UNIVERSIDAD POLITÉCNICA DE VALENCIA

Departamento de Ingeniería Electrónica

UNIVERSIDAD
POLITECNICA
DE VALENCIA

Implementación de Funciones Elementales en Dispositivos FPGA

TESIS DOCTORAL

Doctorando: Roberto Gutiérrez Mazón

Directores: Dr. Javier Valls Coquillat

Dr. Vicente Torres Carot

Julio - 2011 - Valencia

El hombre nada puede aprender……
sino en virtud de lo que ya sabe

Aristóteles

A mis padres,
hermanos,
Esther
y a María

Agradecimientos

Esta tesis ha sido realizada en el Grupo de Investigación de Sistemas Electrónicos Digitales (GISED) de la Universidad Politécnica de Valencia. Quisiera aprovechar esta oportunidad para agradecer a mis directores de Tesis, Dr. Javier Valls y Dr. Vicente Torres, por su ayuda, esfuerzo y paciencia infinita que me han prestado de forma desinteresada todos estos años para lograr que al final esta tesis llegue a buen puerto. También me gustaría agradecer la ayuda que me han prestado mis compañeros del área de electrónica y amigos de la Universidad Miguel Hernández y al resto de compañeros del grupo GISED.

Quiero mostrar mi agradecimiento a las instituciones que han permitido realizar este trabajo de investigación, como son el Ministerio de Educación y Ciencia, la Generalitat Valenciana y a la Universidad Politécnica de Valencia.

También quiero agradecer a mis padres, hermanos y demás familia, por estar ahí en todo momento apoyándome durante el transcurso de la realización de la tesis. Por último, y no menos importante agradecer a mi mujer Esther y a mi pequeña María por estar siempre a mi lado, confiar siempre en mi y darme esa sonrisa que necesitaba los días en los que no salían bien las cosas.

Muchas gracias a todos.

Resumen

En esta tesis doctoral se han diseñado arquitecturas hardware de algunos subsistemas digitales característicos de los sistemas de comunicaciones de elevadas prestaciones, buscando implementaciones optimizadas para dichos sistemas. El trabajo realizado se ha centrado en dos áreas: la aproximación de funciones elementales, concretamente el logaritmo y la arcotangente, y el diseño de un emulador de canal de ruido Gaussiano aditivo. Las arquitecturas se han diseñado en todo momento teniendo como objetivo lograr una implementación eficiente en dispositivos Field Programmable Gate Arrays (FPGAs), debido a su uso creciente en los sistemas de comunicaciones digitales de elevadas prestaciones. Para la aproximación del logaritmo hemos propuesto dos arquitecturas, una basada en la utilización de tablas multipartidas y la otra basada en el método de Mitchell sobre el que añadimos dos etapas de corrección: una interpolación lineal por rectas con pendientes potencias de dos y mantisa truncada, y una tabla para la compensación del error cometido en la interpolación por rectas. Una primera arquitectura para la aproximación de la atan(y/x) está basada en el cómputo del recíproco de x y en el cálculo de la arcotangente, utilizando básicamente tablas Look-up (LUT) multipartidas. Esta propuesta ya permite reducir el consumo de potencia con respecto a las mejores técnicas recogidas en la bibliografía, como las basadas en CORDIC. Una segunda estrategia para la aproximación de la atan(y/x) está basada en transformaciones logarítmicas, que convierten el cálculo de la división de las dos entradas en una sencilla resta y que hacen necesario el cómputo de atan(2^w). Esta segunda estrategia se ha materializado en dos arquitecturas, una primera en la que tanto el logaritmo como el cálculo de atan(2^w) se han implementado con tablas multipartidas, combinado además con el uso de segmentación no-uniforme en el cálculo de atan(2^w), y una segunda arquitectura que emplea interpolación lineal por tramos con pendientes potencias de dos y tablas de corrección. Los resultados obtenidos con esta estrategia mejoran los de la primera arquitectura comentada. Dos arquitecturas para la aproximación de la arcotangente y una de las del logaritmo han dado lugar a tres publicaciones en revistas internacionales. También se han propuesto varias arquitecturas para un generador de ruido blanco Gaussiano. Los diseños están basados en el método de la Inversión, concretamente aproximando la función de distribución acumulada inversa mediante interpolación polinómica y segmentación no-uniforme. Estas arquitecturas ofrecen a su salida una desviación estándar de $\pm 13.1\sigma$ y 13 bits fraccionarios, valores superiores a la práctica totalidad de los generadores hardware presentes en la bibliografía, empleando para ello, en comparación, menos recursos del dispositivo FPGA. Comparadas con las implementaciones del canal Gaussiano basadas en el método de la inversión presentadas por otros autores, nuestras arquitecturas consiguen una notable reducción de área eliminando parcial o completamente el *barrel-shifter*. Los resultados relativos al emulador de canal Gaussiano han sido enviados a una revista internacional, encontrándose en proceso de revisión.

Palabras Clave: Aproximación Funciones elementales, aproximación atan(y/x), aproximación logaritmo, comunicaciones inalámbricas, FPGA, AWGN

Abstract

In this thesis we have designed several hardware architectures of some typical digital subsystems for high performance communications systems, aiming at optimized implementations for these systems. The work has focused on two areas: the approximation of elementary functions, specifically the logarithm and arctangent, and the design of an emulator of additive Gaussian noise channel. The architectures have been designed at all times with the objective to achieve an efficient implementation in Field Programmable Gate Arrays devices (FPGAs), due to its increasing use in digital high performance communications systems. For the approximation of logarithm we have proposed two different architectures, one based on the use of multipartite table methods and the other based on the Mitchell's approximation with two correction stages: a piecewise linear interpolation with power-of-two slopes and truncated mantissa, and a LUT-based correction stage that corrects the piecewise interpolation error. A first architecture for the approximation of atan(y/x) is based on the computation of the reciprocal of x and the arctangent approximation, using multipartite Look-up tables (LUTs). This proposal reduces the power consumption compared to the best techniques described in the literature, such based on CORDIC. A second strategy for the approximation of atan(y/x) is based on logarithmic transformations, which transforms the calculation of the division of two inputs in a simple subtraction and requires the computation of atan(2^w). This second strategy has resulted in two architectures, a first in which both the logarithm and atan(2^w) have been implemented using multipartite LUTs, also combined with the use of non-uniform segmentation techniques in the calculation of atan(2^w), and a second architecture using a piecewise linear interpolation with power-of-two slopes and correction tables. The results obtained with this strategy improve the first architecture discussed. Two architectures for the approximation of arctangent and one for the logarithm have resulted in three publications in internationals journals. We also have proposed several architectures for a Gaussian white noise generator. The designs are based on the Inversion method, specifically approximating the inverse of cumulative distributions functions by polynomial interpolation and non-uniform segmentation. These architectures offer its output a standard deviation of $\pm 13.1\sigma$ and 13 fractional bits, higher values than practically all hardware generators present in the literature, employing, in comparison, fewer FPGA resources. Compared to Gaussian channel implementations based on the inversion method presented by other authors, our architectures achieve a significant reduction in area partially or completely eliminating the *barrel-shifter*. The results for Gaussian channel emulator have been sent to an international journal, being currently under review process.

Keywords: Elementary functions approximations, atan(y/x) approximation, logarithm approximation, wireless communication, FPGA, AWGN,

Resum

Aquesta tesis doctoral s'han dissenyat arquitectures hardware d'alguns subsistemes digitals característics dels sistemes de comunicacions d'elevades prestacions, buscant implementacions optimitzades per als dits sistemes. El treball realitzat s'ha centrat en dues àrees: l'aproximació de funcions elementals, concretament el logaritme i l'arcotangent, i el disseny d'un emulador de canal de soroll Gaussià additiu. Les arquitectures s'han dissenyat en tot moment tenint com a objectiu aconseguir una implementació eficient en dispositius Field Programmable Gate Arrays (FPGAs), a causa del seu ús creixent en els sistemes de comunicacions digitals d'elevades prestacions. Per a l'aproximació del logaritme hem proposat dos arquitectures, una basada en la utilització de taules multipartides i l'altra basada en el mètode de Mitchell sobre el qual afegim dues etapes de correcció: una interpolació lineal per rectes amb pendents potències de dos i mantissa truncada, i una taula per a la compensació de l'error comés per la interpolació per rectes. Una primera arquitectura per a l'aproximació de l'atan(y/x) està basada en el còmput del recíproc de x i en el càlcul de l'arcotangent, utilitzant bàsicament taules Look-up (LUT) multipartides. Esta proposta permet reduir el consum de potència respecte a les millors tècniques recollides en la bibliografia, com les basades en CORDIC. Una segona estratègia per a l'aproximació de l'atan(y/x) està basada en transformacions logarítmiques, que convertixen el càlcul de la divisió de les dues entrades en una senzilla resta i que fan necessari el còmput d'atan(2^w). Esta segona estratègia s'ha materialitzat en dues arquitectures, una primera en què tant el logaritme com el càlcul d'atan(2^w) s'han implementat amb taules multipartides, combinades amb l'ús de segmentació no-uniforme al càlcul d'atan(2^w), i una segona arquitectura que empra la interpolació lineal per trams amb pendents que són potències de dos i taules de correcció. Els resultats obtinguts amb aquesta estratègia milloren els de la primera arquitectura comentada. Dues arquitectures per a l'aproximació de l'arcotangent i una de les del logaritme han donat lloc a tres publicacions en revistes internacionals. També s'han proposat diverses arquitectures per a un generador de soroll blanc Gaussià. Els dissenys estan basats en el mètode de la Inversió, concretament aproximant la funció de distribució acumulada inversa per mitjà d'interpolació polinòmica i segmentació no-uniforme. Estes arquitectures oferixen a la seua eixida una desviació estàndard de $\pm 13.1\sigma$ i 13 bits fraccionaris, valors superiors a la pràctica totalitat dels generadors hardware presents en la bibliografia, emprant per a això, en comparació, menys recursos del dispositiu FPGA. Comparades amb les implementacions del canal Gaussià basades en el mètode de la inversió presentades per altres autors, les nostres arquitectures aconseguixen una notable reducció d'àrea eliminant parcial o completament el *barrel-shifter*. Els resultats relatius a l'emulador de canal Gaussià han sigut enviats a una revista internacional, trobant-se en procés de revisió.
.

Paraules Clau: Aproximació Funcions elementals, aproximació atan(y/x), aproximació logaritme, comunicacions sense fils, FPGA, AWGN

INDICE

LISTA DE FIGURAS

LISTA DE TABLAS

Capítulo 1. Introducción

1.1. Presentación y Contexto

La presente tesis se enmarca en el diseño de sistemas digitales de altas prestaciones, más específicamente sub-sistemas de aplicación en comunicaciones digitales. Dentro de estos sistemas la tendencia actual de implementación es el *Software Defined Radio* (SDR). El *SDR* se define como "una radio en el que alguna o todas las funciones de la capa física son definidas por software". Los sistemas SDR utilizan dispositivos digitales programables para realizar el procesado de señal necesario para transmitir y recibir la información. La principal idea detrás de los sistemas SDR es ampliar la parte digital de cualquier sistema de comunicación desplazándola hacia la antena. Esto significa que muchos de los componentes analógicos utilizados tradicionalmente en los trasmisores y receptores son sustituidos por Procesado digital de Señal implementado mediante dispositivos Field Programmable Gate Arrays (FPGAs) o Procesadores Digitales de Señal (DSPs). La gran ventaja de los sistemas SDR es la posibilidad de cambiar la configuración del sistema sin cambiar el hardware.

Los receptores SDR implementan algoritmos de procesado digital de la señal que requieren altas prestaciones. En estos algoritmos aparecen frecuentemente funciones elementales como $\log(x)$, $\exp(x)$, $1/x$, $\text{atan}(y/x)$, etc. Por ejemplo, las funciones $\log(x)$ y $\exp(x)$ se suelen usar en los controles automáticos de ganancia y la $\text{atan}(y/x)$ es típica en las estimaciones de frecuencia, fase y tiempo en los esquemas de sincronización *feed-forward*, en la detección de fase y frecuencias en esquemas realimentados [1] y en estimación de los desplazamientos en frecuencias de los receptores OFDM (Orthogonal Frequency Division Multiplexing) [2]. Otras áreas donde encontramos la $\text{atan}(y/x)$ son el campo de las aplicaciones biomédicas, concretamente en la detección de actividades cardiopulmonares humanas mediante radares Doppler de microondas [3], el procesado de imágenes utilizado en el análisis de patrones de bordes [4], los sensores interferométricos [5], los sistemas de cámaras en 3D [6], los sistemas de guerra electrónica [7], etc.

En esta tesis nos vamos a centrar en la implementación hardware de las funciones $\log(x)$ y $\text{atan}(y/x)$ apropiada para su utilización en sistemas SDR, concretamente su uso en algoritmos para realizar el procesado de señal en banda base en sistemas de comunicaciones de banda ancha. Estos algoritmos se caracterizan por trabajar a una tasa de una muestra o dos muestras por símbolo, lo que supone una frecuencia de trabajo de entre 20 y 50 MHz, no requiriendo una precisión muy elevada (usualmente entre 10 y 16 bits, dependiendo del sistema). Actualmente podemos encontrar en la literatura multitud de métodos y arquitecturas para la implementación hardware de estos operadores (basados en Tablas Look-Up (LUT), en aproximaciones polinómicas, en recurrencia de dígitos, CORDIC, etc.). En muchos casos se busca alcanzar una elevada precisión con mínimo consumo de recursos a costa de alcanzar velocidades de operación bajas en el operador, lo que no los hace útiles para los sistemas de comunicaciones digitales de banda ancha. En estos últimos sistemas los operadores se suelen implementar mediante el algoritmo CORDIC [8] o con métodos basados en tablas si la precisión que se requiere en los cálculos es baja. El algoritmo CORDIC tiene ciertas virtudes, como que sólo requiere de las operaciones suma-resta y desplazamiento para realizar el cómputo de las operaciones elementales y además su estructura hardware es muy regular, lo que simplifica enormemente el diseño del operador. Esto hace que CORDIC sea el primer candidato para su utilización en implementaciones hardware. Sin embargo sus puntos débiles son su alto consumo de potencia y que requiere mucha área si se realizan los cómputos con una alta precisión.

Otra aplicación en la que se requiere la computación de funciones elementales a alta velocidad es la generación de ruido aditivo Gaussiano (Additive White Gaussian Noise - AWGN), que es una

herramienta de gran utilidad para caracterizar sistemas de comunicaciones. La generación de muestras aleatorias Gaussianas se puede realizar con métodos digitales o con analógicos. A diferencia de los métodos digitales, los métodos analógicos permiten generar números verdaderamente aleatorios. Sin embargo, éstos son muy sensibles a cambios de temperatura y además proporcionan bajas tasas de muestras. Estos métodos suelen ser utilizados para aplicaciones de criptografía [9]. Por el contrario, los métodos digitales son más utilizados debido a su velocidad, flexibilidad y posibilidad de poder repetir la medida. Estos métodos generan secuencias pseudoaleatorias, a diferencia de las implementaciones analógicas. Estas secuencias se caracterizan por tener un periodo de repetición, por lo que deben escogerse para que dicho periodo sea lo suficientemente grande y no afecte a las medidas que se realicen.

Evaluar con herramientas software las prestaciones de un sistema de comunicaciones requiere simulaciones de elevada duración cuando la tasa de error de bit (BER) es pequeña. Esta duración se puede reducir considerablemente emulando el sistema completo (canal AWGN, sistema a evaluar y el sistema de medida de BER) gracias a uno o varios dispositivos FPGA. Un ejemplo en el que surge la necesidad de realizar simulaciones con un BER bajo lo encontramos en la evaluación de bloques de corrección de errores (Forward Error Correction - FEC) de los sistemas de comunicaciones ópticas. Evaluar las prestaciones de determinados decodificadores de códigos de matriz de Paridad de Baja Densidad (LDPC) precisa medir valores de BER menores de 10^{-10}. Para ello necesitamos generar más de 10^{12} símbolos para obtener al menos unos 100 errores y que los resultados de las simulaciones sean estadísticamente significativos. Esto puede requerir varios meses de computación en un ordenador personal de última generación.

En estos sistemas con muy baja probabilidad de error son los valores de ruido de gran amplitud los que provocan los errores. Estos valores se dan en las colas de la distribución Gaussiana. Por tanto, la correcta implementación de esas colas es la que asegurará la correcta evaluación del sistema a tasas de error bajas. Por tanto un generador Gaussiano que sea útil para estos sistemas debe ser capaz de generar muestras cuyas amplitudes sean varias veces mayores que la desviación típica de la distribución. Por ejemplo, en un sistema de comunicaciones que utilice la modulación BPSK (Binary Phase-Shift Keying) con un Eb/No=14 dB son las muestras con amplitud mayor de 7σ las que generan los errores, siendo σ la desviación típica de la distribución Gaussiana.

El reto que supone la implementación hardware de un simulador AWGN es el de conseguir emular una distribución Gaussiana lo más fiel posible en los valores grandes, es decir en las colas. Por ejemplo, en la Figura 1.1 se representa la función de densidad de probabilidad Gaussiana entre los valores 6σ y 10σ. Cabe destacar que la escala del eje de ordenadas es logarítmica y por tanto la función se caracteriza por una pendiente con un rango dinámico muy grande.

Otro problema que nos podemos encontrar cuando diseñamos un emulador de canal AWGN es su verificación, es decir, cuantificar la bondad de la distribución Gaussiana conseguida. En sistemas con un periodo elevado es computacionalmente muy costoso o imposible realizar un análisis completo del mismo. Existen varios métodos de verificación, como pueden ser la medida del factor de Curtosis de la distribución, la cual únicamente nos indicará en qué medida la distribución parece Gaussiana, o tests estadísticos complejos, como el cálculo de la función chi-cuadrado (χ^2) y el test de Anderson-Darling (A-D).

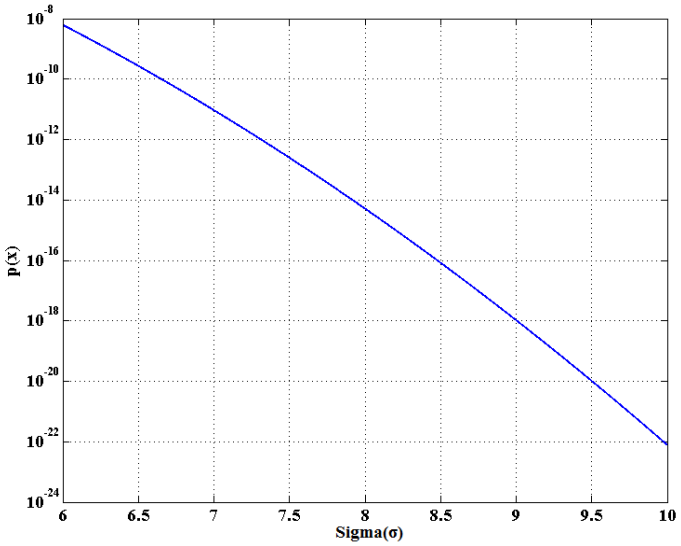


Figura 1.1 – Valores de la Función de Distribución de Probabilidad (PDF) de la distribución Gaussiana comprendida entre 6σ y 10σ. Representación en escala logarítmica del eje de las ordenadas.

1.2. Objetivos

El objetivo principal de esta tesis doctoral es el diseño eficiente de funciones elementales para su uso en sistemas de comunicaciones que utilizan como plataforma de implementación los dispositivos FPGA. Este objetivo general se concreta en los dos siguientes:

1. Diseño de arquitecturas de bajo coste y consumo de potencia para la implementación en dispositivos FPGA de las funciones elementales $\log_2(x)$ y $\text{atan}(y/x)$ cuyas especificaciones correspondan a las necesarias para su integración en un sistema de banda ancha típico.

2. Diseño de un emulador de canal AWGN de altas prestaciones basado en FPGA

Es importante resaltar que a diferencia de otros ámbitos de aplicación en los sistemas de comunicaciones no se suele necesitar una precisión elevada pero sí existen requerimientos como la frecuencia de trabajo, consumo de recursos, consumo de potencia y latencia de los operadores. Gran parte de los diseños propuestos en la bibliografía científica están enfocados a obtener operadores con gran precisión que no son de utilidad práctica para los actuales sistemas de comunicaciones. En los sistemas de comunicaciones de banda ancha el procesado de señal en banda-base se realiza típicamente a tasas de entre 20 y 40 Msps, y resoluciones de entre 10 y 16 bits a la salida de los operadores pueden ser suficientes.

Para abordar los objetivos anteriores se han empleado diversas herramientas. Las diferentes arquitecturas se han modelado con Matlab/Simulink/System Generator y en algunos casos directamente con lenguaje VHDL. En el caso concreto del canal AWGN se recurrió al lenguaje python y su librería mpmath para obtener precisiones en ciertos cálculos que Matlab no era capaz de proporcionar. Todas las arquitecturas se han implementado en dispositivos FPGA de Xilinx utilizando la herramienta ISE de este fabricante, si bien los resultados pueden ser fácilmente extrapolables a otros fabricantes. Las medidas de área y

frecuencia máxima de funcionamiento se han tomado de la herramienta ISE pero las medidas del consumo de potencia han sido realizadas utilizando una placa de pruebas Virtex-2 FG676 y el dispositivo XC2V3000-4 FG676.

1.3. Estructura de la Memoria

El trabajo realizado en esta tesis se ha dividido en dos partes: por un lado realizamos un estudio sobre aproximación de funciones en hardware y proponiendo una serie de arquitecturas para la aproximación eficiente de las funciones atan(y/x) y log(x). Por otro lado proponemos una implementación en hardware de un canal de ruido Gaussiano.

Dentro del bloque de aproximación de funciones, en el capítulo 2 se presentan los distintos métodos disponibles para la aproximación de funciones en hardware. Para todos los métodos expuestos se ha realizado un ejemplo de aproximación basado en la implementación de la atan(y/x) en FPGA. Para todas las arquitecturas se presenta una tabla con los resultados de la implementación.

En el capítulo 3 mostramos una nueva arquitectura para la aproximación de la atan(y/x) basada en la aproximación indirecta de la función mediante LUTs, en la cual ha primado sobre todo la reducción del consumo de potencia y del área de la implementación. Los resultados obtenidos para la arquitectura propuesta han sido comparados con los métodos presentados en el capítulo 2.

En el capítulo 4 proponemos otra arquitectura para la aproximación de la atan(y/x) en la que mediante la utilización de transformaciones logarítmicas evitamos la realización de la división (y/x), consiguiendo una reducción considerable del área necesaria para su implementación. Para la aproximación del logaritmo se han propuesto dos arquitecturas. La primera está basada en el uso de tablas multipartidas (*multipartite*) y la segunda en el uso de aproximación por múltiples rectas más una LUT que reduce el error cometido en la aproximación. Puesto que se emplea una transformación logarítmica para evitar el cómputo directo de la división se necesita realizar posteriormente el cálculo del antilogaritmo, un cálculo para el que también se propone un método optimizado y varias arquitecturas.

El capítulo 5 está relacionado con el segundo de los objetivos de la presente tesis doctoral. En él presentamos los distintos métodos propuestos en la bibliografía científica para la generación hardware de muestras aleatorias Gaussianas. En este capítulo analizamos los principales Generadores de Números Aleatorios Uniformemente Distribuidos (GNUD) así como los métodos más extendidos en los Generadores de Ruido Gaussiano (GNG). Por último, mostramos distintos tests estadísticos útiles para comprobar si un conjunto de muestras sigue una distribución Gaussiana.

El capítulo 6 proponemos varias arquitecturas para la generación de números aleatorios Gaussianos basadas en la utilización del método de la inversión, es decir empleando la Función de Distribución Acumulada Inversa (ICDF). En nuestros diseños se usa un esquema de segmentación no-uniforme de la ICDF e interpolación polinómica. Los resultados de implementación mostrados en este capítulo mejoran en muchos aspectos las prestaciones de las arquitecturas de GNGs presentes en la bibliografía.

Finalmente presentaremos las conclusiones sobre los resultados obtenidos y avanzaremos las futuras líneas de investigación.

BLOQUE 1.

APROXIMACION DE FUNCIONES ELEMENTALES

Capítulo 2.
Métodos para la Aproximación de Funciones Elementales.

En este capítulo se van a presentar los diferentes métodos disponibles en la literatura para la aproximación de funciones elementales en hardware. Para la aproximación de funciones elementales en hardware normalmente se divide la implementación en tres etapas: reducción de rango, aproximación de la función elemental y reconstrucción de rango. Este capítulo se centra en los métodos utilizados para la aproximación de las funciones, si bien se comentan brevemente los métodos utilizados para la reducción de rango. La reconstrucción de rango básicamente realizará la operación inversa de la realizada en la reducción de rango.

La atan(y/x) es un operador muy utilizado en las etapas de sincronización de los receptores digitales, independientemente de la modulación utilizada. Se han comparado los resultados de implementar la atan(y/x) con los distintos métodos presentados en el capítulo. Esta implementación se ha realizado utilizando dispositivos FPGAs. Los resultados de las distintas implementaciones nos servirán para poder comparar la bondad de las arquitecturas propuestas con respecto a los resultados expuestos en los capítulos 3 y 4.

2.1. Introducción

La aproximación de funciones elementales se puede llevar a cabo de dos maneras: podemos realizar las aproximaciones por software o directamente la implementación de la aproximación en hardware dedicado. Dentro de las aproximaciones por software podremos utilizar las instrucciones en coma flotante disponibles por los microprocesadores de propósito general [10], [11] o bien utilizar una serie de librerías matemáticas sobre las cuales se han implementado mediante lenguaje C una serie de funciones para el cálculo matemático de dichas funciones elementales. Dentro de las librerías más importantes tenemos CRLIBM [12], LIBULTIM, etc. Estas librerías tienen la desventaja de estar orientadas para su utilización en microprocesadores de propósito general y en muchos casos suelen ser demasiado lentas para poder ser utilizadas en aplicaciones en tiempo real, ahora bien, permiten generar resultados con una precisión muy elevada. Por ejemplo, la librería LIBULTIM realiza cálculos intermedios con hasta 800 bits de precisión de ahí la lentitud de la misma. Normalmente, estas librerías suelen utilizar una estrategia "Ziv's multinivel" [13]. Primero las funciones son evaluadas con una precisión relativamente baja y en el caso de que la función no pueda ser redondeada correctamente, se aumentará la precisión. Estas librerías se suelen utilizar en aplicaciones gráficas en 3D por ordenador, computación científica, aplicaciones multimedia, etc., donde no es tan importante la velocidad y sí lo es la precisión de los resultados.

Por otro lado, podemos implementar dichas aproximaciones directamente en hardware para poder acelerar el cálculo de las funciones y ser capaces de trabajar en aplicaciones en tiempo real. Los algoritmos utilizados en las implementaciones software suelen diferir de los utilizados en las implementaciones hardware, ya que en estos casos no son necesarias precisiones tan grandes y sí velocidades elevadas. En este trabajo de investigación nos centraremos exclusivamente en las implementaciones hardware. Para la aproximación de estas funciones será necesario el cumplimiento de varias de las propiedades que pasamos a comentar a continuación.

- Velocidad
- Precisión
- Un uso razonable de los recursos hardware utilizados (ROM/RAM, Área de silicio, y en muchos casos el consumo de potencia, etc.).
- Preservación de las propiedades matemáticas como pueden ser la monótocidad y la simetría.
- Límites del rango de trabajo de las funciones.

Para la aproximación de funciones elementales mediante hardware hay disponibles multitud de métodos, los cuales pueden ser clasificados en dos grandes grupos: métodos **No-iterativos** y métodos **Iterativos**. Los métodos no-iterativos son interesantes para realizar aproximaciones con bajas precisiones (valores en coma flotante simple o coma fija de hasta 32 bits). Los métodos iterativos serán necesarios cuando necesitemos más precisión (coma flotante doble o doble extendido y coma fija de 64 bits o más). A continuación presentamos los métodos utilizados para aproximar funciones elementales clasificadas en los dos grandes bloques comentados anteriormente:

- **Métodos No-iterativos**
  - Tablas Look-Up (LUTs) directas.
  - Interpolación Lineal.
  - Métodos basados en tablas (LUTs).
  - Aproximaciones por polinomios o aproximaciones racionales.
- **Métodos Iterativos**
  - Métodos multiplicativos.
  - Métodos Convergencia lineal.
  - Métodos recursivos digito a digito.
  - CORDIC.

Las aproximaciones a las funciones elementales no pueden ser calculadas exactamente con un número finito de bits. La precisión en la aproximación vendrá determinada por el error cometido en la aproximación y los errores de redondeo que ocurran durante el cálculo de la función [14]. Éste será uno de los factores más importantes en la elección de uno u otro método. La gran diferencia entre los métodos no-iterativos y los iterativos reside en que en los primeros pre-calculamos los valores de las aproximaciones y los almacenamos en memorias o los calculamos directamente mediante su aproximación por polinomios. En los segundos calculamos los valores de las aproximaciones mediante una serie de operaciones recursivas hasta obtener el valor correcto. Dentro de los métodos iterativos, un factor muy importante será la velocidad de convergencia hacia el resultado correcto.

Dependiendo de la precisión de trabajo que queramos obtener, velocidad de convergencia, área o consumo de potencia, deberemos utilizar un método u otro de aproximación. Estos métodos de aproximación no tienen por qué ser excluyentes ya que muchas veces podremos combinar distintos métodos sin ningún problema, obteniendo las ventajas de los dos métodos. Muchas de las implementaciones prácticas están basadas en combinaciones de varios métodos. Por ejemplo, en la operación del cálculo del recíproco implementada en el microprocesador AMD-K7 [15], primero se parte de un valor inicial (semilla) almacenado en una LUT, este valor es almacenado con una precisión de pocos bits, y posteriormente mediante un algoritmo basado en iteración de funciones se obtiene el valor del recíproco aproximado a la precisión pedida, de esta manera se mejora la velocidad de convergencia hacia el resultado.

Para la realización hardware de la aproximación de funciones elementales de una manera óptima seguiremos una serie de pasos propuestos por P.T.P. Tang [16]. Estos tres pasos son:

- **Reducción de Rango:** Reducción del intervalo de trabajo de la entrada [a, b] a otro menor [a', b']. Esta reducción de rango normalmente está basada en propiedades como la simetría, periodicidad, etc.
- **Aproximación:** Calculamos el valor de la función elemental aproximada dentro del rango de entrada reducido [a', b'].
- **Reconstrucción:** Extendemos el valor convenientemente hacia en rango de entrada inicial [a, b].

Como podemos apreciar, la aproximación a la función elemental será realizada sobre un rango menor que el original, con lo cual tendremos que añadir etapas de pre-procesado y post-procesado para reducir el dominio de la entrada y expandir el resultado de la aproximación, respectivamente. Normalmente evaluaremos una función elemental *f(x)* para un rango de entrada [a, b] y para una precisión determinada.

2.1.1. Reducción de Rango

Las aproximaciones de las funciones elementales que normalmente utilizamos trabajan con un rango de entrada muy pequeño, tal y como se muestra en la tabla 2.1.

f(x)	**Rango aproximación**
$1/x$	*[1, 2)*
$\sqrt{x}$	*[1, 2)*
sin(x)	*[0, 1)*
2^x	*[0. 1)*
$log_2(x)$	*[1, 2)*
ln(x)	*[1, 2)*
e^x	*[0, ln(2))*
$1/\sqrt{x}$	*(0.5, 1]*

Tabla 2.1 - Rango de valores de entrada óptimos para la aproximación de las funciones elementales más utilizadas.

Para poder evaluar una función *f(x)* para cualquier valor de *x* necesitamos de poder obtener esa función a partir de algún valor *g(x*)* el cual tiene un determinado rango de entrada mucho menor.

En la práctica tenemos dos clases de reducciones:

- ***Reducción aditiva***. *x** será igual a *x-kC*, donde *k* es un entero y C una constante. Por ejemplo, para las funciones trigonométricas, C será un múltiplo de $\pi/4$.
- ***Reducción Multiplicativa***. *x** será igual a x/C^k, donde *k* es un entero y C una constante. Por ejemplo, para la función logaritmo, una buena elección de C será la base del sistema numérico.

• Cálculo de la aproximación a la función e^x

Asumiendo que vamos a trabajar en base-2 y el rango de entrada utilizado por la aproximación [0, ln(2)), elegiremos *C=ln(2)* y el cálculo del valor de la exponencial será el siguiente:

- Calcularemos $x^* \in [0, \ln(2))$ y *k* para que $x^* = x - k\ln(2)$.
- Calcularemos $g(x^*) = e^{x^*}$.
- Calcularemos $e^x = 2^k \cdot g(x^*)$.

Al trabajar en base binaria la última multiplicación se puede realizar mediante desplazamientos.

• Cálculo de la función coseno (seno)

En este caso asumiremos que el rango de entrada de nuestra función de aproximación del coseno será $\left[-\frac{\pi}{4},\frac{\pi}{4}\right]$ y elegiremos $C=\frac{\pi}{2}$. Los pasos a seguir serán los siguientes:

- Calcularemos $x^* \in \left[-\frac{\pi}{4},\frac{\pi}{4}\right]$ y k para que $x^* = x - k\frac{\pi}{2}$.
- Calcularemos $g(x^*,k)$

$$g(x^*,k)=\begin{cases}\cos(x^*) & \text{si mod k=0}\\ -\sin(x^*) & \text{si mod k=1}\\ -\cos(x^*) & \text{si mod k=2}\\ \sin(x^*) & \text{si mod k=3}\end{cases} \tag{2.1}$$

- Calcularemos $\cos(x)=g(x^*,k)$

• Cálculo de la aproximación a la función logaritmo

En este caso asumiremos que el rango de entrada de nuestra función de aproximación del logaritmo neperiano será [1, 2), trabajando siempre con números positivos y siempre en base binaria. Elegiremos C=2 y los pasos a seguir serán los siguientes:

- Calcularemos $x^* \in [1,2]$ y k para que $x^* = x/2^k$.
- Calcularemos $g(x^*,k)=\ln(x^*)$.
- Calcularemos $\ln(x)=g(x^*,k)+k\ln(2)$.

La reducción multiplicativa únicamente será utilizada cuando se quieran calcular logaritmos y raíces cuadradas. Para estas reducciones multiplicativas utilizaremos siempre la base del sistema numérico.

La reducción de rango es ampliamente utilizada, especialmente en CORDIC y en los microprocesadores en coma flotante. Además disponemos de métodos más complejos para la reducción de rango. Cody y Waite en [17] proponen un primer método sobre el cual se trabaja con dos constantes C_1 y C_2. Lefévre y Muller en [18] definen un método para la realización de la reducción de rango "*on the fly*", solapando el cálculo de la aproximación con la recepción de los bits de entrada para sistemas que trabajan en aritmética serie. Payne y Hanek's en [19] presentan un método optimizado para trabajar con tamaños de palabra IEEE de doble precisión. Daumas y otros en [20] proponen un método de reducción de rango modular (MRR) en el cual se realizan varias reducciones.

2.2. Aproximación de Funciones Elementales

Para la compresión y comparación de los distintos métodos de aproximación de funciones descritas en este capítulo vamos a realizar un ejemplo práctico de cada método. Para ello vamos aproximar la función atan(y/x). Para las implementaciones siguientes hemos optado por trabajar con las especificaciones de un sistema de banda ancha estándar. Las entradas estarán normalizadas [0, 1), de esta manera simplificamos la

implementación del circuito ya que únicamente queremos comparar resultados entre los distintos métodos de aproximaciones de funciones. Además vamos a añadir la restricción de que el valor de la entrada *x* siempre será mayor que el valor de la entrada *y*. La fase de salida con la que trabajaremos estará normalizada a uno. Los tamaños de palabra utilizados en las implementaciones serán indicados en las figuras utilizando el formato [N. F], siendo N el numero de bits utilizados y F el numero de bits de la parte fraccional. Todas las implementaciones las realizaremos mediante la utilización de FPGAs, concretamente dispositivos de la familia Xilinx Virtex-2 XC2V3000-4FG676 de la empresa Xilinx. Para el diseño de los distintos componentes hemos trabajado a partir de modelos en VHDL y Xilinx System Generator. Para la Síntesis y Place & Route hemos utilizado la aplicación Xilinx ISE 10.1 SP3.

A continuación pasamos a revisar las características más importantes de cada método para la aproximación de funciones elementales. Además, estudiaremos los resultados de implementación modificando el tamaño de entrada/salida y segmentando el operador completamente.

2.2.1. CORDIC

CORDIC es el acrónimo de COordinate Rotations Digital Computer. Este algoritmo fue desarrollado por Volder [21] en 1959 para el cálculo de funciones trigonométrica. Posteriormente este método fue generalizado añadiendo funciones logarítmicas, funciones exponenciales e hiperbólicas, multiplicaciones, divisiones y raíces cuadradas por Walther [22] que trabajaba en la empresa Hewlett Packard.

CORDIC no es la manera más fácil para la realización de multiplicaciones o el cálculo de logaritmos/exponenciales, pero el mismo algoritmo permite el cálculo de muchas funciones elementales utilizando operaciones muy básicas, lo cual es una característica muy importante desde el punto de vista de las implementaciones hardware. CORDIC ha sido implementado en multitud de aplicaciones, desde calculadoras como puede ser HP35 de Hewlett Packard hasta el coprocesador aritmético utilizado en el microprocesador Intel 80486. Además es utilizado en muchas aplicaciones de procesado de señal, como pueden ser la Transformada Discreta del Coseno, la Transformada Discreta de Hartley, realización de filtrados o para la resolución de sistemas lineales.

Este método está basado en una serie de ecuaciones iterativas basadas en desplazamientos y sumas y búsquedas en tablas. El método presentado por Volder está basado en las siguientes ecuaciones (2.2), las cuales permiten rotar un vector en el plano cartesianos por un ángulo θ.

$$\begin{aligned} x' &= x\cos(\theta) - y\sin(\theta) \\ y' &= y\cos(\theta) + x\sin(\theta) \end{aligned} \tag{2.2}$$

El algoritmo CORDIC generalizado consiste en las siguientes ecuaciones iterativas:

$$\begin{aligned} x_{n+1} &= x_n - m d_n y_n 2^{-\sigma(n)} \\ y_{n+1} &= y_n + d_n x_n 2^{-\sigma(n)} \\ z_{n+1} &= z_n - \omega_{\sigma(n)} \end{aligned} \tag{2.3}$$

donde las constantes m, d_n, ω_n y $\sigma(n)$ dependerán de la función a aproximar. En la tabla 2.2 se pueden ver los distintos valores de las constantes.

Para usar estas ecuaciones recursivas debemos utilizar unos valores de inicio x_1, y_1 y z_1 apropiados como se puede ver en la tabla 2.2. Para todos los casos, estos valores iníciales deben estar restringidos a un determinado rango para asegurarnos la convergencia. Como en todo procedimiento iterativo, una de las variables tenderá hacia cero y la otra hacia el valor aproximado deseado.

El operador CORDIC normalmente trabajará en uno de los dos modos posibles [23]. El primero, llamado modo rotación por Volder, rota la entrada por un ángulo especificado (ese valor también es una entrada). En este modo, la variable z_n es inicializada con el ángulo de rotación deseado. La decisión de rotación en cada iteración se hace para reducir la magnitud del ángulo residual en la variable acumulador. La decisión de cada iteración se basa en el signo del ángulo residual de cada iteración.

El otro modo, llamado vectorización, rota el valor de entrada hacia el eje *x*, mientras va almacenando el ángulo requerido para hacer esa rotación. El resultado de la operación de vectorización es el ángulo rotado y la magnitud escalada del vector original (la componente *x* del resultado). Este modo trabaja intentando minimizar la componente *y* del vector residual en cada iteración. El signo del residuo *y* es usado para determinar la dirección de la siguiente rotación.

Tipo	**m**	ω_k	**d_n=signo de z_n Modo Rotación**	**d_n=-signo de y_n Modo Vectorización**	$\sigma(n)$
Circular	1	atan 2^{-k}	$x_n \to K(x_0 \cos z_0 - y_0 \sin z_0)$ $y_n \to K(y_0 \cos z_0 - x_0 \sin z_0)$ $z_n \to 0$	$x_n \to K\sqrt{x_0^2 + y_0^2}$ $y_n \to 0$ $z_n \to z_0 - \arctan \frac{y_0}{x_0}$	$\sigma(n) = n$
Lineal	0	2^{-k}	$x_n \to x_0$ $y_n \to y_0 + x_0 z_0$ $z_n \to 0$	$x_n \to x_0$ $y_n \to 0$ $z_n \to z_0 - \frac{y_0}{x_0}$	$\sigma(n) = n$
Hiperbólico	-1	$\tanh^{-1} 2^{-k}$	$x_n \to K'(x_1 \cosh z_1 - y_1 \sin z_1)$ $y_n \to K'(y_1 \cos z_1 - x_1 \sin z_1)$ $z_n \to 0$	$x_n \to K'\sqrt{x_1^2 - y_1^2}$ $y_n \to 0$ $z_n \to z_1 - \tanh^{-1} \frac{y_1}{x_1}$	$\sigma(n) = n - k$ [1]

Tabla 2.2 – Valores de las constantes utilizadas en el operador CORDIC

Los algoritmos de rotación y vectorización están limitados a ángulos de rotación entre $-\pi/2$ y $\pi/2$. Esta limitación es debida al hecho de usar 2^0 para la tangente en la primera iteración. Para aumentar el ángulo de rotación a valores mayores de $\pi/2$ es necesario realizar una rotación adicional. A partir de la modificación propuesta por Walther, se permite ampliar el número de funciones elementales aproximadas. Por ejemplo, podemos calcular el logaritmo neperiano y la raíz cuadrada por medio de la relación (2.4) y (2.5).

$$\ln(x) = 2\tanh^{-1}\left(\frac{x-1}{x+1}\right) \tag{2.4}$$

$$\sqrt{x} = K'\sqrt{\left(x + \frac{1}{4K'^2}\right)^2 - \left(x - \frac{1}{4K'^2}\right)^2} \tag{2.5}$$

[1] *K* será el mayor entero que cumpla la condición $3^{k+1} + 2k - 1 \le 2n$

El mayor inconveniente del algoritmo CORDIC es su convergencia lineal para obtener el resultado deseado y este tiempo será proporcional a la precisión que se desee alcanzar en el cálculo.

- **Ejemplo de Implementación**

Para el primer ejemplo de implementación hemos realizado la aproximación de la atan(y/x) mediante CORDIC trabajando en modo vectorización. Si la variable z_n es inicializada a cero, las variables x_n y y_n las inicializaremos con los valores x e y. El CORDIC implementado está basado en un core-IP [24] diseñado por Xilinx y generado el hardware mediante la herramienta System Generator de Xilinx. El core-IP generado se ha paralelizado completamente permitiendo generar resultados síncronamente (el dato es válido en cada ciclo de reloj), aumentando el área requerida. El numero de etapas utilizadas para el cálculo de CORDIC ha sido de 12 (el algoritmo CORDIC necesita aproximadamente realizar una operación de desplazamiento-suma/resta por cada bit de precisión a la salida. En la figura 2.1 podemos ver un esquema del bloque CORDIC implementado.

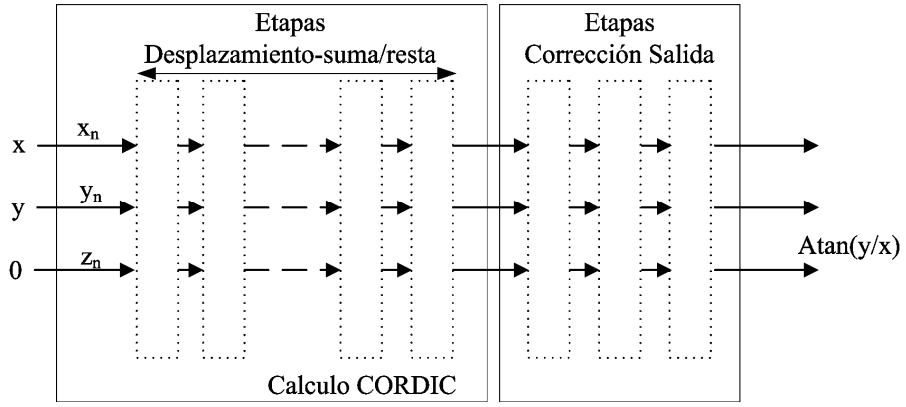


Figura 2.1 – Distribución del numero de etapas necesarias para la implementación del operador CORDIC mediante el Core-IP de Xilinx.

Los resultados de implementación obtenidos para las precisiones definidas son los indicados en la tabla 2.3. Hemos variado el número de etapas de segmentación del bloque que realiza el cálculo del CORDIC (registrando las distintas etapas de desplazamiento-suma/resta para reducir el camino crítico) hasta 12 etapas de registros. Para el caso de no registrar ninguna etapa la latencia mínima del core-IP es de 3 ciclos de reloj, ya que las entradas y salidas están registradas, además de dos etapas intermedias.

Segmentación	**0**	**2**	**4**	**6**	**12**
Slices	339	340	340	342	346
LUT4	510	513	513	512	511
F.F.	91	183	246	334	569
Block RAM			0		
Fmax (MHz)	20.3	52.7	73.5	98.7	165.4
Latencia	3	5	7	9	15
Throughput (Msps)	20.3	52.7	73.5	98.7	165.4

Tabla 2.3 – Resultados de implementación de la aproximación de la atan(y/x) mediante CORDIC y modificando el numero de etapas de segmentación.

Como podemos apreciar, conforme vamos aumentando el número de etapas de segmentación aumentamos el número de biestables utilizados y permitiendo aumentar la frecuencia máxima de funcionamiento. Al realizar una implementación completamente paralela, el *throughput* es el mismo que la frecuencia máxima de funcionamiento y por cada ciclo de reloj obtenemos un dato valido. Podemos apreciar que la implementación completamente segmentada puede trabajar fácilmente con tasas de datos de hasta 165 Mbps. El gran inconveniente será la latencia del operador que dependerá de la precisión requerida. Esta latencia no será muy importante en sistemas donde la cadencia de datos es constante. En la tabla 2.4 podemos ver los resultados de implementación variando el tamaño de los operandos de entrada/salida y únicamente utilizando dos etapas de segmentado. A partir de los resultados obtenidos, tenemos que aumentar el tamaño de las muestras de 12 a 22 bits, implica duplicar aproximadamente el área necesaria y por consiguiente, la tasa de datos ha disminuido a la mitad. Vemos como el operador CORDIC cumple las especificaciones de velocidad ya que superamos los 20Msps requeridos en nuestras especificaciones.

Precisión	**12**	**14**	**16**	**18**	**20**	**22**
Slices	340	387	413	514	577	712
LUT4	513	620	684	784	955	1160
F.F.	183	208	215	227	244	262
Block RAM	0					
Fmax (MHz)	52.7	47.9	42.1	36.1	32.9	28.1
Latencia	5					
Throughput (Msps)	52.7	47.9	42.1	36.1	32.9	28.1

Tabla 2.4 - Resultados de implementación de la aproximación de la atan(y/x) mediante CORDIC y modificando la precisión de salida de la aproximación.

2.2.2. Métodos de Convergencia Lineal

En los métodos de convergencia lineal las aproximaciones de las funciones elementales están basadas en la utilización de funciones auxiliares que nos permiten converger hacia el valor de la función. Para controlar esta convergencia la función auxiliar tenderá hacia '1' o '0'. Dependiendo del valor hacia el que converja la función auxiliar el método se denominará "de normalización multiplicativa" si tiende a '1' y "de normalización aditiva" si tiende a '0' [25].

En estos algoritmos de convergencia lineal las operaciones utilizadas son principalmente multiplicaciones por potencias de la base de trabajo (desplazamientos) y sumas.

2.2.2.1. Normalización Multiplicativa

Este método es utilizado para la aproximación de multitud de funciones, como puede ser el recíproco, raíz cuadrada, inversa de la raíz cuadrada, logaritmo, etc. En todos los casos, tenemos una secuencia que convergerá hacia '1' y ésta controlará la convergencia de otra secuencia que tenderá hacia el resultado.

Este algoritmo iterativo básicamente consiste en la determinación de una secuencia Q(n) tal que la secuencia *X(n)* (2.6) converja hacia '1'

$$X(n+1) = x(n) \cdot Q(n), \tag{2.6}$$

partiendo del valor inicial $X(0) = x$. Para conseguir una convergencia lineal definiremos

$$Q(n) = (1 + b_n \cdot r^{-n}), \tag{2.7}$$

donde *r* es la base de trabajo del algoritmo. Como podemos ver esta normalización multiplicativa genera un "producto continuo" del valor de recíproco de *x* (2.8).

$$\frac{1}{x} \approx \prod_{n=0}^{m} Q(n) = \prod_{n=0}^{m} (1 + b_n r^{-n}) \tag{2.8}$$

Por ejemplo, vamos a calcular la aproximación del logaritmo. Partiendo de (2.8) obtenemos

$$\ln(x) \approx -\sum_{n=0}^{m} \ln Q(n) = -\sum_{n=0}^{m} (1 + b_n 2^{-n}) \tag{2.9}$$

Consecuentemente, el valor b_n es obtenido de la normalización multiplicativa que hemos utilizado para obtener el $\ln(1 + b \ln 2^{-n})$ a partir de una tabla y de los valores que son añadidos. Por último obtenemos la recurrencia

$$y(n+1) = y(n) - \ln(Q(n)) \tag{2.10}$$

Y el resultado obtenido es

$$y(m+1) = y(0) + \ln(x) \tag{2.11}$$

2.2.2.2. Normalización Aditiva

Para la explicación de la normalización aditiva vamos aproximar la función exponencial (e^x). Para ello obtendremos la secuencia b_n, que cumple:

$$x - \sum_{n=1}^{m} \ln(b_n) \to 0. \tag{2.12}$$

Entonces

$$e^x \approx \prod_{n=1}^{m} (1 + s_n \cdot r^{-n}). \tag{2.13}$$

Obteniendo la recurrencia

$$w(n+1) = r(w(n) - r^n \ln(1 + s_n \cdot r^{-n})), \tag{2.14}$$

el valor de la exponencial será obtenido a partir de

$$y(n+1) = y(n) \cdot (1 + s_n \cdot r^{-n}), \quad y(0) = 1. \tag{2.15}$$

- **Ejemplo de Implementación**

Para la aproximación de la atan*(y/x)* mediante métodos de convergencia lineal, vamos a dividir la operación en dos pasos. En la figura 2.2 podemos ver un esquema de la implementación realizada.

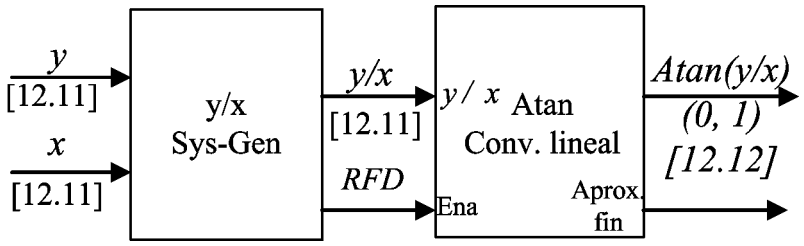


Figura 2.2 – Diagrama de bloques de la aproximación de la atan(*y*/*x*) mediante métodos de convergencia lineal.

Primero realizaremos la aproximación de la división de las dos entradas mediante un divisor generado mediante System Generator. Este core-IP [26] implementará la división entera basada en el algoritmo *non-restoring* en base 2. Este algoritmo calcula un bit del cociente por cada ciclo de reloj por medio de sumas y restas. Los dígitos utilizados para el cociente son (-1, 1) en vez de (0, 1). La división *non-restoring* tiene la ventaja de que los restos parciales negativos no tienen que ser restaurados por medio de la suma del divisor antes de desplazar el resto parcial y restando del divisor para la próxima iteración. La figura 2.3 podemos ver un pseudocódigo que presenta el algoritmo básico de la división Q=N/D *non-restoring* en base 2.

```
P[0] := N
i := 0
while i < n do
  if P[i] >= 0 then
    Q[n-(i+1)] := 1
    P[i+1] := 2*P[i] - D
  else
    Q[n-(i+1)] := -1
    P[i+1] := 2*P[i] + D
  end if
  i := i + 1
end while
```

Figura 2.3 - Algoritmo básico para la realización de la división *non-restoring* en base 2.

El divisor utilizado está completamente segmentado y lo hemos configurado para que una vez llenado el *pipeline* genere una división cada ciclo de reloj. Este core-IP puede ser configurado para generar divisiones cada 1, 2, 4 y 8 ciclos. Conforme vamos aumentando el número de ciclos de reloj por división el área utilizada en el dispositivo se va reduciendo En la tabla 2.5 podemos ver los resultados de implementación para distintos tamaños de palabra.

Precisión	12	14	16	18	20	22
Slices	601	761	827	929	1328	1444
LUT4	367	476	603	754	908	1082
F.F.	937	1201	1484	1854	2233	2628
Fmax (MHz)	183.8	177	175.3	173.6	163.7	144.9
Latencia	16	18	20	22	24	26
Throughput (Msps)	183.8	177	175.3	173.6	163.7	144.9

Tabla 2.5 - Resultados de implementación del divisor *non-restoring* para distintas precisiones de salida.

En la tabla 2.6 podemos ver comparados los resultados de implementación para el divisor con una precisión de 22 bits y modificado el número de ciclos de reloj necesarios por cada división. Como podemos apreciar al aumentar el número de ciclos por división reducimos el área utilizada (esta reducción es muy acusada en el numero de *slices*). El diseñador deberá elegir un compromiso entre los recursos utilizados y *throughput* del divisor.

Precisión	22	22	22	22
Slices	182	383	542	1444
LUT4	292	368	601	1082
F.F.	250	593	873	2628
Fmax (MHz)	178	153.8	135.1	144.9
Latencia	24	27	27	26
ciclos/Div.	22	8	4	1
Throughput (Msps)	8.1	19.2	33.8	144.9

Tabla 2.6 - Resultados de implementación del divisor modificando el número divisiones por ciclo.

Una vez realizada la división utilizaremos el resultado para el cálculo de la arcotangente. En este caso realizaremos la aproximación mediante el uso de la normalización multiplicativa de la exponencial inversa. Las ecuaciones iterativas utilizadas son las siguientes

$$\begin{aligned} U_{i+1} &= U_i - s_i 2^{-i} \cdot V_i \quad ; \quad U_0 = 1 \\ V_{i+1} &= V_i + s_i 2^{-i} \cdot U_i \quad ; \quad V_0 = C \\ y_{i+1} &= y_i + s_i \tan^{-1}(2^{-i}) \quad ; \quad y_0 = 0 \\ s_i &\in \{-1,1\}. \end{aligned} \tag{2.16}$$

Para el cálculo de la arcotangente para una precisión de 10 bits utilizaremos una tabla donde almacenaremos los valores atan(2^{-i}) para $i=0, 1, 2, 3, \ldots, 10$ con una precisión de 10 bits por cada elemento de la tabla. En la tabla 2.7 podemos ver los valores calculados para dicha precisión.

La tabla 2.8 muestra los resultados de la implementación en FPGA de la aproximación de la atan(y/x) mediante métodos de convergencia lineal. Para esta implementación hemos variado el tamaño de palabra de las entradas/salidas. Además se ha realizado una versión especial en VHDL

de la operación de división de las dos entradas. Para este caso, en cada iteración calculamos un único bit del resultado de la división. Podemos observar cómo el *throughput* es muy bajo con respecto a las frecuencias máximas de trabajo debido a la convergencia lineal hacia el resultado correcto de la aproximación de la arcotangente. Este diseño no podrá ser utilizado en los sistemas de comunicaciones en banda base ya que no se superan los 20Msps. Este método será interesante para implementaciones de funciones con una sola variable y bajos requerimientos de velocidad, ya que se necesita muy poca área.

i	2^{-i}	atan(2^{-i})
0	1.0000000000	0.1100100100
1	0.1000000000	0.0111011011
2	0.0100000000	0.0011111011
3	0.0010000000	0.0001111111
4	0.0001000000	0.0001000000
5	0.0000100000	0.0000100000
6	0.0000010000	0.0000010000
7	0.0000001000	0.0000001000
8	0.0000000100	0.0000000100
9	0.0000000010	0.0000000010
10	0.0000000001	0.0000000001

Tabla 2.7 - Valores almacenados para la aproximación de la atan(2^{-i}) para una precisión de 10 bits.

Precisión	12	14	16	18	20	22
Slices	156	219	248	285	325	360
LUT4	224	288	289	354	379	425
F.F.	218	269	289	368	405	439
Fmax (MHz)	196	192	185.8	180.6	175.5	162.8
Latencia	26	30	34	38	42	46
Throughput (Msps)	7.5	6.4	5.4	4.75	4.2	3.5

Tabla 2.8 - Resultados de implementación de la aproximación de la atan*(y/x)* mediante métodos de convergencia lineal y modificando la precisión de salida de la aproximación.

2.2.3. Métodos Basados en Recurrencia de Dígitos

Los métodos de recurrencia digito-digito y algoritmos *on-line* [27] pertenecen al mismo tipo de métodos para la aproximación de funciones elementales en hardware basados en métodos de convergencia lineal. Esta convergencia lineal implica que un número fijo de bits de la aproximación es obtenido en cada iteración. Implementaciones de este tipo de métodos son normalmente de baja complejidad, consumen muy poca área y tienen largas latencias. La elección fundamental en el diseño de estos métodos es la

elección de la base, los dígitos permitidos y la representación del resto parcial. La elección de la base de trabajo es importante ya que permite reducir la latencia del operador aumentando su tamaño. Típicamente se suelen utilizar potencias de 2 para poder realizar las multiplicaciones por la base como desplazamientos de los operandos. Este aumento en la base de trabajo tiene como inconveniente el aumento de la complejidad del operador, aumentando el área y disminuyendo la frecuencia máxima de funcionamiento.

Para este caso no vamos a realizar ningún ejemplo de implementación ya que la implementación presentada anteriormente de la división está basada en la utilización de estos métodos y los resultados no difieren de los métodos basados en convergencia lineal.

2.2.4. Métodos Basados en Tablas

2.2.4.1. Tablas Directas

El método más fácil de implementar para la aproximación una función es el basado en la utilización de una tabla Look-Up (LUT) directa. La precisión de la aproximación dependerá del número de bits utilizados para almacenar las muestras (k) y del número de bits utilizados para direccionar la tabla Look-up (n). Para una función $f(x)$ dada necesitaremos una memoria de $2^n \cdot k$ bits en las cuales almacenaremos los valores pre-calculados de la función a aproximar.

Para obtener los valores de la LUT podemos utilizar el método presentado por DasSarma y Matula [28] para el cálculo de los valores de una LUT directa para la aproximación de la función recíproco. Los resultados de este método pueden ser extrapolados a cualquier otra función elemental. En este algoritmo se busca el valor medio entre el valor actual y el siguiente valor de la entrada. Se calcula el recíproco de este valor medio y el resultado se redondea a k bits, tal como se muestra en (2.17).

$$LUT = \frac{\left\lfloor 2^{(k+1)} \cdot \frac{1}{ent_{Trunc} + 2^{-(n+1)}} \cdot 2^{-(k+2)} \right\rfloor}{2^{(k+1)}} \tag{2.17}$$

Este algoritmo minimiza el error máximo relativo en el resultado final. Además, estos autores muestran que el error máximo relativo en la aproximación del recíproco para una tabla Look-up directa de *m-bit* por *m-bit* se define como

$$\left|e_{rel}\right| = \left|LUT_0 - \frac{1}{ent}\right| \leq 1.5 \cdot 2^{-(m+1)}. \tag{2.18}$$

Las implementaciones basadas en tablas directas pueden ser aconsejables para aproximaciones de una sola variable y tamaños de palabra de hasta 20 bits. Para funciones de dos variables (x^y, $\text{atan}(y/x)$, …) podrían ser implementadas utilizando precisiones de trabajo mucho más pequeñas (hasta 10 bits) [27]. Para más de dos variables y tamaños mayores de los comentados anteriormente es inviable una implementación mediante tablas directas debido al crecimiento exponencial del tamaño de las tablas utilizadas. En la figura 2.4 podemos ver el crecimiento de la LUT necesaria para aproximar una función de una variable.

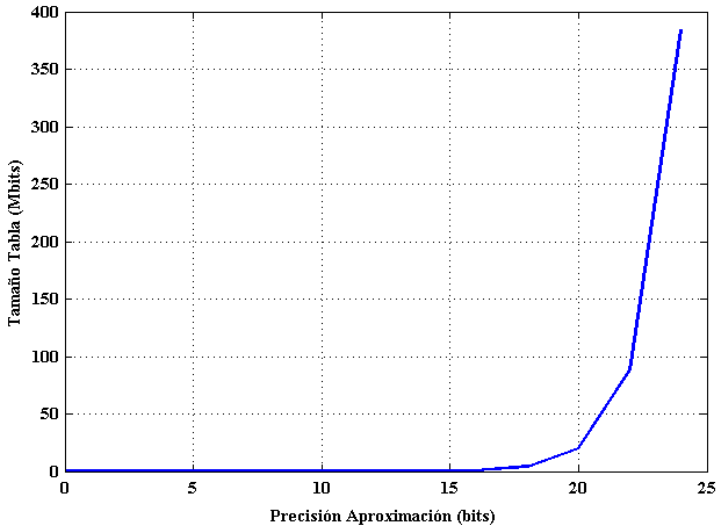


Figura 2.4 - Tamaño de las tablas necesarias para realizar la aproximación por LUTs de una función elemental.

- **Ejemplo de Implementación**

  Para aproximar la atan*(y/x)* mediante tablas directas tendremos que utilizar una tabla de tamaño $2^{(a+b)} \bullet k$ bits, siendo *a* y *b* el tamaño en bits de las dos entradas y *k* el tamaño de la precisión de salida. Los resultados de implementación obtenidos para una FPGA Virtex-2 los podemos ver en la tabla 2.9. La ventaja de la utilización de LUTs directas es que no se necesita realizar ningún cálculo, obteniendo de esta manera una alta velocidad en la aproximación. El gran problema de aproximar una función mediante tablas es la gran cantidad de memoria necesaria. Para el caso de aproximaciones de funciones de dos variables la memoria necesaria aumenta. En la tabla 2.9 podemos ver la memoria necesaria para aproximar una función de dos variables. Para el caso de 12 bits son necesarios 384 Mbits o 21845 Block-RAM. Actualmente no hay disponible ninguna FPGA con esa cantidad de Block-RAM y no se ha podido sintetizar ningún circuito basado en tablas directas.

Precisión	**12**	**14**	**16**
Memoria (Mbits)	384	3584	4096
Block-RAM	21845	203889	233016
Fmax (MHz)	-----	-----	-----
Latencia	1	1	1
Throughput (Msps)	-----	-----	-----

Tabla 2.9 - Resultados de implementación de la aproximación de la atan*(y/x)* mediante LUT directas.

Para poder aproximar funciones eficientemente mediante tablas debemos aplicar nuevos métodos que nos permitan reducir requerimientos de memoria, como sucede en el caso de las aproximaciones por polinomios, y obtener las altas velocidades de las aproximaciones mediante tablas directas. Dependiendo

de qué parámetro queramos enfatizar en nuestros diseños tendremos tres tipos de algoritmos basados en tablas [14]:

- *Métodos In-Between*. Se basan en la utilización de tablas de tamaño medio y un significativo, aunque reducido número de operaciones aritméticas complementarias. Este método engloba a aproximaciones lineales (interpoladores de primer orden), interpoladores de segundo orden, etc.
- *Métodos Table bound*, en los cuales se utilizan grandes tablas además de unas pequeñas operaciones matemáticas. Dentro de este tipo encontramos el método Partial Product Arrays (PPAs) [29] y los métodos basados en tablas Bipartidas [30]. Los métodos mediante tablas Bipartidas [31] y Bipartidas Simétricas [32] (Symmetric Bipartite Table Method - SBTM) se basan en la utilización dos tablas y una suma final para generar el resultado correcto. Con ello conseguimos reducir la memoria respecto a los métodos basados en las tablas directas. A partir de estos métodos que trabajan con dos tablas se desarrollaron métodos más genéricos, los cuales aumentan el número de tablas y el número de sumas con la consiguiente disminución en el tamaño de las tablas a utilizar. Entre estos métodos más genéricos tenemos las tablas Multipartidas [33] y Symmetric Table Addition Method (STAM) [34]. Estos métodos son muy rápidos ya que únicamente utilizan un sumador y su uso es recomendable para cálculos con una precisión no superior a 24 bits, debido al crecimiento del tamaño de las memorias necesarias y el aumento en el numero de sumadores necesarios.
- *Método Compute-bound* [25]. Utilizan tablas en las que se almacenan los valores de los coeficientes de los polinomios. En este caso predominan las operaciones aritméticas. Este método puede ser recomendable para implementaciones en las cuales tengamos disponible una unidad MAC, como pueden ser en los microprocesadores PowerPC, Intel IA64 y procesadores de señal DSP. Además será una propuesta interesante de implementación en FPGAs, al disponer internamente de multiplicadores embebidos y bloques de memoria.

Vamos a pasar a ver más en detalle los distintos métodos disponibles para la aproximación de funciones elementales mediante el uso de tablas.

2.2.4.2. Interpolación Lineal

Una alternativa más eficiente al uso de las LUTs directas es la utilización de las aproximaciones lineales o polinómicas en conjunto con las LUTs. Mediante esta técnica los coeficientes de las aproximaciones polinómicas son almacenados en LUTs y mediante operaciones de multiplicación y suma calculamos las aproximaciones.

En las aproximaciones lineales el operando de entrada x de *n-bits* es dividido en dos partes $x = x_1 + x_2$, los cuales tendrán *n/2* bits. Una aproximación a la función *f(x)* dentro del rango $x_1 \le x \le x_1 + 2^{n/2}$ puede ser obtenida por el primer término de la serie de Taylor en el punto medio $x = \left(x_1 + 2^{-(n/2+1)}\right)$:

$$f(x) \approx C_1(x_1) \cdot x_2 + C_0(x_1). \tag{2.19}$$

Los coeficientes C_1 y C_0 dependerán del valor x_1 y estarán almacenados en sendas LUT. Las tablas serán direccionadas por los *n-bits* más significativos de la entrada. Una vez obtenidos los coeficientes tendremos

que realizar una multiplicación y una suma para obtener el valor aproximado. El intervalo de entrada será dividido en $2^{n/2}$ sub-intervalos y la aproximación lineal será realizada en el centro de cada sub-intervalo. El tamaño de estas tablas será de $2^{n/2} \cdot (n + n/2)$ bits.

