocer también un grupo de problemas tipo utilizado en la literatura.

2.1 Dispositivos FPGA

El vocablo FPGA es el resultante del acrónimo inglés Field Programmable Gate Arrays. Se distingue por ser un dispositivo formado por circuitos integrados donde se diferencian bloques lógicos cuya interconexión y funcionalidad puede ser configurada a través de lenguajes de descripción especializados llamados HDL (Hardware Description Language). En términos generales, una FPGA se describe como una matriz de bloques lógicos donde su interconexión es programable.

Desde la primera FPGA comercial desarrollada por los cofundadores de Xilinx, Ross Freeman and Bernard Vonderschmitt a mediados de los 80, el crecimiento en la industria ha sido exponencial. El principal motivo de ello se debe a que las FPGAs combinan lo mejor de los circuitos integrados de aplicación específica (ASICs) con sistemas basados en procesadores, y todo con una reprogramación flexible y reutilizable.

Además de las ventajas anteriores, las FPGAs ofrecen una elevada velocidad y fiabilidad hardware que huyen del alto gasto inicial que requieren un diseño ASIC. También, presentan una naturaleza claramente paralela donde a diferencia de los procesadores/multiprocesadores no existe una competición por los recursos, ello están implícito en el diseño. De este modo, los cómputos en una FPGA están asignados a una parte del chip y puede ejecutarse de forma autónoma sin ser afectados por otros bloques lógicos.

Es importante destacar varios aspectos que muestran la relevancia de las FPGAs sobre otros componentes:

- Elevada fiabilidad en los diseños que se convierten en circuitos personalizados aportando un excelente rendimiento. Además, sus aplicaciones se posibilitan con mayor facilidad gracias a la cantidad de herramientas que las acompañan, como por ejemplo simuladores, software de testeo, enrutadores, software de análisis, etc.
- Alto nivel de determinismo en la ejecución de algoritmos a velocidades muy elevadas, donde la frecuencia del reloj toma gran importancia, por lo que su rendimiento en cada ciclo es máximo. A pesar de sus elevados cómputos por ciclos, esto no es un problema para una implementación segura y robusta de un circuito, junto con su posterior puesta en marcha.
- Mejora de la concurrencia gracias a un verdadero paralelismo en la ejecución de operaciones donde se facilita en los diseños la segmentación o pipeline a nivel de instrucción.

Una vez citadas las principales ventajas que muestran las FPGAs, existen varios puntos donde su debilidad es notoria. Por ello, a continuación se darán a conocer aquellos más relevantes:

- Uno de las limitaciones más importantes es la cantidad de recursos que existe en una FPGA, totalmente ligado al diseño de dicha FPGA o modelo. Los recursos son los elementos entorno a los que gira la implementación de los circuitos, una óptima utilización de éstos es de vital importancia ya que son la pieza clave para el diseño. Es un factor determinante para realizar un diseño, ya que los recursos están acotados.
- Otro aspecto negativo de la FPGA es el elevado consumo energético en comparación con los circuitos integrados para una aplicación específica conocidos como ASIC (Application-Specific Integrated Circuit). Desventaja que se convierte en ventaja cuando se compara el consumo con una CPU, ya que las FPGA tienen un consumo relativamente inferior.
- Nuevamente al compara una implementación FPGA con una implantación ASIC, la FPGA presenta un buen ratio coste por unidad para una única producción, es decir para el diseño

de cantidades reducida. Sin embargo, si la cantidad producida es elevada, las FPGAs incrementan tanto el coste del producto que no pueden competir con los diseños basados en ASIC.

- Desde la perspectiva de diseño e implementación, las FPGA son un poco más complejas de programar en comparación con los microcontroladores. Asimismo, la portabilidad de diseños no es totalmente trivial, es más, está altamente ligada al modelo o familia de la FPGA.

2.1.1 Arquitectura General de la FPGA

La arquitectura más habitual de las FPGAs se fundamenta en una serie de bloque lógicos, denominados CLB (Configurable Logic Block) o slices, que siguen una estructura de rejilla o matriz a lo largo del chip. Los slices son una de las piezas más importantes de la FPGA, su cantidad y diseño tendrá repercusión en múltiples aspectos del dispositivo. La estructura bidimensional en la FPGA tiene su razón lógica, permitir la comunicación entre los bloques lógicos a través del mar de interconexiones programable.

En la Fig. 2.1 se muestra un ejemplo de cómo están conectados las celda lógicas a través de una estructura bidimensional, además de otros componentes descritos a continuación. Realmente, existe una jerarquía de interconexión bloques programables para permitir la conexión según la necesidad de diseñador del sistema, permitiendo la reutilización.

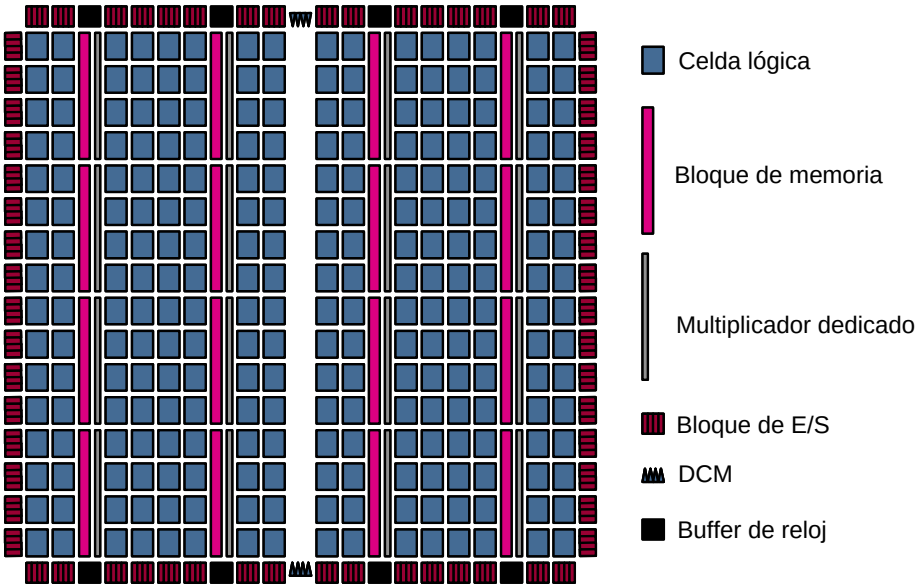


Figura 2.1: Arquitectura General de una FPGA.

Los componentes de la FPGA están directamente relacionados con el productor o vendedor, pero existen ciertos componentes comunes para varios tipos de arquitecturas y modelos. Es verdad que en el listado que se va a presentar no son exactamente iguales o tal vez no son nombrados de ese modo, pero sí convergen en funcionalidades comunes.

Es relevante y razón de peso conocer los componentes de los que se dispone en una FPGA, ya que para realizar cualquier nuevo diseño se debe de conocer de antemano las unidades de recursos de las que disponemos. También es importante conocerlos para saber cómo optimizar su utilización

el beneficio del sistema.

Los principales componentes de una FPGA son listados a continuación:

- LUT's (Look Up Table).
- Flip-Flops, biestables y lógica combinacional.
- Multiplexores.
- Terminales de entrada/salida a través de celdas especiales de E/S.
- Multiplicadores y Segmentos DSP.
- BRAM (Bloques de Memoria RAM).
- Recursos de interconexión programable.

En un FPGA toda la lógica combinacional se implementa a través de la utilización de tablas de lógicas o LUTs. Las LUTs son una conjunto de puertas lógicas cableadas entre sí que sirven para almacenar/computar una lista predefinida de salidas a partir de una combinación de entradas. Básicamente su comportamiento es similar al de una función lógica. Véase la Fig. 2.2.

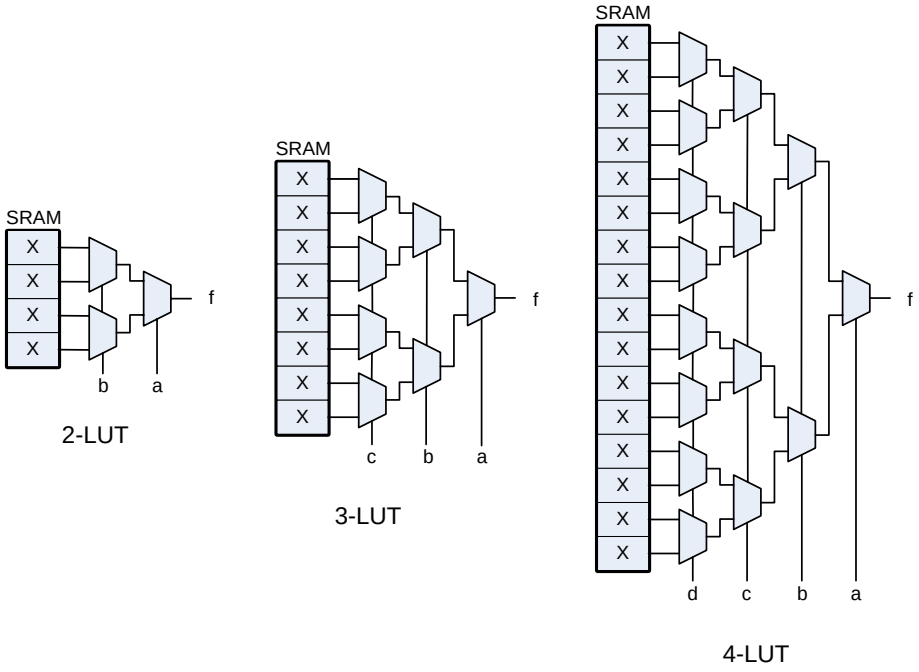
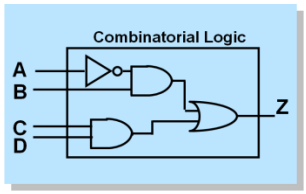


Figura 2.2: Descripción del componente LUT de 2, 3 y 4 variable.

En la Fig. 2.3 se detalla la similitud entre una tabla de verdad y una LUT. La columna de valores Z, valores de la función combinacional, son los valores que realmente se almacenan en la LUT de cuatro variables. Vale la pena recordar que, salvo razones para hacerlo manualmente, el almacenamiento de los valores en las tablas LUTs lo realiza el Software del fabricante del FPGA, siendo el proceso totalmente transparente al diseñador del sistema digital.

Cada salida de la LUT puede encontrase de forma opcional conectado a un Flip-Flop o biestables. Éstos forman un circuito capaz de almacenar uno de los dos estados posibles de forma estable, es decir, representa el dato equivalente a un bit. Pueden ser síncronos o asíncronos, aunque lo más



A	B	C	D	Z
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
	.	.	.	
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figura 2.3: Ejemplo de implementación de una función combinacional en LUT.

habitual es que estén temporizados por una señal de reloj debido a un flanco de subida, bajada o ambos. Los más habituales y conocidos son los flip-flop de tipo D y Q.

Un multiplexor, es un componente que permite seleccionar entre dos o más entradas para escoger una única salida.

Aunque los componentes están bien distinguidos, una agrupación de recursos lógicos tiene especial relevancia en el mundo de las FPGAs, los denominado slices. Los slices están formados por un agregado de dos LUTs, dos Flip-Flops, cuatro salidas donde hay dos combinacionales y dos registros, entradas/salidas, acarreo y multiplexores, siendo específicos en cada familia de FPGA. En los slices se realiza la mayor parte de la funcionalidad de la FPGA y suelen estar agrupados de 2 en 2 o 4 en 4 formando bloques lógicos configurables denominados CLBs. En la Fig. 2.4 se muestra un ejemplo de slice genérico.

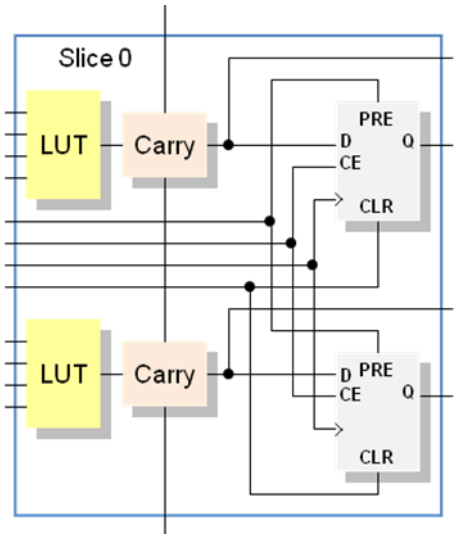


Figura 2.4: Descripción genérica del componente slice.

Regresando al listado de componentes de una FPGA, los bloques de Entrada/Salida son los recursos lógicos que permiten el paso de una señal hacia dentro o hacia el exterior del dispositivo, utilizando registros especiales para ello denominado Buffer E/S.

Los bloques de Memoria RAM (BRAM) son bloques especiales dentro de la FPGA cuyo obje-

tivo es almacenar información tanto para lectura cómo para escritura. Se presenta como una matriz de datos direccionada por índice configurable por el tamaño de palabra utilizado. La cantidad de bloques RAM depende del fabricante, al igual que el tamaño de estos. Por ejemplo para una FPGA Virtex 6 existen dos opciones de tamaño, 18 Kb y 36 Kb utilizándose 36 bits de direccionamiento para los 18 Kb y 72 bits para los bloques de 36 Kb.

Un detalle muy importante es el doble puerto de lectura y escritura que presentan los BRAM, posibilitando realizar dos operaciones simultáneamente e independientes. De acuerdo con el diseño las función más habitual con que nos encontramos son las siguientes: como memoria estática ROM y de sólo lectura, como memoria dinámica RAM con lectura y escritura, apilador de datos FIFO o LIFO, buffer circulares y registros de desplazamiento, soportado por infinidad de combinaciones de anchos de palabra y tamaños del bus de direccionamiento.

Los BRAM son de gran ayuda en el caso de las aplicaciones que requieres de un gran almacenamiento de datos para lectura y escritura, por ejemplo para el almacenamiento de variables durante cálculos matemáticos, almacenamiento de los coeficientes de un filtro FIR, registros de desplazamiento, memorias direccionables por contenido de alto rendimiento (CAM), etc.

Los recursos de interconexión programable son aquel conjunto de líneas e interruptores configurables a través de los cuales las señales se transmiten entre los propios bloques lógicos de la FPGA, en otras palabras, interconectan los bloques slices entre sí con los multiplicadores DSP, con los bloques RAM, etc. Además, se encargan de trazar la ruta entre los bloques lógicos internos y los de entrada/salida. Este paso es importante porque el rutado influye sobre el diseño hardware.

Las FPGAs cuentan también con componentes específicos pensados para resolver cálculos matemáticos complejos de forma eficiente denominados DSP (Digital Signal Processor). Según el fabricante y modelo su nomenclatura puede variar, al igual que el diseño del circuito aritmético. Por ejemplo para un dispositivo Virtex-4 se les denomina DSP48 mientras para que para una Virtex-5 se les denomina DSP48E1 en los que se han mejorado sus características. La Fig. 2.5 muestra a modo de ejemplo la arquitectura de un DSP48E1.

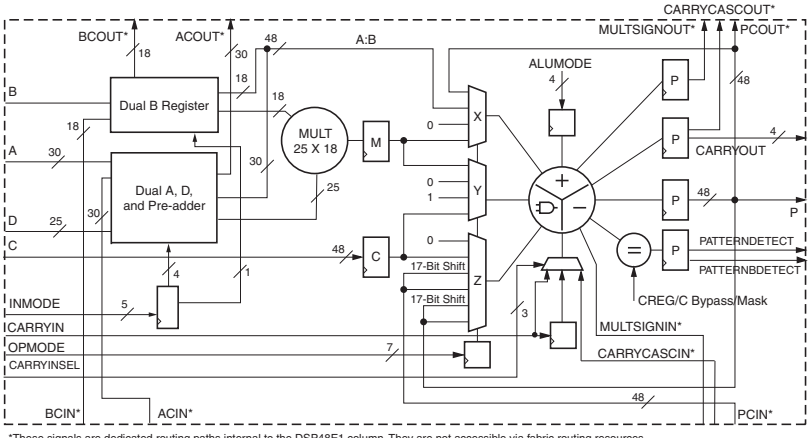


Figura 2.5: Descripción del componente DSP48E1.

2.2 Introducción a la Descomposición QR

En este apartado se mostrará en qué consiste la descomposición QR de una matriz y las ventajas de su rápida convergencia y estabilidad. Además enunciaremos varías metodologías a partir de las cuales conseguir dicha factorización.

En álgebra lineal, una factorización de una matriz es la descomposición de la misma como producto de dos o más matrices según una forma canónica. Por definición, toda matriz $A \in \mathbb{R}^{n\times m}$ con un $rang(A) = m$, es decir, las columnas forman un conjunto linealmente independiente, puede factorizarse como un producto de dos matrices. Como enuncia el teorema 2.2.1 para el caso de la descomposición QR, una primera matriz Q de tamaño $n \times m$ es de tal forma que sus columnas forman un conjunto ortonormal y una segunda matriz denominada R de tamaño $m \times m$ que es una matriz triangular superior y no singular (diagonal con elementos distintos de cero).

Teorema 2.2.1. Dada $A \in \mathbb{R}^{n\times m}$ co $rang(A) = m$, la descomposición QR de A coincide con la siguiente factorización, $A = Q \cdot R$ donde $Q \in \mathbb{R}^{n\times m}$ y $R \in \mathbb{R}^{m\times m}$ tales que $Q^t \cdot Q = Q \cdot Q^t = I$ y R es una matriz triangular superior.

Las principales aplicaciones de la descomposición QR en el álgebra lineal se centran en la resolución de sistemas de ecuaciones lineales, resolución de problemas de mínimos cuadrados, cálculos de pseudo-inversa de una matriz, cálculo de autovalores, etc. De todas las aplicaciones citadas, la resolución de sistemas de ecuaciones es la que presenta un mayor peso. Por ello, se ha decidido hacer una breve descripción del proceso de resolución de la ecuación $A\overrightarrow{x} = \overrightarrow{b}$ en el Algoritmo 2.2.1.

Algoritmo 2.2.1. Algoritmo de resolución de sistemas a través de la factorización QR.

1. $A = Q \cdot R$ de tal forma que $Q = Q' \Rightarrow Q^{-1} \cdot Q' = I$.
2. $A\overrightarrow{x} = \overrightarrow{b} \Rightarrow QR\overrightarrow{x} = \overrightarrow{b} \Rightarrow Q'QR\overrightarrow{x} = Q'\overrightarrow{b}$
3. Calcular $\overrightarrow{b'} = Q'\overrightarrow{b}$.
4. Resolver por sustitución regresiva $R\overrightarrow{x} = \overrightarrow{b'}$, R es una matriz triangular superior.

Existe una gran cantidad de métodos para realizar la factorización QR de una matriz. Destacar tres de los más importantes en cuanto a su aplicación en la computación:

1. El método de **Gram-Schmidt** se aplica por columnas sobre la matriz y consiste en ir realizando proyecciones de vectores para crear una base ortonormal del espacio generado. Las proyecciones son conocidas como ortogonalizaciones de Gram-Schmidt.

2. El método de **Householder**, se basa en realizar una serie de transformaciones unitarias sobre la matriz original A para conseguir la matriz Q y R. Las trasformaciones se fundamentan en la construcción de una matriz a través de las simetrías de un hiperplano, denominadas reflexiones de Householder.

3. El método de **Givens** es otro método que alcanza la factorización a partir de ciertas transformaciones unitarias sobre una matriz origen, denominas rotaciones de Givens. La rotación de Givens consiste la creación de una matriz de rotación basada en la matriz identidad y fundamentada en funciones trigonométricas para algunas posiciones. A través de la premultiplicación de dicha matriz de rotación se consiguen cambios sobre la matriz entrante hasta conseguir las matrices R y Q paso a paso.

Además, el método de Givens presenta una especial ventaja sobre los demás, su implementación puede ser eficientemente paralelizada y muestra buena predisposición a arquitecturas segmentadas. Otro dato importante es que las funciones trigonométricas requeridas por las rotaciones de Givens son fácilmente resolubles gracias al conocido algoritmo CORDIC (COordinate Rotation DIgital Computing), descrito en el apartado 2.4.

2.3 Descomposición QR mediante rotaciones de Givens

2.3.1 Rotaciones de Givens

En los años 50 Wallace Givens (1910-1993) fue el epónimo de las conocidas rotaciones de Givens, que en términos generales es una rotación de un vector abarcada por dos ejes de coordenadas en el plano.

El uso principal de la rotación de Givens en álgebra lineal numérica es transportar la coordenada Y de un vector n bidimensional sobre el eje de abscisas, es decir, introducir un cero en una coordenada del vector que se desea rotar.

La rotaciones de Givens se representa de forma matricial, en la Fig. 2.6 se muestra el esqueleto para construir una matriz de rotación. De forma indicativa, en la figura citada se rotan las fila i y j, que darán lugar a un nuevo 0 en la matriz origen, exactamente en la fila j.

$$G(i,j) = \begin{array}{c} \\ \\ i \\ \\ j \\ \\ \\ \end{array} \begin{array}{c} \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \\ \begin{array}{ccccccc} & & i & & j & & \end{array} \end{array}$$

Figura 2.6: Patrón para la construcción de la matriz encargada de realizar la rotación de Givens. En ella, los elementos $c = \cos(\theta)$ y $s = \sin(\theta)$.

De forma aclaratoria, la matriz de Givens puede construirse siguiendo las siguiente premisas:

- $g_{kk} = 1$ para $k \neq i, j$
- $g_{ii} = \cos(\theta)$
- $g_{jj} = \cos(\theta)$
- $g_{ji} = \sin(\theta)$
- $g_{ij} = -\sin(\theta)$ para $i > j$

Alternativamente se pueden deducir las siguiente ecuaciones para computar el coseno y el seno siguiendo las ecuaciones Eq. 2.1 y Eq. 2.2 respectivamente.

$$c = cos(\theta) = \frac{a_{i-1,j}}{\sqrt{a_{i-1,j}^2 + a_{i,j}^2}} \qquad (2.1) \qquad s = sin(\theta) = \frac{-a_{i,j}}{\sqrt{a_{i-1,j}^2 + a_{i,j}^2}} = \qquad (2.2)$$

El proceso de rotación se puede resumir en una pre-multiplicación, donde una matriz denominada G multiplica a otra matriz original A. El resultado es una nueva matriz A' que cuenta con un cero insertado de forma premeditada. En la Fig. 2.7 se muestra el proceso de rotación y sus efectos sobre la matriz A.

$$G \cdot A = G \cdot \begin{pmatrix} a_{1,1} & a_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} = \begin{pmatrix} \widetilde{a}_{1,1} & \widetilde{a}_{1,2} \\ 0 & \widetilde{b}_{2,2} \end{pmatrix}$$

$$donde\ G = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}\ y\ \theta = \arctan\begin{pmatrix} b \\ \overline{a} \end{pmatrix}$$

Figura 2.7: Cómputo de una rotación de Givens sobre una matriz A y su posterior efecto en la matriz.

Una vez comprendido cómo se realiza una rotación de Givens, destacar la relevante dependencia operacional que encontramos en la rotación. Como se ha visto, inicialmente se ha de calcular el ángulo de rotación θ, necesario para el posterior cálculo del $cos(\theta)$ y el $sin(\theta)$. Seguido, se construye la matriz de rotación con los valores previos, y seguidamente, se pre-multiplica la matriz.

Del razonamiento anterior se deducen dos operaciones básicas que deben realizarse obligatoriamente de forma consecutiva para llevar a cabo una rotación de Givens:

OB.I Ángulo de rotación: El cálculo del ángulo de rotación θ a partir de un par de elementos seleccionados de dos filas adyacentes, donde $\theta = \arctan(\frac{a_{i,j}}{b_{i-1,j}})$.

OB.II Rotación de vectores: La pre-multiplicación por la matriz creada con el ángulo de rotación (obtenido en OB.I) se encarga de rotar el resto de vectores e inserta un cero en la posición del elemento con el que se ha calculado tal ángulo.

Ejemplo de Rotación de Givens

Para facilitar la compresión, a continuación se muestra el proceso completo de una rotación de Givens sobre una matriz con datos reales. Visualmente ayudará al lector a hacerse una mejor idea de los cálculos necesarios para implementar una rotación.

Dada la siguiente matriz A:

$$A = \begin{pmatrix} 12 & -51 & 4 \\ 6 & 167 & -68 \\ -4 & 24 & -41 \end{pmatrix}$$

El objetivo es generar una rotación de Givens para insertar un cero sobre A en la posición escogida es $A_{31} = -4$. Para ello se tiene que rotar los vectores [(6,-4);(167,24);(-68,-41)] una vez hallado el ángulo de rotación $\theta = \arctan(\frac{-4}{6})$.

Completando la matriz de rotación de Givens se obtiene G_1:

$$G_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta)\cos(\theta) & \end{pmatrix} \approx \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0.83205 & -0.55470 \\ 0 & 0.55470 & 0.83205 \end{pmatrix}$$

El paso final consiste en pre-multiplicar la matriz origina A por la rotación generada en el paso anterior G_1, resultando la siguiente matriz:

$$G_1 \cdot A \approx \begin{pmatrix} 12 & -51 & 4 \\ 7.21110 & 125.63959 & -33.83671 \\ 0 & 112.60414 & -71.83368 \end{pmatrix}$$

2.3.2 Algoritmo Givens QR

La clave del algoritmo de Givens QR son las rotaciones de Givens, gracias a ellas el algoritmo puede insertar nuevos valores nulos en una matriz de forma selectiva. Cada rotación de Givens se interpreta como una transformación unitaria, el conjunto de todas las transformaciones unitarias completan el algoritmo Givens QR.

El algoritmo comienza siempre por la primera columna, aquella que se encuentra más a la izquierda de la matriz. El recorrido siempre sigue un mismo itinerario, se van calculando las rotaciones de Givens a modo de transformaciones unitarias siguiendo la dirección abajo hacia arriba, hasta alcanzar el elemento diagonal de la matriz. Una vez, encontrado el elemento diagonal, se avanza una columna hacia la derecha, volviendo a comenzar desde la fila inferior hasta el siguiente elemento diagonal.

Por lo que el método de Givens QR se define como un proceso secuencial, iterativo y bien determinado. La figura Fig.2.8 muestra cómo avanza el algoritmo en el proceso de factorización, transformando una matriz original del tamaño 3×3.

Observando la figura Fig.2.8, en el paso (a), se ha encuadrado aquellos elementos utilizado para calcular el primer ángulo de rotación θ_1, utilizado a su vez para crear la matriz de rotación de Givens $G_1(2,3)$.

En el paso (b), ya ha sido aplicada la primera transformación (G_1), por lo que la matriz ya cuenta con el primer valor nulo insertado. En este instante también ha de calcularse una nueva

matriz de rotación de Givens utilizando los dos nuevos elementos encapsulados. Por tanto, la matriz de rotación $G_2(1,2)$ del instante (b) utiliza el ángulo de rotación θ_2, es decir, los elementos de primera columna correspondientes a las filas 1 y 2. En este paso ya se ha alcanzado un elemento diagonal de la matriz, entonces el algoritmo ya no ha de realizar más rotaciones en esa columna, por ello, se realiza un desplazamiento hacia la derecha, volviendo a comenzar desde la fila inferior.

En la Fig. 2.8 instante (c), todos los ceros para la primera columna ya han sido insertados. Entonces se inicia el procesamiento de los elementos de la filas 2 y 3, pero de la columna 2, generando el ángulo de rotación θ_3. La nueva matriz de rotación construida con θ_3 es $G_3(2,3)$, último paso del algoritmo para insertar el último valor nulo. Como en este instante, la primera y única rotación utiliza el elemento diagonal de la matriz, el algoritmo finaliza. Aplicando la última rotación se alcanza el instante nombrado como (d), en el que ya se obtiene una matriz triangular superior R.

$$\underset{(a)}{\begin{bmatrix} \times & \times & \times \\ \boxed{\times} & \times & \times \\ \times & \times & \times \end{bmatrix}} \xrightarrow[\theta_1]{G_1(2,3)} \underset{(b)}{\begin{bmatrix} \boxed{\times} & \times & \times \\ \times & \times & \times \\ 0 & \times & \times \end{bmatrix}} \xrightarrow[\theta_2]{G_2(1,2)} \underset{(c)}{\begin{bmatrix} \times & \times & \times \\ 0 & \boxed{\times} & \times \\ 0 & \times & \times \end{bmatrix}} \xrightarrow[\theta_3]{G_3(2,3)} \underset{(d)}{\begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{bmatrix}} = R$$

Figura 2.8: Secuencia de las rotaciones del método QR Givens para factorizar una matriz de dimensiones 3×3.

Por definición del algoritmo de Givens, la matriz R se consigue por la acumulación de trasformaciones unitarias sobre la matriz entrante. Por consiguiente la ecuación Eq. 2.3 describe cómo alcanzar la matriz R. De forma similar, la matriz Q se consigue con las mismas rotaciones de Givens, siguiendo el mismo orden, sólo que en este caso la pre-multiplicación es sobre la matriz identidad I. Esto se encuentra recogido en la Eq 2.4.

$$R = \prod_{k=1}^{t} G_k \cdot A \qquad (2.3) \qquad\qquad Q' = \prod_{k=1}^{t} G_k \cdot I \qquad (2.4)$$

Algorítmicamente la factorización por el método de Givens QR se define como se indica en el algoritmo Alg. 2.1 [6]. Dicho algoritmo se basa en la rotación de Givens descrita en la subsección 2.3.1, coincidiendo con las líneas 3 y 4.

Algorithm 2.1 Algoritmo de Givens QR. Dado una matriz $A \in \mathbb{R}^{m \times n}$ siendo $m \geq n$, el algoritmo va sobrescriendo sobre la matriz entrante A hasta producir el siguiente resultado $Q^t \cdot A = R$.

1: **for** $j = 1$ **to** n **by** $(j+1)$ **do**
2: **for** $i = m$ **to** $j+1$ **by** $(i-1)$ **do**
3: $[c, s] = givens(A(i-1, j), A(i, j))$
4: $A(i-1 : i, j : n) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \times A(i-1 : i, j : n)$
5: **end for**
6: **end for**
7: {NOTA: $givens(Y, X)$ es la función que calcular el ángulo de rotación θ y computa su $sin(\theta)$ y $cos(\theta)$.}

2.4 Algoritmo CORDIC

El algoritmo CORDIC (COordinate Rotation DIgital Computing) es un muestra de eficiencia y eficacia algorítmica en la implementación hardware. El algoritmo fue propuesto por Jack Volder casi en los 60 [7]. Su propósito es la rotación vectorial mediante sólo dos tipos de operaciones, sumas y desplazamientos.

CORDIC utiliza esa rotación vectorial para realizar el cómputo de funciones trigonométricas básicas como son el seno, coseno y tangente, funciones trigonométricas inversas $\tan^{-1}(a/b)$ y la raíz cuadrada $\sqrt{(a^2+b^2)}$. Pero su funcionalidad alcanza más horizontes, con pequeñas modificaciones CORDIC puede también computar funciones hiperbólicas, multiplicaciones, divisiones, cálculos exponenciales, logarítmicos, y raíces cuadradas.

Cierto es que para este algoritmo presenta algunos inconvenientes como su lenta velocidad provocado por su naturaleza iterativa. Para ello, se ha propuesto varias modificaciones y nuevas configuraciones que agilizan su comportamiento, como veremos más adelante.

Sus ventajosas características son la razón por la que ha sido ampliamente utilizado en gran variedad de campos como por ejemplo: DSP (procesamiento digital de señales) [8, 9, 10, 11], procesamiento de vídeo e imagen [12, 13, 14, 15], aplicaciones gráficas [16, 17, 18] y comunicaciones [19, 20, 21], entre otras. Se debe recalcar un creciente interés en las comunicaciones, especialmente este notable repunte se centra en las comunicaciones inalámbricas [22, 23, 24, 25, 26, 27, 28, 29, 30, 31].

2.4.1 Diseño CORDIC estándar

La base del algotitmo CORDIC estándar se basa en la rotación de un vector a través de un plano bidimensional utilizando coordenadas circulares, extensible en otras versiones a coordenadas hiperbólicas o lineales. El algoritmo CORDIC utiliza como entrada un vector inicial (x_I, y_I) y un ángulo θ, generando un nuevo vector resultante (x_R, y_R). Dicho vector resultante, es producido por el desplazamiento del vector inicial sobre el eje de coordenadas circulares exactamente el ángulo θ deseado. A modo de ejemplo, dicha rotación se encuentra ilustrada en la Fig. 2.9.

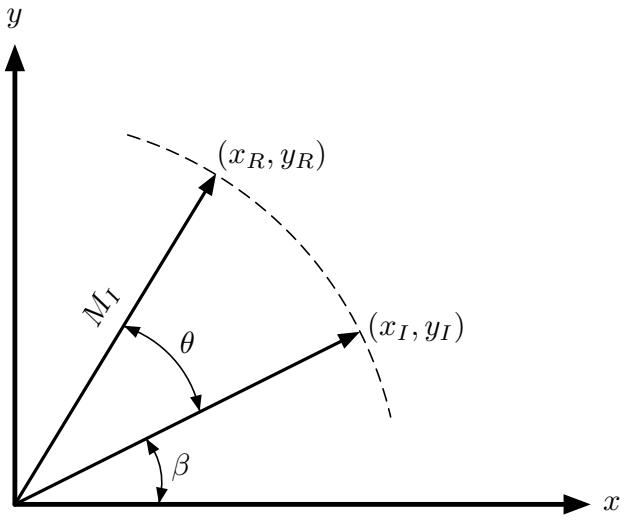


Figura 2.9: Rotación estándar de un vector a través del algoritmo CORDIC.

Matemáticamente hablando, la ecuación Eq. 2.5 expresa de forma analítica la rotación de un

vector. En este caso M_I es el modulo del vector y β es el ángulo inicial.

$$\begin{aligned} x_R &= M_I \cos(\beta + \theta) = x_I \cos(\theta) - y_I \sin(\theta) \\ y_R &= M_I \sin(\beta + \theta) = x_I \sin(\theta) + y_I \cos(\theta) \end{aligned} \tag{2.5}$$

Para una mejor compresión, es habitual encontrarnos con la ecuación anterior (Eq. 2.5) expresada en formato matricial, tal y como refleja la Eq. 2.6. Como se aprecia en la ecuación, para realizar la rotación es necesario calcular el seno y coseno, además de cuatro multiplicaciones y dos sumas. Todo ese conjunto de operaciones son llevadas a cabo si se ejecuta el algoritmo CORDIC.

$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_I \\ y_I \end{bmatrix} = \mathrm{ROT}(\theta) \begin{bmatrix} x_I \\ y_I \end{bmatrix} \tag{2.6}$$

Un detalle a tener en cuenta con respecto al algoritmo CORDIC son los ángulos elementales. Estos se definen sobre una rotación vectorial, que a su vez se compone de una secuencia de micro-rotaciones, las cuales denominamos ángulos elementales α_j. De este modo, el ángulo inicial θ se descompone en una suma de ángulos elementales como se describe en la ecuación Eq. 2.7.

$$\theta = \sum_{j=0}^{\infty} \alpha_j \tag{2.7}$$

Por lo tanto, la rotación dada en la Eq. 2.6 puede calcularse de la siguiente forma:

$$\mathrm{ROT}(\theta) = \prod_{j=0}^{\infty} \mathrm{ROT}(\alpha_j) \tag{2.8}$$

Siendo ROT(α_j) una función definida por la ecuación Eq. 2.9.

$$\begin{aligned} x_R[j+1] &= x_R[j]\cos(\alpha_j) - y_R[j]\sin(\alpha_j) \\ y_R[j+1] &= x_R[j]\sin(\alpha_j) + y_R[j]\cos(\alpha_j) \end{aligned} \tag{2.9}$$

Como puede apreciarse, las micro-rotaciones también requiere del cómputo de varias multiplicaciones, alcanzándose nuevamente una compleja implementación. No obstante, la situación puede mejorar considerablemente, ya que las multiplicaciones pueden evitarse mediante el siguiente procedimiento:

1. Factorizando el término $\cos(\alpha_j)$:

$$\begin{aligned} x_R[j+1] &= \cos(\alpha_j)(x_R[j] - y_R[j]\tan(\alpha_j)) \\ y_R[j+1] &= \cos(\alpha_j)(y_R[j] + x_R[j]\tan(\alpha_j)) \end{aligned} \tag{2.10}$$

2. Escogiendo la secuencia apropiada para los ángulos elementales:

$$\alpha_j = \tan^{-1}(\sigma_j(2^{-j})) = \sigma_j \tan^{-1}(2^{-j}) \tag{2.11}$$

donde $\sigma_j \in \{-1, 1\}$.

Con la definición de ángulos elementales dada por la ecuación Eq. 2.11, cada extensión de rotación se puede calcular de la siguiente manera:

$$\begin{aligned} x[j+1] &= x[j] - \sigma_j 2^{-j} y[j] \\ y[j+1] &= y[j] + \sigma_j 2^{-j} x[j] \end{aligned} \tag{2.12}$$

Como puede verse, la ecuación Eq. 2.12 puede calcularse utilizando simplemente sumas y desplazamientos. Sin embargo debido a la usencia del termino $\cos \alpha_j$ en la ecuación Eq. 2.12, la micro-rotación realizada produce un escalado en el modulo del vector $M[j]$. Llamándole $K[j]$ al factor de escala inferido en cada rotación:

$$M[j+1] = K[j]M[j] = \frac{1}{\cos \alpha_j} M[j] = \sqrt{1 + \sigma_j^2 2^{-2j}} M[j] = \sqrt{1 + 2^{-2j}} M[j] \tag{2.13}$$

En definitiva, el algoritmo CORDIC consiste en aplicar una secuencia estas micro-rotaciones. Teniendo en cuenta que $\sigma_j \in \{-1, 1\}$, entonces el factor de escala permanece constante e independiente del ángulo que va a ser rotado. La ecuación Eq. 2.14 delimita el valor de dicha constante o factor de escala.

$$K = \prod_{j=0}^{\infty} \sqrt{1 + 2^{-2j}} \approx 1.64676025812107 \tag{2.14}$$

Resumiendo, el algoritmo CORDIC estándar basado en micro-rotaciones o iteraciones, produce un resultado escalado definido como un conjunto de tres ecuaciones de coordenadas:

$$\begin{aligned} x[j+1] &= x[j] - \sigma_j 2^{-j} y[j] \\ y[j+1] &= y[j] + \sigma_j 2^{-j} x[j] \\ z[j+1] &= z[j] - \sigma_j \tan^{-1}(2^{-j}) \end{aligned} \tag{2.15}$$

En la Fig. 2.10 se representa desde un punto de vista hardware el diseño de un CORDIC estándar. Como ya se había anunciado este se compone de desplazadores o $shifters$, desplazadores/sumadores o add/sum y una tabla que contiene los ángulos elementales $\tan^{-1}(2^{-j})$. En este caso, como puede apreciarse, no se utiliza ningún multiplicador para corregir el factor de escala.

