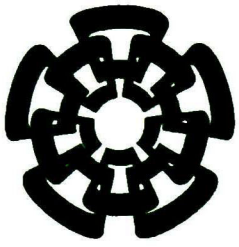
CT 815-551
DON. 7015



Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional
Unidad Guadalajara

Diseño e implementación de un igualador de canal MMSE con restricciones de área

Tesis que presenta:

Alberto Rodríguez García

para obtener el grado de:

Maestro en Ciencias

en la especialidad de:

Ingeniería Eléctrica

Directores de Tesis

Dr. Ramón Parra Michel
Dr. Joaquín Cortez González

CINVESTAV del IPN Unidad Guadalajara, Guadalajara, Jalisco, Enero de 2014.

Diseño e implementación de un igualador de canal MMSE con restricciones de área

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

Por:
Alberto Rodríguez García
Ingeniero en Electrónica
Instituto Tecnológico de Sonora 2007-2011

Becario de CONACYT, expediente no. 263586

Directores de Tesis
Dr. Ramón Parra Michel
Dr. Joaquín Cortez González

CINVESTAV del IPN Unidad Guadalajara, Enero de 2014.

Contenido

Resumen

Las señales que viajan a través de canales dispersivos sufren de la llamada interferencia intersimbólica, que degrada en gran medida el rendimiento del sistema. Uno de los principales enfoques para contrarrestar este problema es mediante el uso de igualadores de canal. Entre los diferentes enfoques para la elaboración de un igualador de canal, el MMSE (Error Cuadrático Medio Mínimo) en uno de los más utilizados, debido a su rendimiento y que es fácil de entender. Sin embargo, las aplicaciones de tiempo real requieren que los igualadores MMSE sean implementados en hardware con un alto rendimiento y reducción de los recursos, lo que requiere un diseño con una buena arquitectura. Esta tesis presenta el diseño e implementación de un igualador de canal utilizando el criterio MMSE. El diseño se llevó a cabo utilizando el HDL (Lenguaje de descripción de Hardware) *Verilog* y *fue sintetizado* utilizando la tecnología FPGA Altera.

El diseño se basa en una arquitectura sistólica que se ha modificado para reducir el consumo de los recursos de hardware. El diseño como resultado se compara con una versión sistólica y una versión no configurable de mismo. Los resultados muestran que el trabajo propuesto logra una reducción significativa en el uso de los recursos de hardware. En particular, un igualador de canal con treinta coeficientes puede ser procesado en aproximadamente 300 μs con una frecuencia de reloj de 50 MHz. La arquitectura MMSE diseñada puede ser utilizada en un sistema de comunicaciones práctico.

Abstract

Signals traveling trough dispersive channels suffer of the so-called inter symbol interference, that greatly degrades system performance. One of the main approaches to counteract this problem is by using channel equalizers. Among the different approaches for devising a channel equalizer, the MMSE (Minimun Mean Square Error) in one of the most used ones, due to its performance and that it is easy to understand. However, real time applications require that MMSE equalizer be implemented in hardware with high throughput and reduced resources, which requires a fine architectural design. This thesis present the design and implementation of a channel equalizer using the MMSE criterion. The design was implemented using the HDL (Hardware Description Language) Verilog and it was synthesized using the Altera FPGA technology.

The design is based on a systolic architecture that is modified for low hardware resources consumption. The resulted design is compared against a systolic version and a non-configurable version of it. The results show that the proposed work achieves a significant reduction in hardware resources usage. Particularly, a thirty coefficients channel equalizer can be processed in about 300 μs with a 50 MHz clock-frequency. The designed MMSE architecture can be used in practical communications system.

Agradecimientos

A mis padres que me han conducido por la vida, con amor y paciencia, por enseñarme lo que han recogido a su paso por la vida y ayudarme a salir adelante en la adversidad.

A mi hermana con quien compartí muchos alegres momentos a lo largo de mi vida.

Al Dr. Ramón Parra por haber confiado en mi persona, por la paciencia en la dirección de este trabajo.

A mis maestros por brindarme su apoyo y compartirme sus conocimientos y experiencias.

Gracias también a mis compañeros de generación, Benjamín, Gustavo, Daniel, Laura, Joaquín, con quienes compartí desvelos y logros durante estos 2 años.

Agradezco al Consejo Nacional de Ciencia y Tecnología por la beca que me proporcionó para realizar esta maestría.

Capítulo 1

Introducción

En este capítulo se da una introducción a un sistema de comunicaciones, aclarando la importancia de un igualador de canal. Se presenta el problema a resolver, los objetivos de proyecto, así como la motivación que llevo a la realización del mismo. También, es presentado el estado del arte, donde se dan a conocer los trabajos realizados en este campo. Finalmente, es presentada la estructura del contenido de este documento.

1.1. Introducción

En un sistema de comunicaciones existen tres bloques básicos el transmisor, el canal de comunicación y el receptor. La información fluye desde el transmisor hacia el receptor, pero dado que no se cuenta con un canal ideal, la señal transmitida se modifica al pasar por el canal. Esto provoca que el receptor tenga que realizar la tarea de estimar la señal transmitida en base a la señal que recibe. Para ayudar a esto una de las tareas importantes en el receptor es la igualación de canal, el cual se encarga de disminuir las distorsiones ocasionadas a la señal transmitida cuando viaja por el canal. Existen diferentes métodos para lograr esto; una de los más usuales es ver el canal de comunicaciones como un filtro de respuesta al impulso finita (FIR), en ese caso los datos transmitidos se convolucionan con el canal y el trabajo del igualador es eliminar los efectos de la convolución para obtener un estimado de la señal transmitida.

$$y(k) = \sum_{-\infty}^{\infty} b(i)h(k-i) \tag{1.1}$$

$$Y(w) = B(w)H(w) \tag{1.2}$$

Las ecuaciones (1.1) y (1.2) muestran la salida de un canal, representado como filtro FIR, en tiempo y en frecuencia, respectivamente. Donde $y(k)$ son los datos recibidos

del canal, $b(k)$ son los datos transmitidos, y $h(k)$ representa el canal. Es fácil notar que el igualador, debe ser un filtro tal que su respuesta al impulso en frecuencia sea el inverso de la respuesta al impulso en frecuencia del canal, así los efectos del canal serán eliminados. Existen varios criterio para realizar la igualación de canal, el escoger uno depende de la aplicación y de los requerimientos que se tenga para el sistema a implementar. Sin embargo, en lo general se buscan algoritmos que sean fáciles de implementar y cuyo cálculo no consuma de una gran cantidad de tiempo. Uno de los más utilizados es el criterio de error cuadrático medio mínimo (MMSE), el cual como se explica en [1] tiene una complejidad baja y facilidad de implementación.

1.2. Estado del Arte

En la actualidad los igualadores de canal con criterio MMSE son utilizados preferentemente por su fácil implementación, este criterio también es implementado para sistemas MIMO [2]. A pesar de ello las formas de implementación son parecidas a la propuesta en esta tesis, como es el caso de [3], cuya arquitectura se muestra en la Fig. 1.1

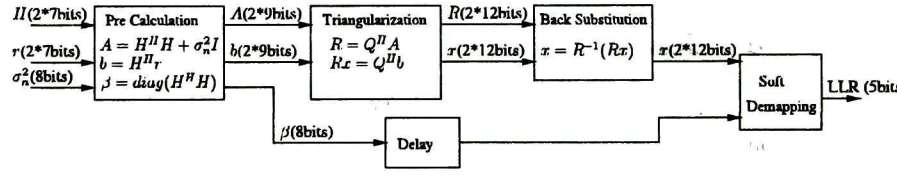


Figura 1.1: Arquitectura de un 4x4 MIMO detector

La arquitectura se basa en el cálculo de una matriz triangular; posteriormente, utiliza sustitución hacia atrás para resolver las ecuaciones para un igualador MMSE. Este método es muy similar al descrito en [1], donde se realiza una eliminación de Gauss para resolver la ecuación para el criterio MMSE. La arquitectura en [1] fue realizada para el estándar de 60 GHz; por lo que su velocidad de procesamiento es muy rápida, pero consume una gran cantidad de recursos de hardware.

También, existe implementaciones para software como las presentadas en [4], donde se puede evaluar el rendimiento al escoger diferentes procesadores y dispositivos para la implementación. Aun así, ninguna de las arquitecturas muestra tener algún grado de configurabilidad y se enfocan en resolver algún problema en específico. En el presente trabajo se buscara crear una arquitectura de propósito general, que pueda ser integrada a un sistema con facilidad y pueda ser configurada una vez se encuentre implementada.

1.3. Motivación

Los sistemas de comunicaciones son de vital importancia en nuestra vida cotidiana, tanto para usar un celular, una computadora, un dispositivo Bluetooth, etc. Pero los requerimientos para los dispositivos han cambiado durante los últimos años. En 1991, Joe Mitola acuño el termino Software Radio, para referirse a las comunicaciones de radio reprogramables o reconfigurables [5]. Donde la misma pieza de hardware puede desarrollar diversas funciones en diferentes tiempos. Se llama Software Radio a los sistemas de radio que en su implementación tienen una parte de hardware y otra de Software, contrario a los sistemas de radio convencionales que se encuentran definidos únicamente por hardware.

Esto ha provocado que los sistemas de comunicaciones en la actualidad, deban ser reconfigurables, es decir puedan realizar diferentes tipos de funciones, y cubrir una gran cantidad de estándares. Esto implica, que el harware diseñado para estos sistemas debe ser en cierta manera configurable, es decir, con el mismo hardware puedan realizar diferentes tareas. Teniendo esto en mente, es obvio notar que los bloques que se usen en el sistema, como sería el igualador de canal, deben ser bloques configurables que brinde al sistema completo una grado más de configurabilidad. Dado que el igualador de canal es un bloque importante en el diseño de sistemas de comunicaciones, contar con un igualador de bajo costo en cuestión de hardware y configurable, es necesario para los actuales requerimientos en los sistemas de comunicaciones.

1.4. Definición del Problema

Dada la importancia de los igualadores de canal en el sistema de comunicaciones, se plantea diseñar un igualador de canal utilizando el criterio MMSE, dada la facilidad de implementación de este criterio, que pueda ser configurable para el número de coeficientes a calcular. Así, obtener un módulo fácilmente integrable en un sistema.

1.5. Objetivos

1.5.1. Objetivo general

Diseñar e implementar un módulo para el cálculo de los coeficientes de un filtro igualador de canal bajo el criterio MMSE, en el lenguaje de descripción de hardware Verilog.

1.5.2. Objetivos particulares

- Implementar una diseño eficiente en el uso de recurso de hardware.
- Realizar una arquitectura con la capacidad de ser configurable para el número de coeficientes.

- Verificar el funcionamiento del diseño comparado con un modelo de oro.

1.6. Organización de la tesis

El documento está organizado en 5 capítulos y su contenido es el siguiente:

- El capítulo 2 habla sobre el objetivo del igualador de canal, se explica cómo funciona el criterio MMSE, y se muestra la adaptación del algoritmo para su implementación.
- En el capítulo 3 se explica detalladamente la arquitectura realizada, la cual toma como base la presentada en [1], por lo tanto solo son mostradas las diferencias y modificaciones realizadas.
- En el capítulo 4 se muestran el plan de verificación para el diseño, así como su resultados de síntesis para distintos dispositivos y una comparativa contra los resultados presentados en [1].
- Finalmente, en el capítulo 5 se presentan las conclusiones sobre la tesis, mostrando sus aportaciones principales. También se da a conocer el trabajo futuro a realizar en esta área.

Capítulo 2

Igualador de Canal

En el presente capitulo se da una introducción a las partes principales de un sistema de comunicaciones, se explica la función de un igualador de canal y se da a conocer el desarrollo matemático para un igualador con el criterio MMSE. También, se muestra la adecuación del algoritmo para realizar la implementación.

2.1. Elementos básicos de un sistema de comunicaciones

Un diagrama a bloques de un típico sistema de comunicaciones se muestra en la Fig. 2.1, la salida de la fuente puede ser una señal analógica o una señal digital, en sistema de comunicaciones digitales el mensaje producido por la fuente es convertido en una secuencia de dígitos binaria. Idealmente la fuente debería de producir los más pocos dígitos binarios que sean posibles, evitando la redundancia en la información. Como esta característica no se tiene directamente de la fuente, el proceso de eficientar la salida de la fuente se conoce como codificación de fuente.

La secuencia binaria del codificador de fuente se pasa por el codificador de canal, este introduce de una manera controlada alguna redundancia en la información, esta será usada en el receptor para disminuir los efectos del ruido e interferencia en la señal transmitida. La secuencia binaria de salida del codificador de canal es pasada a un modulador digital, este sirve como una interfaz con el canal de comunicaciones. Todos los canales de comunicaciones son capaces de transmitir alguna forma de onda, como puede ser una señal eléctrica; el propósito del modulador es transformar la secuencia binaria en una forma de onda adecuada para ser transmitida. El caso más simple para un modulador digital es convertir un dígito 0 en una forma de onda s_0 y un dígito 1 en otra forma de onda s_1, de esta manera cada bit dado por el codificador de canal es transmitido separadamente. También existen moduladores que puede transmitir b dígitos usando de 2^b distintas formas de ondas.

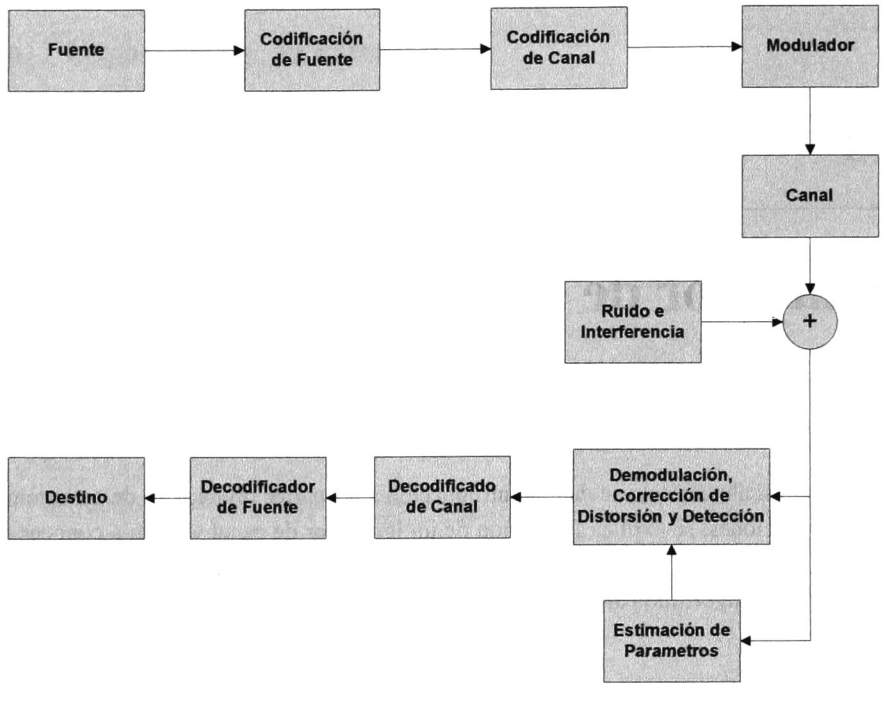


Figura 2.1: Diagrama a bloques de un sistema de comunicaciones.

El canal de comunicación es el medio físico que se usa para enviar la señal transmitida al receptor, por ejemplo el aire, líneas de fibra óptica, líneas de transmisión, etc. En el canal de transmisión la señal es corrompida de alguna manera aleatoria como puede ser por ruido e interferencia. En el receptor, el demodulador recibe la señal corrompida por el canal, y realiza la mejor aproximación de la secuencia binaria producida por el codificador de canal. Esta secuencia es enviada al decodificador de canal, el cual intenta corregir los errores que podrían aparecer en la transmisión, usando la redundancia introducida por el codificador de canal. Por último, el decodificador de fuente toma la salida del decodificador de canal y genera una secuencia lo más parecida a la fuente que sea posible.

El sistema de comunicaciones descrito es apropiado para un simple camino de comunicación entre una fuente y un destino, pero ejemplifica las funciones principales para un sistema en general.

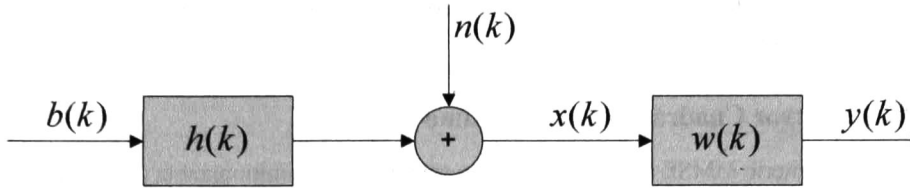


Figura 2.2: Modelo de un sistema de comunicación digital.

2.2. Filtro Igualador

El objetivo de un filtro igualador de canal es disminuir las distorsiones creadas por el canal en la señal transmitida, es decir, se busca obtener un estimado de la señal transmitida a partir de la señal recibida. Si se considera el caso de un simple transmisor enviando información ($b(k)$) a una antena receptora, el más simple dispositivo igualador es normalmente implementado como un filtro de respuesta la impulso finita (FIR por sus siglas en ingles). Usando el modelo de sistema discreto mostrado en la Fig. 2.2, donde $b(k)$ es la señal transmitida, $h(k)$ representa el canal de comunicaciones, $n(k)$ es la componente de ruido blanco aditivo gaussiano, $x(k)$ representa la señal ruidosa recibida, $w(k)$ es la respuesta al impulso del igualador y $y(k)$ son los datos recuperados.

A partir de la Fig. 2.2, la salida del igualador se muestra en (4.1) donde $*$ representa la operación de convolución. Así, los datos de entrada se convolucionan con el canal; después, se les añade ruido. Posteriormente, se convolucionan los datos ruidosos con la respuesta al impulso del igualador para obtener los datos estimados

$$y(k) = w(k) * h(k) * b(k) + w(k) * n(k) \tag{2.1}$$

En base a (4.1), si se escoge una respuesta la impulso del igualador tal que se cumpla (2.2).

$$w(k) * h(k) = \delta(k) \tag{2.2}$$

La salida del igualador será dada por (2.3), donde $n'(k)$ es un término de ruido y $\hat{b}(k)$ son el estimado de los símbolos transmitidos.

$$y(k) = b(k) + n'(k) = \hat{b}(k) \tag{2.3}$$

Es necesario seleccionar un criterio apropiado para el cálculo de los coeficientes $w(k)$, existe mucho criterios como son el valor cuadrático medio del error estimado, la esperanza del valor absoluto, el promedio de la probabilidad de error [6], etc. No obstante, cuando se diseñan sistemas de comunicaciones se requiere de un criterio simple

de implementar, lo que hace al error cuadrático medio mínimo (MMSE por sus siglas en ingles) el más preferentemente escogido.

2.2.1. Error Cuadrático medio mínimo

En el criterio MMSE los coeficientes son calculados para minimizar la media cuadrática del error entre la señal transmitida y la señal estimada, definiendo este como:

$$J_{MSE} = E[|b(k-d) - y(k)|^2] \tag{2.4}$$

donde E es el operador de valor esperado y d representa el retardo de igualación, este último debido a que la señal transmitida sufre retardos provocados por el canal, el filtro acoplado del receptor, el filtro igualador y el filtro formador del transmisor:

Para el caso sin ruido la salida del igualador es

$$y(k) = \sum_{i=0}^{Q-1} w(i)x(k-i) \tag{2.5}$$

escribiendo (2.5) en forma matricial queda

$$y(k) = \mathbf{x}^T\mathbf{w} \tag{2.6}$$

donde el vector $\mathbf{x}$ está compuesto de Q elementos de la señal recibida y está dada por

$$\mathbf{x} = [x(k), x(k-1), ..., x(k-Q+1)]^T \tag{2.7}$$

y el vector $\mathbf{w}$ son los coeficientes del igualador.

$$\mathbf{w} = [w(0), w(1), ..., w(Q-1)]^T \tag{2.8}$$

Desde (4.1), la señal recibida es dada por

$$x(k) = \sum_{i=0}^{M-1} h(i)b(k-i) + n(k) \tag{2.9}$$

donde $h(i)$ es la respuesta al impulso de orden M. Expresando (2.9) en forma matricial queda

$$\mathbf{w} = \mathbf{Hb} + \mathbf{n} \tag{2.10}$$

donde $\mathbf{H}$ es la matriz de canal estimado, con estructura tipo toeplitz definida como

$$\mathbf{H} = \begin{bmatrix} h(0) & h(1) & \cdots & h(M-1) & 0 & \cdots & 0 \\ 0 & h(0) & \cdots & h(M-2) & h(M-1) & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & h(0) & h(1) & \cdots & h(M-1) \end{bmatrix} \tag{2.11}$$

y

$$\mathbf{b} = [b(k), b(k-1), ..., b(k-Q-M+2)]^T \tag{2.12}$$

$$\mathbf{n} = [n(k), n(k-1), ..., n(k-Q+1)]^T \tag{2.13}$$

Sustituyendo (2.10) y (2.6) en (2.4) tenemos

$$J_{MSE} = E[|b(k-d)|^2] - E[b^*(k-d)\mathbf{x}^T\mathbf{w}] - E[\mathbf{w}^H\mathbf{x}^*b(k-d)] + E[\mathbf{w}^H\mathbf{x}^*\mathbf{x}^T\mathbf{w}] \tag{2.14}$$

donde $(^*)$, $(^H)$ y $(^T)$ denotan conjugación sin transposición, conjugación con transposición y transposición respectivamente. Desarrollando 2.14 y agrupando términos tenemos

$$J_{MSE} = \sigma_b^2 - \sigma_b^2\mathbf{u}_{d+1}^T\mathbf{H}^T\mathbf{w} - \sigma_b^2\mathbf{w}^H\mathbf{H}^*\mathbf{u}_{d+1} + \sigma_b^2\mathbf{w}^H\mathbf{R}\mathbf{w} + \sigma_n^2\mathbf{w}^H\mathbf{w} \tag{2.15}$$

donde $\mathbf{R}$ representa la matriz de correlación del canal estimado dada por

$$\mathbf{R} = \mathbf{H}^*\mathbf{H}^T \tag{2.16}$$

Aplicando el operador de gradiente complejo a (2.15), el vector $\mathbf{w}$ que minimiza J_{MSE} es dado por

$$\mathbf{w} = (\mathbf{R} + \frac{\sigma_n^2}{\sigma_b^2}\mathbf{I})^{-1}\mathbf{H}^*\mathbf{u}_d \tag{2.17}$$

donde el $\mathbf{u}_d$, es un vector unitario de orden N, donde el uno aparece en la *d-th* posición. El desarrollo de (2.15) para llegar a (2.17) se muestra detalladamente en [7].

2.2.2. Adecuación de Algoritmo para Implementación

Para poder realizar un diseño para implementación, es necesario analizar (2.17), para posteriormente determinar una forma de implementar que requiera un mínimo de operaciones. Según el análisis presentado en [1] se selecciona el siguiente método. Primero la matriz $\mathbf{R}$ definida en (2.16), tiene la siguiente estructura:

$$\mathbf{H} = \begin{bmatrix} r(0) & r(1) & r(2) & \cdots & r(Q-1) \\ r^*(1) & r(0) & r(1) & \cdots & r(Q-2) \\ r^*(2) & r^*(1) & r(0) & \cdots & r(Q-3) \\ \vdots & \vdots & . & \vdots & \vdots \\ r^*(Q-1) & r^*(Q-2) & r^*(Q-3) & \cdots & r(0) \end{bmatrix} \tag{2.18}$$

Esta es una matriz diagonal dominante, se observa además que un solo renglón de la matriz contiene toda la información, tomando esto en consideración se puede calcular

solo el primer renglón de la matriz, para a partir de este formar la matriz completa por medio de corrimientos y conjugación. Para realizar esto solo es necesario calcular la correlación del canal estimado **h** de la cual se tomaran lo valores para el primer renglón de la matriz y posteriormente formar la matriz **R**.

La siguiente operación es el termino $\mathbf{R} + \frac{\sigma_n^2}{\sigma_b^2}\mathbf{I}$ en (2.17), este realiza la suma de dos matrices dado que **I** es la matriz identidad solo se está sumando a la diagonal de **R** la relación señal a ruido. Dadas las propiedades de **R** y la estructura mostrada en (2.18), se observa que los valores en la diagonal siempre son $r(0)$; sabiendo esto es posible calcular el primer renglón de **R** y luego sumar al elemento $r(0)$ la relación señal a ruido, como se muestra en (2.19). Al hacer esto solo es necesario realizar una suma al formar la matriz.

$$\mathbf{r} = [r(0) + \frac{\sigma_n^2}{\sigma_b^2}, r(1), r(2), \cdots, r(Q-1)] \tag{2.19}$$

El siguiente termino es $\mathbf{H}^*\mathbf{u}_d$ dado que el vector $\mathbf{u}_d$ solo tiene el valor de uno en la $d-th$ y el resto son ceros, esta operación se convierte simplemente en la selección de un columna de $\mathbf{H}^*$. Finalmente, queda el problema de calcular la inversa de la matriz $\mathbf{R} + \frac{\sigma_n^2}{\sigma_b^2}\mathbf{I}$, para no tener que realizar la operación de la inversa de una matriz, se modifica (2.17). Primero se nombrara **A** al termino $\mathbf{R} + \frac{\sigma_n^2}{\sigma_b^2}\mathbf{I}$, el cual es una matriz diagonal dominante, y **b** será el termino $\mathbf{H}^*\mathbf{u}_d$, el cual es un vector. De esta manera se puede rescribir (2.17) como sigue

$$\mathbf{w} = (\mathbf{A})^{-1}\mathbf{b} \tag{2.20}$$

y multiplicando por **A** se tiene

$$\mathbf{A}\mathbf{w} = \mathbf{b} \tag{2.21}$$

De esta manera, se puede ver (2.21) como encontrar la solución a un sistema lineal de ecuaciones. Con lo cual se pueden utilizar diferentes métodos para resolver el problema sin la necesidad del cálculo de la inversa. Como se muestra en [1] el método menos costoso, en cuanto al número de operaciones para este problema en específico, es utilizar eliminación de Gauss, en el cual primero se calcula la matriz diagonal superior de la matriz aumentada formada por **A** y el vector **b**, después por medio de sustitución hacia atrás se encuentran los valores para **w**.

Con estas modificaciones se puede realizar un diseño cuyos bloques funcionales principales serán los siguientes:

- Autocorrelación del canal estimado
- Selección de columna de la matriz $\mathbf{H}^*$

- Diagonalización para eliminación de Gauss
- Sustitución hacia atrás

Así, primero se realiza la autocorrelación del canal estimado para encontrar **h**, luego se le suma el valor de la relación señal a ruido para formar la matriz **A**, después se selecciona la columna de la matriz $\mathbf{H}^*$ para formar el vector **b** y finalmente utilizando eliminación de Gauss resolver (2.21).

Capítulo 3

Diseño de una arquitectura para el cálculo de coeficientes

En el presente capítulo se describen la arquitectura para un filtro igualador MMSE, con un número de coeficientes a calcular configurable. Este se basa en la arquitectura presentada en [1], por lo que a lo largo del capítulo se explicaran los módulos modificados para esta implementación, los detalles sobra la implementación en [1] son omitidos, y solo interesan las diferencia entre esta y la implementación configurable, si se desea una explicación más detallada de todos los bloques que componen la arquitectura base se puede consultar en [1].

El capítulo maneja la siguiente estructura, se inicia con una explicación general de la arquitectura base, después cada bloque funcional de la arquitectura es explicado, mostrando los cambios y modificaciones hasta llegar a su versión configurable. Una vez explicadas las modificación a los bloques funcionales que compone la arquitectura se da una explicación del diseño configurable completo.

3.1. Arquitectura base para el calculo de coeficientes con el criterio MMSE

La Arquitectura del módulo que calcula los coeficientes para el filtro igualador de canal bajo del criterio MMSE, está basada en 4 bloques:

- **Autocorrelacionador**: Realiza la autocorrelación del canal estimado.
- **Eliminación de Gauss**: Forma la matriz de canal y después calcula la matriz triangular.
- **Selector de Columna**: Selecciona una columna de la matriz de canal basado en el retardo de igualación y el número de coeficientes a calcular.

Tabla 3.1: Descripción de señales de entrada y salida del módulo MMSE

Entradas	**Descripción**
Canal_Estimado_PR	Parte real del canal estimado.
Canal_Estimado_PI	Parte imaginaria del canal estimado.
Cargar_Canal_Estimado	Señal para la carga del canal estimado activa en alto.
Retardo_De_Igualacion	Valor del retardo de igualación.
Retardo_Igualacion_Listo	Cuando es igual a 1 indica que el retardo de igualación está listo.
SNR_Estimado	Valor de la relación señal a ruido.
SNR_Estimado_Listo	Cuando es igual a 1 indica que el valor de la relación señal a ruido es válido.
Reloj	Señal de reloj.
Reinicio	Reinicio del módulo, esta señal es activa en estado bajo.
Salidas	**Descripción**
Coeficientes_PR	Parte real de los coeficientes calculados.
Coeficientes_PI	Parte imaginaria de los coeficientes calculados.
Coeficiente_Válido	Cuando es igual a 1 indica que un coeficiente calculado es válido.
Igualador_Listo	Cuando es igual a 1 indica que el módulo está disponible para procesar información.

- **Sustitución hacia atrás**: Resuelve el sistema de ecuaciones proveniente de la **Eliminación de Gauss**.

Esta arquitectura fue presentada en [1] y se muestran en la Fig. 3.1. La Tabla 3.1 muestra las señales de entrada y salida para el bloque MMSE. En las siguientes secciones se realizan las modificaciones a los bloques para reducir el uso de recursos de hardware, para tener una arquitectura más eficiente en cuestión de área, manteniendo la misma interfaz de entrada y salida presentada en [1]; esto permitirá que el nuevo módulo puede integrarse fácilmente a diseños donde se esté utilizando la arquitectura base. También, se muestra la modificación de los bloques para realizar una arquitectura configurable en el número de coeficientes para el bloque MMSE.

