



Universidad de Buenos Aires



Diseño e Implementación de un procesador de Descomposición QR
aplicado a filtros de Beamforming

por

Federico Damián Camarda

Tesis presentada para optar al Título de

Ingeniero Electrónico

por la

Facultad de Ingeniería de la Universidad de Buenos Aires

Director:

Ing. Nicolás Alvarez

Co-Director:

Ing. Octavio Alpago

Miembros del Jurado:

Ing. Alberto Dams

Ing. Carlos Belaustegui Goitia

Ing. Leonardo Rey Vega

Calificación: _____

Fecha: _____

Universidad de Buenos Aires, Facultad de Ingeniería

Diseño e Implementación de un procesador de Descomposición QR aplicado a filtros de
Beamforming

por

Federico Damián Camarda

Resumen

El presente trabajo constituye la Tesis de Grado necesaria para obtener el título de Ingeniero Electrónico de la Facultad de Ingeniería de la Universidad de Buenos Aires.

El objetivo de este trabajo es el diseño y la implementación en hardware digital de un procesador de descomposición QR para ser utilizado en el diseño de un filtro para el procesamiento de un *Beamformer* Adaptativo. Se presenta la teoría de base necesaria para poder explicar los desarrollos realizados. Por último, se analizan los resultados obtenidos, se contrastan los mismos contra trabajos similares, y se proponen trabajos futuros.

Agradecimientos

Le quiero agradecer al Ing. Alberto Dams, al Ing. Carlos Belaustegui Goitia, y al Ing. Leonardo Rey Vega, por haber aceptado ser miembros del jurado de la presente tesis de grado. Tuve la oportunidad de asistir a los cursos de algunos de ellos, de los cuales me llevo buenos recuerdos.

Quiero agradecerle mi novia Cecilia, quien fue mi mayor compañera a lo largo de toda mi carrera. Supo entender la gran cantidad de tiempo que le dediqué, me tuvo una gran paciencia y me acompañó y apoyó en los momentos más difíciles, lo cual valoro mucho.

A mi familia, por ayudarme siempre que los necesité. Sin ellos no habría podido completar mi carrera. A mi papá Carlos y a mi mamá Ana principalmente por aconsejarme y guiarme en mis elecciones, y a mis hermanas Laura y Julieta por darme su apoyo.

A mis compañeros de facultad y amigos Matías Stahl, Hernán Fernández Brando, Nicolás Sanguineti y Natan Ottavianoni. Ellos me acompañaron en los primeros años de la carrera, viviendo juntos el gran esfuerzo requerido en la carrera de Ingeniería Electrónica, donde fue necesario dedicar mucho tiempo de estudio. Con ellos logramos armar un excelente grupo y no solo compartir horas de estudio sino también muy buenos momentos. De ellos me llevo una gran amistad. A mis compañeros Fernando Peña, Sergio Hinojosa, Nicolás Ronis, Guillermo Makar, Diego Martín, Leandro Arana, a quienes conocí en los últimos años de mi carrera y con quienes compartí varias horas de trabajo en diferentes proyectos además de muchas charlas en los pasillos de la facultad. Me llevo de todos ellos los mejores recuerdos.

Agradezco a mi director Nicolás Alvarez, co-director Octavio Alpago, y al Colo Federico Giordano Zacchigna, por haberme guiado y ayudado en el desarrollo de la presente tesis. Siempre que los necesité estuvieron ahí, y compartieron conmigo su experiencia en un ámbito completamente nuevo para mí.

Quiero agradecer a Ariel Lutemberg por el apoyo que me dió al ingresar al Laboratorio de Sistemas Embebidos y también a todo el equipo que lo conforma. En el momento que decidí tocar la puerta para trabajar con ellos fueron más que amables. Logré encontrar un excelente lugar para trabajar y hacer consultas en un ambiente muy ameno.

Agradezco a los profesores Enrique Jorge Velo, Martín Cardozo, Gustavo Bongiovanni,

Ariel Burman, Leandro Santi, Carlos Belaustegui, Federico Roasio y Ezequiel Espósito, siendo algunos de ellos quienes tuvieron gran importancia en mis elecciones a lo largo de la carrera. Compartieron conmigo sus conocimientos y experiencias y tienen mi mayor admiración. Encontré en sus cursos clases muy entretenidas y didácticas, de las cuales me quedan grandes enseñanzas. Gracias a todos ellos me llevo los mejores recuerdos de la Facultad de Ingeniería de la Universidad de Buenos Aires.

Y por último les doy las gracias a mis colegas de trabajo y amigos Gerónimo Acevedo, Tomás Lopez, Nicolás Barbalán, Guillermo Preciado, Emiliano Camarotta y Carlos Vitrano. Ellos estuvieron presentes en los últimos años de mi carrera, con quienes compartí juntadas y muy buenos momentos que sirvieron para recargar energía y mirar siempre hacia adelante.

A mi familia

Índice

1. Introducción al trabajo de tesis	1
1.1. Objetivo	2
1.2. Alcance	2
1.3. Organización del trabajo	3
2. Teoría de los sistemas MIMO	5
2.1. Antenas	6
2.1.1. Patrón de radiación de una antena	7
2.1.2. Antenas direccionales	8
2.1.3. Arreglos lineales	9
2.1.4. Arreglos dirigidos electrónicamente	12
2.1.5. Parámetros básicos de arreglos de antenas	13
2.2. <i>Multipath fading</i>	14
2.3. <i>Beamforming</i>	16
2.3.1. <i>Beamforming</i> digital	16
2.3.2. <i>Beamforming</i> adaptativo	18
2.3.3. Criterios para obtener los pesos óptimos	20
2.4. Algoritmos adaptativos	24
2.4.1. Algoritmo LMS - <i>Least Mean Squares</i>	24
2.4.2. Algoritmo RLS - <i>Recursive Least Squares</i>	25
2.4.3. Adquisición de la señal de referencia	27

2.5.	Otras técnicas de diseño MIMO	28
2.5.1.	<i>Space-Time block coding</i>	28
2.5.2.	<i>Spatial multiplexing</i>	30
3.	Arquitecturas digitales para la implementación de sistemas MIMO	33
3.1.	Resolución del algoritmo RLS	34
3.2.	Descomposición QR	34
3.3.	Del algoritmo a una arquitectura sintetizable	39
3.3.1.	Gráfico de dependencias	40
3.3.2.	Gráfico de flujo de señal	41
3.4.	Implementación sistólica de las rotaciones de givens	42
3.4.1.	Mapeo de Gentleman y Kung	42
3.4.2.	Mapeo de proyección horizontal	43
3.4.3.	Mapeo de Rader	44
3.4.4.	Mapeo de Walke	45
3.4.5.	Planificación de las operaciones QR	48
3.5.	Implementación de las operaciones BC e IC	49
3.5.1.	El algoritmo CORDIC	49
3.6.	Análisis de publicaciones	50
3.6.1.	Publicación de Altera	50
3.6.2.	Publicación de Xilinx	51
3.6.3.	Publicación de Universidad de Victoria	52
3.7.	Arquitectura implementada	53
4.	Implementación a nivel de microarquitectura	55
4.1.	Diagrama en bloques	56
4.2.	Parámetros comunes a los distintos módulos	57
4.3.	Pre-procesamiento de entradas para el módulo CORDIC	57
4.4.	CORDIC iterativo	59

4.5.	Post-multiplicador de salidas x e y de CORDIC	61
4.6.	Unidad de rotación completa	62
4.7.	Procesador boundary cell	63
4.8.	Procesador internal cell	63
4.9.	Procesador de descomposición QR	63
4.9.1.	Interfaz del módulo	64
4.9.2.	Uso del procesador QR	65
4.9.3.	Unidad de registros	67
4.9.4.	Unidad de control	69
4.10.	Herramientas utilizadas	71
5.	Estudio, simulación y validación del IP core generado	73
5.1.	Emulador de hardware en lenguaje C	73
5.2.	Simulación de módulos en Modelsim	77
5.3.	Interfaz del procesador de descomposición QR	78
5.4.	<i>Testbench</i> desarrollado en Verilog	80
5.5.	Síntesis de hardware	84
5.6.	<i>Testbench</i> en lenguaje C	86
5.7.	Correcciones sobre el hardware implementado	89
5.7.1.	Algoritmo QR-RLS	89
5.7.2.	Factor de olvido en el procesador de descomposición QR	90
5.8.	Resultados	92
5.8.1.	Integración del hardware con Octave para la estimación de un sistema	92
5.8.2.	Ánalisis del error en la estimación de múltiples sistemas	94
5.8.3.	Precisión del sistema	96
5.8.4.	Parámetros de Timing	101
5.8.5.	Recursos de FPGA utilizados	107

6. Conclusiones y trabajos a futuro	111
A. Código fuente Verilog del procesador QR desarrollado	115
A.1. Fuente qr_processor.v	115
A.2. Fuente boudary_cell.v	131
A.3. Fuente internal_cell.v	134
A.4. Fuente rotator.v	137
A.5. Fuente iterative_cordic.v	142
A.6. Fuente preprocessor.v	147
A.7. Fuente post_multiplier.v	150
B. Script Octave de análisis de resultados	153
C. El algoritmo CORDIC	159
C.1. Convergencia del algoritmo CORDIC	164
C.1.1. Convergencia en norma	165
C.1.2. Convergencia en fase	167
Acrónimos	171

Índice de Tablas

2.1.	Ejemplos de bandas de frecuencias y sus longitudes de onda	7
4.1.	Tabla de Arcotangentes	60
5.1.	Tabla comparativa de dispositivos FPGA Xilinx	85
5.2.	Ciclos requeridos por cada estado de procesamiento	102
5.3.	Resultados de <i>timing</i> para distintos dispositivos FPGA	103
5.4.	Máximo <i>clock</i> de operación	107

Índice de Figuras

2.1.	Esquema de un sistema de comunicación MIMO	5
2.2.	Representación de los distintos sistemas de múltiples antenas	6
2.3.	Dipolo de media onda - Intensidad de Campo vs Dirección	8
2.4.	Dipolo de media onda - Esquema	8
2.5.	Antena Parabólica - Patrón de Radiación	9
2.6.	Antena Parabólica - Diagrama	9
2.7.	Arreglo Lineal $D = \lambda/2$ - Patrón de Radiación	10
2.8.	Arreglo Lineal $D = \lambda/2$ - Esquema	10
2.9.	Arreglo Lineal $D = \lambda$ - Patrón de Radiación	11
2.10.	Arreglo Lineal $D = \lambda/2$ - 7 elementos - Patrón de Radiación	11
2.11.	Arreglo Lineal $D = \lambda/2$ - 7 elementos - Esquema	11
2.12.	Arreglo dirigido electrónicamente - Patrón de Radiación	12
2.13.	Arreglo dirigido electrónicamente - Esquema	12
2.14.	Múltiples Reflexiones de la señal transmitida	15
2.15.	Esquema de un sistema de <i>Beamforming</i>	16
2.16.	Esquema de un sistema de <i>Beamforming</i> Adaptativo	18
2.17.	Arreglo de dos elementos para la supresión de interferencia	19
2.18.	Representación gráfica del flujo de señal para el algoritmo LMS	25
2.19.	Representación gráfica del flujo de señal para el algoritmo RLS	27
2.20.	Una configuración genérica para <i>beamforming</i> adaptativo en un sistema de comunicación wireless CDMA	28

2.21.	Ganancia de diversidad espacio-tiempo en sistemas MIMO	29
2.22.	Esquema de un transmisor de Alamouti	29
2.23.	Spatial Multiplexing con tres antenas en transmisor y receptor	30
2.24.	Tipos de codificación para Spatial Multiplexing	31
3.1.	Gráfico de dependencias en alto nivel para la solución QR-RLS	36
3.2.	Transformación por Rotaciones de Givens	37
3.3.	Descomposición QR aplicada a una matriz de 3x3	38
3.4.	Proceso de desarrollo del circuito digital	39
3.5.	Gráfico de Dependencias y Gráfico de Flujo de Señal	40
3.6.	Esquema de mapeo directo	42
3.7.	Boundary Cell	43
3.8.	Internal Cell	43
3.9.	Esquema de Mapeo de Proyección Horizontal	44
3.10.	Esquema de Mapeo de Rader	44
3.11.	Esquema de Mapeo de Walke - Primera Transformación	45
3.12.	Esquema de Mapeo de Walke - Segunda Transformación	46
3.13.	Esquema de Mapeo de Walke - Tercera Transformación	47
3.14.	Esquema de Mapeo de Walke - Cuarta Transformación	47
3.15.	Mapeo de Walke - Iteraciones	48
3.16.	Tabla de métricas del hardware de Altera	50
3.17.	Arquitectura implementada por Xilinx	52
3.18.	Tabla de métricas del hardware presentado por Xilinx	52
3.19.	Arreglo de descomposición QR para una matriz de 4×4	53
3.20.	Tabla de métricas del hardware presentado	54
4.1.	Diagrama en Bloques del Hardware Desarrollado	56
4.2.	Diagrama RTL de la implementación de CORDIC en hardware	61
4.3.	Diagrama en Bloques del Módulo de Rotación	62

4.4.	Diagrama de Tiempos en el uso del Hardware	65
4.5.	Esquema de los primeros 8 ciclos de procesamiento	66
4.6.	Esquema de registros utilizados	67
4.7.	Detección de nueva entrada y estado inicial	69
4.8.	Resultado del desplazamiento hacia abajo	70
4.9.	Resultado de la copia	70
5.1.	Extracto de la función de emulación <i>QR Decomposition</i>	74
5.2.	Matriz de referencia elegida para las pruebas	75
5.3.	Resultado provisto por Bluebit	75
5.4.	Resultado provisto por el emulador QR	76
5.5.	Resultados de la simulación del procesador	77
5.6.	Diagrama en bloques de la interfaz de integración qr_eval	78
5.7.	Diagrama en estados simplificado de la unidad de control utilizada en la interfaz	79
5.8.	Resultados de la simulación de la interfaz del procesador	80
5.9.	Ejecución de una simulación del cálculo de una matriz en el procesador	82
5.10.	Fin de la simulación del cálculo de una matriz en el procesador	83
5.11.	Relación de rendimiento y funcionalidades en dispositivos FPGA Xilinx	84
5.12.	Kit de desarrollo de Spartan 3E	84
5.13.	Resultado de la síntesis en la herramienta Xilinx ISE	85
5.14.	Proceso de programación del archivo de síntesis en el dispositivo FPGA	86
5.15.	Testbench: Macbook y el kit de desarrollo con el hardware sintetizado calculando	88
5.16.	Máximo valor de la matriz R en función del número de iteración, $\lambda = 1$	90
5.17.	Máximo valor de la matriz R en función del número de iteración, $\lambda = 0,95$	91
5.18.	Señal de salida del sistema original y señal de salida del sistema estimado por el hardware	93
5.19.	Señal de error basada en la función de costo utilizando el hardware	93

5.20.	Pesos originales vs Pesos estimados utilizando el hardware	94
5.21.	Histograma de la métrica E_1	95
5.22.	Histograma de la métrica E_2	95
5.23.	Salidas ideal y estimada superpuestas	99
5.24.	Valor medio de la matriz de error	100
5.25.	Valor medio de la matriz de error relativo al máximo número representable	100
5.26.	Valor medio de la matriz de error en porcentaje	101
5.27.	Máximo <i>clock</i> de operación para distintos dispositivos FPGA	104
5.28.	<i>Throughput</i> para distintos dispositivos FPGA	104
5.29.	<i>Update delay</i> para distintos dispositivos FPGA	105
5.30.	Latencia para distintos dispositivos FPGA	105
5.31.	Número de <i>lookup tables</i> utilizadas	107
5.32.	Número de <i>flip flops</i> utilizados	108
5.33.	Número de <i>slices</i> utilizados	108
C.1.	Rotación de un vector en un ángulo ϕ	159
C.2.	$\Delta\phi$ en función de la cantidad de iteraciones n	162
C.3.	Pseudorotación producida por la i -ésima iteración.	164

Capítulo 1

Introducción al trabajo de tesis

El presente trabajo se encuentra enfocado en el contexto del diseño de hardware digital. El mismo fue motivado por el creciente avance en las nuevas tecnologías implementadas en los sistemas de comunicaciones digitales y en los sistemas *Software Defined Radio*¹. Los avances en la escala de integración de circuitos integrados ante la constante reducción de las líneas de conducción en los procesos de tratamiento del silicio y el aumento de la frecuencia de operación, hacen que hoy en día sea posible implementar sistemas que en el pasado eran sólo desarrollos teóricos.

En los últimos años, celulares, redes 4G, puntos de acceso de redes inalámbricas tales como Wi-Fi y WiMAX, se encuentran haciendo uso de la tecnología MIMO². Una de las técnicas para implementar MIMO es el *beamforming* adaptativo, que consiste en la capacidad de separar señales en el dominio del espacio. Este hecho provee un medio para separar la señal deseada de señales de interferencia.

Un *beamformer* adaptativo logra optimizar automáticamente el patrón de un arreglo de múltiples antenas al ajustar el control de los pesos de ponderación hasta que se satisface una determinada función de objetivo preestablecida. Los medios por los cuales esta optimización es lograda están especificados por un algoritmo diseñado para tal propósito. Estos dispositivos utilizan mucha más información disponible en la antena con respecto a un *beamformer* convencional.

El núcleo de un *beamformer* utiliza un sistema de descomposición de matrices para lograr obtener las constantes de ponderación de las señales recibidas en cada antena, siendo la descomposición QR la más utilizada y sobre la cual se basa el presente trabajo.

¹ *Software Defined Radio*: Sistema de comunicaciones donde los componentes típicamente implementados en hardware (mezcladores, filtros, amplificadores, moduladores / demoduladores, detectores, etc) son implementados en software

² *Multiple-input Multiple-output*: Consiste en un sistema de múltiples entradas y salidas, el cual aprovecha fenómenos físicos como la propagación multicamino para incrementar la tasa de transmisión y reducir la tasa de error

1.1. Objetivo

La Tesis tiene como objetivo principal diseñar e implementar, en hardware digital, un procesador capaz de resolver un algoritmo utilizado para la descomposición de matrices conocido como RLS-QRD (*Recursive Least Squares - QR Decomposition*). Dicho diseño será realizado a través del uso del lenguaje de descripción de hardware Verilog, y la implementación se obtendrá a través de su síntesis en un dispositivo FPGA ³.

El enfoque de la tesis se basará en una parte teórica y una experimental. Se analizarán los modelos y desarrollos algorítmicos para la implementación de una descomposición QR y luego se realizará un análisis para el diseño de un IP core que ponga en práctica uno de ellos. Una vez codificado el hardware, se procederá a realizar su síntesis en un dispositivo FPGA utilizando un kit de desarrollo, se realizarán los bancos de prueba correspondientes y se analizarán los resultados obtenidos.

El procesador a implementar será sometido a diferentes pruebas con el objetivo de definir parámetros tales como el error, máxima frecuencia de operación, cantidad de operaciones por segundo y consumo de potencia.

1.2. Alcance

Como resultados a obtener de la presente tesis se tienen los siguientes:

- IP Core codificado en el lenguaje Verilog de un procesador de descomposición QR.
- Resultado de mediciones pertinentes al diseño del procesador.
- Análisis comparativo de procesamiento entre el procesador desarrollado y desarrollos de terceros.
- Proposición de trabajos futuros y/o mejoras.

³ *Field Programmable Gate Array*: dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada “in situ” mediante un lenguaje de descripción especializado.

1.3. Organización del trabajo

En esta sección se describe la organización de la presente tesis. Con el objetivo de que la misma sea mínimamente autocontenido, los primeros capítulos se ocupan de presentar las bases o conocimientos necesarios para comprender la totalidad del trabajo.

El desarrollo de la tesis se organiza de la siguiente forma:

- En el capítulo 2 se hará referencia a los conceptos requeridos para comprender la teoría de los sistemas MIMO. Se describirá el contenido teórico requerido para exponer los conceptos de funcionamiento del hardware a desarrollar, el cual se basa en antenas, *beamforming*, criterios para el diseño del filtro adaptativo y algoritmos.
- En el capítulo 3 se analizarán las diferentes arquitecturas digitales propuestas para implementar el sistema. Se elegirá una de ellas para realizar el desarrollo en base a un determinado criterio.
- En el capítulo 4 se hará referencia a la implementación de la arquitectura seleccionada en Verilog y se desarrollarán los bancos de prueba de simulación para verificar su correcta funcionalidad. Se generará un IP core en RTL para implementar un sistema MIMO. Dicho RTL cumplirá con ciertas condiciones de portabilidad y legibilidad del código, para que el mismo sea efectivamente un IP core.
- En el capítulo 5 se experimentará el IP core en un ambiente de simulación y en campo. Se realizará la síntesis del mismo para distintos dispositivos FPGA, y se medirán los recursos utilizados, la máxima frecuencia de operación y la potencia consumida.
- En el capítulo 6 se extraerán las conclusiones pertinentes sobre los resultados obtenidos y se propondrán futuras mejoras de la arquitecturas a partir del análisis realizado.

Capítulo 2

Teoría de los sistemas MIMO

En términos generales, se puede definir a un sistema MIMO (multiple input multiple output) como un sistema caracterizado por poseer múltiples entradas y salidas. Debido a la abstracción de esta definición, la misma puede ajustarse a distintos campos de estudio. El tema que se abordará será el estudio de sistemas MIMO en las radiocomunicaciones, en las cuales se trabajará con señales que provienen de ondas electromagnéticas.

En este caso particular de comunicaciones, los sistemas MIMO se refieren a sistemas que poseen múltiples antenas para captar señales en la entrada y múltiples antenas para transmitir las señales de salida.

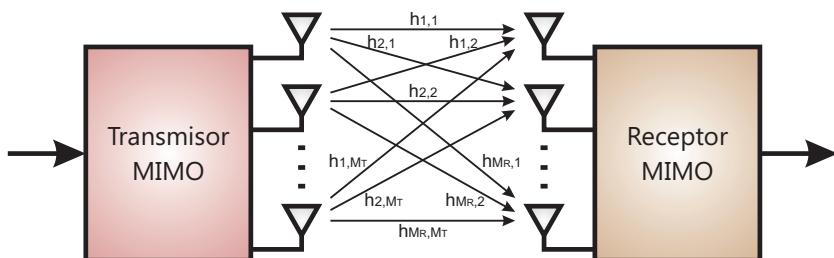


Figura 2.1: Esquema de un sistema de comunicación MIMO

Una comunicación inalámbrica se compone por tres elementos. Un transmisor, un receptor, y el canal, en el cual se propagan las señales a transmitir/recibir [1]. Existe una clasificación para los distintos tipos de sistemas MIMO en función de la cantidad de antenas que poseen en los lados emisor y receptor. MIMO propiamente dicho, se refiere a un sistema en el cual se tiene más de una única antena tanto en el receptor como en el emisor del sistema. Los sistemas en los cuales se tiene una única antena en el receptor y múltiples antenas en el transmisor son conocidos como MISO (multiple input single output). Por otro lado, los sistemas SIMO (single input multiple output), son aquellos en los cuales se tiene una única antena en el lado emisor y múltiples antenas en el lado receptor. SISO (single input single output) es el caso en el cual se tiene una única antena

tanto en la entrada como en la salida.

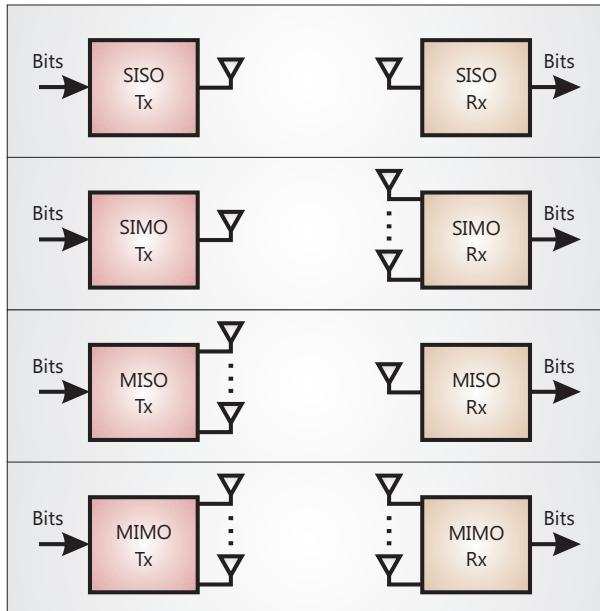


Figura 2.2: Representación de los distintos sistemas de múltiples antenas

El concepto de utilizar múltiples antenas en un receptor de comunicaciones inalámbricas surgió en 1960. La idea escencial consistía en la provisión de múltiples copias de una señal transmitida en el lado receptor y sus combinaciones para obtener una señal de mejor desempeño [1]. Ésta técnica, es conocida como **diversidad espacial**. Para lograr la técnica de diversidad espacial, se emplean múltiples antenas en el lado receptor. Las señales de las antenas de salida son luego combinadas aplicando distintos coeficientes de ponderación. Por otro lado, el estudio de la implementación de diversidad a través de múltiples antenas en el lado transmisor fue introducido en 1990 [2].

A través de la utilización de múltiples antenas es posible lograr una mayor eficiencia en distintas características de un sistema de comunicación, siendo las más importantes el aumento en la capacidad del canal o en la relación señal ruido. También es posible resolver conflictos vinculados a distintos fenómenos asociados con una comunicación inalámbrica. Uno de estos fenómenos es el conocido como **“Multipath Fading”**. Antes de abordar dicho fenómeno se hará una breve reseña de algunos conceptos sobre antenas.

2.1. Antenas

Una antena es un dispositivo eléctrico que convierte potencia eléctrica en ondas de radio, y viceversa. Usualmente, es utilizada con un radio transmisor o radio receptor. En una transmisión, el transmisor de radio provee una corriente eléctrica oscilante de

radio frecuencia a las terminales de la antena, y la misma irradia la energía proveniente de dicha corriente como ondas electromagnéticas (ondas de radio). En la recepción, una antena intercepta parte de la potencia de una onda electromagnética con el objetivo de producir un voltaje muy pequeño en sus terminales, el cual es aplicado al receptor para luego ser amplificado [7].

La antena consiste de conductores eléctricos (cables, tubos o superficies reflectantes) que crean campos eléctricos y magnéticos en el espacio alrededor de los mismos. [4]. Si los campos son variantes, se propagan hacia el espacio como ondas electromagnéticas a la velocidad de la luz:

$$\text{Velocidad de la Luz} \quad c \approx 3 \cdot 10^8 \frac{\text{metros}}{\text{seg}} \quad (2.1)$$

Toda antena que transmite también recibe. Las ondas electromagnéticas que atraviesan la misma excitan corrientes en los conductores de la antena. La antena captura parte de la energía de las ondas que la atraviesan y la convierte en señales eléctricas en el cable.

Al realizarse el diseño de una antena, sus dimensiones son especificadas en términos de la longitud de onda de las señales de radio que serán transmitidas o recibidas. La longitud de onda es la distancia desde el inicio de un ciclo electromagnético hasta el próximo.

$$\lambda = \frac{c}{f_c} \quad (2.2)$$

λ es la longitud de onda en metros y f_c es la frecuencia portadora de señal de radio en Hz . c es la velocidad de la luz ($3 \cdot 10^8 \text{metros/seg}$).

Señal	Frecuencia	Longitud de Onda
Radio AM	$1MHz$	300 metros
Radio FM	$100MHz$	3 metros
Teléfono Celular	$850MHz$	35 centímetros
Access Point Wi-Fi	$2,4GHz$	12,5 centímetros

Tabla 2.1: Ejemplos de bandas de frecuencias y sus longitudes de onda

2.1.1. Patrón de radiación de una antena

Una antena transmisora genera ondas electromagnéticas más fuertes en algunas direcciones con respecto a otras. Para analizar este comportamiento se obtiene un diagrama de intensidad del campo en función de la dirección, el cual es llamado “Patrón de radiación de la antena”. Siempre es igual para la recepción y para la transmisión.

Una onda electromagnética medida en un punto lejos de la antena es la suma de la radiación de todas las partes de la antena. Cada parte de la misma, irradia ondas de diferentes amplitudes y fases, y cada una de esos ondas viaja distintas distancias hasta el punto donde se encuentra el receptor. En algunas direcciones, estas ondas se suman constructivamente para dar una ganancia. En otras, se suman destrutivamente para dar una pérdida.

Un dipolo de media onda es una antena simple que consiste de media longitud de onda de cable, cortado en el centro para realizar la conexión. En la siguiente figura se muestra su patrón de radiación:

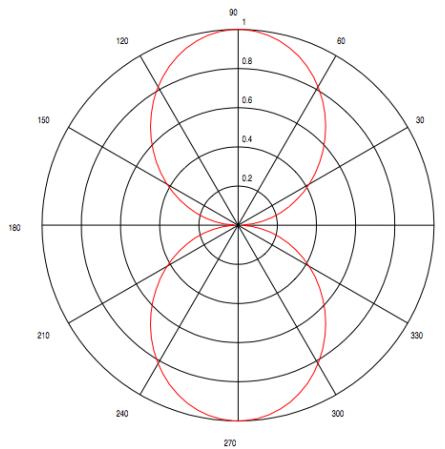


Figura 2.3: Dipolo de media onda - Intensidad de Campo vs Dirección

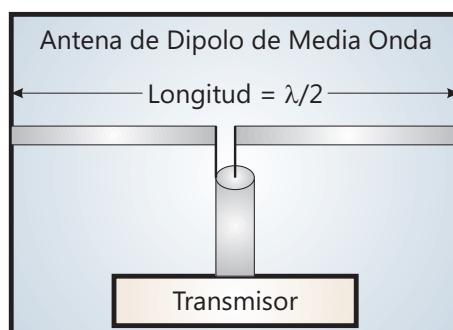


Figura 2.4: Dipolo de media onda - Esquema

2.1.2. Antenas direccionales

Un ejemplo de antena direccional es una antena diseñada para tener ganancia en una dirección y pérdida en otras. Una antena se vuelve direccional al aumentar su tamaño. Al hacer esto, se amplian los conductores radiantes de la antena para cubrir una dis-

tancia mayor, de forma tal que las interferencias constructivas y destructivas pueden ser controladas de mejor forma para dar un patrón de radiación direccional.

Una antena de plato satelital puede ser considerada, simplisticamente, una superficie circular que irradia ondas electromagnéticas igualmente desde todas partes. Tiene un haz central angosto de alta ganancia, como se muestra en la siguiente figura, que se apunta al satélite.

A medida que se incrementa el diámetro del plato en longitudes de onda, el haz central se vuelve más angosto. Notar los lóbulos pequeños, llamados “lóbulos laterales”, a cada lado del haz central. Las direcciones en las cuales la intensidad de señal es cero son llamadas ‘nulls’.

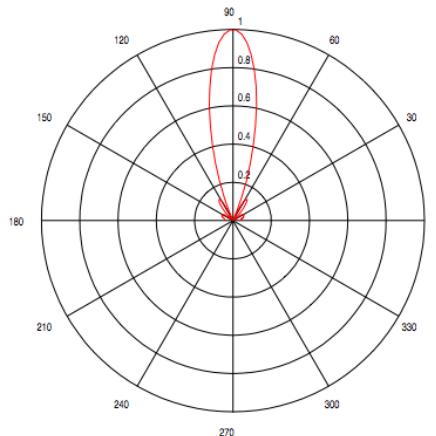


Figura 2.5: Antena Parabólica - Patrón de Radiación

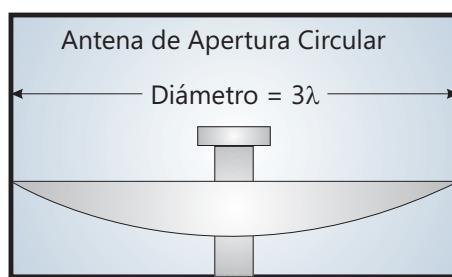


Figura 2.6: Antena Parabólica - Diagrama

2.1.3. Arreglos lineales

Una antena direccional simple consiste de un arreglo lineal de pequeños elementos radiantes de antena, cada uno alimentado con señales idénticas (misma amplitud y fase) desde un transmisor. A medida que el ancho total del arreglo crece, el haz central se

vuelve más angosto. A medida que el número de elementos se incrementa, los lóbulos laterales se vuelven más pequeños.

La siguiente figura es el patrón de radiación para una línea de 4 elementos (antenas pequeñas) espaciadas $1/2$ longitud de onda.

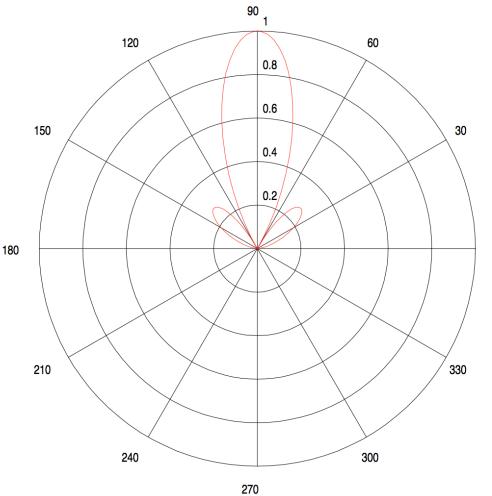


Figura 2.7: Arreglo Lineal $D = \lambda/2$ - Patrón de Radiación

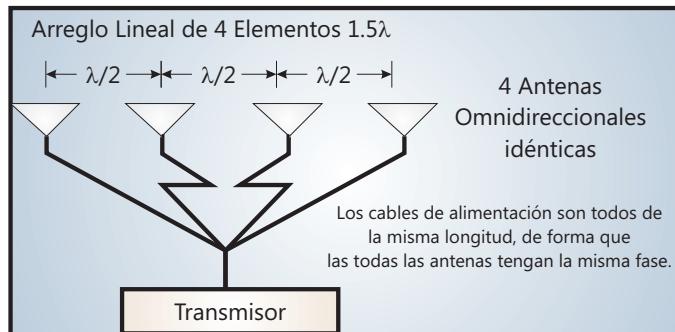
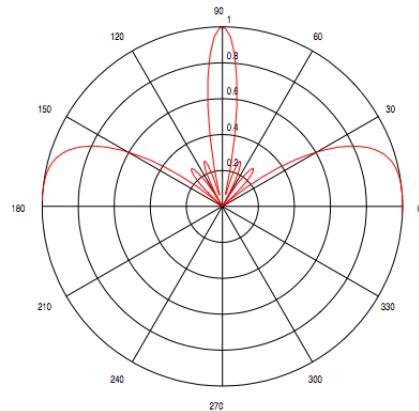
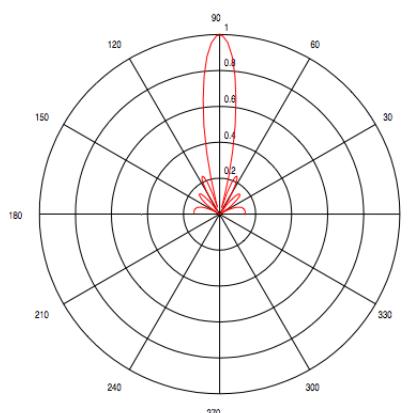
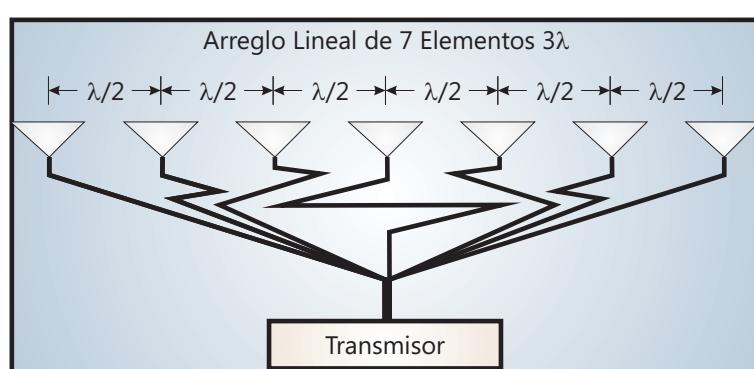


Figura 2.8: Arreglo Lineal $D = \lambda/2$ - Esquema

Si el espaciado es incrementado a más de $1/2$ longitud de onda, empiezan a aparecer lóbulos laterales más largos en el patrón de radiación. De todas formas, el haz central se vuelve más angosto debido a que la longitud total de la antena es incrementada. A continuación se ilustra el siguiente patrón de radiación, para 4 elementos espaciados 1 longitud de onda:

Al mantener la longitud total de la antena igual, y agregar elementos para reducir el espaciado nuevamente a $1/2$ longitud de onda, los lóbulos laterales se reducen. A continuación se muestra el patrón de radiación si se agregan 3 elementos más a la antena anterior para reducir el espaciado entre elementos.

Figura 2.9: Arreglo Lineal $D = \lambda$ - Patrón de RadiaciónFigura 2.10: Arreglo Lineal $D = \lambda/2$ - 7 elementos - Patrón de RadiaciónFigura 2.11: Arreglo Lineal $D = \lambda/2$ - 7 elementos - Esquema

2.1.4. Arreglos dirigidos electrónicamente

El haz central del arreglo lineal puede ser direccionado al variar las fases de las señales en los elementos del mismo. La forma más simple para controlar la fase de las señales es variar sistemáticamente las longitudes de los cables hacia los elementos. El cable retrasa la señal, por lo cual desplaza la fase. De todas formas, esto no permite que la antena sea dirigida dinámicamente.

En un arreglo electrónicamente dirigido, se utilizan desplazadores de fase electrónicos programados en cada elemento del arreglo. La antena es dirigida al programar el valor de desplazamiento de fase requerido para cada elemento. El patrón del haz a continuación es para un arreglo lineal de 8 elementos con un desplazamiento de fase progresivo de $0,7\pi$ por elemento. El haz central fue dirigido 45 grados a la izquierda. Un desplazamiento de fase de 2π corresponde a una longitud de onda o un periodo de onda de portadora, y valores más positivos son equivalentes a decir que la señal es transmitida más tempranamente.

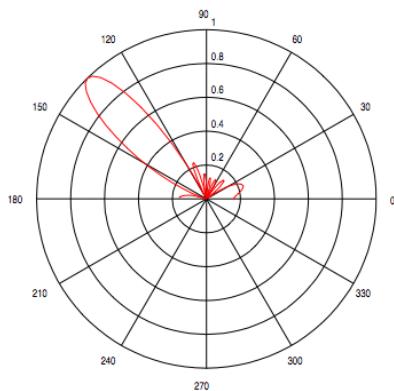


Figura 2.12: Arreglo dirigido electrónicamente - Patrón de Radiación

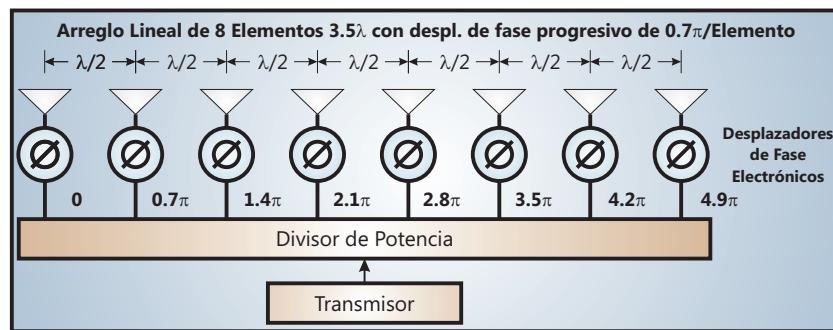


Figura 2.13: Arreglo dirigido electrónicamente - Esquema

2.1.5. Parámetros básicos de arreglos de antenas

A continuación se explicarán algunos parámetros y definiciones básicas que serán de interés relevante para los conceptos tratados en la presente tesis [5]. A pesar de que varios de ellos son definidos en términos de antenas transmisoras, la reciprocidad asegura que estas definiciones sean también aplicables a antenas receptoras.

Patrón de radiación

Se refiere a la distribución relativa de potencia irradiada como función de la dirección en el espacio.

Factor del arreglo

El factor del arreglo representa el patrón de radiación a campo lejano de un arreglo isotrópico de elementos radiantes. El factor del arreglo se denotará a lo largo de la tesis como $F(\phi, \theta)$, donde ϕ representa el ángulo de azimuth y θ representa el ángulo de elevación en el espacio.

Lóbulo principal

El lóbulo principal de un patrón de radiación de una antena es el lóbulo que contiene la dirección de máxima radiación de potencia.

Lóbulos laterales

Los lóbulos laterales son lóbulos en cualquier dirección distinta de aquella del lóbulo principal. Para un arreglo lineal con ponderación uniforme, el primer lóbulo lateral (el cual es el más cercano al lóbulo principal) en el patrón de radiación se encuentra 13dB por debajo del valor pico del lóbulo principal.

Ancho del haz (Beamwidth)

El ancho del haz de una antena es el ancho angular del lóbulo principal en su patrón de radiación a campo lejano. El HPBW (Ancho del haz a mitad de potencia), o ancho del haz a $-3dB$, es el ancho del haz medido entre los puntos del lóbulo principal que se encuentran a $-3dB$ del valor pico del lóbulo principal:

$$HPBW = \frac{0,88\lambda}{A} \quad (2.3)$$

Donde A representa la longitud de la apertura del arreglo.

Eficiencia de la antena

La eficiencia de la antena se define como la relación entre la potencia total irradiada por la antena y la potencia total de entrada de la misma.

Ganancia directiva

La ganancia directiva es una cantidad definida para campo lejano como la relación entre la densidad de radiación en una dirección angular particular en el espacio y la densidad de radiación de la misma potencia irradiada isotrópicamente, lo que es:

$$D(\phi, \theta) = \frac{4\pi \text{ potencia irradiada por unidad de ángulo sólido en dirección } \phi, \theta}{\text{Total de potencia irradiada por la antena}} \quad (2.4)$$

Directividad

La directividad es la máxima ganancia directiva de la antena, es decir, la ganancia directiva en la dirección de máxima densidad de radiación.

Ganancia de la antena

La ganancia de la antena es definida como la relación entre la densidad de radiación en una dirección particular del espacio y la potencia total de entrada de la antena:

$$G(\phi, \theta) = \frac{4\pi \text{ potencia irradiada por unidad de ángulo sólido en dirección } \phi, \theta}{\text{Total de potencia irradiada por la antena}} \quad (2.5)$$

La ganancia máxima G , o simplemente ganancia, es el producto entre la directividad y la eficiencia de la antena, es decir:

$$G = D\eta \quad (2.6)$$

2.2. Multipath fading

Cuando se transmite una señal a través de una antena, la misma se propaga en el medio y enfrenta distintos tipos de obstáculos, tales como el suelo, edificios, autos, personas

e incluso las distintas capas de la atmósfera. Se producirán reflexiones y desfasajes que darán lugar a señales que provienen de distintos caminos o direcciones, las cuales se sumarán para obtener una señal resultante en la antena receptora. A este efecto se lo conoce como ***Multipath Fading*** [3].

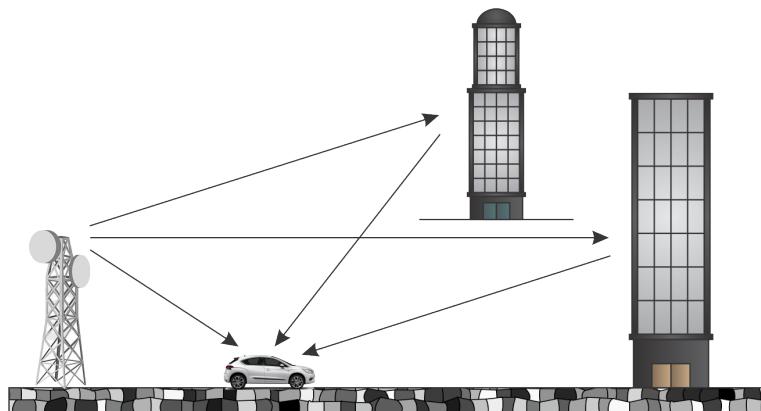


Figura 2.14: Múltiples Reflexiones de la señal transmitida

La presencia de reflectores en el ambiente que rodea al transmisor y receptor, crea múltiples caminos que la señal transmitida puede recorrer. La amplitud y la fase de la señal que llega desde cada camino estará relacionada con la longitud del camino y las condiciones, lo cual resulta en fluctuaciones considerables en la amplitud de la señal compuesta recibida. Esto puede resultar en una interferencia constructiva o destructiva, amplificando o atenuando la potencia de señal vista desde el receptor.

El análisis preciso de la propagación por múltiples caminos puede ser abordado a través del uso de las ecuaciones de Maxwell con condiciones de contorno que representen las propiedades físicas de la arquitectura y el medio ambiente. De todas formas, este análisis es muy complejo y se suele abordar el problema haciendo una derivación a las leyes de óptica geométrica, representando a la propagación de ondas electromagnéticas como la propagación de ondas ópticas.

Cuando se produce una interferencia destructiva de gran intensidad, se la reconoce como ***deep fade*** y puede resultar en una falla temporal de la comunicación debido a una severa caída en la relación señal ruido del canal.

Por dichos motivos, es deseable encontrar una forma de poder abordar el fenómeno de *multipath fading* con el objetivo de poder garantizar estabilidad y confiabilidad en la comunicación.

2.3. Beamforming

Beamforming consiste en la combinación de distintas señales de radio provenientes de un conjunto de antenas no direccionales, espacialmente separadas, con el objetivo de simular una gran antena direccional [4]. La antena simulada puede ser apuntada electrónicamente, a pesar de que físicamente, la antena no se mueva. En comunicaciones, la técnica de *beamforming* es utilizada para apuntar una antena hacia la fuente de la señal para reducir la interferencia y mejorar la calidad de la comunicación. En aplicaciones de descubrimiento de dirección, *beamforming* puede ser utilizado para dirigir una antena para determinar la dirección del origen de una señal.

2.3.1. Beamforming digital

La técnica de *beamforming* digital está basada en la conversión de una señal de RF proveniente de cada antena a dos flujos de señales binarias en banda base representando los canales I y Q [5]. Las señales digitales en banda base luego representan las amplitudes y fases de las señales recibidas en cada elemento del arreglo. El proceso de *beamforming* implica la ponderación de estas señales digitales, ajustando sus amplitudes y fases de forma tal que al ser sumadas en conjunto formen el haz deseado. Básicamente, un sistema de *beamforming* digital consta de un arreglo de antenas, receptores individuales para cada una de ellas, y un procesador digital de señales.

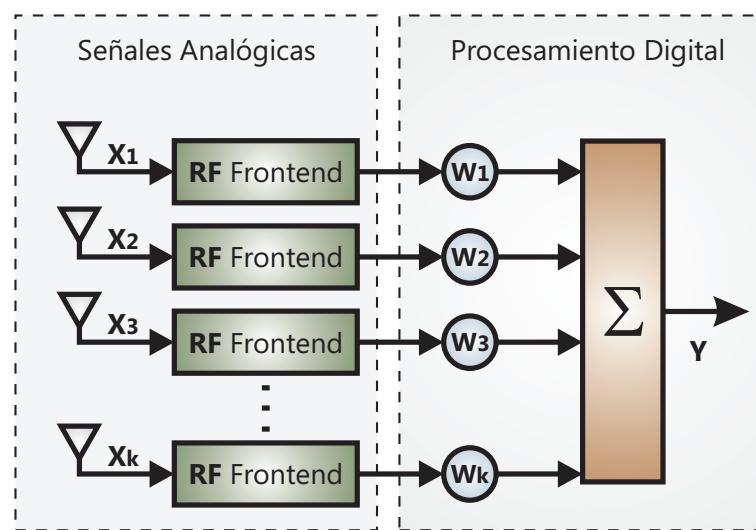


Figura 2.15: Esquema de un sistema de *Beamforming*

Se puede observar en la **figura 2.15** que las señales provenientes de cada una de las antenas son registradas por un frontend de RF en el cual se considera que está presente toda la circuitería analógica requerida para adaptar la señal y el conversor A/D utilizado

para transformarla a un stream digital. Las señales digitales obtenidas ingresan luego al procesador digital en el cual se implementan los pesajes w_1, w_2, \dots, w_k para obtener una suma ponderada de las mismas.

Tanto la amplitud como la fase de cada antena pueden ser controladas. La combinación del control de fase y amplitud puede ser utilizada para ajustar los lóbulos laterales y dirigir nulos mejor que lo que se puede lograr con el control de fase por si solo. La amplitud relativa a_k y desplazamiento de fase θ_k para cada antena es llamado un “peso complejo” y es representado por una constante compleja w_k (para la antena en la $k^{\text{ésima}}$ posición)

Un *beamformer* para un transmisor de radio aplica los coeficientes de ponderación a la señal a transmitir (desplaza la fase y aplica la amplitud) para cada elemento del arreglo de antenas.

Un *beamformer* para la recepción en radio aplica los pesos complejos a la señal de cada antena, y luego suma todas las señales para formar aquella que tiene el patrón direccional deseado.

Las primeras ideas que formaron las bases del *beamforming* digital fueron desarrolladas por investigadores en las áreas de sistemas de radar y sonar. El *beamforming* digital es una unión entre las tecnologías de antena y las tecnologías digitales. Una antena puede ser considerada un dispositivo que convierte señales espacio-temporales en señales estrictamente temporales, haciéndolas disponibles a una gran variedad de técnicas de procesamiento de señales. De esta forma, toda la información deseada que está siendo transportada por estas señales puede ser extraída.

Una ventaja mayor del *beamforming* digital radica en el hecho de que una vez que la información de RF es capturada en la forma de un flujo digital, dicho contenido está disponible para la aplicación de una gran cantidad de técnicas de procesamiento digital de señales.

El haz deseado es formado en el dominio digital (dentro del sistema de procesamiento). Para diseñar el sistema de procesamiento pueden utilizarse distintas tecnologías, tales como FPGA, DSPs de propósito general o chips específicamente diseñados para *beamforming*. El procesamiento digital requiere que la señal de cada antena sea digitalizada utilizando un conversor A/D. Debido a que las señales de radio por encima de las frecuencias de onda corta ($> 30\text{MHz}$) son muy altas para ser digitalizadas directamente a un costo razonable, los receptores de *beamforming* digital utilizan “traductores de RF” para desplazar la frecuencia de la señal hacia niveles inferiores antes de los conversores A/D. La clave de esta tecnología es lograr una traducción precisa de la señal analógica al régimen digital. Esto se logra utilizando receptores heterodinos completos, los cuales deben estar cercanamente apareados en amplitud y fase. Los receptores deben realizar el pasaje a frecuencias inferiores, filtrado y amplificación a un nivel de potencia adaptado a los requerimientos de entrada de los conversores A/D.

2.3.2. Beamforming adaptativo

Un *beamformer* adaptativo es un dispositivo que tiene la capacidad de separar señales en el dominio del espacio. Este hecho provee un medio para separar la señal deseada de señales de interferencia. Un *beamformer* adaptativo logra optimizar automáticamente el patrón del arreglo al ajustar el control de los pesos de ponderación hasta que se satisface una determinada función de objetivo prestablecida. Los medios por los cuales esta optimización es lograda están especificados por un algoritmo diseñado para tal propósito. Estos dispositivos utilizan mucha más información disponible en la antena con respecto a un *beamformer* convencional.