Además CORDIC puede trabajar en dos modalidades, vectorización y rotación. La decisión de la modalidad dependen de la vía para conseguir el siguiente ángulo σ_{j+1}. Esto va a ser aclarado en la siguiente sección 2.4.2.

2.4.2 Modos de operación CORDIC: Vectorización y Rotación

Como ya se ha dicho anteriormente, el algoritmo CORDIC puede operar en modo vectorización o rotación, también conocidos en inglés como $Vectoring\ mode$ y $Rotation\ mode$ respectivamente. También sabemos que la decisión del modo de operación influye directamente en el desarrollo del algoritmo afectando al cómputo consecutivo de ángulos .

CORDIC modo vectorización

En el modo vectorización es el proceso a través del cual un vector de entrada denominado (x_I, y_I) es rotado por el algoritmo CORDIC hasta que la coordenada y alcanza el valor 0. La

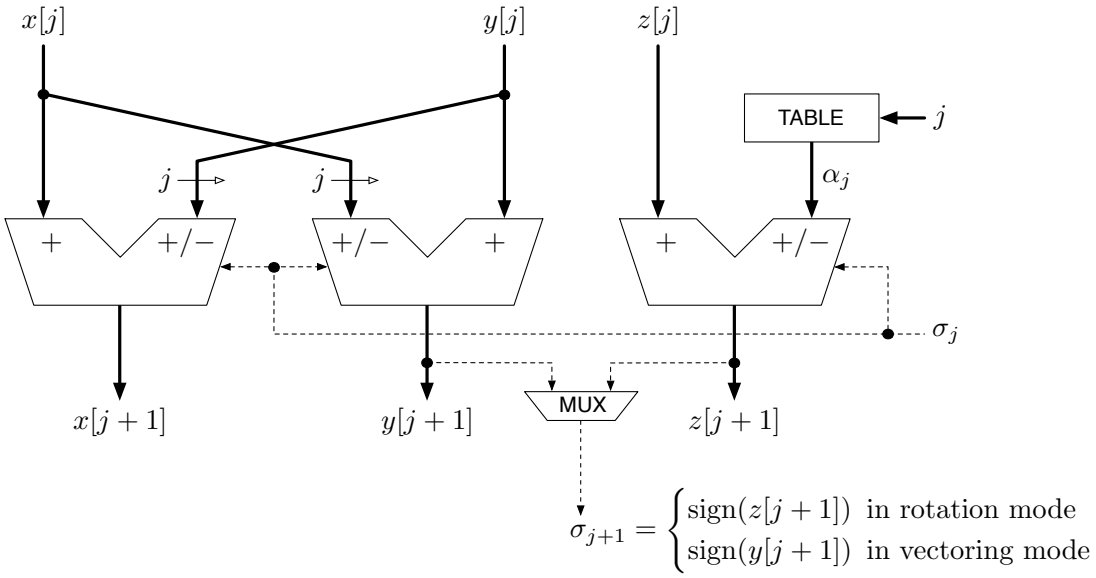


Figura 2.10: Iteración o micro-rotación de un algoritmo CORDIC.

Fig. 2.11 describe el proceso de rotación para el vector inicial dado. Como puede verse, no es algo que suceda en un único cómputo, ya que para alcanzar el vector resultante (x_f, y_f), este ha de pasar por varias iteraciones descritas en la imagen como (x_1, y_1), (x_2, y_2), etc.

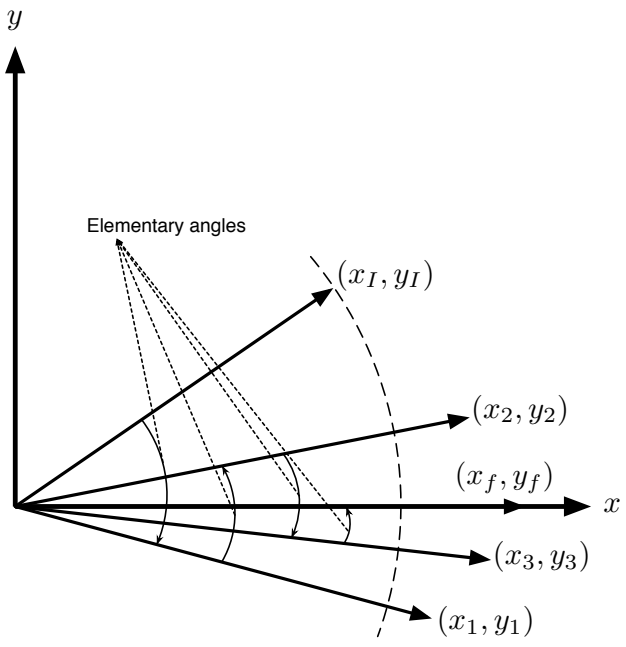


Figura 2.11: Algoritmo CORDIC modo vectorización.

En este modo, la coordenada z almacena el correspondiente ángulo acumulado, es decir aquel obtenido una vez que el vector ha sido rotado. Para un vector que está situado en el primer cuadrante, como el de la Fig. 2.11, el sentido de las rotaciones viene determinado por la siguiente

fórmula:

$$\sigma_j = \begin{cases} 1 & \text{if } y[j] < 0 \\ -1 & \text{if } y[j] \geq 0 \end{cases} \tag{2.16}$$

Evidentemente, la coordenada y es quien rige el flujo del algoritmo, ya que es el objetivo a cumplir por este modo. Para comprender mejor el comportamiento del modo vectorización, la ecuaciones Eq. 2.17 muestran para un vector inicial $(x[0], y[0], z[0]) = (x_I, y_I, z_I)$ las operaciones matemáticas utilizadas para calcular el vector resultante (x_f, y_f, z_f). Donde z_I es el ángulo inicial para reducir a cero, ya que la coordenada y ha de posarse sobre el eje de las abscisas o eje horizontal. Completar la ecuación añadiendo que x_f es un vector escalado por el factor de escala K.

$$\begin{aligned} x_f &= K\sqrt{x_I^2 + y_I^2} \\ y_f &= 0 \\ z_f &= z_I + \tan^{-1}\left(\frac{y_I}{x_I}\right) \end{aligned} \tag{2.17}$$

CORDIC modo rotación

En este modo de operación la entrada del algoritmo es un vector inicial (x_I, y_I) que será rotado por un ángulo de θ. La figura Fig. 2.12 muestra dicha situación, donde el resultado final es el vector rotado (x_f, y_f). Para lograr tal cometido, el ángulo θ es literalmente descompuesto en un conjunto de ángulos elementales α_j, que hacen oscilar al vector original de izquierda a derecha hasta obtener la rotación deseada.

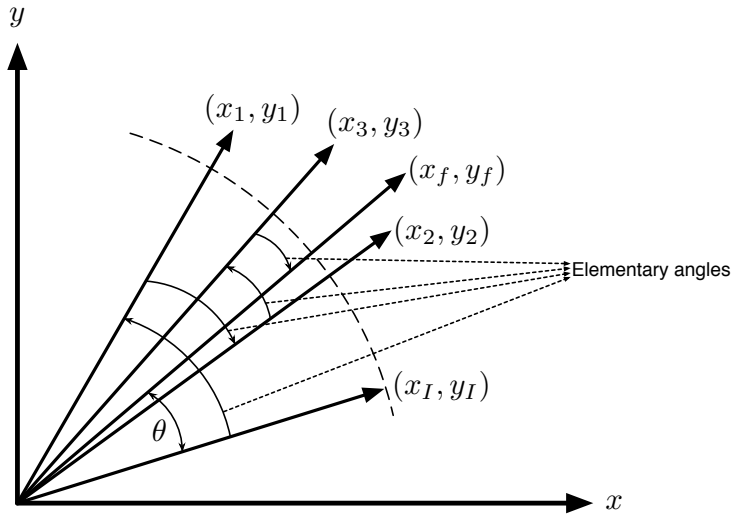


Figura 2.12: Algoritmo CORDIC modo rotación.

La coordenada z toma como valor inicial el ángulo objetivo θ, y su cometido es alcanzar el valor nulo. Entonces, en cada micro-rotación o iteración del algoritmo ha de escoger el sentido

de la rotación de forma adecuada basándose en los elementos de la tabla z, tal y como indica el parámetro σ_j en la ecuación Eq. 2.18.

$$\sigma_j = \begin{cases} 1 & \text{if } z[j] \geq 0 \\ -1 & \text{if } z[j] < 0 \end{cases} \tag{2.18}$$

El resultado final es el descrito por la ecuación Eq. 2.19, donde $(x[0], y[0], z[0]) = (x_I, y_I, \theta)$ son los valores iniciales. Una vez que la coordenada z alcanza el valor 0, el algoritmo finaliza devolviendo los siguientes valores:

$$\begin{aligned} x_f &= K(x_I \cos\theta - y_I \sin\theta) \\ y_f &= K(x_I \sin\theta + y_I \cos\theta) \\ z_f &= 0 \end{aligned} \tag{2.19}$$

Añadir que además de conseguir que el vector haya rotado un determinado ángulo, ha de saberse que este ha aumentado su tamaño debido al factor de escala K. Destacar que el factor de escala es independiente del ángulo.

2.4.3 Arquitectura generales del algoritmo CORDIC

Existen dos tipos de configuraciones arquitecturales para el diseño del algoritmo CORDIC. Por un lado tenemos el formato iterativo, donde un único cómputo es realizado tras múltiples ciclos, eso sí, cuenta con la ventaja de ocupar una área muy reducida. La otra opción es una arquitectura segmentada o pipeline, en ella el volumen de cómputo (throughput) es uno por ciclo a costa de ocupar un mayor espacio.

CORDIC Iterativo (Word-Serial CORDIC)

La arquitectura iterativa para el algoritmo CORDIC se denomina así porque en cada etapa de la rotación se utilizan los mismos componentes, se itera sobre ellos. La arquitectura es muy simple, consiste en un desplazador, sumador algebraico y unos registros para almacenar los valores de salida.

En la figura Fig. 2.13 se ilustra un ejemplo de CORDIC iterativo. Como puede verse, cuando se inicia el registro los valores x_0, y_0 y z_0 ingresan de forma paralela en los registros posteriores gracias a los multiplexores. Según el modo en el que actúe el algoritmo la entrada de los sumadores puede tomar dos opciones en cada etapa, si la entrada es el $sgn(y)$ entonces actuará en modo vectorización, y si la entrada es $sgn(z)$ actuará en modo rotación.

Seguidamente los valores se desplazarán y se sumarán/restarán. Una vez finalizado este primer cómputo, se obtendrán nuevos valores x_1, y_1 y z_1, correspondientes a la primera iteración. Pero una iteración no es suficiente para poder tener un buen resultado, de hecho, se recomienda una operación de suma y desplazamiento por cada bit de precisión aproximadamente. Operaciones descritas a través de la ecuación Eq. 2.15.

Por tanto, para mejorar la precisión, las salidas realimentan de nuevo el circuito para conseguir una nueva iteración, y una mejor aproximación. Una vez se han realizado n iteraciones, el valor final se puede obtener de x_n, y_n y z_n y deja de realimentar el circuito CORDIC.

La ventaja de esta arquitectura se debe al eficiente uso de hardware, ya que los recursos son reutilizados en cada iteración. Eso sí, con este diseño se obtiene un valor final tras cada n ciclos.

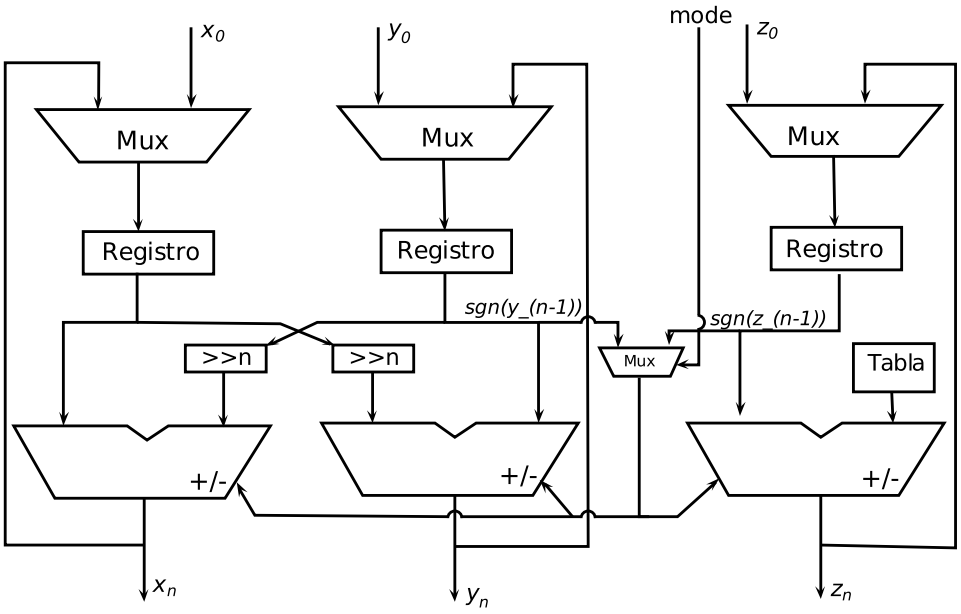


Figura 2.13: Arquitectura iterativa o word-serial del algoritmo CORDIC, en ella se aprecia la realimentación de las señales x,y y z en el circuito.

CORDIC segmentado (Parallel CORDIC)

La arquitectura segmentada o paralela consiste un conjunto de varias etapas del algoritmo CORDIC encadenadas una tras otra, de tal modo que se despliegan linealmente para enlazar las salidas de una etapa anterior como entradas de la siguiente. En ella, se utilizan en cada etapa los mismos recursos que en el diseño iterativo, donde la salida de la etapa n consiste en la entrada de la etapa $n + 1$. Es decir, cada etapa contiene los mismos componentes: dos desplazadores, tres sumadores/restadores y sus correspondientes multiplexores y registros.

En la figura Fig. 2.14 se muestra un diseño hardware de la arquitectura segmentada con n etapas. Este tipo de diseño presenta dos grandes ventajas. Primero, la arquitectura es completamente combinacional y su control depende las propias señales. Segundo, su latencia es de n ciclos y se genera una salida por ciclo, siempre y cuando el caudal del circuito esté completo.

Aunque ha de citarse que este diseño conlleva un gran inconveniente, ya que la cantidad de recursos hardware se dispara. La recursos utilizados son directamente proporcionales a la cantidad de etapas del diseño. Por lo tanto, aproximadamente consume n veces los recursos de la arquitectura iterativa para conseguir el mismo resultado.

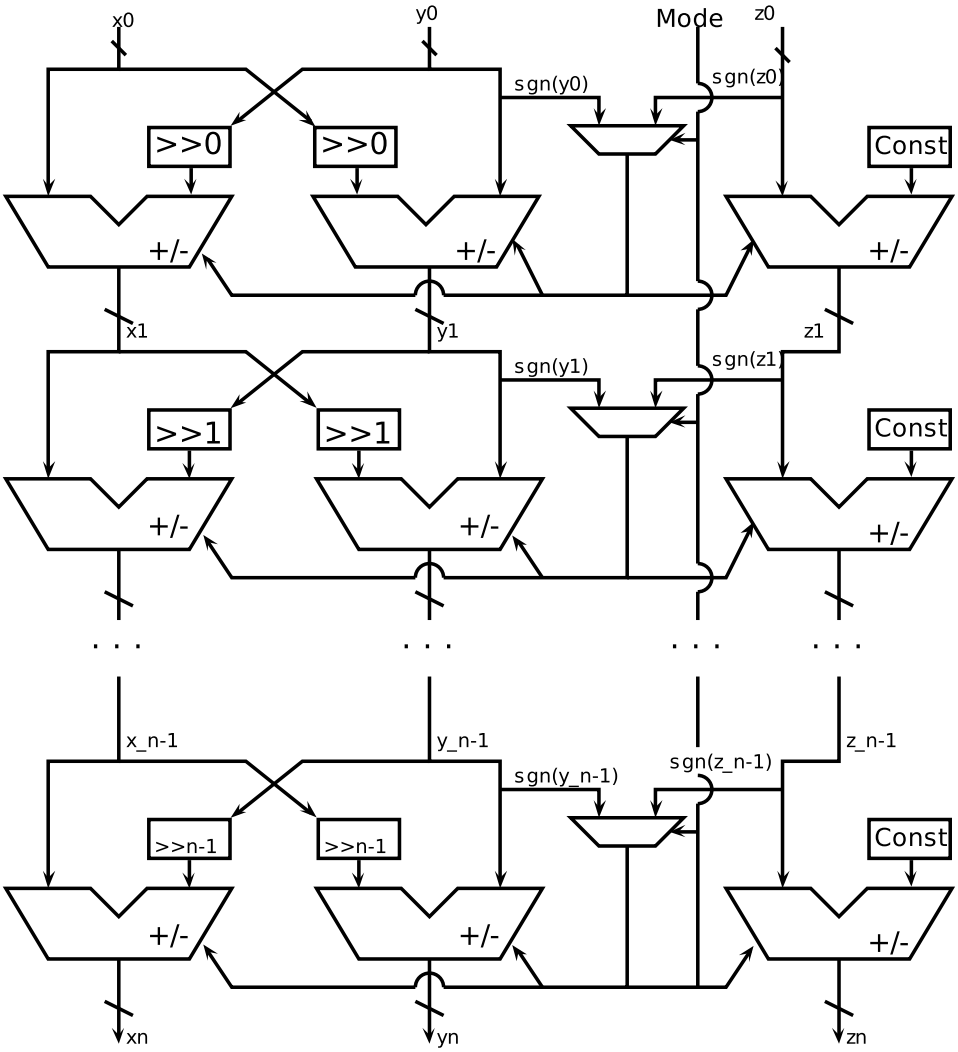


Figura 2.14: Arquitectura segmentado o pipeline del algoritmo CORDIC.

2.5 Problemas de planificación (Scheduling)

En los problemas de planificación o scheduling, este segundo término es utilizado de forma más general, la capacidad de las máquinas se encuentra definida por un tiempo determinado y el problema consiste en cubrir la demanda de máquinas necesarias para realizar las operaciones en el tiempo, sin exceder la capacidad disponible. De tal definición surgen cuatro variantes bien distintas, diferenciadas principalmente por el flujo operacional.

A continuación se citan las cuatro variantes clasificadas por el flujo operacional:

- Con un flujo aleatorio encontramos los problemas conocidos como Open Shop Scheduling (OSS), donde no existe ningún tipo de restricción en cuanto al orden de máquinas utilizadas por los trabajos realizados.
- Con un flujo general se define Job Shop Scheduling Problem (JSSP), en el que cada trabajo es procesado por un conjunto definido de máquinas siguiendo un orden predefinido.
- Siguiendo un flujo regular se definen los Flow Shop Scheduling (FSS), en éstos los trabajos

utilizan las máquinas en el mismo orden, pero una pieza no visita una misma máquina más de una vez.

- El flujo de permutación es el que define los Permutational Flow Shop (PFSS), es un caso particular del flujo regular con la particularidad de que la secuencia de trabajos es siempre la misma en todas las máquinas.

2.6 Descripción de la problemática Job Shop Scheduling

Los JSSP pertenecen al tipo de problemas optimización donde el objetivo es realizar una asignación de recursos reduciendo los tiempos de producción. Un problema JSS $n \times m$ se expresa de la siguiente forma: existe un conjunto de m trabajos (Jobs) diferentes $\{J_i|1 \leq i \leq m\}$ que han de ser procesados según una planificación por un conjunto de n maquinas (Machines) distintas $\{M_j|1 \leq j \leq n\}$ [32] [33]. Generando un total de posibles combinaciones indicado por la siguiente expresión $(M!)^N$, donde M es el número total de trabajos y N la cantidad de máquinas.

Además, los JSSP están condicionados por las siguientes restricciones:

- Un trabajo no visita la misma máquina dos veces, es decir, cada trabajo sólo es procesado por máquinas distintas.
- Las rutas tecnológicas o recorridos entre máquinas está predefinido y se considera invariable.
- No existen restricciones de precedencia entre operaciones de trabajos diferentes.
- Una máquina sólo es capaz de procesar un trabajo a la vez, y ésta no puede verse interrumpida por otro trabajo.
- Los tiempos de proceso son independientes de la secuencia seguida. Y además, los tiempos de configuración y ajuste de la maquinaria están excluidos y no serán considerados, al igual que los tiempos de transporte.

Entonces, cada trabajo es procesado por una secuencia previamente definida de máquinas, también llamada tarea, en la que el trabajo de procesamiento J_i en la máquina M_j se llama operación $O_{i,j}$. Esta operación requiere el uso exclusivo de la máquina M_j durante un tiempo de procesamiento fijo $p_{i,j}$.

La planificación o scheduling, consiste en ir seleccionando todas las operaciones $O_{i,j}$, una vez todas las operaciones han sido elegidas, el orden de éstas configura una posible solución del problema donde $S_{m \times n} = \{O_{i,j}|1 \leq i \leq m\, 1 \leq j \leq n\}$. Evidentemente una operación $O_{i+1,j}$ no podrá continuar hasta que la operación $O_{i,j}$ haya sido completada, tal y como indica la restricción anterior de uso exclusivo e ininterrumpido de las máquinas.

Cada solución S tiene un tiempo de ejecución total para completar todos los trabajos, este tiempo es calculable y se conoce como **makespan**, o alternativamente **tiempo total de producción**. Los diagramas de Gantt son una forma sencilla de cuantificar y visualizar la duración máxima para una solución dada, es decir, una forma sencilla de calcular el makespan.

En definitiva, los JSSP son un problema de minimización de tiempos de procesamiento, en el que el objetivo es conseguir ordenar las tareas a fin de reducir el tiempo de procesamiento.

Una descripción más cerrada y precisa del objetivo de los JSSP es posible gracias a las matemáticas, dada a continuación. Teniendo en cuenta que $t_{i,j}$ es el momento inicial de la operación $O_{i,j}$

obtenida para una planificación S, es posible la resolución del problema a través de las minimización del makespan como muestran la Ec. 2.20, sin olvidar las restricciones descritas anteriormente a través de las ecuaciones Ec. 2.21 y Ec. 2.22.

Objetivo:

$$\min\ makespan(S) \tag{2.20}$$

Restricciones:

$$t_{i+1,j+1} \geq t_{i,j+1} + p_{i,j+1} \quad i = 1 \ldots m; j = 1, \ldots n \tag{2.21}$$

$$t_{i+1,j+1} \geq t_{i+1,j} + p_{i+1,j} \quad i = 1 \ldots m; j = 1, \ldots n \tag{2.22}$$

Como consecuencia de todo esto, los problemas JSSP tienden a ser descritos por medio de dos tablas, una tabla de secuencia de operaciones y una tabla de tiempos de procesamiento. El primero representa el orden o secuencia que debe seguir un trabajo ($O_{i,j}$). Y la segunda tabla lista todos los tiempos de operación para cada trabajo de operación, es decir, el tiempo de procesamiento ($p_{i,j}$).

2.6.1 Batería de problemas

Un hecho interesante es la existencia de varias baterías de problemas o benchmarks testeados para los problemas JSSP. Por un lado, Muth y Thompson formularon una prueba de uso común para medir la efectividad de un determinado método. Algunos de ellos como mt10 y mt20 [34] fueron un buen desafío computacional durante mucho tiempo debido a su dificultad.

Applegate y Cook propusieron un grupo problemas con los que habían hecho múltiples pruebas denominados “Ten tough problems” (diez problemas difíciles). Éstos suponen un mayor desafío computacional en comparación con los problemas mt10, debido en gran parte a su mayor orden de magnitud. La nomenclatura dada a los problema de Applegate y Cook fue La01-la40 [34].

2.6.2 Ejemplo de JSSP 3×3

Para tener una mejor visión del tipo de problemas JSSP se va a presentar un ejemplo muy básico [35]. El ejemplo dado trata de un problema JSSP 3×3 en el que se han de realizar un total de 3 jobs o trabajos en 3 máquinas. Por tanto se han de planificar las 9 operaciones sin posibilidad de interrumpir máquinas y sin volver a visitar las máquinas por donde ya han pasado.

Los trabajos están representados de los nombre nombres $j1$, $j2$ y $j3$. La secuencia de cada trabajo está reflejada en la Tab. 2.6.2, en este caso el recorrido es diferente para cada uno de ellos pero no tiene porque ser así. Observando el trabajo $j1$, la secuencia de máquinas que debe de recorrer es $m1$, $m2$ y $m3$. La duración que ha de permanecer en cada una de las máquinas se obtiene de la Tab. 2.6.2, por lo que el primer trabajo de $j1$ deberá de permanecer 3 unidades de tiempo en la máquina 1, 2 unidades de tiempo en la máquina 2 y finalmente 2 unidades de tiempo en la máquina 3.

A modo de ejemplo queremos proponer una posible solución para el problema anteriormente definido. Un posible orden basado en las tareas podría ser el siguiente: $t_{1,1}$, $t_{3,1}$, $t_{2,1}$, $t_{1,2}$, $t_{3,2}$, $t_{2,2}$, $t_{2,3}$, $t_{1,3}$, y $t_{3,3}$, es decir, primero se empezaría por la primera tarea del trabajo 1, se continuaría con la primera tarea del trabajos 3 y así sucesivamente, siguiendo el orden indicado en la solución.

En la Fig. 2.15 se muestra a través de un diagrama de Gantt como quedaría el orden de las tareas y como las procesarían cada una de las máquinas. Se ha utilizado un código de colores basado el

Secuencia de Máquinas

	Operaciones		
Job	1	2	3
$j1$	$m1$	$m2$	$m3$
$j2$	$m1$	$m3$	$m2$
$j3$	$m2$	$m1$	$m3$

Tabla 2.1: Secuencia de máquinas recorridas para realizar los trabajos $j1$, $j2$ y $j3$, en cada una de sus operaciones.

Tiempo de Procesamiento

	Operaciones		
Job	1	2	3
$j1$	3	2	2
$j2$	1	5	3
$j3$	3	2	3

Tabla 2.2: Tiempo de procesamiento en cada operación para los trabajos $j1$, $j2$ y $j3$.

espectro de color, siendo las tareas azules aquellas procesadas antes que las tareas coloreadas en tonalidades rojas.

Existen varios detalles que anotar en Fig. 2.15, en el instante inicial tanto la máquina 1 como la máquina 2 están activas con las tareas t_{11} y t_{31} respectivamente, mientras que la máquina 3 se encuentra ociosa. En la fase final ocurre lo contrario, solo la maquina 3 está ocupada con la tarea t_{33}. Destacar en el gráfico el instante de tiempo 6 donde se muestran un hueco ocioso en la máquina 2, este retardo es debido a la dependencia operacional, donde la tarea t_{23} no pude dar comienzo hasta que la t_{22} haya sido procesada. El total de unidades de tiempo para procesar la solución aportada es de 14. Debido a que el JSSP 3×3 ha sido ampliamente estudiado, se ha demostrado que 14 unidades de tiempo no es la solución óptima.

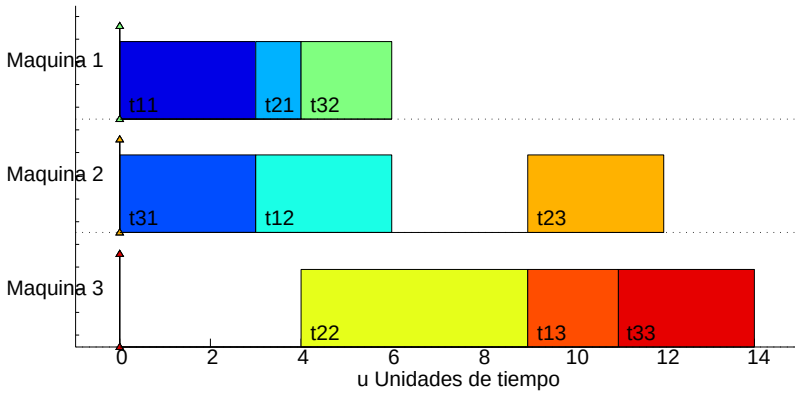


Figura 2.15: Solución basada en un diagrama de Gantt para ejemplo para el JSSP 3x3, donde tij representa la tarea realizada para el trabajo i en la operación j.

Una de las soluciones óptimas al problema presentado viene dada por la secuencia de operaciones: $t_{2,1}$, $t_{3,1}$, $t_{1,1}$, $t_{2,2}$, $t_{3,2}$, $t_{1,2}$, $t_{3,3}$,$t_{2,3}$, y $t_{1,3}$. En la Fig. 2.16 se presenta dicha solución donde se describe la secuencia de operaciones y su orden de ejecución, completándose todas ellas en un total de 11 unidades de tiempo, es decir, 3 unidades menos que la solución anterior.

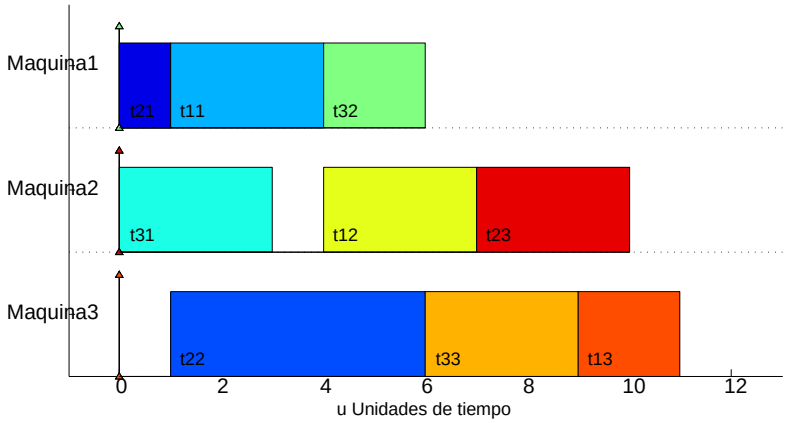


Figura 2.16: Solución basada en un diagrama de Gantt para ejemplo para el JSSP 3x3, donde tij representa la tarea realizada para el trabajo i en la operación j.

3 Implementación iterativa para la descomposición QR sobre FPGA

3.1 Antecedentes y trabajos previos

En la sección 2.3 (Descomposición QR mediante rotaciones de Givens) se ha definido en qué consiste la descomposición QR y la importancia de este algoritmo en el álgebra lineal para la resolución de sistemas lineales, cálculo de autovalores, obtención de una matriz inversa, etc. Ahora queremos centrarnos en conocer su utilización y en cómo ha sido implementada y diseñada esta función en distintos ámbitos por otros.

La utilización de la función QR está presente en múltiples ambientes que a continuación iremos indicando. Por ejemplo, en el ámbito de la industria se utiliza para algoritmos de control en los que se han de resolver una cierta cantidad de sistemas lineales [36]. Otro ejemplo que encontramos en la literatura es su uso para la compresión de imagen [37] y para marcas de agua en imágenes [38][39]. Aunque el procesamiento de la señal digital (DSP) en las comunicaciones inalámbricas es con diferencia donde encontramos un mayor número de aplicaciones, destacar los siguientes: algoritmos de beamforming o filtrado adaptativo espacial para la transmisión y recepción de señales direccionales, el algoritmo MIMO, algoritmos de detección y cancelación multiusuarios, etc. todos ello bien descritos en el artículo [40]. Adicionalmente, citar varios artículos en los que se aplica la descomposición QR de forma exitosa para la tecnología MIMO [41] y el filtrado adaptativo [42].

El uso de FPGAs para acelerar la computación de algoritmos es un hecho cada vez más presente, aunque su complejidad implica el uso de herramientas de síntesis automática que generan arquitecturas que no tienen porque ser óptimas [43]. Por ello, pensamos que es beneficioso conseguir optimizar la descomposición QR sobre una FPGA para su posterior aplicación en los ámbitos presentados.

La mayoría de las implementaciones de descomposición QR en FPGA que hemos encontrado en la literatura son usualmente sistólicas y resuelven tamaños de matrices bastante limitado. La

matriz sistólica es una estructura de cuadrícula paralela donde las celdas están vinculadas con otras celdas vecinas. Cada celda es un elemento de procesamiento implementado en la mayoria de los casos a través de algoritmo CORDIC, entre otros ejemplos implementados sobre una FPGA pueden encontrarse los siguientes artículos: [44], [45], [46], [47] o [48]. No entraremos más en profundidad sobre este tema ya que va ser tratado con mayor detenimiento en el capítulo 4.

Es cierto que todos ellos muestran resultados positivos con respecto a la factorización de pequeñas matrices, sin embargo, el problema es resolver matrices de una longitud mediana-grande. El tamaño máximo de matrices factorizado por las estructuras sistólicas no excede 16×16 filas, siendo la gran cantidad de recursos utilizados en FPGA su falta más notable. Aunque dicho exceso es compensado con un alto rendimiento computacional y eficiencia en la productividad.

Como alternativa al diseño sistólico, las arquitecturas iterativas son menos voluminosas y utilizan menos recursos del sistema. Debido a su configuración es complicado conseguir el elevado rendimiento de los diseños sistólicos, pero por ahora es la única forma de alcanzar tamaños de matriz que sobrepasen un tamaño de 16×16 fila. En la literatura hay algunos ejemplos de descomposición QR a través de arquitectura iterativas tales como [49], [50], [51] o [52], pero todavía tienen ciertos aspectos que pulir.

Los trabajos [49] y [50] realizan una descomposición QR basada en números complejos para aplicaciones específicas basadas en circuitos integrados (ASIC) donde se destaca la baja utilización del área. El aspecto más negativo es que dicha arquitectura presenta una limitación en cuanto al tamaño de las matrices que se computan, siendo este de 4×4.

En nuestro caso, nosotros pensamos que sería positivo no limitar la arquitectura a una longitud de matriz única o demasiado pequeña, queremos ofrecer una mayor versatilidad con respecto al tamaño de matriz. Por otro lado, es importante lograr una superposición entre operaciones que agilizará el procesamiento. Anunciar que el cuello de botella del algoritmo se encuentra en la rotación de las filas para aquellas matrices que presenta un tamaño elevado, puesto que tal rotación implica de una gran cantidad de operaciones para cada uno de los elementos de cada fila.

En nuestro diseño previo descrito en el artículo [52] y utilizado como hilo conductor, se presenta un diseño con una buena reutilización de datos a través de FIFOs, habilitado para una longitud de matriz variable. Su principal inconveniente es su falta de computación paralela, mejorada en nuestra nueva propuesta (véase sección 3.4). Hemos utilizado más recursos para aumentar las rotaciones y reducir los tiempos de inactividad. Las matrices medias requieren un mayor cálculo de potencia debido a la considerable diferencia en el tiempo de procesamiento entre Q y R. Por las razones anteriores, proponemos una arquitectura genérica iterativa más ligera para matrices de tamaño medio-grande donde la cantidad de elementos rotadores es configurable. Además, la arquitectura propuesta es capaz de calcular múltiples longitudes de matriz sin la necesidad de volver a sintetizar el diseño gracias al parámetro definido para limitar el tamaño máximo de fila, todo ello se ha descrito en las secciones 3.3 y 3.4.

3.2 Diseño iterativo basado en FPGA para la Descomposición QR

En esta sección, el objetivo es implementar la factorización matricial QR siguiendo el método de Givens desde una perspectiva hardware. Por ello, es importante considerar de inicio que cualquier dispositivo hardware cuenta unos recursos limitados, por lo que la optimización de ellos y el buen uso favorecerán la computación del algoritmo en conjunto con el solapamiento de operaciones.

El dispositivo de partida con el trabajaremos es una FPGA Virtex VI que se adapta perfectamente a nuestras necesidades. Principalmente, su fácil y rápido prototipado hace que sea un medio ideal para implementar sistemas finales, aunque no ha de menospreciar su potencia de cómputo como acelerador hardware. Recalcar que cada FPGA cuenta con un número limitado de elementos y de interconexiones programables para implementar los circuitos.

De acuerdo con el método de Givens descrito en la sección 2.3.2, éste se compone de dos operaciones básicas que se llevan a cabo de forma ordenada, el cálculo del ángulo de rotación (OB. 1) y la multiplicación por una matriz con los elementos formados por el ángulo calculado (OB. 2).

Aunque parecen dos simples operaciones, desde un punto de vista hardware no tienen una fácil implementación. La primera de ellas conlleva el cálculo de un seno y coseno, además de la arcotangente para el ángulo; todas ellas consideradas operaciones de elevada complejidad. En cuanto a la segunda operación básica destinada a la rotación, ésta consiste en premultiplicar la matriz objetivo por una de iguales dimensiones construida en base a los valores anteriores, seno y coseno, a fin de poder insertar un cero en la matriz destino.

Alternativamente, existe la posibilidad de evitar el cómputo de ángulos, senos y cosenos a partir de las ecuaciones Ec. 2.1 y Ec. 2.2, descritas en el sección 2.3.1. Realmente, este método alternativo no resulta mucho mejor que el anterior, ya que se compone de otras tres operaciones distintas que nuevamente enmarañan el camino y no lo compensan. La división, la potencia, y raíz cuadrada presentan complejidades de índole asintótica, y deben ejecutarse al mismo tiempo.

Afortunadamente CORDIC, descrito en la sección 2.4.1, es un algoritmo ampliamente conocido capaz de realizar y hacer converger ese mismo cómputo de una forma sencilla. Basado en sumas y restas, realiza las rotaciones de los vectores un determinado ángulo o simplemente llevar una de sus componentes a cero. Por lo tanto, CORDIC se convierte en el gran aliado, el componente ideal para realizar la rotación de Givens que necesitamos, evitando de este modo el tedioso cálculo de raíces cuadradas, divisiones, arcotangentes, senos o cosenos.

De este modo, aplicar la función de rotación de Givens toma una nueva perspectiva a nivel Hardware, viéndose ampliamente potenciada y simplificada por el algoritmo CORDIC. Las operaciones básicas que computan el ángulo de rotación (OB 1) y la operación básica para rotar filas (OB 2) descritas en 2.3.1 se realizan por mediación de componentes CORDIC diferenciados por su modo de operación.

La OB 1 es factible su implementación por mediación de un CORDIC vectorización, que devuelve como dato un vector rotado y el ángulo de rotación. Mientras que la OB 2 encargada de la rotación se llevará a cabo a través de un CORDIC rotación. Destacar nuevamente la imposibilidad de rotar nuevos elementos hasta que el CORDIC vectorización ha finalizado el cómputo del ángulo.