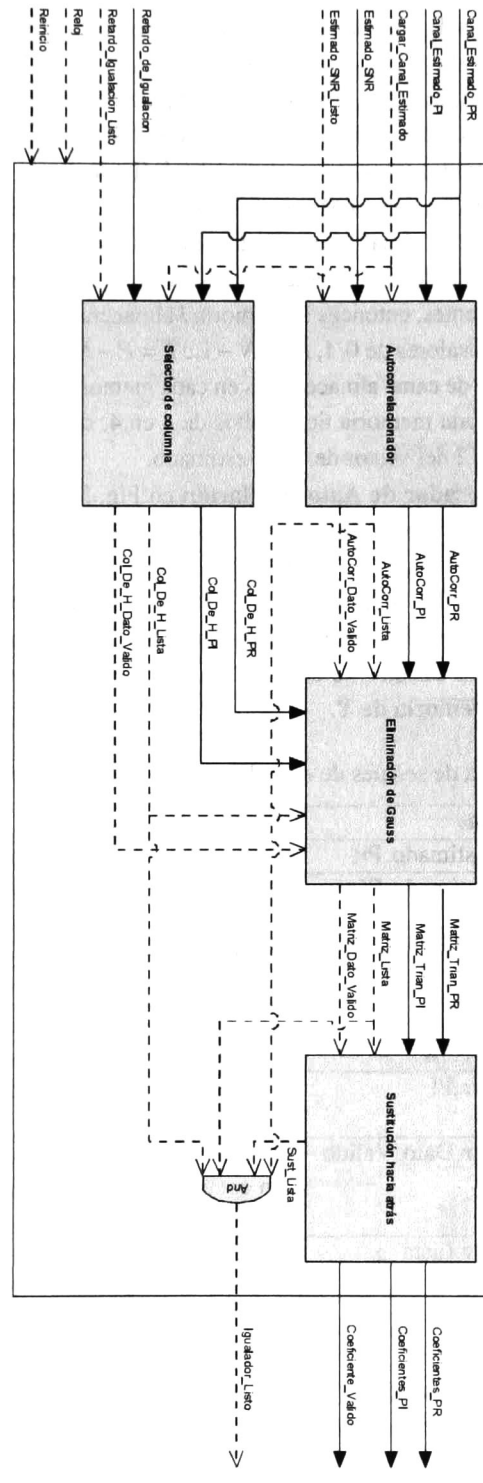


Figura 3.1: Arquitectura para el Módulo MMSE

3.2. Autocorrelacionador

El módulo autocorrelacionador se encarga de realizar la operación de autocorrelación del vector de canal estimado, para a partir de ella obtener la matriz de autocorrelación del canal. La arquitectura propuesta en [1] para este módulo está diseñada para reducir el tiempo de procesamiento. Esta arquitectura se muestra en la Fig. 3.2 y una descripción de sus entradas y salidas se observa en la Tabla 3.2.

El módulo se compone de una **Memoria de canal**, que es la responsable de guardar el conjugado del vector de canal estimado. Las memorias de coeficientes almacenan el vector de canal estimado de la siguiente manera, si existen N memorias para un vector de canal con P coeficientes, entonces la memoria i almacena los coeficientes $i, i+N, i+2N, ..., I$, donde i tiene valores de $0, 1, 2, ..., N-1$ e $I = P-N+i$. La Fig. 3.3 muestra los coeficientes del vector de canal almacenados en cada memoria para el caso de $N = 4$ con $P = 16$; se ve como cada memoria tiene saltos de 4 en 4, así la memoria 0 almacenara los coeficientes 0,4,8,12 del vector de canal estimado.

Los bloques **Procesador de Autocorrelación** en Fig. 3.2, realizan una multiplicación acumulación de números complejos para el procesamiento de la correlación. La **Memoria de X** y la **Memoria de Y** almacenan el canal estimado, dado por el ultimo procesador, y los resultados parciales de la correlación respectivamente. Los multiplexores **MUX Y**, **MUX S** y **MUX X** son usados por la **Unidad de control** para determinar el flujo de los datos. La **Unidad de control** controla, además, la lectura y escritura de **Memoria de X** y la **Memoria de Y**.

Tabla 3.2: Descripción de señales de entrada y salida del módulo Autocorrelacionador

Entradas	**Descripción**
Canal_Estimado_PR	Parte real del canal estimado.
Canal_Estimado_PI	Parte imaginaria del canal estimado.
Cargar_Canal_Estimado	Señal para la carga del canal estimado cuando es igual a 1.
Salidas	**Descripción**
AutoCorr_PR	Parte real de la autocorrelación.
AutoCorr_PI	Parte imaginaria de la autocorrelación.
AutoCorr_Dato_Válido	Cuando es igual a 1 indica que un coeficiente de la autocorrelación es válido.
AutoCorr_Lista	Cuando es igual a 1 la autocorrelación y el módulo se encuentran listos.

El funcionamiento del autocorrelacionador es como sigue: primero el canal estimado es cargado en la **Memoria de canal** y las **Memorias de coeficientes**, esto se logra

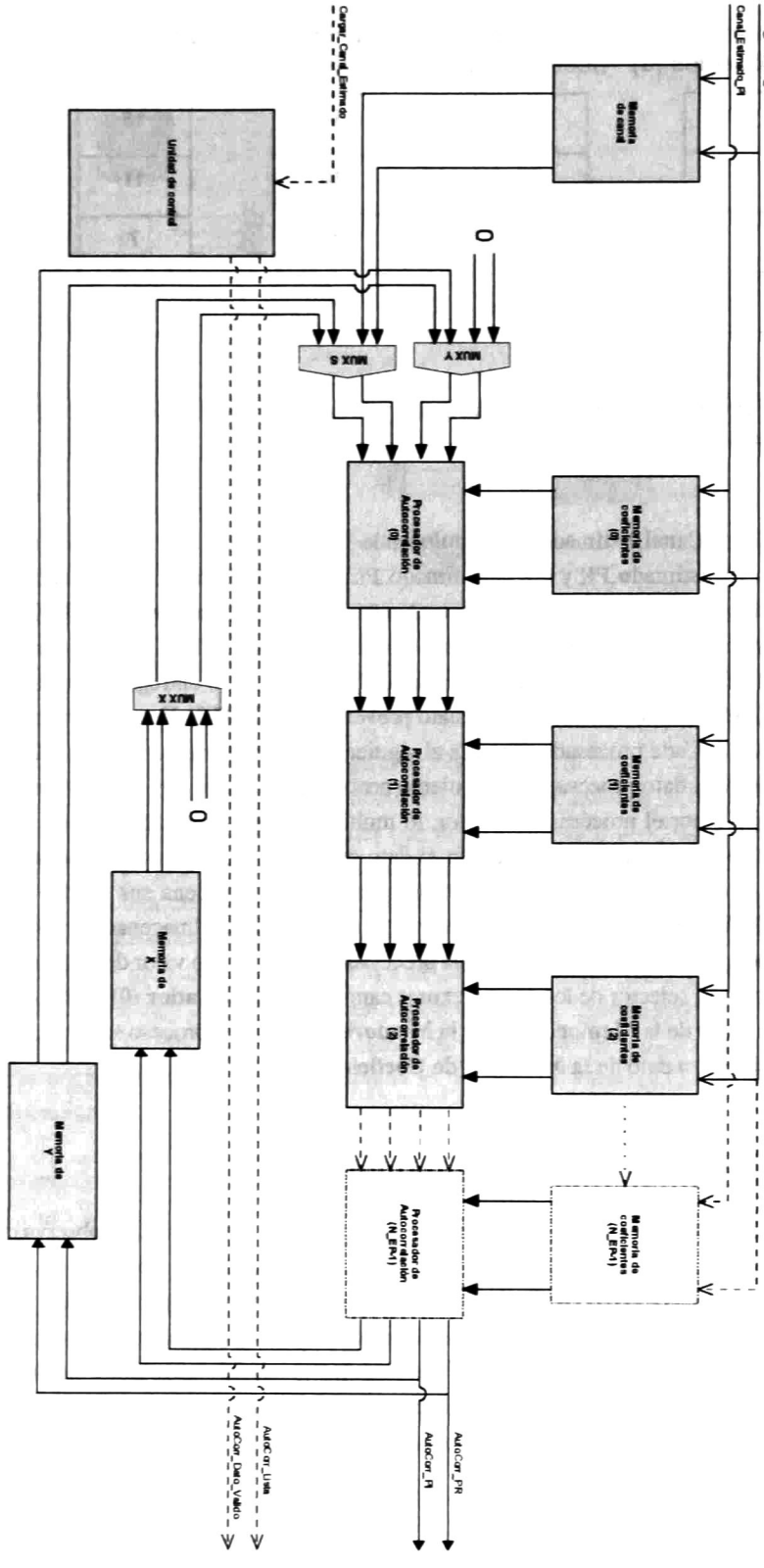


Figura 3.2: Arquitectura del módulo Autocorrelacionador.

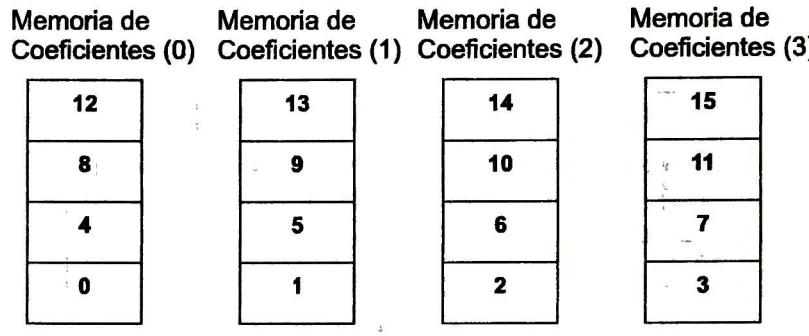


Figura 3.3: Orden de los coeficientes del canal guardados en las Memorias de coeficientes para $N = 4$ y $P = 16$

haciendo Carga_Canal_Estimado = 1 y colocando los valores de canal estimado en las entradas Canal_Estimado_PR y Canal_Estimado_PI. Después, el primer dato de cada memoria es leído por su procesador correspondiente, los datos en la **Memoria de Canal** son leídos y pasados al **Procesador (0)** a través del multiplexor (**MUX**) **S**; el procesador toma el dato de la **Memoria de canal**, lo multiplica por el dato entregado por la **Memoria de Coeficientes** y lo suma con el dato proveniente del **MUX Y**, que inicialmente selecciona un 0. Cada procesador entrega al siguiente el dato proveniente de la **Memoria de Canal** y el dato procesado; el siguiente procesador toma el dato de la **Memoria de Canal**, dado por el procesador anterior, lo multiplica por el dato almacenado en su **Memoria de Coeficientes** y lo suma con el dato procesado por un procesador previo; el proceso se repite hasta llegar al último procesador, éste almacena sus salidas en la **Memoria de X** y la **Memoria de Y**. Una vez que todos los datos almacenados en la memoria de coeficientes son procesados, cada procesador lee un nuevo valor de su memoria de coeficientes; el selector de los multiplexores cambia y el **Procesador (0)** recibe como entradas los datos de la **Memoria de X** y la **Memoria de Y**, así el proceso se repite hasta que se lea el último dato de la **Memoria de Coeficientes**.

3.3. Autocorrelacionador Secuencial

En la arquitectura del autocorrelacionador mostrada en la sección 3.2, se observa que el elemento con un consumo mayor de recursos de hardware es el procesador de autocorrelación, por el uso de un multiplicador complejo. Por tanto si se restringe el módulo autocorrelacionador a un solo procesador, se obtiene una nueva arquitectura con una menor cantidad de recursos de hardware. Se puede notar que el número de **Memorias de Coeficientes** es dependiente del número de procesadores, por tanto, solo deberá existir una sola **Memoria de Coeficientes**, pero esta contendrá todos los coeficientes del vector

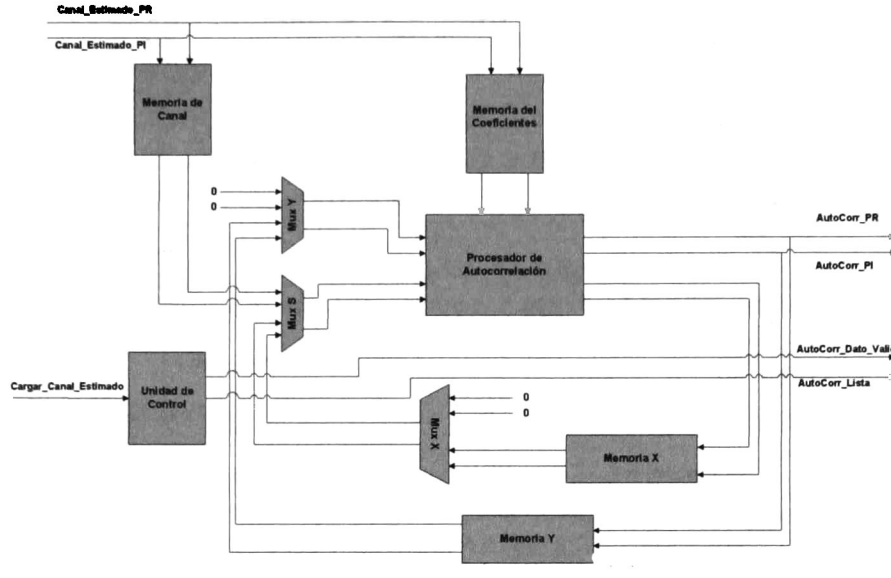


Figura 3.4: Arquitectura para el módulo Autocorrelacionador

estimado como sucedería en el ejemplo de la Fig. 3.3 si $N = 1$.

La arquitectura para este nuevo autocorrelacionador se muestra en la Fig. 3.4, se observar que conserva los mismos bloques que la mostrada en Fig. 3.2, solo quitando las réplicas de los bloques **Procesador de Autocorrelación** y las **Memorias de Coeficientes**; se agrega una entrada (Recolector_Listo) para sincronizar el autocorrelacionador con el bloque **Eliminador de Gauss**. Al solo existir un Procesador la latencia para esta arquitectura se incrementa con respecto a la mostrada en la sección previa. Todos los bloques exceptuando la **Unidad de Control**, conservan la misma estructura interna que se muestra en [1]. Solo la máquina de estado para la **Unidad de Control** fue modificada para la nueva arquitectura, esta se muestra en la Fig. 3.5 y los estados se describen a continuación:

- Estado **S1**: En este estado solo se espera a que llegue la señal de Cargar_canal_estimado para pasar al estado **S2**.
- Estado **S2**: Es un estado de espera hasta que se terminen de ingresar todos los coeficientes del canal, es decir, cuando la **Memoria de Coeficientes** y la **Memoria de Canal** estén llenas.
- Estado **S3**: Se espera a que la entrada Recolector_Listo sea uno para pasar al Estado **S4**.

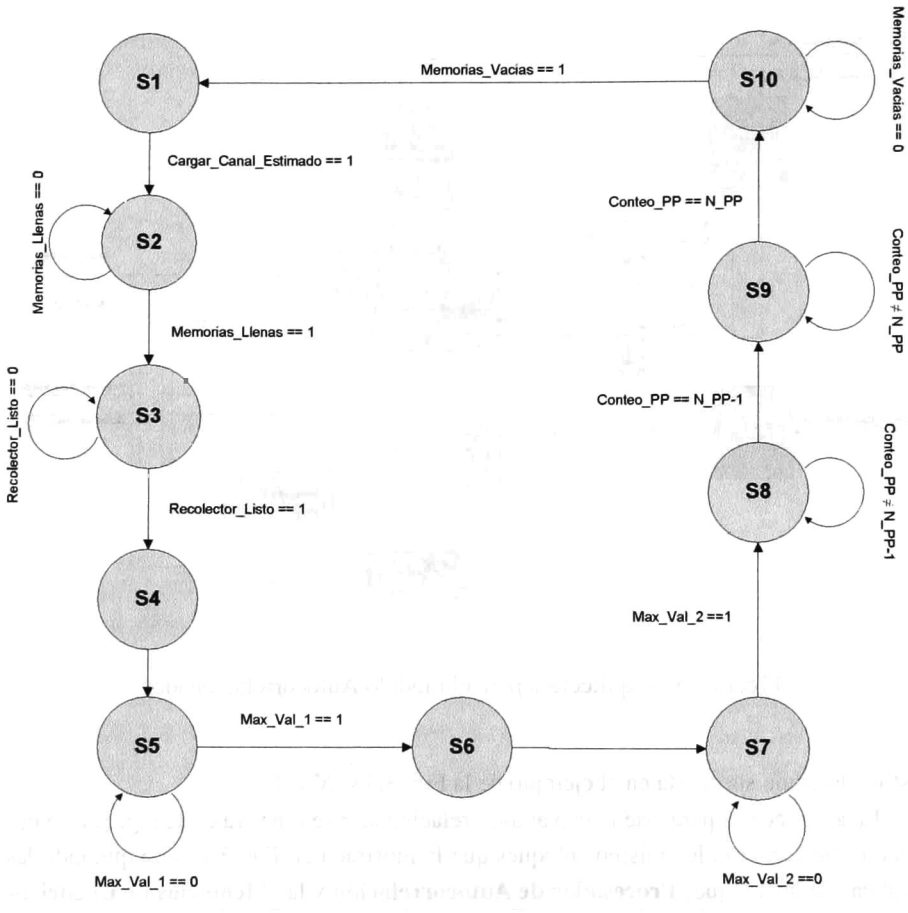


Figura 3.5: Diagrama de transiciones de estados para la Unidad de Control del autocorrelacionador

- Estado **S4**: Se activa los módulos que controlan la lectura de la **Memoria de Coeficientes** y la **Memoria de Canal** para iniciar con el procesamiento de la correlación.
- Estado **S5**: En este estado se habilitan el procesador para que se realice el procesamiento, también, el **MUX S** deja pasar los datos de la **Memoria de Canal** y el **MUX Y** deja pasar ceros hacia el procesador.
- Estado **S6**: En este estado se encienden los módulos que controlan la escritura de la **Memoria X** y la **Memoria Y**, para escribir los resultados parciales de la correlación.
- Estado **S7**: Se espera mientras termina la primera escritura de la **Memoria X** y la **Memoria Y**.

- Estado **S8**: En este estado se encienden los módulos que generan las señales de lectura para la **Memoria X** y la **Memoria Y**; se espera a que solo falte el último proceso parcial.

- Estado **S9**: En este estado se deshabilitan los módulos que controlan la escritura y lectura de la **Memoria X** y la **Memoria Y**, para que solamente se termine de realizar el último proceso parcial; una vez terminado se pasa al estado **S10**.

- Estado **S10**: En este estado se reinician los submódulos de la **Unidad de Control** y se verifica que las memorias estén vacías antes de regresar al estado inicial **S1**.

3.4. Eliminador de Gauss

Como se muestra en [1], este módulo se compone de dos submódulos el **Alimentador de Gauss**, encargado de formar la matriz de autocorrelación y concatenar la columna de H para formar la matriz aumentada, y el **Arreglo de Procesadores de Gauss**, encargado de triangularizar la matriz aumentada. La Fig. 3.6 muestra la arquitectura interna para el módulo **Arreglo de Procesadores de Gauss**, se observa que este se compone por tres bloque principales: las **Memorias de Gauss**, en donde se guarda inicialmente la matriz aumentada que posteriormente sera convertida en una matriz triangular; el **Procesador de Gauss Tipo 1**, es el encargado generar los pivotes en la matriz triangular, es decir, para la fila n tomara el elemento (n, n) para luego dividir toda la fila n entre este elemento; y los **Procesadores de Gauss Tipo 2**, que son los encargados de formar los ceros para la matriz triangular, es decir, toman la fila procesada por el **Procesador Tipo 1** y la restan k veces, donde k es el elemento de la fila que se requiera hacer cero. Siendo M el número de columnas de la matriz aumentada, existen $M - 1$ **Procesadores de Gauss Tipo 2** y cada elemento procesador tiene su propio control para poder leer y escribir su **Memoria de Gauss** correspondiente.

El funcionamiento del módulo es el siguiente, primero el **Procesador de Gauss Tipo 1** lee de la **Memoria de Gauss(0)** la primera fila de la matriz aumentada, para generar a la salida la primera fila de la matriz triangular, esta es pasada al **Procesador de Gauss Tipo 2 (0)**, quien posteriormente la pasara al **Procesador (1)**, el *i-th* **Procesador de Gauss Tipo 2** lee la fila de la **Memoria de Gauss** $(i + 1)$, luego le resta k veces la fila procesada por el **Procesador Tipo 1**, donde k representa el primer valor de la fila guardada en la **Memoria de Gauss** $(i + 1)$. Cada **Procesador Tipo 2** guarda sus resultados en la **Memoria de Gauss** (i). Al terminar este proceso la última memoria estará vacía, esto es llamado el primer proceso parcial. Repetimos el mismo procedimiento para el siguiente proceso parcial, solo que en este caso se desactiva el ultimo **Procesador de Gauss Tipo 2**, dado que para el segundo proceso parcial se requiere procesar una fila menos de la matriz aumentada. Así, en cada proceso parcial se debe de desactivar un procesador hasta solo quedar activo el **Procesador de Gauss Tipo 1**. Una explicación

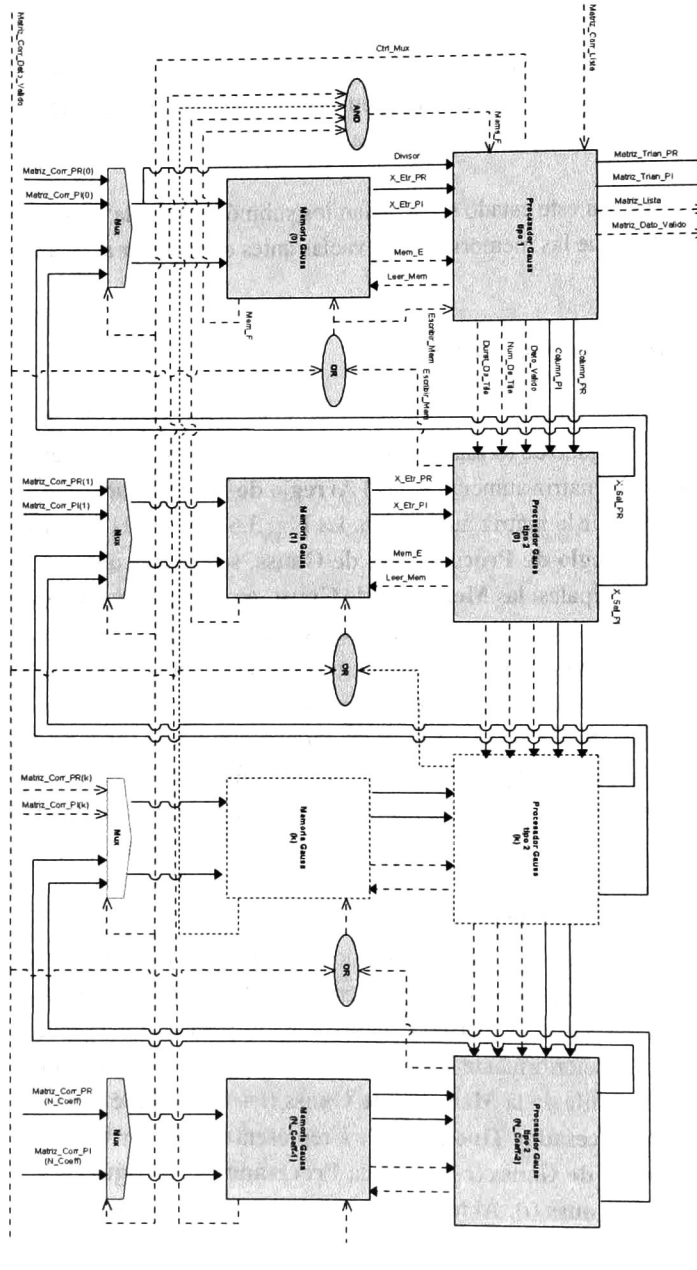


Figura 3.6: Arquitectura del módulo Arreglo de Procesadores de Gauss

detallada de este procedimiento y una descripción más amplia de los módulos se muestra en las secciones 3.5.1.3 y 4.2.2 de [1].

3.5. Eliminador de Gauss Secuencial

El módulo **Eliminador de Gauss** se compone de dos bloques como se vio en la sección anterior, de estos el **Arreglo de Procesadores de Gauss** es el módulo con más alto costo de recursos de hardware, por ser el encargado de realizar el procesamiento para formar la matriz triangular y contener un módulo divisor y multiplicadores complejos. Tomando la estructura presentada en Fig. 3.6, se observa que es posible realizar una arquitectura utilizando un solo elemento **Procesador de Gauss Tipo 2 (PG2)** y un **Procesador de Gauss Tipo 1 (PG1)** para todo el procesamiento, dando como resultado la arquitectura mostrada en Fig. 3.7. La nueva arquitectura añade un multiplexor para controlar de que memoria se leerán los datos procesados por **PG2** y un demultiplexor para determinar a qué memoria le llegaran las señales de lectura y escritura dadas por **PG2**, esto hace posible realizar una arquitectura que no requiera el hardware de los $M-1$ **PG2**.

Sin embargo, como solo existirá un **PG2** es necesario una memoria para guardar los datos provenientes de **PG1**, ya que estos son necesarios para el procesamiento de las filas guardadas en cada una de las memorias conectadas al multiplexor de **PG2**. La escritura de la memoria de datos procesados es controlada por el **PG2**, usando la señal de Dato_Valido, y la lectura es controlada por medio de la señal de lectura que se usa en las **Memorias de Gauss**. También se agregó otro bloque de control para controlar el multiplexor y el demultiplexor, así, como determinar el encendido y apagado de los procesadores. Por lo tanto, las máquinas de estados para los procesadores deben ser modificadas para lograr esta nueva funcionalidad. Estos cambios y los nuevos bloques agregados se muestran en las siguientes secciones.

3.5.1. Memoria de Datos Procesados

La Fig. 3.8 muestra el diagrama de entrada y salida para el bloque de **Memoria de Datos Procesados**, y una descripción de cada entrada se muestra en la Tabla 3.3. Esta memoria es una memoria tipo FIFO, modificada para no tener banderas de memoria vacía y memorias llena, tiene la capacidad de ser leída cuantas veces se requiera, de esta manera, **PG2** podrá leer esta memoria para cada fila que deba procesar. También con la entrada Limite_Memoria, dada por la salida Duracion_de_PP de **PG1**, controla cuantas localidades pueden ser escritas o leídas, así, el **PG1** solo escribe los elementos no cero de la fila procesada, y el**PG2** puede leer estos elementos las veces que lo requiera.

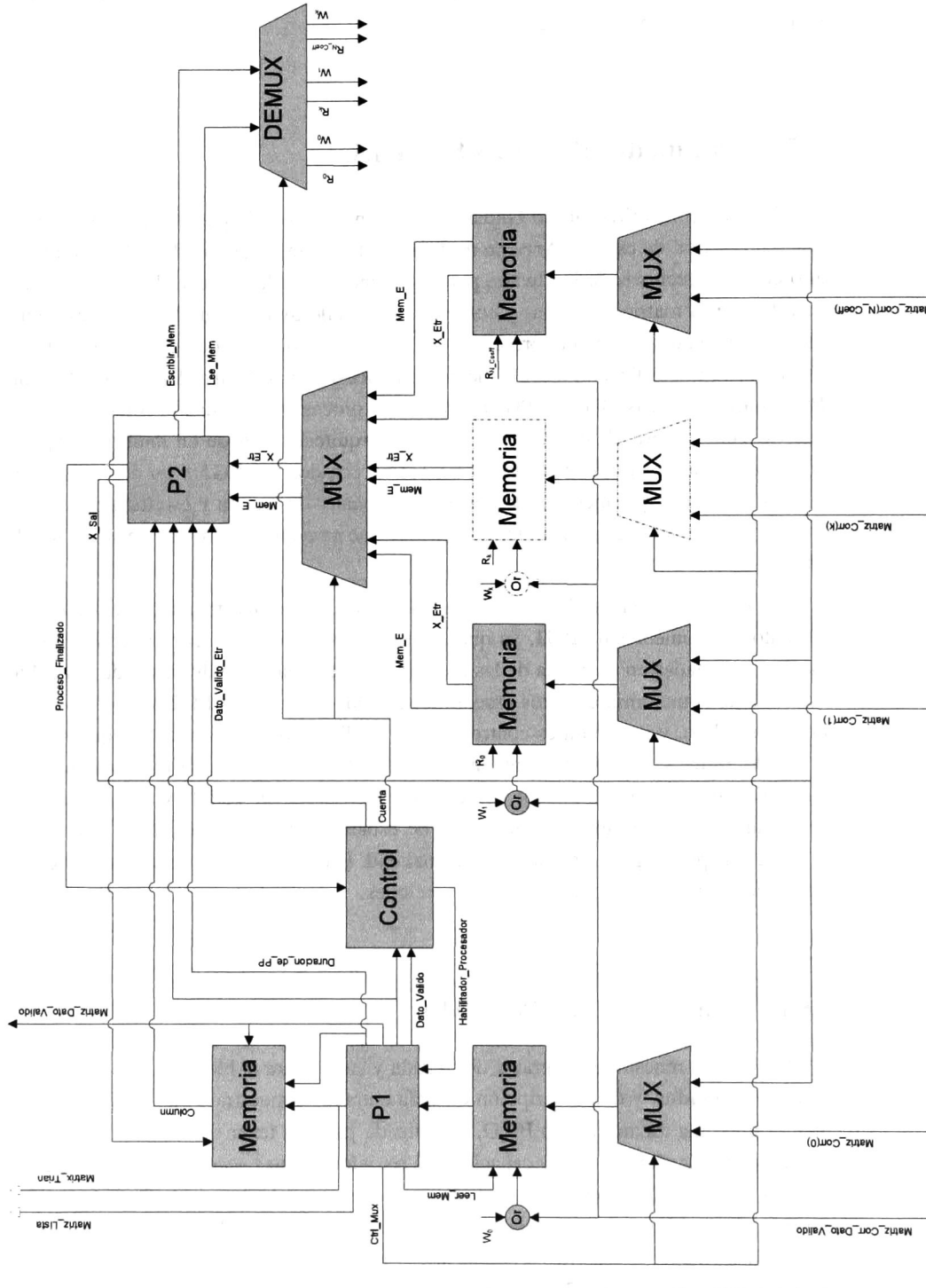


Figura 3.7: Arquitectura del módulo Arreglo de Procesadores de Gauss Reducido

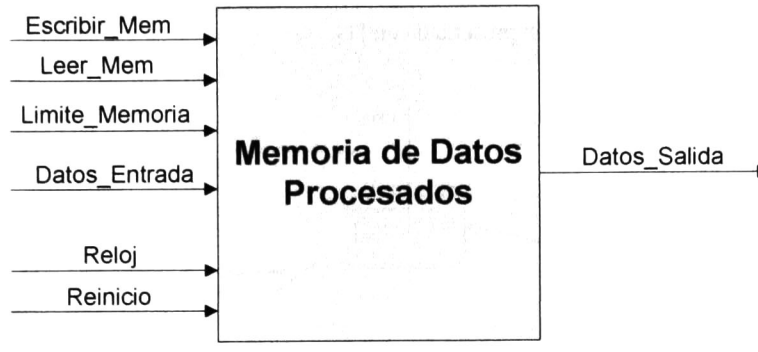


Figura 3.8: Diagrama de señales de entrada y salida para la Memoria de Datos Procesados

Tabla 3.3: Descripción de señales de entrada y salida para la Memoria de Datos Procesados

Entradas	**Descripción**
Escribir_Mem	Señal de escritura de la memoria, se escribe cuando es igual a 1.
Leer_Mem	Señal de lectura de la memoria, se escribe cuando es igual a 1.
Limite_Memoria	Señal para indicar el límite de localidades que pueden ser guardadas en la memoria.
Datos_Entrada	Datos de entrada de la memoria.
Salidas	**Descripción**
Datos_Salida	Datos de salida de la memoria.