En el caso de los interpoladores de segundo orden la función *f(x)* será aproximada por un polinomio de segundo orden:

$$f(x) \approx C_2(x_1) \cdot x_2^2 + C_1(x_1) \cdot x_2 + C_0(x_1). \tag{2.20}$$

En este caso obtenemos una reducción en el tamaño de las tablas de $2^{2/3} \cdot (n + 2n/3 + n/3)$, a costa de aumentar el número de operaciones aritméticas necesarias.

D. DasSarma y D. W. Matula [35] presentan un algoritmo para el cálculo de los valores almacenados en las tablas utilizadas para la interpolación lineal. En este artículo además proponen un método para comprimir el tamaño de la LUT utilizada para la aproximación del recíproco. Al igual que en el caso anterior, estos resultados se pueden extrapolar para el uso de otras funciones elementales. Este método de interpolación utiliza LUTs comprimidas y multiplicadores de pequeño tamaño para generar la aproximación del recíproco en una única operación. Por ejemplo, para una aproximación al recíproco de *2k* bits utilizamos una tabla de 2^k posiciones con un tamaño de *(2k+2)* bits y un multiplicador con un tamaño de las entradas de *(k+3)* bits.

- **Ejemplo de Implementación**

  Para la aproximación de la atan*(y/x)* utilizaremos el divisor utilizado en los anteriores ejemplos para la realización de la división de las dos entradas y realizaremos la aproximación de la arcotangente mediante un interpolador lineal. En la figura 2.5 podemos ver el esquema de interpolador implementado. Para almacenar los valores utilizamos una memoria de doble puerto embebida para almacenar los coeficientes de interpolación.

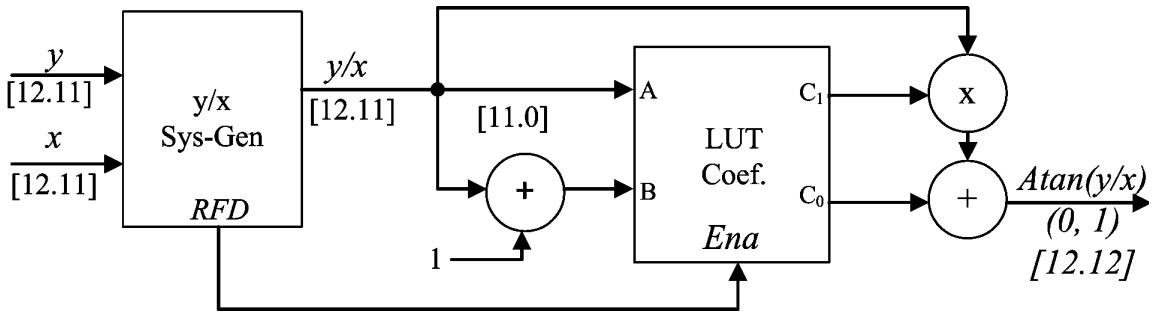


Figura 2.5 – Diagrama de bloques de la aproximación de la atan(*y/x*) mediante el uso de interpolación lineal.

Los resultados de implementación de la aproximación mediante interpoladores lineales se pueden ver en la tabla 2.10. Podemos apreciar de los resultados obtenidos cómo nuestro operador tiene una alta latencia debida sobre todo al utilizar el divisor *non-restoring*. Debido a que únicamente utilizamos tablas y multiplicadores embebidos las pérdidas de velocidad conforme aumentamos la precisión del operador no son muy grandes. Los multiplicadores utilizados están completamente segmentados. Gracias a la utilización de las memorias de doble puerto podemos acceder a los dos coeficientes a la vez. Vemos que para altas precisiones tenemos que aumentar el número de

multiplicadores utilizados para poder realizar la operaciones ya que el tamaño de las entradas es de únicamente 18 bits. Por último comentar, que esta implementación puede trabajar en sistemas de comunicaciones en banda-base, ya que superamos ampliamente la tasa mínima exigida de 20 Msps para cualquier tamaño de muestra.

Precisión	**12**	**14**	**16**	**18**	**20**	**22**
Slices	648	819	879	1046	1420	1580
LUT4	408	537	653	812	998	1220
F.F.	978	1270	1540	1920	2395	2750
Mult18x18	1	1	1	1	3	3
Block RAM	1	1	1	2	2	4
Fmax (MHz)	145.5	138.2	136	135.1	118	115.7
Latencia	20	22	24	26	30	33
Throughput (Msps)	145.5	138.2	136	135.1	118	115.7

Tabla 2.10 - Resultados de implementación de la aproximación de la atan*(y/x)* mediante el uso de interpolación lineal.

2.2.4.3. Tablas Bipartidas

Este método fue presentado por D. Das Sarma y D. W. Matula [30] para el caso especifico de la aproximación para el cálculo del recíproco y generalizado por J. M. Muller [31] y Michael J. Schulte y James E. Stine [32] y. En este método la tabla utilizada para almacenar el valor de la aproximación se divide en dos tablas más pequeñas y las salidas de ambas tablas se suman para obtener la aproximación:

$$f(x) = f(x_0 + x_1 + x_2) \approx a_0(x_0, x_1) + a_1(x_0, x_2). \tag{2.21}$$

Para la aproximación de la función $f(x)$ mediante tablas bipartidas la entrada se divide en tres partes de k-bits, x_0, x_1 y x_2, donde $x = x_0 + x_1 \cdot 2^{-k} + x_2 \cdot 2^{-2k}$, asumiendo $0 \leq x \leq 1$. En la figura 2.6 se puede ver el particionado de la entrada x.

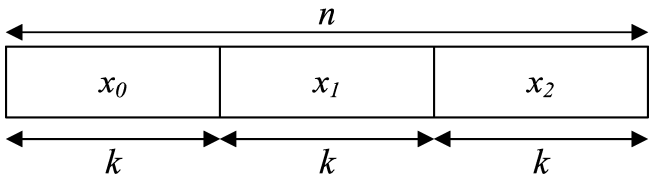


Figura 2.6 – División de la palabra de entrada en tres partes para el direccionamiento de las dos LUTs necesarias.

Para calcular los valores de los coeficientes que almacenaremos en las tablas LUT partiremos de la expansión de Taylor de primer orden (2.22) de la función elemental a aproximar y de los cuales nos quedaremos con los dos primeros términos de la aproximación para obtener (2.23).

$$f(x) = f(x_0 + x_1 \cdot 2^{-k}) + x_2 \cdot 2^{-2k} f'(x_0 + x_1 \cdot 2^{-k}) + x_2^2 \cdot 2^{-4k} f''(\xi), \quad \xi \in [x_0 + x_1 \cdot 2^{-k}, x] \tag{2.22}$$

$$f(x) = f\left(x_0 + x_1 \cdot 2^{-k}\right) + x_2 \cdot 2^{-2k} \cdot f'(x_0) \tag{2.23}$$

Por último, partiendo de (2.23), podremos generar la suma bipartida (2.21) de la aproximación de *f(x)* mediante la suma de las dos funciones a_0 y a_1:

$$\begin{aligned} a_0(x_0, x_1) &= f\left(x_0 + x_1 \cdot 2^{-k}\right) \\ a_1(x_0, x_2) &= x_2 \cdot 2^{-2k} \cdot f'(x_0) \end{aligned} \tag{2.24}$$

Los coeficientes a_0 y a_1 serán almacenados en dos LUTs y serán direccionados por (x_0, x_1) y (x_0, x_2), respectivamente. Mediante un sumador de n-bits sumaremos los dos valores de las tablas para generar la aproximación de la función. En la figura 2.7 podemos ver los valores almacenados en cada una de las dos tablas *(a_0 y a_1)* para la aproximación del recíproco. En la tabla a_0 podemos identificar fácilmente los valores de la función recíproco y en la tabla a_1 podemos ver los valores utilizados para el ajuste fino de la aproximación. El error cometido en la aproximación será

$$\varepsilon = (2^{-(4k+1)} + 2^{-3k}) \cdot \max_{[\alpha,1]} |f''| + 2^{-3k}. \tag{2.25}$$

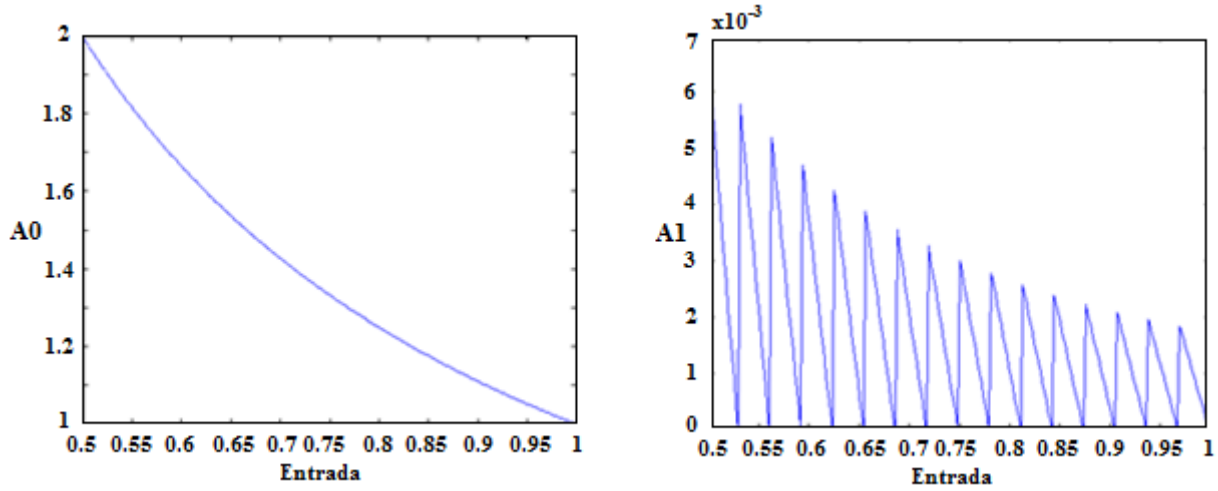


Figura 2.7 - Valores de la tablas a_0 y a_1 utilizadas en el cálculo de la aproximación del recíproco.

Si el valor de la segunda derivada de *f(x)* es cercano a uno, como puede ocurrir en el caso de la función recíproco, el error cometido será aproximadamente igual al número de bits utilizados para representar dicho número $\varepsilon \approx 2^{-n}$. Definiendo el tamaño de las tres partes de la entrada igual la memoria necesaria será de:

$$2 \cdot \left(\left({}^{4n}/_{3} \right) \cdot 2^{{}^{2n}/_{3}} \right) bits \tag{2.26}$$

Para una precisión de *n=16 bits*, necesitaremos una tabla directa con un tamaño de aproximadamente 512 kbits. En cambio, utilizando el método bipartido únicamente serán necesarias dos tablas con un tamaño total de aproximadamente 33 kbits.

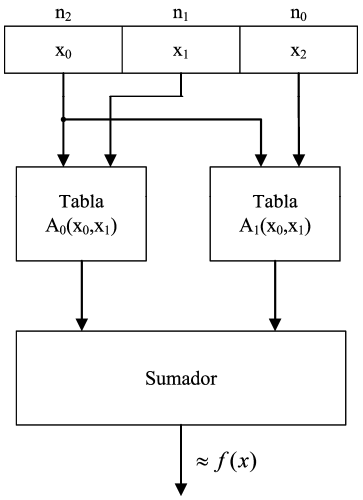


Figura 2.8 - Estructura hardware para la implementación de la aproximación mediante LUTs bipartidas.

Como podemos ver en la figura 2.8, los requerimientos hardware utilizados para la implementación de este método son bastante bajos ya que únicamente necesitamos dos tablas a_0 y a_1 de 2^{n0+n1} posiciones y 2^{n0+n2} posiciones, respectivamente, y un sumador final para generar la aproximación. Los tamaños de tablas obtenidos pueden ser de hasta 10 veces mayores que los métodos basados en interpolación lineal, pudiendo ser interesante la utilización de estos métodos para precisiones de hasta 20 bits.

- **Ejemplo de Implementación**

  Para la aproximación de la atan*(y/x)* utilizaremos el mismo divisor que en los casos anteriores para que las diferencias en cuanto su implementación sean debidas exclusivamente a la utilización de un método para aproximar la arcotangente. En la figura 2.9 podemos ver el esquema de aproximación implementado mediante tablas bipartidas. Para almacenar los valores podemos utilizar tanto memorias Block-RAM como memorias distribuidas, esto dependerá del tamaño de las tablas utilizadas. Para la implementación de las tablas pequeñas será más eficiente la utilización de memoria distribuida y las tablas grandes deberán ser implementadas mediante memoria embebida Block-RAM Para una precisión de entrada de 12 bits el particionado óptimo obtenido es igual a $n_0 = 4$, $n_1 = 4$ y $n_2 = 4$. La señal RFD será utilizada para habilitar las memorias y realizar la aproximación con los datos válidos del divisor.

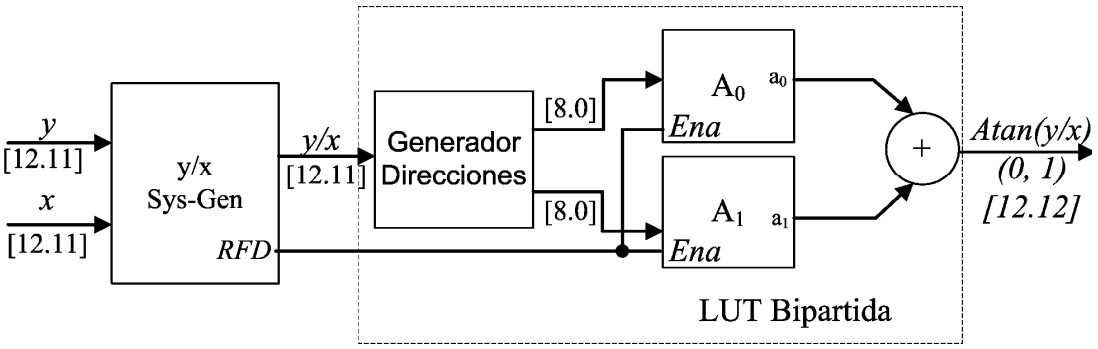


Figura 2.9 - Implementación de la aproximación de la atan(*y/x*) mediante LUT bipartidas.

Los resultados de implementación de la aproximación mediante tablas bipartidas se pueden ver en la tabla 2.11. La gran ventaja de esta implementación es que no necesitamos realizar ninguna multiplicación con el consiguiente aumento de velocidad con respecto a los interpoladores lineales. Por el contrario podemos ver como necesitamos un mayor número de Block-RAM para almacenar los valores de las tablas, sobre todo para grandes precisiones.

Precisión	**12**	**14**	**16**	**18**	**20**	**22**
Slices	618	782	845	945	1359	1520
LUT4	375	490	648	768	975	1620
F.F.	950	1210	1501	1795	2357	2735
Mult18x18	0					
Block RAM	1	1	1	3	6	12
Fmax (MHz)	175.5	172.7	172	168.2	165	164
Latencia	18	20	22	24	26	28
Throughput (Msps)	175.5	172.7	172	168.2	165	164

Tabla 2.11 - Resultados de implementación de la aproximación de la atan*(y/x)* mediante LUTs bipartidas.

2.2.4.4. Tablas Bipartidas Simétricas - (SBTM).

Este método de implementación de Tablas Bipartidas Simétricas (*Symmetric Bipartite Table Method* – SBTM) está basado en el método original desarrollado *Debjit Das Sarma* y *David W. Matula* en [30] sobre el cual *Michael J. Schulte* y *James E. Stine* en [32] sugirieron una serie de modificaciones para aumentar el rendimiento de dichas tablas. Para la aproximación de la función $f(x)$ mediante tablas SBTM, la entrada es dividida en tres partes x_0, x_1 y x_2, de n_0, n_1 y n_2 bits donde $x = x_0 + x_1 \cdot 2^{-n_0} + x_2 \cdot 2^{-(n_0+n_1)}$, asumiendo $0 \le x \le 1$.

Básicamente, sugirieron realizar la aproximación del primer término de Taylor sobre el punto $x_0 + x_1 \cdot 2^{-n_0} + 2^{-(n_0+n_1+1)}$ en vez $x_0 + x_1 \cdot 2^{-n_0}$, para el segundo término de Taylor utilizaron $x_0 + 2^{-(n_0+1)}$ en vez de x_0, reduciendo de esta manera el error cometido en la aproximación. Por tanto, la expansión de Taylor (2.22) para la aproximación de la función, quedará de la siguiente forma

$$f(x) = f\left(x_0 + x_1 \cdot 2^{-n_0} + 2^{-(n_0+n_1+1)}\right) + x_2 \cdot 2^{-(n_0+n_1)} f'\left(x_0 + 2^{-(n_0+1)}\right) + \epsilon, \tag{2.27}$$

y los valores que se almacenan en las tablas serán:

$$\begin{aligned} a_0(x_0, x_1) &= f\left(x_0 + x_1 \cdot 2^{-n_0} + 2^{-(n_0+n_1+1)}\right) \\ a_1(x_0, x_2) &= x_2 \cdot 2^{-(n_0+n_1)} \cdot f'\left(x_0 + 2^{-(n_0+1)}\right). \end{aligned} \tag{2.28}$$

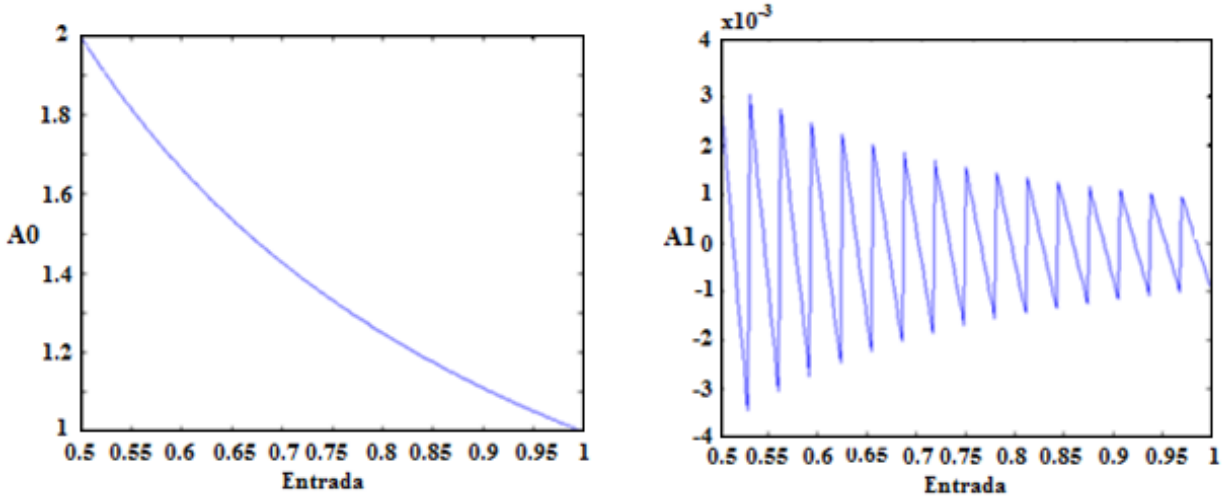


Figura 2.10 - Valores de la tablas a_0 y a_1 utilizadas en el cálculo del recíproco mediante el método SBTM.

Como podemos apreciar de la figura 2.10, los valores de la tabla $a_1(x_0, x_2)$ son simétricos pudiendo reducir el número de valores almacenados a la mitad ya que únicamente almacenaríamos los valores positivos, realizando el complemento para los valores negativos. Además el error absoluto máximo cometido en la aproximación es menor que el obtenido con el método original [32]. El error cometido en la aproximación SBTM estará definido a partir de (2.25). Analizando los valores de la tabla a_1, los bits más significativos (MSB) son todos "unos" o "ceros", los cuales no tienen que ser almacenados en las tablas. El número de "unos" o "ceros" repetidos se puede aproximar por

$$L = n_0 + n_1 + 1 + \log_2\left(\max|f'| \Big/ \max|f''|\right). \tag{2.29}$$

Para calcular el tamaño de las tablas mediante el método SBTM definiremos, al igual que en caso anterior, un tamaño de segmento igual para las tres partes de la entrada. El tamaño total de la tabla será de

$$\left(\frac{2n}{3}\right)\cdot 2^{2n/3} + (n-L)\cdot 2^{(2n/3)-1}\, bits. \tag{2.30}$$

Para una precisión de n=16 bits necesitaremos una tabla directa con un tamaño de aproximadamente 512 kbits. En cambio, utilizando el método SBTM únicamente serán necesarias dos tablas con un tamaño total de aproximadamente 25 kbits.

En la figura 2.11 podemos ver la estructura hardware de este tipo de implementaciones. Son prácticamente parecidos a la implementación Bipartida con la salvedad de la inclusión de las dos XOR para poder aprovecharnos de las simetrías aparecidas en la tabla $a_1(x_0, x_2)$. Para ello evaluamos el bit más significativo de x_2. Si es cero, el resto de las direcciones son usadas para direccionar la tabla $a_1(x_0, x_2)$ y el valor de esta tabla es sumado al valor de $a_0(x_0, x_1)$. Si por el contrario este valor es '1', el resto de los bits de x_2 son complementados y usados para direccionar la tabla $a_1(x_0, x_2)$ y el valor leído será nuevamente complementado y sumado al valor de $a_0(x_0, x_1)$.

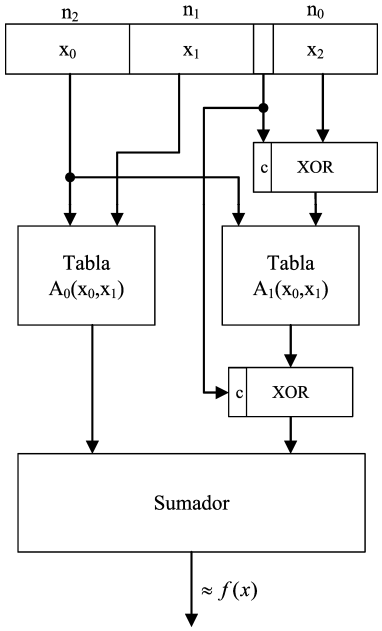


Figura 2.11 - Estructura hardware para la implementación de la aproximación mediante el método SBTM.

- **Ejemplo de Implementación**

Para la aproximación de la atan*(y/x)* utilizaremos el divisor usado en los anteriores ejemplos para la realización de la división de las dos entradas y realizaremos la aproximación de la arcotangente mediante la utilización de una memoria bipartida. En la figura 2.12 podemos ver el esquema de aproximación implementado mediante tablas bipartidas simétricas. Para almacenar los valores utilizaremos tanto memorias Block-RAM como memorias distribuidas. Esto será debido a la diferencia de tamaños que obtendremos al calcular las tablas necesarias para obtener las precisiones pedidas de las aproximaciones. La gran ventaja del método SBTM con respecto al método mediante tablas bipartidas es que únicamente necesitamos la mitad de los valores de la tabla A_1 (valores positivos) ya que los valores negativos los podemos generar por media de la XOR (complemento a 1). Para una precisión de entrada de 12 bits, el particionado óptimo obtenido es igual a $n_0 = 4$, $n_1 = 4$ y $n_2 = 4$.

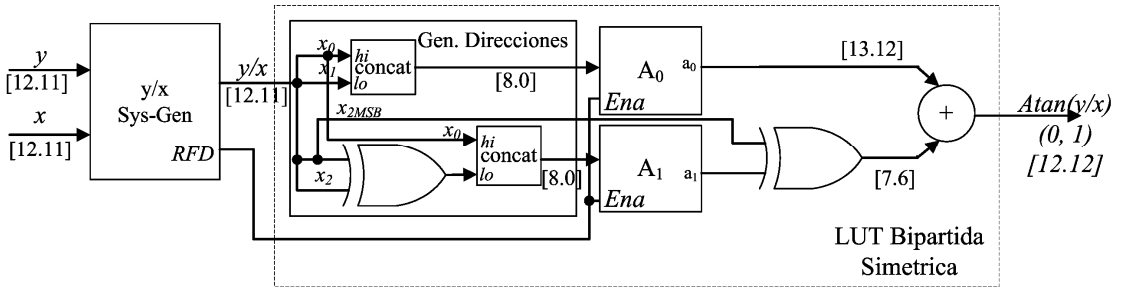


Figura 2.12 - Implementación de la aproximación de la atan(*y/x*) mediante el método SBTM.

Los resultados de implementación de la aproximación mediante tablas SBTM se pueden ver en la tabla 2.12. Como se aprecia a partir de los resultados obtenidos este método nos permite reducir la cantidad de memoria necesaria con respecto al método de las memorias bipartidas añadiendo una pequeña cantidad de lógica que nos permita generar el resto de valores de A_1. Esta reducción se puede apreciar cuando trabajamos con precisiones relativamente altas. Para bajas precisiones no

apreciamos reducción en el número de Block-RAMs utilizadas debido al tamaño de las mismas. Además, podemos ver como la pérdida de velocidad del diseño no se ha reducido por incluir esta pequeña lógica adicional

Precisión	**12**	**14**	**16**	**18**	**20**	**22**
Slices	610	780	810	905	1316	1486
LUT4	365	480	588	750	906	1500
F.F.	950	1210	1501	1795	2357	2735
Mult18x18				0		
Block RAM	1	1	1	3	5	10
Fmax (MHz)	175.5	172.2	172	168	165	162
Latencia	18	20	22	24	26	28
Throughput (Msps)	175.5	172.2	172	168	165	162

Tabla 2.12 - Resultados de implementación de la aproximación de la atan*(y/x)* mediante el método SBTM.

2.2.4.5. Tablas Multipartidas - (STAM)

Este método llamado Symmetric Table Addition Method (STAM) es una ampliación del método STBM propuesto por Michael J. Schulte y James E. Stine en [32] en el cual aumentan el número de tablas hasta un máximo de seis. J.M. Muller en [31] también presenta un método basado en tablas multipartidas con ligeros matices con respecto al STAM. Por último, F. Denechin y A. Tisserand en [33] presentan una unificación y generalización de las dos propuestas de tablas multipartidas comentadas anteriormente.

En este caso dividiremos la entrada en $m+1$ partes de n-bits, $x_0, x_1, \ldots, x_n$ con $n_0, n_1, \ldots, n_m$ bits respectivamente, donde $x = x_0 + x_1 \cdot 2^{-(n_0)} + \ldots + x_m \cdot 2^{-(n_0+n_1+\ldots+n_{m-1})}$, asumiendo $0 \leq x \leq 1$. Dicha división se muestra en la figura 2.13.

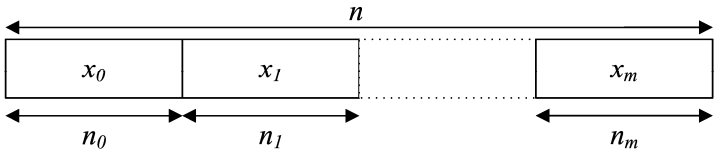


Figura 2.13 - División de la palabra de entrada en $m+1$ partes para el direccionamiento de las m LUTs necesarias.

A partir de esta división en $m+1$ partes de la entrada obtendremos una aproximación a la función de la siguiente forma:

$$f(x) \approx \sum_{i=1}^{m} a_{i-1}(x_0, x_i). \tag{2.31}$$

Como podemos ver, en la implementación del método STAM básicamente lo que hacernos es utilizar más tablas para realizar la aproximación permitiendo una reducción mayor del tamaño de las mismas. Para calcular los valores de las tablas utilizaremos, al igual que en el método STBM, la expansión de Taylor de primer orden,

$$f(x) = f\left(x_0 + x_1 \cdot 2^{-n_0} + \ldots + x_{\left(m/2-1\right)} \cdot 2^{-\left(n_0+n_1+\ldots+n_{(m-1)/2}\right)} + 2^{-\left(n_0+n_1+\ldots+n_{(m-1)/2}+1\right)} \right)$$
$$+\left(x_{m/2} \cdot 2^{-\left(n_0+n_1+\ldots+n_{m/2}\right)} + \ldots + x_m \cdot 2^{-(n_0+n_1+\ldots+n_m)} \right) \cdot \qquad (2.32)$$
$$f'\left(x_0 + x_1 \cdot 2^{-n_0} + \ldots + x_{m/2-1} \cdot 2^{-(n_0+n_1+\ldots+n_{m/2})} \right) + \epsilon.$$

En este caso, el error cometido en la aproximación será aproximadamente

$$\varepsilon \le \frac{1}{2} \cdot 2^{-(m+1)} \max\left|f''\right|, \qquad (2.33)$$

y los valores que almacenaremos en las tablas serán:

$$\begin{aligned}
a_0(x_0, x_1) &= f\left(x_0 + x_1 \cdot 2^{-n_0} + \ldots + x_{\left(m/2-1\right)} \cdot 2^{-\left(n_0+n_1+\ldots+n_{(m-1)/2}\right)} + 2^{-\left(n_0+n_1+\ldots+n_{(m-1)/2}+1\right)} \right) \\
a_1(x_0, x_1) &= x_{m/2} \cdot 2^{-\left(n_0+n_1+\ldots+n_{(m-1)/2}\right)} \cdot f'\left(x_0 + x_1 \cdot 2^{-n_0} + \ldots + x_{m/2-1} \cdot 2^{-(n_0+n_1+\ldots+n_{(m-1)/2})} \right) \\
a_2(x_0, x_2) &= x_{m/2+1} \cdot 2^{-\left(n_0+n_1+\ldots+n_{m/2+1}\right)} \cdot f'\left(x_0 + x_1 \cdot 2^{-n_0} + \ldots + x_{m/2} \cdot 2^{-(n_0+n_1+\ldots+n_{m/2+1})} \right) \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
a_m(x_0, x_m) &= x_m \cdot 2^{-(n_0+n_1+\ldots+n_m)} \cdot f'\left(x_0 + x_1 \cdot 2^{-n_0} + \ldots + x_m \cdot 2^{-(n_0+n_1+\ldots+n_m)} \right).
\end{aligned} \qquad (2.34)$$

Esta implementación utilizará *m* tablas en paralelo y un sumador de *m* entradas, además, al igual que en el caso de la implementación STBM, podemos aprovechar las simetrías de los coeficientes almacenados en las tablas $a_1(x_0, x_2)$, $a_2(x_0, x_3)$, … , $a_m(x_0, x_m)$ para almacenar únicamente los coeficientes positivos. Utilizaremos las puertas XOR para generar los valores complementarios de dichas tablas.

En la figura 2.14 podemos ver la estructura hardware de este tipo de implementaciones. Son prácticamente similares a la implementación SBTM, exceptuando el número de tablas a utilizar, que en este caso puede llegar a seis. En el método STAM la mayor complejidad estriba en el diseño del sumador multi-operando, ya que valores grandes de *m* implica encadenar varios sumadores con el consiguiente aumento en la complejidad y el retardo combinacional en su diseño.

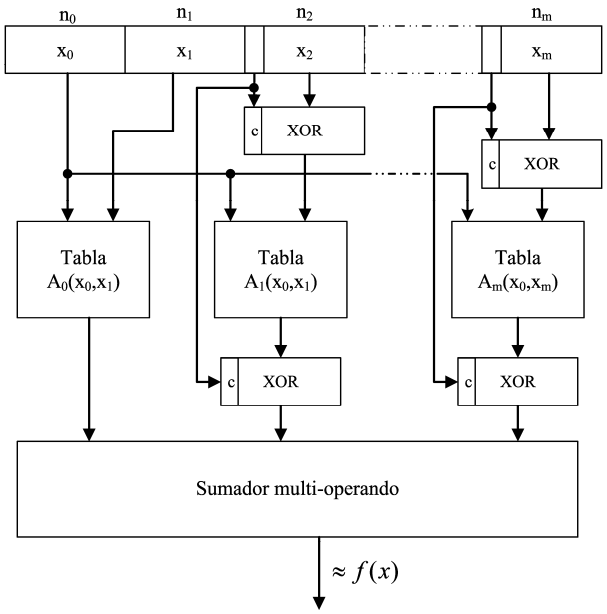


Figura 2.14 - Estructura hardware para la implementación de la aproximación mediante el método STAM.

- **Ejemplo de Implementación**

Para la aproximación de la atan*(y/x)* utilizaremos el divisor utilizado en los anteriores ejemplos para la realización de la división de las dos entradas y realizaremos la aproximación de la arcotangente mediante la utilización de una memoria bipartida. En la figura 2.15 podemos ver las tablas utilizadas para la aproximación mediante tablas multipartidas. Para almacenar los valores utilizaremos tanto memorias Block-RAM como memorias distribuidas. Esto será debido a la diferencia de tamaños que obtendremos a la hora de calcular las tablas necesarias para obtener las precisiones pedidas de las aproximaciones. En las operaciones aritméticas siempre extendemos el signo de las tablas pequeñas a los tamaños de las tablas más grandes. Para una precisión de entrada de 12 bits, el particionado óptimo obtenido es igual a $n_0 = 3$, $n_1 = 3$, $n_2 = 3$ y $n_3 = 3$. Como podemos apreciar con las implementaciones multipartidas obtenemos una reducción considerable del tamaño de las tablas utilizadas a costa de aumentar el número de sumadores necesarios. En la figura 2.16 vemos la implementación de tablas multipartidas implementadas.

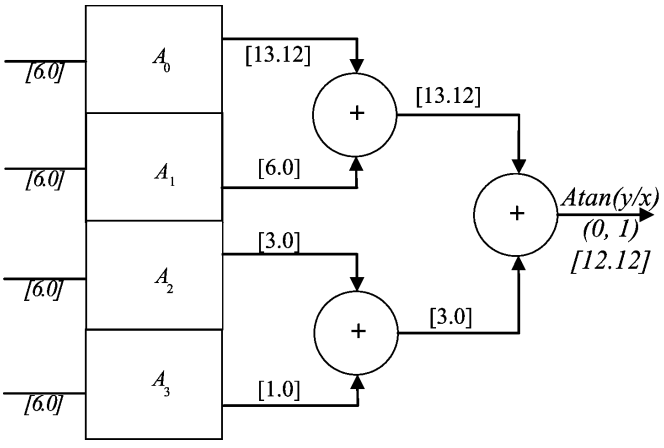


Figura 2.15 – Tamaño de las tablas utilizadas para la aproximación de la atan(*y/x*).

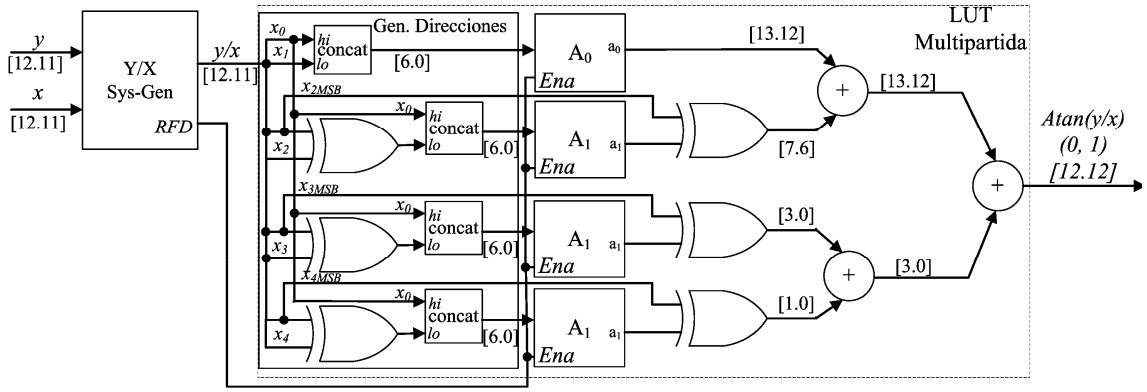


Figura 2.16 - Implementación de la aproximación de la atan(*y*/*x*) mediante el método STAM.

Los resultados de implementación de la aproximación mediante tablas multipartidas se pueden ver en la tabla 2.13. Para el caso de la implementación multipartida vemos como podemos llegar a reducir a la mitad el número de Block-RAM necesarias con respecto a la implementación SBTM. Por otro lado, vemos que el *throughput* obtenido permite su utilización en sistemas SDR ya que superamos la tasa mínima de 20 Msps. Sobre la frecuencia máxima de funcionamiento no tenemos una reducción en la frecuencia máxima por aumentar el número de sumadores en la generación de la aproximación final ya que el camino crítico lo obtenemos en la implementación del divisor.

Precisión	**12**	**14**	**16**	**18**	**20**	**22**
Slices	610	775	852	1012	1335	1472
LUT4	390	495	634	914	934	1134
F.F.	945	1235	1534	1882	2267	2668
Mult18x18	0					
Block RAM	1	1	1	2	5	5
Fmax (MHz)	175.3	172	171.7	166	164.6	162.7
Latencia	20	22	24	26	28	30
Throughput (Msps)	175.3	172	171.7	166	164.6	162.7

Tabla 2.13 - Resultados de implementación de la aproximación atan*(y/x)* mediante el método STAM.

2.2.5. Aproximaciones por Polinomios

Las aproximaciones por polinomios implican la aproximación de una función continua f con uno o más polinomios p de grado n dentro de un intervalo [a, b]. Dentro de estos métodos encontramos dos clases de aproximaciones: "*least squares approximations*" que minimizan el error medio y "*least maximum approximations*" que minimizan el error en el peor caso [14]. En ambos casos, el objetivo primordial es reducir la "distancia" $\|p-f\|$. Para las aproximaciones *least squares* esta distancia es

$$\|p-f\| = \sqrt{\int_a^b w(x)(f(x)-p(x))^2\,dx} \tag{2.35}$$

donde *w* será una función de ponderación continua que será utilizada para seleccionar las partes del rango de la aproximación donde queremos que la función sea más precisa. Para las aproximaciones *least máximum*, también llamadas aproximaciones *minimax*, esta distancia será

$$\|p - f\| = \max_{a \le x \le b} w(x) \cdot |f(x) - p(x)|. \tag{2.36}$$

En este punto vamos a utilizar las aproximaciones *minimax* ya que al trabajar en coma fija será más interesante minimizar el error máximo. Para el caso de trabajar en coma flotante, deberíamos minimizar el error medio [14].

Los polinomios *p* utilizados serán de la forma

$$p(x) = c_d x^d + c_{d-1} x^{d-1} + c_{d-2} x^{d-2} + ... + c_1 x + c_0 \tag{2.37}$$

y mediante la aplicación de la regla de Horner podremos reducir el número de operaciones necesarias, obteniendo

$$p(x) = (((c_d x + c_{d-1})x + c_{d-2})x + ...)x + c_0, \tag{2.38}$$

siendo *x* la entrada.

Para obtener una alta precisión y poder aproximar una función en un amplio dominio es necesario utilizar un polinomio de alto grado, con el consiguiente aumento de complejidad en la aproximación. Para reducir esa complejidad podemos segmentar el dominio de la función en varios sub-dominios de igual tamaño (segmentación uniforme) y utilizar diferentes polinomios para cada segmento. Los coeficientes del polinomio serán almacenados en una o varias LUT's y dependiendo del segmento de trabajo, seleccionaremos unos u otros coeficientes.

Esta segmentación uniforme puede ser problemática e ineficiente en regiones de la función con grandes no-linealidades (funciones donde la primera derivada ó derivadas de un alto orden tengan valores absolutos grandes). Estas regiones necesitarán más segmentos que en las regiones con una alta linealidad. Esta segmentación mediante técnicas de segmentación no-uniforme permite optimizar las cantidades de memoria necesarias para almacenar los coeficientes, ya que adaptamos el tamaño del segmento al comportamiento de la función. Muchos autores han propuesto deferentes métodos para la segmentación no uniforme de funciones. Por ejemplo, Combet [36] propone el uso de segmentos que se incrementan en potencias de dos para la aproximación de los logaritmos binarios. D. Lee y otros [37] proponen el uso de jerarquías que contienen segmentos uniformes y segmentos que varían el tamaño en potencias de dos.

El polinomio *minimax* lo calcularemos mediante la utilización del algoritmo iterativo de Remez, el cual es utilizado normalmente para determinar de una manera óptima los coeficientes de los filtros digitales. La gran desventaja de las aproximaciones por polinomios es la gran cantidad de multiplicaciones secuenciales necesarias o, en el caso de querer paralelizar operaciones, la cantidad de multiplicadores necesarios.

- **Ejemplo de Implementación**

Para la aproximación de la atan*(y/x)* se ha implementado un algoritmo que nos calcule la aproximación por polinomios utilizando el algoritmo de Remez cumpliendo las especificaciones definidas en la introducción. El polinomio obtenido es el siguiente:

$$p(z) = 0.03861z^5 - 0.18171z^3 + 0.63328z, \text{ siendo z} = \left(y/x\right). \tag{2.39}$$

Para la realización del diseño se ha implementado una arquitectura completamente en paralelo que nos permita reducir el consumo. El polinomio implementado, expresado utilizando la regla de Horner, es el siguiente:

$$p(z) = z\left(0.63328 + z^2\left(0.18171 + 0.3861z^2\right)\right). \tag{2.40}$$

Esta aproximación ha sido generada mediante la herramienta Xilinx System Generator para diferentes precisiones de salida y añadiendo dos etapas de segmentación. La tabla 2.14 muestra los resultados obtenidos después del Place & Route y utilizando una etapa de registros. Podemos ver cómo gracias a la utilización de la regla de Horner únicamente son necesarios tres multiplicadores. Al igual que en los casos anteriores la velocidad alcanzada es superior a las especificaciones marcadas con valores de latencia relativamente altas. Para precisiones de más de 18 bits el número de multiplicadores utilizados aumenta debido al tamaño de 18 bits de los multiplicadores embebidos disponibles en la FPGAs de Xilinx.

Precisión	**12**	**14**	**16**	**18**	**20**	**22**
Slices	650	805	862	975	1365	1492
LUT4	420	560	685	794	958	1146
F.F.	950	1256	1535	1924	2312	2805
Mult18x18	3	3	3	3	9	9
Block RAM	0					
Fmax (MHz)	152.1	148.9	145.4	141.8	128.6	123.8
Latencia	26	28	30	32	42	44
Throughput (Msps)	152.1	148.9	145.4	141.8	128.6	123.8

Tabla 2.14 - Resultados de implementación de la aproximación de la atan*(y/x)* mediante polinomios y modificando la precisión de salida de la aproximación.

2.2.6. Aproximaciones por División de Polinomios

Cualquier función elemental continua puede ser aproximada por un polinomio de grado *d* ó por una aproximación racional o división de polinomios.

$$p(x) = \frac{c_n x^n + c_{n-1} x^{n-1} + c_{n-2} x^{n-2} + ... + c_1 x + c_0}{d_m x^m + d_{m-1} x^{m-1} + d_{m-2} x^{m-2} + ... + d_1 x + d_0} \tag{2.41}$$

Las aproximaciones racionales tienen una mayor precisión que las aproximaciones de polinomios utilizando el mismo número de coeficientes además de permitir una mayor paralelización de las operaciones. Sin embargo, surge la necesidad de realizar una división final. Las aproximaciones racionales son preferibles a las aproximaciones por polinomios para funciones con polos, como puede ser tangente(x) ó asintóticas como la arcotangente(x).

El gran inconveniente de estas aplicaciones es la aparición de la operación de división, haciendo estas implementaciones menos interesantes en dispositivos que no tengan operadores de división como puede ser el caso de FPGAs. Para este caso no hemos realizado implementaciones ya que los resultados obtenidos son parecidos a la aproximación por polinomios realizada anteriormente, añadiendo el retardo de realizar una división adicional.

2.2.7. Aproximaciones por Series de Taylor

El teorema de Taylor fue formulado por Brook Taylor en 1715 y nos permite representar una función como una serie infinita de términos calculados a partir de los valores de las derivadas en un único punto. Para ello la función tiene que ser infinitamente derivable y puede ser tanto real como compleja. Si la serie la centramos en cero, esta será llamada serie Maclaurin.

La serie de Taylor de una función *f* centrada en un punto *t* se representa de la siguiente manera

$$f(x) = f(t) + f'(t) \cdot (x-t) + f''(t) \cdot \frac{(x-t)^2}{2!} + ... + f^{(n)}(t) \cdot \frac{(x-t)^n}{n!} \tag{2.41}$$

que en forma compacta queda

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(t)}{n!} (x-a)^n \quad . \tag{2.42}$$

La serie de Taylor se puede generalizar a funciones de más de una variable a partir de

$$\sum_{n_0}^{\infty} \cdots \sum_{n_d}^{\infty} \frac{\partial^{n_0}}{\partial x^{n_0}} \cdots \frac{\partial^{n_d}}{\partial x^{nd}} \frac{f(t_0, \cdots, t_d)}{n_0 ! \cdots n_d} (x_0 - t_0)^{n_0} \cdots (x_d - t_d)^{n_d} \quad . \tag{2.43}$$

A continuación podemos ver algunas series de Taylor para varias funciones elementales:

$$\begin{aligned}
\sqrt{x} &= 1+\frac{x}{2}-\frac{x^2}{8}+\frac{x^3}{16}-\frac{15x^4}{128}+\cdots\cdots \\
\ln(1+x) &= x-\frac{x^2}{2}+\frac{x^3}{3}-\frac{x^4}{4}+\cdots\cdots \\
\cos(x) &= 1-\frac{x^2}{2}+\frac{x^4}{24}-\cdots\cdots \\
\operatorname{atan}(x) &= x-\frac{x^3}{3}+\frac{x^5}{5}-\cdots\cdots
\end{aligned} \tag{2.44}$$

- **Ejemplo de Implementación**

Para la aproximación de la atan*(y/x)* una primera aproximación sería la utilización de la aproximación de Taylor de primer orden (2.41) para una función de dos variables:

$$\begin{aligned}
f(x,y) = f(a,b)+(x-a)f_x(a,b)+(y-b)f_y(a,b)+ \\
+\frac{1}{2!}((x-a)^2 f_{xx}(a,b)+2(x-a)(y-b)f_{xy}(a,b)+(y-b)^2 f_{yy}(a,b))+\ldots
\end{aligned} \tag{2.45}$$

donde f_x(a,b), f_y(a,b) son las derivadas parciales con respecto a *x, y* evaluadas en *x=a*, *y=b*. Aplicando el primer término de (2.43) sobre la función atan(y/x) obtendríamos la aproximación:

$$\operatorname{atan}\left(Y/X\right) = \operatorname{atan}\left(a/b\right)+(y-b)\left(\frac{2}{x\left(1+\frac{y^2}{x^2}\right)\pi}\right)-(x-a)\left(\frac{2y}{x^2\left(1+\frac{y^2}{x^2}\right)\pi}\right). \tag{2.46}$$

La aproximación (2.45) únicamente será válida si $|x|,|y|<1$. Para generar los valores negativos de la función arcotangente utilizaremos la propiedad:

$$\operatorname{atan}(-x) = -\operatorname{atan}(x). \tag{2.47}$$

Como se puede apreciar de la figura 2.17 los requerimientos hardware para esta implementación son grandes, ya que necesitamos tres tablas pre-calculadas. A partir de (2.46) podemos apreciar que necesitamos tres tablas ya que esta función será aproximada mediante planos al ser una función bidimensional, en vez de rectas como en el caso de funciones de una variable. En la primera tabla tenemos el valor de la función en el punto medio del plano (*x=a*, *y=b*), las otras dos tablas son utilizadas para aproximar el valor de la arcotangente dentro de los planos aproximados.

Básicamente seguimos el mismo criterio que en los métodos bipartidos pero trabajando con dos dimensiones. Los bits más significativos de las dos entradas son utilizados para direccionar las tablas (el numero de bits a utilizar dependerá del numero de planos que vamos a utilizar) y los bits menos significativos los utilizaremos para calcular el valor dentro del plano (multiplicamos este valor por el valor almacenado en las tablas). Necesitaremos para esta implementación un número de tablas elevado, además de dos multiplicadores y dos sumadores para obtener la aproximación completa, con el consiguiente retardo combinacional. Además deberemos añadir unas entradas que nos permitan normalizar las entradas ya que la aproximación (2.46) únicamente es válida si el valor de las dos entradas es menor que uno. La gran ventaja de esta implementación es que no

necesitamos implementar la operación de división obteniendo una implementación con una estructura muy regular.

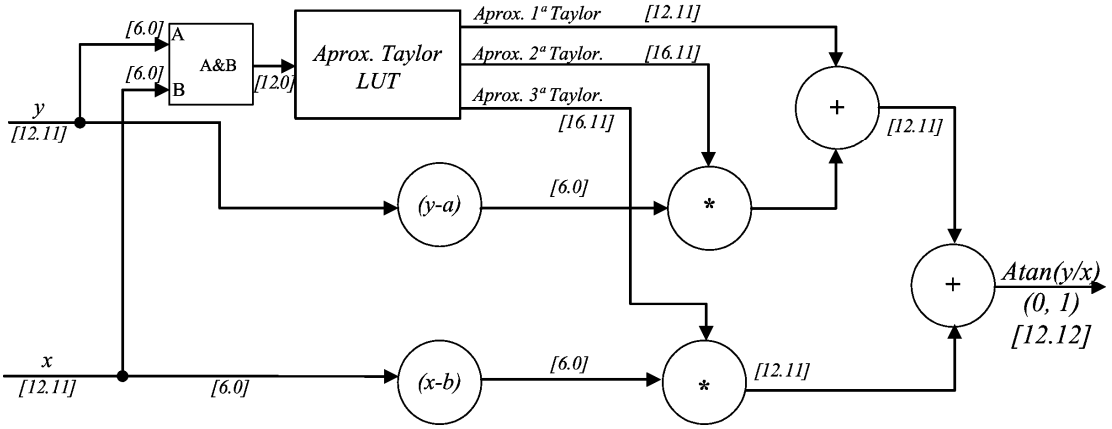


Figura 2.17 - Implementación de la aproximación de la atan(*y*/*x*) mediante la serie de Taylor de dos variables.

Esta aproximación ha sido generada mediante la herramienta System Generator de Xilinx para diferentes precisiones de salida y dos etapas de segmentado. La tabla 2.15 muestra los resultados obtenidos después del Place & Route. Para la implementación de 20 bits hemos utilizado un dispositivo XC2V6000. Para precisiones de 22 bits no hemos podido implementarlo en FPGA, ya que no había dispositivos dentro de la familia Virtex-2 con suficiente número de Block RAM.

Precisión	**12**	**14**	**16**	**18**	**20**	**22**
Mult18x18	2	2	6	6	6	
Slices	185	207	242	298	330	-----
LUT4	295	338	411	467	541	-----
F.F.	190	215	228	242	270	-----
Block RAM	12	33	132	132	528	-----
Fmax (MHz)	125.1	118.7	105.4	98.9	94.3	-----
Latencia	10	10	13	13	13	
Throughput (Msps)	125.1	118.7	105.4	98.9	94.3	-----

Tabla 2.15 - Resultados de implementación de la aproximación de la atan*(y/x)* mediante el método de Taylor y modificando la precisión de la aproximación.

2.3. Conclusiones

En este capítulo hemos presentado los diferentes métodos para aproximación de funciones como puede ser CORDIC, métodos basados en tablas bipartidas y multipartidas, aproximaciones por polinomios, aproximaciones por división de polinomios, interpolaciones lineales, etc. Para todos los métodos de aproximación se ha realizado una implementación de una función elemental atan*(y/x)* la cual es ampliamente utilizada en los sistemas de comunicaciones digitales. Además, podemos ver los grandes

inconvenientes que tenemos cuando aproximamos funciones de más de una variable. Todas estas implementaciones han sido realizadas en FPGAs mediante su modelado en Xilinx System Generator. El gran problema encontrado en todas las implementaciones es la realización de la operación de la división ya que ha aumentado la complejidad de la implementación. Para la realización de la división se ha optado por utilizar un core-IP de Xilinx que realiza la división *non-restoring* en base 2. Este divisor esta segmentado completamente y genera una división completa por cada ciclo de reloj. Para el caso de los métodos de convergencia lineal se ha implementado un divisor *non-restoring* modelado en VHDL y que genera un bit de la división en cada iteración. De esta manera, los resultados de implementación del operador de división no son tan significativos con respecto a la aproximación de la arcotangente. Utilizando este divisor podemos alcanzar grandes velocidades pero con un coste hardware excesivamente grande. Para realizar divisiones con una precisión de 22 bits utilizamos un 10% del número de *slices* disponibles en la FPGA utilizada (Virtex-2 v3000).

La aproximación por Taylor evita realizar la operación de división pero tiene el inconveniente de la gran cantidad de memoria necesaria, sobre todo para precisiones elevadas, haciendo inviable su implementación en FPGAs. Para bajas precisiones el operador CORDIC ofrece las mayores velocidades. Segmentando completamente el operador podemos llegar a velocidades de hasta 165 Msps. Por otro lado los métodos basados en tablas y más concretamente los métodos basados en tablas Multipartidas nos ofrecen la ventaja de los bajos requerimientos de memoria necesaria pero no soluciona el problema de la división. Los métodos de convergencia lineal tienen la ventaja del bajo área necesaria pero tienen el inconveniente de la alta latencia del operador y bajo *throughput* obtenido. La arquitectura basada en aproximación por polinomios no necesita ninguna Block-RAM pero necesitamos hasta 12 multiplicadores. Todas las implementaciones obtienen velocidades superiores a las marcadas en las especificaciones del problema pero en todos los casos el área y la latencia del operador son excesivamente grandes. En la tabla 2.16 y 2.17 presentamos una comparación de los resultados obtenidos por los distintos métodos para una baja (14 bits) y alta precisión de salida (20 bits), respectivamente.

Método	CORDIC	Conv. lineal	Interpol. lineal	LUTs Bipartidas	SBTM	STAM	Polinomios	Taylor
Mult18x18	0	0	1	0	0	0	3	2
Slices	402	219	819	782	780	775	805	207
LUT4	680	288	537	490	480	495	560	338
F.F.	689	269	1270	1210	1210	1235	1256	215
Block RAM	0	0	1	1	1	1	0	33
F_{max} (MHz)	160.5	192	138.2	172.7	172.2	172	148.9	118.7
Latencia	17	30	22	20	20	22	28	10
Throughput (Msps)	160.5	6.4	138.2	172.7	172.2	172	148.9	118.7

Tabla 2.16 - Comparativa de resultados de las diferentes implementaciones de la aproximación de la atan*(y/x)* para una salida con una baja precisión.

A partir de los resultados para bajas precisiones presentados, vemos como la implementación basada en convergencia lineal es del orden de 18.5 veces más lenta que la siguiente menos rápida (Taylor), por el contrario es la que menos área ocupa. Los métodos basados en LUTs, en sus distintas vertientes, presentan resultados parecidos en cuanto al número de Block-RAMs utilizadas, esto es debido a las pequeñas cantidades de memoria utilizadas. Como vemos el método de Taylor, al aproximar directamente

la función de dos variables necesita una cantidad de memoria mucho mayor que el resto. Por otro lado, al no necesitar de la realización de la operación de división, es la implementación que menos área necesita.

Método	CORDIC	Conv. lineal	Interpol. lineal	LUTs Bipartidas	SBTM	STAM	Polinomios	Taylor
Mult18x18	0	0	3	0	0	0	9	6
Slices	801	325	1420	1359	1316	1335	1365	330
LUT4	1091	379	998	975	906	934	958	541
F.F.	1234	405	2395	2357	2357	2267	2312	270
Block RAM	0	0	2	6	5	5	0	528
F_{max} (MHz)	155.4	175.4	118	165	165	164.6	128.6	94.3
Latencia	23	42	30	26	26	28	42	13
Throughput (Msps)	155.4	4.2	118	165	165	164.6	128.6	94.3

Tabla 2.17 - Comparativa de resultados de las diferentes implementaciones de la aproximación de la atan*(y/x)* para una salida con una alta precisión.

Los resultados de implementación para altas precisiones de trabajo muestran que al igual que para bajas precisiones, el método basado en convergencia lineal, es el que menos área ocupa y también sigue siendo el que menos velocidad presenta (22.5 veces más lento que el segundo más lento). También vemos que el número de Block-RAMs necesarias para la aproximación de Taylor ha aumentado considerablemente, haciendo inviable su implementación en gran cantidad de FPGAs. La implementación de Taylor de 20 bits se ha realizado utilizando una FPGA Virtex-2 V6000, el dispositivo más grande de la familia de FPGAs de Xilinx Virtex-2. Vemos que los requerimientos de área de los distintos métodos de compresión de tablas, presentas resultados parecidos. Por último, en la implementación por polinomios, al aumentar el tamaño de palabra por encima de los 18 bits de los multiplicadores embebidos de la Virtex-2, ha aumentado el número de multiplicadores (3 por cada multiplicación) y aumentando por consiguiente la latencia de la implementación.

A partir de los resultados obtenidos vamos a centrarnos en los siguientes capítulos en proponer una serie de arquitecturas que nos permitan obtener la aproximación de la atan*(y/x)* de una manera más eficiente. De los resultados obtenidos por las distintas implementaciones, vemos que será muy importante la realización de la división de las dos entradas, ya que esta operación es el principal cuello de botella de todas las implementaciones. También podemos apreciar, que los métodos basados en LUTs ofrecen las mejores velocidades, a costa, de grandes tamaños de memorias. Para ello, será importante la utilización de los métodos de compresión de tablas ya que permiten reducir el tamaño de las LUTs necesarias sin aumentar el área necesaria.

Capítulo 3.

Aproximación de la Atan(*y/x*) basada en Métodos de Tablas

En este capítulo se presenta una arquitectura para la aproximación de la función elemental atan(y/x) mediante la utilización de métodos basados en Tablas Look-up (LUTs). Esta arquitectura puede ser utilizada para sistemas de comunicaciones Software Defined Radio (SDR) con lo cual se debe poder trabajar con tasas de 20Msps. Para su implementación utilizaremos FPGAs de Xilinx, en concreto una Virtex-2 XC2V3000. La arquitectura propuesta tiene la ventaja de utilizar los hard-cores embebidos disponibles en la FPGA (memorias Block-RAM y Multiplicadores mult18x18), permitiendo reducir el consumo de potencia de la implementación con respecto a otras posibles soluciones, todo ello con una baja área. La arquitectura propuesta estará compuesta por tres bloques bien diferenciados. Primero realizaremos una reducción de rango de las entradas para aproximar posteriormente las funciones elementales de una manera eficiente. Segundo, realizaremos las aproximaciones mediante tablas de las funciones elementales de la división y arcotangente sobre las nuevas entradas de un rango menor (y', x'), permitiendo reducir el tamaño de las tablas necesarias. Por último, una vez aproximada la atan(y'/x'), expandiremos el resultado al rango original de las entradas mediante la utilización de propiedades de la arcotangente. De esta manera reducimos considerablemente el tamaño de las tablas necesarias.

3.1. Introducción

La manera más fácil para aproximar una función es mediante el uso de LUTs, debido sobre todo a su baja latencia y alta velocidad. Pero tienen el inconveniente de que el tamaño de las LUTs depende directamente de la precisión necesaria, como se ha visto en el capítulo dos. Para la gran mayoría de los casos este tamaño puede ser excesivamente grande, sobre todo en el caso de funciones con más de una variable, haciendo inviable su implementación, así que necesitamos de métodos que nos permitan reducir el tamaño de las tablas utilizadas. Un esquema de reducción "dos-entradas a una-entrada" puede ser aplicado para reducir el tamaño de las LUTs [27], permitiendo que la función de dos variables pueda ser evaluada por medio de una función auxiliar unaria y permitiendo, de esta manera, reducir el tamaño de la LUT implementada. Nuestra arquitectura constará de una etapa de pre-procesado sobre los dos operandos de entrada que nos permita reducir el rango de las dos entradas de la función, posteriormente realizaremos la aproximación mediante tablas Look-up en el rango reducido de las entradas y una etapa de post-procesado que nos permita expandir la aproximación sobre el rango original. P.T.P. Tang [16] propone una serie de consejos generales para reducir el rango y aproximar funciones elementales como se ha visto en el capítulo anterior. Métodos más específicos para la reducción rango han sido propuestos por Shulte & Swartzlander [38] y Ferguson [39]. Para reducir el tamaño de las tablas LUT podemos utilizar métodos de particionado de tablas o utilizar interpoladores lineales [35]. Un método para dividir las tablas LUT consiste en dividir el operando x entrada en n dígitos (como se ha visto en el capítulo dos) y construir una tabla con cada digito y sumar las salidas con su peso correspondiente para generar el valor correcto de la aproximación. Los métodos Bipartidos [30] y sus variaciones [31][32] y los métodos multipartidos [33][34] son ejemplos de particionado de tablas que reducen enormemente el tamaño de la misma dividiendo la tabla LUT original en hasta 6 tablas para precisiones de hasta 24 bits con una buena precisión.

Centrándonos en la función elemental arcotangente se han propuesto diferentes arquitecturas que pasamos a presentar. Para números en coma flotante J.M. Muller [14] y P.T.P. Tang [16] proponen diferentes polinomios de alto grado los cuales combinados con técnicas basadas en tablas permiten obtener precisiones de 0.6 *ulp*[1] trabajando con valores en coma flotante de 80 bits. S. Rajan y otros [40] proponen diferentes polinomios para aproximar la arcotangente de una variable para bajos requerimientos de error en la aproximación. El problema de estas aproximaciones es que necesitamos realizar la división entre las dos entradas. Los mismos autores en [41] proponen diferentes aproximaciones para la atan(y/x) mediante el uso de interpoladores de Lagrange y realizando una optimización *minimax* de los coeficientes propuestos, permitiendo obtener aproximaciones de hasta un error de 0.00075 rad. En [42] realizan una aproximación de la arcotangente de dos variables mediante el uso de la transformación de Möbius, permitiendo aumentar la precisión de la aproximación con respecto a [41]. Arquitecturas en las cuales se intenta reducir el consumo de potencia han sido propuestas por [43], utilizando para ello interpoladores lineales e implementando la división de las dos entradas mediante un divisor paralelo *non-restoring*. Hwang y otros presentan en [44] una arquitectura que aproxima la atan(y/x) dividiendo el cálculo en dos etapas, utilizando para ello LUTs y multiplicadores de reducido tamaño. A continuación pasamos a presentar nuestra arquitectura hardware para la aproximación de la atan(y/x).

[1] *ulp*. Unit of Least Precision. Diferencia existente entre el número en coma flotante representado y el valor real dado.

3.2. Arquitectura Propuesta

Como se ha comentado previamente la manera más fácil para aproximar una función es mediante el uso de una LUT donde almacenamos previamente los datos de la función a aproximar con una precisión requerida y mediante la entrada direccionaremos la tabla para obtener el valor de la aproximación. Para el caso de una función de dos variables *f(x,y)*, como la atan(y/x), y utilizando un tamaño de palabra para las entradas de x e y bits, la aproximación mediante una tabla LUT requerirá un tamaño $2^{(x+y)} \cdot k$ bits, siendo k el tamaño de la palabra almacenada. Por ejemplo, el sistema de comunicación seleccionado suele trabajar con precisiones de 12 bits, tanto en la entrada como en la salida, siendo necesaria una tabla de 176 Mbits. Es por ello que necesitamos implementar alguna técnica que nos permita reducir el tamaño de las tablas para poder implementar dicha aproximación eficientemente. Para solucionar este crecimiento exponencial del tamaño de las tablas vamos a aplicar etapas de pre-procesado y post-procesado sobre las entradas y salidas de la LUT, respectivamente. Este esquema se puede ver en la figura 3.1.

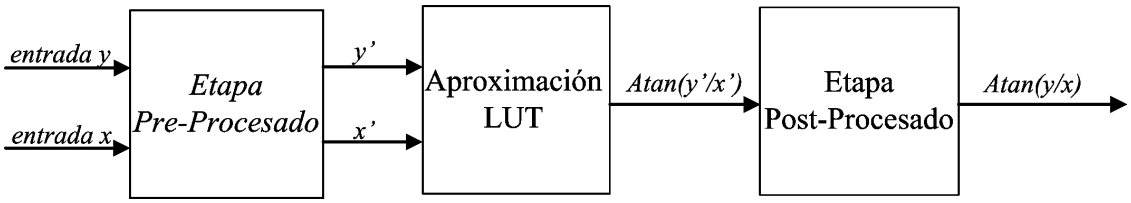


Figura 3.1 - Arquitectura propuesta para la aproximación de la atan(y/x) mediante LUTs.

Para ello vamos a realizar la aproximación de la función de dos entradas por medio una función auxiliar de una única entrada, con lo cual el tamaño de memoria será únicamente $2^n \cdot k$ bits, siendo n el tamaño de palabra de la entrada auxiliar y k el tamaño de palabra de su salida. Gracias a la tabla de una única entrada y las etapas de pre-procesado y post-procesado podremos evaluar sin problemas nuestra función de dos entradas. De esta manera, la función de dos entradas puede ser evaluada por medio de una función auxiliar de una entrada y así conseguimos reducir el tamaño de la LUT necesaria. Básicamente, la arquitectura propuesta consistirá en la aplicación de una etapa de pre-procesado sobre los operandos de entrada que nos permitirá reducir el rango de trabajo de la función, una aproximación de la función mediante una tabla sobre el rango reducido y por último, una etapa de post-procesado que nos permitirá expandir la aproximación sobre el rango original de las entradas de la función.

Para el cálculo de la atan(y/x) vamos a dividir la operación en dos partes: primero realizaremos la operación ($z=y/x$) para obtener una variable y segundo calcularemos la atan(z). Para el cálculo de la variable z, realizaremos primero el cálculo del recíproco de $1/x$ y posteriormente multiplicaremos ese resultado por la entrada y. Para el cálculo de recíproco de $1/x$ y de la atan(z) haremos uso de métodos basados en LUTs de una sola variable. En ambos casos, el método escogido para la aproximación de las dos funciones elementales mediante tablas LUT será el método bipartido desarrollado por [32] que nos permitirá reducir el tamaño de las mismas.