3.2.1 Hilo conductor precedente para la factorización QR y resultados

Tal y como se ha citado en la introducción de este capítulo, el hilo conductor que inició el desarrollo de capítulo surge del artículo [52], presentado en el congreso DCIS (Design of Circuits and integrated Systems) de 2012 en Avignon. El planteamiento que se muestra se centra un diseño para la factorización QR en el que se destaca la gran reutilización de los datos a través de estructuras FIFOs y la rotación de vectores basado en el algoritmo CORDIC. A pesar de ser una propuesta inicial el enfoque propuesto es capaz de descomponer a través de QR un tamaño de matrices variable sobre una arquitectura FPGA.

Este primitivo diseño, presenta un diagrama de computación que encaja completamente con la definición descrita en el apartado 2.3.2, basado la operación cálculo de ángulo de rotación y rotación de Givens de dos filas adyacentes completas. La minimización de tiempos se consigue fácilmente a través de un básico solapamiento operacional que se puede conseguir en el algoritmo de factorización QR cuando se ejecutan en paralelo la operación de cálculo de ángulo y rotación de Givens para el resto de vectores. Para conseguirlo, mientras se está computando un nuevo ángulo de rotación a la vez se está produciendo la rotación de elementos de forma segmentada. Esta superposición operacional sólo es posible en consecutivas etapas del algoritmo, ya que es la única forma de aprovechar ese instante de inactividad que presenta el algoritmo cuando primero se calcula el ángulo de rotación y posteriormente se utiliza ese dato para la rotación de Givens.

En la figura 3.1 se muestra el diagrama de la traza de ejecución para la factorización del algoritmo QR propuesto como hilo conductor y desarrollado en los próximos apartados. En el diagrama se aprecian dos bloques diferenciados por el tipo de operación necesaria para llevar a cabo una rotación de Givens. El algoritmo comienza con el cómputo del ángulo de rotación θ_1, en el que intervienen los dos primeros elementos de las dos filas adyacentes inferiores de la matriz entrante. Una vez calculado el primer ángulo de rotación, el algoritmo progresa rotando los elementos de las filas a las que hace referencia el ángulo θ_1 junto el cálculo un nuevo ángulo de rotaciónθ_2. Recalcar que la operación de rotación se hace de forma segmentada o pipeline, es decir en cada ciclo de reloj un nuevo dato es generado, para indicarlo se han insertado unas rayas verticales en el nodo rotación. Todo ello se realiza un número k de veces hasta conseguir completar la factorización QR de la matriz entrante.

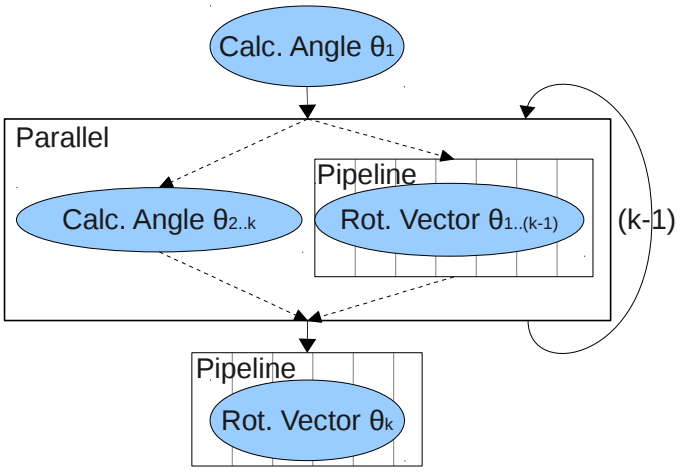


Figura 3.1: Diagrama primitivo utilizado como hilo conductor para la factorización QR.

La arquitectura propuesta que apoya el diagrama anterior se representa en la Fig. 3.2. Esta arquitectura ya se ve diferenciada en dos secciones independientes, coincidiendo en las operaciones básicas para generar una rotación de Givens:*unidad de cálculo de ángulo* y *unidad de rotación*. Ambas secciones leen y escriben los datos de la matriz que se almacena en una RAM de doble puerto y trabajan en paralelo como indicamos anteriormente. La primera sólo opera sobre el primer elemento no nulo de las filas, pero la segunda opera sobre el resto de los elementos. Para equilibrar el tiempo de cálculo de ambas unidades, la *unidad de cálculo de ángulos* es iterativa mientras que la *unidad de rotación* es una implementación en forma de pipeline.

De forma general, la unidad de cálculo de ángulos se compone en un componente de rotación de elemento CORDIC en modo vectorización y un registro para el elemento rotado de la fila su-

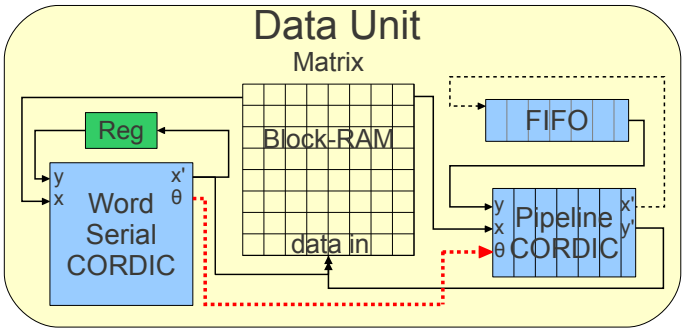


Figura 3.2: El esquema de la unidad de datos propuesto como hilo conductor.

perior. Este elemento será el elemento a poner a cero en la siguiente rotación de Givens. Esta sección ocupa un área pequeña ya que las operaciones se implementan en serie. La sección de rotación consiste en un CORDIC segmentado en modo de rotación y una FIFO para almacenar temporalmente la fila superior rotada que a su vez será reutilizada en la siguiente rotación.

La razón principal para usar una FIFO es porque la matriz se almacena en una RAM de doble puerto, estructura eficientemente implementada en la FPGA usando bloques RAM especializados. Sin embargo, la sección de rotación requiere leer dos elementos produciendo a su vez dos resultados en cada ciclo. Teniendo en cuenta que la fila superior rotada será nueva entrada de la siguiente rotación dada, una FIFO es la estructura ideal para almacenar temporalmente estos datos de una rotación a la siguiente. De tal manera que sólo los elementos de la fila superior se leen directamente desde la matriz y sólo los elementos rotados de la fila inferior se almacenan de nuevo en la RAM. Sin embargo, en la última iteración para cada columna, la fila almacenada en la FIFO contiene el valor final factorizado a través de QR Givens de la matriz.

Es importante señalar que esta arquitectura propuesta es independiente del tamaño de la matriz, realmente quién se encarga de controlar la rotación de filas es la unidad de control, por lo que bastará con indicar en cada factorización las dimensiones de la matriz para que la unidad de control se adapte a dicho tamaño. Sin embargo, hay que tener en cuenta que el tamaño de la RAM utilizada para almacenar la matriz, sí depende fuertemente del tamaño de la matriz.

Se ha diseñado un núcleo VHDL parametrizable para la descomposición QR basado en la arquitectura propuesta como hilo conductor. El diseño inicial propuesto permite configurar tanto el número de bits de precisión y el tamaño de la matriz. Este núcleo ha sido simulado, evaluado y sintetizado usando Xilinx ISE 13.4 dirigido a un FPGA Virtex-6. Los dos algoritmos CORDIC y la estructura FIFO se han implementado utilizando la Herramienta Xilinx CORE Generator. Para confirmar la idoneidad del núcleo propuesto para matrices de tamaño medio, se han realizado una variedad de implementaciones para una amplia gama de tamaños de matriz y longitud de tres bits (16, 24, 32). A continuación, se muestran los resultados obtenidos.

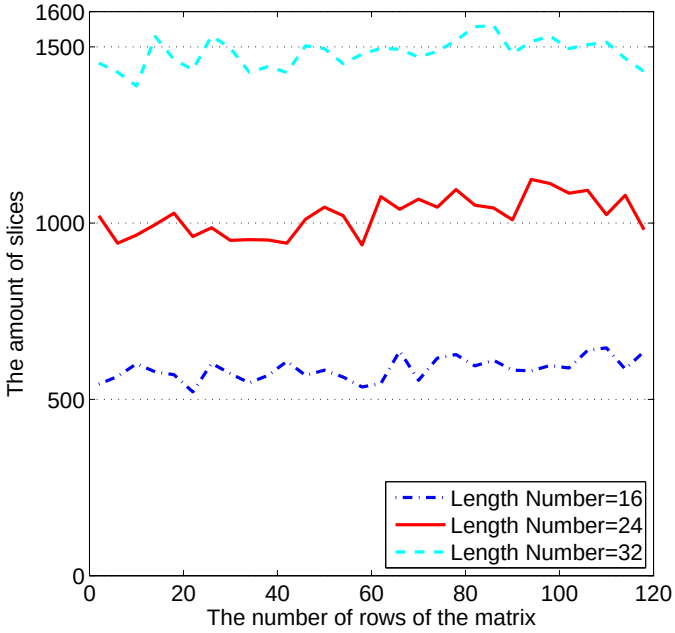


Figura 3.3: Cantidad de Slice necesarios para implementar la arquitectura propuesta, sin tener en cuenta los slices necesarios para la implementación de los algoritmos CORDIC.

La Fig. 3.3 muestra la cantidad de slices necesarios para implementar el hardware limitado para un tamaño de matriz que no exceda las 120 filas sin tener en cuenta el tamaño de los algoritmo CORDIC. Para el diseño de los elementos CORDIC se ha utilizado una herramienta de generación de componentes de forma automática de la herramienta ISE. En el gráfico se muestra como el número de slices no depende del tamaño de la matriz, sino de los bits de precisión utilizados para representar los elementos de la matriz. Todas las líneas tienen fluctuaciones menores debido a los elementos de control de la unidad y el enrutamiento, pero sus valores son bastante similares independientemente del tamaño de la matriz. Por lo tanto, como se esperaba, la arquitectura propuesta podría utilizarse para cualquier tamaño de la matriz, si esta matriz encaja en el bloque interno RAM del dispositivo seleccionado.

Para ver la influencia del tamaño de la matriz en la Fig. 3.4 se muestra la cantidad de BlockRAM18 requeridos dependiendo del tamaño de la matriz. Como puede verse, cuanto mayor sea el número de filas, mayor será el número de BlockRAM18 que se necesite. Es importante notar que el tamaño más grande probado, matrices de tamaño de palabra de 32 bits y 120 elementos por fila, la cantidad de bloques RAM utilizados es aproximadamente el 10 % de la cantidad total disponible en el dispositivo FPGA Virtex-6 más pequeño, por lo que aún hay margen para seguir creciendo.

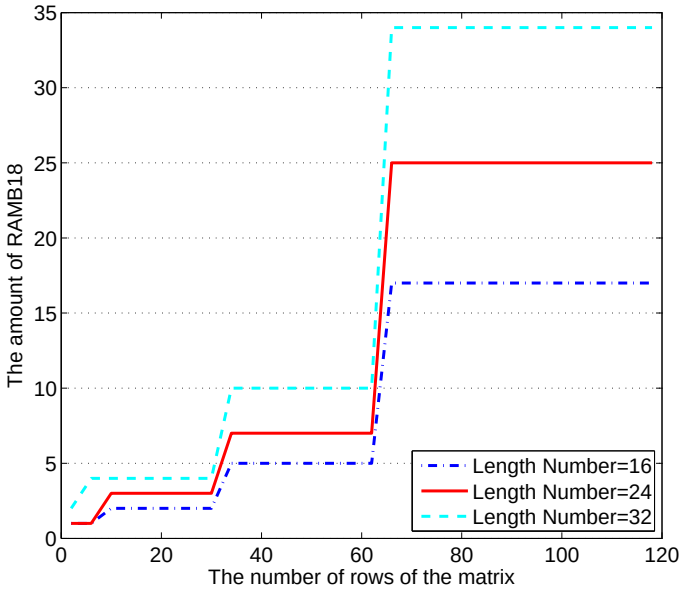


Figura 3.4: El número de bloques RAMB18 utilizados en respecto las dimensiones de la matriz de entrada.

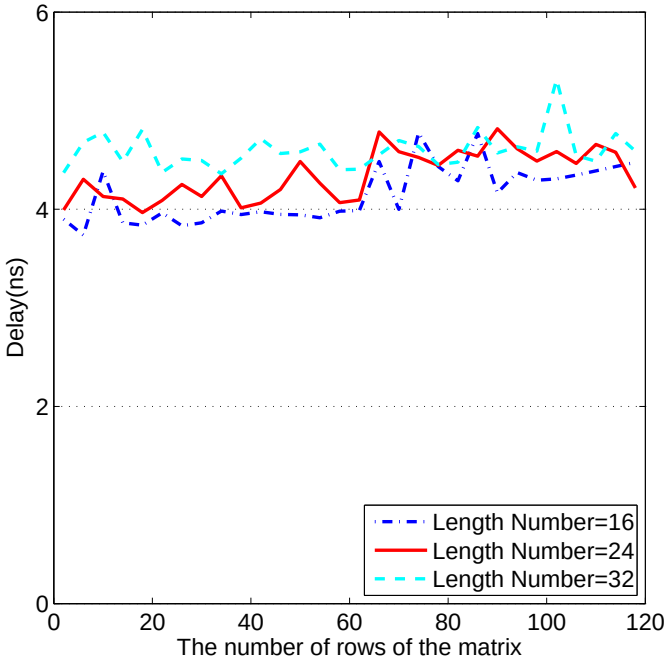


Figura 3.5: Retardo en nanosegundos del diseño propuesto en relación con el tamaño de la matriz entrante.

Con respecto a la velocidad, la Fig. 3.5 representa la frecuencia máxima de operación que nuestro diseño puede alcanzar, dependiente del tamaño de matriz. La mayoría de los valores están por encima de 200 Mhz. Este límite lo establece la arquitectura CORDIC y su frecuencia máxima de funcionamiento. En este caso, la representación numérica no afecta significativamente a la frecuencia.

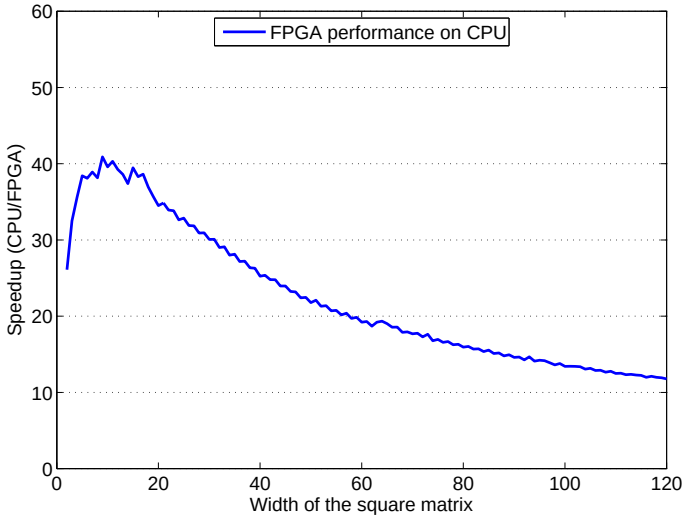


Figura 3.6: Aceleración del FPGA con respecto a la CPU del diseño propuesto en relación con cantidad de filas de una matriz.

La figura 3.6 muestra la aceleración del FPGA en relación con la CPU. Por ello, el algoritmo de rotación de Givens se ha implementado en Matlab 2010 a fin de poder medir el tiempo de procesamiento de la CPU. Este tiempo se ha calculado para cada tamaño de la matriz, de 2 a 120 filas y se ha dividido por el tiempo de procesamiento FPGA.

Como se puede ver, el tiempo de procesamiento de la CPU es mayor que el tiempo de procesamiento mostrado por FPGA, concretamente un 12 % a 41 % mejorado. En este caso, se utilizó un Core 2 Duo (E 8400, 3Ghz) para estimar los tiempos de la CPU.

Todos estos datos positivos hacen desarrollar nuevas estructuras basadas en el primitivo diseño con el objetivo de mejorarlo en la medidad de lo posible. En los sucesivos apartados se muestran nuevos diseños en los que se trata de incrementar el procesamiento de matrices con el coste de incrementar el uso de recursos de la FPGA.

3.3 Arquitectura iterativa elemental propuesta para la Factorización Givens QR

De un modo primario y más elemental, el diseño de la arquitectura para llevar a cabo una rotación de Givens es factible a través de la comunicación entre los dos elementos CORDIC, ejecutándose primero el CORDIC vectorización y después el CORDIC rotación. Este orden viene impuesto por el algoritmo de Givens y sus dependencias (OB.I y OB. II), el primer CORDIC computará el ángulo de rotación para que el segundo CORDIC tenga el ángulo de referencia para la rotación.

En nuestro artículo [52], implementamos una arquitectura iterativa donde las operaciones se superponen para llevar a cabo una rotación de Givens. La pauta es muy sencilla y de una traza lineal en [52], ya que primero se computa el ángulo, y a continuación se produce la rotación de los elementos restantes a la vez que se computa el ángulo para la próxima etapa del método Givens.

Sin embargo, esa linealidad presenta ciertas desventajas y es posible desarrollar un mejor plan para la ejecución de las operaciones. A partir de las siguientes dos directrices el diseño puede incrementar su rendimiento:

1. Reducción de tiempos ociosos y mejora en la sincronización. El primer problema detectado es el alto tiempo de inactividad en ambas operaciones básicas debido a la dependencia operacional. Es cierto que se consigue un buen solapamiento operacional en [52] [49] y [50]; pero aún se puede conseguir más, si aumentamos el grado de flexibilidad independizando las dos operaciones en distintas secciones, prácticamente no habrá tiempos de espera entre ambas. La conexión entre las distintas secciones puede ser mediante una estructura FIFO (First Input First Ouput) que almacena los ángulos, mejorando al mismo tiempo la sincronización entre operaciones.

2. Replicación hardware y paralelización del algoritmo. La segmentación del algoritmo en la rotación es un buen paso inicial para reducir tiempo, tal y como se ha realizado en el [52]. Sin embargo, esta técnica no tiene porque ser suficiente. Para matrices de un tamaño medio, la operación de rotación tiene una mayor carga temporal, por ello, aumentar la cantidad de elementos rotadores reducirá tiempos de cómputo. Además, como bien es sabido, la matiz triangular R requerirá de una menor cantidad de rotaciones que la matriz Q, este desequilibrio debe de reflejarse y ser compensado de alguna forma en el algoritmo.

Para dar a conocer la estructura de la versión simplista del algoritmo de factorización QR, la Fig 3.7 representa cómo es el diseño. En dicha figura se diferencia la sección izquierda encargada del cómputo del ángulo y coincide con la operación básica 1. Mientas que en la parte derecha, la otra sección hace alusión a la operación básica 2 (OB. I) donde se computa la rotación del vector (OB. II).

En la Fig 3.7 ambas secciones tienen dos elementos en común. El primero de ellos es el dispositivo que almacena los datos situado en la parte central superior, éste se encuentra compuesto por la matriz de entrada (M) junto con una matriz identidad (I) de igual tamaño. El almacenamiento de los datos (matriz de entrada) es a través de bloques dual–port RAM, que presenta la ventaja de tener dos puertos activos para realizar dos operaciones simultáneas, tanto lecturas como escrituras. En este caso concreto, uno es utilizado para la lectura de elementos entrantes, mientras que el otro está destinado a la escritura de los elementos procesados.

El otro elemento en común es una estructura de almacenamiento de datos intermedio FIFO. Su

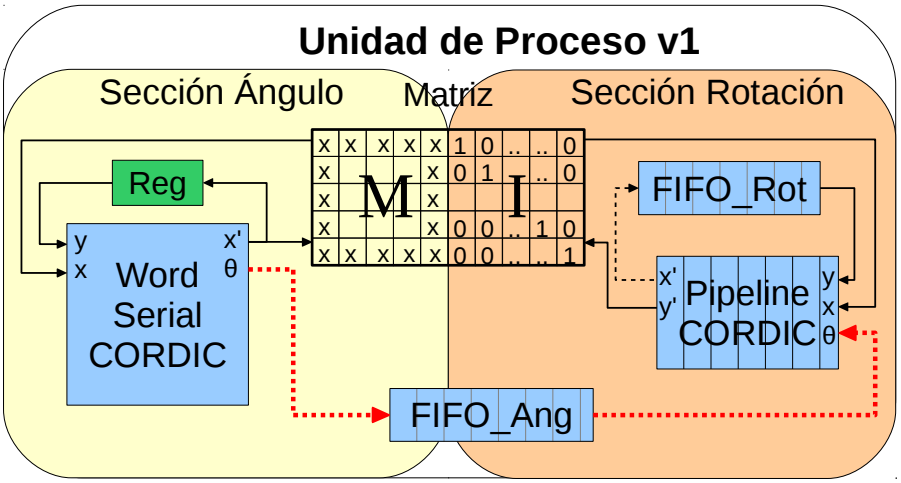


Figura 3.7: Arquitectura simplista para la factorización QR según el método Givens.

cometido es almacenar todos aquellos ángulos de rotación insertados por la sección del ángulo, los cuales se irán extrayendo a medida que sean necesarios para la rotación.

La sección del ángulo es quién rige el algoritmo, ya que se intenta que dicha sección este la menor cantidad de tiempo ociosa, de tal modo que los ángulos se vayan calculándose con la mayor brevedad posible. Consiguiendo de este modo que siempre haya un ángulo disponible para comenzar nuevas rotaciones. Se ha de tener en cuenta que no es posible computar nuevos ángulos de rotación sobre posiciones que aún no han sido recalculadas o rotadas para la columna actual.

Hacer notar que entre ambas secciones existe un sincronismo de retroalimentación o feedback, en el que por un lado se requieren nuevos ángulos de rotación que no pueden avanzar en el cálculo mientras no se hayan rotado ciertas posiciones de la matriz. Por tanto, una aceleración en la sección de rotación, reducirá el tiempo de espera de ángulos para realizar el cómputo en la sección del cálculo de ángulos.

3.3.1 Sección Ángulo

En esta sección se desempeña la calculación de los ángulos de rotación, a través de un CORDIC word-serial en modo vectorización (véase 2.4.2) y un registro que almacena el valor rotado por el CORDIC para la coordenada X'. El objetivo es la reutilización de ese dato en la siguiente etapa como entrada de la componente Y. La utilización del registro intermedio es meramente de almacenamiento temporal, evitando así, la doble lecturas de los mismos bloques RAM con valor que realmente se obtiene en un ciclo anterior.

En la Fig 3.7se muestra como esta sección presenta dos entradas, una señal X y otra Y. La entrada X del CORDIC proviene directamente del conjunto de datos entrantes compuesto por la concatenación de la matriz entrante y la su matriz identidad $(M|I)$. Mientas que la entrada Y coincide con la realimentación de la salida X', exceptuando en la primera iteración que proviene directamente de la matriz entrante. En cuanto a las salidas del CORDIC el valor Y' se desprecia porque ya sabemos que al vectorizar las entradas, su salida es 0. La salida X' sólo se escribe en la matriz cuando coincide con un valor diagonal.

Aunque ya se ha entredicho, el ángulo de salida θ es almacenado directamente en la cola de

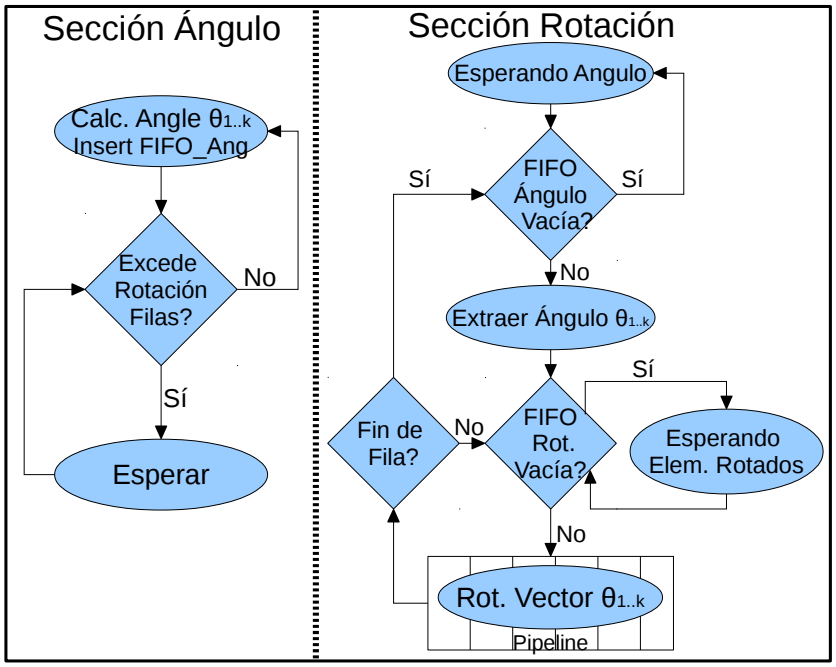


Figura 3.8: Diagrama de secuencia de operaciones propuesto para la sección ángulo y la sección rotación, ambos se llevan a cabo a la vez.

ángulos denominada en la Fig 3.7 como FIFO_Ang. El orden de lecturas se ajusta al descrito en la sección 2.3.2, de abajo hacia arriba y una vez alcanzado el elemento diagonal, se comienza por el elemento inferior desplazado una columna hacia la derecha.

El diagrama ilustrado en la Fig 3.8 se muestra cómo es el comportamiento de la máquina de estados para la sección ángulo. Tales comparaciones se efectúan para evitar no cumplir con las restricciones del algoritmo y sus dependencias posicionales.

El número de ángulos de rotación calculados para una matriz $m \times n$ es $k = \frac{m \cdot (m-1)}{2}$. En último lugar, hacer notar que el tamaño de esta sección es cuantiosamente inferior que la sección rotación debido al uso de un CORDIC Word-Serial o iterativo.

3.3.2 Sección Rotación

Como ya es sabido, la misión de esta sección es la rotación de filas adyacentes y se encuentra esquematizado en la Fig 3.7. Para ello, se va a utilizar un elemento CORDIC segmentado y en modo rotación, véase la sección 2.4.2 para más detalle. En este caso se ha optado por la opción segmentada debido a que los datos entrantes no están formados por un único valor como sucede con el cómputo de ángulos, ahora se ha de tratar la fila completa de la matriz junto con su adyacente. Al ser segmentado se obtiene un dato por ciclo y se consigue una mayor equilibro entre ambas secciones.

Las entradas del CORDIC modo rotación son tres: X,Y y θ. Y siempre es la fila adyacente e inmediatamente inferior a la fila X de la matriz, mientras que θ es el ángulo de rotación obtenido de la FIFO_Ang. En esta sección, la salida los valores devueltos son compuestos por la doble salida del componente CORDIC (X',Y') paralelamente. El resultado de Y' forma parte de los valores finales y será escrito sobre la matriz. Contrariamente, los elementos X' rotados corresponden

a la entrada de la siguiente fila que queremos leer, por lo que serán reasignados como entradas sobre la componente Y. De este modo se consigue evitar una escritura y posterior lecturas en los bloques RAM, consecuencia de un posible colapso de puertos de entrada/salida evitada gracias a la estructura FIFO intermedia, denominada FIFO_Rot.

Nuevamente, tal y como sucede en la sección ángulo, la salida del CORDIC almacena una fila rotada pero en este caso está compuesta por una ristra de valores, por ello el diseño de una cola mediante registros es una estructura ideal. Además, no es factible ocupar ambos puertos para leer los datos, de este modo y por medio de la FIFO_ROT reducimos las lecturas en los bloques RAM.

El esquema de la secuencia de operaciones se presenta en el lado derecho de la Fig. 3.8. El diagrama muestra cuándo es posible rotar un nuevo par de elementos adyacentes, esta acción está restringida por las diferentes condiciones y dependencias del algoritmo. En el inicio del esquema de la Fig. 3.8, siempre empieza con la extracción de un ángulo de rotación de la FIFO_Ang, en otro caso el algoritmo está en estado de espera. Una vez extraído dicho ángulo, se comprueba si la FIFO_Rot contiene elementos o aún están en el proceso de rotación. Si no contiene elementos, espera hasta que estos sean computados, en otro caso, se inicia la rotación de los vectores hasta que se alcanza el final de fila. Una vez finalizamos con la fila, volvemos al punto de inicio del algoritmo para seguir rotando nuevos elementos.

El número de elementos rotados ($nrot$) relativos a una matriz de entrada $A^{m\times n}$ es equivalente al de la Ec. 3.1, donde la primera parte de la suma corresponde a la matriz R y la segunda a Q. Como se puede ver, el número de rotaciones de esta segunda parte es considerablemente mayor que la sección de ángulo.

$$nrot = \sum_{i=1}^{(m-1)} (i) + \sum_{j=1}^{(m-1)} (j \cdot n) \tag{3.1}$$

3.4 Arquitectura iterativa avanzada para la Factorización Givens QR

Una vez conocida una arquitectura básica para factorizar una matriz, es posible llegar más lejos. En este nuevo diseño la novedad reside en la replicación de elementos CORDIC a fin de aumentar las rotaciones. En este caso, queremos mitigar el cuello de botella del algoritmo aumentando el cómputo dedicado a la rotación de filas. Éste cambio tendrá gran repercusión y será especialmente notable para matrices de tamaño medio.

En la Fig. 3.9 se muestra una arquitectura genérica para la factorización de matrices. En comparación con la Fig 3.7, la arquitectura avanzada se descompone en tres áreas, una más que la versión simple. En la parte superior izquierda se muestra un área exactamente igual que en la versión sencilla dedica exclusivamente al cómputo de ángulos de rotación. Las otras dos secciones se dedican a la rotación de las matrices de entrada M y I para alcanzar las matrices Q y R. A simple vista, comparando este diseño con la versión simplista del apartado 3.3, la sección rotación se ha segregado en dos nuevas secciones diferenciadas por la matriz de entrada. La principal razón de esta división es debido al ritmo de extracción de ángulos, la sección de matriz Q es mucho más lenta que la otra, como refleja en la Ec. 3.1.

Describiendo con mayor detalle las secciones de la Fig. 3.9 nos encontramos con las siguientes características:

- **Sección Ángulo**. Esta sección en la encargada de calcular los ángulos de rotación a través

de un CORDIC vectorización word-serial y un registro intermedio, exactamente igual que la versión más simplista. En esta nueva propuesta, los datos entrantes se han desvinculado de la matriz I, ya que para el cómputo del ángulo solo son requisito los de la matriz entrante M.

- **Sección rotación R**. Esta parte del diseño se encarga sólo y exclusivamente de la rotación de la matriz R, compartiendo puertos con la sección Ángulo. Los ángulos se extraen como en la versión anterior, a partir de una estructura FIFO en la que éstos son acumulados. En esta parte de la arquitectura se permite la replicación de componentes de rotación formados por CORDIC rotación segmentado y su FIFO_Rot correspondiente, capaces de llevar a cabo la rotación de elementos. El número de componentes depende de un parámetro configurable dentro de la arquitectura distinguido por la variable n. Un dato importante es el hecho que todos los componentes replicados están conectados a la misma FIFO que almacena los ángulos, por lo que la extracción es simultánea para las n-replicas.

- **Sección rotación Q**. Esta sección está situada en la parte inferior de la figura. Su arquitectura es prácticamente un calco de la sección R, con la diferencia que exclusivamente trabaja con la matriz I además de no compartir puertos de lectura y escritura con ningún otro componente. Nuevamente utiliza una FIFO_Ang para ir leyendo lo ángulos de rotación. Sus elementos de rotación también están formados por un CORDIC rotación segmentado y una FIFO_Rot, los cuales son replicados a través del parámetro genérico p, designado al control de replicaciones hardware de la sección Q.

Destacar la flexibilidad de la arquitectura, ya que gracias a dos parámetros, n y p, es posible configurar la cantidad de elementos encargados de rotar elementos por sección. Estos números están expresados como potencia de dos en la arquitectura y no tienen porque coincidir. Que el parámetro p sea mayor que el n será un hecho relevante, ya que la carga operacional se encuentra en desequilibrio, ganado con mayor peso la matriz I.

Recalcar que al disgregar la matriz originalmente concatenada en dos partes $(M|I)$, se consigue duplicar la posibilidad de realizar lecturas y escrituras sobre los bloques RAM a través de los puertos, donde el tamaño de éstos dependerá de los parámetros p y n. Aunque la paralelización obligue a replicar la FIFO_Rot, desde el punto de vista de la utilización de recursos no es significante, ya que la suma de estos componentes equivale al tamaño de diseño más simplista propuesto en el apartado 3.3.

3.5 Diseño y sincronismo de las arquitecturas

El objetivo del diseño es construir una arquitectura con la menor cantidad de recursos, optimizando el uso de estos, a la vez que es flexible y reconfigurable.

Los elementos de la matriz son representados en formato punto fijo, en nuestro caso han servido de análisis los tamaños de 16, 32 y 48 bits de tamaño. Por ello se considera importante que la arquitectura pueda implementarse para un valor genérico de tamaño de palabra. Como consecuencia se ha definido el parámetro *Word-Length* que reconfigura el núcleo con el tamaño de palabra deseado.

Además, la dimensión máxima de la matriz también es configurable a través del parámetro *Maximum Range*. Gracias a este parámetro, podemos computar las matrices dentro del tamaño máximo establecido sin la necesidad de re-sintetizar el diseño. Cierto es que mientras mayor sea este parámetro, mayor será la cantidad de bloques RAM utilizados. Lo parte positiva de seguir

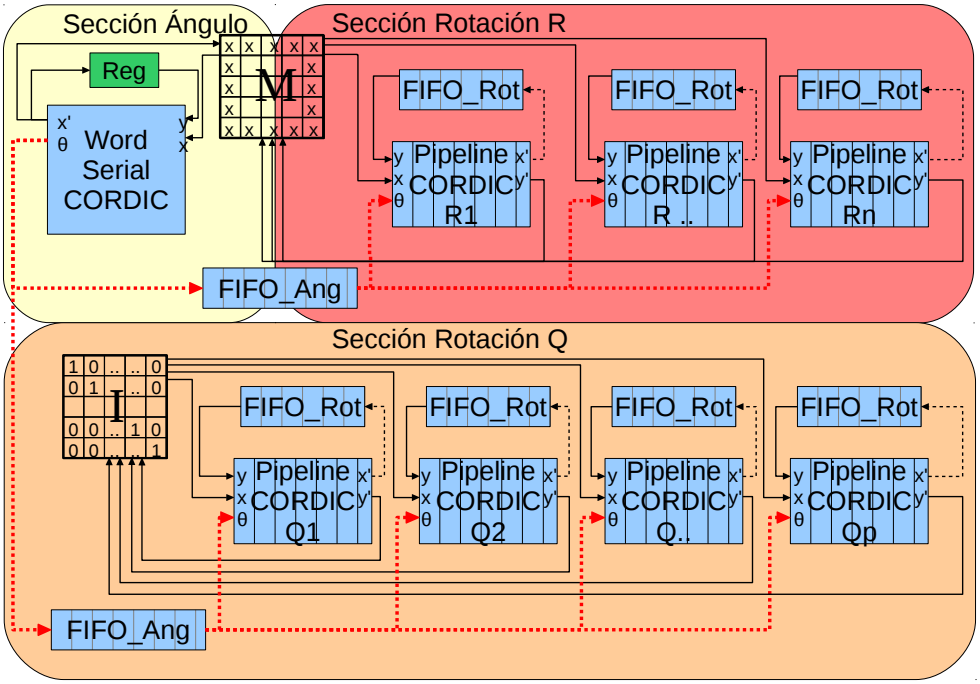


Figura 3.9: Arquitectura propuesta donde las matrices Q y R son factorizadas en paralelo a través de varios CORDICs.

una filosofía parametrizable es que cualquier matriz que se encuentre en el rango establecido por el parámetro *Maximum Range* puede ser factorizada.

Recordad los parámetros n y p de la arquitectura genérica, que aportarán mucho juego en las distintas configuraciones. Con estos parámetros elegimos la cantidad de elementos rotadores en cada sección, permitiendo atacar al cuello de botella del algoritmo. El hecho de poder configurar la cantidad de CORDIC y colas para los elementos de las filas n y p tendrán un peso importante la cantidad de recursos de la FPGA utilizados.

Ya por último hacer mención al sincronismo y control que se realiza mediante una máquina de estados de Moore. La máquina de estados recuerda al modelo de paso de mensajes. De acuerdo con el modelo la *Sección ángulo* (remitente) envía los ángulos (mensaje) de rotación a la FIFO de ángulos (buffer de comunicación), cuya función es regular el flujo de rotaciones en la *sección Q* y *sección R* (receptores). Aunque el papel principal del control es la FIFO_Ang, existe contadores para el control de lecturas y escrituras en la matriz, evidentemente con un itinerario distinto para cada sección, recorrido de una matriz triangular superior para R y recorrido de una matriz completa para la matriz I.

3.6 Rendimiento, análisis y valoración

Basándonos en los diseños propuestos anteriormente, se han diseñado dos núcleos VHDL parametrizables para la descomposición QR. Ambas arquitecturas, simples y multi-CORDIC, nos permiten configurar: la precisión del bit, el tamaño de la matriz de entrada y el número de elementos encargados de la rotación. Estos núcleos han sido simulados, evaluados y sintetizados usando Xilinx ISE 14.3, para su posterior implementación en Virtex-6 XV6VLX240T FPGA. El

Dispositivo	**Xilin Virtex 6 XC6VLX240T**			
Conf. Núm. CORDIC (n, p)	**R:1, Q:1**	**R:1, Q:2**	**R:2, Q:2**	**R:2, Q:4**
Núm. Slice Registers	10.081(3%)	14.197(4%)	18.370(6%)	26.553(8%)
Núm. Slice LUTs	10.592(7%)	14.731(9%)	18.748(12%)	26.671(17%)
Slice LUTs/N. Replicaciones (10^3)	5,3	4,9	4,7	4,4
Núm. fully used LUT-FF pairs	8.715(75%)	12.513(78%)	16.126(78%)	24.075(85%)
Período Mínimo	3,560ns	3,441ns	3,294ns	3,547ns
Núm. Block RAM36	156(40%)	132(35%)	142(37%)	130(34%))
Núm. Block RAM18	5(1%)	5(1%)	8(1%)	10(1%)
Configuración FIFO	×	×	×	×
Tamaño Palabra	32	32	32	32
Profundidad Ángulo.	16.384x2	2.048x2	8.192x2	10.24x2
Profundidad Rotación	256x3	256x5	128x6	128x10

Tabla 3.1: Evaluación del rendimiento hardware para las arquitecturas de factorización QR con diferentes configuraciones de los parámetros de replicación n y p, para un longitud máxima de matriz 256 con una representación numérica de 32 bits de tamaño de palabra.

algoritmo CORDIC y las estructuras FIFO se han implementado utilizando la Herramienta Xilinx CORE Generator.

Para confirmar la idoneidad del núcleo propuesto para matrices de tamaño medio, se han realizado diversas implementaciones para una amplia gama de tamaños de matriz con diferentes configuraciones.