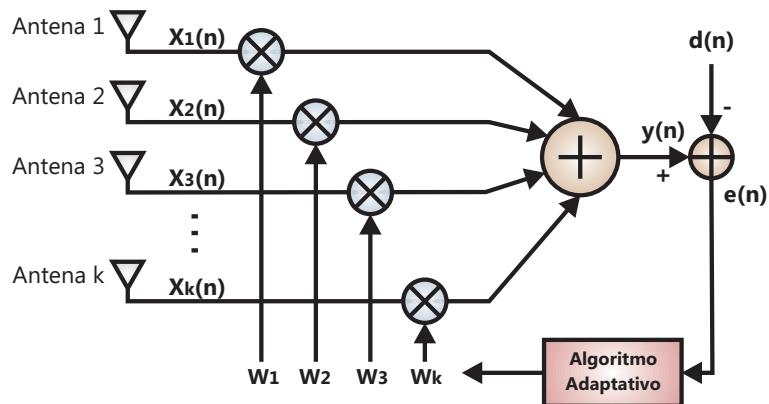


Figura 2.16: Esquema de un sistema de *Beamforming* Adaptativo

Conceptos básicos

El procedimiento utilizado para direccionar y modificar el patrón del haz del arreglo con el objetivo de mejorar la recepción de la señal deseada, mientras simultáneamente se suprimen las señales de interferencia a través de la selección de pesos complejos, se puede exponer como se explica a continuación.

Se tiene un arreglo como el expuesto en la figura 2.17, que consiste de dos antenas omnidireccionales con espaciado $\lambda_0/2$. La señal deseada, $S(t)$ llega desde la dirección de máxima ganancia ($\Theta_S = 0$), y la señal de interferencia, $I(t)$ llega desde el ángulo ($\Theta_1 = \pi/6$).

Ambas señales tienen la misma frecuencia f_o . La señal de cada antena es multiplicada por una variable de ponderación compleja, y las señales ponderadas son luego sumadas para formar la salida del arreglo. La contribución de señal deseada en la salida del arreglo es:

$$y_d(t) = Ae^{j2\pi f_o t}(w_1 + w_2) \quad (2.7)$$

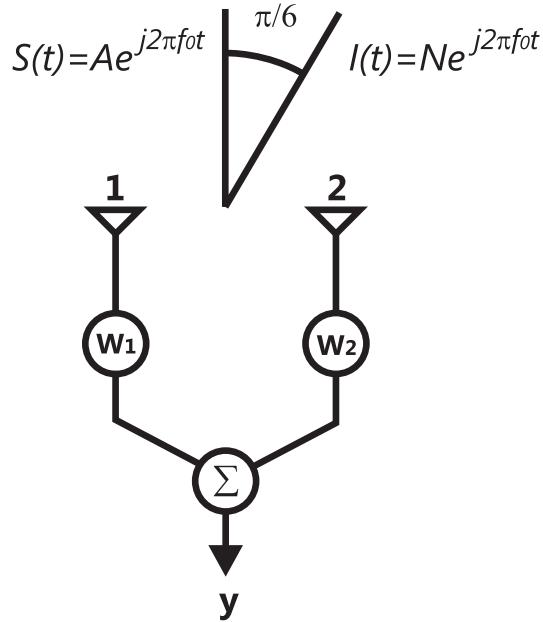


Figura 2.17: Arreglo de dos elementos para la supresión de interferencia

Para que \$y_d(t)\$ sea igual a \$S(t)\$, es necesario que

$$\begin{aligned}\Re[w_1] + \Re[w_2] &= 1 \\ \Im[w_1] + \Im[w_2] &= 0\end{aligned}\tag{2.8}$$

Donde \$\Re[]\$ e \$\Im[]\$ corresponden a las componentes real e imaginaria respectivamente. La señal de interferencia incidente llega al elemento 2 con una dirección de fase con respecto al elemento 1 de valor \$2\pi \frac{1}{\lambda_0} dsin(\pi/6) = \pi/2\$. Consecuentemente, la contribución de señal de interferencia en la salida del arreglo es:

$$y_i(t) = Ne^{j2\pi f_o t} w_1 + Ne^{(j2\pi f_o t + \pi/2)} w_2\tag{2.9}$$

Para que la respuesta de interferencia del arreglo sea cero es necesario que

$$\begin{aligned}\Re[w_1] + \Re[jw_2] &= 0 \\ \Im[w_1] + \Im[jw_2] &= 0\end{aligned}\tag{2.10}$$

La solución simultánea para las ecuaciones (2.8) y (2.10) lleva a:

$$w_1 = \frac{1}{2} - j\frac{1}{2} \quad w_2 = \frac{1}{2} + j\frac{1}{2}$$

Con estos pesos, el arreglo aceptará la señal deseada mientras que simultáneamente rechazará la interferencia.

El ejemplo expuesto toma ventaja en el hecho de que sólo hay un único origen de interferencia direccional y que utiliza la información a priori sobre la frecuencia y dirección de ambas señales. Un procesador práctico no debería requerir ese nivel de detalles en la información a priori sobre la ubicación, número y naturaleza de las fuentes de señal.

De todas formas, el ejemplo muestra que un sistema compuesto por un arreglo, que es configurado con pesos complejos, provee posibilidades innumerables para realizar los objetivos del sistema. Sólo se necesita desarrollar un procesador adaptativo práctico para llevar a cabo el ajuste en los pesos complejos.

2.3.3. Criterios para obtener los pesos óptimos

A continuación se mencionan algunos criterios existentes para la elección de los pesos óptimos para el filtro adaptativo. Se verá que cada uno de ellos se puede expresar como una solución a la **ecuación de Wiener Hopf**.

Mínimo error cuadrático medio

Considerar inicialmente un arreglo de antenas uniformemente espaciado, el cual opera en un ambiente de señal donde se tiene una señal de comunicación deseada $s(t)$, así como tambien N_u señales de interferencia $\{u_i(t)\}_{i=1}^{N_u}$. Luego considerar que las señales deseadas llegan al arreglo con un ángulo espacial θ_0 y que la señal de interferencia i llega con un ángulo θ_i . La salida del arreglo es representada por:

$$\mathbf{x}(t) = s(t)\mathbf{v} + \mathbf{u} = \mathbf{s} + \mathbf{u} \quad (2.11)$$

\mathbf{v} consiste en el vector de propagación del arreglo para la señal deseada:

$$\mathbf{v}^T = [1, e^{jkd\sin\theta_0} \dots e^{jk(K-1)d\sin\theta_0}] \quad (2.12)$$

\mathbf{u} representa la suma de todos los vectores de señales de interferencia:

$$\mathbf{u}^T = \sum_{i=1}^{N_u} u_i(t) \boldsymbol{\eta}_i \quad (2.13)$$

y $\boldsymbol{\eta}_i$ es el vector de propagación del arreglo para la $i^{\text{ésima}}$ señal de interferencia.

$$\boldsymbol{\eta}_i^T = [1, e^{jkdsin\theta_i} \dots e^{jk(K-1)dsin\theta_i}] \quad (2.14)$$

Si la señal deseada $s(t)$ es conocida, uno puede elegir minimizar el error entre la salida del *beamformer* $\mathbf{w}^H \mathbf{x}(t)$ y la señal deseada. Por supuesto, el conocimiento de la señal deseada eliminaría la necesidad del *beamformer*. De todas formas, para muchas aplicaciones, las características de la señal deseada pueden ser conocidas con suficiente detalle para generar una señal $d^*(t)$ que la representa cercanamente, o que por lo menos se correlaciona con la señal deseada en un cierto nivel. Esta señal es llamada la **señal de referencia**. Se expresa a la señal de referencia como un complejo conjugado sólo por conveniencia matemática. Los pesos son elegidos para minimizar el error cuadrático medio (MSE) entre la salida del *beamformer* y la señal de referencia:

$$\epsilon^2(t) = [d^*(t) - \mathbf{w}^H \mathbf{x}(t)]^2 \quad (2.15)$$

Al tomar la esperanza en ambos lados de la ecuación (2.15) y llevar a cabo algunas manipulaciones algebráicas, se tiene:

$$E\{\epsilon^2(t)\} = E\{d^*(t)\} - 2\mathbf{w}^H \mathbf{r} + \mathbf{w}^H \mathbf{R} \mathbf{w} \quad (2.16)$$

donde $\mathbf{r} = E\{d^*(t)x(t)\}$ y $\mathbf{R} = E\{\mathbf{x}(t)\mathbf{x}^H(t)\}$. \mathbf{R} es usualmente referida como la matriz de covarianza. El mínimo error cuadrático medio se obtiene al igualar a cero el gradiente del vector de la ecuación (2.16) con respecto a \mathbf{w} .

$$\nabla_{\mathbf{w}}(E\{\epsilon^2(t)\}) = -2\mathbf{r} + 2\mathbf{R}\mathbf{w} = 0 \quad (2.17)$$

La solución a esta ecuación es

$$\mathbf{w}_{opt} = \mathbf{R}^{-1} \mathbf{r} \quad (2.18)$$

la cual es referida como la ecuación de Wiener-Hopf o la solución óptima Wiener. Si $s(t) = d^*(t)$ luego $\mathbf{r} = E\{d^2(t)\}\mathbf{v}$. Se puede expresar $\mathbf{R} = E\{d^2(t)\}\mathbf{v}\mathbf{v}^H + \mathbf{R}_u$, donde $\mathbf{R}_u = E\{\mathbf{u}\mathbf{u}^H\}$ y aplicar la *Identidad de Woodbury* a \mathbf{R}^{-1} y se tiene:

$$\mathbf{R}^{-1} = \left[\frac{1}{1 + E\{d^2(t)\}\mathbf{v}^H \mathbf{R}_u^{-1} \mathbf{v}} \right] \mathbf{R}_u^{-1} \quad (2.19)$$

Luego, la solución de Wiener se puede generalizar como:

$$\mathbf{w}_{opt} = \beta \mathbf{R}_u^{-1} \mathbf{v} \quad (2.20)$$

Donde β es el coeficiente escalar. En el caso del mínimo MSE:

$$\beta = \frac{E\{d^2(t)\}}{1 + E\{d^2(t)\}\mathbf{v}^H\mathbf{R}_u^{-1}\mathbf{v}} \quad (2.21)$$

Máxima relación señal interferencia

Los pesos pueden ser elegidos para maximizar la relación señal interferencia (SIR). Si se asume que $\mathbf{R}_s = E\{\mathbf{s}\mathbf{s}^H\}$ y $\mathbf{R}_u = E\{\mathbf{u}\mathbf{u}^H\}$ son conocidas, se puede elegir maximizar la relación de la potencia de señal de salida σ_s^2 y el total de potencial de señal de interferencia σ_u^2 . La potencia de señal de salida puede expresarse como:

$$\sigma_s^2 = E\{|\mathbf{w}^H\mathbf{s}|^2\} = \mathbf{w}^H\mathbf{R}_s\mathbf{w} \quad (2.22)$$

y la potencia de salida de ruido como:

$$\sigma_u^2 = E\{|\mathbf{w}^H\mathbf{u}|^2\} = \mathbf{w}^H\mathbf{R}_u\mathbf{w} \quad (2.23)$$

Luego, la SIR está dada por:

$$\text{SIR} = \frac{\sigma_s^2}{\sigma_u^2} = \frac{\mathbf{w}^H\mathbf{R}_s\mathbf{w}}{\mathbf{w}^H\mathbf{R}_u\mathbf{w}} \quad (2.24)$$

Tomando la derivada de (2.24) con respecto a \mathbf{w} e igualándola a cero se obtiene:

$$\mathbf{R}_s\mathbf{w} = \frac{\mathbf{w}^H\mathbf{R}_s\mathbf{w}}{\mathbf{w}^H\mathbf{R}_u\mathbf{w}}\mathbf{R}_u\mathbf{w} \quad (2.25)$$

lo cual consiste en un problema de autovalores. El valor de $\frac{\mathbf{w}^H\mathbf{R}_s\mathbf{w}}{\mathbf{w}^H\mathbf{R}_u\mathbf{w}}$ está delimitado por los autovalores mínimo y máximo de la matriz simétrica $\mathbf{R}_u^{-1}\mathbf{R}_s$. El máximo autovalor λ_{\max} que satisface

$$\mathbf{R}_u^{-1}\mathbf{R}_s\mathbf{w} = \lambda_{\max}\mathbf{w} \quad (2.26)$$

es el óptimo valor de SIR. Correspondiente a este valor, existe un único autovector, w_{opt} que representa a los pesos óptimos. Luego:

$$\mathbf{R}_s\mathbf{w}_{\text{opt}} = \text{SIR } \mathbf{R}_u\mathbf{w}_{\text{opt}} \quad (2.27)$$

Notando que $\mathbf{R}_s = E\{d^2(t)\}\mathbf{v}\mathbf{v}^H$, se obtiene:

$$\mathbf{w}_{\text{opt}} = \beta \mathbf{R}_u^{-1} \mathbf{v} \quad (2.28)$$

donde

$$\beta = \frac{E\{d^2(t)\}}{\text{SIR}} \mathbf{v}^H \mathbf{w}_{\text{opt}} \quad (2.29)$$

Lo cual representa que el criterio del máximo SIR puede ser también expresado en términos de la solución de Wiener.

Varianza mínima

Si la señal deseada y su dirección son desconocidas, una forma de asegurar una buena recepción de señal es minimizar la varianza del ruido de salida. Recordando que la salida del *beamformer* es:

$$\begin{aligned} y(t) &= \mathbf{w}^H \mathbf{x} \\ &= \mathbf{w}^H \mathbf{s} + \mathbf{w}^H \mathbf{u} \end{aligned} \quad (2.30)$$

Para asegurar que la señal deseada es enviada con una ganancia y fase específica, se puede utilizar una restricción de forma que la respuesta del *beamformer* para la señal deseada sea:

$$\mathbf{w}^H \mathbf{v} = g \quad (2.31)$$

La minimización de las contribuciones a la salida a causa de la interferencia es lograda a través de la elección de los pesos para minimizar la varianza de la potencia de salida

$$\begin{aligned} \text{Var}\{y\} &= \mathbf{w}^H \mathbf{R} \mathbf{w} \\ &= \mathbf{w}^H \mathbf{R}_s \mathbf{w} + \mathbf{w}^H \mathbf{R}_u \mathbf{w} \end{aligned} \quad (2.32)$$

sujeta a la restricción definida en (2.31). Esto es equivalente a minimizar la cantidad $\mathbf{w}^H \mathbf{R}_u \mathbf{w}$. Utilizando el método de Lagrange, se tiene

$$\nabla \mathbf{w} \left(\frac{1}{2} \mathbf{w}^H \mathbf{R}_u \mathbf{w} + \beta [1 - \mathbf{w}^H \mathbf{v}] \right) = \mathbf{R}_u \mathbf{w} - \beta \mathbf{v} \quad (2.33)$$

de forma tal que

$$\mathbf{w}_{\text{opt}} = \beta \mathbf{R}_u^{-1} \mathbf{v} \quad (2.34)$$

donde

$$\beta = \frac{g}{\mathbf{v}^H \mathbf{R}_u^{-1} \mathbf{v}} \quad (2.35)$$

La solución 2.34 es también una solución de Wiener. Si $g = 1$, la respuesta del *beamformer* es normalmente conocida como la respuesta de mínima varianza sin distorsión (MVDR).

2.4. Algoritmos adaptativos

En la sección anterior, se mostró que los criterios óptimos están íntimamente relacionados entre sí. Por lo cual, la elección del criterio no es crítica en términos de rendimiento.

Por otro lado, la elección de **algoritmos adaptativos** para derivar los pesos adaptativos es muy importante, dado que determina tanto la velocidad de convergencia como la complejidad de hardware requerida para implementar el algoritmo. A continuación se discutirán algunas de las técnicas adaptativas más comunes.

2.4.1. Algoritmo LMS - *Least Mean Squares*

El algoritmo adaptativo más común para adaptatividad continua es el **algoritmo LMS**. Se basa en el método de descenso más pronunciado, que computa y actualiza recursivamente el vector de pesos. Es razonablemente intuitivo que correcciones sucesivas al vector de pesos en la dirección negativa del vector gradiente deberían eventualmente llevar al MSE, hasta un punto tal que el vector de pesos asume su valor óptimo. De acuerdo al método, el valor actualizado del vector de pesos en el momento $n + 1$ es computado utilizando la siguiente relación recursiva simple

$$\mathbf{w}(n+1) \approx \mathbf{w}(n) + \frac{1}{2}\mu[-\nabla(E\{\epsilon^2(n)\})] \quad (2.36)$$

Reemplazando con (2.17) se obtiene

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu[\mathbf{r} - \mathbf{R}\mathbf{w}(n)] \quad (2.37)$$

En la realidad, una medición exacta del vector gradiente no es posible, dado que esto requeriría un conocimiento previo tanto de \mathbf{R} como de \mathbf{r} . La estrategia más obvia consiste en usar estimaciones instantáneas, las cuales son definidas respectivamente como:

$$\hat{\mathbf{R}}(n) = \mathbf{x}(n)\mathbf{x}^H(n) \quad (2.38)$$

y

$$\hat{\mathbf{r}}(n) = \hat{d}(n)\mathbf{x}(n) \quad (2.39)$$

Los pesos pueden ser actualizados como

$$\begin{aligned} \hat{\mathbf{w}}(n+1) &= \hat{\mathbf{w}}(n) + \mu\mathbf{x}(n)[\hat{d}(n) - \mathbf{x}^H(n)\hat{\mathbf{w}}(n)] \\ &= \hat{\mathbf{w}}(n) + \mu\mathbf{x}(n)\hat{e}(n) \end{aligned} \quad (2.40)$$

La constante de ganancia μ controla las características de convergencia del vector de secuencia aleatorio $\mathbf{w}(n)$. Se debe notar que este es un enfoque continuamente adaptativo, en el cual los pesos son actualizados a medida que los datos son muestreados de forma tal que el vector resultante converja a la solución óptima. La adaptatividad continua funciona bien cuando las estadísticas relacionadas con el ambiente de la señal son **estacionarias**. La figura 2.18 muestra la representación gráfica del flujo de la señal del **algoritmo LMS**. La virtud principal de este algoritmo es su simplicidad. El rendimiento es aceptable en diversas aplicaciones. De todas formas, sus características de convergencia dependen en la estructura de autovalores de $\hat{\mathbf{R}}$. Cuando éstos están ampliamente difundidos, la convergencia puede ser lenta y debería considerarse utilizar otros algoritmos adaptativos con tasas de convergencia más altas.

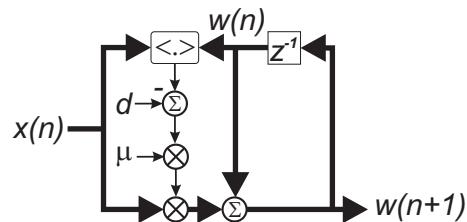


Figura 2.18: Representación gráfica del flujo de señal para el algoritmo LMS

2.4.2. Algoritmo RLS - *Recursive Least Squares*

En este algoritmo se realiza la estimación de \mathbf{R} y \mathbf{r} realizando una suma ponderada:

$$\hat{\mathbf{R}}(n) = \sum_{i=1}^N \gamma^{n-1} \mathbf{x}(i) \mathbf{x}^H(i) \quad (2.41)$$

y

$$\hat{\mathbf{r}}(n) = \sum_{i=1}^N \gamma^{n-1} \hat{d}(i) \mathbf{x}(i) \quad (2.42)$$

El factor de ponderación, $0 < \gamma \leq 1$ tiene la función de asegurar que los datos en el pasado son “olvidados” con el objetivo de permitir que el procesador siga las variaciones estadísticas de los datos observables. Factorizando los términos correspondientes a $i = n$ tanto en (2.41) y (2.42), se tiene la siguiente recursión para actualizar tanto $\hat{\mathbf{R}}(n)$ y $\hat{\mathbf{r}}(n)$:

$$\hat{\mathbf{R}}(n) = \gamma \mathbf{R}(n-1) + \mathbf{x}(n) \mathbf{x}^H(n) \quad (2.43)$$

y

$$\hat{\mathbf{r}}(n) = \gamma \hat{\mathbf{r}}(n-1) + \hat{d}(n) \mathbf{x}(n) \quad (2.44)$$

Utilizando la identidad de Woodbury, se puede obtener la siguiente ecuación recursiva para llegar a la inversa de la matriz de covarianza:

$$\mathbf{R}^{-1}(n) = \gamma^{-1} [\mathbf{R}^{-1}(n-1) - \mathbf{q}(n) \mathbf{x}(n) \mathbf{R}^{-1}(n-1)] \quad (2.45)$$

donde el vector de ganancia $\mathbf{q}(n)$ está dado por:

$$\mathbf{q}(n) = \frac{\gamma^{-1} \mathbf{R}^{-1}(n-1) \mathbf{x}(n)}{1 + \gamma^{-1} \mathbf{x}^H(n) \mathbf{R}^{-1}(n-1) \mathbf{x}(n)} \quad (2.46)$$

Para desarrollar la ecuación recursiva para actualizar el estimador de cuadrados mínimos $\hat{\mathbf{w}}(n)$ se usa (2.18) para expresar $w(n)$ como sigue:

$$\begin{aligned} \hat{\mathbf{w}}(n) &= \mathbf{R}^{-1}(n) \mathbf{r}(n) \\ &= \gamma^{-1} [\mathbf{R}^{-1}(n-1) - \mathbf{q}(n) \mathbf{x}(n) \mathbf{R}^{-1}(n-1)] \\ &\quad \times [\gamma \mathbf{r}(n-1) + \hat{d}(n) \mathbf{x}(n)] \end{aligned} \quad (2.47)$$

Reorganizando la ecuación, se puede actualizar el peso como sigue:

$$\hat{\mathbf{w}}(n) = \hat{\mathbf{w}}(n-1) + \mathbf{q}(n)[\hat{d}(n) - \hat{\mathbf{w}}^H(n-1)\mathbf{x}(n)] \quad (2.48)$$

Una característica importante del algoritmo RLS es que la inversión de la matriz de covarianza $\mathbf{x}(n)$ es reemplazada en cada paso por una división escalar simple. La figura 2.19 muestra la representación del flujo de señal del algoritmo RLS. La tasa de convergencia para el algoritmo RLS es típicamente un orden de magnitud más alta que la del algoritmo LMS.

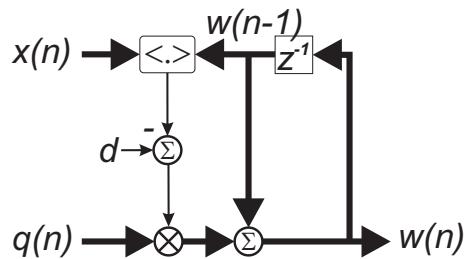


Figura 2.19: Representación gráfica del flujo de señal para el algoritmo RLS

2.4.3. Adquisición de la señal de referencia

Se han planteado distintos criterios y algoritmos para realizar *beamforming* adaptativo. Todos ellos requieren algún tipo de señal de referencia en su proceso de optimización adaptativa. Cuando se habla de señales de referencia, normalmente se refiere a información a priori y explícita o conocimientos sobre las señales de interés. Una referencia explícita puede ser dividida en dos categorías: referencia espacial y referencia temporal. La primera es referida normalmente como la información de ángulo de arribo (AOA - *Angle of Arrival*) de la señal deseada. Una señal de referencia temporal puede ser una señal piloto que está correlacionada con la señal deseada, una secuencia especial incluida en un paquete por la señal deseada, o un código pseudo-ruido (PN - *pseudo-noise*) conocido en un sistema CDMA. La forma de referencia disponible depende del sistema particular donde se implementará el *beamforming* adaptativo. Si una señal de referencia explícita está disponible en el sistema, se la debe utilizar tanto como sea posible para lograr menor complejidad, mayor precisión y convergencia rápida.

Es valioso describir el proceso de adquisición de la señal de referencia en un sistema CDMA. El *beamforming* adaptativo es muy adecuado para un sistema CDMA, debido a que los códigos de dispersión pueden ser usados como referencias para *beamforming*. La implementación común de *beamforming* adaptativo en un sistema de comunicaciones wireless CDMA es el uso del lazo de generación de referencia de Compton. Una configuración de implementación genérica para *beamforming* adaptativo en CDMA se muestra en la figura 2.20. En esta configuración, la demodulación es realizada luego del *beamforming*. Esto es, la salida del arreglo es primero mezclada con una señal de un

oscilador local codificada CDMA, la cual es filtrada y limitada. La señal limitada es re-modulada a través de una mezcla con la señal del oscilador local codificado CDMA correspondiente. La señal remodulada es usada como señal de referencia, la cual es comparada con la salida retrasada del arreglo para producir una señal de error. La señal de error dirige al procesador adaptativo para actualizar los pesos de *beamforming*. El loop de retroalimentación es no lineal debido a la operación de limitación.

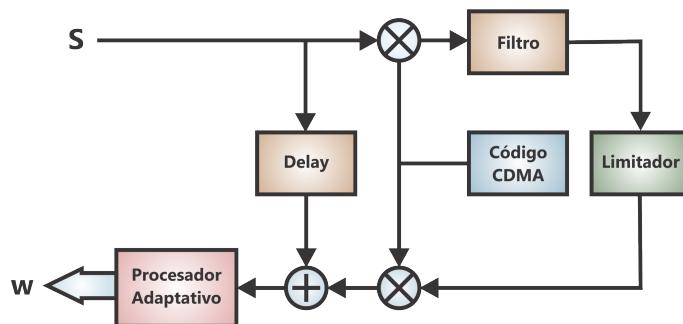


Figura 2.20: Una configuración genérica para *beamforming* adaptativo en un sistema de comunicación wireless CDMA

2.5. Otras técnicas de diseño MIMO

Los sistemas MIMO pueden ser diseñados para proveer máxima diversidad para incrementar la confiabilidad de la transmisión, o para alcanzar una máxima ganancia de multiplexado para soportar altos niveles de tasas de transmisión.

La capacidad de un canal MIMO puede ser mucho más alta que la de un sistema SISO. Este rendimiento de un sistema MIMO es cuantificado a través de la ganancia de multiplexado espacial.

Si distintas copias de la señal son transmitidas o recibidas desde múltiples antenas, los sistemas pueden mejorar la confiabilidad de un enlace inalámbrico. Esta ganancia es llamada ganancia de diversidad.

Los sistemas MIMO pueden ser utilizados simultáneamente para proveer tanto ganancia por diversidad como por multiplexado. De todas formas, existe una relación de compromiso entre ellos.

2.5.1. *Space-Time block coding*

En este tipo de diseño, se toman copias de la señal, las cuales son transmitidas desde múltiples antenas o son recibidas en más de una antena en sistemas multi-antena espacio-tiempo. La probabilidad de error promedio de símbolo de un sistema de comunicaciones

MIMO para detección de máxima probabilidad tiene un límite superior en niveles altos de SNR [6]:

$$p_e \leq \overline{N}_e \left(\frac{\gamma d_{min}}{4M} \right)^{-M} \quad (2.49)$$

donde \overline{N}_e es el número de vecinos cercanos en una constelación escalar, d_{min} es la distancia mínima de separación de la constelación escalar, y $M = \min\{M_R, M_T\}$.

Como se puede ver en la figura 2.21, incrementar el número de antenas aumenta la pendiente de las curvas del BER (bit error rate) y mejora la confiabilidad de la comunicación inalámbrica.

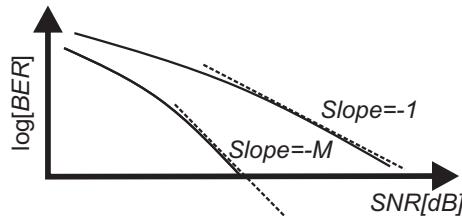


Figura 2.21: Ganancia de diversidad espacio-tiempo en sistemas MIMO

Uno de los ejemplos más claros de *space-time block coding* es el conocido como código de Alamouti. El código de Alamouti es un space-time block code ortogonal (O-STBC). Es implementado utilizando dos antenas en el transmisor y un número arbitrario de antenas en el receptor. Las palabras del código para múltiples antenas son escritas de la siguiente forma:

$$X = \frac{1}{\sqrt{2}} \begin{pmatrix} \chi_1 & -\chi_2^* \\ \chi_2 & \chi_1^* \end{pmatrix} \quad (2.50)$$

El transmisor de Alamouti se muestra en la 2.22. El código de Alamouti tiene un rango de spatial multiplexing igual a uno, dado que se transmiten dos símbolos en el tiempo de duración de dos símbolos.

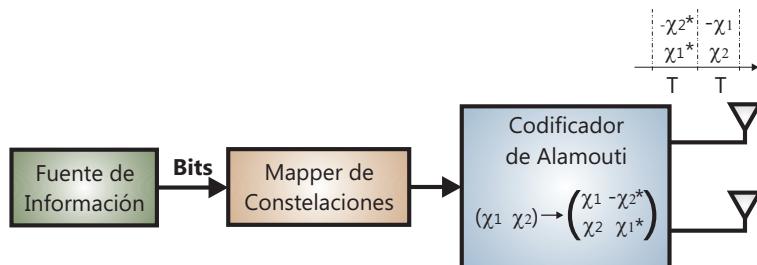


Figura 2.22: Esquema de un transmisor de Alamouti

2.5.2. Spatial multiplexing

Spatial Multiplexing es una de las técnicas que se encuentra en el grupo de los códigos de capas de espacio-tiempo [6]. En la misma, la secuencia de información es dividida en subflujos precisos. Los subflujos donde se lleva a cabo el procesamiento son conocidos como capas. Debido a ello a esta se la conoce como técnica de capas de espacio tiempo (LAST: layer space-time technique). Como se observa en la figura 2.23, se transmiten M_T subflujos independientes a través de M_T antenas. En esta técnica, el número de antenas receptoras debe ser igual o mayor al número de antenas transmisoras.

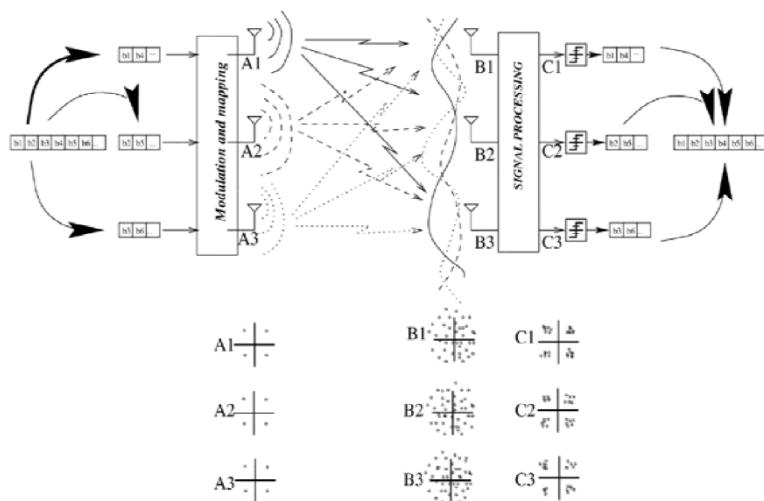


Figura 2.23: Spatial Multiplexing con tres antenas en transmisor y receptor

El proceso de dividir la secuencia de información en subflujos es realizado por un demultiplexor. El proceso de demultiplexado debe ser aplicado a bits o símbolos. Luego, la codificación puede ser realizada en tres formas diferentes de acuerdo a la posición del demultiplexor en la cadena del transmisor y la dirección de la capa. Estos procesos de codificación son referidos como horizontal, vertical y diagonal. Las distintas técnicas de codificación se muestran en la figura 2.24.

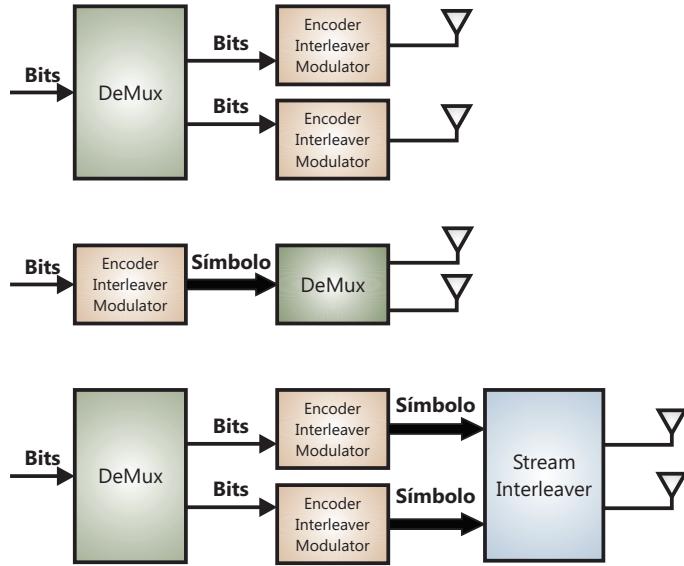


Figura 2.24: Tipos de codificación para Spatial Multiplexing

En el codificado horizontal, los bits de datos son demultiplexados en M_T subflujos que son independientemente codificados, intercalados y modulados. Por otro lado, en la realización vertical, el flujo de datos es codificado, intercalado y modulado; y, los símbolos resultantes son luego demultiplexados en M_T subflujos. El proceso para realizar spatial multiplexing diagonal es similar al codificado horizontal, con la única diferencia de que, luego de la etapa final, los frames de símbolos son sometidos a un intercalado de flujo, el cual rota los frames transmitidos.

Capítulo 3

Arquitecturas digitales para la implementación de sistemas MIMO

En el capítulo anterior fueron expuestos los beneficios de la implementación de la técnica de *beamforming* en los sistemas MIMO. Se llegó a la conclusión de que, para la implementación de un filtro de *beamforming*, se requiere el cómputo una de matriz inversa. Para este objetivo, existen diferentes algoritmos. La elección del algoritmo adaptativo para derivar el vector de pesos es muy importante, dado que determina tanto la velocidad de convergencia como la complejidad de hardware requerido para implementar el algoritmo.

Existen dos algoritmos muy populares para resolver esta necesidad, **el algoritmo LMS** (*Least Mean Squares*) y **el algoritmo RLS** (*Recursive Least Squares*), entre otros, los cuales fueron abordados en el capítulo anterior. Entre los diferentes algoritmos, generalmente el algoritmo RLS es preferido por su propiedad de convergencia rápida.

Entre los diversos trabajos analizados, los cuales se mencionan en las últimas secciones de este capítulo, se encontró que en la gran mayoría de los casos el algoritmo implementado es el algoritmo RLS. Existen diversas formas de implementar las ecuaciones en hardware. Diferentes técnicas encontradas mencionan el uso del método de **Gram Schmidt**, las **transformaciones de Householder**, el método **Squared Givens Rotations** y las rotaciones de Givens. Las diferencias sustanciales entre los trabajos analizados se encuentran en la forma de implementar los procesadores de cómputo de las ecuaciones (CORDIC, squared givens rotations, Gram-Schmidt), el tipo de mapeo de la arquitectura junto con la cantidad de procesadores utilizados, y la forma de interconexión de los mismos.

En este capítulo, se abordarán los distintos tipos de arquitecturas existentes para la

implementación de procesadores de descomposición QR.

3.1. Resolución del algoritmo RLS

El algoritmo RLS estándar requiere el cómputo explícito de una matriz de correlación [8]. Este es un cálculo intensivo que tiene el efecto de elevar al cuadrado el número de condición del problema, causando un efecto negativo en la longitud de palabra para la estabilidad en sistemas de longitud de palabra finita. Los pesos pueden ser calculados en un modo más estable, evitando el cálculo de la matriz de correlación y su inversa al utilizar la **descomposición QR**, una forma de triangularización ortogonal con buenas propiedades numéricas.

3.2. Descomposición QR

Existe una familia de algoritmos RLS numéricamente estables y robustos que evolucionó en un rango de métodos de descomposición QR, tales como las rotaciones de Givens y las transformaciones de Householder. Las rotaciones de Givens son rotaciones ortogonales planas, utilizadas para eliminar elementos en una matriz. Al aplicar una serie de rotaciones de Givens sucesivas, una matriz puede ser triangularizada al eliminar los elementos debajo de la diagonal. Esta operación es conocida como factorización QR, en la cual una matriz $X(n)$ es descompuesta en una matriz triangular superior $R(n)$ y una matriz ortogonal $Q(n)$ (sus columnas son vectores unitarios ortogonales indicando que $Q^T Q = I$) de forma tal que:

$$X(n) = Q(n)R(n)$$

La matriz $X(n)$ de dimensión $P \times N$ es descompuesta en una matriz triangular superior $R(n)$ de dimensión $N \times N$ a través de la aplicación de una matriz unitaria $Q^T(n)$ de forma tal que:

$$Q^T(n)X(n) = \begin{bmatrix} R(n) \\ 0 \end{bmatrix}$$

Donde 0 es la matriz cero que resulta si $N < P$. Debido a que $Q(n)$ es una matriz unitaria, entonces:

$$\phi(n) = X^T(n)X(n) = X^T(n)Q(n)Q^T(n)X(n) = R^T(n)R(n)$$

La matriz triangular, $R(n)$, es el factor Cholesky (raíz cuadrada) de la matriz de correlación de información $\phi(n)$. Debido a que $Q(n)$ es unitaria el sistema de ecuaciones original puede expresarse como:

$$\|J(n)\| = \|Q(n)e(n)\| = \left\| \underbrace{Q^T(n)X(n)}_{R(n)} W_{LS}(n) + \underbrace{Q^T(n)y(n)}_{u(n)} \right\|$$

Se sigue que el vector de cuadrados mínimos $w_{LS}(n)$ debe satisfacer la ecuación:

$$R(n)w_{LS}(n) + u(n) = 0$$

Debido a que $R(n)$ es una matriz triangular superior, los pesos pueden ser resueltos utilizando sustitución. Esta descomposición permite que la matriz sea triangularizada nuevamente cuando nueva información entra en la matriz, sin la necesidad de computar la triangularización desde el formato de matriz cuadrada original. La matriz de información $X(n)$ y el vector de medida $y(n)$ en un tiempo n pueden ser representados en un modo recursivo a través de la matriz resultante previa y el vector de nueva información, de forma tal que:

$$X(n) = \begin{bmatrix} \lambda(n)X(n-1) \\ \underline{x}^T(n) \end{bmatrix}$$

y

$$y(n) = \begin{bmatrix} \lambda(n)y(n-1) \\ \underline{y}(n) \end{bmatrix}$$

donde $\underline{x}^T(n)$ e $\underline{y}^T(n)$ forman la fila concatenada en el tiempo n . Una forma de raíz cuadrada del algoritmo es lograda como sigue:

$$\begin{aligned} Q^T(n) & \begin{bmatrix} \lambda^{0.5}R(n-1) \\ \underline{x}^T(n) \end{bmatrix} W_{LS}(n) = \\ & Q^T(n) \begin{bmatrix} \lambda^{0.5}u(n-1) \\ \underline{y}(n) \end{bmatrix} + Q^T(n)e(n) \end{aligned}$$

donde $\beta = \lambda^{0.5}$. Luego, esto nos da:

$$Q^T(n) \begin{bmatrix} \beta(n)R(n-1) & \beta(n)u(n-1) \\ \underline{x}^T(n) & \underline{y}(n) \end{bmatrix} = \begin{bmatrix} R(n) & u(n) \\ 0 & \alpha(n) \end{bmatrix}$$

Esto se computa para dar:

$$\begin{bmatrix} R(n) \\ 0 \end{bmatrix} W_{LS}(n) = \begin{bmatrix} u(n) \\ \alpha(n) \end{bmatrix}$$

$\alpha(n)$ está relacionado con el residuo de cuadrados mínimos a posteriori, $e(n)$, en el tiempo n de forma que:

$$e(n) = \alpha(n)\gamma(n)$$

donde $\gamma(n)$ es el producto de cosenos generado en el proceso de eliminar $\underline{x}^T(n)$.

El gráfico de dependencia a alto nivel en la realización de la solución QR para RLS se puede ver en la figura 3.1:

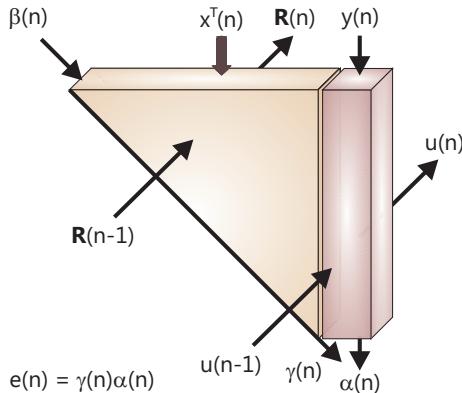


Figura 3.1: Gráfico de dependencias en alto nivel para la solución QR-RLS

La matriz $X(n)$ es pre-multiplicada por matrices de rotación, un elemento a la vez. Los parámetros son calculados de forma que los elementos por debajo de la diagonal en la primera columna sean convertidos a cero. Luego se sigue con los elementos de la siguiente columna, hasta que la matriz triangular superior equivalente es conformada.

Givens logra esta operación a través de una secuencia de rotaciones, que se pueden explicar utilizando el siguiente ejemplo para una matriz de 2×3 :

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

La matriz es transformada en una matriz pseudo-triangular al eliminar el elemento a_{21} . Esto se logra a través de la multiplicación de la siguiente matriz de rotación con la matriz M :

$$G = \begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix}$$

Luego:

$$\begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} = \begin{bmatrix} a_{11} \cos \alpha - a_{21} \sin \alpha & a_{12} \cos \alpha - a_{22} \sin \alpha & a_{13} \cos \alpha - a_{23} \sin \alpha \\ a_{11} \sin \alpha + a_{21} \cos \alpha & a_{12} \sin \alpha + a_{22} \cos \alpha & a_{13} \sin \alpha + a_{23} \cos \alpha \end{bmatrix}$$

Para eliminar a_{21} se debe resolver $a_{11} \sin \alpha + a_{21} \cos \alpha = 0$. Luego, por trigonometría, se puede plantear la siguiente solución:

$$\cos \alpha = a_{11} / \sqrt{a_{11}^2 + a_{21}^2} \quad (3.1)$$

$$\sin \alpha = -a_{21} / \sqrt{a_{11}^2 + a_{21}^2} \quad (3.2)$$

$$a_{11new} = \sqrt{a_{11}^2 + a_{21}^2} \quad (3.3)$$

$$a_{21new} = 0 \quad (3.4)$$

Aplicando la rotación para eliminar a_{21} resulta en la matriz pseudo-triangular:

$$M = \begin{bmatrix} a_{11new} & a_{12new} & a_{13new} \\ 0 & a_{22new} & a_{23new} \end{bmatrix}$$

Esta función puede implementarse en un arreglo sistólico, como se ve en la figura 3.2, el cual consiste en dos tipos de celdas, referidas como *boundary cell* (BC - representada con un círculo) e *internal cell* (IC - representada con un cuadrado).

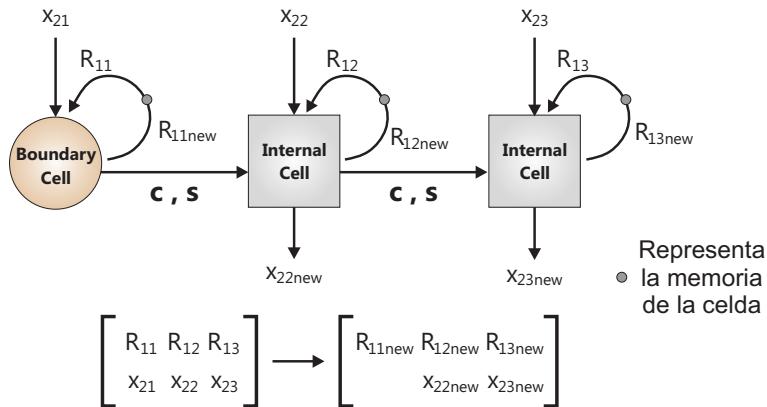


Figura 3.2: Transformación por Rotaciones de Givens

Los elementos en los cuales se aplican las rotaciones son x y R , donde x es el valor de entrada en la celda, y R es el valor retenido en la memoria para esa celda. Los parámetros

de rotación, $\cos \alpha$ y $\sin \alpha$, son calculados en una *boundary cell*, de forma tal que el valor de x que ingresa a una *boundary cell* sea convertido a 0, y el valor de R de dicha celda sea actualizado acorde a la rotación y almacenado para la próxima iteración.

Los parámetros de rotación son enviados sin modificar a lo largo de toda una fila a cada una de las *internal cells* para continuar con la rotación. El gráfico de dependencia de la figura 3.2 representa la eliminación de x_{21} , relacionado con el elemento de la matriz a_{21} del ejemplo anterior. Los valores de R y x son considerados una coordenada polar (R, x).

Eliminar la entrada x en una *boundary cell*, se logra a través de rotar el vector un ángulo α tal que:

$$R_{new} = R \cos \alpha - x \sin \alpha = \frac{R^2 + x^2}{\sqrt{R^2 + x^2}} = \sqrt{R^2 + x^2}$$

donde

$$\cos \alpha = \frac{R}{R_{new}} = c \quad \sin \alpha = \frac{-x}{R_{new}} = s$$

Los mismos parámetros de rotación utilizados en BC son aplicados en las ICs:

$$R_{new} = cR - sX \quad x_{new} = cx + sR$$

Al concatenar rotaciones de Givens sucesivas, la factorización QR puede ser implementada para una matriz $N \times N$. El diagrama de la figura 3.3 muestra un ejemplo para una matriz de 3×3 . Se trata de un esquema *pipeline* en el cual los valores de R son realimentados a sus propias celdas para la próxima iteración.

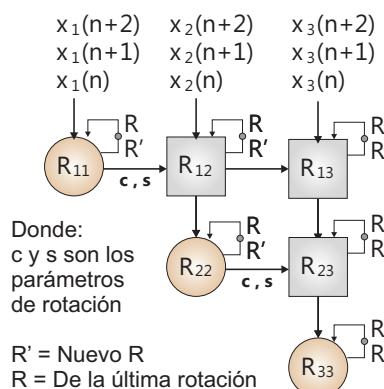


Figura 3.3: Descomposición QR aplicada a una matriz de 3×3

A continuación se explorará en un nivel más detallado el proceso de desarrollar una arquitectura en hardware desde el algoritmo RLS y sus ecuaciones resuelto por descomposición QR utilizando rotaciones de Givens.

3.3. Del algoritmo a una arquitectura sintetizable

Con el objetivo de lograr la implementación, es necesario transformar las ecuaciones presentadas en una arquitectura sintetizable. En este proceso, es un aspecto clave el lograr una implementación de un circuito de alta *performance*, para asegurar una traducción eficiente del algoritmo a hardware en silicio. Esto normalmente conlleva el hecho de desarrollar una arquitectura en la cual operaciones independientes se desarrolle en paralelo para aumentar el rendimiento.

En la figura 3.4 se presenta el proceso de transformar las ecuaciones a un procesador en hardware.

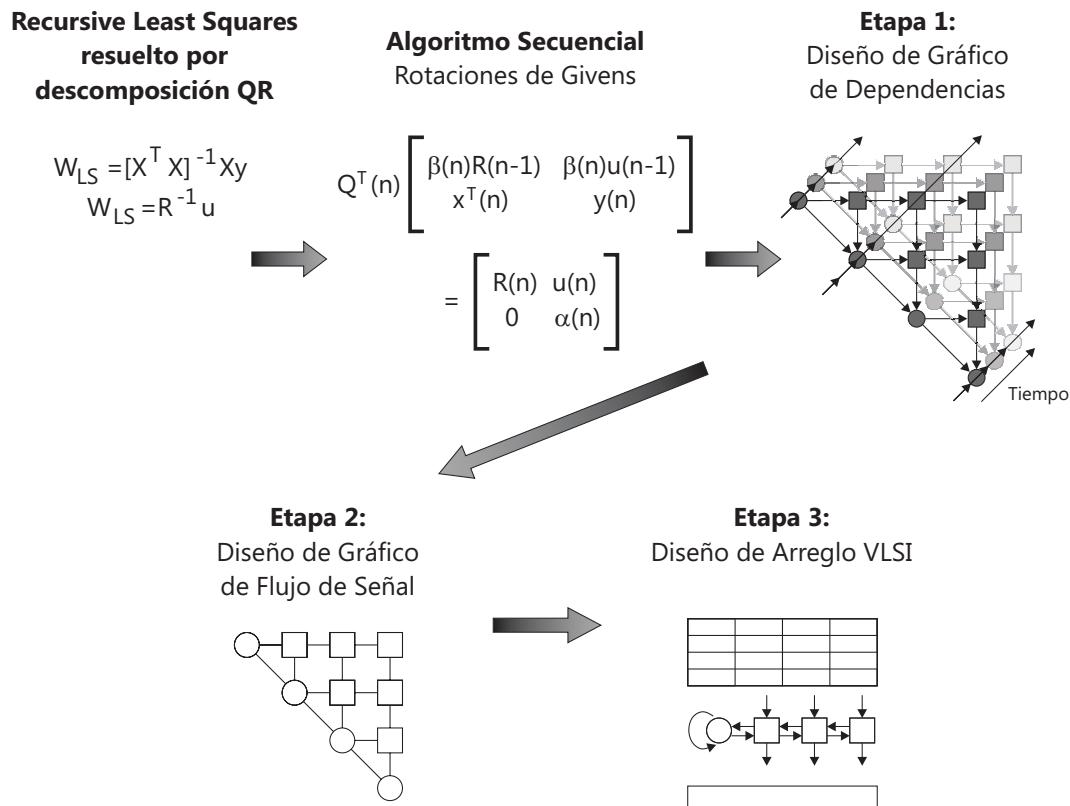


Figura 3.4: Proceso de desarrollo del circuito digital

El punto de inicio en la figura es la resolución del algoritmo RLS a través de la descomposición QR. La siguiente etapa muestra dicha resolución a través del uso de un

algoritmo secuencial. Esto es, por cada iteración del algoritmo, un nuevo conjunto de valores ingresa a las ecuaciones y evoluciona a una nueva solución.

La operación QR puede representarse como un arreglo triangular de operaciones. La matriz de información es la entrada en la parte superior del triángulo, y en cada fila son eliminados los términos para resultar en una matriz triangular superior.

El gráfico de dependencias en la figura 3.4 expone este proceso de triangularización. Los arreglos triangulares en cascada representan **cada iteración en el tiempo**. Las flechas muestran la dependencia a través del tiempo. Desde el diagrama de dependencias, se desarrolla posteriormente un diagrama de flujo de señal adecuado, y con el mismo se deriva una arquitectura sintetizable.

3.3.1. Gráfico de dependencias

En este diagrama se pueden detectar las dependencias entre los datos, permitiendo identificar el máximo nivel de concurrencia posible al desensamblar el algoritmo en nodos y flechas. Los nodos representan los cálculos y la dirección de las flechas la dependencia de las operaciones. En la figura 3.5 se muestra un ejemplo de 3×3 . Cada esquema en perspectiva representa una iteración en el tiempo (para n , $n + 1$ y $n + 2$). Las flechas demuestran que, por ejemplo, para hacer el cálculo de a_{12} , se requiere la salida del procesamiento realizado en a_{11} , o que para hacer el cálculo de a_{22} , se requiere la salida de los procesamientos realizados en a_{11} y a_{12} .