Además de los métodos bipartidos para la reducción de las tablas, haremos uso de varias propiedades de la arcotangente para reducir aún más dicho tamaño. Primero, únicamente será necesario calcular los valores positives de z ya que los valores negativos pueden ser obtenidos a partir de

$$\text{atan}(-z) = -\text{atan}(z). \tag{3.1}$$

Segundo, únicamente debemos calcular la aproximación para el rango comprendido (0, 1) y extenderemos en rango de trabajo mediante

$$\operatorname{atan}\left(\frac{y}{x}\right) = \operatorname{sign}\left(\frac{y}{x}\right)\cdot\left(\frac{\pi}{2}\right) - \operatorname{atan}\left(\frac{x}{y}\right). \tag{3.2}$$

A través de la etapa de pre-procesado detectaremos el caso en el que la división sea negativa y mayor que uno para convertirlos a valores positivos y menores que uno. Posteriormente, en la etapa de post-procesado, corregiremos las transformaciones realizadas previamente aplicando (3.1) y (3.2). Este esquema se puede ver en la figura 3.2. Los anchos de palabra utilizados están indicados en todas las figuras mediante el formato [N. F], siendo N el tamaño de palabra y F el numero de bits que indican la parte fraccionaria.

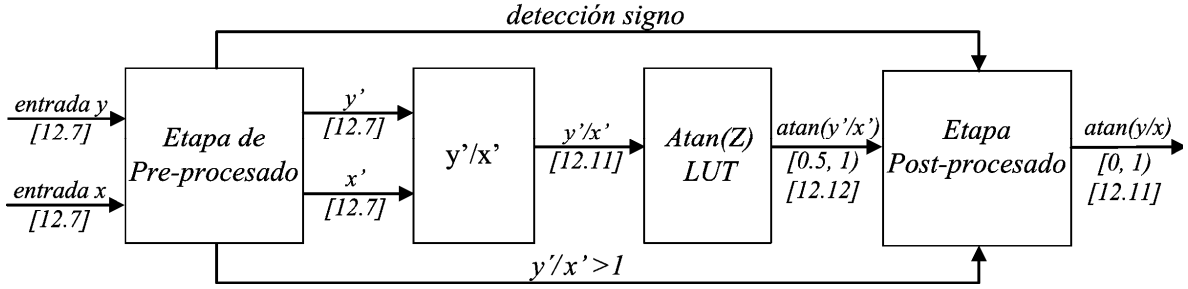


Figura 3.2 – Diagrama de bloques de la aproximación de la atan(y/x) basada en la división de las dos entradas y la utilización de LUTs.

3.2.1. Etapa de Pre-procesado

La etapa de pre-procesado consiste de dos circuitos que realizan el complemento a dos (CMP2s) de la entrada, en el caso de ser valores negativos, o los deja pasar tal cual en el caso de ser positivos. Mediante un comparador y dos multiplexores detectamos si $|y|>|x|$ e intercambiamos las entradas para asegurar que la división siempre es menor que uno. De este bloque saldrán dos salidas que serán enviadas a la etapa de post-procesado: una señal con la información del signo de la división (3.1) y otra señal con la información de la salida del comparador. Este esquema se puede ver en la figura 3.3.

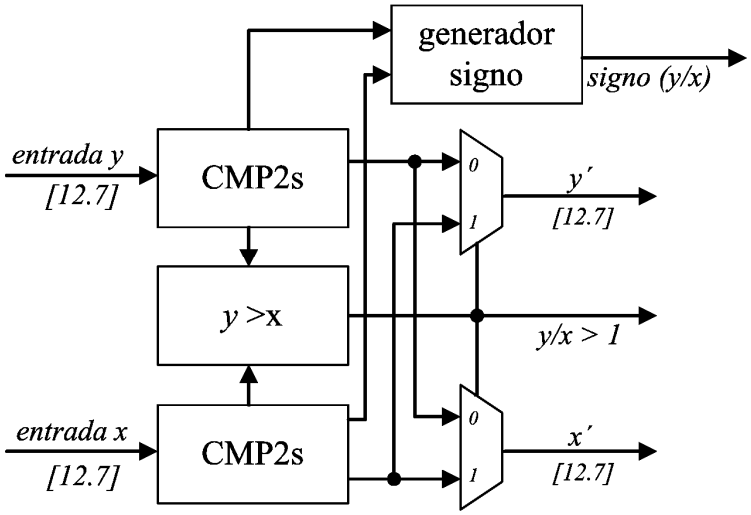


Figura 3.3 - Etapa de pre-procesado.

3.2.2. Etapa de Post-procesado.

En la etapa de post-procesado se computan las ecuaciones (3.1) y (3.2) a partir de las señales de control enviadas por la etapa de pre-procesado. Esta etapa se compondrá de un circuito que realiza el complemento a dos de la entrada, un sumador y un multiplexor. Al trabajar con la fase normalizada a la salida y en vez de restar (π/2) en (3.2), utilizaremos el valor de uno. Este esquema se puede ver en la figura 3.4.

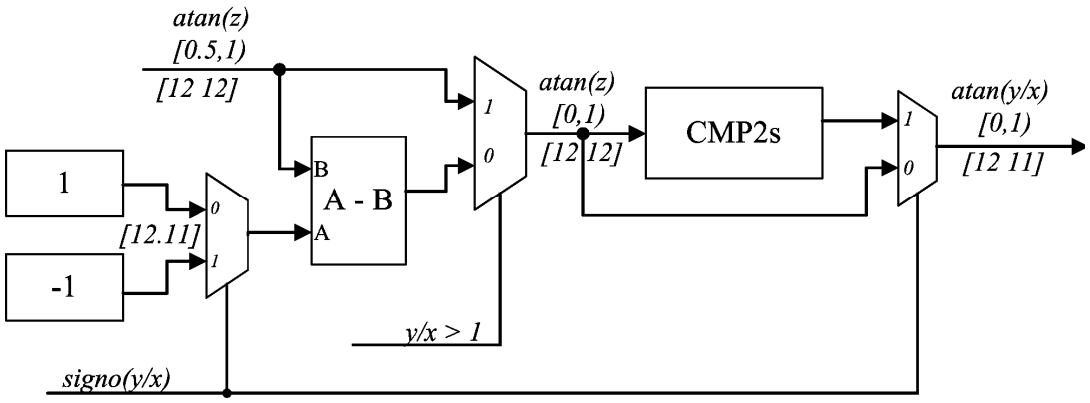


Figura 3.4 - Etapa de post-procesado.

3.2.3. Cálculo de la División

Para el cálculo de la atan(*y*/*x*) primero realizaremos la división de las dos entradas y a partir de este resultado calcularemos la aproximación de la arcotangente. Para realizar la división seguiremos las recomendaciones propuestas por P.T.P. Tang [16] para la implementación del cálculo del recíproco de la entrada *y* mediante métodos basados en LUT. Básicamente estas recomendaciones consisten en la división del cálculo en tres etapas: primero, una reducción en el rango de trabajo; segundo, una aproximación basada en LUTs y, por último, una extensión del resultado sobre el rango original de trabajo. Este esquema se puede ver en la figura 3.5

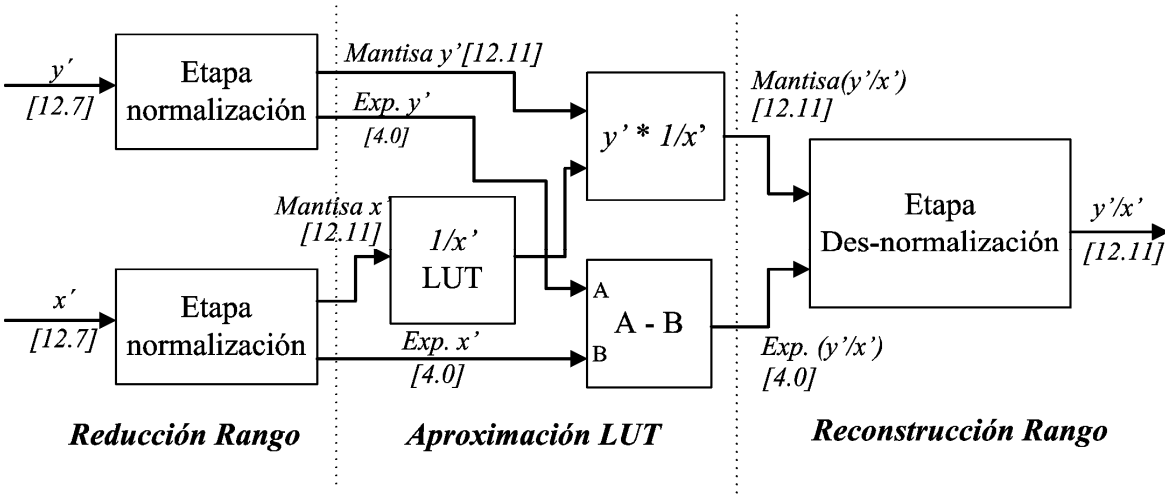


Figura 3.5 - Cálculo de la división y'/x'.

A partir del esquema definido en la figura 3.7, lo primero será reducir el rango de de las entradas. En el caso de nuestras especificaciones basadas en la implementación de un receptor en OFDM [45], el rango de las entradas estará comprendido en ±[0, 16) y reduciremos ese rango de trabajo a uno más conveniente: [0.5, 1). Esta reducción de rango la realizaremos en el bloque normalizador. Este bloque estará compuesto de un circuito *Leading Zero Detector (LZD)* basado en los esquemas propuestos por Oklobdzija [46], el cual nos permitirá detectar cuantos desplazamientos hacia la izquierda necesitamos para que nuestra entrada este comprendida dentro del rango [0.5 – 1) y mediante un *barrel-shifter* realizaremos el desplazamiento indicado. De esta manera el bit más significativo siempre será '1' y estaremos dentro del rango exigido. Este bloque de reducción de rango se comportará como un bloque conversor de coma-fija a coma-flotante. En la figura 3.6 se puede ver la estructura de la etapa normalizadora.

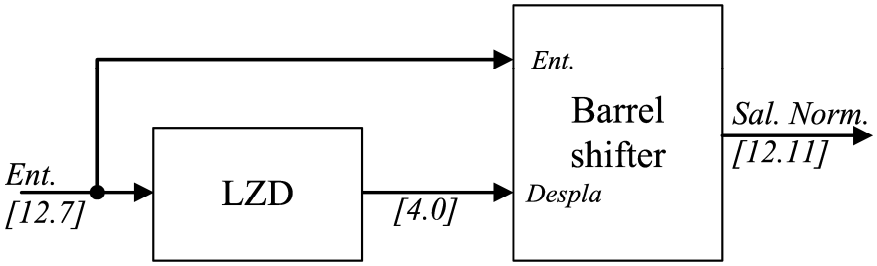


Figura 3.6 - Bloque normalizador de las entradas para el divisor.

Se han utilizado los LZD propuestos por Oklobdzija debido a la facilidad de diseñar cualquier tamaño de LZD a partir de bloques básicos de LZD de dos bits. En la figura 3.7.a y 3.7.b se pueden ver cómo se han implementado LZD de 2 y 8 bits respectivamente. En estos esquemas, la salida 'p' indica el valor del desplazamiento y 'v' es una señal de habilitación que indica que la entrada es válida. El valor obtenido 'p' es utilizado para la realización del desplazamiento hacia la izquierda de las entradas *x* e *y*. Para la realización de la operación del desplazamiento mediante el *barrel-shifter* utilizaremos un multiplicador embebido de la FPGA como se indica en la figura 3.8. El *barrel-shifter* implementado mediante multiplicadores puede desplazar vectores de hasta 18 bits hacia la izquierda indicando el desplazamiento por medio de la entrada `despla`. El valor del desplazamiento indicado en binario debe ser convertido a codificación "*one-hot*" y para ello utilizamos una LUT. Se ha optado por implementar el barrel-shifter con un multiplicador embebido en vez de implementarlo mediante las *logic-cells* para reducir el consumo de potencia y alcanzar alta velocidad.

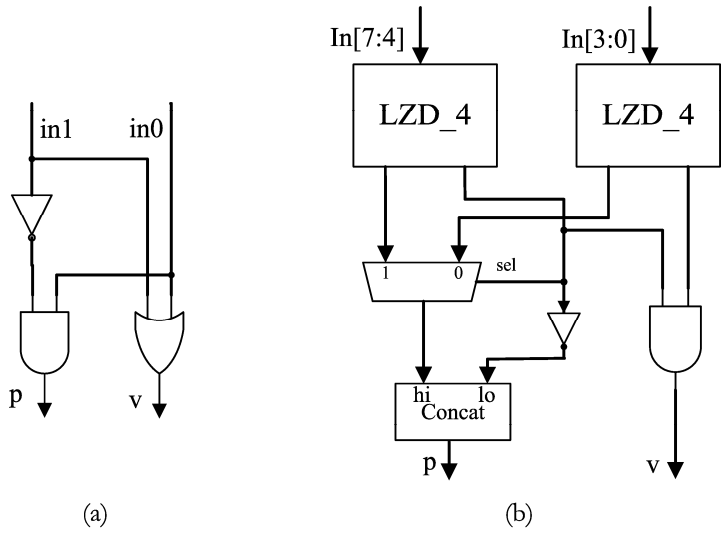


Figura 3.7.a - LZD de dos bits. **Figura 3.7.b** - LZD de ocho bits.

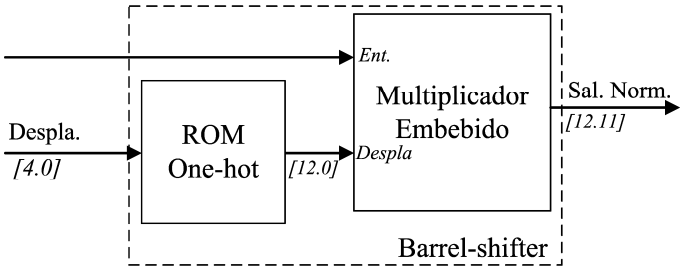


Figura 3.8 - *Barrel-shifter* implementado mediante multiplicadores.

Una vez tenemos normalizada la entrada *x'*, la cual siempre estará comprendida en el rango de trabajo $0.5 < x' \leq 1$, la utilizaremos para direccionar la tabla en la que tenemos almacenada la aproximación al valor del recíproco. La salida del recíproco estará comprendida en el rango $1 \leq 1/x' < 2$, con lo cual el bit más significativo del recíproco siempre será uno y no será necesario almacenarlo en la LUT. Posteriormente será añadido a la aproximación poniendo a "uno" el bit más significativo.

Además de las reducciones comentadas anteriormente podemos reducir el tamaño de la tabla LUT utilizada para aproximar el recíproco mediante particionado de tablas [32]. El método SBTM ha sido el utilizado para la implementación de la tabla usada en la aproximación. Se ha optado por este método bipartido y no otros métodos basados en más de dos tablas debido al mejor rendimiento que obtenemos con el uso de las memorias embebidas de doble puerto disponibles en la FPGA. De esta forma implementamos las dos tablas con una única memoria de doble puerto, utilizando cada uno de los dos puertos para acceder a cada tabla, permitiendo reducir el consumo de la implementación con respecto a una implementación basada en memoria distribuida. En el punto 3.2, hemos implementado las aproximaciones de la atan(z) y $1/x$ utilizando métodos basados en tablas multipartidas utilizando el algoritmo STAM [34], el cual nos permite reducir el tamaño de las tablas aumentando el número de éstas.

En el método bipartido la tabla con la aproximación es dividida en dos tablas de menor tamaño y las salidas son sumadas para obtener la aproximación:

$$f(x) = f(x_0 + x_1 + x_2) \approx a_0(x_0, x_1) + a_1(x_0, x_2). \tag{3.3}$$

La entrada x de n-bits se dividirá en tres palabras x_0, x_1, x_2 con n_0, n_1, n_2 bits, respectivamente donde $x = x_0 + x_1 \cdot 2^{-n_0} + x_2 \cdot 2^{-(n_0+n_1)}$ asumiendo, $0 \leq x \leq 1$. La expansión de Taylor de primer orden será,

$$f(x) = f(x_0 + x_1 \cdot 2^{-n_0}) + x_1 \cdot 2^{-(n_0+n_1)} \cdot f'(x_0 + x_1 \cdot 2^{-n_0}) + \varepsilon, \tag{3.4}$$

y se utilizará para aproximar la ecuación (3.3) y obtener los valores de las dos tablas:

$$\begin{aligned} a_0(x_0, x_1) &= f(x_0 + x_1 \cdot 2^{-n_0}) \\ a_1(x_0, x_1) &= x_2 \cdot 2^{-(n_0+n_1)} \cdot f'(x_0). \end{aligned} \tag{3.5}$$

El error cometido en dicha aproximación es $\varepsilon \approx 2^{-n} \, max|f''|$, que para el caso de la división es igual $\varepsilon \approx 2^{-n}$, siendo n el tamaño de palabra de la memoria.

Los valores almacenados en cada una de las dos tablas han sido calculados a partir del algoritmo propuesto por DasSarma y Matula [35], el cual garantiza el resultado con un error menor de 1 *ulp*. La

partición óptima ha sido estudiada para minimizar el tamaño de las dos tablas. Para una precisión de entrada de 12 bits el particionado óptimo obtenido es igual a $n_0 = 4$, $n_1 = 4$ y $n_2 = 4$, siendo necesarios 5632 bits. Tanto el particionado óptimo como los valores de las dos tablas han sido calculados a partir de una función realizada en Matlab. En la tabla 3.1 podemos ver las particiones óptimas obtenidas para la aproximación del recíproco para diferentes precisiones de salida.

Precisión	10	12	14	16	20
Particionado óptimo	n_0=4,n_1=3, n_2=3	n_0=4,n_1=4, n_2=4	n_0=5,n_1=5, n_2=4	n_0=6,n_1=4, n_2=6	n_0=8,n_1=6, n_2=6
ROMs	a_0 128 x 9 bits	a_0 256 x 11 bits	a_0 512 x 13 bits	a_0 2048 x 15 bits	a_0 8192 x 19 bits
	a_1 64 x 9 bits	a_1 256 x 11 bits	a_1 512 x 13 bits	a_1 1024 x 5 bits	a_1 4096 x 7 bits

Tabla 3.1 - Particionado óptimo de la LUT bipartida utilizada para la aproximación del recíproco.

Esta memoria puede ser implementada mediante una memoria de doble puerto disponible en las FPGAs de Xilinx y será configurada como una ROM de 1kx16 bits. Cada una de las dos tablas que componen la memoria bipartida será implementada como una página de memoria y cada uno de los dos puertos direccionará una página. Los valores de la tabla A_1 se han almacenado con todos los bits en la Block-RAM (igual tamaño que los coeficientes de la tabla A_0) debido a que no obteníamos ninguna reducción en el número de Block-RAM utilizadas, para precisiones bajas. Para precisiones altas al utilizar diferentes Block-RAM para las tablas A_0 y A_1 sí que hemos eliminado los "unos" o "ceros" más significativos iguales. Este esquema de implementación puede verse en la figura 3.9 para un tamaño de salida de 12 bits.

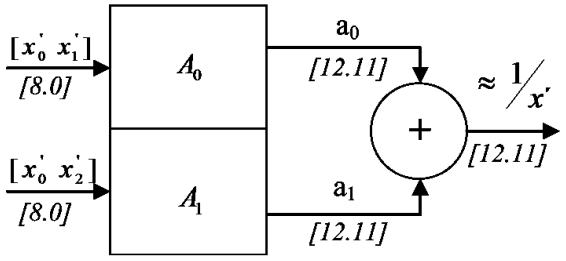


Figura 3.9 – Esquema de memoria bipartida utilizada para la aproximación del recíproco.

Por último, nos quedará multiplicar el recíproco $1/x'$ por la otra entrada normalizada y' para obtener la división. A partir del resultado de la división normalizada, sólo nos quedara extender el resultado al rango original. Esta operación se ha implementado mediante un *barrel-shifter* combinacional ya que el sentido de la operación es distinto que para el caso de la etapa normalizadora y, por eso es imposible utilizar otro multiplicador. La diferencia de los dos exponentes (la salida de los dos circuitos LZD) es utilizada para calcular el desplazamiento hacia la derecha necesario para expandir la división a su rango original. Este resultado será utilizado en la etapa siguiente para direccionar la aproximación de la arcotangente mediante tablas.

3.2.4. Cálculo de la Atan(z)

El método bipartido también ha sido utilizado para la implementación de la aproximación de la atan(z). Para este caso, no es necesario una reducción del rango de la función debido al hecho de que la entrada está dentro del rango deseado $0 \leq z < 1$. El rango de salida estará comprendido entre 0 y 0.5, ya que trabajamos con la fase normalizada. La partición que mejor minimiza el tamaño de la memoria es n_0, n_1 y n_2 de 4 bits. El esquema de la implementación es parecido al del cálculo del recíproco. En la tabla 3.2 podemos ver los particionados óptimos para distintas precisiones de salida.

Precisión	10	12	14	16	20
Particionado óptimo	n_0=4,n_1=3, n_2=3	n_0=4,n_1=4, n_2=4	n_0=5,n_1=4, n_2=5	n_0=5,n_1=5, n_2=6	n_0=7,n_1=6, n_2=7
ROMs	a_0 128 x 10 bits	a_0 256 x 12 bits	a_0 512 x 14 bits	a_0 1024 x 16 bits	a_0 4096 x 19 bits
	a_1 128 x 10 bits	a_1 256 x 12 bits	a_1 1024 x 5 bits	a_1 2048 x 6 bits	a_1 8192 x 6 bits

Tabla 3.2 - Particionado óptimo de la LUT bipartida utilizada en la aproximación de la atan(z).

3.3. Implementación y Resultados

Una vez se ha diseñado la arquitectura propuesta para la aproximación de la atan(y/x) necesitamos validar el diseño para ver si cumplimos las especificaciones marcadas, para ello, se ha desarrollado un modelo de precisión finita mediante la herramienta System Generator de Matlab. En la Figura 3.10 se muestra el error resultante cuando el modelo de precisión finita es excitado con todas las posibles entradas (2^{24} posibles entradas). Se obtiene que para el peor caso tenemos una precisión de salida de 10.3 bits, cumpliendo con las especificaciones.

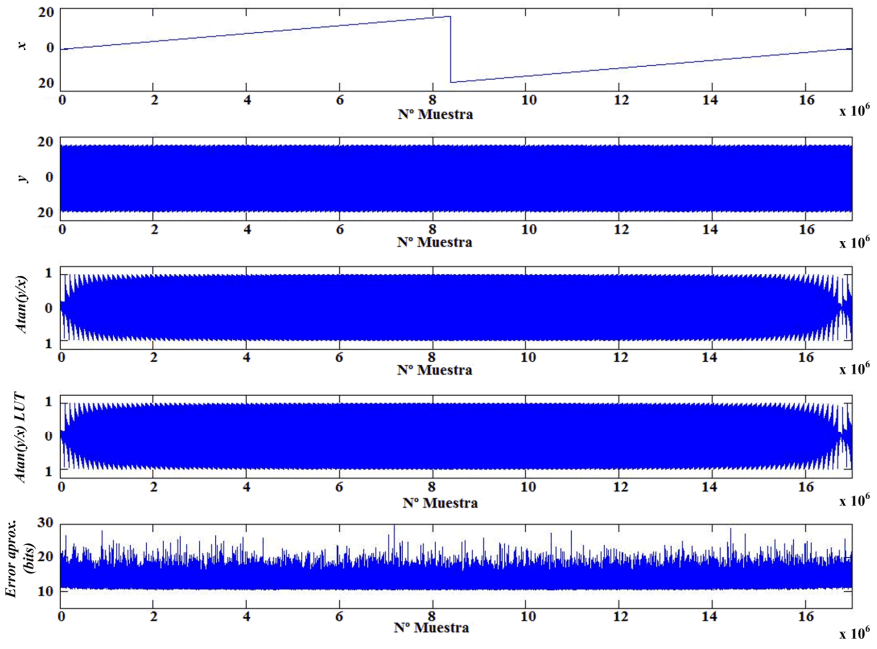


Figura 3.10 - Simulación del error cometido en la arquitectura propuesta expresado en bits.

La arquitectura propuesta ha sido implementada en el dispositivo Virtex 2 XC2V3000-4 de Xilinx. El circuito ha sido implementado modificando el número de etapas de segmentación. Los resultados de la implementación se muestran en la tabla 3.3 y han sido obtenidos mediante la herramienta Xilinx ISE 10.1.

Etapas de Segmentación	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Slices	157	160	160	170	176	183	180	175
LUT4	260	252	285	254	258	269	264	275
F.F.	62	96	91	102	112	120	124	148
Mult18x18	3							
Block RAM	2							
Fmax (MHz)	44.5	48.5	56.6	66.7	83.3	84.9	91.3	93.6
Latencia	3	4	5	6	7	8	9	10
Throughput (Msps)	44.5	48.5	56.6	66.7	83.3	84.9	91.3	93.6

Tabla 3.3 - Resultados de implementación de la arquitectura propuesta modificando el número de etapas de segmentación.

A partir de los resultados indicados en la tabla 3.3 podemos ver que para bajas latencias cumplimos los requerimientos en frecuencia para poder trabajar en sistemas OFDM típicos (20 Msps). Tres ciclos de reloj es la latencia mínima que podemos obtener debido al uso de las memorias embebidas de doble puerto usadas en el diseño, ya que éstas son síncronas y tienen una latencia de un ciclo de reloj. Cabe indicar que las entradas y salidas de la arquitectura propuesta están también registradas. Vemos como únicamente son necesarias dos Block-RAM para realizar la aproximación. De los tres multiplicadores embebidos, dos son utilizados para la implementación de los *barrel-shifter* de la etapa de normalización del cálculo de la división. Por último, vemos que conforme vamos aumentando el número de etapas de segmentado vamos aumentando la tasa de muestras generadas hasta valores cercanos a 100Msps. Para poder hacer los resultados más genéricos, se ha implementado la arquitectura con diferentes tamaños de salida desde 12 a 22 bits. Estos resultados se muestran en la tabla 3.4. Podemos ver que para precisiones de hasta 18 bits, los requerimientos hardware son relativamente bajos. Para precisiones más altas serian necesarios otros esquemas de particionado que nos permitan reducir el número de memorias utilizadas. Por ejemplo, para las precisiones de 20 y 22 bits podemos reducir el número de Block-RAM de 20 y 60 a 4 y 6 Block-RAM respectivamente aplicando un esquema de tablas multipartidas, con un aumento pequeño en el número de *slices* utilizados. Por otro lado podemos ver como al incrementar el tamaño de palabra también aumenta considerablemente el número de multiplicadores embebidos utilizados. Y por último, podemos apreciar cómo la tasa de muestras generadas para el caso de 22 bits ya no cumplimos el mínimo de los 20 Msps requeridos en nuestras implementaciones. Para aumentar este valor deberíamos incrementar el número de etapas de segmentación.

Precisión	**12**	**14**	**16**	**18**	**20**	**22**
Slices	157	180	201	243	294	342
LUT4	260	278	310	356	405	452
F.F.	62	68	74	86	96	110
Mult18x18	3	3	3	3	12	12
Block RAM	2	2	4	6	20	60
Fmax (MHz)	44.5	41.1	36.3	33.8	21.1	15.7
Latencia	3					
Throughput (Msps)	44.5	41.1	36.3	33.8	21.1	15.7

Tabla 3.4 - Resultados de implementación de la aproximación de la atan(y/x) mediante LUTs y modificando la precisión de la aproximación generada.

Para poder comparar resultados de la arquitectura propuesta hemos realizado una implementación en la cual se ha segmentado completamente el circuito. La latencia del operador es de únicamente 14 ciclos de reloj, valor muy inferior a las arquitecturas implementadas en el capítulo anterior. Los resultados para diferentes tamaños de salida se muestran en la tabla 3.5.

Precisión	**12**	**14**	**16**	**18**	**20**	**22**
Slices	182	185	212	256	304	376
LUT4	264	281	312	360	404	465
F.F.	142	170	186	195	260	288
Mult18x18	3	3	3	3	12	12
Block RAM	2	2	4	6	20	60
Fmax (MHz)	145.4	142.9	139.7	135.9	130.2	126.5
Latencia	14	14	14	14	16	16
Throughput (Msps)	145.4	142.9	139.7	135.9	130.2	126.5

Tabla 3.5 - Resultados de la implementación de alta velocidad de la atan(y/x) mediante LUTs y modificando la precisión de la aproximación generada.

En la tabla 3.6 hemos realizado la misma arquitectura para altas precisiones e implementado las LUTs mediante tablas multipartidas. Para este caso vemos como se reduce considerablemente el número de Block-RAM necesarias, pero como veremos a continuación aumentando el consumo de potencia. Esta opción será necesaria cuando sea más importante el área utilizada del dispositivo que el consumo de potencia.

Precisión	**20**	**22**
Slices	300	342
LUT4	422	472
F.F.	96	110
Mult18x18	12	12
Block RAM	5	8
Fmax (MHz)	23.1	15.7
Latencia	3	
Throughput (MSPS)	23.1	18.7

Tabla 3.6 - Resultados de la implementación para altas precisiones de la atan(y/x) basada en LUTs y utilizando LUTs multipartidas.

Para las medidas de consumo hemos realizado dos implementaciones utilizando un tamaño de palabra de 12 bits y utilizando diferentes etapas de segmentación. Una utilizando todos los elementos disponibles en la FPGA como son, *slices*, Block-RAM y Mult18x18 y otra en la que todos los componentes utilizados han sido implementados utilizando únicamente *slices*. De esta manera podemos comprobar la influencia de la utilización de los componentes embebidos en términos de consumo energético. Los resultados

obtenidos se muestran en la tabla 3.7 y 3.8 para las implementaciones con elementos embebidos y con *slices* únicamente, respectivamente.

Etapas de Segmentación	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Slices	157	160	160	170	176	183	180	175
Mult18x18	3							
Block RAM	2							
Fmax (MHz)	44.5	48.5	56.6	66.7	83.3	84.9	91.3	93.6
Consumo (mW/MHz)	1.6	1.45	1.35	1.27	1.22	1.26	1.23	1.21

Tabla 3.7 - Resultados de las medidas consumo de la aproximación de la atan(y/x) mediante LUTs y modificando el número de etapas de segmentación y utilizando elementos embebidos de la FPGA.

Etapas de Segmentación	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Slices	1038	1071	1044	1040	1044	1053	1080	1104	1091	1096
Fmax (MHz)	16.1	26.4	36.6	41.1	48.2	56.4	59.9	78.4	78.4	81.8
Consumo (mW/MHz)	12.8	9.13	6.32	6.4	5.91	5.81	5.71	5.31	4.89	4.54

Tabla 3.8 - Resultados de las medidas consumo de la aproximación de la atan(y/x) mediante LUTs y modificando el número de etapas de segmentación y únicamente con slices.

El primer resultado que podemos obtener es la reducción de consumo debido a añadir etapas de segmentación. Para el caso de 9 etapas de segmentación la reducción de consumo se puede cuantificar en un 24.3 %. Vemos que este efecto también se repite en los resultados de la tabla 3.7. En este caso la reducción es más acusada ya que reducimos un 64.5%. Vemos que la técnica de segmentación nos permite, o aumentar la velocidad del circuito, o reducir el consumo de potencia si mantenemos la frecuencia. Los circuitos con una alta segmentación son más inmunes a la aparición de *glitches* que los circuitos que no están segmentados, ya que normalmente estos tendrán menos lógica entre registros. Además al aparecer menos *glitches* implicará una menor potencia dinámica disipada durante cada ciclo de reloj [47].

La restricción más importante sobre nuestro diseño era minimizar el consumo de potencia. Para ello hemos realizados varias implementaciones utilizando diferentes métodos con los mismos tamaños de entrada y precisiones y hemos comparado sus consumos de potencia con respecto a nuestra arquitectura. Todas las implementaciones se han realizado utilizando diferentes valores de segmentación (hasta un máximo de 9). El primer candidato ha sido la utilización de tablas multipartidas (hemos utilizado 3 tablas (3.6) y 4 tablas (3.7)). De esta manera evaluamos el impacto ocasionado por la utilización de esquemas bipartidos con respecto a esquemas multipartidos. Hemos visto en el capítulo 2 como el uso de memorias multipartidas conseguía reducir enormemente el tamaño de la memoria necesaria para aproximar una función mediante tablas. Al aumentar el número de tablas debemos forzosamente aumentar el número de Block-RAM utilizadas pero viendo el tamaño tan pequeño de las tablas adicionales no estaríamos utilizando eficientemente las Block-RAM por lo han sido implementadas únicamente utilizando *slices*. Estas particiones han sido obtenidas mediante una aplicación programada en Matlab y nos permiten

minimizar el tamaño de las memorias generando los valores aproximados a la precisión pedida. En las tablas 3.9 y 3.10 podemos ver los tamaños de las tablas utilizadas para la aproximación del recíproco y de la arcotangente, respectivamente.

$$f(x) = f(x_0 + x_1 + x_2 + x_3) \approx a_0(x_0, x_1) + a_1(x_0, x_2) + a_2(x_0, x_3) \tag{3.6}$$

$$f(x) = f(x_0 + x_1 + x_2 + x_3 + x_4) \approx a_0(x_0, x_1) + a_1(x_0, x_2) + a_2(x_0, x_3) + a_3(x_0, x_4) \tag{3.7}$$

	1/x			
Particionado entrada	**3 Tablas**		**4 Tablas**	
	n_0=2,n_1=3, n_2=3, n_3=3		n_0=2,n_1=2, n_2=2, n_3=2, n4=2	
ROMs	a_0	32 x 12 bits	a_0	32 x 12 bits
	a_1	32 x 6 bits	a_1	16 x 6 bits
	a_2	32 x 3 bits	a_2	16 x 3 bits
			a_3	16 x 3 bits

Tabla 3.9 - Particionado óptimo de la LUT multipartida utilizada para la aproximación del recíproco.

	atan(*z*)			
Particionado entrada	**3 Tablas**		**4 Tablas**	
	n_0=3,n_1=3, n_2=3, n_3=3		n_0=3,n_1=3, n_2=2, n_3=2, n4=2	
ROMs	a_0	64 x 12 bits	a_0	64 x 12 bits
	a_1	64 x 5 bits	a_1	32 x 5 bits
	a_2	64 x 2 bits	a_2	32 x 3 bits
			a_3	32 x 1 bits

Tabla 3.10 - Particionado óptimo de la LUT multipartida utilizada para la aproximación de la atan(*z*).

En las siguientes figuras podemos ver el esquema de implementación de dichas tablas, en este caso para aprovechar la reducción de tablas se han implementado todas en memoria distribuida. En la figura 3.11 podemos ver la implementación de la memoria multipartida de 3 tablas y en la figura 3.12 podemos ver el esquema de la memoria multipartida utilizando 4 tablas.

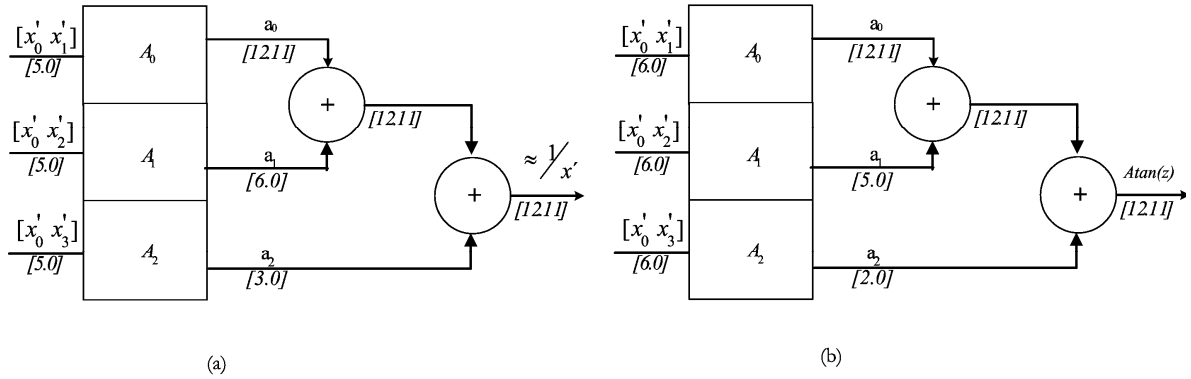


Figura 3.11 - Esquema de las memorias multipartidas (3 LUTs) utilizadas en la aproximación propuesta. (a) recíproco, (b) atan(*z*)

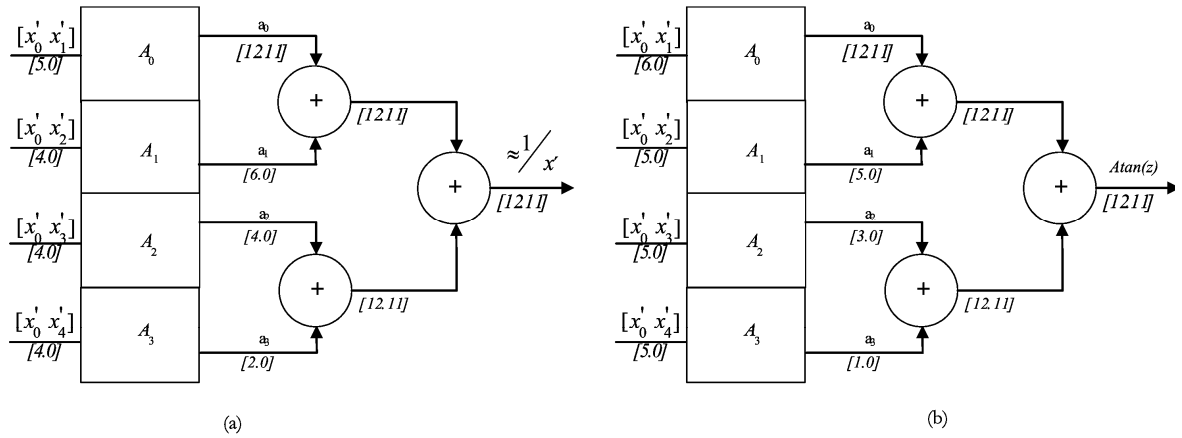


Figura 3.12 – Esquema de las memorias multipartidas (4 LUTs) utilizadas en la aproximación propuesta. (a) recíproco, (b) atan(z)

En la tabla 3.11 vemos los resultados obtenidos de la medida de consumo de potencia de la implementación multipartida de tres LUTs para una precisión de 12 bits y diferentes etapas de segmentado. Se aprecia como se ha duplicado el número de Block-RAM necesarias con respecto a la implementación bipartida. Para el caso de únicamente dos etapas de segmentación, se observa como el consumo ha aumentado un 55.8% con respecto a nuestra propuesta. Conforme se aumenta el número de etapas de segmentación esta diferencia es menos acusada. Para nueve etapas de segmentación el consumo de potencia ha aumentado 22.7%. La diferencia en cuanto al número de *slices* utilizados es debido a la necesidad de añadir un nuevo sumador en cada aproximación.

Etapas de Segmentación	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Slices	172	175	198	196	219	222	254	250
Mult18x18	3							
Block RAM	4							
Fmax (MHz)	40.6	45.8	49.8	62.8	73.2	80.6	85.6	98.8
Consumo (mW/MHz)	2.52	2.1	1.88	1.75	1.65	1.54	1.51	1.48

Tabla 3.11 -Resultados de implementación de la aproximación de la atan(y/x) mediante LUTs multipartidas (3 LUT) y modificando el numero de etapas de segmentación.

En la tabla 3.12 podemos ver los resultados obtenidos en la medida de consumo de potencia de la implementación multipartida de cuatro LUTs para una precisión de 12 bits y diferentes etapas de segmentación. Para este caso el número de LUTs utilizadas es idéntico que en caso de la implementación con tres tablas, debido a que hemos podido meter dos tablas por cada Block-RAM. En el caso de dos etapas de segmentación el aumento del consumo de potencia es un 63.4% con respecto a nuestra arquitectura. El aumento de potencia entre las implementaciones de tres y cuatro tablas no es tan grande, ya que únicamente obtenemos un aumento del 4.3%. Vemos como el número de *slices* ha aumentado debido a que para este caso necesitamos más sumadores para generar la aproximación.

Etapas de Segmentación	2	3	4	5	6	7	8	9
Slices	205	210	249	250	268	260	262	270
Mult18x18	3							
Block RAM	4							
Fmax (MHz)	37.5	42.6	45.9	58.9	68.4	75.9	81.4	84.9
Consumo (mW/MHz)	2.66	2.28	1.98	1.87	1.73	1.64	1.58	1.55

Tabla 3.12 - Resultados de implementación de la aproximación de la atan(*y*/*x*) mediante LUTs multipartidas (4 LUT) y modificando el numero de etapas de segmentación.

3.3.1. Comparación de Consumos por las Distintas Implementaciones

Por último, vamos a añadir los resultados de consumos de potencia obtenidos por algunas de las implementaciones realizadas en el capítulo 3, a las que hemos añadido algunas implementaciones realizadas en el capítulo anterior. Todas las implementaciones se han realizado para una precisión de 12 bits y diferentes etapas de segmentación. En la figura 3.13 podemos ver una grafica en la que hemos comparado los resultados de consumo de potencia obtenidos por las diferentes implementaciones realizadas anteriormente a la que hemos añadido los resultados de las implementaciones basadas en CORDIC, aproximación por polinomios y aproximación por Taylor. La frecuencia de reloj utilizada en las implementaciones ha sido de 20 MHz para todos los casos. Hemos expresado los valores absolutos de las medidas de potencia para resaltar las diferencias entre las distintas implementaciones trabajando en la tasa de transferencia propuesta en nuestras especificaciones. La operación de la división en la aproximación por polinomios es la basada en el cálculo del recíproco de la entrada *x* y la posterior multiplicación por la entrada *y*.

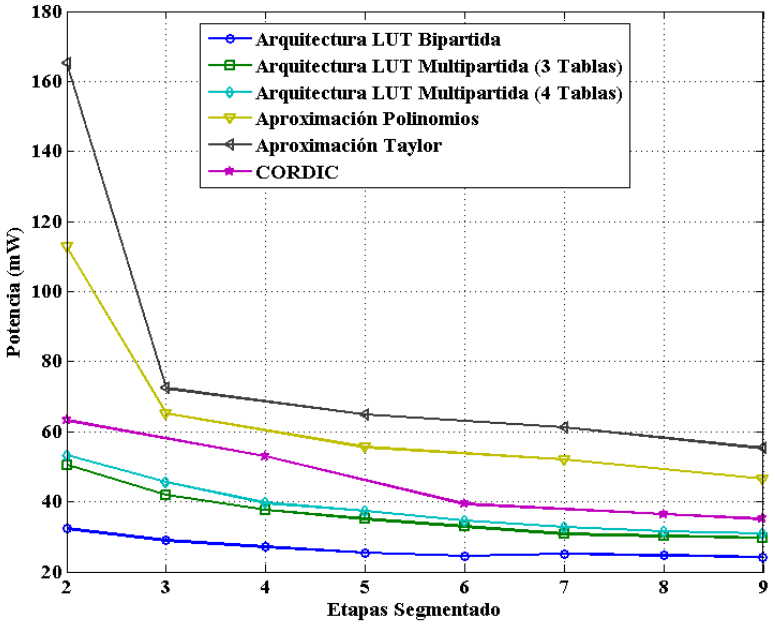


Figura 3.13 - Comparativa del consumo de potencia obtenido por las diferentes implementaciones y modificando el numero de etapas de segmentación.

Podemos apreciar que para todos los casos la arquitectura que menos consume es la basada en memorias bipartidas. Las implementaciones que más consumen son las basadas en la aproximación de Taylor y por polinomios debido al gran numero de recursos utilizados. Conforme vamos aumentando el número de etapas de segmentado, vemos como esa diferencia es menos acusada y en todos los casos siempre se reduce el consumo. Estos resultados se han tomado utilizando los elementos embebidos disponibles en las FPGAs Virtex-2. En la figura 3.14 se muestra una comparativa de los mismos métodos pero implementando todas las arquitecturas únicamente con *slices*. Para este caso la arquitectura que menos consume es la basada en CORDIC, seguida de las implementaciones multipartidas. Nuestra implementación es de las que más consume debido a que utilizamos un mayor número de *slices* para almacenar los valores de la tabla bipartida.

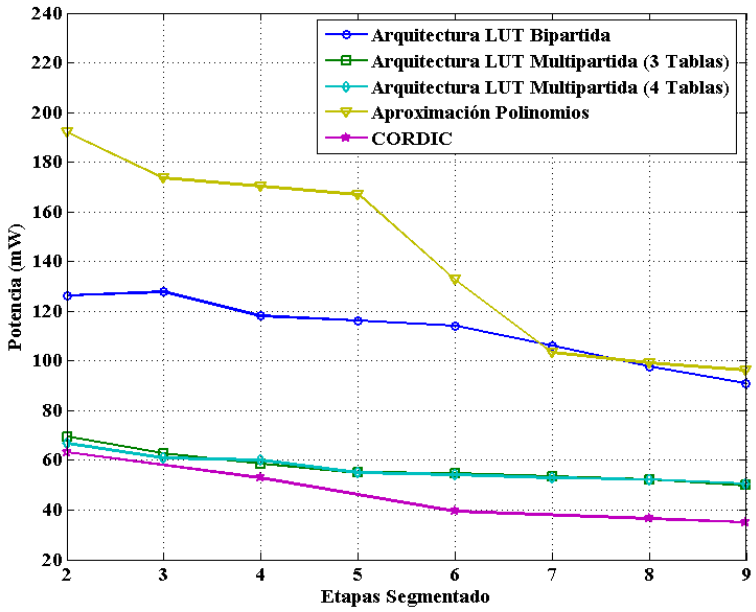


Figura 3.14 – Comparativa del consumo de potencia obtenido por las diferentes implementaciones y modificando el numero de etapas de segmentación y utilizando únicamente *slices*.

3.4. Conclusiones

En este capítulo hemos presentado una arquitectura para realizar el cálculo de la atan(y/x) basada en la utilización de LUTs que puede ser utilizada en aplicaciones de comunicaciones. Para el cálculo de la atan(y/x) hemos dividido la operación de dos partes: primero calculamos la función de una variable $z=y/x$ por medio de la aproximación del recíproco de la entrada x y el resultado lo multiplicamos por la entrada y; segundo realizamos el cálculo de la atan(z). Para la aproximación de la función recíproco $1/x$ y atan(z) se han utilizado métodos basados en tablas. Para reducir el tamaño de las tablas utilizadas se han utilizado métodos bipartidos. Se han descartado los métodos multipartidos debido al aumento de consumo de potencia con respecto a la implementación bipartida a pesar de la reducción en el tamaño de las memorias necesarias. Estos esquemas multipartidos pueden ser necesarios si queremos reducir el tamaño de la memoria necesaria sin importar el consumo de potencia.

La arquitectura ha sido implementada utilizando la herramienta Xilinx System Generator y para la síntesis y emplazado y rutado hemos utilizado la herramienta Xilinx ISE 10.1. El dispositivo utilizado para la implementación es la FPGA Virtex-2 V3000 de Xilinx. De los resultados expuestos vemos que es la implementación que menos consumo ha generado. Para ello hemos comparado diferentes implementaciones CORDIC, aproximación por polinomios, multipartidas, etc. y hemos medido su consumo. El principal uso de la arquitectura propuesta es su utilización en sistemas de comunicaciones los cuales trabajaran con tasas de datos de 20 Msps. La arquitectura presentada supera sin problemas esas tasas de datos, incluso aumentado la precisión de trabajo. La latencia mínima de la implementación es de tres ciclos de reloj ya que tenemos todas las entradas y salidas registradas. Una implementación completamente segmentada puede llegar a generar tasas de datos de hasta 145 Msps con una latencia de 14 ciclos de reloj. Además, tenemos que añadir que las Block-RAM utilizadas son síncronas añadiendo dos ciclos de latencia. Con respecto al consumo de potencia, nuestra implementación reduce un 50% el consumo con respecto a las implementaciones basada en multipartidas, por ello, se ha optado por utilizar el método basado en tablas bipartidas con respecto a los métodos multipartidos. Hay que tener en cuenta que un elemento muy importante para la reducción del consumo de potencia ha sido la utilización de los multiplicadores embebidos para la realización de los *barrel-shifter* en vez de su implementación mediante *slices*.

Capítulo 4.

Aproximación de la Atan(*y/x*) basada en la División Logarítmica

En el capítulo 4 presentamos una nueva arquitectura para el cálculo de la atan(y/x) que puede trabajar en sistemas de comunicaciones donde sean necesarias tasas de procesado de 20 Msps. Como hemos visto en capítulos anteriores el principal cuello de botella para la aproximación de la atan(y/x) es la realización de la operación de la división de las dos entradas. En la arquitectura que se presenta vamos a evitar la realización de la operación de división realizando una transformación logarítmica. De esta forma la operación de la división se convierte en una simple resta de las dos entradas. A partir de este resultado calcularemos la arcotangente del antilogaritmo para obtener el resultado deseado.

Para la transformación logarítmica vamos a proponer dos arquitecturas que permiten calcular eficientemente el logaritmo. La primera estará basada en la utilización de tablas multipartidas y en la segunda propondremos, a partir de la aproximación de Mitchell, una etapa de corrección que nos permitirá aumentar la precisión con un bajo coste hardware. Esta etapa de corrección estará basada en la utilización de interpolación por rectas con una corrección adicional basada en tablas.

Para la aproximación de la arcotangente vamos a proponer dos arquitecturas diferentes. La primera estará basada en el uso de técnicas de segmentación no-uniforme adaptadas para poder ser utilizadas en tablas multipartidas. La segunda arquitectura estará basada en el uso de interpolación por rectas con una tabla de corrección adicional. Las arquitecturas propuestas utilizan los hard-cores embebidos de las FPGAs, alcanzando altas velocidades y unos consumos reducidos de potencia, como ha quedado constatado en el capítulo anterior.

4.1. Introducción

En el capítulo 4 vamos a presentar una nueva arquitectura para la aproximación eficiente de la atan(y/x) para su utilización en FPGAs. Las arquitecturas presentadas en el capítulo 2 presentan el problema de la complejidad en la realización de la división de las dos entradas aumentando la latencia del operador y el área utilizada. La arquitectura propuesta en el capítulo 3 presenta la ventaja de su bajo consumo debido a la utilización de métodos basados en tablas y unos bajos requerimientos de área. Para este caso el cuello de botella lo seguimos teniendo en la realización de la división, sobre todo cuando aumentamos el tamaño de palabra de la aproximación. Básicamente, la división la realizamos por medio de un esquema de conversión coma-fija/coma-flotante, realizamos la división y volvemos a utilizar un esquema de conversión coma-flotante/coma-fija. Estas operaciones de conversión son realizadas por medio de un *barrel-shifter*, los cuales añaden un retardo relativamente alto para tamaños de palabra grandes.

Para evitar este inconveniente vamos a proponer una nueva arquitectura que elimine este problema. Para ello vamos a realizar la operación de división de una manera indirecta, mediante la utilización de funciones logarítmicas. En un sistema logarítmico la división entre dos variables puede ser realizada a partir de la resta de las mismas (4.1). Cuando utilizamos sistemas logarítmicos, las operaciones de multiplicación y división son substituidas por operaciones de suma y resta, respectivamente, simplificando enormemente la realización de las operaciones.

$$\log_2\left(y/x\right) = \log_2 y - \log_2 x \tag{4.1}$$

Para la aproximación de atan(y/x), primero realizaremos la conversión logarítmica de las dos entradas y después la resta entre ellas, a esa división logarítmica aplicaremos la aproximación de la arcotangente del antilogaritmo de la diferencia (4.2), de esta forma pasamos el resultado del sistema logarítmico al sistema decimal.

$$\operatorname{atan}\left(2^{(\log_2(y)-\log_2(x))}\right) \tag{4.2}$$

Para simplificar la notación durante el presente capítulo utilizaremos "log" en vez de "$\log_2$" para referirnos al logaritmo binario.

4.2. Arquitectura Propuesta

A continuación vamos a presentar una nueva arquitectura que nos va a permitir realizar una aproximación eficiente en FPGAs de la función atan(y/x). Esta arquitectura estará basada en la implementación de la operación de división mediante el uso de funciones logarítmicas. Mediante la utilización de logaritmos podemos realizar la operación de división de dos entradas por medio de la operación de la resta, de esta manera evitamos la realización de la operación de la división la cual es una operación bastante compleja. A partir del resultado de la división logarítmica, realizaremos la aproximación de la arcotangente del antilogaritmo. Como podemos apreciar la arquitectura necesita aproximar dos funciones elementales, la función log(x) y la atan(2^w). Para ello hemos desarrollado diferentes propuestas que comentamos a continuación.

Primero, para la realización de la aproximación del log(x) vamos a presentar dos propuestas de implementación. La primera estará basada en la utilización de tablas multipartidas para la aproximación del logaritmo. La segunda propuesta está basada en la aproximación de Mitchell [48] sobre la que hemos añadido una etapa de corrección del error para aumentar la precisión de la aproximación. Esta etapa de corrección consiste en dividir el error cometido por la aproximación de Mitchell en cuatro regiones y aproximarlo mediante rectas. Para el cálculo de esta etapa de corrección utilizaremos una versión truncada de la entrada x y este error aproximado será sumado a x. Adicionalmente, si con la aproximación obtenida no generamos valores con suficiente precisión para cumplir nuestros requerimientos podemos añadir una pequeña LUT con el error cometido en la aproximación por rectas y de esta manera aumentaremos la precisión de la aproximación. A la etapa de corrección por rectas la llamaremos "corrección gruesa" y a la etapa que utiliza LUTs la llamaremos "corrección fina". El logaritmo tiene la gran ventaja de que es una función relativamente lineal y fácilmente puede ser aproximada mediante aproximaciones por rectas. El tamaño y los límites de las rectas han sido elegidos mediante prueba y error, eligiendo los coeficientes que han minimizado el error absoluto de la aproximación.

Segundo, para la aproximación de la atan(2^w) también hemos optado por dos propuestas. Por un lado, hemos utilizado una aproximación mediante tablas multipartidas sobre la cual se ha implementado un esquema de segmentación no-uniforme permitiendo adaptar el tamaño de las tablas utilizadas al comportamiento de la función. Se ha optado por utilizar la segmentación no-uniforme debido al comportamiento tan dispar de la función en todo el rango de la aproximación. Normalmente en los métodos basados en interpoladores y aproximaciones con tablas, el intervalo donde la función va a ser aproximada es dividido en varias sub-regiones/segmentos más pequeños y los valores necesarios para obtener el valor de la función dentro de la sub-región son almacenados en una tabla. Este método tiene la gran ventaja de un direccionamiento muy simple de la tabla pero puede ser problemático e ineficiente en regiones de la función donde ocurran grandes no-linealidades. Estas no-linealidades aparecerán en funciones donde la primera o n-esima derivada tengan valores absolutos grandes. Es por ello que será interesante que el tamaño de los segmentos se adapte a las no-linealidades de la función ya que regiones con altas no-linealidades requerirán de tamaños de segmentos menores que las regiones más lineales. De esta manera reduciremos las cantidades de memoria necesarias para la aproximación de la función, permitiendo diseños más eficientes y compactos. Muchos autores han propuesto deferentes métodos para la segmentación no-uniforme de funciones elementales. Por ejemplo, [36] propone el uso de segmentos que incrementan su tamaño en potencias de dos para la aproximación de los logaritmos binarios. Lee y otros [37] proponen el uso de jerarquías que contengan segmentos uniformes y segmentos que varían el tamaño en potencias de dos. Sasao y otros [49] presentan un esquema de segmentación no-uniforme basado en el uso de LUT en cascada que son utilizadas para el cálculo de las direcciones de los coeficientes. La gran ventaja de este método es que está especialmente indicado para su implementación en FPGAs ya que el tamaño de las tablas es pequeño pudiendo ser implementadas eficientemente en las LUTs disponibles en las *slices* de las FPGAs. Estos esquemas de segmentación no-uniforme son utilizados en aproximaciones por polinomios e interpoladores lineales. Nosotros vamos a utilizar estos métodos adaptándolos para el uso de aproximaciones a funciones mediante tablas multipartidas. En la figura 4.1.a y 4.1.b podemos ver la aproximación de la atan(2^w) utilizando segmentación uniforme y no-uniforme, respectivamente. Podemos apreciar la diferencia de pendiente de la función en la parte central y el los extremos de la misma. Por otro lado hemos utilizado el mismo esquema de aproximación mediante rectas más LUT de error. Vemos como en este caso el número de regiones ha aumentado a cinco debido a las diferencias de pendiente de la función durante todo el rango de la misma.

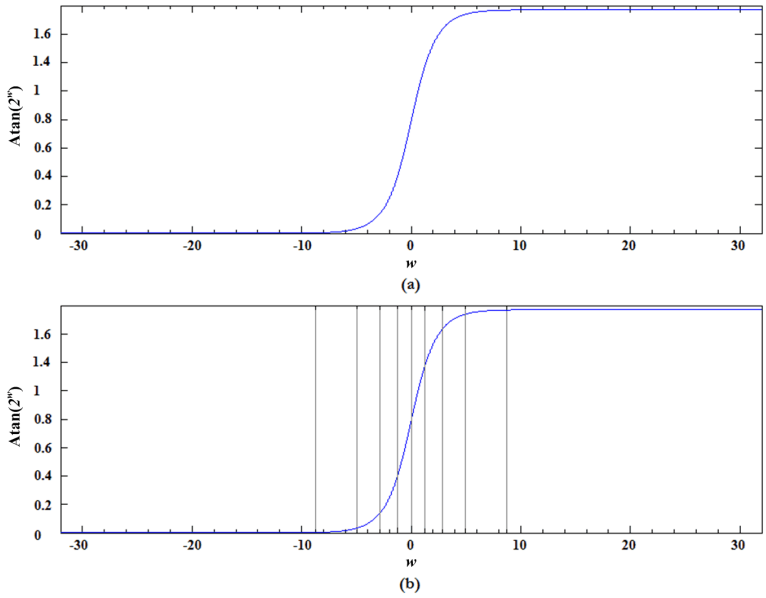


Figura 4.1.a - Segmentación uniforme de la aproximación de la atan(2^w).
Figura 4.1.b - Segmentación no-uniforme de la aproximación de la atan(2^w).

4.2.1. Aproximación del Logaritmo

El algoritmo de Mitchell está basado en la interpolación lineal en intervalos entre potencias de dos [48]. Si N es un numero binario $z_n z_{n-1} z_{n-2} \ldots z_0 . z_{-1} z_{-2} \ldots z_{-p}$, y siendo z_n, el "uno" en la posición más significativa de N. Entonces

$$N = 2^n + \sum_{i=-p}^{n-1} 2^i z_i . \tag{4.3}$$

Sacando factor común 2^n quedará:

$$N = 2^n (1 + \sum_{i=-p}^{n-1} 2^{i-n} z_i) = 2^n (1+x) \; ; \; 0 \le x < 1. \tag{4.4}$$

El logaritmo binario de N será:

$$\begin{aligned} \log N &= n + \log(1+x) \\ \log N &= b_n \ldots\ldots b_0 . b_{n-1} \ldots\ldots b_{-p} \end{aligned} \tag{4.5}$$

En (4.5), $b_n \ldots\ldots b_0$ es la representación binaria de n y $b_{n-1} \ldots\ldots b_{-p}$ es la representación binaria del $\log(1+x)$. El valor n en (4.5) normalmente es referido como la *característica* del logaritmo y contiene la información sobre el numero de bits entre el "uno" más significativo y el punto fraccionario de la representación binaria de N. Para el término $\log(1+x)$, Mitchell propone la utilización del único término lineal de la serie de Taylor del logaritmo ($\log(1+x) \approx x$). Este método tiene la gran ventaja de la velocidad y la facilidad de implementación en hardware pero tiene un error máximo de 0.08639 (3.53 bits) para un valor x=0.44269. El error cometido por la aproximación de Mitchell vendrá dado por (4.6) y está representado en la figura 4.2.

$$\varepsilon(x) = \log(1+x) - x, \qquad 0 \le x < 1 \tag{4.6}$$

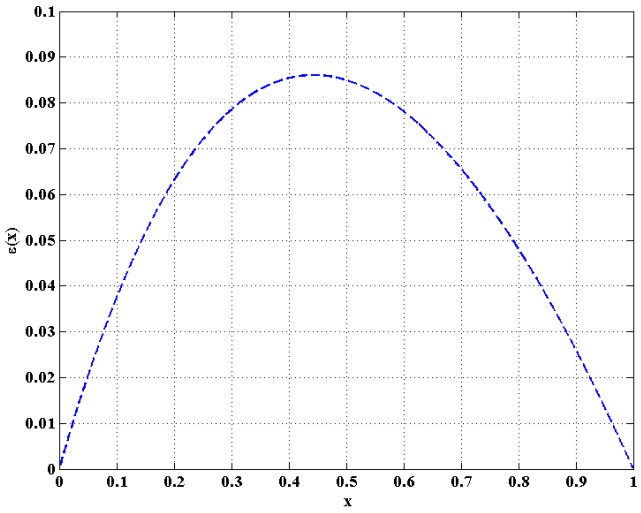


Figura 4.2 - Error cometido por la aproximación de Mitchell.

Para muchas aplicaciones este error es muy alto, imposibilitando la utilización de este método. A continuación vamos a ver los métodos propuestos para aumentar la precisión de la aproximación del logaritmo. Posteriormente presentaremos los métodos basados en rectas y tablas multipartidas propuestos.

4.2.1.1. Métodos para la Corrección de la Aproximación de Mitchell.

Para mejorar la precisión de la aproximación de Mitchell se puede añadir una etapa de corrección de error basada en la aproximación del error cometido por Mitchell mediante la utilización de rectas. Antes de presentar nuestra etapa de corrección vamos a ver en profundidad los métodos propuestos en la literatura para posteriormente presentar nuestra aproximación.

4.2.1.1.a. Métodos basados en interpolación lineal mediante desplazamientos y sumas.

Varios métodos para la aproximación del log(*1*+*x*) mediante la división en dos regiones han sido propuestos en [50][51][52]. La ecuación (4.7) muestra el valor de la etapa de corrección $\varepsilon(x) \approx \varepsilon_{2L}(x)$ utilizada por los distintos métodos.

$$\varepsilon_{2L}(x) = \begin{cases} S_0 \cdot x'_0 + b_0, & 0 \le x < 0.5 \\ S_1 \cdot x'_1 + b_1, & 0.5 \le x < 1 \end{cases} \tag{4.7}$$

La diferencia entre los tres métodos se debe a que utilizan diferentes pendientes y constantes en la aproximación lineal de cada región. En la tabla 4.1 podemos ver esos valores, donde $x_{n.0}$ indica los n+1bits más significativos de la entrada x y $\overline{x}_{n.0} = 1 - x_{n.0} - 2^{-n}$.

	[50]	[51]	[52]
S_0	2^{-2}	$(2^{-3}+2^{-4})$	2^{-2}
S_1	2^{-2}	$(2^{-3}+2^{-5})$	2^{-2}
x_0'	$x_{3:0}$	$x_{3:0}$	$x_{2:0}$
x_1'	$\overline{x}_{3:0}$	$\overline{x}_{3:0}$	$\overline{x}_{2:0}$
b_0	0	0	0
b_1	0	$(2^{-3}+2^{-5})$	0

Tabla 4.1 - Constantes utilizadas para la aproximación logaritmo por los métodos de dos regiones.

Como podemos observar, los métodos propuestos de dos regiones utilizan únicamente operaciones de desplazamiento y suma evitando la operación de multiplicación necesaria en la interpolación lineal. Además, al utilizar únicamente los bits más significativos en las operaciones se reduce el hardware necesario. En la figura 4.3 podemos ver las distintas etapas de corrección $\varepsilon_{2L}(x)$ comparadas con el error de Mitchell $\varepsilon(x)$ y su error cometido $e=\varepsilon(x)-\varepsilon_{2L}(x)$.

En [36] y [53] dividen el intervalo de $\log(1+x)$ en cuatro regiones diferentes. Las rectas utilizadas en la etapa de corrección $\varepsilon_{4L}(x)$ están definidas en (4.8) y los valores de las pendientes y constantes están definidas en la tabla 4.2. Los coeficientes utilizados en el método [36] fueron elegidos para minimizar el error cometido en la aproximación y en [53] para minimizar el error cuadrático medio. Las operaciones son realizadas utilizando todos los bits de la entrada x. En la figura 4.4 vemos el comportamiento de las etapas de corrección de cuatro regiones $\varepsilon_{4L}(x)$ y el error cometido $e=\varepsilon(x)-\varepsilon_{4L}(x)$.

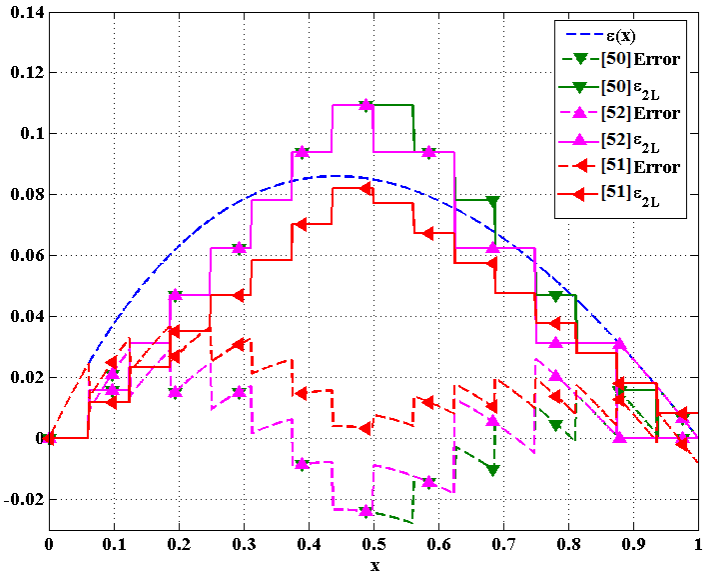


Figura 4.3 - Interpolación lineal utilizada en los métodos de dos regiones y su error cometido.

$$\varepsilon_{4L}(x)=\begin{cases} S_0\cdot x_0'+b_0, & 0\le x<0.5 \\ S_1\cdot x_1'+b_1, & 0.25\le x<0.5 \\ S_2\cdot x_2'+b_2, & 0.5\le x<0.75 \\ S_3\cdot x_3'+b_3, & 0.75\le x<1 \end{cases} \tag{4.8}$$

	[36]	**[53]**
S_0	(5/16)	(37/128)
S_1	(5/64)	(3/64)
S_2	(1/8)	(7/64)
S_3	(1/4)	(29/128)
x_0', x_1'	x , 1	x
x_2', x_3'	$\overline{x}$	$\overline{x}$
b_0 , b_1	0	0
b_2 , b_3	(3/128) , 0	(1/32) , 0

Tabla 4.2 – Constantes utilizadas para la aproximación logaritmo por los métodos de cuatro regiones.