3.6.1 Evaluación para distintas configuraciones QR

En la Tab. 3.1 y Tab. 3.2 se presentan los resultados de la evaluación de la arquitectura simple en comparación de varias configuraciones para la *sección Q* y *sección R*. La primera fila de la tabla denominada Conf. Núm. CORDIC (n, p), indica la cantidad de elementos CORDIC y FIFOS replicados en cada una de las secciones. Por lo tanto, para una configuración (R:2,Q:4), coincidiendo con la columna 5, significa que en la *sección R* está compuesta por 2 CORDICs y 2 FIFOS_Rot, mientras que la *sección Q* cuenta con 4 replicaciones. Como puede observarse, esta fila sólo indica el valor de los parámetros n y p.

Observando las filas en las que queda reflejado el consumo de slices en las tablas Tab. 3.1 y Tab. 3.2, se percibe un claro incremento de los registros y LUTs. Esto se debe especialmente al incremento de CORDICs en cada una de las secciones. Resulta de especial interés la relación Slices LUTS/ Número de replicaciones donde se aprecia una disminución del uso de LUTs a medida que se incrementan la replicación del hardware. Este hecho también es confirmado en la fila Núm. fully used LUT-FF pairs donde se muestra el porcentaje de Slice LUTs y Slice Registers emparejados, comenzando con un 75% para la configuración R:1,Q:1 y alcanzando hasta un 87% en la

Dispositivo	**Xilin Virtex 6 XC6VLX240T**	
Conf. Núm. CORDIC (n, p)	**R:4, Q:4**	**R:4, Q:8**
Núm. Slice Registers	34.788(11%)	50.867(16%)
Núm. Slice LUTs	34.801(23%)	50.692(33%)
Slice LUTs/N. Replicaciones (10^3)	4,3	4,2
Núm. fully used LUT-FF pairs	31.287(84%)	46.500(87%)
Período Mínimo	3,602ns	3,602ns
Núm. Block RAM36	136(36%)	128(34%)
Núm. Block RAM18	12(1%)	22(2%)
Configuración FIFO	×	×
Ancho Palabra	32	32
Profundidad ángulo	4.096x2	256x2
Profundidad rotación	64x12	64x20

Tabla 3.2: Evaluación del rendimiento hardware para las arquitecturas de factorización QR con diferentes configuraciones de los parámetros de replicación n y p, para un tamaño máximo de matriz 256 con una representación numérica de 32 bits de tamaño de palabra.

configuración R:4,Q:8; evidencia clara del buen mapeo y enrutamiento de las celdas de la FPGA.

Otro dato de gran notoriedad es la frecuencia máxima de trabajo que la arquitectura puede soportar sin que se produzcan retardos entre las componentes. Como se aprecia en la Tab. 3.1 y Tab. 3.2, el periodo mínimo comprende entre los 3,3 y 3,6 nanosegundos, sobrepasando el valor de los 300 Mhz.

En la parte central de las dos tablas se muestra la relación entre el número de bloques de memoria utilizados y el número de colas FIFO replicadas. La mayoría de bloques RAM36 son para almacenar las matriz Q y la matriz R de un tamaño de 256×256 , mientras que la cantidad de bloques RAM18 se han utilizado para implementar las FIFOs del circuito. La significativa reducción en el consumo de bloques RAM36 desde la configuración (R:1,Q:1) hasta configuraciones con mayores cantidad de elementos se debe a la profundidad necesaria de la FIFO_Ang.

Aquella configuraciones cuyo parámetro p es el doble que el n presentan un menor consumo de Bloques RAM36, ya que no requiere de una FIFO_Ang tan profunda, hecho provocado por una mayor cantidad de rotaciones en paralelo que afecta a la velocidad de extracción de ángulos mayor.

Por último, en la parte inferior de las tablas Tab. 3.1 y Tab. 3.2 se recopilan los datos de implementación de las FIFOs, el primer campo corresponde al tamaño de palabra del número almacenado, el segundo muestra la profundidad de las FIFO_Ang, y el tercero muestra la profundidad de la FIFO_Rot. Como puede verse, aquellas arquitecturas con número distinto de elementos rotadores en R y Q requieren de FIFOs más pequeñas para un correcto funcionamiento. Este hecho muestra como aquellas arquitecturas con mayor número de elementos de rotación reparten la carga y aligeran la estructura provocando a su vez la reducción del número de Bloques RAM18 en aquellas configuraciones dispares.

3.6.2 Evaluación para distintos tamaños máximo de matriz

Dispositivo	**Xilin Virtex 6 XC6VLX240T**					
Configuración	**2 Replicas R Section**			**2 Replicas Q Section**		
Max. Rango Matriz	8	16	32	64	128	256
Núm. Slice Registers	17.717	17.730	17.741	17.806	17.915	18370
Núm. Slice LUTs	17.561	17.658	17.661	17.760	17.997	18748
Núm. Block RAM18	8	8	8	8	8	8
Núm. Block RAM36	4	4	4	8	36	142
Tamaño FIFO Ángulo	16	32	64	256	2064	8192
Tamaño FIFO Rotación	16	16	16	32	64	128

Tabla 3.3: Evaluaciones de rendimiento hardware para la descomposición QR con distintos rangos de tamaños de matriz, desde 8 hasta 256 filas, con 32 bits de longitud de palabra por elemento bajo una configuración de 2 replicaciones en R y 2 replicaciones en Q.

Dispositivo	**Xilin Virtex 6 XC6VLX240T**					
Configuración	**2 Replicas Q Section**			**4 Replicas Q Section**		
Máx. Rango Matriz	8	16	32	64	128	256
Núm. Slice Registers	25.881	25.915	26.014	26.010	26.073	26.553
Núm. Slice LUTs	25.596	25.666	25.737	25.870	25.966	26.671
Núm. Block RAM18	12	12	12	12	12	10
Núm. Block RAM36	6	6	6	8	32	132
Tamaño FIFO Ángulo	16	32	32	64	256	1024
Tamaño FIFO Rotación	16	16	16	32	64	128

Tabla 3.4: Evaluaciones de rendimiento hardware para la descomposición QR con distintos rangos de tamaños de matriz, desde 8 hasta 256 filas, con 32 bits de longitud de palabra por elemento bajo una configuración de 2 replicaciones en R y 4 replicaciones en Q.

En este subapartado se analizará cómo afecta el tamaño de una matriz al rendimiento y a los recursos utilizados en función del tamaño máximo de matriz admitido bajo una misma arquitectura. Para realizar la evaluación se han utilizado dos arquitecturas que se encuentran en la mitad del rango estudiado anteriormente, las configuraciones (R:2, Q:2) y (R2, Q:4). Los resultados de las tablas Tab. 3.6.2 y Tab. 3.6.2 muestran qué ocurre en el diseño cuando se modifica el parámetro genérico *Máx. Range* que coincide con la fila Máx. Rango Matriz. Los valores testeados como límite del tamaño de la matriz fueron 8 , 16, 32, 64, 128 y 256 filas.

Ambas tablas, Tab. 3.6.2 y Tab. 3.6.2, muestran un crecimiento similar para la cantidad de Slices registros y LUTS a medida que aumenta el tamaño máximo de matriz admitido. En las dos tablas, la diferencia de slices entre un tamaño de matriz de máximo de 8 filas y 256 filas no supera los 1.000, tanto para registros como para LUTs.

Todo lo contrario, ocurre con los bloques RAM36, para una misma configuración y distintos

tamaños de matriz el crecimiento es considerablemente grande. Esto es debido a la necesidad de adquirir un mayor espacio de almacenamiento adicional para la matriz, representando un crecimiento cuadrático.

Si comparamos la profunidad de las FIFO en las dos tablas, Tab. 3.6.2 y Tab. 3.6.2, se deduce la siguiente premisa, el consumo de bloques RAM aumenta a un mayor ritmo para aquellas arquitecturas que sus elementos rotadores están equilibrados en la sección Q y en la sección R. Para la configuración (R2, Q:4) con un tamaño de matriz máximo de 256 como muestra la tabla Tab. 3.6.2 se consumen 132 bloques RAM36, sin embargo, este número aumenta hasta en 10 bloques RAM más para la configuración (R2, Q:2) con un tamaño máximo de 256 filas. Razonablemente, esta reducción se debe a una velocidad mayor de extracción de ángulos para aquellas configuraciones con más elementos destinados a la rotación en la sección Q que en R.

En resumen, la arquitectura muestra una gran independencia del tamaño máximo de la matriz permitido. Por lo que computar matrices de un tamaño superior al mencionado afectará notablemente al crecimiento de los bloques RAM18 y RAM36 de la FPGA, teniendo una menor repercusión sobre otros recursos.

3.6.3 Evaluación del tiempo de procesamiento

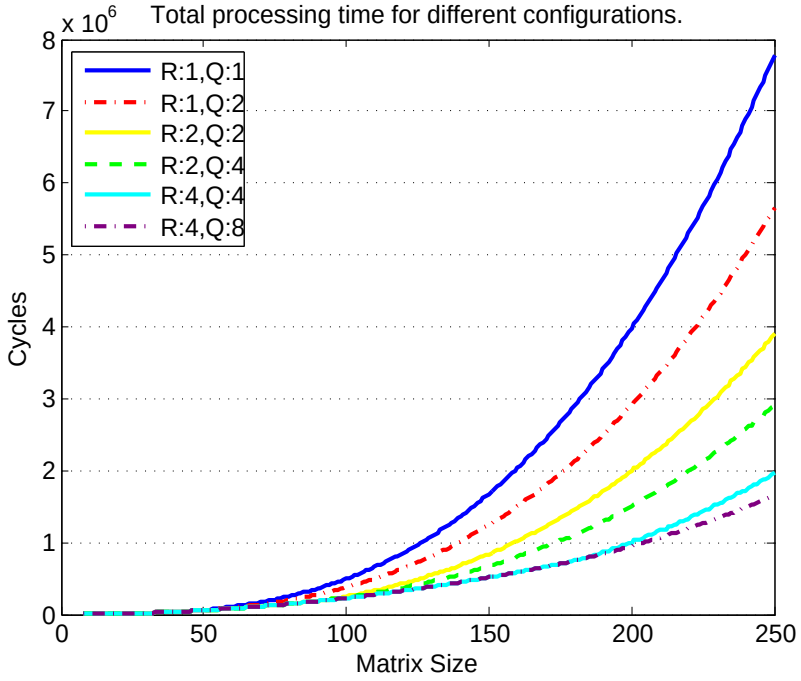


Figura 3.10: Tiempo de procesamiento de diferentes tamaños de matriz con varias configuraciones propuestas.

La Fig. 3.10 representa el número total de ciclos de cómputo para una selección de configuraciones con respecto a un tamaño de matriz predefinido. Las configuraciones seleccionadas son aquellas evaluadas en las subsecciones y tablas anteriores.

Continuando con el análisis de la Fig. 3.10, se observa que todas las arquitecturas presentan un crecimiento exponencial, siendo las arquitecturas con una cantidad mayor de componentes destinados a la rotación las que presenta una curva de crecimiento menos acentuada. Es destaca-

ble el amplio hueco existente entre la peor de las configuraciones (R:1, Q:1) y las más completa (R:4, Q:8) para un tamaño de matriz de 250 filas y columnas; el intervalo supera la cifra de los seis millones de ciclos. Hecho que hace suponer que ocurrirá lo mismo con arquitecturas superiores.

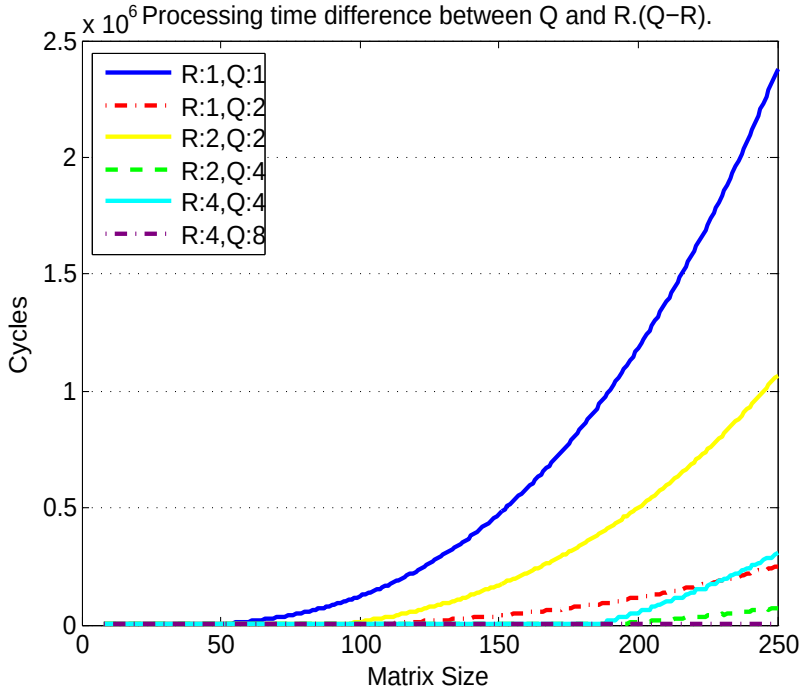


Figura 3.11: Representación del intervalo de finalización entre Q y R (Q-R) para diferentes configuraciones y tamaños de matriz.

Como la arquitectura propuesta es dividida en dos secciones de rotación Q y R, el tiempo de procesamiento resultante es la cantidad máxima de ciclos en cada una de las secciones. Evidentemente siempre coincide con el tiempo de procesamiento de la sección Q, puesto que es la sección con una mayor carga de cómputos. Para examinar esta diferencia de tiempos de procesamiento entre secciones de la arquitectura propuesta en la Fig. 3.11 se ilustran tales resultados.

Resulta interesante apreciar en la Fig. 3.11 cómo para las arquitecturas con un número de replicaciones iguales en ambas secciones, la principal causa de su dilatación en el tiempo de procesamiento es provocado por la sección Q. Por ejemplo para configuración (R:1, Q:1) y un tamaño de matriz de 250, la diferencia de tiempo de procesamiento entre secciones en enorme, situándose el valor en torno a los 2.370.000 ciclos de diferencia, tiempo en el que la sección R se encuentra ociosa. En el caso opuesto se encuentra la configuración (R:4, Q:8) en el que a simple vista su crecimiento parece lineal; el intervalo entre la sección Q y R para un tamaño de matriz de 250×250 no supera los 64 ciclos. Resaltar el punto de cruce que tiene lugar para un tamaño de matriz de 230×230 entre las configuraciones (R:1, Q:2) y (R:4, Q:4), ya que el tiempo de inactividad de la arquitectura con un total de 8 CORDICs supera a la arquitectura con tan solo 3 CORDICs.

Por último, enfatizar sobre las diferencias de inactividad que sufren las distintas configuraciones de la arquitectura, realzando aquellas con un número dispar de replicaciones CORDIC entre Q y R, es decir (R:1, Q:2), (R:2, Q:4) y (R:4, Q:8).

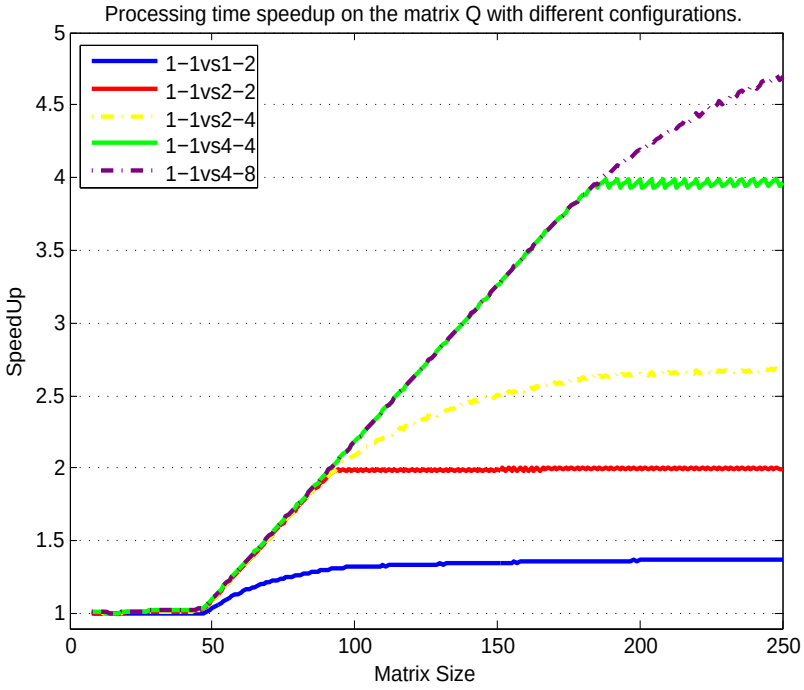


Figura 3.12: Comparación de las diferentes configuraciones para el procesamiento de la matriz Q con respecto a la configuración más básica (R:1, Q:1).

3.6.4 Aceleración del tiempo de procesamiento

La Fig. 3.12 ilustra la aceleración del tiempo de procesamiento entre las diferentes configuraciones utilizando como referencia la configuración más simple (R:1, Q:1). Como es de esperar, la configuración (R:4, Q:8), con un mayor nivel de paralelismo, es la que muestra un mayor rendimiento al procesar medias y grandes matrices. El rendimiento alcanzado es de hasta 4,7 veces más rápido que la arquitectura (R:1, Q:1).

En el gráfico de la Fig. 3.12 se presentan tres puntos predominantes en los que la variación de rendimiento es bien notable. El primero de ellos aparece para el rango de matrices comprendido entre 45 y 46 filas, en el que se aprecia una primera bifurcación y estancación por la arquitectura (R:1, Q:2) en comparación con las demás. El segundo escalón se produce para tamaños de matriz que están entorno a los 94-96 filas, donde la arquitectura (R:2, Q:2) y (R:2, Q:4) comienza a mermar su aceleración. Y por último, aparece un nuevo desvío para entorno a las 188 filas. Todos estos cambios son provocados por una mejor explotación de los recursos, especialmente CORDIC.

Los puntos de inflexión de la Fig. 3.12 no son coincidencia, están relacionados con el tiempo de rotación de los elementos CORDIC segmentados. En este caso, como se ha utilizado un diseño por defecto basado en las herramientas para generación de núcleos la latencia de los elementos CORDIC es de 42 ciclos o etapas, que coincide con el primer escalón mencionado anteriormente. Como puede observase, los siguientes instantes de cambio de ritmo coinciden con múltiplos del número de etapas de los dispositivos CORDIC.

Para cerrar este tema, añadir que tal y como se ve en la Fig. 3.12 parece que las líneas representadas para las distintas configuraciones se encuentra superpuestas, aunque no es un hecho de gran trascendencia, si se hace zoom sobre éstas líneas se aprecia cómo las arquitecturas con mayor cantidad de elementos rotadores muestran un mejor resultado que las demás.

Evaluación del error para distintos tamaños de palabra

El error de los resultados está relacionado con dos aspectos, el error de redondeo y el error asociado a los componentes CORDIC ligado al número de micro-rotaciones. Si el número de etapas que a su vez coincide con las micro-rotaciones es demasiado bajo, la precisión también será baja.

En la tabla 3.6.4 se muestra un resumen con el error estimado para las longitudes de palabra para 32 y 48 bits. La prueba para estimar tal error se ha realizado para un total de 100.000 matrices aleatorias con un tamaño de 128×128. La comprobación se calcula fácilmente, multiplicando Q^t por la matriz R obtenida menos la matriz original A ($Q' * R - A$). Dicha comprobación ha sido realizada a través del software MATLAB que utiliza una representación flotante de los números y su precisión se considera prácticamente exacta.

En los resultados obtenidos de la Tab. 3.6.4, se muestran valores más que aceptables, siendo de hasta 8 dígitos correctos para 32 bits y de 13 dígitos para una tamaño de palabra de 48 bits.

Fórmula Medición Error.	$Q' * R - A$
Error Medio 32bits	1.325e-8
Error Medio 48bits	1.995e-13

Tabla 3.5: Evaluación del error para 32 bits y 48 bits de tamaño de palabra.

3.6.5 Comparación Experimental

Arquitecturas sistólicas

Este apartado ha sido añadido porque realmente hay otro tipo de arquitecturas para realizar la factorización QR de matrices, los arrays sistólicos. Aunque la comparación es bastante compleja hay varios detalles que son realmente relevantes y deben de tenerse en cuenta. Para conocer más detalle de este tipo de arquitecturas véase el apartado 4.1.

Observando a groso modo en la literatura encontramos varios tipos de arquitecturas que se fundamentan en el diseño sistólico para factorizar matrices. Todo ellos son propuestos para resolver un tamaño de matriz fijo como puede ser 4×4[44][45], 8×8[46][47] o como máximo 16×16[53]. Lo primero que nos damos cuenta es el reducido tamaño de las matrices de entrada. Esto es debido a la gran cantidad de área y recursos que estos diseños requieren en una FPGA.

En [46], el consumo de slices para factorizar una matriz a lo sumo de 8×8 matrix es de 16.581 LUTs, 22.978 registros y 224 bloques DSP. Sin embargo, nuestro diseño propuesto logra resolver matrices 256×256 utilizando una cantidad de slices similar y sin utiliza ni un bloque DSP.

Otro ejemplo es [44] donde el número de slices utilizados está entre 2,609 (11 %) y 3,617 (15 %) con una longitud de palabra que varía entre los 18 a 28 bits sobre una FPGA Virtex5 XC5VTX150T y un tamaño 4×4. Sin embargo, si esta área es extrapolada para calcular una matriz mayor y con una longitud de palabra más grande, la matriz más grande que tiene cabida en una FPGA Virtex-6 XC6VLX240T no supera un tamaño de 14×14 con un formato de 16 bits. Y si además se incrementara el tamaño de representación numérica a 28 bits, ni siquiera podría sintetizarse una arquitectura capaz de resolver una matriz de tamaño 11×11. Este hecho demuestra nuevamente el elevado coste en recursos con respecto a estas arquitecturas y sus dificultades para resolver matrices de tamaño medio y la necesidad de crear nuevas alternativas como las

arquitecturas iterativas.

Arquitecturas similares

En la literatura también existen arquitecturas que coinciden en ciertos aspectos de nuestra arquitectura propuesta, como por ejemplo las citadas en los siguientes trabajos [49] [50][51][52]. Pero ninguna de ellas muestra la versatilidad que nosotros proponemos.

En los diseños propuestos [49] [50] se llevan a cabo la factorización a través de un comportamiento iterativo. Como en nuestro diseño se han basado en las rotaciones de Givens a través del algoritmo CORDIC. Aunque también diferencia entre dos secciones de cómputo, su alcance está limitado a matrices 4×4, sin llevar a cabo ningún tipo de paralelización.

No es fácil lograr una equidad comparativa entre nuestra arquitectura y la del artículo [51], puesto que se han implementado los diseños sobre distintas plataformas FPGA. Pero hay ciertas evidencias que infieren que nuestra propuesta resulta ventajosa.

La primera evidencia destacable es la estrategia seguida en nuestra arquitectura, el objetivo siempre ha sido rotar las filas lo antes posible para evitar tiempos de inactividad en lo ángulos, incrementando el número de elementos destinados a la rotación en la medida de lo posible. Por el contrario, la arquitectura [51] sólo utiliza dos elementos rotadores como máximo, conectado directamente la sección del cómputo del ángulo con la rotación. Para matrices pequeñas puede ir bien y ocupa muy poca área de la FPGA, pero para matrices de mayor envergadura no es la opción más correcta. En una matriz de tamaño medio, la sección del cómputo de ángulo se queda ociosa bastante tiempo y todo el circuito realmente se ralentiza en comparación con nuestra propuesta. Además en [51] no se ha tenido en cuenta los beneficios de reutilizar datos evitando nuevas lecturas a través de las FIFOs intermedias, tampoco se ha conseguido el nivel de paralelismo computacional de nuestra arquitectura.

Para el caso de [52], nuestro artículo que ha sido el hilo conductor de esta parte, guarda similitudes estructurales ya que a partir de él se ha conseguido un mejor desarrollo, sin embargo, al ser una versión previa presenta deficiencias con respecto eficiencia y eficacia.

4 Implementación sistólica para la descomposición QR sobre FPGA

4.1 Antecedentes y trabajos previos

El mundo del procesamiento de la señal está en constante avance, siempre presentando nuevos retos y desafíos donde la fiabilidad, velocidad y calidad son esenciales. La inmensa mayoría de algoritmos avanzados se fundamentan en operaciones algebraicas para llevar a cabo su propósito, de los que cabe destacar aquellos que utilizan operaciones matriciales. Múltiples ejemplos ya fueron citados anteriormente en secciones anteriores. Entre todos ellos, los que guardan una mayor relación con esta sección son aquellos que están vinculados con las comunicaciones inalámbricas, ejemplos como el formato de transmisión y recepción de antenas en dispositivos inalámbricos denominado MIMO, filtros de señales, filtros adaptativos, etc. [40] [42] [41].

Muchos de los trabajos utilizan la descomposición QR y requieren de una elevada cantidad de factorizaciones y cómputos, por ello, no es de extrañar que utilicen tecnologías que aporten un alto rendimiento operacional, como son implementaciones hardware basadas en ASIC o FPGA.

La idea de tener una gran variedad de operadores matriciales optimizados y disponibles a nivel hardware para ser utilizados por cualquier algoritmo es necesaria para el avance en las comunicaciones, de hecho, pensamos que es la base para ir creciendo y avanzando. Nos gusta el concepto que se manifiesta en el artículo [40] en el que se proponen conseguir arquitecturas hardware configurables y escalables , con vistas a que sea posible hacerse uso de ellas como si fueran módulos o librerías, facilitando la construcción de posteriores diseños.

En la literatura para el diseño e implementación hardware de la descomposición QR encontramos dos tipos de enfoques principales, iterativas o sistólicas. La primera de ellas fue bien descrita en la sección 3.2 (Diseño iterativo basado en FPGA para la Descomposición QR), mientras la segunda será descrita en la siguiente sección. En la propuesta de esta sección la arquitectura sistólica ha sido la escogida para el desarrollo de la descomposición QR en FPGAs con el objetivo de conseguir un elevado rendimiento para matrices pequeñas.

Nuevamente, dentro de las múltiples formulaciones para llevar a cabo la descomposición QR nos hemos decantado por el método de las rotaciones de Givens, puntualizado en las secciones 2.2 y 2.3. Los motivos de tal elección se deben a la robustez y precisión numérica junto con su facilidad de paralelización e implementación desde un punto de vista hardware.

Existen varios trabajos que comparten similitudes con nuestra propuesta sistólica para la descomposición QR de matrices pequeñas citados a continuación: [54], [55], [41], [56] y [57]. Todos ellos utilizan arrays sistólicos bidimensionales donde sus elementos de procesamiento trabajan en paralelo y se encuentran interconectados locamente entre sí. El nivel de explotación del paralelismo de estas arquitecturas hace que el rendimiento de todas ellas sea muy elevado y la latencia realmente baja, sin embargo, el coste de tales implementaciones presenta un el elevado consumo de los recursos disponibles, mermando de este modo los beneficios de tales diseños.

4.1.1 Arquitecturas sistólicas y la descomposición QR

La ejecución paralela de algoritmos es un área que afecta a una gran variedad de campos de la investigación y desarrollo, dentro de las técnicas más comunes destacar los sistemas multiprocesador, segmentación de instrucciones y las matrices sistólicas.

De un modo general, se define como matrices sistólicas o arrays sistólicos aquellas estructuras formadas por Elementos de Proceso (PE) con una disposición celular y con un flujo de datos encauzado. La transmisión de los datos se realiza de forma síncrona entre los distintos PEs. La característica de sistólica se debe a la semejanza del flujo de información entre los PEs con el flujo sanguíneo, desde un punto de vista arquitectural, cada pulso de reloj se iguala a un latido o bombeo del corazón originando que la información fluya de celda en celda.

Este formato de procesamiento fue acuñado por Richard P. Brent and H. T. Kung, quienes lo implementaron para el cómputo del máximo común divisor de grandes números enteros y polinomios [58]. Sus aplicaciones más comunes son aquellas en las que se realiza una masiva integración operacional en paralelo, convoluciones, correlaciones, multiplicación de matrices o clasificación de tareas, entre muchas.

Las principales ventajas de las arquitecturas arquitecturas sistólicas son las siguientes:

- Los datos de los operandos y resultados parciales se almacenan dentro del conjunto ordenado de procesadores.
- No es necesario el acceso a buses externos, memoria principal o cachés internas.
- La dependencia de los datos se encuentra implícitamente relacionaos por el nodo de interconexión.
- No hay pasos secuenciales en la gestión del flujo de datos que en respuesta son altamente paralelo.

En concordancia con la arquitectura sistólica, encontramos varios trabajos [54] y [57], en los que se implementa el algoritmo QR a través del método de rotaciones de Givens. En ambos trabajos se procesan los datos siguiendo la estructura de un array sistólico de dos dimensiones (2D) similar al que se muestra en la Fig. 4.1.

Observando detalladamente la Fig. 4.1 se aprecia el elevado nivel de paralelismo, ya que la entrada de la arquitectura es el doble del tamaño de la matriz objetivo, diferenciándose dos sec-

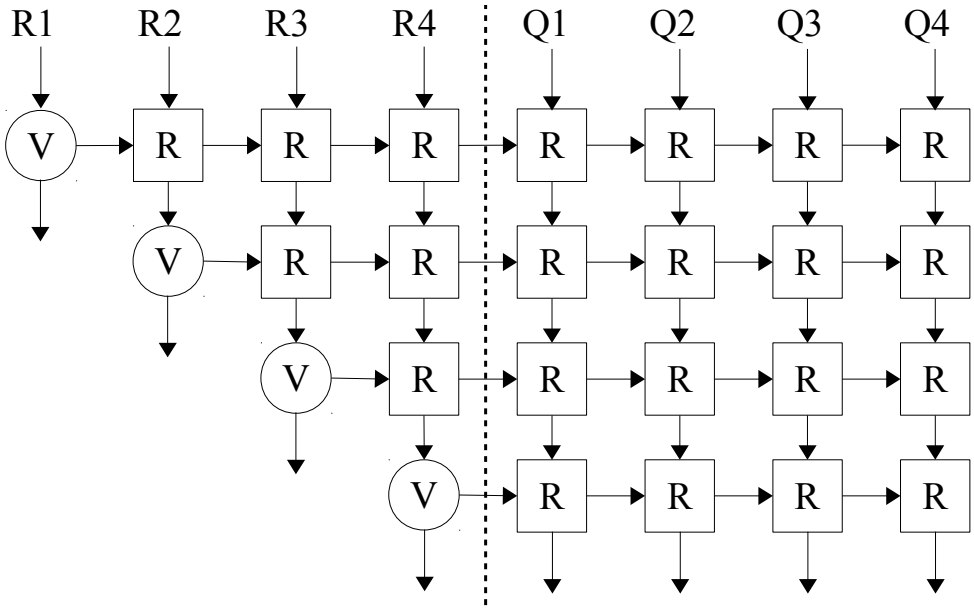


Figura 4.1: Matriz 2D-sistólica basada en columnas para matrices 4×4.

ciones separadas por una línea discontinua. Mostrando un evidente procesamiento de los PE's por columnas en [54] y [57].

En la sección izquierda de la Fig. 4.1, la entrada está formada por una fila completa de la matriz objetivo para obtener la matriz resultante R. Mientras que en la sección derecha tiene como entrada una matriz identidad de igual tamaño para generar la matriz Q.

Destacar el elevado número de unidades PE que se contabiliza, donde evidentemente, la primera fila siempre será el doble del tamaño de la matriz ya la primera fila está compuesta por la matriz entrante junto con la matriz identidad. Las PEs que encontramos en los sucesivos niveles irá decrementándose en una unidad en la sección izquierda. Comportamiento que no sucede para el procesamiento para conseguir la matriz Q, ya que en todos los niveles siempre se requiere de la misma cantidad de PEs.

Las rotaciones de Givens se pueden paralelizar tantas veces como elementos distintos de cero están dentro de la fila de la matriz. Siguiendo con la descripción de la arquitectura de la Fig. 4.1, se distinguen dos tipos de unidades de PEs: vectorizadoras y rotadoras.

Por un lado, las unidades encargadas de vectorizar tiene como objetivo rotar un par de elementos para hallar el ángulo de rotación, identificadas con una V. Mientras que las marcadas con una R son las unidades PEs que efectúan la rotación de un par de elementos a partir del ángulo enviado por la unidad rotadora R. Cada fila de la arquitectura sólo necesita una unidad de vectorización V, mientras que el resto elementos requerirá de una unidad de procesamiento para la rotación. En la Fig. 4.1 se muestra lo descrito hasta el momento, el número de PEs va reduciéndose hasta formar una diagonal superior de una matriz donde la diagonal está cubierta por unidades de vectorización.

La diferencia entre [54] y [57] reside en el modo que han sido desarrolladas las PE's. Para el caso de la arquitectura [54] se utilizan operaciones aritméticas estándar para llevar a cabo las rotaciones. Mientras que en la [57] fue implementada mediante elementos CORDIC iterativos (Véase 2.4.3) presentado una reducción de área considerable.

A pesar de las ventajas comentadas anteriormente para [54] y [57], ambas arquitecturas sistólicas presentan una importante inactividad por parte de los PEs de las últimas filas provocada por los intervalos de tiempo entre rotaciones consecutivas. Todo ello se traduce como un significativo derroche de recursos que se encuentra parados y en espera de datos. Además de esto, la misma dependencia de los datos evita el uso de segmentación interna limitando el rendimiento alcanzable.

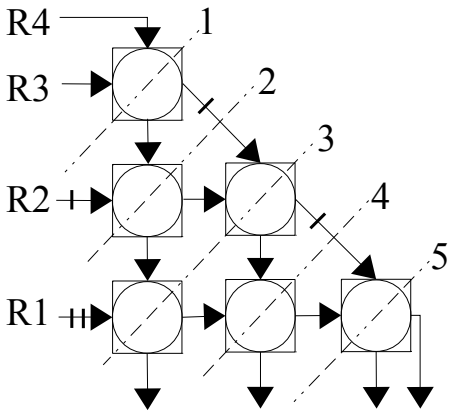


Figura 4.2: Matriz 2D-sistólica basada en filas para matrices 4×4.

Otro enfoque distinto es el que describen los artículos [55] y [59] en los que la Fig. 4.2 representa la arquitectura utilizada, siendo en este caso aplicable a la matriz por filas y no por columnas como se ha detallado anteriormente. En este esquema una PE realiza una rotación de Givens completa para todos los elementos. Por lo tanto, las dos operaciones básicas para realizar una rotación de Givens han de estar implementadas dentro de la PE. Cierto es que se utiliza una lógica más compleja, pero la ganancia aparece con la disminución de unidades elementales de procesamiento. La principal ventaja de un enfoque por filas es que la existente dependencia entre los datos sólo es la producida por la rotación de Givens, es decir, aquella que existe entre el cálculo del ángulo de rotación y su propia rotación. De esta forma es más fácil aplicar técnicas de pipeline o segmentación, algo que no es tan directo en las arquitecturas sistólicas por columnas.

Además en el artículo [59] se proponen intercalar columnas de diferentes matrices de entrada para superar esta dependencia, pero esto no es práctico para muchas aplicaciones, especialmente para las de una larga segmentación.

Por último, las operaciones matemáticas para realizar la factorización QR, raíces cuadradas y divisiones son utilizadas en [55], donde la dependencia entre ángulos y rotaciones se elimina mediante un pre- y post-procesamiento que permite pipelining de los PE a través de un módulo denominado Square Root and Division Free Givens Rotations (SDFG) [60]. Gracias a ello, esta arquitectura logra un mayor rendimiento, pero la complejidad de las operaciones involucradas también requiere un elevado uso de recursos disponibles.

4.2 Arquitectura propuesta para la factorización QR a través de rotaciones de Givens paralelizado

El diseño propuesto se fundamenta en un diseño sistólico bidimensionales donde cada nodo de procesamiento PE trabaja con todos los elementos de la misma fila, además tiene como objetivo crear un mayor caudal de datos utilizando unidades de procesamiento segmentadas. Adicionalmente para optimizar las rotaciones de Givens, y a su vez huir de ciertas funciones complejas a nivel hardware como la raíz cuadrada, senos y cosenos, se ha hecho uso del conocido algoritmo CORDIC. En este caso se ha optado por una versión segmentada para ganar en sencillez, rendimiento, y en área, ésta última mucho más reducida que en otras arquitecturas.

De forma aclaratoria, indicar que el diseño guardar alguna similitud estructural con el diseño [55], pero internamente presenta relevantes diferencias que ser verán a continuación. Todo este desarrollo concluyó en la publicación del artículo [61] en el año 2015.

4.2.1 Descripción y planificación del algoritmo Givens paralelizado

A estas alturas ya conocemos el algoritmo clásico de Givens para la factorización de matrices, bien descrito en la sección 2.3.2. En la Fig. 4.3 se muestra el esquema de cómo avanza el algoritmo de factorización QR a través del método de Givens.

La entrada es una matriz 4×4, que una vez finalizado el algoritmo mantiene el tamaño, pero adquiere la característica de ser una matriz triangular superior en tan sólo 6 etapas. Cada rotación de Givens es ilustrada mediante una flecha acompañada con la siguiente nomenclatura $G_k(i,j)$, ésta especifica las filas involucradas en la rotación (i,j), siendo k la columna en la se va a insertar el nuevo valor 0.

A modo de ayuda visual se ha trazado sobre la matriz un perímetro circular sobre aquellos elementos de la fila con los que el ángulo de rotación es calculado, mientras que con un perímetro rectangular se remarcan los elementos que rotan con dicho ángulo.

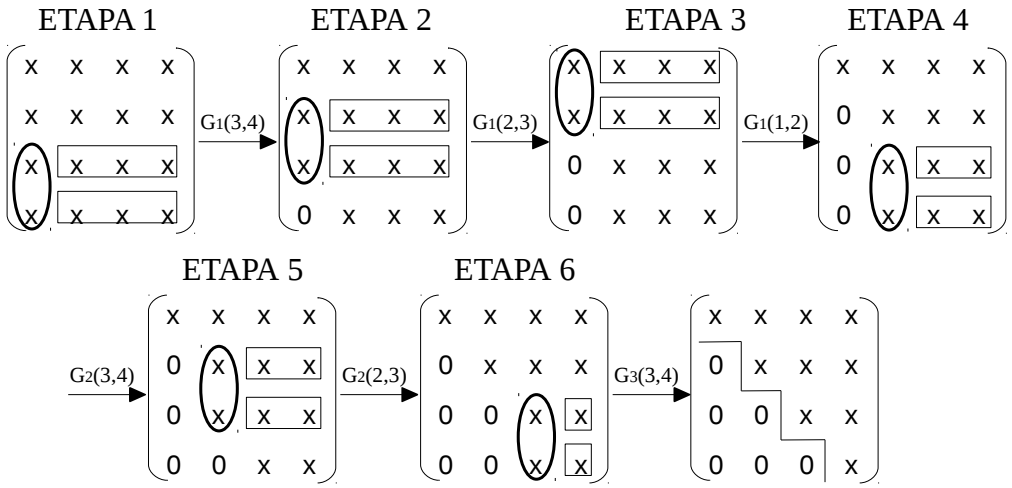


Figura 4.3: Planificación de la aplicación de las rotaciones de Givens sobre una matriz 4×4.