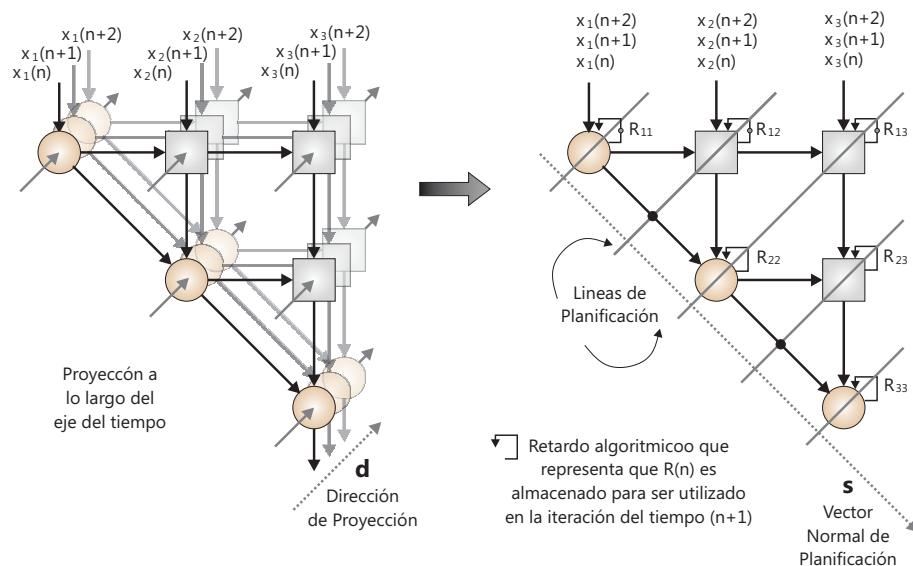


Figura 3.5: Gráfico de Dependencias y Gráfico de Flujo de Señal

3.3.2. Gráfico de flujo de señal

En la figura 3.5, se puede observar la transición del gráfico de dependencias al gráfico de flujo de señal. Este es el gráfico de mayor importancia para derivar a una arquitectura, dado que representa el sentido en el cual deben transmitirse las señales, y el orden en el cual se deben realizar las operaciones. Los nodos del gráfico de dependencias son asignados a procesadores, y sus operaciones son planificadas entre los mismos.

Una de las formas más sencillas para lograr esta transición es la **proyección lineal de todos los nodos idénticos** a lo largo de una linea recta en un único procesador. Esto es representado por el vector de proyección \mathbf{d} . La planificación lineal es utilizada para determinar el orden en el cual las operaciones son realizadas en estos procesadores. Las **líneas de planificación** en la figura 3.5 **indican las operaciones que se pueden realizar en paralelo** en cada ciclo de reloj. Matemáticamente están representadas por un **vector de planificación** \mathbf{s} normal a las líneas de planificación, el cual apunta en la dirección de la dependencia de las operaciones (muestra el orden en el cual cada línea de operaciones es realizada).

Existen dos reglas que gobiernan la proyección y la planificación:

- Todos los arcos de dependencia fluyen en la misma dirección a lo largo de las líneas de planificación.
- Las líneas de planificación no son paralelas con el vector de proyección \mathbf{d} .

En el ejemplo QR, cada arreglo triangular de celdas en el diagrama de dependencias representa una actualización QR. Al estar en cascada, el diagrama representa una **secuencia de actualizaciones QR**. Al proyectar a lo largo del eje del tiempo, todas las actualizaciones QR pueden ser asignadas a un diagrama de flujo de señal triangular, como se observa en la imagen.

Los valores R son transmitidos a lo largo del tiempo de una actualización QR a la otra, representados por la cascada de arreglos triangulares. Esta transición se observa mejor en el diagrama de flujo de señal en la realimentación de los valores R en la celda a partir de un *delay* algorítmico utilizado para retener los valores hasta la próxima iteración. Esto es referido como un lazo recursivo.

La simplicidad del diagrama de flujo de señal es que asume que todas las operaciones en los nodos ocupan un único ciclo, así como los *delays* algorítmicos, representados por pequeños círculos oscuros. Estos *delays* algorítmicos partitionan las iteraciones del algoritmo y son una parte necesaria del mismo. El resultado del diagrama de flujo de señal es una representación más concisa del algoritmo con respecto al diagrama de dependencias.

El camino a seguir, una vez definido el diagrama de flujo de señal, es derivar una arquitectura eficiente e implementar en hardware el algoritmo.

3.4. Implementación sistólica de las rotaciones de givens

A partir del gráfico de flujo de señal, se debe profundizar el mismo a un nivel de detalle que defina la forma en la cual se implementarán los procesadores, la cantidad de los mismos que van a ser utilizados, la forma en la cual se almacenarán los datos, y cómo se realizará el control de los mismos. Existen diferentes mapeos desarrollados para derivar del arreglo triangular una arquitectura sintetizable. Veremos a continuación algunos de ellos.

3.4.1. Mapeo de Gentleman y Kung

El modo más directo de transformar el algoritmo en una arquitectura sintetizable consiste en la implementación de un arreglo sistólico en el cual cada *boundary cell* e *internal cell* sea calculada por un procesador independiente. Dicho esquema se presenta en la figura 3.6.

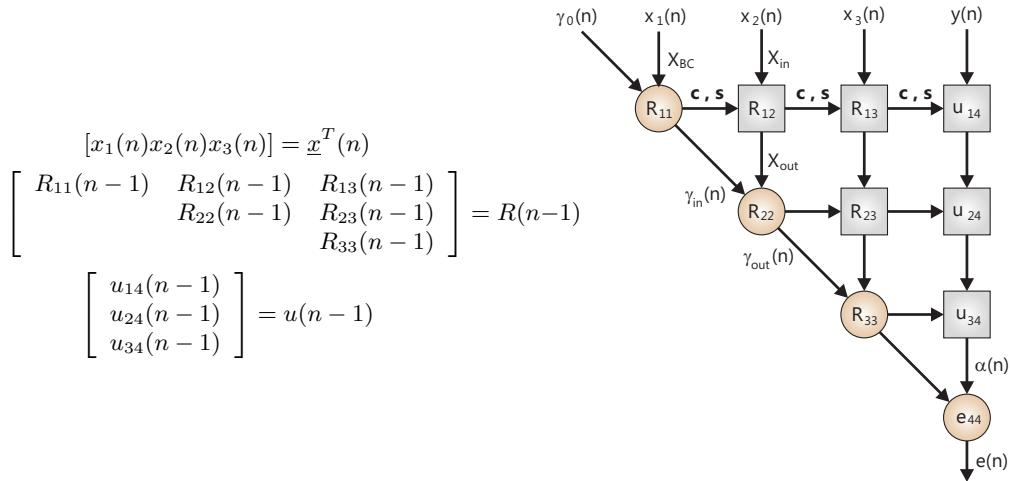
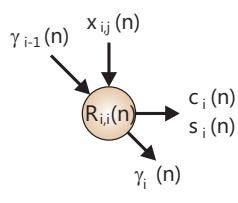


Figura 3.6: Esquema de mapeo directo

Esta arquitectura fue desarrollada por Gentleman y Kung y modificada por McWhirter. En la misma, el vector de información $\underline{x}^T(n)$ ingresa al sistema desde la parte superior del arreglo y es progresivamente eliminado al computar la rotación en cada fila de la matriz en proceso. Los parámetros c y s son calculados en una BC de forma que eliminan la entrada $x_{i,i}(n)$. Estos parámetros son posteriormente enviados a las ICs de la misma fila para actualizar las componentes según el resultado de la rotación aplicada. Los valores de salida de las ICs $x_{i+1,j}(n)$ se convierten en los valores de entrada para la próxima fila. Al mismo tiempo, una nueva entrada ingresa en la parte superior del arreglo y el proceso se repite. En el proceso, los valores de $R(n)$ y $u(n)$ son actualizados para registrar la rotación y luego almacenados en el arreglo para ser utilizados en el próximo ciclo.

Para el algoritmo RLS, la implementación del factor de olvido λ y el producto de cosenos γ debe ser incluido en las ecuaciones. Por lo tanto, las operaciones de BC e IC son modificadas acordemente. Una notación es asignada a las variables en el arreglo. Cada término R y u posee un sub-índice denotado por (i,j) , que representa la ubicación de los elementos en la matriz R y el vector u . Una notación similar es asignada a las entradas X y a las variables de salida. Las descripciones de las celdas para las BCs e ICs son presentadas en las figuras 3.7 y 3.8 respectivamente.



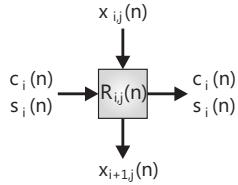
$$R_{i,i}(n) = \sqrt{\beta^2 R_{i,i}^2(n-1) + x_{i,i}^2(n)} \quad (3.5)$$

$$c_i(n) = \beta \frac{R_{i,i}(n-1)}{R_{i,i}(n)} \quad (3.6)$$

$$s_i(n) = \frac{x_{i,i}(n-1)}{R_{i,i}(n)} \quad (3.7)$$

Figura 3.7: Boundary Cell

$$\gamma_i(n) = c_i(n)\gamma_{i-1}(n-1)$$



$$x_{i+1,j}(n) = c_i(n)x_{i,j}(n) - s_i(n)\beta R_{i,j}(n-1) \quad (3.8)$$

$$R_{i,j}(n) = c_i(n)\beta R_{i,j}(n-1) + s_i(n)x_{i,j}(n) \quad (3.9)$$

Figura 3.8: Internal Cell

Se puede destacar que, si bien la implementación de este mapeo es simple y es posible lograr que las celdas operen en un esquema *pipeline*, la implementación requiere una gran cantidad de procesadores (en el ejemplo se tienen 10 para una matriz de 4×4) por lo cual ocupará un alto porcentaje de recursos del FPGA comparada con otras.

3.4.2. Mapeo de proyección horizontal

En la figura 3.9 se puede observar un ejemplo de un mapeo de celdas QR en una arquitectura lineal proyectando de izquierda a derecha en N procesadores. Existen dos conflictos con dicho mapeo.

En primer lugar, tanto las operaciones de BC como de IC están mapeadas en un mismo procesador. De la figuras 3.7 y 3.8 se puede observar que existen diferencias entre dichas operaciones. En segundo lugar, los procesadores de la arquitectura mapeada no son utilizados eficientemente, siendo únicamente el primero aprovechado a máxima capacidad. La eficiencia se reduce en la columna de procesadores, llevando a una eficiencia total en la región del 60 %, lo cual no es un óptimo uso de recursos.

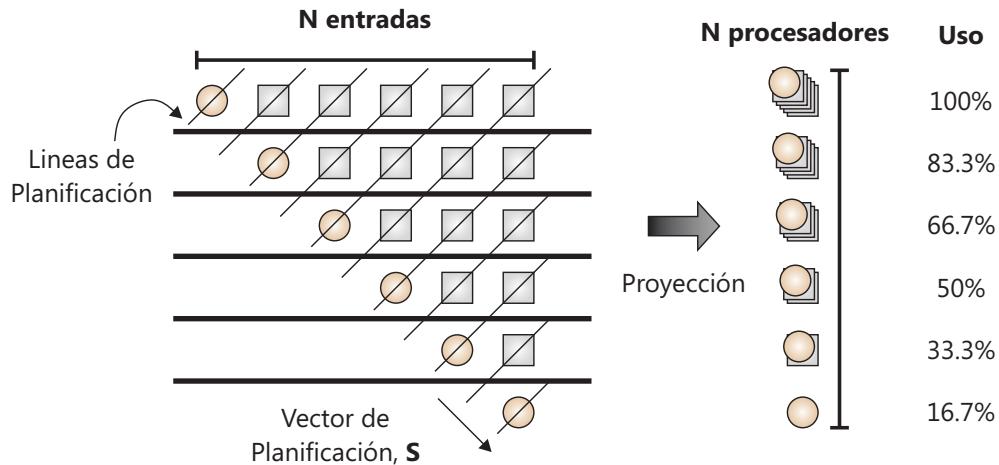


Figura 3.9: Esquema de Mapeo de Proyección Horizontal

3.4.3. Mapeo de Rader

Rader (1992,1996) produjo una arquitectura eficiente al manipular la forma triangular, antes de asignar las operaciones a procesadores, como se observa en la figura 3.10.

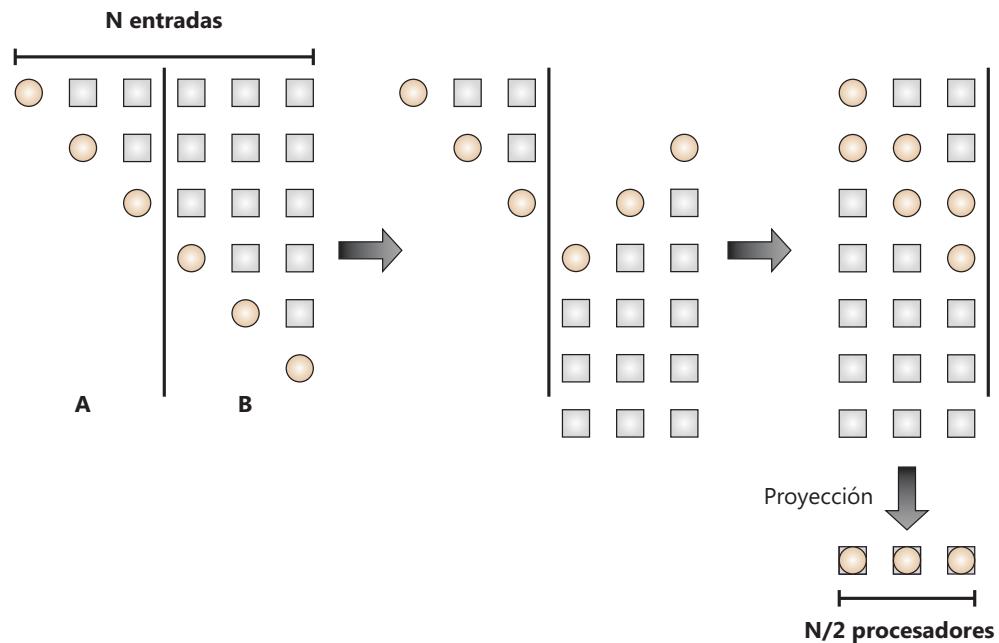


Figura 3.10: Esquema de Mapeo de Rader

La parte B del arreglo QR es espejada en el eje x y luego este resultado es espejado en el eje y para resultar en un arreglo de celdas rectangulares que puede ser mapeado hacia

abajo en una arquitectura lineal que consta de $N/2$ procesadores. Se puede observar que este mapeo presenta una mejora con respecto al mapeo de proyección horizontal, dado que los $N/2$ procesadores son utilizados el 100 % del tiempo. De todas maneras, los procesadores aun deben realizar ambas operaciones QR (BC e IC). Una opción para implementar este mapeo, es el uso de CORDIC, que logra arquitecturas similares tanto para BC como para IC.

3.4.4. Mapeo de Walke

Otro mapeo (Walke 1997 [11]), logra una arquitectura eficiente al mantener las operaciones BC e IC en procesadores diferentes. Esto es logrado al manipular el arreglo triangular de forma inteligente a través de diversas transformaciones para alinear todas las operaciones BC en una columna y el resto de las operaciones IC en otras.

Dichas transformaciones constan de doblar, espejar y rotar las partes del arreglo QR. Las mismas se explicarán en el siguiente ejemplo para un arreglo triangular de 7 entradas. El resultado mapea un arreglo triangular de $2m^2 + 3m + 1$ celdas ($N = 2m + 1$ entradas) en una arquitectura lineal, con interconexiones locales, consistiendo de un único procesador BC y m procesadores IC, todos utilizados al 100 % de eficiencia.

Los pasos a seguir para obtener el arreglo rectangular son los siguientes:

1. El arreglo triangular inicial es dividido en dos triángulos más chicos A y B. El corte A es realizado luego de la BC $m + 1^{ava}$ en una línea perpendicular a la diagonal de BCs.

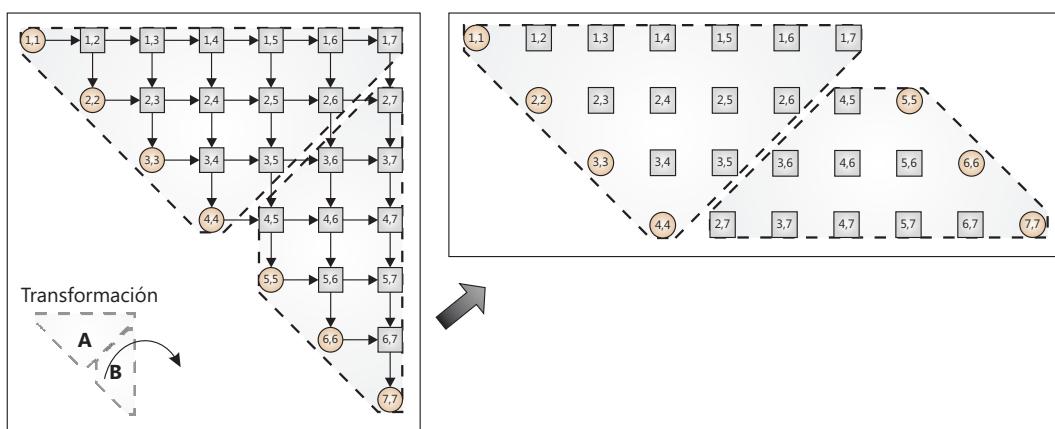


Figura 3.11: Esquema de Mapeo de Walke - Primera Transformación

2. El triángulo B debe ser manipulado para que pueda formar la parte superior del arreglo rectangular. Esto se logra en dos pasos. Al espejar el triángulo B en el eje x primero, las BCs son alineadas de forma que sean paralelas a las BCs en el triángulo

A, formando un paralelogramo, como se observa en la figura 3.12. El triángulo espejado B es luego movido hacia arriba a lo largo del eje y, hacia la izquierda en el sentido del eje x, para posicionarse en la parte superior de A, formando el arreglo rectangular. Como se puede observar, las operaciones BC están alineadas en dos columnas, por lo cual el arreglo rectangular debe ser aún modificado para ser adecuado para una proyección lineal.

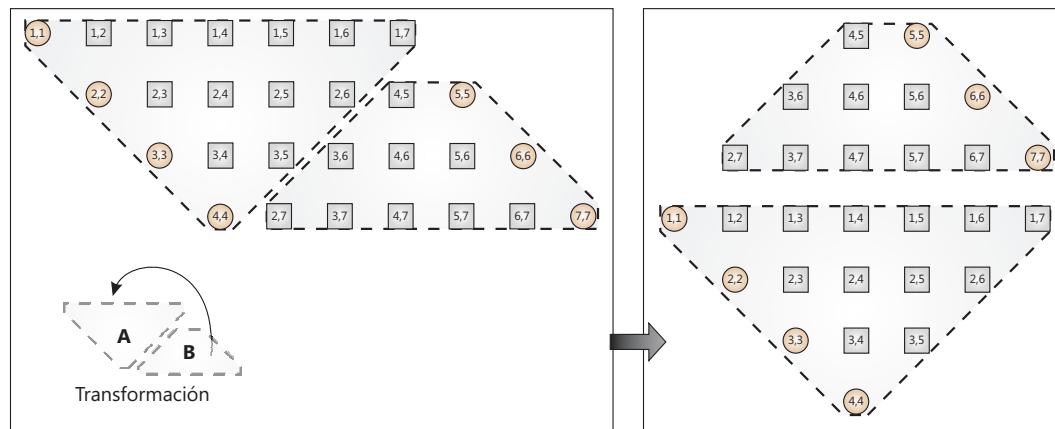


Figura 3.12: Esquema de Mapeo de Walke - Segunda Transformación

3. La siguiente etapa apunta a plegar el arreglo rectangular, de forma que las dos columnas de operaciones BC sean posteriormente alineadas a lo largo de una única. Este pliegue logra que las dos columnas de operaciones BC sean alineadas a izquierda.

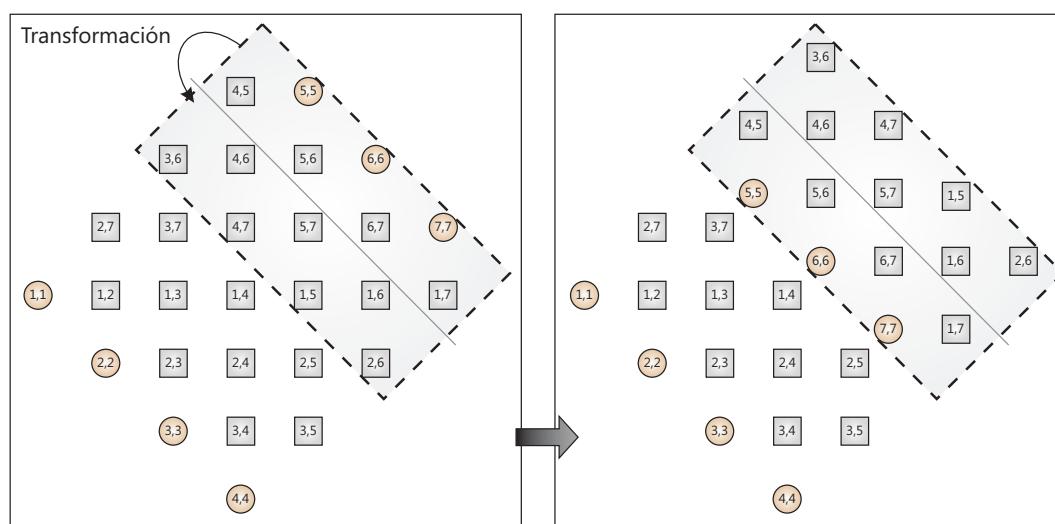


Figura 3.13: Esquema de Mapeo de Walke - Tercera Transformación

4. El último paso, consiste en intercalar las celdas de forma tal que se produzca un arreglo rectangular compacto. De este arreglo rectangular de procesador, se puede producir una arquitectura reducida al proyectar hacia abajo de la diagonal a un arreglo lineal, con todas las operaciones BC asignadas a un procesador de BC, y todas las operaciones de IC asignadas a una fila de m procesadores IC (ver figura 3.14).

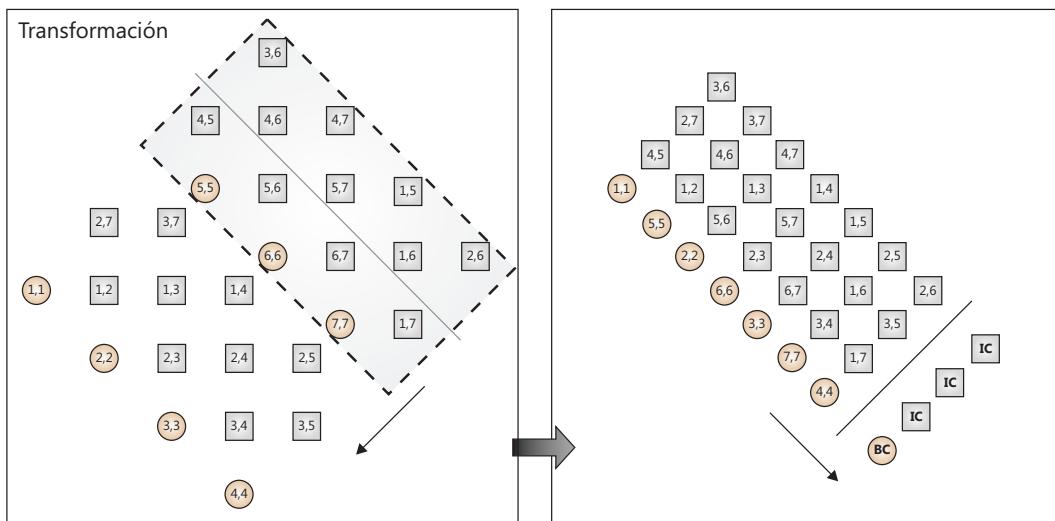


Figura 3.14: Esquema de Mapeo de Walké - Cuarta Transformación

La arquitectura lineal resultante se muestra con mayor detalle en la figura 3.15. Cada fila de procesadores dentro del arreglo rectangular de operaciones en la figura sigue la planificación vista anteriormente bajo la definición del vector de planificación o de *schedule s*, una flecha perpendicular a las líneas de planificación.

En esta etapa del análisis, se considera que cada celda de procesamiento toma un ciclo de *clock*. Existen registros presentes en todas las salidas de las celdas de procesamiento del arreglo lineal resultante para mantener la planificación. Se colocan multiplexores en las entradas de las celdas QR para controlar la entrada de datos, sea de las entradas del sistema o de las celdas adyacentes.

Los multiplexores inferiores definen las diferentes direcciones del flujo de datos que ocurre entre filas y el arreglo original. Las celdas QR del arreglo original guardan los valores de R de una iteración a la siguiente. Este mismo almacenamiento también necesita ser realizado en la arquitectura reducida, requiriendo entonces almacenar un número de valores R entre ciclos recursivos de celda por múltiples ciclos de *clock*.

Una solución es **mantener los valores localmente** dentro de los caminos de datos recursivos de las celdas QR, en lugar de utilizar una memoria externa. Los valores son almacenados en registros locales para retrasarlos hasta que sean necesarios.

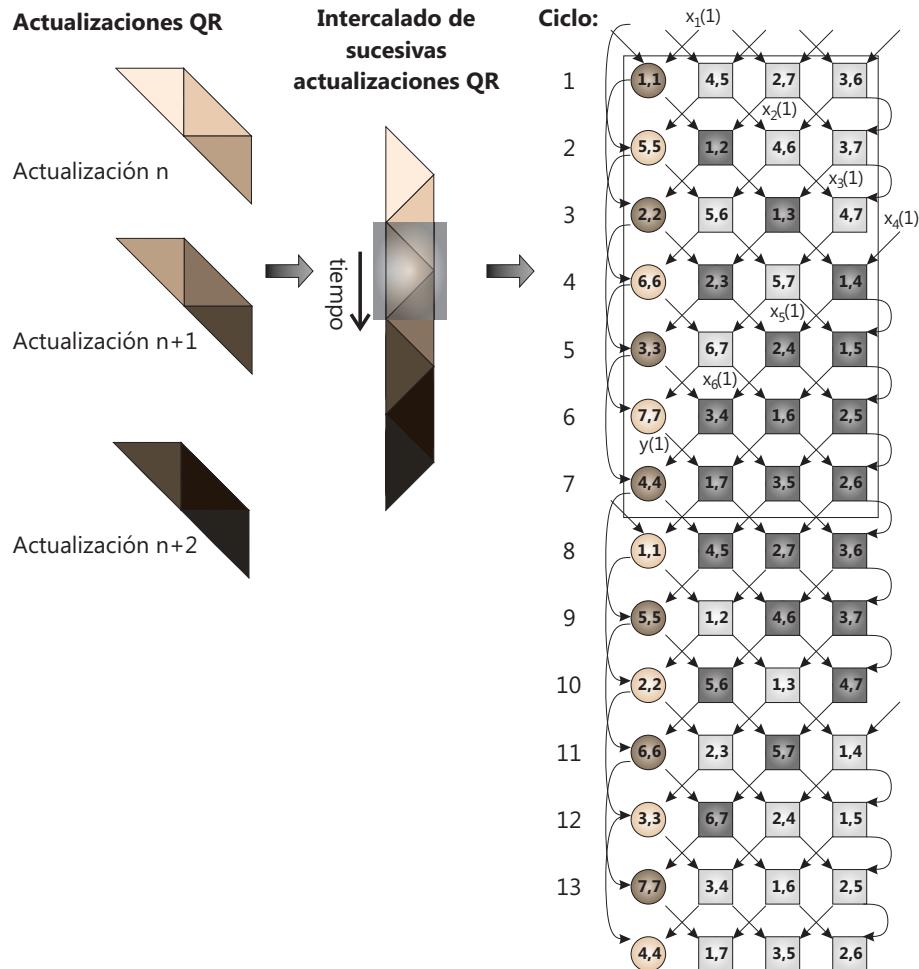


Figura 3.15: Mapeo de Walke - Iteraciones

3.4.5. Planificación de las operaciones QR

La derivación de la arquitectura es sólo una parte del desarrollo necesario. Una tarea más compleja es la determinación de una planificación válida que asegure que la información requerida para cada set de operaciones esté disponible en el momento de ejecución, mientras se siga manteniendo la eficiencia.

El arreglo rectangular de procesamiento de la figura 3.15 contiene todas las operaciones requeridas por el algoritmo QR, mostrando la secuencia en la cual se van a implementar en la arquitectura lineal. Por lo tanto, este diagrama puede ser utilizado para mostrar la planificación de las operaciones a ser realizadas en el diagrama lineal. Al ver la primera línea de planificación, se puede observar que las operaciones de **dos actualizaciones diferentes** han sido intercaladas. Las celdas oscurecidas representan la actualización

QR actual en tiempo n , y las no oscurecidas representan la actualización QR anterior no terminada, en tiempo $n - 1$. Efectivamente, las actualizaciones QR han sido intercaladas. La primera operación QR empieza en el ciclo = 1, luego después de $2m + 1$ ciclos de la arquitectura lineal, la siguiente operación QR empieza. De la misma forma, luego de $2m + 1$ ciclos, la tercera operación QR es iniciada. En total, toma $4m + 1$ ciclos de la arquitectura lineal para completar la primera actualización QR, y $2m + 1$ las siguientes.

Las celdas QR necesitan tener la posibilidad de tomar las entradas de x desde entradas externas del sistema, por ejemplo, desde muestras de información, formando la entrada de la matriz $x(n)$ y el vector $y(n)$, como se observa en la figura 3.15. Las entradas externas son alimentadas en la arquitectura lineal cada $2m + 1$ ciclos de reloj. Las mismas también van a tomar entradas que son internas al arreglo lineal.

Si cada celda QR toma un único ciclo de *clock* para producir una salida, luego no existirá una violación a la planificación expuesta en la figura 3.5. De todas formas, se deben realizar análisis adicionales dado que las celdas QR poseen requerimientos de *timing* detallados. Un análisis de *timing* de mayor profundidad es realizado en el Capítulo 5.

3.5. Implementación de las operaciones BC e IC

3.5.1. El algoritmo CORDIC

El algoritmo CORDIC provee un método iterativo para realizar rotaciones de vectores en diferentes ángulos utilizando únicamente sumas y desplazamientos [9]. El algoritmo, acreditado a Volder [10], es derivado de la transformación de rotación general de Givens.

El análisis teórico del algoritmo CORDIC puede encontrarse en el Apéndice C.

Los algoritmos de rotación y vectorización CORDIC como fueron planteados originalmente están limitados a ángulos de rotación entre $-\pi/2$ y $\pi/2$. Esta limitación es debida al uso de 2^0 como el primer valor de tangente de iteración. Para ángulos compuestos mayores a $\pi/2$ en módulo, una rotación adicional es requerida. Un enfoque, consiste en aplicar una rotación inicial de π para estos casos, el cual será adoptado en el presente trabajo.

Esta reducción asume una representación en módulo 2π del ángulo de entrada. La implementación de las operaciones BC e IC requieren utilizar los modos rotación y vectorización de CORDIC.

3.6. Análisis de publicaciones

Durante la realización del presente trabajo, se hizo una investigación enfocada en publicaciones presentadas en el ámbito de la implementación de filtros de *beamforming* y procesadores de descomposición QR. Se analizaron las técnicas implementadas en cada uno de ellos y se tomaron como referencia para el planteo de métricas que permitieran una posterior comparación de resultados.

3.6.1. Publicación de Altera

Título: *Implementation of CORDIC-Based QRD-RLS Algorithm on Altera Stratix FPGA with Embedded Nios Soft Processor Technology*

Autor: Altera

En este trabajo [13] se presenta la realización de dos tipos de hardware diferentes: la implementación de un procesador de descomposición QR en hardware y la implementación de un software de sustitución sobre un *soft processor* de Altera llamado Nios. El uso conjunto de los dos sistemas implementa un filtro de *beamforming*.

Si bien el documento no entra en grandes niveles de detalle, se presentan métricas para 3 mapeos diferentes (directo, mezclado y discreto) en hardware para la descomposición de una matriz de 64×9 , con entradas complejas de ancho de palabra de 16 bits. Cada celda compleja es implementada con 3 módulos CORDIC reales, a diferencia de uno solo, que es lo requerido para celdas reales.

Si bien dicho hardware presenta grandes diferencias con respecto al que se implementó en esta tesis, se tienen en cuenta los resultados como parte de la información disponible de análisis.

Implementation Technique	CORDIC usage		Throughput		Cost LE/Update
	No. of Blocks	No. of LEs	Update Delay (us)	Updates/s	
Direct Mapping	54	70200	5.1	196078	0.358
Mixed Mapping	4	5200	250.85	3986	1.305
Discrete Mapping	2	2600	198.11	5047	0.515

Figura 3.16: Tabla de métricas del hardware de Altera

Entre las diferentes métricas, se encuentran las siguientes:

Update Delay: Tiempo requerido antes de que todas las celdas en el arreglo sistólico sean actualizadas.

Throughput: Número de matrices de entrada (cada una de $M \times N$) que son procesadas por segundo ($= 1/update\ delay$).

Cost: Número de celdas básicas LE¹ que ocupan las celdas CORDIC.

3.6.2. Publicación de Xilinx

Título: *FPGA Implementation of Matrix Inversion Using QRD-RLS Algorithm*

Autor/es: Marjan Karkooti, Joseph R. Cavallaro, Chris Dick. Center for Multi-media Communication, Department of Electrical and Computer Engineering, Rice University / Xilinx

Este trabajo [14] detalla la realización de un filtro de *beamforming* y fue desarrollado por un laboratorio de investigación de Xilinx en conjunto con la Universidad Rice. Su hardware utiliza dos CORDICs en *vectoring mode* para cada una de las *boundary cells* y la arquitectura de los mismos, a diferencia del hardware de la presente tesis, es desenrollada.

Para las *internal cells*, la rotación no es implementada en hardware a través de un módulo CORDIC, sino que utiliza *multiply accumulate (MAC) functional units*. Se utiliza el dispositivo FPGA Virtex 4, el cual consta de un gran arreglo de unidades MAC referidas como *DSP48 slices*. Según menciona el documento, utilizar los bloques embebidos DSP48 en reemplazo de un enfoque basado en CORDIC para las *internal cells* reduce la latencia de esta fase de cómputo y minimiza la cantidad de tablas de lógica FPGA (*look-up tables LUTs* y registros) requeridas para la implementación. De todas maneras, se considera importante destacar el contexto de dicho tamaño dado que la arquitectura CORDIC es desenrollada.

Un dato interesante que presenta el trabajo para la comparativa es el tiempo requerido para el cálculo de la descomposición de una matriz utilizando varios valores de M y N , entre los cuales se encuentra la matriz de 7×7 . Sin embargo, no especifica la cantidad de bits de ancho de palabra utilizados.

Adicionalmente, detalla que se desarrolló un banco de pruebas con MATLAB. El script simula un objetivo dinámico y genera las muestras del patrón de radiación en campo lejano para el objetivo en movimiento. Las muestras del campo eléctrico en cada sensor se generan en MATLAB y son enviadas al procesador FPGA QRD. Se produce una nueva estimación del vector de peso conformador de haz y se envía a MATLAB para su posterior procesamiento.

¹Logic Element: Los elementos lógicos son las unidades más pequeñas de lógica en la arquitectura de la familia de dispositivos FPGA Cyclone III de Altera.

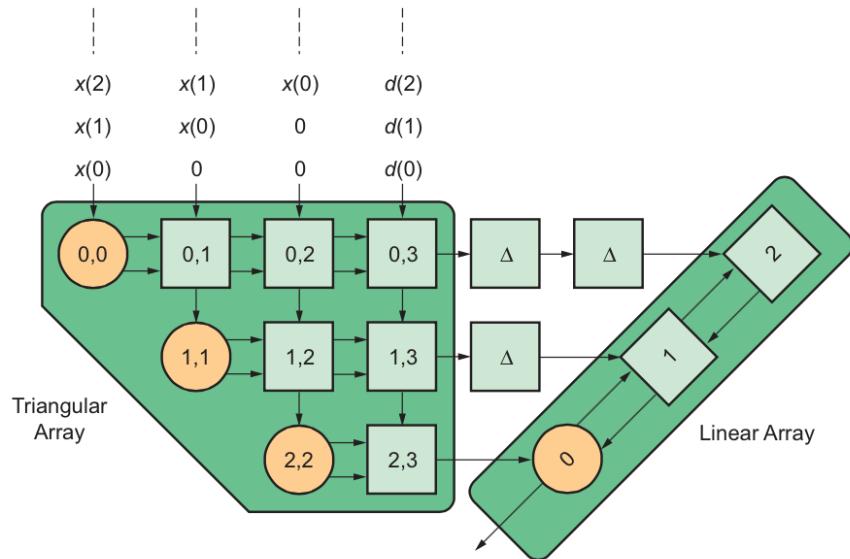


Figura 3.17: Arquitectura implementada por Xilinx

M	N	Cycles for Triangularization	Cycles for Back Substitution	Total Cycles	Time (μs) for 250-MHz Clock
3	3	792	147	939	3.76
8	3	2,112	147	2,259	9.04
5	5	2,540	255	2,795	11.18
9	5	4,572	255	4,827	19.31
7	7	5,656	371	6,027	24.11
10	7	8,080	371	8,451	33.80
9	9	10,476	495	10,971	43.88
11	9	12,804	495	13,299	53.20
10	10	13,630	560	14,190	56.76

Figura 3.18: Tabla de métricas del hardware presentado por Xilinx

3.6.3. Publicación de Universidad de Victoria

Título: *Fixed-Point CORDIC-Based QR Decomposition by Givens Rotations on FPGA*

Autores: Dongdong Chen, Mihai SIMA. Department of Electrical and Computer

Engineering, University of Victoria

Este trabajo [15] contiene definiciones precisas sobre cómo fue desarrollado el hardware, cómo fue evaluado, y las diferentes métricas obtenidas. El hardware consta de un procesador de descomposición QR para matrices de 4×4 , con ancho de palabra variable. Se utilizó un **mapeo directo** para desarrollar la arquitectura QR, la cual consta de un total de 21 procesadores CORDIC.

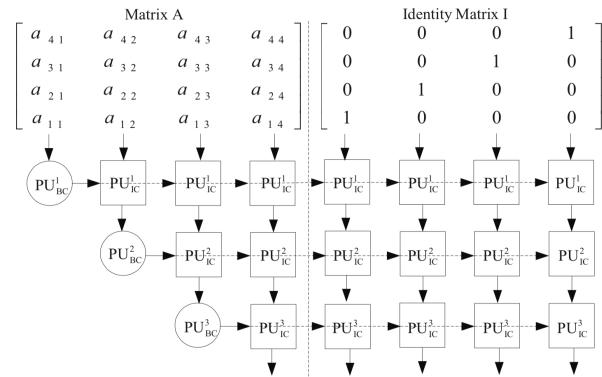


Figura 3.19: Arreglo de descomposición QR para una matriz de 4×4

Una diferencia sustancial con la arquitectura que fue descripta, es que en ésta se agrega una sección de celdas/procesadores a derecha, la cual al ser inicializada con la matriz identidad, permite que al finalizar el cálculo se llegue al resultado de la matriz Q^T (localizada en dichas celdas). Esta sección está compuesta por $3 \times 4 = 12$ celdas, siendo las 9 celdas restantes utilizadas para el cálculo de la matriz R .

Se presenta un análisis teórico del *timming* de la unidad, y una evaluación del error para diferentes anchos de palabra.

En forma similar al proyecto de Xilinx, desarrollaron un banco de pruebas basado en una simulación de MATLAB para contrastar resultados, utilizando un dispositivo FPGA Virtex 5 para realizar la síntesis, y sometiéndolo al cálculo de 100.000 matrices con elementos aleatorios.

Entre las métricas presentadas se encuentran distintos parámetros de evaluación del error, número de *slices* ocupados, *delay* de camino crítico, tiempo de procesamiento y *throughput*.

3.7. Arquitectura implementada

La elección de la implementación utiliza RLS-QRD (algoritmo RLS a través de descomposición QR) como el sistema central para calcular adaptativamente los pesos del

Error evaluation	Matrix R						Matrix Q					
	18	20	22	24	26	28	18	20	22	24	26	28
Word length (<i>FB-bit</i>)	6.32	5.28	1.42	1.32	1.32	0.92	6.77	5.80	1.76	1.60	1.47	1.58
Max. absolute error (10^{-3})	5.39	1.99	1.41	1.34	1.32	1.33	5.09	1.69	1.12	1.06	1.06	1.06
Mean absolute error (10^{-5})	5.47	2.87	1.16	1.13	1.11	0.95	5.85	2.95	1.30	1.26	1.24	1.24
Stand. deviation (10^{-5})	2910	312	109	95	94	93	3424	266	134	113	111	109
Worst cases (<i>No.</i>)												
Device	Xilinx Virtex5 XC5VTEX150T											
Word length (<i>FB-bit</i>)	18	20	22	24	26	28						
No. of occupied slices LUTs (%)	7,811 (8%)	8,593 (9%)	9,122 (9%)	9,758 (10%)	10,512 (11%)	11,513 (12%)						
No. of occupied slices (%)	2,609 (11%)	2,684 (11%)	2,810 (13%)	3,070 (13%)	3,379 (14%)	3,617 (15%)						
Critical path delay (<i>ns</i>)	8.70	8.78	8.82	8.97	9.21	9.27						
Processing time (<i>us/update</i>)	0.469	0.474	0.476	0.484	0.497	0.500						
Throughput (<i>M/sec</i>)	2.13	2.11	2.10	2.06	2.01	2.00						

Figura 3.20: Tabla de métricas del hardware presentado

filtro.

A continuación se definen las premisas de la arquitectura elegida para la implementación del procesador de descomposición QR, sobre la cual se hará hincapié en el próximo capítulo:

- **Tipo de Algoritmo:** Algoritmo Recursive Least Squares a través de descomposición QR.
- **Tipo de Rotaciones:** Rotaciones de Givens.
- **Hardware de Rotación:** CORDIC iterativo.
- **Cantidad de Procesadores de Celdas:** 4 módulos CORDIC.
- **Distribución de Celdas:** Mapeo de Walke [11].
- **Dimensión de la matriz:** 7×7 .
- **Ancho de Palabra:** Hasta 64 bits.

Capítulo 4

Implementación a nivel de microarquitectura

En el capítulo anterior se analizaron diferentes tipos de arquitecturas existentes para la implementación de un procesador de descomposición QR. Se describieron sus características a nivel conceptual y se detallaron las definiciones de la arquitectura elegida para realizar la implementación.

En este capítulo se describen las herramientas que fueron utilizadas y se explica en detalle, a nivel de microarquitectura, la forma en la cual fue implementado cada módulo del procesador desarrollado, y la forma en la cual los módulos fueron interconectados para conformar al mismo. Adicionalmente, se describen aquellos desarrollos que fueron el resultado de la presente tesis, utilizados para resolver diferentes necesidades encontradas, los cuales representan un contenido intelectual propio.

La metodología utilizada para el desarrollo del procesador consistió en analizar las necesidades de la arquitectura elegida, y los diferentes componentes requeridos. Se procedió a la definición de dicha arquitectura a través de un diagrama en bloques y se desarrolló cada uno de los módulos por separado, gradualmente. Una vez desarrollado un módulo, se procedió a simular su comportamiento aislado a través de un *testbench* que lograra contemplar el mayor conjunto de los posibles vectores de entrada. Una vez finalizada la simulación individual de los distintos componentes, se procedió a realizar una simulación conjunta de algunos de ellos.

El detalle del código fuente Verilog puede encontrarse en el Apéndice A.

4.1. Diagrama en bloques

A continuación se expone un diagrama en bloques a grandes rasgos del hardware desarrollado:

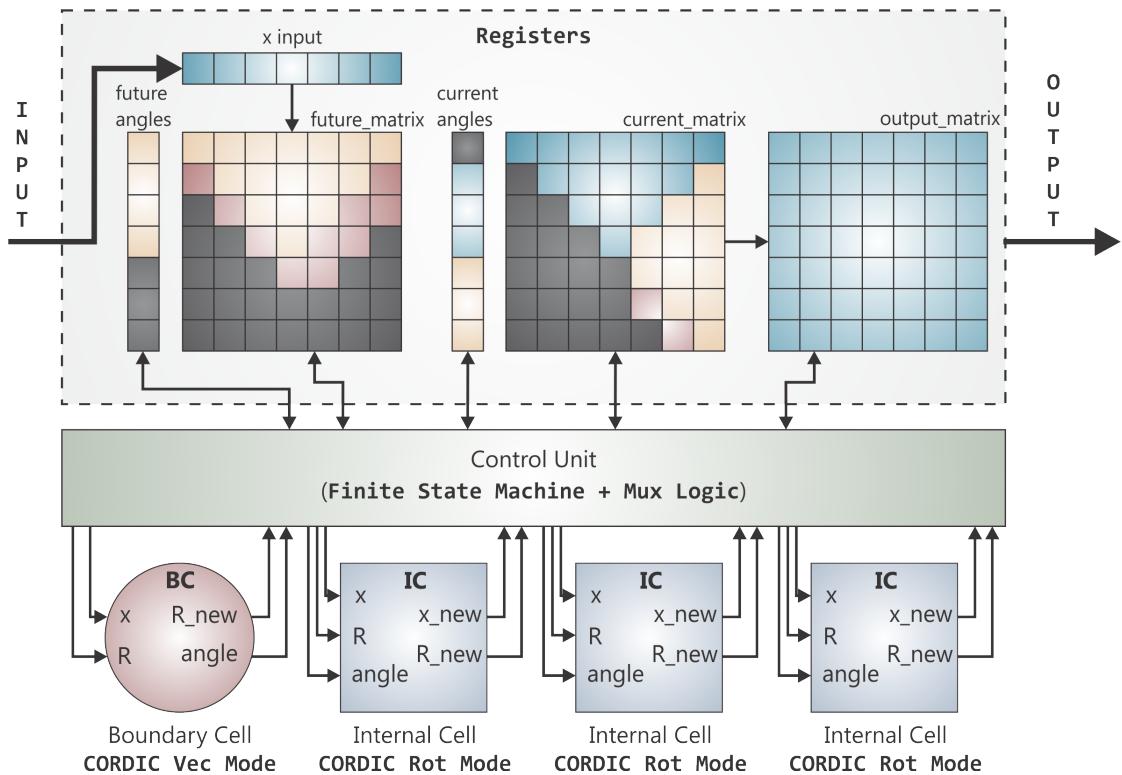


Figura 4.1: Diagrama en Bloques del Hardware Desarrollado

En el diagrama se pueden observar las unidades desarrolladas para implementar la arquitectura elegida. Las mismas pueden dividirse en 3 grupos:

- La unidad de registros contiene el hardware requerido para almacenar los valores de entrada y salida de los procesadores, registrando cada uno de los cálculos realizados. Los datos se encuentran organizados en formato matricial. La unidad fue cuidadosamente desarrollada para adaptarse al comportamiento multiprocesamiento de la arquitectura, y para realizar operaciones de desplazamiento requeridas durante el procesamiento del algoritmo.
- La unidad de control es el bloque de hardware que implementa la lógica necesaria para llevar a cabo los pasos del algoritmo según el mapeo de Walke, teniendo en cuenta el vector y las líneas de planificación. Esto se logra a partir del uso de una

máquina de estados finita y multiplexores, para direccionar adecuadamente, según el estado actual, las entradas y salidas del hardware.

- Los procesadores de *boundary cell* e *internal cell* son implementados utilizando núcleos CORDIC. En el caso de las operaciones BC, se utiliza CORDIC en *vector-ring mode*, y en el caso de las operaciones IC, el modo utilizado es *rotation mode*. La *boundary cell* es la unidad que toma como entrada los valores de R y x y realiza la rotación de los mismos para obtener el módulo del vector, el cual representa el nuevo valor de R , y el ángulo del mismo, el cual es utilizado para enviarlo como entrada a las *internal cells* de la misma fila. Por otro lado, las *internal cells* toman como entrada los valores de R , x , y el ángulo calculado por las *internal cells*, para dar como resultado el nuevo valor de R , y el valor de x para enviar como entrada a la *internal cell* de la siguiente fila.

La descripción individual de cada uno de los componentes, empezando desde los módulos más básicos y continuando hasta evolucionar en el *top level*, se encuentra en las siguientes secciones.

4.2. Parámetros comunes a los distintos módulos

- WORD_WIDTH: Es el parámetro utilizado para definir el ancho de palabra de los datos que representan cada componente de la matriz. El valor por defecto es 16 bits y puede soportar hasta 64. El número de iteraciones depende directamente de este valor, por lo cual al aumentar la precisión, aumenta el tiempo de procesamiento requerido.
- ROWS: Es el parámetro con el cual se define la cantidad de filas de la matriz. Actualmente, la implementación sólo admite que este valor sea 7.
- COLUMNS: Es el parámetro con el cual se define la cantidad de columnas de la matriz. Actualmente, la implementación sólo admite que este valor sea 7.

4.3. Pre-procesamiento de entradas para el módulo CORDIC

El código **preprocessor.v** describe el hardware desarrollado para hacer el pre-procesamiento de las entradas $x;y;z$ requerido, antes de enviarlas al procesador CORDIC. Es la primera etapa del módulo de rotación “*rotator*”. Según fue descripto en la sección 3.5.1, el procesador CORDIC sólo puede manejar rotaciones entre $-\pi/2$ y $\pi/2$ en cualquiera de sus dos modos de operación. Para aquellos casos en los cuales es necesario hacer rotaciones con ángulos mayores a $\pi/2$ en módulo, es requerido aplicar una rotación inicial de π .

En el modo rotación, esta transformación se logra negando ambas entradas, x e y , y enviándolas al procesador CORDIC. Dicha transformación logra la rotación inicial de π deseada, siendo la rotación restante manejada por el procesador CORDIC. El resultado correcto es extraído directamente desde las salidas del mismo.

En el modo vector, es necesario abordar la transformación desde dos bloques distintos. En primer lugar, se detecta si el vector de entrada posee un ángulo mayor a $\pi/2$ en módulo observando el signo de la componente x . Un ejemplo de dicho caso podría ser el vector $(-1000; 1000)$. Estos casos se identifican cuando el signo de x es negativo. En dicho caso, se aplica la misma transformación sobre x e y que se utilizó en el modo rotación, negando las componentes y enviándolas al modulo CORDIC. En segundo lugar, se debe tomar el ángulo de salida de CORDIC z_o , y restarle π . Esta operación, es manejada por el módulo superior **rotator.v**.

Para realizar la implementación, se utiliza un *bit* de decisión, el cual según el modo de CORDIC elegido, define qué parámetro será utilizado para determinar si es necesario modificar las entradas:

```
1 | vec_rot_n ? d_i <= x_in[WORD_WIDTH-1] : ^z_in[WORD_WIDTH:WORD_WIDTH-1];
```

Luego, con el bit de decisión se controlan las entradas x e y y z siguiendo la siguiente lógica:

```
1 | Si d_i se encuentra en estado alto entonces
2 |   /* Angulo fuera del dominio valido de CORDIC */
3 |   Asignar x de CORDIC con -x_i y extender 1 bit el signo
4 |   Asignar y de CORDIC con -y_i y extender 1 bit el signo
5 |   Si el modo es rotacion:
6 |     Asignar z de CORDIC con z_i - pi
7 |   sino
8 |     Asignar z de CORDIC con z_i
9 |   sino
10 |  /* Angulo dentro del dominio valido de CORDIC */
11 |  Asignar x de CORDIC con x_i y extender 1 bit el signo
12 |  Asignar y de CORDIC con x_i y extender 1 bit el signo
13 |  asignar z de CORDIC con z_i
```

Al observar el pseudocódigo es posible identificar que se extienden en un *bit* las entradas del sistema. El motivo de dicha extensión se basa en que, debido a la ganancia de CORDIC, pueden darse casos de overflow para ciertas entradas, y así llegar a resultados erróneos. Para evitarlo, la solución elegida consistió en:

1. Extender las entradas en un *bit* en la etapa de pre-procesamiento.
2. Realizar el cálculo de CORDIC.
3. Atenuar la ganancia de CORDIC en el módulo de post multiplicación.
4. Eliminar el *bit* de signo.