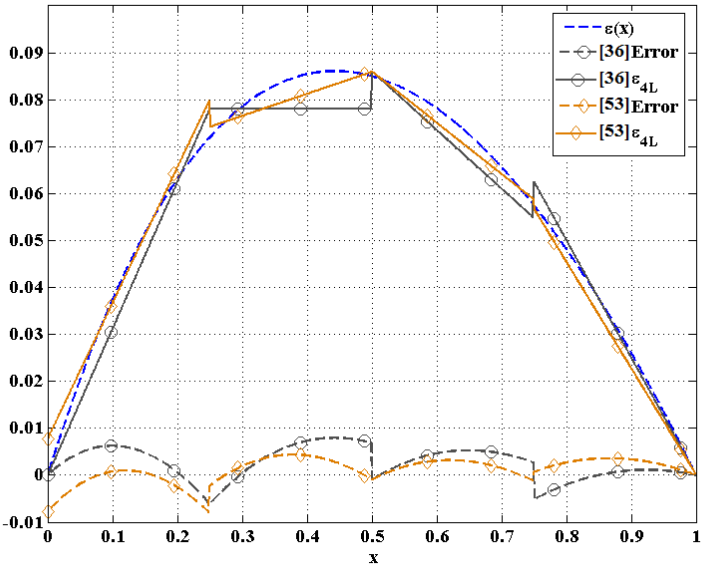


Figura 4.4 - Interpolación lineal utilizada en los métodos de cuatro regiones y su error cometido.

Abed y otros [52] proponen diferentes etapas de corrección variando el número de regiones utilizadas. Para dos regiones ya se ha comentado previamente. La aproximación para seis regiones $\varepsilon_{6L}(x)$ de $\varepsilon(x)$ la podemos ver en (4.9), para este caso únicamente se utilizan los 6 bits MSB.

$$\varepsilon(x) \approx \begin{cases} 2^{-2}\cdot x_{5:0}, & 0 \le x < 0.0625 \\ 2^{-2}\cdot x_{5:0} + 2^{-6}, & 0.0625 \le x < 0.25 \\ \left(2^{-4}+2^{-7}+2^{-8}\right), & 0.25 \le x < 0.375 \\ \left(2^{-4}+2^{-6}+2^{-7}\right), & 0.375 \le x < 0.625 \\ \left(2^{-4}+2^{-7}\right), & 0.625 \le x < 0.75 \\ 2^{-2}\cdot \overline{x}_{5:0}, & 0.75 \le x < 1 \end{cases} \tag{4.9}$$

En la figura 4.5 podemos ver el error cometido en la aproximación de Mitchell comparado con las rectas utilizadas $\varepsilon_{6L}(x)$ y el error cometido en la aproximación $e=\varepsilon(x)-\varepsilon_{6L}(x)$.

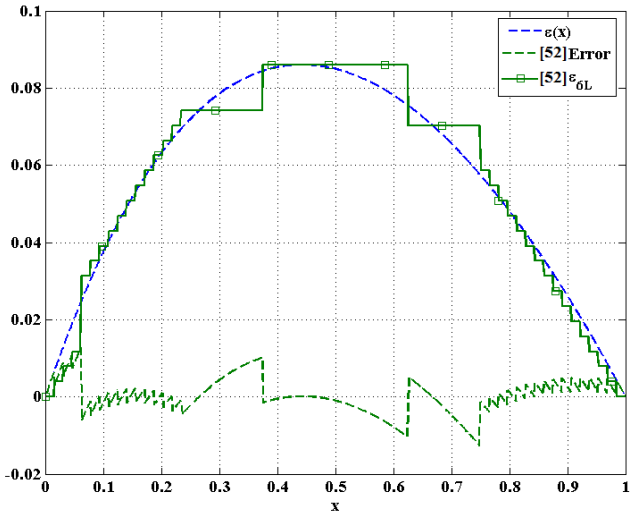


Figura 4.5 - Interpolación lineal utilizada en los métodos de seis regiones y su error cometido.

4.2.1.1.b. Métodos basados en LUTs.

El primer método basado en LUTs fue desarrollado por Brubaker y Becker [54], el cual está basado en la utilización de una LUT donde se almacena la aproximación del $\log(1+x)$. En [55] utilizan una LUT pero en este caso únicamente almacenan el error cometido por la aproximación de Mitchell y lo suman a la aproximación de Mitchell. Una implementación eficiente de este método ha sido propuesta por [56]. En este caso además de la etapa de corrección mediante el uso de LUTs añaden una etapa de interpolación lineal para aumentar la precisión en la aproximación. Esta etapa de interpolación se define como

$$\varepsilon(x) \approx a + \frac{(b-a)\cdot n_1}{2^{k-t}}, \quad 0 \le x < 1. \tag{4.10}$$

El error de Mitchell es muestreado utilizando 2^t puntos (este parámetro dependerá del tamaño de la LUT utilizada). Los 2^t puntos del error de Mitchell es almacenado en la LUT y direccionado por los t bits MSB de la entrada x. b será el valor adyacente de a, k es el número total de bits de x y n_1 es el valor decimal de los últimos k-t bits de x. Para evitar realizar la operación de multiplicación, la multiplicación en (4.10) es transformada en una suma trabajando con un sistema logarítmico obteniendo la expresión:

$$(b-a)\cdot n_1 = \text{antilog}(\log(b-a) + \log(n_1)). \tag{4.11}$$

Para implementar este método necesitamos tres LUTs donde almacenamos $\log(a+b)$, $\log(n_1)$ y la función antilogaritmo y dos sumadores consiguiendo reducir los requerimientos en cuanto al tamaño de las memorias utilizadas con respecto a métodos basados en tablas bipartidas (SBTM) [32] o los métodos propuestos por Brubaker [54], Kmetz [55] y Maenner [57].

4.2.1.2. Método Propuesto.

La arquitectura propuesta para la aproximación del logaritmo está basada en el método de Mitchell sobre el que se añaden dos etapas de corrección: primero realizamos una interpolación lineal del error cometido por Mitchell ($\varepsilon_{4L}(x)$) en cuatro regiones y una segunda etapa en la que aproximamos el error residual ($\varepsilon_R(x)$)

por medio de una tabla LUT para aumentar la precisión de la aproximación. Nuestra aproximación tendrá dos términos:

$$\varepsilon(x) \approx \varepsilon_{4L}(x) + \varepsilon_R(x), \quad 0 \le x < 1 \tag{4.12}$$

En nuestra propuesta el error cometido por Mitchell es dividido en cuatro regiones uniformes, las cuales son aproximadas por rectas. Las pendientes de estas rectas únicamente podrán ser potencias de dos y utilizaremos una versión truncada de la entrada x (8 MSB). La ecuación que modela esta interpolación lineal $\varepsilon_{4L}(x)$ la podemos ver en (4.13). El número de bits utilizados y el valor de los coeficientes han sido elegidos mediante prueba y error para minimizar el error cometido. En la figura 4.6 podemos ver las rectas propuestas comparadas con el error de Mitchell $\varepsilon(x)$ y el error cometido en nuestra aproximación. Añadiendo esta etapa de corrección incrementamos la precisión del método de Mitchell hasta los 7 bits, siendo el que mayor precisión obtiene dentro de los métodos basados en interpolación lineal [36] y [50]-[53].

$$\varepsilon_{4L}(x) = \begin{cases} 0.008 + x_{7:0} \cdot 2^{-2}, & 0 \le x < 0.25 \\ 0.07 + x_{7:0} \cdot 2^{-5}, & 0.25 \le x < 0.5 \\ 0.15 - x_{7:0} \cdot 2^{-3}, & 0.5 \le x < 0.75 \\ 0.25 - x_{7:0} \cdot 2^{-2}, & 0.75 \le x < 1 \end{cases} \tag{4.13}$$

Para aumentar la precisión se ha añadido una pequeña tabla Look-up, llamada "error-LUT", en la que almacenamos el error residual cometido por la aproximación de Mitchell más nuestra etapa de corrección basada en la interpolación lineal sobre cuatro regiones. La ventaja es que la magnitud del error residual almacenado en la tabla (ε_R) es menor que la magnitud del error de Mitchell permitiendo reducir el tamaño de las palabras almacenadas. Por ejemplo, utilizando una tabla de 128x5 bits conseguimos aumentar la precisión de la aproximación hasta los 10.4 bits. En la figura 4.7 podemos ver el error cometido por la aproximación de Mitchell añadiendo una etapa de corrección formada $\varepsilon(x) \approx \varepsilon_{4L}(x) + \varepsilon_R(x)$, donde $\varepsilon_R(x)$ es la LUT de 128x5 bits. Si queremos aumentar la precisión el tamaño de la LUT también aumenta, para estos casos podemos utilizar métodos multipartidos y reducir el tamaño de la memoria necesaria para almacenar la error-LUT. Además podemos aumentar el tamaño de la entrada truncada para aumentar la precisión de la aproximación. En nuestra arquitectura utilizamos estos métodos a partir de precisiones de 16 bits.

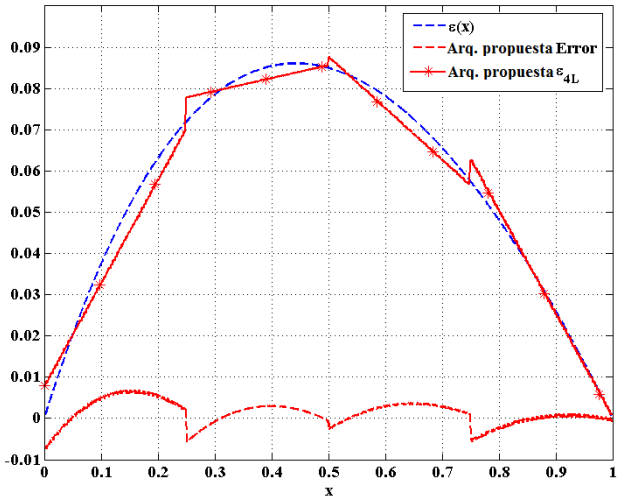


Figura 4.6 - Interpolación lineal utilizada en el método de cuatro regiones propuesto y su error cometido.

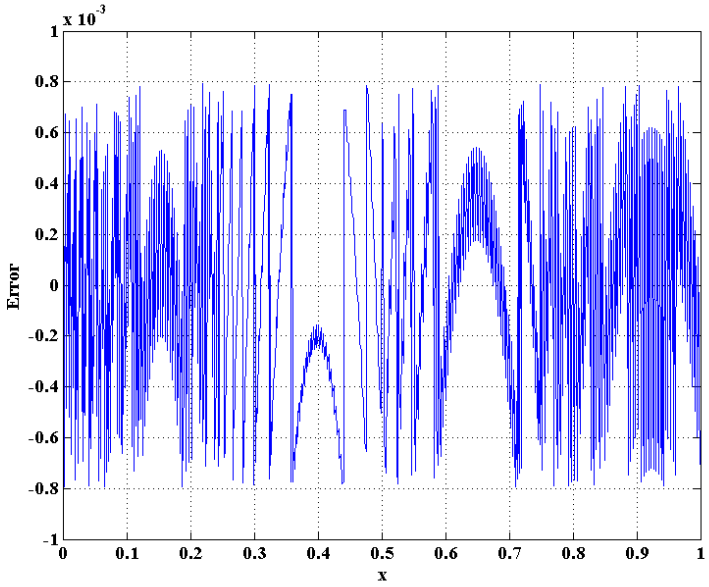


Figura 4.7 - Error cometido en la aproximación del logaritmo mediante interpolación rectas y error-LUT.

El circuito necesario para la realización de la aproximación del logaritmo se muestra en la figura 4.8. La primera etapa consistirá en la localización del "uno" más significativo. Esta operación se realiza mediante un *Leading to One Detector* (LOD) basado en la metodología propuesta por Oklobdzija [46]. La característica del logaritmo (parte entera) es obtenida a partir de la resta del número de bits de la parte entera del logaritmo con respeto al valor de la posición del "uno" más significativo. El valor del "uno" más significativo es además utilizado por un *barrel-shifter* para re-alinear el valor $(1+x)$. La salida del *barrel-shifter* contendrá el valor $log(1+x)-\varepsilon(x)$ de la aproximación de Mitchell. El MSB del *barrel-shifter* será ignorado ya que siempre será "uno" y los dos bits siguientes son utilizados para seleccionar una de las cuatro rectas de la aproximación. El multiplexor es utilizado para seleccionar uno de los cuatro desplazamientos cableados además de seleccionar el coeficiente de la LUT; los n siguientes bits son utilizados para direccionar la tabla con el error. El valor de n es obtenido por medio de simulaciones hasta obtener la precisión necesaria. Los coeficientes almacenados en la error-LUT han sido generados utilizando dos bits de guarda y las memorias utilizadas han sido memorias distribuidas disponibles en las FPGAs de Xilinx. La aproximación de $\log(1+x)$ será concatenada con el valor de la característica del logaritmo calculado previamente para formar la salida.

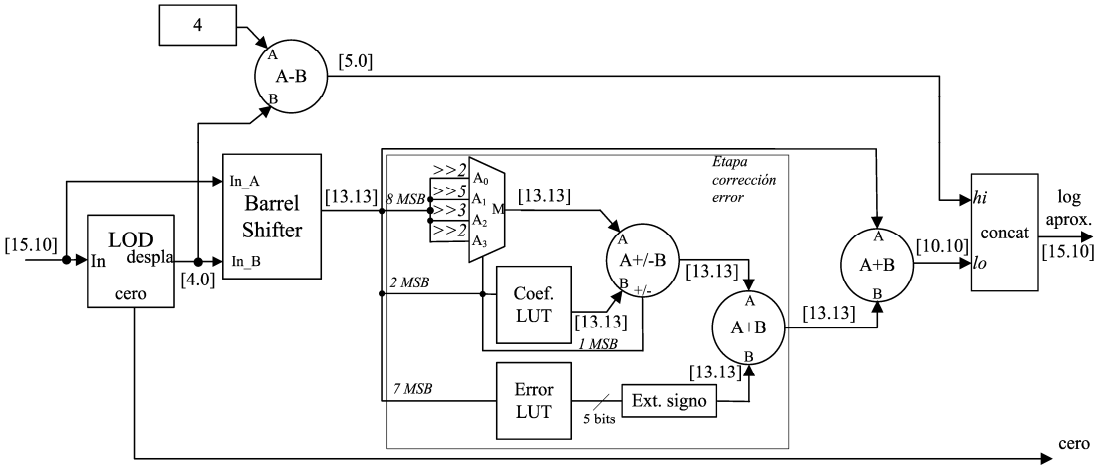


Figura 4.8 - Arquitectura propuesta para la aproximación del $\log(x)$.

4.2.1.3. Resultados de Implementación de la Aproximación del Logaritmo Propuesta y Comparación de Resultados con los Distintos Métodos.

La arquitectura propuesta ha sido modelada utilizando el lenguaje VHDL y ha sido implementada en una FPGA Virtex-II Pro XC2VP30-7 de Xilinx. Se ha optado por este dispositivo para poder comparar los resultados con otras implementaciones propuestas por distintos autores. El área y velocidad han sido obtenidas utilizando la herramienta Xilinx ISE 10.1. En la tabla 4.3 podemos ver los resultados de implementación obtenidos modificando la precisión de salida entre 10 y 20 bits. La parte entera del logaritmo siempre será de 5 bits y la parte fraccionaria variará de tamaño para poder alcanzar la precisión requerida. La tabla error-LUT será única para precisiones comprendidas entre 10 – 14 bits y utilizaremos los 8 MSB bits de la entrada. Para precisiones de 16 – 20 bits utilizaremos los 14 MSB bits de la entrada y la error-LUT será descompuesta en tres tablas mediante métodos multipartidos. Las dos tablas más grandes serán implementadas utilizando Block-RAM de doble puerto y la otra tabla más pequeña será implementada mediante memoria distribuida.

Precisión	**10**	**12**	**14**	**16**	**18**	**20**
Slices	90	120	162	180	248	298
LUT4	165	215	302	345	465	560
Tamaño LUT (bits)	640	896	2368	5632 (3LUTs)	12800 (3LUTs)	25600 (3LUTs)
Block RAM	0	0	0	1	1	1
Fmax (MHz)	65.8	65.1	63.8	60.8	57.5	54.9
Latencia	1					

Tabla 4.3 – Resultados de implementación de la aproximación del log(x) para diferentes precisiones de salida.

Para comparar nuestro método con los métodos de bajos requerimientos hardware [36] y [50 – 53] presentados anteriormente, hemos realizado una implementación de todos los métodos en VHDL. En la figura 4.9 podemos ver la implementación del log($1+x$) realizada. Para poder comparar nuestra aproximación con las distintas arquitecturas hemos modificado el tamaño de entrada, utilizando un tamaño de 26 bits fraccionarios y 5 enteros. Este formato es el utilizado en los métodos basados en rectas y que hemos utilizado para la comparación. En la tabla 4.4 podemos ver el área utilizada por los distintos métodos y la precisión obtenida. De los resultados obtenidos podemos ver como obtenemos la mayor precisión con una utilización de recursos bastante baja (únicamente 36 LUTs). Comparando con los otros métodos de interpolación de cuatro regiones la reducción de área es considerable.

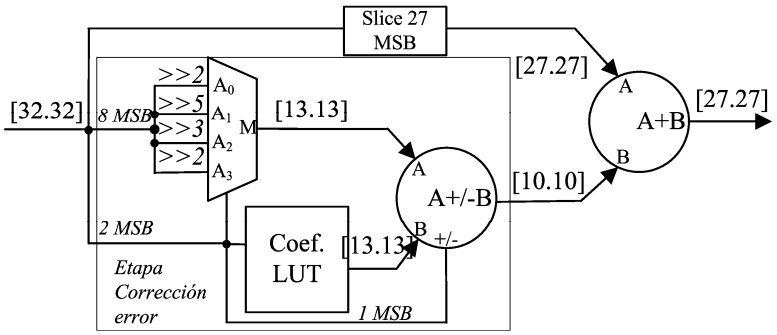


Figura 4.9 - Arquitectura propuesta para la aproximación del log($1+x$) mediante rectas para una entrada de 32 bits.

	[50]	[51]	[52]	[36]	[53]	[52]	Nuestra Propuesta
Nº regiones	2	2	2	4	4	6	4
Error (bits)	4.1	4.5	4	6.3	6.1	6.2	7
Slices	5	15	5	57	65	13	18
LUT4	9	27	8	109	128	24	36

Tabla 4.4 – Comparación de resultados de implementación para las distintas aproximaciones del log(*1*+*x*) basadas en rectas.

Por último, para mostrar las bondades de nuestra propuesta basado en interpolación lineal combinado con una LUT donde almacenamos el error residual de la aproximación, hemos realizado una implementación con los mismos tamaños de palabra y precisiones de salida que la propuesta por [56]. Para esta implementación se ha trabajado en coma flotante simple. Al tener la mantisa normalizada no se ha utilizado ni el *barrel-shifter*, ni el LOD. Hemos comparado con la arquitectura basada en la aproximación por LUT ya que es la que menor número de recursos de almacenamiento utiliza, como se indica en [56], comparándose con otros métodos basados en LUTs. Los tamaños de memoria utilizados por ambos métodos se muestran en la tabla 4.5.

		Nuestra Propuesta	[56]
Memoria Necesaria		6272	6656
Tamaños Tablas	A_0	256x13	128x18
	A_1	256x9	128x18
	A_2	128x5	128x16

Tabla 4.5 - Tamaños de las memorias utilizadas por la aproximación log(*1*+*x*) para implementación en coma flotante simple.

En la figura 4.10 podemos ver la implementación realizada para comparar con [56]. Para ello hemos modificado nuestra implementación para trabajar en coma flotante simple (mantisa de 23 bits). Las líneas punteadas verticales indican que se han inferido registros para segmentar el diseño. En la tabla 4.6 podemos ver los resultados de implementación de los dos métodos, obteniendo una precisión de salida superior utilizando para ello menos memoria (-5.7%) y LUTs (-4.7%). Ambas implementaciones se han segmentado completamente obteniendo frecuencias de trabajo relativamente parecidas (aproximadamente 350 MHz) pero nuestra implementación tiene una menor latencia. Las líneas verticales punteadas indican una etapa de segmentación. El camino crítico en la implementación lo encontramos en el retardo de la Block-RAM, de ahí que no se haya segmentado más el operador.

	Nuestra Propuesta	[56]
Slices	165	287
LUT4	200	210
F.F.	265	415
Fmax (MHz)	358	350
Latencia	8	12
Error (Bits)	16.8	16.7

Tabla 4.6 - Comparación de los resultados obtenidos por la arquitectura propuesta y la arquitectura propuesta en [56].

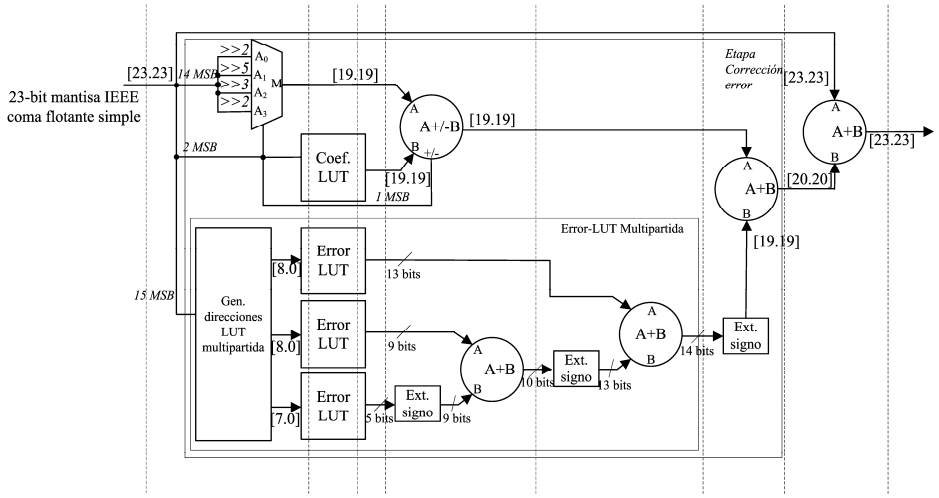


Figura 4.10 - Arquitectura realizada para comparar resultados de implementación con [56].

4.2.1.4. Aproximación del log(*1+x*) mediante LUTs

Como hemos visto anteriormente, para poder aumentar la precisión de la aproximación del logaritmo mediante el método de Mitchell deberemos aumentar la precisión de la aproximación del log(*1+x*). Para aumentar la precisión de la conversión vamos a aproximar el log(*1+x*) mediante una tabla multipartida. En este método, la tabla con los valores utilizados para la aproximación será dividida en tres o más tablas pequeñas y mediante una suma generaremos el valor de la aproximación (4.14):

$$f(x) = f(x_0 + x_1 + x_2 + x_3) \approx a_0(x_0, x_1) + a_1(x_0, x_2) + a_2(x_0, x_3) \tag{4.14}$$

La entrada de n-bits será dividida en cuatro palabras x_0, x_1, x_2 y x_3 de n_0, n_1, n_2 y n_3 bits, respectivamente, siendo $x = x_0 + x_1 \cdot 2^{-n_0} + x_2 \cdot 2^{-(n_0+n_1)} + x_3 \cdot 2^{-(n_0+n_1+n_3)}$ y asumiendo $0 \le x \le 1$. Los coeficientes que componen las distintas tablas han sido calculados mediante el método propuesto en [34], obteniendo un error en la aproximación de $\varepsilon \approx 2^{-n}$. El particionado óptimo de la entrada x ha sido estudiado para minimizar el tamaño de las tablas utilizadas. Los resultados muestran que para una entrada de 10 bits el particionado óptimo es con $n_0 = 3$, $n_1 = 2$, $n_2 = 2$ y $n_3 = 2$, necesitando unos 544 bits. El bit más significativo de la entrada siempre será "uno" y no es utilizado para direccionar las distintas tablas. Estas memorias pueden ser implementadas utilizando memoria distribuida. La tabla 4.7 muestra los tamaños de las distintas tablas utilizadas en la aproximación para precisiones de salida desde 10 a 20 bits.

	log(1+x)					
Precisión	10	12	14	16	18	20
Particionado óptimo	n_0=3, n_1=2, n_2=2, n_3=2	n_0=3, n_1=3, n_2=3, n_3=2	n_0=3, n_1=4, n_2=3, n_3=3	n_0=4, n_1=4, n_2=4, n_3=3	n_0=5, n_1=4, n_2=4, n_3=4	n_0=4, n_1=5, n_2=5, n_3=5
ROMs	A_0 32 x 12 bits	A_0 64 x 12 bits	A_0 128 x 12 bits	A_0 256 x 12 bits	A_0 512 x 16 bits	A_0 512 x 18 bits
	A_1 16 x 6 bits	A_1 64 x 6 bits	A_1 64 x 6 bits	A_1 256 x 6 bits	A_1 512 x 8 bits	A_1 512 x 10 bits
	A_2 16 x 4 bits	A_2 32 x 4 bits	A_2 64 x 3 bits	A_2 128 x 3 bits	A_2 512 x 5 bits	A_2 512 x 7 bits

Tabla 4.7 – Tamaños de las memorias utilizadas para la aproximación del log(*1+x*) mediante LUTs multipartidas para diferentes precisiones.

La implementación hardware de la aproximación log(*1+x*) es muy simple, ya que únicamente es necesario un detector de "unos" (LOD), una pequeña ROM, un *barrel-shifter* el cual se implementará mediante un multiplicador embebido de la FPGA (como se ha indicado en el capitulo tres), tres tablas y dos sumadores. El primer paso para el cálculo del logaritmo será calcular la posición del "uno" más significativo de la entrada, esta operación será realizada por el LOD. Restando la posición del "uno" más significativo al bit más significativo de la parte entera, obtenemos la parte entera del logaritmo. En los logaritmos binarios, la parte fraccionaria del logaritmo será el resto de los bits de la derecha del "uno" más significativo. La posición del "uno" más significativo será usada por el *barrel-shifter* para normalizar la entrada *(1+x)*. La entrada normalizada será usada para direccionar la tabla multipartidaria utilizada en la aproximación. El bit más significativo de la aproximación multipartida será ignorado y el resto de los bits serán concatenados a la derecha de la característica calculada previamente para formar la aproximación del logaritmo. En la figura 4.11 podemos ver dicho esquema.

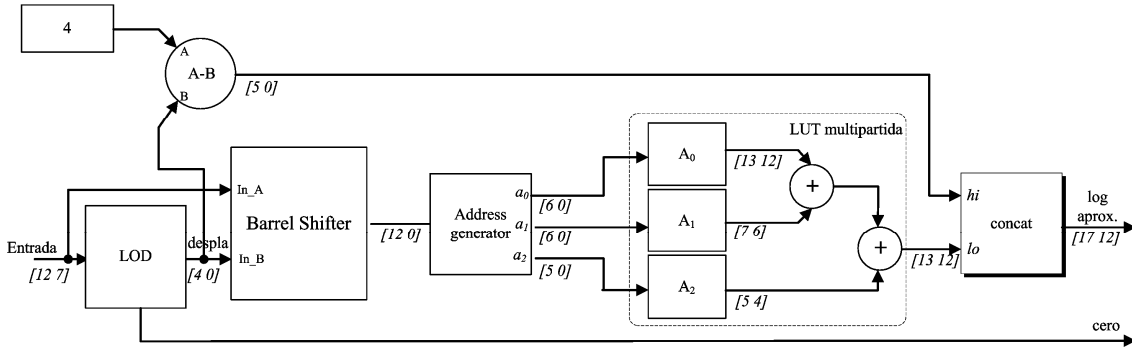


Figura 4.11 - Arquitectura propuesta para la aproximación del log(*x*) mediante LUTs multipartidas.

La arquitectura propuesta ha sido modelada utilizando el lenguaje VHDL y ha sido implementada en una FPGA Virtex-II Pro XC2VP30-7 de Xilinx. En la tabla 4.8 podemos ver los resultados de implementación obtenidos modificando la precisión de salida entre 10 y 20 bits. Podemos apreciar los bajos recursos incluso para altas precisiones. Las entradas y salidas están registradas y no se ha añadido ninguna etapa de segmentado intermedia.

Precisión	**10**	**12**	**14**	**16**	**18**	**20**
MULT18X18	1	1	1	1	1	1
Slices	75	85	98	110	128	172
LUT4	130	145	165	184	213	320
Tamaño LUT (bits)	544	1280	2112	4992	14848	19920
Block RAM	0	0	0	1	1	1
Fmax (MHz)	70,5	69,2	67,7	65,8	62,7	60,9

Tabla 4.8 – Resultados de implementación de la aproximación del log(*x*) basada en LUTs multipartidas para diferentes precisiones de salida.

4.3. Aproximación de la Atan(*y*/*x*)

Una vez visto los métodos propuestos para la aproximación del logaritmo vamos a presentar en esta sección la arquitectura propuesta para la aproximación de la atan(*y*/*x*) utilizando aproximaciones logarítmicas. Para realizar el cálculo de la atan(*y*/*x*) vamos a dividir la operación en dos etapas: primero realizaremos la división logarítmica de las dos entradas y segundo calcularemos la arcotangente del antilogaritmo. La figura 4.12 muestra esta arquitectura.

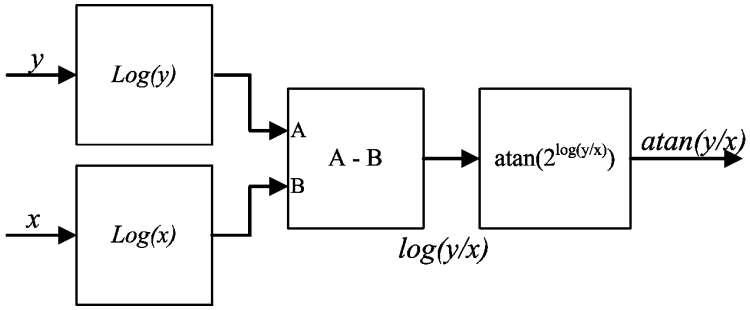


Figura 4.12 - Arquitectura propuesta para la aproximación atan(*y*/*x*) mediante división logarítmica.

La gran ventaja de nuestra arquitectura es que evitamos la necesidad de realizar operaciones de multiplicación o división permitiendo segmentar fácilmente el operador para aumentar el *throughput* del diseño. Además, nos permite poder multiplexar el operador del logaritmo para reducir el área del circuito aumentando en un ciclo de reloj la latencia del circuito y reduciendo el *throughput* a la mitad.

Para la aproximación de la arcotangente hemos utilizado varias propiedades. Primero, únicamente es necesario calcular los valores positivos de las entradas ya que los valores negativos pueden ser obtenidos a partir de (4.15). Además, el logaritmo de un valor negativo no existe.

$$atan\left(-\frac{y}{x}\right) = -\,atan\left(\frac{y}{x}\right) \tag{4.15}$$

Por otro lado, sólo los valores positivos de la división logarítmica son utilizados en la aproximación de la arcotangente. Los valores negativos son generados restando a $(\pi/2)$ el valor de la división (4.16). Si trabajamos con la fase normalizada este valor será igual a uno. Además, debemos tener en cuenta que no se pueden calcular los logaritmos de los números negativos.

$$\begin{aligned} &atan\left(2^{\log\left(y/x\right)}\right) \;\; ; \;\; 0 \le \log\left(y/x\right) < \infty \\ &\left(\frac{\pi}{2}\right) - atan\left(2^{abs\left(\log\left(y/x\right)\right)}\right) \;\; ; \;\; -\infty < \log\left(y/x\right) < 0 \end{aligned} \tag{4.16}$$

Para la aplicación de estas propiedades necesitamos una etapa de pre-procesado que nos permita trabajar únicamente con los valores positivos de las entradas y una etapa de post-procesado que nos permita corregir las transformaciones aplicadas a las entradas mediante (4.15) y (4.16). El esquema propuesto se puede ver en la figura 4.13.

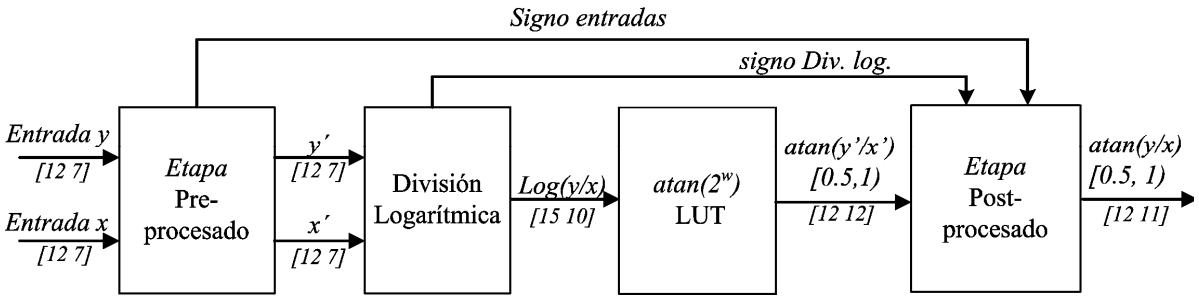


Figura 4.13 – Diagrama de bloques de la implementación propuesta para la aproximación de la atan(y/x).

4.3.1. Etapas de Pre-procesado y Post-procesado

La etapa de pre-procesado consta de dos circuitos que realizan el complemento a dos de cada una de las dos entradas, pasando los datos de formato en complemento a dos a signo y magnitud. La señal con la información sobre el signo se envía a la etapa de post-procesado. La etapa de post-procesado calcula las ecuaciones (4.15) y (4.16) de acuerdo con la señal signo enviada por la etapa de pre-procesado y por el divisor logarítmico, respectivamente. Un circuito que realice el complemento a dos (CMP2), un restador y dos multiplexores son los bloques necesarios para la realización de esta tarea. Las etapas de pre-procesado y post-procesado se pueden ver en las figuras 4.14.a y 4.14.b, respectivamente.

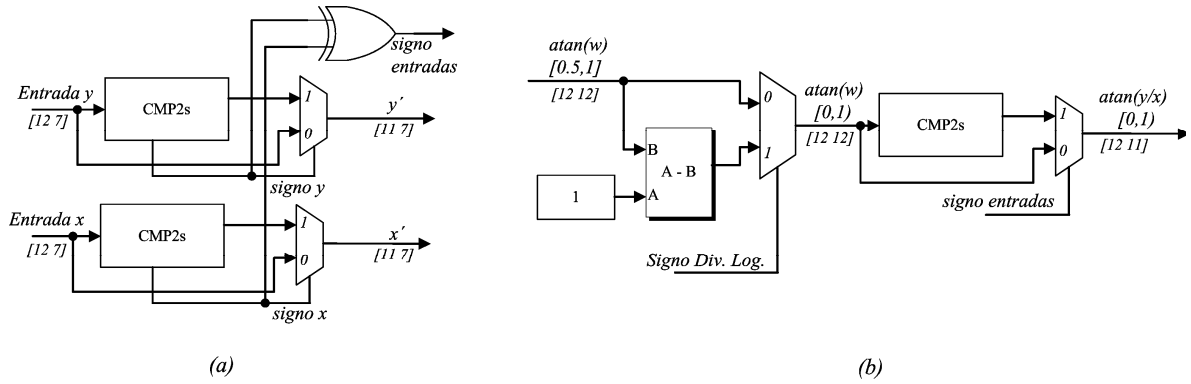


Figura 4.14.a - Etapa de pre-procesado. **Figura 4.14.b** - Etapa de Post-procesado.

4.3.2. Divisor Logarítmico

Una vez realizada la aproximación del $\log(1+x)$ de las dos entradas, realizamos su resta y el resultado lo utilizamos en la siguiente etapa para direccionar la tabla del cálculo del arcotangente del antilogaritmo. La gran ventaja de esta implementación es la posibilidad de reducir el área del sistema, multiplexando este operador entre las dos entradas utilizadas. La salida `cero` del bloque de conversión logarítmica la utilizaremos para asegurarnos que ninguna de las dos entradas es válida ($x, y = 0$). El esquema del divisor logarítmico y del divisor logarítmico multiplexado puede verse en la figura 4.15.a y 4.15.b respectivamente. Para multiplexar la operación de logaritmo de las dos entradas se ha añadido un multiplexor a la entrada el cual selecciona en cada instante una de las dos entradas por medio de una pequeña lógica de control. Por último se añaden unos registros a la salida para mantener el valor del cálculo de una entrada y poder realizar la división con la otra entrada.

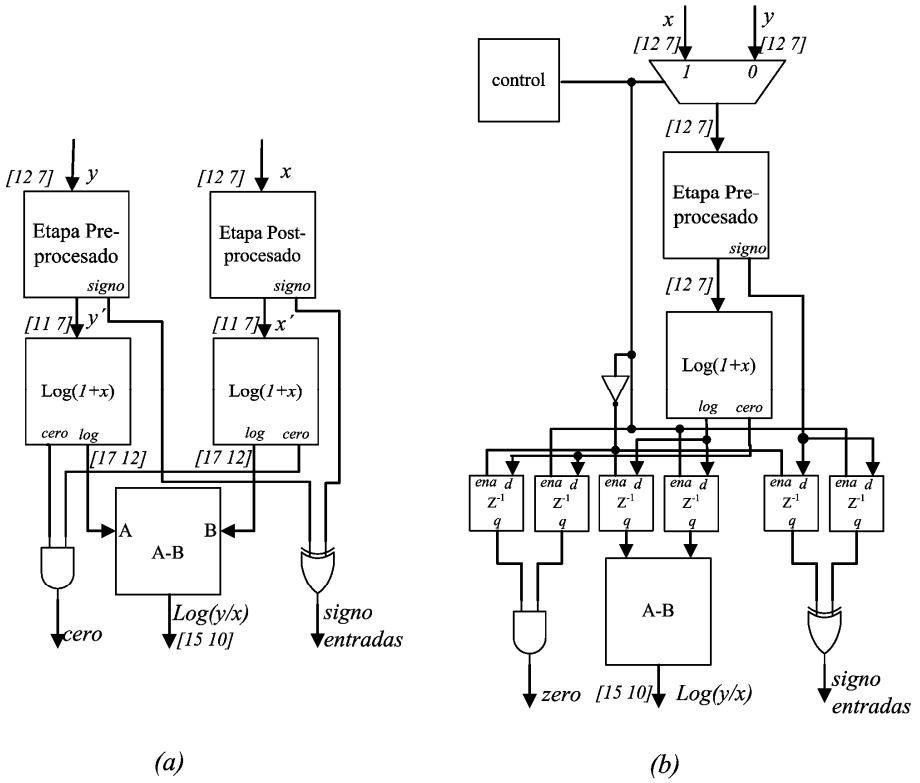


Figura 4.15.a - Divisor Logarítmico. **Figura 4.15.b** - Divisor Logarítmico multiplexado.

4.3.3. Aproximación de la Atan(2^w)

Como se ha comentado anteriormente hemos definidos dos métodos para la aproximación de la atan(2^w). Debido a nuestras especificaciones, el rango para el cual la aproximación debe ser válida es muy grande (-16, 16) implicando unos requerimientos de memoria grandes si queremos aproximar la función utilizando LUTs. Este rango vendrá definido por los bits de la parte entera del divisor logarítmico. Gracias a la simetría de la función, únicamente se ha almacenado la parte positiva ya que la parte negativa puede ser generada a partir de (4.16). El rango de la salida será $0 \leq atan(2^w) < 1$, ya que trabajamos con la fase normalizada.

4.3.3.1. Aproximación basada en LUT con Segmentación No-uniforme

Los métodos basados en LUTs son buenos para la aproximación de una función elemental con altas precisiones y una buena velocidad de cálculo. La precisión de la aproximación generada únicamente dependerá del tamaño de la LUT. Altas precisiones requerirán grandes tablas y viceversa. Para este caso, también hemos aplicado el método multipartido para minimizar el tamaño de la tabla. Además, hemos utilizado un esquema de segmentación no-uniforme [37] adaptado para poder ser utilizado con LUTs. De esta manera conseguimos reducir el tamaño de las LUTs necesarias. Los métodos multipartidos están basados en la utilización de segmentación uniforme y el número de segmentos utilizados son potencias de dos. La segmentación uniforme tiene la gran ventaja del fácil cálculo de las direcciones de memoria de los coeficientes. El gran inconveniente de este esquema de segmentación es que puede ser ineficiente para regiones de funciones con altas no-linealidades. Como hemos podido observar de la figura 4.1 el comportamiento de la atan(2^w) es muy diferente en la parte central que en los extremos, necesitando menos muestras en los extremos (menos pendiente) que en la parte central.

Mediante la utilización de las técnicas de segmentación no-uniforme podemos adaptar el tamaño de las tablas a las no-linealidades de las funciones, obteniendo una reducción significativa de la cantidad de memoria necesaria con respecto a una segmentación uniforme. La precisión obtenida en cada segmento será controlada variando el tamaño de la tabla que a su vez vendrá dado por el tamaño del segmento. El emplazamiento óptimo de los segmentos ha sido calculado utilizando un esquema de error balanceado [37]. El método del error balanceado es una forma de segmentación no-uniforme en el cual el error máximo de la aproximación es igual en todos los segmentos (4.17). En la figura 4.16.a podemos ver los segmentos óptimos obtenidos para la aproximación atan(2^w) sobre el rango [0, 16) con un error máximo de 2^{-11} utilizado en este diseño y en la figura 4.16.b podemos ver el error cometido en dicha aproximación. Podemos observar que para la aproximación de la arcotangente necesitamos más segmentos en la mitad inferior y únicamente un segmento en la mitad superior. Para cada segmento y utilizando la información referente a la pendiente de la función podemos realizar una mejor estimación del error cometido y optimizar el tamaño de las LUT usadas, permitiendo un cálculo de la función más preciso. Como podemos ver en la figura 4.16.a, únicamente son necesarios seis segmentos para aproximar la función con una precisión de 11 bits para todo el domino de la aproximación.

$$\varepsilon_{seg} = \max_{s_{2,n} \le x \le s_{2,n+1}} \left| f(x) - p_d(x) \right| \tag{4.17}$$

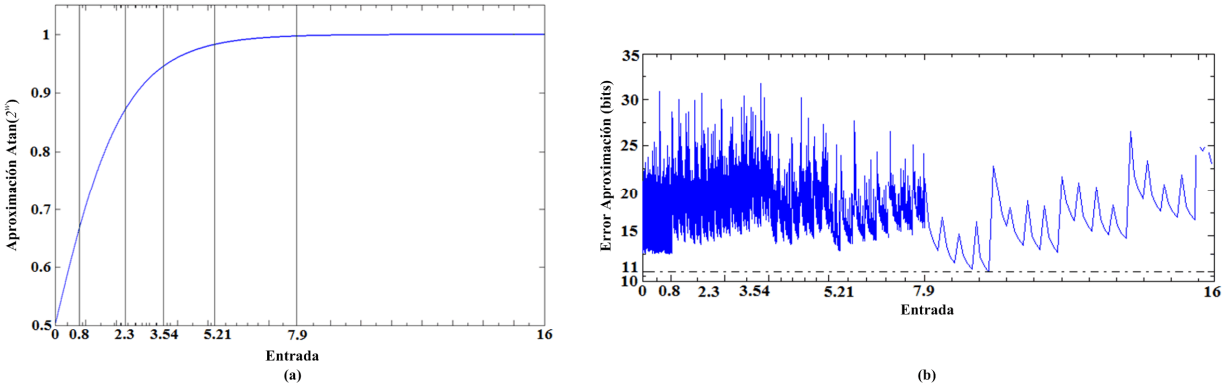


Figura 4.16.a - Emplazamiento óptimo de los segmentos para la aproximación de atan(2^w).
Figura 4.16.b - Error cometido en la aproximación utilizando segmentación no-uniforme expresado en bits.

Este método de segmentación no-uniforme ha sido implementado en Matlab. Se ha realizado una función que realiza la segmentación de la arcotangente a partir de varios parámetros: rango de entrada, precisión de salida, error máximo y número de tablas utilizadas [2 – 5]. Utilizando la información referente a la pendiente de la función en cada segmento realizamos el particionado óptimo para el número de tablas definido por n_0, n_1, n_2, n_3 y n_4 en cada segmento, quedándonos con el que nos permite un menor tamaño de las tablas. En nuestra implementación hemos dividido la salida del divisor logarítmico en cinco palabras para generar la tabla multipartida de cuatro LUTs (4.18), x_0, x_1, x_2, x_3 y x_4 con un tamaño de n_0, n_1, n_2, n_3 y n_4 bits, respectivamente. A estas cuatro tablas las llamaremos $LUT\ A_m$.

$$f(x) = f(x_0 + x_1 + x_2 + x_3 + x_4) \approx A_0(x_0, x_1) + A_1(x_0, x_2) + A_2(x_0, x_3) + A_3(x_0, x_4) \tag{4.18}$$

Lo primero que tendremos que realizar es el segmentado óptimo de la función atan(2^w) que cumpla la condición de error máximo pedido. Una vez tenemos los límites óptimos de los segmentos procederemos al cálculo del particionado óptimo de las tablas multipartidas utilizadas en cada uno de los seis segmentos. Para ello aplicamos el algoritmo STAM [34] de cuatro entradas. Los valores de las diferentes tablas han

sido calculados utilizando el esquema de redondeo (4.19), siendo *t* el número de bits utilizados para almacenar las muestras. Para cada segmento calcularemos la aproximación multipartida y obtendremos las LUT segmento A_{mn}, siendo *m* el número de tabla (0 – 3) y *n* el número de segmento (1 – 6) al que corresponde. Una vez tenemos calculados los valores de las tablas multipartidas necesarias para cada segmento, concatenaremos las distintas tablas LUT segmento correspondiente al mismo valor de *m* para obtener la tabla LUT A_m. Al concatenar las distintas tablas de cada segmento tendremos el problema de los distintos tamaños de palabra necesarios para la aproximación, para evitarlo, extenderemos el signo de las tablas de menor tamaño al tamaño de la tabla con mayor tamaño de palabra. Este esquema se puede apreciar en la figura 4.17

$$round(x,n) = \left\lfloor x \cdot 2^{t} + 0.5 \right\rfloor \cdot 2^{-t} \tag{4.19}$$

Una vez presentado como hemos realizado la segmentación no-uniforme de la aproximación por tablas multipartidas vamos a calcular las tablas necesarias para nuestra aproximación. En la tabla 4.9 tenemos los distintos tamaños utilizados por las cuatro tablas multipartidas utilizadas en cada segmento. El tamaño total de memoria necesario para la aproximación es de 4096 bits. Esta misma aproximación con los mismos requerimientos y utilizando segmentación uniforme obtendríamos un tamaño de memoria de 6272 bits. Vemos como hemos reducido la memoria necesaria un 34,7% con respecto a la segmentación uniforme.

atan(2^w)							
Segmento		1	2	3	4	5	6
Particionado Óptimo (bits)		3,3,2,2,2	3,3,2,2,2	3,3,2,2,2	2,2,2,2,2	2,2,2,1,1	2,2,1,1,1
ROMs	A_0	64*14	64*14	32*14	16*14	16*14	16*14
	A_1	32*10	32*10	32*10	16*10	16*10	8*10
	A_2	16*6	16*6	16*6	8*6	4*6	4*6
	A_3	16*4	16*4	16*4	8*4	4*4	4*4

Tabla 4.9 – Tamaños de las tablas utilizadas por los distintos segmentos de la aproximación atan(2^w).

Las tablas A_0 y A_1 han sido implementadas mediante Block-RAM de doble puerto. Esta memoria estará configurada como una memoria ROM de 1kx16 bits. Cada una de las dos tablas será implementada como una página de memoria independiente y cada uno de los dos puertos direccionará una página. Las tablas A_2 y A_3 serán implementadas mediante memoria distribuida debido a su pequeño tamaño.

El esquema de segmentación no-uniforme requiere un bloque hardware adicional que nos permitirá seleccionar los diferentes segmentos y a partir del valor del segmento poder generar el valor de la dirección de memoria correcto. En la figura 4.18 podemos ver la implementación en hardware de la aproximación no-uniforme. Vemos que por un lado tendremos que calcular a qué segmento tendremos que acceder. El hardware necesario para realizar este direccionamiento está basado en un esquema presentado por Dong-U Lee [37]. A partir del segmento en el que estamos y por medio de una ROM, traduciremos el numero de segmento a la posición de memoria corresponde ese segmento. Para este caso la ROM será de un tamaño muy pequeño ya que únicamente tiene que direccionar 6 segmentos. Por otro lado, una vez tenemos la posición de memoria del segmento (dirección base) y a partir del segmentado óptimo y por medio de multiplexores obtendremos la posición de memoria dentro del segmento (offset).

Estos dos valores serán concatenados para obtener la posición de memoria que accederá a cada una de las cuatro tablas.

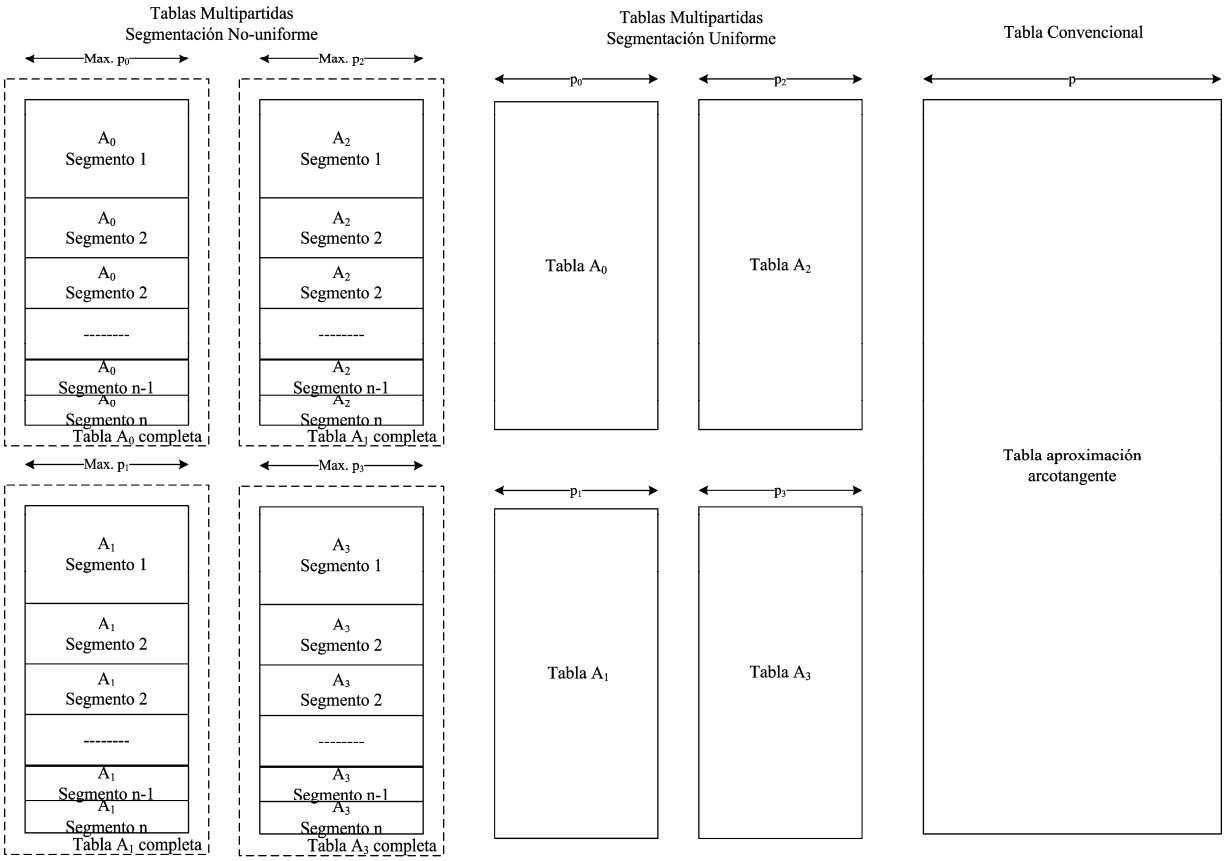


Figura 4.17 - Esquema de memoria utilizado para la aproximación atan(2^w) mediante LUTs.

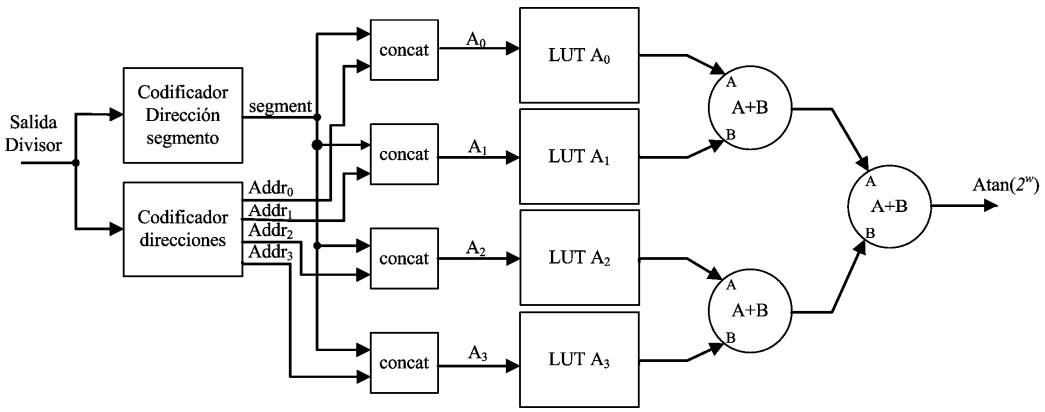


Figura 4.18 - Implementación del generador direcciones de la aproximación basada en LUTs y segmentación no-uniforme.

En la figura 4.19 podemos ver el circuito utilizado para acceder a los distintos segmentos. Este pequeño bloque calculará el valor del segmento, comprobando la cantidad de "unos" y "ceros" que tendremos a la salida del divisor. Los componentes necesarios serán una "cascada" de puertas OR utilizadas para los segmentos que crecen en un factor de dos y una "cascada" de puertas AND para los segmentos que decrecen en un factor de dos. El sumador final contará el número de unos que hay a la

salida de los dos circuitos. Por último, la salida del sumador ira a una pequeña ROM que traducirá el valor del segmento y generará los bits utilizados para direccionar las distintas tablas.

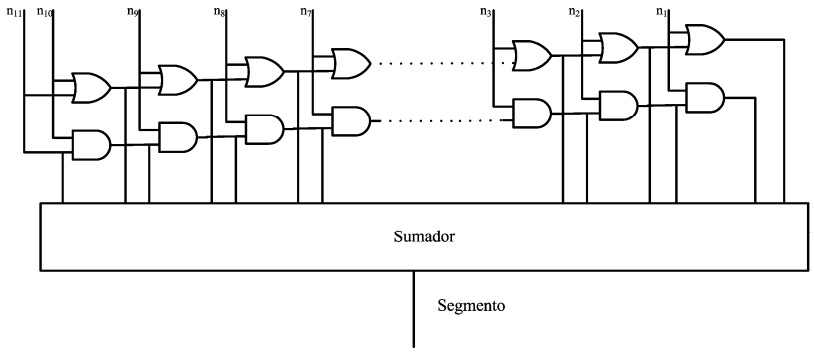


Figura 4.19 - Bloque utilizado para el cálculo segmento direccionado.

4.3.3.2. Aproximación basada en Rectas y Error-LUT

En esta sección vamos a utilizar la aproximación de la atan(2^w) implementada con el método propuesto anteriormente para la aproximación del logaritmo. Debido al comportamiento de la arcotangente dentro del rango de trabajo puede ser factible utilizar el método basado en la aproximación de rectas y una LUT de error. Los tamaños de las rectas y los extremos de las mismas han sido seleccionados mediante prueba y error hasta minimizar el error máximo cometido por la aproximación. Además, los valores de las pendientes de las rectas se han restringido a potencias de dos, permitiendo de esta manera evitar las operaciones de multiplicación necesarias en las interpolaciones lineales. Por último, resaltar una pequeña diferencia en la implementación de la aproximación de la arcotangente con respecto a la aproximación del logaritmo anteriormente propuesta. Para la aproximación del logaritmo utilizábamos el método de Mitchell junto con una etapa de corrección que aumentaba la precisión de la aproximación por medio de la interpolación lineal de su error y añadimos una tabla adicional con el error cometido por la interpolación lineal. Para este caso, hemos aproximado directamente la arcotangente mediante rectas. Para la aproximación de la arcotangente se ha dividido el intervalo de entrada ($0 \le w < 16$) en ocho regiones y en cada una de ellas hemos estimado una recta. La expresión para la aproximación por rectas resultante $y(w)$ es:

$$\begin{aligned}
y_1(w) &= 0.485 + w_{9:0} \cdot 2^{-2}, && 0 \le w < 0.5 \\
y_2(w) &= 0.47 + w_{9:0} \cdot 2^{-2}, && 0.5 \le w < 1 \\
y_3(w) &= 0.59 + w_{9:0} \cdot 2^{-3}, && 1 \le w < 2 \\
y_4(w) &= 0.73 + x_{9:0} \cdot 2^{-4}, && 2 \le w < 3 \\
y_5(w) &= 0.8325 + w_{9:0} \cdot 2^{-5}, && 3 \le w < 4 \\
y_6(w) &= 0.9 + w_{9:0} \cdot 2^{-6}, && 4 \le w < 6 \\
y_7(w) &= 0.965 + w_{7:0} \cdot 2^{-8}, && 6 \le w < 8 \\
y_8(w) &= 0.996 + w_{5:0} \cdot 2^{-10}, && 8 \le w < 16
\end{aligned} \tag{4.20}$$

En la figura 4.20 podemos ver la implementación hardware de la aproximación por rectas propuesta. El bit más significativo de la división logarítmica no es utilizado en esta aproximación. Los cinco bits siguientes seleccionaran una de las ocho rectas mediante una ROM de conversión. Para direccionar la tabla con el error emplearemos los 10 (realmente serán 8 ya que en vez de usar los 5 MSB manejaremos

los tres bits de la salida del conversor) MSB. Dependiendo del tamaño de la tabla, podemos usar los métodos multipartidos para reducir su tamaño.

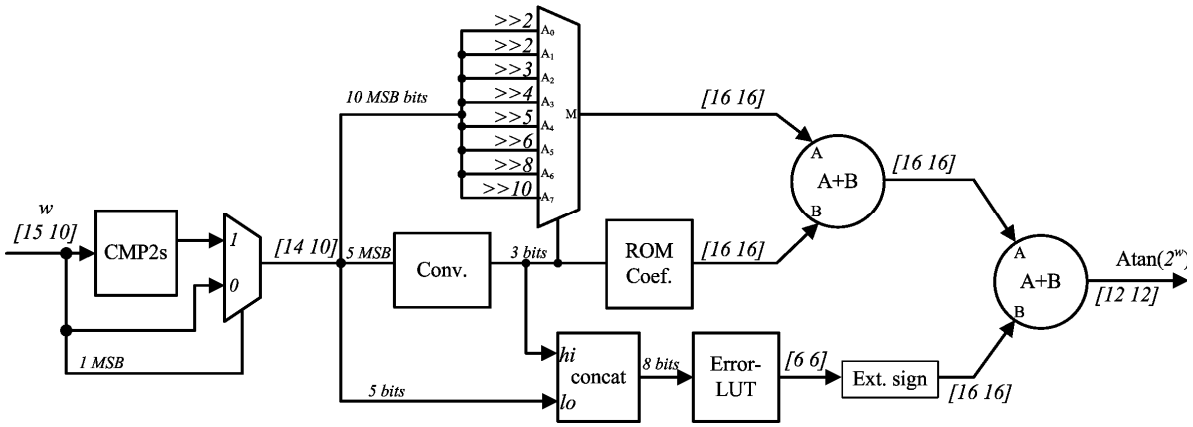


Figura 4.20 – Diseño propuesto para la aproximación de la atan(2^w) mediante rectas y error-LUT.

En la tabla 4.10 podemos ver los distintos tamaños de LUT necesarios para almacenar los valores de la error-LUT utilizados en la aproximación de la arcotangente para distintas precisiones de salida.

Precisión	**12**	**14**	**16**	**18**	**20**	**22**
Tamaño Error-LUT (bits)	256x6	512x9	512x9	512x10 256x9 256x7	1024x12 512x10 256x9	1024x14 1024x11 512x9

Tabla 4.10 – Tamaños de la memoria error-LUT utilizada para la aproximación atan(y/x) mediante rectas.

En la figura 4.21 podemos ver las rectas propuestas para la aproximación comparadas con el valor de la con la atan(2^w) sobre el intervalo [0, 16). En la figura 4.22 podemos ver el error cometido en la aproximación únicamente por rectas. Este error será almacenado en la error-LUT para aumentar la precisión en la aproximación.

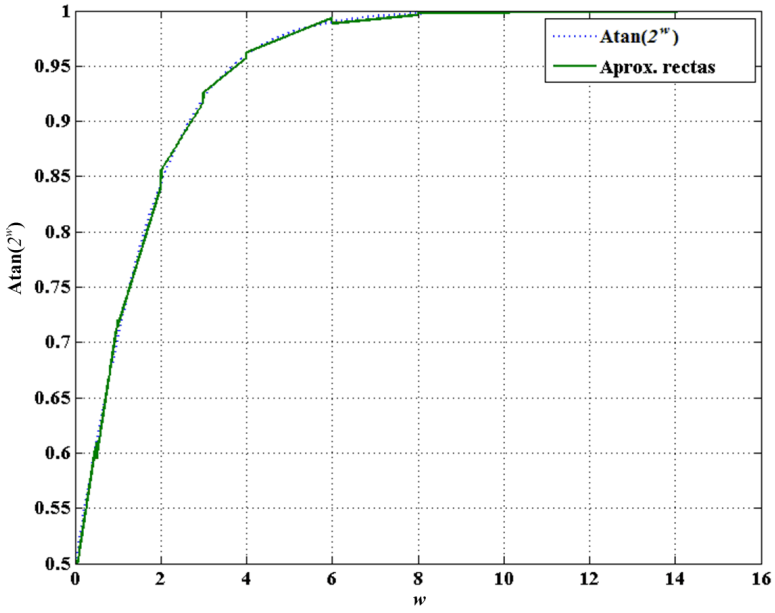


Figura 4.21 - Rectas utilizadas para la aproximación de la atan(2^w).

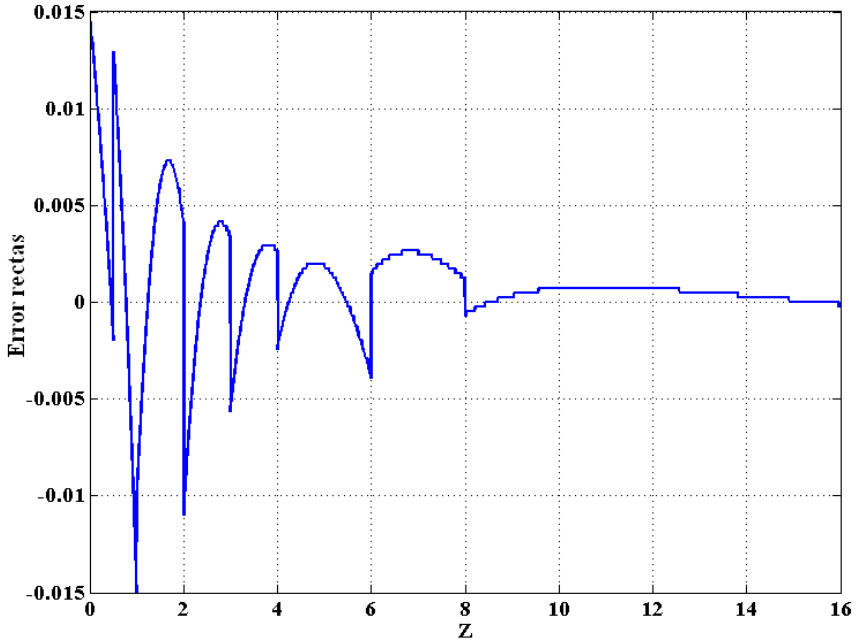


Figura 4.22 – Error cometido en la aproximación por rectas y que será almacenado en la error-LUT.

Por último, en la figura 4.23 vemos el error cometido en la aproximación por rectas y el error cometido por rectas más la error-LUT. Los valores de error los hemos expresado en bits.

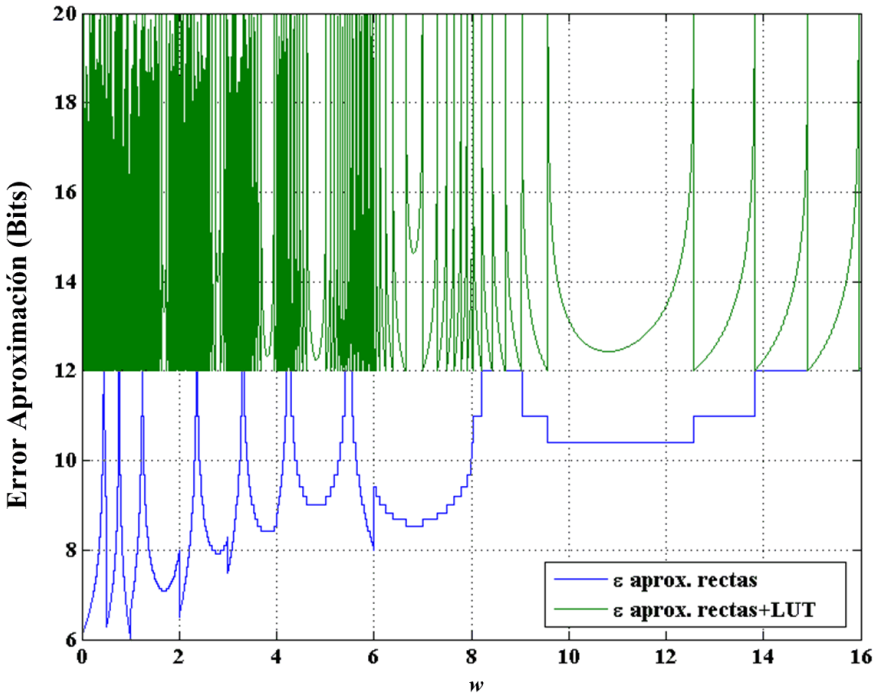


Figura 4.23 - Error cometido por la aproximación por rectas y rectas más error-LUT expresado en bits.