3.5.2. Procesador de Gauss Tipo 1

En la arquitectura propuesta en [1] para el módulo de **Arreglo de Procesadores de Gauss**, se tienen $N-1$ bloques **PG2**, con esto el módulo **PG1** podía trabajar de forma continua, dado que el procesamiento del resto de las filas se realizaba en forma paralela. En la arquitectura mostrada en Fig. 3.7, es necesario que este bloque tenga una entrada que permita habilitar y deshabilitar este módulo, de esta manera, podrá procesar una fila de la matriz y esperar a que **PG2** termine de procesar el resto de las filas, antes de continuar con la siguiente. Para lograr esto solo la máquina de estado debe ser modificada. La Fig. 3.9 tomada de [1] muestra la máquina de estados utilizada para el **PG1** a esta solo es necesario agregar una condición extra en el estado **S2**, que involucra a la nueva entrada agregada al bloque Habilitador_Procesador, quedando el diagrama de estados mostrado en Fig. 3.10. Con esta modificación el módulo tendrá que esperar a estar habilitado cada

vez que se desee procesar una nueva fila. La función de cada estado no fue modificada quedando la misma descripción presentada en [1].

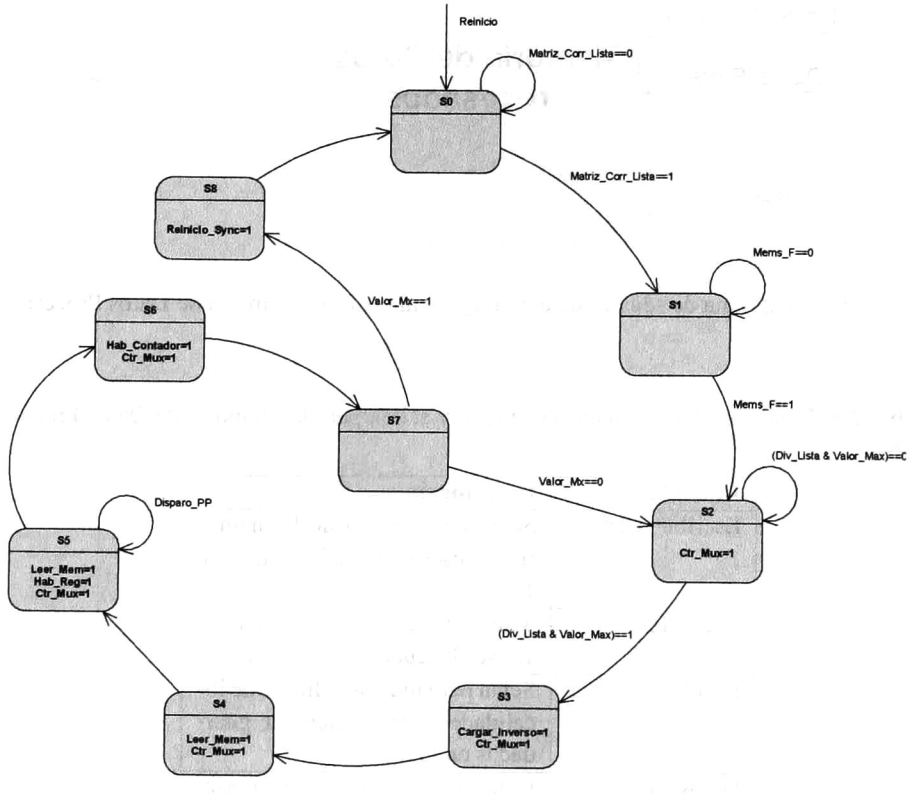


Figura 3.9: Diagrama de transición de estados para el Procesador de Gauss Tipo 1

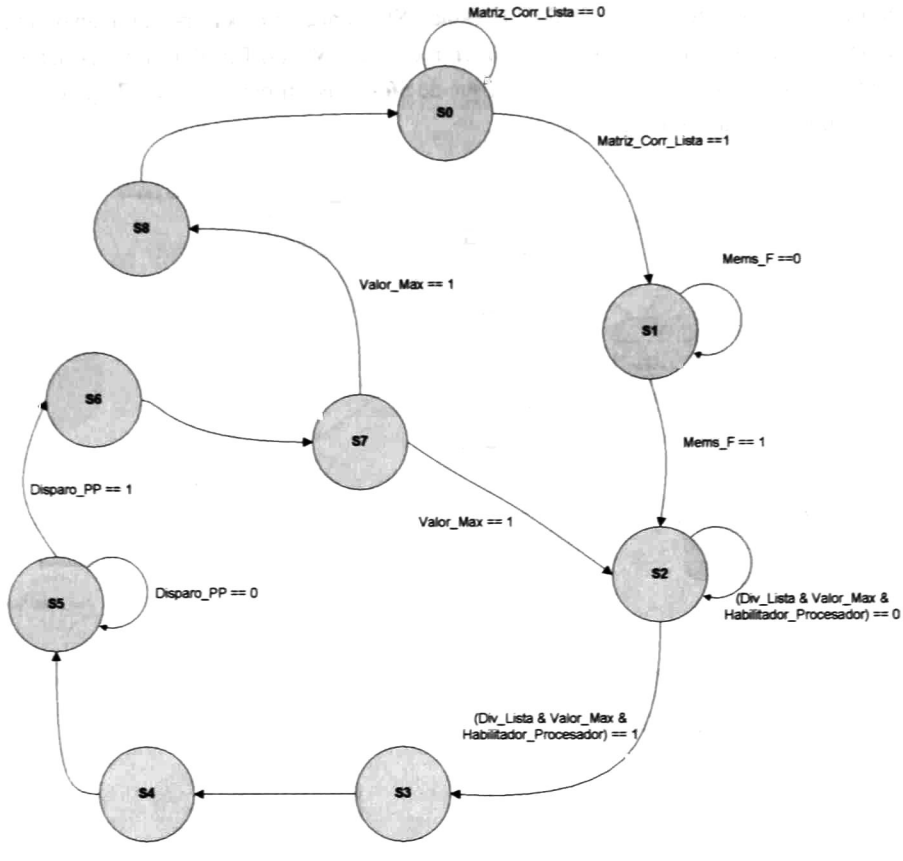


Figura 3.10: Diagrama de estados modificado para el Procesador de Gauss Tipo 1

3.5.3. Procesador de Gauss Tipo 2

Para poder realizar el control del multiplexor y demultiplexor en la Fig. 3.7 y determinar que memoria debe ser leída y escrita en cada tiempo; es necesario, que el **Procesador Tipo 2** pueda informar al control cuando a terminado de procesar una memoria. Para ello se agrega al procesador la señal Proceso_Finalizado, esta bandera se colocara en alto cada vez que se termina de procesar una fila de la matriz. Esta salida debe ser agregada a la máquina de control mostrada en sección 4.2.2.11 de [1] cuyo diagrama de estados se muestra en la Fig. 3.11. Esta salida tomara el valor de 1 en el estado **S8**, que es donde se a terminado de procesar la fila de la matriz, así el bloque de control puede cambiar el selector del multiplexor para procesar la siguiente fila.

Este procesador también debe de poder repetir su ciclo de procesamiento para el resto de las filas. Esto puede ser fácilmente logrado, usando la señal de Dato_Valido_Etr, con la cual, se da inicio al ciclo de procesamiento, que comprende los estado desde el **S1** al **S8**. Como los datos procesados por **PG1** son guardados en una memoria, siempre están

listo para ser utilizados, por lo cual en el estado **S6**, donde se espera por un nuevo dato del **GP1**, no es necesario que se utilice la entrada Dato_Valido_Etr. Por tanto, podemos modificar las transiciones de estados para que de **S6** se pase directamente a **S7**, quedando el diagrama mostrado en Fig. 3.12.

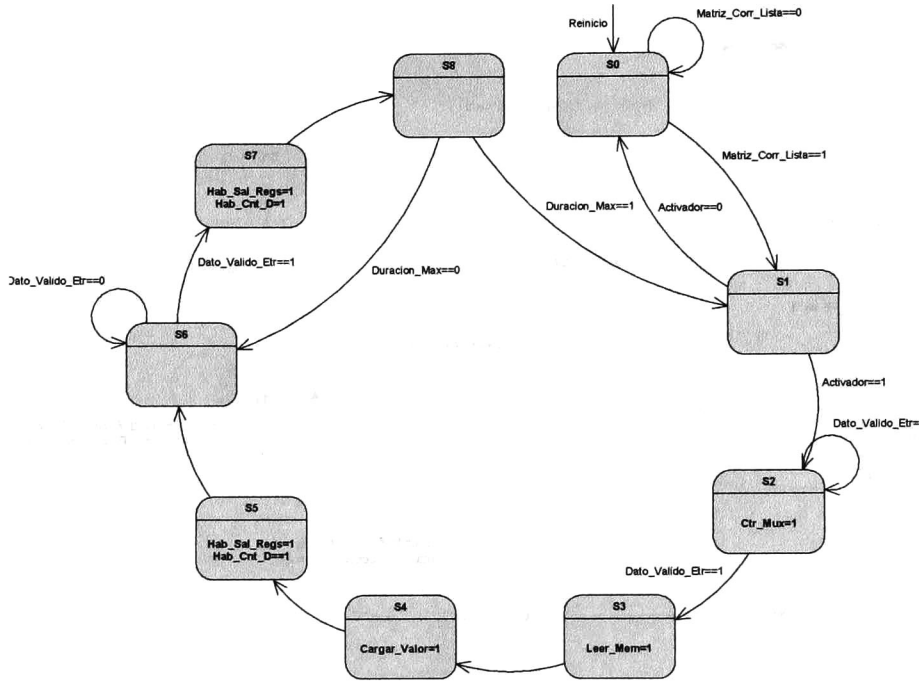


Figura 3.11: Diagrama de transición de estados para el Procesador de Gauss Tipo 2

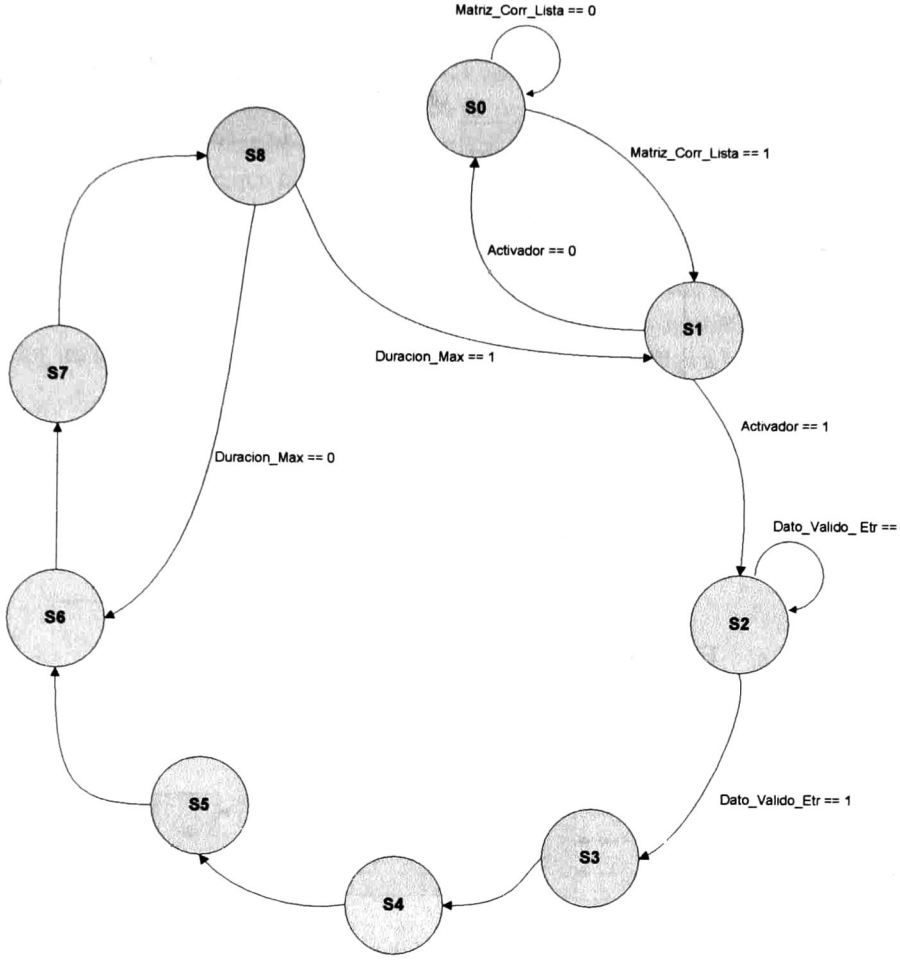


Figura 3.12: Diagrama de estados modificado para el Procesador de Gauss Tipo 2

3.5.4. Control

El módulo de control se muestra en la Fig. 3.13, este bloque es el encargado de controlar el multiplexor y demultiplexor agregados, así como determinar en qué momento los procesadores deben de funcionar. El control es realizado por un bloque contador ascendente, que tiene una entrada para determinar el límite de la cuenta, esta es tomada de la salida Numero_PP del bloque **PG1**; así, según el proceso parcial que se esté realizando se puede determinar cuántas memorias deben de ser procesadas. Usando la salida Cuenta de este bloque es posible controlar el multipexor y el demultiplexor. El bloque contador es habilitado por la salida de Proceso_Finalizado agregada al bloque **PG1**, así cada vez que se termina de procesar una fila el contador se incrementa, entonces el multiplexor y demultiplexor cambian su salida para procesar la siguiente fila. La habilitación para

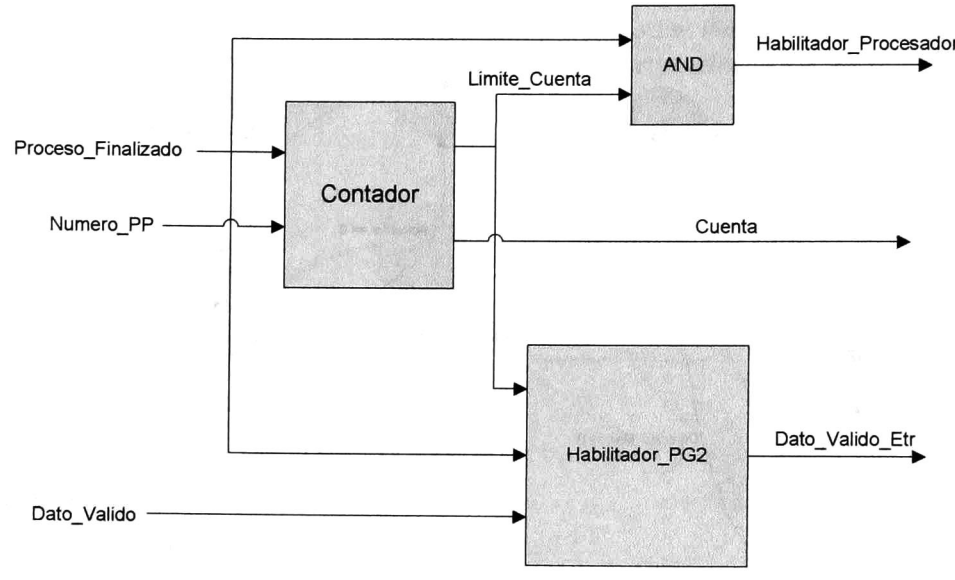


Figura 3.13: Arquitectura para el bloque Control

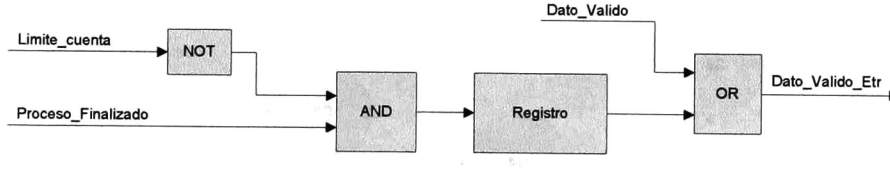


Figura 3.14: Arquitectura para el bloque Habilitador_PG2

PG1 se hace con la salida Habilitador_Procesador, la cual es una operación AND con la entrada Proceso_Finalizado y Limite_Cuenta, esto activa **PG1** cuando se termina de procesar la última fila en **PG2**.

La habilitación para **PG2** se realiza con el módulo **Habilitador_PG2** que se muestra en la Fig. 3.14, inicialmente Dato_Valido_Etr = 0, por lo que **PG2** no está trabajando; la activación de **PG2** se puede realizar por dos caminos. El primero, cuando **PG1** termina de procesar una fila activa la señal de Dato_Valido con lo cual Dato_Valido_Etr = 1, así **PG2** procesara la fila siguiente. Como **PG2** tiene que procesar más de una fila, también se puede activar usando la señal de Proceso_Finalizado=1 de **PG2**, siempre y cuando no se trate de la última fila a procesar, esto lo indica la señal Limite_de_Cuenta, ya que después de terminada la última fila a procesar se debe de esperar a que **PG1** procese una nueva fila.

3.5.5. Division

Según se muestra en [1] el módulo **PG1** cuenta con una bloque divisor basado en CORDIC, el cual calcula el reciproco de un número, esto sirve para el cálculo de los pivotes de la matriz aumentada. La implementación para este bloque divisor fue diseñada como pipeline, por lo que tiene un consumo excesivo de recursos, por tal razón se modifica por un arquitectura basada en CORDIC, pero en una forma secuencial, así solo se requiere de un solo módulo para hacer la iteración de CORDIC. La Fig. 3.15 tomada de [1], muestra la estructura del módulo Division. Este se compone de una módulo **Dispara**, el cual convierte la señal de Escribir_Mem en un pulso de la duración de una ciclo de reloj, así se empieza a hacer la división cuando el primer dato llega a las memorias.

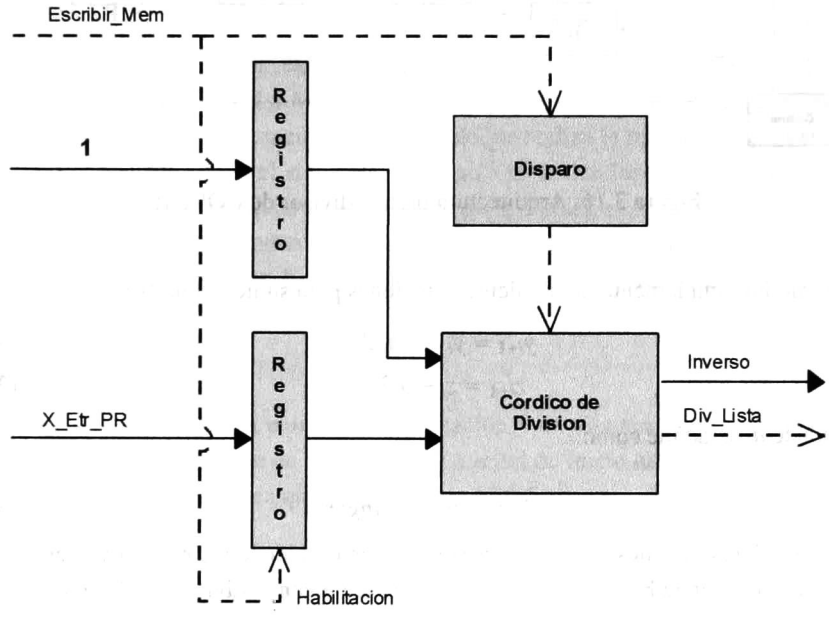


Figura 3.15: Arquitectura para el divisor en [1]

El módulo **Cordico de Division** es el que realiza la división en base a al algoritmo de CORDIC [8]. Este módulo cuenta con tres entradas una para el Divisor, otra para el Dividendo, el cual es siempre uno, y una más de Inicio para indicar el comienzo del cálculo, como salidas tiene Inverso, que es donde se da el resultado del cálculo, y Div_Lista donde se indica con un pulso en uno cuando el resultado es válido.

La Fig. 3.16 muestra la nueva arquitectura para el bloque **Cordico de Division**, se deja la misma interfaz que el bloque anterior para solo tener que remplazar el módulo y no cambiar nada en el control del procesador.

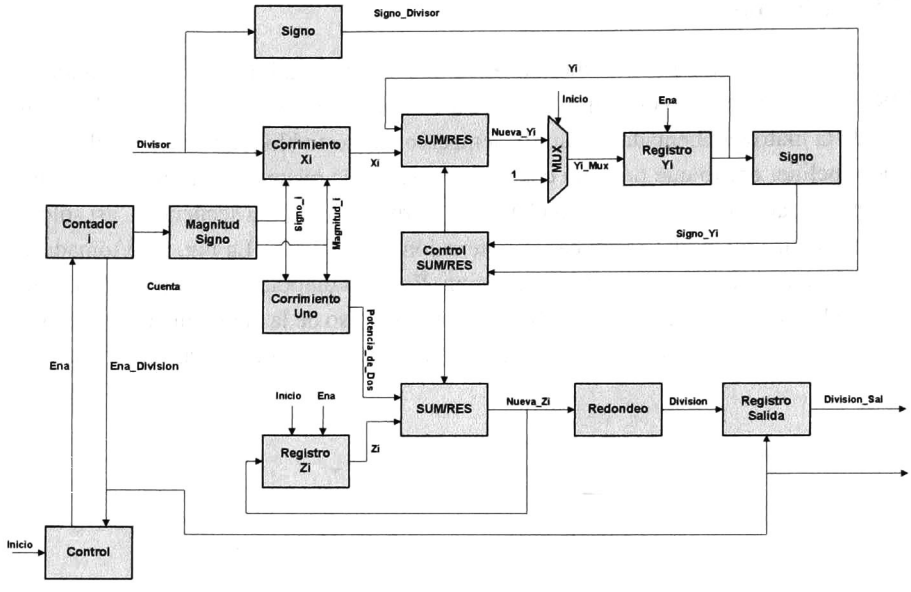


Figura 3.16: Arquitectura para el divisor de CORDIC

El módulo implementa las siguientes funciones para su iteración [9]:

$$y_{i+1} = y_i + x_i d_i 2^i \tag{3.1}$$

$$z_{i+1} = z_i - d_i 2^{-i} \tag{3.2}$$

Donde d_i se define como:

$$d_i = -sign(y_i) sing(x_i) \tag{3.3}$$

y los valores iniciales en x_0, y_0 y z_0 son divisor, dividendo y 0 respectivamente. Como se ve en la Fig. 3.16 los bloques que componen el divisor son los siguientes:

- **Contador i**

  Es un contador ascendente de números enteros con signo, diseñado para controlar la iteración que se está realizando. La descripción de las entradas y salidas del bloque se muestra en la Tabla 3.4. Los límites de conteo para el bloque se definen como un parámetro en base al formato de palabra y el número de iteraciones que se desean realizar.

- **Magnitud Signo**

  Este bloque convierte la salida cuenta del contador de un formato complemento a dos a un formato magnitud y signo. Tiene dos salidas Magnitud_i que es la magnitud de cuenta y Signo_i que es signo de cuenta, es decir, si es positivo es 0 y si es

Tabla 3.4: Descripción de señales de entrada y salida para Contador i

Entradas	**Descripción**
Ena	Habilitador de contador.
Salidas	**Descripción**
Cuenta	Valor de la cuenta del contador.
Ena_Division	Señal que indica cuando el valor llega a su cuenta máxima.

negativo es 1. Estas salidas son usadas para controlar los bloques de **Corrimiento X_i** y **Corrimiento Uno**.

- **Corrimiento X_i y Corrimiento Uno**

  Este bloque se encarga realizar un corrimiento a la entrada Divisor, dependiendo del valor de Signo_i y Magnitud_i, dado que un corrimiento de bits es igual a multiplicar por una potencia de dos, este bloque realiza la operación $x_0 2^{-}i$ en 3.1. Su funcionamiento es el siguiente, si Signo_i es 1, desplazara la entrada divisor Magnitud_i bits a la izquierda, de lo contrario la entrada es desplazada Magnitud_i bits a la derecha. El bloque **Corrimiento Uno** es idéntico al bloque **Corrimiento X_i** solo que la entrada es 1, así este bloque realiza la operación $2^{-}i$ en 3.2.

- **Registro Zi**

  Es un registro con una entrada de habilitación y un reinicio síncrono, dadas por las señales Ena y Inicio en la Fig. 3.16. La señal de Inicio hace que el registro se reinicie en cero cada vez que se termina un cálculo.

- **Registro Yi y Registro Salida**

  Son simples registros con entrada de habilitación.

- **Signo**

  Este bloque solo toma el signo del dato de entrada y lo da como salida, es decir la salida del bloque es el bit más significativo del dato de entrada.

- **SUM/RES 1 y 2**

  Es un bloque que puede ser configurado como sumador o restador según su entrada de control.

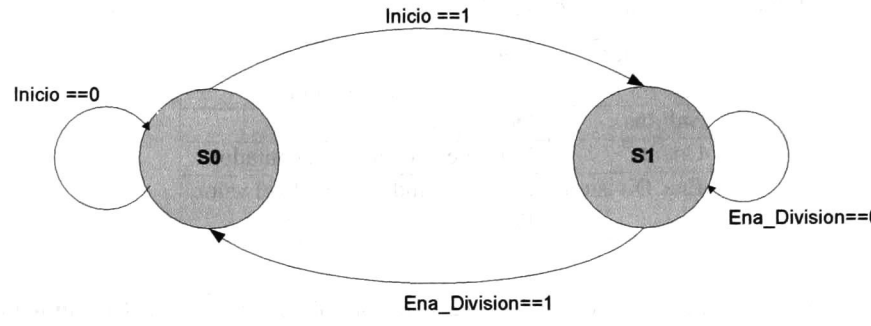


Figura 3.17: Transiciones de estados para el bloque de control de la división

- **Control SUM/RES**

  Este bloque toma la decisión si los bloques **SUM/RES 1** y **SUM/RES 2** deben de operar como sumadores o restadores. Es decir este bloque realiza la operación en 3.3, así, si d_i es positiva SUM/RES 1 debe sumar y SUM/RES 2 debe restar, de lo contrario SUM/RES 1 resta y SUM/RES 1 suma.

- **Redondeo**

  Realiza un redondeo al valor más cercano del dato obtenido de **SUM/RES 2**, con tamaño de palabra de 32 bits, a un dato de salida con tamaño palabra 16 bits.

- **Control**

  Este bloque es una máquina de estados, para controlar cuando habilitar el bloque **Contador i**. La Fig. 3.17 muestra la transición de estados para el bloque Control. En el estado **S0**, el **Contador i** no está funcionando y en **S1**, se habilita el contador.

Este bloque divisor se integra al **Procesador de Gauss Tipo 1** sin tener que realizar ningún cambio extra.

3.6. Eliminador de Gauss Secuencial Configurable

El número de coeficientes a calcular determina el tamaño de la matriz a procesar por el **Arreglo de Procesador de Gauss**, por lo cual para una implementación cuyo número de coeficientes sea configurable es necesario modificar tanto el **Alimentador de Gauss** como el **Arreglo de Procesadores**. Estas modificaciones se muestran las siguientes secciones.