De esta forma se adecúa el tamaño de las entradas del procesador CORDIC, evitando la posibilidad de llegar a una condición de overflow.

4.4. CORDIC iterativo

El código **iterative_cordic.v** describe el hardware desarrollado para implementar el algoritmo de CORDIC según se encuentra descripto en el Apéndice C en su versión iterativa. Lo que se requiere es lograr un hardware que replique las ecuaciones planteadas en C.7. La composición del hardware se encuentra diagramada en la figura 4.2 y se logra como sigue:

- Por un lado, se tiene un contador que registra el número de iteración dentro del algoritmo. La cantidad de cuentas de dicho contador depende directamente del ancho de palabra utilizado.
- Se utiliza un *bit* de decisión *di*, el cual trabaja utilizando el signo de *z* o de *y* según el modo sea rotación o vectorización.
- Cuando el hardware se encuentra en el estado IDLE, las entradas *x*, *y*, *z*, ingresan en los registros de acumulación. Al salir del estado IDLE, el acumulador es actualizado con el contenido de la salida del circuito en cada iteración. Esta decisión es implementada por un multiplexor para cada componente *x*, *y*, *z*.
- *x* e *y*, son direccionadas a registros de desplazamiento que operan según el número de iteración (número de *bits* desplazados = número de iteración).
- Se agregan dos circuitos *add/subtract* para implementar las dos ecuaciones de CORDIC. La elección de suma o resta es regida por el *bit* de decisión. Los términos para cada uno son (*x;y* desplazado) por un lado, e (*y;x* desplazado) por el otro.
- Los valores de arcotangentes elementales son almacenados en una ROM. El acceso a la misma es controlado por el contador de iteraciones del algoritmo. La representación binaria se logra al multiplicar los valores absolutos por una constante para normalizarlos. Se utilizó una constante que normaliza los valores de la tabla para una representación en 64 *bits*, el cual es el máximo valor aceptado por el hardware. En caso de que se utilicen longitudes de palabra menores, se hace un desplazamiento de dichas constantes a derecha (dividir por 2^n) una cantidad $n = 64 - \text{longitud de palabra}$.

Iteración	Valor de Arcotangente	Representación Decimal de 64 bits
1	0,785398163397448000000000	4611686018427387904
2	0,463647609000806000000000	2722437224269746380
3	0,244978663126864000000000	1438461061161762076
4	0,124354994546761000000000	0730185295051043895
5	0,062418809995957400000000	0366509582986589657
6	0,031239833430268300000000	0183433460584072433
7	0,015623728620476800000000	0091739112965426259
8	0,007812341060101110000000	0045872355853501789
9	0,003906230131966970000000	0022936527896151662
10	0,001953122516478820000000	0011468307695752815
11	0,000976562189559319000000	0005734159316382967
12	0,000488281211194898000000	0002867080341756270
13	0,000244140620149362000000	0001433540256323779
14	0,000122070311893670000000	0000716770138842597
15	0,00006103515617420880000	0000358385070756387
16	0,00003051757811552610000	0000179192535545079
17	0,00001525878906131580000	0000089596267793400
18	0,00000762939453110197000	0000044798133899308
19	0,00000381469726560650000	0000022399066949980
20	0,00000190734863281019000	0000011199533475031
21	0,00000095367431640596100	0000005599766737520
22	0,00000047683715820308900	0000002799883368761
23	0,00000023841857910155800	0000001399941684380
24	0,00000011920928955078100	0000000699970842190
25	0,00000005960464477539060	0000000349985421095
26	0,00000002980232238769530	0000000174992710547
27	0,00000001490116119384770	0000000087496355274
28	0,00000000745058059692383	0000000043748177637
29	0,00000000372529029846191	0000000021874088818
30	0,00000000186264514923096	0000000010937044409
31	0,00000000093132257461548	0000000005468522204
32	0,00000000046566128730774	0000000002734261102

Tabla 4.1: Tabla de Arcotangentes

- La salida de z se implementa independientemente en forma similar a la de x e y , sumando o restando ángulos según el *bit* de decisión, pero tomando como segundo término de la suma/resta el valor de la ROM de arcotangentes, según el valor para la iteración correspondiente.

El diagrama del circuito de CORDIC implementado es el siguiente:

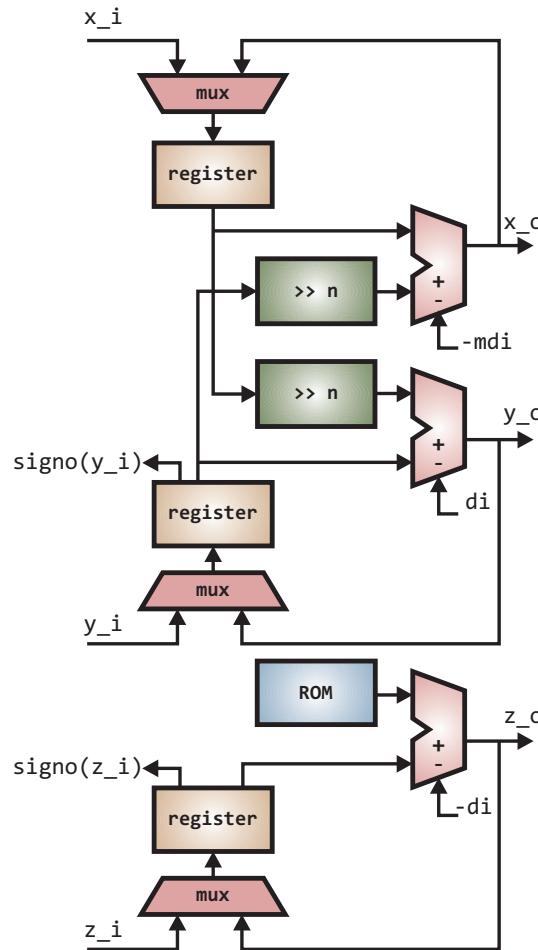


Figura 4.2: Diagrama RTL de la implementación de CORDIC en hardware

4.5. Post-multiplicador de salidas x e y de CORDIC

El código **postmultiplier.v** describe el hardware desarrollado para hacer la atenuación requerida para compensar la ganancia de CORDIC. Es la última etapa del módulo de rotación “*rotator*”, y se aplica a las salidas de x e y de CORDIC. Segundo fue descripto en

la sección 3.5.1, el hardware CORDIC introduce una ganancia A_n al aplicar la rotación. Como en el hardware implementado es requerido que los vectores rotados no se vean escalados por la misma, se debe implementar una post-multiplicación que la atenúe.

Para implementar dicha multiplicación se hizo uso de la primitiva de producto del lenguaje Verilog, siendo la implementación definida al momento de la síntesis.

4.6. Unidad de rotación completa

El código **rotator.v** describe el hardware desarrollado para implementar un rotador el cual, utilizando los módulos de CORDIC, pre-procesamiento y post-multiplicación, sea capaz de manejar las operaciones BC e IC correctamente, sin importar el ángulo de los vectores de entrada (conformados por los valores de R y x) y sin producir escalamientos. Para lograrlo, dicho hardware instancia un pre-procesador, un módulo CORDIC iterativo, dos post-multiplicadores (uno para x y otro para y), y adicionalmente contiene el hardware de ajuste de ángulo requerido para los casos en que se utilice el modo vectorización y el vector de entrada tenga un ángulo mayor $\pi/2$ en módulo.

A continuación se expone un diagrama en bloques de la composición del rotador:

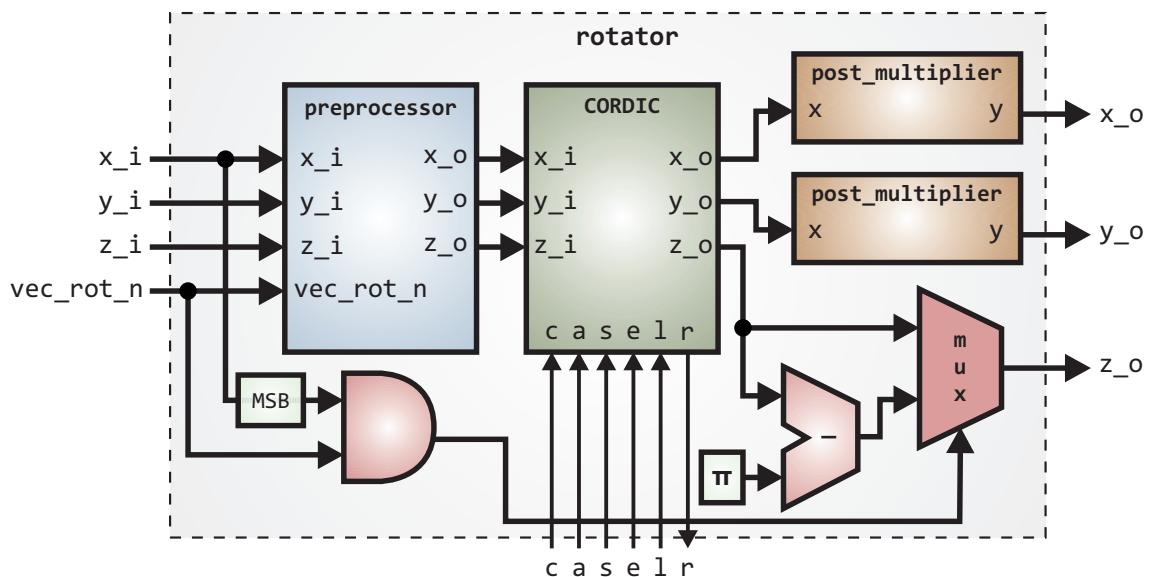


Figura 4.3: Diagrama en Bloques del Módulo de Rotación

4.7. Procesador boundary cell

El código **boundary_cell.v** describe el hardware desarrollado para implementar las operaciones BC del procesador de descomposición QR. Las mismas fueron descriptas en la sección 3.2 y se rigen por las ecuaciones 3.5, 3.6 y 3.7. Para lograr la implementación de dichas ecuaciones, en el mismo se instancia un módulo rotador y se direccionan adecuadamente los valores de R y x de entrada, y los valores de R_{new} y $angle$ de salida. El modo de operación del rotador se define de forma constante como modo vectorización.

4.8. Procesador internal cell

El código **internal_cell.v** describe el hardware desarrollado para implementar las operaciones IC del procesador de descomposición QR. Las mismas fueron descriptas en la sección 3.2 y se rigen por las ecuaciones 3.8 y 3.9. Para lograr la implementación de dichas ecuaciones, en el mismo se instancia un módulo rotador y se direccionan adecuadamente los valores de R , x y $angle$ de entrada, y los valores de R_{new} y x_{new} de salida. El modo de operación del rotador se define de forma constante como modo rotación.

4.9. Procesador de descomposición QR

El código **qr_processor.v** es el *top module* del proyecto y describe el hardware desarrollado para implementar el procesador de descomposición QR. A través de la definición de la unidad de registros, la lógica de control y la instancia e interconexión de los módulos de celda que fueron descriptos, el mismo logra la implementación en hardware del algoritmo descripto.

La organización del código está distribuida como sigue:

1. Definición del *layout* del módulo, el cual incluye sus parámetros, entradas y salidas.
2. Definición de los estados de control.
3. Definición de registros e interconexiones.
4. Instancia de circuitos (*boundary_cell* e *internal_cell*).
5. Definición de la arquitectura.

4.9.1. Interfaz del módulo

A continuación se describen las entradas y salidas del módulo:

Entradas:

- **clk** : Entrada de *clock* del sistema. El hardware opera por detección de flanco ascendente.
- **arst** : *Reset* asincrónico activo en alto.
- **srst** : *Reset* sincrónico activo en alto.
- **enable** : *Enable* sincrónico activo en alto.
- **start** : *Flag* para indicar que se está suministrando un nuevo vector de entrada, el cual dispara el cálculo para una nueva matriz R.
- **lambda_option** : Selector de factor de olvido.
- **x_in_1** : Primer elemento del vector de entrada.
- **x_in_2** : Segundo elemento del vector de entrada.
- **x_in_3** : Tercer elemento del vector de entrada.
- **x_in_4** : Cuarto elemento del vector de entrada.
- **x_in_5** : Quinto elemento del vector de entrada.
- **x_in_6** : Sexto elemento del vector de entrada.
- **x_in_7** : Séptimo elemento del vector de entrada.

Salidas:

- **ready** : Cuando se encuentra en estado alto, indica que el resultado se encuentra disponible. La matriz de salida va a ser enviada, de a una fila por ciclo de clock, en las salidas **y_out** donde cada una de ellas representa cada columna.
- **y_out_1** : Primera columna de la fila correspondiente de la matriz de salida.
- **y_out_2** : Segunda columna de la fila correspondiente de la matriz de salida.
- **y_out_3** : Tercera columna de la fila correspondiente de la matriz de salida.
- **y_out_4** : Cuarta columna de la fila correspondiente de la matriz de salida.
- **y_out_5** : Quinta columna de la fila correspondiente de la matriz de salida.
- **y_out_6** : Sexta columna de la fila correspondiente de la matriz de salida.
- **y_out_7** : Séptima columna de la fila correspondiente de la matriz de salida.

4.9.2. Uso del procesador QR

La implementación del procesador desarrollada es capáz de calcular la descomposición QR de una matriz de 7×7 . La salida es la matriz R , la cual contiene la información requerida de la matriz de entrada. La matriz Q forma una base ortogonal que al ser multiplicada por R equivale a la matriz de entrada.

Si bien es posible computar la matriz Q haciendo uso de los ángulos de salida calculados en las *boundary cells*, o agregando hardware de triangularización adicional, no se realiza dicha implementación dado que la misma no es requerida para la aplicación elegida, en la cual el objetivo es el cálculo del vector de pesos de un *beamformer*.

En esta aplicación, las columnas de la matriz de entrada se generan por las últimas 7 muestras de 6 antenas (primeras 6 columnas, vector x) y una secuencia de entrenamiento (última columna, elemento y).

Dado el carácter recursivo de la implementación, para suministrar la matriz de entrada es necesario hacerlo de a una fila por vez. Se comienza suministrando la primera fila, activando el *flag start*, y esperando a que la señal de *ready* se coloque en estado en alto antes de suministrar la próxima fila. Se presenta en la figura 4.4 un diagrama de tiempos al insertar una primera fila $A = [a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6 \ a_7]$. Dicho proceso se debería repetir con las 6 filas restantes de la matriz.

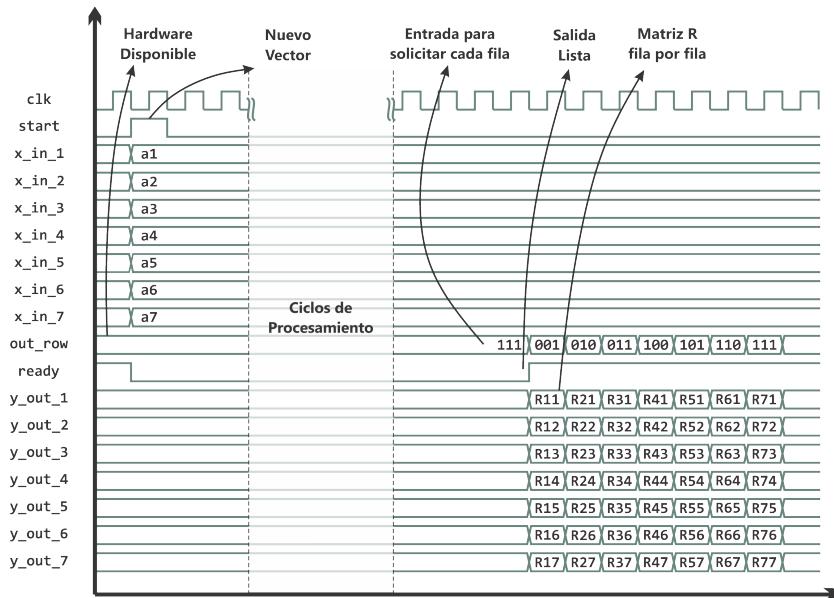
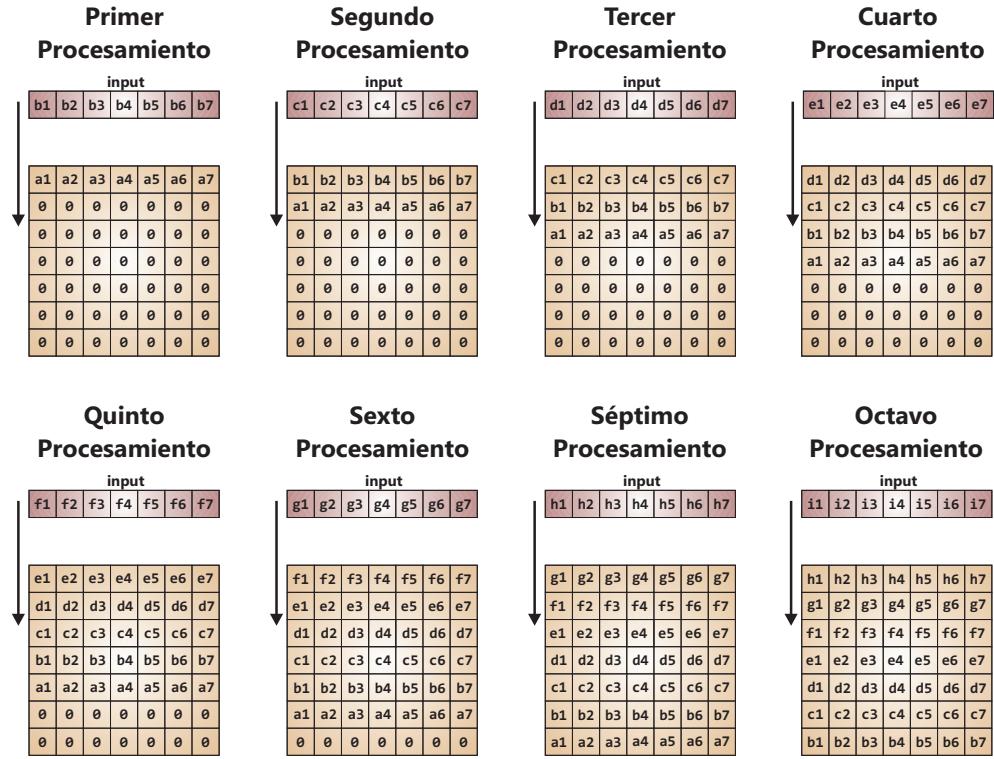


Figura 4.4: Diagrama de Tiempos en el uso del Hardware

La composición de la matriz es siempre generada por un desplazamiento hacia abajo de las filas, en el cual se pierde la última fila, e ingresa en la primera el nuevo vec-

tor de entrada suministrado. Esta implementación recursiva se adapta al proceso de *beamforming*, donde las filas que se mantienen representan las últimas 6 muestras. Este comportamiento se observa en la figura 4.5. En la imagen se debe notar, por un lado el cambio en cada procesamiento, donde se observa el resultado anterior desplazado, y por el otro, la pérdida de la última fila en el Octavo Procesamiento.



4.9.3. Unidad de registros

A continuación se hará una descripción detallada de la forma en la cual se almacenan y utilizan los elementos de una matriz en hardware. En la sección 3.4.4, se explicó que es necesario retener los valores de R de una iteración, para la siguiente. En el mapeo de Walke, al reducir la arquitectura, se requiere que no sólo los valores de R se almacenen, sino también los de x , y al ser varios pares (x, R) asignados a un mismo procesador, esto es requerido para varios ciclos.

Esta implementación podría haber sido realizada dentro de las celdas, pero en lugar de hacerlo de esta forma, se optó por crear una unidad de registros, que tenga ciertas propiedades particulares para dar ventajas a la ejecución del algoritmo. Tanto la unidad de registros, como la forma en la cual se opera, es un diseño propio que presenta diversas ventajas para el mapeo, y fue ideado específicamente durante el desarrollo.

Como se vio en el diagrama en bloques, el esquema de almacenamiento de datos es el siguiente:

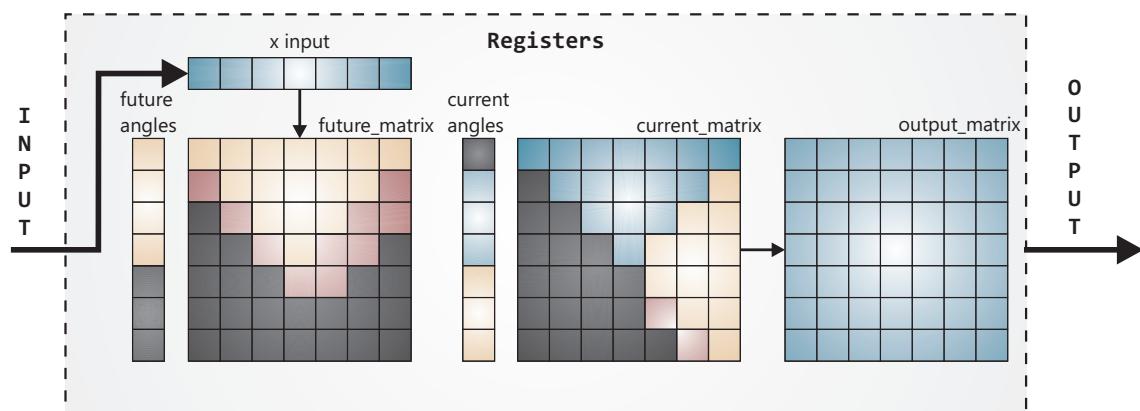


Figura 4.6: Esquema de registros utilizados

Una de las características del mapeo de Walke, es que para lograr la máxima eficiencia de los procesadores, mientras algunos de ellos están procesando el ciclo actual, otros están procesando datos para el ciclo futuro. Por lo cual, en un determinado ciclo de procesamiento se deben almacenar los elementos calculados para dos entradas diferentes. Por este motivo existen las unidades de registros **current_matrix** y **future_matrix**, las cuales son de tamaño $M \times N \times WORD_WIDTH$. Respectivamente, los ángulos calculados en las operaciones de *boundary cell* son uno por fila, y son almacenados en los registros **current_angles** y **future_angles**.

Finalmente, el bloque de registros **output_matrix** es utilizado para retener el resultado del cálculo de descomposición de una matriz durante toda la etapa de procesamiento, desde que se coloca en alto la señal de *ready* al final de un ciclo, hasta que ocurre el

mismo suceso en el siguiente.

Es importante destacar que la lógica de control y los pasos requeridos para la implementación del algoritmo serán descriptos con mayor detalle, y en este caso sólo se explican brevemente las características de los registros.

Se puede observar en el diagrama de la figura 4.6 que los elementos de la matriz poseen diferentes colores. Cada uno de ellos representa una característica del registro. Las mismas se describen a continuación:

- **Color Azul:** El color azul representa un registro que se actualiza una única vez durante todo el procesamiento de una matriz. El primer caso es el del registro `x_input`, donde se almacena el nuevo vector de entrada, el cual, concatenado con la matriz anterior, crea la nueva matriz. Luego se tiene un bloque de forma triangular en la parte superior de `current_matrix`. En el mismo, se realiza una copia de los valores de `future_matrix` para luego tenerlos disponibles en los estados de procesamiento. Esto es realizado en una de las etapas tempranas del algoritmo. Este es el mismo caso que se da con la primera mitad de `current_angles`. Finalmente, `output_matrix` permanece constante durante todo el procesamiento con el último cálculo de matriz realizado, y se actualiza en el último estado del algoritmo.
- **Color Dorado:** Representa un bloque de registros que forman parte del procesamiento del algoritmo, y cambia continuamente durante el mismo. En ellos, se almacenan cada una de las entradas y salidas modificadas por las *boundary_cells* e *internal_cells*. El primer caso se da en la matriz `future_matrix`, donde se empieza a calcular una matriz que será finalizada en el próximo ciclo. Por ejemplo, en el ciclo 1 de procesamiento, se utilizan los registros de las posiciones (1; 1) y (2; 1) de esta unidad. Por otro lado, se tiene la matriz `current_matrix`, donde se almacena el cálculo de la matriz actual, y se utilizan los valores previamente calculados en el sitio anterior. Para seguir con el ejemplo del ciclo 1, se utilizan los registros de las posiciones (4; 5), (5; 5), (2; 7), (3; 7), (3; 6), (4; 6). Finalmente, se tienen los registros donde se leen y escriben los ángulos calculados.
- **Color Rojo:** Además de ser registros con las mismas características que poseen los registros dorados, este color representa un registro utilizado para mantener un valor que es requerido en el próximo procesamiento, que sería perdido al realizar el desplazamiento hacia abajo.
- **Color Negro:** Representa un registro que no es utilizado. Estos registros se encuentran presentes dado que en Verilog los mismos son creados definiendo los bloques en forma de arreglos bidimensionales. Se verá que, como posibles mejoras, se puede lograr unificar la matriz de salida con estos registros en conjunto con otros adicionales.

4.9.4. Unidad de control

A continuación se describirá la unidad codificada para el control del procesador de descomposición QR. La misma fue codificada como una máquina de estados finita. Dado que los estados son consecutivos, la síntesis resulta en un contador, pero la descripción como máquina de estados permite, además de una lectura y entendimiento del código más amigable, el escalamiento del mismo en el futuro. A continuación se describen cada uno de los estados:

- **wait_vector:** Se trata del estado *idle* del procesador en el que se encuentra a la espera de un nuevo vector, el cual es detectado a través de la señal **start** en alto. El vector que es suministrado desde las entradas del procesador es almacenado en el bloque de registros **future_matrix** en la fila 0. Esto representa una diferencia con el diagrama de registros visto, en el cual se observa un registro separado denominado **x_input**. Se debe tener presente que la fila 0 **no** es una fila válida de matriz, dado que tanto filas como columnas comienzan desde el subíndice 1. El registro 0, presente en todos los casos, es agregado como un auxiliar que en este caso es utilizado para almacenar una nueva entrada.

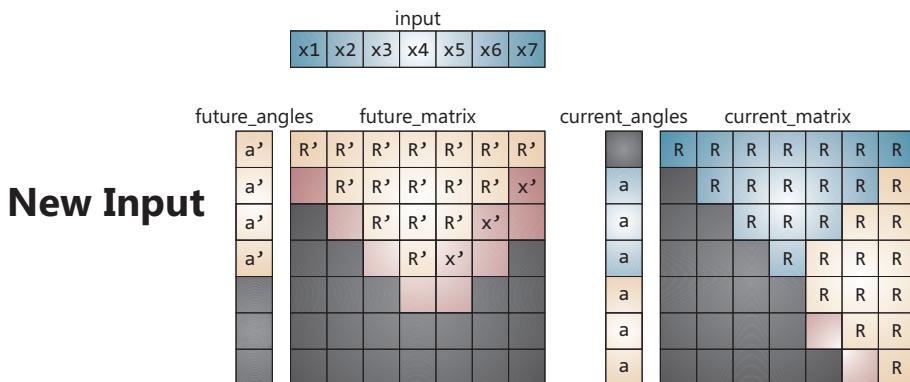


Figura 4.7: Detección de nueva entrada y estado inicial

- **shift_down:** En este estado, con excepción del bloque de registros de salida, se desplazan todos los registros hacia abajo. En **future_matrix**, dado que en la posición 0 se tiene el nuevo vector x , la nueva fila ingresa a la matriz en la fila 1, se desplazan hacia abajo las demás, y se pierde la última. Los otros registros, con excepción de **output_matrix**, donde no es necesario, también son desplazados hacia abajo para alinearse con el cambio originado por la nueva entrada.

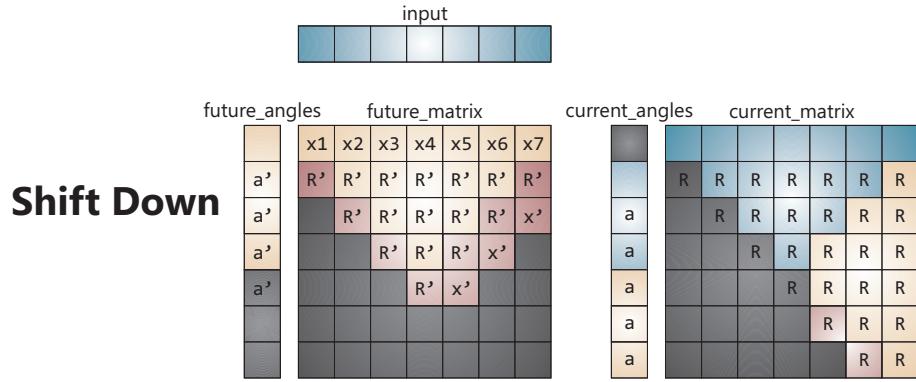


Figura 4.8: Resultado del desplazamiento hacia abajo

- **copy_matrix:** Una vez realizado el desplazamiento, se debe realizar una copia de los valores actuales en **future_matrix** a la matriz **current_matrix**. Dichos valores son aquellos que fueron calculados en el procesamiento anterior y conforman la primera mitad del cálculo actual. Se copian los valores de R y de x . Adicionalmente, en este estado se comprueba si el vector recibido es el primer vector. En dicho caso, no es necesario hacer ningún procesamiento dado que para que el mismo sea válido es requerido que ingresen en los registros por lo menos dos vectores. Si la comprobación es acertada, se vuelve al estado **wait_vector**.

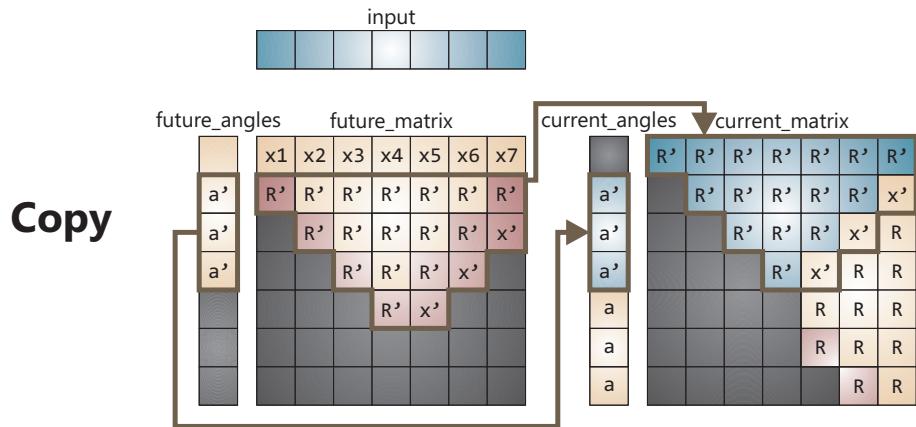


Figura 4.9: Resultado de la copia

- **load_cycleN** y **process_cycleN**: Posterior a la copia, se tiene una serie de estados denominados **load_cycle** y **process_cycle** junto con un número, los cuales consisten en los N estados de procesamiento de los valores R y x de la matriz. En los mismos, se redireccionan las entradas y salidas de las celdas BC e IC, y se controlan las señales de **load** y **ready** para que cada una de las celdas lleve a cabo

la rotación correspondiente según las rotaciones de Givens.

Básicamente, el estado *load* consiste en colocar en estado alto la señal `load` suministrando las entradas correspondientes al estado en proceso. Al pasar al siguiente estado, denominado *process*, se pone en estado bajo la señal de *load* y se espera a que todas las celdas coloquen en alto la señal de *ready*. Una vez detectado dicho suceso, las salidas de las celdas son almacenadas en los registros y el proceso se repite en la siguiente iteración.

- **output_results:** En este estado, se realiza una copia de la imagen actual del bloque de registros `current_matrix` al bloque `output_matrix`. Dado que la unidad `current_matrix` es constantemente modificada durante los ciclos de procesamientos, es requerido copiarlos si se desea retenerlos. Existe un hardware adicional, activado por la señal `send_output`, que lee los valores de la `output_matrix` y los escribe en las señales de salida, uno por ciclo de *clock* del procesador QR.

En el siguiente capítulo se describirán todos los resultados obtenidos en las simulaciones y síntesis del hardware implementado.

4.10. Herramientas utilizadas

A continuación se hace una breve reseña de las herramientas utilizadas para la implementación del procesador:

Sublime Text 2

Sublime Text fue el principal editor de texto utilizado para codificar en lenguaje Verilog. El mismo presenta una interfaz gráfica con diferentes características de gran utilidad.



Sublime Text

Vim

Vim es un editor de texto nativo en entornos de tipo UNIX y fue utilizado en algunos casos en conjunto con Sublime Text 2.



Vim

ModelSim

ModelSim fue la herramienta utilizada para la simulación del hardware descripto en código fuente Verilog, con la cual se generaron los archivos de salida de tipo *wave*.

ModelSim

gtkwave

Gtkwave fue la herramienta utilizada para visualizar los archivos de tipo *wave* que fueron generados por la simulación realizada en ModelSim.



Xilinx ISE

ISE Design Suite es la herramienta proporcionada por la compañía Xilinx para crear los archivos de síntesis “.bit” a partir de un hardware codificado en un lenguaje HDL. Dichos archivos .bit implementan el hardware en los dispositivos FPGA que Xilinx provee (Spartan y Virtex, entre otros). También fue utilizada para transferir los archivos “.bit” al kit de desarrollo FPGA.



Bitbucket

Para realizar el desarrollo se utilizó un repositorio hospedado en <http://www.bitbucket.org>, el cual fue utilizado a través de la herramienta de versionado Mercurial. Dicho sistema aportó diversas ventajas, entre las cuales principalmente se encuentran el mantener un registro de todas las modificaciones realizadas entre cada una de las versiones y el poder trabajar en la nube. En caso de algún comportamiento no deseado al hacer un cambio de versión, el mismo podía ser identificado en el registro proporcionado por la herramienta. El poder trabajar en la nube, facilitó el hecho de poder hacer aportes en el desarrollo desde distintos equipos, ya sea por estar en un equipo de la facultad, del hogar o del trabajo.



Capítulo 5

Estudio, simulación y validación del IP core generado

En el capítulo anterior se explicó la forma en la cual fue implementado el procesador de descomposición QR en hardware. Se hizo hincapié en las principales características de cada uno de los códigos fuente que definen los módulos que lo conforman.

En el presente capítulo, se expondrán las diferentes herramientas diseñadas para poner a prueba el hardware, tanto aquellas que fueron utilizadas durante su desarrollo como aquellas utilizadas una vez finalizado el mismo para la obtención de métricas. Se implementaron diferentes tipos de bancos de pruebas, cada uno de ellos en base a una necesidad diferente.

Por otro lado, se presentarán todos los resultados de cada una de las pruebas, se plantearán diferentes métricas y se harán comparaciones con otras arquitecturas de procesadores mencionadas en capítulos anteriores (ver sección 3.6).

5.1. Emulador de hardware en lenguaje C

Previamente al desarrollo del hardware en lenguaje Verilog, se optó por poner a prueba el mapeo de Walke del algoritmo utilizando el lenguaje C. Dicho programa iba a aportar diversas ventajas para el proyecto:

- Comprender la mecánica del algoritmo y del mapeo de Walke [11], teniendo la posibilidad de extraer resultados parciales paso por paso.
- Proveer una fuente de referencia para la comparación de resultados.
- Posibilitar la expansión del programa para crear un *testbench* que evalúe el hardware sintetizado en un dispositivo FPGA.

- Dado que la sintaxis de Verilog está basada en el lenguaje C, la lógica del mapeo podría ser extraída de este código para ser directamente insertada con mínimas correcciones en el código Verilog y evitar potenciales errores en dicho bloque de desarrollo.

El costo de desarrollar dicha herramienta sería mínimo en comparación con el desarrollo del hardware, por lo cual, en base a las ventajas que aportaría, se decidió realizarlo. El mismo se encuentra descripto en el código fuente `qr_decomposition.c`.

La estructura del código fue concebida, no para estar optimizada en cuanto a su funcionalidad, sino para que la sintaxis fuera similar a Verilog, y para que las funciones y datos representaran módulos y registros. Como principales diferencias con respecto al hardware, se tiene que las rotaciones fueron implementadas haciendo uso de las funciones trigonométricas de la librería `math.h`, en lugar de utilizar el algoritmo CORDIC. Asimismo, los datos utilizados fueron de tipo double, con el objeto de trabajar con la máxima precisión disponible.

```
qr_decomposition.cpp ×
1 #include "qr_decomposition.h"
2
3 extern char enable_debug;
4
5 /* qr_decomposition: Function to calculate in software, simulating the hardware
6 behaviour, the qr_decomposition of a 7 x 7 matrix. It uses static variables
7 defined on this code. */
8 void
9 qr_decomposition(int16_t input_vector[COLUMNS+1], double output_matrix[ROWS+1][COLUMNS+1]) {
10    /* Variables. */
11    int state = 0;
12    int input_counter = 0;
13    int j;
14
15    /* Registers. */
16    static int first_vector = 0;
17    static double future_matrix[ROWS+2][COLUMNS+1];
18    static double current_matrix[ROWS+2][COLUMNS+1];
19    static double future_angles[ROWS+2];
20    static double current_angles[ROWS+2];
21
22    /* Processors inputs and ouputs definition. */
23    double bc_x, bc_R, *bc_R_new, *bc_angle;
24    double ic1_x, ic1_R, ic1_angle, *ic1_x_new, *ic1_R_new;
25    double ic2_x, ic2_R, ic2_angle, *ic2_x_new, *ic2_R_new;
26    double ic3_x, ic3_R, ic3_angle, *ic3_x_new, *ic3_R_new;
27
28    /* Reset. */
29    if(!first_vector){
30        reset_matrix(output_matrix , ROWS+1, COLUMNS+1);
31        reset_matrix(future_matrix , ROWS+1, COLUMNS+1);
32        reset_matrix(current_matrix, ROWS+1, COLUMNS+1);
33        reset_vector(future_angles , COLUMNS+1);
34        reset_vector(current_angles, COLUMNS+1);
35    }
}
```

Figura 5.1: Extracto de la función de emulación *QR Decomposition*

Con el objeto de validar los resultados de la implementación, se efectuaron pruebas haciendo uso de las herramientas *online* Wolphram Alpha y Bluebit Matrix Calculator. Dichas herramientas poseen motores de cálculo de diferentes funciones matemáticas, entre ellas, la descomposición QR. Se definió una matriz de referencia, la cual fue utilizada en las pruebas realizadas en cada uno de los *testbenches*:

$$\begin{bmatrix} 11012 & 2210 & 2130 & 1140 & 5320 & 6240 & 9870 \\ 8771 & 7722 & 5663 & 4524 & 2695 & 1276 & 1727 \\ 5751 & 18000 & 15578 & 12290 & 15311 & 12260 & 7121 \\ 11110 & 11111 & 13232 & 3245 & 2282 & 13644 & 13373 \\ 13261 & 2223 & 12322 & 9222 & 11115 & 11226 & 12217 \\ 15510 & 11119 & 16553 & 6544 & 6560 & 18861 & 12217 \\ 2571 & 7222 & 11360 & 12650 & 16225 & 12226 & 7912 \end{bmatrix}$$

Figura 5.2: Matriz de referencia elegida para las pruebas



Input matrix:

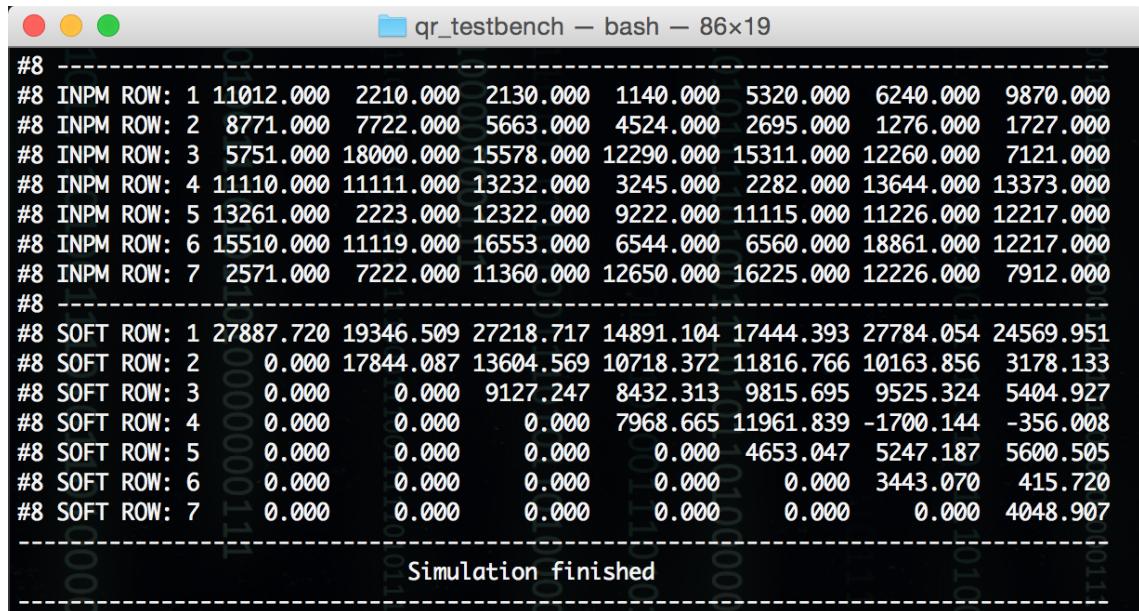
11012.000	2210.000	2130.000	1140.000	5320.000	6240.000	9870.000
8771.000	7722.000	5663.000	4524.000	2695.000	1276.000	1727.000
5751.000	18000.000	15578.000	12290.000	15311.000	12260.000	7121.000
11110.000	11111.000	13232.000	3245.000	2282.000	13644.000	13373.000
13261.000	2223.000	12322.000	9222.000	11115.000	11226.000	12217.000
15510.000	11119.000	16553.000	6544.000	6560.000	18861.000	12217.000
2571.000	7222.000	11360.000	12650.000	16225.000	12226.000	7912.000

QR Decomposition:

R:

27887.720	19346.509	27218.717	14891.104	17444.393	27784.054	24569.951
0.000	17844.087	13604.569	10718.372	11816.766	10163.856	3178.133
0.000	0.000	9127.247	8432.313	9815.695	9525.324	5404.927
0.000	0.000	0.000	7968.665	11961.839	-1700.144	-356.008
0.000	0.000	0.000	0.000	4653.047	5247.187	5600.505
0.000	0.000	0.000	0.000	0.000	3443.070	415.720
0.000	0.000	0.000	0.000	0.000	0.000	4048.907

Figura 5.3: Resultado provisto por Bluebit



```
#8
#8 INPM ROW: 1 11012.000 2210.000 2130.000 1140.000 5320.000 6240.000 9870.000
#8 INPM ROW: 2 8771.000 7722.000 5663.000 4524.000 2695.000 1276.000 1727.000
#8 INPM ROW: 3 5751.000 18000.000 15578.000 12290.000 15311.000 12260.000 7121.000
#8 INPM ROW: 4 11110.000 11111.000 13232.000 3245.000 2282.000 13644.000 13373.000
#8 INPM ROW: 5 13261.000 2223.000 12322.000 9222.000 11115.000 11226.000 12217.000
#8 INPM ROW: 6 15510.000 11119.000 16553.000 6544.000 6560.000 18861.000 12217.000
#8 INPM ROW: 7 2571.000 7222.000 11360.000 12650.000 16225.000 12226.000 7912.000
#8
#8 SOFT ROW: 1 27887.720 19346.509 27218.717 14891.104 17444.393 27784.054 24569.951
#8 SOFT ROW: 2 0.000 17844.087 13604.569 10718.372 11816.766 10163.856 3178.133
#8 SOFT ROW: 3 0.000 0.000 9127.247 8432.313 9815.695 9525.324 5404.927
#8 SOFT ROW: 4 0.000 0.000 0.000 7968.665 11961.839 -1700.144 -356.008
#8 SOFT ROW: 5 0.000 0.000 0.000 0.000 4653.047 5247.187 5600.505
#8 SOFT ROW: 6 0.000 0.000 0.000 0.000 0.000 3443.070 415.720
#8 SOFT ROW: 7 0.000 0.000 0.000 0.000 0.000 0.000 4048.907
-----
Simulation finished
```

Figura 5.4: Resultado provisto por el emulador QR

En las imágenes se puede observar que los resultados del emulador son idénticos a la fuente de referencia utilizada. Al obtener los mismos, se lograron todos los objetivos planteados para este desarrollo.

5.2. Simulación de módulos en Modelsim

Como se explicó en capítulos anteriores, al desarrollar un módulo en Verilog se validaba su funcionamiento a través de una simulación utilizando ModelSim. Una vez realizado este proceso con todos los módulos individuales, se procedió a su integración. Luego de una etapa de corrección de diversos errores propios del proceso de desarrollo de hardware, se logró que el mismo entregara el resultado correcto de la descomposición de la matriz de referencia:

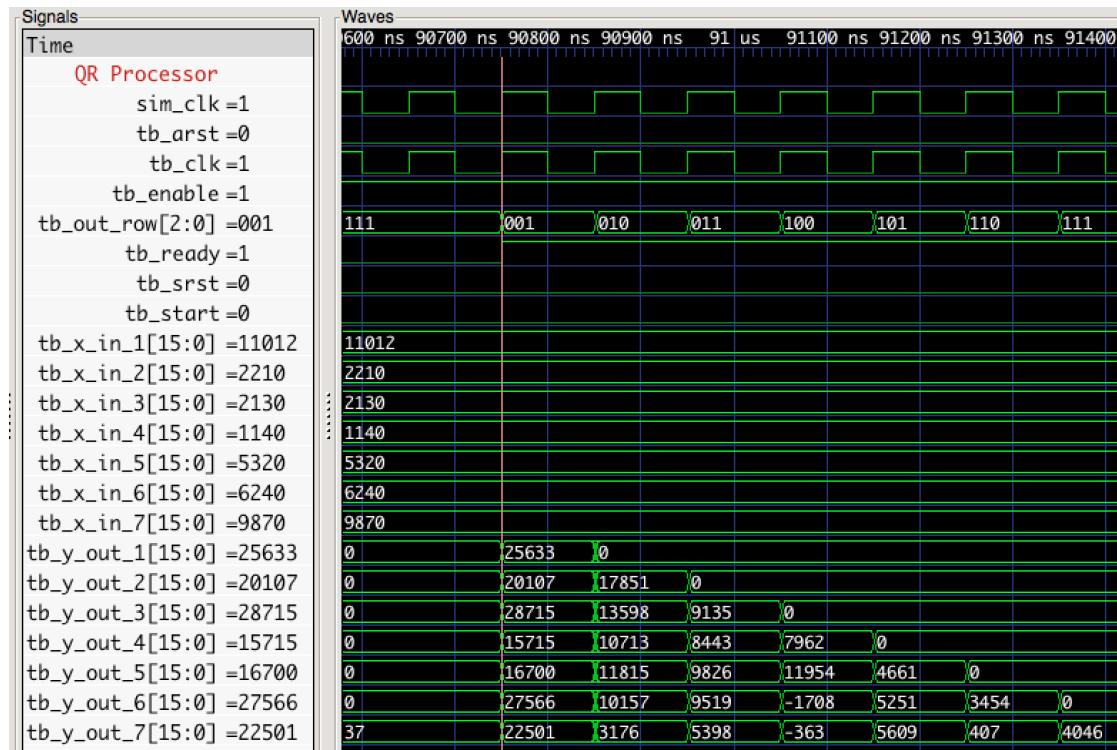


Figura 5.5: Resultados de la simulación del procesador

En la figura 5.5 se puede observar el resultado de las señales de salida del hardware desarrollado, las cuales pueden ser contrastadas contra la función de emulación de la figura 5.4. Con cierto margen de error, los resultados del emulador desarrollado en lenguaje C y el hardware coinciden. Debido a que la simulación fue realizada utilizando una precisión de 16 bits en punto fijo, la existencia de un margen de error es esperable con respecto a un resultado de 64 bits en punto flotante. En la sección 5.8.3 se hace un análisis de la precisión del procesador.

Una vez validado el funcionamiento del hardware para algunas matrices, se procedió a armar un hardware *top level* para utilizar y comunicar el procesador con otro dispositivo, como por ejemplo una PC. Para ello, se incluyó el procesador de descomposición QR, una unidad UART para realizar la comunicación, y una unidad de control que contenga la lógica necesaria para que el sistema pueda recibir valores, realizar un cálculo y enviar los resultados. Dicho hardware se encuentra descripto en el código `qr_eval.v`. A continuación se presenta un diagrama en bloques del mismo:

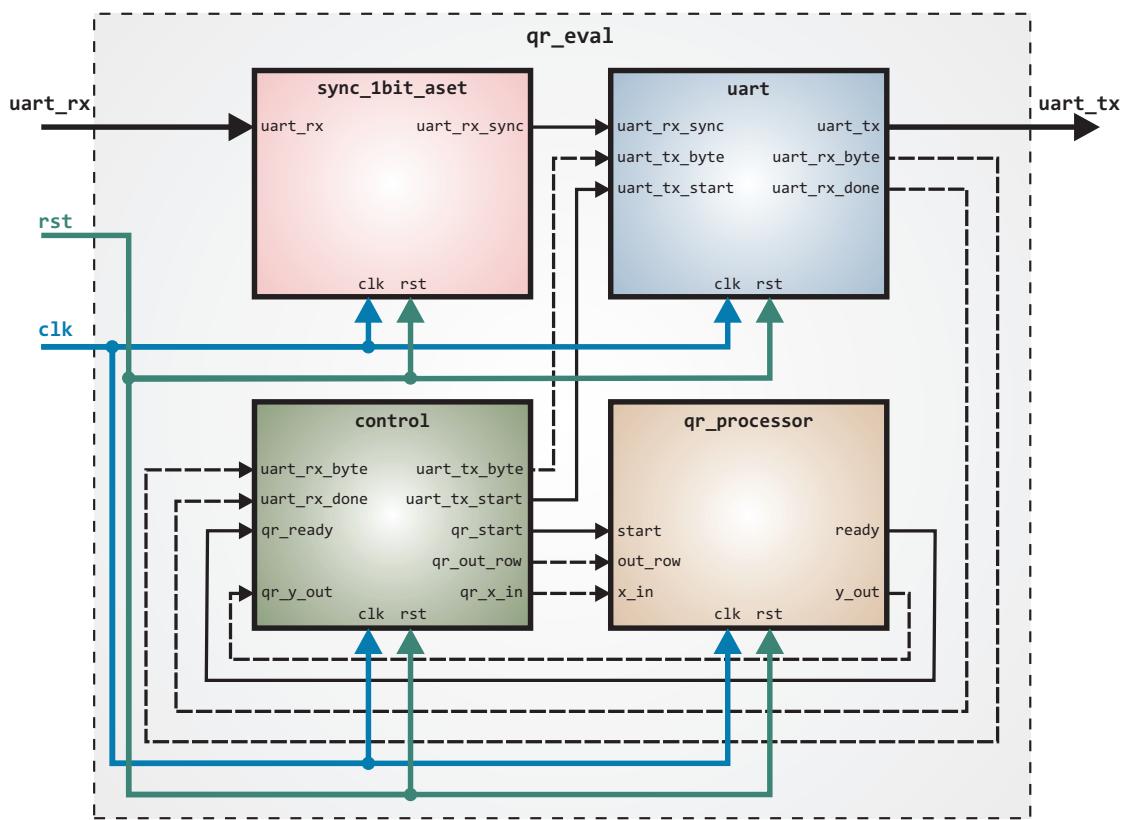


Figura 5.6: Diagrama en bloques de la interfaz de integración `qr_eval`

Los estados de la unidad de control siguen el siguiente esquema:

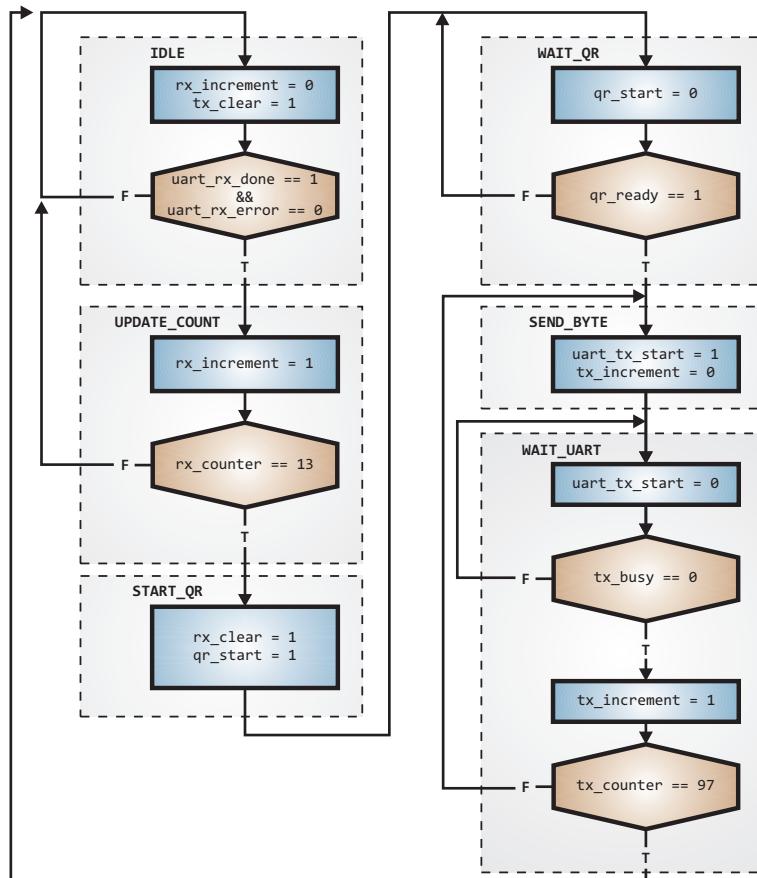


Figura 5.7: Diagrama en estados simplificado de la unidad de control utilizada en la interfaz

Una de las características del *top level* es su simplicidad, dado que sólo requiere el conexionado de las líneas de CLK, RST (para la cual se utiliza un *push button*), UART RX y UART TX para operar.