4.4. Implementación y Resultados

Para validar la arquitectura propuesta hemos realizado diferentes modelos de precisión finita mediante System Generator y hemos realizado simulaciones generando todos los posibles valores de las dos entradas, 2^{24} valores. En la figura 4.24 podemos ver los resultados de la simulación. En el peor caso, hemos obtenido una precisión de 10.2 bits, cumpliendo los requerimientos de diseño. Para la implementación de la arquitectura hemos seleccionado la FPGA Virtex-2 XC2V3000FG676-4 de Xilinx. Los resultados de área y velocidad se han obtenido de la herramienta Xilinx ISE 10.1.

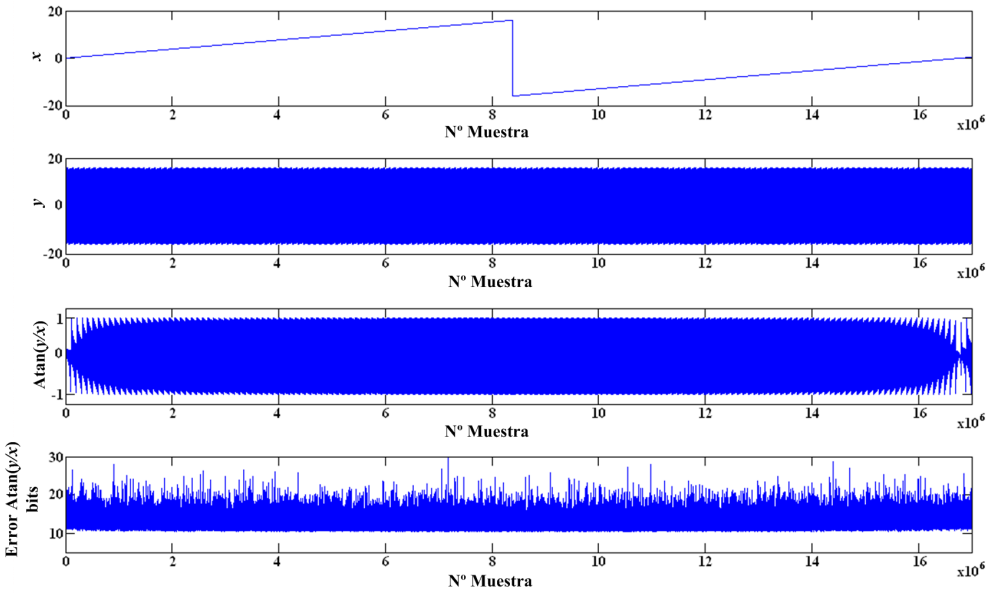


Figura 4.24 - Simulación completa de la aproximación de la atan(y/x) mediante la aproximación del logaritmo por rectas y aproximación de la atan(2^w) mediante LUT multipartida y segmentación no-uniforme .

En las tablas 4.11 y 4.12 podemos ver los resultados obtenidos para la implementación de la atan(y/x) utilizando segmentación no-uniforme basada en tablas y aproximación por rectas de la arcotangente, respectivamente. Para estos dos primeros casos, se ha utilizado para aproximar el logaritmo el método basado en rectas. En la implementación mediante LUTs no-uniforme, las tablas A_0 y A_1 han sido implementadas usando Block-RAM y A_2 y A_3 mediante memoria distribuida ya que son de menor tamaño. El número de segmentos para cada valor de precisión ha sido: 7 segmentos para 12 bits, 10 para 14 bits, 16 para 16 bits, 24 para 18 bits, 38 para 20 bits y 60 para 22 bits. El número de etapas de segmentado utilizadas en los dos casos ha sido de dos, para poder comparar resultados con las arquitecturas del capítulo anterior. Comparando los dos resultados podemos ver como la implementación basada en rectas de la arcotangente es de media un 15% más rápida que la basada en LUTs. Esta diferencia debe exclusivamente a la etapa que calcula el direccionamiento de la segmentación no-uniforme de aproximación de la atan(2^w), ya que las dos arquitecturas utilizan la misma aproximación al logaritmo. Para altas precisiones vemos como el número de Block-RAMs es la mitad para el caso de rectas con respecto a la aproximación por LUTs, como podemos apreciar la reducción es considerable. Sobre el área utilizada, vemos como para bajas precisiones (12-16) el número de *slices* utilizados en la aproximación por rectas aumenta de media aproximadamente un 25% con respecto a la aproximación por LUTs. Esta diferencia es debida a que las memorias error-LUT han sido mapeadas en la FPGA como memorias distribuidas. Para altas precisiones vemos como esta tendencia cambia y el área utilizada es muy similar.

Precisión	12	14	16	18	20	22
Slices	253	325	362	403	458	538
LUT4	431	552	616	690	779	925
F.F.	95	101	110	120	135	150
Block RAM	1	1	1	2	4	4
Fmax (MHz)	58,8	56,4	55,1	53,6	52	50,4
Latencia	3					
Throughput (Msps)	58,8	56,4	55,1	53,6	52	50,4

Tabla 4.11 - Resultados de implementación de la aproximación de la atan(*y*/*x*) basada en LUTs y segmentación no-uniforme.

Precisión	12	14	16	18	20	22
Slices	314	425	514	406	445	560
LUT4	570	780	990	694	825	1080
F.F.	130	140	160	170	200	220
Block RAM	0	0	0	1	1	2
Fmax (MHz)	68,5	65,9	64,4	63,6	60,9	58,8
Latencia	3					
Throughput (Msps)	68,5	65,9	64,4	63,6	60,9	58,8

Tabla 4.12 - Resultados de implementación de la aproximación de la atan(*y*/*x*) basada en rectas y error-LUT.

Para comparar los resultados obtenidos por las arquitecturas, hemos realizado la aproximación de la atan(*y*/*x*) utilizando aproximaciones para el logaritmo y arcotangente basadas en la utilización de segmentación uniforme y LUTs multipartidas con tres y cuatro tablas, respectivamente. Estos resultados se presentan en la tabla 4.13. Para este caso, como las memorias eran bastante pequeñas hemos implementado las tres memorias utilizadas para la aproximación del log(*1*+*x*) mediante memoria distribuida.

Precisión	12	14	16	18	20	22
Slices	254	333	365	400	538	575
LUT4	397	531	550	642	938	1010
F.F.	135	145	130	138	135	140
Block RAM	1	1	2	2	5	5
Fmax (MHz)	58,9	52,9	49,8	48,6	46,8	45,6
Latencia	3					
Throughput (Msps)	58,9	52,9	49,8	48,6	46,8	45,6

Tabla 4.13 - Resultados de implementación de la aproximación de la atan(*y*/*x*) basada en LUTs multipartidas.

Podemos apreciar como el número de Block-RAMs utilizadas para esta implementación es igual o superior al caso de la implementación basada en LUTs y segmentación no-uniforme. Esta reducción es más significativa para los casos donde la precisión de salida es grande. Para el caso de la velocidad vemos como no hay diferencias entre las dos implementaciones basadas en LUTs, excepto para precisiones grandes, debido al retardo extra de la estructura para el cálculo del segmento. Para comparar las tres arquitecturas hemos realizado una implementación únicamente con *slices*. En la figura 4.25 podemos ver el número de *slices* utilizados y el *throughput* obtenido variando la precisión de salida.

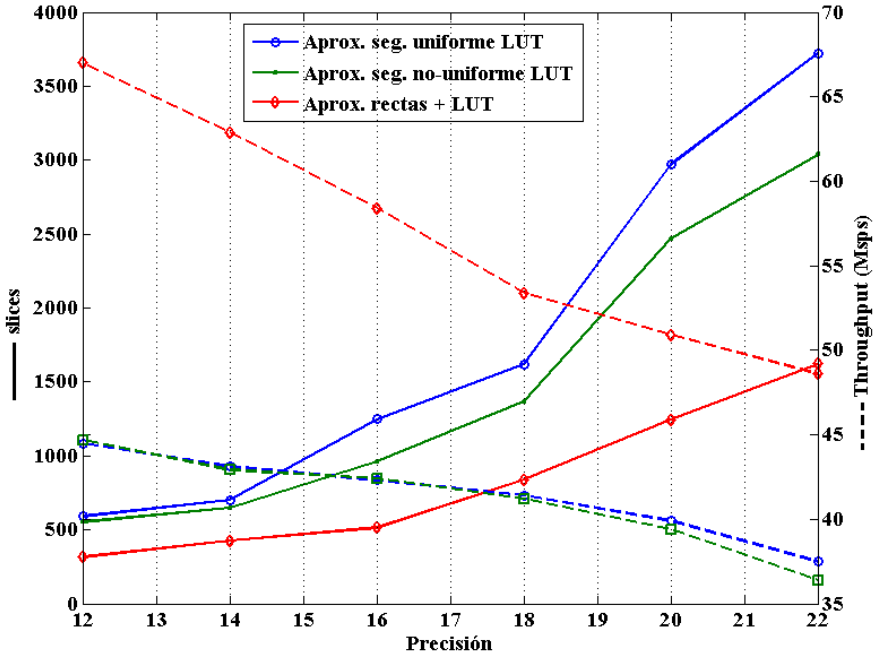


Figura 4.25 - Comparativa de área y velocidad de los distintos métodos propuestos para la aproximación atan(y/x).

A partir de los resultados mostrados por las implementaciones utilizando solo *slices*, vemos como el área utilizada es mucho menor para el caso de la aproximación por rectas que para los dos casos basados en LUTs. Esta diferencia aún es mayor conforme vamos aumentando la precisión de la salida. En cuanto a la velocidad, vemos como también va siguiendo la misma tendencia, en este caso la velocidad disminuye conforme aumentamos la precisión de salida. En las implementaciones basadas en tablas vemos como las diferencias de velocidad son mínimas excepto para los dos últimos casos (20 y 22 bits).

En las tablas 4.14 y 4.15 podemos ver los resultados de implementación de la aproximación de la atan(y/x) mediante LUTs multipartidas (segmentación no-uniforme) y aproximación por rectas de la arcotangente, respectivamente. Para este caso la etapa del cálculo del logaritmo está implementada mediante una aproximación multipartida formada por tres tablas. Vemos como el número de Block-RAMs utilizadas es el mismo que en el caso anterior (tabla 4.11), ya que las tres tablas utilizadas para la aproximación del logaritmo se han mapeado en memoria distribuida. El resto de resultados obtenidos de las dos implementaciones son bastante parecidos con la diferencia en el número de Block-RAMs utilizadas.

Precisión	12	14	16	18	20	22
Slices	362	425	460	525	585	685
LUT4	410	590	658	795	910	1210
F.F.	105	115	120	126	132	145
Block RAM	1	1	1	2	4	4
Fmax (MHz)	57,5	54,5	50.4	47,9	44,4	43,7
Latencia	3					
Throughput (Msps)	57,5	54,5	50.4	47,9	44,4	43,7

Tabla 4.14 - Resultados de implementación de la atan(*y*/*x*) mediante LUTs y segmentación no-uniforme. El operador logaritmo esta implementado con LUTs multipartidas.

Precisión	12	14	16	18	20	22
Slices	402	465	740	585	635	835
LUT4	580	790	1046	845	1056	1380
F.F.	156	165	172	188	204	215
Block RAM	0	0	0	1	2	2
Fmax (MHz)	59,8	56,8	53.1	49,7	47,4	45,9
Latencia	3					
Throughput (Msps)	59,8	56,8	53.1	49,7	47,4	45,9

Tabla 4.15 - Resultados de implementación de la atan(*y*/*x*) mediante rectas y error-LUT. El operador logaritmo esta implementado con LUTs multipartidas.

En las tablas 4.16 y 4.17 podemos ver las implementaciones con el operador multiplexado del logaritmo para las arquitecturas de la arcotangente mediante LUTs y rectas, respectivamente. Podemos ver como el número de Block-RAMs utilizadas no ha reducido en una Block-RAM para altas precisiones. También podemos apreciar como el *throughput* de muestras generadas es la mitad de la frecuencia máxima debido a que para este caso necesitamos dos ciclos de reloj por cada división logarítmica. En cuanto a la reducción de área obtenida es de aproximadamente un 30,7% para el caso de la implementación por LUTs y de un 26.5% para el caso de la aproximación por rectas. Esta diferencia en cuanto a las dos implementaciones es debida a la herramienta de Place & Route más que a diferencias en el diseño ya que en los dos casos el operador del logaritmo es el mismo. Las diferencias de velocidad obtenidas con respecto a la versión sin multiplexar se deben mayormente a tener que añadir el multiplexor a la entrada de la etapa de complemento a dos y a modificar la posición de la etapa de biestables.

Precisión	12	14	16	18	20	22
Slices	165	212	245	280	335	405
LUT4	274	354	425	490	545	690
F.F.	105	110	120	135	140	150
Block RAM	1	1	1	2	3	3
Fmax (MHz)	53,3	51,8	50	47,9	46,7	46
Latencia				4		
Throughput (Msps)	26,6	25,9	25	23,9	23,3	23

Tabla 4.16 - Resultados de implementación de la aproximación de la atan(y/x) basada en LUTs y segmentación no-uniforme. Operador logaritmo multiplexado.

Precisión	12	14	16	18	20	22
Slices	230	295	380	300	340	415
LUT4	419	541	730	512	598	756
F.F.	91	100	108	120	126	140
Block RAM	0	0	0	1	1	2
Fmax (MHz)	58,6	57,4	55,6	54,2	52	50,3
Latencia				4		
Throughput (Msps)	29,3	28,7	27,8	27,1	26	25,1

Tabla 4.17 - Resultados de implementación de la aproximación de la atan(y/x) basada en rectas y error-LUTs. Operador logaritmo multiplexado.

Para analizar la bondad del diseño, se han realizado dos implementaciones de las arquitecturas propuestas (LUT no-uniforme y rectas) completamente segmentadas. Para la aproximación del logaritmo se ha utilizado el método basado en rectas. Se ha utilizado únicamente esta arquitectura para la aproximación del logaritmo debido a que los resultados de las dos arquitecturas propuestas no difieren mucho. En las tablas 4.18 y 4.19 mostramos los resultados obtenidos. Podemos apreciar el aumento considerable de velocidad obtenido con respecto a las implementaciones con mínima latencia. Los resultados de área no difieren de las dos implementaciones excepto en el numero de Block-RAMs. Para estas implementaciones con un alto grado de segmentación (hemos añadido 13 etapas de registros) la velocidad ha aumentado hasta aproximadamente 166.7 MHz de media para la implementación con LUTs no-uniforme y para el caso de la aproximación por rectas hemos llegado hasta los 154.1 MHz. Hay que tener en cuenta que el *speed-grade* de los dispositivos utilizados es el más bajo.

Precisión	12	14	16	18	20	22
Slices	270	327	362	405	460	545
LUT4	435	560	620	694	782	930
F.F.	373	401	432	465	485	520
Block RAM	1	1	1	2	4	4
Fmax (MHz)	166,7	163,5	159,2	156,8	152,1	150,9
Latencia	14					
Throughput (Msps)	166,7	163,5	159,2	156,8	152,1	150,9

Tabla 4.18 - Resultados de implementación de la aproximación de alta velocidad de la atan(y/x) basada en LUTs y segmentación no-uniforme.

Precisión	12	14	16	18	20	22
Slices	289	432	516	407	450	561
LUT4	581	795	1001	697	828	1084
F.F.	289	321	348	412	430	460
Block RAM	0	0	0	1	1	2
Fmax (MHz)	154,1	152,6	149,9	148,2	146,2	144,5
Latencia	14					
Throughput (Msps)	154,1	152,6	149,9	148,2	146,2	144,5

Tabla 4.19 - Resultados de implementación de la aproximación de alta velocidad de la atan(y/x) basada en rectas y error-LUT.

4.4.1. Comparación de Resultados Obtenidos de la Implementación Propuesta con Diferentes Diseños.

Por último, vamos a comparar las distintas arquitecturas presentadas en el capítulo 4, con las arquitecturas propuestas en el capítulo 3 y CORDIC. Para ello vamos a utilizar los resultados basados en una implementación de baja precisión (14 bits) y otra, con una alta precisión de salida (20 bits). Se han añadido los resultados de consumo de potencia de las distintas implementaciones. Para analizar la influencia de los elementos embebidos en el consumo de potencia se han realizado todas las arquitecturas implementando los *barrel-shifter* mediante *slices* y MULT18X18S disponibles en la FPGA Virtex-II. De esta manera podemos analizar la influencia de la utilización de los bloques embebidos para la reducción del consumo de potencia en las arquitecturas propuestas. En la tabla 4.20 y 4.21 presentamos los resultados obtenidos para las implementaciones con *barrel-shifter* con *slices* para una baja y alta precisión de salida, respectivamente. Todas las implementaciones han sido realizadas añadiendo dos etapas de segmentación y el número mínimo de Block-RAM necesarias. Las medidas de consumo se han realizado para una frecuencia de reloj de 20 MHz.

Método	Arq. capítulo 4				Arq. capítulo 3		CORDIC
	rectas[1]	LUT seg. no-unif[1].	LUT seg. unif[2].	LUT seg. no-unif[2]	Bipartida	Multipartida	
Mult18x18	0	0	0	0	1	1	0
Slices	425	325	333	425	260	278	387
LUT4	780	552	531	590	388	438	620
F.F.	140	101	145	115	114	115	208
Block RAM	0	1	1	1	2	4	0
F_{max} (MHz)	65.9	56.8	52.8	54.5	46.6	40.5	47.9
Latencia	3	3	3	3	3	3	5
Throughput (Msps)	65.9	56.4	52.8	54.5	46.6	40.5	47.9
Consumo (mW/MHz)	3.97	2.85	2.75	3.05	2.08	2.54	3.16

Tabla 4.20 - Comparativa de resultados de implementación de la aproximación de la atan(y/x) para baja precisión de salida.

A partir de los resultados obtenidos de la tabla 4.20, vemos que la implementación más rápida es la basada en rectas aunque es la que más área (LUTs) utiliza. Este aumento en el numero de LUTs utilizadas es debido a que no hemos utilizado ninguna Block-RAM para almacenar las error-LUT. Por el contrario, la implementación del capítulo 3 es la que menos LUTs utiliza aunque necesita dos Block-RAM. De las arquitecturas presentadas en el capítulo 4 basadas en la aproximación de la arcotangente mediante LUTs vemos que el hecho de utilizar segmentación uniforme/no-uniforme no afecta para nada al número de Block-RAMs utilizadas. Esto es debido al tamaño de las Block-RAM de Xilinx de 18 kbits, aunque reduzcamos la cantidad de memoria necesaria, siempre necesitaremos una Block-RAM. Vemos como la aproximación bipartida es la implementación que menos consumo de potencia ha generado, pero es de las implementaciones más lentas. En general las implementaciones basadas en tablas son las que menos consumo de potencia han generado. Podemos apreciar que la utilización de Block-RAM permite reducir los consumos de potencia con respecto a la utilización de memoria distribuida (*slices*). Las arquitecturas basadas en aproximación por rectas y CORDIC, son las que más consumo de potencia generan debido a que únicamente se utilizan *slices*.

Al aumentar la precisión de trabajo cuyos resultados se muestran en la tabla 4.21, la implementación basada en rectas sigue siendo la que mejor velocidad ofrece, además de ser la que menos Block-RAM necesita. Ofreciendo el mejor ratio velocidad, área y potencia de todas las implementaciones. Para el caso de la implementación bipartida del capítulo anterior debido a la utilización del método bipartido y su baja compresión de tablas, el número de Block-RAM necesarias aumenta considerablemente (20), siendo además, la implementación más lenta de todas. Podemos apreciar que a pesar del aumento considerable de Block-RAM utilizadas no influye significativamente en el aumento de consumo de potencia de las implementaciones, siendo más importante en número de *slices* utilizados. Para estos tamaños de palabra el uso de la segmentación uniforme sí que ha dado sus frutos debido a que hemos reducido el número de Block-RAMs utilizadas de 5 a 4.

[1] Aproximación del logaritmo mediante interpolación por rectas más error-LUT.
[2] Aproximación del logaritmo mediante una LUT multipartida de 3 LUTs.

Método	Arq. capítulo 4				Arq. capítulo 3		CORDIC
	rectas[3]	LUT seg. no-unif[3].	LUT seg. unif[4].	LUT seg. no-unif[4]	Bipartida	Multipartida	
Mult18x18	0	0	0	0	3	3	
Slices	445	458	538	585	378	380	577
LUT4	825	779	938	910	560	580	955
F.F.	200	135	135	132	164	65	244
Block RAM	1	4	5	4	20	5	0
F_{max} (MHz)	60.9	52	46.8	44.4	26.7	28.8	32.9
Latencia	3	3	3	3	3	3	5
Throughput (Msps)	60.9	52	46.8	44.4	26.7	28.8	32.9
Consumo (mW/MHz)	4.3	4.25	5.18	4.96	4.98	4.16	4.87

Tabla 4.21 - Comparativa de resultados de implementación de la aproximación de la atan(y/x) para alta precisión de salida.

Método	Arq. capítulo 4				Arq. capítulo 3		CORDIC
	rectas[3]	LUT seg. no-unif[3].	LUT seg. unif[4].	LUT seg. no-unif[4]	Bipartida	Multipartida	
Mult18x18	2	2	2	2	3	3	0
Slices	360	228	235	240	157	210	387
LUT4	685	365	375	386	260	318	620
F.F.	88	80	80	78	62	68	208
Block RAM	0	1	1	1	2	4	0
F_{max} (MHz)	63.9	53.8	51.6	52.9	44.5	35.8	47.9
Latencia	3	3	3	3	3	3	5
Throughput (Msps)	63.9	53.8	51.6	52.9	44.5	35.8	47.9
Consumo (mW/MHz)	3.77	2.3	2.36	2.4	1.7	2.3	3.16

Tabla 4.22 - Comparativa de resultados de implementación de la aproximación de la atan(y/x) para baja precisión de salida. Barrel-shifter implementado mediante multiplicadores embebidos.

En las tablas 4.22 y 4.23, presentamos los resultados de implementación obtenidos por las distintas implementaciones con baja y alta precisión de salida utilizando para los *barrel-shifter* los multiplicadores embebidos MULT18X18. Como podemos apreciar el número de LUT4 se ha reducido un 27.5% de media para bajas precisiones y un 24% para altas precisiones, exceptuando CORDIC. Al reducir el número de *slices* utilizados, vemos como se ha reducido el consumo de potencia en un 14.5% de media para bajas precisiones y un 8.1% para altas precisiones. Para altas precisiones debido al aumento en el número de multiplicadores necesarios esta reducción es menos significativa. Estas implementaciones de *barrel-shifter* serán interesantes en dispositivos que dispongan de multiplicadores embebidos.

[3] Aproximación del logaritmo mediante interpolación por rectas más error-LUT.
[4] Aproximación del logaritmo mediante una LUT multipartida de 3 LUTs.

Método	Arq. capítulo 4				Arq. capítulo 3		CORDIC
	rectas[5]	LUT seg. no-unif[5].	LUT seg. unif[6].	LUT seg. no-unif[6]	Bipartida	Multipartida	
Mult18x18	6	6	6	6	12	12	0
Slices	310	390	467	458	294	300	577
LUT4	580	624	747	732	405	422	955
F.F.	170	160	162	162	155	156	244
Block RAM	1	4	5	4	20	5	0
F_{max} (MHz)	58.2	51.9	46.1	48.5	24.8	23.1	32.9
Latencia	3	3	3	3	3	3	5
Throughput (Msps)	58.2	51.9	46.1	48.5	24.8	23.1	32.9
Consumo (mW/MHz)	3.6	4.1	4.73	4.56	4.89	3.85	4.87

Tabla 4.23 - Comparativa de resultados de implementación de la aproximación de la atan(y/x) para alta precisión de salida. Barrel-shifter implementado mediante multiplicadores embebidos.

4.5. Conclusiones

En este capítulo se ha presentado una arquitectura que nos permite aproximar la atan(y/x) de una manera eficiente para su implementación en FPGAs. Para ello se ha dividido el cálculo de la función en dos etapas. Primero, se ha realizado la operación unaria $w=y/x$ por medio de logaritmos evitando la realización de la operación división. Para ello se han propuesto dos métodos para realizar eficientemente el cálculo del logaritmo binario de la entrada. El primero está basado en la aproximación de la función $\log(1+x)$ utilizada en la conversión logarítmica por medio de la utilización de una tabla multipartida, permitiendo reducir la memoria necesaria para su aproximación por LUTs. La segunda arquitectura utilizada para la conversión del logaritmo está basada en el método de Mitchell sobre la que se ha añadido una etapa de corrección compuesta por una interpolación lineal del error cometido por el método de Mitchell sobre la que se añade una LUT donde almacenamos el error cometido por la interpolación lineal. De esta manera, conseguimos aumentar la precisión del método de Mitchell con un bajo coste hardware ya que las pendientes de las rectas utilizadas son potencias de dos y podemos realizar la multiplicación por medio de desplazamientos. La aproximación propuesta por rectas más una error-LUT permite aumentar la precisión del logaritmo con respecto a otros métodos propuestos con los mismos recursos.

Segundo, a partir de la división logarítmica realizamos el cálculo de la arcotangente del antilogaritmo de la división. Para esta aproximación también hemos realizado dos aproximaciones. Por un lado, hemos realizado la aproximación de la atan(2^w) mediante tabla multipartidas sobre las que hemos realizado una segmentación no-uniforme que ha permitido reducir la cantidad de memoria utilizada para la aproximación. La segunda arquitectura está basada en la arquitectura propuesta para la aproximación del logaritmo, con una pequeña diferencia, en este caso aproximamos la función y no el error cometido. Al igual que en el caso anterior también hemos añadido una tabla con el error cometido por la interpolación por rectas que nos permitirá aumentar la precisión de la aproximación. Una ventaja adicional es que si el tamaño de la tabla crece enormemente podemos reducir su tamaño aplicando un método de tablas multipartidas. Comparando las dos propuestas para la aproximación de la arcotangente podemos ver

[5] Aproximación del logaritmo mediante interpolación por rectas más error-LUT.
[6] Aproximación del logaritmo mediante una LUT multipartida de 3 LUTs.

como la implementación basada en la utilización de rectas utiliza menos memoria que la basada en tablas utilizando un esquema de segmentación no-uniforme, además de ofrecer un mayor *throughput* tanto para bajas como para altas precisiones. La utilización de segmentación no-uniforme puede ser beneficiosa cuando vamos a trabajar con altas precisiones, ya que para bajas precisiones la reducción no es tan apreciable debido en parte al tamaño de 18 kbits de las Block-RAM.

Todas las arquitecturas han sido implementadas mediante System Generator y VHDL utilizando los dispositivos Xilinx (Virtex II XC2V3000 y Virtex II Pro XC2VP30). Los resultados obtenidos muestran que nuestra arquitectura puede trabajar perfectamente con el *throughput* requerido en SDR de 20 MHz con una latencia de dos ciclos de reloj. También hemos implementado versiones con un alto grado de segmentación (14 ciclos de reloj de latencia) llegando a velocidades de hasta 150 MHz. Por último, multiplexando el operador logaritmo podemos reducir el área de las implementaciones aproximadamente un 30%, con el inconveniente de la bajada del *throughput* obtenido a la mitad, ya que únicamente podemos realizar una división cada dos ciclos de reloj.

BLOQUE 2.

GENERACION DE NUMEROS ALEATORIOS GAUSSIANOS.

Capítulo 5.

Generadores de Números Aleatorios Gaussianos

En los sistemas de comunicaciones el canal es el medio físico utilizado para enviar información entre el emisor y el receptor. Dentro de las comunicaciones digitales, una medida de cómo afecta el ruido a la información transmitida es el BER. El BER es el cociente entre el número de bits recibidos con error y el número total de bits transmitidos en un intervalo de tiempo. Actualmente es muy importante evaluar los sistemas de comunicaciones en los que el BER es muy bajo. Esto conlleva tiempos de simulación altos. Para evitar este inconveniente, será necesario poder modelar en hardware el canal de comunicación. Uno de los modelos más ampliamente empleados para el canal es el de ruido aditivo blanco Gaussiano (Additive White Gaussian Noise – AWGN). En el capítulo 5 vamos a presentar distintos métodos propuestos por otros autores para la Generación de Números Aleatorios con una distribución Gaussiana (GNAG) en hardware.

El capítulo 5 está dividido en tres grandes bloques: primero vamos a revisar los distintos métodos disponibles para la generación de muestras aleatorias con una distribución Gaussiana. Un elemento común de todos los métodos es que su punto de partida es la utilización de muestras aleatorias uniformemente distribuidas. En el segundo bloque se presentarán los métodos más usados para la generación de muestras pseudoaleatorias con una distribución uniformemente distribuida. Por último, presentaremos varios test estadísticos que nos van a permitir caracterizar el comportamiento aleatorio de las muestras generadas.

5.1. Introducción

En los sistemas de comunicaciones, utilizamos un canal para transmitir información entre un transmisor y un receptor. La figura 5.1 muestra el diagrama funcional y los elementos básicos de un sistema de comunicación digital.

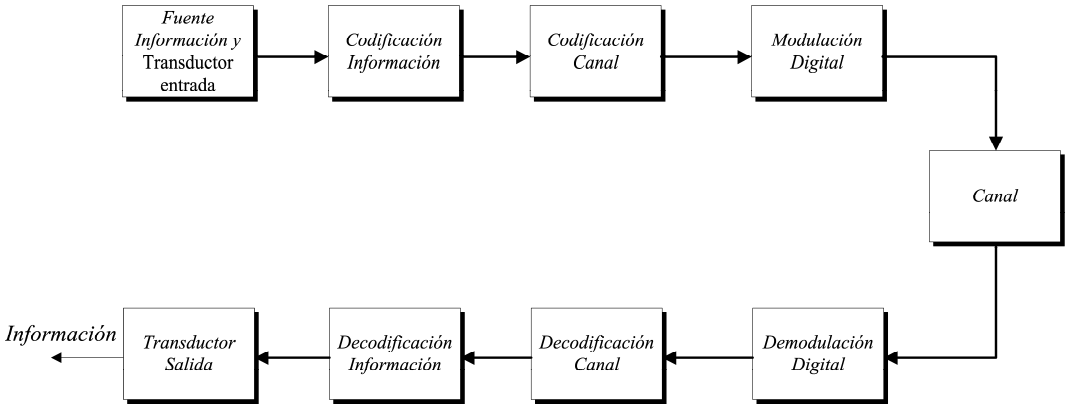


Figura 5.1 - Elementos básicos de un sistema de Comunicaciones digital.

El transmisor procesa la información de los datos a transmitir de acuerdo a las características del canal y el receptor acepta la señal del canal y recupera la información transmitida. El canal es el medio físico usado para la transmisión de la señal entre el transmisor y el receptor. Un problema asociado al canal es que la información transmitida se puede corromper de una manera aleatoria debido a una gran variedad de posibles mecanismos, como puede ser el ruido térmico generado por los dispositivos electrónicos, ruido atmosférico, etc. Debido a estos efectos, el canal de comunicación puede introducir ruido que causará errores sobre la información transmitida. El parámetro BER indica la tasa de bits erróneos recibidos divididos por el número de bits transmitidos durante un intervalo de tiempo determinado. El BER dependerá de las características del código utilizado, de la forma de onda utilizada para transmitir los datos, potencia transmitida, características del canal y los métodos de demodulación y decodificación. De todos los factores comentados, el principal factor que afecta al parámetro del BER es el ruido en el canal. Este ruido presente en el canal está normalmente descrito mediante una **Función de Densidad de Probabilidad** (PDF) de la distribución Gaussiana. Si podemos representar matemáticamente esta función, podemos predecir el parámetro BER de un sistema. El BER es un parámetro que está íntimamente relacionado con el parámetro señal-ruido (SNR). Análisis teóricos [58] para un canal en banda base, relacionan el parámetro BER con el parámetro SNR mediante la relación

$$BER = Q\sqrt{SNR} \tag{5.1}$$

siendo Q, el área bajo la cola de la **Función Distribución Probabilidad o Función Densidad Probabilidad Acumulada** (CDF) de la distribución Gaussiana, también llamada función *Quantile*.

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_{x}^{\infty} e^{-t^2/2} dt \tag{5.2}$$

Si en vez de trabajar en banda base, añadimos alguna modulación sobre la información transmitida, el parámetro BER variará dependiendo del sistema de modulación utilizado. En (5.3) vemos unos ejemplos para modulaciones QPSK y M-PSK.

$$\begin{aligned} BER &= 2Q()\left(\sqrt{2SNR}\right)\cdot\left[1-\frac{1}{2}Q\left(\sqrt{2SNR}\right)\right] \;\rightarrow\; \text{QPSK} \\ BER &\approx 2Q\left(\sqrt{2\cdot\log M\cdot SNR}\sin\left(\frac{\pi}{M}\right)\right) \;\rightarrow\; \text{M-PSK} \end{aligned} \tag{5.3}$$

De acuerdo con la ley de Shannon, la capacidad del canal C para un ancho de banda dado B, dependerá del parámetro SNR mediante la relación

$$C = B\cdot\log_2\left(1+SNR\right) \tag{5.4}$$

En la práctica, los diseñadores de sistemas de comunicaciones deberán ajustar los parámetros de ancho de banda y SNR para maximizar la capacidad del canal con un BER aceptable. Es por ello que será interesante poder obtener rápidamente las curvas tanto teóricas como prácticas que relacionan el parámetro SNR con el BER obtenido y de esta manera poder evaluar la capacidad del canal y la ganancia de la codificación utilizada. Esto será equivalente a un aumento en el SNR del sistema. De ahí, que sea necesario poder calcular el parámetro BER de un dispositivo fabricado para todos los casos posibles.

A partir de (5.1) y (5.3), podemos observar que para obtener bajos valores de BER será necesario trabajar con altos valores de SNR. En un canal basado en ruido Gaussiano, el parámetro SNR vendrá definido por la varianza del generador AWGN. Para emular sistemas con bajo BER, utilizaremos Gaussianas estrechas que nos permitirán obtener bajas varianzas. Bajos valores de BER vendrán dados por la distribución en la cola de la Gaussiana o más específicamente, por el valor máximo de la variable ruido (σ) o desviación estándar de la distribución Gaussiana. En un sistema digital donde los valores transmitidos son '0' y '1', el valor máximo del generador de ruido tendrá que ser mayor que 0.5 para poder generar bits de error. Implementaciones hardware de generadores AWGN obtienen valores máximos de salida de 7σ [58]. Si utilizamos este generador en una transmisión en banda base obtendremos un valor de SNR=16.9 dB, valor que se traducirá aplicando (5.1) en un valor teórico de BER$\approx 10^{-12}$. Ahora bien estos generadores únicamente serán adecuados para trabajar con valores de BER de hasta 10^{-10}. Los valores de la distribución generados cerca de los límites del valor máximo de salida de cualquier canal AWGN con una cola truncada, no serán completamente Gaussianos ya que la distribución ideal Gaussiana es infinita. De ahí que sea necesario para aplicaciones de muy bajo BER implementar canales AWGN con un alto valor máximo de salida.

En las siguientes secciones del capítulo vamos a estudiar los distintos métodos definidos para poder generar canales AWGN necesarios para las implementaciones de muy bajo BER. Todo generador de muestras aleatorias Gaussianas estará compuesto por dos grandes bloques: Un generador de valores aleatorios uniformemente distribuidos y otro bloque que transformará las muestras uniformemente distribuidas en muestras aleatorias distribuidas Gaussianamente. Para ello vamos a revisar los diferentes métodos para la generación de muestras aleatorias y los métodos utilizados para su transformación.

5.1.1. Caracterización de un Canal de Ruido Aditivo Blanco Gaussiano

Los canales de comunicación proveen una conexión entre un transmisor y un receptor. Hay diferentes tipos de canales de comunicación físicos: aire, fibra óptica, líneas cableadas, etc. Es conveniente la realización de un modelo matemático que nos permita reproducir las características más importantes del medio de transmisión. Este modelo del canal será utilizado para realizar un diseño óptimo del codificador del canal y modulador en el transmisor y el demodulador y decodificador del canal en el receptor.

El modelo de canal basado en ruido blanco Gaussiano es el modelo más utilizado para el estudio y diseño de los sistemas de comunicaciones. El modelo matemático del ruido aditivo blanco Gaussiano (AWGN – *Additive White Gaussian Noise*) se muestra en la figura 5.2.

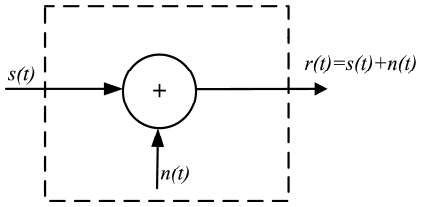


Figura 5.2 - Modelo de un canal AWGN utilizado en comunicaciones.

En un modelo de canal AWGN, la señal transmitida *s(t)* es corrompida por un ruido *n(t)*. Este ruido será generado tanto por el canal, como también por los componentes electrónicos utilizados tanto en el transmisor como en el receptor. Este tipo de ruido es caracterizado como un ruido térmico o estadísticamente, como un proceso de ruido Gaussiano. La salida del canal de comunicación es la suma de la señal determinista y del ruido aleatorio, y viene dada por

$$r(t) = s(t) + n(t), \tag{5.5}$$

siendo *s(t)* una señal analítica con amplitud A y *n(t)* un valor complejo con valor medio igual a cero y ruido Gaussiano. La parte real e imaginaria de *n(t)* se asume que son mutuamente independientes y con una varianza σ^2 y por consiguiente tendremos la relación señal a ruido (SNR):

$$SNR = \frac{A^2}{2\sigma^2}. \tag{5.6}$$

- **<u>Propiedades teóricas de AWGN</u>**

Antes de caracterizar las propiedades teóricas del ruido Gaussiano, vamos a comenzar revisando las características de las variables aleatorias y a las definiciones de **función de distribución de probabilidad** (CDF) y **función de densidad de probabilidad** (PDF). Las variables aleatorias son normalmente descritas por medio de sus estadísticas, entre las propiedades más importantes tenemos la media o esperanza (5.7), la media al cuadrado (5.8) y su varianza (5.9). La definición de estos parámetros para una variable *x* caracterizada por su PDF – (P(*x*)) serán:

$$E(x) = m_x = \sum_n x_n \cdot P(x_n) \tag{5.7}$$

$$m_x^2 = \sum_n x_n^2 \cdot P\left(x_n\right) \tag{5.8}$$

$$\sigma^2 = E(x^2) - m_x^2 \tag{5.9}$$

Dada una variable *X*, para el evento ($X \leq x$), donde *x* es cualquier valor comprendido en el intervalo ($-\infty$, ∞), la probabilidad de que este evento se produzca, estará definido como la probabilidad P(X) de que el evento X será menor que la probabilidad P(*x*) del evento *x*, o simplificando como F(x). La función F(*x*) es llamada **función de distribución de la probabilidad** de una variable aleatoria X (también es llamada función de distribución acumulativa *Cumulative distribution function* – CDF). Como F(*x*) indica probabilidad, su rango estará definido en el intervalo $0 \leq F(x) \leq 1$. La derivada de F(x), escrita como p*(x)*, es llamada **función de densidad de probabilidad** (*Probability Density function* - PDF) de una variable aleatoria.

$$p(x) = \frac{dF(x)}{dx}. \qquad -\infty \leq x \leq \infty \tag{5.10}$$

De todas las funciones de probabilidad de los sistemas de comunicaciones digitales, la función de densidad de la distribución Gaussiana es la más utilizada. La distribución Gaussiana también es llamada distribución normal. La función de densidad de probabilidad (PDF) de una variable aleatoria Gaussiana vendrá dada por

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-m_x)^2 / 2\sigma^2}, \tag{5.11}$$

siendo m_x y σ^2 la media y la varianza de una variable Gaussiana, respectivamente. En la figura 5.3 se puede ver una representación de la distribución Gaussiana de una variable aleatoria para un valor $m_x = 0$ y $\sigma^2 = 1$.

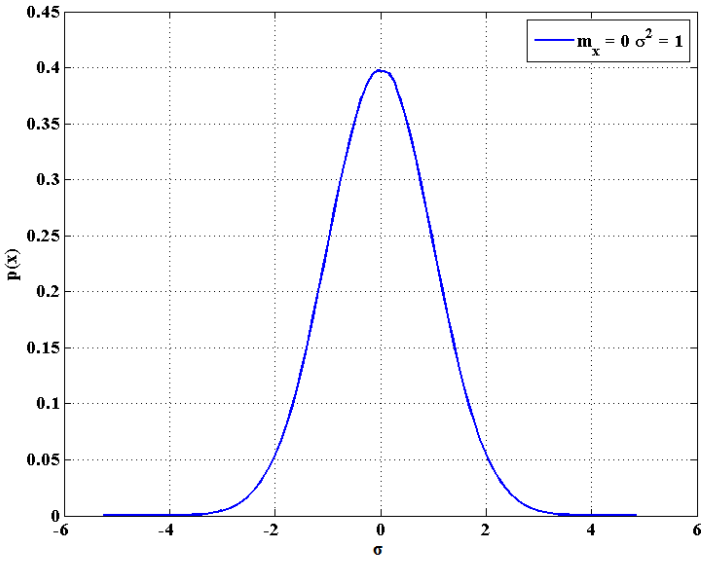


Figura 5.3 – Valores de la Función Distribución Probabilidad (PDF) de la distribución Gaussiana de media cero y varianza uno.

La función de distribución acumulativa de una variable aleatoria será

$$
\begin{aligned}
F(x) &= \int_{-\infty}^{x} p(u)\,du \\
&= \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-(u-m_x)^2/2\sigma^2}\,du \\
&= \frac{1}{2}\frac{2}{\sqrt{2\pi}} \int_{-\infty}^{(u-m_x)/\sigma\sqrt{2}} e^{-t^2}\,dt \\
&= \frac{1}{2} + \frac{1}{2} erf\left(\frac{x-m_x}{\sigma\sqrt{2}}\right).
\end{aligned}
\tag{5.12}
$$

Siendo *erf(x)* (función error), definida como

$$
erf(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{t^2}\,dt. \tag{5.13}
$$

La función de distribución acumulativa también puede ser definida mediante la función de error complementaria

$$
\begin{aligned}
F(x) &= 1 - \frac{1}{2} erfc\left(\frac{x-m_x}{\sigma\sqrt{2}}\right) \\
erfc &= \frac{2}{\sqrt{2\pi}} \int_{x}^{\infty} e^{-t^2}\,dt = 1 - erf(x).
\end{aligned}
\tag{5.14}
$$

Para $x > m_x$, la función de error complementaria es proporcional al área bajo la cola de la función de distribución de probabilidad Gaussiana. En la figura 5.4 se puede ver representada la CDF de una variable Gaussiana, vemos que es una función antisimétrica.

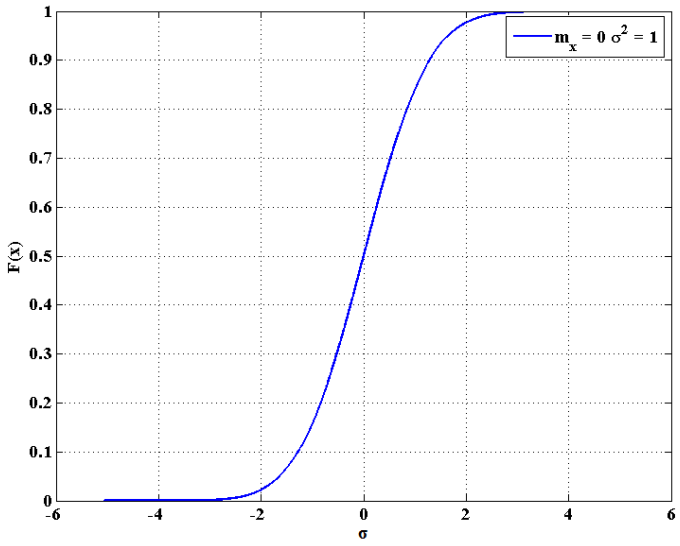


Figura 5.4 – Valores de la Función de Distribución Acumulada Probabilidad (CDF) de la distribución Gaussiana de media cero y varianza uno.

Otra importante función utilizada para la caracterización de la distribución Gaussiana es la función Q, la cual representa el área bajo la cola de la función de densidad Gaussiana. Q(x) es la función más importante para el cálculo de la probabilidad de error en los sistemas de comunicaciones y está definida para una media de cero y una varianza de uno, como:

$$\begin{aligned} Q(x) &= \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-t^2/2} dt, \qquad x \geq 0 \\ &\approx \frac{1}{x\sqrt{2\pi}} \left[1 - \frac{1}{x^2} \right] e^{-x^2/2} \\ Q(x) &= \frac{1}{2} erfc\left(\frac{x}{\sqrt{2}} \right). \end{aligned} \tag{5.15}$$

Por último, tendremos que $F(x)+Q(x)=1$.

5.2. Generación de Variables Aleatorias Gaussianas

Una vez caracterizado el canal de ruido blanco Gaussiano, vamos a estudiar las distintas técnicas que nos van a permitir generar valores aleatorios con una distribución Gaussiana. Un amplio rango de métodos para la Generación de Números Aleatorios Gaussianos (GNAG) han sido descritos en la literatura. Estos métodos estarán basados en la aplicación de principios básicos matemáticos, normalmente transformaciones a partir de números aleatorios uniformemente distribuidos. Por cada variable uniformemente distribuida, podemos obtener una variable no-uniformemente distribuida. Para algunas distribuciones tenemos multitud de algoritmos que nos permiten generar muestras aleatorias. Para el caso de la generación de variables aleatorias Gaussianas, vamos a clasificar los distintos algoritmos en cuatro categorías básicas [59]: Inversión de la Función de Distribución Acumulada (ICDF) [60], métodos basados en transformaciones [61], métodos aceptación-rechazo [62][63] y los métodos recursivos [64]. Los métodos basados en la ICDF simplemente invierten la CDF de la distribución Gaussiana para generar muestras aleatorias Gaussianas. Los métodos basados en Transformaciones implican el uso de transformaciones directas de muestras aleatorias uniformemente distribuidas para generar las muestras aleatorias Gaussianas. Los métodos recursivos, utilizan combinaciones lineales de muestras Gaussianas previamente generadas para generar las nuevas muestras. Por último, los métodos aceptación-rechazo, aplican transformaciones sobre muestras uniformemente distribuidas añadiendo una etapa final de rechazo condicional de alguna de las muestras generadas.

Otro criterio para la clasificación de los métodos para la generación de variables aleatorias Gaussianas es el de si los métodos son "exactos" o "aproximados" [59]. Los métodos exactos son aquellos que producirán números aleatorios perfectamente Gaussianos si su implementación se realiza en un entorno "ideal". Por ejemplo, el algoritmo Box-Muller [61] es un método basado en la realización de diversas transformaciones mediante funciones elementales sobre variables uniformemente distribuidas para generar variables aleatorias. Si las muestras son realmente uniformemente distribuidas e infinitamente precisas, las funciones se evalúan utilizando precisión infinita, para ese caso, las muestras generadas serán perfectamente Gaussianas. Los métodos aproximados, producirán muestras aproximadamente Gaussianas si la aritmética utilizada es perfecta. Un ejemplo es el caso del Teorema del Limite Central (CLT), el cual únicamente será exacto cuando un número infinito de muestras uniformemente distribuidas sean sumadas y por ello debe ser aproximado en cualquier implementación práctica.

Los algoritmos normalmente diferirán en velocidad, precisión, requerimientos de memoria o complejidad en la codificación/implementación. Por ejemplo, Box-Muller es un método más complejo de implementar pero tiene la ventaja de que genera dos muestras por iteración. El método CLT es un algoritmo fácil de implementar, únicamente utiliza un acumulador pero la calidad de las muestras generadas es muy baja. Por último, comentar que los métodos que vamos a comentar en el presente capítulo son llamados métodos "universales" en el sentido que pueden ser utilizados para la generación de variables con un distribución Gaussiana como para otro tipo de distribución (Weibull, Gamma, Beta, etc.). Algunos autores suelen llamarlos métodos *black-box*. Algunos de estos métodos serán mejores que otros para una distribución en particular o para un rango especifico de una distribución.

Estos métodos, especialmente los basados en inversión de la CDF, únicamente se pueden utilizar para la generación de variables aleatorias univariable, mientras otros métodos se pueden aplicar directamente sobre variables aleatorias multivariables. A continuación pasamos a comentar los métodos más utilizados para la generación de números aleatorios Gaussianos.

5.2.1. Método de la Inversión de la Función de Distribución Acumulada (ICDF).

El método basado en la inversión de la función de distribución acumulada (ICDF) trabaja a partir de la utilización de números aleatorios uniformemente distribuidos entre [0, 1) y aplicando la inversión de la función (5.12) $y = F^{-1}(x)$. Mientras que F(x) asocia números Gaussianos con una probabilidad comprendida entre cero y uno, $F^{-1}(x)$ permite generar valores Gaussianos a partir de valores comprendidos entre cero y uno. En la figura 5.5 podemos ver el funcionamiento, sobre el rango donde la derivada de la CDF es grande, hay más probabilidad de realizar una variación uniforme.

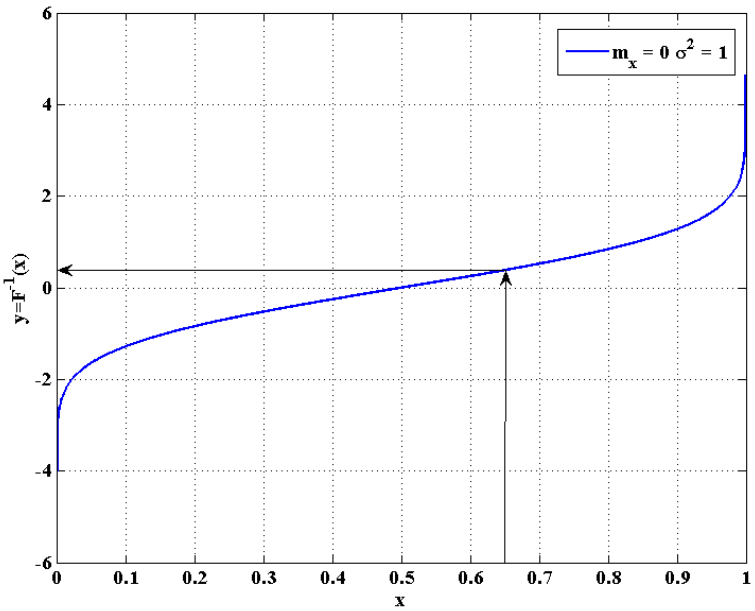


Figura 5.5 - Funcionamiento del método de la Inversión

La ICDF permite relacionar dos variables aleatorias continuas entre ellas. Si X es una variable aleatoria continua con una CDF P_X e Y es una variable aleatoria continua con una CDF P_Y, entonces

$$X = P_X^{-1}(P_Y(Y)). \tag{5.16}$$

La gran ventaja de este método es que permite generar variables aleatorias con CDF arbitrarias [60]. Además la calidad de las muestras generadas únicamente dependerá de la calidad de las muestras del generador de números uniformemente distribuidos. Este hecho difícilmente pueden ofrecerlo otros métodos de generación de números aleatorios Gaussianos [65]. Por otro lado, el método de la ICDF tiene un gran problema debido a la dificultad de poder aproximar la ICDF para algunas distribuciones de interés. Este método no es muy utilizado a pesar de la simplicidad de su funcionamiento. Para estos casos se han desarrollado métodos basados en la integración numérica que generan aproximaciones con una alta precisión pero con unos costes computacionales muy grandes haciendo impracticables su utilización para implementaciones de generación de números aleatorios, sobre todo en las regiones de la cola de la Gaussiana. Para el caso de distribuciones como la Gaussiana (normal), exponencial o log-normal se suelen utilizar aproximaciones por polinomios y en menor medida aproximaciones por tablas.

Uno de los primeras aproximaciones fue presentada por Muller [66], el cual aproxima la ICDF con una precisión de $4 \cdot 10^{-4}$ para un rango de entrada de ($6 \cdot 10^{-7}$, $1-6 \cdot 10^{-7}$), correspondiente a un rango de salida de $\pm 5\sigma$. Para su implementación divide el rango de trabajo en 64 pares de segmentos simétricos y utiliza una interpolación lineal. En este caso, el autor pretende que sea más importante la velocidad de generación que la precisión de las muestras. Otra aproximación propuesta por [67] utiliza una aproximación polinómica por medio de la integración numérica de la PDF. En este caso, proponen la división de la CDF en 99 sub-intervalos los cuales tienen una probabilidad de 1/64 en el centro y pequeñas probabilidades en las colas. Dentro de los intervalos, se aproxima la ICDF re-escalada por medio de interpolación de Lagrange mediante nueve puntos y esos coeficientes son convertidos en polinomios de Chebyshev.

Más recientemente en [82] se proponen dos métodos en los cuales divide el rango de la aproximación en dos regiones y utiliza para cada región un polinomio racional. Para las entradas comprendidas en el rango (0.075, 0.925) se utiliza el polinomio racional (5.17) y para los valores fuera de este rango se utiliza (5.18). La gran ventaja de esta aproximación, es que como la gran mayoría de las muestras caen en la primera región, las funciones raíz cuadrada y logaritmo neperiano, únicamente necesitan ser calculadas el 15% del tiempo. El primer método llamado PPND7 genera 7 dígitos decimales de precisión en el rango (10^{-316}, $1-10^{-316}$) utilizando polinomios racionales de grado 2 y 3 y el segundo PPND16, obtiene 16 dígitos decimales de precisión utilizando polinomios racionales de grado 7 sobre el mismo rango.

$$(x-0.5)^2 \tag{5.17}$$

$$\sqrt{-\ln(x)} \tag{5.18}$$

5.2.2. Método Box-Muller

La transformada Box-Muller [61] (George Edward Pelham Box y Mervin Edgar Muller, 1958) es uno de los primeros métodos exactos desarrollados basados en transformación de variables. Este método produce un par de números aleatorios Gaussianos a partir de un par de números aleatorios uniformemente

distribuidos. Este método está basado en la propiedad de que la distribución 2D de dos variables independientes aleatorias Gaussianas de media cero son simétricas, si ambas componentes Gaussianas tienen la misma varianza. Si U_1 y U_2 son dos variables aleatorias uniformemente distribuidas e independientes, y

$$\begin{aligned} Y_1 &= \sqrt{-2\ln(U_1)}\cos(2\cdot\pi\cdot U_2) \\ Y_2 &= \sqrt{-2\ln(U_1)}\sin(2\cdot\pi\cdot U_2), \end{aligned} \tag{5.19}$$

entonces, Y_1 e Y_2 serán independientemente distribuidas con una distribución Gaussiana y una desviación estándar de uno. El algoritmo Box-Muller puede ser fácilmente entendible como un método en el cual los números Gaussianos generados representan coordenadas en un plano bi-dimensional. La magnitud del correspondiente vector R^2 es obtenida por la transformación de un número uniformemente aleatorio y la fase aleatoria Φ es obtenida por medio de la transformación de un segundo número uniformemente aleatorio y escalado por 2π, como se puede apreciar en (5.20).

$$\begin{aligned} R^2 &= -2\cdot\ln(U_1) \\ \Phi &= 2\cdot\pi\cdot U_2. \end{aligned} \tag{5.20}$$

El método Box-Muller es un algoritmo relativamente lento en aplicaciones software ya que necesita aproximar las funciones raíz cuadrada y el logaritmo, además de las funciones seno y coseno. Por otro lado, tiene la ventaja de que se generan dos muestras por cada iteración. Otro inconveniente del método Box-Muller, es que no es muy recomendable su utilización para implementaciones con un alto valor de desviación estándar de la distribución, σ. Como podemos ver (5.19), el mayor valor de Y vendrá determinado por el valor de $\sqrt{-2\ln(U_1)}$, el cual tenderá a infinito conforme U_1 tienda a cero. Si utilizamos un generador de números uniformemente distribuidos de 32 bits, el número generado más pequeño será 2^{-32} y obtendremos un valor de aproximadamente 6.56. Si aumentamos el tamaño a 64 bits el valor únicamente crecerá hasta los 9.4. Como podemos apreciar la complejidad de las implementaciones puede llegar a ser relativamente alta si queremos trabajar con valores altos de σ debido al tamaño de los operadores y de la cantidad de funciones elementales necesarias. Este problema también aparece en el método de la inversión.

5.2.3. Método Polar

El método Polar (definido por G. Marsaglia) [62], es un algoritmo con cierto parecido al algoritmo Box-Muller, ya que trabaja con dos variables aleatorias U_1 y U_2. Estas dos variables estarán uniformemente distribuidas dentro del rango [-1,1]. Además, añadiremos la condición $s = r^2 = U_1^2 + U_2^2$. Si $s = 0$ o $s \geq 1$, descartamos las muestras generadas y volvemos a generar muestras hasta que $r^2 \in (0,1)$. Debido a que las muestras U_1 y U_2 están uniformemente distribuidas y únicamente serán admitidas muestras que están dentro del circulo unidad, los valores de r^2 estarán uniformemente distribuidos dentro del intervalo (0, 1). A partir de (5.19) y añadiendo la condición $s = r^2 = U_1^2 + U_2^2$, tenemos

$$Y_1 = \sqrt{-2\ln(U_1)}\cos(2\cdot\pi\cdot U_2) = \sqrt{-2\ln(s)}\cdot\left(\frac{U_1}{\sqrt{s}}\right) = U_1\cdot\sqrt{\frac{-\ln(s)}{s}}$$
$$Y_2 = \sqrt{-2\ln(U_1)}\sin(2\cdot\pi\cdot U_2) = \sqrt{-2\ln(s)}\cdot\left(\frac{U_2}{\sqrt{s}}\right) = U_2\cdot\sqrt{\frac{-\ln(s)}{s}}. \quad (5.21)$$

El método Polar difiere del método básico o Box-Muller en que se implementa en combinación con métodos aceptación-rechazo. La ventaja del método polar es que no necesita realizar la aproximación de las funciones trigonométricas. Por el contrario, la magnitud *s* tiene que ser menor que uno, lo cual ocurre con una probabilidad de $\pi/4$, requiriendo $4/\pi\approx1.2732$ muestras uniformes por cada muestra Gaussiana generada. El método Box-Muller necesita tres multiplicaciones, un logaritmo, una raíz cuadrada y una función trigonométrica por cada muestra generada. El método Polar necesita dos multiplicaciones, un logaritmo, una raíz cuadrada y una división por cada muestra generada.

5.2.4. Método Ziggurat

Los algoritmos más rápidos para la generación de muestras aleatorios son los basado en el uso de métodos *ratio-of-uniforms* o una mezcla con los métodos aceptación/rechazo. El método Ziggurat [63] es un método exacto englobado dentro de los algoritmos de aceptación/rechazo. Estos métodos están basados en la generación aleatoria de un punto. A continuación, comprobamos si el punto ha caído dentro o fuera de la distribución deseada. Si ha caído fuera volvemos a intentarlo hasta que caiga dentro. A partir de la PDF p*(x)* monótonamente decreciente y definida para todo $x \geq 0$, la base de la Ziggurat estará definida como todos los puntos que hay dentro de la distribución y debajo de y_1=p(x_1). Esta consistirá en una región rectangular desde (0, 0) hasta (x_1, y_1), y la cola de la distribución, donde $x>x_1$ y $y<y_1$. Este primer rectángulo tendrá una área A. Encima de este rectángulo, añadiremos otro de anchura x_1 y altura A/x_1, de esta manera su área también será A. La altura de estos dos cuadrados será de $y_2=y_1+A/x_1$ e interseccionará con la distribución en el punto (x_2, y_2). En este nuevo nivel, contendrá cualquier punto de la distribución entre Y_1 e Y_2, pero además contendrá como (x_2, y_2) que están fuera de la distribución. Iremos añadiendo niveles hasta que lleguemos a la parte alta de la distribución.

De esta manera tendremos que la curva de la PDF encerrará a las distintas secciones R_i $(1\leq i\leq n)$, hechas a partir de *n-1* rectángulos además de la región de la cola. Todos los rectángulos serán divididos en dos regiones: sub-rectángulo el cual estará completamente dentro de la PDF y la otra región (en la parte derecha del sub-rectángulo) con forma de cuña, que incluye partes inferior y superior de la PDF. Para un valor n=256, la probabilidad de obtener una región rectangular es del 99%. La figura 5.6 muestra un ejemplo del funcionamiento del método Ziggurat.

El algoritmo Ziggurat funciona de la siguiente manera. Cada vez que vamos a generar un valor aleatorio, una de las *n* secciones es aleatoriamente elegida y una muestra uniformemente aleatoria de *x* es generada y evaluada para ver si está dentro del sub-rectángulo de la sección elegida, que estará completamente dentro de la PDF. Si esta condición se cumple, la muestra *x* es una muestra aleatoria Gaussiana. Si no se cumple la condición significa que *x* está cerca de la región de la cuña y calcularemos un valor de escalado apropiado. Si los valores *x* e *y* están por debajo de la PDF en la región de la cuña, entonces *x* es un valor válido. Si los valores están por encima se descartan y comienza de nuevo el algoritmo. En la figura 5.7 presentamos un pseudocódigo que modela el funcionamiento del método Ziggurat.

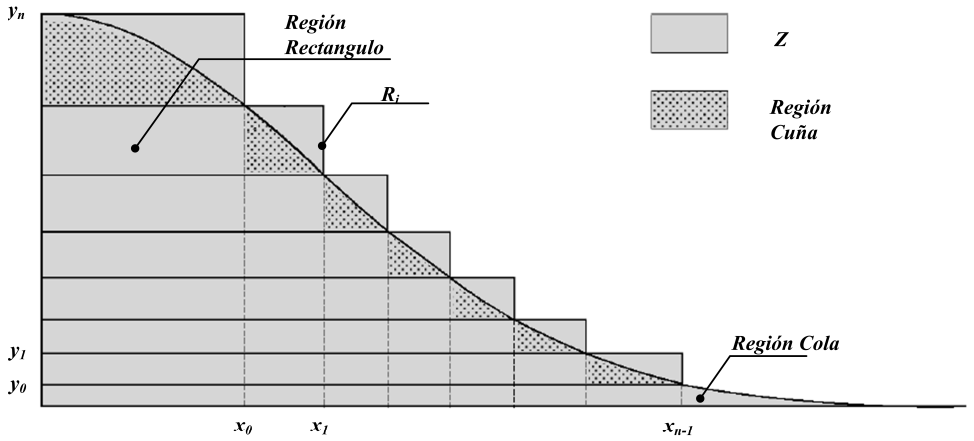


Figura 5.6 - División de la distribución Gaussiana por medio de rectángulos, cuñas y la región de cola.

Algoritmo Ziggurat

```
1: Blucle infinito
2:   i = 1 + ⌊nU1⌋ ; -- Siendo U1 una variable uniformemente distribuida
                       y n una potencia de 2
3:   x = xi · U2 ; -- Siendo U2 una variable uniformemente distribuida
4:  if |x| < xi-1 :
5:    return z ; -- El punto esta dentro de un rectangulo
6:  elsif i ≠ n :
7:    y = (φ(xi-1) − φ(xi)) · U ;-- generamos posicion vertical aleatoria
8:    if y < (φ(x) − φ(xi)) : -- Test posicion dentro PDF
9:      return x ; -- El punto esta dentro de la cuña.
10: elsif i = 0 : generamos valor cola
11:   x = −ln(U1)/x1 ;
12:   x = −ln(U2) ;
13:   if 2y > x^2 :
14:     return x+x1
15:   else
16:     jump linea 11;
17: Fin Blucle infinito
```

Figura 5.7 - Pseudocódigo del algoritmo para la realización del método Ziggurat.

5.2.5. Método Iterativo o de Wallace

El método de generación de números aleatorios Gaussianos propuesto por Wallace [64] está basado en la propiedad de que una combinación lineal números aleatorios Gaussianamente distribuidos, es en sí misma una distribución Gaussiana, evitando de esta manera la evaluación de funciones elementales necesarias en los otros métodos.

El método de Wallace, al ser iterativo, utiliza una tabla inicial de K=MN números aleatorios independientes de una distribución Gaussiana previamente calculados y normalizados para que su valor medio cuadrático sea uno. N será el numero de etapas de transformación necesarias y M serán los números Gaussianos inicialmente calculados y que contendrán los M nuevos valores a partir de la expresión X'=AX, siendo A una matriz Ortogonal. Si los M valores iniciales son Gaussianamente distribuidos, los nuevos M valores también lo serán. El proceso de generación del nuevo juego de valores Gaussianamente distribuidos es conocido como "pasada" y se suelen realizar R pasadas antes de hacer que los valores generados sean utilizados permitiendo una mejor incorrelación de las muestras generadas. Los valores iniciales utilizados en el algoritmo estarán normalizados para que su valor cuadrático medio sea igual a 1. Además, como A será una matriz ortogonal, las siguientes iteraciones no modificarán la suma de los cuadrados. En la figura 5.8 podemos ver un pseudocódigo del método de Wallace.