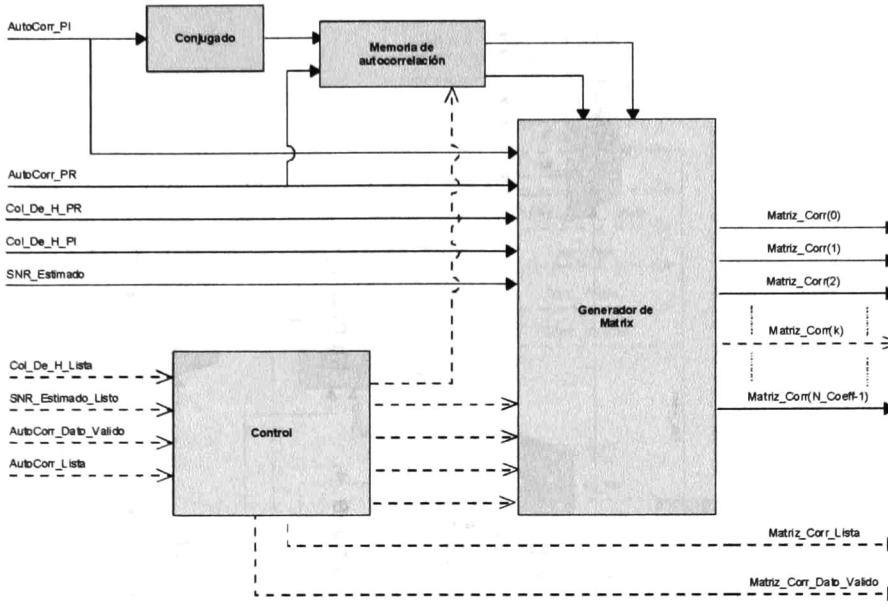


Figura 3.18: Arquitectura para el bloque Alimentador de Gauss_PG2

3.6.1. Alimentador

El **Alimentador de Gauss** es el módulo encargado de formar la matriz aumentada, compuesta por la matriz de correlación del canal y la columna *h*. La Fig. 3.18 tomada de [1] muestra la arquitectura interna para este bloque. Las partes más importantes son la **Memoria de autocorrelación**, la cual guarda el conjugado del vector de autocorrelación; el bloque **Generador de Matriz**, el cual se compone de una serie de registros que son utilizados para guardar en el orden correcto los datos en las **memorias de Gauss**, y el bloque de **control** es el encargado de dar la habilitación al resto de los bloques y sincronizar el módulo de la manera correcta. Para realizar la adaptación para un número de coeficientes configurables, solo la **Memoria de autocorrelación** debe de ser modificada, agregando una entrada que permita modificar su tamaño, es decir, cuantas localidades puede guardar. Esta nueva entrada será dada por la señal Num_Coeficientes, así, el control no tendrá que ser modificado.

3.6.2. Arreglo de Procesador de Gauss

La arquitectura del arreglo de procesadores debe ser modificada si se quiere un número de coeficientes configurables, esta nueva arquitectura se muestra en la Fig. 3.19. Se observa que las **Memorias de Gauss** y el **PG1** deben de ser modificados, así como la agregación de un nuevo bloque el **Bloqueador**. Estas modificaciones se presentan en las siguientes secciones.

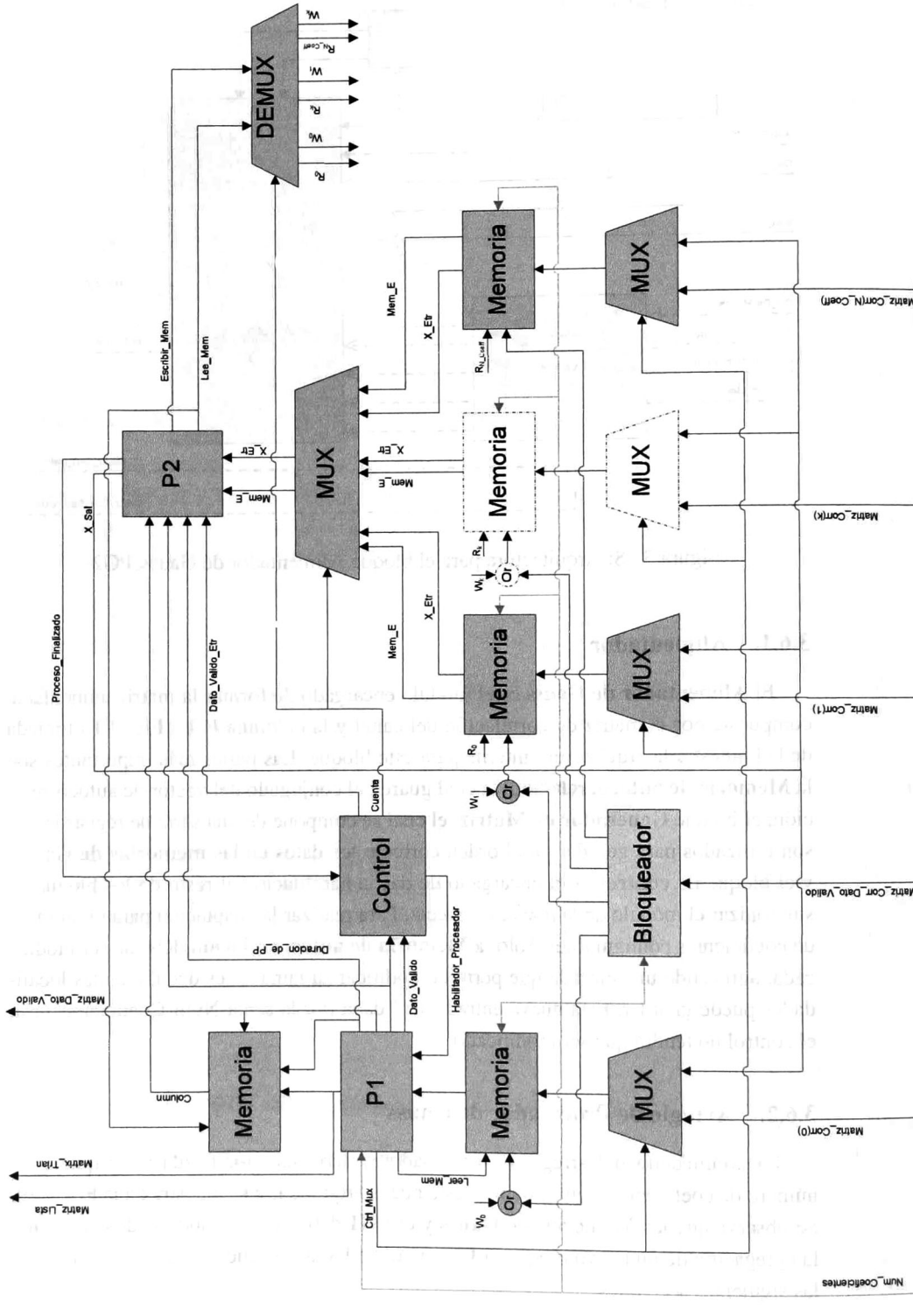


Figura 3.19: Arquitectura para el bloque Arreglo de Procesadores de Gauss Configurable_PG2

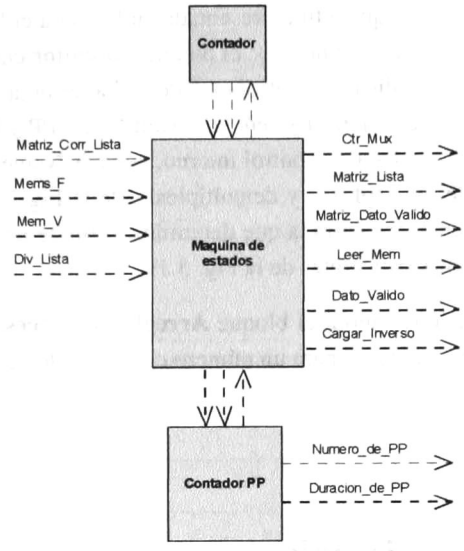


Figura 3.20: Arquitectura para el bloque Control de **PG1**

3.6.2.1. Memorias de Gauss

Se agrega a la memoria original una entrada para determinar el tamaño de la memoria, es decir, el número de localidades que podrá guardar, haciendo esto se logra que el control de los procesadores no deba ser modificado ya que depende de las señales de memoria vacía y memoria llena. El resto de las entradas y salidas no se modifica y permanecen con la misma función.

3.6.2.2. Bloqueador

Este bloque se encarga de decidir a cuantas memorias debe ser escritas por el alimentador de Gauss, esto dependiendo de la entrada número de coeficientes, es decir, si existen N memorias solo las primeras Num_Coeficientes memorias recibirán la entrada Matriz_Corr_Dato_Valido, la cual controla la escritura de las memorias por el **Alimentador de Gauss**. Así, solo se trabajara con las memorias que se necesite cuando la entrada Num_Coeficeintes cambie.

3.6.2.3. Procesador de Gauss tipo 1

Dado que este procesador es el encargado de determinar el proceso parcial que se está realizando, así como su duración, es decir, el número de datos que se procesan en la fila, es necesario que los bloques que calculen estos valores sean modificados, esto lo realiza el bloque de control dentro de **PG1**.

La Fig. 3.20 muestra la arquitectura presentada en [1] para el bloque de control, se observa que se compone de dos contadores, el bloque **Contador** cuenta hasta el número de procesos parciales para indicar al control cuando se ha termina el procesamiento. El **Contador PP** se encarga de generar las señal de Numero_de_PP y Duracion_de_PP que son usadas por **PG2** para realizar su control interno, además Numero_de_PP también es usada para el control del multiplexor y demultiplexor de la Fig. 3.7. Estos contadores son modificados para tener una entrada que determina el limite de conteo, esta entrada es dada por la señal Num_Coeficeintes de la Fig. 3.19.

Con estas modificaciones ahora el bloque **Arreglo de Procesadores de Gauss** es capaz de realizar el procesamiento para un número configurable de coeficientes.

3.7. Sustitución hacia atrás

El módulo de sustitución hacia atrás es el encargado de tomar la matriz triangular del bloque Eliminación de Gauss y calcular los coeficientes para el filtro igualador de canal. Este bloque se forma por dos submódulos el **Alimentador** y el **Arreglo de procesadores** mostrados en la Fig. 3.21 tomada de [1].

El **Alimentador** es el encargado del correcto almacenamiento de los renglones de la matriz triangular dados por el **Eliminador de Gauss**. Como se muestra en la Fig. 3.22 este bloque está formado principalmente por comparadores, que usan la salida del **Contador de Renglón** para determinar cuál renglón se está escribiendo, este contador también determina cuantos datos deben de escribirse en cada fila, ya que solo se escriben los datos no cero de la matriz triangular, y el bloque contador sirve a la máquina de estados para determinar cuándo se han escrito todos los renglones.

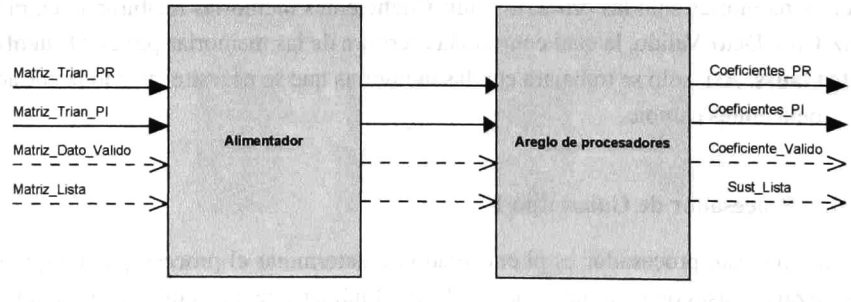


Figura 3.21: Arquitectura para el bloque Sustitución hacia Atrás

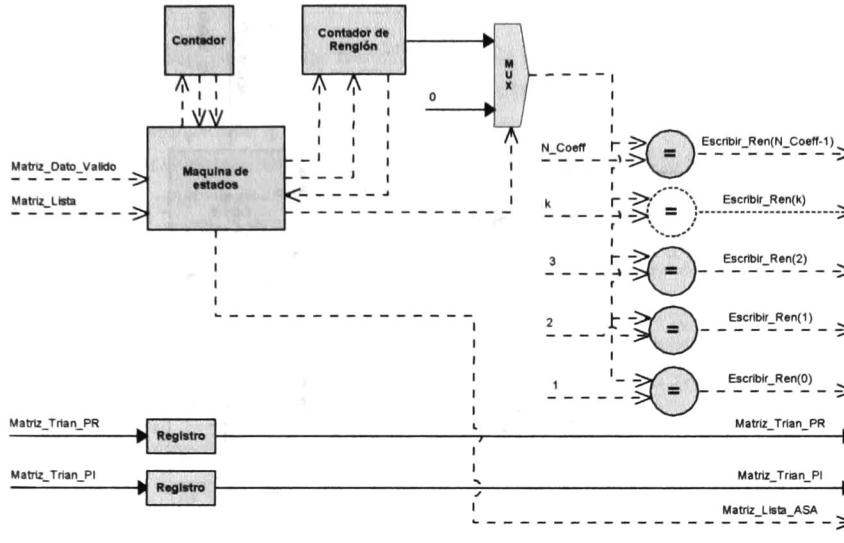


Figura 3.22: Arquitectura para el bloque Alimentador de Sustitución hacia Atrás

El bloque **Arreglo de procesadores** es el encargado de realizar la sustitución hacia atrás para encontrar los valores de los coeficientes; su arquitectura es mostrada en la Fig. 3.23, se observa que solo está formado por memorias y los procesadores. Existen tantas memorias como número de coeficientes se deseen calcular, esta memorias son escritas por el bloque **Alimentador** y son leídas por cada procesador. Cada procesador tiene su propio control para la lectura de las memorias y la transferencia de los datos al siguiente procesador. Existen dos tipos de procesadores el tipo 1, por ser el primer procesador no tiene que realizar ningún procesamiento dado que el último renglón ya tiene el resultado para el primer coeficiente, los procesadores tipo 2 usan el valor calculado del procesador anterior para realizar la sustitución hacia atrás y resolver las ecuaciones.

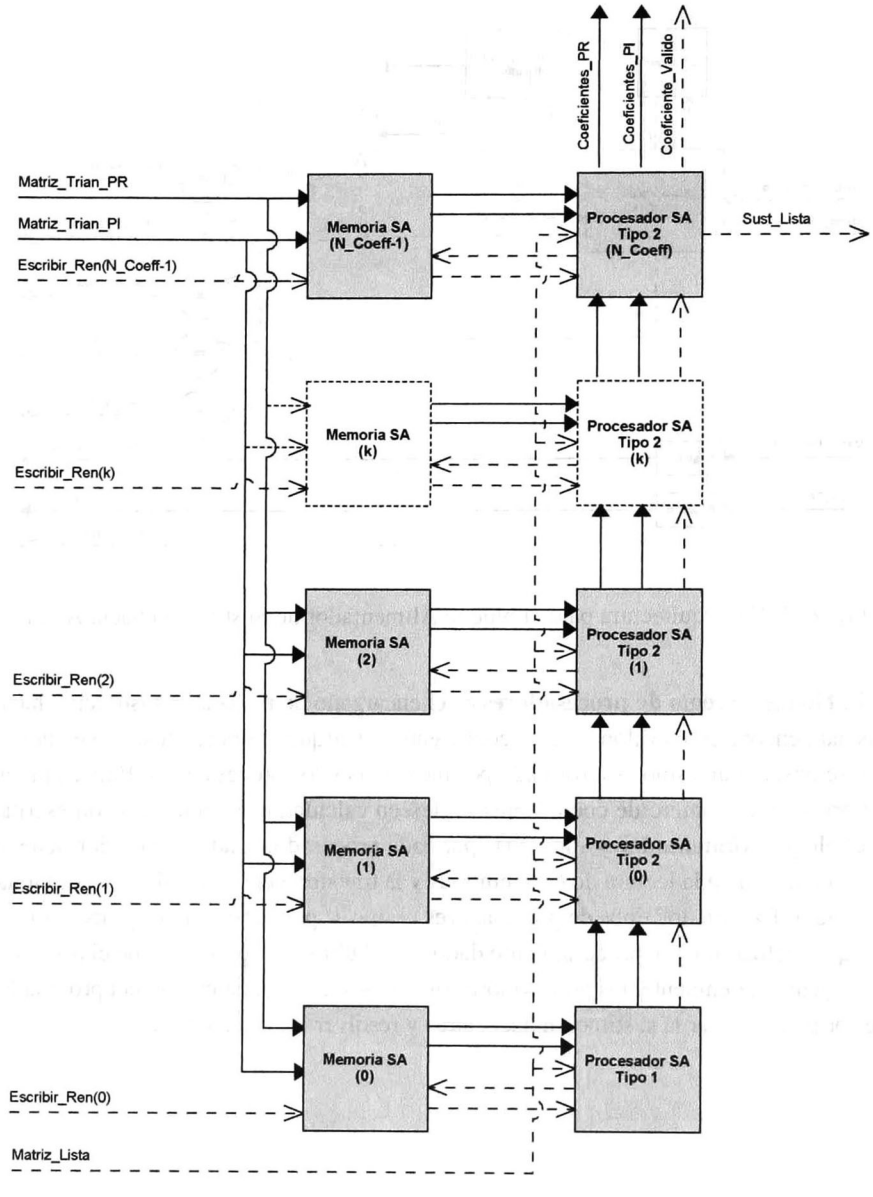


Figura 3.23: Arquitectura para el bloque **Arreglo de Procesadores** de Sustitución hacia Atrás

3.8. Sustitución hacia atrás Secuencial

En base al módulo Sustitución hacia Atrás de la sección anterior, se observa que el módulo que realiza el procesamiento es el **Procesador Tipo 2**, cuyas replicas podemos reducir para disminuir el uso de recursos de hardware. La nueva arquitectura se muestra en la Fig. 3.24. En esta arquitectura se usa el mismo **Procesador Tipo 2** con algunas modificaciones para realizar el procesamiento, se siguen conservando todas las memorias que almacenan la matriz triangular, solo que se añade un multiplexor y un demultiplexor, los cuales determina cual memoria es la que debe de procesarse, así el procesador podrá leer y recibir las señales de la memoria correcta. También se agregó una nueva memoria de datos de salida, esta memoria guarda los datos de salida procesados, que son los coeficientes calculados por el procesador, estos son usados para procesar la siguiente fila.

El bloque de Control se encarga de cambiar el selector del multiplexor y demultiplexor según la fila que se esté procesando; también controla la salida de Dato_Listo la cual se activa cuando el procesador está procesando la última fila y solo falta calcular el último coeficiente. Los cambios realizados a los bloques se muestran en las siguientes secciones.

3.8.1. Memoria de Salida

Es una simple memoria tipo FIFO sin salidas de memoria vacía y memoria llena, tiene una capacidad máxima de localidades del número de coeficientes. Las entradas y salidas de la memoria se muestra en la Tabla 3.5.

Tabla 3.5: Descripción de señales de entrada y salida para la Memoria de Salida

Entradas	**Descripción**
Escribir_Mem	Señal de escritura de la memoria activa en alto.
Leer_Mem	Señal de lectura de la memoria, se escribe cuando es igual a 1.
Datos_Entrada	Datos de entrada de la memoria.
Salidas	**Descripción**
Datos_Salida	Datos de salida de la memoria.

3.8.2. Procesador Tipo 2

Este bloque es el encargado de realizar la sustitución hacia atrás, para lograr que este bloque funcione para la nueva arquitectura es necesario que su máquina de control pueda ser reiniciada para procesar cada una de las filas. La Fig. 3.25 tomada de [1] presenta el bloque de control, el cual está constituido por dos bloques, un contador el cual determina cuantos datos deben de procesarse en cada fila y la máquina de estados la cual controla

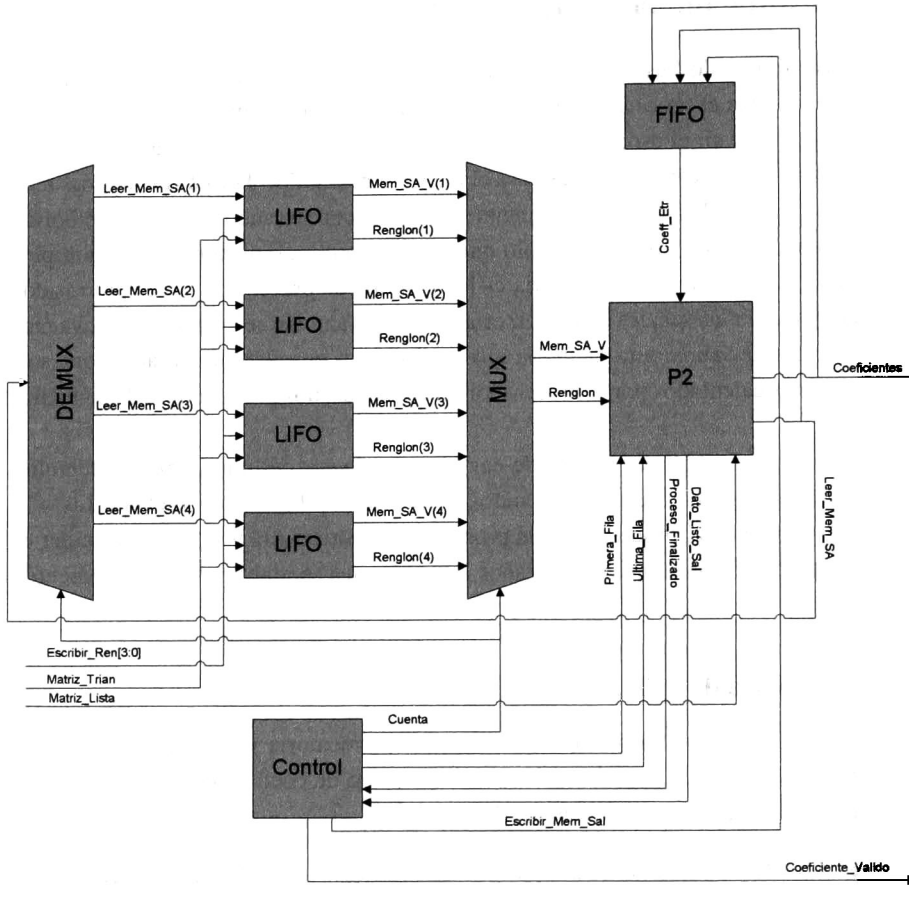


Figura 3.24: Arquitectura para el bloque **Sustitución hacia Atrás**

el bloque procesador, además de la lectura y escritura de las memorias asociadas al procesador. Para lograr la modificación es necesario que el contador tenga un límite de conteo diferente para cada fila. Para la primera fila no se debe procesar ningún dato, dado que el dato guardado es directamente el coeficiente esperado, en la segunda fila se debe de procesar un dato y así sucesivamente en cada fila se va añadiendo un dato más. La nueva entrada para el límite de conteo será dada por el bloque de **Control** en la Fig. 3.24, siendo la misma que se usa como selector para el multiplexor.

También se modifica la máquina de estados propuesta en [1] que se muestra en la Fig. 3.26; se ve que el procesamiento inicia después de que llega la señal de Matriz_lista y dura desde el estado **S1** hasta el estado **S11**. Para hacer que este proceso pueda repetirse para cada fila es necesario que exista un estado, **S12**, en el cual se pregunte si se ha terminado de procesar todas la filas, si no es así se debe regresar al estado **S1** e iniciar

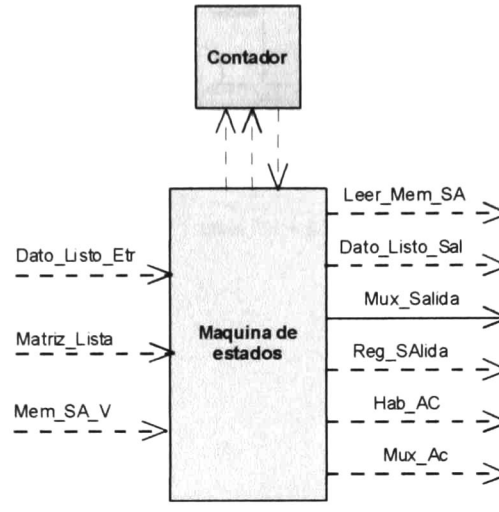


Figura 3.25: Arquitectura para el bloque de **Control** dentro del **Procesador Tipo 2** de Sustitución hacia Atrás

un nuevo ciclo de procesamiento. En este nuevo estado, también se enciende la salida Proceso_Finalizado agregada a la máquina de estados para avisar al bloque de Control en Fig. 3.24 que se ha terminado de procesar una fila. Como este nuevo procesador también debe de sustituir al **Procesador Tipo 1**, el cual no realiza ningún procesamiento, es necesario que el nuevo procesador pueda solo leer la memoria cuando se esté procesando la primer fila. Esto se logra agregando una nueva transición del estado **S3**, donde ya se tiene leído el valor la memoria, al estado **S9**, donde se habilita la salida de datos. Las modificaciones a la máquina de estados se muestran en la Fig. 3.27, se observa que la transición en el estado **S4** fue cambiada, esto debido a que en ese estado se esperaba a recibir un dato del procesador previo, dado que los datos ya están en la memoria no es necesario esperar, por lo cual se hace un cambio directo al estado **S5**, todos los estados conservan la misma función descrita en [1]. Para poder realizar la lectura de la memoria de salida el procesador utiliza la misma señal de lectura dada a la **Memoria SA** y la escritura es controlada por el bloque de **Control** en la Fig. 3.24.

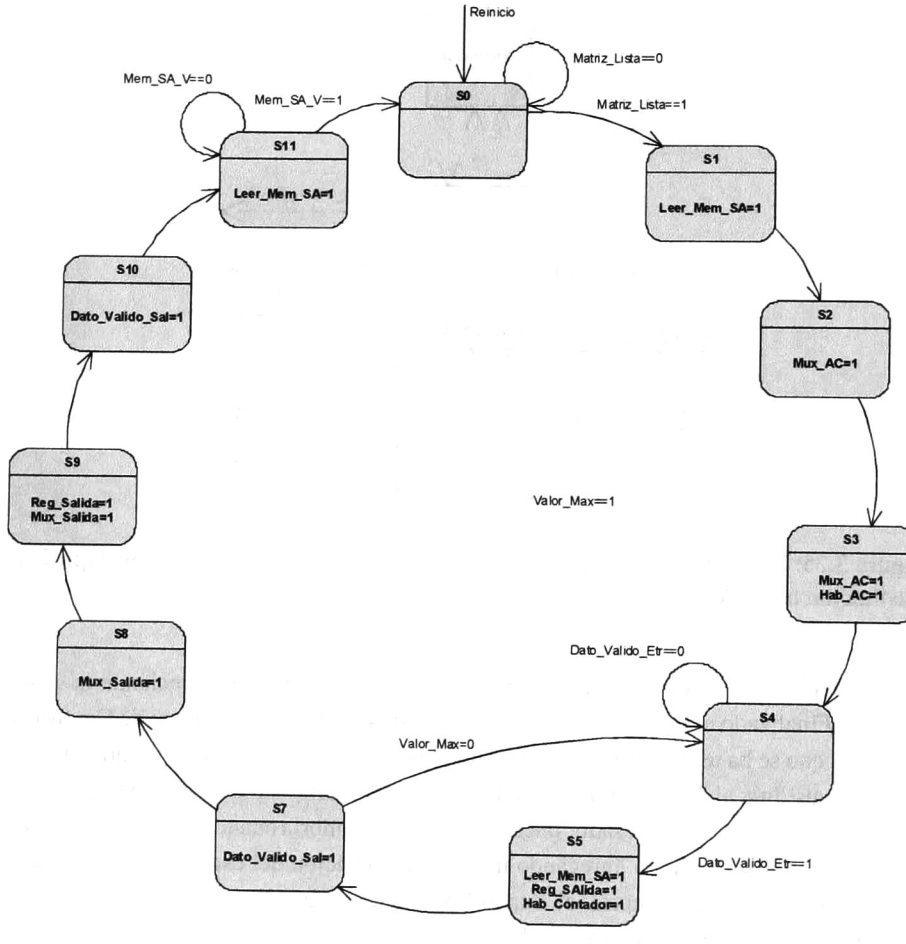


Figura 3.26: Diagrama de estados del **Procesador Tipo 2** de Sustitución hacia Atrás

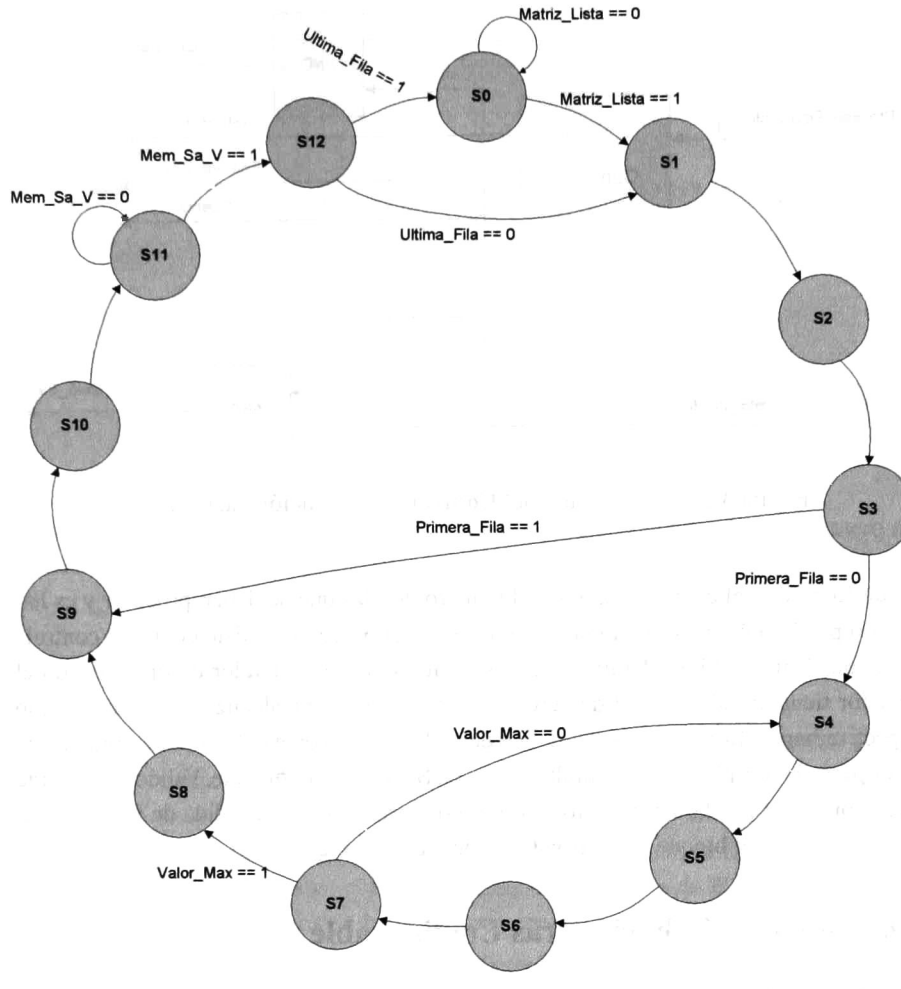


Figura 3.27: Diagrama de estados modificado del **Procesador Tipo 2** de Sustitución hacia Atrás

3.8.3. Control

El bloque de control es el encargado de controlar el multiplexor y el demultiplexor, así como de dar las señales de Primera_Fila y Ultima_Fila, las cuales se usan en la máquina de estado para el procesador, también controla la salida de Coeficiente_Valido, la cual se habilita cuando la señal Ultima_Fila y señal Dato_Listo_Sal dada por el procesador son válidas. Además, activa la escritura de la memoria de salida cuando Dato_Listo_Sal = 1 y Ultima_Fila = 0, con ello la memoria se escribirá cada vez que se procese una fila excepto en la última, dado que el procesador ya no necesitara calcular otro coeficiente, por lo que no es necesario guardar los datos. Este bloque está compuesto simplemente

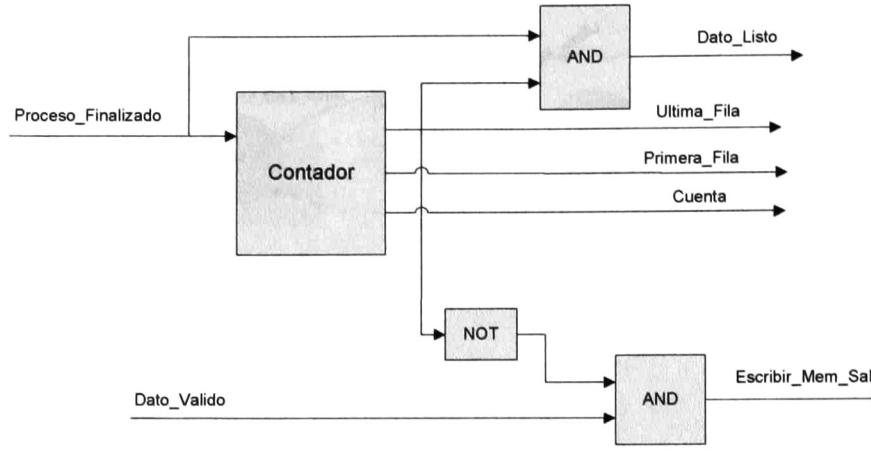


Figura 3.28: Arquitectura del **Control** de Sustitución hacia Atrás

por un Contador, el cual cuenta hasta el número de filas que se deben procesar y es habilitado por la señal Proceso_Finalizado dada por el procesador. Este contador controla las salidas Primera_Fila y Ultima_Fila, las cuales solo tiene el valor de uno, cuando el contador tiene el valor inicial de cero y cuando el contador alcanza su valor máximo respectivamente. También se usa la salida del valor de la cuenta del contador como el selector para el multiplexor y el demultiplexor. La Salida de Coeficiente_Valido es solo una operación AND con la Salida Dato_Listo_Sal del procesador y la salida de Ultima_Fila. La arquitectura del bloque de **Control** se muestra en la Fig. 3.28.