El recorrido que se representa la Fig. 4.3 se corresponde con el algoritmo de Givens clásico,

donde se empieza con la inserción del primer cero en la fila inferior izquierda, avanzando por la filas superiores hasta alcanzar en elemento diagonal. Una vez procesado el elemento diagonal, se produce un desplazamiento a nivel de columna hacia la derecha y se comienza con un nuevo procesamiento desde la fila inferior hasta volver a un nuevo elemento diagonal.

El recorrido del algoritmo clásico es el primer aspecto que dista de nuestra propuesta. Las 6 etapas con la que se lograba la factorización QR de una matriz 4×4 es reducible a tan sólo 4. Para ello, bastará con ejecutar en paralelo más rotaciones de Givens al mismo tiempo.

En la Fig. 4.4 se ilustra cómo es este nuevo recorrido acelerado para una misma matriz 4×4 en menos etapas, llevándose a cabo dos rotaciones de Givens en paralelo, algo que no se hacía en el algoritmo estándar.

Cierto es que los arrays bidimensionales sistólicos mejoran el paralelismo de la factorización clásica. Concretamente en la Fig. 4.2, todas las PEs de la misma diagonal computan elementos en paralelo, reduciendo significativamente el número de etapas para un encadenamiento de matrices. Pero el esquema de la Fig. 4.4 permite un mayor nivel de paralelismo donde más filas son rotadas de forma simultánea.

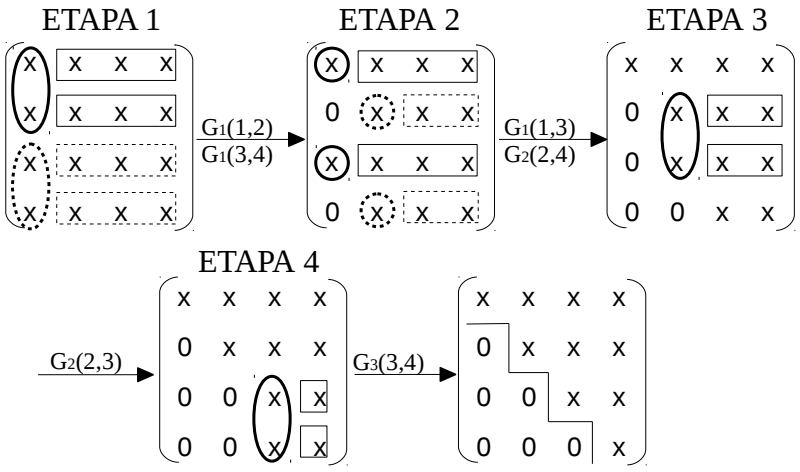


Figura 4.4: Planificación acelerada de la aplicación de las rotaciones de Givens sobre una matriz 4×4.

Observando con detalle la Fig. 4.4, se aprecia cómo en la primera etapa, a diferencia del método clásico, ahora han aparecido dos áreas circulares, una línea continua y otra punteada. Con ello indicamos la doble computación de los dos ángulos de rotación que se está realizando, ambos utilizados simultáneamente por las áreas rectangulares, que nuevamente son dos. Las dos rotaciones están indicadas como $G_1(1,2)$ y $G_1(3,4)$.

El algoritmo es acelerado y paralelizado nuevamente en la siguiente etapa, ya que de nuevo se producen dos operaciones simultáneas. Éstas se corresponden con $G_1(1,3)$ y $G_2(2,4)$, su peculiaridad reside en la inserción de dos ceros en distintas columnas al mismo tiempo. El algoritmo sigue con un procesamiento normal hasta conseguir una matriz triangular superior, por lo tanto, llevándose a cabo $G_2(2,3)$, seguido de la rotación $G_3(3,4)$, igual que en el esquema anterior.

Éste nuevo recorrido se fundamenta en el uso de aquellas filas que se encuentran ociosas, siempre y cuando no afecte al cómputo de las demás rotaciones de Givens.

Nuevamente, el incremento del paralelismo en algoritmo de Givens sigue un recorrido algo diferente del anterior. Aunque en este caso los efectos son claramente notorios, ya que al reducir el número de etapas la latencia también ser ve reducida. Otro aspecto positivo, es que es válido para cualquier tamaño de matriz, siendo las matrices de tamaño potencia de 2 aquellas que mejor aprovecharan los recursos.

4.2.2 Procesamiento basado en el algoritmo CORDIC optimizado

En el capítulo 2.4 se ha descrito el funcionamiento del algoritmo CORDIC (COordinate Rotation DIgital Computer). Además, se ha presentado una arquitectura general para mostrar cómo a partir de sumas y desplazamientos realiza las tareas de vectorización y rotación de vectores. También, en el capítulo 2.4 se presentan las dos posibles arquitecturas con las que el algoritmo CORDIC cuenta, iterativa y pipeline, junto con sus ventajas e inconvenientes.

Existe una gran variedad de propuestas en las que CORDIC es utilizado para implementar la función de rotación Givens. Para nuestra propuesta se ha creado una nueva versión de algoritmo CORDIC que coincide en varios aspectos con el Bi-z CORDIC [62]. En este caso el componente CORDIC se ha utilizado para implementar operaciones para la inversión de una matriz siguiendo una arquitectura sistólica bidimensional.

Por lo general, en otras arquitecturas hemos visto el uso de dos tipos de componentes CORDIC, vectorizador o rotador (véase 2.4.2). No obstante, en nuestra implementación es algo distinto, el CORDIC construido es capaz de realizar la vectorización y rotación en un mismo componente implementado como una arquitectura segmentada.

En el diseño [62] la dependencia de los datos del método Givens para invertir una matriz requiere de una matriz entrelazada para que el diseño segmentado sea ventajoso. Sin embargo, en nuestro diseño la segmentación no requiere de elementos intermedios, ya que la computación va en concordancia con el modelo segmentado siguiendo un flujo natural.

Observando el caso con detalle, el enfoque de este nuevo CORDIC se fundamenta en sustituir el cálculo del ángulo de rotación θ y desglosarlo en la dirección de cada una de las micro-rotaciones. La dirección de la rotación está indicada por el signo de la coordenada Y del vector en cada una de las etapas. Dicha dirección se almacena temporalmente en un registro σ que indicará a los siguiente vectores la dirección de la rotación. Con ello, se consigue que desde el primer ciclo de reloj ya se tenga información suficiente para comenzar con la primera rotación. Por lo tanto, su consigue una solapamiento del cálculo del ángulo con la rotación de las filas entrantes en la rotación.

En la Fig. 4.5 se muestra la arquitectura de la que hablamos en el párrafo anterior, distinguiendo entre dos secciones: izquierda y derecha. En la sección derecha se muestran los componentes y rutas de datos X e Y típica iteración de una etapa CORDIC. En la sección izquierda se representa el hardware que trabaja en modo vectorización, seleccionando en cada etapa la dirección de la rotación y cuándo actualizar los registros σ. Los registros σ contienen la dirección del ángulo de rotación almacenado, éstos están conectados con los sumadores/restadores (add/sub), elementos que se encargaran de realizar la rotación del vector entrante a partir del ciclo posterior.

La señal nda es utilizada para indicar que el dato que está entrando es un nuevo ángulo, por lo que en esa etapa del pipeline debe de calcularse el cómputo del nuevo ángulo de rotación, es decir, se corresponde con las operaciones de vectorización del CORDIC. A medida que la señal nda avanza en el recorrido de la Fig. 4.5, nuevamente se actualizan los registros θ con la dirección de rotación.

Por lo tanto, se ha conseguido implementar una arquitectura donde conviven la vectorización y rotación al mismo tiempo. Además, dependiendo del tamaño de la segmentación pueden estar computándose varias rotaciones de Givens independientes al mismo tiempo, solapando el final de una con el inicio de la otra.

En la Fig. 4.5 no se muestran la cantidad de iteraciones o etapas que completan la rotación, en la imagen ha sido definida como n fases. La cantidad de micro-rotaciones o iteraciones puede depender de dos factores, la longitud de palabra o representación utilizada para los elementos de vector y la precisión que desee obtener. Todo ello, se aclarará en el apartado de Análisis de rendimiento de la propuesta y comparación con implementaciones similares (subsección 4.2.4).

Por último, añadir que cada componente CORDIC ha de realizar una compensación del factor de escala [63]. Para ello se realiza una multiplicación los valores resultantes por la inversa del factor de escala. Adelantando cuestiones resueltas en la siguiente sección, dichos multiplicadores podrán implementarse como constantes de multiplicación basadas en registros o utilizando los multiplicadores embebidos.

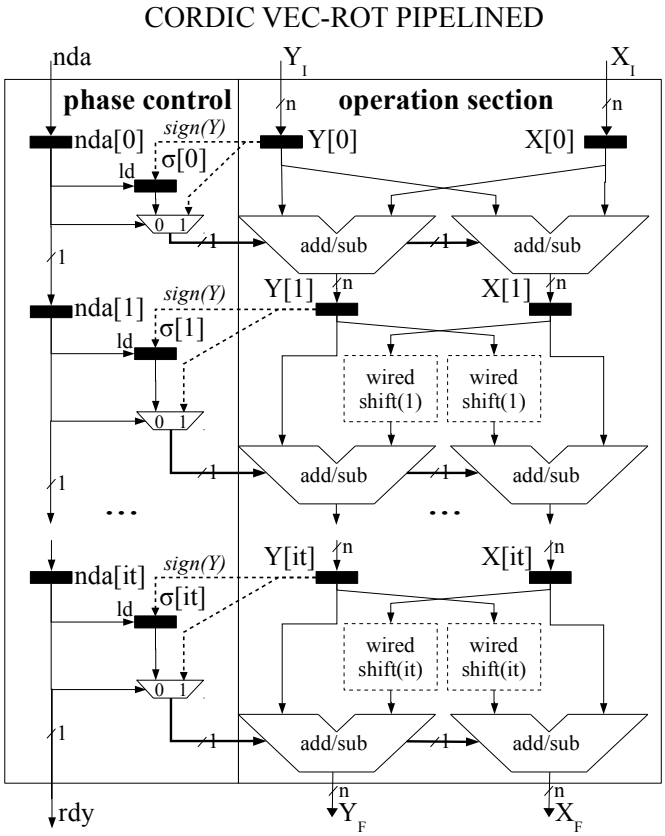


Figura 4.5: Implementación del algoritmo CORDIC vectorización-rotación.

4.2.3 Descripción de la arquitectura e implementación propuesta

A través de la unión de las dos secciones anteriores 4.2.1 y 4.2.2 se deriva, casi de forma directa, en la contrucción de la estructura y localización de las unidades de procesamiento PEs encargadas de realizar cada una de las rotaciones.

La Fig. 4.6 ilustra la asociación de las anteriores conjeturas, presentando en tan solo 4 eta-

pas bien definidas en el algoritmo QR Givens acelerado de la subsección 4.2.1. Tal y como se describe en la planificación del algoritmo, la primera y segunda etapa cuentan con 2 unidades de procesamiento PE, debido a que cada una de ellas realiza dos rotaciones de Givens en paralelo.

También se aprecia en la Fig. 4.6 cómo es la conexión de los buses de entrada y salida entre PEs de las dos etapas iniciales, totalmente directa y enlazando la salida de la primera etapa con la segunda. Recalcar que el CORDIC vectorización-rotación de la etapa 2 denominado *CORDIC Vec-Rot 2,2*, está rotando los elementos de la segunda columna de la matriz, mientras el inmediato superior 1,2 ya devuelve la fila 1 factorizada.

En la tercera etapa es necesario la utilización de un nuevo elemento en la arquitectura propuesta, una lista o cola de registros que sigue una filosofía FIFO (First Input First Output). Esta lista de registros es necesaria para almacenar de forma temporal los elementos de la fila inferior de la matriz, la fila 4. Estos elementos en esa etapa no pueden utilizarse por ningún elemento factorizador pero son necesarios para la etapa 4. Además, al finalizar la etapa 3 obtenemos la fila 2 como elemento resultante factorizado. Indicar que la longitud de tal FIFO coincide con el número de etapas del componente CORDIC que se encuentra en paralelo.

Para acabar con el algoritmo, en la etapa 4 se realiza la última rotación de Givens. En ella se ven involucradas las filas 3 y 4, ya con sus elementos totalmente rotados. Como puede apreciarse la sincronización de la arquitectura es directa y sencilla gracias a la segmentación, sin requerir de ninguna máquina de estados para el control.

Las señales nda de las PE, aquellas que indican la llegada de elementos para vectorizar, se fijan externamente a medida que llegan los datos justo al inicio de la arquitectura, aunque también es posible utilizar un contador que se ajuste al tamaño de la matriz para activar dicha señal. En las siguientes etapas, la señal nda es tratada como una entrada más, aunque es cierto que en el caso donde se computan 2 columnas distintas a la vez o hay un cambio de columna, esta se ve retrasada un ciclo para poder compensar el elemento 0 que no entra en el cauce.

Finalmente, gracias al tipo de arquitectura completamente pipeline es posible comenzar un nuevo cálculo de matriz justo después de que otro haya sido introducido. Ello permite que el rendimiento sea muy elevado, ya que si enlazamos varias matrices de un tamaño 4×4, cada 8 ciclos obtenemos una matriz QR factorizada por el método Givens.

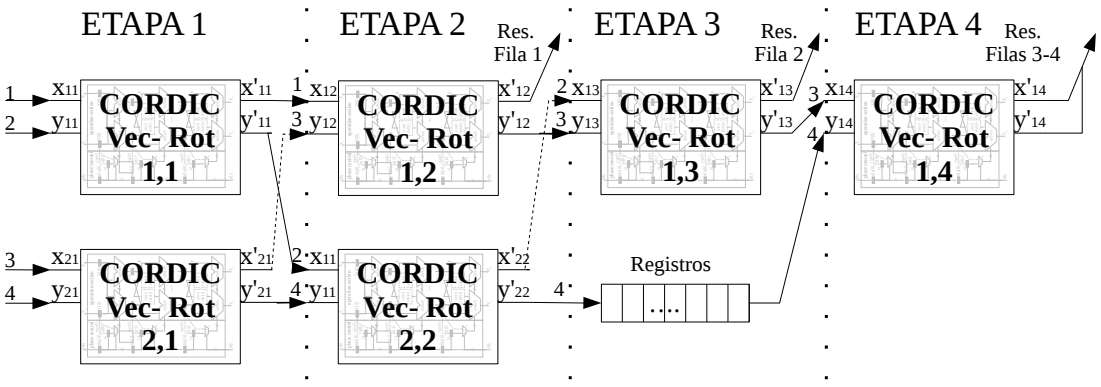


Figura 4.6: Arquitectura basada en CORDIC acelerado para factorizar por QR una matriz 4×4.

4.2.4 Análisis de rendimiento de la propuesta y comparación con implementaciones similares

Una vez descrito la arquitectura y sus detalles en la implementación queremos dar paso al análisis y resultados que ésta presenta. Para ello, se ha diseñado en VHDL un núcleo de descomposición QR con representación en punto fijo para matrices de 4×4. El diseño muestra varios parámetros configurables: el ancho de palabra, es decir la longitud de bits para la representación en punto fijo, y la cantidad de iteraciones o fases donde se produce una micro-rotación en los elementos CORDIC.

Para el diseño y simulación de la arquitectura propuesta se ha hecho uso del software ISE 14.3. La prueba final ha sido verificada a través de una plataforma hardware FPGA Virtex-6 XV6VLX240T velocidad -2. La entrada del sistema ha sido una amplia gama de matrices aleatorias con distintas configuraciones. Para la verificación del correcto funcionamiento y análisis posterior de los resultados se ha empleado uno de los programas por excelencia para cálculo matricial MATLAB R2014b, haciendo especial mención a su gran precisión numérica al trabajar con representaciones flotantes de doble precisión.

Antes de comenzar con los distintos resultados de la arquitectura propuesta, se ha llevado a cabo un estudio previo sobre influencia del número de micro-rotaciones que debe de realizar un algoritmo CORDIC dentro de la arquitectura propuesta. El objetivo perseguido es conseguir la óptima configuración de los componentes CORDIC con respecto al área, latencia y error, aspectos de mayor relevancia en el circuito. Por ello, se han implementado distintas configuraciones para los elementos rotadores basándose en 3 tipos de longitud numérica de palabra: 16 bits, 24 bits y 32 bits. Además, cada longitud de palabra ha sido analizada con una amplia cantidad de iteraciones.

El resultado final de dicho estudio previo se ha presentado en la Tabla 4.1 en el que se muestra el error máximo obtenido para cada una de las configuraciones. Este error máximo calculado ha sido obtenido a partir de la factorización QR de varias baterías de pruebas con 50.000 matrices aleatorias. Una vez descompuestas cada una de las matrices se ha realizado la verificación multiplicando $Q^t * R$, siendo éste la matriz original A. Es decir, se ha calculado el error máximo absoluto entre la matriz de entrada y la factorización como se indica la siguiente fórmula $max(|A - Q^t * R|)$.

Examinando la Tabla 4.1 se presentan todas las configuraciones analizadas, en ellas se aprecia el siguiente comportamiento común para todos los tamaños de palabra: a medida que la cantidad de rotaciones aumenta, el error máximo disminuye hasta alcanzar un mínimo absoluto; una vez sobrepasado ese mínimo, el error vuelve a aumentar de nuevo. Este efecto se produce por el exceso de iteraciones en el algoritmo CORDIC y la acumulación de error de redondeo en las fases finales [64] [63]. Consecuentemente, la Tabla 4.1 nos indica las configuraciones que minimizan el error en nuestro circuito, siendo éstas de un tamaño de 16, 24 y 32 bits de tamaño de palabra y 10, 18, 26 iteraciones CORDIC, respectivamente.

En la Tabla 4.2 se muestran los resultados de implementación para las longitudes de palabra del estudio previo junto con la cantidad de iteraciones óptima calculada. En este caso se han barajado 3 escenarios posibles para realizar la corrección de escala de los elementos CORDIC. Como se dijo en la sección 4.2.2, la salida de los CORDIC ha de ser multiplicada por una constate de corrección denominada $K = lim_{n\to\infty} K(n) \approx 0.607252935008881256169 4$ y que se calcula a partir de la siguiente forma. Siendo $K(n) = \prod_{i=0}^{n-1} 1/\sqrt{1 + 2^{-2i}}$ [65].

Entonces, en la Tabla 4.2 se presentan la alternativa de implementar la arquitectura utilizando multiplicadores dedicados DSP48E que se encuentran embebidos en la FPGA implementado ca-

Tamaño Palabra	**16 bits**				
CORDIC-Iteraciones.	8	10	12	14	16
Latencia (ciclos)	44	52	60	68	76
Máx. Error	1.4e-3	5.8e-4	6.9e-4	7.3e-4	8.9e-4
Tamaño Palabra	**24 bits**				
CORDIC-Iter.	12	16	18	20	24
Latencia (ciclos)	60	76	84	92	108
Máx. Error	8.06e-5	6.01e-6	3.5e-6	3.7e-6	4.7e-6
Tamaño Palabra	**32 bits**				
CORDIC-Iter.	20	24	26	28	32
Latencia (ciclos)	92	108	116	124	140
Máx. Error	3.3e-7	2.4e-8	9.4e-9	1.6e-8	2.3e-8

Tabla 4.1: Error máximo para la factorización QR para 16, 24 y 32 bit de longitud de palabra en función del número de iteraciones en los elementos CORDIC

paces de trabajar en modo no-pipeline o pipeline, Multiplicador A y Multiplicador B. Además, también se ha barajado un tercer enfoque (Multiplicador C) diseñado a través del Xilinx Core Generator que de forma automática realiza una multiplicación por un coeficiente constate y segmentado. Aclarar que este tercer diseño C no utiliza ningún tipo de DSP48E.

Los resultados son muy variados según el tipo de multiplicador adquirido, por ejemplo, para un tamaño de palabra de 16 bits con 10 iteraciones CORDIC, el multiplicador A es el que presenta una menor latencia pero, su frecuencia máxima está muy por debajo de los otros dos, eso sí el área utilizada es prácticamente la misma que el multiplicador B, rondando los 2.500 slices que aproximadamente forman el 1% del área de la FPGA.

Observando en la Tabla 4.2 la configuración de 24 bits con 18 iteraciones CORDIC, el multiplicador A presenta un desfavorable valor de frecuencia máxima, aunque su latencia es mejor no aporta una gran mejora si hay una densa cantidad de matrices entrantes. Nuevamente el multiplicador A es quién consume una menor cantidad de slices Registers y LUTs aunque ocupará un 3% de DSP48E. En el caso del multiplicador B, guarda parámetros bastante similares en cuanto a consumo de área con el multiplicador A, pero su principal diferencia con éste es la severa mejora de la frecuencia máxima. En cuanto al tercer multiplicador, el C, el consumo del área se incrementa en un punto porcentualmente y nuevamente la frecuencia es más del doble que el multiplicador A.

Para acabar con la Tabla 4.2, en la configuración de 32 bits y 26 iteraciones CORDIC sucede algo similar a lo que venimos viendo hasta ahora. El multiplicador A tiene un inferior consumo de área, pero muestra peores resultados en cuanto a rendimiento, siendo la frecuencia máxima más de tres veces peor que en los otros multiplicadores. En esta tercera comparación, el multiplicador C sobrepasa el 50% en cuanto a consumo de slices, eso sí, no utiliza ni un DSP48E.

Resulta interesante los datos de la Tabla 4.2, ya que demuestra que si se quiere reducir en área consumida es importante utilizar el tipo de multiplicadores embebidos. Aunque si nuestra aplicación requiere de multiplicadores para otras operaciones y se quiere ahorrar su uso, conviene utilizar un multiplicador por coeficiente.

Para finalizar con esta subsección, la Tabla. 4.3 se expone a una comparación que justifica

Dispositivo	**Xilinx Virtex 6 XC6VLX240T -2**		
Tamaño Palabra	**16 bits**	**24 bits**	**32 bits**
CORDIC-Iteraciones	10	18	26
Ciclos/Matriz	8	8	8
Multiplicador A	**Dedicated Non-Pipelined DSP48E**		
Latencia (ciclos)	52	84	116
DSP48E	12(1%)	24(3%)	48(6%)
Max. Freq. (Mhz)	234.5	170.3	117.1
Slice Registers	2,402(1%)	5,888(1%)	10,844(3%)
Slice LUTs	2,498(1%)	6,109(4%)	11,337(7%)
Multiplicador B	**Dedicated Pipelined DSP48E**		
Latencia (ciclos)	60	100	140
DSP48E	12(1%)	24(3%)	48(6%)
Stages Pipe. Mult.	2	4	6
Máx. Frec. (Mhz)	421.1	398	377.6
Slice Registers	2,457(1%)	6,046(1%)	11,520(3%)
Slice LUTs	2,587(1%)	6,220(4%)	11,225(7%)
Multiplicador C	**Const. Coef. Pipelined (without DSP48E)**		
Latency (ciclos)	60	96	132
Stages Pipe. Mult.	2	3	4
Máx. Frec. (Mhz)	421.1	398	377.6
Slice Registers	3,262(1%)	8,400(2%)	16,259(5%)
Slice LUTs	3,641(2%)	8,596(5%)	16,100(10%)

Tabla 4.2: Resultados de una implementación sobre FPGA para 16, 24 y 32 bits de longitud de palabra.

la efectividad de nuestra propuesta en comparación con otras opciones similares que proporcionan datos suficientes para realizar una justificación razonable. Nuestra opción escogida para la demostración ha sido la configuración con el multiplicador A, por ser la que más se aproxima a los trabajos realizados. Además para que la comparación sea justa hemos sintetizado nuestra arquitectura en FPGAs equivalentes a aquellas de trabajos anteriores, concretamente para Virtex4 (XC4VFX60-11) y Virtex5 (XC5VTX150T-2).

En cuanto al desempeño, el único diseño que muestra un rendimiento relativamente cercano al nuestro es [55]. En este caso la propuesta es algo similar a la nuestra, utilizando unidades de procesamiento PEs segmentadas y guarda una latencia muy cercana a nuestros valores. Sin embargo, su reducida frecuencia máxima supone un lastre, ya que está un 35% por debajo de la nuestra, además de empeorar en un 50% en el rendimiento en comparación con la arquitectura propuesta. Si nos fijamos ahora en los trabajos [54] y [57], el rendimiento es de un orden de

Artículos	[54]	[55]	[57]	Este Trabajo (Mult. A)	
Dispositivo	**Virtex5**	**Virtex4**	**Virtex5**	**Virtex4**	**Virtex5**
W-Length	**16 bits**	**16 bits**	**18 bits**	**16 bits**	
Latencia(cl)	180	51	80	52	
Latencia(μs)	0.73	0.35	0.72	0.23	0.20
Máx. Frec. (Mhz.)	246	144	111	222	254
Throughput MMatrices/sec	1.36	18.05	2.13	27.70	31.70
Slice Reg.	16,929	5,891	7,811	2,311	2,085
Slice LUTs	10,899	9,810	2,609	2,085	2,671
DSP48	28	41	—	12	
Máx. Error	4.2e-3*	—	6.32e-3	5.8e-4	

*Nota. El error máximo para [54] se obtiene de una muestra matricial factorizada dada como ejemplo, no se ha realizado un estudio de error completo.

Tabla 4.3: Comparación de la arquitectura propuesta con otras implementaciones sobre FPGA

magnitud inferior que el nuestro, ello es debido a que sus unidades de procesamiento son iterativas en vez de segmentadas. En nuestro caso, el camino crítico disminuye debido a la simplicidad de la arquitectura CORDIC.

Si nos centramos en los valores de área de la Tabla. 4.3, nuestro diseño está por encima de todos. La cantidad de recursos es varias veces inferior a cualquiera de las demás. Siendo el peor de todos [54] que ocupa hasta entre 5 y 8 veces más. El más cercano de todos es [57] y también utiliza una arquitectura basada en CORDIC, aún así, requiere el mismo número de Slices LUTS sin usar multiplicadores pero necesita hasta casi 4 veces más Slice Registers que nuestro diseño propuesto. La gran diferencia en el consumo de LUTs se explica por sí sola, ya que el número de unidades de procesamiento PEs presentados es mucho mayor y al uso de la aritmética carry-saved.

Teniendo en cuenta todos estos resultados, se concluye que la arquitectura propuesta, a pesar de ser ampliamente variable en cuanto a configuraciones, presenta un rendimiento mucho mejor y una utilización de recursos óptima en comparación con otros trabajos. También se ha de considerar que se podría duplicar el rendimiento mediante el uso de DSP48E a costa de un ligero aumento en la latencia. Este caso no ha sido comparado, ya que no ha habido otros trabajos que utilizaran tales recursos.

5 Descomposición QR sistólica sobre FPGA con formato HUB

5.1 Formato Half-Unit Biased

En este apartado queremos dar a conocer un reciente formato de representación numérica denominado "Half-Unit Biased"(HUB)[66][67] [68]. Este tipo de notación, propuesto por J. Hormigo y J. Villalba a principios del 2015, se basa en el desplazamiento de los formatos convencionales donde se elimina la lógica de redondeo en unidades aritméticas a través un bit menos significativo implícito (ILSB).

Los números HUB son representados como cualquier número convencional y cuentan con el mismo tamaño que el original, la diferencia es que son representados con un bit implícito adicional. Como consecuencia del ILSB, todas las operaciones aritméticas han de adaptarse al nuevo formato para poder trabajar con número HUB, teniendo en cuenta que el ISLB en ningún momento es almacenado o transmitido.

La principal ventaja del formato HUB reside en que la operación complemento a dos que se implementa con una simple inversión binaria, consiguiendo redondeando al valor más cercano por el hecho de truncar. Entonces, cuando un número es truncando, de forma directa el número HUB producido es siempre el más cercano al valor inicial gracias al tipo de formato.

Gracias a ello, sumadores, multiplicadores, conversores, etc. mantienen o mejoran la precisión de los formatos convencionales para una misma cantidad de bits para la representación numérica. A su vez, el formato HUB favorece la reducción de área, aumento de velocidad y reducción de consumo[66][67]. Además, en [66] se demuestra con datos reales como por ejemplo un sumador HUB es un 14% más rápido, consume un 38% menos de área y tiene un menor consumo en comparación con un sumador convencional.

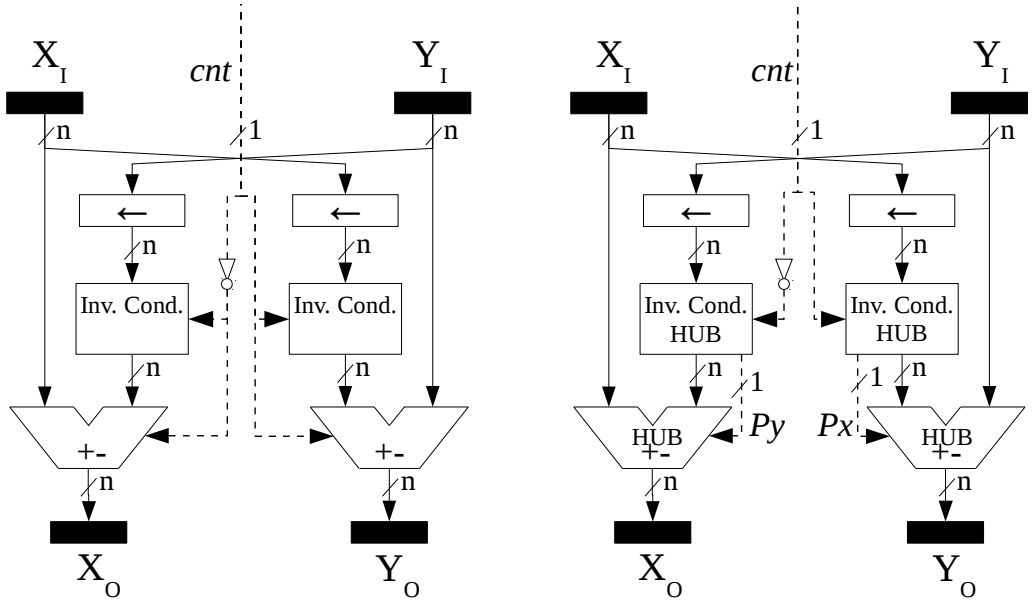
En la literatura ya existen varias aplicaciones en las que utilizan la notación HUB. Por ejemplo ha sido aplicada de forma satisfactoria en filtros FIR (Finite Impulse Response) [69], donde al reducir ancho de bits y manteniendo una misma precisión, las consecuencias presentan múltiples beneficios, menos ocupación de recursos y dilatación de los caminos críticos. Anotar que el formato HUB también ha sido aplicado para formatos punto flotante (FP) [70], denominado HUB FP.

Demostrando nuevamente cómo un sumador básico puede optimizarse en un 70% para frecuencias elevadas, también para valores flotantes.

Debido a todas las anteriores ventajas, una implementación HUB será positiva en nuestros anteriores diseños. Por ello se plantea el cambio de formato de punto fijo a formato HUB en el diseño propuesto para la factorización de matrices QR basado en el método Givens, donde los buenos resultados del formato HUB volverán a ser evidentes [71].

5.2 Implementación HUB para la factorización QR

El formato HUB exige ciertos cambios sobre la arquitecturas propuesta para factorizar matrices QR tal y como se describe en la sección 4.2.1. El primer paso es adaptar y tener en cuenta que los números entrantes son como los normales, sólo que ahora tiene un bit adicional por la derecha oculto o implícito (ILSB) que siempre, siempre, es equivalente a uno. Con la simple operación de la negación de los bits se consigue realizar el complemento a dos, lo bueno es que al ser el bit implícito siempre a cero nunca se propaga al siguiente bit explícito. Este hecho simplifica las operaciones aritméticas repercutiendo de forma directa sobre el tiempo de procesamiento y reduciendo el retardo mínimo.



(a) Iteración CORDIC Normal

(b) Iteración CORDIC HUB

Figura 5.1: Diferencias entre arquitectura de una iteración CORDIC normal y HUB.

Por ello, la implementación del formato HUB tendrá una importante repercusión sobre los componentes CORDIC vectorización y rotación desarrollados en 4.2.2 a fin de acelerar el cómputo de QR. En la parte derecha de la Fig. 5.1(a) se muestra la implementación de una iteración CORDIC normal, mientras que la Fig. 5.1(b) se presenta la arquitectura de una iteración CORDIC adaptada al formato HUB. Evidentemente los inversores utilizados difieren entre las arquitecturas, al igual que los sumadores/restadores. Por ejemplo, para realizar la inversión en el CORDIC normal requiere llevar a cabo la función complemento a dos, mientras que en HUB es una simple negación. Para la implementación del sumador/restador se ha seguido la descripción del articulo [72].

A simple vista, en la Fig. 5.1 se muestra el principal cambio estructural que se lleva a cabo a la salida de los inversores HUB. Para la arquitectura HUB el bit de acarreo no está conectado a la señal de control de rotación cnt como ocurre en una iteración normal. En este caso, es el bit más significativo (MSB) de los bits descartados quien representa el acarreo una vez sucedido el desplazamiento, ilustrado con las señales Px y Py en la Fig. 5.1(b). Ya que este primer bit más significativo dentro de los descartados (Px y Py) es el bit implícito en el formato HUB, y ha de sumarse/restarse a la siguiente valor obtenido. Mencionar que en la primera iteración, este bit implícito es siempre uno.

5.3 Resultados de la implementación HUB

Para conocer los resultados de la implementación HUB en la descomposición QR (QRD) sobre un dispositivo FPGA, se ha implementado la arquitectura pipeline propuesta en 4.2.1 y 4.2.3 para matrices 4×4 utilizando CORDIC HUB.

N. Bits	**QRD Convencional**		**QRD HUB**	
	máx.	***media.***	***máx.***	***media***
16	692.829e-6	362.938-6	265.896e-6	153.388e-6
32	16.627e-9	7.542e-09	7.5785e-9	3.2057e-9

Tabla 5.1: Parámetros de error de redondeo para QRD

En la Tab. 5.1 se presenta los resultados del error obtenido para una muestra de 50,000 matrices aleatorias. El testeo ha sido llevado a cabo a través de MATLAB, tomando como dato correcto aquella aportada por el software con una representación numérica validada por 64 bits de precisión. El error ha sido obtenido de la verificación $Q' \times R = A$, siendo A la matriz entrante. La diferencia entre la matriz A y la matriz calculada $Q' \times R$, genera el error máximo establecido. En el caso del formato Hub se ve ampliamente reducido, coincidiendo prácticamente con el valor medio del formato convencional.

Como se muestra en Tab. 5.1, el error máximo y medio prácticamente se ve reducido a la mitad tanto para un tamaño de palabra de 16 bits y 32 bits, y todo gracias al redondeo añadido. Resulta bastante significativo, ya que gracias a la aplicación del formato HUB se podría disponer de un dígito más de precisión.

Por tanto, al verse mermado a la mitad el error medio y máximo en la implementación HUB, resulta interesante ver y comparar los datos en FPGA para una arquitectura HUB con la misma precisión que la convencional sobre un mismo dispositivo. Es decir, utilizar 15 bits en HUB frente a la de 16 bits en el diseño convencional, y de igual modo con 31 bits y 32 bits respectivamente.

Los principales resultados de dicha comparación se recogen en la Tab. 5.2 en la que se muestran la arquitectura propuesta con resultados equivalentes, por una lado HUB-15 Bits frente Convencional-16 bits, y por otro lado, HUB-31 bits versus Convencional-32 bits. Para ambos casos los resultados son significativos, en ambos casos las arquitecturas HUB reducen la cantidad de slices LUTs consumidas, en torno al 10% para 15 bits y 5% para 31 bits. Con el número de slices Registers ocurre lo mismo, aunque la cantidad es un poco inferior que en la de los LUTs. Otro detalle interesante es la frecuencia máxima alcanzada, resulta que las arquitecturas HUB incrementan la frecuencia máxima (Frec.Máx.) en un 35% y 30% sobre las arquitecturas de 16 bits y 32 bits convencionales, respectivamente.

Arquitectura	Slice Reg.	Slice LUTs	Frecuencia Máx.	Máx. Error
HUB-15 Bits	2,576	2,666	568 Mhz.	687.6e-6
Convencional-16 bits	2,810	2,960	421 Mhz.	692.8e-6
HUB-31 bits	10,900	11,188	492 Mhz.	15.33e-9
Convencional-32 bits	11,240	11,792	378 Mhz .1	6.63e-9

Tabla 5.2: Datos de implementación FPGA para QRD formato convencional contra un formato HUB con un error similar.

En resumen, se consigue una factorización QR con redondeo numérico con mejores resultados que la típica fundamentada en el truncamiento. Además se logran mejores tiempos de procesamiento al verse reducida la latencia junto con una rentable optimización de recursos. Una de las cosas más interesantes es ver cómo gracias al formato numérico HUB, una arquitectura donde se truncan los valores se ve superada por otra donde teóricamente al aplicar redondeo debería presentar peores resultados con respecto a consumo de área y caminos críticos.

6 Resolución de JSSP a través de GA en FPGA

6.1 Fusión de los problemas de planificación JSSP, Algoritmos Genéticos y FPGA

La teoría de complejidad computacional es una rama de la teoría de computación que se centra en la clasificación de problemas computacionales a partir de la dificultad propia de la definición del problema. La clasificación de la complejidad se define en base a factores, tales como el tipo de problema, el modelo de cómputo, precisión, finitud, los recursos acotados y sus cotas, etc.

La clase NP (Non-deterministic Polynomial-time) es una clasificación fundamentada en la complejidad computacional para describir ciertos tipos de problemas de decisión. La definición formal indica que el conjunto de problemas NP son aquellos que son resolubles en un tiempo polinomial por una máquina no determinista de Turing, es decir, dada una evidencia o solución ésta es verificada en un tiempo polinomial.

Dentro de la clase NP, existe un subgrupo denominado NP-completo en la que están presentes una gran cantidad de problemas de complejidades intratables. Algunos de los problemas que nos encontramos en NP-Completo son por ejemplo: problemas de satisfacibilidad booleana, problemas del ciclo hamiltoniano, problemas de la mochila, problemas coloración de grafos, etc.

Los avances tecnológicos han permitido explorar nuevos caminos para resolver los problemas NP-completos. La inmensa mayoría son resueltos a través de un software que explotan al máximo los recursos del sistema, memoria, CPU, GPU, etc. Sin embargo, también se han de tener en cuenta las FPGA como opción para la resolución de problemas NP-completos, ya que éstos dispositivos cuenta con un alto nivel de paralelismo capaz de competir con la CPU.