Algoritmo Método de Wallace

```
1: for i= 1..R : Numero de pasadas
2:   for j= 1..L : --(L=N/K);
3:     for z= 1..K : -- K = tamaño matriz ;
3:       x[z] = tabla_ini_gauss(U); -- U valor aleatorio unif.
4:     end for:
5:     x' = transf_muestras;
6:     for z= 1..K : -- Actualizamos tabla valores Gauss.
7:       tabla_ini_gauss(U) = x';
8:     end for:
9:   end for:
10:end for:
11:S = √(tabla_ini_gauss[N]/N) ; -- Calculamos coeficiente corrector
12:return tabla_ini_gauss(1..N-1)·S; valores nueva tabla escalados
```

Figura 5.8 - Pseudocodigo del algoritmo para la realización del método Wallace.

Los parámetros normalmente utilizados del pseudocódigo que suelen dar un buen resultado estadístico sin comprometer el rendimiento del algoritmo son R=2, L=1024 y K=4. En la implementación de Wallace la matriz ortogonal utilizada estará basada en la matriz de Hadamard. La matriz Hadamard es una matriz ortogonal con la propiedad de que todos los elementos son +1 y -1, haciendo particularmente eficiente la implementación. Las matrices utilizadas las podemos ver en (5.22) para el parámetro K=4.

$$A_1 = \frac{1}{2}\begin{pmatrix} 1 & 1 & -1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & -1 & -1 & -1 \\ -1 & -1 & -1 & 1 \end{pmatrix} \quad A_2 = \frac{1}{2}\begin{pmatrix} 1 & -1 & -1 & -1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ -1 & -1 & -1 & 1 \end{pmatrix}$$
$$A_3 = \frac{1}{2}\begin{pmatrix} 1 & -1 & 1 & 1 \\ -1 & -1 & 1 & -1 \\ -1 & -1 & -1 & 1 \\ -1 & 1 & 1 & 1 \end{pmatrix} \quad A_4 = \frac{1}{2}\begin{pmatrix} -1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 \\ -1 & 1 & 1 & 1 \\ 1 & 1 & 1 & -1 \end{pmatrix} \tag{5.22}$$

La ventaja del uso de las matrices Hadamard es que evita el uso de multiplicaciones permitiendo una mayor velocidad de implementación. Por último, el método Wallace no es un método exacto ya que las

muestras generadas provienen de otras muestras previamente generadas y pueden aparecer correlaciones entre muestras.

5.2.6. Método CLT (Central Limit Theorem)

El método CLT está basado en el Teorema del Limite Central (CLT) [69]. De acuerdo con dicho Teorema, si X es una variable real aleatoria de media m_x y una desviación estándar σ_x, la variable aleatoria X_N estará definida como

$$X_N = \frac{1}{\sigma_x \sqrt{N}} \sum_{i=0}^{N-1} (x_i - m_x) \tag{5.23}$$

y tenderá hacia una distribución Gaussiana de media cero y una varianza de uno, cuando N tienda hacia infinito. En (5.23) x_i serán N instancias independientes de la variable X. En la práctica, la suma de K valores aproximará una Gaussiana de media cero y una varianza de $\sqrt{K/12}$. El gran problema de este método es que converge muy lentamente hacia la Gaussiana conforme aumentamos K, además de proporcionar una aproximación muy mala de la cola de la Gaussiana. Normalmente, este método suele ser utilizado en implementaciones hardware [59] combinado dos o más métodos de generación de números Gaussianos de baja calidad para obtener un generador de alta calidad. El método CLT es un método aproximado ya que para un numero finito K de muestras la salida nunca será Gaussiana.

5.3. Generación de Variables Aleatorias Uniformemente Distribuidas

De acuerdo con los algoritmos vistos previamente, el primer paso para la generación de una variable aleatoria Gaussiana es la generación de una o más variables aleatorias uniformemente distribuidas dentro del rango (0, 1). En el pasado, todos los métodos desarrollados estaban basados en implementaciones software [70][71], pero tenían el problema de la necesidad de realizar complejas operaciones aritméticas con la imposibilidad de una implementación hardware eficiente. En el presente apartado, vamos a examinar los métodos hardware desarrollados para la implementación óptima de Generadores de Números Aleatorios Uniformemente Distribuidos (RNG). En la actualidad disponemos de dos tipos de RNG: RNG "verdadero" (T-RNG) y Pseudo RNG (P-RNG) [62]. Idealmente, las variables aleatorias generadas deberían ser valores aleatorias incorreladas y deben satisfacer cualquier test de aleatoriedad.

Los T-RNGs utilizan fenómenos físicos, como pueden ser, ruidos térmicos de los componentes electrónicos o el "*jitter*" de un oscilador, para la generación de los valores aleatorios. La gran ventaja es que las muestras generadas no son predecibles y son de gran interés en el campo de la criptología. El problema de estos generadores es que no consiguen altas tasas de bits y es imposible repetir simulaciones con los mismos estímulos.

Una solución, es la utilización de los Pseudo Generadores de Números Aleatorios (P-RNG), estos pueden ser descritos mediante la utilización de cuatro parámetros (S, f, R, r) [70]. El parámetro S será el estado actual del generador. F será una función que calculara el siguiente estado a partir del estado actual.

R será el juego de valores aleatorios generados y r será la función que generará el valor aleatorio a partir del estado actual del generador. Un vector aleatorio $x_1, x_2, x_3, \ldots \in R^{\infty}$ será generado a partir de un estado inicial $s_0 \in S$, llamado *semilla*, a partir de ese vector pasaremos al siguiente estado por medio de una ecuación recursiva $s_{i+1} = f(s_i)$ y este nuevo estado generará una nueva salida aleatoria mediante $x_i = r(s_i)$.

La gran ventaja de los PRNG es que *f* y *r* son funciones determinísticas, a partir de la misma semilla, siempre obtenemos el mismo resultado. El periodo del generador vendrá dado por el número de bits utilizados para almacenar el estado *s* del PRNG. Los P-RNGs están basados en la utilización de un algoritmo para la generación de los números aleatorios. Los algoritmos más utilizados son los Generadores Congruenciales Lineales (*Lineal Congruential Generator* – LGC) [71], Generadores Congruenciales de Fibonacci (*Lagged Fibonacci Congruential Generator* - LFCG) [62], Generador suma-con acarreo, resta-con-acarreo y multiplicacion-con acarreo [72], *Linear Feedback Shift Register Generator – LFSR* [73], *Generalized Feedback Shift Registers Generators – GFSR* [74], *Mersenne-Twister* [75], *Tausworthe* [76], etc.

5.3.1. Generadores Congruenciales Lineales – Lineal Congruential Generator

D. H. Lehmer [71] propone la generación de muestras uniformemente distribuidas por medio de los Generadores Congruenciales Lineales (LGC). Estos generadores producirán una secuencia pseudoaleatoria de números x_1, x_2, …, x_m por medio de una función lineal seguido de una operación de modulo *m* (5.24).

$$x_n = (a x_{n-1} + b) \bmod m \quad , \; 0 \le x_i \le m \tag{5.24}$$

Siendo *a*, *b* y *m* los parámetros que caracterizan al generador y x_0 la semilla. El periodo de este generador será como máximo *m*, aunque normalmente este periodo es menor. El problema de este generador, es la alta correlación de las muestras producidas, de ahí que no debería ser utilizado para implementaciones donde se requiera una alta calidad "aleatoria", como pueden ser simulaciones Monte-Carlo o aplicaciones criptográficas. Por último, cuando no tenemos disponible la operación de división, podemos utilizar potencias de dos y en este caso el periodo del generador se reducirá por cuatro.

5.3.2. Generadores Congruenciales de Fibonacci

Los Generadores Congruenciales de Fibonacci [62] son un tipo de generador lineal que permite aumentar la calidad de las muestras generadas con respecto al LGC, haciendo uso de la secuencia de Fibonacci $x_n = x_{n-1} + x_{n-2}$ y combinándola con (5.24) tenemos la siguiente ecuación recursiva

$$x_n = \left(x_{n-j} + x_{n-k}\right) \bmod m \;, \; 0 < j < k. \tag{5.25}$$

Como podemos apreciar de (5.25), el nuevo valor generado será una combinación de dos valores previamente calculados. El modulo de trabajo normalmente son potencias de dos, aunque también se utilizan los valores 232 o 264. Si *m* es un valor primo y k>j, el periodo del generador puede ser tan grande como m^k-1. Si la operación de módulo la realizamos con potencia de dos, el periodo se reduce a $(2^k\text{-}1)\cdot 2^{p\text{-}1}$. Una extensión del generador congruencial de Fibonacci permite la utilización de cualquier operación

binaria además de la suma (resta, multiplicación, la operación or-exclusiva bit a bit, etc). Estos generadores utilizan 2 semillas para generar correctamente los valores (este método necesita recordar los 2 valores anteriores, de ahí que se identifique con el nombre de Fibonacci).

5.3.3. Generador Suma-con-acarreo, resta-con-acarreo y multiplicación-con-acarreo

Marsaglia y Zaman [72] presentaron dos variantes de un generador que ellos llaman suma-con-acarreo (*add-with-carry - AWD*) y resta-con-acarreo (*subtract-with-borrow* – SWB). La implementación AWD esta modelada con la siguiente ecuación

$$x_i = \left(x_{i-s} + x_{i-r} + c_i\right) \mod m \tag{5.26}$$

donde $c_1=0$ y $c_{i+1}=0$ si $x_{i-s} + x_{i-r} + c_i < m$. C será la entrada de acarreo. Dependiendo de los valores de *s*, *r* y *m* podemos llegar a obtener periodos de hasta 10^{43}. Marsaglia también propone el método multiplicación-con-acarreo (*multiply-with-carry – MWC*) el cual es una generalización del método AWD y está definido por medio de

$$x_i = \left(a \cdot x_{i-1} + c_i\right) \mod m. \tag{5.27}$$

5.3.4. Lineal Feedback Shift Register (LFSR)

Tausworthe [73] presentó un generador basado en secuencias de unos y ceros generadas por medio de la ecuación

$$x_i = \left(a_p x_{i-p} + a_{p-1} x_{i-p+1} + \ldots + a_1 x_{i-1}\right) \mod 2, \tag{5.28}$$

debido a que el valor del modulo es primo, el generador estará relacionado por medio del polinomio

$$f(z) = z^p - \left(a_1 z^{p-1} + a_2 z^{p-2} + \ldots + a_{p-1} z + a_p\right), \tag{5.29}$$

sobre un campo de Galois GF(2) definido a partir de los valores enteros 0 y 1 con operaciones de suma y multiplicación seguidos de una operación de división en modulo 2. Un elemento muy importante de (5.28) es que si en el vector inicial *x* todos los elementos son distinto de cero, el periodo del polinomio será $(2^p - 1)$, si y solo si, el polinomio es primitivo. Para simplificar la implementación de (5.28), gran parte de los términos *a* serán cero. Además, si la operación de módulo es igual a dos, sólo tendremos binomios que serán primitivos. La ecuación (5.28) normalmente será reescrita como

$$x_i = x_{i-p} \oplus x_{i-p+q} \tag{5.30}$$

siendo $\oplus$ la operación or-exclusiva. La ecuación (5.28) normalmente será implementada por medio de un registro de desplazamiento realimentado, en el cual, un vector de bits será desplazado una posición hacia la izquierda en cada instante y el bit saliente será combinado con otros bits del polinomio para generar el bit que introduciremos por el bit más hacia la derecha. La gran ventaja de estos generadores, es la facilidad

de implementación en hardware ya que únicamente es necesario un registro de desplazamiento de n-bits y realizar la operación or-exclusiva sobre alguno de los bits. En la figura 5.9 podemos ver la implementación del polinomio $g(x) = x^9 + x^5 + 1$ el cual tiene un periodo de 511.

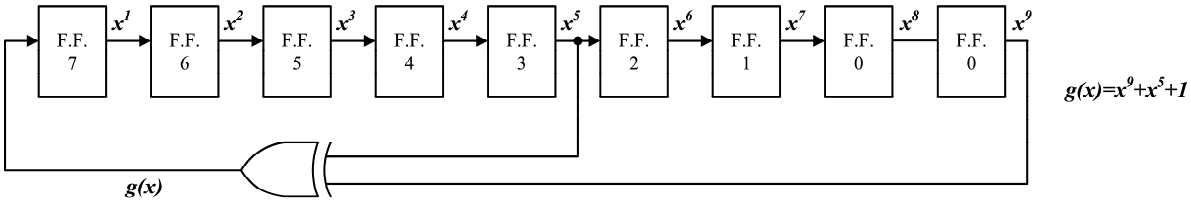


Figura 5.9 - Implementación hardware de un LFSR de 9 bits.

El gran inconveniente de estos generadores es que de las *n* salidas generadas en cada iteración, solo una puede ser considerada aleatoria (la salida de la or-exclusia) ya que las salidas del registro son valores desplazados de la iteración anterior. Si necesitamos generar un valor aleatorio de k-bits necesitamos utilizar k-LFSRs. Además el utilizar k generadores no aumenta el periodo del mismo 2^{kn}-1, sino que sigue siendo 2^n-1. A partir de esta primera implementación del LFSR se han desarrollado diferentes implementaciones basándose en la misma idea. Una primera variación son los *Generalized Feedback shift registers – GFSR* [74], los cuales están basados en la utilización de polinomios característicos de la forma

$$x_i = x_{i-p} \oplus x_{i-r_1} \oplus x_{i-r_2} \oplus ... \oplus x_{i-r_n}. \tag{5.31}$$

La figura 5.10 muestra un GSFR de polinomio $g(x) = x^8 + x^6 + x^5 + x^4 + 1$ de periodo 255. El problema de los GFSR, es que a pesar de añadir más términos a la ecuación recursiva el periodo del generador sigue siendo 2^n-1. Otras modificaciones sobre los LFSR se han hecho en el sentido de aumentar el periodo de los generadores, como son los *Twisted* GFSR, siendo la implementación más conocida la de Mersene Twister [75]. Mersene Twister, es un generador de números aleatorios desarrollado por Matsumoto y Nishimura basado en una iteración matricial lineal $x_{i+t} = x_{i+m} \oplus YR$, siendo $Y = x_t(31):x_{t+1}(30:0)$ y R una matriz 32x32, *m* y *t* son parámetros específicos del generador con $0 \le n < t$, sobre un campo finito binario GF2. Este generador ha sido desarrollado para su implementación en CPUs de 32 bits (MT19937) obteniendo un periodo de 2^{19937}-1. El problema de este método es que a pesar de tener un periodo muy grande la calidad de las muestras es relativamente baja. Implementaciones software de este método podemos encontrarlas para casi todos los lenguajes de programación. Implementaciones hardware basadas en la utilización de FPGAs las podemos encontrar en [77] para una implementación de 32 bits.

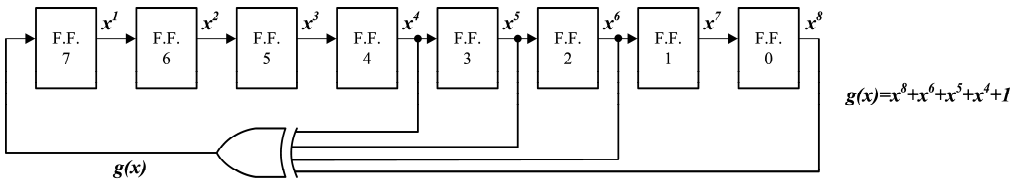


Figura 5.10 - Implementación Hardware de un GFSR de 8 bits.

Por último, los generadores Tausworthe [76] producen números aleatorios a partir de la utilización de k-generadores LFSR de n-bits R_1….R_k, los cuales trabajan independientemente y mediante la realización de la or-exclusiva de todos los valores genera la muestra de salida correcta. El periodo de cada generador

se suele elegir para que sea primo y de esta manera el periodo total será la suma de todos los periodos, teóricamente (2^{kn}-1). En la práctica este periodo será mucho menor. La calidad de los valores generador es mucho mejor que los LFSR y su implementación en hardware es relativamente fácil. En la figura 5.11 podemos ver una implementación de un generador Tausworthe en hardware.

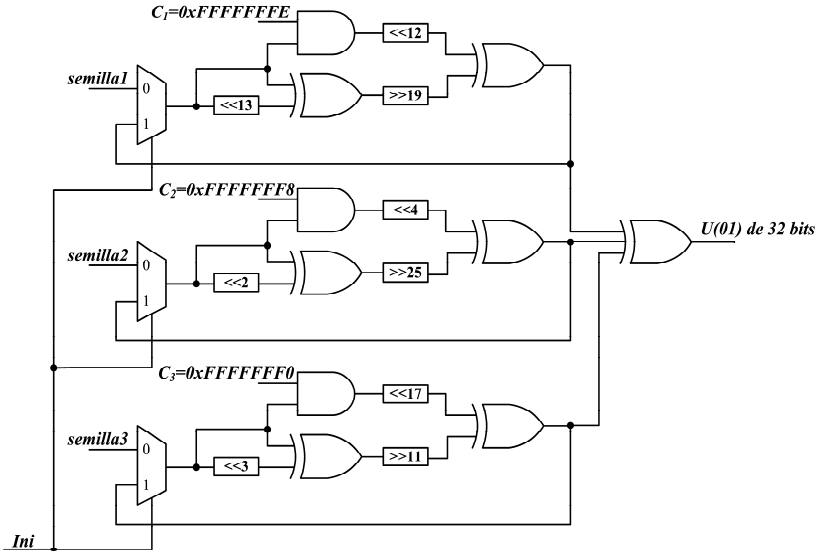


Figura 5.11 - Generador Tausworthe de 32 bits y periodo 2^{88}.

Los valores *semilla1*, *semilla2* y *semilla3* son valores aleatorios. Las constantes están indicadas en la figura con valores hexadecimales. Con esta implementación obtenemos variables aleatorias uniformemente distribuidas de un tamaño de 32 bits. Para otros tamaños y periodos, L'ecuyer [78] propone una serie de tablas para implementaciones de 32 y 64 bits y periodos de hasta 2^{274}.

Además de los métodos vistos en el presente apartado, los cuales son ampliamente utilizados en el campo de las comunicaciones debido a su velocidad, hay multitud de métodos generadores más complejos utilizados en criptografía. Dentro de estos métodos podemos destacar el Blum-Blum-Shub, RSA, etc. Estos métodos están basados en la realización de operaciones aritméticas con números primos relativamente grandes.

5.4. Test Estadísticos

En esta sección vamos a presentar diversos test diseñados para medir la calidad de los generadores de números aleatorios definidos previamente. La calidad de los números aleatorios generados dependerá de cuanto se parezcan las muestras generadas con respecto a muestras realmente aleatorias. En el punto anterior hemos garantizado la obtención de secuencias con un periodo máximo, sin garantizar la aleatoriedad de las muestras. Es por ello que tendremos que decidir si las muestras generadas son lo suficientemente aleatorias como para poder ser utilizadas en nuestras aplicaciones. No podemos juzgar por nosotros mismos si una secuencia de números es o no aleatoria, sino que necesitamos mecanismos matemáticos que nos permitan decidirlo. La aleatoriedad es una propiedad descrita en la probabilidad y por ello las propiedades de una secuencia aleatoria se pueden caracterizar y describir por medio de términos de probabilidad.

Una secuencia puede comportarse de forma aleatoria respecto a una serie de test y aun así no podemos asegurar que ésta siga este comportamiento al someterla a otro test diferente. En la práctica, las muestras generadas las someteremos a un conjunto de test y si los pasan satisfactoriamente, podremos considerarla como aleatoria.

Podemos distinguir entre dos tipos de test:

- Test empíricos: se generan una serie de valores aleatorios y se obtienen ciertas medidas estadísticas. Estas medidas normalmente serán llamadas *goodness-of-fit*.
- Test teóricos: se estudia el método utilizado para la generación de las muestras.

Un test estadístico estará definido para comprobar una hipótesis específica de trabajo o "hipótesis nula" (H_0). Normalmente esta hipótesis a testear será si la secuencia es aleatoria o no. Asociada a la hipótesis nula, tenemos la "hipótesis alternativa" (H_a), la cual indica que la secuencia no es aleatoria. El resultado del test será básicamente aceptar o rechazar la hipótesis nula. Los test estadísticos serán utilizados además para el cálculo del valor-P, el cual resume la "calidad" de la evidencia en contra de la hipótesis nula. Para estos test, el valor-P es la probabilidad de que un generador de números aleatorios perfecto puede producir una secuencia menos aleatoria que la secuencia que ha sido testeada, a partir de la clase de no-aleatoriedad definida en el test. Si en el test obtenemos un valor-P igual a 1, la secuencia generada es perfectamente aleatoria. Por otro lado, si el valor-P es igual a cero, la secuencia es completamente no-aleatoria. Para el cálculo del valor-P definimos un parámetro llamado "nivel significativo" (α). Si valor-P $\geq \alpha$, la hipótesis nula es aceptada y la secuencia parece aleatoria. Si valor-P $< \alpha$, la hipótesis nula es descartada y la secuencia parecerá no-aleatoria. Este valor estará definido entre el rango (0.001, 0.05).

Si se realiza un test con un valor α igual a 0.001, indicará que una secuencia de mil secuencias será rechazada por el test si la secuencia es aleatoria. Para un valor-P $\geq$ 0.001, la secuencia será considerada aleatoria con una confianza del 99.9%. Para un valor-P $<$ 0.001, la secuencia debería ser considerada no-aleatoria con un nivel de confianza del 99.9%. Si el mismo test lo realizamos con un valor α igual a 0.01, en este caso una secuencia de cien secuencias será rechazada. Un valor-P $\geq$ 0.01 indicará que la secuencia debería ser considerada aleatoria con un nivel de confianza del 99%. Un valor-P $<$ 0.01 indicará que la secuencia es no-aleatoria con una confianza del 99%. A continuación vamos a ver los test estadísticos más utilizados.

5.4.1. Test Chi-cuadrado (χ^2)

El test chi-cuadrado es probablemente el test estadístico más utilizado para test de secuencias aleatorias. Este test estadístico fue definido por Karl Pearson. Este test calcula si se cumple la hipótesis nula mediante la agrupación de los valores observados en la secuencia en $k+1$ categorías (bins). Los valores observados serán mutuamente excluyentes y tendrán una probabilidad total de 1.

Este test se basa en el cálculo de la estadística chi-cuadrado y es calculada a partir de

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}, \tag{5.32}$$

siendo

- O_i = los valores de la muestra observada.
- E_i = los valores teóricos esperados.
- n = numero de categorías (bins) utilizadas en el test.

Los valores E_i teóricos serán calculados a partir de

$$E_i = N(F(Y_u) - F(Y_l)), \tag{5.33}$$

siendo F la CDF de la muestra a testear, Y_u el límite superior de la categoría *i*, Y_l el límite inferior de la categoría *i*, y N el número de muestras. Los resultados obtenidos dependerán de la elección del número de categorías utilizadas. Una vez hemos calculado el valor χ^2 lo compararemos con el valor de la distribución chi-cuadrado con *k-c-1* grados de libertad. El parámetro *c* será el número de parámetros necesarios para definir la distribución a testear. Por ejemplo, una distribución normal siempre estará definida por medio de la media y la varianza, con lo cual el parámetro *c* será igual a dos. El test será pasado con éxito si el valor calculado en (5.32) es menor que el valor correspondiente de la distribución chi-cuadrado $\left(\chi > \chi_{(\alpha,k-c-1)}\right)$.

La tabla 5.1 muestra los valores de la distribución chi-cuadrado para distintos grados de libertad y niveles significativos de α. El resultado obtenido será utilizado para calcular el valor-P. Si el valor-P es mayor que el nivel significativo, la hipótesis es aceptada y las muestras son aleatorias. Para el correcto funcionamiento del test, el número de muestras debe ser suficientemente grande y el número de muestras por categoría debe ser mayor que 5. El test se suele realizar al menos tres veces sobre diferentes conjuntos de datos.

5.4.2. Test Kolmogorov-Smirnov

Hemos visto que el test chi-cuadrado se aplica a situaciones en las que las observaciones pueden caer en un número finito de categorías. Esto no suele ser lo usual, sino que lo normal es considerar que las cantidades aleatorias pueden asumir infinitos valores. El test Kolmogorov-Smirnov (K-S test) es un test estadístico en el cual comparamos una muestra de valores observados con respecto a una distribución de probabilidad de referencia. La ventaja de este método es que permite trabajar con distribuciones continuas. Si queremos especificar la distribución de los valores de una cantidad X, podemos hacerlo en términos de su función de distribución *F(x)*. Si tenemos *n* observaciones independientes de una cantidad aleatoria X, con los valores X(*n*) (las muestras estarán ordenadas de menor a mayor), podemos obtener una función de distribución acumulada empírica ECDF*(x)* aplicando (5.34) y siendo N el numero de muestras.

$$F_n(x) = {X(n)}/{N} \tag{5.34}$$

El resultado del test K-S calcula la distancia entre la función de distribución empírica ECDF*(x)* de la muestra y la función de distribución acumulada de la distribución referencia CDF*(x)*. Para la realización del test K-S, calculamos las siguientes estadísticas:

$$K_N^+ = \sqrt{N} \max_{-\infty<x<+\infty} \left(F_n(x) - F(x)\right)$$
$$K_N^- = \sqrt{N} \max_{-\infty<x<+\infty} \left(F(x) - F_n(x)\right), \tag{5.35}$$

donde K_N^+ mide la mayor cantidad de desviación cuando ECDF*(x)* es mayor que CDF*(x)* y K_N^- mide la máxima desviación cuando ECDF*(x)* es menor que CDF*(x)*. Para un valor *x* fijo la desviación típica es proporcional a $1/\sqrt{N}$, de esta manera el termino $\sqrt{N}$ aumenta el valor de las estadísticas K-S de tal forma que esta desviación es independiente del numero de muestras N. En la figura 5.10 podemos ver la diferencia entre ECDF*(x)* y la CDF*(x)* para 500 números aleatorios.

k	α			
	0.1	**0.05**	**0.1**	**0.001**
1	2.7055	3.8415	6.6349	10.8276
2	4.6052	5.9915	9.2103	13.8155
3	6.2514	7.8147	11.3449	16.2662
4	7.7794	9.4877	13.2767	18.4668
5	9.2364	11.0705	15.0863	20.5150
6	10.6446	12.5916	16.8119	22.4577
7	12.0170	14.0671	18.4753	24.3219
8	13.3616	15.5073	20.0902	26.1245
9	14.6837	16.9190	21.6660	27.8772
10	15.9872	19.3070	23.2093	29.5883
11	17.2750	19.6751	24.7250	31.2641
12	18.5493	21.0261	26.2170	32.9095
13	19.8119	22.3620	27.6882	34.5282
14	21.0641	23.6848	29.1412	36.1233
15	22.3071	24.9958	30.5779	37.6973
16	23.5418	26.2962	31.9999	39.2524
17	24.7690	27.5871	33.4089	40.7902
18	25.9894	28.8693	34.8053	42.3124
19	27.2036	30.1435	36.1909	43.8202
20	28.4120	31.4104	37.5662	45.3147
21	29.6151	32.6706	38.9322	46.7970
22	30.8133	33.9244	40.2894	48.2679
23	32.0069	35.1725	41.6384	49.7282
24	33.1962	36.4150	42.9798	51.1786
25	34.3816	37.6525	44.3141	52.6197
26	35.5632	38.8851	45.6417	54.0520
27	36.7412	40.1133	46.9629	55.4760
28	37.9159	41.3371	48.2782	56.8923
29	39.0875	42.5570	49.5879	58.3012
30	40.2560	43.7730	50.8922	59.7031
31	41.4217	44.9853	52.1914	61.0983
63	77.7454	82.5287	92.0100	103.4424
127	147.8048	154.3015	166.9874	181.9930
255	284.3359	293.2478	310.4574	330.5197
511	552.3739	564.6961	588.2978	615.5149
1023	1081.3794	1098.5208	1131.1587	1168.4972

Tabla 5.1 - Valores calculados de la distribución χ^2 para los diferentes grados de libertad (k,α).

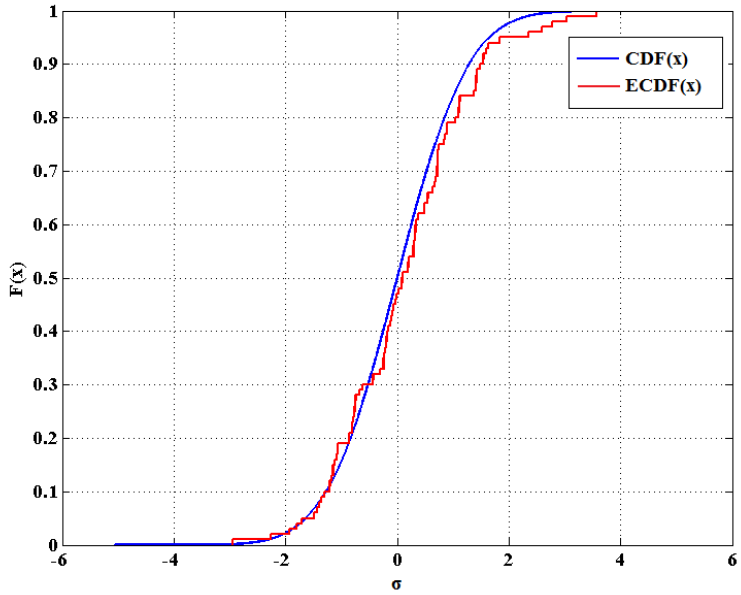


Figura 5.12 - Comparación entre la CDF empírica (ECDF) y la CDF ideal de la distribución Gaussiana..

Una ventaja de este test es que es un test exacto (la veracidad del test no depende del número de muestras utilizado, a diferencia del test chi-cuadrado). Por el contrario, el test K-S tiene serias limitaciones, ya que no permite comprobar correctamente las colas de las distribuciones y únicamente se puede utilizar en distribuciones continuas. Normalmente, se suele utilizar el test K-S en conjunto con el test Chi-cuadrado [79]. Debido a las limitaciones anteriormente comentadas se suele utilizar el test Anderson-Darling que veremos a continuación.

5.4.3. Test Anderson-Darling

El test Anderson-Darling (A-D) es otro tipo de test basado en la diferencia entre la PDF de una distribución y la EPDF. Este test está basado en la utilización de

$$\int_{-\infty}^{\infty} \left(EPDF(x) - PDF(x)\right)^2 w(x) dPDF(x). \tag{5.36}$$

Para el caso del test Anderson-Darling el valor de w(x) es igual a

$$w(x) = \frac{1}{PDF(x)(1 - PDF(x))}. \tag{5.37}$$

El valor en la diferencia calculada en el test A-D dependerá del tipo de distribución, de ahí que únicamente esté definido para unas distribuciones como pueden ser la normal o Gaussiana, lognormal, exponencial, Weibull y valor extremo tipo I. Debido a los valores utilizados en el denominador de (5.37), este test es mucho más preciso para el test de las colas de las distribuciones, además de ser mucho más potente que el test estándar Kolgomorov-Smirnov [80].

Para un numero N de muestras aleatorias, el Test A-D puede ser calculado como

$$A^2 = -N - S, \tag{5.38}$$

siendo S

$$S = \sum_{i=1}^{N} \frac{(2i-1)}{N} \left[\ln F\left(Y_i\right) + \ln\left(1 - F(Y_{N+1-i})\right) \right], \tag{5.39}$$

donde, F es la función de distribución acumulada de la distribución especificada e Y_i las muestras ordenadas de menor a mayor. A partir del valor A obtenido aplicaremos un ajuste que dependerá del numero de muestras

$$A'^2 = A^2 \left(1 + \frac{4}{N} - \frac{25}{N^2} \right) \tag{5.40}$$

Si el valor ajustado A'² supera el valor crítico, el cual dependerá de la confianza en el test ó nivel significativo y de la distribución, la hipótesis será rechazada. Para el caso de la distribución Gaussiana y un nivel α=0.05 este valor es igual a 0.751. En la tabla 5.2 podemos ver los valores críticos de la distribución normal para distintos valores de α.

α	**0.05**	**0.01**	**0.025**	**0.001**
A'²crit	0.751	0.632	0.870	1.029

Tabla 5.2 - Valores críticos del test Anderson-Darling para α=0.05.

5.4.4. Paquetes de Test

Por último, comentar que aparte de los métodos estadísticos que hemos visto en el presente capítulo, disponemos de una serie de test que verifican correctamente los generadores de números aleatorios uniformes. Entre los más importantes están, el test DIEHARD [81], NIST [82] y el TestU01 [83]. El TestU01 contiene todos los test realizados por DIEHARD y NIST. El problema de estos entornos de test es que únicamente pueden trabajar con enteros de hasta 32 bits. El código y el manual para el TestU01 está disponible en http://www.iro.umontreal.ca/~simardr/testu01/tu01.html. Estos paquetes cubren una amplia variedad de test que permiten verificar la aleatoriedad de un juego de muestras aleatorias.

5.5. Conclusiones

En este capítulo se han revisado los distintos métodos disponibles para la generación de variables aleatorias con una distribución Gaussiana. Entre las implementaciones más utilizadas podemos encontrar el método de Box-Muller, el método de la Inversión, el algoritmo de Ziggurat, etc. Todos estos algoritmos están basados en la aplicación de diferentes principios básicos matemáticos. Normalmente el punto de partida de estos métodos será la utilización de variables aleatorias uniformemente distribuidas sobre las que se aplican una serie de transformaciones. Para ello también se han estudiado los distintos generadores de números uniformemente distribuidos (GNUD). Por último, también se han descrito los diferentes test

estadísticos que nos van a permitir medir la "calidad" estadística de las muestras generadas por los diferentes métodos. Estos test serán necesarios ya que no podemos decidir por nosotros mismos si las muestras generadas son aleatorias.

Capítulo 6.

Generador de ruido AWGN de altas prestaciones basado en el método de la Inversión.

En este capítulo vamos a presentar varias arquitecturas de generadores hardware de ruido blanco Gaussiano con gran precisión en las colas de la función de distribución (desviación estándar de hasta ±13.1σ) y bajo coste Hardware. Los diseños utilizan el método de la Inversión, basado en la aproximación de la Inversa de la Función de Distribución Acumulada (ICDF) de la distribución Gaussiana. Los generadores se han implementado en diversas FPGAs de Xilinx como son Virtex-5, Virtex-4 y Virtex-2.

Las arquitecturas se componen de tres grandes bloques. Primero se ha diseñado un generador de números aleatorios uniformemente distribuidos (GNUD). A partir de estos valores se han generado las muestras aleatorias Gaussianas por medio de la aproximación de la ICDF de la distribución Gaussiana, empleando un esquema de segmentación no-uniforme e interpolación polinómica en cada segmento. La segmentación no-uniforme es básica en la aproximación de la ICDF, pues reduce significativamente los recursos hardware necesarios cuando se desean alcanzar altos valores de desviación estándar de la Gaussiana. Para verificar el correcto comportamiento aleatorio del generador se han realizado simulaciones y tests estadísticos (χ2 y Anderson-Darling).

6.1. Introducción

Para la generación hardware de muestras aleatorias Gaussianas son preferidos los métodos digitales con respecto a los métodos analógicos. Los métodos analógicos permiten generar números aleatorios "reales" a diferencia de los métodos digitales, pero son muy sensibles a cambios de temperatura. Además proporcionan bajas tasas de muestras aleatorias. Por el contrario, los métodos digitales son más utilizados debido a su velocidad, flexibilidad y posibilidad de poder repetir la medida. Estos métodos generan secuencias pseudoaleatorias, a diferencia de las implementaciones analógicas. Un generador pseudoaleatorio, es un algoritmo determinista[1], el cual a partir de una secuencia binaria aleatoria de longitud *k,* siempre generará una secuencia binaria de longitud *l*>>*k*, la cual parece ser aleatoria. El gran problema de las implementaciones digitales es que el periodo de los generadores debe ser lo suficientemente largo para que las secuencias no se repitan durante las simulaciones.

Para la implementación de un canal AWGN en hardware podemos utilizar diferentes métodos: Box-Muller [84 - 87], Teorema del Limite Central (CLT) [88][89], inversión de la función acumulada de probabilidad (ICDF) [90 - 92], Polar [93], etc. De todos, el método más conocido para la generación de ruido Gaussiano es el método de Box-Muller.

- El método Box-Muller (B-M) ha sido ampliamente utilizado en implementaciones hardware. Este método se basa en la transformación de dos números uniformemente distribuidos U(0-1) mediante la aplicación de diferentes funciones elementales [61]. Normalmente el método Box-Muller es implementado añadiendo una etapa CLT, permitiendo de esta manera aumentar la calidad de las muestras, además de corregir los posibles errores cometidos en la aproximación de las funciones elementales utilizadas. El primer autor en implementar el método Box-Muller más una etapa de CLT fue Boutillon [84]. Esta implementación tiene como inconveniente una amplitud en las muestras de ruido generado menor que $\pm 4\sigma$, además de una baja calidad de las muestras generadas (no pasan el test χ^2). Xilinx ha desarrollado un core-IP [94] basado en la arquitectura propuesta por [84]. En [85] proponen la implementación del método B-M más una etapa CLT y utilizando interpolación lineal y técnicas de segmentación no-uniforme de las funciones elementales utilizadas (logaritmo neperiano y raíz cuadrada), permitiendo aumentar la calidad de las muestras generadas y reduciendo los requerimientos de memoria necesarios. Los valores de sigma (σ) obtenidos para esta implementación llegan hasta $\pm 6.7\sigma$. Una posterior implementación realizada por los mismos autores [86] permite generar muestras hasta un valor de $\pm 8.2\sigma$ con un tamaño de palabra de 16 bits y una precisión de las muestras generadas de 11 bits. Gracias a un exhaustivo análisis del error y del tamaño de los operadores utilizados, no es necesario el uso de la etapa CLT. Este generador produce dos muestras por ciclo trabajando a frecuencia de 375 MHz y utilizando una FPGA Virtex-4 de Xilinx. Otra implementación basada en B-M ha sido propuesta por [87] y consiste en el uso técnicas de segmentación no-uniforme hibridas (logarítmica y uniforme) sobre las funciones elementales y obtienen una amplitud en las muestras de ruido generado de hasta $\pm 9.4\sigma$, con unos requerimientos de área relativamente bajos.
- Método CLT. Tradicionalmente, este método se implementa utilizando un acumulador. Implementaciones hardware de este método la podemos encontrar en [88] y utiliza cuatro generadores de números uniformemente distribuidos, tres sumadores y un acumulador. Los

[1] Determinista implicará que para un determinado valor inicial (semilla) el generador siempre producirá la misma secuencia de salida.

generadores de valores uniformes están implementados mediante la utilización de registros LFSR (Lineal Feedback Shift Registers) de longitudes 28, 29, 30 y 31. Los últimos 10 bits de cada registro son interpretados como enteros en complemento a dos y el circuito produce una salida válida cada 12 ciclos de reloj por medio de la suma de 48 números aleatorios de 10 bits. Otra implementación propuesta por Andraka y Phelps [89], realiza la suma de 128 variables de un bit, esta aproximación puede ser buena si la precisión de la parte fraccional no necesitamos que sea alta.

- El método inversión de la CDF (ICDF) está basado en la generación de variables Gaussianas mediante la transformación de variables aleatorias uniformemente distribuidas $x=[0,1)$ mediante la aplicación de la función de distribución acumulada de una distribución Gaussiana de la forma $y=CDF^{-1}(x)$. La gran ventaja de este método es que permite generar variables aleatorias con CDF arbitrarias [60]. El mayor problema para la implementación de este método es la gran complejidad en la aproximación de la ICDF de la distribución Gaussiana debido a las altas no-linealidades presentes en los extremos de la misma. Será muy importante generar correctamente dicha función en los extremos, ya que esos valores serán los utilizados en la generación de la cola de la Gaussiana. Chen [90] proponen la aproximación de la ICDF Gaussiana mediante la utilización de una LUT. Este método tiene el inconveniente de que se necesitan grandes cantidades de memoria si queremos generar con gran precisión la cola de la Gaussiana. Para una implementación con 16bits de entrada y 16 bits de salida sería necesaria una memoria de 1 MByte. Mcollum [91] propone el uso de interpolación lineal para la aproximación de la ICDF. De esta manera se reduce el tamaño de las memorias necesarias. En este caso únicamente se utiliza una tabla de 262 KBytes Por último, Cheung [92] propone una implementación más eficiente de la ICDF, en términos de área y amplitud de las muestras de ruido generado llegando hasta $\pm 8.2\sigma$. Para ello utilizan técnicas de segmentación no uniforme e interpolación lineal.

- Los métodos aceptación-rechazo "*rejection-acceptance*", como pueden ser el método Polar [62] y el algoritmo Ziggurat [63], son ampliamente utilizados sobre todo en aplicaciones software (Matlab), debido a la facilidad de su implementación utilizando instrucciones condicionales (`if-then-else`). El gran inconveniente es que la tasa de muestras aleatorias generadas no es constante, debido precisamente a ese funcionamiento condicional. Y. Fan y Z. Zilic [93] proponen una arquitectura basada en la implementación del método Polar y añadiendo una etapa final CLT que permite mejorar la calidad de las muestras generadas. Además propone una serie de arquitecturas para poder evaluar el BER en hardware. Para evitar el problema de la generación con tasa no-constante de muestras, los autores proponen la utilización de una memoria FIFO con una velocidad de lectura que es la mitad de la velocidad de escritura. El algoritmo Ziggurat ha sido implementado por [95] y proponen una etapa de test que permite evaluar la "calidad" de las muestras aleatorias generadas en hardware (χ^2). Estos dos métodos están basados en la aproximación de funciones elementales mediante interpolación polinómica. Las funciones aproximadas son, el logaritmo neperiano y la exponencial.

- El método Wallace, a diferencia del resto de métodos, no requiere de la evaluación de ninguna función elemental ya que las muestras son generadas mediante la aplicación de transformaciones lineales sobre una matriz de muestras Gaussianas previamente calculadas. Debido a la realimentación utilizada por este método pueden aparecer correlaciones inesperadas entre transformaciones sucesivas. Una implementación hardware basada en este método ha sido propuesta por [96], en la cual se presentan diferentes parámetros que minimizan estos efectos de correlación. Esta implementación genera valores de hasta $\pm 7\sigma$.

Para elegir qué arquitectura es la más apropiada deberemos establecer un compromiso entre la simplicidad del algoritmo implementado, el error cometido en la aproximación y la eficiencia (área de la FPGA, latencia y tasas de valores generados). Por ejemplo, la tasa de valores aleatorios generados será la mitad en una implementación basada en inversión CDF con respecto a una implementación Box-Muller. Wallace permite un hardware simple pero tiene el problema de cadencia de datos no constante.

6.2. Implementación de la Aproximación de la Inversa de la Función de Distribución Acumulada (ICDF)

A continuación vamos a exponer la propuesta para el generador de muestras aleatorias Gaussianas basado en el método de la inversión de la CDF de la distribución Gaussiana. Como hemos visto del capítulo anterior, el método ICDF está basado en la transformación de una variable aleatoria uniformemente distribuida en una variable aleatoria Gaussiana aplicando la inversa de la función de distribución acumulada de una Gaussiana. La figura 6.1 muestra este proceso.

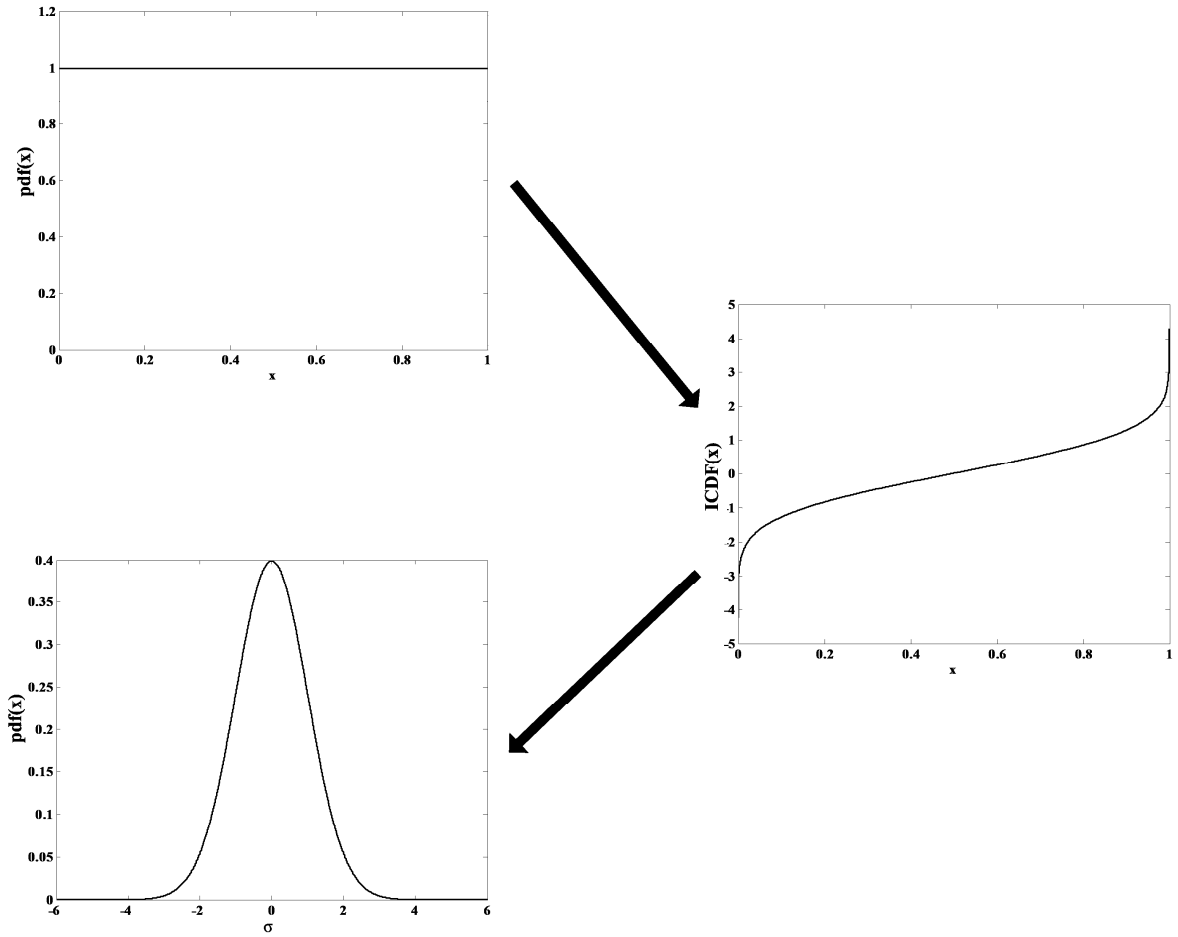


Figura 6.1 - Generación de muestras aleatorias Gaussianas mediante el método de la inversión.

El generador de variables aleatorias Gaussianas estará compuesto por dos grandes bloques:

- **Generador de Números Uniformemente Distribuidos (GNUD)**. Este bloque será el encargado de generar las variables aleatorias uniformemente distribuidas que utilizaremos como entrada de la aproximación de ICDF.
- **Aproximación ICDF**. El diseño de la aproximación de la ICDF de una Gaussiana estará basado en la utilización de una interpolación polinómica mediante un esquema de segmentación no uniforme, que se ha aplicado debido a las altas no-linealidades que tenemos en los extremos de la función. Utilizaremos un mayor número de segmentos en los extremos de la función que en la parte central. De esta manera podemos reducir el número de segmentos necesarios manteniendo una muy buena precisión en las colas de la Gaussiana.

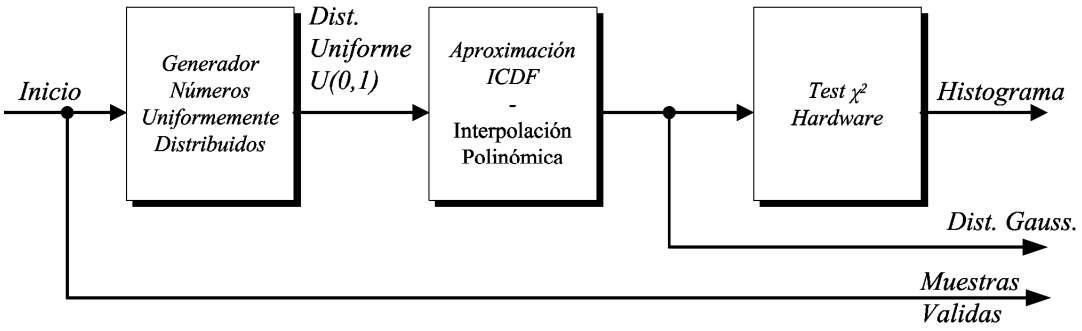


Figura 6.2 – Esquema de implementación y validación del método de inversión propuesto.

En la figura 6.2 podemos ver un esquema de la implementación y verificación basado en la realización del método de la ICDF. Dentro de las líneas futuras de la presente tesis doctoral a la implementación propuesta se podría añadir una etapa para la realización del test χ^2 en hardware. Este bloque estaría compuesto de una Block-RAM y de una pequeña maquina de estados para el control del llenado de la memoria. Los valores de inicio de la Block-RAM serían cero. Los MSB del GNAG direccionarán una posición de memoria de la Block-RAM de doble puerto y el valor almacenado se incrementará. De esta manera obtendríamos cuántas veces se han producido los eventos y por consiguiente obtendríamos el histograma de las muestras aleatorias generadas. El otro puerto de la Block-RAM podría ser utilizado para acceder al histograma de las muestras sin interrumpir el funcionamiento del GNAG. Como podemos apreciar de la figura, este GNAG se inicializará por medio de una señal *Inicio* y a partir de ese momento el GNUD comenzará a generar valores aleatorios partiendo de un valor inicial (semilla) previamente definido. El tamaño del GNUD vendrá determinado por el valor máximo de la desviación estándar de la distribución Gaussiana. El objetivo inicial era realizar un GNAG con una desviación de 10σ y aplicando la ecuación (6.1) obtenemos que la variable uniforme aleatoria necesaria debe tener un tamaño de 78bits ($x > 1\text{-}2^{-78}$). Con este tamaño son necesarios dos GNUD de 64 bits. Con el fin de optimizar el área de la implementación se tomó la decisión de utilizar todos los bits de los dos generadores y el nuevo valor de desviación que podemos obtener con 128 bits es de $\Phi \geq 13$.

$$\Phi^{-1} = \sqrt{2} erf^{-1}(2x-1) \tag{6.1}$$

Un elemento muy importante que deberemos tener en cuenta es que para llegar a un valor de $\approx \pm 13\sigma$ en nuestro generador AWGN, la pendiente de la función en esos puntos será del orden 10^{38}, siendo inviable la aproximación lineal directa debido al tamaño excesivo de los multiplicadores/sumadores necesarios. Para evitar este problema se ha optado por realizar un cambio de variable en cada segmento de

forma que tanto la entrada como los coeficientes de la aproximación tienen un formato manejable. Esta técnica de escalado nos permitirá reducir el tamaño en bits de la entrada de 128 bits a tan solo 18 bits. Se ha elegido el tamaño de 18 bits buscando una implementación eficiente (es el tamaño de los multiplicadores embebidos disponibles en FPGAs de Xilinx, bloques MULT18X18S y DPS48).

A partir de los valores generados en el GNUD y mediante un bloque de lógica adicional identificaremos el segmento y calcularemos la dirección de memoria de los coeficientes necesarios para la realizar la aproximación a la función en ese segmento. Este bloque posee cierta complejidad debido a la utilización de una segmentación no-uniforme de la ICDF. Por último, a partir de los coeficientes direccionados y del resto de bits procedentes del GNUD realizaremos la interpolación polinómica.

6.2.1. Segmentación No-uniforme de la ICDF

Para implementar la arquitectura propuesta tenemos que calcular la inversa de la función de probabilidad acumulada de la distribución Gaussiana. Además necesitamos la correcta aproximación de las colas de la ICDF ya que es en la cola donde se darán los eventos aleatorios necesarios para generar errores para altos valores de SNR. Este funcionamiento será muy importante para realizar simulaciones en sistemas de muy bajo BER como es el caso de los sistemas de comunicaciones ópticos. Para realizar la aproximación de dicha función se han barajado diferentes métodos de aproximación de funciones elementales, de los cuales se han descartado las aproximaciones basadas en LUTs debido al excesivo tamaño de las memorias necesarias para poder realizar la aproximación de las colas de la ICDF para valores altos de desviación y con un bajo error.

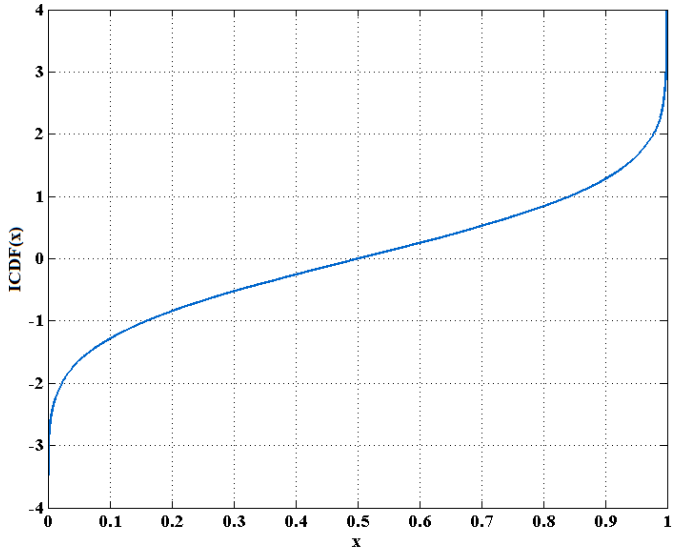


Figura 6.3 – Inversa de la Función Distribución Acumulada de la distribución Gaussiana.

Para aproximar la ICDF hemos utilizado un esquema de interpolación polinómica dentro del rango de entrada [0 – 1). La figura 6.3 muestra la inversa de la Función de Distribución Acumulada de la distribución Gaussiana. En esta explicación nos centraremos en el uso de la interpolación polinómica cuadrática. Se ha optado por utilizar polinomios de grado dos, debido a que proporcionan un número de segmentos bajo con una latencia del operador no muy alta. Para un polinomio de grado dos únicamente es necesario realizar dos multiplicaciones si reordenamos el polinomio de interpolación aplicando la regla de Horner. Si utilizamos otros polinomios de distinto grado, el esquema de segmentación será el mismo y lo

único que aumentará/disminuirá será el número de segmentos necesarios si disminuimos/aumentamos el grado del polinomio con respecto a la interpolación de grado dos. Para poder aproximar la función con la precisión pedida se ha dividido el rango de la función en segmentos de tamaño cada vez más reducido y en cada segmento se ha utilizado un polinomio. Si el número de segmentos utilizados es pequeño, repercutirá en la necesidad de polinomios de un grado alto, aumentando la complejidad del diseño. Por el contrario, si el número de segmentos es muy grande podemos utilizar una interpolación lineal simplificando la realización de la aproximación, pero la cantidad de memoria necesaria para almacenar los coeficientes puede ser tan alta que haga imposible su realización. Es por ello que será muy importante la elección del esquema de segmentación de la aproximación de la ICDF.

Debido a las altas no-linealidades presentes en los extremos de la ICDF (regiones cercanas a 0 y a 1) hemos utilizado un esquema de segmentación no-uniforme como el comentado en el capítulo 4 aunque con algunas diferencias que se detallan más adelante. Como podemos apreciar de la figura 6.4 la pendiente en la región central de la función tiene un comportamiento bastante lineal, implicando un pequeño número de segmentos necesarios para su aproximación. Por el contrario en los extremos de la función será necesario aumentar significativamente el número de segmentos para poder aproximar ésta con suficiente precisión. Además, en esos extremos de la función es importante realizar una buena aproximación ya que es ahí donde se producen los eventos aleatorios de interés en las simulaciones de elevado SNR. En la figura 6.5 podemos ver una representación del esquema de segmentación no-uniforme utilizado.

El esquema propuesto de segmentación está basado en la utilización de segmentación no-uniforme. El rango para el cual la función está definida será dividido en dos partes:

$$s = \begin{cases} s_1 \in [0 - 0.5) \\ s_2 \in [0.5 - 1) \end{cases} \tag{6.2}$$

Como podemos apreciar de la figura 6.4, los valores de $y = ICDF(x)$ de la región [0, 0.5) (a esta región la denominaremos región s_1), son los mismos que los valores de la región [0.5, 1) (a esta región la denominaremos región s_2), exceptuando el cambio de signo experimentado. Por tanto únicamente será necesario aproximar la función en la región s_2 (parte positiva) y complementando los valores de la región s_2 generaremos el resto de valores de la ICDF original. El paso siguiente será realizar la segmentación de la región s_2. El número de segmentos necesarios para realizar la aproximación dependerá de la precisión deseada (parte fraccionaria) y del valor máximo de desviación estándar de la distribución Gaussiana (parte entera) de las muestras generadas. En la figura 6.5 podemos ver la relación del número de bits del generador de muestras uniformes y el valor de la desviación obtenido en la distribución Gaussiana. Podemos apreciar cómo para generar valores de sigma superiores a 10 el tamaño de la palabra generada por el GNUD es considerable. Estos valores los hemos obtenido aplicando (6.1).

Para la identificación de los segmentos utilizaremos la referencia $s_{r,n}$, siendo r una de las dos regiones comentadas anteriormente y n el sub-segmento. Para cada segmento que cumpla la condición de error máximo permitido, calcularemos los coeficientes de polinomios de Chebyshev de primer tipo (T_n). Los polinomios de Chebyshev son una familia de polinomios ortogonales que permiten minimizar el error medio cometido en la aproximación [14].

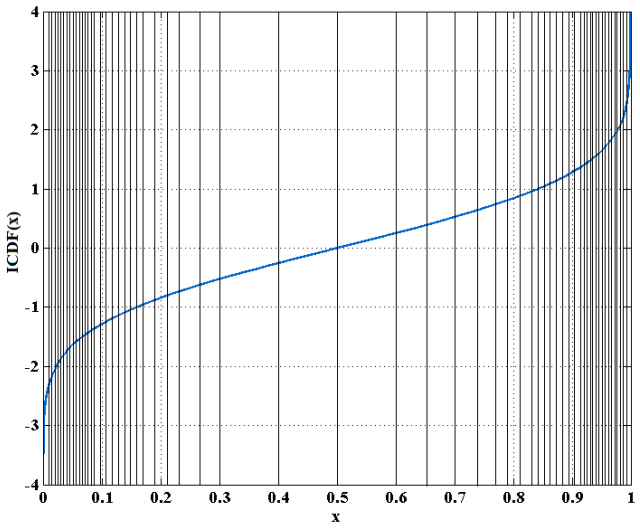


Figura 6.4 - Segmentación no-uniforme utilizada para la aproximación de la ICDF de la distribución Gaussiana.

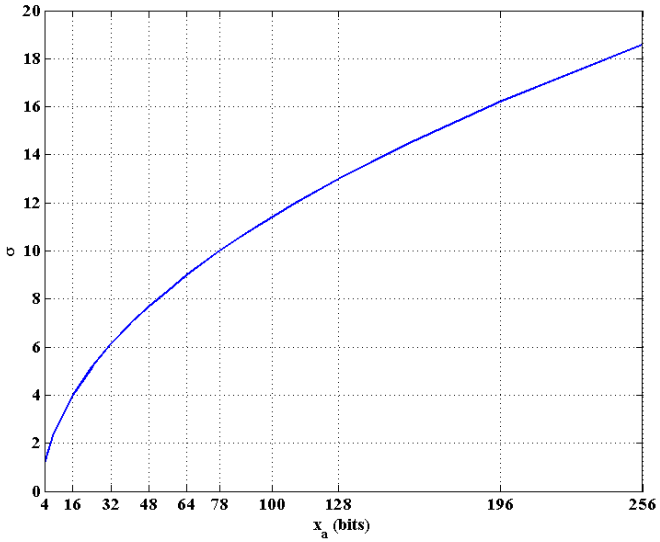


Figura 6.5 - Relación entre la desviación máxima (σ) de la distribución Gaussiana y el número de bits necesarios en el GNUD.

Estos polinomios están definidos mediante la relación de recurrencia:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_n(x) &= 2xT_{n-1}(x) - T_{n-2}(x) = \cos(n \cdot \arccos(x)). \end{aligned} \tag{6.3}$$

La segmentación no-uniforme estará basada en un esquema de error balanceado (6.4). En este esquema, el error cometido en todos los segmentos será siempre el mismo. Esto lo conseguiremos modificando el tamaño del segmento.

$$\varepsilon_{aprox} = \max_{s_{2,n} \le x \le s_{2,n+1}} \left| f(x) - p_d(x) \right| \tag{6.4}$$

Partiendo de las premisas comentadas anteriormente, nuestro esquema de segmentación tendrá como parámetros de entrada, la función continua a aproximar (en nuestro caso la ICDF de la distribución Gaussiana pero podemos pasar cualquier otra función elemental), el rango para el cual la función está definida [a, b), el grado de los polinomios utilizados (*d*) y el error deseado $\varepsilon_{deseado}$ para cada segmento. La función calculará los limites de cada segmento $s_{2,n}$ $(a = s_{2,0} \le s_{2,1} \le s_{2,2} \le \ldots\ldots \le s_{2,T-1} \le b)$ y los coeficientes del polinomio de grado *d* utilizados en la interpolación polinómica, cumpliendo en todo momento que el error del polinomio calculado sea menor que el error definido en el parámetro de entrada de la función. En la figura 6.6 podemos ver el algoritmo implementado empleando pseudocódigo. El algoritmo implementado se divide en dos partes bien diferenciadas. La primera parte calculará los limites de los segmentos necesarios para la aproximación (líneas 10 – 11). Para cada segmento calcularemos los coeficientes del polinomio de Chebyshev de grado d. Para la aproximación de la función ICDF únicamente vamos a aproximar la parte positiva, por lo cual el rango inicial de la aproximación será $s_2 \in \left[0.5,\ 0.5+2^{-m}\right)$, con m=1. Si el ε_{aprox} es mayor que el error deseado en el segmento ($\varepsilon_{deseado}$), saltaremos a las líneas 41-44 para reducir el tamaño del segmento (aumentando el valor de *m*) y volveremos a calcular el polinomio para el nuevo segmento de menor tamaño. Iremos reduciendo el tamaño del segmento hasta que el error $\varepsilon_{aprox.}$ sea inferior al error $\varepsilon_{deseado}$. La segunda parte del algoritmo (líneas 13-40), serán las encargadas de generar los valores utilizados para el direccionamiento mediante una tabla de traducción de direcciones (*ROM_trans*) y de generar el escalado de los coeficientes calculados. Hay que resaltar que conforme vamos llegando a valores de entrada cercanos a uno, la pendiente de la ICDF tenderá a infinito y por consiguiente, los coeficientes de Chebyshev calculados tenderían a valores excesivamente grandes, $c_x \in \left(0,\ 10^{38}\right)$. Esto imposibilitaría la realización de las operaciones aritméticas debido a las altas necesidades de almacenamiento y al gran tamaño de los operadores aritméticos necesarios. Para evitar este problema, en [85] se propone la realización de un escalado dinámico sobre los coeficientes. De esta manera podemos realizar las operaciones aritméticas sin una perdida apreciable de precisión. Para ello se almacenan tanto los coeficientes escalados como el valor de escalado (estos factores serán potencias de dos) y en tiempo de ejecución se incrementa/decrementa el valor del coeficiente dependiendo del valor del escalado aplicado. Normalmente los valores de los nuevos coeficientes escalados estarán comprendidos dentro del rango [0, 1) y al utilizar potencias de dos como valores de escalado, éstos se realizaran mediante desplazamientos, reduciendo el hardware necesario.

En nuestra arquitectura realizamos un cambio de variable en la función a aproximar que reducirá la magnitud de los coeficientes necesarios para la aproximación. La tarea del escalado la realizamos mediante la función `func_coef` (línea 26). De esta manera reducimos la cantidad de memoria necesaria manteniendo la precisión de los cálculos realizados. Para explicar el cambio de variable realizado vamos a basarnos en un polinomio de segundo grado (6.5), aunque el concepto es aplicable a cualquier otro grado de polinomio.

$$y = c_2 \cdot x^2 + c_1 \cdot x + c_0 \tag{6.5}$$

El valor de x lo obtendremos del GNUD, y está comprendido en [0, 1), con un tamaño de palabra de n=128 bits. La entrada x de *n-bits* la dividimos en tres partes, x_a, x_o y x_t con tamaños *Bxa*, *Bxo* y *Bxt* bits, respectivamente. Los tamaños son variables. x_a comprende todos los bits a 1 de mayor peso de la palabra, incluido el 0 de mayor peso (cero más significativo o CMS). x_o son los bits que le faltan a x_a para completar la secuencia de bits que identifica cada segmento y los denominaremos bits de *offset*. El *Bxo* adecuado según se ha obtenido experimentalmente es 1 ó 2, dependiendo del segmento (es 1 en los segmentos más cercanos al extremo). x_t serán el resto de bits. Utilizaremos los *Bxa*+*Bxo* bits más

significativos de la entrada x para determinar el segmento y así acceder a los coeficientes precalculados para él. Como el valor de x_a y x_o está implícitamente dado por el segmento de que se trate, únicamente será necesario x_l (en vez del de x) para la aproximación del polinomio en cada segmento. Tras el cambio de variable x_l estará comprendido dentro del rango [0, 1).