En la siguiente figura, se observa la simulación del módulo desarrollado, destacando los instantes más importantes de su funcionamiento:

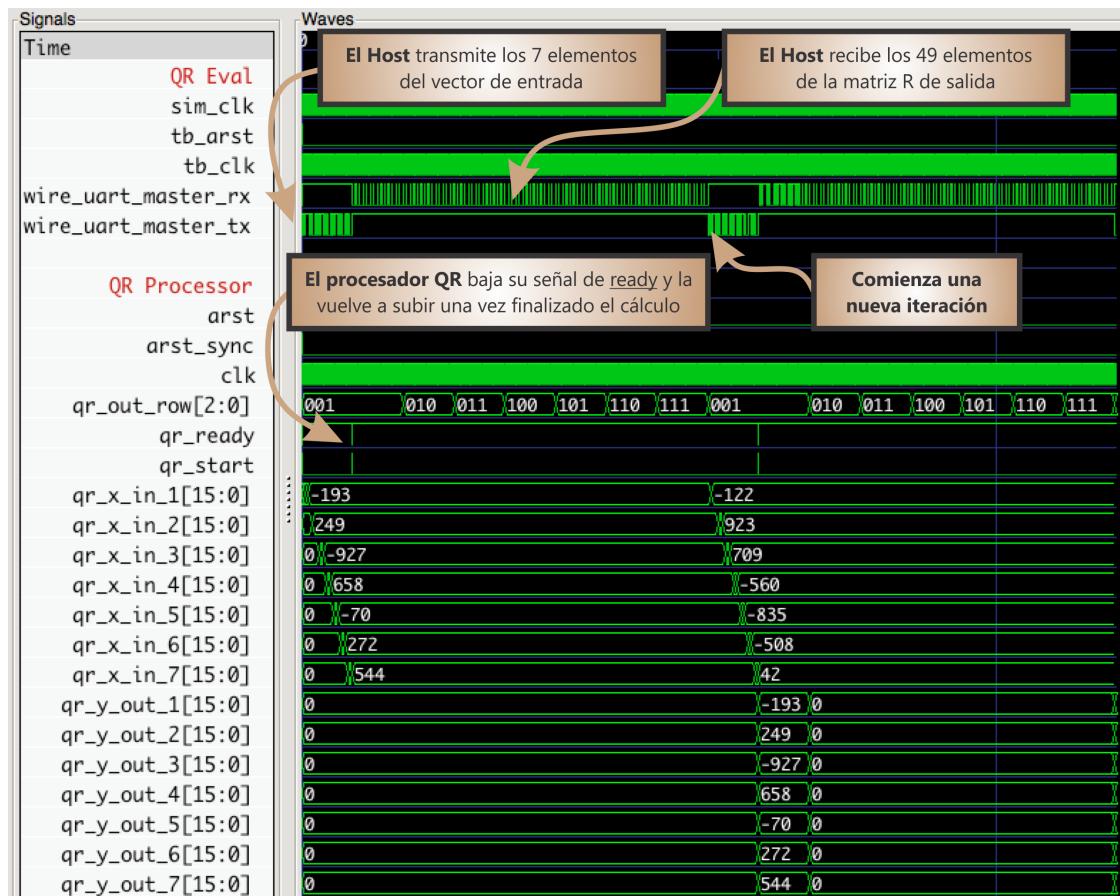


Figura 5.8: Resultados de la simulación de la interfaz del procesador

5.4. Testbench desarrollado en Verilog

Teniendo disponible la codificación del hardware completa, se procedió a desarrollar un *script* en lenguaje Verilog para lograr llevar a cabo una prueba intensiva del hardware. Dicho *script* representaría la función que cumple una PC al evaluar el hardware sintetizado, en un entorno de simulación. El mismo debería ser capaz de:

- Tomar 7 líneas de un archivo de entrada para la lectura de vectores de entrada x .
- Tomar 49 líneas de un archivo de entrada para la lectura del valor esperado de la matriz R .

- Enviar el vector de entrada utilizado a través de una unidad UART.
- Esperar a que el procesador QR realice el cálculo y luego recibir la matriz R a través de la unidad UART.
- Presentar los resultados en pantalla en un formato amigable de lectura y escribir los mismos en un archivo de salida simple para su análisis.

Esta lógica se encuentra descripta en el código `qr_host_transactor.v`, el cual es instanciado en la simulación. El *testbench* utiliza 3 archivos diferentes:

- **input_vector.dat**: Contiene los elementos del vector de entrada separados línea por línea. Luego de 7 líneas, la siguiente representa el primer elemento del siguiente vector.
- **software_result.dat**: Contiene los elementos de la matriz R calculada en software separados línea por línea. Luego de 7 líneas, la siguiente representa el primer elemento de la siguiente fila. Luego de 49 líneas, la siguiente representa una nueva matriz R .
- **hardware_result.dat**: Se guardan los valores calculados por el hardware utilizando el mismo formato del archivo de software.

Los archivos de entrada fueron generados utilizando un programa escrito en lenguaje C denominado `qr_testbench.c`, el cual se explica en la sección 5.6. A continuación se expone un ejemplo de un archivo de *input vectors* en el cual se observan los elementos de las primeras 3 filas ingresadas:

```
1 2571
2 7222
3 11360
4 12650
5 16225
6 12226
7 7912
8 15510
9 11119
10 16553
11 6544
12 6560
13 18861
14 12217
15 13261
16 2223
17 12322
18 9222
19 11115
20 11226
21 12217
```

Durante la ejecución del *script* en Verilog, se observa la siguiente salida en pantalla:

```
#-----#
#-----# QR Processor Testbench
#-----#
# 1 -----
# 1 INPUTVEC: 2571 7222 11360 12650 16225 12226 7912
# 1
# 1 INPM ROW 1: 0 0 0 0 0 0 0
# 1 INPM ROW 2: 0 0 0 0 0 0 0
# 1 INPM ROW 3: 0 0 0 0 0 0 0
# 1 INPM ROW 4: 0 0 0 0 0 0 0
# 1 INPM ROW 5: 0 0 0 0 0 0 0
# 1 INPM ROW 6: 0 0 0 0 0 0 0
# 1 INPM ROW 7: 0 0 0 0 0 0 0
# 1
# 1 SOFT ROW 1: 0 0 0 0 0 0 0
# 1 SOFT ROW 2: 0 0 0 0 0 0 0
# 1 SOFT ROW 3: 0 0 0 0 0 0 0
# 1 SOFT ROW 4: 0 0 0 0 0 0 0
# 1 SOFT ROW 5: 0 0 0 0 0 0 0
# 1 SOFT ROW 6: 0 0 0 0 0 0 0
# 1 SOFT ROW 7: 0 0 0 0 0 0 0
# 1
# 1 HARD ROW 1: 0 0 0 0 0 0 0
# 1 HARD ROW 2: 0 0 0 0 0 0 0
# 1 HARD ROW 3: 0 0 0 0 0 0 0
# 1 HARD ROW 4: 0 0 0 0 0 0 0
# 1 HARD ROW 5: 0 0 0 0 0 0 0
# 1 HARD ROW 6: 0 0 0 0 0 0 0
# 1 HARD ROW 7: 0 0 0 0 0 0 0
# 2 -----
# 2 INPUTVEC: 15510 11119 16553 6544 6560 18861 12217
# 2
# 2 INPM ROW 1: 2571 7222 11360 12650 16225 12226 7912
# 2 INPM ROW 2: 0 0 0 0 0 0 0
# 2 INPM ROW 3: 0 0 0 0 0 0 0
# 2 INPM ROW 4: 0 0 0 0 0 0 0
# 2 INPM ROW 5: 0 0 0 0 0 0 0
# 2 INPM ROW 6: 0 0 0 0 0 0 0
# 2 INPM ROW 7: 0 0 0 0 0 0 0
# 2
```

Figura 5.9: Ejecución de una simulación del cálculo de una matriz en el procesador

Se diseñó la interfaz de salida con dos propósitos. En primer lugar se deseaba que rápidamente se pudiera observar de forma organizada cada uno de los elementos de la matriz, siendo posible identificar a qué salida pertenecían. Por el otro lado, se deseaba que la salida pudiera ser procesada a través de expresiones regulares simples con el objetivo de buscar algún resultado en particular. En la misma, es posible identificar los siguientes elementos:

- **N:** El número al inicio de cada línea representa el número de iteración en el cual se obtuvieron los resultados en dicha línea.
- **INPUTVEC:** Representa el *input vector* que se está ingresando en la iteración actual.
- **INPM ROW:** Representa las filas de la matriz formada con los últimos 7 *input vectors* ingresados.
- **SOFT ROW:** Representa las filas de la matriz R que se obtienen como resultado del emulador en software. Los elementos fueron pre-cargados a través del archivo `software_result.dat`.
- **HARD ROW:** Representa las filas de la matriz R que es calculada por el procesador QR simulado. Los elementos son almacenados en el archivo `hardware_result.dat`.

Una vez finalizada la simulación, se observa la siguiente salida en pantalla:

```

# 8 -----
# 8 INPUTVEC: 11012 2210 2130 1140 5320 6240 9870
# 8
# 8 INPM ROW 1: 11012 2210 2130 1140 5320 6240 9870
# 8 INPM ROW 2: 8771 7722 5663 4524 2695 1276 1727
# 8 INPM ROW 3: 5751 18000 15578 12290 15311 12260 7121
# 8 INPM ROW 4: 11110 11111 13232 3245 2282 13644 13373
# 8 INPM ROW 5: 13261 2223 12322 9222 11115 11226 12217
# 8 INPM ROW 6: 15510 11119 16553 6544 6560 18861 12217
# 8 INPM ROW 7: 2571 7222 11360 12650 16225 12226 7912
# 8 -----
# 8 SOFT ROW 1: 27887 19346 27218 14891 17444 27784 24569
# 8 SOFT ROW 2: 0 17844 13604 10718 11816 10163 3178
# 8 SOFT ROW 3: 0 0 9127 8432 9815 9525 5404
# 8 SOFT ROW 4: 0 0 0 7968 11961 -1700 -356
# 8 SOFT ROW 5: 0 0 0 0 4653 5247 5600
# 8 SOFT ROW 6: 0 0 0 0 0 3443 415
# 8 SOFT ROW 7: 0 0 0 0 0 0 4048
# 8 -----
# 8 HARD ROW 1: 27901 19347 27226 14889 17445 27793 24571
# 8 HARD ROW 2: 0 17851 13598 10713 11815 10157 3176
# 8 HARD ROW 3: 0 0 9135 8443 9826 9519 5398
# 8 HARD ROW 4: 0 0 0 7962 11954 -1708 -363
# 8 HARD ROW 5: 0 0 0 0 4661 5251 5609
# 8 HARD ROW 6: 0 0 0 0 0 3454 407
# 8 HARD ROW 7: 0 0 0 0 0 0 4046
#
#
# TIME      : 468603342
# WARNINGS : 0
# ERRORS   : 0
# INFOS    : 0
# NOTES    : 0
# MESSAGES : 0
# TESTSEED : 952104738
#
#
# ..... TEST PASSED .....

```

Figura 5.10: Fin de la simulación del cálculo de una matriz en el procesador

5.5. Síntesis de hardware

Como se mencionó anteriormente, para llevar a cabo la síntesis del hardware desarrollado, se utilizó la herramienta Xilinx ISE Design Suite 14.1. En principio, se configuró el proyecto para trabajar sobre un dispositivo FPGA Spartan 3E, del cual se disponía un kit de desarrollo para programar el archivo de síntesis. Se utilizó el *kit* de desarrollo Spartan3E Starter Kit. Es importante destacar que Xilinx cuenta con 4 líneas activas de dispositivos FPGA en función de sus prestaciones, y Spartan3E se trata de un dispositivo discontinuado, que actualmente fue reemplazado por Spartan6, y se encuentra en la línea más básica de ellas:

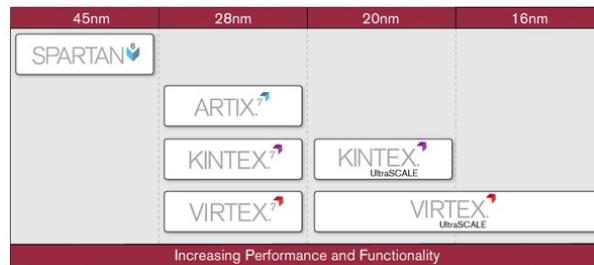


Figura 5.11: Relación de rendimiento y funcionalidades en dispositivos FPGA Xilinx

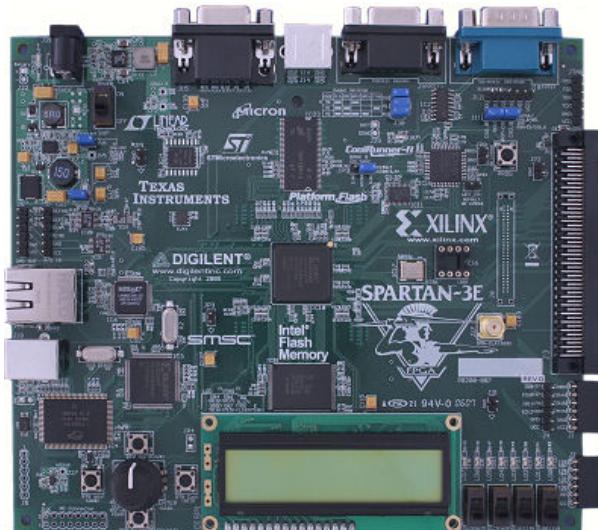


Figura 5.12: Kit de desarrollo de Spartan 3E

A continuación se presentan algunas de las características principales de los dispositivos FPGA Xilinx [16]:

	Spartan-6	Artix-7	Kintex-7	Virtex-7	Kintex UltraScale	Virtex UltraScale
Logic Cells	147,443	215,360	477,760	1,954,560	1,160,880	4,432,680
BlockRAM	4.8Mb	13Mb	34Mb	68Mb	76Mb	132.9Mb
DSP Slices	180	740	1,920	3,600	5,520	2,880
DSP Performance (symmetric FIR)	140GMACs	930GMACs	2,845GMACs	5,335GMACs	8,180 GMACs	4,268 GMACs
Transceiver Count	8	16	32	96	64	120
Transceiver Speed	3.2 Gb/s	6.6 Gb/s	12.5 Gb/s	28.05 Gb/s	16.3 Gb/s	32.75 Gb/s
Total Transceiver Bandwidth (full duplex)	50 Gb/s	211 Gb/s	800 Gb/s	2,784 Gb/s	2,086 Gb/s	5,886 Gb/s
Memory Interface (DDR3)	800	1,066	1,866	1,866	2,400	2,400
PCI Express® Interface	x1 Gen1	x4 Gen2	x8 Gen2	x8 Gen3	x8 Gen3	x8 Gen3
Analog Mixed Signal (AMS)/XADC	-	XADC	XADC	XADC	System Monitor	System Monitor
Configuration AES	Yes	Yes	Yes	Yes	Yes	Yes
I/O Pins	576	500	500	1,200	832	1,456
I/O Voltage	1.2V – 3.3V	1.2V – 3.3V	1.2V – 3.3V	1.2V – 3.3V	1.0 – 3.3V	1.0 – 3.3V

Tabla 5.1: Tabla comparativa de dispositivos FPGA Xilinx

Posteriormente, se realizó la síntesis en otros modelos de FPGA con el objetivo de tomar métricas, sin ser el archivo de síntesis de salida utilizado sobre dispositivos FPGA reales. A continuación se muestra la interfaz de la herramienta durante el proceso de síntesis y programación del dispositivo FPGA.

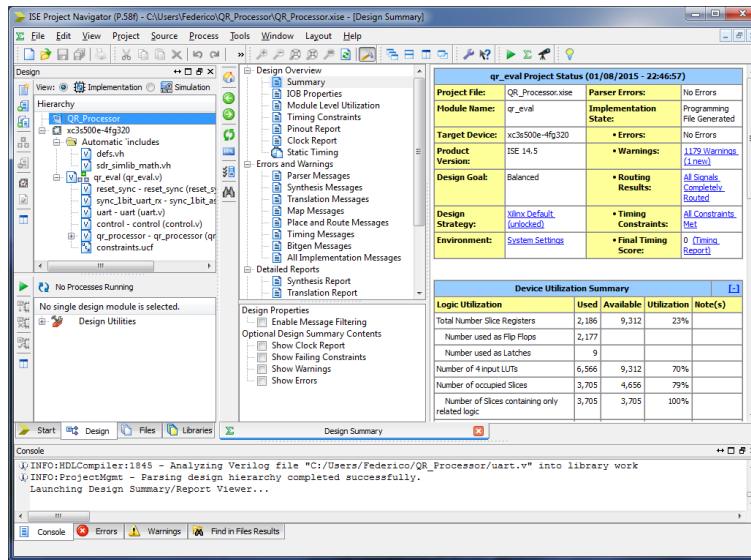


Figura 5.13: Resultado de la síntesis en la herramienta Xilinx ISE

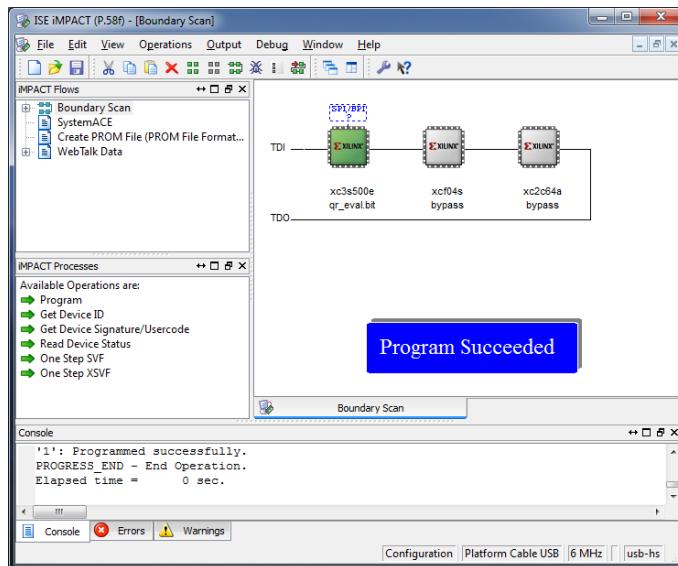


Figura 5.14: Proceso de programación del archivo de síntesis en el dispositivo FPGA

5.6. *Testbench* en lenguaje C

La última herramienta desarrollada fue un programa escrito en lenguaje C que pudiese funcionar como un banco de pruebas del hardware sintetizado. El mismo debía poseer las siguientes características:

- Generar un dato que representara vectores de entrada aleatorios, o vectores de entrada de ejemplo, definiendo parámetros tales como la norma máxima de cada elemento y número total de iteraciones.
- Utilizar la función `qr_decomposition()` desarrollada anteriormente, para tomar su resultado como una fuente de comparación con máxima precisión.
- Abrir una comunicación utilizando un puerto serial para enviar al *kit* de desarrollo el vector de entrada, y recibir del mismo el resultado de la matriz R .
- Crear archivos numéricos para almacenar los resultados y poder hacer una comparación posterior.
- Presentar en pantalla de forma amigable los resultados de las matrices computadas.

La herramienta se encuentra definida en el código fuente `qr_testbench.c`. La misma utiliza la línea de comandos para definir el comportamiento del *testbench* que se va a realizar. A continuación se muestra un extracto de la ayuda:

```

1 NAME
2     qr-testbench - A testbench software to assess a QR processor hardware.
3
4 SYNOPSIS
5     qr-testbench [options]
6
7 DESCRIPTION
8     qr-testbench is a program created to assist the development and testing
9     of an implementation of a hardware QR processor. It creates a random
10    vector
11    and makes use of a function that replicates the hardware's behaviour
12    using
13    the C language.
14
15 OPTIONS
16     -n, --norm
17        Defines the maximum value of the input vector elements. The default
18        value if ommited is 1000.
19
20     -d, --debug
21        Prints special output for debugging purposes.
22
23     -i, --iterations
24        Specifies the number of matrices that will be computed. The default
25        value if ommited is 10.
26
27     -s, --serial
28        Indicates that the serial port will be used to gather the information
29        of the QR hardware. The development kit with the synthesized hardware
30        must be connected for this option to work.
31
32     -w, --hwfile
33        Indicates that an input file will be used to gather the hardware
34        information. This option may be used to compare the information
35        with the results of a simulation.
36
37     -r, --random
38        By using this option the input vector generated will be random.
39
40     -m, --matrix
41        Specifies that the input vector will be generated from a predefined
42        input matrix file supplied by argument.
43
44     -f, --factor
45        Specifies the forgetting factor that will be used. The default
46        value if ommited is 1.
47
48     -h, --help
49        Shows a short help about how to use this program.
50
51 EXAMPLES
52     QR decomposition with lambda = 1 of a matrix formed with 10 vectors with
53     non-random elements with norm less than 1000, without using hardware
54     results :
55
56         qr-testbench
57
58     QR decomposition with lambda = 0.95 of a matrix formed with 200 vectors
59     with random elements with norm less than 100, without using hardware
      results :
```

```

60      qr_testbench -n 100 -i 200 -f 0.95 -r
61
62      QR decomposition with lambda = 0.90 using a samples input matrix, without
63      using hardware results:
64
65      qr_testbench -m sample_matrix.txt -f 0.90
66
67      QR decomposition with lambda = 0.90 using an input matrix, connecting the
68      testbench to the QR processor hardware using serial port /dev/tty.usbserial:
69
70      qr_testbench -m sample_matrix.txt -f 0.90 -s /dev/tty.usbserial
71
72      QR decomposition with lambda = 0.90 using an input matrix, taking the
73      hardware results from a hardware results file:
74
75      qr_testbench -m sample_matrix.txt -f 0.90 -w hardware_results.dat
76
77 AUTHOR
78     Written by Federico Damian Camarda.
79
80 REPORTING BUGS
81     Report bugs to <fededamian@gmail.com>

```

Durante el proceso de síntesis se detectaron errores que no habían sido evidenciados en la etapa de simulación. Luego de la corrección de los mismos se logró el comportamiento adecuado. A continuación se expone una imagen del banco de pruebas en funcionamiento:



Figura 5.15: Testbench: Macbook y el kit de desarrollo con el hardware sintetizado calculando

Finalmente, con todas las herramientas descriptas fue posible realizar las diferentes pruebas requeridas para someter el hardware a un continuo proceso de correcciones, evaluar

su comportamiento y desempeño, tomar métricas y compararlo con los trabajos analizados.

5.7. Correcciones sobre el hardware implementado

Al poner a prueba el hardware implementado, se verificó que los resultados obtenidos eran correctos para un número reducido de iteraciones. Sin embargo, al aumentar dicho número, se evidenciaba que los elementos de la matriz comenzaban a incrementarse, hasta finalmente llegar a una condición de *overflow*¹. Esto es claramente una condición no deseada, dado que desde el momento en que los cálculos alcanzan la condición de *overflow*, todos los resultados obtenidos comienzan a ser erróneos.

Se estudió este fenómeno desde un punto de vista teórico. Luego de dedicar una gran cantidad de tiempo a analizar el problema, dado que el mismo resultó difícil, se detectó que el conflicto en el hardware desarrollado radicaba en el hecho de que poseía un factor de olvido del algoritmo $\lambda = 1$. De esta forma, si el número de vectores se incrementa, esto corresponde a descomponer una matriz R con un orden n que aumenta constantemente. Por lo cual, la condición de *overflow* es inevitable. La misma no es propia del hardware, sino que es propia de la definición teórica del algoritmo al utilizar aritmética de precisión finita.

Con el objetivo de obtener una mejor visión de estos resultados, se desarrolló un *script* Octave, en el cual se ponen en práctica todos los conceptos asociados al algoritmo RLS, y al algoritmo QR-RLS. Dicho *script* permitiría observar gráficamente los imprevistos encontrados, y la forma con la cual se podría resolverlos. El objetivo consistió en definir un sistema a través de un vector de pesos W , y utilizar tanto RLS como QR-RLS para estimar dicho vector de pesos. El detalle del *script* puede encontrarse en el Apéndice B.

5.7.1. Algoritmo QR-RLS

El primer resultado que se deseaba evidenciar era el constante aumento de los elementos de la matriz R cuando se utilizaba un factor de olvido $\lambda = 1$. Se ejecutó la simulación, y a continuación se expone una gráfica del máximo elemento de la matriz R en función del número de iteración:

¹ *Overflow*: Condición en la cual el resultado de una operación de punto fijo produce más bits que sus operandos, en la cual los mismos exceden la longitud de palabra utilizada, lo que resulta en pérdida de información.

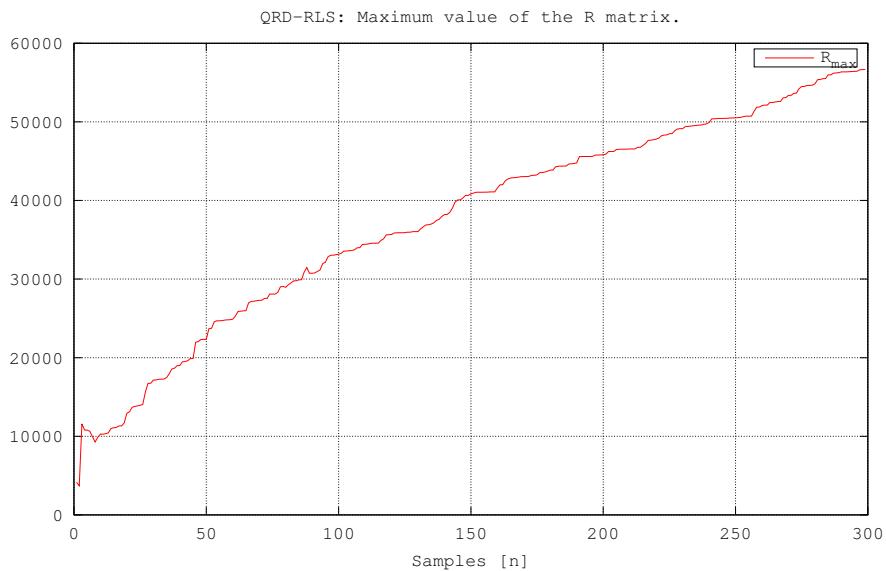


Figura 5.16: Máximo valor de la matriz R en función del número de iteración, $\lambda = 1$

En la figura 5.16 se puede ver claramente el carácter creciente de los elementos de la matriz R . Al analizar la bibliografía, se verificó que este conflicto puede solucionarse al utilizar un factor de olvido menor a 1. A continuación se exponen los resultados de la misma simulación, utilizando un factor de olvido $\lambda = 0,95$, un valor utilizado en varias de las diferentes referencias consultadas.

En contraste con la figura 5.16, se puede observar en la figura 5.17 que el máximo valor de la matriz R se mantiene por debajo del valor 30000. Analizando este resultado, se evidenció que era necesario implementar en el hardware desarrollado el factor de olvido para valores distintos de 1.

5.7.2. Factor de olvido en el procesador de descomposición QR

Una vez comprobada la necesidad de implementar un factor de olvido en el hardware desarrollado, se puso en práctica el diseño del mismo. Analizando las ecuaciones presentadas en la sección 3.4.1, se puede observar que el cambio necesario consiste en multiplicar a cada valor $R_{i,j}(n - 1)$ por el factor de olvido (referenciado como β en la ecuación), y luego realizar el procesamiento en las *boundary cells* e *internal cells* de la misma forma que antes con dicho valor.

Para lograr una arquitectura eficiente, se implementó dicho producto a través del uso de desplazamientos y sumas. Cada vez que se realiza un desplazamiento a derecha de los bits de un registro, esto corresponde a dividir el valor por 2, por lo tanto:

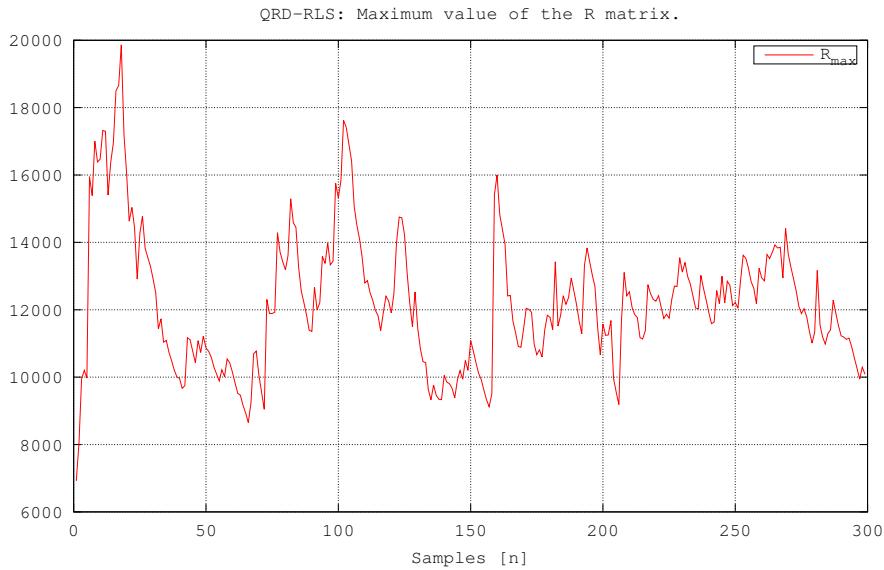


Figura 5.17: Máximo valor de la matriz R en función del número de iteración, $\lambda = 0,95$

$$R \gg 1 = 0,5 \cdot R \quad (5.1)$$

$$R \gg 2 = 0,25 \cdot R \quad (5.2)$$

$$R \gg 3 = 0,125 \cdot R$$

$$R \gg 4 = 0,0625 \cdot R$$

Luego, teniendo dichos resultados y sumándolos, se podría llegar a un factor de olvido λ equivalente a 0,9375:

$$(R \gg 1) + (R \gg 2) + (R \gg 3) + (R \gg 4) = \quad (5.3)$$

$$0,5 \cdot R + 0,25 \cdot R + 0,125 \cdot R + 0,0625 \cdot R =$$

$$0,9375 \cdot R$$

Lo cual corresponde a un factor de olvido que podría ser utilizado para procesos no estacionarios. Utilizando la técnica descripta anteriormente, se incorporó una entrada de dos bits en el hardware desarrollado que brinda la posibilidad de elegir como factor de olvido los valores 1, 0,9375, 0,875 y 0,75. Una vez realizada esta corrección, se llegó a la implementación final de hardware de descomposición QR.

5.8. Resultados

En la presente sección se detallan los resultados obtenidos durante el ensayo del procesador implementado.

5.8.1. Integración del hardware con Octave para la estimación de un sistema

El *script* desarrollado en la plataforma Octave se divide en cuatro etapas. La primera consiste en la inicialización del sistema. En la misma, se borra la memoria y se crean las señales que se utilizarán en las distintas implementaciones del algoritmo RLS. La señal de entrada $u(n)$ se genera utilizando una distribución normal, y posteriormente es normalizada dividiéndola por el máximo valor del arreglo. El sistema es definido a través del vector de pesos W , el cual posee seis elementos definidos de forma arbitraria. El uso de seis elementos es en concordancia con el hardware desarrollado, el cual descompone matrices de dimensión 7×7 (seis elementos corresponden a los pesos y un elemento a la señal deseada). Luego, a través de la función `filter()` se obtiene la señal deseada $d(n)$. El objetivo del *script* es obtener una estimación del sistema, a través de la definición de un vector de pesos estimado W_{est} , de forma tal que su salida, $y(n)$ minimice la función de costo $J(n)$:

$$J(n) = |e(n)|^2 = |d(n) - y(n)|^2 \quad (5.4)$$

La segunda etapa del *script* implementa el algoritmo RLS original [12], sin el uso de descomposición QR. Las ecuaciones se plantean aplicando el lema de inversión de matrices, a partir de las cuales se obtiene el vector de pesos y la señal de salida estimados recursivamente.

La tercera etapa del *script* implementa el algoritmo QR-RLS, de la misma forma que se describe en [12]. Como resultados de esta etapa, no sólo se obtiene el vector de pesos y la señal de salida estimados, sino también la matriz R .

Finalmente, en la última etapa del *script*, se pausa el sistema y se solicita el uso del hardware para el cálculo de la matriz R utilizando los vectores de entrada generados. Se realiza este cálculo utilizando las herramientas descriptas previamente (software qr_testbench conectado vía serial al kit de desarrollo), y posteriormente se comparan los resultados. Para obtener el vector de pesos estimado, se aplica una inversión de la matriz R provista por el hardware utilizando una función de Octave.

A continuación se exponen los gráficos de resultados:

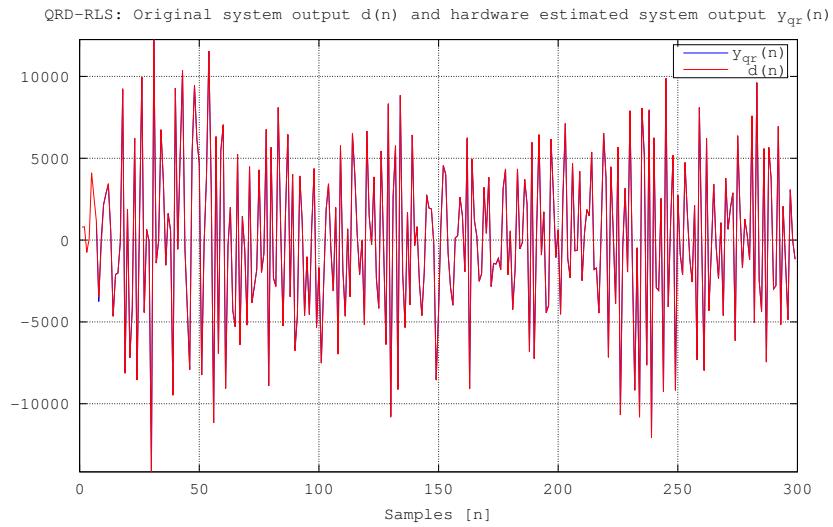


Figura 5.18: Señal de salida del sistema original y señal de salida del sistema estimado por el hardware

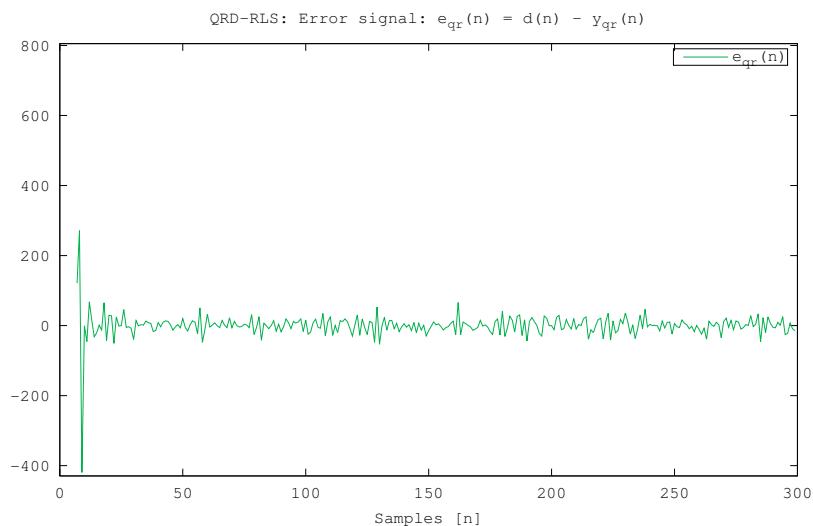


Figura 5.19: Señal de error basada en la función de costo utilizando el hardware

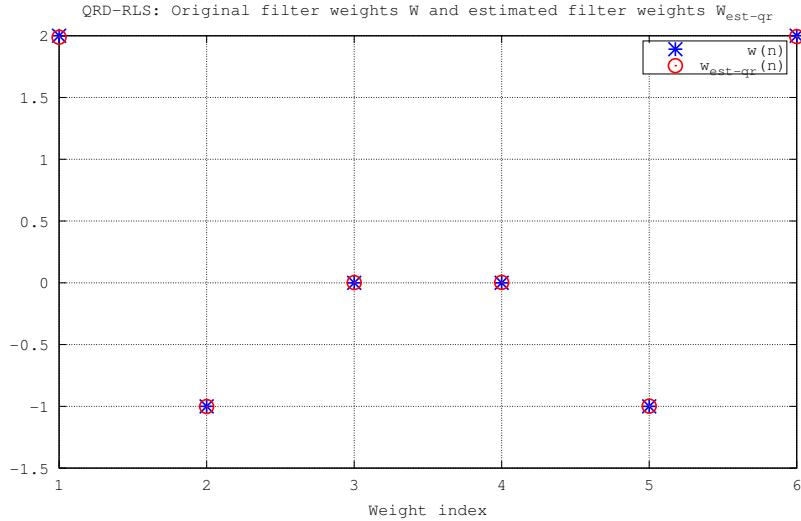


Figura 5.20: Pesos originales vs Pesos estimados utilizando el hardware

Como se puede apreciar en los gráficos, los resultados obtenidos a través de la matriz calculada por hardware coinciden con los parámetros definidos en el sistema original con un mínimo margen de error, por lo cual es posible utilizar satisfactoriamente el hardware desarrollado para la estimación del vector de pesos de un sistema.

5.8.2. Análisis del error en la estimación de múltiples sistemas

Una vez comprobado el funcionamiento adecuado para la estimación de un sistema, se procedió a realizar el cálculo para 1000 sistemas, obteniendo métricas del error. Las métricas tomadas en este caso, para cada sistema estimado, fueron las siguientes:

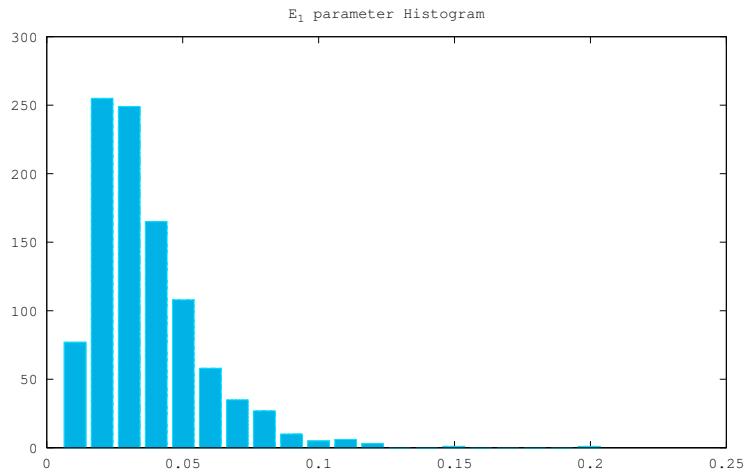
$$E_1 = \frac{1}{N_w} \sqrt{\sum_{i=1}^{N_w} (W_i - \hat{W}_i)^2} \quad (5.5)$$

donde N_w corresponde al número de elementos del vector de pesos (6 para el presente trabajo), y:

$$E_2 = \text{Max}\{|W_i - \hat{W}_i| \quad \forall \quad 1 \leq i \leq N_w\} \quad (5.6)$$

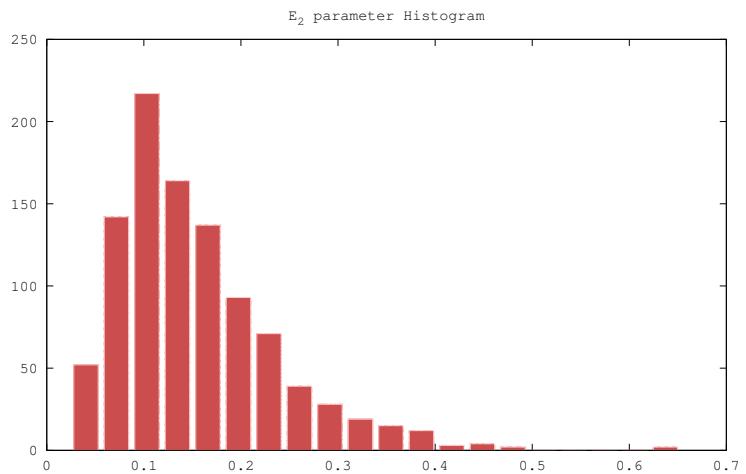
Para ambas métricas, se presenta a continuación el histograma, la media y la varianza que resultan del cálculo de las 1000 iteraciones.

Métrica E_1

Figura 5.21: Histograma de la métrica E_1

- **Media:** 0,036533
- **Varianza:** $4,1081 \cdot 10^{-4}$

Métrica E_2

Figura 5.22: Histograma de la métrica E_2

- **Media:** 0,15334
- **Varianza:** $6,8030 \cdot 10^{-3}$

Se puede observar que, para 1000 iteraciones, los resultados son consistentes con el análisis realizado para la estimación de un sistema.

5.8.3. Precisión del sistema

El uso de punto flotante es costoso en términos de hardware, y lleva a diseños que son ineficientes para dispositivos FPGA. La aritmética de punto fijo, utilizada en el presente trabajo, resulta en una implementación en hardware eficiente.

En contrapartida, la aritmética de punto fijo reduce la precisión e introduce dos tipos de errores, de redondeo y de truncamiento. Ambos errores ocurren cuando el resultado requiere más bits que aquella longitud de bits reservada luego de un cálculo. Estos conflictos deben ser manejados cuidadosamente para prevenir *overflow*, lo que desembocaría en resultados incorrectos.

El análisis del error se realiza para todo el procesador de descomposición QR para determinar la relación entre precisión y área. Uno de los mayores problemas al trabajar con aritmética de punto fijo es prevenir el *overflow*.

Para hacerlo, todas las entradas son normalizadas antes de empezar la descomposición. La normalización es realizada dividiendo cada elemento por el más grande, en valor absoluto. De esta forma los elementos estarán siempre entre -1 y 1 . Posterior a este proceso, se aplica una constante que asegure que los números se mantengan por debajo del límite numérico del hardware.

El análisis del error es realizado al comparar la aritmética de punto fijo, que resulta del hardware, contra la aritmética de punto flotante de doble precisión, que resulta de la implementación en lenguaje C. El análisis del error es realizada para matrices de 7×7 utilizando precisión de punto fijo de 16 bits.

Análisis de la condición de *overflow*

Con el objetivo de obtener una cota para evitar alcanzar la condición de *overflow*, se realizó el siguiente análisis. La matriz R calculada por el hardware resuelve la matriz $\Phi^{1/2}(n)$. La matriz $\Phi(n)$ se define a través de la siguiente fórmula:

$$\Phi(n) = \sum_{i=1}^n \lambda^{n-i} \mathbf{u}(i) \mathbf{u}^H(i) \quad (5.7)$$

Si se plantea un vector $\mathbf{u}(i)$, conformado en todos sus elementos por un único valor considerado máximo, al cual llamaremos U_{MAX} , la matriz toma la siguiente forma:

$$\Phi(n) = \sum_{i=1}^n \lambda^{n-i} U_{\text{MAX}}^2 \mathbf{1} = U_{\text{MAX}}^2 \mathbf{1} \sum_{i=1}^n \lambda^{n-i} \quad (5.8)$$

donde

1: Corresponde a la matriz $M \times N$ conformada por unos

Para encontrar el valor máximo que pueden tomar los elementos de la matriz, se debe encontrar el máximo valor que puede alcanzar la serie, la cual será referenciada como $s(n)$. Se analiza la misma observando los primeros tres valores:

$$s(n) = \sum_{i=1}^n \lambda^{n-i} \quad (5.9)$$

$$\begin{cases} n = 1 & \implies s(n) = \lambda^{1-1} = \lambda^0 \\ n = 2 & \implies s(n) = \lambda^{2-1} + \lambda^{1-1} = \lambda^1 + \lambda^0 \\ n = 3 & \implies s(n) = \lambda^{3-1} + \lambda^{2-1} + \lambda^{1-1} = \lambda^2 + \lambda^1 + \lambda^0 \end{cases}$$

Al analizar los valores que adopta la serie, se observa que se trata de la definición de una serie geométrica, con la diferencia de que los términos aparecen ordenados del mayor al menor. Luego, para una serie geométrica $g(n)$ se tienen los siguientes valores:

$$g(n) = \sum_{i=0}^n \lambda^n \quad (5.10)$$

$$\begin{cases} n = 0 & \implies g(n) = \lambda^0 \\ n = 1 & \implies g(n) = \lambda^0 + \lambda^1 \\ n = 2 & \implies g(n) = \lambda^0 + \lambda^1 + \lambda^2 \end{cases}$$

El máximo valor que puede adoptar la serie, es el valor para la cual converge. Dado que λ será un número menor a uno, la serie convergerá a:

$$\frac{1}{1 - \lambda} \quad (5.11)$$

Aplicando 5.11 a la ecuación 5.8, se deduce que el máximo valor de la matriz $\Phi(n)$, el cual llamaremos Φ_{MAX} , será:

$$\Phi_{\text{MAX}} = U_{\text{MAX}}^2 \frac{1}{1 - \lambda} \quad (5.12)$$

Finalmente, dado que la matriz R resuelve la raíz cuadrada, se tiene:

$$\Phi_{\text{MAX}}^{1/2} = U_{\text{MAX}} \sqrt{\frac{1}{1 - \lambda}} \quad (5.13)$$

Ejemplo: Si se utilizan 16 bits de resolución, y un valor de factor de olvido $\lambda = 0,9375$, el máximo valor que deben poseer los elementos del vector de entrada se obtiene a partir de la siguiente resolución:

$$2^{16-1} = U_{\text{MAX}} \sqrt{\frac{1}{1 - 0,9375}} \quad (5.14)$$

$$32768 = 4U_{\text{MAX}} \implies U_{\text{MAX}} = 8192$$

Matrices R

Si se define al resultado utilizando aritmética de punto flotante de doble precisión como el valor ideal, y al resultado de precisión de punto fijo de 16 bits como el valor estimado, es posible realizar un cálculo del error como sigue.

En primer lugar, se debe caracterizar el resultado de una matriz R a través de un único valor finito. Para ello se calcula el valor promedio de todos los elementos de la matriz R, de la siguiente forma:

$$R_{\text{double}} = \frac{\sum_{i=1}^7 \sum_{j=1}^7 R_{i,j}}{7 \cdot 7} \quad (5.15)$$

$$R_{\text{int}} = \frac{\sum_{i=1}^7 \sum_{j=1}^7 \hat{R}_{i,j}}{7 \cdot 7} \quad (5.16)$$

Se generan 300 iteraciones de descomposición QR y se grafican a continuación los resultados:

Se puede observar en la figura 5.23 que las señales de ambos sistemas se encuentran muy cercanas una a la otra, presentando un mínimo margen de error.