En general, las FPGAs ofrecen un buen número de recursos de computación distribuida donde se llegan a explotar niveles de paralelismo difícilmente alcanzables por una CPU. Otra ventaja es alta capacidad de reconfiguración que ofrece el lenguaje basado en HDL junto con una reducción del consumo frente al de una CPU. Añadir que las FPGAs permiten de forma sencilla implementar un circuito de forma más económica respecto a la relación coste/eficiencia que presentan las aplicaciones específicas basadas en circuitos integrados (ASIC).

Por las razones anteriores las FPGAs han servido de gran utilidad para la aceleración de algoritmos en gran variedad de áreas [73], como por ejemplo en filtros activos, procesamientos de imágenes y vídeo, visión artificial, robótica, criptografía, redes, etc.

En la literatura encontramos varios trabajos en los que las FPGAs resuelven problemas NP-completos de forma eficaz, pero la tendencia siempre ha girado en torno a tres tipos de problemas: problema de satisfacibilidad (SAT)[74], problema de la mochila [75] y el problema del viajante (TSP) [76][77][78]. El motivo de ello se debe a la fácil representación de las soluciones y simplicidad de los operandos.

Entonces, con el objetivo de expandir una implementación basada en FPGA a nuevas problemáticas no tan trabajadas hemos optado por la resolución del problema JSSP (Job Shop Scheduling Problem). Los JSSP son problemas que no sólo pertenecen a la clase NP-completo, sino que forman parte de los NP-Hard, subgrupo de NP con aún mayor dificultad. Como bien se indica en [33], los JSSPs se consideran uno de los peores miembros de la clase NP-Hard.

Los problemas JSSP han sido ampliamente estudiados dentro del mundo de la industria de producción y fabricación. Su origen proviene del comienzo de la Segunda guerra mundial, debido a la necesidad urgente de asignación de recursos escasos en las operaciones militares, extrapolado a la manufacturación. Éstos forman un grupo de conocidos problemas de optimización combinatoria en los que el objetivo es reducir los costes y minimizar los retrasos en la producción. Por lo que la solución consiste en encontrar la mejor planificación para la producción y utilización de recursos definidos de la mejor forma posible.

Los enfoques software para la resolución de los problemas JSSP son muy diversos. Comenzando por los métodos exactos que exploran todas las soluciones, los algoritmos de ramificación y poda son un buen ejemplo de resolución, sin embargo, su principal inconveniente es el extenso tiempo que éstos tardan en encontrar la solución óptima, a pesar de contar con una refinada búsqueda exhaustiva del conjunto de soluciones[79].

Aunque los métodos exactos exploran una gran cantidad de soluciones hasta alcanzar la óptima, en este caso son una opción poco viable puesto que sus tiempos de resolución son extremadamente prolongados. Por ello, la inmensa mayoría de ejemplos que hemos encontrado para solucionar JSSP se fundamentan en métodos aproximados. Los siguientes ejemplos muestran distintas metodologías de aproximación que se han aplicado en JSSP: optimización de colonia de hormigas (ACO) [80], método de búsqueda tabú (TSM) [81] [82], algoritmo genético (GA) [83][84][85], algoritmo memético (MA) [86], recocido simulado (SA) [87] o redes neuronales [88].

A pesar de una amplia variedad algorítmica citada anteriormente para lograr una resolución de los problemas JSSPs, sólo encontramos [89] que se haya llevado a cabo en una FPGA satisfactoriamente. En el artículo [89] se resuelven los JSSPs mediante una metodología de recocido simulado.

Con respecto a los GA en FPGA, encontramos algunos ejemplos en los siguientes artículos [90] [91], en ellos se define una implementación de un núcleo para FPGA, pero en ningún momento se han encontrado desarrollos concretos para JSSP. La principal razón de ello se debe a la singular codificación de cromosomas, conllevando que la operación de reproducción o la función de evaluación no sean tan sencillas de implementar como en otro tipo de problemas.

Por los motivos anteriores se propone la unión de éstos tres conceptos capaces de enlazarse entre sí, siendo el nexo de unión la utilización de un algoritmo genético el elemento que vincule la resolución de los problemas JSSP a través del diseño sobre plataformas FPGA. El objetivo es conseguir una arquitectura hardware parametrizable basada en FPGA para la resolución de forma

genérica de los problemas JSSP. Dando por hecho que se ha de intentar aprovechar al máximo los beneficios que aportan las FPGA como aceleradores de cómputo.

6.2 Diseño basado en FPGA para JSSP a través de GA

En este apartado se presenta una prueba de conceptos, vinculando y fusionando GA, JSSP y FPGA, sin embargo no es una tarea fácil, puesto que el denominador común es el algoritmo genético condicionado por los otros dos extremos.

Consecuentemente, en el momento de diseñar el algoritmo hemos de tener presente su diseño factible desde un punto de vista hardware y su contribución positiva en el GA. Todo ello, ha sido decisivo en el momento de escoger el formato de representación y los operandos que los acompañan.

6.2.1 Algoritmos Genéticos con diseño Hardware

Existe múltiples arquitecturas hardware que implementan algoritmos genéticos [84][85]. Es complejo crear estructuras para solventar una amplia variedad de problemas, puesto que muchos de ellos presentan índoles muy dispares. Pero lo que no es correcto es crear estructuras no parametrizables, ya que en la medida de lo posible se ha de aportar la mayor flexibilidad a los diseños realizados para un rango de problemas.

Resulta de gran interés los trabajos [90] [91] [92][93] donde se define un GA de propósito general a nivel hardware. Se puede configurar un GA con múltiples opciones con respecto a la población, generación de cromosomas, cruces, mutación, etc., creando algoritmos genéticos en FPGAs de gran flexibilidad. Sin embargo, ninguno de ellos se ha propuesto resolver complejas funciones que requieren los JSSPs.

Por lo tanto, en la literatura no se encuentra un diseño general y flexible que pueda solucionar los problemas Job Shop. De ahí, que nuestra propuesta cuente con un diseño moldeable y configurable a través de varios parámetros, buscando el mayor grado de solapamiento entre las operaciones que requiere un AG, cruce (Crossover), mutación y cálculo del Fitness o evaluación. Siendo este último mucho más diferente de todas las evaluaciones vistas hasta el momento.

6.2.2 Algoritmos Genéticos para JSSP

A groso modo, el algoritmo 6.1 describe los pasos elementales de un GA en el contexto de la planificación. El Algoritmo 6.1 sigue un esquema genérico para cualquier GA, pero sus operadores son específicos de los problemas JSSP.

Las GAs comienzan con la creación de una población inicial aleatoria formando un conjunto dado de posibles soluciones, también referidas como individuos, cromosomas o planificaciones para JSSPs. El GA progresa hacia mejores soluciones debido a la repetición de cuatro pasos básicos descritos en el algoritmo 6.1: selección (3), operadores de recombinación (4), función de evaluación (5) y reemplazo (6). Descritos en las sucesivas secciones donde se muestran algunas propuestas de operadores y justificando la elección tomada en cada uno de los casos.

Algorithm 6.1 GA Genérico para JSSP

Require: Instancia del problema de planificación
Ensure: Mejor planificación evaluada
1. Generar la población inicial con valores aleatorios.
2. Evaluación del makespan de la población inicial.
while Criterio de parada **do**
3. Seleccionar planificaciones de la población actual.
4. Generar nuevas planificaciones con los elementos seleccionados a través del crossover y mutación (operadores recombinacionales).
5. Evaluar los nuevos valores de makespan (fitness) para las nuevas planificaciones.
6. Reemplazar las peores planificaciones por las planificaciones mejoradas.
end while

6.2.3 Representación cromosómica de la planificación

En primer lugar, uno de las decisiones más importante antes de embarcase con un GA para JSSP, es seleccionar la forma más eficiente para representar las planificaciones o cromosomas. La codificación de JSSPs debe representar una planificación específica de todas las tareas, que dará forma a la base de todos los operadores. En el resumen [35], Cheng y los demás, dividen la codificación del cromosoma en dos posibles enfoques, directos e indirectos.

En el modo directo, una planificación consiste en una secuencia de datos obtenidos del problema, tales como trabajos (J_i), máquinas (M_j), operaciones ($O_{i,j}$), etc. Por otro lado, el enfoque indirecto no codifica una planificación como antes, en este caso, la codificación se lleva a cabo con reglas de eliminación derivadas del propio problema.

En nuestro caso, nosotros hemos optado por la representación operativa correspondiente a la aproximación directa [94], siendo de las más comunes en la literatura. Esta codificación sigue un esquema de permutaciones con repetición donde cada trabajo (m) se repite n veces (número de máquinas) para un problema JSSP de dimensione $m \times n$. Entonces, la planificación se representa como un vector $m \times n$ con enésimas ocurrencias de un trabajo concreto, donde el orden indica cada una de las operaciones que lo componen. La Fig. 6.1 muestra una representación del ejemplo JSSP 3×3 . Obsérvese que al usar esta representación es factible que dos permutaciones diferentes puedan representar una misma planificación.

$$\begin{array}{cccccccccc} \langle & 2 & 1 & 1 & 3 & 2 & 2 & 1 & 3 & 3 \rangle \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \langle & O_{21} & O_{11} & O_{12} & O_{31} & O_{22} & O_{23} & O_{13} & O_{32} & O_{33} \rangle \end{array}$$

Figura 6.1: Representación base de una planificación operación para una instancia del problema JSSP 3×3. La parte superior coincide con la codificación y la parte inferior corresponde a las operaciones decodificadas.

Las mejores características de este esquema son la simplicidad de construcción donde la generación de planificaciones siempre cumple con las restricciones, evitando las operaciones de corrección y reparación. Además, este esquema presenta una mayor facilidad para los demás operadores.

6.2.4 Operador selección

En la fase de selección, correspondiente a la tercera etapa en el algoritmo 6.1, el reto es simular la supervivencia de las planificaciones más aptas. Esta selección no es completamente aleatoria ya que la probabilidad de permanecer aumenta si la solución es mejor, es decir, la solución presenta mejores características. Este es el paso que coincide con la selección natural tal y como transcurre en la naturaleza, donde aquellos individuos mejor adaptados perduran por encima de los que no.

Las dos estrategias de selección más comunes desde un punto de vista algorítmico son el torneo y la ruleta. El primero de ellos es un método basado en el elitismo en el que se compara un cierto número de soluciones aleatorias para elegir la mejor. Mientras que el segundo método implica asignar una probabilidad de elección para cada individuo de acuerdo con su calificación.

Obviamente, la selección de torneos es más sencilla de llevar a cabo en hardware, ya que sólo se requiere de la comparación de valores. Para ese motivo, el torneo es el operador de selección utilizado para el diseño propuesto.

6.2.5 Operadores recombinacionales

Los operadores recombinacionales aparecen en el estado 4 del algoritmo 6.1, existe una extensa cantidad de información que habla de operadores de recombinación: cruce (denominado crossover) y mutación. Ambos operadores tienen el papel de crear soluciones a partir de otros cromosomas, es decir, se encargan de generar nuevas soluciones que conformaran a nuevos descendientes del conjunto delimitado como población. El cruce combina, mezcla, casa, etc. varios segmentos de soluciones para conseguir de nuevas, mientras que la mutación genera pequeños cambios o variaciones con el fin de mantener una cierta variabilidad.

Para los casos más simples, el cruce es tan fácil como intercambiar parte del cromosoma por el de otro, a partir de un punto de referencia, denominado punto de corte. Sin embargo, aquí nos encontramos con una situación nada deseable, ya que el caso de la representación por permutación no permite el uso de operadores tan simples. El problema es que al intercambiar bloques entre dos planificaciones, se habilita la posibilidad de crear soluciones no válidas.

En el trabajo [95] basado en técnicas de permutación se proponen varios crossover o cruces del que hemos elegido el denominado como Job-based Order Crossover(JOX). Es un método de fusión diseñado para JSSPs por Bierwirth [94], donde su principal ventaja es producir siempre planificaciones válidas, evitando la reparación, y su implementación es factible en FPGA.

Inicialmente, JOX comienza seleccionando al azar un trabajo que preservará su locus en el cromosoma de siguiente generación. Por lo tanto, los cromosomas descendientes copia el orden de operaciones para un trabajo en concreto del padre, insertándolos en la misma posición. Las posiciones restantes se completan según su orden en el segundo padre. De este modo, se genera otro descendiente factible si el proceso se repite con los padres intercambiados. Como ejemplo, la Fig. 6.2 representa un crossover JOX de dos planificaciones, P1 y P2 (cromosomas padres), para un JSSP de dimensiones 3×3. El resultado son dos nuevos elementos descendientes o hijos, H1 y H2.

Con respecto a la implementación, es muy difícil implementar la operación de cruce JOX de forma parelelizada. Por lo tanto, el cruce se lleva a cabo un modo incremental, donde paso a paso se construye una nueva planificación.

Realmente el operador de mutación es menos complejo de realizar en comparación con otros operadores. Se han propuesto varios operadores de mutación para la representación escogida en

Job Seleccionado: **2**
Padres:

P1:⟨ **2** 1 1 3 **2** 3 1 **2** 3 ⟩
P2:⟨ 3 3 1 **2** 1 3 **2** **2** 1 ⟩

Descendientes:

H1:⟨ **2** 3 3 1 **2** 3 1 **2** 1 ⟩
H2:⟨ 1 1 3 **2** 3 1 **2** **2** 3 ⟩

Figura 6.2: Ejemplo del crossover JOX para una instancia del problema 3×3 JSSP.

[95] basado en la inversión, inserción, desplazamiento, mutación de intercambio recíproco y mutación de desplazamiento cuya meta es modificar la planificación.

Para una mayor simplicidad y buena adaptación al crossover de JOX, hemos elegido la mutación por inserción. Es un caso especial de mutación de desplazamiento donde una subcadena que contiene un gen se coloca en una ubicación aleatoria y posterior dentro del cromosoma. La Fig. 6.3 ilustra un ejemplo de mutación de inserción para un JSSP 3×3, en ella se muestra como el trabajo 1 (J_1) correspondiente a $O_{1,1}$ se ha desplazado 6 posiciones para modificar el resultado de la planificación. Con ello se consigue el incrementando de la variabilidad genética a través de la aleatoriedad.

C1:⟨ 2 **1** 1 3 2 3 1 2 3 ⟩
↑ ↓
M1:⟨ 2 1 3 2 3 1 2 **1** 3 ⟩

Figura 6.3: Ejemplo de mutación por inserción en una representación 3×3 JSSP, el Job 1 (J_1) ha sido desplazado seis posiciones a la derecha para lograr una nueva planificación.

6.2.6 Función evaluación

La función evaluación es una forma de estimar si el cromosoma presenta buenas cualidades, es decir, es la expresión de sus genes al que se le denomina como fenotipo. Su localización encaja con el ítem 5 del algoritmo 6.1 y a veces se suele nombrar como fitness. En el mundo de la fabricación, el makespan es la diferencia de tiempo entre el inicio y el final de una secuencia de trabajos o tareas. Como se mencionó anteriormente, es el método adoptado para calcular la duración de una planificación. Aunque hay otras opciones como: el máximo de tardanza, la tardanza ponderada y el tiempo de flujo total, el makespan es con mucha diferencia la función más estudiada.

Basándose en la representación elegida, su desarrollo requiere de la utilización de la tabla de secuencias y la tabla de tiempos de procesamiento. Esto implica realizar un continuo control especial para indicar el estado en el que se halla cada trabajo. Esta situación es característica en los problemas JSSP, pero también es causada por el tipo de representación que no puede enumerase directamente, por lo que no puede calcularse de una forma más inmediata.

Calcular el makespan no es sencillo, ya que requiere de varios contadores para controlar los trabajos que se procesan y en qué instante del procesamiento se encuentra junto con el total de tiempo. Realmente se parece a la función que implementa un diagrama de Gantt. Como se aprecia, el comportamiento mostrado es totalmente iterativo gobernado por las siguientes premisas: en cada

etapa un nuevo trabajo ($J_i \rightarrow O_{ij}$) es asignado, siempre y cuando, el trabajo previo (O_{ij-i}) haya sido finalizado en ese instante de tiempo y además la máquina asignada a la operación O_{ij} se encuentre inactiva o disponible.

6.3 Arquitectura propuesta JSSP-AG-FPGA

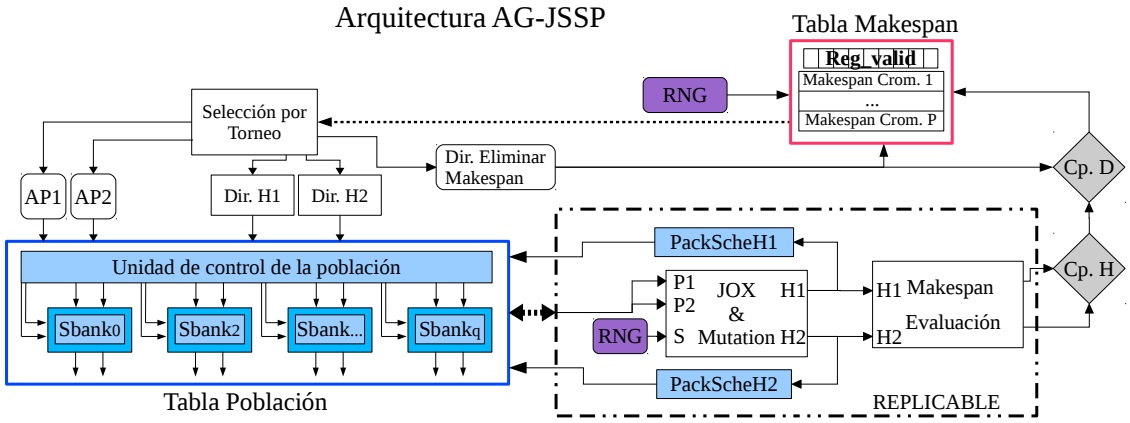


Figura 6.4: Arquitectura general del algoritmo genético para la resolución de JSSP en FPGA.

La arquitectura propuesta es la que coincide con la Fig. 6.4, en este caso se encuentra dividida en varias secciones según su funcionalidad. En primer lugar, las planificaciones (cromosomas) y su makespan o tiempo de producción, son almacenados en dos memorias distintas. Ambas se han construido siguiendo una filosofía de doble puerto para incrementar el solapamiento de operaciones. Al ser memorias de doble puerto se permite realizar dos operaciones simultáneamente, lecturas y/o escrituras.

La tabla situada en la parte inferior izquierda de Fig. 6.4 representa la población con la que trabaja el GA, su objetivo es almacenar individuos y proporcionarlos a la sección denominada como JOX&Mutation. La tabla población se ha construido siguiendo una organización de memoria distribuida. El motivo de utilizar una organización distribuida se debe especialmente al manejo de datos muy extensos. De ello se dará más detalle en las siguientes secciones 6.3.1 y 6.3.2. En cierto modo, la tabla población puede verse como una matriz en la que cada fila se corresponde con una solución del problema JSSP.

Volviendo a la Fig. 6.4, en la parte superior derecha se esboza un diagrama de la tabla encargada de almacenar la función de coste makespan de cada planificación de la población. A diferencia con la tabla anterior que almacena la población, ésta no presenta una organización distribuida, ya que simplemente almacena un valor. En este caso, la tabla que almacena el makespan se considera un vector, donde cada posición corresponde a una fila de tabla población. Además, la sección de la tabla makespan cuentan con un registro para controlar y contrastar si el cromosoma es válido o no, determinado por el operando de recombinación y reemplazo. A efectos prácticos, este registro indica si la posición está libre para escribir una nueva planificación en dicho índice, o en su defecto, ésta se encuentra ocupada por una solución activa en ese momento.

En la parte superior izquierda de la Fig. 6.4 se muestra la sección de selección por torneo. En la sección 6.2.4, ya hablamos sobre la operación de selección y el método de torneo. En este

caso la memoria sólo se comunica con la tabla Makespan, quien aportará la valoración de cada planificación seleccionada, dato que determinará la competición entre individuos. La selección de torneo tiene varias funciones: seleccionar dos cromosomas dentro de la población para un nuevo crossover JOX, determinar las posiciones libres donde se insertarán las nuevas generaciones y escoger al elemento excluido de la población por presentar vagas característica.

Los operadores recombinacionales descritos en 6.2.5 encajan con el módulo JOX&Mutation de Fig. 6.4. Aproximadamente, su objetivo es generar nuevas planificaciones a partir de los padres seleccionados junto con un trabajo aleatorio S, todo esto se aclara mejor en la siguiente subsección 6.3.3. Es importante enfatizar que JOX es un operador incremental donde los genes de la siguiente generación se generan iterativamente, uno a uno, hasta que es completado un nuevo cromosoma.

Mientras que una nueva generación se está creando en el módulo JOX & Mutation, el submódulo Makespan Evaluation está calculando el nuevo valor del makespan a medida que va avanzando la creación de las planificaciones, para ello, en este submódulo se requiere de los datos que definen el problema (tabla secuencia y tabla tiempos). La evaluación del makespan está situada en la parte inferior derecha de Fig. 6.4 y sigue la descripción definida en la sección 6.2.6. La sincronización es de gran relevancia porque la entrada de la evaluación del makespan es la salida del módulo JOX&Mutation, por lo que el ritmo de cruce de $P1$ y $P2$, generando $H1$ y $H2$, ha de coincidir con la velocidad de cómputo del makespan. Como resultado, la evaluación es incremental y segmentada, a fin de aligerar los caminos críticos. Con un mayor detalle en la sección 6.3.4 muestra una descripción de la arquitectura.

Como se muestra en la Fig. 6.4, JOX&Mutation está conectado a la tabla de población a través de los submódulos PackScheH1 y PackScheH2. JOX&Mutation produce genes (operaciones de los trabajos) uno a uno, pero el tamaño del banco de memoria es mucho más amplio. Por esta razón, el submódulo PackScheH apila en un vector un cierto número de genes, los cuales una vez han completado el tamaño del banco de memoria son almacenados en él.

Adicionalmente, se podrían obtener nuevas soluciones posibles simultáneamente si los operadores de recombinación, la evaluación makespan y los empaquetadores PackScheH se pueden replicar de forma paralela. Se ha delimitado la zona replicable mediante una línea discontinua indicando la zona que podría replicarse para acelerar el algoritmo, a través de la generación de más individuos.

El reemplazo o substitución de cromosomas se realiza por los comparadores situados a la derecha de Fig. 6.4, representados con forma de rombo. En primer lugar, el valor comparado es el makespan de los dos nuevos cromosomas hijos, seleccionando el menor makespan entre ellos. Seguidamente, el mejor de ellos compara su makespan con el del cromosoma objetivo y disponible para la exclusión. Si alguno de los nuevos cromosomas es mejor que el seleccionado para eliminar, entonces se produce el reemplazado siendo substituido por el de la nueva generación. Esta operación es tan simple como modificar los registros Reg_Valid de la tabla Makespan.

Finalmente, el término del algoritmo está regido por un contador de iteración. Una iteración consiste en el proceso siguiente: seleccionar a los padres, crear y evaluar nuevas generaciones y reemplazarlas. Una vez alcanzado el número deseado de iteraciones, el algoritmo se detiene. Por lo que el contador es quién determina el criterio de parada.

6.3.1 Esquema de representación

Anteriormente se ha descrito la representación para un problema JSSP de tamaño $m \times n$ como un array con n ocurrencias de los m trabajos. Cada ocurrencia, también denominada operación

o gen, está formada por la cantidad suficiente de bits para representar los m trabajos, es decir, la potencia de dos más cercana al número total de trabajos (Jobs).

Por tanto, la envergadura del problema será quien configure el tamaño de las planificaciones. Por ejemplo un 3×3 JSSP requiere 9 posiciones, mientras que un problema 6×6 JSSP, requiere de 36 genes respectivamente. Para el primer caso con un tamaño de 2 bits por gen es suficiente, alcanzando un total de 18 bits. Para el segundo caso se necesita un total de 3 bits por gen, necesitando 108 bits para representar una planificación. En ambos casos se trata de tamaños manejables para los buses de la arquitectura.

Sin embargo, los problemas La26-30 o la31-35 [34] se encuentra a una escala mucho mayor que los anteriores, sus tamaños son de 20×10 y 30×10. Aunque no son los problemas de mayor envergadura, su representación ya requiere 1000 y 1800 bits para un único cromosoma. Valores demasiado elevados para los buses y para la gestión de la información. Por este motivo se justifica la fragmentación de las soluciones por partes, como bien es mostrado en la figura Fig. 6.5.

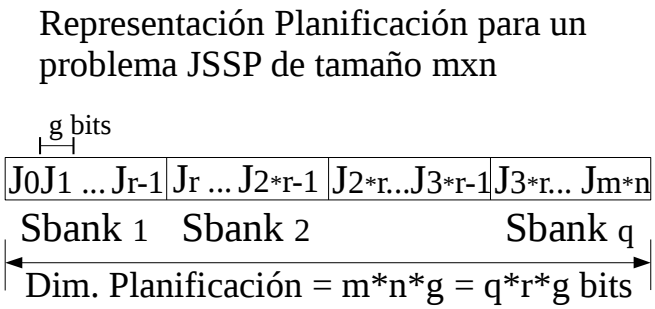


Figura 6.5: Representación genérica de una planificación o cromosoma para la arquitectura propuesta.

La Fig. 6.5 describe cómo un cromosoma está representado por trozos en un problema $m \times n$ JSSP. El número de trozos en los que es fraccionada una planificación corresponde con la variable q, resultado de dividir el tamaño del cromosoma por el ancho de banco de memoria. El valor r describe el número de genes o trabajos que se ajusta o encaja para un el ancho de palabra de un banco de memoria. En la parte inferior de la figura se muestra las dos formas de obtener el tamaño de la planificación, teniendo en cuenta que la cantidad total de bits es redondeada hacia arriba hasta acoplarse al tamaño de los bancos de memoria.

6.3.2 Tabla Población

La tabla de población es donde se mantienen todas las soluciones temporales del problema en un instante determinado, además en ella se distinguen aquellas planificaciones activas para futuros cruces. Debido al gran tamaño de los cromosomas se almacenan en bancos de memoria. El ancho del banco se define mediante un parámetro interno configurable denominado Bank-width, éste se corresponde con el valor q definido en el apartado anterior 6.3.1. Realmente, q viene determinado por el tamaño de la planificación dividido entre el ancho del banco de memoria (Bank-width).

La profundidad de cada banco de memoria es determinada por el parámetro MaxPob, que se encarga de acotar el tamaño máximo de la población, es decir determina la cantidad máxima de elementos que podrá tener la población.

Los bancos de memoria funcionan independientemente y permiten leer y escribir en múltiples bancos con diferentes direcciones. Además, su implementación es dual-port, permitiendo así la doble lectura o escritura para cada banco de memoria simultáneamente.

Las lecturas y escrituras son regidas por la Unidad de Control de Población a través de señales que vienen de JOX&Mutation y PackScheH. Las posiciones leídas vienen indicadas por las posiciones preestablecidas por los registros de dirección AP1 y AP2, seleccionando los segmentos solicitados. A su vez las escrituras se realizan sobre las direcciones indicadas por los registros Dir.H1 y Dir.H2, posiciones donde se escriben las nuevas planificaciones creadas por los operandos de recombinación.

6.3.3 Submódulo JOX Y Mutation

En la sección 6.2.5 ya se ha mencionado la funcionalidad de este módulo, sólo se ha presentado desde un punto de vista conceptual. A continuación, aclararemos ese punto mostrando una eficiente implementación hardware puesto en práctica en nuestra propuesta.

Inicialmente es necesario conocer las entradas para poder realizar la combinación JOX, en este caso son tres: P1, P2 y S. Las dos primeras son las que aportan una parte de la planificación o cromosoma objetivo que será recombinados. Los datos llegan desde la tabla población direccionados por los registros AP1 y AP2, posiciones que ocupan las planificaciones padres dentro de la tabla población.

Para llevar a cabo las lecturas sobre la tabla de población se ha utilizado una Unidad de Control de la Población, quien rige y controla los bancos de memoria requeridos en cada momento. La entrada S se corresponde con un trabajo aleatorio devuelto a partir de un RNG (Generador de números aleatorios), necesario para realizar la recombinación JOX. Realmente S es el gen que será preservado al realizar el JOX en cada uno de los padres.

Este módulo genera dos salidas en paralelo, una por descendiente H1 y H2. El proceso trabaja de forma iterativa a fin de acelerar el avance de la recombinación, generándose una nueva operación de la planificación hijo en cada iteración. Por lo tanto, cada gen de la descendencia se calcula uno por uno basándose en la segmentación.

El algoritmo implementado para realizar el cruce se describe en la Fig. 6.6. Su implementación ha sido modelado a través de la máquina de estado de Mealy, donde en cada transición de estados se introduce una nueva operación $J_i \rightarrow O_{i,j}$. Por definición el cruce JOX mantiene en las posiciones originales los valores que coinciden con el valor S escogido, por ello cuando coinciden en el intercambio dos posiciones donde una de ellas equivale a S, realmente no se produce intercambio se mantiene el valor en la posición, dejando el descendiente en pausa. Consecuentemente, la salida puede ser válida para un descendiente o ambos a la vez en cada instante, de ahí la necesidad de unos empaquetadores de genes (PackScheH1 y PackScheH2) inmediatamente después.

Una vez iniciado, los primeros trabajos de la secuencia de tareas que forma la planificación pasan a ser leídos para compararlos con el valor preseleccionado S. Si ambos son idénticos a S o diferentes de S, entonces H1 es igual a P2 y H2 es igual a P1, se intercambian los valores entre P1 y P2. Seguidamente se avanza a la siguiente posición o tarea, tanto para P1 como P2. En otro caso, si la posición leída es igual a S, en su descendencia inserta el valor S y el padre afectado avanza una posición. En la Fig. 6.2 se muestra un ejemplo de cruce por el método JOX.

La mutación no sucede en todos los cruces, ya que está basado en una función probabilística dependiente del RNG. Realmente, el RNG genera valores constantemente que van siendo acumu-

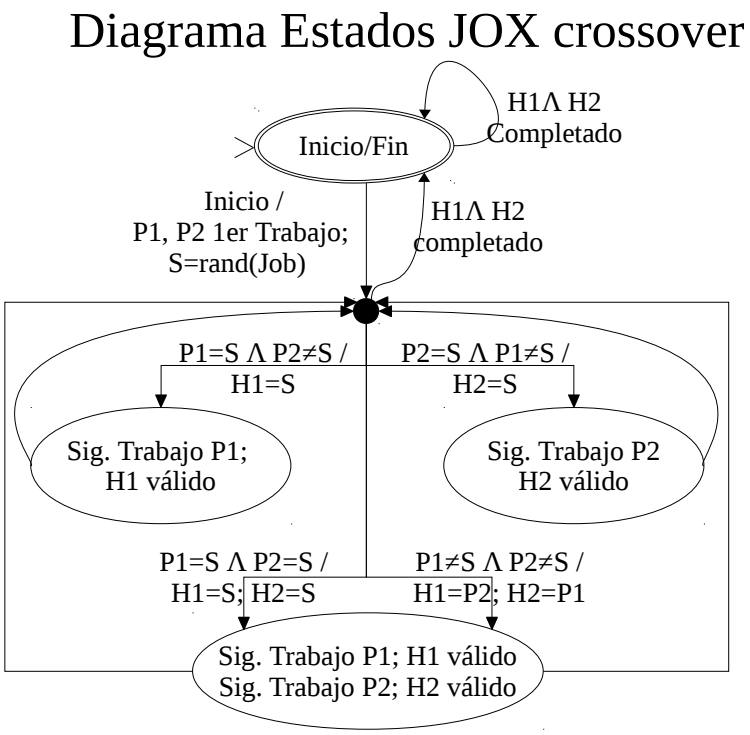


Figura 6.6: Diagrama de estados del elemento encargado de realizar el crossover entre dos soluciones.

lados en un contador que cuando superan un umbral delimitado, la mutación tiene lugar. El diseño de la implementación se puede crear de forma sencilla, debido a que el proceso consiste en retrasar en el tiempo la salida de una de las tareas exactamente un número aleatorio de posiciones en la planificación.

La implementación de la mutación sólo requiere de un registro que almacene la operación que se va retrasar, decrementa el registro un número aleatorio de posiciones hasta que sea cero. Una vez alcanzado el valor se devuelve el valor almacenado en el registro y todas las operaciones siguientes de la planificación también son retrasadas una posición, pasando por dicho registro. Si se da el caso que el cromosoma ha alcanzado su última posición, pero la mutación ha comenzado sin que se haya insertado aún el trabajo mutado, éste pasará a ser insertado al final de la planificación.

La implementación de la mutación presenta ciertas facilidades debido a que las planificaciones se generan como una secuencia de trabajos, uno por uno. Retrasar simplemente una salida del trabajo válida en cualesquiera de los descendientes, ya altera la secuencia del cromosoma y puede cambiar el valor de la solución al problema, convirtiéndose en el factor determinante para la variabilidad genética.

6.3.4 Submódulo de evaluación makespan

Como se indicó de forma teórica en el apartado 6.2.6, uno de las mejores formas de valorar una planificación es a través del diagrama de Gantt, pero para ello se han de realizar varios cómputos en paralelo. De inicio requiere de acceso a los datos del problema porque debe de conocer en cada instante la tabla de secuencia de operaciones ($O_{i,j}$) y la tabla de tiempo de procesamiento ($p_{i,j}$).

La evaluación del makespan está guiada por las salidas del módulo JOX&Mutation dado que dicha salida se decodifica como una tarea dentro de la secuencia de operaciones. En cada ciclo se obtiene una nueva tarea para evaluar, por lo tanto, la valoración del makespan se ve obligada a sincronizarse de la misma forma. Por ello, se ha diseñado una arquitectura segmentada capaz de solaparse con el crossover JOX donde se produce una entrada en cada ciclo.

Para el cálculo de Makespan se han de considerar tres datos relevantes: tareas realizadas (Job-Task), tiempo acumulado en cada trabajo (TAJob) y tiempo acumulado en cada máquina (TAMac). Por eso cada uno de los datos es almacenado como si fuera un vector. Si lo vemos desde el punto de vista hardware, simplemente se traduce como un vector de registros, tal y como se ha representado en la Fig. 6.7.

El diseño propuesto para el cómputo del makespan se muestra en la Fig. 6.7, ésta consta de 5 etapas, donde la segmentación gobierna la arquitectura en la que los registros se propagan entre las distintas etapas. Para simplificar y no saturar la figura se han evitado ciertas indicaciones que se dan por hecho, por ejemplo, el registro J[1] lee el valor J[0], siendo así en cada ciclo sin que sea necesario ninguna flecha para su señalización. Esta medida se toma para evitar la confusión con una imagen demasiado sobrecargada.

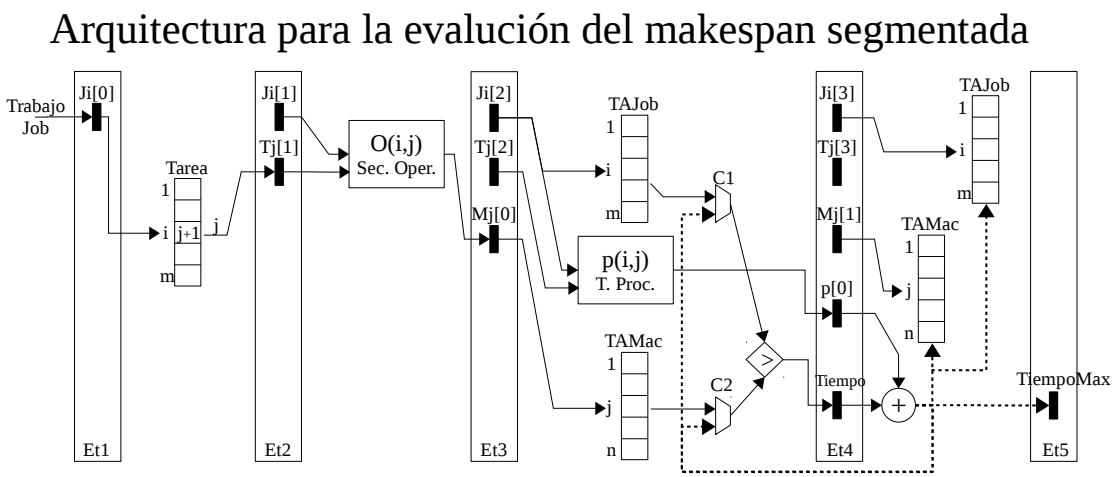


Figura 6.7: Arquitectura para la evaluación del makespan de una planificación implementada en 5 etapas para cada tarea de la planificación. Para no enmarañar la figura los registros con el mismo nombre son leídos en la siguiente etapa.

El número de etapas generado es provocado por la dependencia de datos y la imposibilidad de realizar algunas operaciones simultáneamente. La siguiente lista explica brevemente las diferentes etapas de la arquitectura propuesta en la Fig. 6.7.

st.1: El registro *J[0]* almacena el trabajo entrante J_i.

st.2: El registro *J[1]* carga el valor *J[0]* y en el mismo instante, el trabajo entrante J_i se utiliza para decodificar la siguiente tarea j utilizando el vector Job-Task, valor que posteriormente se almacena en el registro denominado Task. Exactamente, la tabla JobTask no es más que un contador de tareas evaluadas o finalizadas por cada Job, por ello ha de leerse en qué estado se encuentra y a su vez incrementar una unidad para dejar constancia de su realización en la

siguiente consulta.

st.3: *J[2]* lee el registro *J[1]*, y además, *J[1]* junto con los registros Task permiten leer la máquina destino de la tabla de secuencia de operaciones ($O_{i,j}$) junto con el tiempo de procesamiento de la tabla de tiempos de procesamiento ($p_{i,j}$). Las lecturas se almacenan en los registros *M[0]* y *p[0]*.

st.4: *J[3]*, *M[1]* y *p[1]* registran los valores de *J[2]*, *M[0]* y *p[0]*, respectivamente. Además, se comparan dos instantes de tiempo obtenidos de TAJob y TAMac, el inicio de una nueva tarea y la máquina inactiva, almacenando el mayor de ellos en el registro Tiempo.

st.5: En la última etapa, el registro TiempoMax almacena el mayor valor del sumador siendo el resultado del makepan. Para ello, se añaden los registros *p[1]* y Tiempo cuyo resultado se guarda en la posición correspondiente de la matriz de registros *TAJob* y *TAMac*.