3.9. Sustitución hacia atrás Configurable

Para realizar un módulo configurable para el número de coeficientes que se van a calcular, es necesario que solo se procesen la cantidad de filas indicadas por la entrada Num_Coeficientes, así el único bloque responsable de decir el número de filas a procesar es el bloque de **Control** de la Fig. 3.24, por lo que solo este bloque debe ser modificado, más precisamente el contador en este bloque debe de tener una entrada Limite_Conteo que será dada por Num_Coeficientes, así ahora la señal de Ultima_Fila se activara en 1 cuando el contador llegue a Num_Coeficientes - 1. Solo con esta modificación tenemos un módulo configurable para el número de coeficientes a calcular. La Fig. 3.29 muestra la modificación al módulo de **Control**.

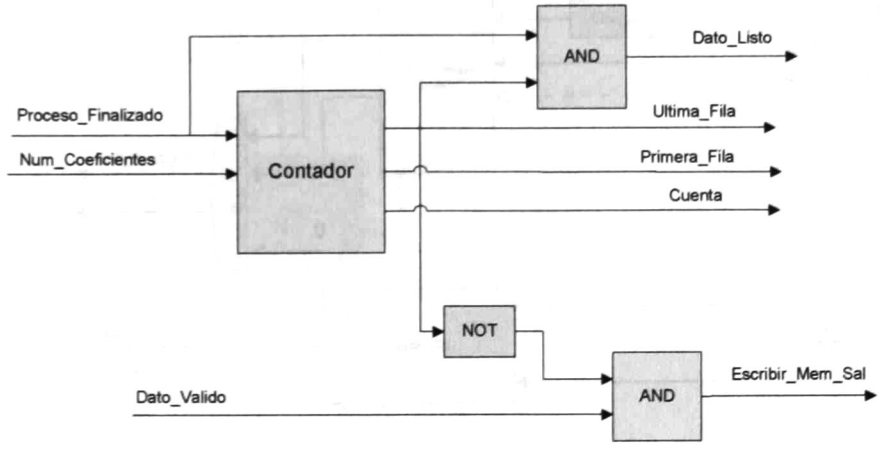


Figura 3.29: Arquitectura del **Control** modificado para ser configurable en número de coeficientes

3.10. Selector de Columna H

El módulo selector de columna es el encargado de seleccionar la columna de la matriz de canal correspondiente según el retardo de igualación y el número de coeficientes que se van a calcular. Este bloque que se muestra en la Fig. 3.30, basa su arquitectura en determinar el caso de la matriz en el cual se encuentra la columna de **H** a seleccionar, es decir, puede suceder tres casos; el primero, cuando la columna de **H** contiene primero ceros y después los elementos de **h**, el segundo cuando la matriz se forma solo de los elementos de **h** y no contiene ceros, y el tercero cuando los primeros forman parte de **h** y luego se concatenan ceros. Una explicación detallada de los casos y las condiciones en las cuales se cumplen se encuentra en la sección 3.5.1.6 de [1].

Dado que este módulo no realiza una gran cantidad de procesamiento y no requiere operaciones como multiplicaciones y divisiones, ni contiene réplicas de módulos como en caso del Autocorrelacionador, este módulo no es necesario reducirlo. Además, en [1] se hace una propuesta, configurable para el número de coeficientes, de este módulo. Tomando este diseño se puede obtener la implementación necesaria para la arquitectura configurable que se desea. La explicación detallada del diseño de este bloque se da en la sección 4.2.3 de [1].

3.11. Arquitectura MMSE configurable

Una vez que se tiene los módulos **Autocorrelacionador, Eliminación de Gauss, Sustitución hacia atrás y Selector de columna**, en su forma reducida y configurable,

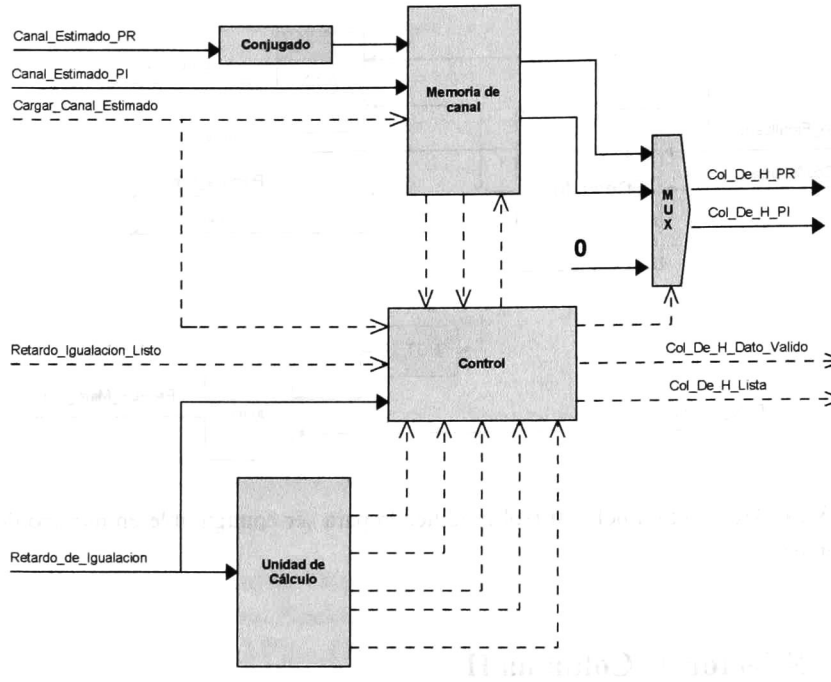


Figura 3.30: Arquitectura del módulo **Selector de Columna**

se unen en la nueva arquitectura MMSE configurable presentada en la Fig. 3.31. Esta conserva la misma interfaz presentada en [1] solo con la agregación de la entrada Num_Coeficientes, la cual se usa en los bloques internos para determinar el número de coeficientes que deben calcular. Como cada bloque tiene su propio control, sus señales de dato valido y módulo listo, no es necesario agregar otro bloque extra de control para la unión de los módulos, esto muestra que los bloques son fácilmente integrables en las arquitecturas en que se añaden. Así, se tiene una arquitectura configurable que reduce el uso de hardware y cuyos bloques pueden ser utilizados separadamente para el cálculo de otros algoritmos.

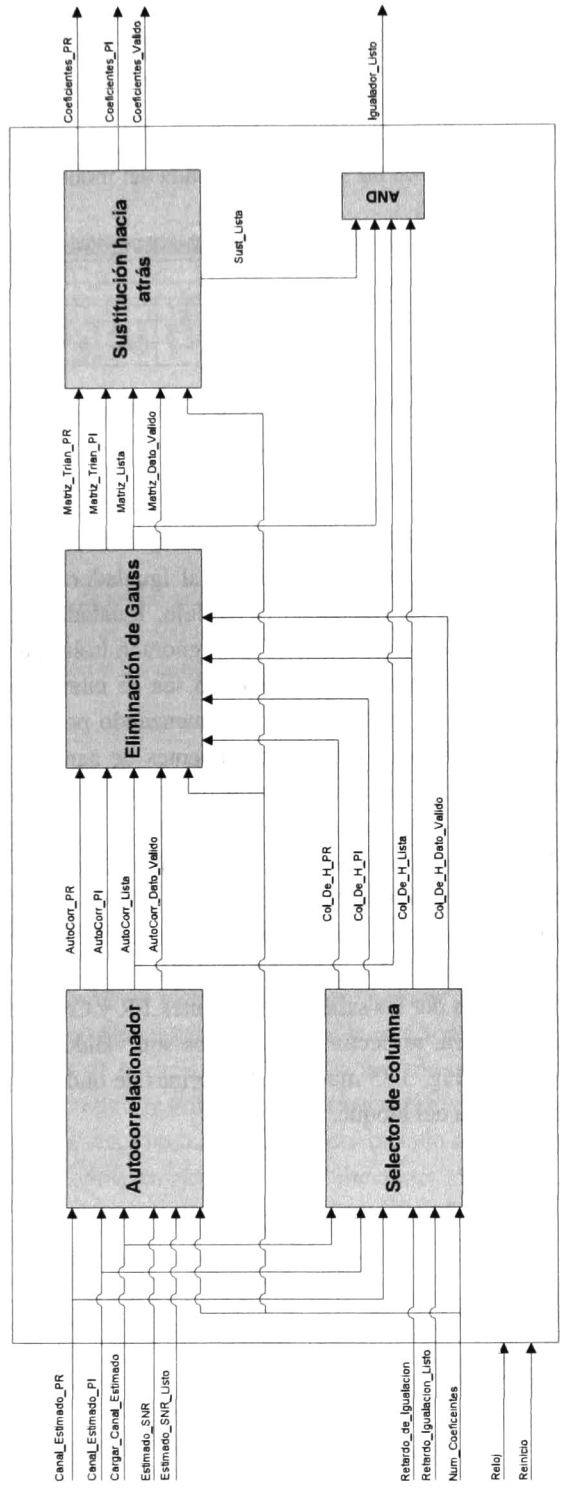


Figura 3.31: Arquitectura del módulo MMSE configurable

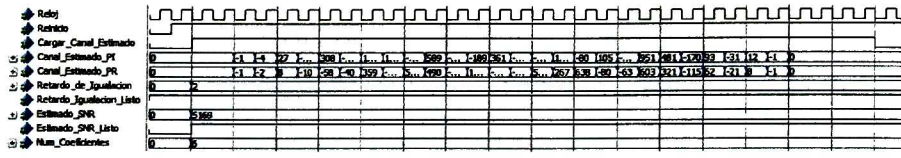


Figura 3.32: Ejemplo de las señales de entrada del módulo MMSE

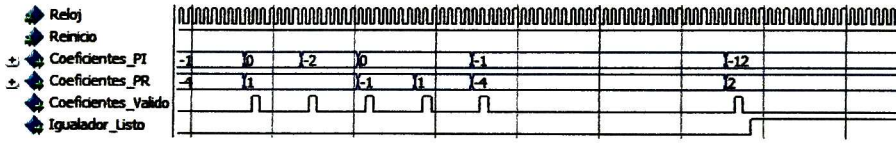


Figura 3.33: Ejemplo de las señales de salida del módulo MMSE

El módulo MMSE tiene como entradas el canal estimado, el estimado de la relación señal a ruido, el retardo de igualación y el número de coeficientes a calcular. Para poder comenzar la ejecución de este módulo la señal Igualador_Listo debe ser igual a 1; cuando se inicia la carga de valores en el módulo, Igualador_Listo cambia a 0, a partir de este momento toda nueva escritura será ignorada hasta que el módulo termine el cálculo de los coeficientes e Igualador_Listo sea de nuevo igual a 1. La carga del canal estimado se realiza en forma serial comenzando por el valor h(0) hasta el valor h(Coeff_Canal 1). La carga de los coeficientes de canal se realiza cuando Cargar_Canal_Estimado=1. Terminada la carga de todos los valores de **h** y si Retardo_De_Igualacion_Listo y SNR_Estimado_Listo son igual a 1, el módulo toma el valor del estimado de la relación señal y del retardo de igualación de las señales SNR_Estimado y Retardo_de_Igualacion respectivamente para comenzar el cálculo de los coeficientes. El valor en la entrada de Num_Coeficientes debe permanecer durante toda la ejecución del módulo es decir mientras la señal Igualador_Listo sea igual a 0. La salida de los coeficientes calculados se dan por las salidas Coeficientes_PR y Coeficientes_PI, siendo la parte real y parte imaginara, respectivamente. Estos son válidos cuando Coeficiente_Valido=1. Las Fig 3.32 y Fig. 3.33 muestran las formas de onda para las señales de entrada y las señales de salida del bloque.

Capítulo 4

Metodología de Verificación y Resultados

En este capítulo se describe la metodología de verificación utilizada, se muestran el plan de pruebas en base los objetivos y requerimientos de la tesis. Además de los resultados de implementación y las comparativas en recursos de hardware con la arquitectura en [1].

4.1. Metodología de Verificación

La metodología de verificación empleada fue usar simulación funcional del diseño para probar las características que nos interesan. Por medio de una cama de prueba podemos inyectar los datos necesarios al módulo y recibir los datos de salida para analizarlos.

Para realizar las pruebas en el diseño se utiliza el esquema mostrado en la Fig. 4.1. En este esquema, mediante un script en Matlab se generan los valores para las entradas de la arquitectura, como son el canal estimado, la relación señal a ruido, el retardo de igualación y el número de coeficientes. Estos valores son transformados a su representación en punto fijo, acorde a la utilizada en la arquitectura. Posteriormente, se crean los archivos para la simulación, los cuales serán leídos cuando se ejecute la simulación de la cama de prueba, este archivo es ejecutado en el simulador ModelSim. Mediante la cama

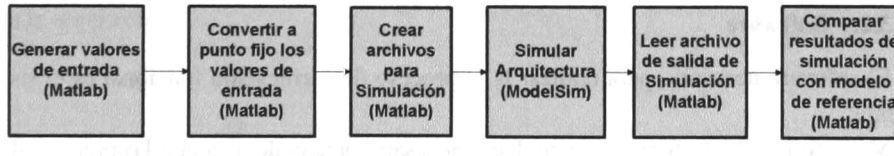


Figura 4.1: Esquema para pruebas del diseño

Tabla 4.1: Objetivos del Proyecto

Objetivo	Descripción
O1	Reducir el costo en términos de hardware con respecto a [1].
O2	Realizar una arquitectura configurable para el número de coeficientes a calcular.

de prueba se generan archivos de salida que contienen el valor de los coeficientes para el filtro igualador, esto archivos son leídos y comparados con el modelo de referencia creado en Matlab.

4.1.1. Objetivos del Proyecto

Objetivos del proyecto reflejan las metas principales del trabajo, estos se presentan en la Tabla 4.1.

4.1.2. Requerimientos

La arquitectura debe cumplir con los requerimientos mostrados en la Tabla 4.2, estos describen las funciones importantes que debe de realizar el módulo, con el fin de cumplir los objetivos de la tesis. Los requerimientos servirán como base para el diseño de un plan de pruebas.

4.1.3. Requerimientos de Verificación

4.1.3.1. Documentación

Alguna información es requerida con el fin de facilitar la realización de la verificación del diseño. La lista de requerimientos contiene las características esperadas de la implementación, las cuales dan a conocer las funcionalidades más importantes de la arquitectura, que reflejan el objetivo de esta tesis. El plan de pruebas proporciona una guía que permite probar la funcionalidad del módulo. Los diagramas a bloque de la arquitectura presentados en el Capítulo 3, ayudan a comprender las interfaces y la comunicación entre los bloques. Contar con estos elementos dará una mayor claridad en el momento de efectuar la verificación.

4.1.3.2. Software

El software necesario para realizar la el proceso de verificación funcional es el siguiente:

ModelSim o QuestaSim, es el simulador necesario para poder ejecutar las pruebas al diseño, con este es posible manipular las entradas y observar las salidas del diseño. De esta manera se puede construir un ambiente de verificación.

Tabla 4.2: Requerimientos de la Arquitectura

Número de Requerimiento	Descripción
R1	El módulo deberá calcular los coeficientes para un filtro igualador bajo el criterio MMSE.
R2	El módulo calculara el número de coeficientes indicado en la entrada Num_Coeficientes.
R3	Los coeficientes deben ser entregados de manera serial por las salidas Coeficientes_PR y Coeficientes_PI, cuando la salida Coeficientes_Valido sea 1, donde Coeficientes_PR es la parte real y Coeficientes_PI la parte imaginaria del coeficiente.
R4	El módulo debe indicar poniendo Igualador_Listo = 1 cuando ha terminado de realizar el cálculo de los coeficientes y está listo para trabajar.
R5	Si el módulo se implementa para calcular hasta un número de coeficientes N debe de poder calcular desde 2 a N, cambiando el valor en la entrada Num_Coeficientes
R6	Después de realizar un cálculo de n coeficientes debe poder realizar otro calculo con un número diferentes de coeficientes.
R7	Mientras no sean cargados el canal estimado, el SNR y el retardo de igualación, no se puede terminar el cálculo.

Matlab, para generar los valores de entrada del módulo, realizar el cálculo de los resultados esperados y compararlos con los resultados obtenidos de la simulación.

4.1.4. Plan de Verificación

Basado en los requerimientos de la Tabla 4.2 se crean una seria de pruebas mostradas en la Tabla 4.3 las cuales permitirán evaluar la funcionalidad del módulo garantizando cumplir con los requerimientos planteados. El plan de pruebas fue creado con el fin de probar la configurabilidad del módulo, y se centra en comprobar el funcionamiento de la arquitectura completa, por lo que no se consideran pruebas para bloques individuales de la arquitectura.

Tabla 4.3: Plan de Pruebas

Número de Prueba	**Descripción**	**Resultado Esperado**
P1	Cargar los valores para el canal estimado, pero no cargar ni el SNR, ni el retardo de igualación; luego esperar 10000 ciclos de reloj.	El módulo debe estar en estado de espera por la carga del SNR y el retardo de igualación, por lo tanto la señal dc Cocficicntc_Valido no debe de presentar el valor de 1 y la señal de Igualador_Listo debe permanecer en cero indicando que el módulo está ocupado.
P2	Cargar los valores para el canal estimado y el SNR, pero no el retardo de igualación; luego esperar 10000 ciclos de reloj.	El módulo debe estar en estado de espera por la carga del SNR y el retardo de igualación, por lo tanto la señal de Coeficiente_Valido no debe de presentar el valor de 1 y la señal de Igualador_Listo debe permanecer en cero indicando que el módulo está ocupado.
P3	Cargar los valores para el SNR estimado y el retardo de igualación, pero no el canal estimado ; luego esperar 10000 ciclos de reloj.	El módulo debe estar en estado de espera por la carga del SNR y el retardo de igualación, por lo tanto la señal de Coeficiente_Valido no debe de presentar el valor de 1 y la señal de Igualador_Listo debe permanecer en cero indicando que el módulo está ocupado.
P4	Cargar los valores para el canal estimado y el retardo de igualación, pero no el SNR; luego esperar 10000 ciclos de reloj.	El módulo debe estar en estado de espera por la carga del SNR y el retardo de igualación, por lo tanto la señal de Coeficiente_Valido no debe de presentar el valor de 1 y la señal de Igualador_Listo debe permanecer en cero indicando que el módulo está ocupado.
P5	Con un módulo sintetizado para 20 coeficientes, cargar el canal estimado, el retardo de igualación, el SNR estimado, y poner 5 en la entrada Num_Coeficientes. Esperar a que se calculen los coeficientes.	EL módulo debe realizar el algoritmo para el cálculo de 5 coeficientes. Después que los cinco coeficientes son entregados la salida Igualador_Listo debe regresar a 1.
P6	Sintetizar un módulo capaz de procesar 30 coeficientes, realizar la secuencia de entrada para el cálculo de 2 coeficientes; cuando el módulo esté listo para un nuevo cálculo incrementar el número de coeficientes en uno e iniciar un nuevo procesamiento, repetir hasta llegar a 30 coeficientes.	El módulo dará como resultado el número de coeficientes indicados para cada cálculo y estos deben concordar con los coeficientes teóricos en Matlab.
P7	Sintetizar un módulo para 30 coeficientes, realizar 100 calculados, donde los valores para las entradas Num_Coeficeintes, Estimado_SNR, Canal_Estimado_PR y Canal_Estimado_PI sean calculados usando funciones pseudoaleatorias. Usar un modelo en Matlab con las mismas entradas para comprobar los resultados.	Se espera que lo valores de coeficientes dados por el módulo concuerden con los dados por el algoritmo en Matlab.
P8	Repetir P7 con módulos sintetizados para 10, 15 , 20 y 25 coeficientes.	Se espera que lo valores dc coeficientes dados por el módulo concuerden con los dados por el algoritmo en Matlab.

4.1.5. Matriz de Trazabilidad

Se debe de garantizar que las pruebas efectuadas cubran todos los requerimientos planteados. La Tabla 4.4 muestra la matriz de trazavilidad que relaciona los requerimientos con las pruebas. Se puede observar que el plan de pruebas cubre todos los requerimientos del diseño. Es importante notar que una sola prueba cubre mas de un requerimiento debido a que en lo general una sola prueba involucra verificar varias funcionalidades del módulo. Con el plan realizado se garantiza que el objetivo de crear un módulo configurable para el número de coeficientes a calcular se cumple, y que los resultados son los correctos, por lo tanto los módulos internos esta funcionando adecuadamente.

Tabla 4.4: Matriz de Trazabilidad

	R1	R2	R3	R4	R5	R6	R7
P1				X			X
P2				X			X
P3				X			X
P4				X			X
P5	X	X	X	X			
P6	X	X	X	X	X	X	
P7	X	X	X	X	X	X	
P8	X	X	X	X	X	X	

4.2. Resultados del costo de Hardware

El hardware se describió usando el lenguaje de descripción de hardware Verilog. Este diseño no fue implementado en un FPGA, por lo que solo se presentan los resultados de síntesis mediante la herramienta Quartus II. La Tabla 4.6 muestran los resultados de síntesis de la arquitectura porpuesta, en un Stratix II para un número diferente de coeficientes. Por otra parte, la Tabla 4.5 muestra los resultados de la arquitectura presentada en [1]. Al comparar ambas tablas, se observa cómo es que ha disminuido el uso de hardware principalmente en el uso de bloques DSP, los cuales representan los bloques multiplicadores del diseño, esto quiere decir que el diseño propuesto, no incrementa el número unidades de procesamiento cuando se incrementa el número de coeficientes. No obstante, la cantidad de bits de memoria utilizada se incrementa, esto concuerda con las modificaciones presentadas en el Capítulo 3, donde la modificación más importante en la mayoría de los módulo fue agregar más memoria, debido a que es necesario almacenar más información porque los cálculos no pueden realizarse en paralelo como en el diseño en [1].

Además de la arquitectura configurable, se tiene una arquitectura simplemente reducida en el número de recursos de hardware sin la capacidad de ser configurable. Esto nos permite medir cual sería la diferencia en cuestión de uso de recurso de hardware entra

Tabla 4.5: Resultados de síntesis para el cálculo de diferentes números de coeficientes con el diseño presentado en [1]

Número de coeficientes	ALUTs Combinacionales	Registros lógicos dedicados	Bits de memoria	Bloques DSP
5	5,261	5,549	7,040	132
10	9,064	9,544	11,360	212
15	13,037	13,554	18,080	292
20	17,246	17,663	27,200	372
25	28,599	21,702	38,720	384

Tabla 4.6: Resultados de síntesis para el cálculo de diferentes números de coeficientes del módulo MMSE configurable

Número de coeficientes	ALUTs Combinacionales	Registros lógicos dedicados	Bits de memoria	Bloques DSP
5	2,315	2,980	7,488	28
10	3,005	4,209	12,224	28
15	3,647	5,398	19,104	28
20	4,677	6,732	28,896	28
25	5,402	7,962	40,576	28

ambas. Para realizar la síntesis se seleccionó un Cyclone IV EP4CGX110DF31C8, esta selección se hizo por ser un modelo de gama mucho más baja que el Stratix, pero cuenta con los recursos suficientes para poder sintetizar los tres diseños. Las Tablas 4.7 y 4.8 muestran los resultados de síntesis para 10 y 30 coeficientes de los diferentes diseños, donde **A** es el diseño reducido en hardware, **B** es el diseño configurable para el número de coeficientes y **C** es el diseño presentado den [1]. Esta tabla también muestra las diferencia y porcentajes de reducción comparados con el diseño **C**, así como la comparación entre **A** y **B**.

En la Tabla 4.7 se puede observar cómo tanto para **A** y **B** se reducen en más del 50 % el uso de elementos lógicos, funciones lógicas y registros dedicados, con respecto a **C**. El uso de bits de memoria se incrementa cerca del 20 %, pero el uso de multiplicadores se reduce en gran medida un 86.79 %. Por otro lado el tiempo de ejecución para el cálculo de 10 coeficientes para los diseños **A** y **B** aumenta cerca de 30 μs.

En la Tabla 4.8 se muestra el resultado para los diseño que calculan 30 coeficientes, en el cual se hace más notoria las ventajas y desventajas de los diseños reducidos, el uso en cuanto a elementos lógicos, funciones combinacionales y registros se reduce por arriba del 70 %, mientras el uso de memoria solo aumento cerca del 6 % para los diseños **A** y **B**. Además, el uso de los multiplicadores se reduce mucho más que con la versión de 10 coeficientes, haciendo que los diseños **A** y **B** utilicen 94.74 % menos multiplicadores que el diseño **C**. No obstante la desventaja obvia de los diseños **A** y **B** es el tiempo de

Tabla 4.7: Resultados de síntesis en Cyclone IV EP4CGX110DF31C8 para un número de 10 coeficientes

	A	B	C	C-A	C-B	B-A	Porcentaje C-A	Porcentaje C-B
Elementos Lógicos	9,097	9,210	29,487	20,390	20,277	113	69.15 %	68.77 %
Funciones Combinacionales	7,711	7,828	26,187	18,476	18,359	117	70.5 %	70.11 %
Registros lógicos dedicados	6,505	6,384	18,743	12,238	12,359	-121	65.29 %	65.94 %
Bits de memoria	12,384	12,600	10,368	-2,016	-2,232	216	-19.44 %	-21.52 %
Multiplicadores de 9 bits	28	28	212	184	184	0	86.79 %	86.79 %
Frecuencia Máxima (MHz)	51.01	55.24	59.73	8.72	4.49	4.23	14.6 %	7.52 %
Tiempo de ejecución (μ s)	44.91	44.47	12.44	-32.47	-29.03	-3.44	-261.01 %	-233.36 %

Tabla 4.8: Resultados de síntesis en Cyclone IV EP4CGX110DF31C8 para un número de 30 coeficientes

	A	B	C	C-A	C-B	B-A	Porcentaje C-A	Porcentaje C-B
Elementos Lógicos	17,456	17,624	77,890	60,434	60,266	168	77.59 %	77.37 %
Funciones Combinacionales	14,667	14,914	72,296	57,629	57,382	247	79.71 %	79.37 %
Registros lógicos dedicados	11,996	11,871	47,534	35,538	35,663	-125	74.76 %	75.03 %
Bits de memoria	60,254	60,336	56,808	-3,456	-3,,528	72	-6.08 %	-6.21 %
Multiplicadores de 9 bits	28	28	532	504	504	0	94.74 %	94.74 %
Frecuencia Máxima (MHz)	48.06	47.92	63.12	15.05	15.2	-0.14	23.86 %	24.08 %
Tiempo de ejecución (μ s)	338.43	339.42	29.96	-308.47	-309.46	-0.99	-1029.61 %	-1032.91 %

ejecución, este se incrementa alrededor del 1000 % para el cálculo de 30 coeficientes.