Matriz de error

Como métrica del error, se toma el valor medio de todos los elementos de la matriz de error, resultante de la diferencia entre la matriz ideal y la matriz estimada, para cada iteración. De esta forma:

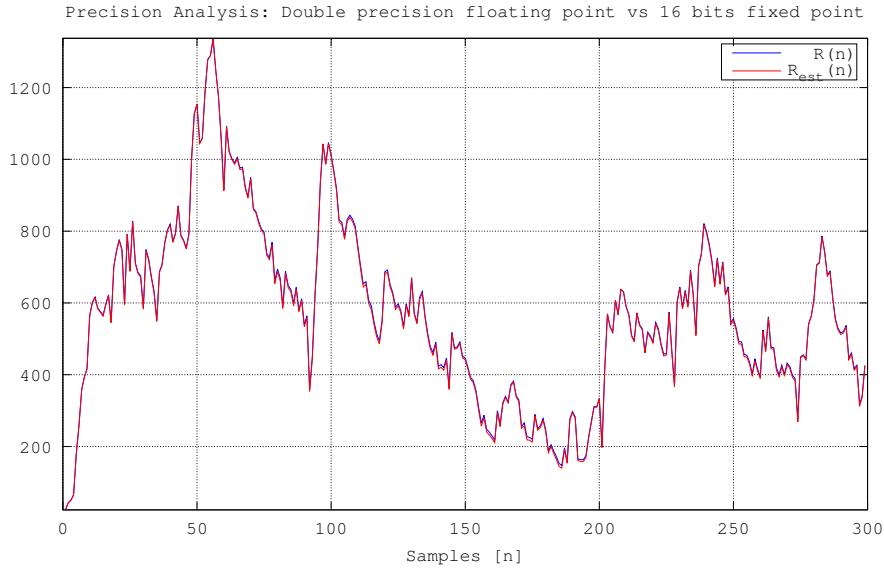


Figura 5.23: Salidas ideal y estimada superpuestas

$$R_{err} = \frac{\sum_{i=1}^7 \sum_{j=1}^7 |R_{i,j} - \hat{R}_{i,j}|}{7 \cdot 7} \quad (5.17)$$

Se generan 300 iteraciones de descomposición QR y se grafican a continuación los resultados:

Se puede observar que, en un hardware en el cual la representación numérica se encuentra limitada para números en el rango de -32768 a $+32767$, los valores de error oscilan entre -2 y 10 . Se presentan a continuación los mismos resultados en términos del valor relativo al mayor número representable, en este caso 32767 , y en porcentaje sobre dicho valor:

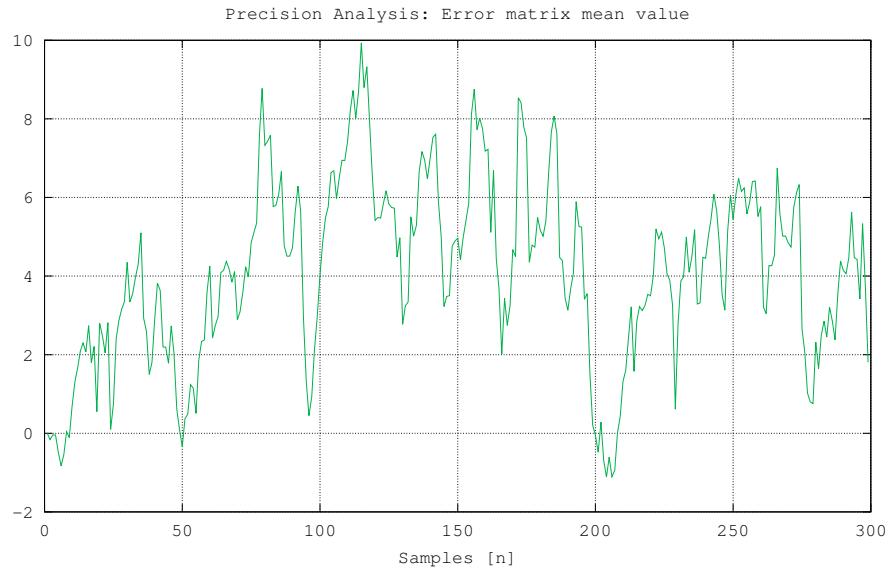


Figura 5.24: Valor medio de la matriz de error

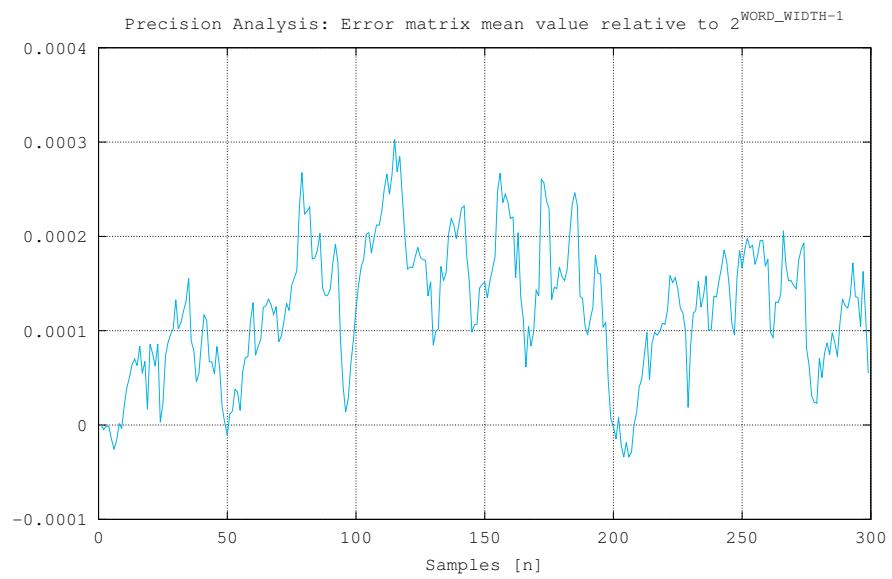


Figura 5.25: Valor medio de la matriz de error relativo al máximo número representable

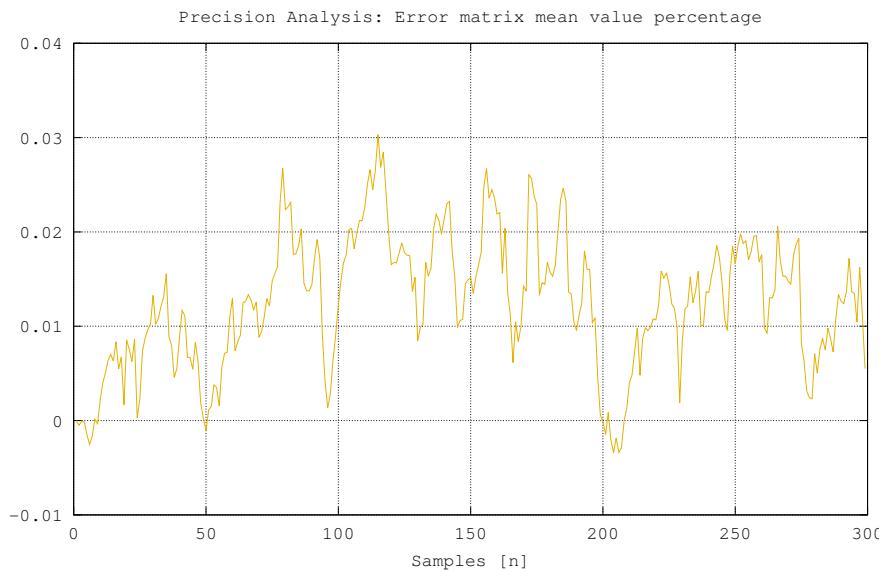


Figura 5.26: Valor medio de la matriz de error en porcentaje

5.8.4. Parámetros de Timing

Máximo clock de operación

Una vez finalizado el proceso de síntesis, la herramienta Xilinx ISE informa el máximo *clock* con el cual se puede operar el hardware implementado. A partir del valor de este parámetro se deducen los restantes.

Throughput

Se define al *throughput* como la máxima cantidad de matrices que es posible computar en un segundo. Una vez inicializado el sistema, la cantidad de ciclos de *clock* requeridos para calcular una matriz en cada uno de los estados es la siguiente:

Por lo tanto, el total de ciclos requeridos responde a la siguiente relación:

$$\text{Ciclos} = 11 + \text{WORD_WIDTH} \times 7 \quad (5.18)$$

Luego, conociendo la máxima frecuencia de operación, es posible calcular el *throughput* a través de la siguiente fórmula:

$$\text{Throughput} = \frac{f}{11 + \text{WORD_WIDTH} \times 7} \quad (5.19)$$

Estado	Ciclos Requeridos
wait_vector	1 ciclo
shift_down	1 ciclo
copy_matrix	1 ciclo
load_cycle1	1 ciclo
process_cycle1	WORD_WIDTH ciclos
load_cycle2	1 ciclo
process_cycle2	WORD_WIDTH ciclos
load_cycle3	1 ciclo
process_cycle3	WORD_WIDTH ciclos
load_cycle4	1 ciclo
process_cycle4	WORD_WIDTH ciclos
load_cycle5	1 ciclo
process_cycle5	WORD_WIDTH ciclos
load_cycle6	1 ciclo
process_cycle6	WORD_WIDTH ciclos
load_cycle7	1 ciclo
process_cycle7	WORD_WIDTH ciclos
output_results	1 ciclo

Tabla 5.2: Ciclos requeridos por cada estado de procesamiento

Update Delay

En las publicaciones citadas, se encontró que una de las métricas utilizadas consiste en la cantidad de tiempo que se requiere para procesar una muestra. Se llama a este parámetro *update delay*. Se deduce que este parámetro corresponde a la inversa del *throughput*:

$$\text{Update Delay} = \frac{1}{\text{Throughput}} \quad (5.20)$$

Latencia

Para analizar la latencia, es necesario calcular la cantidad de segundos requeridos desde que se inicializa el sistema hasta lograr el primer resultado deseado. En el caso del hardware desarrollado, en el proceso de inicialización del sistema, se requiere que se procesen 8 vectores de entrada hasta lograr el primer resultado correcto. Luego, de la ecuación 5.18 que resulta del cálculo de *throughput*, se deduce que la latencia obedece a la siguiente relación:

$$\text{Latencia} = 8 \cdot \frac{11 + \text{WORD_WIDTH} \times 7}{f} = \frac{8}{\text{Throughput}} \quad (5.21)$$

En la siguiente tabla se exponen los resultados de cada uno de los parámetros presentados anteriormente:

FPGA	Palabra	Máximo Clock	Throughput	Update delay	Latencia
Virtex 7	8 bits	133,806 MHz	1.99710 M	0.50072 μ s	4.0058 μ s
Virtex 7	12 bits	121,522 MHz	1.27918 M	0.78175 μ s	6.2540 μ s
Virtex 7	16 bits	119,202 MHz	0.96912 M	1.03186 μ s	8.2549 μ s
Artix 7	8 bits	104,461 MHz	1.55912 M	0.64139 μ s	5.1311 μ s
Artix 7	12 bits	89,314 MHz	0.94015 M	1.06366 μ s	8.5093 μ s
Artix 7	16 bits	87,563 MHz	0.71189 M	1.40470 μ s	11.2376 μ s
Spartan 6	8 bits	74,425 MHz	1.11082 M	0.90024 μ s	7.2019 μ s
Spartan 6	12 bits	71,398 MHz	0.75156 M	1.33057 μ s	10.6446 μ s
Spartan 6	16 bits	70,332 MHz	0.57180 M	1.74885 μ s	13.9908 μ s
Spartan 3E	8 bits	41,120 MHz	0.61373 M	1.6294 μ s	13.035 μ s
Spartan 3E	12 bits	39,336 MHz	0.41406 M	2.4151 μ s	19.321 μ s
Spartan 3E	16 bits	39,503 MHz	0.32116 M	3.1137 μ s	24.910 μ s

Tabla 5.3: Resultados de *timing* para distintos dispositivos FPGA

Resultados en formato gráfico

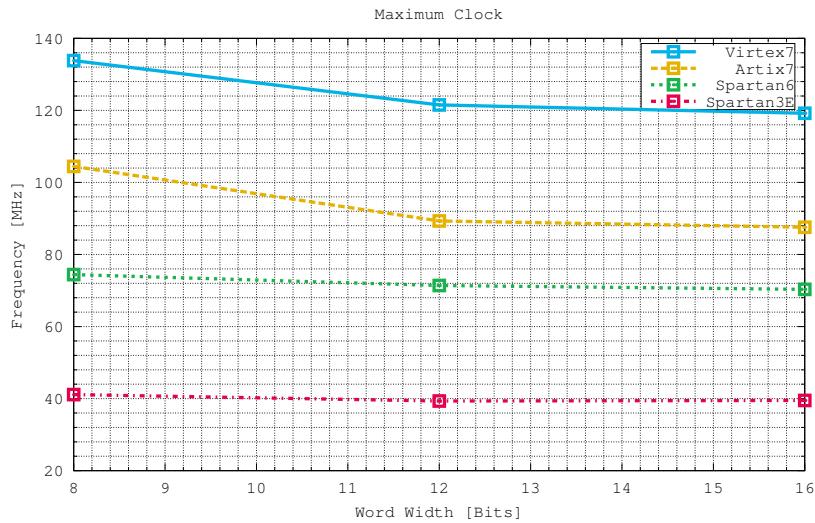


Figura 5.27: Máximo *clock* de operación para distintos dispositivos FPGA

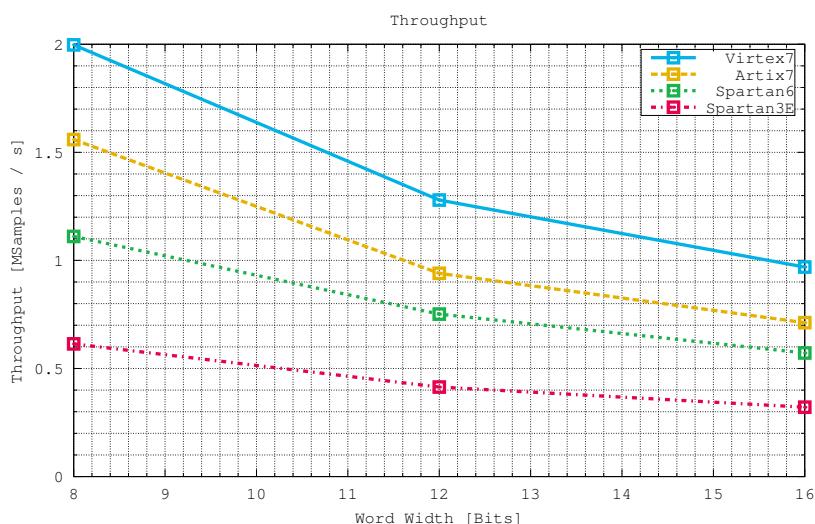


Figura 5.28: *Throughput* para distintos dispositivos FPGA

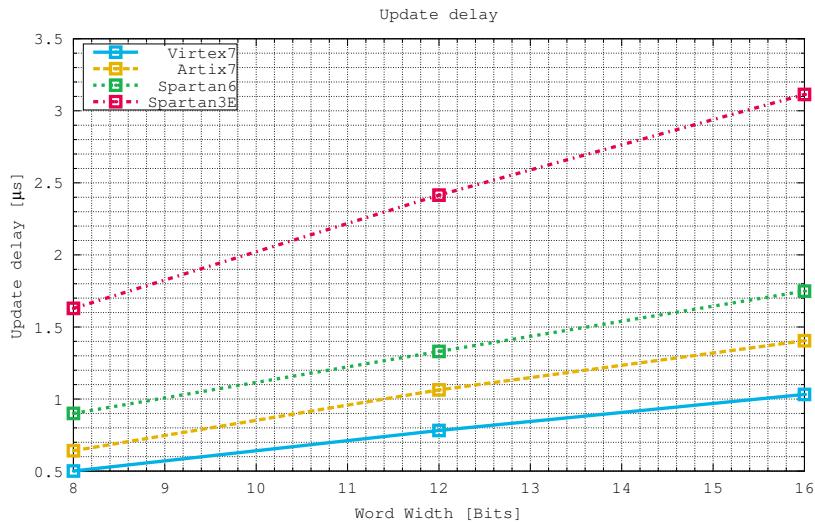
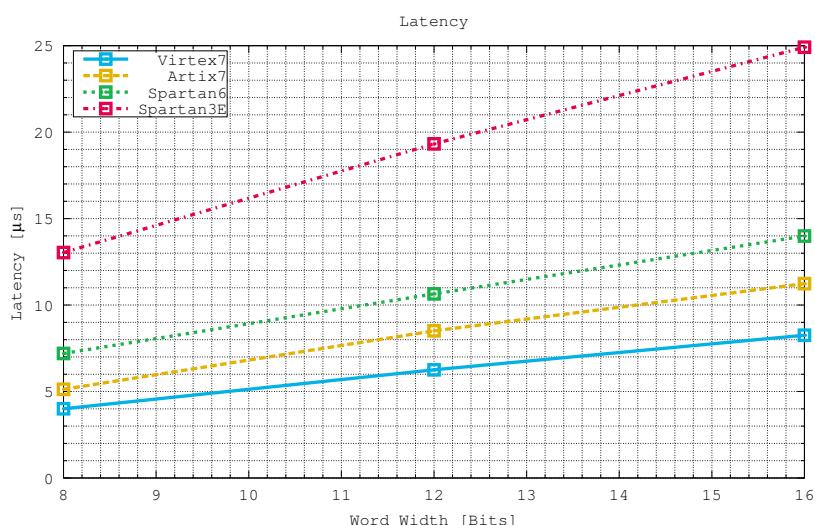
Figura 5.29: *Update delay* para distintos dispositivos FPGA

Figura 5.30: Latencia para distintos dispositivos FPGA

Contraste de resultados de parámetros de timing

La primera publicación presentada [13] corresponde a un procesador sintetizado en dispositivos Altera. Considerando que dicho hardware calcula matrices de 64×9 y números complejos, es esperable que el rendimiento del mismo sea menor. En este caso, el dato que es posible comparar corresponde al *throughput*. El mejor caso de los tres listados que corresponde al uso de mapeo directo, que corresponde a aproximadamente 0.2 *MSamples* por segundo. El hardware implementado en el presente trabajo puede alcanzar un *throughput* 10 veces mayor, pero cabe aclarar que el mismo calcula matrices de menor rango (7×7) en el dominio de los números reales.

Posteriormente se presentó una publicación de Xilinx [14] en la cual se sintetiza un procesador que incluye la dimensión 7×7 dentro de los resultados. No se especifica el número de bits de ancho de palabra utilizado. El *update delay* presentado para dicho procesador corresponde a $22,62 \mu s$, valor que es 22 veces más lento que el valor calculado para el procesador del presente trabajo utilizando 16 bits. El procesador del presente trabajo responde con mayor velocidad, pero es probable que el procesador presentado por Xilinx posea un mayor número de bits de precisión, lo cual explicaría el origen de esta diferencia.

Finalmente, la publicación de la Universidad de Victoria [15] expone resultados para un procesador de matrices de dimensión 3×4 , utilizando como mínimo 18 bits. Podemos comparar dichos resultados con el presente procesador para 16 bits. En el caso de la publicación, el procesador presenta un *throughput* de $2,13 \text{ MSamples} / s$, resultado muy similar al del presente trabajo que corresponde a $1,99 \text{ MSamples} / s$. El hardware de la Universidad es ligeramente más rápido, pero menos eficiente, dado que calcula matrices de menor dimensión. Cómo se explicó en la sección 3.4.1, un mapeo directo implica procesadores que no se utilizan el 100% y presentan tiempos ociosos debido a que, para realizar el cálculo, deben esperar a que otros procesadores actualicen la entrada de datos.

5.8.5. Recursos de FPGA utilizados

A continuación se presentan los resultados de área de chip obtenidos del reporte de síntesis del hardware desarrollado para 4 tecnologías de dispositivos FPGA Xilinx.

FPGA	Ancho de Palabra	LUTs	FFs	Slices
Virtex 7	8 bits	2257 (1 %)	2323	636 (1 %)
Virtex 7	12 bits	2619 (1 %)	3076	967 (1 %)
Virtex 7	16 bits	3560 (1 %)	4119	1306 (1 %)
Artix 7	8 bits	2218 (1 %)	2612	1002 (2 %)
Artix 7	12 bits	2640 (1 %)	3172	1036 (3 %)
Artix 7	16 bits	3554 (2 %)	4269	1783 (5 %)
Spartan 6	8 bits	2309 (8 %)	2647	794 (11 %)
Spartan 6	12 bits	2699 (9 %)	3184	1004 (14 %)
Spartan 6	16 bits	3564 (13 %)	4313	1329 (19 %)
Spartan 3E	8 bits	3539 (38 %)	1211	1903 (40 %)
Spartan 3E	12 bits	5299 (56 %)	1793	2856 (61 %)
Spartan 3E	16 bits	7053 (75 %)	2310	3813 (81 %)

Tabla 5.4: Máximo *clock* de operación

Resultados en formato gráfico

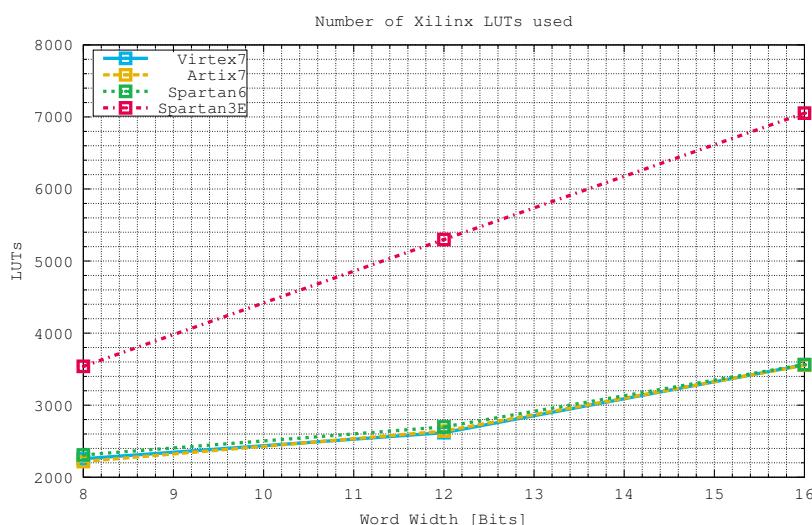
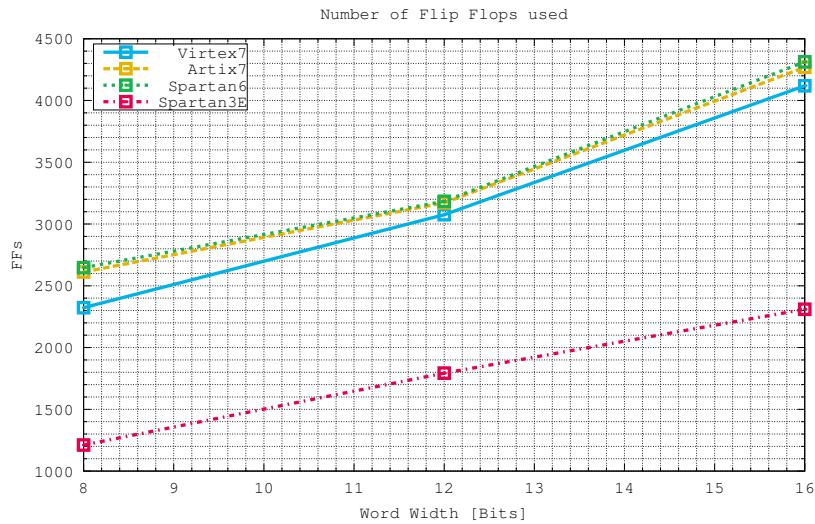
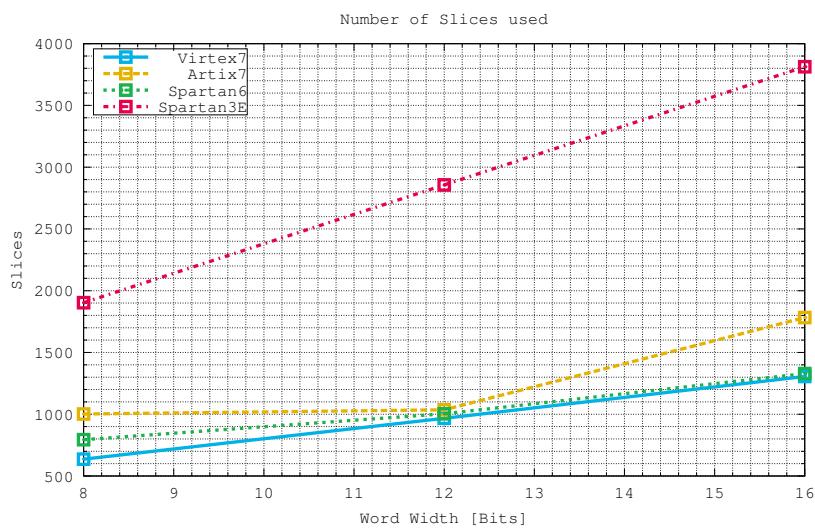


Figura 5.31: Número de *lookup tables* utilizadas

Figura 5.32: Número de *flip flops* utilizadosFigura 5.33: Número de *slices* utilizados

Contraste de resultados de área de chip

El análisis comparativo de área se realiza únicamente contra la última publicación citada [15], correspondiente a la Universidad de Victoria. Dado que la primera publicación citada corresponde a un hardware desarrollado en dispositivos FPGA Altera, sería difícil hacer una comparación entre sus elementos lógicos “LEs” y los elementos lógicos “LUTs” de la tecnología Xilinx. Por lo cual en este caso, el análisis es omitido. Por otro lado, la segunda publicación, realizada por Xilinx, no incluye detalles de porcentajes de área de chip utilizados, por lo cual tampoco se considera.

La publicación [15] presenta diversas diferencias con la arquitectura desarrollada en el presente trabajo. Al tratarse de arquitecturas diferentes, una comparación directa resulta imposible. En principio, la matriz de la publicación es de menor dimensión (3×4), y el mapeo es directo, implicando un mayor número de procesadores. Asimismo, se incluye un bloque de hardware de descomposición adicional utilizado para extraer la matriz Q , la cual no se incluye en el hardware del presente trabajo debido a que no es utilizada para la implementación del filtro adaptativo.

Teniendo en cuenta estas consideraciones, se observa que el hardware presentado en [15] utiliza un 8% de LUTs y un 11% de *slices* para una implementación de 18 bits de ancho de palabra en un dispositivo FPGA Virtex5. Naturalmente, los porcentajes son mayores a los presentados en el presente trabajo, que se encuentran en el 1% tanto para LUTs como para *slices*, lo cual puede explicarse por el hecho de que el hardware desarrollado posee sólo 4 procesadores, en contraste con los 21 procesadores de la implementación presentada en [15].

Capítulo 6

Conclusiones y trabajos a futuro

El alcance propuesto para el presente trabajo se basó en los siguientes puntos:

- IP Core codificado en el lenguaje Verilog de un procesador de descomposición QR.
- Resultado de mediciones pertinentes al diseño del procesador.
- Análisis comparativo de procesamiento entre el procesador desarrollado y desarrollos de terceros.
- Proposición de trabajos futuros y/o mejoras.

Se logró implementar satisfactoriamente un procesador de descomposición QR basado en la arquitectura de Walke[11]. Dicha implementación se resume en 14 códigos fuente que contienen la lógica y descripción del hardware en lenguaje Verilog. Adicionalmente, se desarrollaron distintas herramientas que permitieron ensayar el procesador en base a diferentes necesidades.

Como parte del ensayo del procesador, se obtuvieron métricas desde diferentes fuentes. Por un lado, se presentó el resumen de síntesis para cuatro tecnologías FPGA diferentes, utilizando tres valores de longitud de palabra. Dicho resumen permitió conocer el porcentaje de área de ocupación en el chip requerida para implementar el procesador. Este dato es de gran importancia dado que permite entender la cantidad de recursos necesarios para integrar el procesador como un módulo de una implementación mayor, como por ejemplo, un *transceiver* digital.

Se verificó que, en dispositivos FPGA de alta gama, la ocupación se encuentra dentro del 1 %, lo cual permite concluir que el mismo podría ser incluido en un proyecto utilizando dichas tecnologías, sin generar un impacto apreciable en el área de chip. Se obtuvieron parámetros de *timing*, como lo es el máximo *clock* de operación, el cual permite definir la latencia y el *throughput* del hardware. Se obtuvo como resultado que el hardware es capaz

de procesar hasta 2 *MSamples* por segundo, el cual es un resultado muy satisfactorio. Por otro lado, se logró identificar la precisión del procesador contrastándolo contra una implementación en software utilizando aritmética de punto flotante de doble precisión, y adicionalmente realizar una estimación de los pesos de un sistema ejemplo mediante el agregado de ecuaciones de sustitución en software. En base a los resultados obtenidos se puede analizar el rendimiento de la implementación y determinar en qué aplicaciones puede ser utilizada.

En base a los resultados de síntesis y de medición, fue posible contrastar la implementación con otras publicaciones similares, con ciertas limitaciones originadas en el hecho de que las arquitecturas y los dispositivos FPGA utilizados son diferentes. Como conclusión general, se puede decir que se implementó una arquitectura capaz de resolver matrices de mayor rango que otras (7×7 en contraste con 3×4) utilizando menor porcentaje de área de chip y velocidad similar, si se toma como referencia la publicación de la Universidad de Victoria [15], la cual presentaba más semejanzas entre las tres analizadas.

Este trabajo presenta una primera implementación de un procesador de descomposición QR, con un gran número de posibilidades para mejorar y optimizar el diseño. Como potenciales trabajos a futuro, en el contexto de la presente tesis, se destacan los siguientes:

1. Complementar el hardware desarrollado, que consta únicamente del procesador de descomposición QR, con el hardware de sustitución, con el objetivo de conseguir el hardware requerido para la obtención de los pesos de un filtro adaptativo.
2. Ensayar el *beamformer* en un ambiente de prueba real, mediante el uso de *front-ends* con antenas y conversores AD.
3. Modificar el *pipeline* y realizar otro tipo de optimizaciones, con el objeto de lograr mejorar las métricas (aumentar el máximo *clock* utilizable, disminuir la latencia y aumentar el *throughput*, o reducir el área de chip).
4. Modificar el hardware para operar con números complejos, representando las componentes IQ de la modulación en cuadratura.
5. Parametrizar el hardware para operar con distintos números de filas y columnas, como por ejemplo, matrices de $N \times N$ igual a 3×3 , 4×4 o 5×5 . En función del número de columnas, se podrán utilizar 2 antenas + 1 señal deseada, 3 antenas + 1 señal deseada ó 4 antenas + 1 señal deseada, respectivamente.
6. Modificar el hardware para operar con otros mapeos diferentes al de Walke[11], con mayor número de procesadores para lograr *throughputs* mas altos.
7. Sintetizar y ensayar el hardware en un dispositivo FPGA *state of the art*, como por ejemplo Virtex7.

8. Incorporar la posibilidad de cambiar el algoritmo de rotación (algoritmo CORDIC en el hardware implementado) por otros, como ejemplo Gram Schmidt, o SGR (Squared Givens Rotations).
9. Modificar el hardware para operar con memoria RAM en lugar de una unidad de registros.

Apéndice A

Código fuente Verilog del procesador QR desarrollado

A.1. Fuente qr_processor.v

```
1 // _____  
2 // Copyright (c) 2014 SDR and Wireless Group FIUBA.  
3 // Confidential property of SDR and Wireless Group FIUBA.  
4 //  
5 // The use of the software, documentation, methodologies and other  
6 // information contained herein is governed solely by the associated  
7 // license agreements. Any inconsistent use shall be deemed to be a  
8 // misappropriation of the intellectual property of SDR and Wireless  
9 // Group FIUBA and treated accordingly.  
10 //  
11 // _____  
12 // Description :  
13 // _____  
14 //  
15 // This hardware describes the IP Core of a QR decomposition processor  
16 // implemented following the Walke architecture.  
17 //  
18 // The processors inputs and outputs are registered  
19 //  
20 // The purpose is to supply an input matrix row by row and recursively  
// compute  
21 // the R matrix of the QR decomposition. This hardware description is ideal  
// to  
22 // implement a beamformer. In that case, each column of the matrix would  
23 // an antenna, except for the last one, which represents the desired input,  
24 // and each row would represent a sample. When a new sample is obtained, the  
25 // whole R matrix doesn't need to be computed, since the recursive  
26 // architecture updates the entire matrix with only one processing cycle.  
27 //
```

```

28 // _____
29 // File name:
30 // _____
31 //
32 // qr-processor.v
33 //
34 //

35 // Interface:
36 // _____
37 //
38 // Clock, reset & enable inputs:
39 //   - clk      : Posedge active clock input (logic, 1 bit).
40 //   - arst     : High active asynchronous reset (logic, 1 bit).
41 //   - srst     : High active synchronous reset (logic, 1 bit).
42 //   - enable    : Synchronous enable (logic, 1 bit).
43 //
44 // Data inputs:
45 //   - start    : Supply a high pulse to start a new calculation.
46 //   - lambda_option : Specifies the forgetting factor option.
47 //   - x_in_1    : First column of the new vector.
48 //   - x_in_2    : Second column of the new vector.
49 //   - x_in_3    : Third column of the new vector.
50 //   - x_in_4    : Fourth column of the new vector.
51 //   - x_in_5    : Fifth column of the new vector.
52 //   - x_in_6    : Sixth column of the new vector.
53 //   - x_in_7    : Seventh column of the new vector.
54 //   - out_row   : Address to select each of the output's rows.
55 //
56 // Data outputs:
57 //   - ready    : When '1' the result is ready (logic, 1 bit) and the
58 //                 output matrix will be sent, each row on each cycle,
59 //                 on the y_out outputs.
60 //   - y_out_1   : First column of the output matrix.
61 //   - y_out_2   : Second column of the output matrix.
62 //   - y_out_3   : Third column of the output matrix.
63 //   - y_out_4   : Fourth column of the output matrix.
64 //   - y_out_5   : Fifth column of the output matrix.
65 //   - y_out_6   : Sixth column of the output matrix.
66 //   - y_out_7   : Seventh column of the output matrix.
67 //
68 // Parameters:
69 //   - WORD_WIDTH      : Word width in number of bits
70 //                         (positive integer, default: 16).
71 //   - CLOG2_WORD_WIDTH : Number of bits of the CORDIC iteration counter
72 //                         (positive integer, for 16 iterations default:
73 //                           4),
74 //   - ROWS            : Number of Rows of the input matrix
75 //                         (positive integer, default: 8).
76 //   - COLUMNS          : Number of Columns of the input matrix
77 //                         (positive integer, default: 8).
78 //

79 // History:
80 // _____
81 //
82 //   - May 06, 2015 - Federico Damian Camarda - Original version.

```

```

83 // 
84 //
85 // -----
86 // ****
87 // Interface
88 // -----
89 module qr_processor #(
90   // Parameters
91   // -----
92   parameter WORD_WIDTH      = 16 ,
93   parameter ROW_IDX         = 3 ,
94   parameter ROWS            = 7 ,
95   parameter COLUMNS         = 7
96 ) (
97   // -----
98   // Clock, reset & enable inputs
99   // -----
100  input  wire                  clk ,
101  input  wire                  arst ,
102  input  wire                  srst ,
103  input  wire                  enable ,
104  // -----
105  // Data inputs
106  // -----
107  input  wire                  start ,
108  input  wire [1                :0] lambda_option ,
109  input  wire [WORD_WIDTH-1:0]  x_in_1 ,
110  input  wire [WORD_WIDTH-1:0]  x_in_2 ,
111  input  wire [WORD_WIDTH-1:0]  x_in_3 ,
112  input  wire [WORD_WIDTH-1:0]  x_in_4 ,
113  input  wire [WORD_WIDTH-1:0]  x_in_5 ,
114  input  wire [WORD_WIDTH-1:0]  x_in_6 ,
115  input  wire [WORD_WIDTH-1:0]  x_in_7 ,
116  input  wire [WORD_WIDTH-1:0]  out_row ,
117  // -----
118  // Data outputs
119  // -----
120  output wire                 ready ,
121  output wire [WORD_WIDTH-1:0] y_out_1 ,
122  output wire [WORD_WIDTH-1:0] y_out_2 ,
123  output wire [WORD_WIDTH-1:0] y_out_3 ,
124  output wire [WORD_WIDTH-1:0] y_out_4 ,
125  output wire [WORD_WIDTH-1:0] y_out_5 ,
126  output wire [WORD_WIDTH-1:0] y_out_6 ,
127  output wire [WORD_WIDTH-1:0] y_out_7
128 );
129 );
130 // -----
131 // ****
132 // -----
133 // Architecture

```

```

134 // ****
135
136 // Symbolic state declaration.
137 localparam [4:0]
138   WAIT_VECTOR      = 5'd0,
139   SHIFT_DOWN       = 5'd1,
140   COPY_MATRIX      = 5'd2,
141   LOAD_CYCLE1     = 5'd3,
142   PROCESS_CYCLE1  = 5'd4,
143   LOAD_CYCLE2     = 5'd5,
144   PROCESS_CYCLE2  = 5'd6,
145   LOAD_CYCLE3     = 5'd7,
146   PROCESS_CYCLE3  = 5'd8,
147   LOAD_CYCLE4     = 5'd9,
148   PROCESS_CYCLE4  = 5'd10,
149   LOAD_CYCLE5     = 5'd11,
150   PROCESS_CYCLE5  = 5'd12,
151   LOAD_CYCLE6     = 5'd13,
152   PROCESS_CYCLE6  = 5'd14,
153   LOAD_CYCLE7     = 5'd15,
154   PROCESS_CYCLE7  = 5'd16,
155   OUTPUT_RESULTS   = 5'd17;
156
157 // _____
158 // Internal signals
159 // _____
160
161 // FSM Signals.
162 reg [4:0]           state_reg , state_next ;
163
164 // Variables to store and operate with matrices and angles.
165 reg shift , first_vector_reg , first_vector_next ;
166 reg [WORD_WIDTH-1:0] future_matrix_reg [ROWS+1:0][COLUMNS+1:0];
167 reg [WORD_WIDTH-1:0] future_matrix_next [ROWS+1:0][COLUMNS+1:0];
168 reg [WORD_WIDTH:0]   future_angles_reg [ROWS+1:0];
169 reg [WORD_WIDTH:0]   future_angles_next [ROWS+1:0];
170 reg [WORD_WIDTH-1:0] current_matrix_reg [ROWS+1:0][COLUMNS+1:0];
171 reg [WORD_WIDTH-1:0] current_matrix_next [ROWS+1:0][COLUMNS+1:0];
172 reg [WORD_WIDTH:0]   current_angles_reg [ROWS+1:0];
173 reg [WORD_WIDTH:0]   current_angles_next [ROWS+1:0];
174 reg [WORD_WIDTH-1:0] output_matrix_reg [ROWS+1:0][COLUMNS+1:0];
175 reg [WORD_WIDTH-1:0] output_matrix_next [ROWS+1:0][COLUMNS+1:0];
176
177 // Iterators for reset, copy and shift functions.
178 integer i,j;
179
180 // Input and output processor variables.
181 reg load , ready_next , ready_reg , start_reg ;
182 reg [WORD_WIDTH-1:0] bc_01_x , bc_01_R , ic_01_x , ic_01_R ,
183                                ic_02_x , ic_02_R , ic_03_x , ic_03_R ;
184 reg [WORD_WIDTH :0]   ic_01_angle , ic_02_angle , ic_03_angle ;
185 wire [WORD_WIDTH :0]   bc_01_angle ;
186 wire [WORD_WIDTH-1:0] bc_01_R_new , ic_01_R_new , ic_01_x_new ,
187                                ic_02_R_new , ic_02_x_new , ic_03_R_new , ic_03_x_new
188                                ;
189 wire bc_01_ready , ic_01_ready , ic_02_ready , ic_03_ready
190                                ;
191 // Instantiate Circuits: One Boundary Cell and three Internal Cells.

```

```

192 // Boundary Cell.
193 boundary_cell #(
194   // _____
195   // Parameters
196   // _____
197   .WORD_WIDTH      (WORD.WIDTH)
198 ) bc_01 (
199   // _____
200   // Clock, reset & enable inputs
201   // _____
202   .clk            (clk),
203   .rst            (rst),
204   .srst           (srst),
205   .enable          (enable),
206   .load            (load),
207   // _____
208   // Data inputs
209   // _____
210   .x              (bc_01_x),
211   .R              (bc_01_R),
212   .lambda_option  (lambda_option),
213   // _____
214   // Data outputs
215   // _____
216   .R_new          (bc_01_R_new),
217   .angle          (bc_01_angle),
218   .ready          (bc_01_ready)
219 );
220
221 // Internal Cell 1.
222 internal_cell #(
223   // _____
224   // Parameters
225   // _____
226   .WORD_WIDTH      (WORD.WIDTH)
227 ) ic_01 (
228   // _____
229   // Clock, reset & enable inputs
230   // _____
231   .clk            (clk),
232   .rst            (rst),
233   .srst           (srst),
234   .enable          (enable),
235   .load            (load),
236   // _____
237   // Data inputs
238   // _____
239   .x              (ic_01_x),
240   .R              (ic_01_R),
241   .angle          (ic_01_angle),
242   .lambda_option  (lambda_option),
243   // _____
244   // Data outputs
245   // _____
246   .R_new          (ic_01_R_new),
247   .x_new          (ic_01_x_new),
248   .ready          (ic_01_ready)
249 );
250
251 // Internal Cell 2.
252 internal_cell #(
253   // _____

```

```

254 // Parameters
255 // _____
256 .WORD_WIDTH      (WORD_WIDTH)
257 ) ic_02 (
258 // _____
259 // Clock, reset & enable inputs
260 // _____
261 .clk              (clk),
262 .arst             (arst),
263 .srst             (srst),
264 .enable            (enable),
265 .load              (load),
266 // _____
267 // Data inputs
268 // _____
269 .x                (ic_02_x),
270 .R                (ic_02_R),
271 .angle             (ic_02_angle),
272 .lambda_option    (lambda_option),
273 // _____
274 // Data outputs
275 // _____
276 .R_new             (ic_02_R_new),
277 .x_new             (ic_02_x_new),
278 .ready              (ic_02_ready)
279 );
280
281 // Internal Cell 3.
282 internal_cell #(
283 // _____
284 // Parameters
285 // _____
286 .WORD_WIDTH      (WORD_WIDTH)
287 ) ic_03 (
288 // _____
289 // Clock, reset & enable inputs
290 // _____
291 .clk              (clk),
292 .arst             (arst),
293 .srst             (srst),
294 .enable            (enable),
295 .load              (load),
296 // _____
297 // Data inputs
298 // _____
299 .x                (ic_03_x),
300 .R                (ic_03_R),
301 .angle             (ic_03_angle),
302 .lambda_option    (lambda_option),
303 // _____
304 // Data outputs
305 // _____
306 .R_new             (ic_03_R_new),
307 .x_new             (ic_03_x_new),
308 .ready              (ic_03_ready)
309 );
310
311 // FSMD state & data registers.
312 always @(posedge clk, posedge arst)
313 begin
314     // Reset behaviour.
315     if (arst) begin

```

```

316     state_reg      <= WAIT_VECTOR;
317     first_vector_reg <= 1'b1;
318     ready_reg       <= 1'b0;
319     start_reg       <= 1'b0;
320
321 // Zero the matrices and angle registers .
322 for (i=0;i<=ROWS+1;i=i+1) begin
323     for (j=0;j<=COLUMNS+1;j=j+1) begin
324         future_matrix_reg[i][j] <= {WORD_WIDTH{1'b0}};
325         current_matrix_reg[i][j] <= {WORD_WIDTH{1'b0}};
326         output_matrix_reg[i][j] <= {WORD_WIDTH{1'b0}};
327     end
328     future_angles_reg[i] <= {WORD_WIDTH{1'b0}};
329     current_angles_reg[i] <= {WORD_WIDTH{1'b0}};
330 end
331
332 else if (srst) begin
333     state_reg      <= WAIT_VECTOR;
334     first_vector_reg <= 1'b1;
335     ready_reg       <= 1'b0;
336     start_reg       <= 1'b0;
337
338 // Zero the matrices and angle registers .
339 for (i=0;i<=ROWS+1;i=i+1) begin
340     for (j=0;j<=COLUMNS+1;j=j+1) begin
341         future_matrix_reg[i][j] <= {WORD_WIDTH{1'b0}};
342         current_matrix_reg[i][j] <= {WORD_WIDTH{1'b0}};
343         output_matrix_reg[i][j] <= {WORD_WIDTH{1'b0}};
344     end
345     future_angles_reg[i] <= {WORD_WIDTH{1'b0}};
346     current_angles_reg[i] <= {WORD_WIDTH{1'b0}};
347 end
348 end
349
350
351 else begin
352     // FSM next-state .
353     state_reg      <= state_next;
354     first_vector_reg <= first_vector_next;
355     ready_reg       <= ready_next;
356     start_reg       <= start;
357
358     if (shift) begin
359         // Shift matrix and angles rows down. The last row is discarded .
360         for (i=ROWS+1;i>0;i=i-1) begin
361             for (j=COLUMNS+1;j>0;j=j-1) begin
362                 future_matrix_reg[i][j] <= future_matrix_reg[i-1][j];
363                 current_matrix_reg[i][j] <= current_matrix_reg[i-1][j];
364             end
365             future_angles_reg[i] <= future_angles_reg[i-1];
366             current_angles_reg[i] <= current_angles_reg[i-1];
367         end
368     end
369     else begin
370         for (i=0;i<=ROWS+1;i=i+1) begin
371             for (j=0;j<=COLUMNS+1;j=j+1) begin
372                 future_matrix_reg[i][j] <= future_matrix_next[i][j];
373                 current_matrix_reg[i][j] <= current_matrix_next[i][j];
374                 output_matrix_reg[i][j] <= output_matrix_next[i][j];
375             end
376             future_angles_reg[i] <= future_angles_next[i];
377             current_angles_reg[i] <= current_angles_next[i];

```

```

378         end
379     end
380   end
381 end
382
383 // Next-state logic & data path functional units/routing
384 // @* cannot be used because Xilinx ISE cannot interprete it for
385 // multidimensional arrays for Spartan3E.
386 always @(state_reg , first_vector_reg , start_reg , ready_reg , out_row ,
387   future_matrix_reg[0][0] , future_matrix_reg[0][1] ,
388   future_matrix_reg[0][2] , future_matrix_reg[0][3] ,
389   future_matrix_reg[0][4] , future_matrix_reg[0][5] ,
390   future_matrix_reg[0][6] , future_matrix_reg[0][7] ,
391   future_matrix_reg[0][8] , future_matrix_reg[1][0] ,
392   future_matrix_reg[1][1] , future_matrix_reg[1][2] ,
393   future_matrix_reg[1][3] , future_matrix_reg[1][4] ,
394   future_matrix_reg[1][5] , future_matrix_reg[1][6] ,
395   future_matrix_reg[1][7] , future_matrix_reg[1][8] ,
396   future_matrix_reg[2][0] , future_matrix_reg[2][1] ,
397   future_matrix_reg[2][2] , future_matrix_reg[2][3] ,
398   future_matrix_reg[2][4] , future_matrix_reg[2][5] ,
399   future_matrix_reg[2][6] , future_matrix_reg[2][7] ,
400   future_matrix_reg[2][8] , future_matrix_reg[3][0] ,
401   future_matrix_reg[3][1] , future_matrix_reg[3][2] ,
402   future_matrix_reg[3][3] , future_matrix_reg[3][4] ,
403   future_matrix_reg[3][5] , future_matrix_reg[3][6] ,
404   future_matrix_reg[3][7] , future_matrix_reg[3][8] ,
405   future_matrix_reg[4][0] , future_matrix_reg[4][1] ,
406   future_matrix_reg[4][2] , future_matrix_reg[4][3] ,
407   future_matrix_reg[4][4] , future_matrix_reg[4][5] ,
408   future_matrix_reg[4][6] , future_matrix_reg[4][7] ,
409   future_matrix_reg[4][8] , future_matrix_reg[5][0] ,
410   future_matrix_reg[5][1] , future_matrix_reg[5][2] ,
411   future_matrix_reg[5][3] , future_matrix_reg[5][4] ,
412   future_matrix_reg[5][5] , future_matrix_reg[5][6] ,
413   future_matrix_reg[5][7] , future_matrix_reg[5][8] ,
414   future_matrix_reg[6][0] , future_matrix_reg[6][1] ,
415   future_matrix_reg[6][2] , future_matrix_reg[6][3] ,
416   future_matrix_reg[6][4] , future_matrix_reg[6][5] ,
417   future_matrix_reg[6][6] , future_matrix_reg[6][7] ,
418   future_matrix_reg[6][8] , future_matrix_reg[7][0] ,
419   future_matrix_reg[7][1] , future_matrix_reg[7][2] ,
420   future_matrix_reg[7][3] , future_matrix_reg[7][4] ,
421   future_matrix_reg[7][5] , future_matrix_reg[7][6] ,
422   future_matrix_reg[7][7] , future_matrix_reg[7][8] ,
423   future_matrix_reg[8][0] , future_matrix_reg[8][1] ,
424   future_matrix_reg[8][2] , future_matrix_reg[8][3] ,
425   future_matrix_reg[8][4] , future_matrix_reg[8][5] ,
426   future_matrix_reg[8][6] , future_matrix_reg[8][7] ,
427   future_matrix_reg[8][8] , current_matrix_reg[0][0] ,
428   current_matrix_reg[0][1] , current_matrix_reg[0][2] ,
429   current_matrix_reg[0][3] , current_matrix_reg[0][4] ,
430   current_matrix_reg[0][5] , current_matrix_reg[0][6] ,
431   current_matrix_reg[0][7] , current_matrix_reg[0][8] ,
432   current_matrix_reg[1][0] , current_matrix_reg[1][1] ,
433   current_matrix_reg[1][2] , current_matrix_reg[1][3] ,
434   current_matrix_reg[1][4] , current_matrix_reg[1][5] ,
435   current_matrix_reg[1][6] , current_matrix_reg[1][7] ,
436   current_matrix_reg[1][8] , current_matrix_reg[2][0] ,
437   current_matrix_reg[2][1] , current_matrix_reg[2][2] ,
438   current_matrix_reg[2][3] , current_matrix_reg[2][4] ,
439   current_matrix_reg[2][5] , current_matrix_reg[2][6] ,

```

```

440      current_matrix_reg[2][7], current_matrix_reg[2][8],
441      current_matrix_reg[3][0], current_matrix_reg[3][1],
442      current_matrix_reg[3][2], current_matrix_reg[3][3],
443      current_matrix_reg[3][4], current_matrix_reg[3][5],
444      current_matrix_reg[3][6], current_matrix_reg[3][7],
445      current_matrix_reg[3][8], current_matrix_reg[4][0],
446      current_matrix_reg[4][1], current_matrix_reg[4][2],
447      current_matrix_reg[4][3], current_matrix_reg[4][4],
448      current_matrix_reg[4][5], current_matrix_reg[4][6],
449      current_matrix_reg[4][7], current_matrix_reg[4][8],
450      current_matrix_reg[5][0], current_matrix_reg[5][1],
451      current_matrix_reg[5][2], current_matrix_reg[5][3],
452      current_matrix_reg[5][4], current_matrix_reg[5][5],
453      current_matrix_reg[5][6], current_matrix_reg[5][7],
454      current_matrix_reg[5][8], current_matrix_reg[6][0],
455      current_matrix_reg[6][1], current_matrix_reg[6][2],
456      current_matrix_reg[6][3], current_matrix_reg[6][4],
457      current_matrix_reg[6][5], current_matrix_reg[6][6],
458      current_matrix_reg[6][7], current_matrix_reg[6][8],
459      current_matrix_reg[7][0], current_matrix_reg[7][1],
460      current_matrix_reg[7][2], current_matrix_reg[7][3],
461      current_matrix_reg[7][4], current_matrix_reg[7][5],
462      current_matrix_reg[7][6], current_matrix_reg[7][7],
463      current_matrix_reg[7][8], current_matrix_reg[8][0],
464      current_matrix_reg[8][1], current_matrix_reg[8][2],
465      current_matrix_reg[8][3], current_matrix_reg[8][4],
466      current_matrix_reg[8][5], current_matrix_reg[8][6],
467      current_matrix_reg[8][7], current_matrix_reg[8][8],
468      output_matrix_reg[0][0], output_matrix_reg[0][1],
469      output_matrix_reg[0][2], output_matrix_reg[0][3],
470      output_matrix_reg[0][4], output_matrix_reg[0][5],
471      output_matrix_reg[0][6], output_matrix_reg[0][7],
472      output_matrix_reg[0][8], output_matrix_reg[1][0],
473      output_matrix_reg[1][1], output_matrix_reg[1][2],
474      output_matrix_reg[1][3], output_matrix_reg[1][4],
475      output_matrix_reg[1][5], output_matrix_reg[1][6],
476      output_matrix_reg[1][7], output_matrix_reg[1][8],
477      output_matrix_reg[2][0], output_matrix_reg[2][1],
478      output_matrix_reg[2][2], output_matrix_reg[2][3],
479      output_matrix_reg[2][4], output_matrix_reg[2][5],
480      output_matrix_reg[2][6], output_matrix_reg[2][7],
481      output_matrix_reg[2][8], output_matrix_reg[3][0],
482      output_matrix_reg[3][1], output_matrix_reg[3][2],
483      output_matrix_reg[3][3], output_matrix_reg[3][4],
484      output_matrix_reg[3][5], output_matrix_reg[3][6],
485      output_matrix_reg[3][7], output_matrix_reg[3][8],
486      output_matrix_reg[4][0], output_matrix_reg[4][1],
487      output_matrix_reg[4][2], output_matrix_reg[4][3],
488      output_matrix_reg[4][4], output_matrix_reg[4][5],
489      output_matrix_reg[4][6], output_matrix_reg[4][7],
490      output_matrix_reg[4][8], output_matrix_reg[5][0],
491      output_matrix_reg[5][1], output_matrix_reg[5][2],
492      output_matrix_reg[5][3], output_matrix_reg[5][4],
493      output_matrix_reg[5][5], output_matrix_reg[5][6],
494      output_matrix_reg[5][7], output_matrix_reg[5][8],
495      output_matrix_reg[6][0], output_matrix_reg[6][1],
496      output_matrix_reg[6][2], output_matrix_reg[6][3],
497      output_matrix_reg[6][4], output_matrix_reg[6][5],
498      output_matrix_reg[6][6], output_matrix_reg[6][7],
499      output_matrix_reg[6][8], output_matrix_reg[7][0],
500      output_matrix_reg[7][1], output_matrix_reg[7][2],
501      output_matrix_reg[7][3], output_matrix_reg[7][4],