Un detalle importante en la Fig. 6.7 son las señales de control de los multiplexores *C1* y *C2*. Existe el riesgo que dos máquinas idénticas o dos trabajos iguales se procesen secuencialmente una detrás de otras, provocando que se lean valores desactualizados. Para evitar este problema se han implementado dos cortocircuitos en la arquitectura, en los que se deriva hacia atrás el valor de *TAJob*, posición donde el multiplexor *C1* devuelve el mayor de los dos trabajos consecutivos e iguales. Del mismo modo se ha realizado el mismo baipás hacia atrás del valor *TAMac* para dos máquinas consecutivas en el multiplexor *C2*, devolviéndose el mayor de los valores nuevamente.

Finalmente, se llega al final del algoritmo cuando todos los trabajos se han completado con el resultado del makepan calculado y almacenado en el registro *TiempoMax*.

6.4 Experimentos y resultados

Para llevar a cabo los experimentos se ha hecho uso de la plataforma hardware FPGA Virtex-6 XV6VLX240T velocidad -2. El diseño ha sido implementado a través del lenguaje VHDL haciendo uso del software ISE 14.3 de Xilinx para su diseño y simulación.

Para realizar las pruebas con una CPU se ha diseño el mismo algoritmo a través del lenguaje de programación C bajo un sistema operativo Linux 64 bits. En el diseño se han implementado cada una de las operaciones necesarias para implementa el GA tal y como se encuentra definido en en el algoritmo 6.1. Además, al realizar la compilación del código se ha hecho uso de la opción gcc -O3 -lpthread para permitir la optimización de código y el uso de hilos de computación.

En este caso el procesador escogido ha sido un Intel Core i7-3537U con una frecuencia base de 2 Ghz capaz de alcanzar los 3.1 Ghz con el Turbo Boost. Las instrucciones son de 64 bits y tiene una cantidad de 2 núcleos permitiendo hasta 4 subprocesos con un tamaño de cache de 4MBytes. La memoria RAM disponible es de 8 GigasBytes

También se ha hecho uso de MATLAB R2014b, haciendo especial mención a su gran precisión numérica al trabajar con representaciones flotantes de doble precisión. Su principal uso ha sido para contrastar los valores de las soluciones, ya que dispone de librerías para realizar diagramas de Gantt y para realizar varios gráficos.

Por último, mencionar que para realizar los experimentos hemos utilizado el conjunto de problemas estándar de los datos de Lawrence (LA01-LA40) [96]. Es la agrupación de problemas tipo de referencia donde se consideran una amplia gama de tamaños de problemas.

6.4.1 Recursos utilizados en la arquitectura propuesta

Dispositivo	**Xilinx Virtex 6 XC6VLX240T -2**				
Problemas	**La01-05**	**La06-10; La16-20**	**La21-25; La36-40**	**La11-15**	**La26-30**
Tamaño	10x5	15x5; 10x10	15x10; 15x15	20x5	20x10
Bits por Trabajo	4	4	4	5	5
Num. Tareas	50	75; 100	150; 225	100	200
Longitud Bits	200	300; 400	600;900	500	1000
Slice Registers	2,433(0%)	2,440(0%)	2,447(0%)	3,347(1%)	3,354(1%)
Slice LUTs	3,965(2%)	4,106(2%)	4,583(3%)	4,657(3%)	5,128(3%)
Bloques RAM	10(2%)	18(4%)	34(8%)	18(4%)	34(8%)
Max. Frec. (Mhz)	221,163	221,163	221,163	212,427	212,427

Tabla 6.1: FPGA resultados de implementación para 16, 24 y 32 bits de tamaño de palabra.

Se han realizado varias configuraciones de la arquitecturas propuesta capaces de resolver el banco de problemas LA01-30, para ello se ha hecho uso de la configuración mínima por cada grupo de problemas presentado. Las configuraciones han sido definidas por dos parámetros, según el tamaño del problema que es dependiente de dos variables: número de Trabajos (Jobs) y número de Máquinas (Machines).

Con respecto al parámetro MaxPob que hace referencia al tamaño máximo de la población, hemos considerado que es fijo para todas las configuraciones testeadas. Se ha tomado el valor estándar para Maxpob igual a 8, pudiéndose alcanzar un total de hasta 256 planificaciones en la tabla población.

Para verlo con un ejemplo, los problemas La01-05 son un grupo de problemas definidos para 10 Trabajos y 5 Máquinas, por lo tanto requieren de un tamaño total de planificación igual o superior a 50 tareas. Cada una de estas tareas representará un valor entre 1 y 10, que será representada con un 4 bits cada una correspondientes al redondeo al número natural superior del logaritmo en base dos, alcanzando un total de 200 bits por planificación.

Los resultados de los recursos utilizados pueden verse en la Tab. 6.1 dividida en 6 columnas. Las 5 últimas columnas hacen referencia a la agrupación de problemas según el tamaño del problema en el siguiente orden: La01-05, La06-10 y La16-20, La21-25 y La36-40, La11-15, y por último La26-30.

Observando los datos de la Tab. 6.1, se aprecia cómo el número de slices registers prácticamente no se altera para una misma cantidad de bits por trabajo. Incrementando el número total de trabajos, como por ejemplo pasar de 16 a 32 trabajos, sólo incrementa en aproximadamente 1000 slices registers.

Otro detalle importante con respecto a los slices registers de la FPGA es que su utilización es muy baja, ni siquiera supera el 1%. Permitiendo que la FPGA pueda replicar el algoritmo en el espacio disponible.

Con el consumo de slices LUTs no sucede lo mismo que con el anterior, las diferencias en este caso entre los problemas se incrementan de forma lineal. El incremento aumenta a medida que crece el problema, pero tampoco son cantidades descomunales. A lo sumo alcanza el 3% de la

FPGA para el mayor de los problemas La26-30, en el que el tamaño de sus planificaciones es de 1000 bits.

La frecuencia máxima es muy estable para todas las configuraciones, rondando los 220 Mhz para las configuraciones La01-06, La06-10, La16-20, La21-25 y La36-40. Se experimenta un mínimo descenso en la frecuencia máxima a un valor de 212 Mhz para las configuraciones La11-15 y La26-30, aunque la diferencia no es significante teniendo en cuenta que aumenta el tamaño de los bits por trabajo.

El recurso de la FPGA que presenta una mayor fluctuación son los Bloques RAM consumidos. Indicar que existen dos parámetros que tienen una gran influencia tiene sobre el consumo de bloques RAM, el tamaño de la población fijado para todas las configuraciones y la longitud total de la planificación o soluciones. Para el menor de todos los problemas, La01-05 con un tamaño máximo de planificación de 200 bits, tan solo consume un 2% de los bloques RAM de la FPGA. Sin embargo, el consumo de bloques RAM se dispara hasta un 8% cuando nos vamos a las peores situaciones La21-25, la 36-40, y La26-30, donde el tamaño de la planificación va desde los 600 bits hasta los 1000 y el consumo de bloques RAM es de 34. Ese crecimiento es debido al incremento del número de bancos de memoria para representar las soluciones.

6.4.2 Comparación CPU versus FPGA

La siguiente Fig. 6.8 muestra el SpeedUp o aceleración del tiempo de procesamiento de un conjunto de problemas en una CPU frente al tiempo de la FPGA. El SpeedUP se consigue dividiendo el tiempo de procesamiento requerido por la CPU descrita en el apartado anterior entre el tiempo de procesamiento del mismo grupo de problemas por la FPGA descrita también en el apartado anterior. En el eje vertical se muestra la relación CPU divido FPGA, significando la cantidad de veces que la FPGA es más rápida que la CPU.

Se han ilustrado 4 líneas distintas en la Fig. 6.8, cada una de ellas representa una agrupación de problemas según su envergadura debido a que se resuelven con la misma estructura. La línea azul representa la aceleración de la FPGA por encima de la CPU para los problemas La01-05, la línea roja discontinua representa la aceleración para los problemas La06-10, la línea amarilla rayadas ilustra los problemas La21-25, y por último la línea verde continua representa cuántas veces es más rápido el procesamiento en la FPGA en comparación con la CPU para los problemas La26-30.

La cantidad de iteraciones en el problema es una variable que afecta directamente al tiempo de ejecución y el condicionante para realizar la parada. Por ello se han llevado a cabo varias pruebas con distintos números de iteraciones, representadas en el eje horizontal, oscilando entre 10 y 5000 iteraciones.

Como resultado, aunque la CPU es bastante rápida, la FPGA muestra un relativo mejor comportamiento en todos los casos. Para el caso de los problemas La01-05 se muestra una mejora de rendimiento en la FPGA, mostrándose una superioridad entre 18 y 8 veces. Para el caso de La26-30, un conjunto de problemas mucho más grande que el anterior, también hay una notable mejora de rendimiento, oscilando entre 12 y 5 veces.

En todos ellos se aprecia una reducción del SpeedUp frente a la CPU a medida que se incrementa la cantidad de iteraciones. Dicho valor acaba estabilizándose para cantidad de iteraciones elevado en un valor superior a 5 y 8, según el agrupamiento de problemas.

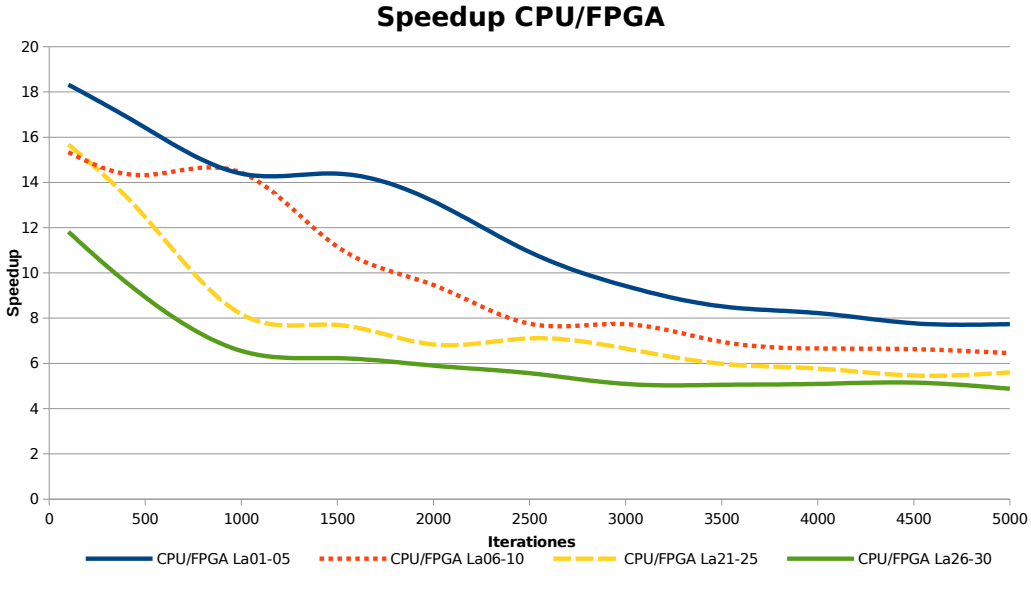


Figura 6.8: Aceleración de la FPGA con respecto la CPU para el rango de problemas definido por Lawrence LA01-LA40.

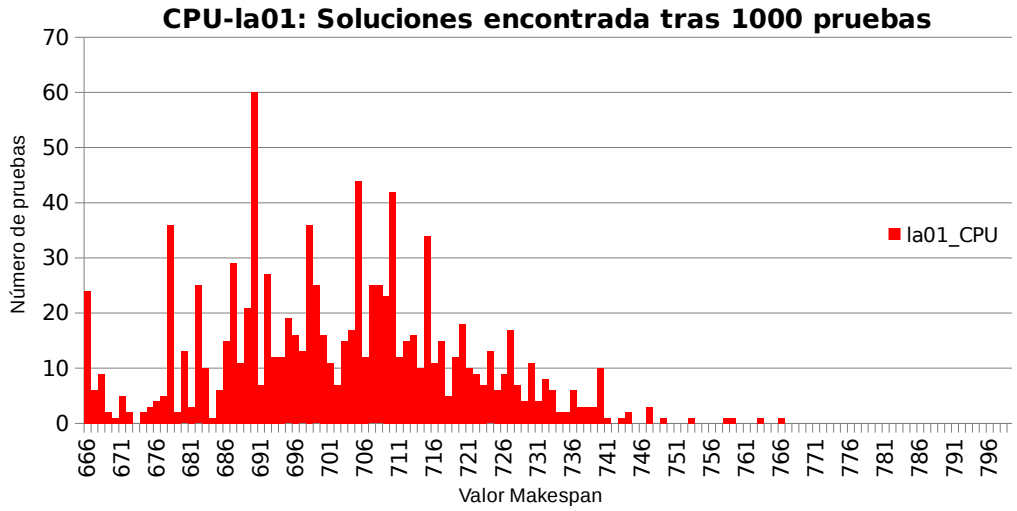


Figura 6.9: Resultado de 1000 ejecuciones del problema La01 sobre la CPU descrita en la sección 6.4.

6.4.3 Alcance de las soluciones

Los algoritmos genéticos operan de forma simultánea con varias soluciones en vez de trabajar de forma secuencial como las técnicas tradicionales. Cuando se utilizan para problemas de optimización en los que su función objetivo en minimizar o maximizar es fácil que caigan en soluciones no óptimas que se tratan de máximos o mínimos locales.

Además, el uso de operadores probabilístico pueden hacer que su convergencia sea prematura o debido a la casuística de ese momento, o todo lo contrario, ya que uno de los valores con los que los GA juegan es con la probabilidad.

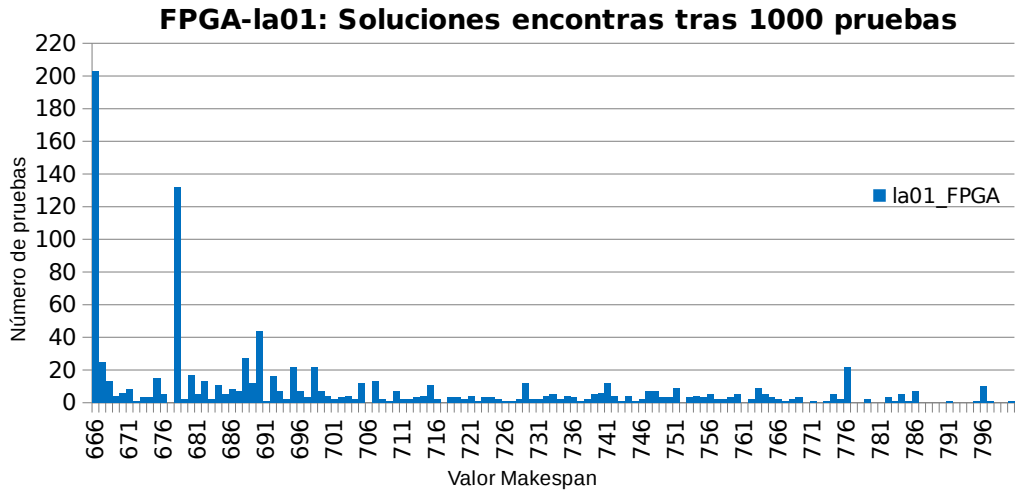


Figura 6.10: Resultado de 1000 ejecuciones del problema La01 sobre la FPGA descrita en la sección 6.4.

En este apartado se ha realizado un pequeño análisis para mostrar un esbozo de la convergencia de los algoritmos, se han contabilizado los casos en los que el algoritmo genera un acierto. El objetivo es conocer qué ocurre en las distintas plataformas testeadas, CPU (software) y FPGA. Evidentemente se han recreado las mismas condiciones a fin de que mantener una misma igualdad, utilizándose los mismos operadores recombinacionales de selección, substitución, etc. Con ello, el tamaño de la población ha sido la misma en ambos casos, 250 planificaciones en la tabla población, y la cantidad de iteraciones en el algoritmo ha sido de 5.000.

El gráfico de la Fig. 6.9 y la Fig. 6.10 muestran un espectro de las soluciones encontradas después de 1000 ejecuciones para el problema La01 sobre una CPU y sobre una FPGA. En el eje horizontal muestra los valores alcanzados como solución, el makespan conseguido, siendo el óptimo el primer valor de ellos (666)[96]. En el eje vertical se muestra la cantidad de repeticiones que ocurre para una solución encontrada.

Al comparar las Fig. 6.9 y la Fig. 6.10 pueden apreciarse ciertas diferencias en el problema La01. Analizando la distribución de la CPU, se puede ver cómo se presentan las soluciones con una dispersión en el histograma. Se muestran 25 casos en los que se produce acierto por parte del software, pero también es verdad que el caso más repetido se centra en un máximo localizado 690 hasta en 60 ocasiones, comprendiendo un rango de soluciones obtenidas que oscila en el rango de 666 y 766.

Sin embargo, en la FPGA la distribución es bastante diferente, también presenta un histograma algo más disperso cubriendo un rango entre el valor makespan 666 y 801, pudiendo alcanzar soluciones mucho peores que la versión software. Aunque la gran diferencia radica en la cantidad de aciertos para el valor solución (666) es de 200 veces, hasta 8 veces más que la solución software.

El mismo análisis se ha realizado para el problema La06, comparando la versión hardware con la versión software representado a través de las figuras Fig. 6.11 y la Fig. 6.12, respectivamente. En este acaso ocurre algo bastante similar al problema La01. Utilizando una misma población, tan sólo se alcanzan 2 situaciones de éxito para el algoritmo software, mientras que para la arquitectura hardware de la FPGA el valor óptimo (926) es alcanzado hasta en 250 ocasiones. En la comparación se puede ver como el rango del histograma de soluciones es bastante similar para ambas gráficas, Fig. 6.11 y la Fig. 6.12. Nuevamente aparece en el algoritmo software un máximo

local altamente repetitivo con valor de makespan 799 unidades de tiempo.

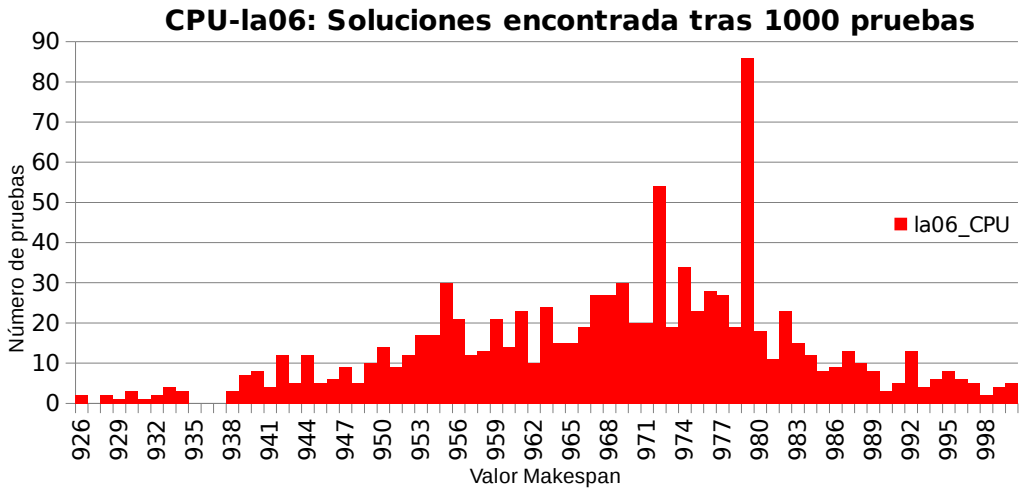


Figura 6.11: Resultado de 1000 ejecuciones del problema La06 sobre la CPU descrita en la sección 6.4.

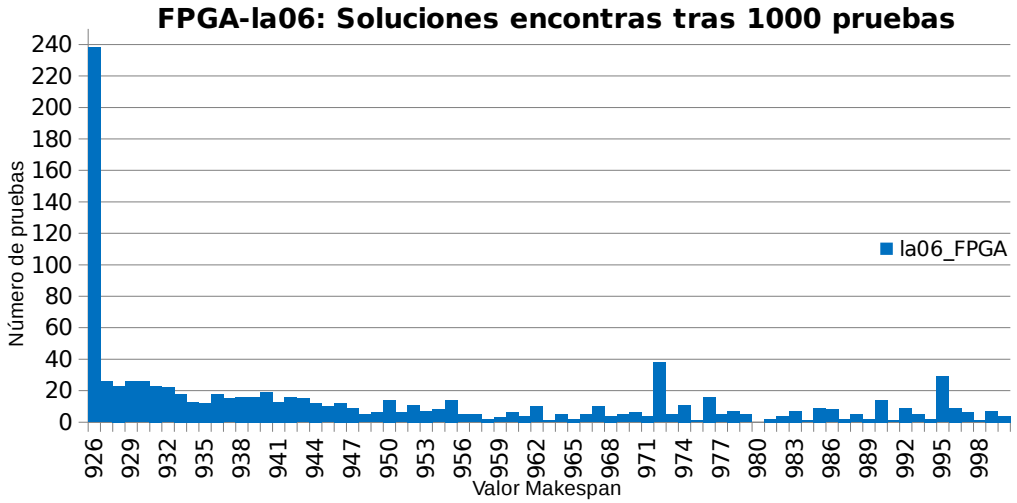


Figura 6.12: Resultado de 1000 ejecuciones del problema La06 sobre la FPGA descrita en la sección 6.4.

7 Conclusiones y trabajos futuros

7.1 Conclusiones Generales

La conclusión principal que queremos destacar es la provechosa ventaja que presentan las FPGA como dispositivos para aceleración de cómputo. A día de hoy, la computaciones heterogéneas más habituales son las que combinan CPU y GPU, principalmente por su facilidad en el momento de su integración y programación. Pero cada día aparecen nuevas herramientas que hacen que la tediosa programación hardware para FPGAs sea más sencilla de unificar con las técnicas habituales. Además con la llegada de chips heterogéneos que integran un chip FPGA, todo hace pensar que va a ser una importante tendencia en cuanto a mejoras para acelerar la computación.

El primer aspecto a resaltar en el momento de llevar a cabo la aceleración de un algoritmo es conocer aquel punto u operación matemática causante del cuello de botella, aspecto que deberá de ser tratado con una mayor relevancia. Con ese punto destacable del algoritmo han de volcarse todos los esfuerzos posibles para conseguir optimizarlo al máximo. A diferencia de las CPU, las FPGA no sólo ejecutan operaciones de forma iterativa, permiten explotar en la medida de lo posible el paralelismo operacional, segmentación operacional, utilización de hardware especifico como DSP, reducir los tiempos de acceso, etc. a fin de perfeccionar cada una de las instrucciones para una fusión mejor.

Entre todo ellos destacar la concurrencia como uno de los aspecto más potentes para la aceleración y optimización de algoritmos. Desde mi punto de vista, en las FPGA es muy provechoso su forma de trabajar basada en la distinción de módulos o cajas que componen el diseño, junto con su clara estructura jerárquica, de hecho gracias a ello, resulta sencillo poner en marcha todos los módulos para que trabajen de forma concurrente y a la vez.

Otro aspecto destacable es la existencia de máquina de estados que actúa cómo unidad de control para respaldar y regular el buen funcionamiento del diseño. Debe fijarse como meta centrase en los diseños con comportamientos interactivos, es decir, es beneficioso pensar en diseños atómicos de máquinas de estados en el que gobierne la concurrencia, para que la unión de ellas forme una unidad de control completa con el mayor nivel de solapamiento operacional con interacción entre módulos.

Es cierto que muchas operaciones presentan dependencias por la definición del algoritmo, como ha ocurrido en varios arquitecturas implementadas, las arquitecturas segmentadas se vuelven ideales donde el consumo de recursos no tiene porque verse afectado. Otro aspecto importante de la segmentación es su simpleza en el momento del diseño, siempre y cuando se logre abstraer las unidades operacionales básicas de cada iteración. Un buen ejemplo es el algoritmo CORDIC vectorización-rotación diseñado, una vez descrito el caso base para una iteración, el lenguajes de programación orientados a hardware (HDL) facilitan enormemente la generación de las demás etapas.

Recalcar la provechosa ventaja de implementar aquellos algoritmos fácilmente paralelizables, y sobretodo, aquellos donde existen operaciones muy repetitivas y requieren de estructuras de carácter n-dimensional. Las FPGA pueden operar paralelamente sobre múltiples puntos al mismo tiempo, algo donde una única CPU presenta dificultades.

Un aspecto notable es tener en cuenta la información y resultados del análisis que ofrecen las herramientas de los entornos de diseño para FPGA en el momento de la implementación. En ellos se presenta datos como por ejemplo el consumo de recursos, frecuencia máxima, etc. resulta importante el hecho de conocer los caminos críticos en la implementación hardware, mostrando aquellos componentes que se ven involucrados en el actual cuello de botella. Con ello es más fácil hacer un seguimiento de dichos componentes que afectan de manera negativa al diseño, muchas veces debido a un exceso de distancia entre componentes, datos que llegan con desfase, etc. Aunque el rutado de las aplicaciones es muy bueno, existes varias posibilidades que tratan de optimizar los diseños en función de varios aspectos como puede ser utilizar la menor cantidad de recursos, realizar el rutado para obtener la menor latencia, o un plan intermedio. Es importante ajustar cada uno de ellos para utilizar el más provechoso en cada situación.

Otro dato importante sobre las FPGAs es que estas disponen de un tamaño de memoria RAM elevado y que cada vez van en aumento, habilitando de este modo la posibilidad de almacenar grandísimas cantidades de datos. En nuestro caso siempre ha sido suficiente con los Bloques RAM, de los que nos gustaría resaltar el bueno número con el que cuenta una FPGA y su rápido acceso, permitiendo la utilización de un doble puerto. Desde mi punto de vista, han de utilizar siempre que sea posible, puesto que actúan como si fueran una memoria cache de una CPU.

Por último, indicar que las FPGAs abren un nuevo mundo por explotar donde ahora el paralelismo es tangible, especialmente con la llegada de nuevos procesadores donde ya tiene integradas FPGA. Ahora será posible liberar al procesador de aquellas tares recurrentes que han de ejecutarse una tras otra vez, ya que en una sección del chip podrán realizarse los cómputos de forma óptima.

7.2 Conclusiones Descomposición QR a través de las rotaciones de Givens

Se han presentado dos arquitecturas bien distintas para la factorización de matrices QR a través del método de las rotaciones de Givens para FPGA, ambas arquitecturas han demostrado ser eficientes y eficaces en cada una de sus contextos.

Comenzando por orden de aparición, la propuesta iterativa diseñado para FPGA es ampliamente flexible gracias a su gran cantidad de parámetros que habilita la opción de crear una arquitectura que se ajuste al problema a resolver. La arquitectura iterativa es capaz de resolver una extensa cantidad de rangos de matrices. El estudio se ha centrado para QR-factorizar matrices de un tamaño grande, en pocas arquitecturas son capaces de resolver tales dimensiones. Con una

mayor predominancia en la literatura, las arquitecturas sistólicas encontradas no permiten factorizar matrices con las dimensiones de nuestra propuesta.

La arquitectura iterativa propuesta está dividida en tres partes, la sección responsable de calcular los ángulos de rotación y dos secciones para la rotación de las filas de la matrices Q y R. Se ha dejado bastante claro los beneficios de realizar esa división por secciones, ya que cada una de ellas presenta distintos niveles de computación, puesto que la sección para la matriz Q es la que más operaciones realiza seguida de la sección R, siendo la más ligera en cuanto a computación la sección del ángulo. Recalcar la ventaja de que cada una de ellas opera de forma concurrente e individual, gobernada por una maquina de estados global.

Las arquitecturas configuradas con un mayor nivel de paralelización presentan un mejor rendimiento, aunque eso implique un aumento de los costes en recursos. Destacar aquellas configuraciones donde la sección Q tiene un mayor número de elementos dedicados a la rotación que la sección R , ya que requieren de un menor uso de recursos de bloques RAM y su distribución de carga de trabajo es más equilibrada que otras arquitecturas, por lo que presentan un menor tiempo de inactividad. A su vez, el tiempo de procesamiento es sumamente mejor para aquellas arquitecturas donde existe un incremento de elementos rotadores en la sección Q en comparación con la sección R.

Destacar la importancia del algoritmo CORDIC para llevar a cabo la factorización QR. El tiempo de procesamiento de los elementos CORDIC tiene una gran influencia en los resultados, ya que el número de etapas definirá el tiempo de vectorización o rotación de los vectores, además de la precisión de los resultados. En otras palabras, si el número de etapas del algoritmo CORDIC es demasiado bajo, la rotación finalizará antes reduciendo la latencia, pero su precisión empeorará por la carencia iteraciones; todo lo contrario se incrementan las etapas del algoritmo CORDIC. Además, nos ayuda a estimar la arquitectura óptima para un rango matricial porque los efectos de la paralelización son más evidentes en las matrices más grandes.

Resaltar que la arquitectura iterativa es configurable a través de tres parámetros genéricos que controlan la longitud de la palabra, el rango máximo de matrices permitido y el número de elementos replicados para calcular las dos matrices de salida R y Q. Gracias a ellos, la flexibilidad del diseño puede ayudar a ahorrar recursos en función del problema, destinando los restantes a nuevas funciones de problemas más generales.

Con respeto a la otra arquitectura para la descomposición QR de matrices a través de una arquitectura sistólica, indicar el buen rendimiento que presenta con la optimización y paralelización de las rotaciones de Givens. En la arquitecturas se prioriza la realización de rotaciones de Givens para que se hagan lo antes posible, con ello conseguimos que la retroalimentación sea más ágil, mitigando tiempos ociosos de los demás dispositivos. Realmente, el cuello de botella del algoritmo se encuentra en el proceso de rotación de filas, por eso, potenciamos el cómputo en esta parte.

Recalcar la importancia de crear diseños propios a medida y que se ajusten a nuestras necesidades del algoritmo CORDIC capaz de realizar la vectorización y rotación a la vez, permitiendo de este modo superponer completamente el cálculo del ángulo y la rotación de las filas. Gracias a ello se alcanza un mayor grado de paralelización entre operaciones. Además con su diseño pipeline ayuda a que la factorización de matrices sea tan elevada, muchos más que las arquitecturas iterativas.

Comparando la arquitectura sistólica propuesta con las demás de la misma índole, en nuestra propuesta se muestran resultados muy positivos en cuanto a recursos utilizados. Nuestra propuesta es con diferencia la arquitecturas más óptima puesto que el rendimiento es un 50% mayor que las demás con una menor utilización de recursos. La alta segmentación permite realizar una descom-

posición de la matriz $n \times n$ cada $2n$ ciclos de reloj.

Realzar el estudio realizado sobre la arquitectura sistólica para matrices 4×4 , ya que se ha optimizado para diferentes longitudes de palabra seleccionando el número apropiado de iteraciones CORDIC. También se ha analizado la cantidad máxima de error del diseño propuesto, permitiendo así conocer la cantidad de bits correctos para una determinada configuración y tamaño de palabra.

7.3 Conclusiones AG para JSSP en FPGA

En el capítulo 6 se han conectado tres conceptos bien diferentes, se ha conseguido fusionar la resolución de problema de planificación a través de algoritmo genéticos desde una perspectiva hardware. Para ello se ha creado un diseño en FPGA que es capaz de encontrar soluciones de uno de los problemas de planificación de complejidad NP-Completa denominados Job Shop Scheduling Problems (JSSP), utilizando como nexo de unión un algoritmo genético.

Realmente pocos son los trabajos que hay en la literatura para la resolución de problemas JSSP a través de hardware, por lo que es un logro crear una arquitectura que sean capaz de resolver tales problemas sobre una FPGA. Además, lo mejor es que se ha creado una implementación ampliamente flexible para el tipo de problemas. Para ello, se ha hecho uso de varios parámetros que configuran la arquitectura para la resolución de un tamaño máximo de los problemas, estableciendo de este modo, una forma de acotar los recursos en base al problema a resolver.

Destacar la implementación del cruce de soluciones a través de un módulo definido para el cálculo del makespan. Dicho diseño denominado JOX&Mutation conforma un implementación segmentada en la que se ofrece un rendimiento óptimo, siendo capaz de calcular el valor de la solución en un tiempo mínimo a medida que ésta es calculada.

Se ha logrado un diseño expresamente para todo tipo de problemas JSSP, siendo no aplicable a otros más generales. Pero con la construcción modular del GA permite substituir dicho módulo por otro que realmente se ajuste a una nueva función de cruce o resolución de otros problemas. Por lo que al ser una arquitectura basada en módulos, cualquier parte puede ser reemplazada por otra que se ajuste a un nuevo tipo de problema, aprovechando el resto de los componentes.

Un aspecto a destacar es cómo se ha conseguido el alto nivel de solapamiento, a medida que se produce el cruce de cromosomas y la mutación, tenemos el valor que cuantifica esa nueva solución a partir del makespan en ese instante. Es muy positivo tal sincronismo y solapamiento para acelerar el cálculo de soluciones.

En nuestra propuesta, la cantidad de recursos utilizados en la FPGA son bastante bajos, siendo el elemento más restrictivo la cantidad de bloques RAM que a lo sumo ocupa el 8% de estos para el mayor de los problemas estudiados (La21-25,La36-40 y La26-30). Por ello aún queda espacio en la FPGA para permitir crear hasta 12 ejecuciones completamente paralelas de instancias completas del algoritmo genético en la FPGA utilizada para el estudio, Virtex 6 XC6VLX240T -2. Los demás datos en cuanto a recursos no son restrictivos, ya que oscila la ocupación de slices entre el 1% y el 3%.

Además a esa capacidad de ejecución paralela ha de añadirse la ventaja que ofrecen las FPGA, entre 18 y 5 veces más rápidas que una CPU actual, mostrándose como buenos ejemplos de aceleradores de cómputo. Aunque no se ha medido el consumo, otro aspecto importante de la aceleración con FPGA es el bajo consumo que éstas siempre muestran en comparación con otros dispositivos, a excepción de los ASIC.

Los resultados son bastante favorables para las pruebas realizadas, como aspecto positivo presentan mayor rango de aciertos que la ejecución software. Sin embargo, en el rango de error es mucho mayor, por ello puede generarse una solución mucho más alejada de la solución óptima que la conseguida por la aplicación software. Es decir, el peor de los casos desfavorece nuestra arquitectura propuesta. Los motivos de ello pueden deberse a varios motivos, como la generación de números aleatorios, la población aleatoria inicial, la selección del gen utilizado para el cruce, etc., o simplemente que el sistema de selección aleatorio podría incrementarse para realizar un torneo con un rango mayor.

Otro detalle a mejorar en el algoritmo genético es el criterio de parada, en nuestro caso éste es cerrado para un número determinado de iteraciones. Pero podría optimizarse de tal modo que parase por otro criterio, como por ejemplo por un estancamiento de la población. En el anexo A y B, se muestra cómo evoluciona la población en cuatro momento de la ejecución, donde puede apreciarse como la tabla población no sufre ningún cambio tanto en la etapa final como en la anterior.

El estancamiento de la población puede darse sin que se haya encontrado la planificación óptima, por lo que podrían añadirse nuevas planificaciones si el algoritmo detecta esa condición y así añadir una mayor variabilidad.

7.4 Trabajos Futuros

Es cuanto a trabajos futuros los posibilidades son múltiples a partir de las secciones propuestas. En función del orden de aparición de las secciones en esta tesis vamos a citar algunos trabajos que podrían realizarse y que se quedan pendientes para el futuro.

En primer lugar aún quedan varios frentes abiertos en base a las arquitecturas QR propuestas. De inicio podría resultar muy interesante continuar con el siguiente paso a la factorización, como por ejemplo con la resolución de sistemas de ecuaciones lineales. Es verdad que este paso requería de nuevas operaciones como divisiones, ya que la resolución al ser una matriz triangular se realiza por medio de substitución hacia atrás.

Otra opción muy buena sería estudiar nuevos métodos de factorización QR, en nuestro caso hemos escogido la opción de la rotaciones de Givens, pero hemos visto otros trabajos en los que también se utiliza Gram-Schmidt y reflexiones de Householder. Podría proponerse un estudio en el que se lleve a cabo un análisis de resultados en función del método utilizado.

Resultaría de gran interés aplicar los resultados de esta tesis a la tecnología MIMO (Multiple Input Multiple Output), con el objetivo de analizar qué ocurre al acelerar la descomposición de la matriz de canales para el manejo de ondas de transmisión y recepción en antenas de dispositivos inalámbricos. También se podría realizar un estudio para verificar si resultaría más útil aplicar una arquitectura sistólica o con una arquitectura iterativa podría ser suficiente.

Otro punto fuerte para seguir trabajando es el de continuar con el estudio de la utilización del formato HUB sobre las arquitecturas QR u otro tipo de arquitecturas. Ver todos los futuros trabajos anteriormente citados para conocer como es su comportamiento y ver cuánto de positiva puede ser, ya que es aplicable a cualquier algoritmo como MIMO, filtros FIR, etc. Desde mi punto de vista, hemos de tener muy en cuenta este nuevo formato, ya que a nivel de aritmética hardware tiene un gran potencial.

Con respecto a los problemas JSSP y la propuesta para su resolución en FPGA, varias son las mejoras y trabajos futuros que se quedan pendientes. Así de inicio, un primer paso podría ser

extender el diseño para solventar otros tipos de problemas de planificación como los citados en la subsección 2.5 denominados como OSS, FSS y PFSS. Todo ese grupo de problemas guardan característica muy similares y el formato de representación podría ser el mismo para todos ellos, solo sería necesario modificar las restricciones de la definición del problema

Algo muy positivo para el futuro sería la posibilidad de crear otras alternativas a los módulos actuales, por ejemplo dar la posibilidad de realizar la selección a través del método de la ruleta en función de probabilidad, implementar nuevos tipos de cruce además de mejorar el actual JOX, etc. Es verdad que todos estos cambios son más bien detalles algorítmicos que pertenecen a otro campo relacionado con la búsqueda de configuraciones de los algoritmo genéticos para aumentar las posibilidades de éxito.

Por último como trabajo futuro aportaría datos interesantes realizar múltiples ejecuciones de problemas y observar el comportamiento de la FPGA, con todas las instancias ejecutándose a la vez aumentaría la posibilidad de encontrar la solución. Además sería de especial interés poder obtener datos sobre el consumo de la FPGA, característica que cada vez gana más peso.

Anexo A: AG-JSPP 3×3

A continuación en este anexo se presenta la ejecución de un GA que resuelve un problema sencillo de JSSP 3×3

A.1 Representacion de la población

En esta sección se ha decidido mostrar la población inicial para una ejecución del problema JSSP 3×3, problema definido en la sección 2.6.2 denominada como Ejemplo de JSSP 3×3. En ella se representa cada una de las planificaciones que se han generado aleatoriamente, correspondiente a cada una de las filas de la tabla. También se ha completado cada uno de las tareas de las planificación con el trabajo al que corresponde al problema.