Habiendo observados los resultados, es facial concluir que los diseños **A** y **B** tiene la ventaja de no aumentar el número de multiplicadores empleados cuando el número de coeficientes aumenta. Además de que tanto las funciones combinacionales como los elementos logicos y los registros dedicados disminuyen más de la mitad del diseño **C** y esta cifra aumenta cuando se incrementa el número de coeficientes a calcular. El uso de memoria será siempre mayor para los diseños reducidos en hardware, pero comparado con el aumento de memoria debido a incrementar el número de coeficientes este es despreciable. El tiempo de ejecución, el cual está ligado a la frecuencia máxima, se vuelve la desventaja principal de las arquitecturas, debido a que al aumentar el número de coeficientes aumentara la diferencia en el tiempo de ejecución entre la arquitectura **C**. Pero para aplicaciones donde el tiempo de ejecución no sea un factor crítico se puede utilizar estos diseños a la perfección.

Un factor importante a determinar es el gasto, en cuestión de hardware, de realizar una arquitectura configurable para el número de coeficientes. Así, las Tablas 4.9 y 4.10, muestran una comparativa entre los diseño **A** y **B**, para el cálculo de 10 y 30 coeficientes. Estos fueron sintetizados en un FPGA Cyclone II, el cual es un modelo de mucho menor

Tabla 4.9: Resultados de síntesis en Cyclone II EP2C20F484C7 para un número de 10 coeficientes

	A	B	B-A	Porcentaje B-A
Elementos Lógicos	8,944	9,049	105	1.17 %
Funciones Combinacionales	7,703	7,806	103	1.34 %
Registros lógicos dedicados	6,321	6,240	-81	-1.28 %
Bits de memoria	12,384	12,600	216	1.74 %
Multiplicadores de 9 bits	28	28	0	0 %
Frecuencia Máxima (MHz)	54.03	47.49	-6.54	-12.1 %
Tiempo de ejecución (μ s)	42.4	48.24	5.84	13.77 %

Tabla 4.10: Resultados de síntesis en Cyclone II EP2C35F672C6 para un número de 30 coeficientes

	A	B	B-A	Porcentaje B-A
Elementos Lógicos	17,120	17,265	145	0.85 %
Funciones Combinacionales	14,637	14,907	270	1.84 %
Registros lógicos dedicados	10,883	10,799	-84	-0.77 %
Bits de memoria	60,264	60,336	72	0.12 %
Multiplicadores de 9 bits	28	28	0	0 %
Frecuencia Máxima (MHz)	59.8	56.21	-3.59	-6 %
Tiempo de ejecución (μ s)	271.99	289.36	17.37	6.39 %

costo y con muchos menos recursos, se escogió este modelo debido a que se cuenta con estas tarjetas y pueden utilizarse para una posterior implementación y pruebas.

La Tabla 4.9 muestra los resultados en el modelo EP2C20F484C7 y la Tabla 4.10 para el modelo EP2C35F672C6. Se observa cómo es que ambos diseños no tiene una gran diferencia en cuanto a uso de recursos de hardware. Existe menos del 2 % de diferencia tanto para los elementos lógicos, las funciones combinacionales, los registros y la memoria utilizada. La diferencia más grande se encuentra en la frecuencia de reloj, dado que ambos diseños necesitan la misma cantidad de ciclos de reloj para terminar una ejecución, el tiempo es solo dependiente de la frecuencia de reloj máxima. En este

Tabla 4.11: Valores de SQNR para diferentes iteraciones de divisor de CORDIC

Número Iteraciones	SQNR
13	64.973
14	70.9916
15	77.0101
16	83.0333
17	83.0044
18	89.0686
19	89.0686
20	89.0686
21	89.0686

aspecto, para la tecnología Cyclone II el diseño no configurable muestra una frecuencia de reloj mucho mayor que la del diseño configurable, pero en la Tabla 4.7 se ve como el diseño **B** tiene una mayor frecuencia de reloj, esto muestra que la diferencia principal de los diseños está determinada solo por la tecnología en que se implementen.

El divisor utilizado para el diseño MMSE, se basa en una arquitectura de CORDIC, también se realizó la implementación de un divisor utilizado el método de Newton-Raphson, este trabajo de muestra en el Apéndice A. Se decidió utilizar un divisor basado en CORDIC dado que como se ve en Apéndice A, utilizan mucho menos recurso de hardware. Además, no se requiere de un muy rápido tiempo de procesamiento para la división y el número de operaciones de división que se realizan es solo el número de coeficientes a calcular.

El bloque de CORDIC puede cambiar la precisión de la operación según el número de iteraciones que se desean realizar; esto también repercute en el número de ciclos de reloj que se tomara el módulo para realizar la operación. Tomando esto en cuenta se consideró que un número de iteraciones aceptable para un tamaño de palabra de 16 bits, es utilizar 18 iteraciones. Esto se puede comprobar con los resultados en la Tabla 4.11. En esta se observa como el incremento en el SQNR después de 18 se detiene, por lo que no tiene mucho sentido incrementar más las iteraciones.

4.3. Prueba de funcionalidad

Para observar la funcionalidad del diseño se tomaran los resultados de la prueba P7, debido que en ella se realiza el cálculo de diferentes coeficientes con valores aleatorios en las entradas. En la Fig. 4.2, se muestran los resultados de la simulación. Para medir el error entre los coeficientes calculados en la arquitectura y el los coeficientes dados por el modelo de referencia, se utiliza la métrica de SQNR, la cual es dada por:

$$SQNR = 10log_{10}(\frac{P_s}{P_e}) \tag{4.1}$$

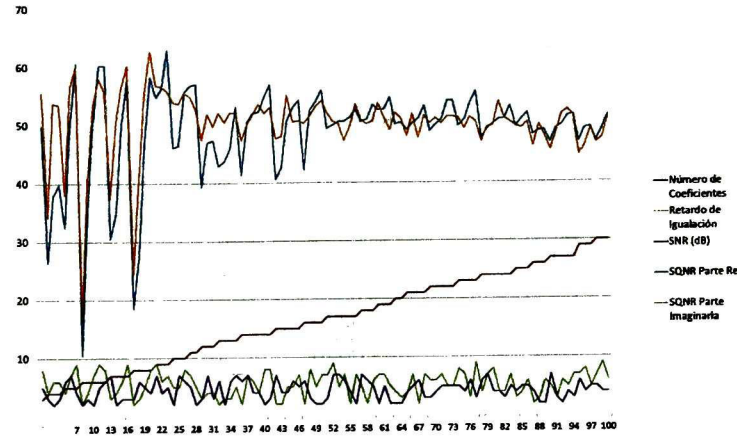


Figura 4.2: Resultados de simulación de P7

Donde P_s y P_e representan la potencia de la señal y la potencia del error, respectivamente. P_s y P_e pueden ser calculadas como sigue

$$Ps = \frac{1}{N}\sum_{i=1}^{N} Vf_i^2 \tag{4.2}$$

$$Pe = \frac{1}{N}\sum_{i=1}^{N} (Vf_i - Vfx_i)^2 \tag{4.3}$$

Donde Vf_i y Vfx_i son el valor en punto flotante y el valor en punto fijo del coeficiente calculado.

Con los resultados mostrados en la Fig. 4.2, se puede observar que el diseño implementado tiene un SQNR promedio de 50 dB, el cual se muestra mucho más estable cuando el número de coeficientes es más grande. Es fácil notar que existen realizaciones donde esta métrica decae por debajo de 30 dB. Esto se debe, a que en esta simulación se están metiendo valores aleatorios en las entradas del diseño sin realizar un análisis de punto fijo, que determine los formatos de palabra más adecuados para las entradas, por lo cual existen valores que no son representables en el formato de punto fijo utilizado, en estos casos el SQNR es muy bajo. Dado que la arquitectura es configurable en síntesis para la longitud y el formato de palabra utilizado, si se hace un análisis de punto fijo para la aplicación en donde se quiera implementar el diseño, se puede obtener mejor desempeño.

Los resultados de la simulación muestran que el módulo es configurable para realizar cálculos con diferentes números de coeficientes y con diferentes valores de entrada para el SNR y el retardo de igualación. Con lo cual se logra un arquitectura configurable tanto en síntesis como en ejecución.

Capítulo 5

Conclusiones y Trabajo Futuro

5.1. Conclusiones

Con la realización de este trabajo se logra comprender el algoritmo de implementación para un igualador de canal con el criterio MMSE. Además de obtener una arquitectura con un mucho menor gasto de hardware, con referencia a la propuesta en [1]. También los 4 bloques principales de la implementación pueden utilizarse por separado para realizar diferentes tipos de algoritmo; es decir, se tiene separadamente módulos de bajo costo para realizar una autocorrelación, tringularizar una matriz, y para realizar una sustitución hacia atrás.

Además, la arquitectura tiene la función de ser configurable para el número de coeficientes, con lo cual se logra que un hardware ya implementado para algún sistema puede variar su funcionamiento, dando al sistema la opción de utilizarlo en diferentes casos sin la necesidad de modificar el hardware. También se muestra, como realizar arquitecturas configurables, no implica necesariamente un incremento considerable en el uso de hardware, como se mostró en el capítulo 4.

El módulo solo quedó en fase de síntesis y no se llego a hacer pruebas con una implementación, pero según los resultados de síntesis prueban que el módulo tiene un gasto reducido de recursos, por lo cual, no aportara un gran consumo cuando se integre a un sistema completo.

5.2. Trabajo Futuro

El diseño fue realizado mediante una modificación a [1], utilizando un menor número de recursos y reduciendo el paralelismo en el diseño. Además, los bloques funcionales principales de diseño los cuales son el autocorrelacionador, el eliminador de Gauss y el bloque sustitución hacia atrás, pueden integrarse a distintos algoritmos dado que funcionan separadamente; esto es una ventaja en cuanto a usabilidad, pero se vuelve una

desventaja si lo que se busca es realizar un solo bloque para un igualador MMSE, con la menor cantidad de recursos.

Dado que los tres bloques trabajan a distintos tiempos, es decir, el bloque Eliminación de Gauss no empezara a procesar hasta que el Autocorrelacionador termina su operación, así como el bloque Sustitución hacia atrás tiene que esperar a que el Eliminador de Gauss realice sus cálculos. Esto ofrece la posibilidad de tener solo una unidad de procesamiento para los tres bloques, es decir, cada bloque procesador en los módulos principales puede combinarse y tener una sola unidad de procesamiento capaz de realizar todas las operaciones necesarias. Esta nueva unidad de procesamiento será compartida según el bloque que la necesite.

Otro punto importante a resaltar es el método para triangularizar la matriz **R**. Como se presentó en [1], este método fue escogido por que el número de operaciones de multiplicación, división y sumas eran más bajo con respecto a los otros métodos probados. Sin embargo, existe implementaciones eficientes para otros métodos, por ejemplo, existen trabajos sobre implementaciones eficientes basadas en CORDIC para una descomposición QR. Si lo que se busca es una reducción en hardware para el igualador podría analizarse implementaciones eficientes para los distintos algoritmos que componen cada bloque y así seleccionar una mejor arquitectura en base a los avances presentados de los diferentes algoritmos.

Bibliografía

[1] J Pizano-Escalante. *Diseño e implementación digital del filtro igualador de canal para el estándar 802.15.3c*. CINVESTAV-GDL, 2010.

[2] J. G. Proakis and J. H. Miller. An adaptive receiver for digital signaling through channels with intersymbol interference. *IEEE Transactions on Information Theory*, 15:484–497, July 1969.

[3] Rodrigue Rabineau Laurent Boher and Maryline Hélard. An efficient mmse equalizer implementation for 4x4 mimo-ofdm systems in frequency selective fast varying channels. *The 18th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–5, September 2007.

[4] Kees van Berkel1 Ozgun Paker and Kees Moerman. Hardware and software implementations of an mmse equalizer for mimo-ofdm based wlan. *Signal Processing Systems Design and Implementation, 2005*, pages 1–6, November 2005.

[5] Joseph Mitola. *Software Radio Architecture*. Wiley-Interscience, 1st edition, January 2002.

[6] J. G. Proakis. *Digital Communications*. McGraw-Hill, 3 ed. edition, 1995.

[7] Aldo G. Orozco Lugo. *BLIND SPATIO-TEMPORAL PROCESSING FOR COMMUNICATIONS*. PhD thesis, University of Leeds, 2000.

[8] P.K. Meher, J. Valls, Tso-Bing Juang, K. Sridharan, and K. Maharatna. 50 years of CORDIC: Algorithms, architectures, and applications. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 56(9):1893–1907, 2009.

[9] Jean-Michel Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, 2nd edition, October 2005.

Apéndice A

Artículo aceptado en ReConFig 2013

Fast Fixed-Point Divider based on Newton-Raphson Method and piecewise polynomial approximation

A. Rodríguez-García*, L. Pizano-Escalante*, R. Parra-Michel*, O. Longoria-Gandara† and J. Cortez‡
*Dept. of Electrical Engineering, CINVESTAV-IPN, Zapopan, Mexico
Email: {arodrigue, jpizano, rparra}@gdl.cinvestav.mx
†Dept. of Electronics, Systems and Informatics, ITESO, Tlaquepaque, Mexico
Email: olongoria@iteso.mx
‡Dept. of Engineering and Technology, ITSON, Cd. Obregon, Mexico

***Abstract*—Division is an operation extensively used in architectures for digital signal processing algorithms, which in portable devices require an implementation using fixed-point format. In this paper, a novel fixed-point divider is proposed. The divider architecture is based on a division algorithm that uses the reciprocal operation and a post-multiplication. In turn, reciprocal operation is based on the Newton-Raphson algorithm, where the seed is provided through piecewise polynomial approximation. Reciprocal operation is performed with only two clocks cycles and division operation requires only three clock cycles. A comparison between the proposed architecture and dividers based on coordinate rotation digital computer, shows that the proposed architecture achieves approximately 5-fold gain in execution time for applications in 50-100 MHz frequency range and higher signal-to-quantization-noise ratio.**

I. Introduction

Division is a fundamental operation in architectures for digital signal processing (DSP) algorithms like singular value decomposition or QR decomposition [1], [2]. In this kind of algorithm, division operation is implemented in fixed-point (FxP) format, due to its shorter execution times and lesser area resources than floating-point (FP) format, also FxP implementation offers good system performance in terms of numerical accuracy, for example in the development of digital communications systems [3].

There are several types of algorithms for implementing the division operation such as SRT (named after Sweeney, Robertson and Tocher) [4], digit recurrence [5], numerical methods and special techniques like coordinate rotation digital computer (CORDIC) [6]. SRT division uses only add/subtract and shift operations for processing. In digit recurrence the quotient is represented in a radix-r form and one digit of the quotient is obtained per iteration. Division by numerical methods has the characteristic of obtaining more than one bit per iteration, examples of these algorithms are Newton-Raphson (NR) or Goldschmidt. CORDIC is one of the most used algorithm for implementing division operation due to its low cost in hardware and the high frequency that it can achieve [7]. However, its iterative nature impacts on its execution time when the system clock is limited to low frequencies for reducing the power consumption.

In this paper a new architecture for a FxP divider based on reciprocal operation and post-multiplication is proposed. The aim is to reduce execution time to speed-up processing at practical clock frequencies. In turn, reciprocal operation uses piecewise polynomial approximation to obtain an approximated value (seed) for the reciprocal value and one iteration of the NR method obtains the final value. Also, it is presented a procedure for scaling and de-scaling the input value, which makes an architecture independent of the FxP format. This processing takes two clock cycles for reciprocal operation and three for division operation. A comparison between the proposed architecture and CORDIC shows an improvement in execution time and numerical accuracy using the proposed architecture, therefore, it is suitable as hardware accelerator.

This paper is organized as follows: Section II explains the procedure to determine the numerical accuracy of FxP algorithm for DSP. Section III introduces the proposed algorithm; Section IV discusses its architecture in detail; Section V presents the implementation results and performance comparison with CORDIC designs; and finally, Section VI presents the conclusions.

II. Signal-to-Quantization-Noise-Ratio

FxP implementation offers less area resources and shorter execution times than FP implementations, therefore FxP format is preferred for the implementation of portable devices [3]. An important issue related with FxP implementations is to determine how much FxP format affects the numerical accuracy of the algorithm been implemented. Numerical errors or an incorrect selection in the FxP format will degrade the system performance. The standard way to analyze the numerical accuracy of a FxP architecture is through simulations using the well-known DPS techniques proposed in [8], [9]. These analysis takes as figure of merit the signal-to-quantisation-noise ratio (SQNR).

SQNR is defined as the logarithm of the ratio between the power of the signal, Ps, and the power of the error, Pe, as follows:

$$\text{SQNR} = 10\log_{10}\left\{\frac{Ps}{Pe}\right\} \tag{1}$$

Assuming ergodicity, Ps and Pe can be calculated with

$$Ps = \frac{1}{N}\sum_{i=1}^{N} Vf_i^2 \tag{2}$$

$$Pe = \frac{1}{N}\sum_{i=1}^{N} (Vf_i - Vfx_i)^2 \tag{3}$$

978-1-4799-2079-2/13/$31.00 ©2013 IEEE

where Vf_i and Vfx_i are the FP and FxP value of the i-th result, respectively, and N is the number of experiments. As the SQNR value is application dependent, it is not possible to know in advance either the minimum number of bits in the FxP representation of the numerical values, or the minimum value of the SQNR required to guarantee system functionality. In this paper the following nomenclature will be used to define a FxP value Q(wl,i, $sign$) where wl is the word length, i is the number of integer bits and $sign = u, s$ defines the signed format, where u represents an unsigned number and s a signed number.

III. Division Algorithm

The proposed divider architecture computes the division operation of two FxP numbers a and x i.e., it computes the operation

$$y = \frac{a}{x}, \tag{4}$$

where the result y is given in FxP format in three clock cycles. The proposed algorithm consists of three steps:

- Firstly, the reciprocal operation is performed:

$$y' = \frac{1}{x}. \tag{5}$$

- Secondly, a post-multiplication is performed using the reciprocal value to obtain:

$$y'' = y' \times a. \tag{6}$$

- Finally, the result is rounded to the nearest as

$$y = round(y''). \tag{7}$$

Reciprocal operation is performed following the next five steps:

1) Scaling: The input data with a certain FxP format is mapped within the reduced working range $wr = [\alpha, \beta]$, where $\alpha = 1$ and $\beta = 2$.
2) Seed computation: The scaled data is used in a polynomial to approximate the division result in the working range. This result is the seed for NR step.
3) NR algorithm: The seed is taken from the polynomial evaluation and it is used to perform one iteration of the NR algorithm.
4) De-scaling: The NR result is de-scaled to the original FxP format.
5) Rounding: This step computes the rounding operation of the resulting data from de-scaling.

This FxP divider based on reciprocal operation, post-multiplication and rounding is described in detail in the following subsections. Each step in the algorithm is mapped with a particular functional unit and its hardware architecture is shown in Fig. 1.

A. Scaling

Different types of FxP implementations need different types of FxP formats (integer and fractional parts may have any number of bits). Therefore, in order to produce an architecture independent of the FxP format, a scaling step is required. Scaling consist on mapping the input data within the wr, which makes possible to design an efficient architecture for reciprocal operation in that specific range, allowing only the scaling step to be format dependent. Scaling is achieved by simple manipulation of (5), as follows:

$$y' = \frac{1}{x} = \frac{1}{2^n}\frac{1}{\frac{x}{2^n}} = \frac{1}{2^n}\frac{1}{z} \tag{8}$$

where n is the scaling factor and $z \in wr$. Therefore, y' can be obtained first by computing $1/z$, i.e., reciprocal operation in the working range, and then by de-scaling the result by $1/2^n$. The restriction of (8) is that n must be integer, if this is met, then the implementation of scaling and de-scaling consists of simple shifting operations. With a scaling $x/2^n$ from (8), it follows that each value of n maps an interval of x within the wr. Particularly, the n-th value divides x in the interval $[\mathrm{LB}_n, \mathrm{UB}_n]$ given by:

$$\mathrm{LB}_n = \alpha 2^n, \qquad \mathrm{UB}_n = \beta\,(2^n) - 1/2^{frac}; \tag{9}$$

where $1/2^{frac}$ corresponds to the minimum fractional number that can be represented in a given format of the FxP word representation, and $frac$ is the number of bits in the fractional part.

B. Seed Computation

Seed computation is based on polynomial approximation, which computes the necessary seed for NR step in the wr. It is performed using a second order polynomial as follows:

$$y_s = a_2 z^2 + a_1 z + a_0, \tag{10}$$

where y_s is the approximation of the reciprocal in the wr. In order to produce a division operation with high SQNR, the approximated reciprocal value is required to have a high SQNR. However, one second order polynomial is not enough for producing this requirement. Therefore, the wr is divided into six segments, which requires a second order polynomial for each segment. The decision of dividing the wr in six segments is based on simulation results. Table I shows the bounds of each segments in wr where LBS and UBS stands for lower bound and the upper bound of each segment, respectively. It can be noted that values in Table I are powers of two, making the selection of the correct polynomial suitable for hardware implementation. Table III shows the FP coefficients and its equivalent value in FxP for polynomials in each different segment.

C. Newthon Raphson

The NR algorithm is a well-known and very efficient technique for finding roots of functions. Particularly, it can be used to evaluate the reciprocal operation in the following iterative way [10]:

$$w_{n+1} = w_n(2 - y_s w_n), \tag{11}$$

TABLE I. BOUNDS OF THE 6 SEGMENTS IN WHICH w_r IS DIVIDED FOR THE PIECEWISE POLYNOMIAL APPROXIMATION

Segment	LBS	UBS
1	1	1.0625
2	1.0625	1.125
3	1.125	1.25
4	1.25	1.5
5	1.5	1.75
6	1.75	2

where w_n is the result at the n-th iteration and w_0 is the initial approximation. This method has quadratic convergence, which means that the error decreases quadratically and the number of accurate bits is doubled in each iteration. Therefore, if a seed w_0 is provided with 8 bits of accuracy, only one iteration of NR algorithm is needed to provide an accurate result in a word length of 16 bits.

D. De-scaling

Since the result of the NR algorithm belongs to the working range, it is necessary to return it to the original FxP format. From (8), it follows directly that

$$y' = \frac{1}{2^n} \times w_1, \tag{12}$$

which corresponds to a simple shift operation of n bits.

E. Rounding

In order to preserved the accuracy that NR algorithm provides, a rounding step is incorporated. Rounded value is represented as:

$$y'_{RND} = round(y'); \tag{13}$$

where y' is the result from NR algorithm The implemented rounding is to the nearest, because it offers a good trade-off between area resources and numerical accuracy.

F. Post-Multiplication

The last step is a post-multiplication between the reciprocal value and the divined value. Also, another rounding to the nearest is performed to preserve numerical accuracy. It can be noted that if it were needed, the divider can provide the reciprocal operation as output, in which case the last multiplication and rounding steps can be avoided. The final procedure in the algorithm is:

$$y = round(y'_{RND} \times a); \tag{14}$$

where y is the final value, a is the dividend which in the case of $a = 1$ reciprocal value is the delivered result in the algorithm.

IV. ARCHITECTURE

The architecture for the proposed divider is shown in Fig. 1. Its interface has two data inputs; **Dividend** and **Divisor**, one output; **Result** and two control inputs; **Start** and **Selector**. When **Start** = 1 the calculation begins, if **Selector** = 1 division operation is performed, otherwise the reciprocal operation

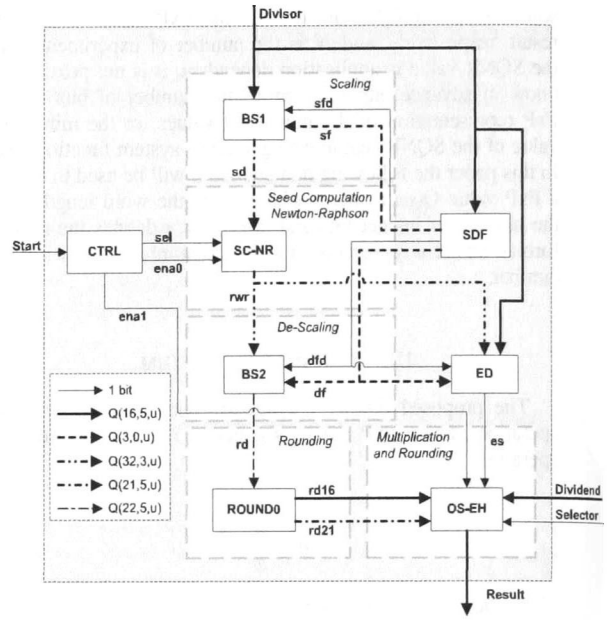


Fig. 1. Hardware Architecture for the proposed algorithm

is computed. Hence, division and reciprocal operations take 3 and 2 clock cycles respectively.

The Architecture is composed of eight functional blocks: Barrel Shifter 1 (**BS1**), Seed Computation and Newton-Raphson (**SC – NR**), Barrel Shifter 2 (**BS2**), Operation Selector and Error Handler (**OS – EH**), Scaling and De-Scaling Factor (**SDF**), Error Detector (**ED**) and Control (**CTRL**). Each block is explained in the following paragraphs. In particular, the architecture is explained taking as case study a FxP format Q(16,5,u) for **Divisor**, **Dividend** and **Result** and with a value of 8.35009765625 (0x42CD in FxP) and 20.4501953125 (0xA39A in FxP) for **Divisor** and **Dividend**, respectively. It can be noted that the selected FxP format has a dynamic range (possible representable values) from 0 to 31.99951171875. All word lengths and FxP formats were obtained through a FxP analysis which is explained in [8] and [9].

A. Scaling and De-Scaling Factor

Scaling and De-scaling factor is calculated by the **SDF** block. Its input is the **Divisor**, its outputs are **sf**, **sfd**, **df** and **dfd** signals. **sf** and **sfd** represent the magnitude of the scaling factor and its direction (left or right) of the shift, respectively. **df** and **dfd** represent magnitude of the de-scaling factor and its direction, respectively. Scaling factor and de-scaling factor are computed based on (9) and shown in Table II for a input format $Q(16, 5, u)$.

B. Barrel Shifter 1

This block is responsible to scale the data input into the w_r implementing only shift operations in combinational way. Its inputs are the signals **Divisor**, **sf** and **sfd**. **Divisor** is

TABLE II. SCALING INTERVALS FOR FXP FORMAT Q(16,5,u) AND BARREL SHIFTERS INPUTS CORRESPONDING TO SCALING FACTOR

Scaling factor n	LB_n	UB_n	sfd	sf	dfd	df
4	16	31.999511171875	0	2	0	4
3	8	15.999511171875	0	1	0	3
2	4	7.99951171875	0	0	0	2
1	2	3.99951171875	1	1	0	1
0	1	1.99951171875	1	2	0	0
-1	0.5	0.99951171875	1	3	1	1
-2	0.25	0.49951171875	1	4	1	2
-3	0.125	0.24951171875	1	5	1	3
-4	0.0625	0.12451171875	1	6	1	4
-5	0.03125	0.06201171875	1	7	1	5

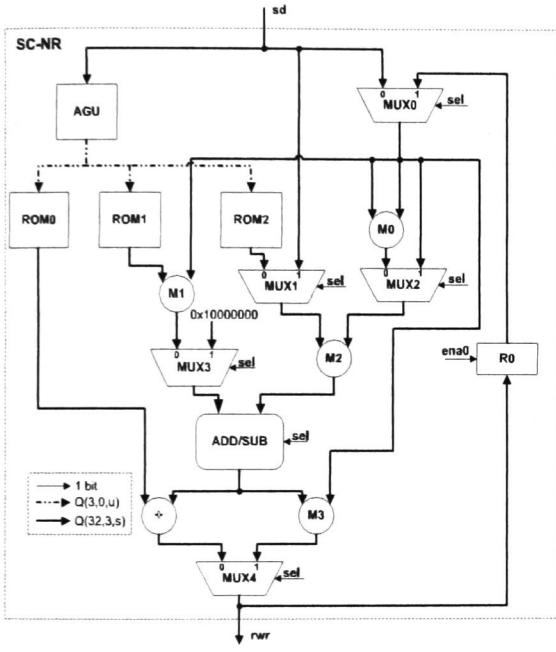


Fig. 2. Architecture for the Seed Computation and Newton-Rapshon

the data to be scaled, **sf** is the scaling factor that indicates the magnitude of the shift operation and **sfd** specify the direction of the shift operation. The result that this block delivers is the scaled divisor (**sd**) in the wr with a FxP format Q(32,3,u). In the case of **Divisor** = 0x42CD, **BS1** receives from **SDF**, **sf** = 1 and **sfd** = 0. The resulting value is **sd** = 1.043762207031250 (0x21668000 in FxP).

C. Seed Computation and Newton-Rapshon

The **SC – NR** block combines the Seed Computation and NR algorithm in one architecture that reuses the multipliers for both steps as show the Fig. 2. Inputs of this block are the **sd** from **BS1** block and **sel** and **ena0** from the **CTRL** block. Its output is the reciprocal value in the wr trough the signal **rwr** with a FxP format Q(32,3,u).