```

```

502     output_matrix_reg[7][5], output_matrix_reg[7][6],
503     output_matrix_reg[7][7], output_matrix_reg[7][8],
504     output_matrix_reg[8][0], output_matrix_reg[8][1],
505     output_matrix_reg[8][2], output_matrix_reg[8][3],
506     output_matrix_reg[8][4], output_matrix_reg[8][5],
507     output_matrix_reg[8][6], output_matrix_reg[8][7],
508     output_matrix_reg[8][8],
509     future_angles_reg[0], future_angles_reg[1], future_angles_reg[2],
510     future_angles_reg[3], future_angles_reg[4], future_angles_reg[5],
511     future_angles_reg[6], future_angles_reg[7], future_angles_reg[8],
512     current_angles_reg[0], current_angles_reg[1], current_angles_reg
513     [2],
514     current_angles_reg[3], current_angles_reg[4], current_angles_reg
515     [5],
516     current_angles_reg[6], current_angles_reg[7], current_angles_reg
517     [8],
518     start, x_in_1, x_in_2, x_in_3, x_in_4, x_in_5, x_in_6, x_in_7,
519     bc_01_ready, ic_01_ready, ic_02_ready, ic_03_ready, bc_01_R_new,
520     bc_01_angle, ic_01_x_new, ic_01_R_new, ic_02_x_new, ic_02_R_new,
521     ic_03_x_new, ic_03_R_new)
522 begin
523   // FSM default outputs and states.
524   state_next           = state_reg;
525   first_vector_next   = first_vector_reg;
526   for (i=0;i<=ROWS+1;i=i+1) begin
527     for (j=0;j<=COLUMNS+1;j=j+1) begin
528       future_matrix_next[i][j] = future_matrix_reg[i][j];
529       current_matrix_next[i][j] = current_matrix_reg[i][j];
530       output_matrix_next[i][j] = output_matrix_reg[i][j];
531     end
532   end
533   for (j=0;j<=COLUMNS+1;j=j+1) begin
534     future_angles_next[j]      = future_angles_reg[j];
535     current_angles_next[j]    = current_angles_reg[j];
536   end
537   shift                = 1'b0;
538   load                 = 1'b0;
539   ready_next            = 1'b0;
540   bc_01_x               = {WORD_WIDTH{1'b0}};
541   bc_01_R               = {WORD_WIDTH{1'b0}};
542   ic_01_x               = {WORD_WIDTH{1'b0}};
543   ic_01_R               = {WORD_WIDTH{1'b0}};
544   ic_02_x               = {WORD_WIDTH{1'b0}};
545   ic_02_R               = {WORD_WIDTH{1'b0}};
546   ic_03_x               = {WORD_WIDTH{1'b0}};
547   ic_03_R               = {WORD_WIDTH{1'b0}};
548
549   // FSM logic.
550   case (state_reg)
551     // WAIT_VECTOR: State to receive a vector an load it into the
552     // matrix registers.
553     WAIT_VECTOR: begin
554       ready_next           = 1'b1;
555       if (start_reg) begin
556         ready_next = 1'b0;
557         future_matrix_next[0][1] = x_in_1;
558         future_matrix_next[0][2] = x_in_2;
559         future_matrix_next[0][3] = x_in_3;
560         future_matrix_next[0][4] = x_in_4;

```

```

561         future_matrix_next[0][5] = x_in_5;
562         future_matrix_next[0][6] = x_in_6;
563         future_matrix_next[0][7] = x_in_7;
564         state_next              = SHIFT_DOWN;
565     end
566 end
567 // SHIFT_DOWN: State to move both matrices down.
568 SHIFT_DOWN: begin
569     shift      = 1'b1;
570     state_next = COPY_MATRIX;
571 end
572 // COPY_MATRIX: On the copy step, specific values of the future
573 // matrix must be copied to the current matrix.
574 COPY_MATRIX: begin
575     current_matrix_next[1][1] = future_matrix_reg[2][1];
576     current_matrix_next[1][2] = future_matrix_reg[2][2];
577     current_matrix_next[1][3] = future_matrix_reg[2][3];
578     current_matrix_next[1][4] = future_matrix_reg[2][4];
579     current_matrix_next[1][5] = future_matrix_reg[2][5];
580     current_matrix_next[1][6] = future_matrix_reg[2][6];
581     current_matrix_next[1][7] = future_matrix_reg[2][7];
582     current_matrix_next[2][2] = future_matrix_reg[3][2];
583     current_matrix_next[2][3] = future_matrix_reg[3][3];
584     current_matrix_next[2][4] = future_matrix_reg[3][4];
585     current_matrix_next[2][5] = future_matrix_reg[3][5];
586     current_matrix_next[2][6] = future_matrix_reg[3][6];
587     current_matrix_next[2][7] = future_matrix_reg[3][7];
588     current_matrix_next[3][3] = future_matrix_reg[4][3];
589     current_matrix_next[3][4] = future_matrix_reg[4][4];
590     current_matrix_next[3][5] = future_matrix_reg[4][5];
591     current_matrix_next[3][6] = future_matrix_reg[4][6];
592     current_matrix_next[4][4] = future_matrix_reg[5][4];
593     current_matrix_next[4][5] = future_matrix_reg[5][5];
594     for (j=0;j<=4;j=j+1) begin
595         current_angles_next[j] = future_angles_reg[j+1];
596     end
597     if (first_vector_reg) begin
598         first_vector_next = 1'b0;
599         state_next       = WAIT_VECTOR;
600     end
601     else begin
602         state_next       = LOAD_CYCLE1;
603     end
604 end
605 // LOAD_CYCLE1: Redirect registers to the cells inputs.
606 LOAD_CYCLE1: begin
607     bc_01_x      = future_matrix_reg[1][1];
608     bc_01_R      = future_matrix_reg[2][1];
609     ic_01_x      = current_matrix_reg[4][5];
610     ic_01_R      = current_matrix_reg[5][5];
611     ic_01_angle  = current_angles_reg[4];
612     ic_02_x      = current_matrix_reg[2][7];
613     ic_02_R      = current_matrix_reg[3][7];
614     ic_02_angle  = current_angles_reg[2];
615     ic_03_x      = current_matrix_reg[3][6];
616     ic_03_R      = current_matrix_reg[4][6];
617     ic_03_angle  = current_angles_reg[3];
618     load         = 1'b1;
619     state_next   = PROCESS_CYCLE1;
620 end
621 // PROCESS_CYCLE1: Redirect the cells outputs to registers.
622 PROCESS_CYCLE1: begin

```

```

623      bc_01_x      = future_matrix_reg [1][1];
624      bc_01_R      = future_matrix_reg [2][1];
625      ic_01_x      = current_matrix_reg [4][5];
626      ic_01_R      = current_matrix_reg [5][5];
627      ic_01_angle  = current_angles_reg [4];
628      ic_02_x      = current_matrix_reg [2][7];
629      ic_02_R      = current_matrix_reg [3][7];
630      ic_02_angle  = current_angles_reg [2];
631      ic_03_x      = current_matrix_reg [3][6];
632      ic_03_R      = current_matrix_reg [4][6];
633      ic_03_angle  = current_angles_reg [3];
634      if (bc_01_ready && ic_01_ready && ic_02_ready && ic_03_ready)
635          begin
636              future_matrix_next [1][1] = bc_01_R_new;
637              future_angles_next [1]   = bc_01_angle;
638              current_matrix_next [4][5] = ic_01_R_new;
639              current_matrix_next [5][5] = ic_01_x_new;
640              current_matrix_next [2][7] = ic_02_R_new;
641              current_matrix_next [3][7] = ic_02_x_new;
642              current_matrix_next [3][6] = ic_03_R_new;
643              current_matrix_next [4][6] = ic_03_x_new;
644              state_next             = LOAD_CYCLE2;
645          end
646      end
647      // LOAD_CYCLE2: Redirect registers to the cells inputs.
648      LOAD_CYCLE2: begin
649          bc_01_x      = current_matrix_reg [5][5];
650          bc_01_R      = current_matrix_reg [6][5];
651          ic_01_x      = future_matrix_reg [1][2];
652          ic_01_R      = future_matrix_reg [2][2];
653          ic_01_angle  = future_angles_reg [1];
654          ic_02_x      = current_matrix_reg [4][6];
655          ic_02_R      = current_matrix_reg [5][6];
656          ic_02_angle  = current_angles_reg [4];
657          ic_03_x      = current_matrix_reg [3][7];
658          ic_03_R      = current_matrix_reg [4][7];
659          ic_03_angle  = current_angles_reg [3];
660          load         = 1'b1;
661          state_next   = PROCESS_CYCLE2;
662      end
663      // PROCESS_CYCLE2: Redirect the cells outputs to registers.
664      PROCESS_CYCLE2: begin
665          bc_01_x      = current_matrix_reg [5][5];
666          bc_01_R      = current_matrix_reg [6][5];
667          ic_01_x      = future_matrix_reg [1][2];
668          ic_01_R      = future_matrix_reg [2][2];
669          ic_01_angle  = future_angles_reg [1];
670          ic_02_x      = current_matrix_reg [4][6];
671          ic_02_R      = current_matrix_reg [5][6];
672          ic_02_angle  = current_angles_reg [4];
673          ic_03_x      = current_matrix_reg [3][7];
674          ic_03_R      = current_matrix_reg [4][7];
675          ic_03_angle  = current_angles_reg [3];
676          if (bc_01_ready && ic_01_ready && ic_02_ready && ic_03_ready)
677              begin
678                  current_matrix_next [5][5] = bc_01_R_new;
679                  current_angles_next [5]   = bc_01_angle;
680                  future_matrix_next [1][2] = ic_01_R_new;
681                  future_matrix_next [2][2] = ic_01_x_new;
682                  current_matrix_next [4][6] = ic_02_R_new;
683                  current_matrix_next [5][6] = ic_02_x_new;
684                  current_matrix_next [3][7] = ic_03_R_new;

```

```

683         current_matrix_next[4][7] = ic_03_x_new;
684         state_next              = LOAD_CYCLE3;
685     end
686 end
687 // LOAD_CYCLE3: Redirect registers to the cells inputs.
688 LOAD_CYCLE3: begin
689     bc_01_x      = future_matrix_reg [2][2];
690     bc_01_R      = future_matrix_reg [3][2];
691     ic_01_x      = current_matrix_reg[5][6];
692     ic_01_R      = current_matrix_reg[6][6];
693     ic_01_angle  = current_angles_reg[5];
694     ic_02_x      = future_matrix_reg [1][3];
695     ic_02_R      = future_matrix_reg [2][3];
696     ic_02_angle  = future_angles_reg [1];
697     ic_03_x      = current_matrix_reg[4][7];
698     ic_03_R      = current_matrix_reg[5][7];
699     ic_03_angle  = current_angles_reg[4];
700     load        = 1'b1;
701     state_next   = PROCESS_CYCLE3;
702 end
703 // PROCESS_CYCLE3: Redirect the cells outputs to registers.
704 PROCESS_CYCLE3: begin
705     bc_01_x      = future_matrix_reg [2][2];
706     bc_01_R      = future_matrix_reg [3][2];
707     ic_01_x      = current_matrix_reg[5][6];
708     ic_01_R      = current_matrix_reg[6][6];
709     ic_01_angle  = current_angles_reg[5];
710     ic_02_x      = future_matrix_reg [1][3];
711     ic_02_R      = future_matrix_reg [2][3];
712     ic_02_angle  = future_angles_reg [1];
713     ic_03_x      = current_matrix_reg[4][7];
714     ic_03_R      = current_matrix_reg[5][7];
715     ic_03_angle  = current_angles_reg[4];
716     if (bc_01_ready && ic_01_ready && ic_02_ready && ic_03_ready)
717         begin
718             future_matrix_next [2][2] = bc_01_R_new;
719             future_angles_next [2]    = bc_01_angle;
720             current_matrix_next[5][6] = ic_01_R_new;
721             current_matrix_next[6][6] = ic_01_x_new;
722             future_matrix_next [1][3] = ic_02_R_new;
723             future_matrix_next [2][3] = ic_02_x_new;
724             current_matrix_next[4][7] = ic_03_R_new;
725             current_matrix_next[5][7] = ic_03_x_new;
726             state_next              = LOAD_CYCLE4;
727         end
728 end
729 // LOAD_CYCLE4: Redirect registers to the cells inputs.
730 LOAD_CYCLE4: begin
731     bc_01_x      = current_matrix_reg [6][6];
732     bc_01_R      = current_matrix_reg [7][6];
733     ic_01_x      = future_matrix_reg [2][3];
734     ic_01_R      = future_matrix_reg [3][3];
735     ic_01_angle  = future_angles_reg [2];
736     ic_02_x      = current_matrix_reg [5][7];
737     ic_02_R      = current_matrix_reg [6][7];
738     ic_02_angle  = current_angles_reg [5];
739     ic_03_x      = future_matrix_reg [1][4];
740     ic_03_R      = future_matrix_reg [2][4];
741     ic_03_angle  = future_angles_reg [1];
742     load        = 1'b1;
743     state_next   = PROCESS_CYCLE4;
744 end

```

```

744 // PROCESS_CYCLE4: Redirect the cells outputs to registers.
745 PROCESS_CYCLE4: begin
746     bc_01_x      = current_matrix_reg [6][6];
747     bc_01_R      = current_matrix_reg [7][6];
748     ic_01_x      = future_matrix_reg [2][3];
749     ic_01_R      = future_matrix_reg [3][3];
750     ic_01_angle  = future_angles_reg [2];
751     ic_02_x      = current_matrix_reg [5][7];
752     ic_02_R      = current_matrix_reg [6][7];
753     ic_02_angle  = current_angles_reg [5];
754     ic_03_x      = future_matrix_reg [1][4];
755     ic_03_R      = future_matrix_reg [2][4];
756     ic_03_angle  = future_angles_reg [1];
757     if (bc_01_ready && ic_01_ready && ic_02_ready && ic_03_ready)
758         begin
759             current_matrix_next [6][6] = bc_01_R_new;
760             current_angles_next [6]   = bc_01_angle;
761             future_matrix_next [2][3] = ic_01_x_new;
762             future_matrix_next [3][3] = ic_01_R_new;
763             current_matrix_next [5][7] = ic_02_R_new;
764             current_matrix_next [6][7] = ic_02_x_new;
765             future_matrix_next [1][4] = ic_03_R_new;
766             future_matrix_next [2][4] = ic_03_x_new;
767             state_next              = LOAD_CYCLE5;
768         end
769     end
770 // LOAD_CYCLE5: Redirect registers to the cells inputs.
771 LOAD_CYCLE5: begin
772     bc_01_x      = future_matrix_reg [3][3];
773     bc_01_R      = future_matrix_reg [4][3];
774     ic_01_x      = current_matrix_reg [6][7];
775     ic_01_R      = current_matrix_reg [7][7];
776     ic_01_angle  = current_angles_reg [6];
777     ic_02_x      = future_matrix_reg [2][4];
778     ic_02_R      = future_matrix_reg [3][4];
779     ic_02_angle  = future_angles_reg [2];
780     ic_03_x      = future_matrix_reg [1][5];
781     ic_03_R      = future_matrix_reg [2][5];
782     ic_03_angle  = future_angles_reg [1];
783     load = 1'b1;
784     state_next = PROCESS_CYCLE5;
785 end
786 // PROCESS_CYCLE5: Redirect the cells outputs to registers.
787 PROCESS_CYCLE5: begin
788     bc_01_x      = future_matrix_reg [3][3];
789     bc_01_R      = future_matrix_reg [4][3];
790     ic_01_x      = current_matrix_reg [6][7];
791     ic_01_R      = current_matrix_reg [7][7];
792     ic_01_angle  = current_angles_reg [6];
793     ic_02_x      = future_matrix_reg [2][4];
794     ic_02_R      = future_matrix_reg [3][4];
795     ic_02_angle  = future_angles_reg [2];
796     ic_03_x      = future_matrix_reg [1][5];
797     ic_03_R      = future_matrix_reg [2][5];
798     ic_03_angle  = future_angles_reg [1];
799     if (bc_01_ready && ic_01_ready && ic_02_ready && ic_03_ready)
800         begin
801             future_matrix_next [3][3] = bc_01_R_new;
802             future_angles_next [3]   = bc_01_angle;
803             current_matrix_next [6][7] = ic_01_R_new;

```

```

804         future_matrix_next [3][4] = ic_02_x_new;
805         future_matrix_next [1][5] = ic_03_R_new;
806         future_matrix_next [2][5] = ic_03_x_new;
807         state_next             = LOAD_CYCLE6;
808     end
809 end
810 // LOAD_CYCLE6: Redirect registers to the cells inputs.
811 LOAD_CYCLE6: begin
812     bc_01_x      = current_matrix_reg [7][7];
813     bc_01_R      = current_matrix_reg [8][7];
814     ic_01_x      = future_matrix_reg [3][4];
815     ic_01_R      = future_matrix_reg [4][4];
816     ic_01_angle  = future_angles_reg [3];
817     ic_02_x      = future_matrix_reg [1][6];
818     ic_02_R      = future_matrix_reg [2][6];
819     ic_02_angle  = future_angles_reg [1];
820     ic_03_x      = future_matrix_reg [2][5];
821     ic_03_R      = future_matrix_reg [3][5];
822     ic_03_angle  = future_angles_reg [2];
823     load         = 1'b1;
824     state_next   = PROCESS_CYCLE6;
825 end
826 // PROCESS_CYCLE6: Redirect the cells outputs to registers.
827 PROCESS_CYCLE6: begin
828     bc_01_x      = current_matrix_reg [7][7];
829     bc_01_R      = current_matrix_reg [8][7];
830     ic_01_x      = future_matrix_reg [3][4];
831     ic_01_R      = future_matrix_reg [4][4];
832     ic_01_angle  = future_angles_reg [3];
833     ic_02_x      = future_matrix_reg [1][6];
834     ic_02_R      = future_matrix_reg [2][6];
835     ic_02_angle  = future_angles_reg [1];
836     ic_03_x      = future_matrix_reg [2][5];
837     ic_03_R      = future_matrix_reg [3][5];
838     ic_03_angle  = future_angles_reg [2];
839     if (bc_01_ready && ic_01_ready && ic_02_ready && ic_03_ready)
840         begin
841             current_matrix_next[7][7] = bc_01_R_new;
842             current_angles_next[7]   = bc_01_angle;
843             future_matrix_next [3][4] = ic_01_R_new;
844             future_matrix_next [4][4] = ic_01_x_new;
845             future_matrix_next [1][6] = ic_02_R_new;
846             future_matrix_next [2][6] = ic_02_x_new;
847             future_matrix_next [2][5] = ic_03_R_new;
848             future_matrix_next [3][5] = ic_03_x_new;
849             state_next             = LOAD_CYCLE7;
850         end
851     end
852 // LOAD_CYCLE7: Redirect registers to the cells inputs.
853 LOAD_CYCLE7: begin
854     bc_01_x      = future_matrix_reg [4][4];
855     bc_01_R      = future_matrix_reg [5][4];
856     ic_01_x      = future_matrix_reg [1][7];
857     ic_01_R      = future_matrix_reg [2][7];
858     ic_01_angle  = future_angles_reg [1];
859     ic_02_x      = future_matrix_reg [3][5];
860     ic_02_R      = future_matrix_reg [4][5];
861     ic_02_angle  = future_angles_reg [3];
862     ic_03_x      = future_matrix_reg [2][6];
863     ic_03_R      = future_matrix_reg [3][6];
864     ic_03_angle  = future_angles_reg [2];
865     load         = 1'b1;

```

```

865     state_next = PROCESS_CYCLE7;
866   end
867   // PROCESS_CYCLE7: Redirect the cells outputs to registers.
868   PROCESS_CYCLE7: begin
869     bc_01_x      = future_matrix_reg[4][4];
870     bc_01_R      = future_matrix_reg[5][4];
871     ic_01_x      = future_matrix_reg[1][7];
872     ic_01_R      = future_matrix_reg[2][7];
873     ic_01_angle  = future_angles_reg[1];
874     ic_02_x      = future_matrix_reg[3][5];
875     ic_02_R      = future_matrix_reg[4][5];
876     ic_02_angle  = future_angles_reg[3];
877     ic_03_x      = future_matrix_reg[2][6];
878     ic_03_R      = future_matrix_reg[3][6];
879     ic_03_angle  = future_angles_reg[2];
880     if (bc_01_ready && ic_01_ready && ic_02_ready && ic_03_ready)
881       begin
882         future_matrix_next[4][4] = bc_01_R_new;
883         future_angles_next[4]   = bc_01_angle;
884         future_matrix_next[1][7] = ic_01_R_new;
885         future_matrix_next[2][7] = ic_01_x_new;
886         future_matrix_next[3][5] = ic_02_R_new;
887         future_matrix_next[4][5] = ic_02_x_new;
888         future_matrix_next[2][6] = ic_03_R_new;
889         future_matrix_next[3][6] = ic_03_x_new;
890         state_next             = OUTPUT_RESULTS;
891       end
892     end
893   // OUTPUT_RESULTS: When the decomposition is completed, the output
894   // is
895   // sent to the output_matrix registers.
896   OUTPUT_RESULTS: begin
897     for (i=0;i<=ROWS+1;i=i+1) begin
898       for (j=0;j<=COLUMNS+1;j=j+1) begin
899         if (j >= i) begin
900           output_matrix_next[i][j] = current_matrix_reg[i][j];
901         end
902         else begin
903           output_matrix_next[i][j] = {WORD_WIDTH{1'b0}};
904         end
905       end
906     end
907     state_next = WAIT_VECTOR;
908   endcase
909 end
910
911   // Redirect registers to the hardware outputs.
912   assign y_out_1 = output_matrix_reg[out_row][1];
913   assign y_out_2 = output_matrix_reg[out_row][2];
914   assign y_out_3 = output_matrix_reg[out_row][3];
915   assign y_out_4 = output_matrix_reg[out_row][4];
916   assign y_out_5 = output_matrix_reg[out_row][5];
917   assign y_out_6 = output_matrix_reg[out_row][6];
918   assign y_out_7 = output_matrix_reg[out_row][7];
919   assign ready   = ready_reg;
920
921   /**
922   ****
923 endmodule

```

A.2. Fuente boudary_cell.v

```

1 // _____
2 // Copyright (c) 2013 SDR and Wireless Group FIUBA.
3 // Confidential property of SDR and Wireless Group FIUBA.
4 //
5 // The use of the software, documentation, methodologies and other
6 // information contained herein is governed solely by the associated
7 // license agreements. Any inconsistent use shall be deemed to be a
8 // misappropriation of the intellectual property of SDR and Wireless
9 // Group FIUBA and treated accordingly.
10 //
11 //

12 // Description :
13 // _____
14 //
15 // This hardware implements a Boundary Cell of a QR Decomposition Processor.
16 //
17 // It's purpose is to take a current value of the R matrix (called R), the
18 // new input for that position (called x), and compute the new R value (
19 // called
20 // R_new) and the angle to feed the internal cells of the same row (called
21 // angle).
22 //
23 // The implementation is done with a CORDIC processor in vectoring mode,
24 // which
25 // accomplishes the calculation of the module and angle of a vector made
26 // using
27 // R as the "CORDIC x" and x as the "CORDIC y". The module of the results is
28 // the R_new value, and the angle is used for the internal cell's angle input
29 //
30 //
31 // boundary_cell.v
32 //
33 //

34 // File name:
35 // _____
36 //
37 // _____
38 // Clock, reset & enable inputs:
39 //   - clk      : Posedge active clock input (logic, 1 bit).
40 //   - arst    : High active asynchronous reset (logic, 1 bit).
41 //   - srst    : High active synchronous reset (logic, 1 bit).
42 //   - enable  : Synchronous enable (logic, 1 bit).

```

```

42 //      - load      : When '1' the cell starts processing .
43 //
44 // Data inputs:
45 //      - x          : New value of the input matrix .
46 //      - R          : Current value of the R matrix .
47 //
48 // Data outputs:
49 //      - R_new     : Cell's R value calculated .
50 //      - angle      : Angle calculated by the boundary cell to send to the
51 //                          internal cells of the same row .
52 //      - ready      : When '1' the result is ready (logic , 1 bit) .
53 //
54 // Parameters:
55 //      - WORD_WIDTH : Word width in number of bits (positive integer ,
56 //                          default: 16) .
57 //
58 //



---


59 // History:
60 // -----
61 //
62 //      - May 06, 2015 - Federico Damian Camarda - Original version .
63 //
64 //



---


65 //
66 // ****
67 // Interface
68 //
69 // ****
70 module boundary_cell #(
71   // Parameters
72   // -----
73   parameter WORD_WIDTH      = 16
74   ) (
75   // -----
76   // Clock , reset & enable inputs
77   //
78   input  wire  clk ,
79   input  wire  arst ,
80   input  wire  srst ,
81   input  wire  enable ,
82   input  wire  load ,
83   //
84   // Data inputs
85   //
86   input  wire [WORD_WIDTH-1:0] x ,
87   input  wire [WORD_WIDTH-1:0] R ,
88   input  wire [1           :0] lambda_option ,
89   //
90   // Data outputs
91   //
92   output wire [WORD_WIDTH-1:0] R_new ,
93   output wire [WORD_WIDTH  :0] angle ,
94   output wire                  ready
95 );

```

```

96 // ****
97 // ****
98 // ****
99 // Architecture
100 // ****
101 // _____
102 // Internal signals
103 // _____
104 wire [WORD.WIDTH:0] z_int;
105 reg signed [WORD.WIDTH-1:0] R_shifted;
106 wire vec_rot_n;
107 // _____
108
109 // Instantiate a CORDIC processor on vector mode to compute R_new
110 // and the angle values.
111 rotator #(
112   // _____
113   // Parameters
114   // _____
115   .WORD_WIDTH (WORD.WIDTH)
116 ) rotator (
117   // _____
118   // Clock, reset & enable inputs
119   // _____
120   // _____
121   // Data inputs
122   // _____
123   .clk (clk),
124   .arst (arst),
125   .srst (srst),
126   .enable (enable),
127   .load (load),
128   .vec_rot_n (vec_rot_n),
129   .x_i (R_shifted),
130   .y_i (x),
131   .z_i (z_int),
132   // _____
133   // Data outputs
134   // _____
135   .x_o (R_new),
136   .y_o (),
137   .z_o (angle),
138   .ready (ready)
139 );
140
141 // The CORDIC operates on vector mode with input angle 0.
142 assign vec_rot_n = 1'b1;
143 assign z_int = {WORD.WIDTH+1{1'b0}};
144
145 // The forgetting factor lambda is implemented as an option which defines
146 // right shifts and adds.
147 always @* begin
148   case(lambda_option)
149     // Forgetting factor lambda = 1.
150     2'b00 : begin

```

```

152     R_shifted <= R;
153   end
154   // Forgetting factor lambda = 1/2 + 1/4 + 1/8 + 1/16 = 0.9375.
155   2'b01 : begin
156     R_shifted <= ($signed(R) >>> 1) + ($signed(R) >>> 2) +
157       ($signed(R) >>> 3) + ($signed(R) >>> 4);
158   end
159   // Forgetting factor lambda = (1/2 + 1/4 + 1/8 = 0.875,
160   2'b10 : begin
161     R_shifted <= ($signed(R) >>> 1) + ($signed(R) >>> 2) +
162       ($signed(R) >>> 3);
163   end
164   // Forgetting factor lambda = 1/2 + 1/4 = 0.75
165   2'b11 : begin
166     R_shifted <= ($signed(R) >>> 1) + ($signed(R) >>> 2);
167   end
168 endcase
169 end
170 //
171 ****
172 endmodule

```

A.3. Fuente internal_cell.v

```

1 // _____
2 // Copyright (c) 2013 SDR and Wireless Group FIUBA.
3 // Confidential property of SDR and Wireless Group FIUBA.
4 //
5 // The use of the software, documentation, methodologies and other
6 // information contained herein is governed solely by the associated
7 // license agreements. Any inconsistent use shall be deemed to be a
8 // misappropriation of the intellectual property of SDR and Wireless
9 // Group FIUBA and treated accordingly.
10 //
11 //
12 // _____
13 // Description:
14 //
15 // This hardware implements an Internal Cell of a QR Decomposition Processor.
16 //
17 // It's purpose is to take a current value of the R matrix (called R), the
18 // new input for that position (called x) and the angle obtained from the
19 // Boundary Cell of the same row, to compute the new R value (called R_new)
20 // and
21 // the new x to feed the internal cell of the following row (called x_new).
22 //
23 // The implementation is done with a CORDIC processor in rotation mode, which
24 // accomplishes the calculation of the x and y coordinates of the rotation of
// a vector made using x as the "CORDIC x" and R as the "CORDIC y". The angle

```

```

25 // of the rotation is provided by the Boundary Cell and the x and y results
26 // of
27 // the rotated vector represent the R_new value, and the x_new value
28 // respectively.
29 //


---


30 // File name:
31 // _____
32 //
33 // internal_cell.v
34 //
35 //


---


36 // Interface:
37 // _____
38 //
39 // Clock, reset & enable inputs:
40 //   - clk      : Posedge active clock input (logic, 1 bit).
41 //   - arst     : High active asynchronous reset (logic, 1 bit).
42 //   - srst     : High active synchronous reset (logic, 1 bit).
43 //   - enable   : Synchronous enable (logic, 1 bit).
44 //   - load     : When '1' the cell starts processing.
45 //
46 // Data inputs:
47 //   - x        : New value of the input matrix.
48 //   - R        : Current value of the R matrix.
49 //   - angle    : Angle calculated by the boundary cell of the same row
50 //
51 // Data outputs:
52 //   - R_new    : Cell's R value calculated.
53 //   - x_new    : Cell's x value calculated.
54 //   - ready    : When '1' the result is ready (logic, 1 bit).
55 //
56 // Parameters:
57 //   - WORD_WIDTH : Word width in number of bits (positive integer,
58 //                 default: 16).
59 //
60 //


---


61 // History:
62 // _____
63 //
64 //   - May 06, 2015 - Federico Damian Camarda - Original version.
65 //
66 //


---


67 //
68 // ****
69 // Interface
70 //
71 module internal_cell #(
72   // -

```

```

73  // Parameters
74  // -----
75  parameter WORD_WIDTH = 16
76  ) (
77  // -----
78  // Clock, reset & enable inputs
79  // -----
80  input wire clk,
81  input wire arst,
82  input wire srst,
83  input wire enable,
84  input wire load,
85  // -----
86  // Data inputs
87  // -----
88  input wire [WORD_WIDTH-1:0] x,
89  input wire [WORD_WIDTH-1:0] R,
90  input wire [WORD_WIDTH : 0] angle,
91  input wire [1           : 0] lambda_option,
92  // -----
93  // Data outputs
94  // -----
95  output wire [WORD_WIDTH-1:0] R_new,
96  output wire [WORD_WIDTH-1:0] x_new,
97  output wire                 ready
98 );
99 // ****
100 //
101 // ****
102 // Architecture
103 // ****
104 //
105 // Internal signals
106 // -----
107 wire vec_rot_n;
108 reg signed [WORD_WIDTH-1:0] R_shifted;
109 //
110 //
111 //
112 //
113 // Instantiate a CORDIC processor on rotation mode to compute R_new and
114 // x_new values.
115 rotator #(
116   // -----
117   // Parameters
118   // -----
119   .WORD_WIDTH (WORD_WIDTH)
120 ) rotator (
121   // -----
122   // Clock, reset & enable inputs
123   // -----
124   // -----
125   // Data inputs
126   // -----
127   .clk          (clk),
128   .rst          (rst),

```

```

129      .srst      (srst),
130      .enable    (enable),
131      .load      (load),
132      .vec_rot_n (vec_rot_n),
133      .x_i       (x),
134      .y_i       (R_shifted),
135      .z_i       (angle),
136      // _____
137      // Data outputs
138      // _____
139      .x_o       (x_new),
140      .y_o       (R_new),
141      .z_o       (),
142      .ready     (ready)
143 );
144
145 // The CORDIC operates on rotation mode with the input angle supplied.
146 assign vec_rot_n = 1'b0;
147
148 // The forgetting factor lambda is implemented as an option which defines
149 // right shifts and adds.
150 always @* begin
151   case (lambda_option)
152     // Forgetting factor lambda = 1.
153     2'b00 : begin
154       R_shifted <= R;
155     end
156     // Forgetting factor lambda = 1/2 + 1/4 + 1/8 + 1/16 = 0.9375.
157     2'b01 : begin
158       R_shifted <= ($signed(R) >>> 1) + ($signed(R) >>> 2) +
159                   ($signed(R) >>> 3) + ($signed(R) >>> 4);
160     end
161     // Forgetting factor lambda = (1/2 + 1/4 + 1/8 = 0.875,
162     2'b10 : begin
163       R_shifted <= ($signed(R) >>> 1) + ($signed(R) >>> 2) +
164                   ($signed(R) >>> 3);
165     end
166     // Forgetting factor lambda = 1/2 + 1/4 = 0.75
167     2'b11 : begin
168       R_shifted <= ($signed(R) >>> 1) + ($signed(R) >>> 2);
169     end
170   endcase
171 end
172
173 // ****
174 endmodule

```

A.4. Fuente rotator.v

```

1 // _____
2 // Copyright (c) 2013 SDR and Wireless Group FIUBA.
3 // Confidential property of SDR and Wireless Group FIUBA.

```

```

4  //-
5  // The use of the software, documentation, methodologies and other
6  // information contained herein is governed solely by the associated
7  // license agreements. Any inconsistent use shall be deemed to be a
8  // misappropriation of the intellectual property of SDR and Wireless
9  // Group FIUBA and treated accordingly.
10 //
11 //

12 // Description :
13 // -----
14 //
15 // Enhanced CORDIC processor. It instantiate a preprocessor, an iterative
16 // CORDIC
17 // and a post_multiplier to successfully perform the rotation or vectoring
18 // CORDIC algorithms, for any angle and any input. The traditional CORDIC
19 // only
20 // works between -PI/2 and PI/2, and overflow may occur if the input
21 // multiplied
22 // by the gain exceeds the number representation. This enhanced version
23 // overcomes those issues by managing the whole angle domain and by
24 // restricting
25 // the inputs with one bit less to avoid overflow.
26 //
27 // Data formats :
28 //   * Input angle (z_0):
29 //     -PI = 100000...0001
30 //     +PI = 011111...1111
31 //     The angle representation is determined by the atan ROM. The first
32 //     value stands for the atan of 1, which represents 45 degrees.
33 //

34 // File name:
35 // -----
36 //
37 // rotator.v
38 //
39 //

40 // Interface :
41 // -----
42 //
43 //   Clock, reset & enable inputs:
44 //     - clk      : Posedge active clock input (logic, 1 bit).
45 //     - arst    : High active asynchronous reset (logic, 1 bit).
46 //     - srst    : High active synchronous reset (logic, 1 bit).
47 //     - enable  : High active synchronous enable (logic, 1 bit).
48 //
49 //   Data inputs:
50 //     - load    : When '1', a new input data is available (strobe, 1 bit).
51 //     - vec_rot_n : When '1' vectoring mode, when '0' rotation mode (logic
52 //                   , 1 bit).
53 //           - x_i    : Initial x component of input vector
54 //                         (two's complement, WORD_WIDTH bits).
55 //           - y_i    : Initial y component of input vector

```

```

55 //          (two's complement, WORD_WIDTH bits).
56 //      - z_i      : Rotation angle (two's complement, WORD_WIDTH+1 bits).
57 //
58 // Data outputs:
59 //      - x_0      : x component of the rotated vector
60 //                      (two's complement, WORD_WIDTH bits).
61 //      - y_0      : y component of the rotated vector
62 //                      (two's complement, WORD_WIDTH bits).
63 //      - z_0      : z component of the rotated vector
64 //                      (two's complement, WORD_WIDTH+1 bits).
65 //
66 // Parameters:
67 //      - WORD_WIDTH      : Word width in number of bits (positive integer,
68 // default: 16).
69 //      - CLOG2_WORD_WIDTH : Ceil of log2 of WORD_WIDTH (positive integer,
70 // default: 4).
71 //
72 // History:
73 // -----
74 //      - May 06, 2015 - Federico Damian Camarda - Original version.
75 //
76 //

77 //
78 // ****
79 // Interface
80 // ****
81 module rotator #(
82     // Parameters
83     // -----
84     parameter WORD_WIDTH      = 16
85     ) (
86     // Clock, reset & enable inputs
87     // -----
88     input  wire                 clk ,
89     input  wire                 arst ,
90     input  wire                 srst ,
91     input  wire                 enable ,
92     input  wire                 load ,
93     input  wire                 vec_rot_n ,
94     // Data inputs
95     // -----
96     input  wire [WORD_WIDTH-1:0] x_i ,
97     input  wire [WORD_WIDTH-1:0] y_i ,
98     input  wire [WORD_WIDTH:0]   z_i ,
99     // Data outputs
100    // -----
101    output wire [WORD_WIDTH-1:0] x_o ,
102    output wire [WORD_WIDTH-1:0] y_o ,
103
104
105
106

```

```

107     output wire [WORD_WIDTH:0]      z_o ,
108     output wire                   ready
109   );
110   // ****
111   //
112   // ****
113 // Architecture
114 // ****
115
116 // _____
117 // Internal signals
118 // _____
119 wire [WORD_WIDTH:0] x_0_int;
120 wire [WORD_WIDTH:0] y_0_int;
121 wire [WORD_WIDTH:0] z_0_int;
122 wire [WORD_WIDTH:0] x_f_int;
123 wire [WORD_WIDTH:0] y_f_int;
124 wire [WORD_WIDTH:0] z_f_int;
125 reg [WORD_WIDTH:0] z_final;
126 // _____
127
128 // _____
129 // Instantiate Circuits:
130 // _____
131
132 preprocessor #(
133   // _____
134   // Parameters
135   // _____
136   .WORD_WIDTH    (WORD_WIDTH)
137 ) preprocessor (
138   // _____
139   // Clock , reset & enable inputs
140   // _____
141
142   // _____
143   // Data inputs
144   // _____
145   .vec_rot_n    (vec_rot_n),
146   .x_in         (x_i),
147   .y_in         (y_i),
148   .z_in         (z_i),
149   // _____
150   // Data outputs
151   // _____
152   .x_out        (x_0_int),
153   .y_out        (y_0_int),
154   .z_out        (z_0_int)
155 );
156
157 iterative_cordic #(
158   // _____
159   // Parameters
160   // _____
161   .WORD_WIDTH    (WORD_WIDTH)
162 ) iterative_cordic (

```

```

163 //////////////////////////////////////////////////////////////////
164 // Clock, reset & enable inputs
165 //////////////////////////////////////////////////////////////////
166 .clk          (clk),
167 .arst         (arst),
168 .srst         (srst),
169 .enable       (enable),
170 //////////////////////////////////////////////////////////////////
171 // Data inputs
172 //////////////////////////////////////////////////////////////////
173 .load         (load),
174 .vec_rot_n   (vec_rot_n),
175 .x_0          (x_0_int),
176 .y_0          (y_0_int),
177 .z_0          (z_0_int),
178 //////////////////////////////////////////////////////////////////
179 // Data outputs
180 //////////////////////////////////////////////////////////////////
181 .x_i          (x_f_int),
182 .y_i          (y_f_int),
183 .z_i          (z_f_int),
184 .ready        (ready)
185 );
186
187 post_multiplier #(
188   .WORD_WIDTH  (WORD.WIDTH)
189 ) real_scaler (
190   //////////////////////////////////////////////////////////////////
191   // Data inputs
192   //////////////////////////////////////////////////////////////////
193   .x           (x_f_int),
194   //////////////////////////////////////////////////////////////////
195   // Data outputs
196   //////////////////////////////////////////////////////////////////
197   .y           (x_o)
198 );
199
200 post_multiplier #(
201   .WORD_WIDTH  (WORD.WIDTH)
202 ) imag_scaler (
203   //////////////////////////////////////////////////////////////////
204   // Data inputs
205   //////////////////////////////////////////////////////////////////
206   .x           (y_f_int),
207   //////////////////////////////////////////////////////////////////
208   // Data outputs
209   //////////////////////////////////////////////////////////////////
210   .y           (y_o)
211 );
212
213 // If the processing is in the vectoring mode and the sign of x_i is
214 // negative, it indicates that an additional PI rotation
215 // must be made to the output angle (e.g vector -1000;1000).
216 always @(*) begin
217   z_final <= z_f_int;
218   if (vec_rot_n) begin
219     if (x_i[WORD.WIDTH-1]) begin
220       z_final <= z_f_int + {{1'b1},{WORD.WIDTH{1'b0}}};
221     end
222   end
223 end
224 assign z_o = z_final;

```

```

225 // ****
226 // ****
227 endmodule

```

A.5. Fuente iterative_cordic.v

```

1 // _____
2 // Copyright (c) 2013 SDR and Wireless Group FIUBA.
3 // Confidential property of SDR and Wireless Group FIUBA.
4 //
5 // The use of the software, documentation, methodologies and other
6 // information contained herein is governed solely by the associated
7 // license agreements. Any inconsistent use shall be deemed to be a
8 // misappropriation of the intellectual property of SDR and Wireless Group
9 // FIUBA and treated accordingly.
10 //
11 //

12 // Description:
13 // _____
14 //
15 // - Iterative CORDIC algorithm. The angle representation is determined by
16 // the
17 // a_tan ROM. The first value stands for the atan of 1, which represents 45
18 // degrees.
19 // - For modifying the maximum WORD_WIDTH the
20 // ./fiubs_sdr/scripts/shell/atan_matrix_calculator shall be run and the
21 // result shall replaced the atan_matrix assignment in this file.
22 //

23 // File name:
24 // _____
25 //
26 // iterative_cordic.v
27 //
28 //

29 // Interface:
30 // _____
31 //
32 //
33 // Clock, reset & enable inputs:
34 //   - clk      : Posedge active clock input (logic, 1 bit).
35 //   - arst     : High active asynchronous reset (logic, 1 bit).
36 //   - srst     : High active synchronous reset (logic, 1 bit).
37 //   - enable   : Synchronous enable (logic, 1 bit).
38 //
39 // Data inputs:

```

```

40 //      - load      : When '1' the algorithm starts .
41 //      - vec_rot_n   : '1' vectoring mode, when '0' rotation mode (logic ,
42 //      1 bit).
43 //      - x_0        : X-axis input (Two's complement, WORD_WIDTH bits).
44 //      - y_0        : Y-axis input (Two's complement, WORD_WIDTH bits).
45 //      - z_0        : Input angle (Two's complement, WORD_WIDTH bits).
46 //
47 //      Data outputs:
48 //      - x_i        : X-axis iteration step (Two's complement, WORD_WIDTH
49 //      bits).
50 //      - y_i        : Y-axis iteration step (Two's complement, WORD_WIDTH
51 //      bits).
52 //      - z_i        : Output angle (Two's complement, WORD_WIDTH bits).
53 //      - ready      : When '1' the algorithm finishes (logic, 1 bit).
54 //
55 //
56 //

57 // History:
58 // -----
59 //
60 //      - May 06, 2015 - Federico Damian Camarda - Original version .
61 //
62 //

63 //
64 // ****
65 // Interface
66 // ****
67 module iterative_cordic #(
68     // Parameters
69     // -----
70     parameter WORD_WIDTH = 16
71 ) (
72     // Clock , reset & enable inputs
73     // -----
74     input  wire          clk ,
75     input  wire          arst ,
76     input  wire          srst ,
77     input  wire          enable ,
78     // Data inputs
79     // -----
80     input  wire          load ,
81     input  wire          vec_rot_n ,
82     input  wire [WORD_WIDTH:0] x_0 ,
83     input  wire [WORD_WIDTH:0] y_0 ,
84     input  wire [WORD_WIDTH:0] z_0 ,
85
86
87
88

```

```

89   // Data outputs
90   // _____
91   output reg [WORD_WIDTH:0] x_i ,
92   output reg [WORD_WIDTH:0] y_i ,
93   output reg [WORD_WIDTH:0] z_i ,
94   output wire ready
95 );
96 //
97 // ****
98 // ****
99 // Architecture
100 // ****
101 //include "sdr_simlib_math.vh"
102
103 // _____
104 // Constants
105 // _____
106 localparam CLOG2_WORD_WIDTH = clog2(WORD_WIDTH);
107 // _____
108
109 // _____
110 // Internal signals
111 // _____
112
113 wire signed [WORD_WIDTH:0] x_mux;
114 wire signed [WORD_WIDTH:0] y_mux;
115 wire signed [WORD_WIDTH:0] z_mux;
116 reg signed [WORD_WIDTH:0] x_sri;
117 reg signed [WORD_WIDTH:0] y_sri;
118 wire signed [WORD_WIDTH:0] atan_i;
119 wire signed [WORD_WIDTH:0] x_ip1;
120 wire signed [WORD_WIDTH:0] y_ip1;
121 wire signed [WORD_WIDTH:0] z_ip1;
122 wire d_i;
123 wire signed [WORD_WIDTH:0] atan_matrix [0:64];
124 reg [CLOG2_WORD_WIDTH-1:0] counter;
125 reg state;
126 // _____
127
128 // _____
129 // FSM states
130 // _____
131 localparam IDLE_STATE = 1'b0;
132 localparam ROTATING_STATE = 1'b1;
133 // _____
134
135 // _____
136 // This implementation supports 32 bits angles as maximum
137 // _____
138 assign atan_matrix[00] = 64'd04611686018427387904>>(64-WORD_WIDTH);
139 assign atan_matrix[01] = 64'd02722437224269746380>>(64-WORD_WIDTH);
140 assign atan_matrix[02] = 64'd01438461061161762076>>(64-WORD_WIDTH);

```

```

141 assign atan_matrix[03] = 64'd00730185295051043895 >>(64-WORD_WIDTH);
142 assign atan_matrix[04] = 64'd00366509582986589657 >>(64-WORD_WIDTH);
143 assign atan_matrix[05] = 64'd00183433460584072433 >>(64-WORD_WIDTH);
144 assign atan_matrix[06] = 64'd00091739112965426259 >>(64-WORD_WIDTH);
145 assign atan_matrix[07] = 64'd00045872355853501789 >>(64-WORD_WIDTH);
146 assign atan_matrix[08] = 64'd00022936527896151662 >>(64-WORD_WIDTH);
147 assign atan_matrix[09] = 64'd00011468307695752815 >>(64-WORD_WIDTH);
148 assign atan_matrix[10] = 64'd00005734159316382967 >>(64-WORD_WIDTH);
149 assign atan_matrix[11] = 64'd00002867080341756270 >>(64-WORD_WIDTH);
150 assign atan_matrix[12] = 64'd00001433540256323779 >>(64-WORD_WIDTH);
151 assign atan_matrix[13] = 64'd00000716770138842597 >>(64-WORD_WIDTH);
152 assign atan_matrix[14] = 64'd00000358385070756387 >>(64-WORD_WIDTH);
153 assign atan_matrix[15] = 64'd00000179192535545079 >>(64-WORD_WIDTH);
154 assign atan_matrix[16] = 64'd00000089596267793400 >>(64-WORD_WIDTH);
155 assign atan_matrix[17] = 64'd0000044798133899308 >>(64-WORD_WIDTH);
156 assign atan_matrix[18] = 64'd00000022399066949980 >>(64-WORD_WIDTH);
157 assign atan_matrix[19] = 64'd0000001199533475031 >>(64-WORD_WIDTH);
158 assign atan_matrix[20] = 64'd00000005599766737520 >>(64-WORD_WIDTH);
159 assign atan_matrix[21] = 64'd00000002799883368761 >>(64-WORD_WIDTH);
160 assign atan_matrix[22] = 64'd00000001399941684380 >>(64-WORD_WIDTH);
161 assign atan_matrix[23] = 64'd00000000699970842190 >>(64-WORD_WIDTH);
162 assign atan_matrix[24] = 64'd00000000349985421095 >>(64-WORD_WIDTH);
163 assign atan_matrix[25] = 64'd00000000174992710547 >>(64-WORD_WIDTH);
164 assign atan_matrix[26] = 64'd0000000087496355274 >>(64-WORD_WIDTH);
165 assign atan_matrix[27] = 64'd00000000043748177637 >>(64-WORD_WIDTH);
166 assign atan_matrix[28] = 64'd00000000021874088818 >>(64-WORD_WIDTH);
167 assign atan_matrix[29] = 64'd00000000010937044409 >>(64-WORD_WIDTH);
168 assign atan_matrix[30] = 64'd0000000005468522204 >>(64-WORD_WIDTH);
169 assign atan_matrix[31] = 64'd0000000002734261102 >>(64-WORD_WIDTH);
170 assign atan_matrix[32] = 64'd00000000001367130551 >>(64-WORD_WIDTH);
171 assign atan_matrix[33] = 64'd00000000000683565275 >>(64-WORD_WIDTH);
172 assign atan_matrix[34] = 64'd00000000000341782638 >>(64-WORD_WIDTH);
173 assign atan_matrix[35] = 64'd00000000000170891319 >>(64-WORD_WIDTH);
174 assign atan_matrix[36] = 64'd0000000000085445659 >>(64-WORD_WIDTH);
175 assign atan_matrix[37] = 64'd0000000000042722830 >>(64-WORD_WIDTH);
176 assign atan_matrix[38] = 64'd0000000000021361415 >>(64-WORD_WIDTH);
177 assign atan_matrix[39] = 64'd0000000000010680707 >>(64-WORD_WIDTH);
178 assign atan_matrix[40] = 64'd0000000000005340354 >>(64-WORD_WIDTH);
179 assign atan_matrix[41] = 64'd0000000000002670177 >>(64-WORD_WIDTH);
180 assign atan_matrix[42] = 64'd0000000000001335088 >>(64-WORD_WIDTH);
181 assign atan_matrix[43] = 64'd000000000000667544 >>(64-WORD_WIDTH);
182 assign atan_matrix[44] = 64'd000000000000333772 >>(64-WORD_WIDTH);
183 assign atan_matrix[45] = 64'd000000000000166886 >>(64-WORD_WIDTH);
184 assign atan_matrix[46] = 64'd000000000000083443 >>(64-WORD_WIDTH);
185 assign atan_matrix[47] = 64'd00000000000041721 >>(64-WORD_WIDTH);
186 assign atan_matrix[48] = 64'd00000000000020861 >>(64-WORD_WIDTH);
187 assign atan_matrix[49] = 64'd00000000000010430 >>(64-WORD_WIDTH);
188 assign atan_matrix[50] = 64'd00000000000005215 >>(64-WORD_WIDTH);
189 assign atan_matrix[51] = 64'd00000000000002607 >>(64-WORD_WIDTH);
190 assign atan_matrix[52] = 64'd0000000000001304 >>(64-WORD_WIDTH);
191 assign atan_matrix[53] = 64'd0000000000000000652 >>(64-WORD_WIDTH);
192 assign atan_matrix[54] = 64'd0000000000000000326 >>(64-WORD_WIDTH);
193 assign atan_matrix[55] = 64'd00000000000000163 >>(64-WORD_WIDTH);
194 assign atan_matrix[56] = 64'd000000000000000081 >>(64-WORD_WIDTH);
195 assign atan_matrix[57] = 64'd000000000000000041 >>(64-WORD_WIDTH);
196 assign atan_matrix[58] = 64'd000000000000000020 >>(64-WORD_WIDTH);
197 assign atan_matrix[59] = 64'd000000000000000010 >>(64-WORD_WIDTH);
198 assign atan_matrix[60] = 64'd000000000000000005 >>(64-WORD_WIDTH);
199 assign atan_matrix[61] = 64'd000000000000000002 >>(64-WORD_WIDTH);
200 assign atan_matrix[62] = 64'd000000000000000001 >>(64-WORD_WIDTH);
201 assign atan_matrix[63] = 64'd000000000000000001 >>(64-WORD_WIDTH);
202 assign atan_matrix[64] = 64'd000000000000000000 >>(64-WORD_WIDTH);