Algoritmo Segmentación no-uniforme ICDF

```
1: -- Parámetros entrada: f, rango entrada [a, b), tamaño bits
2:-- coeficientes ppg, grado polinomio d, error segmento (ε_deseado),
3:-- valor sigma v_σ
5: seg_ini=0.5; seg_fin=1; -- Valor inicio y fin región
6: LZD=0;LZD_ant=0 -- Valor posición del "cero" MSB.
7: n=0; m=1; -- Contador segmentos (n), (m) tamaño segmento
8: fin=0 – hasta que no segmentemos todo el rango no terminamos
9: while fin==0 :
10:  segmento=[seg_ini seg_fin];
11:  coef,ε_aprox=cheby_aprox(f, segmento, d);
12:  if ε_aprox < ε_deseado :
13:    seg_ini_bin = conv_integer_bin(seg_ini);
14:    seg_fin_bin = conv_integer_bin(seg_fin);
15:    LZD= Leadding_zero_detector(seg_ini_bin,seg_ini_fin);
16:      if LZD |= LZD_ant :
17:        ROM_trans = n;
18:   LZD_ant=LZD
19:   seg_ini_esc = seg_ini_bin << LZD;
20:   seg_fin_esc = seg_ini_bin << LZD;
21:   offset_bits=calc_offset(seg_ini_esc,seg_fin_esc);
22:   escalado=LZD + offset_bits;
23:   seg_ini_esc =round(seg_ini_esc << offset_bits,ppg);
24:   seg_fin_esc =round(seg_fin_esc << offset_bits,ppg);
25:   coef_esc=func_coef_esc(coef);
26:   ROM_coef=round(coef_esc,ppg);
27:   s_ini=polyeval(seg_ini_esc,ROM_coef);
28:   s_fin=polyeval(seg_fin_esc,ROM_coef);
29:   ε_ini=abs(ICDF(seg_ini)-s_ini);
30:   ε_fin=abs(ICDF(seg_fin)-s_fin);
31:   ε_med=(ε_ini +ε_fin)/2;
32:     if ε_med > ε_deseado :
33:       salta 42;
34:     else:
35:       seg_ini=seg_fin;
36:       seg_fin=seg_fin+cambio;
37:       n=n+1;
38:   if s_fin > v_σ :
39:     fin=1;
40:      esc_ROM_vhdl(ROM_coef,ROM_trans)
41:  else:
42:    m=m+1;
43:    cambio=2**-m
44:    seg_fin=seg_fin-cambio; -- Reducimos el segmento
```

Figura 6.6 - Pseudocódigo del algoritmo de cálculo de la segmentación no-uniforme de la función ICDF.

Solo los 17 bits más significativos de x_t se emplearán en los cálculos (parámetro *ppg* de nuestra función) pues éste es el tamaño máximo sin signo de las entradas de los multiplicadores embebidos disponibles en las FPGAs de Xilinx. Como mínimo *Bxt* será 1. El cambio de variable consiste en eliminar los *Bxa+Bxo* bits más significativos de *x* (desplazar hacia la izquierda *Bxa+Bxo* posiciones y desechar los bits enteros, según la ecuación (6.8). Gracias al cambio de variable los coeficientes se escalan de forma que su magnitud se reduce, según la ecuación (6.10). Por ejemplo, para unos requerimientos de desviación de $\pm 13.1\sigma$, un error máximo por segmento de 2^{-13}, un polinomio interpolador de segundo grado y un *Bxo* de tamaño variable son necesarios 298 segmentos. Con esas mismas especificaciones y un *Bxo* siempre fijo de dos bits necesitamos 510 segmentos. Utilizar un *Bxo* variable nos permite por tanto reducir el número de segmentos necesarios para la aproximación. Como podemos apreciar la reducción en el número de segmentos necesarios es bastante significativa y únicamente es necesario añadir al sistema un comparador para decidir si tomamos dos bits o un bit de *offset* en cada instante. La figura 6.7 muestra un ejemplo del esquema de direccionamiento implementado. Todas estas tareas las realizamos en las líneas 13-24 de nuestro pseudocódigo.

		GNUD				
		CMS	offset	x_t	$X_{[105:0]}$	
Segmento 1 [0.5 – 0.565)	Seg_ini	10	00	000000000000000000	000000……...00000	CMS=2 B_{offset}=2 Escalado=2^4
	Seg_ini	10	00	111111111111111111	111111……...11111	
Segmento 2 [0.565 – 0.625)	Seg_ini	10	01	000000000000000000	000000……...00000	CMS=2 B_{offset}=2 Escalado=2^4
	Seg_ini	10	01	111111111111111111	111111……...11111	
		GNUD				
		CMS	offset	x_t	$X_{[104:0]}$	
Segmento 5 [0.78125 – 0.8125)	Seg_ini	110	01	000000000000000000	000000……...00000	CMS=3 B_{offset}=2 Escalado=2^5
	Seg_ini	110	01	111111111111111111	111111……...11111	
Segmento 6 [0.8125 – 0.84375)	Seg_ini	110	10	000000000000000000	000000……...00000	CMS=3 B_{offset}=2 Escalado=2^5
	Seg_ini	110	10	111111111111111111	111111……...11111	
		GNUD				
		CMS	offset	x_t	$X_{[100:0]}$	
Segmento 12 [0.9531 – 0.96093)	Seg_ini	11110	10	000000000000000000	000000……...00000	CMS=5 B_{offset}=2 Escalado=2^7
	Seg_ini	11110	10	111111111111111111	111111……...11111	
Segmento 13 [0.96093 – 0.96874)	Seg_ini	11110	11	000000000000000000	000000……...00000	CMS=5 B_{offset}=2 Escalado=2^7
	Seg_ini	11110	11	111111111111111111	111111……...11111	
		GNUD				
		CMS	offset	x_t	$X_{[94:0]}$	
Segmento 33 [0.999633 – 0.999694)	Seg_ini	111111111110	10	000000000000000000	00……...000	CMS=12 B_{offset}=2 Escalado=2^{14}
	Seg_ini	111111111110	10	111111111111111111	11……...111	
Segmento 34 [0.999694 – 0.999755)	Seg_ini	111111111110	11	000000000000000000	00……...000	CMS=12 B_{offset}=2 Escalado=2^{14}
	Seg_ini	111111111110	11	111111111111111111	11……...111	

Figura 6.7 – Esquema de direccionamiento utilizado en la aproximación de la función ICDF.

El cambio de variable que nos permite usar x_t en vez de x implica una transformación en los coeficientes obtenidos de la aproximación de Chebyshev (línea 11), tal y como se detalla a continuación. Partimos del polinomio de grado dos y utilizando la nueva variable x_t reescribimos (6.5)

$$y = c_2^{'} \cdot x_t^2 + c_1^{'} \cdot x_t + c_0^{'}, \tag{6.6}$$

y para cualquier segmento dado, tendremos que

$$x_t = (x - x_a - x_o) \cdot 2^{(Bxa+Bxo)}, \tag{6.7}$$

y reordenando (6.7) obtenemos

$$x = \frac{x_t}{2^{(Bxa+Bxo)}} + (x_a + x_o). \tag{6.8}$$

Substituyendo (6.8) en (6.6)

$$y = \frac{c_2}{2^{2(Bxa+Bxo)}} \cdot x_t^2 + \frac{c_1 + 2 \cdot c_2 \cdot (x_a + x_o)}{2^{(Bxa+Bxo)}} \cdot x_t + c_2 \cdot (x_a + x_o) + c_1 \cdot (x_a + x_o) + c_0. \tag{6.9}$$

Por inspección de (6.6) y (6.9) los nuevos coeficientes escalados serán:

$$\begin{aligned} c_2^{'} &= \frac{c_2}{2^{2(Bxa+Bxo)}} \\ c_1^{'} &= \frac{c_1 + 2 \cdot c_2 \cdot (x_a + x_o)}{2^{(Bxa+Bxo)}} \\ c_0^{'} &= c_2 \cdot (x_a + x_o) + c_1 \cdot (x_a + x_o) + c_0 \end{aligned} \tag{6.10}$$

Todas estas operaciones las hemos agrupado dentro de la función `func_coef_esc` en la línea 25 del pseudocódigo. En la línea 26 redondeamos los coeficientes a `ppg` bits mediante el siguiente esquema de redondeado

$$round(c_n^{'}, ppg) = \left\lfloor c_n^{'} \cdot 2^{ppg} + 0.5 \right\rfloor \cdot 2^{-ppg} \tag{6.11}$$

Por último, nuestro algoritmo reconstruye la aproximación de la ICDF (función `polyeval` línea 27) con los coeficientes escalados y redondeados a `ppg` bits y utilizando x_t en vez de x y calculando el error cometido por la aproximación. Si el error ε_{recons} es menor que $\varepsilon_{deseado}$, aceptamos los coeficientes y pasamos al siguiente segmento, si no se descartan los coeficientes y se reduce el tamaño del segmento. Finalmente comprobamos si hemos llegado al valor de sigma v_σ pedido y si es así finalizamos el bucle `while` y llamamos a una función `esc_rom_vhdl` que genera los ficheros vhdl que modelan el comportamiento de la memoria *ROM_coef* y *ROM_trans* utilizados en nuestra implementación. En la figura 6.8 podemos ver el error cometido en la segmentación de la función ICDF para un error $\varepsilon_{deseado}=2^{-13}$ (1 *ulp*), en terminos de *ulp*. Podemos apreciar como nunca superamos ese valor. La figura 6.9 muestra un esquema de la arquitectura propuesta para una interpolación de segundo grado. Los polinomios calculados han sido implementados aplicando la regla de Horner, permitiéndonos reducir el número de operaciones aritméticas necesarias.

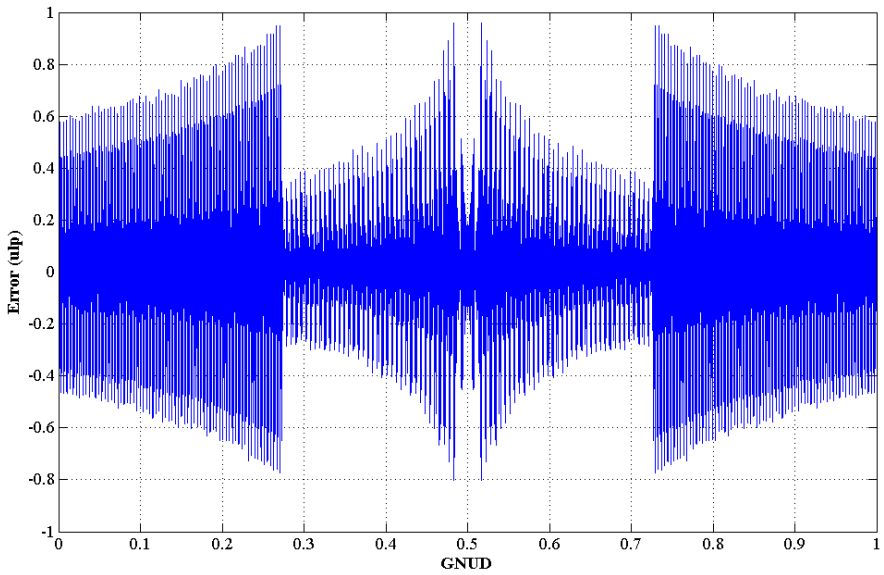


Figura 6.8 - Error cometido (*ulp*) en la aproximación no-uniforme de la función ICDF.

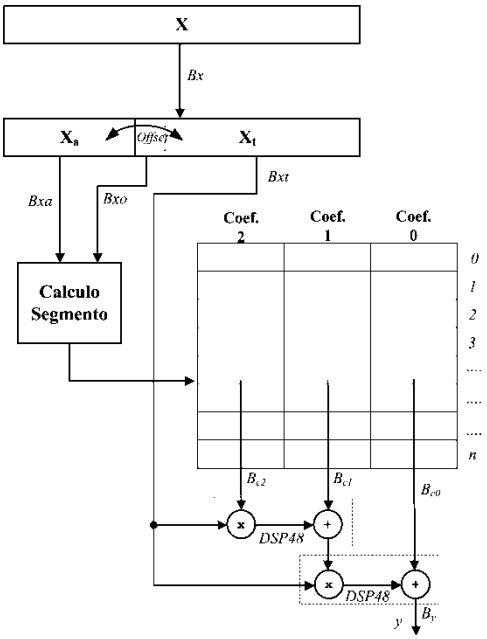


Figura 6.9 – Arquitectura propuesta para la aproximación de la función ICDF mediante segmentación no-uniforme y con polinomio interpolador de grado dos.

En la figura 6.10 podemos ver el número de segmentos necesarios para la aproximación de la ICDF para diversos valores de precisión de las muestras Gaussianas generadas y un valor fijo de desviación de la distribución Gaussiana de $\pm 10\sigma$. Como podemos observar el número de segmentos necesarios para la utilización de una interpolación lineal es un orden de magnitud superior para precisiones a partir de 14 bits, respecto a las interpolaciones cuadráticas y grado 3. Con lo cual únicamente será interesante esta interpolación para bajas precisiones (8 y 10 bits). Para altas precisiones será más recomendable utilizar interpolaciones cuadráticas o de grado 3. La aproximación grado 3 será interesante para altas precisiones (20 bits) ya que reduce aproximadamente a un tercio el número de segmentos necesarios a costa de aumentar el número de multiplicadores utilizados.

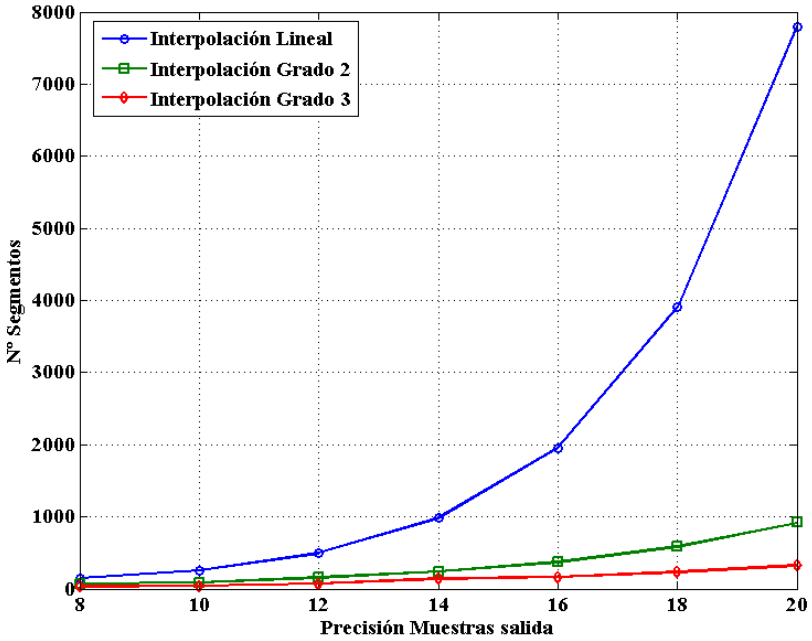


Figura 6.10 – Variación del numero de segmentos para diferentes precisiones de salida y distintos grados de polinomio interpolador.

En la figura 6.11 podemos ver el número de segmentos necesarios para la aproximación de la ICDF para una precisión fija de 13 bits en las muestras Gaussianas generadas y diferentes grados de polinomio. Al igual que en el caso anterior, el número de segmentos necesarios en el caso de interpolación lineal es muy grande con respecto a los otros polinomios. Para los otros dos casos, el comportamiento sigue siendo idéntico al caso anterior y la reducción en el número de segmentos necesarios para el polinomio de grado tres implicará un aumento de la latencia de la implementación. La opción más equilibrada a partir de los resultados anteriores será la utilización de un polinomio de grado dos ya que es la que menos multiplicadores necesita para unos requerimientos de memoria contenidos.

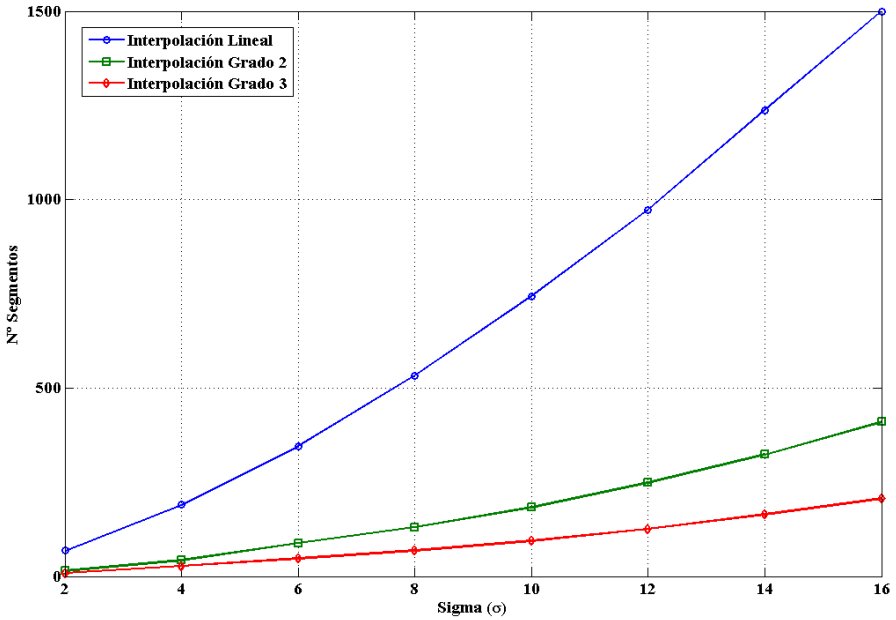


Figura 6.11 – Variación del numero de segmentos para diferentes valores de desviación y distintos grados de polinomio interpolador.

Por último, queremos indicar que todo el algoritmo antes descrito ha sido implementado mediante el lenguaje de programación Python. Python permite realizar operaciones aritméticas con tamaños de palabra grandes mediante la utilización de la librería *mpmath*. *Mpmath* es una librería que puede realizar operaciones aritméticas en coma flotante trabajando con multiprecisión.

6.3. Arquitectura Propuesta

En la figura 6.12 podemos ver con más detalle la arquitectura propuesta para el generador AWGN. Las especificaciones definidas para nuestro generador de números aleatorios Gaussianos serán de una periodicidad de 10^{53}, un tamaño de las muestras de 18 bits codificadas en complemento a dos con una precisión de 13 bits y una desviación estándar de $\pm 13.1\sigma$. La periodicidad del GNAG vendrá dada por la periodicidad de GNUD y ésta deberá ser superior a la inversa de la probabilidad de que ocurra un evento en $\pm 13\sigma$, que es aproximadamente 10^{38}. La periodicidad escogida ha sido de 10^{53}. Como hemos visto anteriormente para generar las muestras Gaussianas, realizaremos una transformación de muestras uniformemente distribuidas aplicando la ICDF de la distribución Gaussiana.

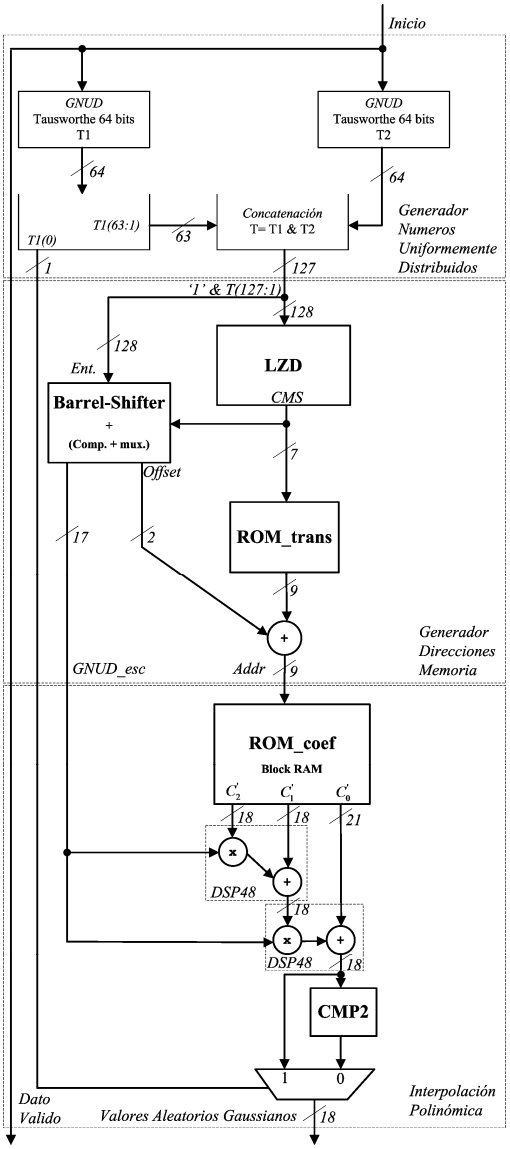
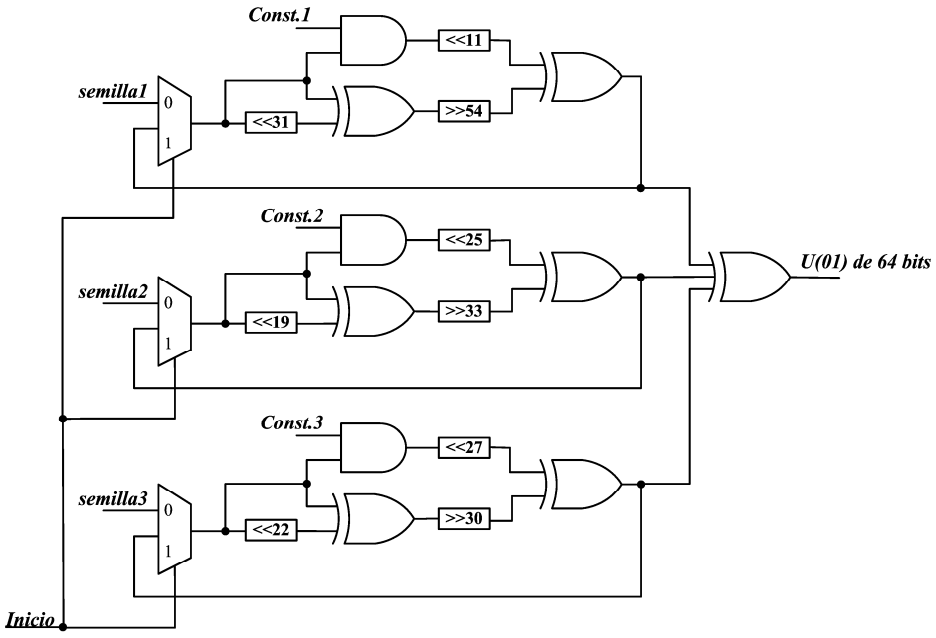


Figura 6.12 – Arquitectura propuesta para la aproximación de la función ICDF de la distribución Gaussiana.

La arquitectura implementada constará de tres etapas. La primera etapa generará números uniformemente distribuidos. La segunda etapa calculará la dirección de memoria correspondiente al segmento generado por el GNUD. Ésta estará compuesta por un *Leading Zero Detector* (LZD) y un *barrel-shifter* que será utilizado para extraer x_t y x_0 del valor generado por el GNUD. Además, mediante la utilización de la *ROM_trans* y el valor del LZD calcularemos el número de segmento que será la dirección de memoria donde están los coeficientes (siguiente etapa). En la tercera etapa una memoria (*ROM_coef*) proporcionará los coeficientes (a partir de la dirección de memoria calculada en la etapa anterior) y se realizará la interpolación polinómica de grado 2.

6.3.1. Generador de Números Uniformemente Distribuidos (GNUD)

Para la generación de los valores aleatorios uniformemente distribuidos hemos utilizado un generador Tauswhorte [73]. El generador Tausworthe está formado a partir de 3 LFSR y de esta manera se obtiene una mejor equi-distribución de los valores aleatorios generados. Para poder generar valores de hasta $\pm 13.1\sigma$ necesitamos generar el valor $1\text{-}2^{-128}$, necesitando para ello un generador de 128 bits. Para nuestro diseño se han utilizado dos generadores Tausworthe de 64 bits (T1 y T2) y concatenando las salidas de ambos se obtienen los 128 bits necesarios. La periodicidad de los generadores utilizados es de 2^{175} [78], valor suficiente para las necesidades de nuestra implementación. La figura 6.13 muestra el esquema del generador Tausworthe de 64 bits utilizado. Los valores **semilla1**, **semilla2**, **semilla3** son tres valores aleatorios de 64 bits y dependiendo de los valores elegidos se generará una secuencia u otra, como se ha podido ver en el capítulo anterior.



Const.1	Const.2	Const.3
0xFFFFFFFFFFFFFFFE	0xFFFFFFFFFFFFFFC0	0xFFFFFFFFFFFFFF80
Semilla1	Semilla2	Semilla3
0x1234567891014569	0x12D5417F2F2564FE	0xF125487FFF2584FE

Figura 6.13 - Generador Tausworthe de 64 bits.

6.3.2. Generador de Direcciones

Esta etapa es la encargada de generar a partir de un valor generado por el GNUD las direcciones de memoria necesarias para acceder a los coeficientes almacenados en la ROM de coeficientes. En la figura 6.14 podemos ver la arquitectura implementada para el cálculo de la dirección de memoria y del valor de *x* escalado. De los 128 bits del GNUD, utilizaremos uno para la generación del bit de signo. Este bit nos permitirá extender el rango de la aproximación generada al rango original de la ICDF de la distribución Gaussiana. Como se ha podido apreciar de la figura 6.3, la función ICDF es simétrica y únicamente aproximamos el rango de entrada [0.5, 1). La aproximación para el rango [0, 0.5) será generada complementando a dos del valor de salida cuando así lo marque el bit de signo. Para calcular qué segmento tenemos que acceder en cada instante utilizaremos el valor de la posición del "cero" más significativo (CMS), como se ha indicado en el punto anterior. Para esta tarea utilizaremos un bloque *Leading Zero Detector* (LZD) basado en la arquitectura propuesta por Oklobdzija [46]. Con la información sobre la posición del CMS y mediante la utilización de un *barrel-shifter* desplazaremos la entrada *x* para quedarnos únicamente con el valor x_t. Una vez desplazado *x*, tendremos que los MSBs (uno o dos bits en el caso de interpolación cuadrática) serán el valor de *offset*. Como este tamaño de *offset* es variable y depende de la posición del CMS, utilizaremos un comparador para detectar el segmento umbral y seleccionar dos bits si el número de segmento es menor que el umbral o uno si es mayor. En caso de seleccionar un bit, concatenaremos un cero como bit de mayor peso para poder implementar correctamente el multiplexor. Este umbral dependerá del $\varepsilon_{deseado}$ definido en el algoritmo de la figura 6.6. Para un $\varepsilon_{deseado} = 2^{-13}$, que es el utilizado en las implementaciones que siguen, este valor umbral es 36. Con una segmentación basada en *Bxo* de tamaño fijo no es necesario añadir este comparador ya que siempre tiene el mismo tamaño. Los siguientes 17 bits serán los que utilizaremos en la etapa de interpolación. Este valor siempre será positivo y estará comprendido en [0 – 1), con lo que añadiremos un '0' en el MSB, obteniendo la entrada *xt* en formato 18.17.

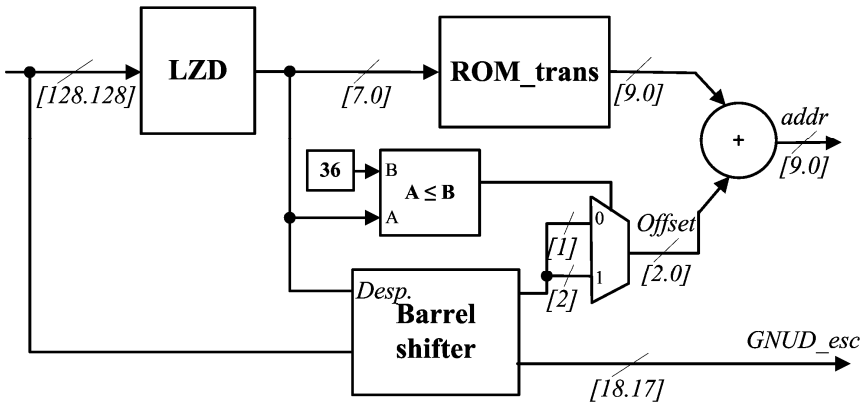


Figura 6.14 – Diagrama de bloques del generador de direcciones utilizado para la segmentación no-uniforme.

Para direccionar la tabla *ROM_trans* utilizaremos el valor del CMS al que le restaremos siempre una posición. Esto es debido a que al utilizar únicamente el rango [0.5 - 1) siempre el primer bit es uno. Evitaremos la implementación de este restador, desplazando el contenido de *ROM_trans* una posición. Esta tabla nos permitirá traducir la posición del CMS en una dirección de memoria base a partir de la cual tenemos almacenados todos los segmentos con mismo escalado (*Bxa*). Una vez tenemos la posición de memoria traducida (`addr_trans`) debemos sumar el valor de *offset* necesario para diferenciar los segmentos con idéntico CMS. Para ello utilizaremos los dos MSB de la salida del *barrel-shifter*. Dependiendo del tramo donde está ubicado el segmento este valor tendrá un tamaño de dos o un bits.

Sumando el valor de la salida de la memoria con este desplazamiento, obtendremos el valor de la posición de memoria donde están almacenados los coeficientes.

6.3.3. Interpolación Polinómica

En la etapa de la interpolación polinómica necesaria para la implementación del polinomio utilizado en la aproximación hemos aplicado la regla de Horner, donde

$$y = \left(\left((C_d \cdot x_t + C_{d-1}) \cdot x_t + C_{d-2}\right) \cdot x_t + ...\right) \cdot x_t + C_0 \tag{6.12}$$

siendo C_i los coeficientes del polinomio, d el grado del polinomio y x_t los bits de la salida x del GNUD que se emplearán como entrada en la interpolación. Este polinomio aproximará la mitad superior de la Gaussiana. Aplicando el bit de signo obtenido de la etapa GNUD generaremos toda la Gaussiana complementando a dos el valor a la salida del polinomio. Para la implementación de los operadores aritméticos hemos utilizado los bloques DSP48E disponibles en las FPGAS Virtex-5 de Xilinx. Estos bloques están compuestos de un multiplicador en complemento a dos de 18x25 bits más un sumador de 48 bits. Si la interpolación es lineal únicamente será necesario un bloque DSP48E y si trabajamos con un polinomio de grado 2 únicamente serán necesarios 2 bloques. Para el resto de dispositivos Virtex-4 y Virtex-2 se han utilizado los bloques DSP48 y MULT18X18S, respectivamente. En el caso de los dispositivos Virtex-2 al no disponer de sumadores, estos se han implementado mediante los elementos disponibles en los *slices* (MUXCY y XORCY). La figura 6.15 muestra el esquema de interpolación lineal y de segundo grado implementados.

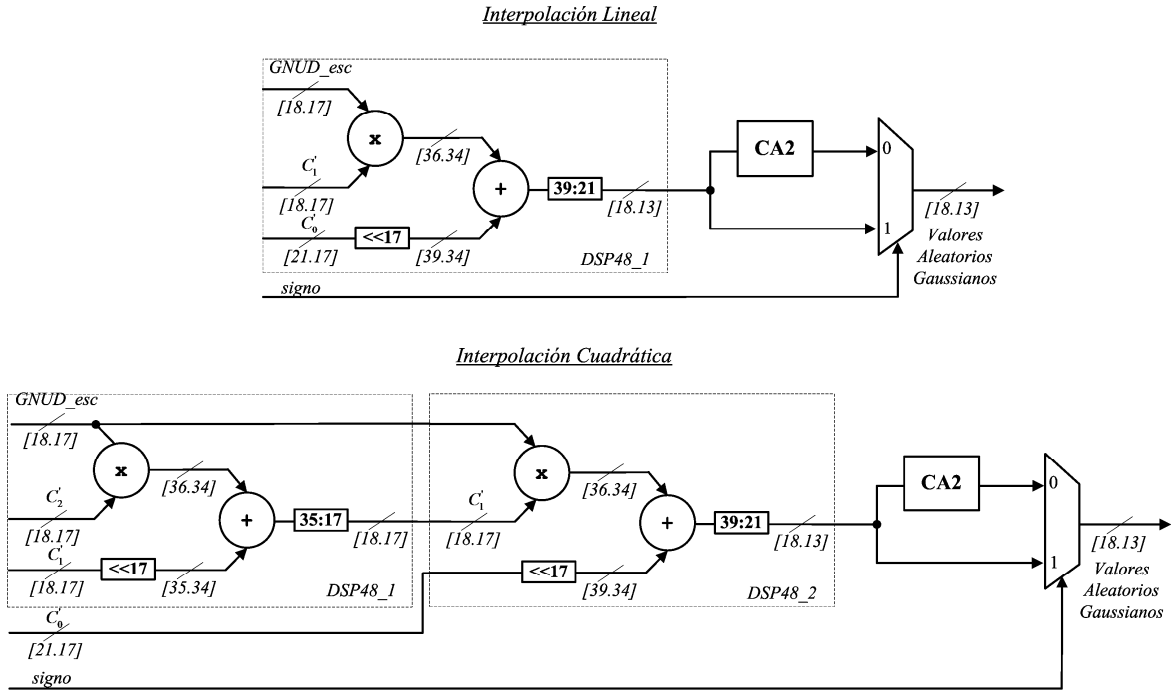


Figura 6.15 - Implementación de la interpolación lineal y cuadrática de la función ICDF.

Para almacenar los coeficientes en el dispositivo hemos utilizado las memorias embebidas Block-RAM de doble puerto disponibles en las FPGAs de Xilinx. Estas memorias nos permiten acceder a dos posiciones de memoria a la vez. El coeficiente C_0 ha sido almacenado con un tamaño de 21 bits y una

precisión de 17 bits. Los coeficientes C_1 y C_2 han sido almacenados con un tamaño de 18 bits y una precisión de 17 bits para el caso de interpolación cuadrática. Si utilizamos una interpolación lineal únicamente serán necesarios los coeficientes C_1 y C_0. Para la implementación con interpolación cuadrática son necesarios 298 segmentos y para el caso de interpolación lineal el numero de segmentos crece hasta 1100 segmentos. Las memorias Block-RAM disponibles en los dispositivos de Xilinx Virtex-2 y Virtex-4, tienen un tamaño de 18432 bits que se pueden configurar de diferentes tamaños (512x36, 1024x18, 2048x9, etc.) y configuraremos la memoria de coeficientes como una memoria ROM de 512 posiciones de 36 bits. Para el caso de la Virtex-5 el tamaño de la memoria se duplica hasta los 36864 bits, permitiendo poder ser configurada como 1024x36. Esta memoria la dividiremos en dos páginas de 512 posiciones y cada uno de los dos puertos de la Block-RAM direccionará una página. Los coeficientes C_2 y C_1 de 18 bits serán concatenados para formar una única palabra de 36 bits. Los coeficientes C_0 los almacenaremos en la página inferior. El resto de posiciones serán rellenadas con ceros. En los dispositivos Virtex-4/2 almacenaremos los coeficientes C_2 y C_1 en un Block-RAM y los coeficientes C_0 en otra Block-RAM. Para la interpolación lineal cada uno de los dos coeficientes será almacenado en Block-RAM independientes, debido al mayor número de segmentos necesarios. En la figura 6.16 podemos ver los esquemas de memorias utilizados para una interpolación de grado dos y para las diferentes FPGAs utilizadas.

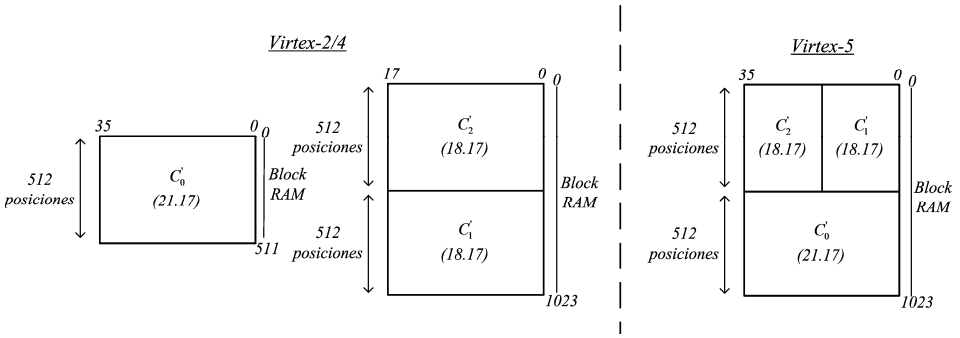


Figura 6.16 – Esquema de memoria utilizados para la interpolación cuadrática en distintas FPGAs.

6.4. Resultados de Implementación

En esta sección presentamos los resultados obtenidos en la implementación de nuestro diseño en la FPGA. Todo el diseño ha sido modelado en VHDL y se ha sintetizado, emplazado y rutado utilizando la herramienta Xilinx ISE 12.4. Para la implementación de la arquitectura propuesta hemos utilizado distintas FPGAs de Xilinx, en concreto Virtex-5 XC5VLX110T-3, Virtex-4 XC4VLX100-12 y una Virtex-II XC2V4000-6. Para las medidas de consumo de potencia se ha utilizado la aplicación Xpower Analyzer de Xilinx. En la Tabla 6.1 podemos ver los resultados de implementación obtenidos después de realizar el Place & Route para un valor de desviación máxima de $\pm 13.1\sigma$ con un tamaño de palabra de 18 bits y una parte fraccional de 13 bits, mediante la utilización de interpolación cuadrática. En la tabla 6.2 presentamos los resultados obtenidos utilizando interpolación lineal. Se ha segmentado el operador para poder trabajar al máximo *throughput* posible. La latencia del operador es de 18 y 15 ciclos de reloj para el caso de utilización de la interpolación cuadrática y lineal, respectivamente. En la figura 6.17 vemos cómo se han distribuido las distintas etapas de segmentado. El elemento que más latencia añade es el *barrel-shifter* con cinco ciclos de reloj, seguido de los operadores *mult-sum* los cuales están implementados mediante bloques DSP48 que añaden una latencia de 3 ciclos de reloj. No hemos segmentado más el operador ya que el camino crítico lo impone el retardo obtenido en la Block-RAM. Para el caso de la interpolación lineal reducimos hasta 15 ciclos de reloj debido a que únicamente utilizamos un bloque *mult-sum*.

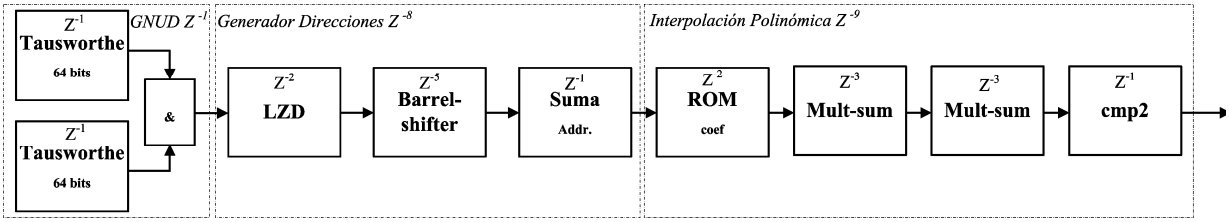


Figura 6.17 - Distribución de las etapas de segmentado en los distintos bloques de la aproximación propuesta.

Dispositivo	Virtex-II XC2V4000-6	Virtex-4 XC4VLX100-12	Virtex-5 XC5VLX110T-3
Desviación máx.		$\pm 13.1\sigma$	
Periodicidad		2^{175}	
Nº Segmentos		298	
Slices	760	745	263
LUT4	1281	1180	795 (LUT6)
F.F.	1148	1058	756
Block-RAM	2	2	1
DSP48	2 (mult18x18)	2	2
Fmax (MHz)	244.2	373	472.5
Throughput (Msps)	244.2	373	472.5
Consumo Potencia	1.85 mW/MHz	0.40 mW/MHz	0.38 mW/MHz
Tamaño palabra		18.13	
Latencia		18	

Tabla 6.1 - Resultados de implementación del GNAG propuesto utilizando interpolación cuadrática para diferentes dispositivos.

Dispositivo	Virtex-II XC2V4000-6	Virtex-4 XC4VLX100-12	Virtex-5 XC5VLX110T-3
Desviación máx.		$\pm 13.1\sigma$	
Periodicidad		2^{175}	
Nº Segmentos		1100	
Slices	832	812	275
LUT4	1380	1348	849(LUT6)
F.F.	1090	1040	760
Block-RAM	3	3	2
DSP48	1 (mult18x18)	1	1
Fmax (MHz)	241	371	475.6
Throughput (Msps)	241	371	475.6
Consumo Potencia	2.15 mW/MHz	0.45 mW/MHz	0.41 mW/MHz
Tamaño palabra		18.13	
Latencia		15	

Tabla 6.2 - Resultados de implementación del GNAG propuesto utilizando interpolación lineal para diferentes dispositivos.

Podemos apreciar en los resultados obtenidos para las implementaciones lineal y cuadrática una reducción significativa en el número de LUTs (*slices*) en el dispositivo Virtex-5 con respecto a los otros dos dispositivos. Esto es debido a que las FPGAS Virtex-5 el tamaño de las LUTs se ha duplicado, pasando de 16 a 32 bits. Además el número de LUTs por *slice* también se ha duplicado pasando de dos a cuatro. Para el caso de las implementaciones en Virtex-II no hemos utilizado bloques DSP48 y únicamente se han utilizado los multiplicadores embebidos disponibles. Los sumadores han sido implementados utilizando los recursos disponibles (MUXCY y XORCY) en los *slices*, de ahí, la pérdida de velocidad con respecto a las implementaciones en Virtex-4/5. Vemos cómo la diferencia más significativa entre los resultados obtenidos de la interpolación lineal y cuadrática es en el número de DSP48 y Block-RAMs utilizadas. Conforme aumentamos el grado de interpolación necesitamos más multiplicadores y menos memoria y viceversa. Para la interpolación lineal se han utilizado 3 Block-RAMs. Esto es debido a que aparte de las dos necesarias para almacenar los coeficientes se ha utilizado una tercera para almacenar la memoria de traducción de direcciones (*ROM_trans*), ya que el número de segmentos necesarios es alto y una implementación mediante memoria distribuida sería ineficiente. La velocidad máxima de generación de muestras es de aproximadamente 475 Msps para las dos implementaciones utilizando un dispositivo Virtex-5.

Virtex-5		**Generador direcciones**			**Aproximación Polinomio**	
XC5VLX110T-3	**GNUD**	**LZD**	**Barrel-shifter**	**ROM_trans**	**Interpolación**	**ROM_coef**
Slices	72	32	118	16	34	0
LUT6	237	44	342	47	110	0
F.F.	175	33	438	16	110	0
Block-RAM	0	0	0	0	0	1
DSP48	0	0	0	0	2	0

Tabla 6.3 – Recursos hardware utilizados por los distintos bloques del GNAG propuesto y utilizando interpolación cuadrática.

En la tabla 6.3 podemos ver los recursos de la FPGA Virtex-5 utilizados en las partes que componen el generador GNAG. Como podemos observar el elemento que más recursos utiliza es el *barrel-shifter* seguido del GNUD debido al tamaño de los operadores necesarios (128 bits) para poder llegar al valor de $\pm 13.1\sigma$. Estos resultados están obtenidos de la implementación mediante interpolación cuadrática y son fácilmente extrapolables a la interpolación lineal. En la tabla 6.4 vemos los recursos utilizados para diferentes valores de precisión de las muestras generadas, utilizando un polinomio de grado dos. Podemos apreciar cómo el número de *slices* utilizados no presenta un crecimiento tan acusado como el número de memorias Block-RAM necesarias. Esto es debido a que la precisión de las muestras generadas únicamente influirá en la etapa de interpolación y más concretamente en el tamaño de los segmentos. Al aumentar la precisión de las muestras necesitamos un mayor número de segmentos. Además, a partir de precisiones de más de 14 bits vemos cómo aumentamos la latencia de la implementación debido al tamaño de los bloques DSP48 necesitando aumentar el número de operadores utilizados y por consiguiente aumentar la latencia del operador. En cuanto al área necesaria vemos cómo para pasar de 8 a 20 bits únicamente ha aumentado un 38.9% el número de *slices* utilizados. Por otro lado, el *throughput* máximo ha bajado únicamente un 5% debido a que hemos de encadenar dos bloques DSP48 por cada multiplicación aumentando el retardo necesario y la latencia del operador. La variación en la precisión mostrada en la tabla únicamente afectará a la parte fraccional de las muestras ya que la parte entera no cambia ($\pm 13.1\sigma$).

Precisión	8	10	12	14	16	18	20
Slices	216	222	232	288	304	320	354
LUT6	692	722	743	814	832	873	963
F.F.	660	690	720	762	785	830	915
Block RAM	1	1	1	1	1	2	3
DSP48	2	2	2	2	4	4	4
Fmax (MHz)	472.14	472.14	472.14	470.3	450.5	450.5	448.9
Latencia	18	18	18	18	26	26	26
Throughput (Msps)	472.14	472.14	472.14	470.3	450.5	450.5	448.9

Tabla 6.4 – Resultados de implementación del GNAG propuesto modificando la precisión de las muestras generadas a la salida y un valor de desviación fijo.

En la tabla 6.5 se muestran los recursos utilizados para diferentes valores de desviación estándar σ de la distribución Gaussiana generada mediante interpolación cuadrática. A diferencia del caso anterior, podemos ver cómo el crecimiento en el número de *slices* es mucho más acusado, ya que el valor σ generado dependerá del tamaño del bloque GNUD. Cuanto más grande sea el GNUD mayor desviación en las muestras podremos generar. Además, conforme vamos aumentando el tamaño del GNUD necesitaremos un *barrel-shifter* mayor. También vemos cómo el número de segmentos aumenta considerablemente para altos valores de σ debido a la marcada no-linealidad de la ICDF para valores cercanos a uno. Por ejemplo, para el caso de una distribución con ±4σ pasar a una distribución de ±14σ supone duplicar aproximadamente en el número de *slices* necesarios. Al igual que el caso anterior el número de Block-RAMs también aumenta debido al incremento del número de segmentos necesarios para llegar al valor de ±14σ. Vemos cómo la latencia de la implementación va creciendo conforme aumenta el valor de la desviación. Este crecimiento se debe exclusivamente al crecimiento en el tamaño del *barrel-shifter*, teniendo que añadir un mayor número de etapas de segmentado. La precisión de las muestras generadas es siempre de 13 bits ya que al modificar el valor de σ, únicamente cambia el tamaño de la parte entera de las muestras generadas. Para el GNAG propuesto, el tamaño de palabra no cambia ya que con 5 bits de la parte entera podemos llegar a generar muestras hasta ±16σ.

Desviación estándar (σ)	±2.4	±4	±6	±8	±10	±12	±14
Slices	119	183	164	204	214	243	385
LUT6	286	509	462	637	708	755	1120
F.F.	329	537	482	668	705	746	1162
Block RAM	0	0	1	1	1	1	3
DSP48	2	2	2	2	2	2	2
Fmax (MHz)	475.3	475.3	475.3	472.1	472.1	472.1	460.2
Latencia	15	16	17	18	18	18	20
Throughput (Msps)	475.3	475.3	475.3	472.1	472.1	472.1	450.2

Tabla 6.5 – Resultados de implementación del GNAG propuesto modificando el valor de desviación máxima y un tamaño de las muestras generadas fijo.

En la tabla 6.6 hemos realizado la misma implementación pero utilizando diferentes tipos de elementos disponibles en las FPGAs. Por ejemplo, las operaciones de multiplicación utilizadas en la interpolación pueden ser implementadas mediante bloques DSP48 o implementando los multiplicadores con LUTs. Vemos como el numero de *slices* utilizados es tres veces superior que utilizando los bloques disponibles en la FPGA. Además vemos como la pérdida de prestaciones también es muy significativa sobre todo debido a la implementación de los multiplicadores mediante LUTs. Por ello, siempre será interesante la utilización de estos elementos para obtener implementaciones eficientes de alta velocidad. Los resultados han sido obtenidos para una implementación en FPGA Virtex-5 y utilizando interpolación cuadrática.

Recursos FPGA utilizados	**Slices (LUT)**	**BRAM**	**DSP**	**F_{max}**
Slices+BRAM+DSP	263 (795)	1	2	472.5
Slices+DSP	450(1069)	0	2	370.1
Slices+BRAM	553(1486)	1	0	180
Slices	720(1972)	0	0	146

Tabla 6.6 - Resultados de la implementación modificando el tipo de recursos de la FPGA.

6.4.1. Arquitectura Propuesta Modificada

Analizando el comportamiento de la arquitectura propuesta anteriormente podemos apreciar que en la interpolación usamos 17 bits del GNUD (x_t). Estos bits siempre los obtenemos a partir de la posición del LZD. Para ello, eliminamos los bits más significativos, que contienen la información del segmento, y nos quedamos únicamente con esos 17 bits por medio de un *barrel-shifter*. Además, este valor x_t siempre estará comprendido en [0, 1). Aprovechando que x_t está comprendido dentro del rango [0, 1) y que es una variable uniformemente distribuida, en un principio, el comportamiento debería de ser el mismo si utilizamos los 17 bits siguientes al CMS o utilizar siempre los 17 bits menos significativos de la salida del GNUD. En la segunda posibilidad conseguimos eliminar de la implementación el *barrel-shifter* permitiendo un ahorro de *slices* significativo, ya que como se ha visto en el punto anterior es el componente que más recursos utiliza. Para el bit de signo siempre utilizaremos el bit menos significativo del GNUD. Para obtener los dos bits de *offset* necesarios para seleccionar los distintos segmentos con la misma posición del CMS, se ha seguido la misma idea y siempre vamos a coger los dos bits siguientes al bit de signo. Los bits para interpolación (x_t) serán siempre los 17 bits siguientes a los bits de *offset*. En este apartado se proponen diferentes modificaciones sobre la arquitectura del GNAG desarrollada hasta este momento. Ninguna de las propuestas utiliza el *barrel-shifter* de 128 bits, con la consiguiente reducción de área. La primera solución utiliza los bits del GNUD tal cual se ha comentado anteriormente. Esta realización tiene el inconveniente que la posición del CMS podrá ser como máximo de 128-19=109, ya que los últimos 19 bits son utilizados exclusivamente para la interpolación (17 bits de x_t, 1 bit de *Bxo* y un bit para el signo). Si utilizáramos segmentos con una posición de CMS mayor estaríamos usando para la interpolación bits que identifican el segmento y que por tanto son constantes en todo el segmento, con el consiguiente error en la interpolación. Esta reducción en el número de bits utilizados para el segmentado respecto del máximo posible, resultará en un valor máximo de amplitud menor, llegando únicamente hasta una desviación de $\pm 12.2\sigma$. En la arquitectura propuesta anteriormente el valor máximo del CMS es de hasta 128 (entrada todo a unos). En la figura 6.18 se muestra la nueva arquitectura definida y la identificaremos como arquitectura A. Como podemos apreciar, la gran mayoría de los bloques son los mismos que en la figura 6.12 con la única diferencia de que no utilizamos el *barrel-shifter*.

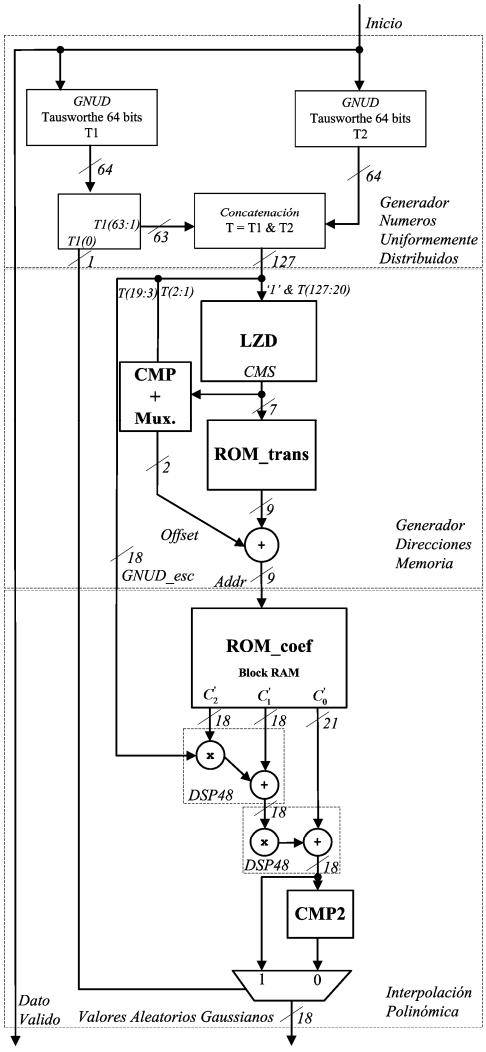


Figura 6.18 - Arquitectura del GNAG propuesto. Arq. modificada A.

La eliminación del *barrel-shifter* ha provocado una reducción en la desviación máxima que puede generar el GNAG con el mismo número de bits utilizados en el GNUD. Para solucionar esta pérdida de prestaciones se han propuesto dos implementaciones, que denominaremos arquitecturas B y C. La arquitectura B constará de los mismos elementos que la arquitectura A con el añadido de un pequeño *barrel-shifter* de 17 bits para la parte utilizada en la interpolación. De esta manera el valor del CMS puede crecer hasta agotar los bits del GNUD, pudiendo generar el valor máximo de desviación correspondiente a 128 bits. En la figura 6.19 vemos el esquema del generador de direcciones de la arquitectura B, ya que el resto de la implementación es el mismo que la arquitectura A. En este esquema detectamos cuándo el CMS está dentro de los 17 bits utilizados para interpolación y si es ése el caso habilitaremos el *barrel-shifter* y calculamos cuántos bits tenemos que desplazar la entrada para eliminar el CMS. Si el CMS está fuera de los bits asignados a la interpolación el funcionamiento es el normal y el *barrel-shifter* no actúa sobre la parte definida para la interpolación. Para nuestra implementación, este valor umbral es de 110. Si el CMS es igual o superior a este valor necesitamos realizar el desplazamiento. El valor de desplazamiento será la posición del CMS menos este valor umbral. Al realizar el desplazamiento de los 17 bits para eliminar la información del CMS estamos rellenando con ceros siempre los LSBs de la parte de interpolación y de esta manera generamos los 17 bits necesarios a la entrada de los multiplicadores.

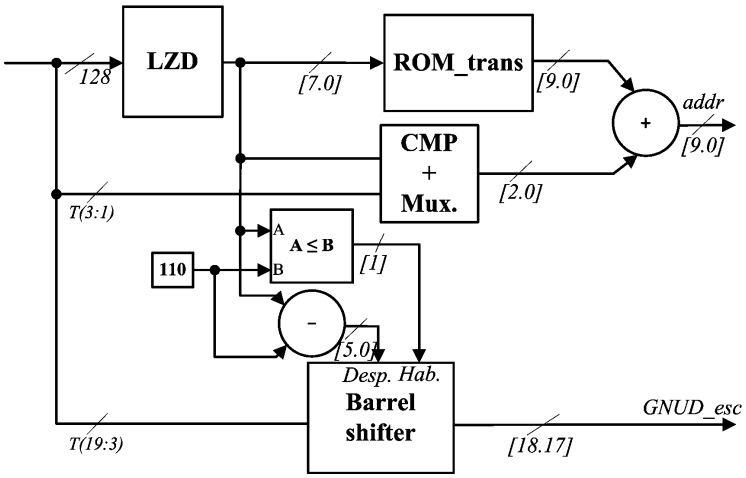


Figura 6.19 - Generador de direcciones utilizado en la arquitectura B.

Por último, se ha definido una nueva arquitectura a la que llamaremos C. A diferencia de la arquitectura B en la que los bits utilizados para direccionamiento son eliminados por medio de un desplazamiento, en la arquitectura C estos bits serán eliminados por medio de una máscara que los pondrá a cero. En esta arquitectura vamos a realizar una modificación en la asignación de pesos a los bits de x_t. En las dos arquitecturas definidas previamente el bit MSB y el LSB de x_t siempre eran los bits 19 y 3, respectivamente. Podemos invertir el orden de los pesos de x_t y el comportamiento del GNAG seguiría siendo el mismo. Para el caso C, el MSB y LSB serán los bits 3 y 19, respectivamente. En la figura 6.20 podemos ver un ejemplo del funcionamiento de este enmascaramiento. Como vemos en el caso (a), el CMS está fuera de los bits de interpolación y los bits a la entrada del multiplicador serán los numerados del 19 al 3 con los pesos invertidos. En el caso (b), al estar el valor del CMS dentro de x_t necesitamos que los bits a la izquierda del CMS sean eliminados y únicamente trabajar con los bits a la derecha del CMS. Para ello calcularemos una máscara en la que los bits a la izquierda del CMS sean cero y los bits de la derecha del CMS sean uno. Al aplicarla únicamente nos quedaremos con los valores a la derecha del CMS y el resto serán cero.

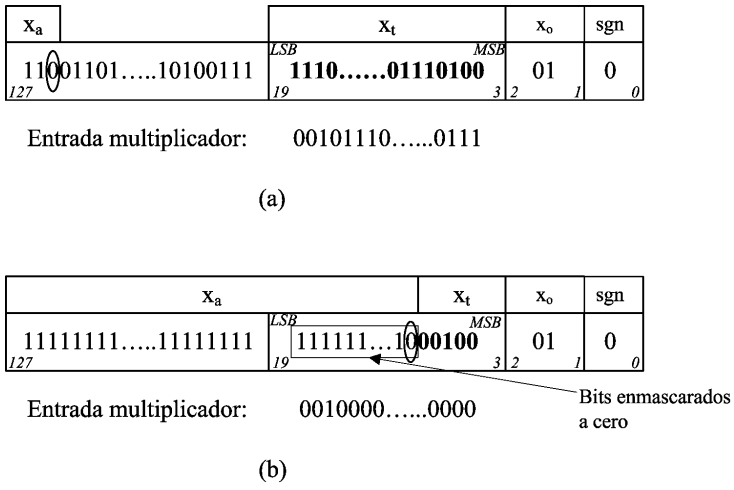


Figura 6.20 – Esquema de asignación de bits para el direccionamiento e interpolación de la arquitectura C: (a) el CMS aparece entre los bits del 127 al 20; (b) el CMS aparece dentro de los bits del 19 al 3.

Para realizar esta tarea se ha implementado un pequeño circuito combinacional, mostrado en la figura 6.21. Como se puede apreciar únicamente es necesaria una puerta AND por cada bit para detectar donde está la posición del CMS y a partir de este bit propagar siempre un cero. El resultado lo invertimos

y por medio de una segunda etapa de ANDs aplicamos la máscara. Los bits a la salida serán los utilizados en la etapa de interpolación. En la figura 6.22 vemos el esquema del generador de direcciones de la arquitectura C. El resto de la implementación es el mismo del caso anterior.

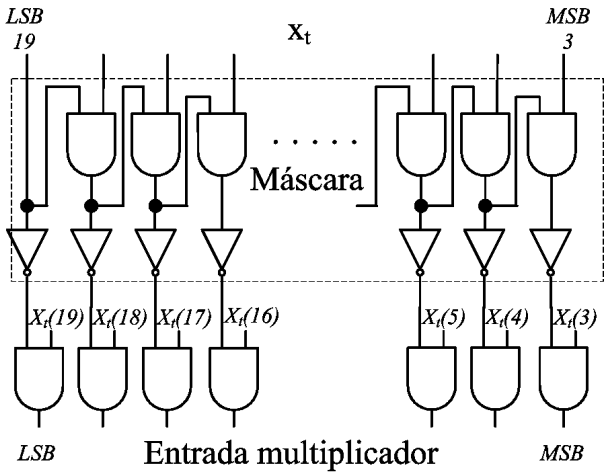


Figura 6.21 – Circuito generador de la máscara utilizado en la arquitectura C.

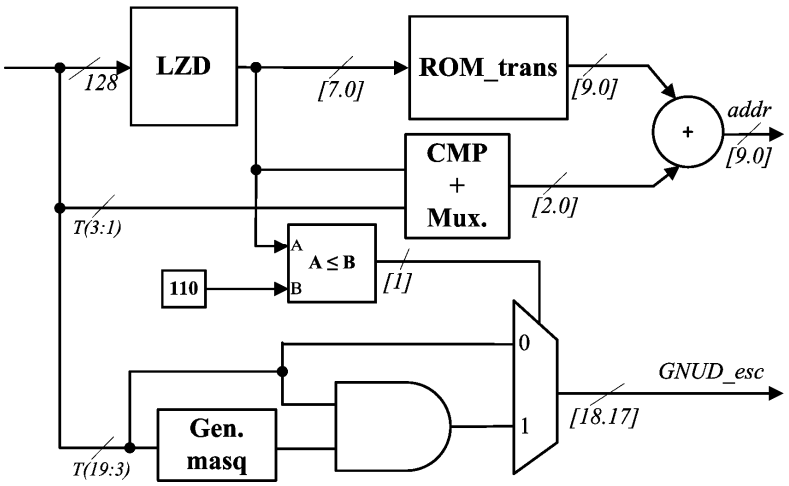


Figura 6.22 - Generador de direcciones utilizado en la arquitectura C.

En la tabla 6.7 presentamos los resultados de las implementaciones de las tres arquitecturas presentadas anteriormente después de realizar el Place & Route, trabajando con un tamaño de palabra de 18 bits y una parte fraccional de 13 bits mediante la utilización de interpolación cuadrática y un GNUD de 128 bits. Sin pérdida de generalidad vamos a dar los resultados únicamente para el dispositivo Virtex-5, ya que las conclusiones que se sacan de las implementaciones sobre otros dispositivos son las mismas. Podemos ver cómo en las tres nuevas arquitecturas ha habido una importante reducción en el número de LUTs utilizados con respecto a la arquitectura original. Para el caso del dispositivo Virtex-5 esta reducción ha sido de aproximadamente un 58.3% para el caso de la arquitectura C y para el caso de la arquitectura B ha sido un poco menor. El número de segmentos necesarios no cambia en las implementaciones B y C, ya que este parámetro únicamente dependerá del error cometido por segmento y del polinomio interpolador utilizado. Para el caso de la implementación A, al generar un valor menor de desviación el número de segmentos es menor. También se ha reducido la latencia en dos ciclos de reloj (16) ya que como se ha visto anteriormente en la figura 6.17 el *barrel-shifter* era el elemento que más latencia añadía al circuito. La frecuencia máxima de la implementación no ha variado debido a que el camino-critico lo tenemos entre la

Block-RAM y los bloques DSP48. En la tabla 6.8 vemos la distribución de los distintos bloques de la aproximación C. A diferencia de la arquitectura original el elemento que más recursos utiliza es el GNUD. En la arquitectura original era el *barrel-shifter*. En el punto 6.4 nos quedará analizar el comportamiento aleatorio de las muestras Gaussianas generadas.

Dispositivo	**Arq. Original**	**Arq. A**	**Arq. B**	**Arq. C**
Desviación máx.	$\pm 13.1\sigma$	$\pm 12.2\sigma$	$\pm 13.1\sigma$	$\pm 13.1\sigma$
Periodicidad		2^{175}		
Nº Segmentos	298	254	298	298
Slices	263	149	172	154
LUT6	795	291	368	331
F.F.	756	496	570	555
Block-RAM	1	1	1	1
DSP48	2	2	2	2
Fmax (MHz)	472.5	472.1	472.1	472
Throughput (Msps)	472.5	472.1	472.1	472
Consumo Potencia	0.38 mW/MHz	0.23 mW/MHz	0.25 mW/MHz	0.24 mW/MHz
Tamaño palabra		18.13		
Latencia	18		16	

Tabla 6.7 - Resultados de implementación de los diferentes GNAG propuestos utilizando interpolación cuadrática.

Virtex-5 XC5VLX110T-3		**Generador direcciones**			**Aproximación Polinomio**	
	GNUD	**LZD**	**Gen. Masq.**	**ROM_trans**	**Interpolación**	**ROM_coef**
Slices	70	27	10	14	33	0
LUT6	167	38	10	36	80	0
F.F.	175	96	154	16	112	0
Block-RAM	0	0	0	0	0	1
DSP48	0	0	0	0	2	0

Tabla 6.8 - Recursos hardware utilizados por la arquitectura C propuesta y utilizando interpolación cuadrática.

Por último, podemos ver en la tabla 6.9 una comparación de los resultados de implementación obtenida por la arquitectura propuesta comparada con otras implementaciones presentadas por diversos autores. Para este caso, hemos utilizado los resultados obtenidos por la FPGA Virtex-II XC2V4000-6, debido a que este dispositivo es el utilizado por el resto de implementaciones. Todas las implementaciones están segmentadas al máximo. A partir de los resultados podemos ver que nuestra implementación es la que genera muestras con un valor mayor de desviación.

Implementación	[94]	[85]	[86]	[87]	[92]	[93]	[96]	[95]	Arq. Orig.	Arq. Mod. C
Método Empleado	B-M+ CLT	B-M+ CLT	B-M	B-M+ CLT	ICDF	Polar +CLT	Wallace	Ziggurat	ICDF	ICDF
Desviación máx.	±4.8σ	±6.7σ	±8.2σ	±9.4σ	±8.2σ	----	±7σ	----	±13.1σ	±13.1σ
Periodicidad	2^{190}	2^{60}	2^{88}	2^{258}	2^{88}	----	2^{88}	2^{88}	2^{175}	2^{175}
Slices	480	2514	1528	852	585	336	770	891	760	460
Block-RAM	5	2	3	3	1	2	6	4	2	2
Mult18x18	5	8	12	3	4	----	4	2	2	2
Fmax (MHz)	245	133	233	248	231	73.5	155	170	244.2	244.2
Throughput (Msps)	245	133	466	496	231	146	155	168	244.2	244.2
Tamaño palabra	16(11)	16(11)	16(11)	16(11)	16(11)	19	32(-)	24(-)	18(13)	18(13)
Test Estadísticos	No Pasado	Pasado	Pasado	Pasado	Pasado	-----	Pasado	Pasado	Pasado	Pasado
$r = \left(\frac{Throughput}{slices}\right) \cdot \sigma$	0.222	0.161	0.633	1.245	0.836	-----	0.218	-----	1.191	1.341

Tabla 6.9 - Comparativa de los resultados de implementación obtenidos por diversas implementaciones de GNAG.

Para poder comparar los resultados de implementación con respecto a las distintas arquitecturas, hemos calculado un parámetro *r*. Este parámetro representara el número de *slices* utilizados dividido por la tasa de datos generada y multiplicado por la desviación obtenida. Para poder cuantificar los elementos embebidos hemos implementado las Block-RAM y los MULT18X18 utilizando únicamente LUTs, obteniendo 780 y 183 *slices*, respectivamente. Podemos ver cómo nuestra implementación (arq. C) es la que obtiene un ratio mayor seguida de la implementación [87]. La gran ventaja de las implementaciones basadas en Box-Muller es que pueden generar dos muestras por iteración, de esta manera a pesar de utilizar más área, obtiene un mejor ratio que el resto de las implementaciones. Comparando con la otra implementación basada en el método de la ICDF [92], nuestra arquitectura obtiene un ratio mucho mayor debido a la menor utilización de *slices* y a la obtención de un valor de desviación mucho mayor. La implementación original a pesar de contar con el *barrel-shifter* de 128 bits obtiene un ratio bastante bueno. El resto de implementaciones obtienen unos ratios mucho más bajos debido al número de Block-RAMs y multiplicadores embebidos utilizados. Nuestra implementación también es que la genera una mayor precisión en las muestras generadas (13 bits). Por último, podemos ver cómo la única implementación que no consigue superar los test estadísticos es la basada en el core-IP AWGN de Xilinx. Estos test los presentaremos en el siguiente punto.