In the first clock cycle of execution **sel** = 0 and **ena0** = 1, enabling the seed computation. Internally, there are three

TABLE III. POLYNOMIAL COEFFICIENTS FOR PIECEWISE POLYNOMIAL APPROXIMATION

I	Format	$a2$	$a1$	$a0$
1	FP	0.917393513301662	-2.8359764333051	2.91866701344667
	FxP	0x1D5B49A4	0xA53FAE5A	0x5D65B85D
2	FP	0.768450318213578	-2.519699687581O1	2.75091540467984
	FxP	0x1897251F	0xA15F9FC3	0x5807F7BE
3	FP	0.601841361908552	-2.1423922010272	2.53749966666238
	FxP	0x134248D1	0xBB7185E9	0x51333280
4	FP	0.390118407952177	-1.60698212392062	2.19928868552267
	FxP	0x0C7BD999	0xCC939A3A	0x466092AA
5	FP	0.2353977687854O8	-1.146413486652I7	1.856702U2833508
	FxP	0x466092AA	0xDB5094AA	0x3B6A1A5F
6	FP	0.152851245267241	-0.859141210204386	1.60685190039622
	FxP	0x04F4284B	0xF481FA4B	0x336B54AD

read only memories named **ROM0**, **ROM1** and **ROM2**. These memories store for each segment the coefficients of the polynomial that approximates the reciprocal operation in the working range. FxP coefficients a_2, a_1 and a_0 are stored in **ROM0**, **ROM1** and **ROM2**, respectively. The specific FxP values a_2, a_1 and a_0 are shown in Table III. All coefficients have FxP format equal to $Q(32,3,s)$.

Signal **sd** is connected to the address generator unit (**AGU**), which is responsible for generating the address to select the correct polynomial coefficients from ROM memories depending in which segment the **sd** value is located. In the case of **sd** = 1.043762207031250 (0x21668000 in FxP) the segment number 1 is selected, therefore, a_2 = 0.917393513301662 (0x1D5B49A4 in FxP), a_1 = -2.8359764333051 (0xA53FAE5A in FxP) and a_0 = 2.91866701344667 (0x5D65B85D in FxP).

Multiplexer 1 to Multiplexer 4 (**MUX1 – MUX4**) select the input marked with zero. Multiplier 0 (**M0**) takes **sd** and performs z^2. Multiplier 2 (**M2**) takes coefficient a_2 from **ROM2** and z^2 from **M0** and computes a_2z^2. Multiplier 1 (**M1**) takes coefficient a_1 from **ROM1** and **sd** to perform a_1z. Data from **M2** and **M1** are added in **ADD/SUB** and the result is finally added with coefficient a_0 from **ROM0** and then stored in register **R0**. The seed for NR algorithm is stored in **R0**, the seed has a value of 0.958026759326458 (0x1EA827BC in FxP) with format $Q(32,3,u)$.

In the second clock cycle with **sel** = 1 and **ena0** = 0 the NR algorithm is computed. Then, all multiplexers select their input marked with number one. The seed in **R0** is multiplied by **sd** in **M2**. **MUX2** selects as output 0x10000000 (2 in format $Q(32,3,u)$) and **ADD/SUB** block configured as substrator performs the operation $2 - y_n w_n$. The result is multiplied in **M3** by the seed in **R0**. **MUX4** selects the value from **M3** which represents the reciprocal approximation **rwr** with a format $Q(32,3,u)$. In the given example with **sd**= 0x21668000 and a seed equal to 0x1EA827BC, **rwr** = 0.958072625100613 (0x1EA887EC in FxP).

D. Barrel Shifter 2

The De-Scaling step is performed by **BS2**. It is a barrel shifter like **BS1**. Its inputs are **rwr** from **SC – NR**, **df** and **dfd** from **SDF** block. Its output is the de-scaled data through the signal **rd** with a format $Q(22,5,u)$. In the case of **rwr** = 0x1EA887EC, **rd** = 0.119758605957031 (0x003D51 in FxP).

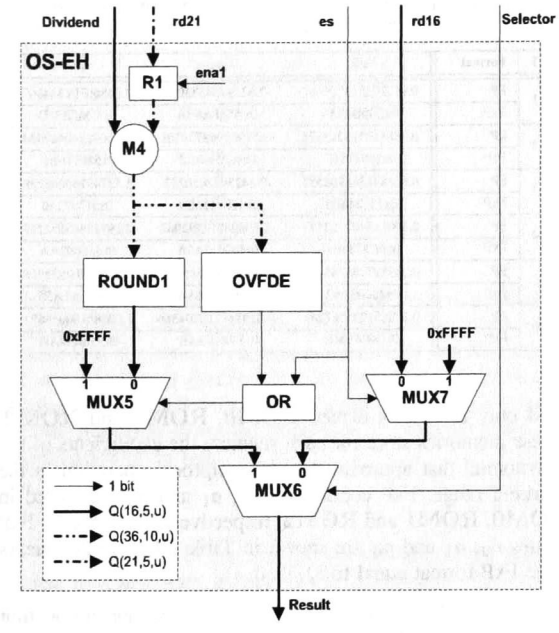


Fig. 3. Hardware Architecture for the Operation Selector and Error Handler

E. Rounding

RND block performs rounding to the nearest. Its input is **rd** from **BS2**, its outputs are **rd16** and **rd21**. **rd16** signal is used when reciprocal value is the selected operation. **rd21** is used when division is the selected operation. **rd21** contains five extra bits in the fractional part and is used in the **OS – EH** for computing the division.

F. Error Detector

The **ED** is responsible of the detection when **Divisor** has a value whose reciprocal cannot be represented in the FxP format for the output format in **BS2**. For example in the Q(22,5,*u*) output format for **BS2**, as has been used in this case, the values of **Divisor** are between 0 to 0.03125, whose reciprocal is greater than 32, which are not representable for Q(22,5,*u*) format.

G. Operation Selector and Error Handler

OS – EH block selects between division or reciprocal operation. It is also responsible of deciding when the output must be represented with the saturation value (maximum representable value corresponding to 0xFFFF). **OS – EH** architecture is shown in Fig 3. When **Selector** = 1, multiplexer 6 (**MUX6**) selects division operation as output. When **Selector** = 0, **MUX6** selects reciprocal operation as output.

In the case when division operation is selected, **Dividend** and **rd21** are multiplied using multiplier 4 (**M4**). The resulting value is analyzed in the overflow detector (**OVFDE**), and if an overflow is detected, multiplexer 5 (**MUX5**) selects its input marked with number one, passing the saturation value to multiplexer 6 (**MUX6**). Also, if es signal from ED block is asserted, **MUX5** and **MUX7** select the saturation value. Otherwise, result from **M5** is rounded to the nearest and is passed to **MUX6** and assigned to **Result**. In the case when reciprocal operation is chosen, multiplexer 7 (**MUX7**) selects between **rd16** signal and the saturation value, the selected value is sent to **MUX6**, and finally assigned to **Result**.

TABLE IV. IMPLEMENTATION RESULTS

Resources	DIV-PPA&NR	DIV-CORDIC	DIV-CORDIC-P
Clocks Cycles	3	16	16
Max. Freq. (MHz)	68.62	351.9	704.72
Execution Time in μs for Max. Freq.	0.04371	0.04546	0.0227
Execution Time in μs for 50 MHz	0.06	0.32	0.32
Adaptive Logic Module (ALM)	339	154	329
Dedicated Register	73	103	483
Multipliers	4(32x32 bits) 1(16x21 bits)	0	0

V. IMPLEMENTATION RESULTS

In order to show the advantages of the proposed algorithm a comparison between it and two architectures of dividers based on CORDIC algorithm is presented. The first one is a very economical hardware module [7], which performs CORDIC algorithm for division in a sequential way, the module takes two FxP value of 16 bits as inputs, and after 16 clock cycles the results is delivered, due to the sequential nature of this architecture a new computation needs another 16 clock cycles, hereafter, this architecture is named DIV-CORDIC. The second architecture is a pipeline version of the CORDIC algorithm for division [7], which represent a very fast hardware module, however with a considerable increment of area. The pipeline architecture takes two 16-bit FxP value as input, and after 16 clock cycles the result is delivered, a new result is delivered each clock cycle if there is a constant flow of data input, hereafter, this architecture is named DIV-CORDIC-P.

The proposed division architecture (DIV-PPA&NR) and the two dividers design based on CORDIC algorithm were implemented using Verilog-HDL. The architectures were synthesized into the FPGA Stratix-V 5SGXMA7N1F45C1 using the 12.1 version of Quartus II. Synthesis results are summarized in Table IV.

It can be seen in Table IV that the DIV-PPA&NR architecture has slower clock frequency and requires more hardware resources than DIV-CORDIC, however the execution time of DIV-PPA&NR for obtaining a division is shorter than DIV-CORDIC, this is taking the maximum clock frequency that each module can reach. In the case of DIV-CORDIC-P hardware resources has incremented in comparison with DIV-CORDIC. In relation with DIV-PPA&NR, DIV-CORDIC-P has less hardware resources and the execution time for obtaining a division is shorter than DIV-PPA&NR, this is taking the maximum clock frequency that each module can reach.

One of the main objectives in portable devices, or in

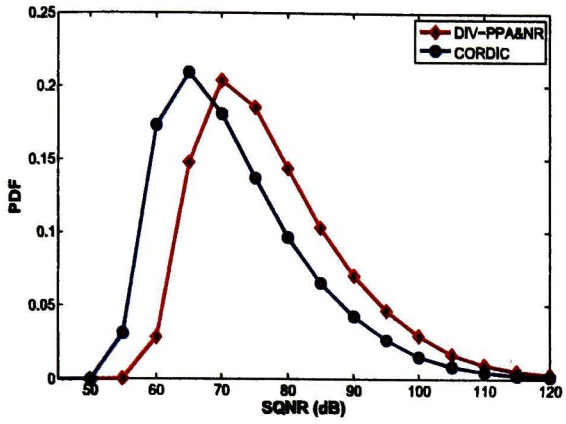


Fig. 4. SQNR distribution for the proposed DIV-PPA&NR and CORDIC architectures.

general, devices that use batteries, is power consumption. Therefore, in order to give a insight in this topic a power analysis is presented, which is made using the tool Powerplay Power Analyzer of Quartus that estimates the thermal power dissipation in the design being implemented. This analysis is performed over the DIV-CORDIC and the proposed architecture. This decision for taking this CORDIC is because it has least hardware resources. The analysis is performed for DIV-CORDIC for the following frequencies 50 MHz, 100 MHz and 250 MHz the results are 1016.16 mW, 1021.64 mW and 1028.14 mW respectively. The DIV-PPA&NR is analyzed only for 50 MHz and the total thermal power dissipation is 1021.91 mW. It can be seen the thermal power dissipation at 50 MHz for the DIV-PPA&NR is similar to the DIV-CORDIC at 100 MHz.

Furthermore, hardware implementations for portable devices must provide a short execution time with the lowest frequency operation. Taking into account this restriction, in Table IV are shown the execution time for the different architecture using a clock frequency of 50 MHz. It can be seen that in this context, DIV-PPA&NR offer the shortest execution time, indeed a 5-fold gain.

Another important feature of digital-arithmetic hardware implementations for portable devices is related with numerical accuracy, because it affects system performance. Fig. 4 shows the probability density function (*pdf*) of SQNR of DIV-PPA&NR and CORDIC implementation (the same curve applies for both CORDIC architectures). These curves were obtained by simulation with the range value of **Dividend** = $[9.765625, 14.65332031251]$ (in FxP [0x4c20, 0x753a]) and for **Divisor** = $[0, 31.99951171875]$ (in FxP [0x0000, 0xFFFF]). It can be seen that DIV-PPA&NR offer an improvement in the numerical accuracy. The *pdf* of the proposed algorithm has minimum SQNR of 55 dB, while CORDIC *pdf* is around 50 dB. Also, the mean value of the DIV-PPA&NR is greater than CORDIC's *pdf*. It can be concluded that in average, DIV-PPA&NR has also a better numerical accuracy.

VI. Conclusion

This paper presents a novel divider architecture that provides improvements on execution time and numerical accuracy in comparison with current divider implementations for FxP arithmetic. In the case of execution time, it was shown that the proposed architecture provides better execution time than DIV-CORDIC at the expense of more hardware resources with similar power consumption, and when speed computation at manageable clock frequencies is a design target, the proposed architecture provides a gain of 5-fold even when it is compared with the pipelined CORDIC version. Furthermore, in the case of numerical accuracy, DIV-PPA&NR produces results with higher SQNR than both version of the CORDIC algorithm, which is an important issue to preserve system performance in architectures for DSP algorithms.

References

[1] Y.-L. Chen, C.-Z. Zhan, T.-J. Jheng, and A.-Y. Wu, "Reconfigurable adaptive singular value decomposition engine design for high-throughput MIMO-OFDM systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 21, no. 4, pp. 747–760, 2013.

[2] P. Luethi, C. Studer, S. Duetsch, E. Zgraggen, H. Kaeslin, N. Felber, and W. Fichtner, "Gram-schmidt-based QR decomposition for MIMO detection: VLSI implementation and comparison," in *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, 2008, pp. 830–833.

[3] S. Gifford, J. E. Kleider, and S. Chuprun, "Broadband OFDM using 16-bit precision on a SDR platform," in *Military Communications Conference, 2001. MILCOM 2001. Communications for Network-Centric Operations: Creating the Information Force. IEEE*, vol. 1, 2001, pp. 180–184 vol.1.

[4] I. Koren, *Computer Arithmetic Algorithms, Second Edition*, 2nd ed. A K Peters/CRC Press, 11 2001.

[5] M. D. Ercegovac and T. Lang, *Digital Arithmetic (The Morgan Kaufmann Series in Computer Architecture and Design)*, 1st ed. Morgan Kaufmann, 6 2003.

[6] Y. Hu, "CORDIC-based VLSI architectures for digital signal processing," *Signal Processing Magazine, IEEE*, vol. 9, no. 3, pp. 16–35, 1992.

[7] P. Meher, J. Valls, T.-B. Juang, K. Sridharan, and K. Maharatna, "50 years of CORDIC: Algorithms, architectures, and applications," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 56, no. 9, pp. 1893–1907, 2009.

[8] W. Sung and K.-I. Kum, "Simulation-based word-length optimization method for fixed-point digital signal processing systems," *Signal Processing, IEEE Transactions on*, vol. 43, no. 12, pp. 3087–3090, 1995.

[9] D. Menard, D. Chillet, and O. Sentieys, "Floating-to-fixed-point conversion for digital signal processors," *EURASIP J. Appl. Signal Process.*, vol. 2006, pp. 77–77, Jan. 2006.

[10] J.-M. Muller, *Elementary Functions: Algorithms and Implementation*, 2nd ed. Birkhuser, 10 2005.

Apéndice B

Funciones en Verilog

En esta sección se presenta la relación de los módulos implementados para el igualador MMSE configurable con los archivos en Verilog. Las siguientes tablas se conforman de 4 columnas la primera muestran el nombre del módulo en la arquitectura, la segunda el nombre de la implementación en Verilog, la siguiente los archivos necesarios para síntesis y la ultima el número de figura donde se muestra el RTL del módulo.

Los nombres en las columnas Implementación en Verilog y Archivos Necesarios se refieren a nombres de archivos con extensión ".v". Los nombres de archivos escritos en negritas son los módulos que forman parte de la arquitectura Top del diseño, la cual se muestra en la Fig. B.1.

La Tabla B.1 muestra los archivos necesarios para utilizar el archivo principal del diseño. La Tabla B.2 muestra los archivos en Verilog para la utilización del módulo autocorrelacionador descrito en la sección 3.3. Las Tablas B.3, B.4 y B.5 hacen referencia a las arquitecturas en las secciones 3.6, 3.9 y 3.10, respectivamente.

Tabla B.1: Relación de archivos en Verilog con el módulo MMSE

Módulo en la Arquitectura	**Implementación en Verilog**	**Archivos necesarios**	**Figura**
Igualador MMSE en Tiempo	MMSE_Equalizer.v	Correlation, Back_Substitution_Array, Back_Substitution_Feeder, Gauss, Gauss_Feeder, Selector_Of_Column	B.1

Tabla B.2: Relación de archivos en Verilog con el módulo Autocorrelacionador

Módulo en la Arquitectura	**Implementación en Verilog**	**Archivos necesarios**	**Figura**
Autocorrelacionador	Correlation	LIFO, Correlation_Control, FIFO, Correlation_PE	B.2
Procesador de autocorrelación	Correlation_PE		B.2
Memoria de canal	LIFO		B.2
Memoria de Coeficientes	FIFO		B.2
Memoria de Y	FIFO		B.2
Memoria de X	FIFO		B.2
Unidad de Control	Correlation_Control	Correlation_State_Machine_ControlV2, Correlation_Counter, Correlation_Tile_Counter, Correlation_Data_Feeder_Selector, Correlation_Timer, Correlation_LIFO_Reader	B.3
MSE Canal	Correlation_LIFO_Reader		B.3
MSL Canal	Correlation_LIFO_Reader		B.3
Contador	Correlation_Counter		B.3
Generador	Correlation_Timer	Correlation_Timer_State_Machine, Correlation_Counter	B.4
DMS	Correlation_Timer_Mux	Correlation_Timer_Mux_State_Machine, Correlation_Counter	B.5
Generador de pulsos	Correlation_Data_Feeder_Selector		B.3
Contador_PP	Correlation_Tile_Counter	Correlation_Counter_Synchronous_Reset	B.6

Tabla B.3: Relación de archivos en Verilog con el módulo Eliminador de Gauss

Módulo en la Arquitectura	**Implementación en Verilog**	**Archivos necesarios**	**Figura**
Alimentador de Gauss	Gauss_Feeder	Gauss_Feeder_Control, Gauss_Feeding_Registers, FIFO_Whit_Depth_FIFO	B.7
Memoria de autocorrelación	FIFO_Whit_Depth_FIFO		B.7
Generador de Matriz	Gauss_Feeding_Registers	Gauss_Feeding_Register	B.8
Control de Alimentador de Gauss	Gauss_Feeder_Control	Gauss_Feeder_Counter, One_Shot, Gauss_Feeder_State_Machine_Data_Collector, Gauss_Feeder_Counter_With_LimitCountIn	B.9
Arreglo de procesadores de Gauss	Gauss	Gauss_PE_1, Gauss_PE_2, DemuxParametrizable, FIFO_Whit_Depth_FIFO_In, LockedMemory, Memory_Output_Gauss, MuxParameterizable, Gauss_SubTile_Counter	B.10
Memoria de Gauss	FIFO_Whit_Depth_FIFO_In		B.10
Procesador de Gauss Tipo 1	Gauss_PE_1	Gauss_Real_Division, Gauss_PE_1_Control	B.11
Control para procesador tipo 1	Gauss_PE_1_Control	Gauss_Counter_Synchronous_Reset _Whit_LimitCountIn, One_Bit_Memory, Gauss_PE_1_State_Machine_Control, Gauss_Tile_Counter	B.12
División para procesador tipo 1	Gauss_Real_Division	Complement_2_to_Magnitud_Sing, Control, Counter_i, Register, Barrel_Shifter, Round, One_Shot_Mealy, Adder_Sub, Register, Adder_Sub, Register_With_Syn_Reset	B.14
Contador de procesos parciles	Gauss_Tile_counter	Gauss_Tile_Down_Counter_Synchronous _Reset_With_LimitCountIn, Gauss_Tile_Counter_Synchronous_Reset	B.13
Procesador de Gauss Tipo 2	Gaus_PE_2	Gauss_PE_2_Control	B.15
Control para procesador tipo 2	Gaus_PE_2_Control	Gauss_PE_2_State_Machine_Counter, One_Bit_Memory, Gauss_PE_2_State_Machine_control	B.16

Tabla B.4: Relación de archivos en Verilog con el módulo Sustitución hacia atrás

Módulo en la Arquitectura	**Implementación en Verilog**	**Archivos necesarios**	**Figura**
Alimentador	Back_Substitution_Feeder	Back_Substitution_Feeder_Control, Back_Substitution_Feeder_Counter, Back_Substitution_Feeder_SM_Counter _with_limitCountIn Gauss_Tile_Counter_Synchronous_Reset	B.17
Contador	Back_Substitution_Feeder_Counter	BS_Down_Counter_synchonous_Reser_Whit _LimitCountIn, BS_Counter_Synchronous_Reset	B.18
Contador de renglón	Back_Substitution_Feeder_SM_Counter_with_limitCountIn		B.17
Control	Back_Substitution_Feeder_Control		B.17
Arreglo de Procesadores de Sustitución hacia atrás	Back_Substitution_Array	DemuxParametrizable, Back_Substitucion_LIFO, MuxParametrizable, FIFO_Whit_Depth_FIFO_In, Back_Substitution_PE_Type_2, Back_Substitution_Process_Counter_Whit _LimitCountIn	B.19
Procesador Sustitución hacia atrás	Back_Substitution_PE_Type_2	Back_Substitution_PE_Type_2_Control, Back_Substitution_PE_Type_2_Data_Path	B.20

Tabla B.5: Relación de archivos den Verilog con el módulo Selector de Columna

Módulo en la Arquitectura	**Implementación en Verilog**	**Archivos necesarios**	**Figura**
Selector de Columna	Selector_Of_Column	Selector_Of_Column_Control, Selector_Of_Column_Data_Path, LIFO, LIFO_Parametrizable	B.21
Control de Selector de Columna	Selector_Of_Column_Control	Selector_Of_Column_State_Machine, Selector_Of_Column_Counter	B.22
Memoria de salida	LIFO_Parametrizable	dual_port_ram, lessOne, Counter, Control	B.23