```

```

203 // _____
204
205 // Muxes
206 assign x_mux = (state == IDLE_STATE)? x_0 : x_i;
207 assign y_mux = (state == IDLE_STATE)? y_0 : y_i;
208 assign z_mux = (state == IDLE_STATE)? z_0 : z_i;
209
210 assign d_i = vec_rot_n ? ~y_mux[WORD_WIDTH] : z_mux[WORD_WIDTH];
211
212 // Right shifters
213 always @(*) begin
214   x_sri = x_mux >>> counter;
215   y_sri = y_mux >>> counter;
216 end
217
218 // Adders / subtractors
219 assign x_ip1 = d_i ? x_mux+y_sri : x_mux-y_sri;
220 assign y_ip1 = d_i ? y_mux-x_sri : y_mux+x_sri;
221 assign z_ip1 = d_i ? z_mux+atan_i : z_mux-atan_i;
222
223 // Counter
224 always @(posedge clk or posedge arst) begin
225   if (arst) begin
226     counter <= {CLOG2_WORD_WIDTH{1'b0}};
227     state <= IDLE_STATE;
228   end
229   else if (srst) begin
230     counter <= {CLOG2_WORD_WIDTH{1'b0}};
231     state <= IDLE_STATE;
232   end
233   else if (enable) begin
234     if (state == ROTATING_STATE) begin
235       counter <= counter + {{(CLOG2_WORD_WIDTH-1){1'b0}},1'b1};
236       if (& counter) begin
237         state <= IDLE_STATE;
238       end
239       else begin
240         state <= ROTATING_STATE;
241       end
242     end
243     else begin
244       if (load) begin
245         counter <= counter + {{(CLOG2_WORD_WIDTH-1){1'b0}},1'b1};
246         state <= ROTATING_STATE;
247       end
248       else begin
249         counter <= {CLOG2_WORD_WIDTH{1'b0}};
250         state <= IDLE_STATE;
251       end
252     end
253   end
254 end
255 assign ready = (state == IDLE_STATE)? 1'b1 : 1'b0;
256
257 // Atan multiplexer
258 assign atan_i = atan_matrix[counter][WORD_WIDTH:0];
259
260 // Output registers
261 always @(posedge clk or posedge arst) begin
262   if (arst) begin

```

```

263      x_i <= {WORD.WIDTH{1'b0}};
264      y_i <= {WORD.WIDTH{1'b0}};
265      z_i <= {WORD.WIDTH{1'b0}};
266  end
267  else if (srst) begin
268      x_i <= {WORD.WIDTH{1'b0}};
269      y_i <= {WORD.WIDTH{1'b0}};
270      z_i <= {WORD.WIDTH{1'b0}};
271  end
272  else if (enable && (load || (|counter))) begin
273      x_i <= x_ip1;
274      y_i <= y_ip1;
275      z_i <= z_ip1;
276  end
277 end
278
279 // ****
280 endmodule

```

A.6. Fuente `preprocessor.v`

```

1  //
2
3  // Copyright (c) 2013 SDR and Wireless Group FIUBA.
4  // Confidential property of SDR and Wireless Group FIUBA.
5  //
6  // The use of the software, documentation, methodologies and other
7  // information contained herein is governed solely by the associated
8  // license agreements. Any inconsistent use shall be deemed to be a
9  // misappropriation of the intellectual property of SDR and Wireless
10 // Group FIUBA and treated accordingly.
11 //
12
13 // Description :
14 //
15 // Angle preprocessor for CORDIC algorithm. It manages the cases when the
16 // angle is not between [-PI/2,+PI/2). It also extends one bit the inputs in order
17 // to avoid overflow on the iterative CORDIC processor.
18 //
19 //
20
21 // File name:
22 //
23 // preprocessor.v
24 //

```

```

25 // _____
26 // Interface:
27 // _____
28 //   Clock, reset & enable inputs:
29 //     - None (pure combinational block).
30 //   Data inputs:
31 //     - vec_rot_n : When '1' vectoring mode, when '0' rotation mode (logic
32 //       , 1 bit).
33 //     - x_in : Initial x component of input vector (two's complement,
34 //       WORD_WIDTH bits).
35 //     - y_in : Initial y component of input vector (two's complement,
36 //       WORD_WIDTH bits).
37 //     - z_in : Rotation angle (two's complement, WORD_WIDTH bits).
38 //   Data outputs:
39 //     - x_out : Fitted x component of input vector (two's complement,
40 //       WORD_WIDTH bits).
41 //     - y_out : Fitted y component of input vector (two's complement,
42 //       WORD_WIDTH bits).
43 //     - z_out : Fitted Rotation angle (two's complement, WORD_WIDTH
44 //       bits).
45 // Parameters:
46 //   - WORD_WIDTH : Word width in number of bits (positive integer,
47 //     default: 16).
48 // _____
49 // History:
50 // _____
51 //   - May 06, 2015 - Federico Damian Camarda - Original version.
52 // _____
53 // ****
54 // ****
55 // Interface
56 // ****
57 module preprocessor #(
58   // _____
59   // Parameters
60   // _____
61   parameter WORD_WIDTH = 16
62   ) (
63   // _____
64   // Clock, reset & enable inputs
65   // _____
66   // None.
67   // _____
68   // Data inputs
69   // _____

```

```

70   input  wire           vec_rot_n ,
71   input  wire [WORD_WIDTH-1:0] x_in ,
72   input  wire [WORD_WIDTH-1:0] y_in ,
73   input  wire [WORD_WIDTH :0] z_in ,
74   // -----
75   // Data outputs
76   // -----
77   output reg [WORD_WIDTH:0]   x_out ,
78   output reg [WORD_WIDTH:0]   y_out ,
79   output reg [WORD_WIDTH:0]   z_out
80 );
81 //
82 // ****
83 //
84 // Architecture
85 //
86   wire           d_i;
87   reg [WORD_WIDTH-1:0] x_aux;
88   reg [WORD_WIDTH-1:0] y_aux;
89
90   // Decision factor. When active, means that changes must be made to the
91   // inputs. For vectoring mode, the decision is made checking the sign of x
92
93   // For rotation mode, the decision is made checking the XOR between the 2
94   // MSB of the input angle, which defines if it is outside the domain
95   // [-pi/2; pi/2].
96   assign d_i = vec_rot_n ? x_in[WORD_WIDTH-1] : ^z_in[WORD_WIDTH:WORD_WIDTH
97   -1];
98
99   // Rotate coordinates to fit them between -90.0 and +89.xxx degs.
100  always @(*) begin
101    if (d_i) begin // x < 0.
102      // Input angle: 90.0 < z_in < 180.0
103      // Input angle: 180.0 < z_in < 270.0
104      x_aux = ~x_in + 1;
105      x_out = {x_aux[WORD_WIDTH-1],x_aux};
106      y_aux = ~y_in + 1;
107      y_out = {y_aux[WORD_WIDTH-1],y_aux};
108      if (~vec_rot_n) begin
109        z_out = z_in + {{1'b1},{WORD_WIDTH{1'b0}}};
110      end
111      else begin
112        z_out = z_in;
113      end
114    end
115    else begin
116      // Input angle: 0.00 < z_in < 90.0
117      // Input angle: 270.0 < z_in < 360.0
118      x_out = {x_in[WORD_WIDTH-1],x_in};
119      y_out = {y_in[WORD_WIDTH-1],y_in};
120      z_out = z_in;
121    end
122  end
123 //
124 // ****

```

```
123 |   endmodule
```

A.7. Fuente post_multiplier.v

```

1 // _____
2 // Copyright (c) 2013 SDR and Wireless Group FIUBA.
3 // Confidential property of SDR and Wireless Group FIUBA.
4 //
5 // The use of the software, documentation, methodologies and other
6 // information contained herein is governed solely by the associated
7 // license agreements. Any inconsistent use shall be deemed to be a
8 // misappropriation of the intellectual property of SDR and Wireless
9 // Group FIUBA and treated accordingly.
10 //
11 //

12 // Description:
13 // _____
14 //
15 // This is a fixed multiplier that will be attached to the CORDIC output to
16 // normalize the output vector.
17 //
18 //

19 // File name:
20 // _____
21 //
22 // post_multiplier.v
23 //
24 //

25 // Interface:
26 // _____
27 //
28 // Clock, reset & enable inputs:
29 // - None (pure combinational block).
30 //
31 // Data inputs:
32 // - x : Input from CORDIC (logic, 16 bits).
33 //
34 // Data outputs:
35 // - y : Normalized OUTPUT (logic, 16 bits).
36 //
37 // Parameters:
38 // - WORD_WIDTH : Word width in number of bits
39 //                  (positive integer, default: 16).
40 //
41 //

```

```

42 // History:
43 // -----
44 //
45 // - May 06, 2015 - Federico Damian Camarda - Original version.
46 //
47 //

48 //
49 // ****
50 // Interface
51 //
52 // ****
53 module post_multiplier #(
54     // Parameters
55     // -----
56     parameter WORD_WIDTH = 16
57     ) (
58     // Clock, reset & enable inputs
59     // -----
60     // None.
61     // -----
62     // Data inputs
63     // -----
64     input wire [WORD_WIDTH:0] x,
65     // -----
66     // Data outputs
67     // -----
68     output reg [WORD_WIDTH-1:0] y
69 );
70 //
71 // ****
72 //
73 // ****
74 // Architecture
75 //
76 // ****
77 // Internal signals
78 //
79 wire signed [WORD_WIDTH:0] correction_factor;
80 reg [WORD_WIDTH :0] aux1;
81 reg [2*WORD_WIDTH+2:0] aux2;
82 reg [WORD_WIDTH-1:0] aux3;
83 //
84 //
85 assign correction_factor = 64'd11202707675963800000 >> (64-WORD_WIDTH);
86
87 always @* begin
88     if (x[WORD_WIDTH]) begin
89         aux1 = ~x + 1;
90         aux2 = aux1 * correction_factor;
91

```

```
92      aux3 = aux2[2*WORD_WIDTH-1:WORD_WIDTH];
93      y[WORD_WIDTH-1:0] = ~aux3 + 1;
94  end
95  else begin
96    aux2 = x * correction_factor;
97    y[WORD_WIDTH-1:0] = aux2[2*WORD_WIDTH-1:WORD_WIDTH];
98  end
99 end
100 // ****
101
102 endmodule
```

Apéndice B

Script Octave de análisis de resultados

```
1 %  
2 % Name:          rls_analysis.m  
3 % Version:       1.1  
4 % Author:        Federico Damian Camarda  
5 % Date:         09/04/2015  
6 % Description:   Framework to work with the RLS algorithm.  
7 % References:    [1] S. Haykin, Adaptive Filter Theory. 3rd edition pp. 562.  
8 %                  RLS Algorithm.  
9 %  
10 %                  [2] S. Haykin, Adaptive Filter Theory. 3rd edition pp. 598.  
11 %                  QRD RLS Algorithm.  
12 %  
13 % Block Diagram:  
14 %  
15 %  
16 % u(n) --> |-----| --> d(n) --> |+|  
17 % |           W           |-----|  
18 % |-----|  
19 % |           W_est       |-----|  
20 % |-----| --> y(n) --> |-|  
21 % |           sum         |-----|  
22 % |-----|  
23 %  
24 % The u(n) signal is filtered with the W system to generate the desired  
25 % signal d(n). The W system is manually defined.  
26 %  
27 % The RLS algorithm is used to compute the W_est filter, an estimated system  
28 % that, when filtering u(n) with W_est it outputs a signal y(n) that  
29 % minimizes  
30 % the cost function J(n) = |e(n)|^2 = |d(n) - y(n)|^2
```

```

31 % Initialize environment.
32 clc % Clean terminal.
33 clear all % Clear environment.
34 close all % Close windows.
35 diary('rls_analysis.term') % File where terminal output will be saved on.
36 diary on % Save terminal output on.
37
38 % Control parameters.
39 enable_plots_rls = 0;
40 enable_plots_rls_qr = 0;
41 enable_plots_rls_qr_hard = 0;
42 enable_plots_precision = 1;
43 enable_debug = 0;
44 enable_hard_file = 1;
45
46 % Samples related signals.
47 N = 300; % Number of samples.
48 n = 1:N; % Samples' index.
49
50 % Filter related signals.
51 W = [2 -1 0 0 -1 2]'; % Unknown System coefficients.
52 N_w = length(W); % Filter weights length.
53 n_w = 1:N_w; % Filter weights index.
54 An = 5000; % Multiplying constant.
55 u = randn(1, N); % Input to the filter.
56 u = An * u / max(abs(u)); % Normalized input.
57
58 % Observed signal.
59 d = filter(W, 1, u);
60
61 %
62 %


---


63 % 1. Standard RLS Algorithm : Equations
64 %


---


65
66 % RLS Algorithm implemented by the equations described in [1].
67 % Zero padding of N_w values is applied to u(n). The reason is that the input
68 % of the algorithm requires that the first vector is [ 0 0 ... 0 u(n) ],
69 % the next [ 0 0 ... u(n) u(n-1) ], and so on until the vector is completed.
70
71 lambda = 0.9375; % RLS forgetting factor.
72 v = 1; % Initial input variance estimate.
73 P = eye([N_w, N_w]) / v; % Initial inverse correlation matrix.
74 W_est = zeros(N_w, 1); % Initial weights.
75 u = [zeros(1, N_w - 1), u]; % System input u(n) with zero padding.
76 y = [ ]; % Output of the estimated system.
77 e = [ ]; % A posteriori error.
78 xi = [ ]; % A priori error. (xi: from the greek letter
79 )
80
81 % Open file descriptor to save input vectors.
82 fid = fopen('input_vectors_matlab.txt','wt');
83
84 % Start algorithm.
85 for i=1:N % All time steps.
86     % Take the last N_w samples from the observation.
87     U = u(i + N_w - n_w)'; % e.g. for i=1, => 6,5,4,3,2,1;

```

```

87 %           for i=2, => 7,6,5,4,3,2;
88
89 % Save vector in a file.
90 fprintf(fid,'%f ',U);
91 fprintf(fid,'%f ',d(i));
92 fprintf(fid,'\n');
93
94 % Save the estimated output.
95 y(i) = W_est' * U;
96
97 % Filter gain.
98 K = (1 / lambda) * P * U / (1 + (1 / lambda) * U' * P * U);
99
100 % Error signal equation. A priori error.
101 xi(i) = d(i) - W_est' * U;
102
103 % Filter coefficients.
104 W_est = W_est + K * xi(i);
105
106 % Error signal equation. A posteriori error.
107 e(i) = d(i) - W_est' * U;
108
109 % Inverse correlation matrix update.
110 P = (1 / lambda) * P - (1 / lambda) * K * U' * P;
111 end
112
113 % Close file descriptor.
114 fclose(fid);
115
116 %

117 %           2. Compute the results with hardware
118 %

119
120 % Process hardware results if flag is enable.
121 if(enable_hard_file)
122     % Give time to the operator to process the samples.
123     disp('Samples are ready to be processed.');
124     pause
125
126     % Read the hardware results and store them in hard_matrix_16.
127     for i=n
128         hard_matrix_16 (:,:,i)=dlmread('hardware_result_matlab.dat',' ',
129                                     [(i-1)*7 0 (i-1)*7+6 6]);
130     end
131
132     % Read the software results and store them in soft_matrix_16.
133     for i=n
134         soft_matrix_16 (:,:,i)=dlmread('software_result_matlab.dat',' ',
135                                     [(i-1)*7 0 (i-1)*7+6 6]);
136     end
137
138     % Mean value of the matrix
139     for i=1:N-1
140         hard_matrix_mean_value(i) = sum(sum(hard_matrix_16 (:,:, i+1)))/49;
141         soft_matrix_mean_value(i) = sum(sum(soft_matrix_16 (:,:, i+1)))/49;
142         error_matrix_mean_value(i) = (sum(sum(soft_matrix_16 (:,:, i+1))) -
143                                         sum(sum(hard_matrix_16 (:,:, i+1))))/49;
144

```

```

144     error_matrix_mean_value_rel(i) = error_matrix_mean_value(i) /
145         2^(16-1);
146     error_matrix_mean_value_per(i) = error_matrix_mean_value_rel(i) *
147         100;
148 end
149 %
150 %
151 %                               3. QRD RLS Algorithm : Equations
152 %
153 %QRD-RLS Algorithm implemented by the matrix equations described in [2].
154 % Since the QR decomposition gives a lower triangular matrix, and Haykin
155 % describes an upper triangular matrix, everything is transposed.
156 %
157 % | lambda_sqrt_qr * Phi_sqrt_qr'(n-1)  lambda_sqrt_qr * P_qr(n-1)| * Q =
158 % |                           U_qr'                                d(n)          |
159 %
160 % |      Phi_sqrt_qr'(n)                      P_qr(n)           |
161 % |          0                            xi_qr * gamma_sqrt(n) |
162 %
163 lambda_sqrt_qr = sqrt(lambda);    % Square root of the forgetting factor.
164 Phi_sqrt_qr    = zeros(N_w, N_w); % Square root of the correlation matrix.
165 P_qr           = zeros(N_w, 1);   % Cross-correlation vector.
166 W_est_qr       = zeros(N_w, 1);   % Initial weights.
167 y_qr           = [ ];            % Output of the estimated system.
168 e_qr           = [ ];            %A posteriori error.
169 xi_qr          = [ ];            %A priori error. (xi: from the greek
170 letter)
171 R_max          = [ ];            %Maximum value of the R matrix.
172 hard_Phi_sqrt_qr = zeros(N_w, N_w); %Square root of the correlation
173             %matrix.
174 hard_P_qr       = zeros(N_w, 1);   %Cross-correlation vector.
175 hard_W_est_qr   = zeros(N_w, 1);   %Initial weights.
176 %
177 % Start algorithm.
178 for i=1:N-1                         % All time steps minus one casue of hard.
179     % Take the last N_w samples from the observation.
180     U_qr        = u(i + N_w - n_w)'; % for i=1, => 3,2,1;
181             % for i=2, => 4,3,2;
182
183     % Save the estimated output.
184     y_qr(i)     = W_est_qr' * U_qr;
185
186     % Create the input matrix to compute the QR decomposition. This matrix
187     % is the result of the concatenation of other matrices as described above
188     %
189     A_qr = [ [U_qr' d(i)] ; [lambda_sqrt_qr * Phi_sqrt_qr' lambda_sqrt_qr *
190                 P_qr] ];
191
192     % Calculate the QR decomposition.
193     [Q,R]        = qr(A_qr);
194
195     % Extract the new Phi_sqrt_qr matrix from the R matrix.
196     for column = n_w
197         Phi_sqrt_qr(column, :) = R(1:N_w, column)';
198     end

```

```

196
197 % Extract the new P_qr vector from the R matrix.
198 P_qr = R(1:N_w, N_w + 1);
199
200 % Error signal equation. A priori error.
201 xi_qr(i) = d(i) - W_est_qr' * U_qr;
202
203 % Filter coefficients.
204 Phi_sqrt_qr_inv = inv(Phi_sqrt_qr);
205 W_est_qr = (P_qr' * Phi_sqrt_qr_inv)';
206
207 % Error signal equation. A posteriori error.
208 e_qr(i) = d(i) - W_est_qr' * U_qr;
209
210 % Save the maximum value of the R matrix:
211 R_max(i) = max(max(abs(R)));
212
213 if(enable_hard_file)
214 % Extract the new Phi_sqrt_qr matrix from the R matrix.
215 for column = n_w
216     hard_Phi_sqrt_qr(column, :) = hard_matrix_16(1:N_w, column, i+1)'
217     ;
218 end
219
220 % Save the estimated output.
221 hard_y_qr(i) = hard_W_est_qr' * U_qr;
222
223 % Extract the new P_qr vector from the R matrix.
224 hard_P_qr = hard_matrix_16(1:N_w, N_w + 1, i+1);
225
226 % Error signal equation. A priori error.
227 hard_xi_qr(i) = d(i) - hard_W_est_qr' * U_qr;
228
229 % Filter coefficients.
230 hard_Phi_sqrt_qr_inv = inv(hard_Phi_sqrt_qr);
231 hard_W_est_qr = (hard_P_qr' * hard_Phi_sqrt_qr_inv)';
232
233 % Error signal equation. A posteriori error.
234 hard_e_qr(i) = d(i) - hard_W_est_qr' * U_qr;
235
236 % Save the maximum value of the R matrix:
237 hard_R_max(i) = max(max(abs(hard_matrix_16(:, :, i+1))));
238
239 end
240
241 % Debug.
242 if enable_debug
243     disp(['Iteration: ', num2str(i)]);
244     U_qr
245     A_qr
246     R
247     Phi_sqrt_qr
248     P_qr
249     W_est_qr
250 end
end

```


Apéndice C

El algoritmo CORDIC

A continuación se detalla el fundamento matemático del algoritmo.

Sea un vector $\mathbf{x} = (x, y)$ perteneciente al plano y una rotación del mismo $\mathbf{x}_\phi = (x_\phi, y_\phi)$ en un ángulo ϕ , los cuales se observan en la Fig. C.1.

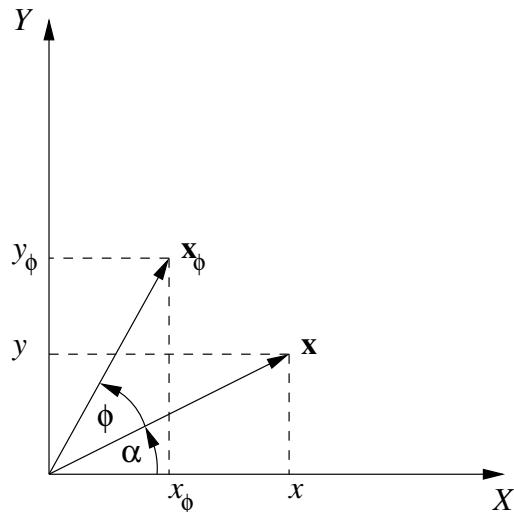


Figura C.1: Rotación de un vector en un ángulo ϕ .

Si \mathbf{x} posee norma unitaria, $\|\mathbf{x}\| = 1$, se cumple

$$\begin{aligned} x &= \cos(\alpha) \\ y &= \sin(\alpha) \end{aligned} \tag{C.1}$$

Dadas las siguientes identidades trigonométricas

$$\begin{aligned}\sin(\alpha + \beta) &= \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta) \\ \cos(\alpha + \beta) &= \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)\end{aligned}\tag{C.2}$$

resulta

$$\begin{aligned}x_\phi &= \cos(\alpha + \phi) = \cos(\alpha)\cos(\phi) - \sin(\alpha)\sin(\phi) = x\cos(\phi) - y\sin(\phi) \\ y_\phi &= \sin(\alpha + \phi) = \sin(\alpha)\cos(\phi) + \cos(\alpha)\sin(\phi) = y\cos(\phi) + x\sin(\phi)\end{aligned}\tag{C.3}$$

o bien en su forma matricial equivalente:

$$\begin{bmatrix} x_\phi \\ y_\phi \end{bmatrix} = \begin{bmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}\tag{C.4}$$

Tomando como factor común $\cos(\phi)$ en la ecuación C.3 se obtiene,

$$\begin{aligned}x_\phi &= \cos(\phi)[x - y\tan(\phi)] \\ y_\phi &= \cos(\phi)[y + x\tan(\phi)]\end{aligned}\tag{C.5}$$

donde $\cos(\phi) > 0$ si $-\pi/2 < \phi < \pi/2$.

Ahora bien, si se restringe el ángulo de rotación a un conjunto discreto de valores ϕ_i de forma tal que su tangente tome los valores

$$\tan(\phi_i) = 2^{-i}, \quad i = 0, 1, 2, \dots, n-1; \quad n \in \mathbb{N}\tag{C.6}$$

entonces

$$\begin{aligned}x_{i+1} &= k_i [x_i - y_i d_i 2^{-i}] \\ y_{i+1} &= k_i [y_i + x_i d_i 2^{-i}]\end{aligned}\tag{C.7}$$

donde

$$k_i = \cos(\phi_i) = \frac{1}{\sqrt{1 + 2^{-2i}}}\tag{C.8}$$

y d_i sólo puede adoptar los valores $+1$ ó -1 .

Demostración de la ecuación C.8:

Si $\phi \neq \pm\pi/2$ entonces,

$$\tan(\phi_i) = \frac{\sin(\phi_i)}{\cos(\phi_i)} \quad (\text{C.9})$$

Restringiendo ϕ_i de forma tal que sólo acepte aquellos valores para los cuales su tangente corresponde a potencias enteras negativas de 2, es decir $\tan(\phi_i) = 2^{-i}$ y elevando al cuadrado la ecuación (C.9) se obtiene

$$\tan^2(\phi_i) = 2^{-2i} = \frac{\sin^2(\phi_i)}{\cos^2(\phi_i)} \quad (\text{C.10})$$

Siendo $\cos^2(\phi_i) + \sin^2(\phi_i) = 1$, se cumple

$$2^{-2i} = \frac{1 - \cos^2(\phi_i)}{\cos^2(\phi_i)} = \frac{1}{\cos^2(\phi_i)} - 1 \quad (\text{C.11})$$

y por lo tanto

$$1 + 2^{-2i} = \frac{1}{\cos^2(\phi_i)} \quad (\text{C.12})$$

Por último,

$$k_i = \cos(\phi_i) = \frac{1}{\sqrt{1 + 2^{-2i}}} \quad (\text{C.13})$$

Si se realizan n rotaciones sucesivas sobre un vector $\mathbf{x}_0 = [x_0 \ y_0]^T \in \mathbf{R}^2$, se obtiene,

$$\begin{aligned} x_n &= x_0 \cos\left(\sum_{i=0}^{n-1} d_i \arctan(2^{-i})\right) - y_0 \sin\left(\sum_{i=0}^{n-1} d_i \arctan(2^{-i})\right) \\ y_n &= x_0 \sin\left(\sum_{i=0}^{n-1} d_i \arctan(2^{-i})\right) - y_0 \cos\left(\sum_{i=0}^{n-1} d_i \arctan(2^{-i})\right) \end{aligned} \quad (\text{C.14})$$

Por lo tanto, si se realiza una rotación en un ángulo ϕ , el ángulo verdadero de rotación obtenido luego de n iteraciones será

$$\phi_n = \sum_{i=0}^{n-1} d_i \arctan(2^{-i}) \quad (\text{C.15})$$

Las siguientes definiciones son ahora necesarias:

Definición C.1. Se define como error ϵ_x entre una magnitud x y una aproximación \hat{x} de la misma a la diferencia $\epsilon_x = x - \hat{x}$.

Definición C.2. Se define como error absoluto e_x entre una magnitud x y una aproximación \hat{x} de la misma al valor $e_x = |\epsilon_x| = |x - \hat{x}|$.

Definición C.3. Se define como incertidumbre Δx de una magnitud x a la inferior de las cotas superiores M tales que $|x - \hat{x}| \leq M$.

Con estas definiciones en mente, se puede escribir

$$e_\phi = |\phi - \phi_n| \leq \frac{1}{2} \arctan(2^{-n+1}) = \Delta\phi \quad (\text{C.16})$$

con lo cual probarse que

$$\lim_{n \rightarrow \infty} e_\phi = 0 \quad (\text{C.17})$$

En la Fig. C.2 se observa $\Delta\phi$ en función de la cantidad de iteraciones n .

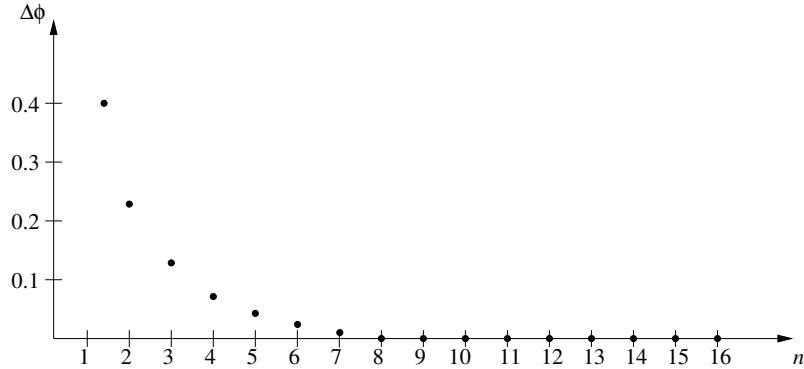


Figura C.2: $\Delta\phi$ en función de la cantidad de iteraciones n .

Por otra parte, definiendo

$$\begin{aligned} x'_{i+1} &= x'_i - y'_i d_i 2^{-i} \\ y'_{i+1} &= y'_i + x'_i d_i 2^{-i} \end{aligned} \quad (\text{C.18})$$

donde $i = 0, 1, \dots, n - 1$, se observa a partir de la ecuación C.7 que

$$\begin{aligned}
x_{i+1} &= k_i (x_i - y_i d_i 2^{-i}) \\
&= k_i [k_{i-1} (x_{i-1} - y_{i-1} d_{i-1} 2^{-i+1}) - k_{i-1} (y_{i-1} + x_{i-1} d_{i-1} 2^{-i+1}) d_i 2^{-i}] \\
&= k_i k_{i-1} (x'_i - y'_i d'_i 2^{-i}) \\
y_{i+1} &= k_i (y_i + x_i d_i 2^{-i}) \\
&= k_i [k_{i-1} (y_{i-1} + x_{i-1} d_{i-1} 2^{-i+1}) + k_{i-1} (x_{i-1} - y_{i-1} d_{i-1} 2^{-i+1}) d_i 2^{-i}] \\
&= k_i k_{i-1} (y'_i + x'_i d'_i 2^{-i})
\end{aligned} \tag{C.19}$$

y por lo tanto

$$\begin{aligned}
x_n &= \left(\prod_{i=0}^{n-1} k_i \right) x'_n \\
y_n &= \left(\prod_{i=0}^{n-1} k_i \right) y'_n
\end{aligned} \tag{C.20}$$

se concluye que resulta conveniente realizar la iteración mediante la ecuación (C.18), obteniendo los valores de x'_n y y'_n . Luego los valores de x_n y y_n pueden hallarse mediante el producto por k_n ,

$$\begin{aligned}
x_n &= k_n x'_n \\
y_n &= k_n y'_n
\end{aligned} \tag{C.21}$$

donde k_n corresponde a la constante dada por:

$$k_n = \prod_{i=0}^{n-1} k_i = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1 + 2^{-2i}}} \tag{C.22}$$

La ecuación (C.18) define el algoritmo CORDIC y provee un método para hallar las componentes x'_n , y'_n sin el uso de productos matriciales, sólo mediante operaciones de suma, resta y desplazamiento como se detallará más adelante. Los dos valores x'_n , y'_n corresponden a las componentes del vector \mathbf{x} rotado un ángulo ϕ_n dado por la ecuación (C.15) y escalado por un valor $A_n = 1/k_n$ conocido como *ganancia de procesamiento*.

En la figura C.3 se observa el efecto de alargamiento del vector en cada iteración del

algoritmo, por lo que cada una de estas es a menudo llamada *pseudorotación*, ya que no conserva la norma.

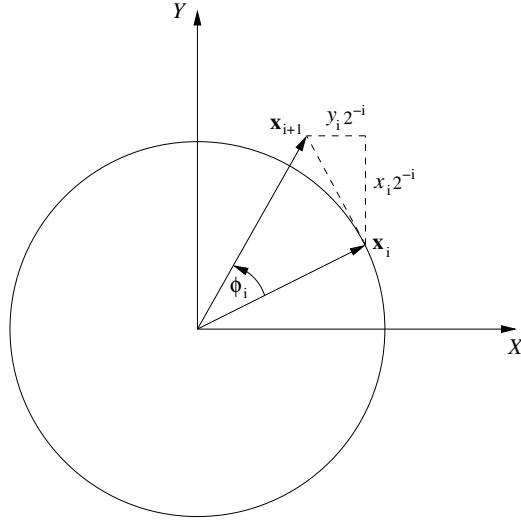


Figura C.3: Pseudorotación producida por la i -ésima iteración.

Por último, si para una rotación dada no se conoce la secuencia de valores $\{d_i\}_{i=0,1,\dots,n-1}$, entonces se utiliza un acumulador de fase z_i donde deben conocerse todos los valores $\arctan(2^{-i})$ para $i = 0, 1, \dots, n - 1$;

$$\begin{aligned} z_{i+1} &= z_i - d_i \arctan(2^{-i}) \\ d_i &= \begin{cases} -1 & \text{si } z_i < 0, \\ +1 & \text{si } z_i \geq 0. \end{cases} \end{aligned} \tag{C.23}$$

C.1. Convergencia del algoritmo CORDIC

Para asegurar la convergencia del algoritmo luego de infinitas iteraciones debe asegurarse que el vector resultante converja en fase, es decir el ángulo ϕ_n se aproxima tanto como se desee al ángulo ϕ , y además la que converja en norma, lo que implica que el factor de escala A_n debe ser finito cuando n tiende a infinito.

La ecuación (C.16) muestra la convergencia del algoritmo respecto del ángulo de rotación debido a la monotonicidad creciente de la función arcotangente y la monotonicidad decreciente de su argumento 2^{-N+1} .

C.1.1. Convergencia en norma

Luego de n iteraciones, ambas salidas del algoritmo CORDIC se ven escaladas por un factor

$$A_n = \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}} = \sqrt{\prod_{i=0}^{n-1} (1 + 2^{-2i})} \quad (\text{C.24})$$

Para probar la convergencia de A_n , alcanza encontrar una cota superior y una cota inferior. Una cota inferior es simple de encontrar, ya que A_n corresponde a la productoria de factores todos ellos positivos independientemente del valor de i , por lo tanto $A_n \geq 0$, $\forall n \in \mathbf{N}$.

Para encontrar una cota superior se analizan los términos parciales de la sucesión $\left\{ \prod_{i=0}^{n-1} (1 + 2^{-2i}) \right\}_{n \in \mathbf{N}}$.

1. $n = 0$

$$\prod_{i=0}^{n-1} 1 + 2^{-2i} = 1 + 2^0 = 2$$

2. $n = 1$

$$\prod_{i=0}^{n-1} 1 + 2^{-2i} = 2 (1 + 2^{-2}) = 2 + 2^{-1}$$

3. $n = 2$

$$\prod_{i=0}^{n-1} 1 + 2^{-2i} = (2 + 2^{-1}) (1 + 2^{-4}) = 2 + 2^{-1} + 2^{-3} + 2^{-5}$$

4. $n = 3$

$$\begin{aligned} \prod_{i=0}^{n-1} 1 + 2^{-2i} &= (2 + 2^{-1} + 2^{-3} + 2^{-5}) (1 + 2^{-6}) \\ &= 2 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-7} + 2^{-9} + 2^{-11} \end{aligned}$$

5. $n = 4$

$$\begin{aligned} \prod_{i=0}^{n-1} 1 + 2^{-2i} &= (2 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-7} + 2^{-9} + 2^{-11}) (1 + 2^{-8}) \\ &= 2 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-6} + 2^{-8} + 2^{-10} + 2^{-12} + 2^{-15} \\ &\quad + 2^{-17} + 2^{-19} \end{aligned}$$

6. $n = 5$

$$\begin{aligned}
\prod_{i=0}^{n-1} 1 + 2^{-2i} &= (2 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-6} + 2^{-8} + 2^{-10} + 2^{-12} + 2^{-15} \\
&\quad + 2^{-17} + 2^{-19}) (1 + 2^{-10}) \\
&= 2 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-6} + 2^{-8} + 2^{-9} + 2^{-10} + 2^{-11} \\
&\quad + 2^{-12} + 2^{-14} + 2^{-15} + 2^{-16} + 2^{-17} + 2^{-18} + 2^{-19} \\
&\quad + 2^{-20} + 2^{-22} + 2^{-25} + 2^{-27} + 2^{-29}
\end{aligned}$$

Se observa que el término de mayor orden corresponde a $2^{-(n^2+n-1)}$. Si bien no están presentes todos los términos hasta $2^{-(n^2+n-1)}$, se puede ver que

$$\prod_{i=0}^{n-1} 1 + 2^{-2i} < 1 + \sum_{i=0}^k 2^{-i} < 1 + \sum_{i=0}^{\infty} 2^{-i} = 3, \quad k = n^2 + n - 1 \quad (\text{C.25})$$

Por lo tanto,

$$A_n = \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}} < \sqrt{3} \cong 1,73205 \quad (\text{C.26})$$

De esta forma se concluye que el algoritmo CORDIC converge luego de infinitas iteraciones alcanzando un error nulo en el ángulo de rotación y un factor de escala $A_n \cong 1,64676$.

Si se desean los valores de x_n y y_n sin el factor de escala, se puede multiplicar la salida del algoritmo, x'_n , y'_n por $k_n = 1/A_n$, el cual adquiere un valor de aproximadamente 0,60725 logrando así que el algoritmo conserve la norma original del vector de entrada \mathbf{x}_0 .

Estas observaciones son formalizadas a continuación. A partir de (C.22) se define

$$k_n^2 = \prod_{i=0}^{n-1} \frac{1}{1 + 2^{-2i}} \quad (\text{C.27})$$

Proposición C.1. *La sucesión $\{k_n^2\}_{n \in \mathbb{N}}$ es una sucesión decreciente de números reales positivos.*

Demostración:

Para demostrar que cada término de la sucesión es positivo basta observar que cada factor de la productoria definida en (C.27) es positivo y para demostrar que es decreciente basta observar a su vez que cada factor es menor a 1.

Estas dos observaciones implica que la sucesión $\{k_n^2\}_{n \in \mathbf{N}}$ está acotada inferiormente con lo cual es convergente y entonces se puede enunciar el siguiente teorema.

Teorema C.2. *La sucesión $\{k_n^2\}_{n \in \mathbf{N}}$ converge en \mathbf{R} . Aún más, converge a un número mayor a cero.*

Demostración:

Al ser una sucesión monótonamente decreciente y acotada inferiormente por cero necesariamente converge por la completitud de \mathbf{R} .

La convergencia a un número mayor a cero se hará por el absurdo. Supóngase que efectivamente $\{k_n^2\}_{n \in \mathbf{N}}$ converge a 0. Entonces por definición de convergencia, para todo $\varepsilon > 0$ existe $N \in \mathbf{N}$ tal que si $n \geq N$ se cumple $|k_n^2 - 0| = k_n^2 < \varepsilon$. Por lo tanto,

$$\prod_{i=0}^{n-1} \frac{1}{1 + 2^{-2i}} < \varepsilon \quad (\text{C.28})$$

Lo cual se cumple si y sólo si

$$1 < \varepsilon \prod_{i=0}^{n-1} 1 + 2^{-2i} \quad \forall \varepsilon > 0 \quad (\text{C.29})$$

Sea $0 < \varepsilon = 1/(2 \prod_{i=0}^{n-1} 1 + 2^{-2i})$ entonces resulta de la ecuación C.29 que $1 < 1/2$ lo cual es absurdo y parte de suponer que $\{k_n^2\}_{n \in \mathbf{N}}$ converge a 0.

De esta forma se concluye que si $\{k_n^2\}_{n \in \mathbf{N}}$ converge entonces también lo hace $\{k_n\}_{n \in \mathbf{N}}$ y en consecuencia $\{A_n\}_{n \in \mathbf{N}}$. Por lo que queda demostrada la convergencia en norma del algoritmo.

C.1.2. Convergencia en fase

La convergencia en fase se demuestra haciendo uso del llamado *Teorema de Convergencia*:

Teorema C.3. *Sea $\{\sigma_0, \sigma_1, \dots, \sigma_n\}$ una secuencia finita decreciente de $n+1$ números positivos, es decir $\sigma_i \geq \sigma_j$ para $i \geq j$, la cual cumple que*

$$\sigma_k \leq \sigma_n + \sum_{j=k+1}^n \sigma_j, \quad 0 \leq k \leq n \quad (\text{C.30})$$

Además, sea r un número positivo que satisface

$$|r| \leq \sum_{j=0}^n \sigma_j \quad (\text{C.31})$$

Sea también la sucesión $s_0 = 0$, $s_{k+1} = s_k + \delta_k \sigma_k$, $k = 0, 1, \dots, n$, donde

$$\delta_k = \operatorname{sgn}(r - s_k) = \begin{cases} 1, & r \geq s_k \\ -1, & r \leq s_k \end{cases} \quad (\text{C.32})$$

Entonces,

$$|r - s_k| \leq \sigma_n + \sum_{j=k}^n \sigma_j, \quad 0 \leq k \leq n \quad (\text{C.33})$$

En particular,

$$|r - s_{n+1}| \leq \sigma_n \quad (\text{C.34})$$

Demostración:

La demostración se realizará por inducción sobre k . Para $k = 0$, se tiene

$$|r - s_0| = |r| \leq \sigma_n + \sum_{j=0}^n \sigma_j$$

Asumiendo el teorema válido para k , se tiene la siguiente hipótesis inductiva (H1)

$$|r - s_k| \leq \sigma_n + \sum_{j=k+1}^n \sigma_j$$

Considerando el caso $k + 1$, se debe demostrar que la siguiente tesis inductiva (T1) es válida

$$|r - s_{k+1}| \leq \sigma_n + \sum_{j=k+1}^n \sigma_j$$

Considérese $|r - s_{k+1}| = |r - s_k - \delta_k \sigma_k|$. Si $r - s_k \geq 0$, entonces $\delta_k = 1$ y $|r - s_k - \delta_k \sigma_k| = ||r - s_k| - \sigma_k||$. Por el contrario, si $r - s_k < 0$, entonces $\delta_k = -1$ y $|r - s_k - \delta_k \sigma_k| = |r - s_k + \sigma_k| = ||r - s_k| - \sigma_k||$. Por lo tanto, en ambos casos

$$|r - s_{k+1}| = ||r - s_k| - \sigma_k| \leq \sigma_n + \sum_{j=k+1}^n \sigma_j$$

Lo que muestra que la tesis inductiva T1 es válida. Finalmente, $-\sigma_n \leq \|r - s_n\| - \sigma_n \leq 2\sigma_n - \sigma_n = \sigma_n$ y entonces $|r - s_{n+1}| = ||r - s_n| - \sigma_n| \leq \sigma_n$, lo cual completa la demostración.

Teorema C.4. *Para $n > 3$, la sucesión $\sigma_k = \arctan(2^{-k})$, $k = 0, 1, \dots, n$ satisface las hipótesis del teorema de convergencia para todo $|r| \leq \pi/2$.*

Demostración:

La secuencia $\arctan(2^0), \arctan(2^{-1}), \dots, \arctan(2^{-n}) = \arctan(1), \dots, \arctan(1/2^n)$ es claramente una secuencia decreciente de valores positivos. El Teorema del Valor Medio establece que existe un número c entre a y b , $a < c < b$, tal que

$$\frac{\arctan(b) - \arctan(a)}{b - a} = \frac{1}{1 + c^2} \quad (\text{C.35})$$

Sean $a = 2^{-(j+1)}$, $b = 2^{-j}$ en (C.35). Entonces, $b - a = 2^{-(j+1)}$ y además

$$\frac{1}{1 + c^2} < \frac{1}{1 + a^2} = \frac{1}{1 + 2^{-2(j+1)}} = \frac{2^{2j+2}}{1 + 2^{2j+2}} \quad (\text{C.36})$$

Entonces,

$$\sigma_j - \sigma_{j+1} < (b - a) \frac{1}{1 + c^2} \leq \frac{1}{2^{j+1}} \frac{2^{2j+2}}{1 + 2^{2j+2}} = \frac{2^{j+1}}{1 + 2^{2j+1}} \quad (\text{C.37})$$

Sean ahora $a = 0$, $b = 2^{-j}$ en (C.35). Entonces,

$$\frac{1}{1 + c^2} > \frac{1}{1 + b^2} = \frac{1}{1 + 2^{-2j}} = \frac{2^{2j}}{1 + 2^{2j}} \quad (\text{C.38})$$

y

$$\sigma_j = b \frac{1}{1 + c^2} \geq \frac{1}{2^j} \frac{2^{2j}}{1 + 2^{2j}} = \frac{2^j}{1 + 2^{2j}} \quad (\text{C.39})$$

Por otra parte,

$$\sigma_k - \sigma_n = (\sigma_k - \sigma_{k+1}) + (\sigma_{k+1} - \sigma_{k+2}) + \dots + (\sigma_{n-1} - \sigma_n) = \sum_{j=k}^{n-1} \sigma_j - \sigma_{j+1} \quad (\text{C.40})$$

Utilizando (C.37) en (C.40),

$$\sigma_k - \sigma_n \leq \sum_{j=k}^{n-1} \frac{2^{j+1}}{1 + 2^{2j+2}} = \sum_{j=k+1}^n \frac{2^j}{1 + 2^{2j}} \leq \sum_{j=k+1}^n \sigma_j \quad (\text{C.41})$$

con lo cual se concluye que

$$\sigma_k \leq \sigma_n + \sum_{j=k+1}^n \sigma_j, \quad \forall 0 \leq k \leq n \quad (\text{C.42})$$

Por último, como la suma de los términos $\arctan(1)$, $\arctan(1/2)$, $\arctan(1/4)$ $\arctan(1/8)$ es mayor a $\pi/2$, entonces se concluye que

$$|r| \leq \frac{\pi}{2} < \sum_{j=0}^3 \arctan(2^{-j}) < \sigma_n + \sum_{j=0}^n \sigma_j \quad (\text{C.43})$$

lo que completa la demostración.

Teorema C.5. *El algoritmo CORDIC converge en fase.*

Demostración:

Sea la secuencia $s_k = \phi - z_k = \sum_{j=0}^{k-1} d_j \sigma_j$. Se observa que $s_0 = \phi - z_0 = 0$ y $s_{k+1} = \sum_{j=0}^k d_j \sigma_j = s_k + d_k \sigma_k$. Para $r = \phi$ se tiene $\rho_k = \text{signo}(r - s_k) = \text{signo}(\phi - s_k) = \text{signo}(z_k) = d_k$. Por lo tanto, la secuencia

$$|\phi - s_{n+1}| \leq \sigma_n = \arctan(2^{-n}) \leq \arctan\left(\frac{1}{2^n}\right) \quad (\text{C.44})$$

satisface el Teorema de Convergencia, lo cual demuestra que el algoritmo CORDIC converge para cualquier secuencia $\{d_i\}_{i=0,\dots,n-1}$.

Acrónimos

MIMO	Multiple Input Multiple Output
MISO	Multiple Input Single Output
SIMO	Single Input Multiple Output
SISO	Single Input Single Output
AM	Amplitude Modulation
FM	Frequency modulation
HPBW	Half Power Beam Width
RF	Radio Frequency
FPGA	Field Programmable Gate Array
DSP	Digital Signal Processor
MSE	Mean Squared Error
SIR	Signal Interference Ratio
MVDR	Minimum Variance Distortionless Response
LMS	Least Mean Squares
RLS	Recursive Least Squares
AOA	Angle of Arrival
LMS	Least Mean Squares
PN	Pseudo-Noise
CDMA	Code Division Multiple Access
BER	Bit Error Rate
O-STBC	Orthogonal Space-time Block Code
LAST	Layer Space-Time Technique
CORDIC	COordinate Rotation DIgital Computer
BC	Boundary Cell
IC	Internal Cell
OFDM	Orthogonal Frequency Division Multiplexing

QRD	QR Decomposition
LE	Logic Element
MAC	Multiply Accumulate
LUT	Look Up Table
HDL	Hardware Description Language
ROM	Read Only Memory
RTL	Register Transfer Level
IP	Intellectual Property
PC	Personal Computer
UART	Universal Asynchronous Receiver Transmitter

Referencias

- [1] Abbas Mohammadi & Fadhel M. Ghannouchi, “RF Transceiver Design for MIMO Wireless Communications”, Springer, p1-5, 2012.
- [2] Paulraj, A., Nabar, R., Gore, D., “Introduction to Space-Time Wireless Communications”. Cambridge University Press, 2003.
- [3] Kaveh Pahlavan & Allen H. Levesque, “Wireless information Network”, John Wiley & Sons, Inc., p55-64, 2005.
- [4] Toby Haynes, Spectrum Signal Processing, “A Primer on Digital Beamforming”, 1998.
- [5] John Litva & Titus Kwok-Yeung Lo, “Digital Beamforming in Wireless Communications”, Artech House, 1996.
- [6] Abbas Mohammadi & Fadhel M. Ghannouchi, “RF Transceiver Design for MIMO Wireless Communications”, Springer, p17-20, 2012.
- [7] [http://en.wikipedia.org/wiki/Antenna_\(radio\)](http://en.wikipedia.org/wiki/Antenna_(radio)), “Antenna (radio)”, Wikipedia, 2013.
- [8] “FPGA-based Implementation of Signal Processing Systems”, Roger Woods, John McAllister, Gaye Lightbody, Ying Yi, Wiley, p271-300, 2008.
- [9] Ray Andraka, “A survey of CORDIC algorithms for FPGA based computers”, Andraka Consulting Group.
- [10] Volder, J., “The CORDIC Trigonometric Computing Technique”, IRE Trans. Electronic Computing, Vol EC-8, 1959.
- [11] G. Lightbody, R.L. Walke, R. Woods, J. McCanny, “Novel Mapping of a Linear QR Architecture”, Proc. ICASSP, Volume IV, p1933-1936, 1999.
- [12] Simon Haykin, “Adaptive Filter Theory”, Prentice Hall, 1986.
- [13] “Implementation of CORDIC-Based QRD-RLS Algorithm on Altera Stratix FPGA with Embedded Nios Soft Processor Technology”, Altera.

- [14] Marjan Karkooti, Joseph R. Cavallaro, Chris Dick, “FPGA Implementation of Matrix Inversion Using QRD-RLS Algorithm”, Center for Multimedia Communication, Department of Electrical and Computer Engineering, Rice University, Xilinx
- [15] Dongdong Chen, Mihai SIMA, “Fixed-Point CORDIC-Based QR Decomposition by Givens Rotations on FPGA”, Department of Electrical and Computer Engineering, University of Victoria
- [16] “All Programmable FPGAs”, <http://www.xilinx.com/products/silicon-devices/fpga.html>
- [17] J. S. Walter, “A Unified Algorithm for Elementary Functions”, AFIPS Conf. Proc., Vol. 38, pp. 379-385, 1971.

Federico Damián Camarda