6.5. Verificación de las Muestras Generadas

En la figura 6.23 podemos ver la función de densidad de probabilidad (PDF) para una población de 100 millones de muestras. Como podemos apreciar la desviación máxima obtenida en esta simulación es de ±6σ. Para poder representar correctamente que la distribución de la cola llega hasta ±13.1σ, sería necesario realizar simulaciones de hasta 10^{38} muestras. Trabajando a una frecuencia de 300 MHz serian

necesarios miles de años para poder validar correctamente la distribución Gaussiana generada con una desviación de hasta ±13.1σ. Para comparar las muestras generadas (azul) hemos superpuesto en la figura la distribución ideal Gaussiana (rojo) y cómo podemos ver no se aprecian diferencias entre las dos curvas. Un primer test que hemos realizado sobre las muestras generadas es el cálculo de los parámetros Curtosis y Skewness [97]. El parámetro Curtosis es una medida de la forma o apuntamiento de las distribuciones que para el caso de la distribución Gaussiana de media cero y varianza uno, es de 3. El Curtosis que se ha obtenido en muestras generadas es de ≈2.999, que como se aprecia es muy cercano al valor teórico. El parámetro Skewness indica la simetría en las muestras generadas obteniendo un valor de $\approx 5.35 \cdot 10^{-4}$, valor cercano al valor esperado de cero.

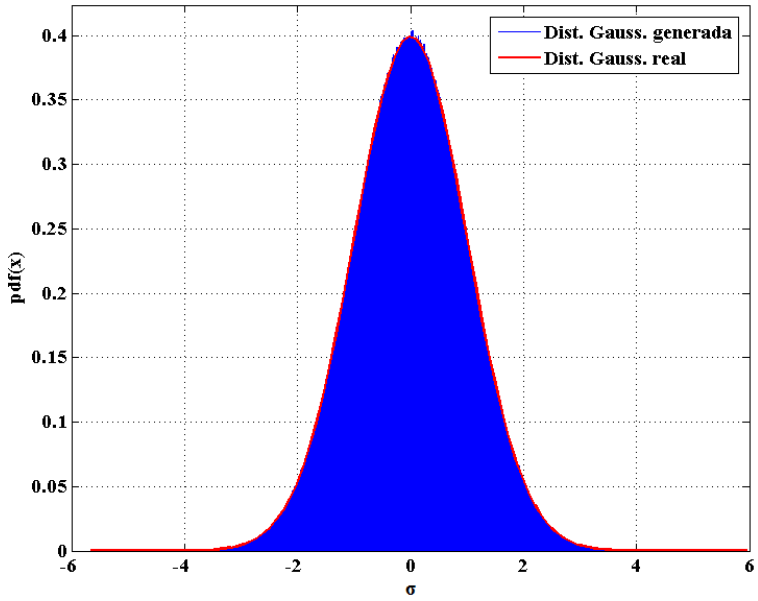


Figura 6.23 – Comparación de la PDF de las muestras generadas con respecto a la PDF ideal de la distribución

Una vez verificadas por inspección las muestras generadas vamos a pasar a comprobar las muestras generadas en la cola de la distribución, para ello vamos a utilizar la CDF de una variable Gaussiana en vez de la PDF. La relación entre las dos funciones es

$$PDF(x) = \frac{dCDF(x)}{dx}. \tag{6.13}$$

La CDF de la función la podemos reescribir aplicando el teorema de Bayes

$$CDF(x) = CDF(x|U_1 < x_{\lim}) \cdot \text{Prob}(U_1 < x_{\lim}) + CDF(x|U_1 \geq x_{\lim}) \cdot \text{Prob}(U_1 \geq x_{\lim}) \tag{6.14}$$

Para medir las muestras de la cola de la Gaussiana haremos que los valores generados en el GNUD sean muy cercanos a uno (dejando fijos a uno, los *n-bits* MSB del generador). En este caso como todas las muestras $U_1 > x_{lim}$ el primer termino de (6.14) tenderán a cero. La PDF generada vendrá dada por

$$PDF(x) = \frac{d}{dx} CDF(x|U_1 < x_{\lim}) \cdot \text{Prob}(U_1 < x_{\lim}) \quad \forall x. \tag{6.15}$$

Aplicando este método hemos dividido la cola de la Gaussiana en cuatro partes: 6.2σ a 7.8σ, 7.8σ a 9.2σ, 9.2σ a 10.6σ, 10.6σ a 12σ y 12σ a 13.1σ. En la figura 6.24 podemos ver la PDF de la cola para una población de 10 millones de muestras generadas (azul) comparada con la distribución Gaussiana ideal (verde). Para poder apreciar correctamente los valores de la cola hemos utilizado una escala logarítmica. No observamos diferencias entre las dos colas, verificando el correcto funcionamiento del generador GNAG. En la figura 6.20.e vemos cómo aparecen ligeras fluctuaciones entre la cola ideal y la cola generada. Esto es debido sobre todo a las pérdidas de precisión por la utilización de la herramienta Matlab para la representación de la Gaussiana ideal.

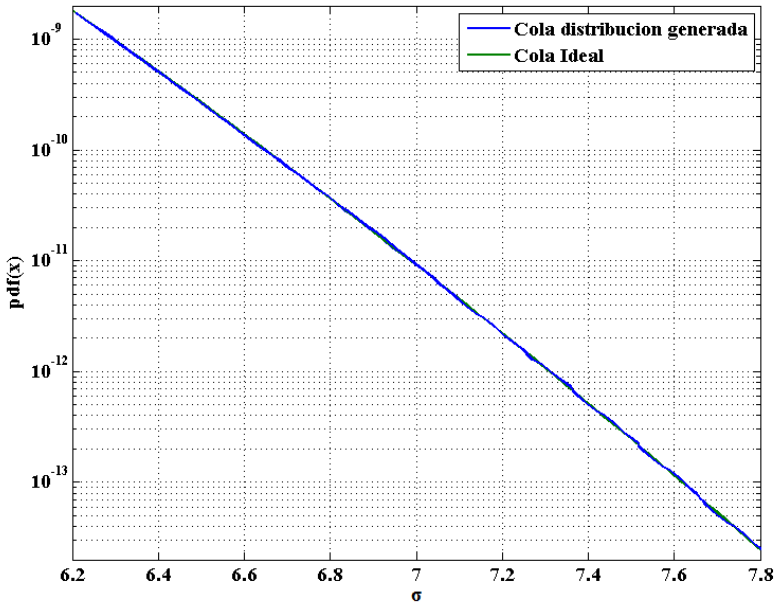


Figura 6.24.a - PDF de las muestras generadas comparadas con la distribución ideal en el tramo 6.2σ a 7.8σ.

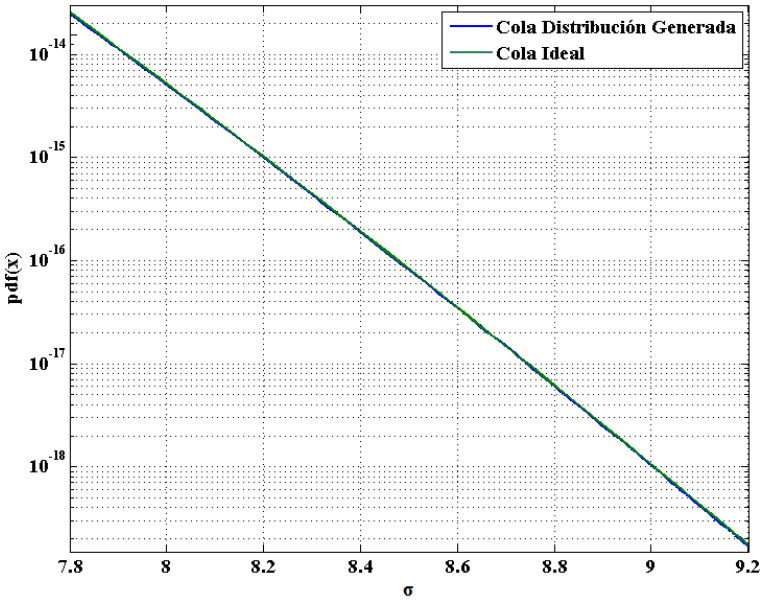


Figura 6.24.b - PDF de las muestras generadas comparadas con la distribución ideal en el tramo 7.8σ a 9.2σ.

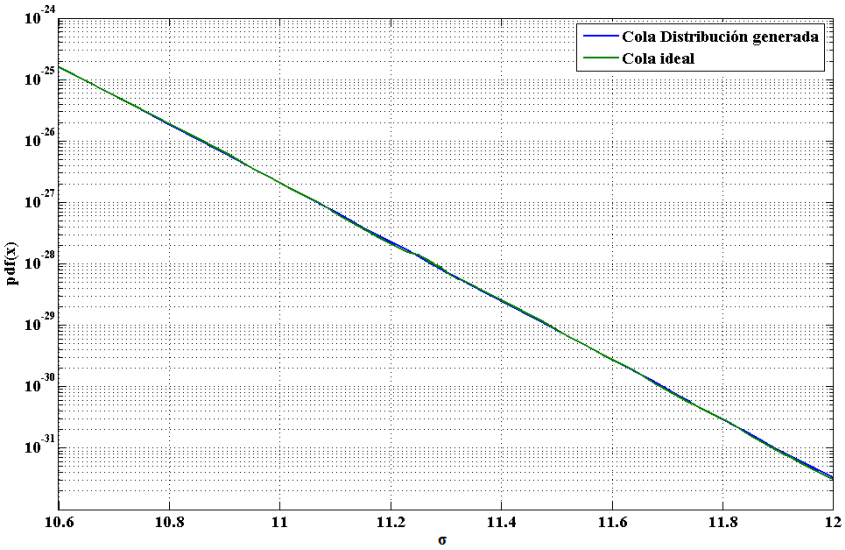


Figura 6.24.c - PDF de las muestras generadas comparadas con la distribución ideal en el tramo 9.2σ a 10.6σ.

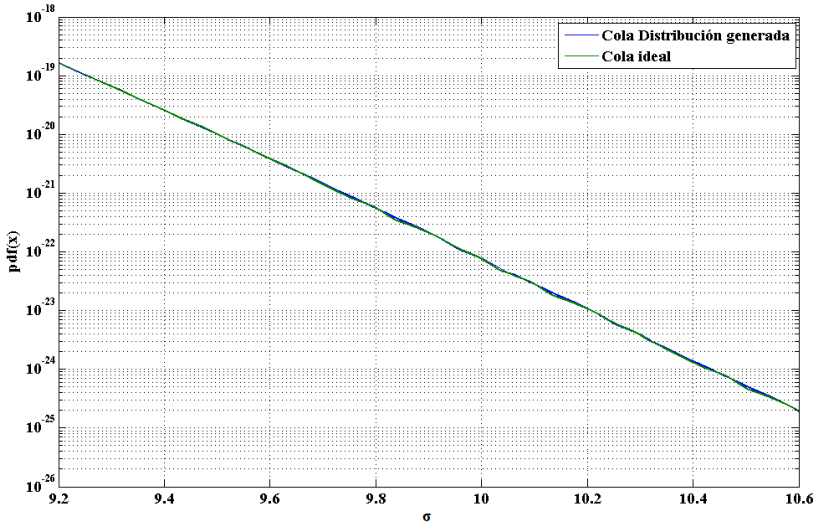


Figura 6.24.d - PDF de las muestras generadas comparadas con la distribución ideal en el tramo 10.6σ a 12σ.

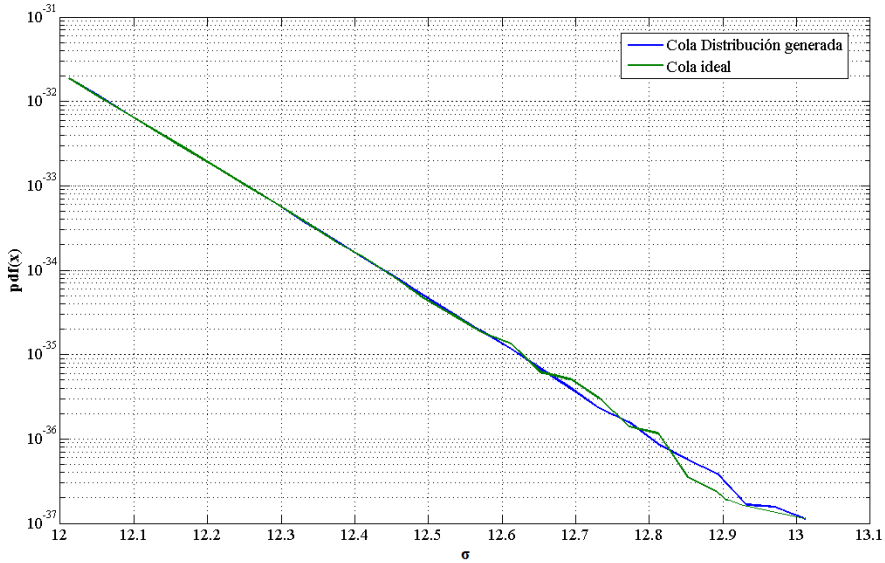


Figura 6.24.e - PDF de las muestras generadas comparadas con la distribución ideal en el tramo 12σ a 13.1σ.

En la figura 6.25 podemos ver el error cometido en una simulación utilizando 200000 muestras generadas e instanciando el modelo VHDL con interpolación cuadrática en System Generator. Este error lo hemos representado en términos de *ulp*. Podemos apreciar como el 49.4% de las muestras generadas están exactamente redondeadas (error<1/2 ulp) y el 50.6% restante están fielmente redondeadas (error<1 ulp).

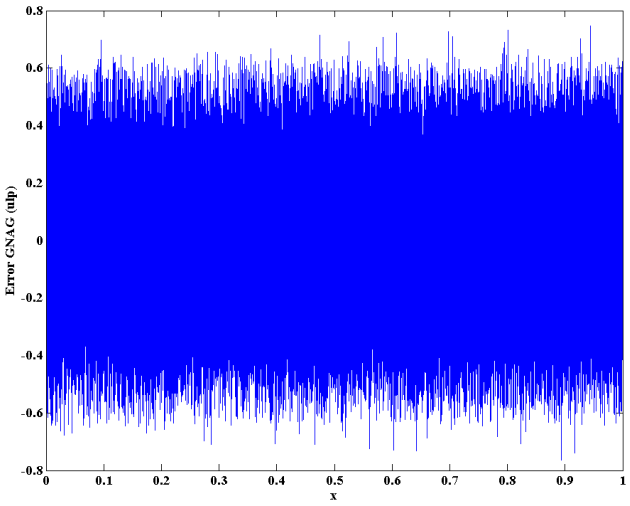
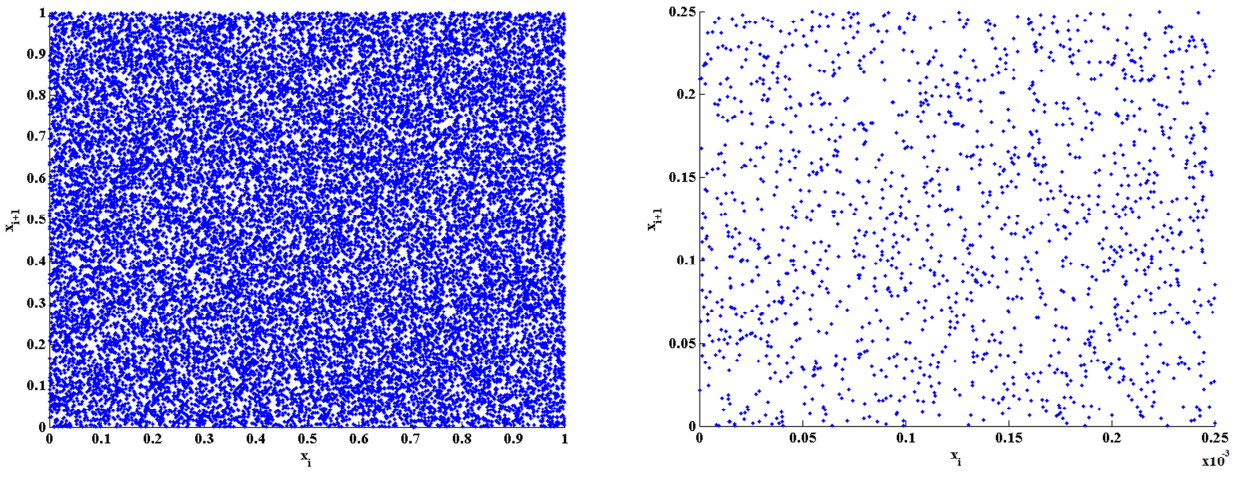


Figura 6.25 - Simulación del error cometido en la implementación del GNAG propuesto en términos de ulp.

Un aspecto muy importante en nuestro diseño es el de asegurar que las muestras generadas sean valores aleatorios Gaussianos, para ello hemos realizado varios test estadísticos. Para poder generar los números aleatorios Gaussianos partimos de la utilización de valores uniformemente distribuidos, con lo cual es importante asegurarnos que estos valores estén uniformemente distribuidos. Para ello hemos utilizado el test Diehard [81] sobre el generador Tausworthe de 64 bits de los cuales nos hemos quedado con los 32 bits centrales (48:17) ya que este test únicamente está disponible para enteros de 32 bits y verificando que todos los test han sido pasados con éxito. En la figuras 6.26.a y 6.26.b podemos ver unas imágenes de la distribución 2D de las muestras a la salida del generador de muestras uniforme. No se aprecia ninguna correlación entre las muestras generadas. Para un mejor análisis deberíamos generar una serie de imágenes de la distribución 2D y crear una secuencia de imágenes y analizar si aparece algún efecto de correlación.



(a) (b)

Figura 6.26.a - Diagrama de dispersión de las muestras generadas por el GNUD. **Figura 6.26.b** - Ampliación del diagrama de dispersión.

En la figura 6.27 podemos ver la desviación 2D de las muestras a la salida del generador AWGN. Vemos cómo tampoco aparece ninguna correlación.

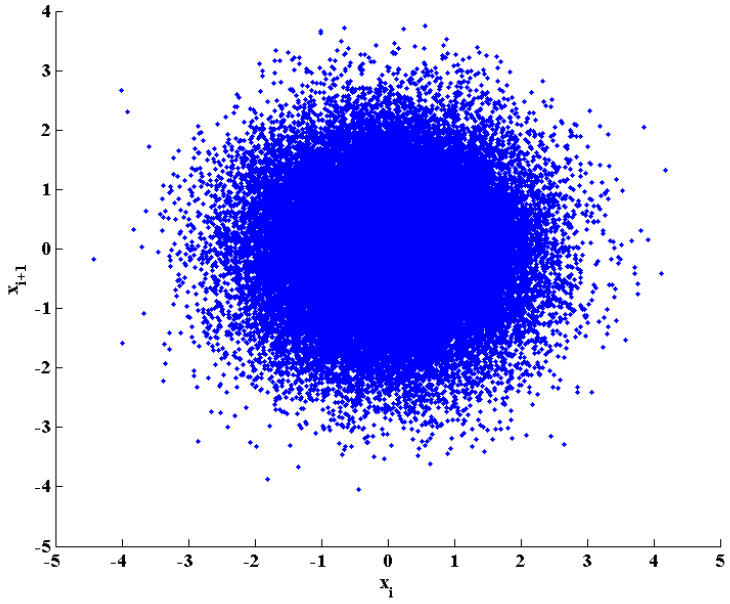


Figura 6.27 - Diagrama de dispersión de las muestras generadas por el GNAG.

Una vez validadas las muestras del generador GNUD, tendremos que verificar la normalidad de las muestras Gaussianas generadas. Para ello utilizaremos los test Chi-cuadrado y el test Anderson-Darling. El test chi-cuadrado, como hemos visto en el capítulo anterior, nos da información sobre cuán parecido es el histograma de las muestras generadas respecto a la PDF ideal. El Test Anderson-Darling (A-D) permite verificar si las muestras generadas provienen de una distribución específica, en nuestro caso de una distribución normal, y comprobar la correcta generación de la cola de la Gaussiana. Para los dos test calculamos el valor-P, el cual es la probabilidad de obtener un resultado al menos tan extremo como el que realmente se ha obtenido (valor del estadístico calculado), suponiendo que la hipótesis nula es cierta. Se rechaza la hipótesis nula si el valor-P asociado al resultado observado es igual o menor que el nivel de significación establecido, convencionalmente se suelen trabajar con valores 0.05 (5%) ó 0.01 (1%). Este parámetro también suele ser llamado potencia del contraste. Es decir, el valor-P nos muestra la probabilidad de haber obtenido el resultado que hemos obtenido si suponemos que la hipótesis nula es cierta. Si el valor-P obtenido es inferior a la potencia del contraste nos indicará que lo más probable es que la hipótesis de partida sea falsa y las muestras no serán aleatorias. Valor-P es un valor de probabilidad por lo que oscila entre 0 y 1. Así, se suele decir que valores altos de valor-P aceptan la hipótesis.

Para realizar el test chi-cuadrado vamos a dividir la PDF en cuatro regiones y cada región la hemos dividido en 250 contenedores (bins) (γ). Para cada región calcularemos el valor $\chi_{2\alpha,\gamma-1}$ aplicando

$$\chi^2_{\alpha,\gamma-1} = \sum_{n=1}^{\gamma} \frac{(S_{osb} - S_{\exp})^2}{S_{\exp}}, \tag{6.16}$$

siendo S_{osb} el numero de muestras observadas por cada contenedor y S_{exp} el numero de muestras esperadas de acuerdo a una distribución normal, α será el nivel significativo para descartar la hipótesis y γ_{-1} son los grados de libertad. Como la distribución normal está definida por su media y su desviación

estándar, los grados de libertad se reducen en 2. Con el valor χ^2 calcularemos el valor-P del test para las distintas regiones. Si superamos el valor significativo, en nuestro caso α=0.05 (95% de confidencia), el test χ^2 se habrá pasado con éxito, indicando que nuestras muestras son normalmente distribuidas. Para realizar los test hemos realizado simulaciones con 50 millones de muestras en cada región. La tabla 6.10 y 6.11 muestra los valores-P obtenidos en los test por la arquitectura propuesta y las versiones modificadas, respectivamente. La arquitectura A modificada al no llegar hasta el valor 13σ, no se han realizado los test estadísticos en el último tramo.

Rango Distribución	**Valor-P**	**Test** $\chi^2_{0.05,247}$
$\lvert\sigma\rvert < 6.2$	0.846	Pasado
$6.2 < \lvert\sigma\rvert \le 7.8$	0.633	Pasado
$7.8 < \lvert\sigma\rvert \le 9.2$	0.556	Pasado
$9.2 < \lvert\sigma\rvert \le 10.6$	0.487	Pasado
$10.6 < \lvert\sigma\rvert \le 12$	0.426	Pasado
$12 < \lvert\sigma\rvert \le 13.1$	0.389	Pasado

Tabla 6.10 – Resultados obtenidos por el GNAG en las simulaciones del test $\chi^2_{0.05,247}$.

	Arq. A		**Arq. B**		**Arq. C**	
Rango Distribución	**Valor-P**	**Test** $\chi^2_{0.05,247}$	**Valor-P**	**Test** $\chi^2_{0.05,247}$	**Valor-P**	**Test** $\chi^2_{0.05,247}$
$\lvert\sigma\rvert < 6.2$	0.824	Pasado	0.824	Pasado	0.824	Pasado
$6.2 < \lvert\sigma\rvert \le 7.8$	0.658	Pasado	0.658	Pasado	0.658	Pasado
$7.8 < \lvert\sigma\rvert \le 9.2$	0.561	Pasado	0.561	Pasado	0.561	Pasado
$9.2 < \lvert\sigma\rvert \le 10.6$	0.478	Pasado	0.478	Pasado	0.478	Pasado
$10.6 < \lvert\sigma\rvert \le 12$	0.398	Pasado	0.398	Pasado	0.398	Pasado
$12 < \lvert\sigma\rvert \le 13.1$	------	-------	0.392	Pasado	0.401	Pasado

Tabla 6.11 - Resultados obtenidos por el GNAG en las simulaciones del test $\chi^2_{0.05,247}$. Arquitecturas modificadas.

Podemos apreciar cómo para todas las simulaciones el valor-P obtenido es superior al valor significativo marcado α=0.05, pasando de esta manera el test χ^2. El valor-P siempre estará comprendido entre cero y uno, siendo uno el valor-P obtenido si pasamos el test sobre la Gaussiana ideal. Conforme vamos aumentando el valor de σ vamos obteniendo un menor valor-P, pero siempre superior al nivel crítico fijado. Podemos apreciar como el hecho de eliminar el *barrel-shifter* de 128 bits no ha afectado al correcto funcionamiento. En todos los caso los valores obtenidos han sido muy parecidos, además de obtener valores bastante cercanos a uno, demostrando la alta calidad de las muestras generadas incluso para valores cercanos al máximo ($\approx \pm 13\sigma$).

Para el cálculo de los valores del test A-D de la distribución normal hemos calculado

$$A^2 = -N - \sum_{i=1}^{N} \frac{(2i-1)}{N}\left[\ln F(Y_i) + \ln\left(1 - F(Y_{N+1-i})\right)\right]$$
$$A'^2 = A^2\left(1 + \frac{4}{N} - \frac{25}{N^2}\right), \tag{6.17}$$

siendo *F()* la CDF de la distribución normal, Y_i los valores ordenados de las muestras Gaussianas y N el número total de muestras. La tabla 6.8 muestra los valores obtenidos para las distintas regiones de la Gaussiana. Para este test compararemos el valor obtenido A'^2 de aplicar (6.18) con el valor A'^2 crítico. Este valor vendrá dado por el nivel significativo de la medida α. Para nuestro test hemos utilizado α=0.05 y el valor crítico es A'^2_{crit}=0.751. La hipótesis será aceptada si el valor obtenido A'^2 es menor que A'^2_{crit}. Vemos a partir de los resultados presentados en la tabla 6.12 y 6.13, muestra los valores críticos y el valor-P obtenido en los test para la arquitectura propuesta y las versiones modificadas, respectivamente. En todos los casos el valor obtenido es inferior al crítico, pasando el test A-D. Podemos apreciar cómo en todos los casos el valor-P obtenido es superior al nivel crítico definido, confirmando el comportamiento gaussiana de las muestras generadas.

Rango Distribución	**A'^2**	**Valor-P**	**Test A-D (α=0.05)**
$\lvert\sigma\rvert < 6.2$	0.204	0.875	Pasado
$6.2 < \lvert\sigma\rvert \leq 7.8$	0.214	0.851	Pasado
$7.8 < \lvert\sigma\rvert \leq 9.2$	0.261	0.708	Pasado
$9.2 < \lvert\sigma\rvert \leq 10.6$	0.350	0.472	Pasado
$10.6 < \lvert\sigma\rvert \leq 12$	0.451	0.274	Pasado
$12 < \lvert\sigma\rvert \leq 13.1$	0.485	0.227	Pasado

Tabla 6.12 – Resultados obtenidos por el GNAG en las simulaciones del test A-D.

	Arq. A			**Arq. B**			**Arq. C**		
Rango Distribución	A'^2	Valor-P	Test A-D (α=0.05)	A'^2	Valor-P	Test A-D (α=0.05)	A'^2	Valor-P	Test A-D (α=0.05)
$\lvert\sigma\rvert < 6.2$	0.190	0.898	Pasado	0.190	0.898	Pasado	0.190	0.898	Pasado
$6.2 < \lvert\sigma\rvert \leq 7.8$	0.211	0.856	Pasado	0.211	0.856	Pasado	0.211	0.856	Pasado
$7.8 < \lvert\sigma\rvert \leq 9.2$	0.289	0.615	Pasado	0.289	0.615	Pasado	0.289	0.615	Pasado
$9.2 < \lvert\sigma\rvert \leq 10.6$	0.325	0.523	Pasado	0.325	0.523	Pasado	0.325	0.523	Pasado
$10.6 < \lvert\sigma\rvert \leq 12$	0.410	0.343	Pasado	0.410	0.343	Pasado	0.410	0.343	Pasado
$12 < \lvert\sigma\rvert \leq 13.1$	------		-------	0.498	0.211	Pasado	0.452	0.272	Pasado

Tabla 6.13 - Resultados obtenidos por el GNAG en las simulaciones del test A-D. Arquitecturas modificadas.

A partir de los resultados obtenidos podemos confirmar que las muestras generadas por las arquitecturas de GNAG propuestas siguen una distribución normal. Para poder realizar correctamente los test para altos valores de sigma hemos tenido que programar dichos test en Python importando los resultados de las simulaciones desde Matlab/System Generator. Por último, para evaluar la correcta precisión del GNAG en una simulación BER, hemos instanciado el generador en un sistema de comunicaciones con codificación BPSK sobre un canal AWGN. El concepto de medida del BER es

bastante simple. Primero, generaremos muestras de datos aleatorias mediante un LFSR. Segundo codificaremos las muestras aleatorias mediante un codificador BPSK. Tercero, añadiremos errores aleatorios a las muestras generadas para simular el canal con ruido. Cuarto, decodificaremos las muestras. Por último, mediremos el número de errores entre los datos originales y los decodificados. La figura 6.28 muestra un diagrama de bloques para la medida del BER.

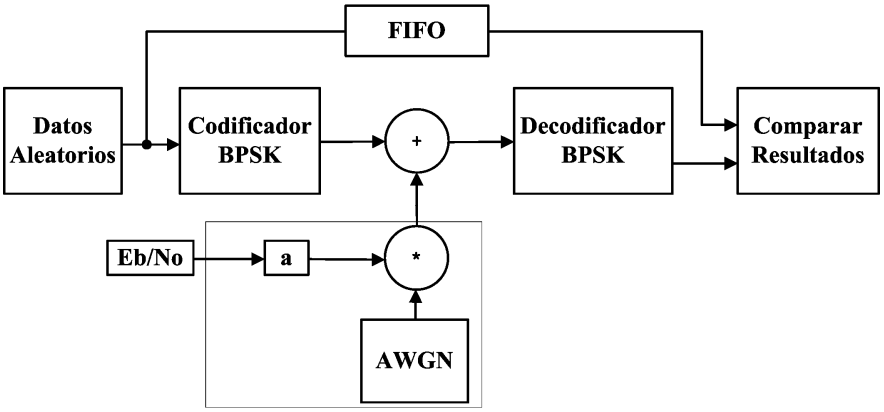


Figura 6.28 – Diagrama de Bloques para la implementación de la medida del BER.

En la figura 6.29 podemos ver la curva del BER teórica para una codificación BPSK comparada con el BER medido en la implementación de la arquitectura C. Para realizar el cálculo de la curva BER a distintos valores Eb/No (relación Energía por Bit/densidad espectral de Potencia de Ruido), se ha escalado la salida del canal Gaussiano por una constante a, definida a partir de (6.18). De esta manera podemos medir el BER a diferentes valores de Eb/No. Podemos apreciar como la curva del BER medido se asemeja bastante a la curva ideal. En menos de un segundo hemos obtenido un valor de BER $4.09 \cdot 10^{-6}$ mientras que para obtener el mismo valor en una simulación software hubiéramos necesitado días para llegar a este valor. Para simulaciones con una duración mucho mayor se ha podido llegar a generar valores de BER de hasta 10^{-13}.

$$a = \frac{1}{\sqrt{\frac{1}{2 \cdot (Eb/No)}}} \tag{6.18}$$

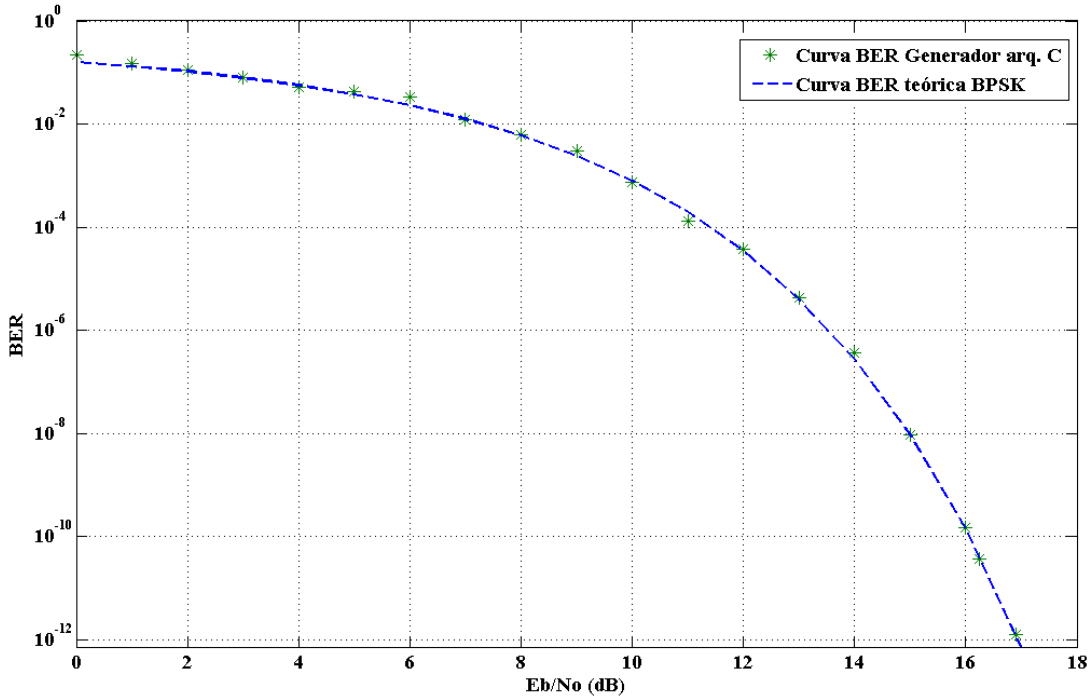


Figura 6.29 - Comparación entre la grafica del BER obtenido por el GNAG propuesto y el BER de la codificación BPSK ideal.

6.6. Conclusiones

En el presente capítulo hemos presentado un Generador de Muestras Aleatorias Gaussianas (GNAG) basado en el método de la inversión de la función de probabilidad acumulada (ICDF) de la distribución Gaussiana. Las arquitecturas propuestas generan muestras aleatorias Gaussianas con una desviación estándar que puede llegar hasta valores de $\pm 13.1\sigma$ con unos requerimientos hardware bajos. Para la aproximación de la función ICDF hemos utilizado métodos de aproximación de funciones sobre la que hemos implementado una segmentación no uniforme. Además, para poder realizar eficientemente las operaciones aritméticas hemos realizado un cambio de variable que ha permitido reducir los requerimientos de memoria y la complejidad de las operaciones, manteniendo la precisión de la aproximación. La gran ventaja de la utilización del método de la Inversión es que fácilmente podemos aproximar otras distribuciones diferentes de la Gaussiana como pueden ser Weibull, exponencial, logarítmica, etc. En principio bastaría con calcular los nuevos coeficientes partiendo de la nueva distribución. Nuestro generador de canal Gaussiano genera muestras aleatorias en complemento a dos de 18 bits con una precisión de 13 bits cada ciclo de reloj, pudiendo llegar una tasa de muestras generadas de 475 Msps con el dispositivo Virtex-5 XC5VLX110T-3.

La arquitectura propuesta ha sido implementada utilizando interpolación lineal y cuadrática. Para cada tipo de interpolación hemos realizado diferentes implementaciones utilizando distintas familias de FPGAs de Xilinx (Virtex-2, Virtex-4 y Virtex-5). Una implementación típica del GNAG propuesto utiliza únicamente el 1.35% de los *slices* en una FPGA Virtex-5 XC5VLX110T, una Block-RAM y dos

multiplicadores DSP48. Si utilizamos una FPGA Virtex-2 XC2V4000 el área utilizada aumenta hasta aproximadamente un 3.1%. También se han realizado implementaciones modificando la precisión de las muestras generadas y el valor de desviación estándar de la distribución Gaussiana, permitiendo una rápida escalabilidad de la arquitectura propuesta. Vemos que podemos aumentar el valor de la desviación estándar hasta valores mucho mayores sin un aumento importante de los recursos utilizados y sin reducir la tasa de muestras generadas. También hemos realizado implementaciones modificando el tipo de recursos utilizados de la FPGA y viendo la importancia de la utilización de estos recursos. Por ejemplo, el no utilizar los bloques DSP48 implica una reducción en la frecuencia máxima de funcionamiento de 200 MHz. También se han propuesto modificaciones a la arquitectura originalmente propuesta permitiendo la eliminación del *barrel-shifter* necesario para el cambio de variable. En este caso la reducción del número de *slices* necesarios ha sido considerable sin perjudicar significativamente el comportamiento aleatorio de las muestras generadas. Se han comparado los resultados de las arquitecturas propuestas con diferentes implementaciones hardware recientes, obteniendo en la comparación un buen resultado en cuanto a precisión en las muestras generada, área necesaria y latencia.

Por último, para verificar correctamente el comportamiento aleatorio de las muestras generadas, se han realizado diversos test estadísticos (Anderson-Darling y χ^2) que nos han permitido verificar el correcto comportamiento de las muestras aleatorias Gaussianas. Para poder comprobar la implementación hasta valores de $\pm 13.1\sigma$, hemos dividido la distribución Gaussiana en cinco partes (6.2σ a 7.8σ, 7.8σ a 9.2σ, 9.2σ a 10.6σ, 10.6σ a 12σ y 12σ a 13.1σ) y en cada parte se han realizados los test estadísticos. Para realizar una simulación por fuerza bruta de toda la Gaussiana llegando hasta valores de desviación de $\pm 13.1\sigma$, hubiéramos necesitado miles de años, imposibilitando su verificación. Para parte de la Gaussiana se ha obtenido el valor-P del test. Este parámetro varía entre 0 y 1. Un valor-P de cero indica una muestra no Gaussiana y un valor-P de uno seria una distribución Gaussiana ideal (tomando infinitas muestras en el test realizado). Realizando los test estadísticos sobre nuestra arquitectura hemos obtenido valores de hasta 0.898, indicando la "alta" calidad de las muestras aleatorias generadas. Conforme vamos aumentando el valor de la desviación máxima, este parámetro se reduce, pero en ningún caso es inferior al nivel significativo marcado $\alpha=0.05$ y por lo tanto pasando los test estadísticos. La arquitectura modificada también ha conseguido pasar los test estadísticos, obteniendo menores valor-P, pero aun así, superiores al nivel significativo marcado y en este caso con una significativa reducción de área necesaria. En la implementación de la medida del BER hemos visto como hemos podido llegar a valores de 10^{-13} en unos tiempos relativamente cortos comparados con simulaciones software, las cuales hubieran necesitado meses para poder llegar a generar eventos para ese valor.

Capítulo 7. Conclusiones

7.1. Conclusiones

En esta tesis doctoral se planteó como primer objetivo estudiar la implementación hardware de funciones elementales (en concreto $\log(x)$ y $\text{atan}(y/x)$) con las prestaciones adecuadas para los sistemas de comunicaciones, concretamente para su uso en algoritmos de procesado de señal en banda base en sistemas de comunicaciones de banda ancha. Además se estableció un segundo objetivo que fue diseñar un emulador hardware de un canal Gaussiano (AWGN) que fuese útil para evaluar las prestaciones de subsistemas de comunicaciones. Los diseños han estado orientados en todo momento a su implementación en FPGAs, dispositivos de creciente uso en sistemas de comunicaciones.

Respecto de la implementación de funciones elementales se ha procedido a analizar las técnicas de aproximación de funciones existentes, CORDIC, métodos basados en convergencia lineal, métodos basados en recurrencia de dígitos, métodos basados en LUTs y todas sus variantes, aproximaciones por polinomios, aproximaciones por división de polinomios y aproximaciones por series de Taylor. Para cada método se ha realizado la aproximación de la función elemental $\text{atan}(y/x)$ y se han analizado los resultados de implementación obtenidos. Desde el punto de vista de las prestaciones necesarias para un sistema de comunicaciones CORDIC y las distintas variantes de los métodos basados en LUTs parecen los algoritmos más eficientes. Las arquitecturas propuestas en la presente tesis doctoral mejoran algunos de los resultados obtenidos con estos métodos, sin que en ningún caso el *throughput* de la implementación se sitúe por debajo de los 20 Msps.

Una de nuestras propuestas para $\text{atan}(y/x)$ obtiene una reducción de consumo de potencia del 50% de media con respecto a implementaciones basadas en LUTs multipartidas y CORDIC. Esta arquitectura divide la aproximación de $\text{atan}(y/x)$ en dos etapas: primero calculamos la operación $z=y/x$ a través del cálculo del recíproco de "x" y multiplicando el resultado por "y"; y posteriormente realizamos el cálculo de la función $\text{atan}(z)$. Para las dos aproximaciones se utilizan LUTs, aplicando el método bipartido. Estos resultados (documentados en el capítulo 3 de la presente memoria) se han publicado en un artículo [T.1] en el congreso *Field Programmable Logic and Application* (FPL07) celebrado en Amsterdam y otro [T.2] en la revista *Journal of Signal Processing Systems* en el año 2009, de la editorial Springer.

Otras dos propuestas de arquitectura para la aproximación de la $\text{atan}(y/x)$ dividen el cálculo en las siguientes dos etapas: 1) se realiza una transformación logarítmica que permite computar la división (y/x) como una resta y 2) se calcula $\text{atan}(2^w)$, siendo $w=\log_2(y)-\log_2(x)$, obteniendo el resultado deseado, $\text{atan}(y/x)$. La primera de las arquitecturas emplea tablas multipartidas tanto para aproximar el logaritmo de las entradas como para $\text{atan}(2^w)$. En el caso de la arcotangente se ha aplicado un esquema de segmentación no-uniforme, reduciendo de esta manera el tamaño de las tablas. Los resultados obtenidos en la implementación de esta arquitectura (documentados en el capítulo 4) se presentaron [T.4] en las *Jornadas de Computación Reconfigurable y Aplicaciones* (JCRA09) celebradas en Alcalá de Henares, obteniendo el premio al mejor artículo. Una versión extendida del artículo fue publicada [T.5] en la revista *Journal of Systems Architecture* de la editorial Elsevier en el año 2010. La segunda arquitectura emplea aproximaciones lineales por tramos tanto para el logaritmo como para $\text{atan}(2^w)$. Ambas aproximaciones se basan en la aproximación lineal por tramos (empleando rectas con pendientes potencias de 2, lo que permite optimizar significativamente los recursos empleados) y compensación del error residual mediante tablas. En el caso del logaritmo se

emplea además la aproximación de Mitchell. Cualquiera de estas dos arquitecturas para la aproximación de la atan(y/x) reduce enormemente el área necesaria para su implementación con respecto a la arquitectura comentada anteriormente. No obstante, la segunda, la basada en aproximaciones lineales por tramos, es la que menos área necesita de todas las arquitecturas propuestas, tanto de las que se encuentran en la bibliografía como de las propuestas en la presente tesis. La aproximación del logaritmo empleada en esta última arquitectura (documentada en el capítulo 4 de la presente memoria) se ha publicado [T.6] en la revista *IEEE Transactions on Very Large Scale Integration* en el año 2011. Se está preparando otro artículo describiendo la arquitectura completa.

Por último se han propuesto varias arquitecturas de un generador de ruido blanco aditivo Gaussiano (AWGN) de altas prestaciones y con un bajo coste hardware. El generador está basado en el uso de la Función de Acumulación de Distribución Inversa (ICDF) de la distribución Gaussiana. Las arquitecturas propuestas permiten emular canales Gaussianos con una elevada precisión (13 bits fraccionales) y desviación (hasta $\pm 13.1\sigma$), valores superiores a la mayoría de las implementaciones presentes en la bibliografía, al tiempo que emplean menos área que ellas y consiguen una alta tasa de muestras. Todas las arquitecturas propuestas realizan la aproximación de la ICDF de la distribución Gaussiana utilizando una interpolación polinómica combinada con el uso de la segmentación no-uniforme de la función. Algunas de las arquitecturas propuestas consiguen reducir de forma significativa el área eliminando parcial o completamente el *barrel-shifter* utilizado en las implementaciones de otros autores del método de la inversión. También se propone una arquitectura en la que el *barrel-shifter* se sustituye por un operador, más eficiente desde el punto de vista de la implementación, que genera los bits de entrada de los multiplicadores aplicando una máscara binaria. Este último caso, que es el que produce mejor resultado en cuanto a área de todas las arquitecturas que alcanzan la máxima desviación de $\pm 13.1\sigma$, necesita un 58.3% menos área que la arquitectura con el *barrel-shifter* completo. Los resultados de implementación de la primera arquitectura (documentados en el capítulo 6 de esta memoria) han sido presentados [T.7] en las *Jornadas de Computación Reconfigurable y Aplicaciones* (JCRA10) que se ha celebrado en Valencia en septiembre del año 2010. Asimismo los resultados de la última arquitectura, la que mejores resultados proporciona, se han enviado a la revista *IEEE Transactions on Very Large Scale Integration*, encontrándose actualmente el artículo en proceso de revisión.

7.2. Líneas Futuras

Existen otras funciones cuya implementación optimizada es de interés en los sistemas de comunicaciones, como es el caso del logaritmo de la arcotangente hiperbólica, empleada en la decodificación de códigos LDPC. El elevado número de instancias de esta función en un decodificador de alta velocidad hace que sea de gran interés buscar soluciones de muy bajo coste.

También resultaría de interés profundizar en las técnicas de reducción de consumo de los diseños hardware. El consumo de potencia es un factor muy importante en la mayoría de los dispositivos portátiles.

Por otro lado los test estadísticos utilizados para validar el comportamiento del emulador de canal presentado en esta tesis doctoral han sido implementados mediante simulaciones software. Esto ha limitado enormemente el número de muestras utilizadas en los test. Es deseable realizar estas pruebas en hardware y de esta manera permitir aumentar el número de muestras utilizadas, incrementando por tanto la calidad de la verificación.

7.3. Resultados Publicados durante la Tesis Doctoral

[T.1] R. Gutiérrez, J. Valls. "Implementation on FPGA of a LUT-based atan(Y/X) operator suitable for Synchronization Algorithms", 2007 International Conference in Field Programmable Logic and Applications (FPL07), pp. 472-475, 2007

[T.2] R. Gutiérrez, J. Valls. "Low-Power FPGA-Implementation of atan(Y/X) using Look-up Table Methods for Communication Applications". Journal of Signal Processing Systems, vol. 56, Nº 1, pp. 25 – 33, 2009

[T.3] R. Gutiérrez, J. Valls, A. Pérez-Juan. "FPGA-implementation of Time-Multiplexed Multiple Constant Multiplication based on Carry-save Arithmetic", 2009 International Conference in Field Programmable Logic and Applications (FPL09), pp. 609-612, 2009

[T.4] R. Gutiérrez, V. Torres, J. Valls. "Implementación en FPGA de la arcotangente(Y/X) usando aproximaciones logarítmicas. IX Jornadas de Computación Reconfigurable y Aplicaciones JCRA09, pp. 445-454, 2009. - Premio Mejor Articulo JCRA09

[T.5] R. Gutiérrez, V. Torres, J. Valls. "FPGA implementation of atan(Y/X) based on logarithmic Transformation and LUT-based techniques. Journal of Systems Architecture, vol. 56, no. 11, pp. 588-596, 2010.

[T.6] R. Gutierrez, J. Valls. "Low Cost Hardware Implementation of Logarithm Approximation". IEEE Transactions on Very Large Scale Integration (VLSI) Systems. DOI: http://dx.doi.org/10.1109/TVLSI.2010.2081387

[T.7] R. Gutiérrez, V. Torres, J. Valls. "Generador de ruido AWGN de altas prestaciones mediante el método de la inversión". X Jornadas de Computación Reconfigurable y Aplicaciones JCRA10, pp. 43-50, 2010.

Publicaciones en Proceso de Revisión

R. Gutiérrez, V. Torres, J. Valls. "Hardware Architecture of a Gaussian Noise Generator Based on Inversion Method".

Bibliografía

[1] U. Mengali, A. N. D'Andrea. Synchronization Techniques for Digital Receivers. Springer, 1997

[2] J. Heiskala, J. Terry. OFDM Wireless LANs. A Theoretical and Practical Guide. Sams, 2001

[3] B. K. Park, O. Boric-Lubeke, V. M. Lubeke. "Arctangent Demodulation with DC offset Compensation in Quadrature Doppler Radar Receivers Systems". *IEEE Transactions on Microwave Theory and Techniques*, vol. 55, no. 5, pp. 1073 - 1078, 2007

[4] H. Guo, G. Liu. "Approximations for the arctangent function in efficient fringe pattern analysis". *Optics express*, vol. 15, no. 6, pp. 3053 - 3066, 2007

[5] F. Liao, M. Zhang, L. Wang, Y. Liao. "The noise and digital realization of arctangent approach of PGC demodulation for optic interferometric sensors". *Proceedings on SPIE*. Vol. 6595, 2007

[6] S. J. Bellis, W. P. Marnane. "A CORDIC Arctangent FPGA implementation for a High-Speed 3D-Camera Systems". *Lecture Notes in Computer Science.* pp. 485 - 494, 2000

[7] J. M. Dunn-Rogers. "An Analysis of the Performance of DRFM for Electronic Warfare Systems". *IEE EW Systems Colloquium.* 1991

[8] F. Angarita, M.J. Canet, T. Sansaloni, A. Perez-Pascual, J. Valls. "Efficient mapping of CORDIC Algorithm for OFDM-based WLAN". *Journal of Signal Processing Systems, vol. 52, no. 2, pp. 181 - 191,* 2008

[9] A. Menezes, P. van Oorschot, S. Vanstone. Handbook of Applied Cryptography. CRC Press. 1996

[10] Richardson, Texas, Cyrix Corporation. *Cyrix 6x86 Processor Data Book*, 1996

[11] Microprocessor Report, Various issues, 1994

[12] C. Daramy, D. Defour, F. de Dinechin, J.M. Muller. "CR-LIBM, a Correctly Rounded Elementary Function Library"**.** *SPIE 48th Annual Meeting International Symposium on Optical Science and Technology*, 2003

[13] A. Ziv. "Fast evaluation of elementary mathematical functions with correctly rounded last bit". *ACM Transactions on Mathematical Software*, vol. 17, no. 17, pp. 410 - 423, 1991

[14] J.M. Muller. Elementary Functions. Algorithms and Implementation**.** Birkhauser 2ª edition, 2006

[15] S. F. Oberman, M. J. Flynn. "An analysis of division Algorithms and Implementations". Technical Report: CSL-TR-95-67. pp. 58, 1995

[16] S.Story, P.T.P.Tang. "New Algorithms for improved Transcendental Functions on IA-64". *14th IEEE Symposium on Computer Arithmetic (ARITH-14 '99)*, pp. 4 - 11, 1999

[17] W. Cody, W. Waite**.** Software Manual for the Elementary Functions. Prentice Hall,1980

[18] V. Lefévre, J.M. Muller. "On-the-fly range reduction**".** *Journal of VLSI signal Processing*, vol. 33, no. 1, pp. 31 - 35, 2002

[19] M. Payne, R. Hanek. "Radian reduction for trigonometric functions". *ACM SIGNUM Newsletter*, vol. 18, no. 1, pp. 19 - 24, 1983

[20] M. Daumas, C. Mazenc, X. Merrheim, J.M. Muller**.** "Modular range reduction: A new algorithm for fast accurate computation of the elementary function". *Journal Universal Computer Science*, vol. 1, no. 3, pp. 162 - 175, 1995

[21] J. E. Volder. **"**The CORDIC trigonometric computing Technique". *IRE Transaction Electronic Computers*, vol. EC-8, pp 330 - 334, 1959

[22] J. S. Walther. **"**A unified algorithm for elementary functions". *Proceeding Spring Joint Computer Conference*, pp. 379 - 385, 1971

[23] R. Andraka. "A Survey of CORDIC algorithms for FPGA based Computers". *Proceedings of the 1998 ACM/SIGDA 6th international symposium on Field Programmable Gate Arrays*, pp. 191 - 200, 1998

[24] Xilinx Inc. "CORDIC IP V4.0 core v1.0 – DS249", 2009

[25] M. D. Ercegovac. T. Lang. Digital Arithmetic. Morgan Kaufmann Publishers. 2003

[26] Xilinx Inc. "Divider Generator V3.0 – DS530", 2009

[27] B. Parhami. Computer Arithmetic. Algorithms and Hardware Design. Oxford University Press, 2000

[28] D. DasSarma, D. W. Matula. "Measuring the accuracy of ROM reciprocal Tables". *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 932 - 940, 1994

[29] M. J. Flynn, S. F. Oberman. Advanced Computer Arithmetic Design. John Wiley & Sons, 2001

[30] D. DasSarma, D.W. Matula**.** "Faithful Bipartite ROM reciprocal tables". *IEEE Symposium on Computer Arithmetic*, pp. 17, 1995

[31] J. M. Muller. "A few results on Table-Based methods". *Journal of Reliable Computing*, vol. 5, no. 3, pp. 279 - 288. Springer Netherlands, 1999

[32] M. J. Shulte, J. E. Stine. "Symmetric bipartite tables for accurate function approximation". *Proceeding of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp.144 - 153. IEEE computer Society Press,1997

[33] F. de Dinechin, A. Tisserand. "Some improvements on Multipartite table Methods". *IEEE Symposium on Computer Arithmetic*, pp. 128 - 135, 2001

[34] M. J. Shulte, J. E. Stine**.** "The Symmetric Table Addition Method for Accurate function Approximation". *Journal of VLSI Signal Processing*, vol. 11, pp. 1 - 11. Kluwer Academic Publishers, Boston, 1999

[35] D. DasSarma y D.W. Matula. "Faithful interpolation in Reciprocal Tables". *13th IEEE Symposium on Computer Arithmetic*, pp. 82 - 91, 1997

[36] M. Combet, H. Van Zonneveld, L. Verbeek. "Computation of the base Two Logarithm of Binary Numbers". *IEEE Transactions on Electronics Computers*, vol. EC-14, no. 6, pp. 863 - 867, 1965

[37] D. Lee, R. C. C. Cheung, W. Luk, J. D. Villasenor. "Hierarchical Segmentation for Hardware Evaluation". *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, vol.17, no. 1, pp. 103 - 116, 2009

[38] M. J. Shulte, E.E. Swartzlander**.** "Hardware Designs for Exactly rounded elementary functions". *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 964 - 973, 1994

[39] W. Ferguson**.** "Exact Computation of a sum or difference with applications to argument reduction". *IEEE Symposium on Computer Arithmetic*, pp. 216 - 221, 1995

[40] S. Rajan, S. Wang, R. Inkol, A. Joyal. "Efficient Approximations for the Arctangent Function". *IEEE signal Processing Magazine*, pp. 108 - 111, 2006

[41] S. Rajan, S. Wnag, R. Inkol. "Efficient approximation for the four-quadrant arctangent function". Canadian *Conference on Electrical and Computer Engineering, 2006. CCECE '06*. pp. 1043 – 1046, 2006

[42] S. Rajan, S. Wang, R. Inkol. "Error Reduction Technique for Four-Quadrant arctangent approximations". *IET Signal Processing*, vol. 2, no. 2, pp. 133 - 138, 2008

[43] M. Saber, Y. Jitsumatsu, T. Kohda. "A Low-Power Implementation of arctangent function for Communication Application using FPGA". *Fourth International Workshop on Signal Design and its applications in Communications (IWSDA'09)*, pp. 60 - 63, 2009

[44] D.D. Hwang, F. Dengwei, A.N. Willson Jr. "A 400-MHz Processor for the conversion of Rectangular to Polar Coordinates in 0.25µm CMOS*". IEEE Journal of Solid-State Circuits*, vol. 38, no. 10, pp. 1771 - 1775, 2003

[45] M. J. Canet, F. Vicedo, V. Almenar, J. Valls. "FPGA implementation of an IF transceiver for OFDM-based WLAN", *2004 IEEE Workshop on Signal Processing Systems* (SiPS 2004), Austin, Texas, pp. 227 - 232, 2004

[46] V. G. Oklobdzija**.** "An Algorithmic and Novel Design of a Leading Zero Detector Circuit. Comparison with Logic Synthesis". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 1, pp. 124 - 128, 1994

[47] S. J. E. Wilton, S. Ang, W. Luk. "The impact of pipelining on energy per operation in field-programmable gate arrays". *Proceedings of the 13th International Workshop in Field-Programmable Logic and Applications, FPL2004*. pp. 719 - 728, 2004

[48] J.N. Mitchell. "Computer Multiplication and Division using a Binary Logarithms". *IEEE Transactions on Electronics Computers*, vol. 11, pp. 512 - 517,1962

[49] S. Nagayama, T. Sasao, J. T. Butler. "Programmable Numerical Function Generator based on Quadratic Approximation: Architecture and Synthesis method". *Proceedings on Asia and South Pacific design automation conference ASPDAC06*, pp. 378 - 383, 2006

[50] S. Sangregory, C. Brothers, D. Gallagher, R. Siferd. "A Fast Low-Power Logarithm Approximation with CMOS VLSI Implementation". *42nd Midwest Symposium on Circuits and Systems*, vol. 1, pp. 388 - 391, 1999

[51] T. B. Juang, S. H. Chen, H. J. Cheng. "A Lower Error and ROM-Free Logarithmic Converter for Digital Signal Processing Applications". *IEEE Transactions On Circuits and Systems – II*, vol. 56, no. 12, pp. 931 - 935, 2009

[52] K. H. Aded, R. E. Siferd. "CMOS VLSI Implementation of Low-Power Logarithmic Converter". *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1221 - 1228, 2003

[53] E. L. Hall, D.D. Lynch, S. J. Dwyer. "Generation of Products and Quotients Using Approximate Binary Logarithms for Digital Filtering Applications". *IRE Transactions Computers*, vol. 19, pp. 97 - 105, 1970

[54] T.A. Brubaker, J.C. Becker. "Multiplication using logarithms implemented with read-only memory". *IEEE Transactions on Computers*, vol. C-24, no. 8, pp. 761 - 766, 1975

[55] G. L. Kmetz. Floating point/logarithm conversion system. U.S. Patent 4583180, 1986

[56] S. Paul, N. Jayakumar, S.P. Khatri. "A Fast Hardware Approach for approximate, efficient logarithm and antilogarithm computations". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 2, pp. 269 - 277, 2009

[57] R. Maenner. "A fast integer binary logarithm of large arguments". *IEEE Micro*, vol. 7, no. 6, pp. 41 - 45, 1987

[58] J. J. Proakis, M. Salehi. Fundamentals of Communications Systems. Prentice Hall. 2004

[59] D. Thomas, W. Luk, P. H.W. Leong, J. Villasenor. "Gaussian Random Number Generator". *ACM Computing Surveys*, vol. 39, no. 11, article. 11. 2007

[60] W. Hörmann, J. Leydold." Continuous random variable generation by fast numerical inversion". *ACM Transactions on Modeling and Computer Simulation*, vol. 13, pp. 347 - 362, 2003

[61] G.E.P. Box, M.E. Muller. "A note on the generation of random normal deviates". *Annals of Mathematical Statistics*, vol. 29, pp. 610 - 611, 1958

[62] L. Devroye. "Non-Uniform Random Variate Generation". Springer-Verlag, New York, 1986

[63] G. Marsaglia, W. W. Tsang. "The Ziggurat Method for generation random variables". *Journal of Statistics Software,* vol. 5, no. 8, 2000

[64] C. S. Wallace. "Fast pseudorandom generators for normal and exponential variates". *ACM Transactions Mathematical Software*, vol. 22, no. 1, pp. 119 - 127, 1996

[65] J. Leydold, H. Leeb, W. Hörmann. "Higher dimensional properties of non-uniform pseudo-random variates. In Monte Carlo and Quasi-Monte Carlo Methods 1998". H. Niederreiter and J. Spanier, Eds. Springer-Verlag, Berlin, Heidelberg, pp. 341 – 355, 1998

[66] M. E. Muller. "An inverse method for the generation of random normal deviates on large-scale computers". *Mathematical Tables and Other Aids to Computation 12*, no. 63, pp. 167 – 174, 1958

[67] J. H. Ahrens, K. D. Kohrt. "Computer methods for efficient sampling from largely arbitrary statistical distributions". *Computing no. 26*, pp. 19 – 31, 1981

[68] M. J. Wichura. "Algorithm AS241: The percentage points of the normal Distribution". *Applied Statics*, vol.37, no. 3, pp. 447 - 484, 1988

[69] G. Marsaglia. "A current view of random number generators". *Computer Science and Statistics. 16th Sysposium on the Interface*, pp. 3 - 10, 1985

[70] P. L'Ecuyer. "Uniform Random Number Generation". *Annals of Operation Research*, vol. 53, pp. 85 - 97, 1994

[71] D. H. Lehmer. "Mathematical methods in large-scale computing units". *Proceedings of the second symposium on large scale digital computer machinery*, pp. 141 – 146, 1951

[72] G. Marsaglia, A. Zaman. "A new Class of Random number generators". *The Annals of Applied Probability*, no. 1, pp. 462 - 480, 1991

[73] R. C. Tausworthe. "Random Numbers Generated by Linear Recurrence Modulo Two". *Mathematics of Computation*, vol.19, no. 90, pp. 201 - 209, 1965

[74] T. G. Lewis, W. H. Payne. "Generalized Feedback shift Registers Pseudorandom Number Algorithm". *Journal of the ACM*, vol. 20, no. 3, pp. 456 - 468, 1973

[75] M. Matsumoto, T. Nishimura. "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator". *ACM Transactions on Modelling and Computer Simulation*, vol.8, pp. 3 - 30, 1998

[76] P.LÈcuyer. "Maximally equidistributed Combined Tausworthe Generators". *Mathematics on Computation*, vol. 65, no. 213, pp. 203 - 213, 1996

[77] V. Sriram, D. Kearney. "An FPGA Implementation of a parallelized MT19937 uniform random number generator. *EURASIP Journal on Embebded Systems*, vol. 2009

[78] P.L'Ecuyer. "Tables of Maximally equidistributed Combined LFSR Generators". *Mathematics on Computation*, vol. 68, no. 225, pp. 261 - 269, 1999

[79] D. E. Knuth. The art of computer programming, Volume 2. Seminumerical Algorithms. Addison-Wesley Publishing Company, 1998

[80] R. D'Agostino. M. Stephens. Goodness-of-fit Techniques. Marcel Dekker Inc. 1986

[81] G. Marsaglia. The Marsaglia Random Number CDROM, including the DIEHARD Battery of test of Randomness, Department of statistics, Florida State University. http://stat.fsu.edu/pub/diehard.html

[82] NIST. A statistical Test suite for Random and Pseudorandom Number generator for Cryptographic Applications, NIST special Publications. National Institute for Standards and Technology.

[83] P. L'Ecuyer, R. Simard. "TestU01: A C library for empirical Testing of Random Number Generation". *ACM Transactions on Mathematical software*, vol.33, no. 4, Art. 22, 2007

[84] E. Boutillon, J.L. Danger, A.Gazel. "Design of High Speed AWGN communication Channel Emulator". *Analog Integrated Circuits Signal Process*, pp. 133 - 142, 2003

[85] D. Lee, W. Luk, J. Villasenor, P.Y.K. Cheung. "A Gaussian Noise Generator for Hardware-Based Simulation". *IEEE Transactions on Computers*, vol. 53, no. 12, pp. 1523 - 1533, 2004

[86] D. Lee, J. Villasenor, W. Luk, P. H.W. Leong. "A Hardware Gaussian Noise Generator Using the Box-Muller Method and Its Error Analysis". *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 659 - 671, 2006

[87] A. Alimohammad, S. F. Fard, B. F. Cockburn, C. Schlegel. "A Compact and Accurate Gaussian Variate generator". *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, vol.16, no. 5, pp. 517 - 527, 2008

[88] P. Atiniramit. "Design and implementation of an FPGA-based adaptative filter single-use receiver". Master Thesis. Virginia Polytechnic Institute and State University, 1999

[89] R. Andraka, R. Phelps. "An FPGA based processor yields a real time high fidelity radar environment simulator". *In military and Aerospace Applications of Programmable Devices and Technologies Conference*, 1998

[90] J. Chen, J. Moon, K. Bazargan. "Reconfigurable Readback-Signal Generator Based on Fiel-programmable Gate Array". *IEEE Transactions on Magnetics*, vol. 40, no. 3, 2004

[91] J. McCollum, J. M. Lancaster, D. W. Bouldin, G. D. Peterson. "Hardware Acceleration of Pseudo-Random Number Generation for Simulation Applications". *35th Southeastern symposium on system Theory*, pp. 299 - 303, 2003

[92] R. C. C. Cheung, D. Lee, W. Luk, J. Villasenor. "Hardware Generation of Arbitrary Random Number Distributions From Uniform Distributions Via the Inversion Method". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 8, pp. 952 - 962, 2007

[93] Y. Fan, Z. Zilic. "BER testing of Communication Interfaces". *IEEE Transactions on Instrumentation and Measurement*, vol. 57, no. 5, pp. 897 - 906, 2008

[94] Xilinx Inc. " Additive White Gaussian Noise (AWGN) core v1.0", 2002

[95] G. Zhang, P. H. W. Leong, D. Lee, J. D. Villasenor, R. C. C. Cheung, W. Luk. "Ziggurat-based hardware Gaussian Random Number Generator". *Proceedings IEEE International Conference Field-Programmable Logic Application*, pp. 275 - 280, 2005

[96] D. Lee, W. Luk, J. Villasenor, G. Zhang, P. H.W. Leong. "A Hardware Gaussian Noise Generator Using the Wallace Method". *IEEE Transactions on Very Large Scale Integration (VLSI) System*, vol. 13, no. 8 , pp. 911 - 920, 2005

[97] C. M. Grinstead, J.L. Snell. "Introduction to Probability". American Mathematical Society, 1996