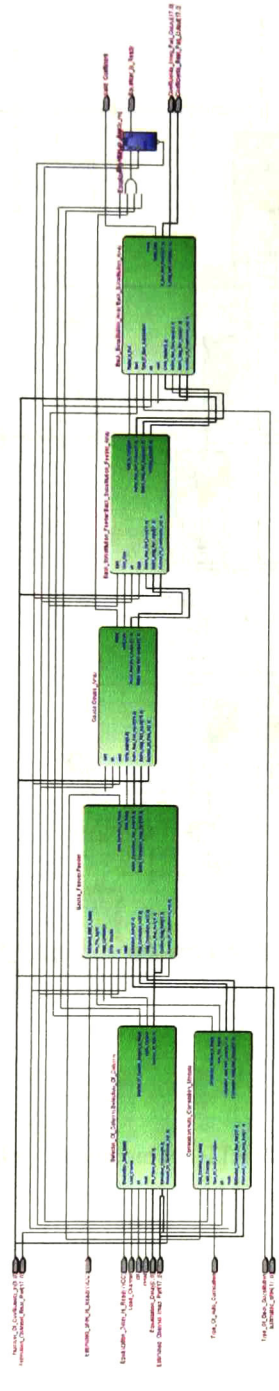


Figura B.1: RTL del módulo MMSE_Equalizer

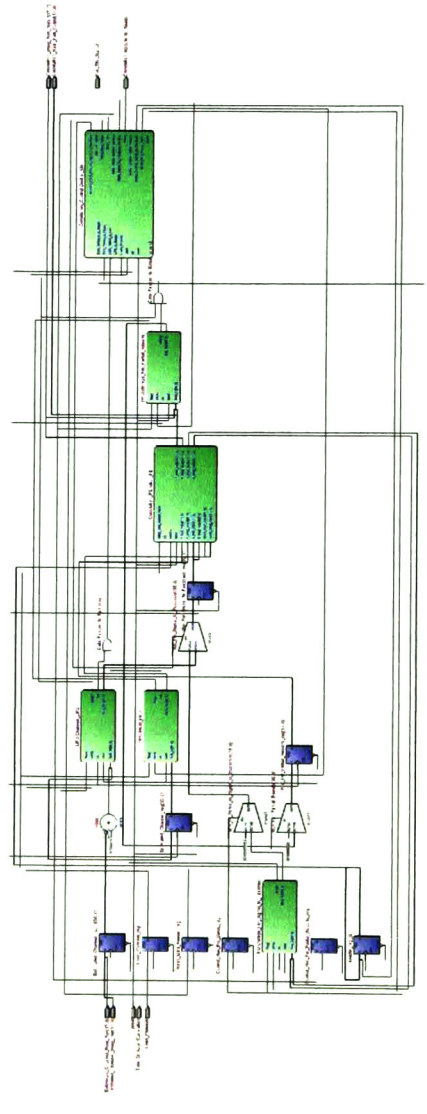


Figura B.2: RTL del módulo Correlation

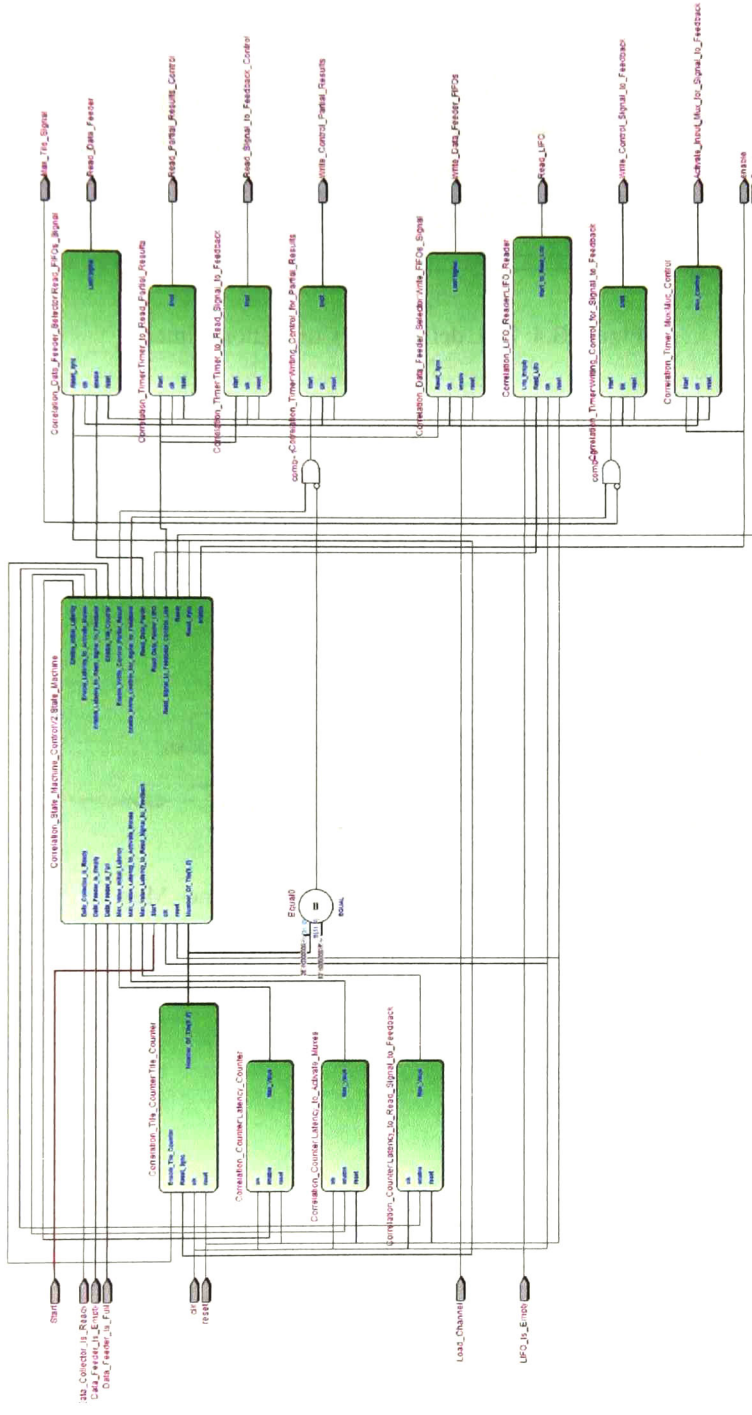


Figura B.3: RTL del módulo Correlation_Control

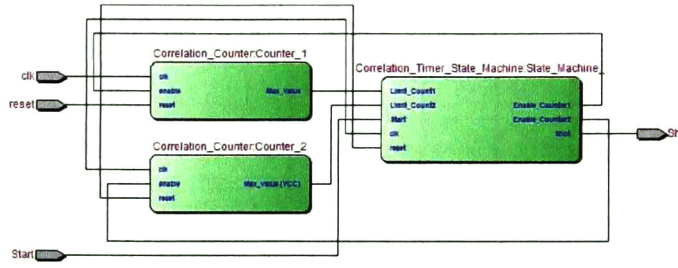


Figura B.4: RTL del módulo Correlation_Timer

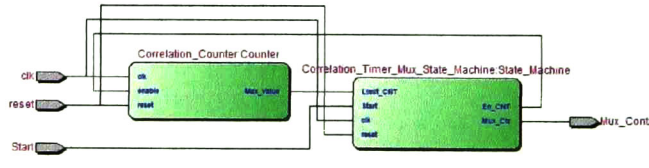


Figura B.5: RTL del módulo Correlation_Timer_Mux

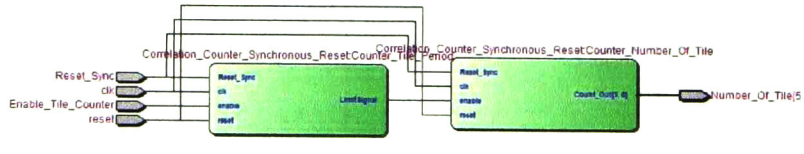


Figura B.6: RTL del módulo Correlation_Tile_Counter

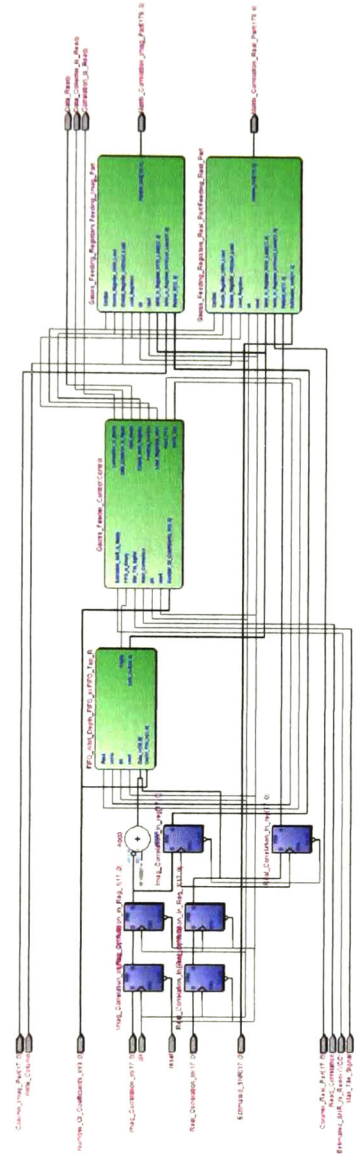


Figura B.7: RTL del módulo Gauss_Feeder

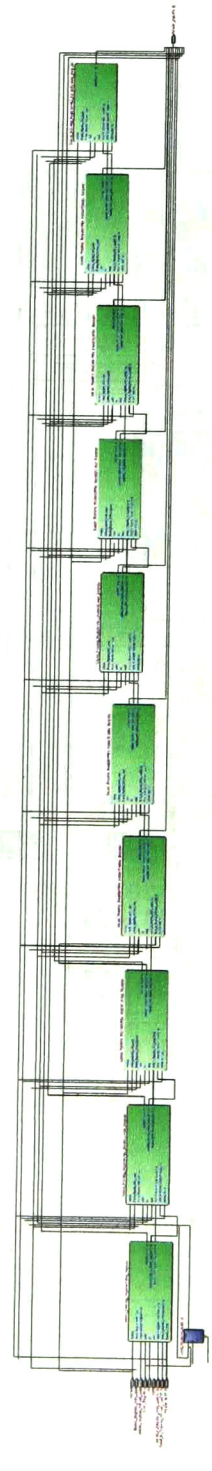


Figura B.8: RTL del módulo Gauss_Feeding_Registers

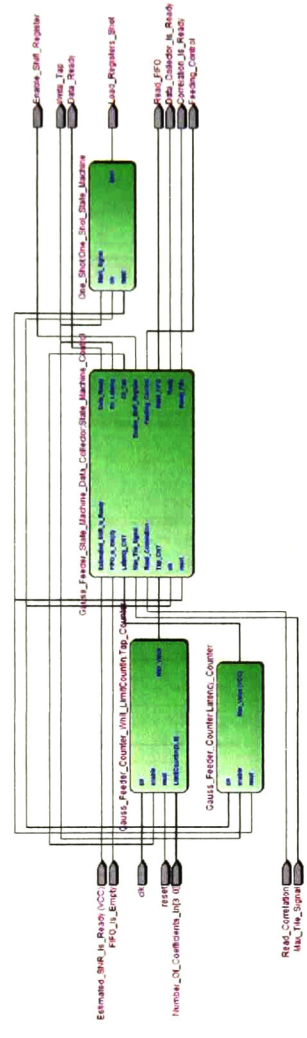


Figura B.9: RTL del módulo Gauss_Feeder_Control

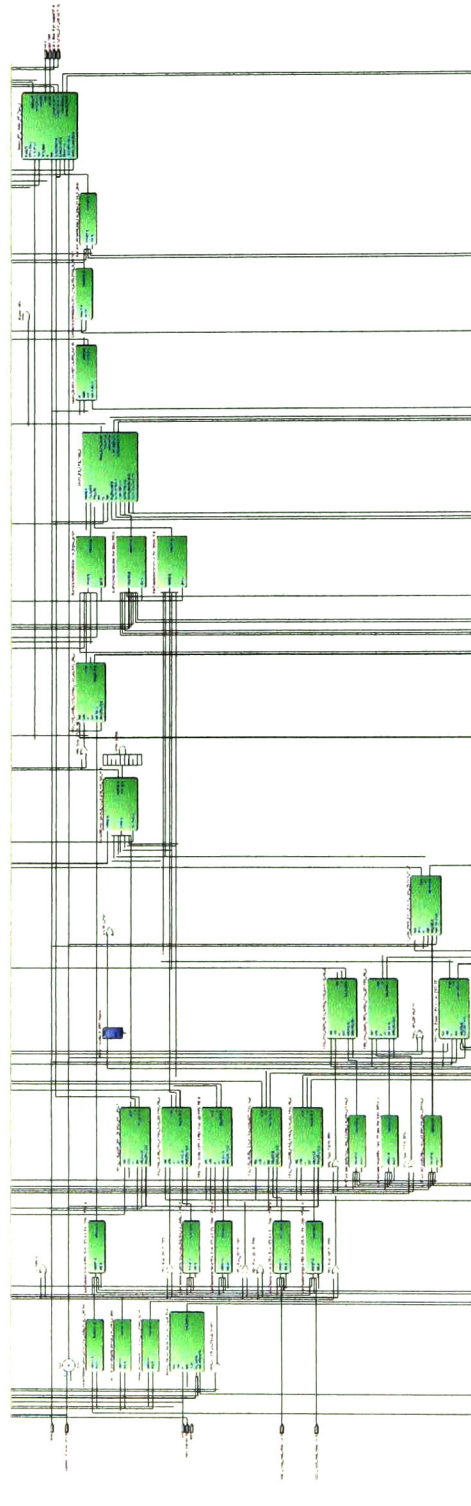


Figura B.10: RTL del módulo Gauss

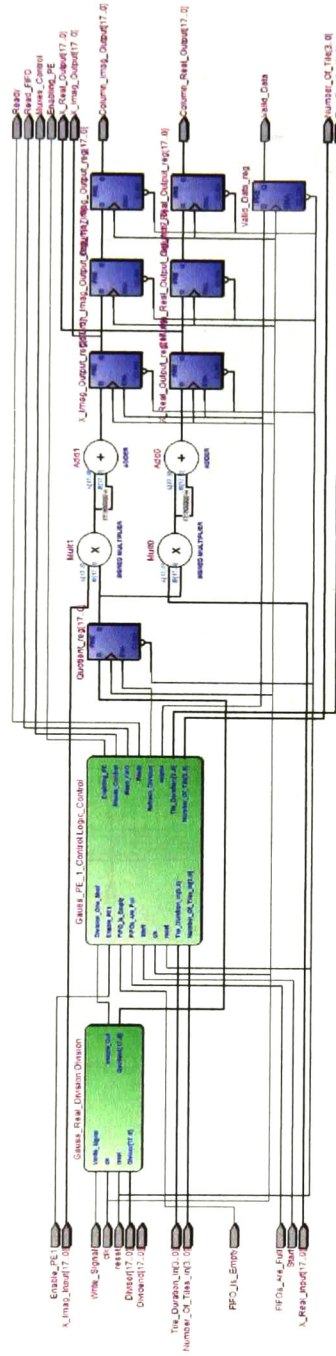


Figura B.11: RTL del módulo Gauss_PE_1

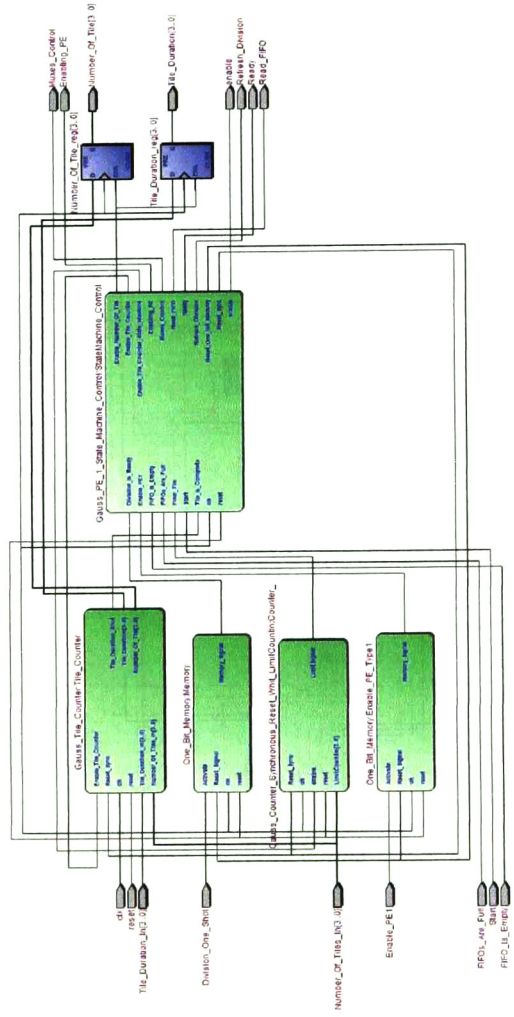


Figura B.12: RTL del módulo Gauss_PE_1_Control

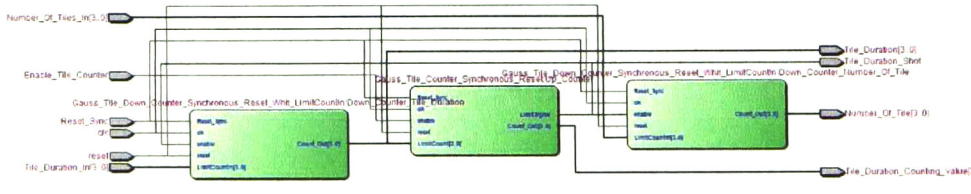


Figura B.13: RTL del módulo Gauss_Tile_Counter

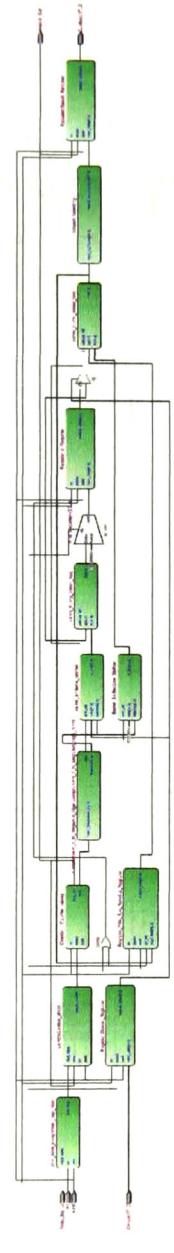


Figura B.14: RTL del módulo Gauss_Real_Division

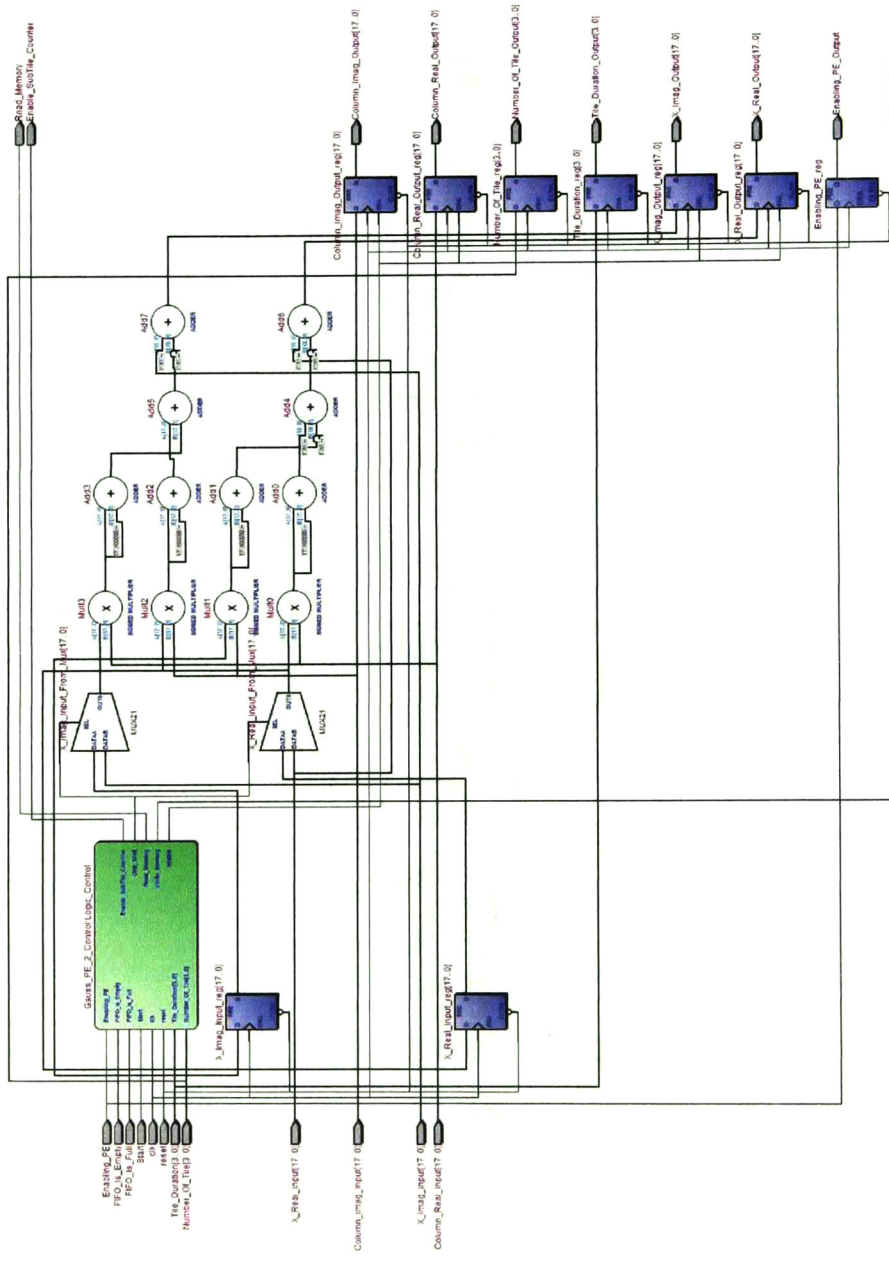


Figura B.15: RTL del módulo Gauss_PE_2

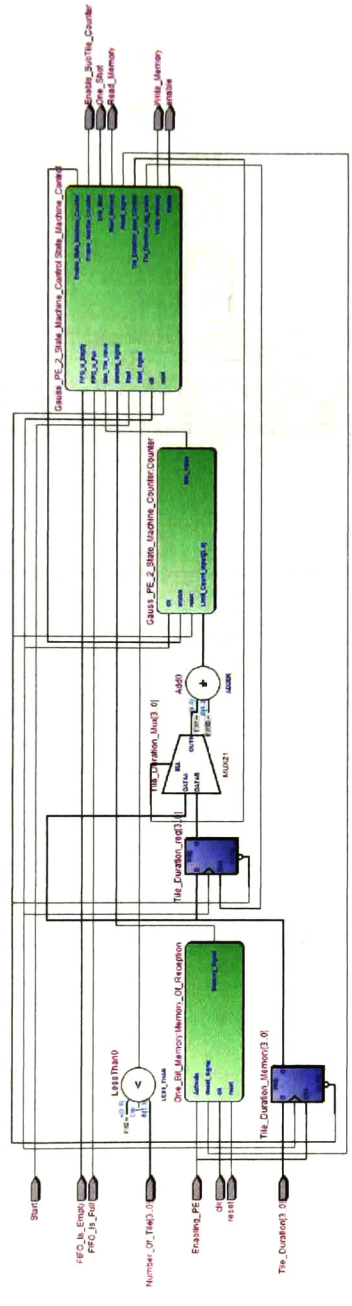


Figura B.16: RTL del módulo Gauss_PE_2_Control

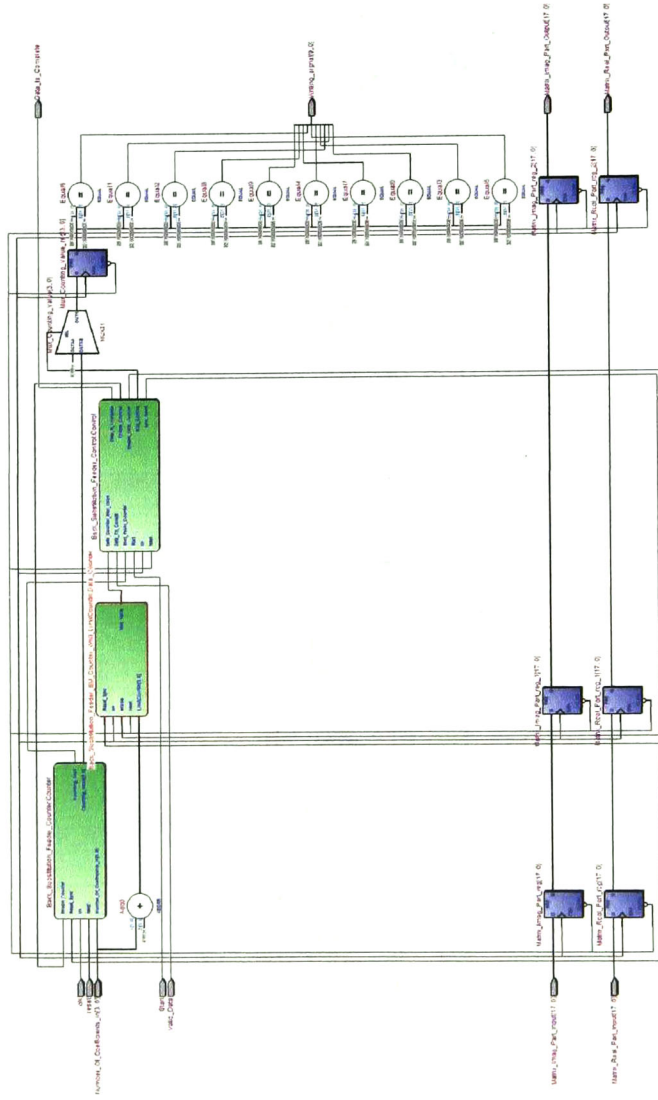


Figura B.17: RTL del módulo Back_Substitution_Feeder

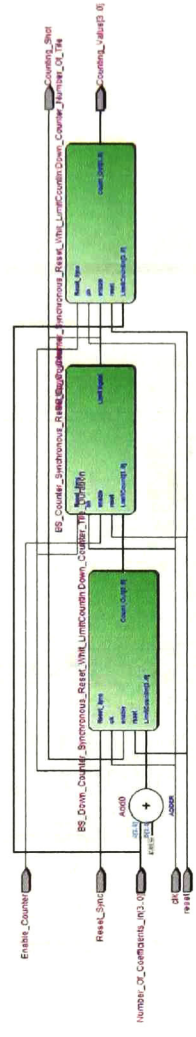


Figura B.18: RTL del módulo Back_Substitution_Feeder_Counter

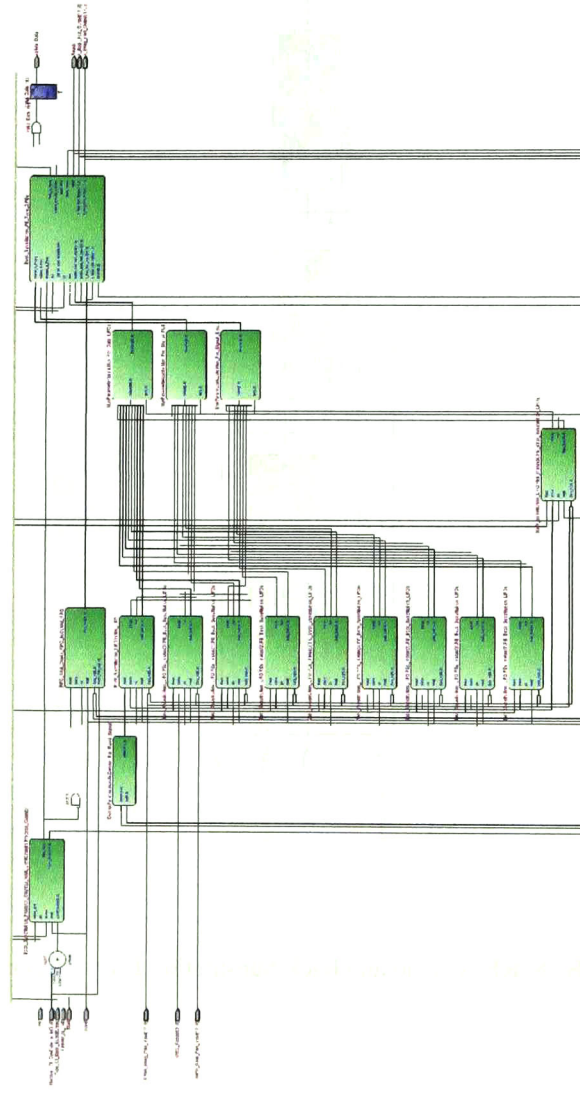


Figura B.19: RTL del módulo Back_Substitution_Array

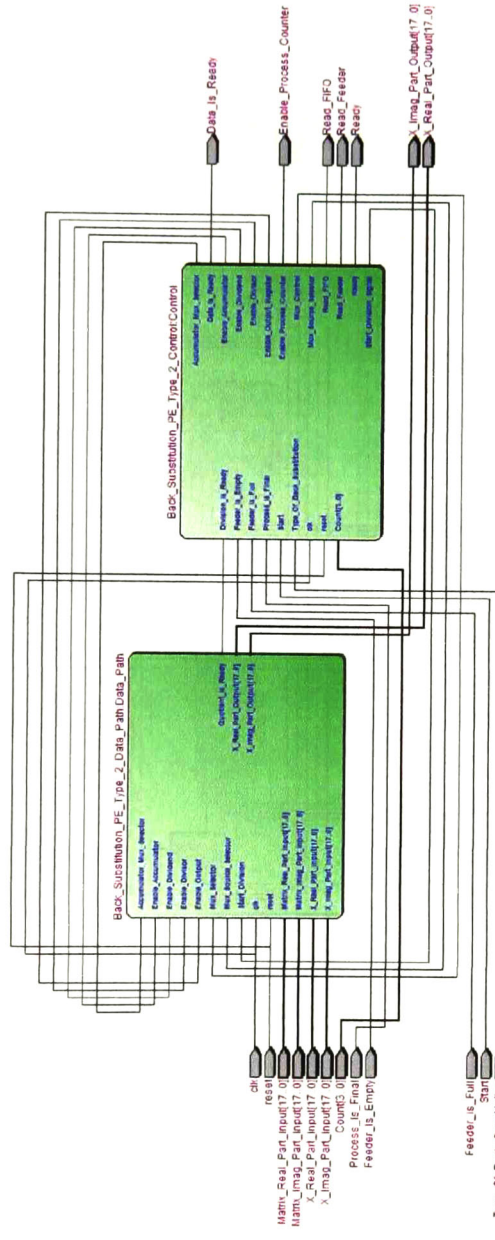


Figura B.20: RTL del módulo Back_Substitution_PE_Type_2

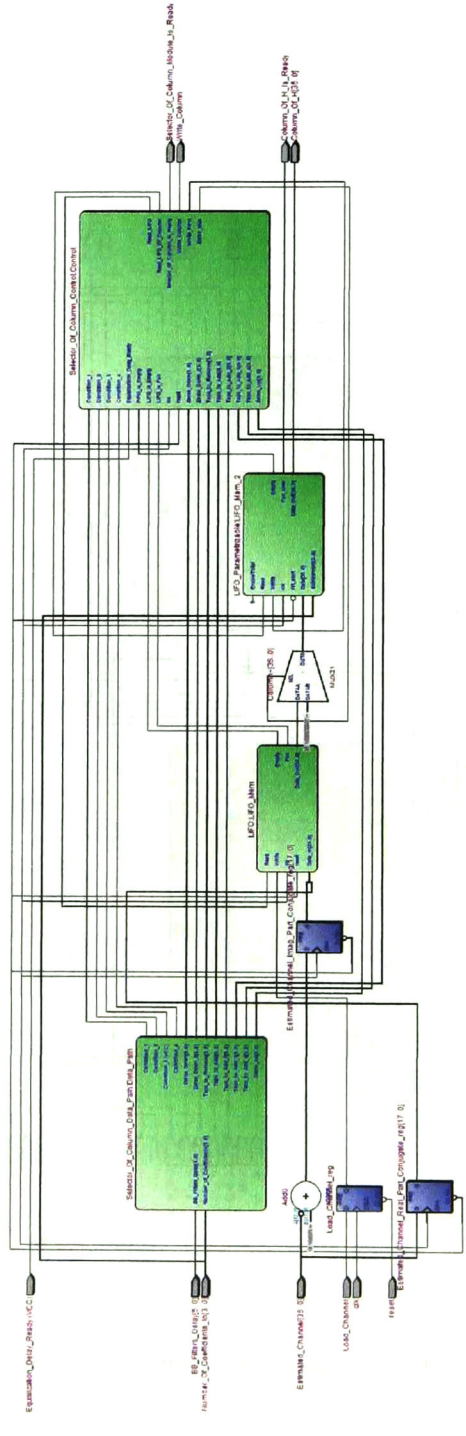


Figura B.21: RTL del módulo Selector_Of_Column

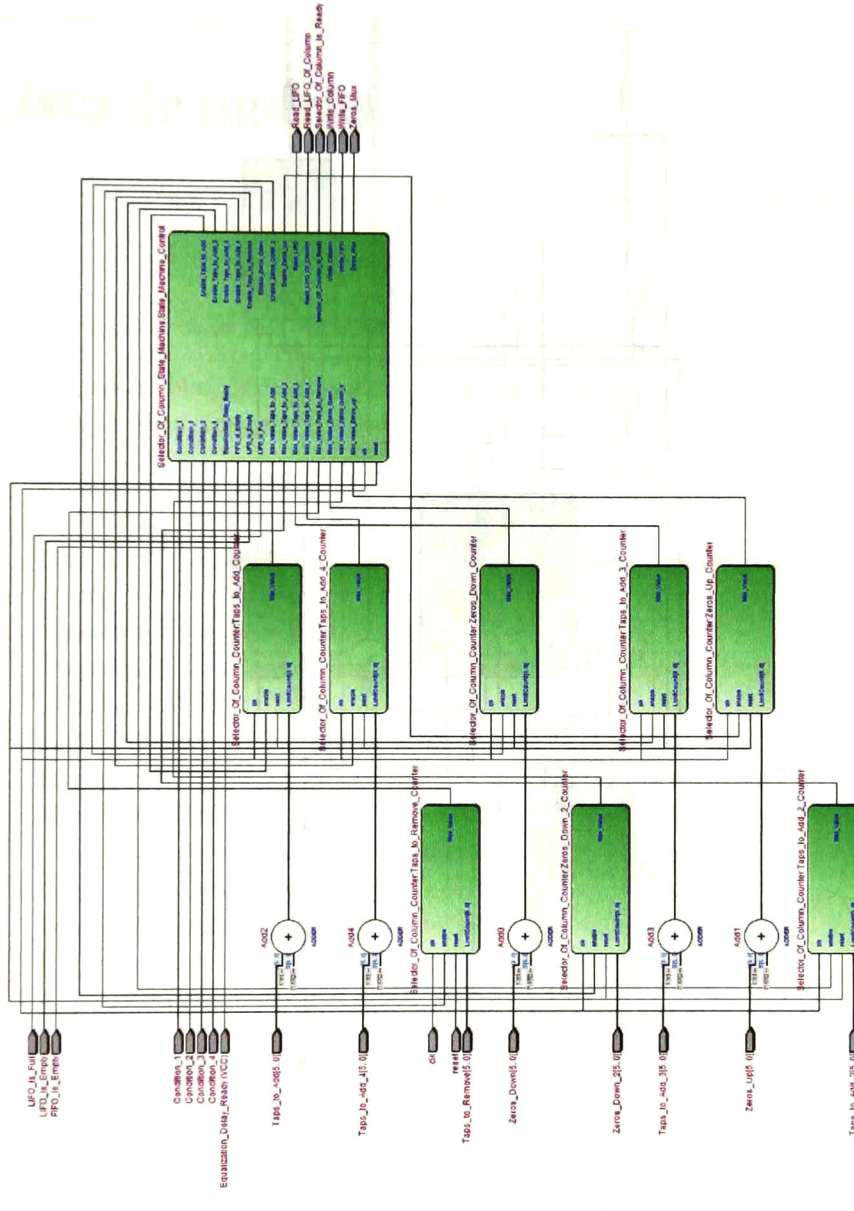


Figura B.22: RTL del módulo Selector_Of_Column_Control

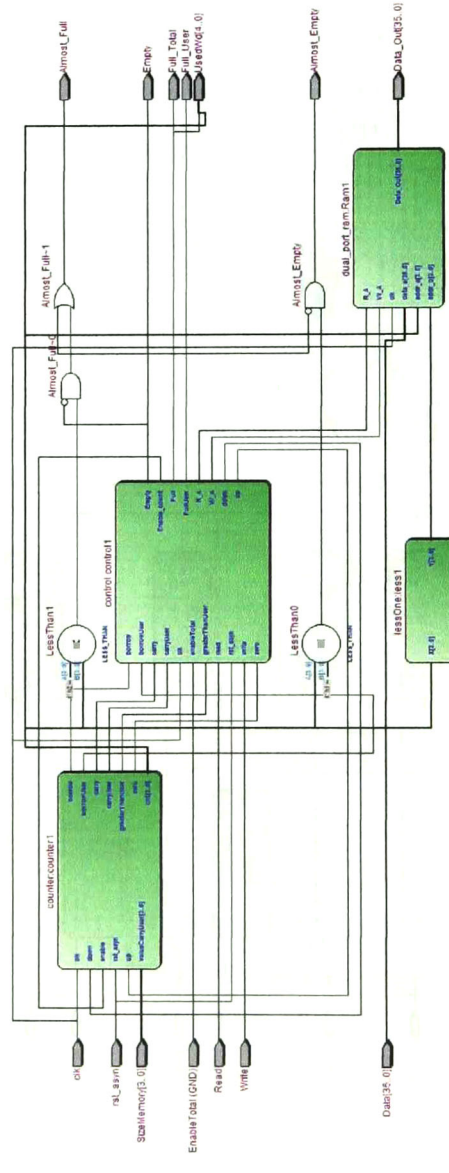


Figura B.23: RTL del módulo LIFO_Parametrizable

Lista de figuras

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL I.P.N.
UNIDAD GUADALAJARA

El Jurado designado por la Unidad Guadalajara del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional aprobó la tesis

Diseño e implementación de un igualador de canal MMSE con restricciones de área

del (la) C.

Alberto RODRÍGUEZ GARCÍA

el día 29 de Enero de 2014.

Dr. Deni Librado Torres Román
Investigador CINVESTAV 3B
CINVESTAV Unidad Guadalajara

Dr. Ramón Parra Michel
Investigador CINVESTAV 3B
CINVESTAV Unidad Guadalajara

Dr. Roberto Carrasco Álvarez
Profesor Investigador
Universidad de Guadalajara