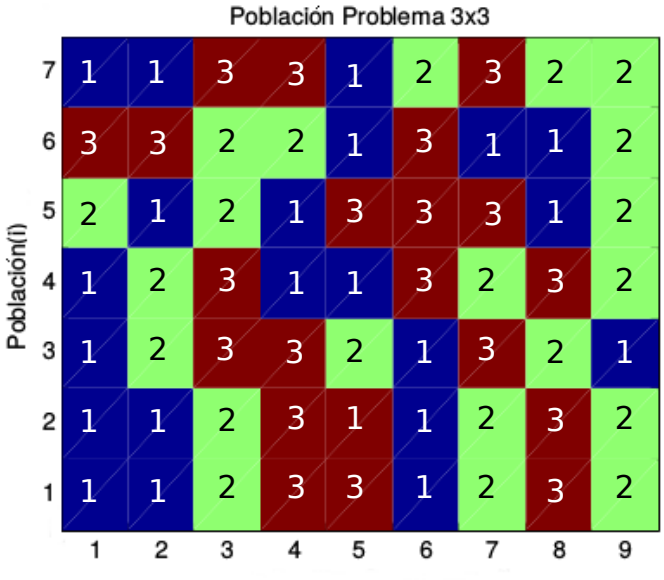
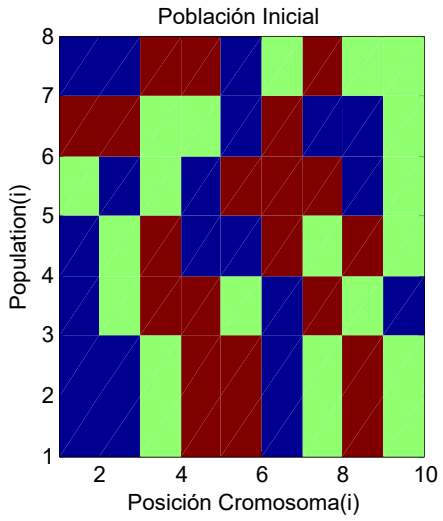


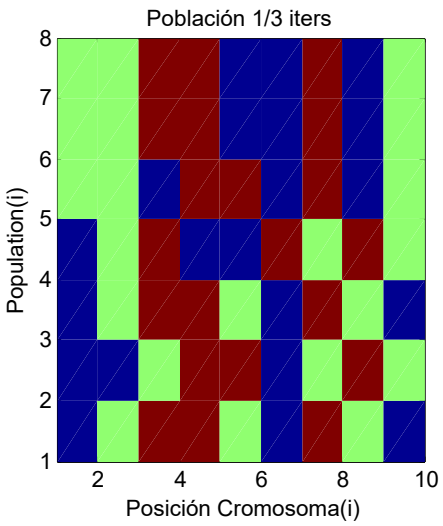
Figura A.1: Decodificación de la población.

A.2 Evolución de la población

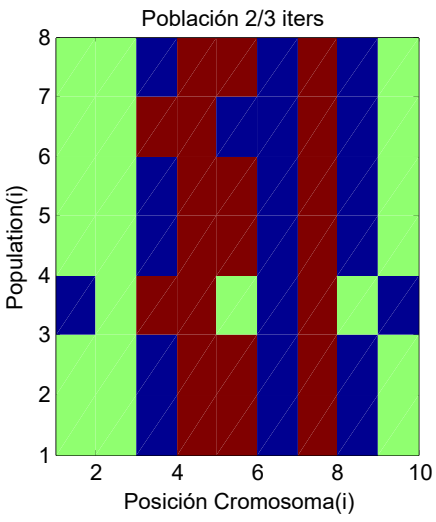
En la figura A.2 se muestra cómo avanza la población en una ejecución 16 iteraciones. Como puede verse en la imagen (a) la variabilidad es mucho mayor, mientras que el la última imagen (d), sólo hay un tipo de planificación, la mejor de todas y que además coincide con la solución al problema.



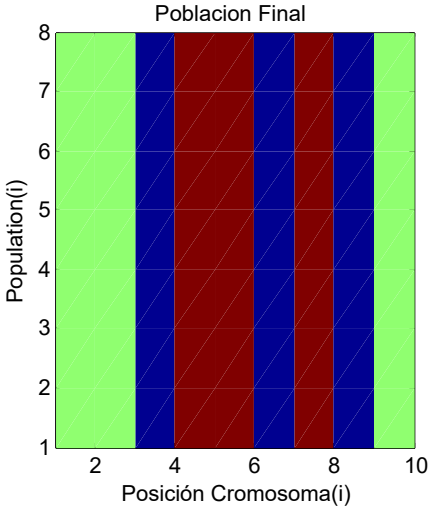
(a) Tabla población del algoritmo en el instante inicial, justo antes de dar comienzo.



(b) Tabla población del algoritmo en el instante 13 de iteraciones.



(c) Tabla población del algoritmo en el instante 23 de iteraciones.



(d) Tabla población del algoritmo en el instante Final.

Figura A.2: Evolución de la población del GA para la resolucion de JSSP.

A.3 Gráfica de evolución del Makespan

En este apartado se ha querido mostrar qué ocurre con el valor del makespan de la tabla población. Simplemente es un análisis de su evolución. Como puede verse en el gráfico A.3 que se corresponde perfectamente con el gráfico visto anteriormente A.2. En el instante inicial, el makespan mínimo tiene un valor de 14 y máximo es de 18 unidades de tiempo. A medida que avanza el problema, es decir, se incrementa el número de iteraciones se van encontrando nuevas soluciones mejores hasta dar con la solución.

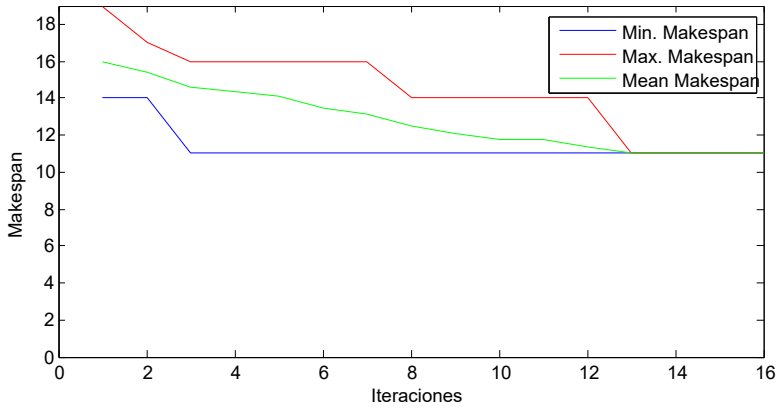


Figura A.3: Grafico de la evolución del Makespan, máximo, mínimo y media.

A.4 Solución encontrada

En esta sección se muestra en la figura A.4 una representación basada en el digrama de Gantt de la solución la problema propuesto. En ella están todas las tareas ordenadas por máquina, alcanzando un total de 11 unidades de tiempo.

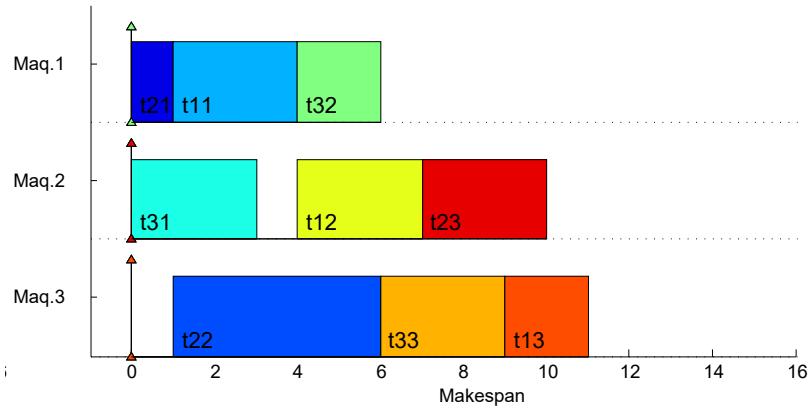


Figura A.4: Planifación óptima encontrada con un valor de makespan igual a 11 unidades de tiempo.

Anexo B: AG-JSPP La01

A continuación en este anexo se presenta la ejecución de un GA que resuelve el problema de JSSP La01.

B.1 Problema JSSP La01

El probla es descrito a partir de la tabla de secuencia de maquinas Tab. B.1.1, junto con la tabla de tiempos de procesamiento Tab. B.1.2.

Secuencia de Máquinas para el problema La01

	Operaciones				
Job	1	2	3	4	5
$j0$	$m1$	$m0$	$m4$	$m3$	$m2$
$j1$	$m0$	$m3$	$m4$	$m2$	$m1$
$j2$	$m3$	$m4$	$m1$	$m2$	$m0$
$j3$	$m1$	$m0$	$m4$	$m2$	$m3$
$j4$	$m3$	$m0$	$m2$	$m1$	$m4$
$j5$	$m1$	$m2$	$m4$	$m0$	$m3$
$j6$	$m3$	$m4$	$m1$	$m2$	$m0$
$j7$	$m2$	$m0$	$m1$	$m3$	$m4$
$j8$	$m3$	$m1$	$m4$	$m0$	$m2$
$j9$	$m4$	$m3$	$m2$	$m1$	$m0$

Tabla B.1: Secuencia de máquinas recorridas para realizar los trabajos $j1$, $j2$ y $j3$, en cada una de sus operaciones.

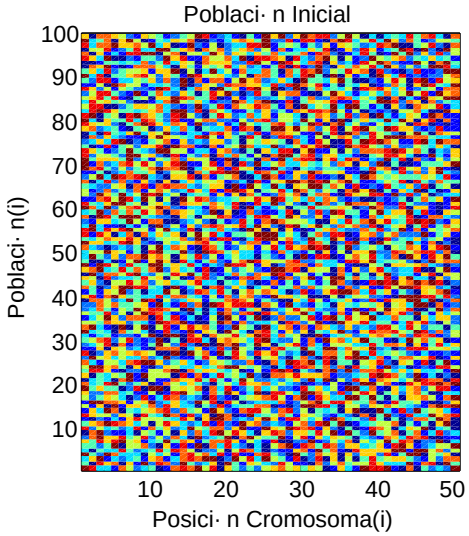
Tiempo de Procesamiento para el problema La01

	Operaciones				
Job	1	2	3	4	5
$j0$	21	53	95	55	34
$j1$	21	52	16	26	71
$j2$	39	98	42	31	12
$j3$	77	55	79	66	77
$j4$	83	34	64	19	37
$j5$	54	43	64	19	37
$j6$	69	77	87	87	93
$j7$	38	60	41	24	83
$j8$	17	49	25	44	98
$j9$	77	79	43	75	96

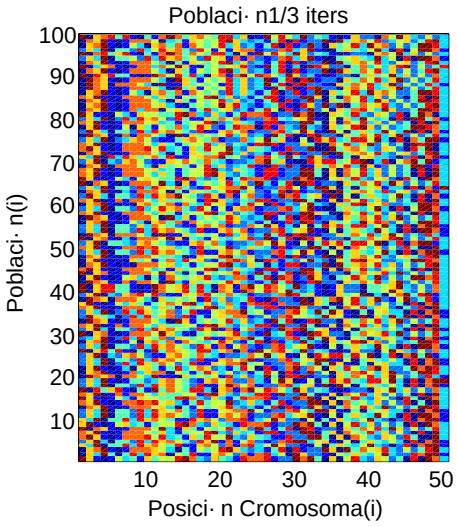
Tabla B.2: Tiempo de procesamiento en cada operación para los trabajos $j1$, $j2$ y $j3$.

B.2 Evolución de la población

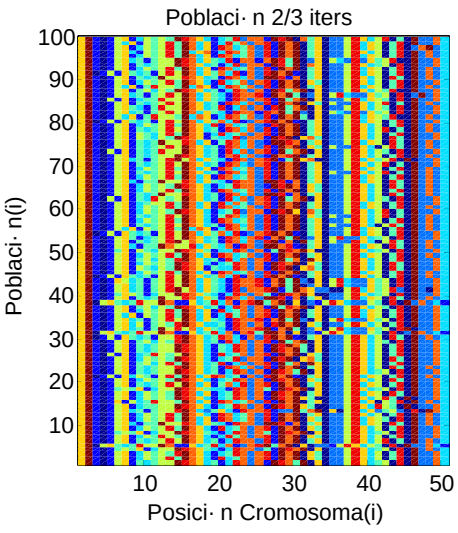
En esta sección se presenta una muestra de ejecución del algoritmo propuesto para la resolución del problema La01. Como puede verse en la figura B.1, se muestran cuatro subfiguras: en la primara de ellas, (a), se representa una instantánea del momento inicial de la población antes de dar comienzo el algoritmo. En la última de las figuras (d), se ha ilustrado el momento final del algoritmo, en una de las filas tiene representada la planificación solución al problema para esa ejecución. En las imágenes intermedias se aprecia una evolución de la población, es característico como existe un estancamiento de la población del instante (c) al (d), en el que no se produce ninguna mejora posiblemente producida porque ya se ha encontrado un valor mínimo.



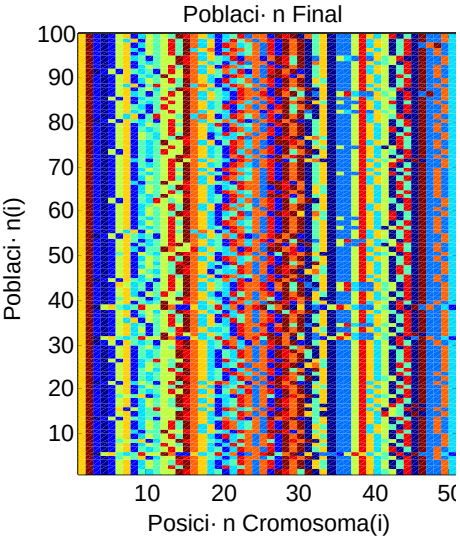
(a) Tabla población del algoritmo en el instante inicial, justo antes de dar comienzo.



(b) Tabla población del algoritmo en el instante 13 de iteraciones.



(c) Tabla población del algoritmo en el instante 23 de iteraciones.



(d) Tabla población del algoritmo en el instante Final.

Figura B.1: Evolución de la población del GA para la resolucion de JSSP.

B.3 Gráfica de evolución del Makespan

En la Fig. B.2 se ha representado un gráfico con la evolución del makespan para la ejecución anterior. En él se muestran cómo va decrementándose el valor del makespan hasta alcanzar la solución, concretamente en el instante que se corresponde con la iteración 450, aproximadamente.

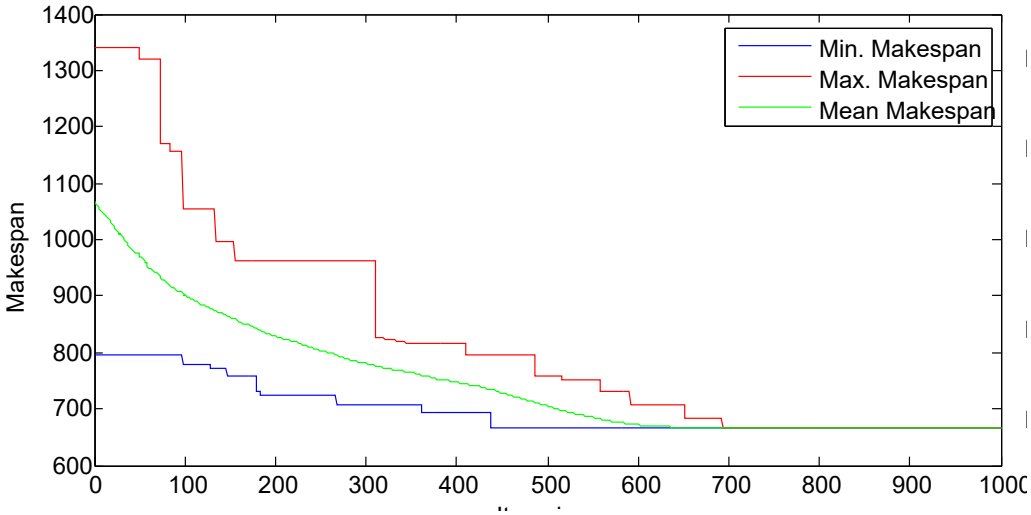


Figura B.2: Grafico de la evolución del Makespan, máximo, mínimo y media.

B.4 Solución encontrada

En la Fig. B.3 se ha representado utilizando un diagrama de Gantt la solución óptima encontrada para el problema definido anteriormente La01.

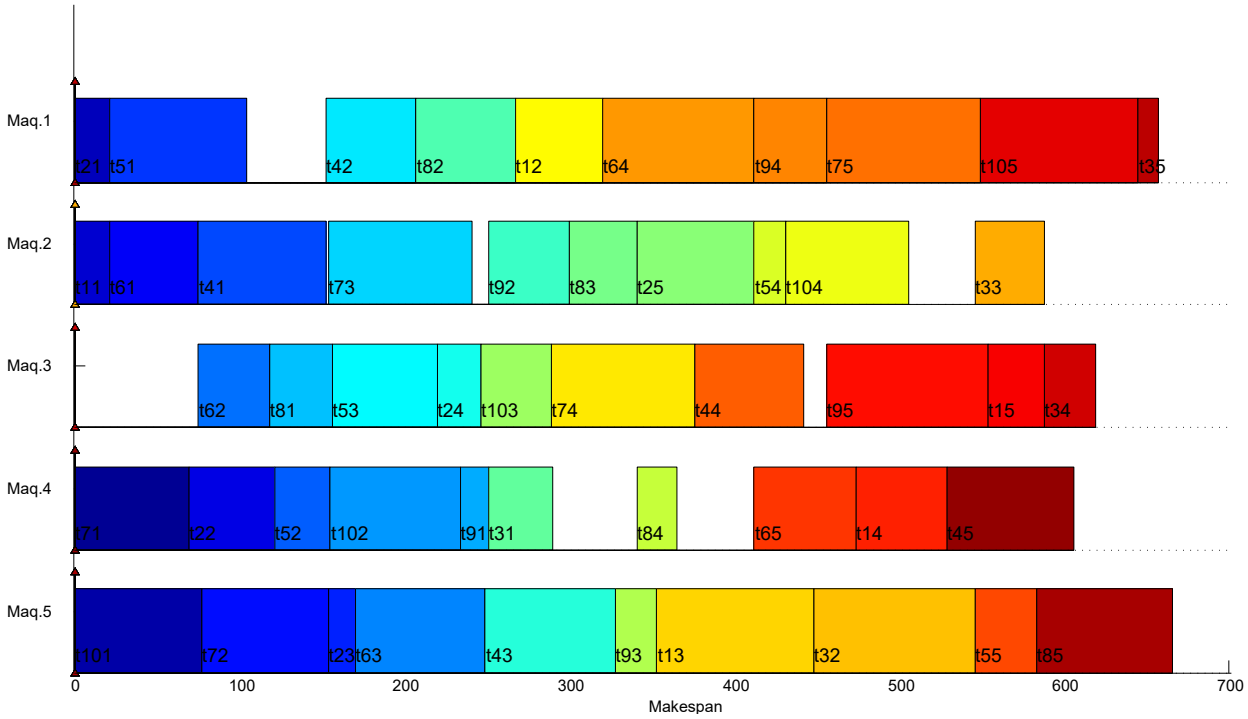


Figura B.3: Planifación óptima encontrada con un valor de makespan igual a 666 unidades de tiempo.

Abreviaturas y nomenclaturas

Abreviaturas

ASIC Application-Specific Integrated Circuit - Circuito Integrado para Aplicaciones Específicas

CORDIC COordinate Rotation DIgital Computing

CPU Central Processing Unit - Unidad central de procesamiento

DSP Digital Signal Processor - Procesamiento Digital de la Señal

Eq. Ecuación

Fig. Figura

FIR Finite Impulse Response

FP Floating Point - Punto Flotante

FPGA Field Programmable Gate Array

GA Genetic Algorithm - Algoritmo Genético

GPU Graphics Processor Unit - Unidad de procesamiento gráfico

HDL Hardware Description Language - Lenguaje de Descripción Hardware

HUB Half-Biased Unit

ILSB Implicit Least Significant Bit - Bit implícito menos significativo

JSSP Job Shop Scheduling Problem

LUT Look Up Table

Máx. Máximo

Mhz. Megahercios

MIMO Multiple-input Multiple-output - Múltiple entrada Múltiple salida

MSB More Significant Bit - Bit más significativo

NP Non-deterministic Polynomial-time - Tiempo polinomial no determinista

QRD Descomposición QR

RNG Random Number Generator- Generador de números aleatorios

Tab. Tabla

Bibliografía

[1] David R. Martinez, Robert A. Bond, and M. Michael Vai, editors. *High performance embedded computing handbook : a systems perspective*. CRC Press, Boca Raton, 2008.

[2] P. H. W. Leong. Recent trends in FPGA architectures and applications. In *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)*, pages 137–141, Jan 2008.

[3] Steve Leibson. Xilinx redefines the high-end microcontroller with its arm-based extensible processing platform-part 1. *Low-PowerDesign. Online May*, 1, 2010.

[4] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *Commun. ACM*, 59(11):114–122, October 2016.

[5] Steven Levy. Google throws open doors to its top-secret data center.

[6] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[7] J. E. Volder. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8(3):330–334, September 1959.

[8] Y. H. Hu. CORDIC-based VLSI architectures for digital signal processing. *IEEE Signal Processing Magazine*, 9(3):13–35, July 1992.

[9] T. Y. Sung. Memory-efficient and high-speed split-radix FFT/IFFT processor based on pipelined CORDIC rotations. *IEE Proceedings ∼ Vision, Image and Signal Processing*, 153(4):405–410, August 2006.

[10] M. C. Mandal, A. S. Dhar, and S. Banerjee. Multiplierless array architecture for computing discrete cosine transform. *Computers and Electrical Engineering*, 21(1):13–19, January 1995.

[11] E. L. Zapata and F. Argüello. A VLSI constant geometry architecture for the fast Hartley and Fourier transforms. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):58–70, January 1992.

[12] H. L. Li and C. Charkrabarti. Hardware design of a 2-D motion estimation system based on hough transform. *IEEE Transactions on Circuits and Systems II*, 45(1):80–95, January 1998.

[13] K. maharatna and S. Banerjee. CORDIC based array architecture for affine transformation of images. In *Proceedings of the International Conference on Communications, Computers and Devices*, volume 2, pages 645–648, Kharagpur, India, December 2000.

[14] J. Chen and K. J. R. Liu. A complete pipelined parallel CORDIC architecture for motion estimation. *IEEE Transactions on Circuits and Systems II*, 45(6):653–660, June 1998.

[15] J. D. Bruguera, N. Guil, T. Lang, J. Villalba, and E. L. Zapata. CORDIC-based parallel/pipelined architecture for Hough transform. *Journal of VLSI Signal Processing*, 12(3):207–221, June 1996.

[16] H. Yoshimura, T. Nakanishi, and Y. Yamaguchi. A 50 MHz CMOS geometrical mapping processor. *IEEE Transactions on Circuits and Systems*, 36(10):1360–1363, October 1989.

[17] T. Lang and E. Antelo. High-throughput CORDIC-based geometry operations for 3D computer graphics. *IEEE Transactions on Computers*, 54(3):347–361, March 2005.

[18] J. Euh, J. Chittamuru, and W. Burleson. CORDIC vector interpolator for power-aware 3D computer graphics. In *IEEE Workshop on Signal Processing Systems, SIPS '02*, pages 240–245, October 2002.

[19] D. C. Kar and V. V. B. Rao. A CORDIC-based unified systolic architecture for sliding window applications of discrete transforms. *IEEE Transactions on Signal Processing*, 44(2):441–444, February 1996.

[20] R. M. Jiang. An area-efficient FFT architecture for OFDM digital video broadcasting. *IEEE Transactions on Consumer Electronics*, 53(4):1322–1326, 2007.

[21] T. C. Chuah. Decoding of low-density parity-check codes in non-Gaussian channels. *IEE Proceedings: Communications*, 152(6):1086–1097, 2005.

[22] L. Boher, R. Rabineau, and M. Hélard. FPGA implementation of an iterative receiver for MIMO-OFDM systems. *IEEE Journal on Selected Areas in Communications*, 26(6):857–866, August 2008.

[23] B. Heyne and J. Götze. CORDIC based algorithms for software defined radio (SDR) baseband processing. *Advances in Radio Science*, 4:179–184, 2006.

[24] Z. Khan, T. Arslan, J. S. Thompson, and A. T. Erdogan. Analysis and implementation of multiple-input, multiple-output VBLAST receiver from area and power efficiency perspective. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(11):1281–1286, November 2006.

[25] H. L. Lin, R. C. Chang, K. H. Lin, and C. C. Hsu. Implementation of synchronization for 2×2 MIMO WLAN system. *IEEE Transactions on Consumer Electronics*, 52(3):766–773, August 2006.

[26] C. S. Peng, Y. S. Chuang, and K. A. Wen. CORDIC-based architecture with channel state information for OFDM baseband receiver. *IEEE Transactions on Consumer Electronics*, 51(2):403–412, May 2005.

[27] A. Troya, K. Maharatna, M. Krsti, E. Grass, U. Jagdhold, and R. Kraemer. Low-power VLSI implementation of the inner receiver for OFDM-based WLAN systems. *IEEE Transactions on Circuits and Systems I*, 55(2):672–686, 2008.

[28] J. Valls, T. Sansaloni, A. Pérez-Pascual, V. Torres, and V. Almenar. The use of CORDIC in software defined radios: A tutorial. *IEEE Communications Magazine*, 44(9):46–50, September 2006.

[29] J. Vankka, M. Kosunen, I. Sanchis, and K. A. I. Halonen. A multicarrier QAM modulator. *IEEE Transactions on Circuits and Systems II*, 47(1):1–10, January 2000.

[30] J. Vankka, J. Ketola, J. Sommarek, O. Väänänen, M. Kosunen, and K. A. I. Halonen. A GSM/EDGE/WCDMA modulator with on-chip D/A converter for base stations. *IEEE Transactions on Circuits and Systems II*, 49(10):645–655, October 2002.

[31] V. Gheorghiu, S. Kameda, T. Takagi, K. Tsubouchi, and F. Adachi. Implementation of frequency domain equalizer for single carrier transmission. In *2008 International Conference on Wireless Communications, Networking and Mobile Compting*, September 2008.

[32] Uday K. Chakraborty. *Computational Intelligence in Flow Shop and Job Shop Scheduling*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[33] KaziShahNawaz Ripon, Chi-Ho Tsang, and Sam Kwong. An evolutionary approach for solving the multi-objective job-shop scheduling problem. In KeshavP. Dahal, KayChen Tan, and PeterI. Cowling, editors, *Evolutionary Scheduling*, volume 49 of *Studies in Computational Intelligence*, pages 165–195. Springer Berlin Heidelberg, 2007.

[34] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278 – 285, 1993. Project Management anf Scheduling.

[35] Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms—i: Representation. *Comput. Ind. Eng.*, 30(4):983–997, sep 1996.

[36] I.M. Yassin, M.N. Taib, H.A. Hassan, A. Zabidi, and N.M. Tahir. Heat exchanger modeling using narx model with binary pso-based structure selection method. In *Computer Applications and Industrial Electronics (ICCAIE), 2010 International Conference on*, pages 368–373, Dec 2010.

[37] V. Smidl and Z. Peroutka. Advantages of square-root extended kalman filter for sensorless control of ac drives. *Industrial Electronics, IEEE Transactions on*, 59(11):4189–4196, Nov 2012.

[38] Qingtang Su, Yugang Niu, Gang Wang, Shaoli Jia, and Jun Yue. Color image blind watermarking scheme based on {QR} decomposition. *Signal Processing*, 94:219 – 235, 2014.

[39] N. Lin, J. Shen, X. Guo, and J. Zhou. A robust image watermarking based on DWT-QR decomposition. In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pages 684–688, May 2011.

[40] Konstantinos Sarrigeorgidis and Jan Rabaey. A scalable configurable architecture for advanced wireless communication algorithms. *Journal of VLSI signal processing systems for signal, image and video technology*, 45(3):127–151, 2006.

[41] Zheng-Yu Huang and Pei-Yun Tsai. Efficient implementation of QR decomposition for gigabit MIMO-OFDM systems. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 58(10):2531–2542, Oct 2011.

[42] S.C. Chan and X.X. Yang. Improved approximate QR-LS algorithms for adaptive filtering. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 51(1):29–39, Jan 2004.

[43] E. Monmasson and M.N. Cirstea. FPGA design methodology for industrial control systems 2014;a review. *Industrial Electronics, IEEE Transactions on*, 54(4):1824–1842, Aug 2007.

[44] D. Sima M. Chen. Fixed-point CORDIC-based QR decomposition by givens rotations on FPGA. pages 327–332, 2011.

[45] S. Aslan, S. Niu, and J. Saniie. FPGA implementation of fast QR decomposition based on givens rotation. In *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, pages 470–473, 2012.

[46] J.E. Guerrero-Ramirez, J. Velasco-Medina, and J.C. Arce-Clavijo. Hardware design of an eigensolver based on the QR method. In *Circuits and Systems (LASCAS), 2013 IEEE Fourth Latin American Symposium on*, pages 1–4, 2013.

[47] Min-Woo Lee, Ji-Hwan Yoon, and Jongsun Park. High-speed tournament givens rotation-based QR decomposition architecture for MIMO receiver. In *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pages 21–24, 2012.

[48] Kuang-Hao Lin, Chih-Hung Lin, R.C.-H. Chang, Chien-Lin Huang, and Feng-Chi Chen. Iterative QR decomposition architecture using the modified gram-schmidt algorithm. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 1409–1412, May 2009.

[49] P. Luethi, A. Burg, S. Haene, D. Perels, N. Felber, and W. Fichtner. VLSI implementation of a high-speed iterative sorted MMSE QR decomposition. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 1421–1424, 2007.

[50] A. Burg. *VLSI Circuits for MIMO Communication Systems*. Series in microelectronics. Hartung-Gorre, Wolfgang, Wissenschaftliche Publikationen, 2006.

[51] F. Sobhanmanesh and S. Nooshabadi. Parametric minimum hardware QR-factoriser architecture for V-BLAST detection. *Circuits, Devices and Systems, IEE Proceedings* -, 153(5):433–441, October 2006.

[52] Sergio D. Muñoz, Javier Hormigo, and Emilio Zapata. FPGA implementation of QR decomposition for medium size matrices. In *Proceedings Conference- XXVIIth Conference on Design of Circuits and Integrated Systems*, pages 50–55, 2012.

[53] P. Biswas, K. Varadarajan, M. Alle, S. K. Nandy, and R. Narayan. Design space exploration of systolic realization of QR factorization on a runtime reconfigurable platform. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 265–272, 2010.

[54] S. Aslan, S. Niu, and J. Saniie. FPGA implementation of fast QR decomposition based on Givens rotation. In *Circuits and Systems, 2012 IEEE 55th International Midwest Symposium on*, pages 470–473, 2012.

[55] M. Abels, T. Wiegand, and S. Paul. Efficient FPGA implementation of a high throughput systolic array QR-decomposition algorithm. In *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, pages 904–908, Nov 2011.

[56] R.C.-H. Chang, Chih-Hung Lin, Kuang-Hao Lin, Chien-Lin Huang, and Feng-Chi Chen. Iterative QR decomposition architecture using the modified Gram-Schmidt algorithm for MIMO systems. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 57(5):1095–1102, May 2010.

[57] Dongdong Chen and M. Sima. Fixed-point CORDIC-based QR decomposition by Givens rotations on FPGA. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 327–332, Nov 2011.

[58] Richard P. Brent and H. T. Kung. Systolic vlsi arrays for polynomial gcd computation. *IEEE Trans. Comput.*, 33(8):731–736, August 1984.

[59] A. El-Amawy and K.R. Dharmarajan. Parallel VLSI algorithm for stable inversion of dense matrices. *Computers and Digital Techniques, IEE Proceedings E*, 136(6):575–580, Nov 1989.

[60] J. Gotze and U. Schwiegelshohn. A square root and division free Givens rotation for solving least squares problems on systolic arrays. *SIAM Journal on Scientific and Statistical Computing*, 12(4):800–807, Jul 1991.

[61] S. D. Muñoz and J. Hormigo. High-throughput FPGA implementation of QR decomposition. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(9):861–865, Sept 2015.

[62] J.W. Luo and C.C. Jong. Scalable linear array architectures for matrix inversion using Bi-z CORDIC. *Microelectronics Journal*, 43(2):141 – 153, 2012.

[63] Miloš D Ercegovac and Tomas Lang. *Digital arithmetic*. Elsevier, 2003.

[64] P. K. Meher, J. Valls, T. B. Juang, K. Sridharan, and K. Maharatna. 50 years of cordic: Algorithms, architectures, and applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 56(9):1893–1907, Sept 2009.

[65] Jean-Michel Muller. *Elementary Functions: Algorithms and Implementation*. Birkhauser Boston, Inc., Secaucus, NJ, USA, 1997.

[66] J. Hormigo and J. Villalba. Measuring improvement when using hub formats to implement floating-point systems under round-to-nearest. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(6):2369–2377, June 2016.

[67] J. Hormigo and J. Villalba. Optimizing DSP circuits by a new family of arithmetic operators. In *2014 48th Asilomar Conference on Signals, Systems and Computers*, pages 871–875, Nov 2014.

[68] J. Villalba-Moreno. Digit recurrence floating-point division under hub format. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 79–86, July 2016.

[69] J. Hormigo and J. Villalba. Hub-floating-point for improving FPGA implementations of DSP applications. *IEEE Transactions on Circuits and Systems II: Express Briefs*, PP(99):1–1, 2016.

[70] J. Hormigo and J. Villalba. Simplified floating-point units for high dynamic range image and video systems. In *2015 International Symposium on Consumer Electronics (ISCE)*, pages 1–2, June 2015.

[71] S. D. Muñoz and J. Hormigo. Improving fixed-point implementation of QR decomposition by rounding-to-nearest. In *2015 International Symposium on Consumer Electronics (ISCE)*, pages 1–2, June 2015.

[72] J. Hormigo and J. Villalba. New formats for computing with real-numbers under round-to-nearest. *IEEE Transactions on Computers*, 65(7):2158–2168, July 2016.

[73] J.J. Rodriguez-Andina, M.J. Moure, and M.D. Valdes. Features, design tools, and application domains of FPGAs. *Industrial Electronics, IEEE Transactions on*, 54(4):1810–1823, Aug 2007.

[74] M. Safar, M.W. El-Kharashi, M. Shalan, and A. Salem. A reconfigurable, pipelined, conflict directed jumping search sat solver. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[75] T. Tachibana, Y. Murata, N. Shibata, K. Yasumoto, and M. Ito. General architecture for hardware implementation of genetic algorithm. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 291–292, April 2006.

[76] I. Mavroidis, I. Papaefstathiou, and D. Pnevmatikatos. A fast FPGA-based 2-opt solver for small-scale euclidean traveling salesman problem. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 13–22, April 2007.

[77] P.V. Santos and J.C. Alves. Fpga based engines for genetic and memetic algorithms. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 251–254, Aug 2010.

[78] T. Ivan and E.M. Aboulhamid. An efficient hardware implementation of a sat problem solver on FPGA. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 209–216, Sept 2013.

[79] Peter Brucker, Bernd Jurisch, and Bernd Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, 49(1–3):107 – 127, 1994. Special Volume Viewpoints on Optimization.

[80] Wang Meilin, Zhang Xiangwei, Dai Qingyun, and He Jinbin. A dynamic schedule methodology for discrete job shop problem based on ant colony optimization. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, pages 306–309, April 2010.

[81] Li-Ning Xing, Ying-Wu Chen, and Ke-Wei Yang. An efficient search method for multi-objective flexible job shop scheduling problems. *Journal of Intelligent Manufacturing*, 20(3):283–293, 2009.

[82] Ferdinando Pezzella and Emanuela Merelli. A tabu search method guided by shifting bottleneck for the job shop scheduling problem. *European Journal of Operational Research*, 120(2):297 – 310, 2000.

[83] M.T. Jensen. Generating robust and flexible job shop schedules using genetic algorithms. *Evolutionary Computation, IEEE Transactions on*, 7(3):275–288, June 2003.

[84] Hongze Qiu, Wanli Zhou, and Hailong Wang. A genetic algorithm-based approach to flexible job-shop scheduling problem. In *Natural Computation, 2009. ICNC '09. Fifth International Conference on*, volume 4, pages 81–85, Aug 2009.

[85] Zhou Yaqin, Li Beizhi, and Wang Lv. Study on job-shop scheduling with multi-objectives based on genetic algorithms. In *Computer Application and System Modeling (ICCASM), 2010 International Conference on*, volume 10, pages V10–294–V10–298, Oct 2010.

[86] Yuan Yuan and Hua Xu. Multiobjective flexible job shop scheduling using memetic algorithms. *Automation Science and Engineering, IEEE Transactions on*, 12(1):336–353, Jan 2015.

[87] Rui Zhang and Cheng Wu. A simulated annealing algorithm based on bottleneck jobs for the open shop scheduling problem. In *Intelligent Control and Automation, 2008. WCICA 2008. 7th World Congress on*, pages 4453–4457, June 2008.

[88] Ye Li and Yan Chen. Neural network and genetic algorithm-based hybrid approach to dynamic job shop scheduling problem. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pages 4836–4841, Oct 2009.

[89] Fei Tao, Lin Zhang, and Yuanjun Laili. Job shop scheduling with FPGA-based F4SA. In *Configurable Intelligent Optimization Algorithm*, Springer Series in Advanced Manufacturing, pages 333–347. Springer International Publishing, 2015.

[90] P.R. Fernando, S. Katkoori, D. Keymeulen, R. Zebulum, and A. Stoica. Customizable FPGA IP core implementation of a general-purpose genetic algorithm engine. *Evolutionary Computation, IEEE Transactions on*, 14(1):133–149, Feb 2010.

[91] G. Andrey and N. Thirer. A FPGA implementation of hardware based accelerator for a generic algorithm. In *Electrical and Electronics Engineers in Israel (IEEEI), 2010 IEEE 26th Convention of*, pages 000578–000580, Nov 2010.

[92] Tatsuhiro Tachibana, Yoshihiro Murata, Naoki Shibata, Keiichi Yasumoto, and Minoru Ito. Flexible implementation of genetic algorithms on FPGAs. In *Proceedings of the 2006 ACMSIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA 06, pages 236–236, New York, NY, USA, 2006. ACM.

[93] D. B. Vernekar, G. Malhotra, and V. Colaco. Reconfigurable FPGA using genetic algorithm. In *Proceedings of the International Conference and Workshop on Emerging Trends in Technology*, ICWET 10, pages 493–497, New York, NY, USA, 2010. ACM.

[94] Christian Bierwirth. A generalized permutation approach to job shop scheduling with genetic algorithms. *Operations-Research-Spektrum*, 17(2-3):87–92, 1995.

[95] Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms, part II: hybrid genetic search strategies. *Computers & Industrial Engineering*, 36(2):343 – 364, 1999.

[96] S. Lawrence. Supplement to Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques. 1984.